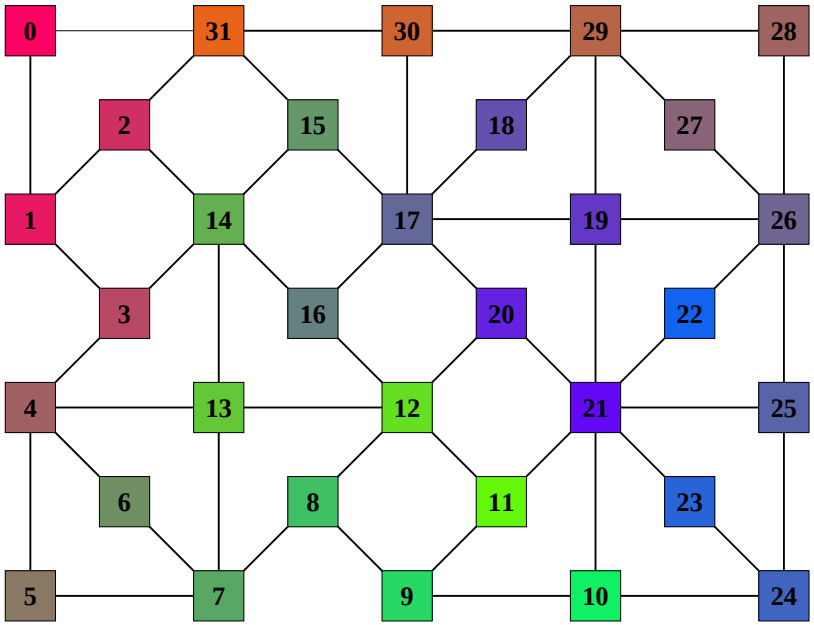
GOBLIN

A Library for Graph Matching and Network Programming Problems



Release 2.7.2 – Reference Manual

April 25, 2006

Contents

I Introduction 9

1 General Statements 11
- 1.1 Scope . . . 11
- 1.2 History . . . 11
- 1.3 Purpose and Applications . . . 12
- 1.4 Project Overview . . . 12
- 1.5 Fundamental Library Concepts . . . 13
- 1.6 Mathematical References . . . 13
- 1.7 Contributions . . . 14

2 Installation 15
- 2.1 Licence Agreement . . . 15
- 2.2 Software Requirements . . . 15
- 2.3 Unpacking the Source . . . 16
- 2.4 Configuration . . . 16
- 2.5 The Makefile and UNIX Installation . . . 17
- 2.6 Tcl Compatibility Issues . . . 18
- 2.7 Cygwin Build . . . 18
- 2.8 Windows Setup Package . . . 19
- 2.9 Download of new GOBLIN Versions . . . 19
- 2.10 Bug Reports . . . 20

3 Getting Started 21
- 3.1 The GOBLET Graph Browser . . . 21
- 3.2 GOSH Shell Scripts . . . 23
- 3.3 Using the Library . . . 24
- 3.4 Solver Executables . . . 25

4 The GOBLET Graph Browser 27
- 4.1 File Management (Menu Item: File) . . . 28
- 4.2 Graph Editor Dialogs (Menu Item: Edit) . . . 28
- 4.3 Editing Graphs (Menu Item: Edit) . . . 29
- 4.4 LP Editor Dialogs (Menu Item: Edit) . . . 30
- 4.5 Composing Graphs (Menu Item: Compose) . . . 30
- 4.6 Graph Visualization (Menu Item: Layout) . . . 32
- 4.7 Problem Solvers (Menu Item: Optimize) . . . 33
- 4.8 Solver Configuration (Menu Item: Optimize) . . . 34
- 4.9 Browser Configuration (Menu Item: Browser) . . . 35

II Data Objects 37

5 Preliminary Statements 39
- 5.1 Some Conventions . . . 39
- 5.2 Base Types . . . 39
  - 5.2.1 Nodes . . . 39
  - 5.2.2 Arcs . . . 40
  - 5.2.3 Capacities . . . 40
  - 5.2.4 Floating Point Numbers . . . 40
  - 5.2.5 Handles . . . 40
  - 5.2.6 Matrix Indices . . . 40
  - 5.2.7 Class Local Types . . . 40
- 5.3 Bounds and Precisions of Numbers . . . 41
- 5.4 Ownership of Objects . . . 41

6 Graph Objects **43**
6.1 Abstract Classes . . . 43
6.1.1 Mixed Graphs . . . 43
6.1.2 Undirected Graphs . . . 44
6.1.3 Digraphs and Flow Networks . . . 44
6.1.4 Bipartite Graphs . . . 45
6.1.5 Balanced Flow Networks . . . 45
6.2 Persistent Objects . . . 46
6.2.1 Struct Objects . . . 46
6.2.2 Dense Graphs . . . 46
6.2.3 Sparse Graphs . . . 47
6.2.4 Sparse Bigraphs . . . 49
6.2.5 Planarity Issues . . . 49
6.3 Logical Objects . . . 50
6.3.1 Canonical Flow Networks . . . 51
6.3.2 Layered Auxiliary Networks . . . 52
6.3.3 Bipartite Matching Problems as Network Flow Problems . . . 53
6.3.4 General Matching Problems as Balanced Flow Problems 54
6.3.5 Layered Shrinking Networks . . . 55
6.3.6 Surface Graphs . . . 55
6.3.7 Suboptimal Balanced Flows . . . 56
6.3.8 Making Logical Objects Persistent . . . 57
6.4 Derived Persistent Objects . . . 57
6.4.1 Copy Constructors . . . 58
6.4.2 Mapping Back Derived Graph Objects . . . 58
6.4.3 Line Graphs and Truncation of the vertices . . . 58
6.4.4 Tearing Apart the Regions of a Planar Graph . . . 59
6.4.5 Complementary Graph . . . 60
6.4.6 Dual Graphs . . . 60
6.4.7 Spread Out Planar Graphs . . . 60
6.4.8 Metric Closure . . . 61
6.4.9 Distance Graphs . . . 61
6.4.10 Complete Orientation . . . 61
6.4.11 Induced Orientation . . . 62
6.4.12 Node Splitting . . . 62
6.4.13 Tilings . . . 62
6.4.14 Split Graphs . . . 63
6.4.15 Subgraph induced by a Node or Arc Set . . . 63
6.4.16 Bigraph induced by two Node Colours . . . 64
6.4.17 Colour Contraction . . . 64
6.4.18 Transitive Closure . . . 64
6.4.19 Intransitive Reduction . . . 65
6.4.20 Explicit Surface Graphs . . . 65
6.4.21 Voronoi Diagram . . . 65
6.4.22 Triangular Graphs . . . 66

7 Iterators **67**
7.1 Incidence Lists . . . 67
7.2 Iterator Objects . . . 67
7.3 Implicit Access . . . 68
7.4 Implementations . . . 69

8 Explicit Data Structures **71**
8.1 Container Objects . . . 71
8.1.1 Queues . . . 73
8.1.2 Stacks . . . 73
8.1.3 Priority Queues . . . 73
8.2 Disjoint Set Systems . . . 73
8.2.1 Static Disjoint Set Systems . . . 74
8.2.2 Shrinking Families . . . 74
8.3 Hash Tables . . . 75
8.4 Dictionaries . . . 75
8.5 Matrices . . . 76

9 Index Sets **79**
9.1 Interface . . . 79
9.2 Templates . . . 79
9.3 Graph Based Implementations . . . 80
9.4 Containers as Index Sets . . . 80

10 Branch and Bound **81**
10.1 Branch Nodes . . . 81
10.2 Generic Algorithm . . . 83
10.3 Implementations . . . 85
10.3.1 Stable Sets . . . 85
10.3.2 Symmetric TSP . . . 86
10.3.3 Asymmetric TSP . . . 87
10.3.4 Node Colouring . . . 87
10.3.5 Maximum Cut . . . 87
10.3.6 Mixed Integer Programming . . . 87

III Methods **89**

11 Prototypes and Data Structures **91**
11.1 Graph Definition . . . 91
11.1.1 Incidences and Adjacencies . . . 91
11.1.2 Arc Capacities and Node Demands . . . 92
11.1.3 Length Labels . . . 92
11.1.4 Geometric Embedding . . . 93
11.1.5 Layout . . . 93
11.1.6 Arc Orientations . . . 93
11.2 Potential Solutions . . . 94
11.2.1 Predecessor Labels . . . 94
11.2.2 Subgraphs . . . 95
11.2.3 Flow Labels . . . 95
11.2.4 Node Degrees . . . 96
11.2.5 Distance Labels . . . 96
11.2.6 Node Potentials . . . 97
11.2.7 Node Colours . . . 98
11.2.8 Partitions of the Node Set . . . 98
11.2.9 Blossoms . . . 99
11.2.10 Props and Petals . . . 99
11.2.11 Odd Cycles . . . 100
11.3 Manipulating Graphs . . . 100
11.3.1 Changes of the Incidence Structure . . . 100
11.3.2 Invalidation Policy . . . 101
11.3.3 Updates on the Node and Arc Labels . . . 101
11.3.4 Merging Graphs . . . 102

12 Graph Drawing **103**
12.1 Preliminary Remarks . . . 103
12.1.1 Layout Models . . . 103
12.1.2 Grid Lines . . . 104
12.1.3 Translations of the Current Drawing . . . 104
12.1.4 Automatic Alignment of Arcs . . . 104
12.2 Circular Layout . . . 104
12.3 Tree Layout . . . 105
12.4 Force Directed Placement . . . 105
12.5 Planar Straight Line Drawing . . . 106
12.6 Orthogonal Drawing . . . 107
12.7 Equilateral Drawing . . . 109

13 High Level Algorithms **111**
13.1 Shortest Paths . . . 111
13.1.1 Eligible Arcs . . . 112
13.1.2 Solver Interface . . . 112
13.1.3 Breadth First Search . . . 113
13.1.4 The Dijkstra Algorithm . . . 113
13.1.5 Discrete Voronoi Regions . . . 113
13.1.6 The Bellman-Ford Algorithm . . . 113
13.1.7 The FIFO Label-Correcting Algorithm . . . 113

13.1.8 The T-Join Algorithm . . . 114
13.1.9 The Floyd-Warshall Algorithm . . . 114
13.1.10 Proposed Extension . . . 114
13.2 Negative Cycles . . . 114
13.2.1 Negative Cycles . . . 115
13.2.2 Minimum Mean Cycles . . . 115
13.2.3 Proposed Extension . . . 115
13.3 DAG Search . . . 115
13.4 Euler Cycles . . . 116
13.5 Spanning Trees . . . 116
13.5.1 The (Enhanced) Prim Algorithm . . . 117
13.5.2 The Kruskal Algorithm . . . 118
13.5.3 Arborescences . . . 118
13.5.4 One Cycle Trees . . . 118
13.5.5 Tree Packings . . . 118
13.5.6 Proposed Extension . . . 118
13.6 Connected Components . . . 119
13.6.1 First Order Connectivity . . . 119
13.6.2 Strong Connectivity . . . 119
13.6.3 Second Order Connectivity . . . 119
13.6.4 Open Ear Decomposition and st-Numbering . . . 119
13.7 Planarity . . . 120
13.7.1 The Method of Demoucron, Malgrange and Pertuiset . 120
13.7.2 Combinatorial Embedding . . . 120
13.7.3 Outerplanar Embedding . . . 121
13.7.4 Connectivity Augmentation . . . 121
13.7.5 Canonically Ordered Partition . . . 121
13.8 Maximum Flows and Circulations . . . 122
13.8.1 The Augmentation Algorithm by Edmonds and Karp 123
13.8.2 The Capacity Scaling Algorithm . . . 123
13.8.3 The Blocking Flow Algorithm by Dinic . . . 123
13.8.4 The Push & Relabel Algorithm by Goldberg and Tarjan 123
13.8.5 Admissible Circulations and b-Flows . . . 124
13.8.6 Proposed Extension . . . 124
13.9 Minimum Cuts and Connectivity Numbers . . . 124
13.10 Minimum Cost Flows . . . 126
13.10.1 The SAP Algorithm by Busacker and Gowen . . . 127
13.10.2 The Refined SAP Algorithm by Edmonds and Karp . 128
13.10.3 The Cycle Canceling Algorithm by Klein . . . 128
13.10.4 The Minimum Mean Cycle Canceling Algorithm . . . 128
13.10.5 The Cost Scaling Algorithm . . . 128
13.10.6 The Multi Terminal SAP Method . . . 129
13.10.7 The Capacity Scaling Method . . . 129
13.10.8 The Primal Network Simplex Method . . . 129
13.11 Balanced Network Search . . . 130
13.11.1 The Algorithm by Kocay and Stone . . . 131
13.11.2 The Breadth First Heuristics . . . 131
13.11.3 The Depth First Heuristics by Kameda and Munro . . 131
13.11.4 The Algorithm by Micali and Vazirani . . . 131
13.12 Maximum Balanced Network Flows . . . 132
13.12.1 The Balanced Augmentation Algorithm . . . 132
13.12.2 The Capacity Scaling Algorithm . . . 132
13.12.3 The Phase-Ordered Algorithm . . . 132
13.12.4 The Cycle Canceling Algorithm . . . 132
13.13 Weighted Balanced Network Flow Algorithms . . . 133
13.13.1 The Primal-Dual Algorithm . . . 133
13.13.2 The Enhanced Primal-Dual Algorithm . . . 134
13.14 Matching Solvers . . . 134
13.15 T-Join and Postman Problems . . . 135
13.15.1 T-Joins . . . 136
13.15.2 The Undirected CPP . . . 136
13.15.3 The Directed CPP . . . 136
13.16 TSP Algorithms . . . 136
13.16.1 The Insertion Heuristics . . . 137
13.16.2 The Tree Approximation . . . 137
13.16.3 The Christofides Approximation . . . 137

13.16.4 Local Search . . . 138
13.16.5 The Subgradient Method by Held and Karp . . . 138
13.16.6 Branch and Bound . . . 139
13.16.7 Application to Sparse Graphs . . . 139
13.17 Graph Colourings and Clique Covers . . . 139
13.18 Stable Sets and Cliques . . . 141
13.19 Discrete Steiner Trees . . . 141
13.20 Maximum Edge Cuts . . . 142

IV Miscellaneous **145**

14 The Object Controller **147**
14.1 Construction . . . 147
14.2 Interaction with Data Objects . . . 147
14.3 Logging . . . 148
14.3.1 Event Handlers . . . 148
14.3.2 Writing Log Entries . . . 149
14.3.3 Structured Source Code . . . 151
14.3.4 Filtering the output . . . 151
14.3.5 Selection of logging information . . . 152
14.4 Method Selection . . . 153
14.4.1 Optional Data Structures . . . 154
14.4.2 Solver Options for NP-hard problems . . . 154
14.4.3 Problem Specific Solver Options . . . 155
14.5 Tracing . . . 156
14.5.1 Trace Level Options . . . 157
14.5.2 Tracing Data Structures . . . 157
14.6 Graphical Display . . . 158
14.6.1 Display Mode Options . . . 158
14.6.2 Export of Graphical Information . . . 158
14.6.3 Device Independent Layout . . . 160
14.6.4 Formatting Arc and Node Labels . . . 161
14.6.5 Arc Display Options . . . 162
14.6.6 Node Display Options . . . 162
14.6.7 General Layout Options . . . 163
14.7 Random Instance Generators . . . 164
14.8 Runtime Configuration . . . 165

15 The Messenger **167**
15.1 Problem Solver Management . . . 168
15.2 The Message Queue . . . 169
15.3 Tracing . . . 169

16 Linear Programming Support **171**
16.1 Public Interface . . . 172
16.1.1 Entry Point . . . 172
16.1.2 LP Instance Retrieval Operations . . . 174
16.1.3 LP Instance Manipulation . . . 176
16.1.4 Basis Dependent Methods . . . 177
16.1.5 Problem Transformations . . . 179
16.1.6 Solving Problems . . . 180
16.1.7 File I/O . . . 181
16.1.8 Text Display . . . 182
16.2 Native LP Solver . . . 184
16.3 GLPK Wrapper . . . 185

17 Ressource Management **187**
17.1 Memory Management . . . 187
17.2 Timers . . . 188
17.2.1 Basic and Full Featured Timers . . . 188
17.2.2 Global Timers . . . 189
17.2.3 Lower and Upper Problem Bounds . . . 189
17.3 Source Code Modules . . . 189
17.3.1 Authorship . . . 190
17.3.2 Bibliography Data Base . . . 190
17.4 Progress Measurement . . . 191

18 Persistency **193**
18.1 Export of Data Objects . . . 193
18.2 Import of General Data Objects . . . 193
18.3 Import of Graph Objects . . . 194
18.4 File Format for Graph Objects . . . 196
18.4.1 Definition . . . 197
18.4.2 Objectives . . . 198
18.4.3 Geometry . . . 199
18.4.4 Layout . . . 199
18.4.5 Potential Solutions . . . 199
18.4.6 Configuration . . . 200
18.5 File Format for Linear Programs . . . 200
18.6 Canvas and Text Form . . . 201
18.7 Support of Standard File Formats . . . 201
18.7.1 Import Filters . . . 202
18.7.2 Export Filters . . . 202

19 Exception Handling **203**

V GOBLIN Executables **205**

20 The GOSH Interpreter **207**
20.1 GOSH Ressources . . . 207
20.2 Context Variables . . . 207
20.3 Root Command . . . 208
20.3.1 Ressource Management . . . 209
20.3.2 Thread Support . . . 209
20.3.3 Messenger Access . . . 210
20.3.4 Accessing Timers . . . 210
20.4 General Object Messages . . . 211
20.5 Graph Retrieval Messages . . . 212
20.6 Graph Manipulation Messages . . . 212
20.7 Sparse Graphs and Planarity . . . 213
20.8 Graph Layout Messages . . . 214
20.9 Graph Node and Arc Messages . . . 215
20.9.1 Node Based Messages . . . 216
20.9.2 Arc Based Messages . . . 217
20.10 Graph Optimization Messages . . . 218
20.11 Derived Graph Constructors . . . 220
20.12 Messages for Undirected Graphs . . . 220
20.13 Messages for Directed Graphs . . . 221
20.14 Messages for Bipartite Graphs . . . 221
20.15 Messages for Balanced Flow Networks . . . 221
20.16 Linear Programming . . . 222
20.16.1 Instance Manipulation Messages . . . 222
20.16.2 Instance Retrieval Messages and Basis Access . . . 223
20.16.3 Row and Column Based Messages . . . 223
20.16.4 Row Based Messages . . . 224
20.16.5 Column Based Messages . . . 225
20.16.6 Optimization Messages . . . 225

21 Solver Applications **227**
21.1 Solver Applications . . . 227
21.1.1 Matching Problems . . . 227
21.1.2 Network Flow Problems . . . 227
21.1.3 Minimum Spanning Tree Problems . . . 228
21.1.4 Shortest Path Problems . . . 229
21.1.5 Chinese Postman Problems . . . 229
21.1.6 Other Solvers . . . 229
21.2 Linear Programming . . . 229
21.3 Random Instance Generators . . . 229
21.3.1 Random Digraphs . . . 229
21.3.2 Random Bigraphs . . . 230
21.3.3 Random Graphs . . . 230
21.4 Graphical Display . . . 230

VI Appendix **231**

22 Computational Results **233**
22.1 Symmetric TSP 233
22.2 Asymmetric TSP 235
22.3 Min-Cost Flow 236
22.4 Non-Weighted Matching 237
22.5 Weighted Matching 238
22.6 Cliques and Node Colouring 238

Part I

Introduction

Chapter 1

General Statements

1.1 Scope

GOBLIN is a `C++` class library focussed on **network programming problems**. Roughly speaking, a network programming problem is a graph optimization problem which can be solved efficiently by linear programming techniques. More explicitly, GOBLIN includes solvers for the following problems:

- Shortest paths
- Negative length cycles
- Minimum mean cycles
- Minimum spanning trees, arborescences and 1-trees
- Maximum packing with arborescences
- Maximum st-flows and min-cost st-flows
- Several types of minimum cuts and connected compenents
- Feasible [min-cost] circulations and b-flows
- Maximum cardinality and min-cost (perfect) assignments
- Directed Chinese postman problems
- Transportation problems
- Maximum cardinality and min-cost (perfect) matchings
- Undirected Chinese postman problems and T-joins
- (Weighted) b-matching problems
- (Weighted) f-factor problems
- (Weighted) capacitated b-matchings

The library also includes algorithms for some NP-hard problems in graph theory, namely:

- Δ-TSP, TSP and ATSP
- Stable sets, vertex covers and maximum cliques
- Graph colourings and clique partitions

There is a generic branch and bound module which is applied for the metric TSP solver and the computation of independent sets. Since GOBLIN does not support cutting planes, the solvers cannot compare with state-of-the-art codes for these problems, but should work for problems up to 100 nodes.

Release 2.6 comes with a basic LP simplex code and a generic interface for integration with more sophisticated LP solvers. So far, this module is utilized by the min-cost flow solver only. Branch and cut applications will follow.

1.2 History

GOBLIN is result from the *Deutsche Forschungsgemeinschaft (DFG)* research project *Balanced Network Flows*. This project is dedicated to the design, analysis and implementation of algorithms for generalized matching problems.

The extensive source code for network flow algorithms in GOBLIN is due to the strong dependencies between network flow and matching problems: Some of the matching algorithms explicitly require solvers for certain

network flow problems. Furthermore, the *layered shrinking graphs* which appear in our matching code reuse the *layered auxiliary networks* which form part of the well-known Dinic max-flow method.

1.3 Purpose and Applications

GOBLIN has been designed for researchers, developers, people who just need to solve network flow or matching (sub)problems, but also for educational purposes. Since the needs of all these potential users are sometimes contradictory, GOBLIN provides several configuration mechanisms, both at compile time and at runtime:

The GOBLIN runtime configuration includes the selection of logging information, of graph layouts, of tracing breakpoints and of the mathematical methods and the data structures which are used.

The graphical display together with the logging module allows the rapid preparation of adequate runtime examples for teaching and documenting network programming algorithms. Of course, this functionality is also helpful for the debugging of such algorithms.

Before GOBLIN is compiled, one may edit the file `config.h` in order to suppress the compilation of this GOBLIN functionality which is not needed for the final version of a problem solver, but which causes considerable computational overhead and large binaries.

Note that this compile time configuration is possible only with open source software. Hence the open source concept is an important prerequisite for the success of this project.

The library comes with source code for executable solver programs which support the runtime configurability. The experienced `C++` programmer, however, will find it easy to build GOBLIN problem instances immediately from his domestic data structures.

The library also comes with source code for a Tcl/Tk based interpreter `gosh` which can process complex scripts and user interactions, and with the graphical front end `goblet`. Both parts heavily depend on the open source Tcl/Tk library which must be installed to get the full functionality of GOBLIN.

1.4 Project Overview

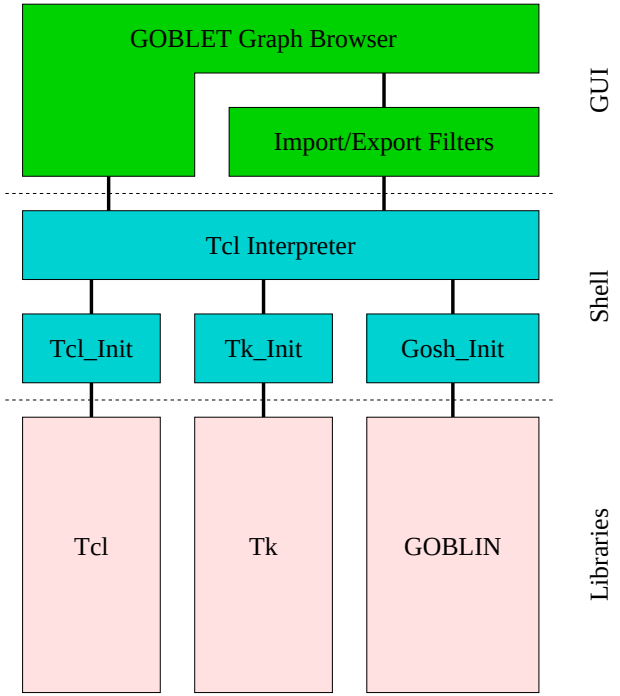


Figure 1.1: 3-Level Architecture

The GOBLIN programming project essentially splits into four parts each of which provides its own interface to the graph optimization methods of the GOBLIN library:

- The C++ Class library (64000 lines of source code)

- An extension of the Tcl/Tk shell sript language to graph objects (6000 lines)
- A graph browser and editor tool (13000 lines)
- Solver executables (2000 lines)

Here we have listed the respective source code sizes which may indicate the efforts of implementation. The Tcl wrapper is indeed a rather simple task and strongly recommended for other mathematical programming projects. Generally, the GOSH shell is compliant with other Tcl/Tk extensions. One only has to merge the project file `goshAppInit.cpp` and the other `AppInit` file. Alternatively, one can build a shared object and load the library dynamically into a Tcl shell.

This document describes the C++ API of the library functions and the Tcl wrapper extensively. The solver programs and the graph browser GOBLET are discussed within a few pages only.

1.5 Fundamental Library Concepts

The design of the GOBLIN library follows the object-oriented paradigm. This means a rather restrictive data encapsulation in order to obtain:

- compliance with other mathematical libraries, especially LP-Solvers.
- a user interface which is as simple as possible.

Merely the configuration parameters associated with controller objects are public.

The extensive use of polymorphisms leads to a class hierarchy which is adequate and intuitive from the point of view of mathematics: High-level methods are separated from data structures, and problem transformations are established by separate classes.

In general, the C++ implementation of mathematical algorithms is somewhat slower than straight C code. This stems from so-called **late binding operations** which assign a method name to a method implementation at runtime.

On the other hand, polymorphism eases the development and debugging of new algorithms a lot. Even more, this mechanism is compliant with the idea of open source projects where nobody is responsible for the correctness of the source code: Every new algorithm which uses an old part of the library is a certificate that this old code fragment works correctly.

We tried a careful trade off between a C and C++ like implementation. That is, to some extent we ignore the OO paradigm: Nodes and arcs are base types, and vectors are implemented as ordinary arrays.

We briefly describe the various classes of GOBLIN: The design distinguishes between graph objects, iterator objects, explicit data structures which are all **data objects**, and **controller objects** which allow to select solution methods as well as logging information and tracing points. Controllers also keep track of the dependencies among the various data objects.

The term **explicit data structure** shall indicate that such objects have a meaning which is independent from graph theory. Apart from this, there are implicit data structures such as incidence lists, subgraphs, distance labels etc. which are encapsulated in graph objects. The explicit data structures are discussed in Chapter 8, the implicit ones in Chapter 11.

The GOBLIN design is completed by **export** and **import** objects which manage the file interface of GOBLIN.

1.6 Mathematical References

Most GOBLIN algorithms are based on the textbooks

Network Flows
R.K.Ahuja, T.L.Magnanti, J.B.Orlin
Prentice Hall (1993)

Combinatorial Optimization
W.J.Cook, W.H.Cunningham, W.R.Pulleyblank, A.Schrijver
Wiley (1998)

Graphs, Networks and Algorithms
D.Jungnickel
Springer (1999)

and

Graphs and Algorithms
M.Gondran, N.Minoux
Wiley (1984)

The matching code is described in a series of papers of Christian Fremuth-Pager and Dieter Jungnickel:

Balanced Networks Flows (I): A unifying framework for design and analysis of matching algorithms. Networks, 33::1-28, 1999

Balanced Networks Flows (II): Simple augmentation algorithms. Networks, 33::29-41, 1999

Balanced Networks Flows (III): Strongly polynomial algorithms. Networks, 33::43-56, 1999

Balanced Networks Flows (V): Cycle canceling algorithms. Networks, 37::202-209, 2001

Balanced Networks Flows (VII): Primal-dual algorithms. To appear in Networks

which constitute part of the theoretical output of the mentioned DFG project.

1.7 Contributions

The core library has been written and is maintained by Dr. Christian Fremuth-Paeger (University of Augsburg). The same applies for this reference manual and the GUI application.

The following people have reviewed earlier versions of this reference manual: Dr. Andreas Enge (now at Ecole Polytechnique, Paris), Prof.Dr. Dieter Jungnickel and Priv.Doz. Bernhard Schmidt (both at the University of Augsburg).

Andreas Hefele (University of Augsburg) has tested Release 2.1, Bernhard Schmidt has tested the releases 2.2, 2.5 and 2.6. Markus Eisensehr (KUKA Controls Augsburg) and Bernhard Schmidt (University Augsburg) have tested the Windows XP setup of release 2.7.

Many thanks to Dr.Petra Huhn (University of Augsburg), Priv.Doz. Dirk Hachenberger (University of Augsburg) and Priv.Doz. Bernhard Schmidt for several helpful talks and their suggestions. Bernhard Schmidt has also contributed the GOBLET overview to this document.

Prof. Fernando de Oliveira Durao (Technical University of Lisboa) has prepared a self-installing GOBLET 2.5 package for Windows 98/2000/XP.

The tree packing method and the ATSP subgradient optimization which is new in GOBLIN 2.2 is written by Markus Schwank (University of Augsburg).

The basic LP simplex code which is attached to this release has been written by Priv.Doz. Bernhard Schmidt and integrated by Christian Fremuth-Paeger.

Birk Eisermann (University Augsburg) has contributed a planarity test, a makefile revision and a doxygen configuration file for release 2.6.2.

Further informations about code authors can be obtained by using the module browser in the GOBLET application.

Chapter 2

Installation

2.1 Licence Agreement

The GOBLIN core library was written by

Christian Fremuth-Paeger
Department of Mathematics
University of Augsburg, Germany

E-Mail: Fremuth@Math.Uni-Augsburg.DE

(C) Dr. Christian Fremuth-Paeger et al. 1998-2005

For details about the contributions by other authors see Section 1.7. All copyrights remain with the authors.

GOBLIN is open source software and covered by the GNU Lesser Public License (LGPL). That is, GOBLIN may be downloaded, compiled and used for scientific, educational and other purposes free of charge. For the details, in particular the statements about redistribution and changes of the source code, note the `LGPL` document which is attached to the package.

2.2 Software Requirements

To unpack and compile the GOBLIN library, the following software is necessary: `gzip`, `tar`, `gmake` and a C++ compiler. All of these programs should be available on any UNIX machine. We have tested the following environments:

- Suse Linux 7.3 with GNU C++ 2.95.3 (GOBLIN 2.6.4)
- Redhat Linux 7.3 with GNU C++ 2.96 (GOBLIN 2.5)
- Redhat Linux 8.0 with GNU C++ 3.2 (GOBLIN 2.5.3)
- Solaris 5.6 with GNU C++ 2.8.1 (earlier GOBLIN versions)
- Aix 4.3 with GNU C++ (previous GOBLIN versions)
- Aix 4.3 with xlC (GOBLIN 2.6.4)
- Cygwin 1.5.9 with GNU C++ 3.3.1 (GOBLIN 2.6.4)

To compile the graphical tool GOBLET, one also needs a Tcl/Tk installation and POSIX threads. You may check whether a Tcl/Tk interpreter is available on your UNIX system by typing `which wish`. Note that the compilation of the GOSH shell tool does not utilize the `wish` interpreter, but requires that the library files `libtcl.a` and `libtk.a` and the include files `tcl.h` and `tk.h` are installed correctly. It might be necessary to manually define links to make the Tcl/Tk library accessible to your C++ compiler.

To compile this reference manual, a `latex` installation is also needed. Finally, we recommend to install the graphical tools `xv` and `xfig` which supply GOBLET with several export filters, especially the postscript filter needed for printing. The `xfig` canvas drawing tool is useful for the postprocessing of figures also.

Problem solvers can be compiled and linked even if the Tcl/Tk package is not present, but the possible graphical output has to be processed manually then. In particular, the `.fig` files can be input to the `xfig` drawing tool.

2.3 Unpacking the Source

The source code is coming as a single zipped file `goblin.<version>.tgz` which can be extracted from a shell prompt by typing

```
tar xfz goblin.<version>.tgz
```

and then generates a folder goblin.¡version¿ including the Makefile. With elder `tar` versions, it may be necessary to extract the file in two steps:

```
gunzip goblin.<version>.tgz
tar xf goblin.<version>.tar
```

2.4 Configuration

Throughout this document, especially in Chapter 14, we will describe the runtime configurability of the core library. This section adresses some some possibilities for configuration at compile time by means of the source file `configuration.h`, and the general build options by means of `Makefile.conf`.

The latter file is intended to do the platform dependent settings. Currently, only Linux and Windows/Cygwin are well-supported. Advanced Unix users will find it obvious how to configure the compiler and linker for their own Unix platform. There are some more build parameters to set but some options are experimental, and the default values achieve the most stable code.

This `Makefile.conf` has been set up to run on a SuSE linux machine with default parameters. Cygwin and Aix are explicitly supported, that is, editing the platform specifier `os` in should be sufficient. Generally, before applying the `Makefile`, you have to edit some further lines in it. You may specify your compiler `CC` and `CXX`, the linker `LD`, the compression tool `zip` and the linkage names of the Tcl/Tk libraries `libtcl` and `libtk` which are installed on your machine. Probably, you need to change only some of of the defaults. If no Tcl/Tk libraries are available, you may build the GOBLIN library but not the the GOSH interpreter and the shared objects.

The file `configuration.h` contains some pragmas which may help to improve the performance and/or stability of the final C++ code. Probably it is not worth reading the following lines unless you encounter respective problems.

First at all in this file, the index types `TNode`, `TArc` are declared implicitly. You can choose from three different scalings by uncommenting one of the rows

```
// #define _SMALL_NODES_
// #define _BIG_ARCS_
// #define _BIG_NODES_
```

The scaling which is adequate for your purposes depends on the kind of problems you want to solve: A large scale (but solvable in a few minutes) spanning tree problem may have several 10000s of nodes, and hence requires the `_BIG_NODES_` pragma. On the other hand weighted matching problems which have a few 1000s of nodes, would require the `_BIG_ARCS_` pragma.

The default configuration is chosen to support the full functionality of GOBLIN. If you want to compile the final version of a problem solver, you may delete the pragma definitions

```
#define _LOGGING_
#define _FAILSAVE_
#define _TRACING_
#define _HEAP_MON_
```

from the file `configuration.h`. In our experience, this may decrease the running times by somewhat like 30 percent. The final code is much smaller, too. We mention what is lost if these pragmas are unset:

The `_LOGGING_` pragma filters only the low level logging information. The `_FAILSAVE_` pragma enables or disables most error detections, including wrong instrumentation of the C++ API and excluding some buffer overflows. This pragma seems to be the most important for code optimization,

but if your solver includes any bugs, you have to recompile the entire library to get some hints. The `_HEAP_MON_` define enables the compilation of special versions of `new` and `delete` and should be omitted in case of incompatibility with other C++ modules.

If the `_TRACING_` pragma is not present, the graphical display and the options `traceLevel>1` are disabled. The option `traceLevel==1` which helps you to decide whether your solver is still alive works even then. If the GOSH interpreter is compiled without the `_TRACING_` pragma, the GOBLET graph browser does not produce trace files.

2.5 The Makefile and UNIX Installation

The GOBLIN `Makefile` controls the compilation and linkage of the library, the GOSH shell tool and the executable solvers, the generation of this documentation and the generation of new GOBLIN packages, which either include all sources or binaries.

In what follows, it is supposed that your current working directory is the root directory of the source code distribution. The GOBLIN library `libgoblin.a` is then generated from console prompt by typing:

```
gmake goblin
```

As the next step of GOBLIN installation, generate the GOSH shell interpreter. For this goal, set in `Makefile` the variables `libtcl` and `libtk` to the Tcl/Tk versions installed on your machine, and then type

```
gmake
```

Similarly,

```
gmake shared
```

creates a shared object `libgoblin.so` which includes the core library functions in `goblin.a` plus the Tcl/Tk command registrations, and which can be dynamically loaded into the original `tclsh` shell. The call

```
gmake manual
```

produces the two files `mgoblin.<version>.ps` and `mgoblin.<version>.pdf`. This is the reference manual which you are just reading. The document can be viewed and printed by using:

```
ghostview mgoblin.<version>.ps &
```

or

```
acroread mgoblin.<version>.pdf &
```

Once the shell tool is available, one can start the GUI by typing `./goblet`, but this works from the `Makefile` directory only. If you don't have root privileges, exectute the **personal installation** by typing

```
gmake private
```

or

```
gmake privclean
```

where the second command also deletes the C++ source files and the build resources. Add the new `bin` directory to your `PATH` variable and the `lib` directory to `LD_LIBRARY_PATH` in your user profile. To perform a **system installation**, become a super user and then type

```
gmake install
```

The default installation directories are `/usr/lib`, `/usr/include` and `/usr/bin`. Take care to set up these directories if you are working in a system other than linux. Any existing installation (if it is not too old) is properly removed from the system. One can manually delete a system installation by typing

```
gmake sysclean
```

or, if the `Makefile` is not available, by executing

```
sh /usr/bin/goblin_uninstall.sh
```

A binary distribution, say `goblin.<version>.<platform>.tbz2`, is installed as follows: Become a super user, copy the archive to the file system root directory `/`, change to this directory and type in

```
tar xfj goblin.<version>.<platform>.tbz2
```

and then

```
sh goblin_install.sh
```

The `gmake install` command discussed before exactly generates such a binary distribution (via the `gmake bin` option) and then tries to execute the `goblin_install.sh`. So there is good hope that your package will install also on other machines. If you have made changes to the source code, you may like to bind a new GOBLIN tarball by typing:

```
gmake pkg
```

The resulting package includes the source code for the library and the executables, the latex sources for the reference manual including the figures, the tk scripts, the definition files for the examples, and a file `doku/history` which keeps track of the ancestor tarballs.

This GOBLIN package contains some source code which helps to generate executable solvers for various optimization problems. In the same manner, one can obtain some instance generators. The respective project names are listed in Table 2.1 and coincide with the file names for the main routines. If you just need a problem solver, say `optflow`, you may generate this executable by typing:

```
gmake exe pr=optflow
```

For all purposes, `gmake` must be called from the GOBLIN root directory where the produced files can be found. On linux computers and in a CYGWIN environment, one can type `make` instead of `gmake`. On other UNIX platforms, `make` possibly cannot interprete the `Makefile`.

Project Name	Purpose
`optmatch`	All kinds of matching problems
`optflow`	Max-Flow, feasible b-flows and min-cost flows
`postman`	Directed and undirected Chinese Postman Problem
`mintree`	Minimum spanning trees and 1-trees
`gsearch`	Shortest paths and shortest path trees
`connect`	(Strongly) connected components
`opttour`	Heuristics and lower bounds for the (metric) TSP
`colour`	Heuristic colouring
`optbflow`	Maximum and min-cost balanced st-flows

Table 2.1: Executable Solver Programs

2.6 Tcl Compatibility Issues

Generally, GOBLIN can be linked with every Tcl/Tk 8.x release. Since Tcl/Tk 8.4, a minor patch of the `Makefile.conf` is necessary: Activate the define

```
tcl_flags = -D_CONST_QUAL_="const"
```

to compensate some changes of the Tcl prototypes between the releases 8.3 and 8.4.

2.7 Cygwin Build

Cygwin is an environment which admits to compile and/or run Unix software on Windows machines. Similar to Linux distributions, Cygwin can be downloaded from internet and installed online. A setup program can be found at:

```
http://www.redhat.com/download/cygwin.html
```

The first manual and non-trivial step is to choose from a large set of module packages. In view of the later GOBLIN installation, select the following packages:

- `gmake`
- `gcc` and `gpp`, including the libraries
- `TclTk` (for building the gosh shell)
- `X11devel` (included by the Tcl/Tk header)
- `transfig`, `ghostscript` and `netpbm` (only for the graphical export of images from GOBLET)

The setup will detect package dependencies and hence add a lot of further packages to your selection. So far, `netpbm` does not form part of the standard Cygwin installation and hence must be downloaded separately. It is not required to install a `X` server.

In a final installation step, one has to extend the Windows system variables: Provided that the Cygwin installation directory is `c:\cygwin`, the `Path` system variable must be extended by a sequence

```
;c:\cygwin\bin;c:\cygwin\usr\X11R6/bin
```

and an environment variable

```
HOME=c:/cygwin/home
```

should be added. Now, Windows is prepared to build the GOBLET graph browser. Before compiling the gosh interpreter, just set `os = cygwin` in the `Makefile`. Then start a `bash` shell and follow the description of the previous section.

Starting with Release 2.7, we will also distribute Cygwin binaries with each major build. This makes some of the comments obsolete, but the packages `TclTk`, `transfig`, `ghostscript` and `netpbm` are still required. Start a `bash` shell or command prompt, copy the downloaded file to the Cygwin (not Windows!) root directory, change to this directory in the shell and type in:

```
tar xfj goblin.<version>.tbz2
```

2.8 Windows Setup Package

There are currently some efforts to make GOBLIN run out of the box on Windows machines. The preliminary setup which is available consists of a compact Cygwin environment, not just a Cywin dll. Unfortunately, this package does not run with any concurrent Cygwin installation because of the `path` variable extensions. Especially, `latex` makes trouble.

To be safe with other programs running Cygwin, check the Windows registry for `cygwin` keys and values before executing the setup. If you have trouble when starting the GOBLET graph browser, check the `path` directories for other `cygwin1.dll`'s and occasionally change the order of directories.

If you are already working with Cygwin, do not run the setup but revert to the description of the previous section.

Since the `tar` and `bunzip2` tools forms part of the GOBLIN setup, an existing installation can be 'patched' with subsequent versions of the `goblin.<version>.cygwin.tbz2` binary distribution (It is not really a patch since all GOBLIN specific files will be replaced).

We mention that there are intrinsic problems with file names including blanks. The graph browser can handle this in the most cases but we did not find a way to save a GIF bitmap to a file in `Documents and settings` yet.

2.9 Download of new GOBLIN Versions

New versions of GOBLIN will be distributed via the internet, URL:

```
http://www.math.uni-augsburg.de/opt/goblin.html
```

The project is presented at

```
http://www.freshmeat.net
```

under the project name `goblin`. By subscribing to the project, you obtain regular infos about updates via e-mail. Do not miss to make a project rating!

2.10 Bug Reports

The authors appreciate any kind of suggestions and bug reports. E-mail to:

`goblin@math.uni-augsburg.de`

In the folder `project` of this installation, you can find a form for bug reports.

Chapter 3

Getting Started

This chapter will give you a first idea of how GOBLIN can apply to your own graph optimization problem. More explicitly, it describes the four different interfaces to the GOBLIN library functions by some instructive examples.

3.1 The GOBLET Graph Browser

GOBLET is the graphical user interface to the GOBLIN library. It can be used to edit graphs, to configure the core library, to run problem solvers and to view the computational results. This graphical output can be printed, and exported to bitmaps but also to canvases.

First, try the following example: Change to the root directory of the GOBLIN installation [1] and type in:

```
goblet samples/strong4
```

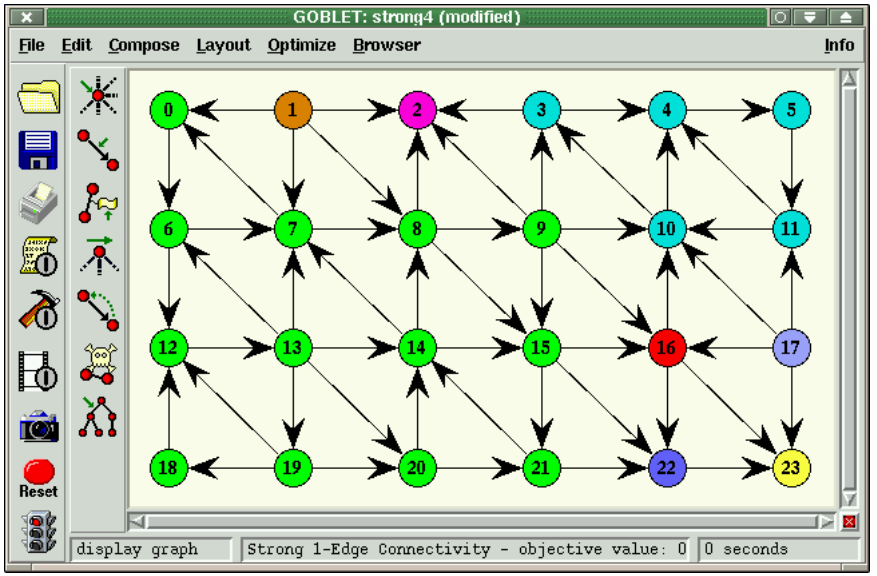
Up to the missing node colours, the browser starts with a screen as depicted in Figure 3.1. The main window is structed as follows:

- The leftmost icon bar refers to general tools for file management, switches for the various operating modes, a reset button for the messenger and a start/stop button for the problem solvers. By clicking on the camera, you can save the current graph object into a trace image. This tool bar is always available.
- The second icon bar and the canvas region form the built-in graph editor. The editor mode is default, but if no geometric embedding is available, GOBLET starts with the messenger window instead.
- The bottom line displays the operating mode, some status info depending on the operating mode and, rightmost, an info about the usage of ressources.

Now click on the `Optimize` menu and, in that menu, select

```
Connectivity... -> Strong Edge Connectivity -> Go
```

If nothing went wrong, the configuration shown in Figure 3.1 results in which the strong components are represented by node colours.



[1] If you are working with a system installation, you can download and unpack the sources to get access to the samples library.

Figure 3.1: GOBLET

Next, type `Control-d` in order to switch to the navigator mode. You now can access a couple of images which illustrate the process of computing the strong components. More explicitly, these images show the iterated depth first search trees. If you like, you can print any of the displayed images by typing `Control-p`.

If you do not like to generate such intermediate results, you can turn off the tracing functionality by selecting:

```
Browser -> Tracing Options...
    -> No Tracing -> Reset -> Done
```

If you want to see a descriptive log file, select

```
Browser -> Logging Options...
    -> Detailed Information -> Done
```

restart the computation by `Control-c` and display the log file by `Control-l`. In this example, the logfile does not provide much additional information compared with the figures. In general, it contains informations about recursive method calls, search orders, variable assignments and, which is also helpful, about the writing of trace images.

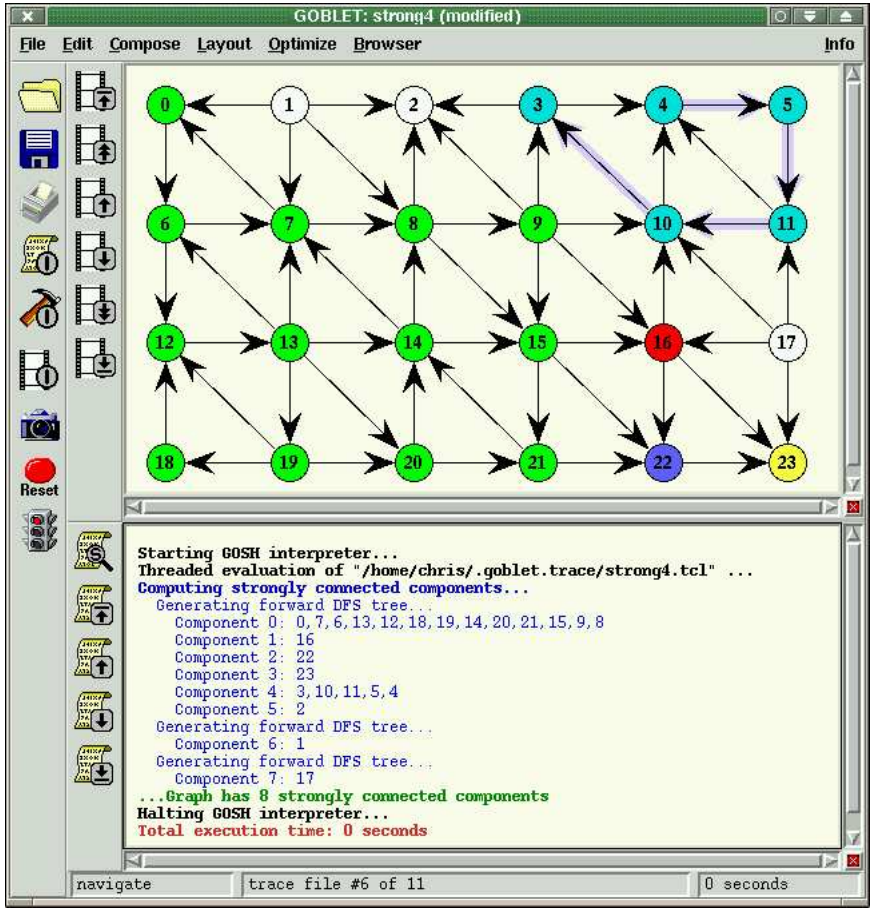


Figure 3.2: GOBLET Browser

Suppose you want to make the graph strongly connected. You can add some arcs by selecting `Edit -> Insert Arcs`. For example, click with the left mouse button on the node 23, place some interpolation nodes, and then click on the node 20. Finally, you are asked to specify the placement of arc labels (click with the left button again) which is immaterial in this example.

These manipulations result in a new graph arc $(23, 20)$. You may run the computation with `Control-c` again, and observe that the number of strong components effectively reduces. Try and find out how many arcs must be added to the original graph to make it strongly connected!

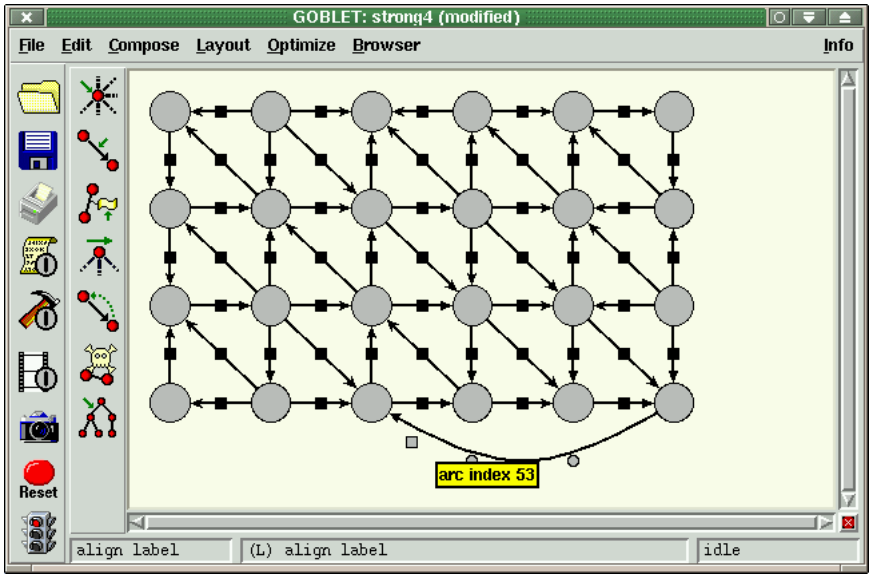


Figure 3.3: GOBLET Editor

We do not give a complete description of the GOBLET editor tool here. The status line helps you stepping through the chosen editor function. The most GOBLET menus and dialogs are intuitive, and this document describes the various components of the GOBLIN library rather than the tool GOBLET.

Note that GOBLET may handle graph objects without any geometrical embedding, but does not provide sophisticted tools for graph layout. Be careful when tracing a computation: Without any special effort, GOBLET may generate several thousands of files and, by that, cause a collapse of your file system.

3.2 GOSH Shell Scripts

The GOSH shell script interpreter extends the well known Tcl/Tk script language by the possibility of defining and manipulating graph objects. Tcl/Tk is an excellent tool to prepare prototype algorithms, instance generators and import/export filters with a minimum of code and effort.

Example:

```
set n [lindex $argv 0]

goblin sparse graph G

for {set i 1} {$i<=$n} {incr i} {
    for {set j [expr $i+1]} {$j<=$n} {incr j} {
        set node($i-$j) [G node insert]

        for {set k 1} {$k<$i} {incr k} {
            G arc insert $node($i-$j) $node($k-$j)
        }

        for {set k [expr $i+1]} {$k<$j} {incr k} {
            G arc insert $node($i-$j) $node($i-$k)
        }

        for {set k 1} {$k<$i} {incr k} {
            G arc insert $node($i-$j) $node($k-$i)
        }
    }
}

set FileName [file rootname [lindex $argv 1]]
G write "$FileName.gob"
G delete
```

```
exit
```

This script generates so-called **triangular graphs** which are interesting for their regularity. The message `goblin sparse graph G` instanciates a graph object `G` which is written to file and disallocated again by the messages `G write` and `G delete` respectively. Before file export, some node and arc insertion operations occur which will not be explained in detail.

Example:

```
set fileName [file rootname [lindex $argv 0]]

set file [open "$fileName.max" r]
goblin sparse digraph G
set n 0

while {[gets $file thisLine] >= 0} {
    if {[scan $thisLine "p max %d %d" n m]==2} {
        for {set i 1} {$i<=$n} {incr i} {G node insert}
    }

    if {[scan $thisLine "n %d %s" u type]==2} {
        if {$type=="s"} {set source [expr $u-1]}
        if {$type=="t"} {set target [expr $u-1]}
    }

    if {[scan $thisLine "a %d %d %d" u v cap]==3} {
        if {$n==0} {
            puts "File conversion failed!"
            exit 1
        }

        set a [G arc insert [expr $u-1] [expr $v-1]]
        G arc $a set ucap $cap
    }
}

close $file

if {$source=="*" || $target =="*"} {
   puts "Missing source and/or target node!"
   exit 1
}

G maxflow $source $target
```

This script reads a graph from a foreign file format, namely the DIMACS max flow format, and computes a maximum *st*-flow.

These two examples illustrate how graph objects can be manipulated easily from within a Tcl/Tk/GOSH script. On the other hand, the variable substitution is sometimes difficult to read, and long scripts are more difficult to handle than equivalent C++ code.

3.3 Using the Library

The bulk of this reference manual deals with the C++ library objects and methods. This is so since direct application of the library produces the most efficient code. Of course we also want to give other researchers the opportunity to develop the GOBLIN library further.

Example:

```
graph G((TNode)0,(TOption)0);
TNode **node = new TNode*[n];

TNode i = NoNode;
for (i=0;i<n;i++)
{
```

```
        node[i] = new TNode[n];
        TNode j = NoNode;
        for (j=i+1;j<n;j++)
        {
            node[i][j] = G.InsertNode();

            TNode k = NoNode;
            for (k=0;k<i;k++)
                G.InsertArc(node[i][j],node[k][j]);
            for (k=i+1;k<j;k++)
                G.InsertArc(node[i][j],node[i][k]);
            for (k=0;k<i;k++)
                G.InsertArc(node[i][j],node[k][i]);
        };
    };

    delete[] node;
```

This C++ code is equivalent to the described GOSH script given before which generates a triangular graph for a set with n elements. Using this instance generator as a benchmark indicates that C++ code is almost five times faster than equivalent Tcl code.

3.4 Solver Executables

GOBLIN comes with source code for solver executables. These main routines do not cover the entire GOBLIN functionality, but only the most frequently asked standard problem solvers. To work with these solvers, you must compile them separately (see Section 2.5).

You can customize the main routines which are distributed with the GOBLIN source code to your own convenience without much effort. This is probably the easiest way to become familiar with the library.

But note that every additional binary may include a lot of library functions, and hence require a lot of disk space. Moreover, if you want to call a GOBLIN solver from another C/C++ program, you may waste a lot of cpu time and disk space for the file export and import.

Chapter 4

The GOBLET Graph Browser

This chapter gives an rough overview about the graphical front end of the GOBLIN library. The GOBLET browser has been designed to test and debug new implementations of graph algorithms, to visualize standard graph methods in undergraduate courses and just to play with the combinatorial structure of graphs.

Intentionally, GOBLET is no graph drawing software. But in order to have a self-contained tool, we have added a graph editor. All graph layout methods provided by the core C++ library can be accessed from the GUI.

The GOBLET tool utilizes the graphical filter software `fig2dev`, `ghostscript` and `transfig` which supply to GOBLET an almost universal export filter. This allows to prepare figures for latex documents which can be included directly or post-processed by the canvas drawing tool `xFig`.

Every table lists a single pull-down menu. There are no inline descriptions of how the tools work but only references to the C++ API functionality for each item. A user manual would be more gentle, but many features are still floating and it is hard to keep this document up to date.

One menu is missing, namely the info menu which provides the problem statistics and system ressources info: The statistics dialog gives some insight about problem type, dimensions and numerics. In any case of trouble, consult the problem statistics and the GOSH transscript. The system ressources info displays some information about the heap (dynamic) memory occupied by GOBLIN.

Note that the browser does not support the entire GOBLIN functionality but somewhat like 95 percent. For example, the matching solver can only be fed with one degree sequence while the C++ API allows to specify upper and lower bounds on the node degrees.

We mention that one can solve moderate size optimization problems without much knowledge of the library, but it requires some care and experience to produce graphical output which is useful for teaching purposes. Then it is the combination of trace objects and the messages which is instructive.

4.1 File Management (Menu Item: File)

Option	Shortcut	Effect	Section
New...		Generate a new graph object or linear program	
Open...	**Ctrl+o**	Read a graph object from file. Supported formats: GOBLIN, DIMACS, TSPLIB and STEINLIB problems. If the check button is unset, the current graph is replaced by the selected object. Otherwise, the graphs are merged	18.4, 18.7
Save	**Ctrl+s**	Write current graph object to a GOBLIN file	18.4
Print Object...	**Ctrl+p**	Print the current graph or trace object. Assign a shell print command	
Save as...		Export the current graph object or the selected trace object to file. The supported file formats include problem instances (GOBLIN, DIMACS, TSPLIB), solutions, bitmaps (GIF,JPEG) and canvasses (Postscript, EPS). If a trace file is exported to a GOBLIN file, the browser switches to the trace object as the current graph	18.7
Compression...		Specify the shell commands used for file compression and decompression	
Save Settings		Export the current configuration to the file `.goshrc` which is read when the GOSH interpreter is started	20.1
Quit	**Ctrl+q**	Quit GOBLET	

4.2 Graph Editor Dialogs (Menu Item: Edit)

Option	Shortcut	Effect	Section
Constant Labels...	**Ctrl+C**	Dialog for constant node and arc labelings	
Metrics		Choose edge length metrics (Only for dense graphs). Either explicit length labels are used during optimization or length labels are computed with respect to the selected metrics. GOBLIN supports Euclidian, Manhattan, coordinate maximum and spheric distances (as specified in the TSPLIB)	11.1.3
Delete Solutions...	**Ctrl+X**	Computational results can be deleted. This is important if algorithms support postoptimization but computation shall be started from scratch	
Extract Solutions...	**Ctrl+E**	Predecessor labels representing trees, 1-matchings or cycles can be extracted from the subgraph labels. Node colourings representing bipartitions and edge-cuts can be extracted from the distance labels.	11.2.1

4.3 Editing Graphs (Menu Item: Edit)

Option	Shortcut	Effect	Section
Insert Arcs	**Ctrl+a**	Left button click selects start node. Subsequent clicks place bend nodes. Final click selects end node. Then the arc label can be placed by another left button click. Alternatively, a right button click enables automatic alignment of the arc label (only available if no bends are present)	6.2.3
Insert Nodes	**Ctrl+v**	Left button click in unoccupied area inserts a new graph node	6.2.3
Redirect Arcs	**Ctrl+r**	For sparse graphs only: Left button click reverts any directed arc, right button click changes undirected edges into arcs and vice versa	6.2.3
Incidences → Reorder Manually	**Ctrl+i**	For sparse graphs only: Left button on a node steps over its incidences, right button click admits to change the ordering	6.2.3
Incidences → Planar Ordering		For sparse planar graphs: Compute a combinatorial embedding. This operation does not produce a plane drawing	13.7
Delete Objects	**Ctrl+x**	Left button click on existing graph nodes and arc labels deletes objects	6.2.3
Explicit Parallels		For sparse graphs only: Replace edges with non-unit capacity labels by simple, parallel arcs	6.2.3
Move Nodes	**Ctrl+m**	Left button drag and drop graph nodes, arc labels and bend nodes	
Edit Labels	**Ctrl+e**	Left button click on nodes and arc labels opens a dialog to manipulate the labels which are associated with graph nodes and arcs	11.2
Set Colours		Left button click decreases, right button click increases the colour index of the highlighted node or edge	11.2.7
Set Predecessors		Left button click selects a node whose predessor arc can be deleted (click right button) or replace by another arc (click with left button on an adjacent node or incident arc)	11.2.1
Randomize → Add Arcs...		Add a specified number of random arcs to the current graph	
Randomize → Add Eulerian Cycle...		Add to the current graph a random Eulerian cycle of specified length	
Randomize → Make Graph Regular...		Complete the current graph to a k-regular graph. For this goal, the degrees in the current graph must not exeed k, and k must be even if the number of nodes is even	
Randomize → Random Generator...	**Ctrl+R**	Generate random labels for the existing graph nodes and arcs, and/or configure the random generator which is also used for arc insertions and the graph composition described in this menu	

4.4 LP Editor Dialogs (Menu Item: Edit)

Option	Shortcut	Effect	Section
Edit Columns...	**Ctrl+C**	Dialog for variable based data: Bounds, cost coefficients, labels. Mark variables as float or integers. Edit restriction matrix	
Edit Rows...	**Ctrl+R**	Dialog for restriction based data: Right hand sides, labels. Edit restriction matrix	
Reset Basis	**Ctrl+X**	Basis solution is set to the lower variable bounds	
Pivoting...	**Ctrl+P**	Perform pivoting steps manually	

4.5 Composing Graphs (Menu Item: Compose)

The composition methods in this pulldown menu generate a new object from the currently controlled graph object. The original graph is not manipulated.

Option	Effect	Section
Underlying Graph	Replace every class of parallel / antiparallel arcs by a single arc	6.4.1
Orientation → Complete	Replace every undirected edge by a pair of antiparallel arcs	6.4.10
Orientation → Induced by Colours	Orient every undirected edge from the lower to the higher colour index	6.4.11
Shrink Colours	Contract nodes by colours	6.4.17
Subgraph → By Node Colours...	Export the subgraph induced by a node colour	6.4.15
Subgraph → By Edge Colours...	Export the subgraph induced by an edge colour	6.4.15
Subgraph → Induced Bigraph...	Export the bigraph induced by a pair of node colours	6.4.16
Subgraph → Explicit Subgraph	Export subgraph into a separate object	6.4.1
Subgraph → Transitive Closure	For directed acyclic graphs only: Add all transitive arcs (arcs which represent non-trivial directed paths	6.4.18
Subgraph → Intransitive Reduction	For directed acyclic graphs only: Remove all transitive arcs	6.4.19
Complement	Switch to the complementary graph	6.4.5
Line Graph	Switch to the line graph	6.4.3
Node Splitting	Switch to the node splitting	6.4.12
Distance Graph	Generate a complete digraph where the length label of any arc uv is the length of a shortest uv-path in the original graph	6.4.9
Metric Graph	Undirected counterpart of the distance graph	6.4.8

Option	Effect	Section
Planar → Undirected Dual Graph	Switch to the dual graph (only for plane graph objects)	6.4.6
Planar → Directed Dual Graph	Switch to the directed dual graph (only for bipolar plane digraphs and for plane graphs with a given st-numbering)	6.4.6
Planar → Planar Line Graph	Replace all original nodes by faces of the same degree. The original arcs are all contracted to nodes	6.4.3
Planar → Truncate Vertices	Replace all original nodes by faces of the same degree, and keep the original edges connecting the new faces	6.4.3
Planar → Tear Regions Apart	Replace the original nodes by faces of the same degree, and the original edges by 4-sided faces	6.4.4
Planar → Tear & Turn Left / Right	As before, but triangulate the faces representing the original edges	6.4.4
Planar → Spread To Outerplanar	Requires an regular graph and a spanning tree. Double the tree arcs to obtain the exterior face of an outerplanar graph. The result is a cutting pattern for the original graph	6.4.7
Tiling...	Compose a graph from tiles. Open one of the templates `tile*.gob` in the example data base (folder `samples`). Specify the number of tiles in x and y-direction	6.4.13
Split Graph...	Swich to the skew-symmetric version of a network flow problem	6.4.14
Integer / Linear →	Generate the (integer) linear formulation for the selected graph based optimization model. Do not actually solve the ILP model	

4.6 Graph Visualization (Menu Item: Layout)

The operations in this pull-down menu manipulate the display coordinates of the current graph object. Partially, pure display entities such as arc bend nodes are added or deleted, and sometimes the order of node incidences are mainipulated to conform with the produced drawing.

Option	Shortcut	Effect	Section
Strip Geometry		Translate the node coordinates so that all coordinate are in the positive orthant. At least one x-coordinate and one y-coordinate are zero. **Attention:** Do not manipulate the geometrical embedding when working with spheric distances, use layout options instead	12.1.3
Scale Geometry...		Scale the geometric embedding to fit into a specified bounding box	12.1.3
Node Grids...		Configure separate, invisible grids for graph nodes, bend nodes and arc label alignment points. Objects are aligned with this grid during editor operations automatically. Optionally move existing nodes to the grid	14.6.7
Fit into Window	**Ctrl+w**	Fits the graph display into the GOBLET main window	
Zoom In	**Ctrl+**	Enlarge the graph display	
Zoom Out	**Ctrl-**	Lessen the graph display	
Planarity		Check for planarity, compute a combinatorial embedding explicitly, maximize the number of exterior nodes or compute a planar drawing from an existing combinatorial embedding	13.7
Force Directed Drawing → Unrestricted		Basically models the graph nodes as loaded particles and the graph arcs as springs. Searches for equilibriance of the nodes.	12.4
Force Directed Drawing → Preserve Geometry		Similar to the previous method but maintains edge crossing properties. That is, if the input is a planar straight line drawing, the result is a planar drawing with the same dual geometry	12.4
Align Arcs		Redraw arcs such that loops become visible and parallel arcs can be distinguished	12.1.4
Predecessor Tree		Manipulate the geometric embedding in order to expose a given tree of predecessor arcs	12.3
Circular Drawing		Draw all nodes on a cycle. The order is either given by the predecessor arcs or by the node colours	12.2
Orthogonal Drawing		Draw the graph on grid lines. The Kandinski model applies to general graphs. The other models are limited to planar graphs and/or small node degrees	12.6
Arc Display...	**Ctrl+A**	Specify the arc and arc label display	14.6
Node Display...	**Ctrl+N**	Specify the node and node label display	14.6
Layout Options...	**Ctrl+W**	Specify layout parameters (scaling, node and arrow size) without changing the geometric embedding and/or activate the graph legenda	14.6.7

4.7 Problem Solvers (Menu Item: Optimize)

This pulldown menu lists all available solvers for graph based optimization models. Calls to a solver can be interupted and / or repeated by pressing **Ctrl+c**. Before repeating a solver call, one can use the node context menus to select a different root, source or target node.

Option	Effect	Section
Vertex Routing → Minimum Spanning Tree	Compute a minimum spanning tree and return it by the predecessor labels	13.5
Vertex Routing → Maximum Spanning Tree	Compute a minimum spanning tree and return it by the predecessor labels	13.5
Vertex Routing → Travelling Salesman	Compute a minimum Hamiltonian cycle return it by the predecessor labels	13.16
Vertex Routing → Minimum 1-Cycle Tree	Compute a minimum 1-cycle tree and return it by the predecessor labels	13.16
Vertex Routing → Minimum Steiner Tree	Compute a minimum Steiner tree and return it by the predecessor labels. The terminal nodes are specified by the node demands	13.19
Edge Routing → Shortest Path Tree	For a given source node s, compute a shortest s-path tree. If a target node t is specified, the computation stops once a shortest st-path has been found. The results are returned by the predecessor and the distance labels	13.1
Edge Routing → Residual Shortest Path Tree	For digraphs only: Similar to the previous operation, but search the residual network as it occurs in min-cost flow algorithms	13.1
Edge Routing → Critical Path	For directed acyclic graphs only: Compute a forest such that every node is reached from a root node by a maximum length path	13.2.3
Edge Routing → Maximum st-Flow	For digraphs only: Compute a maximum st-flow. Return the subgraph and a minimum st-cut by the distance labels. A subgraph must be given in advance which satisfies the node demands other than for s and t (usually the zero flow)	13.8
Edge Routing → Feasible b-Flow	For digraphs only: Compute a subgraph which satisfies all node demands	13.8
Edge Routing → Minimum Cost st-Flow	For digraphs only: Compute a maximum st-flow of minimum costs. Return the optimal subgraph and node potentials. A subgraph must be given in advance which satisfies the node demands other than for s and t and which is optimal among all st-flows with the same flow value (usually the zero flow)	13.10
Edge Routing → Minimum Cost b-Flow	For digraphs only: Compute a subgraph of minimum costs satisfying all node demands. Return the optimal subgraph and node potentials	13.10
Edge Routing → Eulerian Cycle	Check if the graph object is Eulerian. Occasionally return an Eulerian walk by the edge colours	13.3
Edge Routing → Minimum Eulerian Supergraph	Increase the capacity bounds so that the graph becomes Eulerian	13.15

Option	Effect	Section
Bipartitions → Maximum Edge Cut	Compute a maximum capacity edge cut and return it by the node colours	13.20
Bipartitions → Maximum Stable Set	Compute a maximum stable set and return it by the node colours	13.18
Bipartitions → Minimum Vertex Cover	Compute a minimum vertex cover and return it by the node colours	13.18
Bipartitions → Maximum Clique	Compute a maximum clique and return it by the node colours	13.18
Graph Partitions...	Compute a node colouring, a cover with node disjoint cliques or an edge colouring. Optionally, the number of sets can be restricted	13.17
Connectivity...	Compute the (strongly) (edge) connected components for a given degree of connectivity or determine some connectivity number	13.6, 13.9
Matching Problems...	Compute a maximum capacitated b-matching, a minimum cost perfect b-matching, a optimal T-join or a minimal Eulerian supergraph (Chinese Postman Problem). The vector b and the set T are determined by the current node demands	13.14, 13.15
Ordering Problems → st-Numbering	Compute an st-numbering and return it by the node colours	13.6
Ordering Problems → Topologic Order	For directed acyclic graphs only: Compute a topological order and return it by the node colours	13.2.3
Balanced Network Flows...	Compute (min-cost) maximum balanced st-flow for a given source node s or (min-cost) st-flow. The sink node t is determined by the graph symmetry	13.12, 13.13
Solve LP Relaxation	Solve a linear program, neglect all integrality requirements	16.1.6

4.8 Solver Configuration (Menu Item: Optimize)

Option	Shortcut	Effect	Section
Restart/Stop Solver	**Ctrl+c**	Resolve problem with the same parameters or stop the current computation	15.1
Optimization Level...	**Ctrl+O**	Restrict the computational efforts when solving NP-hard problem. Attention: Candidate sets work for weighted matching problems also.	14.4.2
Method Options...	**Ctrl+M**	Configure the various problem solvers	14.4.3
Data Structures...	**Ctrl+S**	Select from alternative data structures for priority queues, union-find processes and node adjacencies	14.4.1

4.9 Browser Configuration (Menu Item: Browser)

Option	Shortcut	Effect	Section
Toggle Editor/Navigator	**Ctrl+d**	Switches from edit mode to display mode or switches between edit and navigation mode	
Snapshot Image	**Ctrl+t**	Generates a new trace image and switches to navigation mode	
View/Update Messenger	**Ctrl+l**	Open messenger window	
Tracing Options...	**Ctrl+T**	Configure the tracing module. That is, specify how often trace objects are generated	14.5
Browser Options...	**Ctrl+B**	Configure the browser, especially the file handling and windowing features	
Logging Options...	**Ctrl+L**	Specify which amount of logging information shall be written by the problem solvers	14.3
Save Browser Options		...to a file in the `.goblet` folder	

Part II

Data Objects

Chapter 5

Preliminary Statements

5.1 Some Conventions

Before we start the description of data objects, we give some general remarks about GOBLIN files, classes and methods which are omitted later in this document.

- If not stated otherwise, any operation is **elementary** , that is, it takes only a constant number of computing steps. Sometimes operations take constant time in practice, but an exact statement about their theoretical complexity is beyond the scope of this document, and therefore omitted.
- If not stated otherwise, the amount of computer storage required by any algorithm is proportional to the number of arcs or less.
- A GOBLIN source code file contains the definition of a single class, and the file name ends with `.cpp`. This class is declared in the header file whose name only differs by the extention `.h`.

  As an exception, a class definition may include a method of another class if this method instanciates the former class, so that an external definition would only complicate the dependencies among the source code files. For example, the TSP branch and bound method is defined in `branchSymmTSP.cpp` which implements the branch node data structure.
- Iterators are declared with their graphs, but defined in a separate file whose name differs from the graph definition file name by a leading `i`.
- Every section starts with a listing of the declaration of the methods which are discussed. The header file where these methods are declared is listed likewise.
- If not stated otherwise, all listed methods are declared `public`.

5.2 Base Types

There are a few GOBLIN objects which are rather basic values than instances of a C++ class. The corresponding types can be configured at compile time. We just considered late binding and dereferencing to be too expensive operations at that low logical level.

5.2.1 Nodes

Graph nodes are distinguished by their indices which are integers of a special type `TNode`. The sequence of node indices associated with a graph is `0,1,..,n-1`, where `n` is a `protected` instance variable of every graph object. In addition to the nodes of a graph, a global constant `NoNode` is defined for the management of undefined node references. This constant appears in GOBLIN files and in GOBLET as an asterisk `*`.

In bipartite graphs, the node set splits into **outer nodes** and **inner nodes**. The outer nodes have the indices `0,1,..,n1-1`, the inner nodes have the indices `n1,n1+1,..,n1+n2-1`. Again, `n1` and `n2` are `protected` instance variables of a bigraph object, and satisfy `n==n1+n2`.

In balanced flow networks, nodes are arranged in **complementary pairs** which consist of one outer node and one inner node. The comple-

mentary node v of the node u can be obtained by the operation `v = (u^1)`, that is, by changing the least significant bit.

With every graph, up to three special nodes can be associated. These nodes can be accessed by the methods `Source()`, `Target()`, `Root()`. For physical objects but not for problem transformations, these nodes can be manipulated by the methods `SetSourceNode()`, `SetTargetNode()`, `SetRootNode()` respectively.

5.2.2 Arcs

Graph arcs are distinguished by their indices which are integers of a special type `TArc`. The sequence of arc indices is `0,1,..,2*m-1`, where `m` is a `protected` instance variable of every graph object. In addition to the arcs of a graph, a constant `NoArc` is defined for the management of undefined arc references. This constant appears in GOBLIN files and in GOBLET as an asterisk `*`.

With every arc, the reverse arc also exists. Both arcs have indices which differ in the least significant bit. **Forward arcs** have even indices, **backward arcs** have odd indices. This is arranged such that a reverse arc is computed by the operation `a2 = (a1^1)`. Note that labels are assigned to forward arcs only.

In balanced flow networks, arcs are arranged in **complementary pairs**. Complementary arcs differ by the second least significant bit, that is, a complementary arc is computed by the operation `a2 = (a1^2)`. Note that flow values are assigned to single arcs, but capacity labels and length labels are assigned to complementary arc pairs.

5.2.3 Capacities

Capacity labels and node demands are held in numbers of a type `TCap`. This may either be an integral type or a floating point type. We do not strictly exclude the possibility of non-integral capacities. But note that matching solvers require integral values.

There is a constant `InfCap` which represents infinite capacities. This constant appears in GOBLIN files an in GOBLET as an asterisk `*`.

5.2.4 Floating Point Numbers

Length labels, distance labels, flow values and subgraph labels are held in floating point numbers of a type `TFloat`. There is a constant `InfFloat` which is used for undefined values and appears in GOBLIN files and in GOBLET as an asterisk `*`.

Explicit length labels are considered integral, and metric distances are rounded to integrality. Even if length and capacity labels are all integral, several algorithms (cost-scaling method for min-cost flow, subgradient optimization for TSP) deal with fractional node potentials and reduced length labels. Weighted matching algorithms deal with half-integral potentials, modified lengths and flow values. Note that the cost-scaling algorithm may end up with a suboptimal solution if the length labels are not integral.

5.2.5 Handles

Handles are integer numbers of a type `THandle` which are used to identify objects. There is a constant `NoHandle` to determine undefined handles which appears in GOBLIN files and in GOBLET as an asterisk `*`.

5.2.6 Matrix Indices

General matrix indices are integer numbers of a type `TIndex`. There is a constant `NoIndex` to determine an undefined index. When working with linear programs, two additional types `TVar`, `TRestr` occur with special constants `NoVar` and `NoRestr`. Although all three types are interchangable, the latter types are helpful to distinguish the primal respectively dual side of a linear program.

5.2.7 Class Local Types

Apart from these global base types, there are some more types which are used with a few methods only and which are declared within one of the root classes `goblinDataObject` and `goblinILPWrapper` respectively. Generally, the scope is obvious and not specified explicitly in this document.

5.3 Bounds and Precisions of Numbers

The length of matrix indices is an upper bound to the length of arc indices which in turn bounds the length of node indices.

Node indic values are bounded by the number of nodes in the corresponding graph object. This number of nodes is in turn bounded by the `maxNode` parameter defined in the context. Finally, `maxNode` is bounded by the constant `NoNode` which cannot be manipulated at runtime.

The method `goblinController::SetMaxNode` manipulates the `maxNode` parameter. There is a method `goblinController::SetMaxArc` which works in the same way for arc indices, the parameter `maxArc` and the constant `NoArc`.

The context variable `goblinController::epsilon` denotes the smallest number which is treated different from zero. It may apply in any situation where the numerical stability needs to be improved.

5.4 Ownership of Objects

Include file: `globals.h`
Synopsis:

```
class goblinAbstractObject
{

    enum TOwnership {OWNED_BY_SENDER, OWNED_BY_RECEIVER};
};
```

When passing an object pointer by a method call or returning an object pointer, it may be necessary for permanent access to specify which context owns the passed object:

- If the calling context specifies `OWNED_BY_SENDER` and passes an object pointer, the message receiver must make a copy of the object for permanent access.
- If the calling context specifies `OWNED_BY_RECEIVER` and passes an object pointer, the passed object is already a copy or not needed by the message sender any longer.
- If the calling context specifies `OWNED_BY_SENDER` and an object pointer is returned, the method instanciates a copy to which the returned pointer refers.
- If the calling context specifies `OWNED_BY_RECEIVER` and an object pointer is returned, the calling object either does not use the reference on the long run or makes a copy.

That is, the `SENDER` denotes the sender of the message rather than the sender of the object pointer.

Chapter 6

Graph Objects

Graph objects can be devided into three groups: **Abstract** classes which hold mathematical methods and prototypes for implementations, **persistent** classes which can be written to and read from a file, and **logical** classes which hold the problem transformations which are so important in network programming. Figure 6.1 shows the GOBLIN classes which model abstract graph objects and logical views.

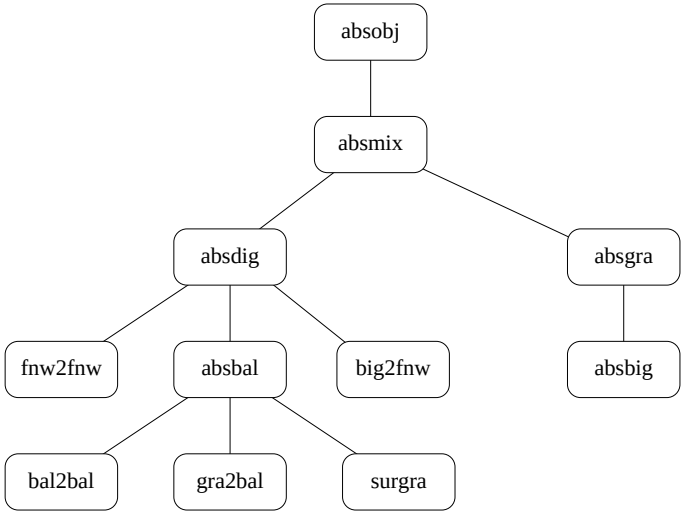


Figure 6.1: GOBLIN Base Classes

6.1 Abstract Classes

Abstract classes allow an high level description of solvers for graph optimization problems. They separate the fundamental algorithms from the data structures which are defined in dedicated classes called **implementation classes** or **concrete classes**.

Every abstract class definition is endowed with file export methods for problem instances and potential solutions. These methods are inherited by all implementation classes. That is, the external formats are implementation independent. Details can be found in Chapter 18.

6.1.1 Mixed Graphs

Include file: `abstractMixedGraph.h`

The class `abstractMixedGraph` is the base class for all graph structures. It handles the management of the implicit data structures which are listed in Table 6.1 and which will be discussed in Chapter 13. Roughly speaking, these data structures represent solutions of graph optimization problems whereas the graph defining data structures are implemented in the various concrete classes.

The first exception to this rule are subgraphs [flows] for which `absmix.h` provides some prototypes, but which are not implemented. This polymorphism results from the fact that subgraphs [flows] are subject to problem (back)transformations, and that dense graph objects should admit a sparse subgraph structure. Based on these prototypes, `abstractMixedGraph` implements methods to extract trees, paths, cycles and 1-matchings from a subgraph data structure.

On the other hand, one may think that adjacencies which are implemented in `abstractMixedGraph` constitute a graph defining data structure. But note that graph defintions are based on incidence lists, and that node adjacencies are defined explicitly just to speed up algorithms: If the context flag `methAdjacency` is enabled, the first call to `Adjacency` generates a hash table for efficient access to node adjacencies. That is, this first call requires $O(m)$ computing steps, but the other calls can be considered elementary operations. This data structure is not useful for dense implementations where the index of an adjacent arc can be computed directly from the node indices. Hence `Adjacency` is overloaded in some classes.

name	public access	description
`adj`	`Adjacency(TNode,TNode)`	Arcs connecting two nodes
`d`	`Dist(TNode)`	Distance labels
`nHeap`	—	Heap in cache
`P`	`Pred(TNode)`	Path predecessor labels
`partition`	`Find(TNode)`	Node partitions
`colour`	`Colour(TNode)`	Node colours
`pi`	`Pi(TNode)`	Node potentials
`sDeg`	`Deg(TNode)`	Node degrees in a subgraph

Table 6.1: Implicit Data Structures

Incidence lists are managed by iterator objects which allow to iterate on the node incidences. The class `abstractMixedGraph` declares prototype methods `First(TNode)` and `Right(TArc,TNode)` which admit a generic class of iterators. More explicitly, `First(v)` returns some arc with start node `v`, and `Right(a,u)` returns the **successor** of the arc `a` in the incidence list of the node `u`.

The class `abstractMixedGraph` also provides methods for the caching of iterator objects, and for the implicit access to graph iterator objects. More information about iterators can be found in Chapter 7.

One of the most important features of this class are the methods `Display()` and `TextDisplay()` on which the tracing of all graph objects depends.

Finally, the definition of `abstractMixedGraph` includes some mathematical methods. These are basic graph search procedures like BFS and other shortest path algorithms which essentially work on the complete orientation of mixed graphs, and methods which totally ignore the arc directions.

6.1.2 Undirected Graphs

Include file: `abstractGraph.h`

Abstract graphs inherit from abstract mixed graphs. Several optimization problems are associated with this class, namely all kinds of matching problems and minimum spanning tree problems, including the 1-tree problem. There are also some algorithms for the symmetric TSP and the metric TSP. The matching code and the Christofides heuristics are defined in the file `gra2bal.cpp`.

6.1.3 Digraphs and Flow Networks

Include file: `abstractDigraph.h`

Abstract digraphs inherit from abstract mixed graphs. This class contains only a few graph theoretical methods, but also models abstract flow networks which supply lot of additional functionality:

- Residual capacities `ResCap(TArc)`
- Node imbalances `Div(TNode)`
- Computation of path capacities `FindCap(TArc*,TNode,TNode)`
- Push operations `Push(TArc,TFloat)` and `AdjustDegree(TArc,TFloat)`
- Augmentation `Augment(TArc*,TNode,TNode,TFloat)`
- Max flow algorithms (Push-Relabel, augmentation, capacity scaling)
- Min cost flow algorithms (SAP, cycle canceling, cost-scaling, minimum mean cycles)

These methods are defined in the file `absdig.cpp`. There are further definition files including network flow algorithms which directly utilize a special problem transformation:

- `auxnet.cpp`: Defines layered auxiliary networks which form part of the well-known Dinic max flow algorithm. This file also defines the method `abstractFlowNetwork::Dinic(TNode,TNode)`.
- `fnw2fnw.cpp`: The reduction of circulation problems to *st*-flow problems: `abstractFlowNetwork::ShortestAugmentingPath(TNode,TNode)`.

6.1.4 Bipartite Graphs

Include file: `abstractBigraph.h`

Abstract bigraphs inherit from undirected graphs, and specify a bipartition by parameters `n1` and `n2`. This parameters can be accessed by the methods `N1()` and `N2()`. Nodes can be checked to be in one of the components by `Outer(TNode)` and `Inner(TNode)` respectively.

Bigraphs overload the matching algorithms by dedicated assignment algorithms. The file `big2fnw.cpp` which defines the reduction of assignment problems to network flow problems also defines these assignment methods. The remaining methods are defined in `absbig.cpp`, including specialized methods for colouring and stable sets.

6.1.5 Balanced Flow Networks

Include file: `abstractBalancedDigraph.h`

Abstract balanced flow networks inherit from digraphs, but have a certain symmetry based on the complementarity of nodes and arcs. The additional functionality is:

- Pairwise push operations (`BalPush(TArc,TFloat)`), symmetrical residual capacities (`BalCap(TArc)`).
- Symmetrical path capacities (`FindBalCap(TNode,TNode)`), pairwise augmentation (`BalAugment(TNode,TNode,TFloat)`).
- Balanced network search methods which constitute the balanced augmentation algorithm (Kocay/Stone, Kameda/Munro and other heuristics).
- Maximum balanced flow algorithms which essentially solve non-weighted matching problems (Anstee, Micali/Vazirani, augmentation, capacity scaling).
- Min-Cost balanced flow algorithms which essentially solve weighted matching problems (Primal-dual, primal-dual starting with min-cost flow optimum).

The new functionality is needed if one is interested in integral symmetric flows only. But this is the case for the reduction of matching problems which is implemented by the class `gra2bal`.

All class methods are defined in the file `absbal.cpp` except for the methods

```
abstractBalancedFNW::MicaliVazirani(TNode);
abstractBalancedFNW::BNSMicaliVazirani(TNode,TNode);
```

which are defined in the file `shrnet.cpp`, the method

```
void abstractBalancedFNW::PrimalDual(TNode)
```

which is defined in the file `surgra.cpp`, and the methods

```
void abstractBalancedFNW::CancelOdd()
void abstractBalancedFNW::CancelPD()
```

which are defined in the file `bal2bal.cpp`.

6.2 Persistent Objects

A data object is **persistent** iff it can be exported to a file, and reimported without (significant) loss of information. The file formats for persistent objects are described in Chapter 18.

To every abstract class of graph objects, potentially two persistent implementations exist: A **sparse implementation** based on incidence lists, and a **dense implementation** based on adjacency matrices. The persistent classes defined in GOBLIN are listed in Table 6.2.

class name	description	include file
`mixedGraph`	mixed graph objects	`"spsmix.h"`
`graph`	sparse graph objects	`"spsgra.h"`
`diGraph`	sparse digraph objects	`"spsdig.h"`
`biGraph`	sparse bigraph objects	`"spsbig.h"`
`balancedFNW`	sparse balanced flow network objects	`"spsbal.h"`
`denseGraph`	dense graph objects	`"dnsgra.h"`
`denseDiGraph`	dense digraph objects	`"dnsdig.h"`
`denseBiGraph`	dense bigraph objects	`"dnsbig.h"`

Table 6.2: Persistent Graph Objects

The definition of persistent classes is of little mathematical interest since algorithms are defined by abstract classes (where all object functionality is specified), and problem reduction principles are defined by dedicated logical classes. We concentrate on the specific data structures and functionality of persistent graphs.

6.2.1 Struct Objects

Include file: `graphStructure.h`

The various sparse graph object classes are not defined independent from each other but are composed of a `sparseGraphStructure` object. In the same way, dense graph objects are composed of `denseGraphStructure` objects.

Both of the mentioned classes inherit by a class `genericGraphStructure` in which most graph data structures are defined: Capacities, arc length labels, node demands, geometrical embedding into the plane, arc orientations. There are methods for loading a data structure, changing values, handling default values and bounds.

Note that persistent graphs do not inherit from `denseGraphStructure` or `sparseGraphStructure` objects, but have such a component object. This avoids multiple inheritance, but makes it neccessary to repeat the declaration of many methods in every class of persistent graph objects.

To avoid such repetitions, there are special include files named `geninc.h`, `spsinc.h`, `dnsinc.h` which declare the interface between struct objects and graph objects.

6.2.2 Dense Graphs

Include file: `denseStructure.h`
Synopsis:

```
class denseGraphStructure : public genericGraphStructure
{
    void        NewSubgraph(TArc);
    void        ReleaseSubgraph();
```

```
    TFloat      Sub(TArc);
    void        AddArc(TArc,TFloat);
    void        OmitArc(TArc,TFloat);
}
```

The data structures defined in `genericGraphStructure` are simply arrays with fixed dimensions. To handle sparse subgraphs in complete and geometrical graph instances, `denseGraphStructure` implements an optional hash table for subgraph labels. This data structure is generated by the first call of `AddArc` automatically. In that case, the number of arcs in the subgraph is restricted to the number of nodes which is satisfactory for working with trees, 1-trees, 1-matchings and 2-factors.

Alternatively, the subgraph data structure may be allocated explicitly by the method `NewSubgraph(TArc)` which takes the maximum size l as a parameter and requires $O(l)$ computing steps.

6.2.3 Sparse Graphs

Include file: `sparseStructure.h`
Synopsis:

```
class sparseGraphStructure : public genericGraphStructure
{
    TArc        First(TNode);
    void        SetFirst(TNode,TArc);
    TArc        Right(TArc);
    void        SetRight(TArc,TArc);
    TArc        Left(TArc);

    TNode       StartNode(TArc);
    TNode       EndNode(TArc);

    void        ReSize(TNode,TArc);

    TArc        InsertArc(TNode,TNode,TCap,TCap,TFloat);
    TArc        InsertArc(TNode,TNode);
    TNode       InsertNode();
    TNode       InsertArtificalNode();
    TNode       InsertAlignmentPoint(TArc);
    TNode       InsertBendNode(TNode);

    void        ExplicitParallels();

    void        SwapArcs(TArc,TArc);
    void        SwapNodes(TNode,TNode);
    void        FlipArc(TArc a);
    void        CancelArc(TArc);
    void        CancelNode(TNode);
    void        ReleaseBendNodes(TArc);
    bool        ReleaseDoubleBendNodes();
    void        ReleaseShapeNodes(TNode);
    void        DeleteArc(TArc);
    void        DeleteNode(TNode);
    void        DeleteArcs();
    void        DeleteNodes();
    void        ContractArc(TArc);
    void        IdentifyNodes(TNode,TNode);
}
```

In the class `sparseGraphStructure`, the node incidence list defining methods `First(TNode)` and `Right(TArc)` are implemented by own data structures. In addition to the general functionality of node incidence lists, sparse graph objects admit the following operations:

The **predecessor** of any arc in the incidence list of its start node is returned by the method `Left(TArc)`. An explicit data structure is gener-

ated from the successor labels by the first call of `Left` which therefore takes $O(m)$ computing steps. Subsequent calls are $O(1)$.

Node incidence lists can be sorted by a method `SwapArcs(TArc,TArc)`. The first arc on a list can be fixed by a method `SetFirst(TNode,TArc)`. Arc directions can be changed by a method `FlipArc(TArc)`.

`SetRight(a1,a2,a3)` makes a_2 the successor of a_1 in the start nodes incidence list and make the original successor of a_1 the new successor of a_3. If no thirst argument is specified, $a_2 = a_3$ is assumed. In any case, for `Left()` and `Right()` circular lists are maintained.

The methods `StartNode(TArc)` and `EndNode(TArc)` are implemented such that arrays for start nodes and end nodes of arcs are generated by the first request. Again, the first call of `StartNode` or `EndNode` takes $O(m)$ computing steps, but subsequent calls take $O(1)$ steps.

Since sparse graphs are usually grown from scratch (only file constructors work somewhat differently), the class `sparseGraphStructure` allows to predefine the final dimensions by the method `ReSize(TNode,TArc)` which effectively prevents the iterated reallocation of the data structures.

The insertion of an arc connecting the nodes with indices `v` and `w` is achieved by `InsertArc(v,w)`, `InsertArc(v,w,uc,ll)` or `InsertArc(v,w,uc,lc,ll)` respectively. Each of the methods return the index of the new arc. One may explicitly assign an upper capacity bound `uc`, a lower capacity bound `lc` and a length label `ll` to the new arc. If no labels are specified, the labels are set to default values or to random values depending on how the random generator is configured.

Once an arc is present, an alignment point for the arc label and bend nodes for the arc drawing can be defined by the methods `InsertAlignmentPoint()` and `InsertBendNode()` respectively.

Example:

```
TArc a = InsertArc(v,v);
TNode x = InsertAlignmentPoint(a);
TNode y = InsertBendNode(x);
TNode z = InsertBendNode(y);
```

introduces a new graph edge, namely a loop, which has two bend nodes y and z, and whose labels are drawn at the position of x. The coordinates have to be specified by the method `SetC` separately.

To delete arcs, the following operations are provided: `CancelArc(TArc)` which deletes an arc and its reverse arc from the incidence lists, and `DeleteArc(TArc)` which eventually deletes the canceled arc from all data structures. Note that the latter operation may change the index of other arcs, and hence must be applied very carefully. A call to `DeleteArcs()` eliminates all canceled arcs. This operation should not be used in algorithms intermediately but rather as a concluding step.

Similarly, a call to `CancelNode(TNode)` cancels all arcs incident with this node and `DeleteNodes()` eliminates all canceled and isolated nodes. The methods `DeleteNode(TNode)` and `DeleteNodes()` potentially change all node and arc indices.

Calling `ReleaseBendNodes(a)` eliminates the alignment point for the arc label and all bend nodes assigned with a. Similarly, `ReleaseShapeNodes(v)` deletes all artifical nodes assigned with the vertex v. The method `ReleaseDoubleBendNodes()` checks for pairs of consecutive bend nodes with the same position in a drawing and occasionally deletes some bend nodes. This check includes the end nodes of all edges.

The method `ContractArc(TArc)` merges the incidence list of the end node into the incidence list of the start node of the given arc and cancels the arc and the end node. If the incidence lists provide a planar embedding, the contraction preserves planarity.

More generally, `IdentifyNodes(x,y)` merges the incidence list of the node `y` into the incidence list of the node `x` and cancels node `y`. The nodes to be identified may be non-adjacent.

The method `ExplicitParallels()` splits the arcs a with capacity `UCap(a)>1` into a couple of arcs which all have capacity ≤ 1. The total upper and lower bounds as well as sum of potential flows remain unchanged.

6.2.4 Sparse Bigraphs

Include file: `sparseBigraph.h`
Synopsis:

```
class biGraph
{
    TNode       SwapNode(TNode);
}
```

For the manipulation of bigraph nodes, an additional method `SwapNode(v)` is provided which moves the passed node v to the other component. The return value is the new index of v, say u. Effectively, the nodes u and v are swapped. Deletions of outer nodes include an implicit `SwapNode()` operation.

6.2.5 Planarity Issues

Include file: `abstractMixedGraph.h`
Synopsis:

```
class abstractMixedGraph
{
    void    MarkExteriorFace(TArc);
    TArc    ExteriorArc();
    bool    ExteriorNode(TNode,TNode = NoNode);

    enum    TOptExtractEmbedding {
                PLANEXT_DEFAULT = 0,
                PLANEXT_GROW = 1,
                PLANEXT_DUAL = 2,
                PLANEXT_CONNECT = 3
            };
    TNode   ExtractEmbedding(
                TOptExtractEmbedding = PLANEXT_DEFAULT,
                void* = NULL);
    TNode   Face(TArc);
    TNode   ND();
    void    ReleaseEmbedding();
}
```

A graph is **planar** if it can be drawn in the plane without any edge crossings. If for every node v (and some virtual plane drawing), the arcs starting at v are listed in clockwise order by the incidence lists, the graph is called **combinatorially embedded**. In embedded graphs, the face left hand of a given arc a_0 can be traversed as follows:

Example:

```
TArc a = a0;
do
{
    a = Right(a^1);
    ...
}
while (a!=a0);
```

Supposed that the graph is connected, all faces are traversed counter clockwise, except for the exterior face which is traversed clockwise.

Every graph object may provide a combinatorial embedding from its own but only sparse graph objects admit manipulation of the incidence lists. Arc deletions and contractions maintain a combinatorial embedding but some care is necessary when edges are inserted into an incidence structure:

The idea is that arcs are always inserted into the exterior face. Calling `MarkExteriorFace(a)` sets the `First()` incidences appropriately so that inserting an arc into the face left hand of a will preserve the embedding. The arc a is saved as a representant of the exterior face and can be retrieved again by calling `ExteriorArc()`.

Of course, it is possible to mark a face exterior, insert edges and then revert to the original exterior face. The running time of `MarkExteriorFace()` is proportional to the number of arcs of the selected face.

Another effect of this method is that the face of a given arc a_0 can be traversed in the converse (usually clockwise) direction of the previous example, but only if the graph is biconnected:

Example:

```
TArc aExt = ExteriorArc();
MarkExteriorFace(a0);

TArc a = a0;
do
{
    TNode v = StartNode(a);
    a = First(v)^1;
    ...
}
while (a!=a0);

MarkExteriorFace(aExt);
```

There is no need to store a planar embedding in a special data structure persistently. If one calls `ExtractEmbedding()`, to every arc the left hand face is saved internally. The procedure also determines a face with a maximum number γ of incident edges, marks this face exterior and returns γ. The running time is $O(m)$ in the default setting.

If the graph is disconnected, the procedure processes each connected component separately and exports the connected components by the node colours. Note that for disconnected graphs, a distinction between **regions** (of the complement of plane drawing) and **faces** (cycles in the boundary of a region) is necessary and that this code handles faces rather than regions.

Depending on the optional parameters, the procedure performs additional operations:

- For `PLANEXT_DUAL`, the `void*` pointer is interpreted as a `abstractMixedGraph*` pointer to an empty graph which is filled with the dual incidence structure.
- For `PLANEXT_GROW`, the incidence lists are manipulated to obtain an embedding with the maximum of exterior nodes. In the extreme case, an outerplanar embedding results. Here the running time is $O(m^2)$ due to nested graph search for exterior separating edges.
- For `PLANEXT_CONNECT`, the connected components are linked such that in the resulting embedding, all original components are exterior. This graph augmentation effectively corrupts the face assignments. Hence a second pass with `PLANEXT_DEFAULT` would be necessary to rebuild the indices.

The number of faces is retrieved by `ND()` and the face left hand to a given arc a is obtained by `Face(a)`. If the embedding has not been extracted explicitly, `Face()` will initiate this operation in its first occurence. So, for connected graphs, a is an exterior arc if

```
Face(a) == Face(ExteriorArc())
```

provided that the graph is implicitly or explicitly embedded. In order to decide if a given node v is on the exterior face, one just calls `ExteriorNode(v)`.

Note that arc insertions and deletions call `ReleaseEmbedding()` and after that the dual incidences must be extracted again.

6.3 Logical Objects

Logical objects describe a special view of another object. Roughly speaking, a logical class defines the reduction mechanism of one optimization problem to another. The referenced object may either be persistent or logical.

Logical objects keep reference of the original object all of their lifetime. A referenced object may not be disallocated while logical views are present. The benefit is a hidden back transformation of potential solutions of the

respective optimization problems. More precisely, the potential solutions are merely logical views of solutions for the original problem.

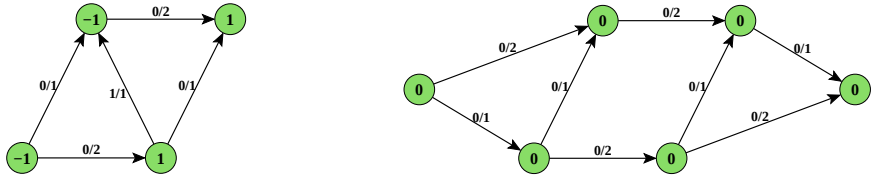


Figure 6.2: Transformation of Network Flow Problems

6.3.1 Canonical Flow Networks

Include file: `digraphToDigraph.h`
Synopsis:

```
class FNW2FNW : public virtual abstractDiGraph
{
    FNW2FNW(abstractDiGraph &);

    TNode       Source();
    TNode       Target();
    bool        Perfect();
}
```

A **canonical** network flow problem is a flow network whose lower capacity bounds are all zero and such that, except for a special node pair, all node demands are zero. The class `FNW2FNW` allows to transform a given network flow problem into an equivalent canonical problem. More explicitly, it manages:

(1) the reduction of the feasible circulation problem to the maximum *st*-flow problem,

(2) the reduction of the min-cost circulation problem to the min-cost *st*-flow problem.

The reduction technique is adding an artificial source node and an artificial target node, and adding some arcs to the network. The artificial nodes may be accessed by the respective methods `Source()` and `Target()`.

Any flow on the logical graph object corresponds to a pseudoflow of the referenced flow network which respects the capacity bounds. For example, a zero flow corresponds to a pseudoflow with `Flow(a)==LCap(a)`.

If a feasible b-flow (circulation) of the referenced networks exists, any maximum flow of the logical object will give such a b-flow. A maximum flow of minimum costs corresponds to a minimum cost b-flow then. Given any logical flow, it may be checked whether it maps to a feasible b-flow or not by a call to the method `Perfect()`.

The constructor method does not initialize the flow on the `FNW2FNW` object to zero, but to the image of the original flow. Some augmentation steps on the artificial arcs are done immediately which do not affect the flow on the original network.

Example:

```
G1 = new diGraph("sample.gob");
G2 = new FNW2FNW(G1);
G2 -> MaxFlow(G2->Source(),G2->Target());
if (G2->Perfect())
{
    F1 = new export("sample.rst");
    G1 -> WriteFlow(F1);
    delete F1;
}
delete G2;
delete G1;
```

6.3.2 Layered Auxiliary Networks

Include file: `auxiliaryNetwork.h`
Synopsis:

```
class layeredAuxNetwork : public abstractDiGraph
{
    layeredAuxNetwork(abstractFlowNetwork &,TNode);

    void            Phase1();
    void            InsertProp(TArc);

    void            Phase2();
    bool            Blocked(TNode);
    TFloat          FindPath(TNode);
    void            TopErasure(TArc);
}
```

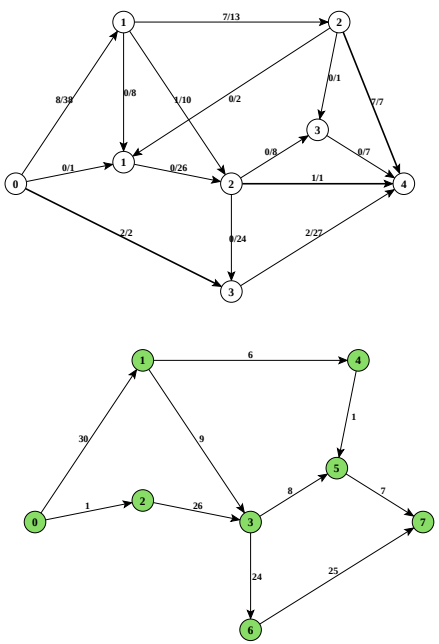


Figure 6.3: A flow and a layered auxiliary network

Layered auxiliary networks are instanciated by the Dinic maximum flow algorithm, and, via inheritance, by the Micali/Vazirani cardinality matching algorithm.

A layered network is a logical view of a flow network, but with a different incidence structure. Nodes and arcs are the same as for the orginal network, and the arc capacities are the residual capacities of the original network. The new incidence structure can be manipulated by two specific operations: Arc insertions which are implemented by the method `InsertProp(TArc)`, and **topological erasure** which is done by the method `TopErasure(TArc)`.

Topological erasure is the arc deletion process, but in a very efficient implementation. If an arc is deleted, some node `v` may become non-reachable from the source node `ss` specified in the constructor. In this case, all arcs with start node `v` are deleted likewise. After the topological erasure of `v`, one has `Blocked(v)==1`.

By that technique, the search procedure `FindPath(t)` which determines an *st*-path with residual capacity is prevented from performing backtracking operations. Note that the information about this path has to be passed from the `layeredAuxNetwork` object to the original network for augmentation. In the Dinic algorithm, both graphs share the predecessor data structure.

During augmentation, `TopErasure(a)` is called for every arc which has no more residual capacity. Finally, the arc insertions and the topological erasure operations are separated by calls to `Phase1()` and `Phase2()` respectively.

The topological erasure process needs $O(m)$ time during the construction of a single blocking flow (called a **phase**), and the time needed for a `FindPath()` operation is proportional to the length of the constructed path.

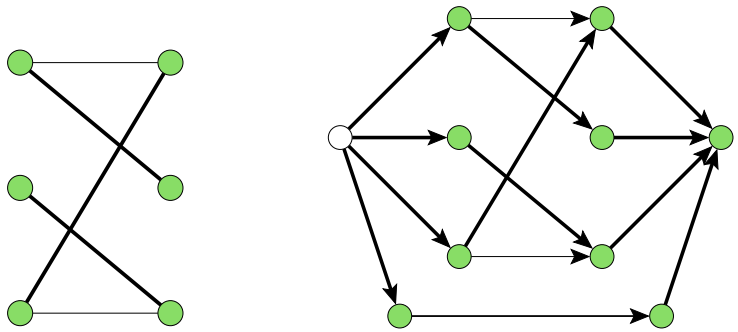


Figure 6.4: A Maximum Assignment with Corresponding Flow

6.3.3 Bipartite Matching Problems as Network Flow Problems

Include file: `bigraphToDigraph.h`
Synopsis:

```
class big2FNW : public virtual abstractDiGraph
{
    big2FNW(abstractBiGraph &,TCap *,TCap * = NULL);
    big2FNW(abstractBiGraph &,TCap);
    big2FNW(abstractBiGraph &);
    TNode       Source();
    TNode       Target();
}
```

This class handles the reduction of bipartite matching problems to network flow problems ans is closely relates to the class of canonical flow networks which were introduced before.

Technically, an artificial source node, an artificial target node, and some arcs are added to the network. The arcs of the original bigraph are directed from one part of the bigraph to the other part. The artificial nodes may be accessed by the respective methods `Source()` and `Target()`.

Any flow on the logical graph object corresponds to a subgraph of the referenced bigraph, and a zero flow corresponds to the empty subgraph. If a perfect matching of the referenced bigraph exists, any maximum flow of the logical object will give such a matching. In that case, a maximum flow of minimum cost corresponds to a minimum cost perfect matching. It may be checked whether a logical flow maps to a perfect matching or not by a call to the method `Perfect()`.

One may pass optional values by the displayed constructor methods: Using the first method, upper and lower degree bounds are defined which appear as capacity bounds of the artificial arcs. Even if lower degree bounds are specified, the `big2fnw` object is always in canonical form.

The second constructor method is used to solve a k-factor problem. If no parameters (up to the bigraph) are specified, the node demand labels encapsulated in the bigraph come into play.

Example:

```
G1 = new bigraph("sample.gob");
G2 = new big2FNW(G1,1);
G2 -> MaxFlow(G2->Source(),G2->Target());
if (G2->Perfect())
{
```

```
    F1 = new export("sample.rst");
    G1 -> WriteSubgraph(F1);
    delete F1;
};
delete G2;
delete G1;
```

effectively determines a mximum 1-matching of the graph object `G1`.

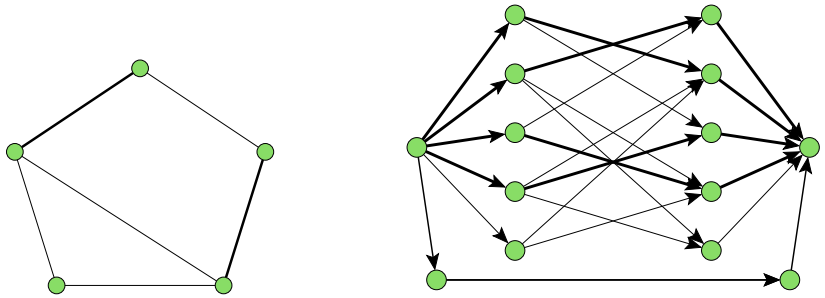


Figure 6.5: A Maximum Matching with Corresponding Balanced Flow

6.3.4 General Matching Problems as Balanced Flow Problems

Include file: `graphToBalanced.h`
Synopsis:

```
class gra2bal : public virtual abstractBalancedFNW
{
    gra2bal(abstractGraph &,TCap *,TCap * = NULL);
    gra2bal(abstractGraph &,TCap);
    gra2bal(abstractGraph &);
    TNode       Source();
    TNode       Target();

    void        InitFlow();
    void        Update();
}
```

The idea of this problem transformation is to split the nodes and arcs into symmetrical pairs and reduce to a balanced network flow problem. Similar to the bipartite situation, artificial nodes and arcs are added. The resulting flow network is bipartite, and the image of an original node consists of an outer and an inner node.

The constructors and some other methods are defined in analogy to the bipartite situation. If lower degree bounds are specified, the elimination of the lower capacity bounds is done immediately. (If we would apply the class `fnw2fnw`, the complementarity relationship would be lost!)

Balanced network flow methods (not ordinary network flow methods!) manipulate the subgraph encoded into the referenced object. However, a `gra2bal` object may maintain a flow which is non-symmetric, and independent from the referenced object. One can generate and access this flow simply by calling any network flow method or by explicit call to the method `InitFlow()`. By that, the flow is initialized to the symmetric logical flow, but can be treated as an ordinary flow afterwards.

Every call to a balanced network flow method requires the symmetric flow. If necessary, `Update()` is called which symmetrizes the flow again. The subgraph of the referenced object is updated, and the physical flow is disallocated. Note that `Update()` is called by the `gra2bal` destructor also.

Hence, there are two kinds of flow associated with a `gra2bal` object, exactly one of these flows is present at each point of lifetime, and the object is generated and destructed with a balanced flow.

Example:

```
G1 = new graph("sample.gob");
G2 = new gra2bal(G1,1);
G2 -> MaxFlow(G2->Source(),G2->Target());
G2 -> CancelEven();
if (G2->CancelOdd()>1) G2->MaxBalFlow(G2->Source());
delete G2;
F1 = new export("sample.rst");
G1 -> WriteSubgraph(F1);
delete F1;
delete G1;
```

determines a maximum 1-matching of the graph in `"sample.gob"` as follows: First an ordinary maximum flow of the object `G2` is computed (starting with a call to `InitFlow`). The call of `CancelEven()` implies a call to `Update()`. All subsequent operations immediately manipulate the subgraph of the object `G1`.

6.3.5 Layered Shrinking Networks

Include file: `shrinkingNetwork.h`
Synopsis:

```
class layeredShrNetwork : public layeredAuxNetwork
{
    layeredShrNetwork(abstractBalancedFNW &,TNode,
                      staticQueue<TNode,TFloat> **,
                      staticQueue<TArc,TFloat> **,
                      staticQueue<TArc,TFloat> **);

    TNode           StartNode(TArc);

    TNode           DDFS(TArc);
    void            ShrinkBlossom(TNode,TArc,TFloat);

    TFloat          FindPath(TNode);
    void            Expand(TNode,TNode);
    void            CoExpand(TNode,TNode);
    void            Traverse(TNode,TNode,TNode,
                             TArc,TArc *,TArc *);

    void            Augment(TArc);
}
```

This class makes the topological erasure technique of layered auxiliary networks available to matching and balanced network flow problems. But this class has a lot of additional data structure and functionality.

If one looks at the constructor interface only, it is obvious that there are a lot of dependencies between the `layeredShrNetwork` objects and the algorithm which constructs the object. We do not go into the details, but need to describe the functionality and the running times of some of the methods.

Roughly speaking, a **double depth first search** `DDFS(a)` determines the blossom which occurs if the arc `a` is added to the layered auxiliary network, and returns the base `b` of this blossom. Then the blossom can either be shrunk by a call `ShrinkBlossom(b,a,..)`, or a minimum length augmenting path is found which can be extracted by a call of `FindPath(s^1)`.

Note that `FindPath()` recursively calls `Expand()`, `CoExpand()` and `Traverse()` which are only needed at this point. The method `Augment()` actually does the augmentation and triggers off the necessary topological erasure operations.

All these operations must be separated from the `InsertProp()` operations by using the methods `Phase2()` and a `Phase1()`. The complexity of these new operations can be bounded for a whole phase, and is $O(m)$ for the `DDFS()` operations. The time needed for `FindPath()` operations is proportional to the length of the constructed paths again.

6.3.6 Surface Graphs

Include file: `surfaceGraph.h`
Synopsis:

```
class surfaceGraph : public abstractBalancedFNW
{
    surfaceGraph(abstractBalancedFNW &);

    TFloat      ModLength(TArc);
    TFloat      RModLength(TArc);
    void        ShiftPotential(TNode,TFloat);
    void        ShiftModLength(TArc,TFloat);
    bool        Compatible();
    void        CheckDual();

    TArc        FindSupport(TNode,TArc,
                            dynamicStack<TNode,TFloat> &);
    void        Traverse(TArc*,TArc,TArc);
    void        Expand(TArc*,TArc,TArc);
    void        ExpandAndAugment(TArc,TArc);

    TFloat      ComputeEpsilon(TFloat*);
    void        PrimalDual0(TNode);
    void        Explore(TFloat*,goblinQueue<TArc,TFloat> &,
                        THandle,TNode);
    TFloat      ComputeEpsilon1(TFloat*);
    void        PrimalDual1(TNode);
}
```

Surface graphs are data structures which are needed by all weighted matching algorithms. A surface graph object keeps a shrinking family of a given balanced flow network and forms a new graph in which some of the original nodes are identified, and some arcs are redirected.

While shrinking families will be discussed later in Section 8.2, we need to describe the other components of the primal-dual algorithm here:

Modified length labels are the reduced costs known from linear programming and are available by the method `ModLength()`. They may be present by an own data structure, or must be computed recursively by `RModLength()` which evaluates the node potentials and the shrinking family data structure. This recursive computation is needed when working with large scale geometrical matching problems and is enabled by the context flag `methModLength`.

If disabled, mismatchs between physical and computed modified lengths can be detected by a call of `CheckDual()`. This is done automatically before the primal-dual methods halt, but only if the context flag `methFailSave` is set. In that case, `Compatible()` is called likewise to check for reduced costs optimality.

Note that a single `RModLength()` call takes $O(n)$ operations, and that exhaustive computation may increase the running time of the whole algorithm by a factor of n. Hence some care is recommended when setting `methModLength` and `methFailSave`. The complexity statements which follow are true only if both variables are zero.

The method `FindSupport` determines the nodes of a blossom, and prepares the data structures which are necessary to reconstruct an augmenting path traversing this blossom. The latter task is managed by the methods `Traverse`, `Expand` and `ExpandAndAugment` which take $O(n \log n)$ time per each augmentation. The `FindSupport` operations take $O(n)$ time per **phase**, that is the period between two augmentations of the PD algorithm.

GOBLIN includes three implementations of the PD algorithm which can be selected by the context flag `methPrimalDual`. The options `methPrimalDual==0` and `methPrimalDual==1` depend on `ComputeEpsilon()`, whereas the option `methPrimalDual==2` depends on `ComputeEpsilon1()`. Both methods determine the amount of a **dual update**, that is an update on the node potentials. The first procedure searches all arcs and takes $O(m)$ time, whereas the second procedure searches only one arc for each node and hence takes $O(n)$ time.

In the current state of development, `methPrimalDual=0` causes the use of `PrimalDual0`, whereas `methPrimalDual=1` and `methPrimalDual=2` cause the use of `PrimalDual1`. Both methods use a dual update technique which takes $O(m)$ time so that the overall complexity is $O(nm)$ per phase, independent of which implementation in used. It is planned, however, to improve `PrimalDual1` to $O(n^2)$ time.

6.3.7 Suboptimal Balanced Flows

Include file: `balancedToalanced.h`

The class `bal2bal` is the symmetrical counterpart of the class `FNW2FNW`, and hence manages:

(1) the reduction of the feasible balanced circulation problem to the maximum balanced *st*-flow problem,

(2) the reduction of the min-cost balanced circulation problem to the min-cost balanced *st*-flow problem.

The main application is the reduction of the **odd cycle canceling problem** for balanced network flows to a balanced *st*-flow problem. This problem occurs if an integral circulation is symmetrized so that some flow values become non-integral.

These reductions eventually extend the Anstee maximum balanced flow algorithm to the general setting of balanced flow networks, and allow a strongly poynomial implementation of the primal-dual algorithm respectively.

6.3.8 Making Logical Objects Persistent

Logical objects turn into persistent objects by writing them to file and loading them again. By running optimization methods on the persistent object, one can avoid the time consuming dereferencing steps to the original data object.

However, the capability of back transformation of computational results to the original problem instance is lost. If necessary, the results can be written to a file and reimported into the logical object.

Example:

```
G1 = new diGraph("sample.gob");
G2 = new FNW2FNW(G1);
F1 = new export("sample.tmp");
G2 -> Write(F1);
delete F1;

G3 = new diGraph("sample.tmp");
G3 -> MaxFlow(G2->Source(),G2->Target());

F1 = new export("sample.tmp");
G3 -> WriteFlow(F1);
delete F1;
delete G3;

F2 = new import("sample.tmp");
G2 -> ReadFlow(F2);
delete F2;

if (G2->Perfect())
{
    F1 = new export("sample.rst");
    G1 -> WriteFlow(F1);
    delete F1;
}

delete G2;
delete G1;
```

It has turned out that file export is rather expensive, and should be used

by extremely search intensive problem solvers only. With some additional efforts for mapping the potential solutions, copy constructors as presented in Section 6.4.1 are highly preferable.

6.4 Derived Persistent Objects

There are some situations where the implementation of a logical class without an own incidence structure is inappropriate for the problem transformation:

- If the transformation mechanisms would be very expensive,
- If the problem to solve is very complicated so that the instances are rather small,
- If the transformation is of academic interest rather than practical need.

Typically, the class definition only consists of a constructor method.

6.4.1 Copy Constructors

Each of the persistent base classes provides a copy constructor which supports the following general purpose options:

- `OPT_CLONE`: Generate a one-to-one copy of the graph. That is, map every node and every arc of the original graph. If this option is absent, arcs with zero capacity are not mapped.
- `OPT_PARALLELS`: Allow parallel edges. If this option is absent, an arbitrary arc of every parallel class is mapped. The option is immaterial if mapping to dense implementations.
- `OPT_SUB`: Export the subgraph labels into a separate object. That is, the capacity of a mapped arc is the subgraph label of the original arc.

The `graph` and the `denseGraph` copy constructors accept arbitrary mixed graphs and just forget about the orientations. The directed classed also accept mixed graphs and, in principle, generate antiparallel arc pairs for undirected arcs in the original graph. But be careful with the constructor method `digraph(G,opt)`: If the input graph is bipartite and `OPT_CLONE` is absent, the arcs are just oriented from one partition (the end node with smaller index) to the other.

6.4.2 Mapping Back Derived Graph Objects

Synopsis:

```
class abstractMixedGraph
{
    TNode   OriginalNode(TNode);
    TArc    OriginalArc(TArc);
    void    ReleaseNodeMapping();
    void    ReleaseArcMapping();
}
```

When generating graph objects from others, it is sometimes useful to maintain the mappings of nodes and arcs from the derived to the original graph. In principle, this information can be accessed by the methods `OriginalNode()` and `OriginalArc()`.

However, these mappings must be generated explicitly by the constructor option `OPT_MAPPINGS` and only few classes implement this option yet now. Even more, the mappings are invalidated by every node or arc deletion operation. If there is no predecessor in the original graph, or if no mappings are available, `NoNode` (`NoArc`) is returned.

6.4.3 Line Graphs and Truncation of the vertices

Include file: `sparseGraph.h`
Synopsis:

```
class lineGraph : public graph
{
    lineGraph(abstractMixedGraph &,TOption option = 0);
}

class planarLineGraph : public graph
{
    planarLineGraph(abstractMixedGraph &,
                    TOption option = 0);
}

class vertexTruncation : public graph
{
    vertexTruncation(abstractMixedGraph&,TOption = 0);

};
```

In a **line graph** the nodes are the arcs of the original graph G, and nodes are adjacent if and only if the arcs share an end node in the original graph. The constructor method `lineGraph(G)` matches this graph-theoretical definition of line graphs.

By the constructor `planarLineGraph(G)`, edges are generated only for pairs of edges which are neighbors in the incidence lists of G. We refer to this as **planar line graphs** since planar input graphs are mapped to planar graphs by this procedure. More explicitly, the faces are mapped to face of the same length, and the boundary cycle is directed counter clockwise. The nodes of the original graph are also mapped to faces where the boundary cycle is directed clockwise and its length is the degree of the original node.

If G is the surface graph of some non-degenerate polyhedron (all vertices have degree 3), both definitions coincide. If G is the surface graph of some regular polyhedron (all faces are equilateral), the planar surface graph has the same geometric interpretation.

By the constructor `vertexTruncation(G)`, the vertices of the original graph are also replaced by cycles of the adjacent edges, and these cycles form faces of the newly generated graph. Other than for the planar line graph, the original arcs are maintained, and all vertices have degree 3.

Loosely speaking, both planar transformations rasp off the vertices of the original polyhedron, and the planar line graph is the extremal case where the original edges collapse the vertices.

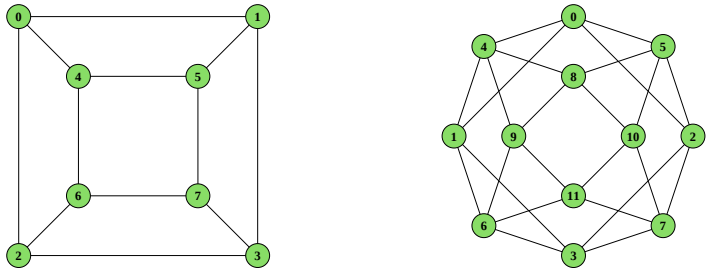


Figure 6.6: A Graph and its Line Graph

6.4.4 Tearing Apart the Regions of a Planar Graph

Include file: `sparseGraph.h`
Synopsis:

```
class facetSeparation : public graph
{
    enum TOptRotation {
        ROT_NONE = 0,
        ROT_LEFT = 1,
        ROT_RIGHT = 2
    };

    facetSeparation(abstractMixedGraph&,
```

```
                    TOptRotation = ROT_NONE);
}
```

This constructor method is another technique to generate regular graphs.

- `ROT_NONE`: Grow the original nodes to faces of the same degree, and the original edges to 4-sided faces. The resulting graph is 4-regular.
- `ROT_LEFT`: As before, but triangulate the faces representing the original edges such that every node is incident with exactly one triangulation arc and the resulting graph is 5-regular. Two different triangulations are possible. Choose the one which can be interpreted as rotating the original faces counterclockwise.
- `ROT_RIGHT`: Analogous to the `ROT_LEFT` option but rotate clockwise.

This construction is well-defined for every planar graph. A topologic embedding is provided for the resulting graph.

As an example, if `G` is a tetrahedron, `facetSeparation(G,facetSeparation::ROT_LEFT)` will produce an icosahedron.

6.4.5 Complementary Graph

Include file: `sparseGraph.h`
Synopsis:

```
class complementaryGraph : public graph
{
    complementaryGraph(abstractMixedGraph &,TOption);
}
```

The **complementary graph** is defined on the same node set as the original graph, and two nodes are adjacent if and only if they are non-adjacent in the original graph. Complementary graphs are used to switch between stable set and clique problems.

Note that the complement of a graph with many nodes but few edges requires a lot of computer storage.

6.4.6 Dual Graphs

Include file: `sparseGraph.h`
Synopsis:

```
class dualGraph : public graph
{
    dualGraph(abstractMixedGraph&,TOption = 0);
}

class directedDual : public diGraph
{
    directedDual(abstractMixedGraph&,TOption = 0);
}
```

To generate a **dual graph**, the input graph must be planar and already provide a combinatorial embedding. A geometric embedding is not required. The nodes of the new graph are the regions of the original graph and the arcs map one-to-one. Nodes are adjacent if and only if the regions share an arc in the original graph.

Dualization preserves the combinatorial embedding. In particular, the dual of a dual graph can be computed instantly and the original graph and combinatorial embedding will result. An existing drawing of the original graph is translated to the dual graph in a very simple way. This drawing does not map back to the original drawing and produces edge crossings at least for the unbounded region.

It is also possible to generate **directed dual graphs** where the arcs are oriented as follows:

- Exterior edges of the primal graph are pointing towards the exterior region.

- If an interior edge is directed in the primal graph, the dual arc will cross from the left-hand face to the right-hand face (provided that the edges are ordered clockwise in the primal graph)
- If an interior edge does not have an explicit orientation, the colours of its end nodes are compared, and the edge is directed from the smaller colour index to the higher one.

By this procedure, **bipolar digraphs** (acyclic digraphs with a unique source and a unique sink node) are mapped to bipolar dual digraphs. The dual source and target nodes are available by `Source()` and `Target()` for further processing. These nodes are adjacent by the `ExteriorArc()`.

6.4.7 Spread Out Planar Graphs

Include file: `sparseGraph.h`
Synopsis:

```
class spreadOutRegular : public graph
{
    spreadOutRegular(abstractMixedGraph&,TOption = 0);
}
```

This class is intended for displaying regular polyhedra in the plane. The input graph must be planar and already provide a combinatorial embedding. Furthermore, a spanning tree must be available by the predecessor labels.

The tree edges are replaced by an Hamiltonian cycle in which every of the former edges occurs twice. The resulting graph is outerplanar with the new cycle forming the exterior face. The graph is drawn with the specialized method described in Section 12.7,

Formally, `spreadOutRegular` objects can be obtained from any planar graph. But the final drawing step produces readable output only in the situation of regular polyhedra.

6.4.8 Metric Closure

Include file: `denseGraph.h`
Synopsis:

```
class metricGraph : public denseGraph
{
    metricGraph(abstractGraph &);
}
```

This class defines the **metric closure** of undirected graphs, in which the length of an arc corresponds to the minimum length of a path in the original graph. The metric closure is used to generate heuristic hamiltonian cycles for sparse graphs.

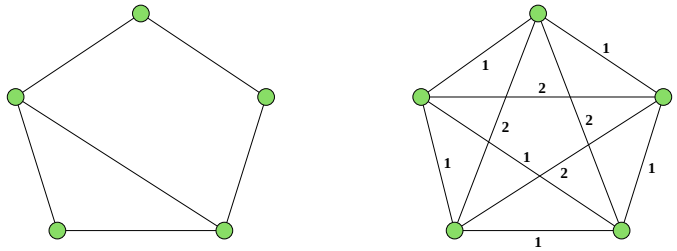


Figure 6.7: A Graph and its Metric Closure

6.4.9 Distance Graphs

Include file: `denseDigraph.h`
Synopsis:

```
class distanceGraph : public denseDiGraph
{
    distanceGraph(abstractMixedGraph &);
}
```

This is the asymetric counterpart to the metric closure, in which the length of an arc corresponds to the minimum length of a directed path in the original (possibly mixed) graph object.

6.4.10 Complete Orientation

Include file: `sparseDigraph.h`
Synopsis:

```
class completeOrientation : public diGraph
{
    completeOrientation(abstractMixedGraph &G,
                            TOption options = 0);

    TArc    OriginalArc(TArc);
}
```

The **complete orientation** of a mixed graph is the digraph in which every undirected edge of the original object is replaced by a pair of antiparallel arcs. If the optional parameter is `OPT_REVERSE`, directed arcs are mapped to a pair of arcs likewise. For every arc `a` in the orientation, the origin can be obtained by the call `OriginalArc(a)`.

6.4.11 Induced Orientation

Include file: `sparseDigraph.h`
Synopsis:

```
class inducedOrientation : public diGraph
{
    inducedOrientation(abstractMixedGraph &G,
                            TOption options = 0);
}
```

The **complete orientation** of a mixed graph is the digraph in which every undirected edge of the original graph object is oriented from the smaller node colour index to the higher index. Since edges with equal colour indices are omitted, this construction can be used to achieve oriented bigraphs. Another application is the generation of st-**orientations** from st-numberings.

6.4.12 Node Splitting

Include file: `sparseDigraph.h`
Synopsis:

```
class nodeSplitting : public diGraph
{
    nodeSplitting(abstractMixedGraph &,TOption = 0);
}
```

The **node splitting** of a mixed graph is similar to its complete orientation. In addition, every node v of the original graph is replaced by a pair v_1, v_2 and an arc v_1v_2 whose capacity bound is the demand of the original node. Every eligible arc uv in the original graph are represented by the arc u_2v_1 in the node splitting. Note that the origins of the arcs in a node splitting cannot be dereferenced.

6.4.13 Tilings

Include file: `sparseGraph.h`
Synopsis:

```
class tiling : public graph
{
    tiling(abstractMixedGraph &,TOption,TNode,TNode);
}
```

A tiling of a given graph consists of several copies of the original. The original graph should provide a plane embedding, and the nodes 0,1,2,3 should form a rectangle with the remaining nodes in the interior. The corner nodes are identified in the tiling.

By this construction principle, one obtains a series of planar triangulated graphs each of which has an exponential number of 1-factors and 2-factors, and also an exponential number of odd cycles.

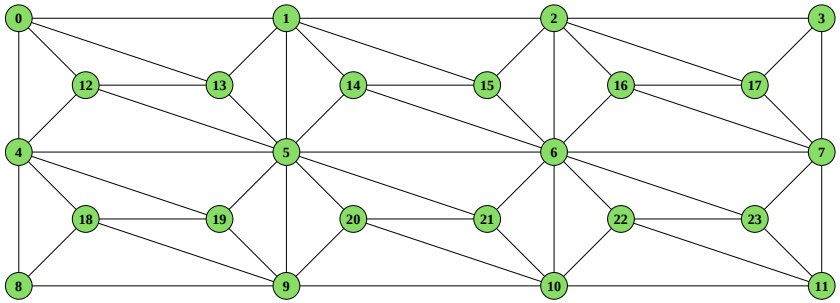


Figure 6.8: A Tiling

6.4.14 Split Graphs

Include file: `balancedDigraph.h`
Synopsis:

```
class splitGraph : public balancedFNW
{
    splitGraph(abstractDiGraph &G,TNode s,TNode t);

    TNode Source() {return n-1;};
    TNode Target() {return n-2;};
}
```

Split graphs establish balanced network flow (matching) formulations of ordinary *st*-flow problems with integral capacities. Since matching algorithms are technically much more complicated than network flow methods, split graphs are not useful for practical computations but rather for the debugging of matching algorithms.

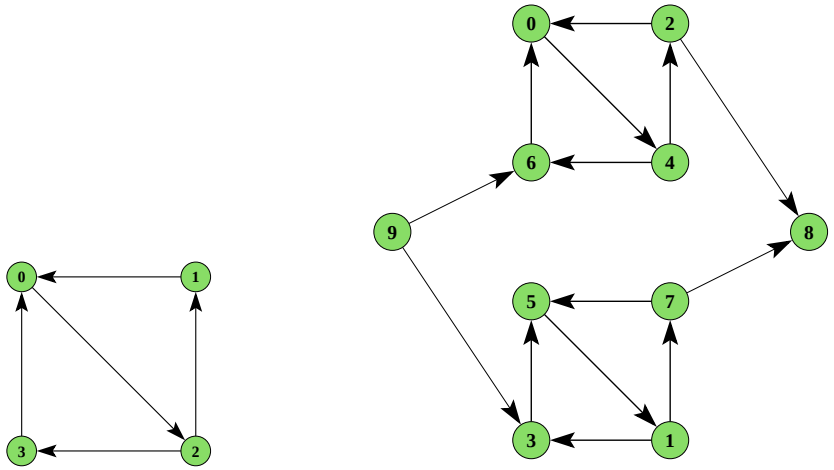


Figure 6.9: A Digraph and its Split Graph

6.4.15 Subgraph induced by a Node or Arc Set

Include file: `mixedGraph.h`
Synopsis:

```
class inducedSubgraph : public mixedGraph
{
    inducedSubgraph(abstractMixedGraph&,indexSet<TNode>&,
```

```
        TOption = OPT_PARALLELS);
    inducedSubgraph(abstractMixedGraph&,indexSet<TNode>&,
        indexSet<TArc>&,TOption = OPT_PARALLELS);
}
```

Other than the subgraphs which can be generated by a `mixedGraph` constructor and which map every node of the original graph, this class handles subgraphs which are induced by a given node set. This node set is passed as an index set (see Chapter 9 for a description) and may be further restricted by an arc index set.

More explicitly, the graph `inducedSubgraph(G,V,A,opt)` consists of all nodes in the index set V. Only the original arcs in A are mapped, namely iff both end nodes are in V. If no arc set A is specified, the resulting **induced subgraph** is as in the literature.

If the option `OPT_PARALLELS` is specified or if the parameter *opt* is omitted, parallel arcs are allowed. Otherwise, some minimum length edge is mapped. The other supported options are `OPT_NO_LOOPS`, `OPT_SUB` and `OPT_MAPPINGS` with the already described semantics.

6.4.16 Bigraph induced by two Node Colours

Include file: `sparseBigraph.h`
Synopsis:

```
class inducedBigraph : public biGraph
{
    inducedBigraph(abstractMixedGraph&,indexSet<TNode>&,
        indexSet<TNode>&,TOption = OPT_PARALLELS);
}
```

This constructor `inducedBigraph(G,U,V,opt)` works much like for the previously described class `inducedSubgraph`. Two specified node sets U and V are mapped. Edges are mapped only if one end node is in U and the other end node is in V and implicitly oriented from U to V then. Both sets must be disjoint; otherwise an exception `ERRejected()` is raised. The options are handled as before.

6.4.17 Colour Contraction

Include file: `mixedGraph.h`
Synopsis:

```
class colourContraction : public mixedGraph
{
    colourContraction(abstractMixedGraph&,TOption = 0);
}
```

The nodes of `colourContraction(G,opt)` are the colour classes of G. That is, all nodes are mapped and equally coloured nodes of G are mapped to the same node. Edges are mapped only if the end nodes belong to different colour classes.

Two options are supported: If the option `OPT_PARALLELS` is specified, parallel arcs are allowed. Otherwise, some minimum length edge is mapped. If the option `OPT_SUB` is specified, the subgraph data structure is exported to an own graph object.

6.4.18 Transitive Closure

Include file: `sparseDigraph.h`
Synopsis:

```
class transitiveClosure : public diGraph
{
    transitiveClosure(abstractDiGraph&G,TOption = 0);
}
```

This constructor `transitiveClosure(G,opt)` copies the input digraph and augments it by all **transitive arcs** (whose end nodes are connected by a directed path with at least two arcs length). The running time is $O(nm)$. If the option `OPT_SUB` is specified, the input graph is encoded by the edge colours.

6.4.19 Intransitive Reduction

Include file: `sparseDigraph.h`
Synopsis:

```
class intransitiveReduction : public diGraph
{
    intransitiveReduction(abstractDiGraph&,TOption = 0);
}
```

For a given acyclic digraph `G`, the constructor `intransitiveReduction(G,opt)` determines a maximal subgraph without any transitive and parallel arcs. The running time is $O(nm)$. If the option `OPT_SUB` is specified, the input DAG is copied, and the intransitive subgraph is encoded by the edge colours.

6.4.20 Explicit Surface Graphs

Include file: `mixedGraph.h`
Synopsis:

```
class explicitSurfaceGraph : public mixedGraph
{
    explicitSurfaceGraph(abstractMixedGraph&,
            shrinkingFamily<TNode>&,TFloat*,TArc*);
}
```

This class has been added for the graphical tracing of the Edmonds' spanning arborescence method only. In a future release, it may also be used to trace matching algorithms.

The constructor parameters are the digraph for which the arborescence is computed, the current shrinking family, the modified length labels and the predecessors of the original digraph.

6.4.21 Voronoi Diagram

Include file: `sparseGraph.h`
Synopsis:

```
class voronoiDiagram : public graph
{
    voronoiDiagram(abstractMixedGraph&);
    ~voronoiDiagram();

    TFloat UpdateSubgraph();
}
```

This class has been introduced for the Mehlhorn Steiner tree Heuristic. Other applications seem obvious, especially to the T-join solver. The name indicates a relationship to the well-known geometric structure, but do not confuse both notions.

The constructor method generates a copy of the given graph in which the node sets of the partition data structure are contracted. The mapping of the nodes and edges is preserved transparently.

The procedure assumes that in the original graph the predecessor labels form partial trees which span the node partition sets and which are rooted at some terminal node (see Section 13.19. The needed data structures for the original graph are implicitly set up by calling the method `VoronoiRegions()`. By that, the partial trees consist of shortest paths, corresponding distance labels are given, and hence the transformed graph edges are shortest paths between different terminal node.

The method `UpdateSubgraph()` considers the predecessor arcs of the transformed graph and maps them back to paths in the original problem instance. The result is a subgraph, not a set of modified predecessor labels!

6.4.22 Triangular Graphs

Include file: `sparseGraph.h`
Synopsis:

```
class triangularGraph : public graph
{
    triangularGraph(TNode,TOption,
            goblinController & = goblinDefaultContext);
}
```

The nodes of a **triangular graph** are the 2-element subsets of a finite ground set. Two nodes are adjacent if they have an element in common. Triangular graphs are interesting for their regularity.

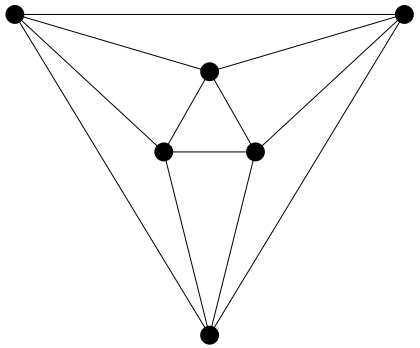


Figure 6.10: A Triangular Graph

Chapter 7

Iterators

An iterator is an object which allows to access listed information which is encapsulated into another data object. In the context of graphs, an iterator supplies with a stream of incident arcs for each of the nodes.

7.1 Incidence Lists

Include file: `abstractMixedGraph.h`
Synopsis:

```
class abstractMixedGraph
{
    TArc                First(TNode);
    TArc                Right(TNode,TArc);
};
```

Node incidence lists are implicitly defined by the methods `First` and `Right` which are available in arbitrary graphs, but implemented differently. In any implementation, the method call `First(v)` should return an arbitrary arc with start node `v`, and the call `Right(u,a)` should return an arc which has the same start node as `a`, namely the node `u`. The repetition of the start node is needed to improve the efficiency of the iterator oprations.

Moreover, node incidence lists must be circular, and contain all arcs which have the same start node. That is, for every pair `a1`, `a2` of arcs with common start node `a1` must be derefencable from `a2` by the method `Right`.

The method name `Right` may suggest that one traverses the node incidences clockwise. In fact, this makes sense for sparse graphs embedded in plane. The most algorithms which run on planar graphs require that `Right` provides a combinatorial embedding of the graph.

Accordingly, the method `Left` which is available for sparse graph objects, supplies with reverse incidence lists. In the mentioned cases, `Left` defines a combinatorial embedding and traverses the node incidences anticlockwise.

7.2 Iterator Objects

Include file: `goblinIterator.h`
Synopsis:

```
class goblinIterator : public virtual goblinDataObject
{
    goblinIterator(abstractMixedGraph &)

    void        Reset() = 0;
    void        Reset(TNode) = 0;
    TArc        Read(TNode) = 0;
    TArc        Peek(TNode) = 0;
    void        Skip(TNode) = 0;
    bool        Active(TNode) = 0;
}
```

Node incidences may either be accessed directly using the methods `First` and `Right`, or by an iterator object which has some advantages:

- The code looks more tidy, more like a high-level description.
- There is a mechanism for caching iterator objects. By that, the frequency of memory allocation and deallocation operations is reduced.
- Development is speeded up since memory faults can be avoided.

The possible iterator methods can be described within a few words: The method `Reset` is used to reinitialize incidence streams, either of a single node or the whole node set.

The method `Active` checks if there are unread incidences of a given node. In that case, the methods `Read` and `Peek` return such an unread incidence. Otherwise, `Read` and `Peek` throw an exception `ERRejected`.

The difference between `Read` and `Peek` is that the latter method does not mark any incidences unread. To do this explicitly, that is, to proceed in the incidence list, one calls `Skip`. A statement `a = I.Read(v)` does the same as the sequence `a = I.Peek(v); I.Skip(v)`.

Example:

```
...
goblinIterator *I = new goblinIterator(G);
TFloat L = -InfFloat;
for (v=0;v<G.N();v++)
   while (G.Dist(v)<InfFloat && I->Active(v))
   {
      a = I->Read(v);
      TNode w = G.EndNode(a);
      if (G.Dist(w)<InfFloat && G.Length(a)>L)
         L=G.Length(a);
   };
delete I;
...
```

determines the maximum length of an arc spanned by the nodes with finite distance labels in the graph `G`. Note that this code is optimal only if the node set is rather small.

7.3 Implicit Access

Include file: `abstractMixedGraph.h`
Synopsis:

```
class abstractMixedGraph
{
    THandle             Investigate();
    goblinIterator &    Iterator(THandle);
    void                Reset(THandle,TNode=NoNode);
    TArc                Read(THandle,TNode);
    bool                Active(THandle,TNode);
    void                Close(THandle);
    void                ReleaseIterators();
};
```

Node incidences are accessed by iterators. This may be done explicitly as described in the previous section. There is, however, an equivalent formulation where all iterator functionality is encapsulated into the referenced graph object:

Example:

```
...
TFloat L = -InfFloat;
THandle H = G.Investigate();
for (v=0;v<N();v++)
   while (G.Dist(v)<InfFloat && G.Active(H,v))
   {
      a = G.Read(H,v);
      TNode w = G.EndNode(a);
      if (G.Dist(w)<InfFloat && G.Length(a)>L)
         L=G.Length(a);
   };
```

```
Close(H);
...
```

The latter approach requires additional effort for dereferencing the iterator. The benefit is caching of the 'used' iterator which effectively decreases the effort of memory allocation and defragmentation.

The method `Investigate()` returns a handle to an iterator object. If there is a cached iterator, the cached object is initialized, and the handle is returned. Otherwise a new iterator is allocated and assigned to a handle.

The method `Close(THandle)` finishes a graph search. If the cache space is exhausted, the iterator is disallocated. Otherwise the iterator object is cached, and can be reused later. If the `Close` statement is omitted, GOBLIN will return an error when the referneced graph object is deleted.

The method `RelaseIterators()` deletes all cached iterator objects. This method is called by destructor methods automatically. The `Reset`, `Read` and `Active` operations work just as if the iterator would be accessed directly.

The most efficient way to work with iterators is to combine the caching functionality with explicit access as described in the previous section. This is accomplished by the method `Iterator(THandle)` which returns the address of the iterator object associated with some handle.

7.4 Implementations

Include files: `abstractMixedGraph.h, auxiliaryNetwork.h, surfaceGraph.h`
Synopsis:

```
class abstractMixedGraph
{
    virtual goblinIterator * NewIterator();
};
```

Just as graph objects, GOBLIN iterators are polymorphic. There is, however, the class `iGraph` which supplies with iterators for most graph objects. Such an iterator is returned by the method `abstractMixedGraph::NewIterator()` and utilizes the methods `First` and `Right` which have been discussed before.

Under some circumstances, the methods `First` and `Right` do not provide an efficient implementation. For this reason, surface graphs and layered auxiliary networks implement own iterators which keep some temporary information.

Accordingly, the method `NewIterator` is overloaded in order to supply `Investigate` with a proprietary iterator object.

Chapter 8

Explicit Data Structures

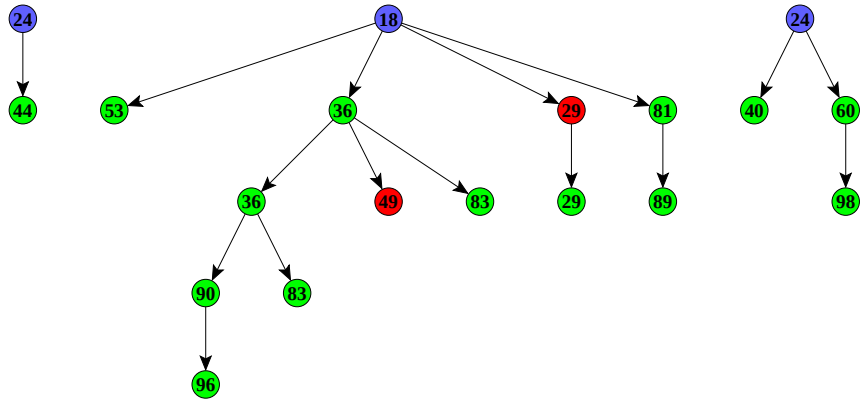


Figure 8.1: Fibonacci Heaps

This chapter describes the GOBLIN classes which are **data structures** in the usual sense and for some of which equivalent data structures can be found in the C++ standard template library (STL). The template data structures which are discussed here support the GOBLIN memory management and tracing functionality.

In general, the template parameter `<TItem>` has to be integral and the GOBLIN library file contains template instances for the types `TNode` and `TArc`. To generate additional template instances, one may include the corresponding `.cpp` file directly.

8.1 Container Objects

Include file: `goblinQueue.h`
Synopsis:

```
template <class TItem,class TPriority>
class goblinContainer : public virtual goblinDataObject
{
    virtual void    Init() = 0;

    virtual void    Insert(TItem,TKey) = 0;
    virtual void    ChangeKey(TItem,TKey) = 0;
    virtual TItem   Delete() = 0;
    virtual TItem   Peek() = 0;
    virtual bool    Empty();
    virtual TItem   Cardinality() = 0;
```

```
}
```

Container objects are either set or multiset objects. One can also classify containers into queues, stacks and priority queues by the order in which elements can be deleted.

The members of a container are `TItem` objects which are inserted and deleted by the listed prototype methods. The second template parameter `TKey` is the optional priority of the members of a priority queue. It is declared in a more general context to preserve compatibility among the various container classes. The same holds for the operation `ChangeKey`.

Example:

```
...
binaryHeap<TArc,TFloat> Q(M());
for (a=0;a<M();a++) Q.Insert(a,Length(2*a));
while (!Q.Empty())
{
    a = Q.Delete();
    ...
}
```

effectively sorts the arcs of a graph object by their length labels. This is simply done by putting the arcs on a priority queue from where they are taken for further processing.

All GOBLIN container classes are defined by **templates**. That is, the member type `TItem` is abstract. This type is not resolved at run time, but by the C++ compiler.

In our example, the template instance `binaryHeap<TArc,TFloat>` can be already found by the linker in the library `goblin.a`. If no pre-compiled code would be available, one would include `binheap.cpp` rather than `binheap.h` to force the compiler to generate such code.

A container may be **dynamic** where every member is represented by an individual struct object. Otherwise the container is **static**, and all members are represented by arrays which are maintained during the entire lifetime of the container object. The latter concept has some serious drawbacks:

- The member type `TItem` must be integral. That is, the members are indices rather than objects.
- No repetitions are allowed. That is, static containers are set objects rather than multisets.
- A maximum index must be passed to the constructor which determines the size of all arrays encapsulated into the set object.

Note that a static data structure is adequate in the example from above. Even if inefficient, static sets may be useful during the testing phase of an algorithm to detect unwanted repetitions, and can be replaced by a dynamic structure in the final version.

For static containers, one can check efficently if an item is missing (`IsMember()`). Furthermore, static containers may share memory with other containers. However, it must be clear that all these sets are disjoint:

Example:

```
...
staticQueue<TNode,TFloat> **Q
    = new staticQueue<TNode,TFloat>*[n];
Q[0] = new staticQueue<TNode,TFloat>(n);
for (v=1;v<n;v++)
   Q[v] = new staticQueue<TNode,TFloat>(Q[0]);
for (v=1;v<n;v++)
   Q[d[v]] -> Insert(v);
...
```

These lines of code form part of the Micali/Vazirani algorithm which distributes the node set of a balanced flow network over all queues `Q[0],Q[1],...,Q[n-1]` where `Q[i]` consists of the node with distance label `i`. In this special situation, the static implementation is indeed the most efficient data structure.

Every container object can be 'emptied' by the method `Init()`. This is particularly useful for static implementations. One could also think of some caching mechanism of dynamic queue member objects, but this is not implemented yet.

8.1.1 Queues

Include files: `staticQueue.h, staticQueue.cpp, dynamicQueue.h, dynamicQueue.cpp`

Queues are container objects which follow the **first-in first-out principle** principle. There are two implementations: The class `staticQueue` which models sets, and the class `dynamicQueue` which models multisets.

For both classes, the GOBLIN library contains precompiled code for the template instances `<TNode,TFloat>` and `<TArc,TFloat>`. Note that the choice of `TKey` is immaterial to some extent. Except for the construction of a static queue and the destruction of a dynamic queue, all operations are elementary, that is, they take $O(1)$ time.

8.1.2 Stacks

Include files: `staticStack.h, staticStack.cpp, dynamicStack.h, dynamicStack.cpp`

Stacks are container objects which follow the **last-in first-out principle**. There are two implementations: The class `staticStack` which models sets, and the class `dynamicStack` which models multisets.

For both classes, the GOBLIN library contains precompiled code for the template instances `<TNode,TFloat>` and `<TArc,TFloat>`. Again, the choice of `TKey` is immaterial. Except for the construction of a static stack, and the destruction of a dynamic stack, all operations are elementary.

8.1.3 Priority Queues

Include files:

`basicHeap.h, basicHeap.cpp, binaryHeap.h, binaryHeap.cpp, fibonacciHeap.h, fibonacciHeap.cpp`

Synopsis:

```
template <class TItem,class TKey>
class goblinQueue : public virtual goblinDataObject
{
    void            Insert(TItem,TKey);
    TKey            Key(TItem);
    void            ChangeKey(TItem,TKey);
};
```

Priority queues are container objects to which `TItem` objects are added together with a specific priority. The item to be deleted is the set member with the highest **priority**. This value is usually called the **key** of an item, a notation which is somewhat misleading since two members may have the same priority.

GOBLIN priority queues are all static and differ only by their run time behaviour. From a theoretical point of view (only), a `fibonacciHeap` performs better than a `binaryHeap` which in turn performs better than a `basicHeap` object in general. Binary and Fibonacci heaps can be traced graphically, see Section 14.5 for some details.

8.2 Disjoint Set Systems

Include file: `abstractFamily.h`
Synopsis:

```
template <class TItem>
class goblinDisjointSetSystem : public goblinDataObject
{
```

```
    virtual void            Bud(TItem) = 0;
    virtual void            Merge(TItem,TItem) = 0;
    virtual TItem           Find(TItem) = 0;
    virtual bool            Reversible() = 0;
};
```

Disjoint set systems are objects which have been designed to perform a so-called **union-find process** on the node set of a graph object. This process is fully described by the listed operations.

The call `Bud(v)` creates a single node set containing `v`, while `Merge(u,v)` effectively merges the sets containing `u` and `v` into a single set. Each of these operations is **elementary**, that is, it requires a constant amount of time.

The call `Find(v)` returns the set containing the node `v` in terms of a **canonical element**. That is, sets are identified with one of their elements. To check whether `u` and `v` are in the same set, one would evaluate the expression `Find(u)==Find(v)`.

Example:

```
...
goblinSetFamily<TNode> F(n);
for (v=0;v<n;v++) F.Bud(v);
for (a=0;a<m;a++) F.Merge(StartNode(a),EndNode(a));
return F.Find(x)==F.Find(y);
...
```

determines the connected components of a graph, and checks whether `x` and `y` are in the same component. The running times of a `Find(v)` operation are implementation dependent.

Disjoint set families can be traced graphically, see Section 14.5 for some more details. The method `Reversible()` helps to distinguish the two available implementations at run time. The notation refers to the fact that shrinking families allow to expand sets in the reverse order.

8.2.1 Static Disjoint Set Systems

Include files: `setFamily.h, setFamily.cpp`

A `Find(v)` operation runs in $O(1)$ amortized time. That is, the running time can be considered constant if the the total number of `Find`s is large enough. If the number m of `Find`s is small, a worst-case bound is $O(\alpha(m,n))$ where α denotes some inverse of the Ackermann function. In practice, `Find(v)` operations can be considered to be elementary operation.

This data structure is particularly useful for non-weighted matching algorithms.

8.2.2 Shrinking Families

Include files: `shrinkingFamily.h, shrinkingFamily.cpp`
Synopsis:

```
template <class TItem>
class shrinkingFamily: public goblinDisjointSetSystem<TItem>
{
    shrinkingFamily(TItem,TItem,
        goblinController &thisContext=goblinDefaultContext);

    void            Bud(TItem);
    TItem           MakeSet();
    void            Merge(TItem,TItem);
    void            FixSet(TItem);

    bool            Top(TItem);
    TItem           Set(TItem);

    TItem           First(TItem);
    TItem           Next(TItem);
```

```
    void            Split(TItem);

    void            Block(TItem v);
    void            UnBlock(TItem v);
};
```

This data structure is required for weighted matching solvers and the minimum spanning arborescence method. Beside the inherited functionality, it allows to split a set S into the subsets which previously were merged into S. Actually there is a lot of new functionality associated with the class `shrinkingFamily`:

We first mention that a constructor call `shrinkingFamily(k,l,...)` specifies two dimensions k and l. The constructed shrinking family has $k+l$ elements where the indices $0, 1, \ldots, k-1$ represent **real items** whereas the indices $k, k+1, \ldots, k+l-1$ represent sets, called **virtual items**.

A `Find(v)` operation runs in $O(\log n)$ time in the worst case. The operations `Block(w)` and `UnBlock(w)` split and then shrink a virtual item again. They are needed for the construction of augmenting paths in the primal-dual method for weighted matching problems. The calls to `Block(w)` and `UnBlock(w)` take $O(n \log n)$ time alltogether for one augmenting path computation.

8.3 Hash Tables

Include file: `hashTable.h`
Synopsis:

```
template <class TItem,class TKey>
class goblinHashTable : public goblinDataObject
{
    goblinHashTable(TItem,TItem,TKey,goblinController &);

    TKey            Key(TItem);

    void            ChangeKey(TItem,TKey);
};
```

A **hash table** is a data structure which allows to store a sparse vector or matrix, say of length r in an array whose dimension is proportional to the maximal number l of non-zero entries. Actually, the size of the hash table is not l but some number $s > l$.

The definition of a hash table includes the choice of k, a **hash function** which maps the index set $\{0, 1, \ldots, r-1\}$ onto the $\{0, 1, \ldots, s-1\}$ so that the preimages of any two indices have approximately equal size, and a strategy for resolving **collisions** between two indices which need to be stored but which have the same image.

In the class `goblinHashTable`, collisions are resolved by searching through implicit set objects which model the images, the hash value is the remainder modulo s and $s = 2l$. The constructor call `goblinHashTable(r,l,k0,...)` specifies the dimensions r, l and a default value k_0 for the vector entries.

There are only two operations to be described here: A statement `Key(i)` returns the current vector entry at index i, and statement `ChangeKey(i,k)` would change this vector entry to k. In practice, only a few collisions occur so that one can treat these operations as if they were elementary. But note that `Key` and `ChangeKey` operations take $O(s)$ steps in the worst case.

A drawback of hash tables is that the number l must be known a priori or reallocations occur. Two main applications of hash tables in GOBLIN are adjacency matrices of sparse graphs and sparse subgraphs (matchings, paths, trees) of geometrical graphs. Here, the maximum size can be easily determined. Additionally, sparse matrices are implemented by hash tables.

The template parameter `TItem` must be an unsigned integer type but there are no restrictions about the data type `TKey`.

8.4 Dictionaries

Include file: `dictionary.h`
Synopsis:

```
template <class TKey>
class goblinDictionary : public goblinDataObject
{
    goblinDictionary(TIndex,TKey,goblinController&);
    TKey    Key(char*,TIndex = NoIndex);
    void    ChangeKey(char*,TKey,TIndex = NoIndex,
                      TOwnership = OWNED_BY_RECEIVER);
}
```

A **dictionary** is the pendant of an hash table which maps arbitrary C strings to values of an unspecified type `TKey`. This data structure is obviously needed to compute object indices from a tuple of node, arc or variable labels.

The constructor call `goblinDictionary(l,k0,CT)` sets the default value k_0 and the maximum number of non-zero entries l. The retrieval operation `Key(pStr,i)` takes a string and an optional object index to compute a hash value. That is, dictionaries do not only apply to the inverse mapping problem but also to support free style node and arc labels. For the first application, no index is specified at all. In the second case, an index denotes an arc or node and a missing index denotes a constant arc or node labelling.

Since references are used, `ChangeKey(pStr,k,i,tp)` operations specify if the dictionary shall work with a copy of the look-up string or if the string ownership moves to the dictionary.

8.5 Matrices

Include file: `matrix.h`
Synopsis:

```
template <class TItem,class TCoeff>
class goblinMatrix : public virtual goblinDataObject
{
    goblinMatrix(TItem,TItem) throw();

    TItem   K();
    TItem   L();

    void    Transpose();

    virtual void    SetCoeff(TItem,TItem,TCoeff) = 0;
    virtual TCoeff  Coeff(TItem,TItem) = 0;

    void    Add(goblinMatrix&);
    void    Sum(goblinMatrix&,goblinMatrix&);
    void    Product(goblinMatrix&,goblinMatrix&);

    void    GaussElim(goblinMatrix&,TFloat=0);
}
```

GOBLIN matrices are declared with two template parameters. The first parameter `TItem` specifies the type of row and column indices, the second `TCoeff` specifies the type of the matrix entries. The only precompiled template instance uses `TIndex` indices and `TFloat` coefficients.

There is a base class `goblinMatrix` which declares the mathematical functionality, and two implementational classes `denseMatrix` and `sparseMatrix` which merely have to implement the methods `Coeff()` and `SetCoeff()`. The sparse implementation is based on hash tables.

Each matrix has a row dimension `K()` and a column dimension `L()`. Matrices can be transposed implicitly by using `Transpose()` without affecting the physical representation.

The very basic matrix algebra is implemented by the methods `Add()`, `Sum()` and `Product()`. The adressed matrix object denotes the place where

the results are stored. Either two input matrices are passed as parameters or the adressed matrix also acts as an input. The running time complexities are $O(kl)$ and $O(klm)$ where m denotes the number of right-hand columns.

The method `GaussElim()` applies to squares matrices only and tries to solve a linear equation system where the matrix parameter acts as the right-hand side. The second parameter denotes the absolut value at which matrix entries are treated as zero. If omitted, the context parameter `epsilon` is used. Of course, both matrices must have compliant dimensions.

Be aware that both input matrices are manipulated by the method. If the initial left-hand matrix is regular, it is transformed to the identity. If the right-hand matrix is a column vector, it is transformed to the unique solution vector. By passing a right-hand identity, the left-hand matrix is effectively inverted. If the initial left-hand matrix is singular, an exception is thrown without reaching a triangular left-hand form.

Since the method indeed implements Gauss elimination, the complexity is $O(k^2(k+l))$ where k denotes the left-hand dimensions and l is the number of right-hand columns.

The matrix functionality may increase in future, but only to speed up certain high-level operations. It is not planned to grow a linear algebra package.

Chapter 9

Index Sets

Index sets encode lists of integers which refer to node or arc indices in a graph or to rows or columns in an LP object. Other than the container objects which have been described in the previous chapter, an index set is basically determined by its constructor call. It is not possible to manipulate the content or order of indices.

The general purpose of index sets is to supply high-level algorithms with input data. The concept is related to STL iterators, though, not as elaborate and, currently, with only few applications. What is passed to algorithms, in the STL language, are rather containers than iterators.

There are basic class templates to specify all, none or single indices of an interval $[0, 1, \ldots, r-1]$. But there are also classes to collect all graph entities with a specific property.

9.1 Interface

Include file: `indexSet.h`
Synopsis:

```
template <class TItem>
class indexSet
{
    virtual bool    IsMember(const TItem) const = 0;

    virtual TItem   First() const;
    virtual TItem   Successor(const TItem) const;
}
```

All index sets provide the following operations:

- `IsMember(i)` checks if the index i is in the set.
- `First()` returns a contained index if one exists, and an arbitrary index out of range otherwise.
- `Successor(i)` returns the successor of index I in an arbitrary but fixed ordering of all contained indices, and an arbitrary index out of range for the final index in that list.

When inheriting from this base class, it is mandatory to implement `IsMember()`. It is recommended to reimplement `First()` and `Successor()` whenever it is possible to enumerate the contained indices more efficiently than enumerating all indices in range (as the default codes do).

9.2 Templates

Include file: `indexSet.h`
Synopsis:

```
singletonIndex(TItem,TItem,
    goblinController& = goblinDefaultContext);
fullIndex(TItem,goblinController& = goblinDefaultContext);
voidIndex(TItem,goblinController& = goblinDefaultContext);
```

These three class templates are almost self explanatory: `TItem` represents the template parameter, that is the type of indices. The constructors require to specify an index range by a `TItem` valued bound. For example,

```
fullIndex<TArc>(G.M())
```

denotes the entire arc set of graph `G`. And

```
singletonIndex<TNode>(G.Root(),G.N())
```

denotes a set of nodes in the same graph, consisting of the predefined root node.

All classes implement the methods `First()` and `Successor()` in the obvious, efficient way.

9.3 Graph Based Implementations

Include file: `abstractMixedGraph.h`
Synopsis:

```
demandNodes(abstractMixedGraph&);
colouredNodes(abstractMixedGraph&,TNode);
colouredArcs(abstractMixedGraph&,TArc);
```

Include file: `abstractDigraph.h`
Synopsis:

```
supersaturatedNodes(abstractDiGraph&);
deficientNodes(abstractDiGraph&);
```

Again, the purpose of the listed classes is the obvious. But other than the basic templates described in the previous section, enumeration of the 'good' indices is not really efficient; the default implementations of the base class apply. No lists of 'good' indices are allocated!

If lists are read more than once, it is worth to generate a container object with the same content and to use this as an index set.

9.4 Containers as Index Sets

Include file: `staticQueue.h`, `staticStack.h`

Sometimes, it makes sense to use containers as index sets:

- There is no predefined index set with the desired property or this set must be post-processed.
- As pointed out in the previous section, if an index set has few elements compared with the value range, it may be inefficient to enumerate its indices several times. Exporting the indices to a container prevents from searching deselected indices.
- In particular, when the index range is divided in different sets, contiguous memory containers may operate on the same chunk of memory.

Using containers as index sets has the following limitations:

- Adding or deleting items from a container can **invalidate** running enumeration processes.
- Only the classes `staticQueue` and `staticStack` provide the index set functionality. Node based containers potentially repeat indices, and elements may be of a non-integral data type also.

Chapter 10

Branch and Bound

Branch and bound is a strategy for solving hard integer optimization problems, not only for problems on graphs. The basic concept is combinatorial and does not involve LP formulations.

The GOBLIN branch and bound module operates on vectors of a specified dimension. To the vector components, we refer as the **problem variables**. These variables have values of a scalar type `TObj` and are indexed by values of an integer type `TIndex`. Initially, there are certain upper and lower integer bounds on the problem variables, but the concrete bounds, variable values and the objective function are unknown to the branch and bound module.

In order to derive a solver for a specific integer programming problem, basically the following must be supplied:

- A fast method which solves a **relaxed problem** to optimality and returns the objective value. That is, an easier problem with fewer restrictions is solved instead of the original problem.
- A method to decide if a given integral vector is feasible for the original problem. This is the only way for the branch and bound module to get access to the combinatorial structure of a specific optimization problem.
- Code to tighten the bounds of problem variables. The generic solver changes only one variable bound at a time, but efficient implementations derive benefit from the combinatorial structure and implicitly restrict further variables.

To the relaxed problem instance, together with the original variable bounds, we refer as the **root node** (of the binary branch tree which is generate as follows).

The branch and bound scheme adds this root node to a list of active **branch nodes** or **subproblems**, and then iteratively deletes one of the active nodes and splits it into two new subproblems by putting disjoint bounds on one of the problem variables. For these new branch nodes, a relaxation is solved which either yields the optimal objective or an infeasibility proof for the relaxed subproblem.

Newly generated branch nodes which admit feasible solutions (for the relaxed subproblem) and objective values not exeeding the best known objective of a feasible solution (for the original problem), are added to the list of active branch nodes.

Sometimes, the optimal solution of a relaxed subproblem is feasible for the original problem and improves the best known solution. Then the new solution is saved (after some post optimzation steps); and the new bound decreases the number of active subproblems.

Implementing a branch and bound solver means implementing a class for the branch nodes which occur. From this class, the root nodes are explicitly instanciated and passed to the `branchScheme` constructor which internally performs the branching operations. That is, the branch nodes keep the problem dependent information, and the branch scheme models the problem independent data and methods.

10.1 Branch Nodes

Include file: `branchScheme.h`
Synopsis:

```
template <class TIndex,class TObj>
class branchNode : public goblinDataObject
{
    branchNode(TIndex,goblinController&,
            branchScheme<TIndex,TObj>* = NULL);

    TObj            Objective();
    virtual bool    Feasible();
    TIndex          Unfixed();

    virtual TIndex  SelectVariable() = 0;

    enum TBranchDir {LOWER_FIRST=0,RAISE_FIRST=1};

    virtual TBranchDir DirectionConstructive(TIndex) = 0;
    virtual TBranchDir DirectionExhaustive(TIndex) = 0;

    virtual branchNode<TIndex,TObj>* Clone() = 0;

    virtual void            Raise(TIndex) = 0;
    virtual void            Lower(TIndex) = 0;
    virtual TObj            SolveRelaxation() = 0;
    virtual TObjectSense    ObjectSense() = 0;
    virtual TObj            Infeasibility() = 0;
    virtual void            SaveSolution() = 0;
    virtual void            LocalSearch() {};
}
```

This class describes the interface between the generic branch scheme and the problem dependent branch nodes. In order to implement a concrete branch and bound solver, one just defines a subclass of `branchNode` which implements all listed prototypes. We describe all methods in the order of occurence in the branch scheme.

The method `SelectVariable()` returns the index `i` of a problem variable for which the lower and the upper bound still differ and which is relevant in the following sense: The current variable value is non-integral, or restricting this variable promises a large change of the optimal objective in one of the new subproblems, and a feasible solution in the other subproblem.

To generate the two new subproblems, the branch scheme first calls `Clone()` which returns a copy of the branch node which is currently expanded. Then, `Lower(i)` is called for the original, and `Raise(i)` is called for the clone. This restricts the value of the variable `i` to disjoint intervals in both subproblems. To the new problems, we refer as the **left** and the **right** successor. The parent node is not needed any longer!

The methods `DirectionConstructive()` and `DirectionExhaustive()` tell the branch scheme which of the two new subproblems is inspected first. A return value `RAISE_FIRST` causes that the left subproblem is inspected first. Note that `DirectionConstructive()` is called before the first feasible solution for the original problem is found; and `DirectionExhaustive()` is called afterwards.

Then, the branch scheme evaluates the left and the right subproblem using the following methods:

- `SolveRelaxation()` actually computes the objective value for the relaxed subproblem while `Objective()` retrieves the cached objective value when possible. The objective value is compared with the class constant `Infeasibility()` in order to detect infeasible relaxed subproblems. It is not allowed that `SolveRelaxation()` operates on the original graph or LP data structures since these are needed to save the best solution found so far.

- `ObjectSense()` is a class constant and specifies either a maximization or minimization problem.

- `Unfixed()` returns the number of variables for which the lower and the upper bound still differ. A subproblem with `Unfixed()==0` must be either feasible for the original problem or infeasible for the relax-

ation. For all other subproblems, let `SelectVariable()` return some branching variable.

- `Feasible()` checks if the optimal solution returned by `SolveRelaxation()` is feasible for the original problem. The default implementation considers every fixed solution to be problem-feasible.
- If the relaxed optimum is feasible and improves the best known solution for the original problem, then `SaveSolution()` is called in order to send this solution to the original graph or LP data structures. It is useful to implement `LocalSearch()` such that a post optimization procedure is applied to all saved solutions. This local search method should be defined for the hidden graph objects rather than the branch nodes so that it can be used independently from branch and bound.

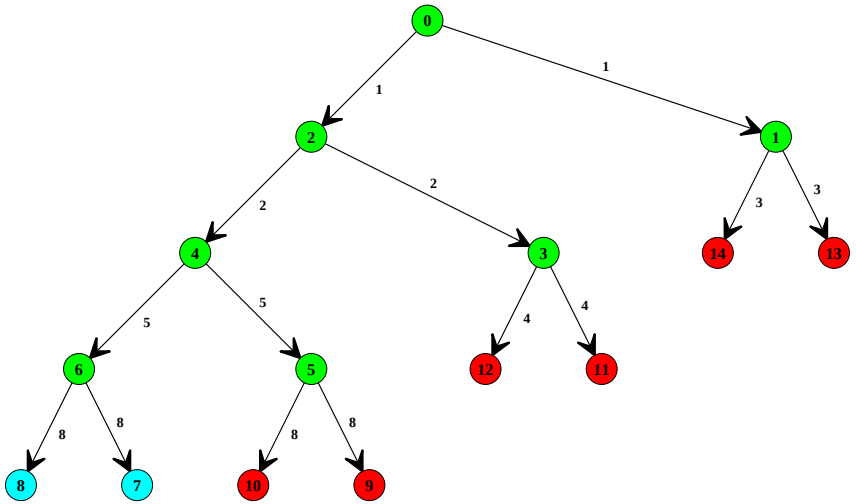


Figure 10.1: A Branch Tree

Nothing else is needed for an executable branch and bound solver. With respect to efficiency, the following should be kept in mind:

- Let the branch nodes consume as few as possible memory if you want to solve large scale instances with optimality proof. In the constructive mode, the number of active nodes is somewhat like the DFS search depth and memory usage is not the most important issue.
- The tradeoff between the running times and the obtained objective of `SolveRelaxation()` can be bothering. It depends on the DFS search depth and hence on the instance sizes. Generally, the quality of the obtained bounds is more important than the running times.
- Consider if `LocalSearch()` is beneficial. It is not obvious whether feasible solutions obtained by the branch scheme are locally optimal. On the other hand, this method will be called only rarely.

10.2 Generic Algorithm

Include file: `branchScheme.h`
Synopsis:

```
template <class TIndex,class TObj>
class branchScheme : public goblinDataObject
{
private:

    branchNode<TIndex,TObj> *firstActive;

    diGraph *       Tree;

protected:

    void Optimize() throw();
```

```
    bool Inspect(branchNode<TIndex,TObj> *);
    branchNode<TIndex,TObj> *SelectActiveNode();
    void QueueExploredNode(branchNode<TIndex,TObj> *);
    void StripQueue();

public:

    TIndex  nActive;
    TIndex  nIterations;
    TIndex  nDFS;
    bool    feasible;

    TObj    savedObjective;
    TObj    bestBound;

    enum TSearchLevel {
        SEARCH_FEASIBLE = 0,
        SEARCH_CONSTRUCT = 1,
        SEARCH_EXHAUSTIVE = 2};

    TSearchLevel level;

    branchScheme(branchNode<TIndex,TObj> *,TObj,
                 TSearchLevel = SEARCH_EXHAUSTIVE);

    enum TSearchState {
        INITIAL_DFS = 0,
        CONSTRUCT_BFS = 1,
        EXHAUSTIVE_BFS = 2,
        EXHAUSTIVE_DFS = 3};

    TSearchState SearchState();

    unsigned long   Size();
    unsigned long   Allocated();
};
```

Once a class of branch nodes is available, the application of the branch and bound algorithm is as simple as possible: Just instanciate the root node (an object which inherits from `branchNode`), and then a `branchScheme` object. The `branchScheme` constructor takes the root node as a parameter and implicitly calls the solver method `Optimize()`. As the second constructor parameter, either pass the objective value of a solution known in advance, or the `Infeasibility()` constant if no feasible solution is known.

The `Optimize()` method consists of the main loop and of iterated calls to `SelectActiveNode()` and `Inspect()`. The method `SelectActiveNode()` selects an active subproblem which is split as described in the previous section; `Inspect()` evaluates the new subproblems; and `StripQueue()` deletes irrelevant branch nodes when an improving feasible solution is found.

We have already described the problem specific parts of branch and bound codes, and how these parts apply to the general algorithm. We have seen that the branch strategy is partially controlled by the methods `SelectVariable()`, `DirectionConstructive()` and `DirectionExhaustive()`. The method `SelectActiveNode()` contributes the general strategy of switching between best-first and depth-first steps. This strategy depends on the specified search level:

- `SEARCH_CONSTRUCT`: Branching starts with a certain (rather large) number of DFS steps in order to obtain an initial bound. After that, series of depth-first steps alternate with a few best-first steps so that the solver cannot get stuck in non-profitable regions of the branch tree. When getting close to `maxBBNodes`, the maximal configured number of active branch nodes, only DFS steps are performed.

- `SEARCH_EXHAUSTIVE`: It is assumed that the initial bound is very close to the optimum. Best-first steps are performed unless the number of

active branch nodes is getting close to `maxBBNodes*100` (in that case, DFS steps are performed).

- `SEARCH_FEASIBLE`: The search strategy is the same as in the `SEARCH_CONSTRUCT` case, but the solver stops when the first feasible solution has been found.

If the number of active nodes exeeds `maxBBNodes*100`, or if the total number `nIterations` of solved subproblems exeeds `maxBBIterations*1000`, then the solver halts in any case.

The current state of computation is given by `SearchState()`. Especially for `SolveRelaxation()` codes, it can be useful to retrieve this search level and to apply a dual bounding procedure which is worse but faster to compute in the inital DFS phase.

The transitions between the search states depend on the `branchNode::depth` parameter which is copied once from the root node to the branch scheme object. It basically denotes an estimation the depth of the branch tree. This may be the maximum number of non-zero problem variables, for example. If this depth is underestimated, the solver may halt prematurely by reaching the configured maximal number of active branch nodes; or the constructive DFS search is interrupted before any leaves of the branch tree have been considered. If the depth is overrated, no best-first steps can take place. By default, the `depth` is the number of problem variables.

Iteration	Objective	Free	Status	Best Bound	Best Lower	Active	Select
0	-5	12	QUEUED	0	-5	1	DEPTH
1	-4	11	QUEUED	0	-5	1	
2	-4	7	QUEUED	0	-5	2	DEPTH
3	-4	6	QUEUED	0	-5	2	
4	-4	3	QUEUED	0	-5	3	DEPTH
5	-3	2	QUEUED	0	-5	3	
6	-4	1	QUEUED	0	-5	4	DEPTH
7	-3	0	SAVED	-3	-5	3	
8	-4	0	SAVED	-4	-5	3	DEPTH
9	-3	1	CUTOFF	-4	-5	2	
10	-3	0	CUTOFF	-4	-5	2	DEPTH
11	-3	5	CUTOFF	-4	-5	1	
12	-4	3	CUTOFF	-4	-5	1	DEPTH
13	-4	10	CUTOFF	-4	-5	0	
14	-4	7	CUTOFF	-4	-5	0	

Figure 10.2: A Branching Protocol

The variable `Tree` maintains the **branch tree** which can be displayed graphically. An example of a branch tree can be found in Figure 10.1 and the corresponding logging information is shown in Figure 10.2. The labels of the tree nodes denote the iteration number, and the arc labels denote the branching variable.

10.3 Implementations

10.3.1 Stable Sets

Include file: `branchStable.h`

In the class `branchStable`, all problem variables are associated with graph nodes. Nodes can either be unfixed, selected or excluded. Every time a node is selected, all of its neighbours are excluded. A node selection corresponds to a call to `Raise()`, excluding a node is done by calling `Lower()`.

Let X denote the set of selected nodes and let $\Gamma(X)$ denote the set of excluded nodes. The method `SelectVariable()` returns the index of a node which has minimum degree in the graph restricted to $V \setminus (X \cup \Gamma(X))$. By the definitions of `DirectionConstructive()` and `DirectionExhaustive()`, the selected node is selected first in the DFS search, and then excluded.

The constructor of the root node determines a heuristic clique cover. This clique cover is maintained for all subproblems. For a given subproblem, the method `SolveRelaxation()` searches for all cliques which contain at least one unfixed node. The number of these cliques plus the number of selected nodes gives an upper bound which is returned.

A possible extension of this class to the weighted stable set problem seems straightforward and desirable.

10.3.2 Symmetric TSP

Include file: `branchSymmTSP.h`

In the class `branchSymmTSP`, all problem variables are associated with graph edges. Every branch and bound node owns a copy of the original graph where the some of the arc capacities have been restricted to 0 or 1. More explicitly, `Lower()` sets the upper capacity bound to 0 and `Raise()` sets the lower capacity bound to 1. A raise operation checks if two incident edges have been selected for one of the end nodes. Both procedures can reduce the complexity by implicitly fixing arcs.

The arc returned by `SelectVariable()` is always in the current 1-tree, and one of the end nodes has degree higher than 2 in the 1-tree. If possible, the arc is chosen such that the high degree end node is already adjacent to a fixed arc. Arcs are selected first in the DFS search, and then excluded.

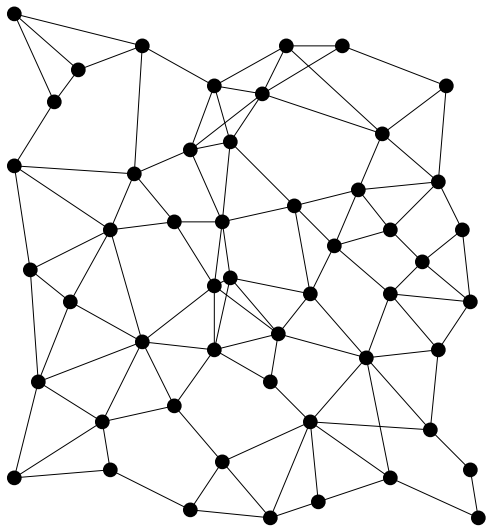


Figure 10.3: A Candidate Graph

The bounding procedure `SolveRelaxation()` is as follows: First, it is checked that for all nodes at most two incident edges have been selected, and that the subproblem is still 2-connected. If one of these conditions is violated, `InfFloat` is returned.

Otherwise, a minimum spanning tree method is called which has been modified to compute the optimal extension of the selected arcs to a 1-tree. The used node potentials are inherited from the parent branch node. The root node for the 1-tree computations is the same for all subproblems.

If `methRelaxTSP2>0`, if the inital DFS phase has been passed, and if the the length of the optimal 1-tree does not exeed the feasibility bound, subgradient optimization is applied to increase the relaxation bound. This procedure stops immediately, when the 1-tree length exeeds the feasibility bound. When setting `methRelaxTSP2=1`, the subgradient method runs in the fast mode. When setting `methRelaxTSP2=2`, the number of branch nodes is minimized.

If the original graph is complete and `CT.methCandidates==k` where $k \geq 0$, the constructor of the root branch node computes a candidate subgraph which consists of

- the current predecessor arcs (usually the best known tour)
- 20 random tours (one should set `methLocal==1` to force the tours to local optimality)
- and the k least cost edges incident with every graph node.

It has been experienced that even for $k = 0$, often an optimal tour can be obtained from the candidate graph. A TSP candidate graph is shown in Figure 10.3.

It is useful to run the branch and bound procedure twice. First, perform a candidate search with a limit on the number of branch nodes (Take care that some sequences of best-first steps can occur). Then either run the candidate search again (The candidate graph includes the tour found before,

so the "good" arcs are accumulated) or run an exhaustive search which can also improve tours if the gap is small.

Experiments show that the TSP solver is able to evaluate complete graphs with less than 150 nodes, and to candidate graphs with less than 200 nodes. See the appendix for some computational results.

10.3.3 Asymmetric TSP

Include file: `branchAsyTSP.h`
Nearly all statements of the previous section also apply to the TSP solver for directed graphs. However, the applied spanning tree method is much less performant, and the subgradient optimization is converging slower than in the undirected setting. Also, only node insertion is available for local search so that optimal tours are found later. Experiments have turned out that it is possible to completely evaluate digraphs up to a size of 50 nodes.

10.3.4 Node Colouring

Include file: `branchColour.h` The class `branchColour` defines an enumeration scheme rather than a branch and bound solver. That is, the solver does not minimize the number k of colours but tries to find a k-colouring for a given number k. The strategy is as follows:

Initially, all nodes are **active**. Nodes become inactive if they are coloured or dominated by the current (partial) colouring. Here a node v is called **dominated** if every consistent extension of the current colouring to the active nodes can be extended to v consistently. Some nodes can be marked dominated a priori. For example, if the graph is planar and $k \geq 6$, then all nodes are marked dominated.

The constructor for the master problem checks if the node colour data structure of the original graph provides a clique. In the positive case, the clique nodes are coloured $0, 1, 2, \ldots$ immediately. Otherwise a maximum degree node is coloured with colour 0, and one of its neighbours is coloured with colour 1.

Here, `SelectVariable()` determines the minimum available colour c and returns an active node u which can be coloured with c. A call `Lower(u)` will fix this colour and `Raise(u)` will forbid this colour for u. The former method calls `SetColour(u,c)` which actually fixes the colour of u and checks for every active neighbour of u if the number of conflicts falls below k. In that situation, `Reduce(u)` is called which marks the node as dominated.

The method `SolveRelaxation()` returns number of colours which are used in this subproblem or detects infeasibility. If there are nodes for which only one colour is available, then `SetColour()` to fix these node colours. If there are no active nodes left, the dominated nodes are coloured by a call to the method `Complete()`.

10.3.5 Maximum Cut

Include file: `branchMaxCut.h`
In a `branchMaxCut` object, the problem variables represent the nodes of a graph. A zero value denotes a left hand node, and a value of 2 denotes a right hand node. A variable value 1 represents a node which is not fixed yet. Depending on the status of the end nodes, arcs are either selected, dismissed or unfixed.

In the undirected case, it is possible to extend the partial cut by at least 1/2 of the unfixed edges. To this end, `SelectVariable()` returns a maximum capacity unfixed node, and `DirectionConstructive()` guides a DFS search to add this node to the more profitable component.

The dual bound computed by `SolveRelaxation()` counts all selected and all unfixed edges. This simple bounding procedure performs really poor, and only allows to evaluate undirected graphs with up to 30 nodes. It is obvious (but not implemented yet) that the bounds can be improved by considering odd length cycles and chains of directed arcs.

10.3.6 Mixed Integer Programming

Include file: `branchMIP.h`

The class `branchMIP` implements a plain integer branch and bound. That is, there is no code for cutting plane generation and pool management yet. The problem variable returned by `SelectVariable()` has a maximum fractional remainder among all integer variables; and a solution is considered `Feasible()` when all fractional remainders fall beyond the context parameter `epsilon`.

Part III

Methods

Chapter 11

Prototypes and Data Structures

11.1 Graph Definition

In this section, we describe how problem instances, namely graph objects, are specified in GOBLIN. In Section 11.2, we will also discuss the potential solutions of graph optimization problems.

We have already mentioned that a class of graph objects may either represent physical objects or logical views of other data objects. Hence we are concerned with prototype methods rather than data structures.

11.1.1 Incidences and Adjacencies

Synopsis:

```
class abstractMixedGraph
{
protected:

    goblinHashTable<TArc,TArc> *        adj;

public:

    virtual TArc    First(TNode) = 0;
    virtual TArc    Right(TArc,TNode) = 0;

    virtual TNode   StartNode(TArc) = 0;
    virtual TNode   EndNode(TArc);

    virtual TArc    Adjacency(TNode,TNode);
    void            MarkAdjacency(TNode,TNode,TArc);
    void            ReleaseAdjacencies();
}
```

Node incidences are the very core of any implementation of graph objects. They can be accessed by iterator objects which were discussed in Chapter 7 and which in turn require an implementation of the methods `First()` and `Right()` (see Section 7.1 for the details).

In a similar way any graph implementation must provide **arc incidences**, that are the end nodes of a given arc, by defining a method `StartNode`. A call to the generic method `EndNode()` effectively determines the start node of the reverse arc. We mention that `StartNode()` utilizes an array in sparse graphs, but merely evaluates the arc indices in dense graphs.

Once node and arc incidences are available, GOBLIN can automatically compute **node adjacencies**, that are arcs joining two given nodes, by maintaining an adequate data structure. Hence node adjacencies are not really graph defining data structures but rather redundant information which can be generated and disposed dynamically.

The data structure used for node adjacencies is a hash table which is generated by the first call to `Adjacency()`. Note that the operations on this hash table are not bounded polynomially, but the computation of an adjacency can be practically considered an elementary operation. The generation of this hash table can be suppressed by disabeling the context flag `methAdjacency`.

The returned arc is always non-blocking. That is, in digraphs, no backward arcs are returned. If the adjacency is ambiguous (that is, if parallel arc exist), the returned arc index is the minimal one.

The method call `MarkAdjacency(u,v,a)` specifies the arc a to be returned by `Adjacency(u,v)`. This is needed to maintain the adjacency table during graph insertion and deletion operations.

Some classes override the generic implementations of `EndNode()` and `Adjacency()` for reasons of efficiency. In any case, the generic code is helpful for the writing of preliminary versions of graph implementations.

11.1.2 Arc Capacities and Node Demands

Synopsis:

```
class abstractMixedGraph
{
    virtual TCap    UCap(TArc) = 0;
    virtual TCap    MaxUCap();
    virtual bool    CUCap() = 0;

    virtual TCap    LCap(TArc) = 0;
    virtual TCap    MaxLCap();
    virtual bool    CLCap() = 0;

    virtual TCap    Demand(TNode);
    virtual TCap    MaxDemand();
    virtual bool    CDemand();
}
```

Arc capacities and node demands are numbers which determine the set of feasible subgraphs respectively flows of a network programming problem. Although not checked exhaustively in GOBLIN, arc capacities and node demands are supposed to satisfy some properties:

For digraphs and flow networks, the node demands must resolve, that is, the sum of demands must be zero. The arc capacities have to be non-negative, but may be non-integral or even infinite. Needless to say that the lower bounds should not exeed the respective upper bounds.

For undirected graphs, all arc capacities and node demands must be non-negative numbers which are either integral or infinite. The sum of the node demands must be an even number which is at most twice the sum of the arc capacities.

The methods `MaxUCap`, `MaxLCap`, `MaxDemand` return the respective maximum label and the methods `CUCap`, `CLCap`, `CDemand` decide whether the labels are constant or not.

11.1.3 Length Labels

Synopsis:

```
class abstractMixedGraph
{
    virtual TFloat  Length(TArc) = 0;
    virtual TFloat  MaxLength();
    virtual bool    CLength() = 0;
}
```

Length labels install linear objective functions which apply to most kinds of network programming problems. For physical graph objects, length labels can either be implemented by a simple array or determined by the geometric embedding of the graph.

More explicitly, if the context variable `methGeometry` is zero, length labels are read from an array data structure. Otherwise a certain metric of the graph embedding is evaluated. The methods `MaxLength` return the maximum length label and the methods `CLength`, `CDemand` decide whether the labels are constant or not.

`methGeometry`	**0**	Explicit length labels
	1	Manhattan distances
	2	Euclidian distances
	3	Maximum coordinate distances
	3	Spheric distances

Table 11.1: Selection of Length Labels

11.1.4 Geometric Embedding

Synopsis:

```
class abstractMixedGraph
{
    virtual TFloat  C(TNode,TDim);
    virtual TFloat  CMax(TDim);
    virtual TDim    Dim();
}
```

Any class may or may not provide a geometrical embedding for their graph objects. This embedding is needed for the graphical display. In case of physical graphs, the geometrical embedding may also determine the length labels.

The method `Dim()` specifies the **dimension** of the embedding, that is the number of coordinates of each graph node. The actual `i`th coordinate of the node `v` can be obtained by `C(v,i)`. A call `CMax(i)` returns the maximum extension of the graph in the `i`th coordinate.

We mention that the graphical display uses the first two coordinates only, and hence logical views are generally embedded into two-dimensional space. Note also that the embedding includes the possible arc bend nodes and the alignment points for arc labels.

11.1.5 Layout

Synopsis:

```
class abstractMixedGraph
{
    virtual TNode   NI();
    virtual TNode   Align(TArc);
    virtual bool    CAlign();
    virtual TNode   Interpolate(TNode);
    virtual bool    CInterpolate();
    virtual bool    HiddenNode(TNode);
    virtual bool    HiddenArc(TArc);
}
```

Here we have listed several graph properties which do not influence the behaviour of any problem solver but which are sometimes necessary to enhance the graphical output.

The boolean functions `HiddenNode` and `HiddenArc` suppress the drawing of certain nodes and arcs. The call `Align(a)` returns the potential first artificial node for the arc `a`. Actually, this point determines the alignment of the arc label. If `CAlign()` is true, no alignment points and no bend nodes are present, and the labels are aligned by a generic strategy.

Using `Align(a)` as the initial point, the ordered list of bend nodes can be reconstructed by the iterated call of `Interpolate`. If `CInterpolate()` is true, no bend nodes are present, and the graph arcs are simple lines.

The interpolation and alignment points together are the **artificial nodes**. The total number of artificial nodes is returned by `NI()`.

So far, the layout of logical views is not too elaborate, in particular, the definitions of `Align` and `Interpolate` are only dummies. In a later release, these two methods may also control the drawing of graph nodes.

11.1.6 Arc Orientations

Synopsis:

```
class abstractMixedGraph
{
```

```
    virtual bool    Blocking(TArc) = 0;
}
```

This functionality is needed to distiguish directed arcs from undirected arcs. More explicitly, `Blocking(a)` is true if the arc `a` is directed, but a backward arc. In most classes, this method returns a constant, but for physical mixed graphs the method is the public interface to an array data structure.

11.2 Potential Solutions

11.2.1 Predecessor Labels

Synopsis:

```
class abstractMixedGraph
{
protected:

    TArc *          P;

public:

    void            InitPredecessors();
    TArc            Pred(TNode);
    void            SetPred(TNode,TArc);
    void            ReleasePredecessors();
    void            WritePredecessors(goblinExport*);
    void            ReadPredecessors(goblinImport*);

    void            ExtractTrees();
    void            ExtractTree(TNode);
    TNode           ExtractPath(TNode,TNode);
    TNode           ExtractCycles();
    void            Extract1Matching();
    void            ExtractEdgeCover();
}
```

The general purpose of this data structure is to keep track of paths, cycles, trees and any disjoint collection of such subgraphs with a minimum of computer storage. Since predecessor labels define arborescences rather than undirected trees, subgraphs can be searched much faster if they are encoded into predecessor labels. Hence at least shortest path algorithms and TSP algorithms depend on this data structure.

There is a public method `Pred` to read the current predecessor arc of a given node, and methods `SetPred` and `InitPredecessors` which manipulate the data structure in the obvious way. In addition, one can assign the complete set of predecessors with a subgraph present by the subgraph data structure. There are several such methods each of which requires a special kind of subgraph:

- `ExtractTrees()` generates a set of rooted trees covering all graph nodes and corresponding to the connected components of the subgraph. An exception `ERCheck` is returned if the subgraph contains cycles.

- `ExtractTree(r)` generates a tree rooted at `r`. If the subgraph is disconnected or if the subgraph contains cycles, an exception `ERCheck` is returned. If the context flag `meth1Tree` is enabled, a unique cycle must exist, and `r` must be on this cycle.

- `ExtractPath(u,v)` generates a directed path starting at `u` and ending at `v`. An exception `ERCheck` is returned if `u` and `v` are disconnected in the subgraph of if the connected component of `u` and `v` contains **branches**, that are nodes with degree at least 3.

- `ExtractCycles()` generates a set of directed cycles which cover all graph nodes. Such a subgraph is called a **2-factor**. An exception `ERCheck` is returned if the original subgraph is not a 2-factor.

- `Extract1Matching()` checks if the arcs of the subgraph are pairwise non-ajacent. If so, the predecessor labels are assigned with this **1-matching** such that predecessors are always arcs with even indices. If there are adjacent arcs, an exception `ERCheck` is returned.
- `ExtractEdgeCover()` checks if the arcs of the subgraph are pairwise non-ajacent. If so, the predecessor labels are assigned with this **1-matching** and augmented to an edge cover. If the input subgraph is a maximum cardinality matching, a minimum edge cover results. The graph must not have isolated nodes.
- `ExtractColours()` generates from the node partition data structure equivalent node colours such that the colour classes occur consecutively.

11.2.2 Subgraphs

Synopsis:

```
class abstractMixedGraph
{
    void            InitSubgraph();
    void            WriteSubgraph(goblinExport*);
    void            ReadSubgraph(goblinImport*);

    virtual void    AddArc(TArc,TFloat) = 0;
    virtual void    OmitArc(TArc,TFloat) = 0;
    virtual TFloat  Sub(TArc) = 0;
    virtual void    SetSub(TArc,TFloat);

    TCap            Cardinality();
    TCap            Length();

    void            AddToSubgraph(TNode = NoNode);
}
```

A **subgraph** is a (possibly fractional) assignment of labels to the graph arcs which satisfies the capacity bounds. If integral, a subgraph label `Sub(a)` may be interpreted as the number of arcs in the subgraph which are parallel to `a`. A subgraph of a directed graph is also called **pseudo-flow**.

This data structure differs from the other potential solutions by the fact that it is implementation dependent. That is, a subgraph of a sparse graph object is a vector, a subgraph of a dense graph object essentially is a hash table, and subgraphs of logical views can be defined completely differently.

More explicitly, every class must implement three methods `AddArc`, `OmitArc` and `Sub`. The first two methods increase respectively decrease the subgraph label by a specified amount. If no subgraph data structure is present, `Sub` should return the lower capacity bound. Every implementation of `AddArc` and `OmitArc` has to check that the resulting subgraph still observes the capacity bounds. The method `SetSub` depends on `AddArc` and `OmitArc` and can be used to set subgraph labels explicitly.

On the other hand, a subgraph may be **infeasible**, that is, node degrees and node demands may differ. A subgraph may also be **non-optimal**, that is, there is a subgraph whose **weight** $\sum_a length(a)sub(a)$ is smaller. The length of a subgraph can be computed in $O(m)$ time by `Weight()`. A corresponding method `Cardinality()` exists which determines the **cardinality** $\sum_a sub(a)$ of a subgraph.

The method `InitSubgraph()` initializes the data structure with a subgraph identical to the lower degree bound. Finally, the method `AddToSubgraph` takes the characteristic vector of the subgraph determined by the predecessor labels, and adds it to the subgraph data structure. If an optional node v is specified, only the way back to the root of v respectively the cycle containing v is added.

When working with subgraphs of dense graph objects, it is necessary either to disable the subgraph hash table or to initialize the subgraph data structure its maximum cardinality `card` by calling the struct method `NewSubgraph(card)`. See also Section 6.2.2.

11.2.3 Flow Labels

Synopsis:

```
class abstractMixedGraph
{
    virtual TFloat  Flow(TArc) = 0;
    virtual void    Push(TArc,TFloat) = 0;
}

class abstractBalancedFNW
{
    virtual TFloat  BalFlow(TArc) = 0;
    virtual void    BalPush(TArc,TFloat) = 0;

    virtual void    Symmetrize() = 0;
    virtual void    Relax() = 0;
}
```

Flow labels are an alias for subgraphs which is used for network flow problems. That is, `Flow(a)` and `Sub(a)` return the same value, and `Push(a,lambda)` does the same as `AddArc(a,lambda)` or `OmitArc(a,lambda)`, depending on the parity of the arc a. Note that the node degrees are affected as well.

In balanced flow networks, a symmetric version of flow labels exists which admit the analogous operations `BalFlow` and `BalPush`. Note that the call `BalPush(a,lambda)` essentially performs both `Push(a,lambda)` and the symmetric operation `Push(a^2,lambda)`.

There is a logical or even physical distinction between symmetric and non-symmetric flow labels. Flow labels can be symmetrized explicitly by calling `Symmetrize()`, and the non-symmetric labels are initilized with their balanced counterparts by calling `Relax()`.

11.2.4 Node Degrees

```
class abstractMixedGraph
{
protected:

    TFloat *    sDeg;
    TFloat *    sDegIn;
    TFloat *    sDegOut;

public:

    void        InitDegrees();
    void        InitDegIO();
    TFloat      Deg(TNode);
    TFloat      DegIn(TNode);
    TFloat      DegOut(TNode);
    TFloat      Divergence(TNode);
    void        AdjustDegrees(TArc,TFloat);
    void        ReleaseDegrees();
}
```

Node degrees are rather an auxiliary data structure than a potential solution. They are completely determined by the subgraph labels. The call `Deg(v)` returns the sum over all subgraph labels of undirected arcs adjacent with the node `v`. In the same manner, `DegIn(v)` is the sum of all directed arcs with end node `v`, and `DegOut(v)` is the sum of all directed arcs with start node `v`.

The necessary data structures are generated by the first calls of `Deg`, `DegIn` or `DegOut` respectively. To keep the degree labels and the subgraph labels compliant, every implementation of `AddArc` and `OmitArc` must include a call to `AdjustDegrees`. If they are not needed any longer, degree labels may be disposed (other than the subgraph data structure).

In order to obtain the node degrees in the original graph rather than in a subgraph, one may set the lower capacity bounds to the value of the upper bounds. Then the subgraph multiplicities and the node degrees will be set implicitly.

11.2.5 Distance Labels

Synopsis:

```
class abstractMixedGraph
{
protected:

    TFloat *        d;

public:

    void            InitLabels(TFloat = InfFloat);
    virtual TFloat  Dist(TNode);
    void            SetDist(TNode,TFloat);
    void            ReleaseLabels();
    void            WriteLabels(goblinExport*);
    void            ReadLabels(goblinImport*);
}
```

Distance labels are not only utilized by shortest path algorithms, but more generally to store the length of the paths which are encoded into the predecessor labels. They are also used to specify cuts (see Section 11.2.7).

A distance label may be read the method `Dist` and changed by `SetDist`. There is an initialization procedure `InitLabels` which sets some default value. This initilization routine supports the reusage of the data structure to avoid repeated reallocation. Note that most algorithms access the data structure directly for reasons of efficiency.

The methods `ReadLabels` and `WriteLabels` admit file import and export of the data structure. The file format forms part of the general file format for graph objects presented in Section 18.4. Equivalent statements hold for the other data structures described in what follows.

11.2.6 Node Potentials

Synopsis:

```
class abstractMixedGraph
{
protected:

    TFloat *        pi;

public:

    void            InitPotentials(TFloat = 0);
    TFloat          Pi(TNode);
    void            SetPotential(TNode,TFloat);
    void            PushPotential(TNode,TFloat);
    void            UpdatePotentials(TFloat);
    void            ReleasePotentials();
    void            WritePotentials(goblinExport*);
    void            ReadPotentials(goblinImport*);

    virtual TFloat  RedLength(TArc);
}
```

Node potentials form the LP dual solutions of network flows and matchings. This data structure can be accessed directly by network flow algorithms. Even if not accessed directly, they come into play via the reduced or modified length labels (see Section 11.1.3). Reduced length labels also appear in the subgradient method `TSPSubOpt1Tree` for the TSP.

The public interface allows to read node potentials (`Pi`), to set a single potential to the value (`SetPotential`) and to add some amount to the current potential (`PushPotential`).

If this data structure is not present, all potentials are treated as zero. Accordingly, a call to `InitPotentials` generates the data structure and sets all potentials to zero. Note that `InitPotentials` may be called by `SetPotential` and `PushPotential` recursively.

A call `UpdatePotentials(alpha)` adds the current distance labels to the current potentials. But note that only those potentials are changed for which the corresponding distance label is less than `alpha`. This procedure is used by the min-cost flow algorithm `EdmondsKarp2` which recursively calls the `Dijkstra` method. The latter procedure searches the reduced length labels but keeps the result via the distance labels. The threshold `alpha` is needed since the `Dijkstra` graph search is incomplete in general.

The reduced length labels combine the length labels and the node potentials to the optimality certificates well-known in linear programming. If `a` denotes some arc with end nodes `u` and `v`, then `RedLength(a)` is defined as `Length(a)+Pi(u)+Pi(v)` in undirected graphs, and as `Length(a)+Pi(u)-Pi(v)` in directed graphs. Shortest path problems and weighted network flow problems are solved optimally if and only if all reduced length labels are non-negative.

There are two further methods `ModLength` and `RModLength` which extend the concept of reduced cost optimality to balanced network flow and matching problems. Since the computation of modified length labels is expensive, `RModLength` allows the recursive computation whereas `ModLength` utilizes an explicit data structure.

11.2.7 Node Colours

Synopsis:

```
class abstractMixedGraph
{
protected:

    TNode *         colour;

public:

    void            InitColours(TNode = NoNode);
    virtual TNode   Colour(TNode);
    void            SetColour(TNode,TNode);
    void            ReleaseColours();
    void            WriteColours(goblinExport*);
    void            ReadColours(goblinImport*);

    void            UpdateColours();
    void            ExtractCut();
    void            ExtractBipartition();
    void            ExtractColours();
}
```

Node colours are not only used to store graph colourings, but can also represent cuts and connected components with a minimum of computer storage. For example, the matching procedures return the blossom structure as node colours. More explicitly, the `gra2bal` destructor assigns to each node the blossom base as its colour.

`ExtractCut()` assigns colour zero to all nodes with finite distance labels, and colour 1 to the remaining nodes. `ExtractBipartition()` assigns colour zero to all nodes with even finite distance labels, and colour 1 to the remaining nodes. `ExtractColours()` saves a (non-persistent) node partition into a consecutive series of node colours.

11.2.8 Partitions of the Node Set

Synopsis:

```
class abstractMixedGraph
{
protected:
```

```
    goblinDisjointSetSystem<TNode> *    partition;

public:

    virtual void    InitPartition();
    virtual void    Bud(TNode v);
    virtual void    Merge(TNode u,TNode v);
    virtual TNode   Find(TNode v);
    virtual void    ReleasePartition();
}
```

The partition data structure is a disjoint set union data structure. In contrast to the most data structures which were discussed here, a partition is a high-level data structure which cannot be written to a file and reconstructed properly.

The methods are only shortcuts for the operations described in Section 8.2.1. That is, `Bud(v)` generates a one elementary set which consists of v, `Merge(x,y)` unifies the sets containing x and y, and `Find(w)` returns the canonical element of the set cotaining w.

11.2.9 Blossoms

Synopsis:

```
class abstractBalancedFNW
{
protected:

    TNode *         base;

public:

    void            InitBlossoms();
    void            ReleaseBlossoms();
    TNode           Base(TNode v);
    void            Shrink(TNode u,TNode v);
}
```

Blossoms are the symmetric specialization of the node partition data structure, and override the general definitions. In contrast to general partitions, complementary nodes are always in the same part of thepartition. The method `Base` does not return an arbitrary canonical element, but a special node which is called the **blossom base**.

This node can be defined algorithmically as follows: `Bud(v)` implies that `Base(v)==v`. If one has `Base(u)==v` and `Base(x)==y`, then the operation `Shrink(u,x)` implies that `Base(u)==Base(x)==v`. That is, the first parameter of `Shrink` determines the blossom base.

11.2.10 Props and Petals

Synopsis:

```
class abstractBalancedFNW
{
protected:

    TArc *          prop;
    TArc *          petal;

public:

    void            InitProps();
    void            ReleaseProps();

    void            InitPetals();
    void            ReleasePetals();
}
```

Props and petals determine augmenting paths in a balanced flow network. The labels are set by the balanced network search methods which are discussed in Section 13.11. Augmenting paths can be extracted by the recursive call of the methods `Expand` and `CoExpand`. The resulting path is assigned to the predecessor labels.

11.2.11 Odd Cycles

Synopsis:

```
class abstractBalancedFNW
{
protected:

    TArc *          Q;

public:

    void            InitCycles();
    void            ReleaseCycles();
}
```

This data structure is used quite analogously to the predecessor labels, in particular, to store a system of disjoint cycles in a balanced flow network. These odd cycles occur during the symmetrization of the flow labels and denote the arcs with non-integral flow labels after a call of `CancelEven()`. The odd cycles are cancelled again by `CancelOdd()` and `CancelPD()`. Both methods form part of the `Anstee` and the `EnhancedPD` method. See Sections 13.12.4 and 13.13.2 for the details.

11.3 Manipulating Graphs

The following methods are available for physical graph objects only. That is, we are now talking about data structures, not only about prototype methods.

11.3.1 Changes of the Incidence Structure

Synopsis:

```
class sparseGraphStructure
{
    TArc    InsertArc(TNode,TNode,TCap,TCap,TFloat);
    TArc    InsertArc(TNode,TNode);
    TNode   InsertNode();
    TNode   InsertartificialNode();
    TNode   InsertAlignmentPoint(TArc);
    TNode   InsertBendNode(TNode);

    void    SwapArcs(TArc,TArc);
    void    SwapNodes(TNode,TNode);
    void    FlipArc(TArc);
    void    CancelArc(TArc);
    void    CancelNode(TNode);
    void    DeleteArc(TArc);
    void    DeleteNode(TNode);
    void    DeleteArcs();
    void    DeleteNodes();
    void    ContractArc(TArc);
    void    IdentifyNodes(TNode,TNode);

    void    ReSize(TNode,TArc);
}

class denseGraphStructure
{
    TArc    InsertArc(TArc,TCap,TCap,TFloat);
}
```

In sparse graph objects, `InsertArc(u,v,uu,ll,cc)` generates a new incidence with start node u, end node v, upper capacity bound uu, lower capacity bound ll and length label cc. In order to avoid multiple reallocation of the data structures when several new arcs are generated, one can call `ReSize(n,m)` initially to set the final dimensions. In the same way, `InsertArc(u,v)` generates a new incidence with random or constant capacities. This depends on the configuration flags `randLength`, `randUCap`, `randLCap`.

Dense graph objects also admit an operation `InsertArc(a,uu,ll,cc)`. Actually, such an operation does not generate a new incidence but increases the lower bound of an existing arc a by an amount of ll, and the upper bound by an amount of uu. The new length label overwrites the old one.

The other operations which apply to sparse graph objects only, have been described in Section 6.2.3. Note that node insertions maintain colours, distance labels and node potentials but destroy node partition data structure. Arc and node canceling operations do not influence the potential solutions, but node deletions effectively destroy all potential solutions.

None of the listed methods does apply if another data object references the graph which has to be manipulated.

11.3.2 Invalidation Policy

When the incidence structure of a graph is modified, the following internal data structures are **invalidated**, that is, they do not apply to the modified graph object:

- Iterators
- Potential solutions
- Dual incidences
- Node ajacencies

There is no exhaustive code that keeps these data structures up to date. It is only guaranteed that invalid data structures are deleted transparently. In the case of node adjacencies, node degrees and graph duality data, this strategy is adequate since the data structure can be rebuilt implicitly.

With some exceptions (required in the library) potential solutions are lost irreversibly if they are invalidated.

Special care is required with iterators if maintained independently from the graph object. To be safe, generate a graph clone, manipulate this copy but iterate on the original graph. If you want to delete nodes or arcs, apply cancel operations instead and delete the canceled items in a final step.

The following is a list of graph manipulation operations ordered by their impact on the discussed data structures:

- Arc insertions
- Node insertions
- Arc cancel and contraction operations
- Arc deletions
- Node deletions

11.3.3 Updates on the Node and Arc Labels

Synopsis:

```
class genericGraphStructure
{
    void            SetUCap(TArc,TCap);
    void            SetLCap(TArc,TCap);
    void            SetDemand(TNode,TCap);
    void            SetLength(TArc,TFloat);
    void            SetOrientation(TArc,char);
    void            SetC(TNode,bool,TFloat);

    void            SetCUCap(TCap);
    void            SetCLCap(TCap);
    void            SetCDemand(TCap);
```

```
    void            SetCLength(TFloat);
    void            SetCOrientation(char);
}
```

For physical graph objects, each of the labels discussed in Section 11.1 can be set to another value by the methods `SetUCap`, `SetLCap`, `SetDemand`, `SetLength`, `SetOrientation` and `SetC` respectively. All methods maintain the respective maximal labels in a way such that an exhaustive computation is avoided.

The methods `SetCUCap`, `SetCLCap`, `SetCDemand`, `SetCOrientation` and `SetCLength` set the current labeling to a constant and disallocate the respective data structures.

Note that the updates described usually lead to non-optimal or even infeasible solutions. Post-Optimality procedures are problem-dependent and hence cannot be supported here.

11.3.4 Merging Graphs

Synopsis:

```
class abstractMixedGraph
{
    void Merge(abstractMixedGraph&);
}
```

This method merges a specified graph object into another graph without identifying any of the graph nodes. The passed graph is not manipulated (but only a copy is generated). The addressed graph which stores the result is layouted.

Chapter 12

Graph Drawing

12.1 Preliminary Remarks

By **graph drawing**, we denote techniques to manipulate the graph node coordinates and to add some artificial points for better readability. This task is distinguished from the **graph display** process which maps a drawing to a computer screen and which assigns some textual information to the nodes and edges in the drawing. When dealing with planar graphs, drawing is also distinguished from the **embedding** phase which determines an appropriate order of the node incidences and selects an exterior face.

In case of **geometric optimization instances**, the node coordinates also define the edge lengths. Here, layout methods generally do not apply.

All produced drawings are 2-dimensional. This is frequently used for inline drawing when graph objects are derived from others.

12.1.1 Layout Models

Include file: `globals.h`, `abstractMixedGraph.h`
Synopsis:

```
enum TLayoutModel {
    LAYOUT_DEFAULT = -1,
    LAYOUT_FREESTYLE_POLYGONES = 0,
    LAYOUT_FREESTYLE_CURVES = 1,
    LAYOUT_ORTHO_SMALL = 2,
    LAYOUT_ORTHO_BIG = 3,
    LAYOUT_VISIBILITY = 4,
    LAYOUT_KANDINSKI = 5,
    LAYOUT_STRAIGHT_2DIM = 6,
    LAYOUT_LAYERED = 7,
    LAYOUT_NONE = 8
};

class goblinController
{
    void    SetDisplayParameters(TLayoutModel);
}

class abstractMixedGraph
{
    void    Layout_ConvertModel(TLayoutModel);
}
```

Layout models denote general drawing styles with precise properties allowing to improve given drawings and to convert to other laout models. All drawing methods activate the appropriate **layout model** by calling `Layout_ConvertModel()` and, recursively, `SetDisplayParameters()`.

The latter procedure effectively overwrites the various display parameters (listed in Section 14.6) with layout model dependent default values. One can customize the display style by setting some of the context parameters. Note that custom values are overwritten with the next call to `SetLayoutParameters()`.

In addition, `Layout_ConvertModel(model)` has the capability to adjust the current drawing: If the target layout model is `LAYOUT_STRAIGHT_2DIM`,

all bends and shape nodes are eliminated. Conversely, if the target model is `LAYOUT_FREESTYLE_POLYGONES` or `LAYOUT_FREESTYLE_CURVES`, the edges are redrawn to exhibit parallel edges and loops. If neither the target model nor the original model is `LAYOUT_STRAIGHT_2DIM`, by default, an intermediary conversion to the `LAYOUT_STRAIGHT_2DIM` model takes place.

Future development will bear more sophisticated conversion rules, but only for particular pairs of layout models. The procedure is not intended to perform drawing algorithms inline.

12.1.2 Grid Lines

Synopsis:

```
class goblinController
{
    int    nodeSep;
    int    bendSep;
    int    fineSep;
}
```

12.1.3 Translations of the Current Drawing

Include file: `abstractMixedGraph.h`
Synopsis:

```
class abstractMixedGraph
{
    void    Layout_StripEmbedding();
    void    Layout_ScaleEmbedding(
                TFloat,TFloat,TFloat,TFloat);
}
```

There are two methods `Layout_StripEmbedding()` and `Layout_ScaleEmbedding(x,X,y,Y)` which shift respectively scale the current drawing. The strip operation shifts the drawing to the non-negative orthant such that every coordinate becomes zero for at least one graph node.

The parameters of the scale operation specify a tight bounding box for the updated coordinates. By taking `X<x` or `Y<y`, the scale operation can be used to flip the layout along the ordinate or abscissa.

Do not confuse this functionality with the display parameters which are discussed in Section 14.6 and which change the view independently from the saved coordiantes and the geometric distance labels.

12.1.4 Automatic Alignment of Arcs

Include file: `abstractMixedGraph.h`
Synopsis:

```
class abstractMixedGraph
{
    void    Layout_ArcAlignment(TFloat = 0);
}
```

The method `Layout_ArcAlignment(d)` eliminates all arc alignment points from the present layout and then redraws the arcs where simple graphs are drawn with straight lines. Parallel arcs are separated by using the parameter `d`. If no value is specified, the context variable `bendSep` comes into play. Loops which are usually invisible (unless interpolation points are associated with them) are also drawn. All of the layout methods which are discussed next perform this arc alignment procedure as a final step.

12.2 Circular Layout

Include file: `abstractMixedGraph.h`
Synopsis:

```
class abstractMixedGraph
{
    void    Layout_Circular(int = 0);
    void    Layout_CircularByPredecessors(int = 0);
    void    Layout_CircularByColours(int = 0);
    bool    Layout_Outerplanar(int = 0);
}
```

The method `Layout_Circular(spacing)` draws the graph nodes as a regular polyhedron in the x-y plane. If `spacing` is specified, this overwrites the context parameter `nodeSep`. There are some variants of the method which differ by the order of the nodes on the resulting circle:

- `Layout_CircularByPredecessors()`: If predecessor arcs are available, the method starts at some node, tracks back the predecessor arcs and consecutively places the nodes until a node is reached with is already placed. Then, a new thread of search is started. In particular, the procedure exhibits Hamiltonian cycles. If no predecessor labels are available, nodes are placed by their indices.

- `Layout_CircularByColours()`: The nodes are displayed by their colour index. By that, colour clusters and special node orderings can be exhibited. If no predecessor labels are available, nodes are placed by their indices.

- `Layout_Outerplanar()`: This checks if an outerplanar embedding is available and, occasionally, exhibits this embedding. If `methLocal` is set, the planar FDP method is called to improve the circular drawing.

When calling `Layout_Circular()`, it applies on of the described methods (in that order of preference).

12.3 Tree Layout

Include file: `abstractMixedGraph.h`
Synopsis:

```
class abstractMixedGraph
{
    enum    TOptAlign {
                ALIGN_LEFT,
                ALIGN_CENTER,
                ALIGN_RIGHT
            };

    void    Layout_PredecessorTree(
                TOptAlign = ALIGN_CENTER,
                TFloat = 0,TFloat = 0);
}
```

The method `Layout_PredecessorTree(method,dx,dy)` determines a x-y plane layout based on the predecessor labels. The predecessor labels must exist and form a forest of rooted trees. Connectivity is not required. The parameters `dx` and `dy` hereby denote the horizontal respectively the vertical node distance. If no values are specified, the context variable `nodeSep` comes into play.

12.4 Force Directed Placement

Synopsis:

```
class abstractMixedGraph
{
    enum    TOptFDP {
```

```
            FDP_DEFAULT = -1,
            FDP_SPRING = 0,
            FDP_GEM = 1,
            FDP_PLANAR = 2
        };

    void    Layout_ForceDirected(
            TOptFDP = FDP_DEFAULT,int = 0);
    void    Layout_PlanarFDP(int = 0);

    void    Layout_SpringEmbedder(TFloat = 0,TFloat = 0);
    void    Layout_GEMDrawing(TOptFDP = FDP_GEM,int = 0);
}
```

The method `Layout_ForceDirected(method,spacing)` is the interface to a couple of layout methods which all are **force directed**. These kind of methods apply to general graph objects and can help to exhibit graph symmetries. If `method` does not specified otherwise, the applied method is determined by the context parameter `methFDP`. The role of the parameter `spacing` is the same as for the circular layout method.

The method `Layout_SpringEmbedder(gamma,delta)` models the graph nodes as loaded particles and the graph arcs as springs in the x-y plane. The parameters work as constants for the respective forces. Starting with the present embedding, a Newton iteration scheme searches for equilibriance of the modelled forces. The main disadvantage of this algorithm is its poor performance. The input graph should be connected for otherwise the connected components diverge.

For practical purposes, it is recommended to apply `Layout_GEMDrawing()` instead of the classic spring embedder. This algorithm moves only one node at a time. In addition to the forces discussed before, attraction to the center of gravity is modelled. The step length is determined by the nodes' temperatures and the sophisticated temperature adjustment rule is the reason for the good performance. The resulting drawings do not differ significantly compared with the spring embedder.

There is another method `Layout_PlanarFDP()` which preserves the edge crossing properties of the initial layout. It applies to general graphs but, of course, is intended to allow post processing of planar layouts. The procedure augments the GEM algorithm by additional forces and certain restrictions. It is less performant than the unrestricted GEM code.

12.5 Planar Straight Line Drawing

Synopsis:

```
class abstractMixedGraph
{
    void    Layout_StraightLineDrawing(
            TArc = NoArc,int = 0);
    void    Layout_ConvexDrawing(
            TArc = NoArc,int = 0);
}
```

The method `Layout_ConvexDrawing(a,spacing)` computes a straight line grid drawing without edge crossings for triconnected planar graphs such that all interior faces are convex. A combinatorial embedding must be already assigned.

The shape of drawing is a triangle with the specified arc a forming the basis. By that, the exterior face is set to the left hand side of a. If no arc is specified, the basis arc is the same as `ExteriorArc()`. If the latter arc is undefined, the exterior arc is chosen on a face which maximizes the number of exterior nodes. The tip node is set implicitly and is "half way" on the exterior face. All nodes are placed at integer coordinates, depending on `spacing` or the context parameter `nodeSep`. The time complexity is $O(m)$.

The procedure uses the canonically ordered partition which is discussed in the next chapter. This structure and the convexity of the interior faces require that the input graph is triconnected.

When calling `Layout_StraightLineDrawing(a,spacing)`, the graph is triangulated and to the triangulation, the convex drawing method is applied. The time complexity is $O(n^2)$. If `methLocal` is set, the planar FDP method is called to improve the drawing of the original graph. This post-processing is necessary at least if a lot of artificial edges have been added. See Section 13.7 for more details about triangulations.

12.6 Orthogonal Drawing

Include file: `abstractMixedGraph.h`
Synopsis:

```
class abstractMixedGraph
{
    enum    TOptOrthogonal {
                ORTHO_DEFAULT = -1,
                ORTHO_EXPLICIT = 0,
                ORTHO_EULER = 1,
                ORTHO_DEG4 = 2,
                ORTHO_4PLANAR = 3,
                ORTHO_VISIBILITY = 4,
                ORTHO_VISIBILITY_TRIM = 5,
                ORTHO_VISIBILITY_GIOTTO = 6
            };

    void    Layout_Orthogonal1Bent(
                TOptOrthogonal = ORTHO_DEFAULT,int = 0);
    void    Layout_OrthogonalDeg4(
                TOptOrthogonal = ORTHO_4PLANAR,int = 0);
    void    Layout_VisibilityRepresentation(
                TOptOrthogonal = ORTHO_VISIBILITY_TRIM,
                int = 0);
}
```

In an orthogonal drawing, every edge is represented by an alternating sequence of horizontal and vertical line segments. If the nodes are drawn without dimension, this approach is restricted to graphs with maximum degree 4 or less. The literature comes up with at least four layout models for drawing high degree graphs:

- **GIOTTO**, where the nodes are rectangles in a square grid. Edges are drawn by placing ports at each end node and a sequence of bend nodes within the same grid as the rectangles. No bound on the size of the rectangles is imposed by this model.

- **Kandinski**, where the nodes are squares of a common size centered in a sparse square grid. Edge ports and bend nodes are placed in a subdivided square grid. The size of the node squares is the maximum number of ports d assigned to one side of a square.

- **Proportional growth**, which is similar to GIOTTO but also requires that the height of a node equals the number of ports on either the left or the right side, and that the width equals the number of ports on the top or the bottom line.

- **Visibility representations**, in which nodes are horizontal and edges are vertical line segments. The length of node segments does not depend on the node degrees.

In every model, the edges must be drawn on the grid lines. Edges and node representation may not overlap or cross each other, but edges may cross other edges when planarity is not required. The known algorithms for graphs with maximum degree 4 produce drawings which fit into the first three models. General methods, applied to degree-4 graphs, are not as smart. That is, large nodes with several ports on one side of a node may result.

GOBLIN provides Kandinski drawings of general graphs, visibility representations and GIOTTO drawings for planar graphs and drawings with

small nodes for 2-connected graphs with maximum node degree 4. Post-processing techniques for the GIOTTO model are desirable but are not available yet.

By calling `Layout_Orthogonal1Bent(opt,grid)`, the following steps are performed in order to obtain a drawing in the Kandinski model:

- The nodes are placed in **general position**, that is with only node in one column or row. This placement preserves the order of coordinate values in the preceding drawing.
- The edges are distributed to the four sides of each node. The node size is also computed. This procedure depends on the `opt` parameter.
- A couple of context parameters are set according to the Kandinski layout model and the grids used for this drawing.
- The edges on each side of a node are ordered so that the drawing does not include crossings of adjacent arcs.
- Every arc is drawn with exactly one bend node and with the arc label in the neighbourhood of this bend node.

If the graph layout and incidence structure is not changed intermediately, calling this layout tool several times results in the same drawing. But the procedure supports some pre- and postprocessing techniques:

- The preceding node placement: It is useful to start with a readable drawing, not necessarily in an orthogonal layout model. One can move nodes manually and rerun the method to improve the node placement.
- When using `opt = ORTHO_EXPLICIT`, directed arcs leave on the top or the bottom side of the start node and enter the end node on the left or the right side. The same is true for the inherent orientation of undirected edges. It follows that one can revert the the orientations of undirected edges in order to reduce the number of edge crossings and the node size d.

The running time is $O(m)$, the number of bends is m and the square area is $(2n(d+1)-1)^2$. The parameter d is trivially bounded by the maximum node degree Δ and for `opt = ORTHO_EULER`, one has $d \leq \Delta/2$.

By calling `Layout_OrthogonalDeg4(opt,grid)`, an st-numbering is computed and the nodes are placed one-by-one with respect to this ordering, each node on a new grid row. Columns may carry several graph nodes and at most two bends. The input graph must be 2-connected and without loops.

If the graph is planar and if `opt = ORTHO_4PLANAR` is used, a plane drawing results and the inherent embedding is visualized. In this case, the running time is $O(n)$. Otherwise, the running time is $O(n^2)$ due to the iterated computation of horizontal coordinates. The achieved grid size is $(m-n+1)n$ in the worst case. Every edge is drawn with at most two bends. Only one edge incident with the final node may be drawn with 3 bends.

By calling `Layout_VisibilityRepresentation(opt,grid)`, the input graph is augmented to a 2-connected planar graph. Then, a bipolar orientation and a directed dual graph are generated. For both the primal and the dual digraph, the distance labels with respect to the source nodes are computed. The primal distances give assigment of rows to the nodes in the drawing, and the dual distances determine the columns for the drawing of arcs. The method option applies as follows:

- Using `opt = ORTHO_VISIBILITY_RAW`, the drawing is as usually described in the literature. That is, the node width may exceed the size required by the vertical edge segments.
- Using `opt = ORTHO_VISIBILITY_TRIM`, the nodes are shrunk in a way that edges are still drawn vertically without bends.
- Using `opt = ORTHO_VISIBILITY_GIOTTO`, the node widths are minimized and every edge is drawn with at most two bends. If the input graph is 2-connected and all nodes have degree 2 or 3, the final drawing is in the small node model. Otherwise, a GIOTTO drawing results.

In any circumstances, the input graph must be without loops. The running time is $O(n)$ and the achieved grid size is $(2n-5)(n-1)$ in the worst case.

12.7 Equilateral Drawing

Include file: `abstractMixedGraph.h`
Synopsis:

```
class abstractMixedGraph
{
    void    Layout_Equilateral(int = 0);
}
```

This method requires an outerplanar, 2-connected graph. All faces other than the exterior region are drawn as equilateral polygones. The constant edge length is either passed explicitly or given by the context parameter `nodeSep`.

The procedure is intended for drawing regular polyhedra, including Platonic solids, after unrolling the surface to plane.

Chapter 13

High Level Algorithms

This chapter shows how GOBLIN problem solvers are called and configured. It also includes a brief introduction to the respective problem settings, and to the general ideas behind the algorithms implemented. With some text book about graph theory at hand and using the GOBLET browser, any interested people should be able to understand the source code.

For the most problems, the implemented algorithms do not reflect the current state of research. But the library covers all of the standard problems and careful implementations of all of the algorithms which one can find in text books about graph optimization. The non-weighted matching code which was the original authors research interest clearly stands out.

Up to the max-cut and the Steiner tree codes which can only be viewed as an interface for future implementations, all solvers are configured to solve practical problem instances. This ranges from several 10000 node instances for shortest path, min-tree, max-flow and non-weighted matching problems, a few thousand node problems for min-cost flow, arborescence and weighted matching problems to 150 node instances for the exact solution of NP-hard problems (we have restricted ourselves to pure combinatorial methods). Note that there was considerable effort to provide codes which support post optimization and which apply to the most general problem formulations possible.

13.1 Shortest Paths

Synopsis:

```
class abstractMixedGraph
{
    enum    TOptSPX {
                SPX_DEFAULT = -1,
                SPX_FIFO = 0,
                SPX_DIJKSTRA = 1,
                SPX_BELLMAN = 2,
                SPX_BFS = 3,
                SPX_DAG = 4,
                SPX_TJOIN = 5
            };

    enum    TOptSPXChar {
                SPX_PLAIN = 0,
                SPX_SUBGRAPH = 1,
                SPX_RESIDUAL = 2,
                SPX_REDUCED = 4,
                SPX_REDUCED_RESIDUAL = 6
            };

    bool    ShortestPath(TNode s,TNode t = NoNode);
    bool    ShortestPath(TOptSPXMeth,TOptSPXChar,
                TNode,TNode = NoNode);

    bool    Eligible(TOptSearch,TArc);

    bool    BFS(TOptSearch,TNode,TNode = NoNode);
    TNode   SPX_Dijkstra(TOptSPXChar,
                const indexSet<TNode>&,
                const indexSet<TNode>&);
```

```
    bool    SPX_FIFOLabelCorrecting(TOptSPXChar,
                TNode,TNode = NoNode);
    bool    SPX_BellmanFord(TOptSPXChar,
                TNode,TNode = NoNode);

    TNode   VoronoiRegions();
}

class abstractGraph
{
    bool    SPX_TJoin(TNode,TNode);
}
```

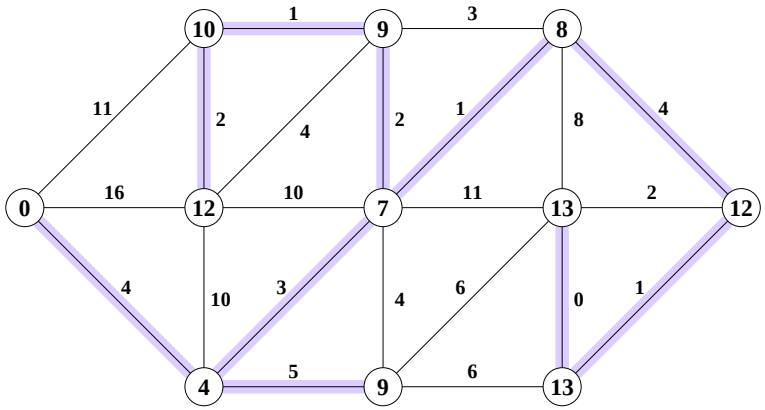


Figure 13.1: A Shortest Path Tree

13.1.1 Eligible Arcs

The method `Eligible()` qualifies the arcs which may appear on a shortest path (tree). Basically, it guides shortest path algorithms to compute directed paths for digraph objects, and arbitrary paths in undirected graphs. If the `SPX_RESIDUAL` option is used, paths with residual capacity are determined. If the `SPX_SUBGRAPH` option is used, the subgraph defined by the arcs a with `Subgraph(a)>0` is searched.

An **eligible** st-path is a simple path starting at node s and ending at node t which entirely consists of eligible arcs. A **shortest** st-path is an eligible st-path of minimum length. A **shortest path tree** is a tree such that any path from the root to another node is a shortest path.

13.1.2 Solver Interface

The solver is called like `ShortestPath(method,characteristic,s,t)`. There is a shortcut `ShortestPath(s,t)` which applies the options `method = SPX_DEFAULT` and `characteristic = SPX_PLAIN`. A `method = SPX_DEFAULT` value is eventually replaced by the value of the context variable `methSearch`.

The parameter s denotes the root of the requested shortest path tree. The parameter t is optional and denotes a node to be reached. If no t is specified, the member `targetNode` is used. If this is also undefined, a full shortest path tree is determined. If t is specified and reachable from s by eligible arcs, the computation may stop prematurely for sake of performance. The return value is `true` if t is s-reachable, and `false` otherwise. Shortest paths are returned by the predecessor labels together with the matching distance labels.

The `characteristic` parameter switches between applying the original

length labels and the reduced length labels (when using `SPX_REDUCED` or `SPX_REDUCED_RESIDUAL`).

A shortest path tree with respect to the reduced length labels is also a shortest path tree for the original length labels. The length of cycles does not depend on the `characteristic` option and the concrete node potentials. But searching the reduced length labels is useful since those labels can be usually kept non-negative, and this is required by some shortest path methods.

In what follows, the particular shortest path algorithms are described. Most methods can be accessed by the hub `ShortestPath()`, but this does mean that the codes are interchangeable: Each algorithms applies to a special (and often restrictive) setting. The label correcting algorithms are the most general ones, but even those cannot handle the computation of a shortest path when the graph admits negative length cycles and, in particular, when undirected edges of negative length exist.

In any situation where the selected method does not apply, an `ERRejected` exception is raised.

13.1.3 Breadth First Search

This method computes an eligible sv-path with a minimum number of arcs for every node v of the graph. So this procedure solves the shortest path problem for graph with constant non-negative edge lengths. The running time is $O(m)$. When calling `BFS()` directly, the length labels are ignored. When calling this method by using `ShortestPath()`, and if the length labels are not constant or negative, an exception is raised.

13.1.4 The Dijkstra Algorithm

This is a multi-terminal version of the Dijkstra method. One index set is passed to specify the root nodes, the second index set specifies the target nodes. The return value is some target node which could be reached from any root node.

The Dijkstra method cannot handle negative length arcs at all.It will, however, complain only if a negative length arc is actually searched.

The algorithm utilizes a priority queue data structure. This may either be passed by the member pointer variable `nHeap` or, if `nHeap==NULL`, generated by the method itself. In the latter case, the type of the priority queue is chosen according to the context parameter `methPQ`. The respective running times are $O(n^2)$, $O(m \log n)$ with the binary heaps and $O(m + n \log n)$ with the Fibonacci heaps.

If one needs to compute all-pair shortest paths, it is reasonable to apply a label-correcting method (but only if negative lengths can occur) and then to compute a shortest path tree for each graph node with the Dijkstra method. This requires $O(n(m + n \log n))$ time and only $O(m)$ storage compared with $O(n^2)$ for the Floyd-Warshall code.

13.1.5 Discrete Voronoi Regions

This is a variation of the Dijkstra method which treats all graph nodes with `Demand()` different from zero a root nodes. The set of target nodes is empty and, by that, all full graph search is performed. If every connected component includes at least one terminal node, the procedure returns with partial trees which connect every node to some terminal, and with corresponding distance labels.

The **Voronoi regions** are also returned by the node partition data structure. The formal return value is the number of terminals and the running times are essentially the same as for `SPX_Dijkstra()`.

13.1.6 The Bellman-Ford Algorithm

This method determines a shortest sv-path for every node v of the graph. Negative length labels are allowed. If a negative length cycle is detected, the procedure returns an exception `ERCheck`. The running time is $O(mn)$.

13.1.7 The FIFO Label-Correcting Algorithm

This method determines a shortest sv-path for every node v of the graph. Negative length labels are allowed. If a negative length cycle is detected, the procedure returns an exception `ERCheck`. The running time is $O(mn)$. This algorithm is a practical improvement of the Bellman-Ford procedure.

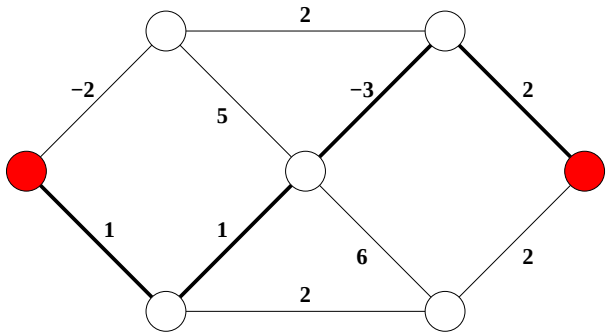


Figure 13.2: A T-Join Shortest Path

13.1.8 The T-Join Algorithm

The method `SPX_TJoin()` differs from the previous shortest path algorithms in several ways: First of all, it does not compute a shortest path tree but only a single simple path. Furthermore, it applies to undirected graphs only.

In any circumsatnces, the method returns an optimal T-join by the subgraph labels, and this subgraph splits into an st-path and a couple of (not necessarily node-disjoint) cycles. The following situations may arise:

- The T-join is a simple st-path. Then this path is a shortest st-path and is returned by the predecessor labels.
- There is no T-join. Then also no st-path exists.
- The T-join splits into a simple st-path plus some more connected components. Then this st-path is returned by the predecessor labels, but this is not necessarily a shortest path.
- In the T-join, s and t are in a common non-trivial component. From this component, an arbitrary st-path is extracted.

That is, the method can deal with negative length arcs, at least, if there are no negative length cycles. When it is not possible to compute an exact solution for the shortest path problem from the optimal T-join, at least, a heuristic solution is returned.

The method consists of $O(n)$ calls of the Dijkstra method and the solution of a weighted 1-matching problem with at most n nodes (the problem size depends on the number negative length edges). This matching problem dominates the running times.

13.1.9 The Floyd-Warshall Algorithm

This method determines the distances between every pair of nodes in $O(n^3)$ time. It is encoded into the constructor of the class `distanceGraph` and, by that, generates a complete digraph which has the original node distances as its length labels.

13.1.10 Proposed Extension

A dequeue label-correcting algorithm.

13.2 Negative Cycles

Synopsis:

```
class abstractMixedGraph
{
```

```
    TNode   NegativeCycle(TNode = NoNode);
}

class abstractDiGraph
{
protected:
    TCap    mu;

public:
    TNode   MinimumMeanCycle();
}
```

Primal min-cost flow algorithm traditionally try to find an augmenting cycle of negative length. GOBLIN supplies two methods which are useful in this context. Both methods apply for digraphs only.

13.2.1 Negative Cycles

The method `NegativeCycle()` determines an arbitrary eligible cycle with negative length. The return value is a node on this cycle. The cycle itself is returned via the predecessor labels. If an optional node is passed, this node is considered to be the root of a graph search, and essentially the FIFO label correcting algorithm results.

13.2.2 Minimum Mean Cycles

The method `MinimumMeanCycle()` determines an eligible cycle such that the ratio of the sum of length labels and the number of arcs on this cycle is minimum. Again, the return value is a node on the cycle, and the cycle itself is returned via the predecessor labels. The minimum ratio is kept by the variable `mu` for further processing. The method also works if this ratio is negative.

An important drawback of the algorithm is that it requires $\Theta(n^2)$ storage units which makes it inapplicable to large-scale problems, say with $n > 10^5$.

13.2.3 Proposed Extension

A mean cycle algorithm which runs with linear storage requirements.

13.3 DAG Search

Synopsis:

```
class abstractDigraph
{
    enum    TDAGSearch {
                DAG_TOPSORT,
                DAG_CRITICAL,
                DAG_SPTREE
            };

    TNode   DAGSearch(TDAGSearch,
                TNode=NoNode,TNode=NoNode);
    TNode   TopSort();
    TNode   CriticalPath();
}
```

A **DAG** is a directed acyclic graph object. The procedure `DAGSearch(opt)` handles the recognition of DAGs and does some additional computations depending on the value of `opt`:

- For `DAG_TOPSORT` and `DAG_SPTREE`, a topological oerdering is exported by the node colour data structure. If the graph contains directed cycles, a node on a cycle is returned.
- For `DAG_SPTREE`, a shortest path tree and the distance labels are exported.
- For `DAG_CRITICAL`, a directed path of maximum length is computed and its end node is returned (unless cycles are found). For every

node, the distance label denotes the maximum path lengths from a root node.

The methods `CriticalPath()` and `TopSort()` are shortcuts which should be used as entry points. The shortest path version is handled by `ShortestPath()`. The running time of a DAG search is $O(m)$ in every instance. Note that eligible (residual capacity) arcs are searched instead of the original directions.

13.4 Euler Cycles

Synopsis:

```
class abstractMixedGraph
{
    bool        EulerCycle(TArc*);
    bool        EulerCycle();
}
```

An Euler cycle is a closed walk which traverses all graph edges exactly once. It can be computed by the call `EulerCycle(pred)` in $O(m)$ time. This method implements the Hierholzer algorithm and returns false if no Euler cycle exists. Otherwise an Euler cycle is returned by the referenced array `pred` which must be allocated by the calling context as `TArc[M()]`. The Euler cycle is decoded from the array as follows:

Example:

```
TArc* pred = new TArc[M()];

if (!EulerCycle(pred)) {/* Handle exception */};

TArc a = pred[0];
for (TArc i=0;i<=M();i++)
{
    a = pred[a>>1];
    // Process the arc a
}
```

If one calls `EulerCycle()` without parameters, the Euler cycle is translated to an edge numbering and saved to the edge colours.

Note that the procedure does not inspect the arc capacities. If capacities are considered as multiplicities (as in the Chinese postman solver), the graph must be preprocessed with `ExplicitParallels()` to eliminate the capacities. Zero capacity arc must be eliminated manually. Be aware of the problem size and the running time which grows linearly with the sum of multiplicities!

13.5 Spanning Trees

Synopsis:

```
class abstractMixedGraph
{
    enum        TOptMSTMeth {
                    MST_DEFAULT = -1,
                    MST_PRIM = 0,
                    MST_PRIM2 = 1,
                    MST_KRUSKAL = 2,
                    MST_EDMONDS = 3
                };

    enum        TOptMSTChar {
                    MST_PLAIN = 0,
                    MST_ONE_CYCLE = 1,
                    MST_REDUCED = 8,
                    MST_MAX = 16
```

```
                };

    TFloat MinTree(TNode r = NoNode);
    TFloat MinTree(TOptMSTMeth,
                TOptMSTChar,TNode = NoNode);

    TFloat MST_Prim(TOptMSTMeth,
                TOptMSTChar,TNode = NoNode);
    TFloat MST_Edmonds(TOptMSTChar,TNode = NoNode);
    TFloat MST_Kruskal(TOptMSTChar,TNode = NoNode);
}

class abstractDiGraph
{
    TCap                TreePacking(TNode);
    abstractDiGraph*    TreePKGInit();
    TCap                TreePKGStripTree(
                            abstractDiGraph*,TCap,TNode);
}
```

A spanning tree of a graph is a subgraph which connects all nodes but does not contain cycles. There is no reason to distinguish between a minimization and a maximization problem for spanning trees. We follow the convention to formulate the minimization problem, at least as the default setting.

The spanning tree solver is called like `MinTree(method,characteristic,r)`. The shortcut `MinTree(r)` applies `method = MST_DEFAULT` and `characteristic = MST_PLAIN`. A `method = MST_DEFAULT` value is eventually replaced by the value of the context variable `methMinTree`. The parameter r is optional and denotes the root node. If r is not specified, the member `rootNode` is used.

Other than in earlier releases, the `MinTree()` interface function is used for both the directed and the undirected setting. Accordingly, the Edmonds arborescence method can run on undirected graphs (with a similar behavior as the Kruskal method, at least when the edge lengths are mutually distinct) and the Prim and the Kruskal method apply to mixed and to directed graphs (ignoring the edge orientation).

There is a couple of options which can be passed with `characteristic` and which all methods allow for:

- `MST_MAX`: Changes the object sense to maximization
- `MST_REDUCED`: Consider the reduced length labels instead of the original length labels
- `MST_ONE_CYCLE`: Construct a one cycle tree instead of a spanning tree

The Kruskal method, the enhanced Prim method and the method for mixed graphs can handle arcs with non-trivial capacity lower bounds which denote mandatory arcs. This mechanism is applied in the branch and bound module of the TSP solver.

13.5.1 The (Enhanced) Prim Algorithm

This procedure grows the spanning tree from the root node to the tips. If no root node is passed, one is selected automatically. The resulting tree is returned by the predecessor labels.

Two versions of this method are provided. The running time of the basic version is $O(n^2)$. The enhanced code differs from the basic version mainly by using a priority queue for node selection. The complexity depends on the special choice of this queue, and matches the running times for the Dijkstra algorithm (see Section 13.1.4).

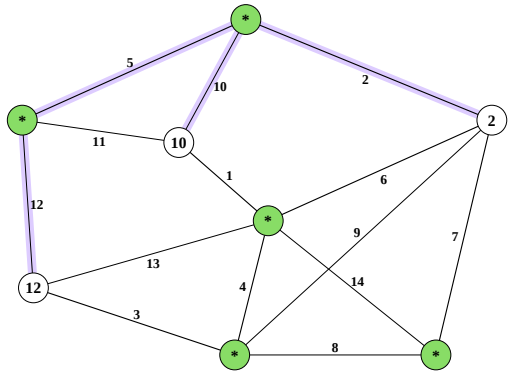


Figure 13.3: Intermediate Step in the Prim Method

13.5.2 The Kruskal Algorithm

The procedure returns a spanning tree via the subgraph data structure if one exists. If not, the connected components are maintained by the partition data structure. The running time is $O(m \log n)$ due to the needed sorting of the edge by their lengths.

13.5.3 Arborescences

The method `MST_Edmonds()` is an $O(nm)$ implementation of Edmond's arborescence algorithm and determines a maximum spanning forest. If a root node is passed, and this is actually a root node, an arborescence is returned by the predecessor labels. If no root node is passed, one is selected automatically. The procedure uses the shrinking family data structure (see Section 8.2.2).

13.5.4 One Cycle Trees

All spanning tree methods described above can be used to compute an r-**tree**. In the undirected setting, this is a minimum spanning tree of the nodes other than r plus the two shortest edges incident with r. In the directed setting, an r-tree is an minimum arborescence rooted at r plus the shortest arc entering r.

When the spanning tree solver is called with the `MST_ONE_CYCLE` and a root node r, it will determine predecessor labels such that r is on the unique directed cycle defined by these labels. For every node v not on this cycle, there is a unique directed path of predecessor arcs connecting the cycle and v.

The worst case complexities are the same as for constructing ordinary spanning trees.

13.5.5 Tree Packings

The method `TreePacking(TNode)` determines a maximum cardinality set of pairwise disjoint arborescences rooted at a specified node. If the arc capacities are non-trivial, the algorithm computes some tree capacities and for every arc the sum of the tree capacities satisfies the capacity bound.

If the main procedure `TreePacking()` is called, the tree capacities are provided by the log file and, if `traceLevel>2`, the found arborescences are written to trace files. But one can also call the component `TreePKGStrip()` which manipulates a copy of the digraph and one-by-one returns the arborescences via the predecessor labels of the original graph and also returns the corresponding tree capacity. The graph copy is generated by the method `TreePKGInit()`. The complete method looks like this:

Example:

```
TCap totalMultiplicity = StrongEdgeConnectivity(r);
abstractDiGraph* G = TreePKGInit();
while (totalMultiplicity>0)
{
```

```
    TreePKGStrip(G,&totalMultiplicity,r);
    // Use the predecessor labels
}
delete G;
```

The tree packing method is non-polynomial, also slow in practice and applies to directed graphs only. Note that the application to the complete orientation does not yield a tree packing in the undirected sense.

13.5.6 Proposed Extension

A cycle canceling method for post-optimization of spanning trees.

13.6 Connected Components

Synopsis:

```
class abstractMixedGraph
{
    bool    Connected();
    bool    StronglyConnected();
    bool    CutNodes(TNode* = NULL,TArc* = NULL,
                TNode* = NULL,TArc = NoArc);
    bool    TwoConnected();
    bool    STNumbering(TArc = NoArc);
}
```

13.6.1 First Order Connectivity

The DFS method `Connected` decides whether a graph is connected or not, and returns 1 or 0 respectively. In the latter case, the connected components are available by the node colour data structure. The running time is $O(m)$.

Note that the Kruskal algorithm also applies to this problem but returns the connected components by the node partition data structure.

13.6.2 Strong Connectivity

Two nodes x, y of a graph are **strongly connected** if there are an eligible xy-path and an eligible yx-path. A **strong component** is a maximal node set such that each pair of nodes is strongly connected.

The method `StronglyConnected` decides whether a graph is strongly connected or not, and returns 1 or 0 respectively. In the latter case, the strong components are available by the node colour data structure. The running time is $O(m)$.

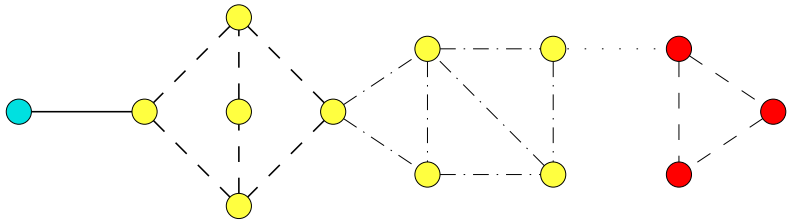


Figure 13.4: Blocks and 2-Edge Connectivity

13.6.3 Second Order Connectivity

A **cut node** [**bridge**] of a graph is a node [arc] whose deletion increases the number of connected components. A **block** is a maximal node set such that each pair of nodes is traversed by a simple cycle. A **2-edge connected component** is a maximal node set such that the induced subgraph contains no bridges.

The method `CutNodes()` checks if the graph has a cut node. In that case, the cut nodes are returned by the node colours with colour 0. The remaining nodes are coloured block by block. For this direct 2-connectivity application, no parameters are passed to `CutNodes()`.

The method `TwoConnected()` checks if the graph is 2-edge connected. It calls `CutNodes()` with some extra data structures and returns a set of subgraph labels which represent the blocks and a colouring which represents the 2-edge connectivity components. To store the blocks, the arc capacities have to be infinite. For both procedures, the running time is $O(m)$.

13.6.4 Open Ear Decomposition and st-Numbering

An **ear decomposition** partitions the edge set into simple paths $P_1, P_2, \ldots, P_k$ such that both end nodes of P_{i+1}, $i > 1$ occur on $P_1, P_2, \ldots, P_i$ but the intermediate nodes do not. Ear decompositions are stored as edge colours. If no cycles occur among $P_1, P_2, \ldots, P_k$, the ear decomposition is called an **open ear decomposition**.

For given nodes s and t, an st**-numbering** is an assignment to the node colours with `colour[s] = 0`, `colour[t] = n-1` and such that for every other node v, two neighbors u and w with

```
colour[u] < colour[v] < colour[w]
```

exist. If s and t denote the end nodes of the path P_1 in an open ear decomposition, one can obtain an st-numbering by inserting the nodes of each path in the order of appearance, right after the start node of the path.

The method `STNumbering()` computes both data structures simultaneously from a DFS tree which is again obtained from `CutNodes()`. It is not possible to choose the nodes s and t independent from each other. Instead, a so-called **return arc** can be passed whose end nodes are t and s then, in that order! The input graph must be 2-connected, the running time is $O(m)$.

13.7 Planarity

Synopsis:

```
class abstractMixedGraph
{
    bool    IsPlanar();
    bool    PlanarityDMP64(TArc*);
    TNode   LMCOrderedPartition(TArc*,TArc*,TNode*)
    void    GrowExteriorFace();
    void    PlaneConnectivityAugmentation();
    void    PlaneBiconnectivityAugmentation();
    void    Triangulation();
}

class sparseGraphStructure
{
    void    Planarize(TArc*);
}
```

A **planar graph** is a graph which can be drawn in the plane without any edge crossings. Many optimization problems admit special solvers for planar graphs which perform much better than the general codes (see the max-cut section for an example). Usually, it is not necessary to know an explicit drawing.

The method `IsPlanar()` is the general entry point for planarity tests. It checks if the graph is planar but does not export an embedding. The pure planarity test is implemented for general graph objects. Explicit planarization is restricted to sparse graph objects which are stored by incidence lists.

13.7.1 The Method of Demoucron, Malgrange and Pertuiset

The implemented planarity test `PlanarityDMP64()` first adds some arcs to obtain a 2-connected graph. Then an initial cycle and two regions are generated. The remaining graph arcs are partitioned into **segments**. In the main loop of the algorithm, a segment is determined which can be embedded into a minimum number of regions. From this segment, an arbitrary path is embedded into some feasible region and this region is split. All loops

and parallel arcs are embedded in a post processing step to prevent from computational overhead. The running time of this method is $O(m^3)$ and, by that, $O(n^3)$ for planar graphs.

Currently, the required storage can be bounded only by $O(nm)$. If the input graph is non-planar, no forbidden configuration is exported yet.

13.7.2 Combinatorial Embedding

For sparse graph objects, a method `Planarize()` exists which calls the planarity test and then exports a **combinatorial embedding** to the node incidences: To this end, the method `Planarize()` is called which actually orders the incidence lists. It takes an array which specifies the predecessor of each arc when traversing the regions. This array must be filled by any prospective planarization method.

13.7.3 Outerplanar Embedding

An **outerplanar graph** is a graph which can be drawn in the plane without edge crossings and such that all nodes are incident with the unbounded region to which we refer as the **exterior face**.

The method `GrowExteriorFace()` requires a combinatorial embedding and selects from this embedding a region with the maximum number of adjacent nodes. Then, all exterior **cut edges** (whose end nodes form cutting pairs and which are no bridges) swapped to the interior. By that, the number of external nodes strictly increases. The running time is bounded by $O(n^2)$.

If the input graph is (implicitly) outerplanar, a respective combinatorial embedding results. But even in the case of general planar graphs, more appealing layouts can be achieved with this procedure. Furthermore, one can determine st-numberings with both v_1v_2 and v_1v_n on the exterior face of the refined embedding.

The procedure also applies if the input graph is disconnected. It is only a wrapper around the call `ExtractEmbedding(PLANEXT_GROW)`. See section 6.2.5 for more details.

13.7.4 Connectivity Augmentation

Most planar graph drawing algorithms require a certain level of connectivity of the input graph. One can link the connected components of a planar embedded graph arbitrarily without destroying the combinatorial embedding. Even more, if the connected components are linked tree like, this gives a minimal connected planar supergraph in linear time. The only advanced feature of the procedure `PlaneConnectivityAugmentation()` is that it selects a maximal exterior face of each component. This procedure is a wrapper around the call `ExtractEmbedding(PLANEXT_CONNECT)` and runs in linear time.

Things are more complicated with the biconnectivity and triconnectivity augmentation problems. One can compute in polynomial time minimal biconnected and triconnected supergraphs but these are, in general, not planar. The planar versions of these problems are NP-hard.

GOBLIN includes a procedure `PlaneBiconnectivityAugmentation()` which computes a (probably not minimal) planar biconnected supergraph in linear time. The restriction to the original graph gives the original combinatorial embedding. The input graph must be connected.

There is also a procedure `Triangulation()` to compute a maximum planar supergraph of the adressed graph object. Such graphs are always triconnected and called **triangulations** since all faces form triangles. As before, the restriction to the original graph gives the original embedding. The time complexity is $O(m)$ up to the lookup of node adjacencies. The input graph must be biconnected.

All procedures modify the incidence structure of the original graph!

13.7.5 Canonically Ordered Partition

The **canonically ordered partition** splits a simple triconnected planar graph G into components which can be inserted one by one into a partial plane drawing. This structure applies to convex as well as orthogonal drawing methods. In some sense, it is the triconnectivity analogon of ear decompositions and st-numberings discussed earlier. More explicitly, the

discussed partition consists of disjoint node sets $X_1, X_2, \ldots, X_k$ such that $V(G) = \bigcup_{i=1}^{k} X_i$ and

- $X_1 = \{v_1, v_2\}$, v_1 and v_2 are neighbors on the exterior face of G and v_1v_2 is called the **basis arc**.
- The induced subgraph G_j of G to $\bigcup_{i=1}^{j} X_i$ is biconnected and **internally triconnected** for every $j = 2, \ldots, k$. That is, deleting two interior nodes preserves connectivity.
- For $j = 1, \ldots, k$, all nodes in X_j are exterior with respect to G_j.
- For $j = 2, \ldots, k-1$, the component X_j is adjacent to $\bigcup_{i=j+1}^{k} X_i$ and has at least two **contact nodes** in G_{j-1}.
- For $j = 2, \ldots, k-1$, the component X_j is either a single node or a path $v_{r_1}, v_{r_2}, \ldots, v_{r_s}$ with exactly two contact nodes, one adjacent to v_{r_1} and the other adjacent to v_{r_s}.
- $X_k = \{v_n\}$ consists of a single node.

Provided that the original graph is simple, triconnected and combinatorially embedded (not just implicitly planar), the call

```
TNode k = LMCOrderedPartition(aLeft,aRight,vRight);
```

returns the following information:

- The number k of components.
- The components by the node colours where `Colour(v)` is the index of the component of v, $i \in \{0, 1, \ldots, k-1\}$.
- The components by the array `vRight` where `vRight[v]` is the right-hand neighbor of the node v in its component or `NO_NODE` if v is right-most.
- The left-most contact arc `aLeft[i]` directed to the component i.
- The right-most contact arc `aRight[i]` directed from the component i.
- If no exterior face and basis arc have been defined in advance, both data are computed and stored internally. See Section 6.2.5 for details about the data structures.

The running time is $O(m)$. We only mention that the procedure computes the left-most canonical ordered partition and refer to the literature for the mathematical details.

13.8 Maximum Flows and Circulations

Synopsis:

```
class abstractFlowNetwork
{
protected:

    TCap            delta;

public:

    TFloat          MaxFlow(TNode,TNode);
    TFloat          EdmondsKarp(TNode,TNode);
    TFloat          CapacityScaling(TNode,TNode);
    TFloat          GoldbergTarjan(TNode,TNode);
    TFloat          Dinic(TNode,TNode);

    bool            AdmissibleCirculation();
}
```

An st-flow is a pseudo-flow such that every node $v \neq s, t$ is **balanced** (`Divergence(v)==Demand(v)`). A **maximum** st-flow is an st-flow such that `Divergence(s)`, the **flow value**, is maximal. There is a generic problem solver `MaxFlow()` which chooses one of the methods listed in Table 13.1 according to the value of the context variable `methMaxFlow`. If the context flag `methFailSave` is enabled, a reduced costs optimality criterion is checked, that is, a minimum cut is computed. All methods return the maximum flow value.

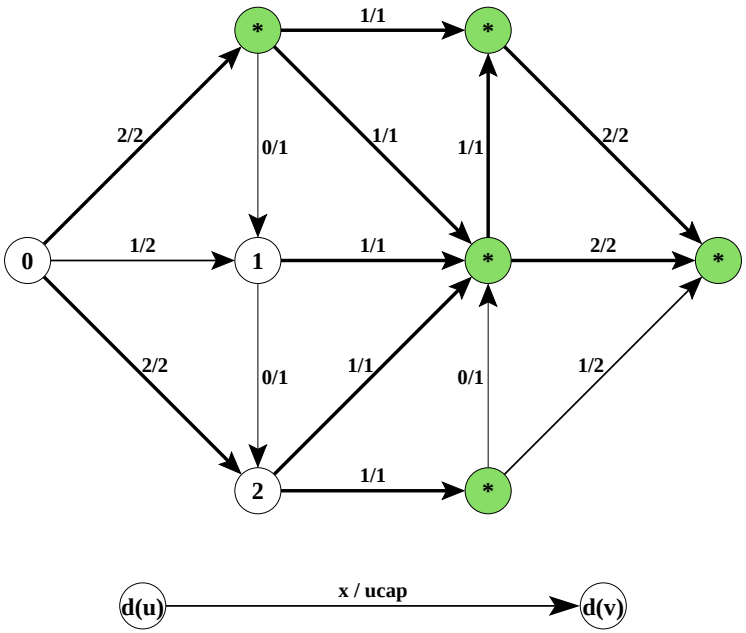


Figure 13.5: A Maximum Flow Problem

All max-flow methods have to be started with a feasible st-flow. Supposed that all lower bound are zero, one can start the solver with the zero flow. If you have already compute an st-flow and you have only increased some capacity or inserted a new arc, no special initialization of the flow labels is required. But if the source or target node have changed, you have to start with the zero flow again. If the lower bounds are non-trivial, you either need a feasible st-flow or you can fix the desired flow value by setting the demands of the root and the target node, and then search for an admissible b-flow instead.

13.8.1 The Augmentation Algorithm by Edmonds and Karp

This is the basic max-flow algorithm which depends on the search method `BFS`. The used data structures are the subgraph, the predecessor labels and the distance labels which determine a corresponding minimum cut eventually. The method runs in $O(nm^2)$ computing steps.

13.8.2 The Capacity Scaling Algorithm

The method `CapacityScaling` splits the balanced augmentation algorithm into scaling phases. In the `delta`-phase, only the arcs with `ResCap(a)>delta` are eligible. The parameter `delta` is initialized to the maximum capacity, and divided by 2 if no more augmenting paths can be found in this scaling phase. The resulting time bound is $O(m^2 \log U)$ where $U := \max_{a=0}^{m-1} ucap(a)$.

13.8.3 The Blocking Flow Algorithm by Dinic

This method heavily depends on the class `layeredAuxNetwork` which is described in Section 6.3.2. In contrast to the augmentation algorithm, the Dinic method does not compute the distance and predecessor labels before every augmentation step, but grows an acyclic incidence structure to perform a couple of augmentations.

The method runs in $O(n^2m)$ computing steps which is inferior to the push and relabel algorithm, but performs better in many practical situations. At least in the case where all arc capacities are one, the Dinic method is the most efficient and robust of the max-flow algorithms implemented.

13.8.4 The Push & Relabel Algorithm by Goldberg and Tarjan

This method iteratively chooses an **active** node, that is a node v which has `Divergence(v)>Demand(v)`. This node can either be relabeled so that the distance label increases, or a certain amount of flow is pushed to an appropriate neighbour of v.

The procedure `GoldbergTarjan()` supports three different strategies:

- If `methMaxFlow==2`, active nodes are selected by a FIFO strategy and an $O(n^3)$ algorithm results.
- If `methMaxFlow==4`, the set of active nodes is restricted to nodes whose flow excess exeeds a lower bound. This bound is decreased everytime when no more active nodes exist. This strategy is known as **excess scaling**. The running time is bounded by $O(nm + n^2 \log U)$.
- Otherwise, the active nodes are stored on a priority queue, and the priority of a node increases with its distance label. Here, the context variable `methPQ` determines the used PQ data structure, and the best possible complexity bound is $O(n^2\sqrt{m})$.

In either case, several push operations from a selected active node are performed and, if no further push is possible, the node is relabelled immediately.

We have experienced that the push & relabel technique can be even more efficient than blocking flow algorithms, but only if no flow has to be pushed back to the source node. In odd cases, not even a percent of the running time is needed to send the maximum flow to the sink node.

13.8.5 Admissible Circulations and b-Flows

An b-**flow** of a flow-network is a pseudo-flow such that all nodes are balanced. In the special situation where all node demands are zero, the b-flows are also called **circulations**.

The method `AdmissibleCirculation()` decides whether an admissible b-flow exists or not. This is achieved by using the class `FNW2FNW` which is described in Section 6.3.1, and the generic solver method `MaxFlow()`. Since the parameters n, m and U grow only by a constant factor during the problem transformation, the complexity bounds are the same as for the used max-flow methods.

13.8.6 Proposed Extension

The MKM blocking flow algorithm.

`methMaxFlow`	0	Successive augmentation
	1	Blocking flows, Dinic
	2	Push & relabel, FIFO
	3	Push & relabel, highest order
	4	Push & relabel, excess scaling
	5	Capacity scaling

Table 13.1: Maximum Flow Solver Options

13.9 Minimum Cuts and Connectivity Numbers

Synopsis:

```
class abstractMixedGraph
{
```

```
    virtual bool    Connected(TCap);
    virtual bool    EdgeConnected(TCap);
    virtual bool    StronglyConnected(TCap);
    virtual bool    StronglyEdgeConnected(TCap);

    TCap            Connectivity();
    TCap            EdgeConnectivity();
    virtual TCap    StrongConnectivity();
    virtual TCap    StrongEdgeConnectivity();

    virtual TFloat  StrongEdgeConnectivity(TNode);
    virtual TFloat  StrongEdgeConnectivity(TNode,TNode);

    TCap            MinCutLegalOrdering(TNode,TNode&,TNode&);
    TCap            MinCutNodeIdentification();
}

class abstractDiGraph
{
    TCap            MinCutHaoOrlin(TNode);
}
```

The **vertex connectivity number** is the minimum number of nodes which must be removed from the graph so that some nodes become disconnected. Correspondingly, the **edge connectivity number** is the minimum number of edges which must be removed so that two nodes become disconnected. The strong counterparts require that all connecting paths are eligible.

`methMinCut`	0	Iterated Max-Flows
	1	Push/Relabel, FIFO
	2	Push/Relabel, Highest Order
	3	Node Identification

Table 13.2: Minimum Cut Solver Options

In GOBLIN, there are two sets of methods which check for graph connectivity and which are listed above. The methods of the first series take the desired connectivity order k as an input parameter and check if the graph actually is k-connected. If k is small, one of the basic methods described in Section 13.6 is used. For (strong) edge connectivity of higher order, the connected components are determined by computing minimum cuts on the subgraph induced by a single colour and splitting the node set with respect to this cut. This requires the solution of $O(n^2)$ max-flow problems. For vertex connectivity of higher order, merely the method `Connectivity()` respectively `StrongConnectivity()` are called since connected components are immaterial here.

The methods of the second series compute a minimum cut and return the cut capacity. The cut is returned by the node colours where the colours of edge cuts are 0-1, and cut edge are directed from colour 1 to colour 0. Node cuts are coloured 0 and directed from the nodes with colour 1 to the nodes with colour 2.

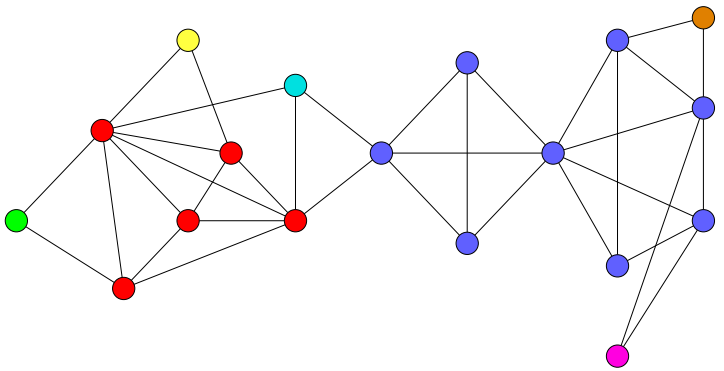


Figure 13.6: Edge Connected Components of Order 3

Note that the methods `Connectivity()` and `StrongConnectivity()` utilize node splittings which were described in Section 6.4.12 and essentially solve $O(n^2)$ max-flow problems. Since the node demands in the original graph map to arc capacities in the node splitting, the vertex connectivity methods observe **node capacities**. In order to compute vertex connectivity in the traditional sense, one must set all node demands to 1.

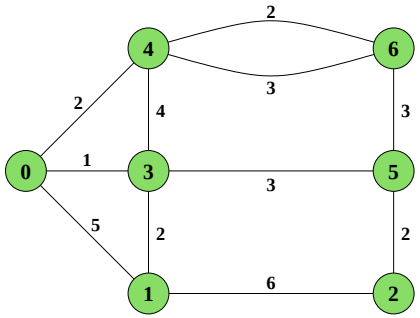


Figure 13.7: A Legal Ordering

If `methMinCut==0` is configured, each of the three edge connectivity methods solves $O(n)$ max-flow problems. If `StrongEdgeConnectivity(TNode)` or `EdgeConnectivity()` is called with `methMinCut>0`, a modified version of the push and relabel method `MinCutHaoOrlin()` is called which has the same worst-case time complexity as the original max-flow algorithm. To our experience, the highest order implementation performs much better than the FIFO version and the iterated max-flow strategy.

The method `EdgeConnectivity()` which works for the global min-cut problem supports a further algorithm: The method `MinCutNodeIdentification()` can be called which iteratively chooses a pair x, y of nodes, determines a minimum (x, y)-cut and then identifies the nodes x and y. These nodes and the (x, y)-cut capacity are supplied by `MinCutLegalOrdering(r,x,y)` where r is an arbitrary root of search. The search for a legal ordering is very similar to the Dijkstra and the enhanced Prim algorithm and hence runs in $O(n^2)$, $O(m \log n)$ or in $O(m + n \log n)$ time depending on the setting of `methPQ`. In practice, the node identification method performs much worse than the push/relabel method.

13.10 Minimum Cost Flows

Synopsis:

```
class abstractDiGraph
{
    enum        TOptMCFST {
                    MCF_ST_DEFAULT = -1,
                    MCF_ST_DIJKSTRA = 0,
                    MCF_ST_SAP = 1,
                    MCF_ST_BFLOW = 2
                };

    TFloat      MinCostSTFlow(TNode,TNode,TOptMCFST);

    TFloat      MCF_BusackerGowen(TNode,TNode);
    TFloat      MCF_EdmondsKarp(TNode,TNode);

    enum        TOptMCFBF {
                    MCF_BF_DEFAULT = -1,
                    MCF_BF_CYCLE = 0,
                    MCF_BF_COST = 1,
                    MCF_BF_TIGHT = 2,
                    MCF_BF_MEAN = 3,
```

```
                    MCF_BF_SAP = 4,
                    MCF_BF_SIMPLEX = 5,
                    MCF_BF_LINEAR = 6,
                    MCF_BF_CAPA = 7
                };

    TFloat      MinCostBFlow(TOptMCFBF);

    TFloat      MCF_CycleCanceling();
    TFloat      MCF_MinMeanCycleCanceling();
    TFloat      MCF_CostScaling(TOptMCFBF);
    TFloat      MCF_ShortestAugmentingPath();
    TFloat      MCF_CapacityScaling();

    TFloat      MCF_NWSimplex();
    void        MCF_NWSimplexCancelFree();
    void        MCF_NWSimplexStrongTree();
}
```

Two formulations of the min-cost flow problem are supported: st-flows and b-flows. All algorithms are accessed by the respective entry methods `MinCostSTFlow()` and `MinCostBFlow()`.

An st-**flow** is a pseudoflow such that all nodes are balanced, up to a fixed pair s, t of nodes, and the imbalance at node t is called the **value** of this flow. An **extreme** or **(ν)-optimal** st-flow is an st-flow which is optimal among all st-flows with the same value ν.

When calling the first solver, `MinCostSTFlow(s,t)`, a series of (ν)-optimal st-flows is computed, and the flow value ν is strictly increasing. This process halts if either a maximum flow has been determined or if the sink node t becomes balanced. This scheme is usually known as the **shortest augmenting path (SAP)** algorithm, referring to the fact that every intermediary extreme flow differs from its predecessor by a shortest augmenting path.

It is required that the input flow is also an extreme st-flow. In many situations, one can call the solver with the zero-flow which is (0)-optimal if the length labels are non-negative. The zero flow is admissible also for negative edge lengths if the digraph is acyclic. If node potentials are not already present, the solver is smart enough to compute a compatible dual solution before starting with augmentations.

The second solver, `MinCostBFlow()`, determines b-**flows** (in which all node imbalances match the given node demand vector b) of minimum costs. This includes the special case of **circulations** where the node demands are zero. The applied method can either be **primal**, that is, it starts with determining an arbitrary feasible b-flow. This solution is improved iteratively, and stays feasible throughout the computation.

Or the solver applies an SAP like algorithm to the b-flow problem with the slight difference to the st-flow solver that multiple terminal nodes occur. This stems from the fact that only complementary slackness but not primal feasibility is maintained, and that all supersaturated nodes are sources when searching for shortest augmenting paths.

The general drawback of SAP methods is that the running time complexity is polynomial only if the number of augmentations can be bounded polynomially. When optimizing from scratch, this is true for the capacity scaling method only.

The generic solver methods `MinCostSTFlow()` and `MinCostBFlow()` accept optional parameters in order to specify a particular algorithm. If these parameter are omitted, the context variables `methMinCFlow` and `methMinCCirc` apply. The possible values match the symbolic enum values which are listed above.

In all possible configurations, the solvers check if the node demamds sum up to zero, and raise an `ERRejected` exception otherwise. If the problem is infeasible for other reasons, `InfFloat` is returned. All methods preserve optimal flows as far as possible. As an exception, starting b-flow SAP codes with an optimal b-flow but with suboptimal node potentials may lead to a different final flow.

13.10.1 The SAP Algorithm by Busacker and Gowen

This method iteratively computes shortest paths using the generic method `ShortestPath()`, and augments on these paths. The running time is $O(\nu mn)$. The code is not for practical computations, but rather for comparison with the refined method which is discussed next.

13.10.2 The Refined SAP Algorithm by Edmonds and Karp

This method depends on the Dijkstra shortest path algorithm and on the reduced length labels. The running time is $O(\nu(m + n \log n))$. This method is of practical importance, since it solves the assignment problem in $O(n(m+n \log n))$ computing steps and can produce a near-optimal solution for the general 1-matching problem in the same order of complexity.

The method may be called with an *st*-flow and with node potentials such that the residual network contains negative length arcs. In that case, a label setting shortest path method is called, and the node potentials are corrected. Doing so, it may turn out that the input flow is not extreme. In that case, an `ERRejected` exception is raised.

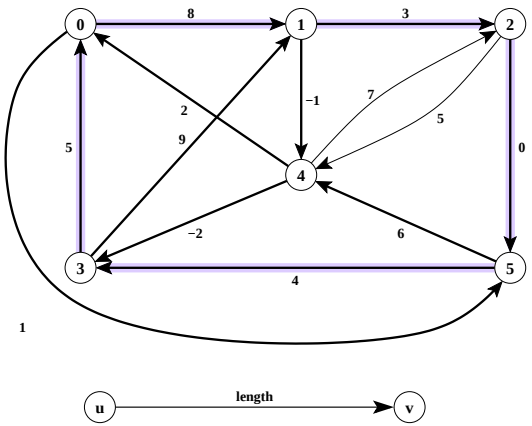


Figure 13.8: A Minimum Mean Cycle

13.10.3 The Cycle Canceling Algorithm by Klein

This is the very basic primal min-cost flow method. It iteratively searches for negative length cycles and augments on these cycles. Consult Section 13.1.10 for information on the method `NegativeCycle()` which is applied. If no negative length cycle exists, the circulation is optimal, and the procedure halts.

The procedure is non-polynomial and also performs very badly in practice. It is only useful for post-optimization and educational purposes.

13.10.4 The Minimum Mean Cycle Canceling Algorithm

This is a strongly polynomial specialization of the cycle canceling method where all augmenting cycles are minimum mean cycles. The running time is $O(n^2m^3 \log n)$ and $O(n^2m \log (nC))$ where $C := \max_{i=0}^{m-1}\{length(a)\}$. The major drawback is the fact that the procedure requires $\Theta(n^2)$ units of storage. In our experience, it is also too slow to solve practical min-cost flow problems.

13.10.5 The Cost Scaling Algorithm

An **ϵ-optimal** b-flow is a b-flow such that one can find node potentials with `RedLength(a)>=-epsilon` for every arc which satisfies `ResCap(a)>0`. A b-flow is **ϵ-tight** if it is ϵ-optimal, but it is not δ-optimal for any $\delta < \epsilon$.

The cost scaling algorithm iteratively transforms a 2ϵ-optimal b-flow and compliant node potentials into an ϵ-optimal b-flow and a corresponding set of potentials.

During such a scaling phase, the method manipulates an ϵ-optimal pseudo-flow rather than a b-flow. It performs push and relabel operations similar to the non-weighted algorithm until every node is balanced.

The running time is $O(n^3 \log(nC))$ for the basic version without *epsilon*-tightening. If tightening operations are enabled (depending on the value of

`methMinCCirc`, the method `MinimumMeanCycle()` is called to check for optimality after each scaling phase, and the running time is $O(mn^3 \log n)$. Each of the tightening steps derives potentials for which the b-flow is $epsilon$-tight. It can been experienced that the method performs much better without applying `MinimumMeanCycle()`.

13.10.6 The Multi Terminal SAP Method

The SAP algorithm starts with sending flow on all arcs which have residual capacity but the reduced length is negative. After that operation, the complementary slackness condition is satisfied, but many nodes are unbalanced. So the remainder of the procedure is sending flow on shortest paths in the residual network and updating the node potentials correspondingly.

Only a non-polynomial complexity order $O(m\ U(m + n \log n))$ for the running times is achieved here, but the method performs well in practice. In particular, it is well-suited for post-optimization.

13.10.7 The Capacity Scaling Method

The capacity scaling method `MCF_CapacityScaling()` is a variant of this SAP method which limitates the number of augmentation steps by choosing augmenting paths with sufficiently high capacity.

The running time is bounded by $O(m \log(U)(m + n \log n))$ but, frankly, can be achieved only for graphs which have infinite capacity paths between each pair of nodes. Practically, the capacity scaling method performs slightly better than the cost scaling method.

13.10.8 The Primal Network Simplex Method

The network simplex method can be considered an adaption of the general simplex method to network flows but also a clever specialization of the Klein cycle canceling algorithm. In order to generate negative cycles, the network simplex method maintains node potentials and a spanning tree which entirely consists of arcs with zero reduced length.

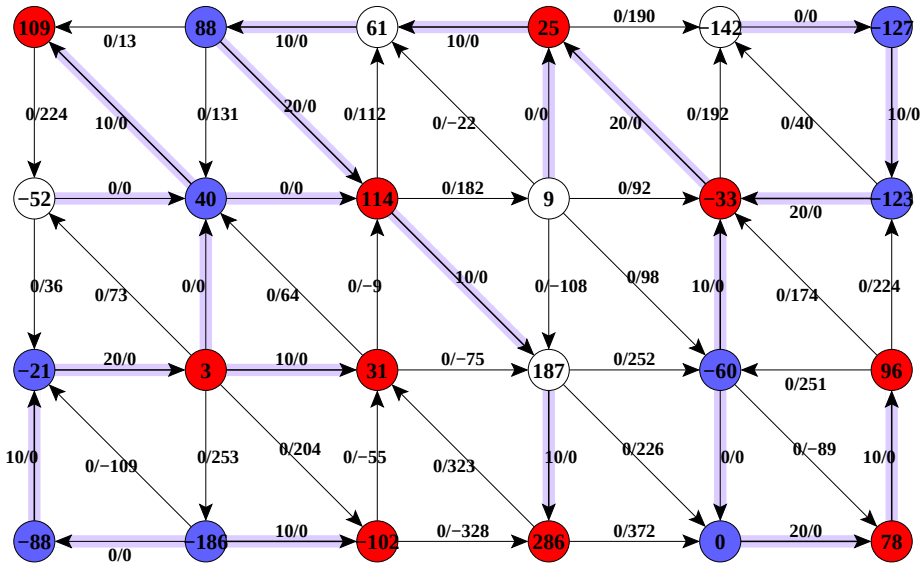


Figure 13.9: A Strongly Feasible Spanning Tree

Every arc not in the tree which has negative reduced length can be extended by tree arcs to a cycle with negative length. One selects such a **pivot arc** with positive residual capacity, but even then it is impossible to guarantee that the **pivot cycle** has residual capacity different from zero. Such **degenerate pivot steps** do not affect the flows but change the node potentials and the spanning tree structure. To fix up the problems with degeneracy, one uses **strongly feasible spanning tree structures** where every arc can send flow up the tree and the leaving arc on the pivot cycle is chosen carefully.

In our implementation, connectivity is neither required for the input graph nor forced by a problem transformation. The method must start with a feasible b-flow. In a first step, a procedure `NWSimplexCancelFree()` transforms the initial b-flow into a **cycle free solution** by a depth first search strategy in $O(m^2)$ time. Then `NWSimplexStrongTree()` computes a strongly feasible spanning tree structure from any given cycle free b-flow in $O(m\ n)$ time.

The main functionality is located in a separate class `networkSimplex`, especially the management of spanning tree indices and the data structures which are needed for the pricing step. The initilization phase ends by calls to methods `InitThreadIndex()` and `ComputePotentials()` of this class which take $O(n)$ steps together. If the network simplex method is started with a cycle free solution, the neccesary data are reconstructed without modifying the b-flow!

The cyclic part consists of alternating calls to `PivotArc()` and `PivotOperation()`. The pivot arc determination takes $O(m)$ steps and the pivot step $O(n)$ steps. The practical perfomance depends on a good **pricing rule** which is fixed by the context parameter `methNetworkPricing`. All possible rules are based on the idea of choosing the arc with the most negative reduced length for pivoting. By the **Dantzig rule**, all arcs are considered. By the **partial pricing** and the **multiple partial pricing** rules, only a few arcs are considered. The extreme case is the first eligible arc rule where only one admissible pivot arc is generated.

The network simplex code performs much better than the cost scaling method up to the case where Dantzigs rule is applied. The other rules show a similar performance and partial pricing performs best of all methods.

`methMinCFlow`	**0**	Revised shortest path
	1	Shortest path
	2	Transformation to b-flows
`methMinCCirc`	0	Klein (cycle canceling)
	1	Cost scaling
	2	Cost scaling with ϵ-tightening
	3	Minimum mean cycle canceling
	4	Shortest Augmenting Path
	5	Primal network simplex
	6	Reduction to linear program
	7	Capacity scaling
`methNetworkPricing`	**0**	Partial pricing
	1	Multiple partial pricing
	2	Dantzig
	3	First eligible arc

Table 13.3: Min-Cost Flow Solver Options

13.11 Balanced Network Search

Synopsis:

```
class abstractBalancedFNW
{
    bool    BNS(TNode,TNode = NoNode);
    bool    BNSKocayStone(TNode,TNode = NoNode);
    bool    BNSKamedaMunro(TNode,TNode = NoNode);
    bool    BNSHeuristicsBF(TNode,TNode = NoNode);
    bool    BNSMicaliVazirani(TNode,TNode = NoNode);
```

```
    void    Expand(TNode,TNode);
    void    CoExpand(TNode,TNode);
}
```

A **valid** path in a balanced flow network is an eligible path which does not traverse a pair of complementary arcs with `BalCap(a)==BalCap(a^2)==1`. A **balanced network search (BNS)** method is a procedure which decides which nodes are reachable by a valid path from a specified root node.

All procedures take one or two parameters. The first one is the root of search while the second optional parameter is a target node which should be reached on a valid path. If such a target t is specified, the method effectively decides whether t is reachable and halts once t has been reached. The BNS methods may either be exact, that is, a valid path is determined for every node which is reachable on a valid path, or heuristic.

For every BNS method and every node with finite distance label, a valid augmenting path can be expanded by using the method `Expand`. This method recursively calls `CoExpand`, and utilizes the `prop` and the `petal` data structures which are returned by all BNS methods. The exact methods `BNSKocayStone` and `BNSMicaliVazirani` also return an odd set system by the partition data structure.

The generic solver method `BNS` calls one of the various BNS methods according to the context variable `methBNS`. Note that the DFS heuristics and the Kameda-Munro heuristics are both encoded into the method `BNSKamedaMunro` which also reads `methBNS`. If a heuristical method is selected but the target is missed, the method `BNSKocayStone` is called to verify the negative result.

13.11.1 The Algorithm by Kocay and Stone

This method is in the tradition of the Edmonds/Gabow cardinality matching algorithm and uses a BFS approach. It is exact, that is, it finds a valid augmenting path if there is one. Although improved to $O(m)$ complexity, it cannot beat the running times of the heuristic methods. It also fails to compute paths of minimum length, but is much simpler than the Micali/Vazirani algorithm.

13.11.2 The Breadth First Heuristics

This is the most simple BNS procedure since it totally ignores the neccessity of blossom shrinking and does not use any high level data structures. It runs in $O(n^2)$ time, but performs better than the Kocay/Stone algorithm up to a size of 5000 nodes approximately. Just as the other heuristics, `BNSHeuristicsBF` does not compute a dual solution (odd set system).

13.11.3 The Depth First Heuristics by Kameda and Munro

This is the most efficient method according to our experiments and runs in $O(m)$ time. It utilizes two stacks for the management of blossoms instead of a disjoint set system. Both version are only an heuristics which may miss to find a balanced augmenting path even if there is one. The enhanced version requires some additional storage for time stamps but misses a node only in pathological situations.

13.11.4 The Algorithm by Micali and Vazirani

This method finds the distance labels with respect to the specified root node in $O(m)$ time. It depends on layered auxiliary shrinking networks, into which the shortest augmenting paths are encoded. For the time being, no st-path is extracted to the `prop` and `petal` data structures.

`methBNS`	**0**	Breadth-First (Kocay/Stone)
	1	Depth-First Heuristics
	2	Depth-First Heuristics (Time Stamps)
	3	Breadth-First Heuristics

Table 13.4: Balanced Network Search Options

13.12 Maximum Balanced Network Flows

Synopsis:

```
class abstractBalancedFNW
{
    TFloat  MaxBalFlow(TNode);
    TFloat  BNSAndAugment(TNode);
    TFloat  BalancedScaling(TNode);
    TFloat  Anstee(TNode);
    TFloat  MicaliVazirani(TNode,TNode = NoNode);

    void            CancelEven();
    virtual TFloat  CancelOdd();
}
```

A **balanced pseudo-flow** on a balanced flow network is a pseudo-flow such that for every arc a, `BalFlow(a)==BalFlow(a^2)` holds. In contrast to the ordinary max-flow solvers, one only specifies the source node `s`. The sink note `t == (s^1)` is determined by the flow symmetry.

The generic solver method `MaxBalFlow` calls one of the actual problem solvers according to the value of `methMaxBalFlow`.

13.12.1 The Balanced Augmentation Algorithm

The method `BNSAndAugment` is the most simple method which iteratively calls `BNS`. It solves a k-factor problem in $O(nm)$ time for fixed k, and the general maximum balanced flow problem in $O(\nu m)$ time.

13.12.2 The Capacity Scaling Algorithm

The method `CapacityScaling` splits the balanced augmentation algorithm into scaling phases. In the `delta`-phase, only the arcs with `BalCap(a)>=delta` are considered. The parameter `delta` is initialized to the maximum capacity, and is divided by 2 if no further augmenting path can be found. The resulting time bound is $O(m^2 \log U)$.

Except for the final scaling phase, every augmenting path is valid. Hence, the balanced network search is replaced by an ordinary BFS for the bulk of the computation. Effectively, the effort decreases to the solution of an ordinary network flow problem.

13.12.3 The Phase-Ordered Algorithm

The method `MicaliVazirani` is the fastest cardinality matching algorithm both in practice and theory achieving the complexity bound $O(\sqrt{n}m)$. The general problem of maximum balanced network flows is solved in $O(n^2 m)$ time.

In fact, the GOBLIN implementation is not the original Micali/Vazirani algorithm but a careful extension to balanced network flows. It is the most involved of the maximum balanced flow solvers and implemented by its own class (see Section 6.3.5 for the details).

13.12.4 The Cycle Canceling Algorithm

The method `Anstee` computes an ordinary maximum st-flow which is symmetrized afterwards resulting in a balanced flow which is half-integral but not integral. The non-integral flow values are deleted successively by calling the methods `CancelEven` and `CancelOdd`. The latter method may decrease the flow value and perform some balanced augmentation steps which conclude the computation.

Note that the generic method `MaxFlow` is used which allows to select from all implemented max-flow algorithms. On the other hand, `CancelOdd` calls the procedure `BNSKocayStone` directly which is needed to perform the balanced augmentations.

Hence the complexity of the `Anstee` algorithm is dominated by the max-flow algorithm used. If one uses the push and relabel algorithm, this yield the best complexity bound, namely $O(\sqrt{m}n^2)$, but the Dinic algorithm performs much better for explicit matching problems. In our experience,

MicaliVazirani and Anstee perform much better than the augmentation methods, but neither beats the other.

methMaxBalFlow	0	Successive augmentation
	1	Phase-Ordered augmentation
	2	Phase-Ordered augmentation with look-ahead
	3	Phase-Ordered augmentation with look-ahead and augmentation
	4	Capacity scaling
	5	Cycle canceling

Table 13.5: Maximum Balanced Flow Options

In the situation of k-factor problems, Anstee runs in $O(nm)$ time which is the same as for the basic method BNSAndAugment. However, it has turned out that CancelOdd decreases the flow value only by a very small amount (probably < 10 for 10^5 nodes). Since balanced augmentation steps are considerably more expensive than ordinary augmentations, the method Anstee performs much better than BNSAndAugment in practice.

Note that the method CancelOdd() has two different implementations. The general procedure depends on the problem transformation class bal2bal. The second, simpler implementation, is in the class gra2bal, and hence applies to explicit matching problems only.

13.13 Weighted Balanced Network Flow Algorithms

Synopsis:

```
class abstractBalancedFNW
{
    TFloat  MinCBalFlow(TNode s);
    TFloat  PrimalDual(TNode s);
    TFloat  EnhancedPD(TNode s);
    TFloat  CancelPD();
}
```

13.13.1 The Primal-Dual Algorithm

The primal-dual algorithm for ordinary flows maintains an st-flow and a set of potentials which satisfy a reduced length optimality criterion. If these solutions admit an augmenting path such that all arcs on this path have zero reduced length, the PD algorithm augments as long as possible. Otherwise several dual updates (updates on the node potentials) are performed, each of which extends the set of s-reachable nodes.

In the setting of balanced networks, the dual solution consists of node potentials, a shrinking family, and variables assigned with the sets of this family. These data structures are managed by a special class surfaceGraph whose incidence structure is the graph in which all blossoms (sets in the shrinking family) are contracted to a single node. This class has been described in Section 6.3.6.

The GOBLIN implementations are still rather basic. That is, they do not use splitable priority queues or multiple search trees. The complexity so far is $O(\nu nm)$ where ν denotes the value of a maximum balanced st-flow (minus the vlaue of the initial flow). A later release of GOBLIN should achieve a $O(\nu n^2)$ implementation. See Table 13.6 for the available variations of the PD algorithm.

Note that all PD methods can run with modified length labels which are not physically present, but computed from the dual variables. This recursive computation takes $O(n)$ time, and hence may increase the complexity of the PD method by a factor n. It has the benefit that only $O(n)$ storage is needed for keeping the modified length labels compared to $O(m)$. This is important for large scale geometrical matching problems, say with $> 10^5$ nodes.

The recursive computation of modified length labels is enabled by the

context flag `methModLength`, and can be performed explicitly if `RModLength` is called.

13.13.2 The Enhanced Primal-Dual Algorithm

This is an improvement of the PD algorithm resambling the cycle canceling method `Anstee` discussed in Section 13.12.4. More explicitly, the method `EnhancedPD` first calls the generic solver `MinCFlow` to compute an ordinary extreme maximum st-flow.

This flow is symmetrized by using `CancelEven` and `CancelPD`. The latter method is essentially the same as the general implementation of `CancelOdd`, but calls `PrimalDual` for the final balanced augmentation steps. The mere symmetrization takes $O(m)$ steps.

By this preprocessing, `PrimalDual` is started with a complementary pair where the flow value is at most n less than the value of a maximum balanced st-flow, and therefore runs in $O(n^2m)$ time. The overall complexity is dominated by the min-cost flow solver which comes into play. If one enables the cost-scaling or minimum-mean cycle canceling method (see Section 13.10), a strongly polynomial procedure results.

If one applies `EnhancedPD` and the shortest path method `EdmondsKarp2` to a k-factor problem, the time complexity is the same as for the straight primal-dual method. However, the actual running times may decrease dramatically, since Dijkstra augmentations are conceptually much more simple and can be performed in $O(m + n \log n)$ time instead of $O(mn)$ time.

`methMinCBalFlow`	0	Primal-Dual
	1	Enhanced primal-dual
`methPrimalDual`	0	Restart BNS after each dual update
	1	Restart BNS after changes in the shrinking family
	2	Restart BNS after blossom expansions
`methModLength`	**0**	Recursive computation
	1	Store modified length labels

Table 13.6: Min-Cost Balanced Flow Options

13.14 Matching Solvers

Synopsis:

```
class abstractGraph
{
    virtual bool    MaximumMatching();
    virtual bool    MaximumMatching(TCap);
    virtual bool    MaximumMatching(TCap*,TCap* = NULL);

    virtual bool    MinCMatching();
    virtual bool    MinCMatching(TCap);
    virtual bool    MinCMatching(TCap*,TCap* = NULL);

    TFloat          MinCEdgeCover();
}
```

Matching problems are solved in GOBLIN by transformation to balanced flow networks generally. This involves the class `gra2bal` which was discussed in Section 6.3.4. Note that the general methods are overridden for bipartite graphs by another problem transformation which depends on the class `big2fnw`.

If the matching solver is called without any parameters, a subgraph with `Deg(v)==Demand(v)` for every node v is computed. It may also be called with an integer k, and then returns a k-factor, that is, `Deg(v)==k` for every node. Finally, the matching solver may be called with two degree sequences a and b. Then `a[v]<=Deg(v)<=b[v]` will hold for the resulting subgraph.

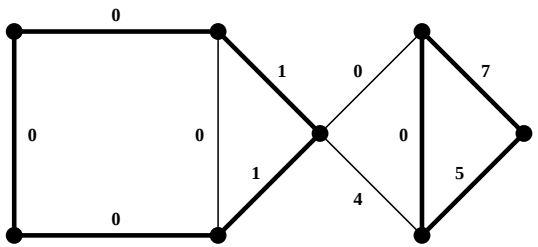


Figure 13.10: An Optimum 2-Factor

For complete graphs and bigraphs, one can solve the matching problem on a sparse subgraph either heuristically (if `methSolve==0`) or to optimality. The candidate graph consists of 10 greedy like heuristic matchings and the `k` nearest neighbours of each node where `k=methCandidates`. Note that this option is provided for optimization with the internal node demands only. If the graph is non-bipartite, only the fractional matching problem is solved on the candidate graph.

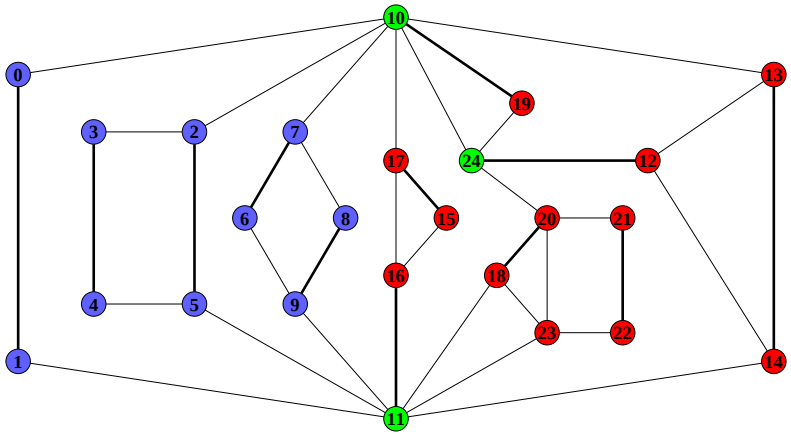


Figure 13.11: Gallai-Edmonds Decomposition

There is a procedure `MinCEdgeCover()` which determines an edge cover with minimum weight. This uses the well-known reduction to the 1-factor problem. In particular, the worst-case time complexity is the same as for the underlying matching solver. A procedure to compute a minimum size edge cover from a given maximum cardinality maching is discussed in Section 11.2.

13.15 T-Join and Postman Problems

Synopsis:

```
class abstractGraph
{
    void    ComputeTJoin();
    void    MinCTJoin();
    void    MinCTJoin(TNode,TNode);
```

```
    void    ChinesePostman();
}

class diGraph
{
    void    ChinesePostman();
}
```

An **Eulerian cycle** is an eligible closed walk which traverses every arc a at least `Cap(a)` times. A graph is **Eulerian** if it admits an Eulerian cycle.

The **Chinese postman problem (CPP)** asks for an eulerian supergraph such that the Eulerian cycle has minimum length. This problem is NP-hard for general mixed graphs, but can be reduced to matching problems if the graph is either directed or undirected. These easy cases are handled in GOBLIN.

Given a node set T of even cardinality, a **T-join** is a subgraph in which all nodes in T have odd degree and all other nodes have even degree. The undirected CPP is a special case of the minimum T-join problem which is actually solved in GOBLIN. Note that the minimum T-join problem has several further interesting special cases: 1-matching, shortest paths and optimization on the **cycle space**.

All methods require $\Theta(n^2)$ storage for the complete graph on which the respective matching problems are solved. Hence the CPP solvers do not work for large scale problems, say with $n > 10^5$ nodes.

13.15.1 T-Joins

The method `ComputeTJoin()` requires non-negative length labels and a set T which is specified by the node demand labels, where a demand 1 denotes a member of T. This procedure reduces the T-join problem to a 1-matching problem so that the running time is dominated by the matching solver used.

The methods `MinCTJoin(TNode,TNode)`, `graph::ChinesePostman()` and `MinCTJoin()` can handle negative length labels. These latter procedures set the demand labels and then call `ComputeTJoin()` which actually determines the T-join.

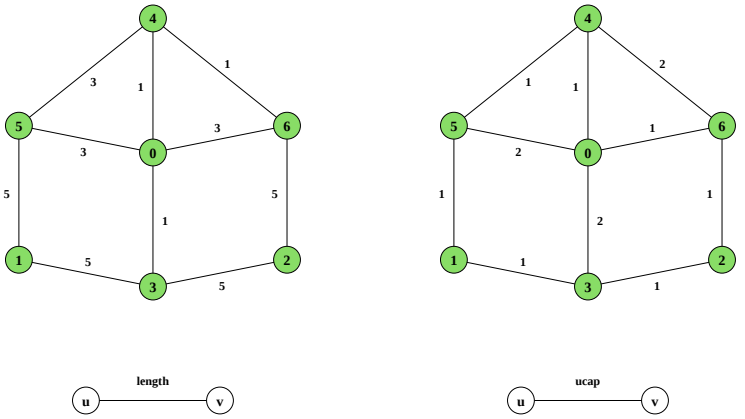


Figure 13.12: A Graph and a Minimum Eulerian Supergraph

13.15.2 The Undirected CPP

The method `abstractGraph::ChinesePostman` returns an Eulerian subgraph which has maximum weight rather than an Eulerian cycle. It calls the method `ComputeTJoin()` which has been described before.

The method `graph::ChinesePostman` increases the capacities of the graph to a minimal Eulerian supergraph and has been added to preserve the analogy of the directed and the undirected case.

13.15.3 The Directed CPP

The method `diGraph::ChinesePostman` reduces the CPP to a bipartite b-matching problem, and the running time is dominated by the matching solver used.

Note that the method is defined for the sparse implementation `diGraph` only. The procedure does not compute an Eulerian cycle but increases the capacities of the graph to a minimal Eulerian supergraph.

13.16 TSP Algorithms

Synopsis:

```
class abstractMixedGraph
{
    TFloat          TSP(TNode = NoNode);
    virtual TFloat  TSPHeuristics(TNode);
    TFloat          TSPHeuristicsRandom();
    TFloat          TSPHeuristicsInsert();
    TFloat          TSPHeuristicsInsert(TNode);
    TFloat          TSPHeuristicsTree();
    TFloat          TSPHeuristicsTree(TNode);
    virtual TFloat  TSPLocalSearch(TArc*);
    bool            TSPNodeExchange(TArc*,TFloat = 0);
    TFloat          TSPSubOpt1Tree(TNode,
                        TFloat = InfFloat,TOption = 1);
    TFloat          TSPBranchAndBound(TNode,
                        TFloat = InfFloat);
}

class abstractGraph
{
    virtual TFloat  TSPHeuristics(TNode);
    TFloat          TSPHeuristicsChristofides(
                        TNode = NoNode);
    TFloat          TSPLocalSearch(TArc*);
    bool            TSP2Exchange(TArc*,TFloat = 0);
    TFloat          TSPSubOpt1Tree(TNode,
                        TFloat = InfFloat,TOption = 1);
    TFloat          TSPBranchAndBound(TNode,
                        TFloat = InfFloat);
}

class denseDiGraph
{
    TFloat          TSPHeuristics(TNode);
}

class denseGraph
{
    TFloat          TSPHeuristics(TNode);
}
```

A **hamiltonian cycle** or **tour** is an eligible cycle which traverses every node exactly once. The **traveling salesman problem (TSP)** asks for a tour of minimum length. In GOBLIN, tours are represented by the predecessor labels.

The general TSP solver is defined by the method `TSP()`. This method is controlled by the configuration parameters `methTSP`, `methSolve` and `methLocal`. The parameters `methTSP` allows to select from several TSP heuristics, and `methLocal` enables or disables local search routines.

The parameter `methSolve` determines the general optimization level. If its value is zero, only a heuristic tour is computed. If its value is one, a subgradient optimization is performed to obtain good lower bounds. For higher values of `methSolve`, the problem can be solved to optimality by branch and bound. When the TSP solver starts, it first checks if a 1-tree exists to sort out some infeasible instances.

The configuration parameters `methSolve` and `methLocal` are designed to control other hard problem solvers too, but there is no application yet.

The optional node passed to `TSP()` is used by the TSP heuristics in different ways and may help to produce good starting solutions. In our experience, the subgradient optimization produces the best heuristic tours.

13.16.1 The Insertion Heuristics

The method `TSPHeuristicsInsert(r)` starts with a cycle through r and a neighbour of r and successively inserts nodes into this cycle. The node to be

inserted is the node with maximum distance from the cycle. It is inserted at the position where it causes the smallest possible costs. If one neglects the computation of node adjacencies, the running time is $O(n^3)$.

13.16.2 The Tree Approximation

The method `TSPHeuristicsTree(r)` expects an r-tree stored in the predecessor labels. This r-tree is transformed into a tour which is at most twice as long as the original tree if the graph is metric. If one neglects the computation of node adjacencies, the running time is $O(n)$.

13.16.3 The Christofides Approximation

The method `TSPHeuristicsChristofides(r)` combines the ideas of the tree heuristics and the Chinese postman algorithm. It first computes a minimum r-tree. Then a complete graph on the nodes with odd degree is instanciated, a perfect 1-matching of this graph is determined, and added to the graph. Then an Eulerian cycle results which can be contracted to a tour which is at most 50 percent longer than the initial r-tree if the graph is metric.

The final tour is returned by the predecessor labels, and its length is the return value. If one neglects the computation of node adjacencies, the running time is dominated by the complexity of the selected matching solver.

13.16.4 Local Search

GOBLIN provides a local search routine `TSPLocalSearch()` which can be used to improve the heuristic tours discussed so far. Local search is enabled by the configuration parameter `methLocal`. One can also start this post-optimization routine with a random tour by calling `TSPHeuristicsRandom`.

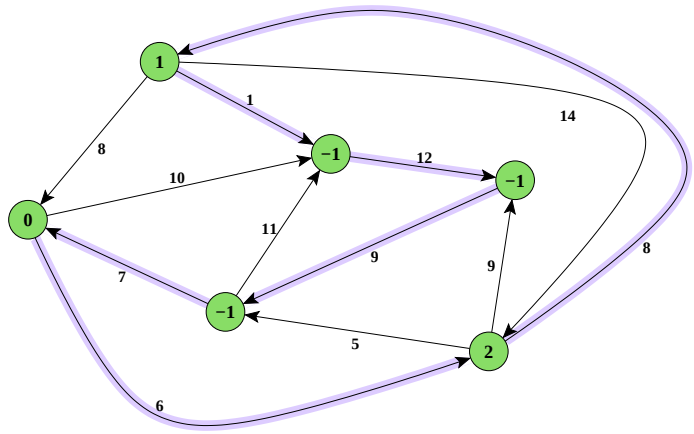
The method `TSPLocalSearch()` iteratively tries to improve the present tour by recursive calls to `TSP2Exchange()` and/or `TSP2NodeExchange()` The first method iteratively tries to improve a given tour by deleting two arbitrary arcs which are replaced by two new (and entirely determined) arcs. The second procedure selects a node which is deleted and inserted at another point of the tour.

Both local search procedures take an array of predecessor labels and an optional parameter which denotes the minimal improvement accepted for a local exchange. If this value is positive, a local exchange may increase the tour length by the specified amount.

Figure 13.13: A 2-Opt Step

13.16.5 The Subgradient Method by Held and Karp

The method `TSPSubOpt1Tree(r)` iteratively calls `MinTree(r)` which returns an r-tree as described in Section 13.5.4 by the subgraph data structure. If all nodes have `Deg(v)==2`, a tour is found, and the procedure halts.



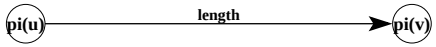


Figure 13.14: An optimal 1-Tree which forms a tour

Otherwise, the potentials of the nodes with `Deg(v)>2` are increased and the potentials of the nodes with `Deg(v)<2` are decreased by some amount, and the min tree solver is called again. If the TSP problem cannot be solved within a certain number of iterations, the procedure returns the best lower bound and the corresponding r-tree by the subgraph data structure.

If it looks promising, `TSPSubOpt1Tree()` calls `TSPHeuristicsTree(r)` which determines a feasible tour and hence an upper bound. The best tour found is returned by the predecessor labels. One may pass the length of a known tour by an optional parameter in order to initialize the upper bound.

This procedure yields very strong lower bounds on the length of an optimal tour, but one cannot expect that an optimal tour is found for practical instances. The quality of the bound depends on the third parameter which acts as follows: If a value of zero is passed, only a single 1-tree is computed without changing any node potentials. If a value of one is passed, good potentials are computed within a reasonable number of iterations, say 100, roughly. For a value of two, a large number of iterations occurs, 3000 approximately. The found bound is much better for some instances, but on the average the fast strategy already achieves the optimal bound.

This calling parameter is matched by the context parameters `methRelaxTSP1` and `methRelaxTSP2` which specify how the 1-tree method is applied to find the initial bound respectively the partial bounds for the branch and bound scheme. A value of `methRelaxTSP2=2` does not yield a practical method.

13.16.6 Branch and Bound

There is a branch and bound solver `TSPBranchAndBound` which depends on the 1-tree relaxation. It also uses the node potentials found by subgradient optimization. If the configuration parameter `methCandidates` is negative, the branch and bound module evaluates the entire graph. Otherwise, a candidate graph is generated which consists of the best tour found so far, several random locally optimal tours and the nearest neighbours of each node. In that case, the value of `methCandidates` denotes a lower bound on the node degrees. See Section 10.3.2 for the details of the branch and bound module.

13.16.7 Application to Sparse Graphs

The TSP solver also applies to sparse graph objects and to graphs with non-trivial capacity bounds. The latter can be used to restrict the set of feasible solutions.

None of the implemented heuristics would be helpful, if applied to the original graph. Instead of this, the method `TSPHeuristics()` computes the metric closure of the graph (see Section 6.4.8 for more details). On this `metricGraph` object, the heuristics and the subgradient optimization are run irrespecitive of the current value of `methSolve`. If the tour of the metric closure maps to the original graph, this tour returned.

If branch and bound is enabled, this applies to the original graph. No candidate search is performed but the entire graph is evaluated.

`methTSP`	**0**	Random tour
	1	Insertion heuristics
	2	Tree heuristics
	3	Christofides (undirected graphs only)
`methRelaxTSP1`	0	Straight 1-tree bound
	1	Subgradient optimization (fast)
	2	Subgradient optimization (stable)

Table 13.7: TSP Solver Methods

13.17 Graph Colourings and Clique Covers

Synopsis:

```
class abstractMixedGraph
{
    TNode           NodeColouring(TNode);
    TNode           PlanarColouring();
    TNode           NCLocalSearch();
    bool            NCKempeExchange(TNode*,TNode,TNode);
    TNode           CliqueCover(TNode);
    TNode           EdgeColouring(TNode);
}
```

A **node colouring** is an assignment of colours to the graph nodes such that the nodes with equal colour are non-adjacent. A **clique cover** is an assignment of colours to the graph nodes such that every colour class forms a clique.

The procedure `NodeColouring` calls the enumeration scheme which is described in Section 10.3.4. The parameter denotes the acceptable number of colours. This value k has strong impact on the practical performance of the solver. For example, for a planar graph and a value of 6, the branch and bound would end within a single iteration. If k is very close to the chromatic number χ, the computational efforts are tremendous even for a 50 node graph. In the case of $k = 5$ and $n \leq 3m - 6$, the method `PlanarColouring` is called, and enumeration occurs only if the specialized method does not return a 5-colouring. The colouring of planar graphs requires $O(nm)$ time.

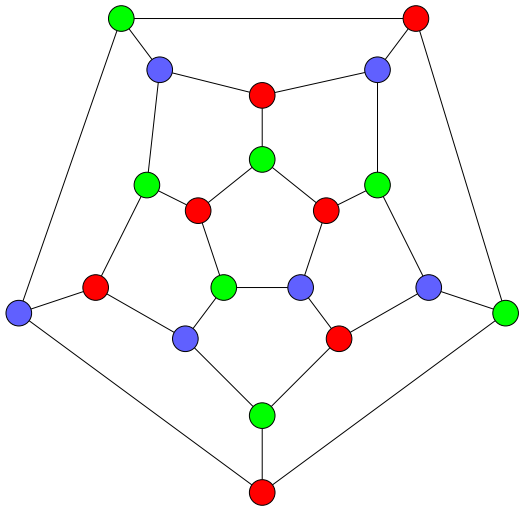


Figure 13.15: A 3-Colouring of the Dodecahedron

If no k is specified, the method produces a decreasing sequence of values for k, for which the enumeration scheme is started. By this strategy, one can produce colourings which come close to χ. Note that cliques sizes are lower bounds for χ. Hence, with some luck, it is possible to bound the chromatic number to a small interval.

The clique cover and the edge colouring method essentially perform a node colouring of the complementary graph and the line graph respectively. More precisely, if Δ denotes the maximum node degree, then `EdgeColouring(k)` computes an approximative edge colouring with either Δ or $\Delta+1$ colours in $O(m\Delta(m+\Delta\log\Delta))$ time. If $k > \Delta$, this colouring is returned by the subgraph labels. If $k < \Delta$, no k-edge colouring exists. Only if $k = \Delta$ and if the approximation method has obtained a $\Delta+1$-colouring, the enumeration scheme is used for an potential improvement.

All described methods return the number of colours in the final solution or the constant `NoNode` if no colouring was found.

If the context flag `methLocal` is set, the procedure `NCLocalSearch` is called with the final colouring obtained by the enumeration scheme. Each of the methods `NCLocalSearch()`, `PlanarColouring` and `EdgeColouring()` depend on the method `NCKempeExchange()` which takes the colours of the two specified nodes and flips the colours in the Kempe component of the first node. If both nodes are in in the same Kempe component, 0 is returned and 1 otherwise. Such an exchange operation needs $O(m)$ time.

13.18 Stable Sets and Cliques

Synopsis:

```
class abstractMixedGraph
{
    void            StableSet();
    void            Clique();
    void            VertexCover();
}
```

A **stable set** is a set of pairwise non-ajacent nodes whereas a **clique** consists of pairwise adjacent nodes and a **vertex cover** is a node set which contains at least one end node of every egde. The three listed methods return the maximum cardinality of a respective node set. The set itself consists of the nodes with colour 1.

All methods call the branch and bound solver for the stable set problem described in Section 10.3.1 and use heuristic graph colouring. Our experiments have turned out that one can compute cliques in graphs with 150-200 nodes depending on the graph density and on the quality of the heuristic colouring.

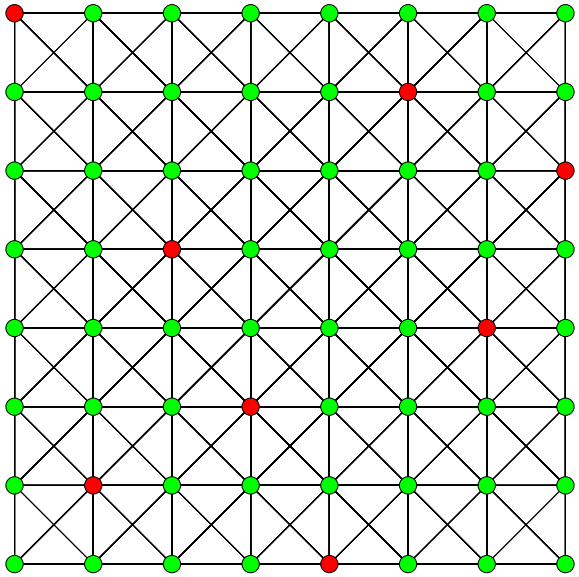


Figure 13.16: A Maximum Stable Set of Queens on a Chessboard

13.19 Discrete Steiner Trees

Synopsis:

```
class abstractMixedGraph
{
    TFloat          SteinerTree(TNode);
    TFloat          SteinerTrimLeaves(TArc*);
    virtual TFloat  SteinerHeuristics(TNode);
    virtual TFloat  SteinerEnumerate(TNode);
}

class abstractGraph
{
    TFloat          SteinerHeuristics(TNode = NoNode);
    TFloat          SteinerEnumerate(TNode = NoNode);
}
```

The method `SteinerTree()` evaluates the node demand labels which have to be either 0 or 1. A **Steiner tree** is a rooted tree or arborescence which spans all nodes with demand 1, the **terminals**. The demand 0 nodes are called **Steiner nodes** and are spanned only if they denote shortcuts.

The method `SteinerEnumerate()` enumerates on all possibilities for the Steiner nodes and iteratively calls the generic min-tree solver. Hence, the algorithm is non-polynomial and the running times are acceptable for at most ten Steiner nodes.

The method `SteinerTrimLeaves(TArc*)` turns a given spanning tree (arborescence) into a Steiner tree by successively deleting all Steiner nodes which are leaves. The running time is $O(n)$, the return value is the sum of lengths of the deleted arcs. The general implementation of `SteinerHeuristics()` does nothing more than calling `MinTree()` and `SteinerTrimLeaves()`.

In undirected graphs, `SteinerHeuristics()` implements the Mehlhorn 2-approximation algorithm. This method calls `Prim2()` with some discrete adaption of the Voronio geometry. The running time is $O(m + n \log n)$ and is dominated by the shortest path problem which must be solved to the compute the Voronoi regions (see Section 13.1.5 for the details).

The compound solver method `SteinerTree()` calls the heuristics and, if `methSolve>1`, the enumeration scheme. Lower bounds can be obtained without complete enumeration in the undirected case only.

13.20 Maximum Edge Cuts

Synopsis:

```
class abstractMixedGraph
{
    TFloat MaxCut(TNode=NoNode,TNode=NoNode);
    virtual TFloat MaxCutHeuristics(
                TNode=NoNode,TNode=NoNode);
    TFloat  MaxCutHeuristicsGRASP(
                TNode=NoNode,TNode=NoNode);
    TFloat  MaxCutLocalSearch(
                TNode*,TNode=NoNode,TNode=NoNode);
    TFloat  MaxCutBranchAndBound(TNode=NoNode,
                TNode=NoNode,TFloat=InfFloat);
}

class abstractGraph
{
    TFloat  MaxCutHeuristics(
                TNode=NoNode,TNode=NoNode);
    TFloat  MaxCutHeuristicsTree(
                TNode=NoNode,TNode=NoNode);
    TFloat MaxCutDualTJoin(TNode=NoNode);
}
```

A **maximum cut** is a strong edge cut with the maximum sum of weights where the **weight** of an arc a is defined as `UCap(a)*Length(a)`. In undirected graphs with unit edge capacities and lengths, a maximum cut corresponds to a maximum bipartite subgraph.

For all described max-cut algorithms, the return value is the cut weight. The cut is returned by the node colours which can be either 0 or 1. Cut arcs are directed from colour 0 to colour 1 and only non-blocking arcs are counted for the cut weight. If one or two optional nodes are specified, the first node is always coloured 0 and the second node is coloured 1.

Apart from the exact methods, GOBLIN provides two starting heuristics:

- In the general setting, `MaxCutHeuristicsGRASP()` applies which assigns colours to the nodes step by step. In each iteration, a candidate list with a few nodes is generated and from this list, an arbitrary node is chosen. Then, this node is always added to the most profitable component. If the graph is undirected, the cut weight is at least 1/2 of the sum of arc weights.

- In undirected graphs, `MaxCutHeuristicsTree()` computes a minimum spanning tree where the length labels are substituted by the arc weights. After that, the bipartition is chosen with respect to the tree.

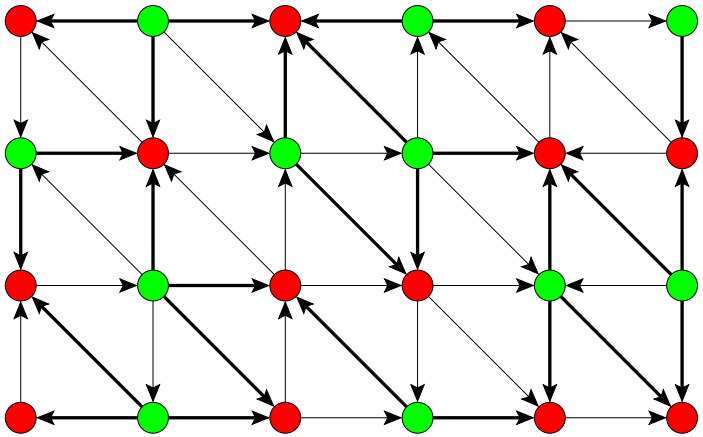


Figure 13.17: A Maximum Edge Cut

The local search procedure `MaxCutLocalSearch()` iteratively shifts a single node from one component to another if the cut capacity strictly increases by that operation. Every iteration takes $O(m)$ computing steps but the number of iterations cannot be bounded polynomially. As for other solvers, the local search procedure is integrated into the heuristics and executed when the context flag `methLocal` is set.

For planar undirected graphs with non-negative arc weights, the method `MaxCutDualTJoin()` determines a maximum $\emptyset$-join of the dual graph and maps it back to an edge cut. This is an exact algorithm, not just a heuristic! The running time is dominated by the used T-join method.

The method `MaxCutBranchAndBound()` applies to the general setting but can solve and proof optimality for small graphs only (with roughly 30 nodes and less).

Part IV

Miscellaneous

Chapter 14

The Object Controller

With any object derived from the base class `goblinDataObject` (graph objects, iterator objects and data structures), a `goblinController` object is associated. To this controller object, we refer as the **context** of the data object.

Data objects may share their controller with other data objects. In particular, iterators, logical views and temporary data structures used in algorithms are in the same context as the referenced graph object.

There is a global controller object, namely the `goblinDefaultContext`. For the most default and file constructors of GOBLIN data objects, a reference to `goblinDefaultContext` appears as a default parameter.

14.1 Construction

Synopsis:

```
class goblinController
{
    goblinController();
    goblinController(goblinController&);
}
```

Whenever a controller object is instanciated, this generates a couple of timer objects and an object hash table which allows to dereference the dependent data objects from a given integer handle. All other context parameters are initialized either with default values or the respective value of the master context.

The **copy constructor** method produces a clone of the controller object passed by its reference. All built-in type and `char*` string values are copied, event handlers and module entry points are inherited from the master context.

For example, a display configuration is a volatile controller object and copied from the context of the object to be displayed. This controller is modified with some class dependent display parameters before the object is actually mapped or written to file:

```
exp2tk E(*this,"dummy.tk");
ConfigDisplay(E.Configuration());
E.DisplayGraph();
```

Controller objects which are constructed by the **default constructor**, are somewhat like clones of the global object `goblinDefaultContext`.

Other than for the subsequent controller instanciations, the construction of `goblinDefaultContext` also generates a controller object hash table which allows to dereference all valid controller and data objects from their handle.

14.2 Interaction with Data Objects

Synopsis:

```
class goblinController
{
```

```
public:

    THandle         InsertObject(goblinDataObject*);
    void            DeleteObject(THandle);

    goblinRootObject*   ObjectPointer(THandle) const;
    goblinRootObject*   Lookup(THandle) const;
}

class goblinDataObject : public goblinRootObject
{
protected:

    goblinController &CT;
    THandle &H;

public:

    goblinDataObject(goblinController &
                           = goblinDefaultContext);
    goblinController &Context();
    THandle &Handle();
}
```

Every constructor of a GOBLIN data object subscribes to the controller object which forms its context. This is managed by the method `InsertObject` which returns a globally unique **object handle**.

This functionality is transparent to the programmer. If a new class is derived from `goblinDataObject` or its descendants, the programmer merely writes a call `goblinDataObject(CT)` into all constructors of the new class. Here, `CT` denotes the desired context.

The context of a `goblinDataObject` and its respective handle can be accessed by the class methods `Context()` and `Handle()`. Conversely, controllers can determine the addresses of the hosted data objects from their handles by means of `ObjectPointer()`. If only the handle `H` but not the context is known, `goblinDefaultContext.Lookup(H)` returns the address.

It is particular useful to store handles instead of addresses if the referenced object may be deleted within the life time of the referencing object. `Lookup()` returns a `NULL` pointer when dereferencing raises a segmentation fault!

Internally, all data objects hosted by the same controller object are in a linked list. Since all controller objects are also in a linked list, it is possible to enumerate all valid GOBLIN data objects. The method `DisplayAll()` is a straight forward application.

To every controller object, one can assign a **master object** by calling `SetMaster()` with the handle of the desired master object. This handle can be questioned by the method `Master()`. The master object determines the context label and, implicitly, the labels of all unnamed objects in that context.

14.3 Logging

GOBLIN is fitted with an elaborate logging module. Like the tracing module which is discussed later, it can be used for debugging, and also for preparation of runtime examples.

14.3.1 Event Handlers

Synopsis:

```
class goblinController
{
private:

    unsigned long suppressCount;
```

```
public:

    void (*logEventHandler)(msgType,TModule,THandle,char*);

    void PlainLogEventHandler(msgType,TModule,THandle,char*);
    void DefaultLogEventHandler(msgType,TModule,THandle,char*);

    void SuppressLogging();
    void RestoreLogging();
}
```

In order to keep any multitasking code out of the core library, we have introduced a function pointer `logEventHandler` which originally references the method `DefaultLogEventHandler()`. This method writes all passed logging information to the file or device referenced by `logStream`. There is an alternative procedure `PlainLogEventHandler()` which processes user-readable output to the same stream but only handles plain message texts.

The messenger and the GOSH shell which are discussed later provide more complex event handlers. These procedures call `DefaultLogEventHandler()` in turn.

The method `SuppressLogging()` saves and deregisters the current event handler, `RestoreLogging()` registers the saved event handler again. Calls must be matching, but it is save to use these methods in a nested way.

14.3.2 Writing Log Entries

Synopsis:

```
class goblinController
{
private:

    THandle LogFilter(msgType,THandle,char*);

public:

    char logBuffer[LOGBUFFERSIZE];

    void    LogEntry(msgType,THandle,char*);
    THandle LogStart(msgType,THandle,char*);
    void    LogAppend(THandle,char*);
    void    LogEnd(THandle,char* = NULL);
}
```

Data objects do not call the registered event handler directly but rather the context methods listed above. If no handler is registered, nothing happens. Otherwise the information delivered by the data object is extended by some structural information and passed to the event handler.

Logging information is grouped into several classes each of which is represented by a token of the enumeration type `msgType`. Table 14.3.5 shows the tokens which are used for the GOBLIN core library logging information. The tokens associated with errors and with the GOSH shell are discussed later in this document.

The parameters of `LogFilter()` are such a token, an object handle and a text line which has to be logged. It manages the filtering of message types and handles the event handlers.

The method `LogEntry()` does nothing else than calling `LogFilter()` and suppressing messages nested into compound log entries. One can use the predefined buffer `logBuffer` to pass the message text but only for strings up to a size of `LOGBUFFERSIZE-1`.

The methods `LogStart()`, `LogAppend()` and `LogEnd()` are used to grow compound messages from a series of strings. To this end, `LogFilter()` and, eventually, the event handler are called with a special message type `MSG_APPEND`. The handle returned from `LogStart()` must be passed for the trailing components. Calls to `LogStart()` and `LogEnd()` must be matching.

To prevent obvious overhead, data objects also implement a method `LogEntry()` which substitutes the own object handle in the context method. Compound messages can be written by data object methods likewise.

The method `Error()` calls the log event handler with a message composed from the two strings passed as arguments. The first string describes the scope where the exception occured and the second one describes the exceptional situation. All information about the most recent exception is saved internally. Finally, `Error()` throws a C++ exception depending on the delivered `msgType` token. See Chapter 19 for more details.

14.3.3 Structured Source Code

Synopsis:

```
class goblinController
{
private:

    TModule nestedModule[MAX_MODULE_NESTING];
    int     moduleNestingLevel;

public:

    char logDepth;
    char logLevel;

    void IncreaseLogLevel();
    void DecreaseLogLevel();

    enum TFoldOptions {
        NO_INDENT = 1,
        SHOW_TITLE = 2
    };

    void OpenFold(TModule,TOption = 0);
    void CloseFold(TModule,TOption = 0);
}
```

The event handlers do some alignment of the log entries depending on the current `logLevel` which can be manipulated by calls to `IncreaseLogLevel()` and `DecreaseLogLevel()`. The maximum indentation level is specified by `logDepth`.

In the same way, `OpenFold()` and `CloseFold()` manipulate the parameter `moduleNestingLevel` and set the current code module context: `OpenFold()` saves the new context on the stack `nestedModule` (if the maximum depth `MAX_MODULE_NESTING` has been reached, the context does not change effectively) and `CloseFold()` recovers the parent context.

If the `NO_INDENT` option is specified, `OpenFold()` [`CloseFold()`] implicitly calls `IncreaseLogLevel()` [`DecreaseLogLevel()`]. The option `SHOW_TITLE` causes `OpenFold()` to send the module name to the log event handler.

Note that data objects also implement `OpenFold()` and `CloseFold()` methods which cover the described functionality and, in addition, start and stop the module timers. See Section 17.3 for more details.

14.3.4 Filtering the output

The information which is actually logged can be filtered by several context parameters. The available flags are listed in Table 14.3.5. Note that all values higher than the default values may result in a tremendous increase of logging information. But for rather small problem instances, the options `logMeth==2` and `logRes==2` allow a good understanding of the various optimization algorithms. Other than the preliminary version of the logging module, the output is now filtered by the controller object internally.

The flags `logWarn` and `logMem` have been added for debugging purposes. The flag `logMem` is discussed in Section 17.1. The flag `logWarn` concerns GOBLIN exceptions (see Chapter 19) which do not affect the general data integrity. More explicitly:

- By `Error(MSG_WARN,..)`, an error message is printed only if `logWarn` is set but no exception is raised.
- By `Error(ERR_CHECK,..)`, an error message is printed only if `logWarn` is set and an exception `ERCheck` is raised in any circumstances.
- By `Error(ERR_REJECTED,..)` an exception `ERRejected` is raised and an error message is printed independently of the value of the flag

`logWarn`.

This is so since the exception class `ERCheck` does not necessarily indicate errors. For example, a call `FlowValue(s,t)` returns an exception `ERCheck` if the subgraph does not form an *st*-flow. Algorithms may check feasibility by this method, and treat the exception as a standard functionality. If tests are needed several times by an algorithm, the log file should not include corresponding error messages.

There is pragma `_LOGGING_` which is defined in the file `config.h`. This definition may be omitted in order to improve the performance. Note, however, that only a certain part of the logging module is compiled conditionally, namely the information which is assigned with `LOG_METH2`, `LOG_RES2`, `MSG_WARN`, and some of the `IncreaseLogLevel()` and `DecreaseLogLevel()` statements.

14.3.5 Selection of logging information

Variable	Def	Token	Information
`logMeth`	1	LOG_METH, LOG_METH2	Course of algorithms (two levels)
`logMem`	0	LOG_MEM, LOG_MEM2	Memory allocations (two levels)
`logMan`	1	LOG_MAN	Object manipulations
`logIO`	1	LOG_IO	File management
`logRes`	1	LOG_RES, LOG_RES2	Computational results (two levels)
`logTimers`	1	LOG_TIMERS	Timer statistics
`logGaps`	1	LOG_GAPS	Duality gaps
`logWarn`	0	MSG_WARN	Warnings

14.4 Method Selection

This section merely summarizes the method selector flags which have been described with the respective problem solver methods. For details, we refer to the Chapters 8 and 11.

Synopsis:

```
class goblinController
{
    int     methFailSave;
    int     methAdjacency;
    int     methDSU;
    int     methPQ;
    int     methModLength;

    int     methGeometry;

    int     methSearch;
    int     methMaxFlow;
    int     methMinCFlow;
    int     methMinCCirc;
    int     methMinTree;
    int     meth1Tree;
    int     methMaxBalFlow;
    int     methBNS;
    int     methMinCBalFlow;
    int     methPrimalDual;
    int     methTSP;

    int     methLocal;
    int     methSolve;

    int     maxBBIterations;
    int     maxBBNodes;
}
```

14.4.1 Optional Data Structures

Variable	Value	Description
`methFailSave`	**0**	No special certificate checking
	1	Network flow and matching solvers are forced to verify a reduced costs optimality criterion
`methAdjacency`	0	Search incidence lists
	1	Generate hash table
`methDSU`	0	Path compression disabled
	1	Path compression enabled
`methPQ`	0	Use basic priority queue
	1	Use binary Heaps
	2	Use Fibonacci Heaps
`methModLength`	0	Recursive computation of reduced length labels
	1	Explicit data structure

14.4.2 Solver Options for NP-hard problems

Variable	Value	Description
`methSolve`	0	Apply only heuristics
	1	Compute lower and upper bounds
	2	Apply branch and bound
`methLocal`	0	Apply only construction heuristics
	1	Apply local search heuristics
`maxBBIterations`	**100**	Maximum number of branch and bound iterations divided by 1000
`maxBBNodes`	**20**	Maximum number of active leaves in the branch tree divided by 100
`methCandidates`	**-1**	Minimum degree in the candidate graph. If negative, candidate search is disabled. Used for TSP and weighted matching.

14.4.3 Problem Specific Solver Options

Variable	Value	Description
`methSearch`	**0**	FIFO label correcting
	1	Dijkstra
	2	Bellman/Ford
	3	BFS
`methMaxFlow`	0	Successive augmentation
	1	Dinic
	2	Push/Relabel, FIFO
	3	Push/Relabel, Highest Order
	4	Capacity scaling
`methMinCFlow`	**0**	Revised shortest path
	1	Shortest path
	2	Capacity scaling (Not implemented)
	3	Transformation to circulations
`methMinCCirc`	0	Klein (cycle canceling)
	1	Cost scaling
	2	Cost scaling with ϵ-tightening
	3	Minimum mean cycle canceling
	4	Transformation to st-flows
`methMinTree`	0	Prim
	1	Enhanced Prim
	2	Kruskal
`meth1Tree`	**0**	Ordinary spanning tree
	1	Minimum 1-trees
`methMaxBalFlow`	0	Successive augmentation
	1	Phase-Ordered augmentation
	2	Phase-Ordered augmentation with look-ahead
	3	Phase-Ordered augmentation with look-ahead and augmentation
	4	Capacity scaling
	5	Max-Flow start up
`methBNS`	**0**	Breadth-First
	1	Depth-First Heuristics
	2	Depth-First Heuristics
	3	Breadth-First Heuristics

Variable	Value	Description
`methMinCBalFlow`	0	Primal-Dual
	1	Enhanced primal-dual
`methPrimalDual`	0	Restart BNS after each dual update
	1	Restart BNS after changes in the shrinking family
	2	Restart BNS after blossom expansion
`methTSP`	**0**	Random tour
	1	Insertion heuristics
	2	Tree heuristics
	3	Christofides (undirected graphs only)

14.5 Tracing

Synopsis:

```
class goblinController
{
    int     traceLevel;
    int     threshold;
    int     fileCounter;
    int     traceStep;
    int     traceData;
    int     commLevel;
    int     breakLevel;

    void    Ping(THandle,unsigned long);
    void    ResetCounter();
    void    IncreaseCounter();
    ostream &Display();
}
```

The tracing functionality is a valuable tool both for debugging and for visualising of the course of an algorithm. The tracing can be controlled by the following members of the controller object:

A class method can be traced only if it defines a **breakpoint**. By this, we denote a method call `CT.Ping(H,priority)` which does the following:

The handle `H` specifies the object to be traced. The value of `priority` is added to the current value of `traceCounter`. If the new value of the counter exeeds `traceStep`, then `traceCounter` is reset to zero, and some information is written to an output device. To this situation, we refer as a **tracing point**. If one has `traceStep == 1`, every breakpoint triggers off a tracing operation.

The concrete output depends on the value of `traceLevel`. Table 14.5.1 gives an overview. A higher trace level generally generates more tracing information. Levels 3 and 4 are reasonable for small examples only, and may generate several megabyte of tracing files.

It is possible to suppress the first k tracing operations by setting the context variable `threshold` to k. Note that any error prompt contains a line

```
Before tracing point #...
```

which allows to debug large problems graphically by setting `threshold` to a reasonable value.

The message `Display()` which is mentioned in Table 14.5.1 is available for every GOBLIN data object. In order to trace an object, its class must implement the `Display()` method. So far, graph objects can be displayed both graphically and textually, and most data structures can be displayed textually. If textual output is configured, the output stream is obtained by the context method `ostream & goblinController::Display()`.

Otherwise a so-called trace file is written which, by default, consists of the graph object and the context information with some modifications in the GOBLIN native format. The file name is the concatenation of the context label obtained by the method `Label()`, the current value of `fileCounter` and one of the extensions `.gob`, `.fig` or `.tk`. Every trace file export will trigger off an increase in the value of `fileCounter`.

Note that the value of `priority` has strong impact on the quality of a tracing session. We propose a value of `1` if the expected time between two breakpoints is $O(1)$, and a value of `n*m` if the expected time is $O(nm)$, for example.

A data object can be traced only if its class implements a method `Display()` which should show relevant information encapsulated into the object. The object to be displayed should reveal some relevant information about the course of an algorithm, and the breakpoint should be placed right behind an update of this object.

Sometimes, it may be useful to have more than one breakpoint in order to trace different objects. For example, the Dinic maxflow algorithm contains two breakpoints `CT.Ping(Aux.Handle(),m)` and `CT.Ping(Handle(),m)`. The first one is placed between the construction of the layered auxiliary network `Aux` (which actually is displayed) and the augmentation step. The second breakpoint displays the flow network which has been augmented just before.

To use the tracing functionality and the graphical display, make sure that the `_TRACING_` pragma is defined in `config.h`. If this pragma is undefined, the breakpoints are still found, but only trace level 1 is available.

14.5.1 Trace Level Options

Level	Description
0	No output is written
1	A dot (`.`) is written to the standard output device
2	As before, but a method `Display()` is called by the constructors of classes which support this functionality. `Display()` either writes information in tabular form to the standard output device or graphical information to trace files which can viewed via Xfig or, GOBLET or the Tcl/Tk script `display`.
3	The method `Display()` is called at each tracing point.
4	The method `Display()` is called at each tracing point. Every output must be prompted by the user. Only useful for console applications.

14.5.2 Tracing Data Structures

The GOBLIN data structures discussed in Chapter 8 can be traced separately from general objects by setting the `traceData` flag. In that case, every elementary operation on the used data structures is subject to graphical tracing. The tracing mechanism is restricted to binary heaps, Fibonnacci heaps and disjoint set families. Stacks and queues do not produce any graphical output.

14.6 Graphical Display

Synopsis:

```
class goblinController
{
    int    displayMode;

    int    xShift;
    int    yShift;
    double xZoom;
    double yZoom;

    int    nodeSize;
    int    nodeStyle;
    int    nodeLabels;
    int    nodeColours;

    int    arcStyle;
    int    arcLabels;
    int    arcLabels2;
    int    arcLabels3;
    int    arrows;
    int    arrowSize;

    int    subgraph;
    int    predecessors;
    int    legenda;

    char*  nodeFormatting;
    char*  arcFormatting;
}
```

Every GOBLIN data object accepts a message `Display()` which may write some tracing information to the standard output device or to a trace file. Graph objects admit textual output which is generated by `TextDisplay()`, but also graphical output which is generated by `Display()`.

The latter method may call `TextDisplay()` again, but may also write trace files which can be read by GOBLET or the Xfig drawing tool. More explicitly, the output depends on the context variable `displayMode` which admits the alternatives shown in Table 14.6.1.

Trace files either consist of the graph object, its current potential solutions and context information in the GOBLIN native format, or an explicit canvas, depending on the value of `diplayMode`. See Section 14.5 for the file naming policy.

14.6.1 Display Mode Options

Mode	Description
0	Textual output
1	Graphical output: A `*.fig` file is written and Xfig is called.
2	Graphical output: A `*.tk` file is written and the tk script `display` is called.
3	Graphical output: A `*.gob` file is written which is processed by the GOBLET graph browser.

14.6.2 Export of Graphical Information

GOBLIN provides two export filters for graph layouts which are implemented by the classes `exp2tk` and `exp2xfig` respectively. As the names suggest, the first class generates some kind of Tcl/Tk scripts while the second class generates canvases for the xFig drawing tool. The xFig files can be converted to other graphics formats by using the tool `fig2dev` which usually forms part of the xFig distribution.

Note that the Tk files generated by `exp2tk` cannot be executed directly, but are input to the GOBLET graph browser. If you want to display a trace file on screen without using GOBLET, you can use the small Tk script `display` which does not require the complete installation of the GOSH interpreter. This script is called if `displayMode=2` is configured. See Section 18.7 for a description of the explicit canvas export methods.

14.6.3 Device Independent Layout

Synopsis:

```
class goblinDisplay
{
protected:

    char* predColour;
    char* inftyColour;

    long int width;
    long int height;

public:

    virtual void goblinDisplay(abstractMixedGraph&,float);

    long int        CanvasCX(TNode v);
    long int        CanvasCY(TNode v);
    long int        AlignedCX(TNode u,TNode v);
    long int        AlignedCY(TNode u,TNode v);

    goblinController &Configuration();

    virtual void    DisplayGraph();

    char*           ArcLabel(TArc,int);
    char*           NodeLabel(TNode);
    char*           ArcLegenda(int);
    char*           NodeLegenda(char*);
    char*           FixedColour(TNode);
    char*           SmoothColour(TNode);

    virtual void    DisplayArc(TArc) = 0;
    virtual void    DisplayNode(TNode) = 0;
    virtual void    DisplayLegenda(long int,long int,
                                   long int) = 0;
}
```

The class `goblinDisplay` organizes the device indepent layout of GOBLIN graph objects. This class is abstract, and instances are implicitly generated by the method `abstractMixedGraph::Display()` which also calls a virtual method `ConfigDisplay()`.

The `goblinDisplay` constructor generates a clone of the controller object. This clone, the configuration, can be accessed by the method `Configuration()`. The method `ConfigDisplay()` which is called with the display configuration makes some class-dependent changes of the layout parameters.

The class `goblinDisplay` provides some other ressources such as colours and the bounding box. This guarantees that the graphical output generated by the classes `exp2tk` and `exp2xfig` has the same appearance. The colours `predColour` and `inftyColour` are used for the display of predecessor arcs and the display of unreachable nodes respectively. The method `FixedColour()` provides an explicit scheme for the node and arc colours. `SmoothColour()` can be used if the fixed colours are not exhausted (20 colours are defined) of if adjacent colour indices should be displayed with similar colours. All returned strings are in 24 bit rgb format.

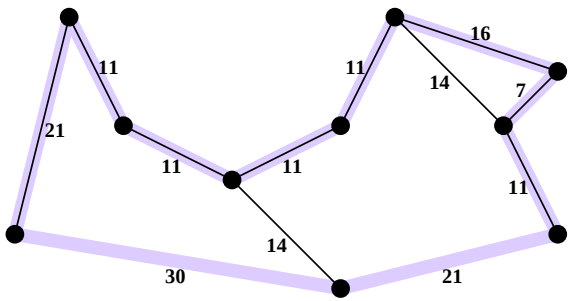


Figure 14.1: A Graph Layout with Subgraph and Predecessor Arcs

The class `goblinDisplay` also supplies with the node and the arc labels and the labels of the legenda. Note that two sets of arc labels can be displayed simultaniously, that is, the `arcLabels` or `arcLabels2` option must be passed to the methods `ArcLabel` and `ArcLegenda` explicitly. Figure 13.5 shows a graph layout with two sets of arc labels and a legenda.

The layout is based on the geometric information which is encapsulated into the graph object. Up to the layout of trees, GOBLIN does not compute any graph embeddings. If no embedding is present, the graph object cannot be viewed. The parameters `xShift`, `xZoom`, `yShift` and `yZoom` define an affine transformation of this embedding, and the transformed coordinates can be accessed by `CanvasCX` and `CanvasCY`.

The parameter `legenda` enables or disables the generation of a legenda. This legenda shows which node and arc labels are displayed in the layout. If `legenda==0`, no legenda is printed. Otherwise, the value of `legenda` specifies the space between the graph and the legenda.

The other layout parameters, together with their possible alternatives (defaults boldfaced), are listed in the following subsections. The layout of graph arcs deserves some further statements:

If the node `x=Align(a)` is undefined, the arc is drawn as a straight line between the geometrical embedding of the two end nodes `u` and `v`, and the potential label is aligned with the center of this line. Otherwise, the label is aligned with the transparent node `x`.

If `y=Interpolate(x)` is undefined, the arc is drawn as line between `u` and `v` again. Finally, if `y!=NoNode`, the interpolation points are `y` and the iterated points `y=Interpolate(y)` (until `y==NoNode` is reached). This list of points either defines a spline or a polyline object in the graph drawing.

The end nodes of a spline or polyline object are aligned with the graph node objects which depend on the `nodeStyle` and the `nodeSize` parameters. This is done by the methods `AlignedCX`, `AlignedCY` which shift the second nodes coordinates in the direction of the first graph node so that it becomes visible.

14.6.4 Formatting Arc and Node Labels

There are two ways how labels can be formatted: By setting the format strings `arcFormatting` and `nodeFormatting`, one can produce almost universal labels. If these strings are left blank, the node and arc labels are computed in the way known from earlier releases.

In the format strings, only two characters are special: The token `%1` refers to the current values of the context variables `arcLabels` or `nodeLabels` respectively. The tokens `#1`, `#2`, .. each represent one of the potential values of `arcLabels` and `nodeLabels` as listed in the Tables 14.6.5 and 14.6.6. As a simple and useful example,

```
arcFormatting = "$e_#7$"
```

would result in a set of arc labels `$e_1$`, `$e_2$`, .. in the GOBLIN canvas. If this canvas is imported to LaTeX, the labels $e_1, e_2, \ldots$ would result. If one sets

```
arcFormatting = "%1 [%2,%3]"
```

the `ConfigDiplay()` methods can determine which data shall be displayed by setting the variables `arcLabels`, `arcLabels2`, `arcLabels3`. The appearance is left to the user. In this example, a label `1.5 [1,3]` may result, a flow value with according capacity bounds.

The default layout of arc labels can be expressed as `%1/%2/%3` provided that none of the context variables `arcLabels`, `arcLabels2`, `arcLabels3` is unset. If `arcLabels` is unset, the equivalent format string is `%2/%3`.

14.6.5 Arc Display Options

Parameter	Value	Description
`arcStyle`	**0**	lines and polygons
	1	interpolated splines
	2	pipes (othogonal polygons)
`arcLabels`, `arcLabels2`, `arcLabels3`	**0**	no labels
	1	indices $0, 1, 2, ...$
	2	capacities
	3	subgraph (flow)
	4	length labels
	5	reduced length labels
	6	lower capacity bounds
	7	indices $1, 2, 3, ...$
`arrows`	**0**	aligned with node objects
	1	centered
`subgraph`	0	draw predecessor arcs only
	1	draw non-empty arcs only
	2	draw fractional arcs dashed
	3	draw empty, free, full arcs with different width
	4	draw all arcs uniformly
	5	different patterns for different subgraph labels
	6	display arc colours with a fixed colour pattern
	7	display arc colours with a dynamic colour pattern
`predecessors`	0	nothing special
	1	highlight predecessor arcs

14.6.6 Node Display Options

Parameter	Value	Description
nodeStyle	0	dots
	1	circles
	2	boxes
nodeLabels	0	no labels
	1	indices $0, 1, 2, ...$
	2	distance labels
	3	node potentials
	4	node colours
	5	node demands
	6	indices $1, 2, 3, ...$
nodeColours	0	no colours
	1	highlight nodes with finite distance labels
	2	node colours
	3	node demands
	4	node partition

Parameter	Default Value	Description
xShift	400	Shift on the ordinate
xZoom	150	Scaling of the ordiante
yShift	400	Shift on the abscissa
yZoom	150	Scaling of the abscissa
nodeSize	100	Diameter of graph nodes
arrowSize	300	Width of arrows
legenda	0	Separator for the legenda. If zero, no legenda is generated
nodeSep	10	Grid size for the graph nodes. Used in several graph layout methods
bendSep	5	Grid size for the bend nodes. Used in AutoArcAlignment()
fineSep	2	Grid size for node and arc labels. Currently used by the browser only

14.6.7 General Layout Options

14.7 Random Instance Generators

Synopsis:

```
class goblinController
{
    unsigned long   Rand(unsigned long);
    TFloat          UnsignedRand();
    TFloat          SignedRand();

    int     randMin;
    int     randMax;

    int     randUCap;
    int     randLCap;
    int     randLength;
    int     randGeometry;
    int     randParallels;
}
```

Many instance generators are prepared to generate random arc and node labels depending on which of the context flags `randLength`, `randUCap`, `randLCap` and `randGeometry` are set. The flag `randParallels` enables or disables the generation of parallel arcs.

Random labels can be generated by every graph constructor method and by every call to the method `InsertArc(TNode,TNode)` and `InsertNode()`. If you do not want to generate labels, keep in mind to unset the respective flags.

Edge length labels and node coordinates are generated by the method `SignedRand()` and arc capacities are genrated by `UnsignedRand()`. The numbers returned by `SignedRand()` are equally distributed integers from the interval `[randMin,..,randMax]`. The numbers returned by `UnsignedRand()` are in the same range if `randMin` is non-negative and from the interval `[0,..,randMax]` otherwise. A method call `Rand(k)` returns equally distributed integers from the interval `[0,..,k-1]`.

14.8 Runtime Configuration

Synopsis:

```
class goblinController
{
    void    Configure(int ParamCount,char* ParamStr[]);
}
```

Throughout this chapter, we have described the various configuration parameters which are available in GOBLIN. We finally need to explain how the controller objects are configured in practice:

If you call GOBLIN from within a C++ program, you can access all variables directly. If you call the library from a GOSH script, the GOBLIN context variables have a prefix `goblin`. For example, the tracing module is switched off by the command `set goblinTraceLevel 0`.

Sometimes, it is more efficient to call the method `Configure` which can change several parameters in one pass. This method is called with an array of strings each of which represents a variable name, value or a general option. One can set a context variable by adding a parameter which is composed from `-`, the variable name and the desired value.

The `Configure` method can be called from any C/C++ main routine and then passes the console input to GOBLIN. It can also be called from GOSH scripts. For example, the GOBLIN branch and bound module is enabled by the command `goblin configure -methSearch 2`.

The logging module admits some general settings which can be selected from the options `-silent`, `-report`, `-details` and `-debug` with increasing order of logging information.

Note that string context variables are generally read-only in the GOSH shell. Strings can be set with the `goblin configure` command only. Even from C++ level it is recommended to use this mechanism to avoid inconsistencies with the memory management.

Chapter 15

The Messenger

A messenger object manages the interaction of two threads of execution, namely a problem solver and a user interface. It implements methods to browse and edit the logging information, provides the possibility to interrupt the solver from the user interface, and it keeps control of the trace files.

Internally, the messenger is **thread safe**, that is, its data structures are locked by so-called **semaphores** to prevent different threads from accessing the data at the same time. The solver thread takes semaphores for a short period only, but the user interface may block the solver in order to read some volatile information and to make online changes.

The GOBLIN core library does not utilise semaphores at all. But for the GOSH shell the function pointers `solverStopSignalHandler`, `logEventHandler` and `traceEventHandler` essentially refer to messenger methods. Hence editing a graph which is subject to some computation can (but does not necessarily) corrupt the object and the solver process.

15.1 Problem Solver Management

Include file: `messenger.h`
Synopsis:

```
class goblinMessenger
{
    bool        SolverRunning();
    bool        SolverIdle();
    void        SolverSignalPending();
    void        SolverSignalStarted();
    void        SolverSignalStop();
    void        SolverSignalIdle();
}
```

The first goal of task communication is to force a solver to stop the optimization with a suboptimal solution. The complete schedule is as follows:

- The user interface checks if the flag `SolverIdle()` is true and, if so, calls `SolverSignalPending()` and then sets up a new thread of execution. At this stage, both `SolverIdle()` and `SolverRunning()` are false.
- The new thread calls `SolverSignalStarted()` and then the solver method. Now `SolverRunning()` is true.
- Occasionally, the user interface calls `SolverSignalStop()` so that the flag `SolverRunning()` becomes false again.
- The solver thread stops the computation before time (if the core library includes `solverStopSignalHandler` retrieval operations). The thread calls `SolverSignalIdle()` and then exits.

The messenger also allows to interrupt the solver temporarily, and this feature is described in Section 15.3.

15.2 The Message Queue

Include file: `messenger.h`
Synopsis:

```
class goblinMessenger
{
    void        MsgAppend(msgType,TModule,THandle,char*);

    void        MsgReset();
    bool        MsgEOF();
    bool        MsgVoid();
    void        MsgSkip();

    char*       MsgText();
    msgType     MsgClass();
    TModule     MsgModule();
    int         MsgLevel();
    THandle     MsgHandle();
}
```

The message queue buffers the most recent log entries generated by the solver thread. It is just large enough to fill a single screen, but does not occupy much system ressources. The log event handler also writes an incremental log file for later evaluations.

A new log entry is added to the queue by calling `MsgAppend()` which takes the class of information and an object handle as parameters. If the class is `MSG_APPEND`, the passed string is appended at the most recent log entry. Otherwise the oldest log entry on the queue is deleted and replaced by the new data.

The other functions are needed by the user interface for reading the messages which are currently queued. The method `MsgReset()` initializes a pointer to the oldest message in the queue. The method `MsgSkip()` then moves from one entry to another. If no more unread log entries exist, the flag `MsgEOF()` becomes true. The flag `MsgVoid()` indicates if no message is queued at all.

The following properties of the currently referenced message can be retrieved: The message text, the class of information, the module index and the object handle which all have been passed by the respective `LogEntry()` and `MsgAppend()` calls. The `MsgLevel()` is the context parameter `logLevel` at the time of writing the log entry.

In later releases, it will be possible to switch between the described online mode and a mode for importing the incremental log file into the messenger and editing.

15.3 Tracing

Include file: `messenger.h`
Synopsis:

```
class goblinMessenger
{
    void        TraceAppend(char*);
    void        TraceSemTake();

    char*       TraceFilename();
    bool        TraceEvent();
    void        TraceUnblock();
}
```

The tracing module has two ressources each of which is locked by an own semaphore: A list of trace file names and a flag which indicates if there are unhandled trace events and which can be read by calling `TraceEvent()`.

To the list of trace files, the solver thread declares every trace file name by calling `TraceAppend()`. This also sets the event flag. The solver thread

then calls `TraceSemTake()` before continuing its computations. The latter method returns only if the trace event is handled in the user interface.

The user interface handles a trace event as follows: It reads the last trace file name by calling `TraceFileName()` and then calls `TraceUnblock()` which effectively resets the event flag.

We mention that `TraceAppend()` and `TraceFileName()` allocate copies of the file name string and that the string returned by `TraceFileName()` must be disallocated by the calling context.

Some future work is at hand: It should be possible to read the complete list of trace file names, and this list should be editable in the same way as the message queue.

Chapter 16

Linear Programming Support

In order to allow development of this library beyond the scope of pure combinatorial algorithms, the author has decided to add some support for linear and integer programming techniques. This currently includes:

- An abstract class `goblinILPWrapper` which models mixed integer problems and the interface to the GOBLIN core library.
- A basic simplex code which applies to problems with a few 100s of variables but which does not utilize LU decomposition and sophisticated pricing techniques yet.
- An LP module entry point which is also designed as an abstract class `goblinILPModule` and which can be overloaded with plugins for other LP codes.
- File import and export filters which can supply to LP solvers other than the native simplex code.

Future releases may come up with a more efficient simplex code as well as with branch and cut techniques. Additionally, plugins for popular LP codes are desirable.

This chapter mainly discusses the method prototypes. Of course, all pure virtual messages must be implemented by any prospective LP wrapper. Some virtual functions provide default implementations which can be overloaded with more immediate code. Others are needed for user interaction only and hence provide dummy implementations which throw exceptions if called from the GOBLET browser.

16.1 Public Interface

16.1.1 Entry Point

Include file: `ilpWrapper.h`
Synopsis:

```
class goblinController
{
    void const*     pLPModule;
}

class goblinILPModule
{
    goblinILPModule();

    virtual goblinILPWrapper*
        NewInstance(TRestr,TVar,TIndex,TObjectSense,
                                goblinController&) = 0;

    virtual goblinILPWrapper*
        ReadInstance(char*,goblinController&) = 0;

    virtual goblinILPWrapper*
        Reinterprete(void*) = 0;

    virtual char*   Authors() = 0;
    virtual int     MajorRelease() = 0;
    virtual int     MinorRelease() = 0;
    virtual char*   PatchLevel() = 0;
    virtual char*   BuildDate() = 0;
    virtual char*   License() = 0;

    enum TLPOrientation {
        ROW_ORIENTED = 0,
        COLUMN_ORIENTED = 1
    };
    virtual TLPOrientation  Orientation() = 0;
}
```

The LP module is accessed by a context pointer to a `goblinILPModule` object. The purpose of this class is instance generation (`NewInstance()` and `ReadInstance()`), explicit runtime time information about the LP instances (`Reinterprete()`) plus some general module information.

The method `ReadInstance()` expects a filename as an input parameter. The input file format may differ among the various implementations. The method `NewInstance()` takes the desired number of restrictions, variables and non-zero matrix coefficients as well as the direction of optimization. Note that the `goblinILPModule` object is in the `goblinDefaultContext` but LP instances may be inserted into other contexts.

In order to generate LP instances from this abstract interface, one needs to cast back the entry pointer before problem instanciation:

Example:

```
...
goblinILPModule* X =
    (goblinILPModule*)goblinController::pLPModule;
goblinILPWrapper* myLP =
    X->ReadInstance(fileName,thisContext);
...
```

Accordingly, the registration of an LP module looks like

Example:

```
...
goblinILPModule* tmpPtr = new myLPModule();
goblinController::pLPModule = (void*)tmpPtr;
...
```

where `myLPModule` may denote some implementation of `goblinILPModule`. The extra assignment is needed to reconstruct a valid pointer later!

The parameter `Orientation()` is needed only for access to the current bases or tableaus of LP instances. Then a return value of `ROW_ORIENTED` indicates that restrictions are treated as artificial columns whereas `COLUMN_ORIENTED` indicates that variables also form restrictions. Row and column indices are partially orientation dependent!

16.1.2 LP Instance Retrieval Operations

Include file: `ilpWrapper.h`
Synopsis:

```
class goblinILPWrapper
{
    virtual TRestr      K();
    virtual TVar        L();
    virtual TIndex      NZ() = 0;

    virtual TFloat      Cost(TVar);
    virtual TFloat      URange(TVar);
    virtual TFloat      LRange(TVar);
    virtual TFloat      UBound(TRestr);
    virtual TFloat      LBound(TRestr);

    enum TVarType {
        VAR_FLOAT=0,
        VAR_INT=1,
        VAR_CANCELED=2
    };
    virtual TVarType    VarType(TVar);

    virtual TObjectSense    ObjectSense();

    virtual TFloat      Coeff(TRestr,TVar);
    virtual TVar        GetRow(TRestr,TVar*,double*);
    virtual TRestr      GetColumn(TVar,TRestr*,double*);

    virtual char*       VarLabel(TVar,TOwnership);
    virtual char*       RestrLabel(TRestr,TOwnership);
}
```

A `goblinILPWrapper` object models a general form **linear program**

$$\begin{aligned}&\textbf{minimize}\\&\quad c^T x\\&\textbf{subject to}\\&\quad a \le Ax \le b\\&\quad l \le x \le u\end{aligned}$$

with the dual form

$$\begin{aligned}&\textbf{maximize}\\&\quad a^T y_+ - b^T y_- + l^T z_+ - u^T z_+\\&\textbf{subject to}\\&\quad A^T(y_- - y_+) + z_- - z_+ = c\\&\quad y_+, y_-, z_+, z_- \ge 0\end{aligned}$$

Each of the vectors a, b, l and u may include symbolic infinite coefficients. In that case, the associated dual variables are fixed to zero implicitly. In the primal form, lower and upper bounds may coincide to represent equality restrictions respectively fixed variables. This mathematical description translates to the C++ model as follows:

- A `TRestr` value denotes a row index running from 0 to either `K()-1` or `K()+L()-1` depending on wether only **structural restrictions** or also **variable range restrictions** are valid arguments.
- A `TVar` value denotes a column index running from 0 to `L()-1` or `K()+L()-1` if **auxiliary variables** are also valid arguments (which then occupy the indices $0, \ldots, K() - 1$).
- The direction of optimization is determined by `ObjectSense()` with the possible values `MAXIMIZE`, `MINIMIZE` and `NO_OBJECTIVE`.
- The method `Cost()` represents the cost vector c.
- The methods `LRange()` and `URange()` represent the vectors l and u.
- The methods `LBound()` and `UBound()` represent the vectors a and b which are extended to the variable range restrictions in the obvious way.

- The matrix A is represented by the method `Coeff()` which is restricted to the structural restrictions. The number of non-zero matrix coefficients is obtained by `NZ()`.
- The `VarType()` of a variable is either `VAR_FLOAT`, `VAR_INT` (which indicate rational or integer variables) or `VAR_CANCELED` (which indicates deleted variables).
- The methods `VarLabel()` and `RestrLabel()` supply with variable names and symbolic row labels. Generally, rows and columns are referenced by indices rather than labels.

16.1.3 LP Instance Manipulation

Include file: `ilpWrapper.h`
Synopsis:

```
class goblinILPWrapper
{
    virtual TVar    AddVar(TFloat,TFloat,TFloat,TVarType);
    virtual TRestr  AddRestr(TFloat,TFloat);

    virtual void    DeleteVar(TVar);
    virtual void    DeleteRestr(TRestr);

    virtual void    SetURange(TVar,TFloat);
    virtual void    SetLRange(TVar,TFloat);
    virtual void    SetUBound(TRestr,TFloat);
    virtual void    SetLBound(TRestr,TFloat);
    virtual void    SetCost(TVar,TFloat);
    virtual void    SetVarType(TVar,TVarType);

    virtual void    SetVarLabel(TVar,char*,TOwnership);
    virtual void    SetRestrLabel(TRestr,char*,TOwnership);

    virtual void    SetObjectSense(TObjectSense);
    void            FlipObjectSense();

    virtual void    SetCoeff(TRestr,TVar,TFloat);
    virtual void    SetRow(TRestr,TVar,TVar*,double*)
    virtual void    SetColumn(TVar,TRestr,TRestr*,double*)

    virtual void    Resize(TRestr,TVar,TIndex);
    virtual void    Strip();
}
```

Every `goblinILPWrapper` object is instanciated with a couple of problem dimensions. These quantities are not the actual dimensions but rather the amount of reserved memory which can be adjusted dynamically by using `Resize(k,l,r)`. This concerns the number of rows k, the number of variables l and the number of non-zero matrix coefficients r. A `Strip()` operation performs a `Resize()` with the actual problem dimensions.

The obvious purpose of this functionality is to save memory reallocations. Any possible implementation class other than the native `goblinLPSolver` may ignore these implicit problem dimensions up to that adding rows and variables must be possible even if this requires a reallocation.

An `AddRestr()` operation sets a lower and an upper bound, an `AddVar()` operation sets the bounds, the cost coefficient and a variable type (in that order). The variable type must be `VAR_INT` or `VAR_FLOAT`. The matrix coefficients associated with a restriction and variable are initialized as zero and have to be set one by one using `SetCoeff()`.

Deleting rows (`DeleteRestr()`) or variables (`DeleteVar()`) may not change the remaining indices. It essentially marks the row or column as canceled. If deletions cannot be implemented otherwise, a delete operation may zero out rows and columns.

Calling `FlipObjectSense()` changes the object sense and inverts the objective vector. By that, optimum solutions are preserved but the objective value changes. Calling `SetObjectSense(MAXIMIZE)` or `SetObjectSense(MINIMIZE)` only changes the object sense whereas `SetObjectSense(NO_OBJECTIVE)` assigns a zero objective vector.

The implementation of the other methods is obvious. Specifying incompatible bounds should raise an exception. If setting a matrix coefficient corrupts the active basis, this should be checked by the next access to some basis dependent data only.

16.1.4 Basis Dependent Methods

Include file: `ilpWrapper.h`
Synopsis:

```
class goblinILPWrapper
{
    virtual void    ResetBasis();
    virtual bool    Initial();

    enum TRestrType {
        BASIC_LB=0,
        BASIC_UB=1,
        NON_BASIC=2,
        RESTR_CANCELED=3
    };
    virtual TRestrType  RestrType(TRestr);
    virtual TRestr      Index(TVar);
    virtual TRestr      RowIndex(TRestr);
    virtual TVar        RevIndex(TRestr);

    enum TLowerUpper {
        LOWER=0,
        UPPER=1
    };
    virtual void    SetRestrType(TRestr,TLowerUpper);
    virtual void    SetIndex(TRestr,TVar,TLowerUpper);
    virtual void    Pivot(TIndex,TIndex,TLowerUpper);

    virtual TFloat  X(TVar) throw(ERRange);
    virtual TFloat  Y(TRestr,TLowerUpper) throw(ERRange);

    virtual TFloat  ObjVal();
    virtual TFloat  Slack(TRestr,TLowerUpper);

    virtual TFloat  Tableau(TIndex,TIndex);
    virtual TFloat  BaseInverse(TIndex,TIndex);

    virtual bool    PrimalFeasible();
    virtual bool    DualFeasible();
}
```

All throughout lifetime, a `goblinILPWrapper` maintains some kind of basis. An initial basis is provided by the method `ResetBasis()`. This basis may consist of the variable range restrictions but other mechanisms are also possible. The flag `Initial()` indicates the state of the basis correspondingly.

The current row **basis** is accessed by the mappings `Index()` and `RevIndex()` which are inverse. The method `Index()` returns the **basis row** assigned with a given variable. More precisely, `RestrType(i)` is either `RESTR_CANCELED` or `NON_BASIC`, or `RevIndex(i)!=NoVar` is defined. In the latter case, the type is either `BASIC_LB` or `BASIC_UB`. In a column oriented implementation, the basis data can be manipulated as follows:

- The operation `SetIndex(i,j,tp)` results in `RevIndex(i)==j` and `Index(j)==i`. The passed type *tp* has to be either `LOWER` or `UPPER`. If previously `Index(k)==i`, then k must be matched elsewhere, ideally with the former `Index(j)`. It is not checked that the basis rows are linear independent after the operation!
- The operation `Pivot(i,j,tp)` has similar effects on the indices but requires that the entering row j is non-basic (Exception: $j == i$). The indexed rows must be linearly independent afterwards.
- Switching between `RestrType==BASIC_LB` and `BASIC_UB` is also achieved by `SetRestrType()`. Of course, this applies to basis rows only.

The methods `Index()` and `SetIndex()` are also mandatory for row oriented implementations. Additionally, the current (column) basis has to be determined by the method `RowIndex()` and the row and column indices have to be partially inverse: If `RowIndex(i)` is a structural variable,

`Index(RowIndex(i))==i` must hold. Both indices are completed by artificial variables and variable range restrictions respectively. Column oriented solver do not need to implement a `RowIndex()`!

The primal and dual solutions which are associated with the current basis are returned by the methods `X()` (only structural variables are handled) and `Y()` respectively. The violation of the primal restrictions is checked with the methods `Slack()` and `PrimalFeasible()`. The method `DualFeasible()` essentially checks the signs of the dual variable values. How the solutions are computed from the basis indices are implementation details.

The methods `Tableau()`, `BaseInverse()` and `Pivot()` have been added for didactic purposes. In order to get a unique interface for both column and row oriented solvers, all methods accept indices running from 0 to $K() + L() - 1$. For `Tableau()` and `Pivot()`, the first parameter specifies a basic index and the second parameter is a non-basic index. For `BaseInverse()`, the first parameter denotes a basic index running from 0 to $K() + L() - 1$ and the second parameter denotes a row index ranged in $0, \ldots, K() - 1$ or a column index ranged in $0, \ldots, L() - 1$ respectively!

Finally, it must be mentioned how the basis changes if the problem definition changes. If the right-hand sides or the cost coefficients are modified, the basis remains intact. If some matrix coefficients are modified, the indexed rows may become linearly dependent, but this may be detected by the next pivoting step only.

If a variable is added, a basis row must be assigned immediately, ideally the variable range restriction (this is always feasible). A new structural restriction does not affect the (row) basis and the primal solution and slacks of the existing restrictions (although the solution may become primally infeasible and the indices must be recomputed). It shall not possible to delete a row in the current basis. The deletion of a variable must mark the matched basis row non-basic.

16.1.5 Problem Transformations

Include file: `ilpWrapper.h`
Synopsis:

```
class goblinILPWrapper
{
    goblinILPWrapper*   Clone();
    goblinILPWrapper*   DualForm();
    goblinILPWrapper*   StandardForm();
    goblinILPWrapper*   CanonicalForm();
}
```

The LP interface supports the well-known transformations of linear programs. All methods do not modify the addressed object but return a new LP instance of the requested form:

- If the original LP instance is a standard or canonical form, the `DualForm()` flips the role of rows and variables but does not introduce new items. Generally, lower and upper bounds are replaced by two variables or two rows, and the object sense is reverted.

- The `CanonicalForm()` replaces all variable range restrictions by structural restrictions and all equality restrictions by a pair of inequalities. Computing the canonical form of a canonical form, does not change anything. Canonical forms are maximization problems.

- The `StandardForm()` fills inequality restrictions with slack variables and substitutes variables with non-trivial bounds. Computing the standard form of a standard form, does not change anything. Standard forms are minimization problems.

- The `Clone()` is a plain copy of the addressed MIP object. It can be used for explicit manipulation without changing the original LP.

All transformations preserve the optimum objective value.

Generally, the new variable [row] names can clash with original names. To be safe, the original names should consist of a letter followed by digits. For example, you can use the internal naming scheme.

16.1.6 Solving Problems

Include file: `ilpWrapper.h`
Synopsis:

```
class goblinILPWrapper
{
    enum TSimplexMethod {
        SIMPLEX_AUTO=0,
        SIMPLEX_PRIMAL=1,
        SIMPLEX_DUAL=2
    };
    enum TStartBasis {
        START_AUTO=0,
        START_LRANGE=1,
        START_CURRENT=2
    };

    virtual TFloat  SolveLP();
    virtual TFloat  SolvePrimal();
    virtual TFloat  SolveDual();
    virtual bool    StartPrimal();
    virtual bool    StartDual();
}
```

This is the most straightforward part of the LP interface description: The entry point `SolveLP()` calls one of the methods `SolvePrimal()` and `SolveDual()` based on the value of the context variable `methLP`. There is a default implementation provided for `SolveLP()` which can be used from later plugins in order to support the GOBLET browser messaging. The relationship between the options `methLP`, `methLPStart` and the types `TSimplexMethod`, `TStartBasis` is the obvious one.

The methods `StartPrimal()` and `StartDual()` can be used to determine feasible rather than optimal solutions.

Variable	Value	Description
`methLP`	**0**	Automatic selection
	1	Primal Simplex
	2	Dual simplex method
`methLPStart`	**0**	Automatic selection
	1	Start with lower bounds
	2	Start with current basis

Table 16.1: LP Solver Options

16.1.7 File I/O

Include file: `ilpWrapper.h`
Synopsis:

```
class goblinILPWrapper
{
    enum TLPFormat {
        MPS_FORMAT=0,
        LP_FORMAT=1,
        MPS_CPLEX=2,
        BAS_CPLEX=3,
        BAS_GOBLIN=4
    };

    void    Write(char*,TOption = 0);
    void    Write(char*,TLPFormat,TOption = 0);
    void    WriteMPSFile(char*,TLPFormat = MPS_CPLEX);
    void    WriteMPSFile(ofstream&,TLPFormat = MPS_CPLEX);
    void    WriteBASFile(char*,TLPFormat = BAS_CPLEX);
    void    WriteBASFile(ofstream&,TLPFormat = BAS_CPLEX);

    void    ReadMPSFile(char*);
    void    ReadMPSFile(ifstream&);
    void    ReadBASFile(char*);
    void    ReadBASFile(ifstream&);
}
```

The LP file interface supports the standard MPS format and the CPLEX MPS variant for both reading and writing files, and the CPLEX LP format for writing files only. The output methods work implementation independent, the input MPS method requires a void LP to run and, by that, a default constructor in the LP plugin. Additionally, one can read and write MPS basis files. Again, reading a basis requires that the LP plugin supports setting a special basis.

The native LP file format generated by the method `Write(char*,TOption)` consists of a certain header part, an MPS problem description and an MPS basis file. In order to implement this efficiently, all file I/O methods exist in two versions, writing to or reading from a file specified either by the file name or an open stream. A more detailed specification of the native format can be found in Section 18.5. The LP format generator is discussed next.

16.1.8 Text Display

Include file: `ilpWrapper.h`
Synopsis:

```
class goblinILPWrapper
{
    enum TDisplayOpt {
        DISPLAY_OBJECTIVE = 1,
        DISPLAY_RESTRICTIONS = 2,
        DISPLAY_BOUNDS = 4,
        DISPLAY_INTEGERS = 8,
        DISPLAY_FIXED = 16,
        DISPLAY_PRIMAL = 32,
        DISPLAY_DUAL = 64,
        DISPLAY_SLACKS = 128,
        DISPLAY_BASIS = 256,
        DISPLAY_TABLEAU = 512,
        DISPLAY_INVERSE = 1024
    };

    void WriteLPNaive(char*,TDisplayOpt = 0);
}
```

There is an implementation independent layout method `WriteLPNaive()` which can display the complete problem description and tableau data. This information is grouped into several sections rather than filled into a single table. The calling parameters are an output file name and a bit field which is composed from the following flags:

- `DISPLAY_OBJECTIVE`: Write the direction of optimization and the linear objective function. Variables with zero coefficients are omitted.
- `DISPLAY_RESTRICTIONS`: Write the structural restrictions. Fields with zero coefficients are left blank. Displayed are either equations or inequalities with one or two right-hand sides.
- `DISPLAY_BOUNDS`: Write the variable range restrictions. Free variables are not listed. Non-negative, non-positive and binary variables are grouped together. The remaining variables are displayed by equations or inequalities with one or two right-hand sides.
- `DISPLAY_INTEGERS`: Write the list of integer variables.
- `DISPLAY_FIXED`: Write the list of fixed variables.
- `DISPLAY_PRIMAL`: Write the variable values. Zero values are omitted.
- `DISPLAY_DUAL`: Write the dual variable values associated with the structural and the range restrictions. Lower and upper bound restrictions are grouped together. Zero values are omitted, especially those of unbounded restrictions.
- `DISPLAY_SLACKS`: Write the primal slacks. Lower and upper bounds are grouped together. Unbounded restrictions and zero slacks are not listed.
- `DISPLAY_BASIS`: Write the mapping from variables to basis restrictions.
- `DISPLAY_TABLEAU`: Write the transposed tableau matrix where the basis column are omitted. Zero matrix entries are not displayed.
- `DISPLAY_INVERSE`: Write the transposed inverse of the basis matrix. Zero matrix entries are not displayed.

If no display option or a zero value is specified, the output is in CPLEX LP format. This essentially consists of the first three listed sections.

All sections list variable and restriction labels rather than indices. The tableau and basis inverse output is always formatted (take care with large

scale problems). If the width does not exeed 120 characters, the objective function and the structural restrictions are aligned together. The remaining sections are written in blocks of 5 or 10 entries.

In this format, a given basis is primally feasible if all displayed slacks are non-negative. Optimality can be checked with the dual variable values which must have the correct sign (depending on the direction of optimization and differing for lower and upper bound restrictions).

16.2 Native LP Solver

Include file: `lpSolver.h`
Synopsis:

```
class goblinLPSolver
{
private:

    bool    baseInitial;
    bool    baseValid;
    bool    dataValid;

    void    DefaultBasisInverse();
    void    EvaluateBasis();
    void    BasisUpdate(TRestr,TVar);
    void    SolutionUpdate();

    void    PrimallyFeasibleBasis();
    TVar    PricePrimal();
    TRestr  QTestPrimal(TVar);

    void    DuallyFeasibleBasis();
    TRestr  PriceDual();
    TVar    QTestDual(TRestr);

public:

    void    Pivot(TRestr,TVar,TLowerUpper);

}
```

The native LP solver is preliminary, and currently only a very basic simplex code is available. For this reason, a detailed documentation of pricing techniques, ratio tests and the used data structures is postponed. We give a few remarks about the basis update strategies so far and about some flags used internally:

- The flag `baseInitial` is equivalent with the method `Initial()`. It is set by constructors and by `ResetBasis()` operations. It indicates the basis consisting of the lower variable bounds. The flag is cleared by every `SetIndex()` operation.
- The flag `baseValid` indicates if a basis inverse matrix exists and if it is up to date with the basis indices and the coefficient matrix. It is set by `DefaultBasisInverse()`, `EvaluateBasis()` and `BasisUpdate()`. The flag is cleared initially and by `ResetBasis()`, `SetIndex()` and `SetCoeff()` operations.
- The flag `dataValid` indicates if the basic solutions are up to date with the problem definiton and the basis inverse matrix. It set by calls to `DefaultBasisInverse()` and `SolutionUpdate()`, and cleared whenever the problem is modified or the basis indices change.

The method `EvaluateBasis()` computes the basis inverse matrix and a pair of basis solutions from scratch. This operation takes $O(l^3)$ time and is used only if optimization is started from a given basis without knowing the initial basis inverse, especially if `SetIndex()` has been called explicitly.

A `Pivot()` operation also calls `SetIndex()` but then updates the basis inverse by a subsequent call to `BasisUpdate()`. The update of the basic solutions is delayed until values are actually requested.

16.3 GLPK Wrapper

Include file: `glpkWrapper.h`
Synopsis:

```
class goblinGLPKWrapper
```

There are some conceptual differences between GLPK and the GOBLIN native code:

- In GLPK, cost coefficients can be associated with restrictions which are considered auxiliary variables.
- In GLPK, efficient access to the constraint matrix is provided by row and column operations.
- GLPK is distributed under the terms of the GNU public licence.

Chapter 17

Ressource Management

17.1 Memory Management

Include files: `globals.h`, `goblinController.h`
Synopsis:

```
long unsigned goblinHeapSize;
long unsigned goblinMaxSize;
long unsigned goblinNFragments;
long unsigned goblinNAllocs;
long unsigned goblinNObjects;

void* operator new(size_t size);
void* operator new[](size_t size);
void* GoblinRealloc(void* p,size_t size);
void operator delete(void *p);
void operator delete[](void *p);

class goblinAbstractObject
{
    virtual unsigned long   Size() = 0;
};

class goblinController
{
    unsigned long   Size();
};
```

The GOBLIN memory management keeps track of all changes of the dynamic memory (heap) referenced by the data objects. Other than in previous releases, the counters are global rather than context relative. The counters inform about the current heap size (`goblinHeapSize`), the maximum heap size (`goblinMaxSize`), the current number of data objects (`goblinNObjects`), the current number of memory fragments (`goblinNFragments`) and the total number of memory allocations (`goblinNAllocs`).

To this end, the operators `new`, `new[]`, `delete` and `delete[]` have been overwritten. If conflicts with other C++ modules arise, the entire functionality can be turned off at compile time via the pragma `_HEAP_MON_`. The function `GoblinRealloc()` does the same as the C function, but a new name has been chosen to separate from C memory management.

Note that a block of memory which was allocated with the default implementation of `new()` cannot be disallocated with the GOBLIN version of `delete()`. Do also take care that `new[]()` and `delete[]()` are matching for sake of later redesigns.

If desired, the calling class method has to provide meaningful logging information about allocation and disallocation of implicit objects (objects which are not GOBLIN data objects). A typical sequence of statements is like follows:

Example:

```
...
thisArray = new TFloat[100];
LogEntry(LOG_MEM,Handle(),"...Array allocated!");
```

```
...
thisArray = TFloat(GoblinRealloc(sizeof(TFloat)*200);
LogEntry(LOG_MEM,Handle(),"...Array resized!");
...
delete[] thisArray;
LogEntry(LOG_MEM,Handle(),"...Array disallocated!");
...
```

Independently from the described heap information one can retrieve the size of any object by calling `Size()`. The returned amount is the actual object size via `sizeof()` plus the amount of heap memory referenced by this object (other than GOBLIN data objects). Calling `Size()` for an object controller would return its actual object size plus the size of all managed data objects.

17.2 Timers

Include files: `timers.h`
Synopsis:

```
class goblinTimer
{
    goblinTimer(goblinTimer** = NULL);

    void        Reset();
    bool        Enable();
    bool        Disable();

    double      AccTime();
    double      AvTime();
    double      MaxTime();
    double      MinTime();
    double      PrevTime();
    bool        Enabled();
};
```

The class `goblinTimer` provides timer objects to keep track of roundtrip times (`PrevTime()`), accumulated times (`AccTime()`), minimum (`MinTime()`), maximum (`MaxTime()`) and average (`AvTime()`) roundtrip times for a special unit of code.

Timer are started by the method `Enable()` and stopped by `Disable()`. A `Reset()` operation clears the timer statistics and also stops the timer. One can check if the timer is currently running by calling `Enabled()`.

If nested starts and stops of the same timer occur, the timer object maintains the nesting depth and effectively stops only if all starts are matched by stop operations.

The compilation of the entire timer functionality is suppressed if the pragma `_TIMERS_` is unset.

17.2.1 Basic and Full Featured Timers

Include files: `timers.h`
Synopsis:

```
class goblinTimer
{
    double      ChildTime(TTimer);

    bool        FullInfo();
};
```

A timer can report about explicit starts and stops but also about relative running times of other timers (**child timers**). For this goal, a pointer to a list of **global timers** must be passed to the constructor method.

Whenever a timer is started, all child running times are reset. Since several timers can be active at a time, the child times do not sum up to the parent timer value.

if no or a `NULL` pointer is passed to the constructor method, a **basic timer** is instanciated. Such timers do not keep track of nested timer starts and stops. A given timer is basic if `FullInfo()` returns `false`.

17.2.2 Global Timers

Include files: `globals.h`, `goblinController.h`
Synopsis:

```
enum TTimer {..., NoTimer};

struct TTimerStruct {
    char*           timerName;
    bool            fullFeatured;
};

const TTimerStruct listOfTimers[];

class goblinController
{
    pGoblinTimer*   globalTimer;
};
```

There is a list of global timers, declared by the enum index type `TTimer` and the global array `listOfTimers`. From this structural information, every controller object instanciates its own timer table. This table can be addressed by the pointer `globalTimer` and the enum index values.

Global timers are intended to split the code into functional units whereas the source code modules discussed later represent special authorship. Several modules may share a global timer.

Some basic global timers are utilized explicitly by the high-level data structures and the file interface whereas the other global timers are (de)activated by `OpenFold()` and `CloseFold()` operations implicitly.

If the context flag `logTimers` is set, every `CloseFold()` operation files the complete timer status including child times. Zero timer values are not displayed.

17.2.3 Lower and Upper Problem Bounds

Include files: `timers.h`, `dataObject.h`
Synopsis:

```
class goblinTimer
{
    bool    SetLowerBound(TFloat);
    bool    SetUpperBound(TFloat);

    TFloat  LowerBound();
    TFloat  UpperBound();
};

class goblinDataObject
{
    void    SetLowerBound(TTimer,TFloat);
    void    SetUpperBound(TTimer,TFloat);
    void    SetBounds(TTimer,TFloat,TFloat);

    TFloat  LowerBound(TTimer);
    TFloat  UpperBound(TTimer);
};
```

With every timer, a pair of problem bounds is associated which can be manipulated in the obvious way. For global timers, an additional wrapper exists which simplifies the access from data object methods.

If the context flag `logGaps` is set, every `SetBounds()` operation which strictly decreases the duality gap writes some logging information.

17.3 Source Code Modules

Include files: `globals.h`
Synopsis:

```
enum TModule {..., NoModule};

struct TModuleStruct {
    char*           moduleName;
    TTimer          moduleTimer;
    TAuthor         implementor1;
    TAuthor         implementor2;
    char*           encodingDate;
    char*           revisionDate;
    TBibliography   originalReference;
    TBibliography   authorsReference;
    TBibliography   textBook;
};

const TModuleStruct listOfModules[];

class goblinDataObject
{
    void    OpenFold();
    void    CloseFold();

    void    OpenFold(TModule,TOption = 0);
    void    CloseFold(TModule,TOption = 0);
};
```

As mentioned before, a **code module** denotes a specific implementation rather than a functional unit. Source code is assigned to a module `modSample` by the method calls `OpenFold(modSample,opt)` and `CloseFold(modSample,opt)` which must match each other.

By default, folds signal indentations to the messenger. To suppress such indentations, one can pass an optional parameter `NO_INDENT`. Conversely, if the module context has already been set, additional indentations can be forced by calling `OpenFold()` and `CloseFold()` without any parameters.

Opening a fold enables the associated timer. If the timer was not already running, the problem bounds are also reset.

17.3.1 Authorship

Include files: `globals.h`
Synopsis:

```
enum TAuthor {..., NoAuthor};

struct TAuthorStruct {
    char*           name;
    char*           affiliation;
    char*           e_mail;
};

const TAuthorStruct listOfAuthors[];
```

17.3.2 Bibliography Data Base

Include files: `globals.h`
Synopsis:

```
enum TBibliography {..., NoBibliography};

struct TBibliographyStruct {
    char*           refKey;
    char*           authors;
    char*           title;
    char*           type;
```

```
        char*           collection;
        char*           editors;
        int             volume;
        char*           publisher;
        int             year;
    }

    const TBibliographyStruct listOfReferences[];
```

17.4 Progress Measurement

A description of this functionality is delayed until the interface has become stable.

Chapter 18

Persistency

18.1 Export of Data Objects

Include file: `fileExport.h`
Synopsis:

```
class goblinExport
{
    goblinExport(char*,goblinController &
                            = goblinDefaultContext);

    void        StartTuple(char*,char,char = 0);
    void        StartTuple(unsigned long,char,char = 0);
    void        EndTuple();
    template <typename T>
                void MakeItem(T value,char length);
    void        MakeNoItem(char);
}
```

This class supports file export of data objects into a hierarchical format. In this format, a data object is essentially a tree. The non-leaf nodes of this tree are called **tuples**; they start and end with a parenthesis. Between these two parenthesis, a label and the child nodes are listed.

All child nodes must be of the same type, that is, either they are all tuples or they are all numbers of a certain type. Needless to say that this simple concept does not only work for graph objects, but is adequate for any data object which essentially consists of vectors.

Every `StartTuple()` operation must be matched by an `EndTuple()` operation and vice versa. These operations write parenthesis `(` and `)` respectively. It is checked if the number of parentheses resolve in the end, and if there are unmatched opening parenthesis intermediately.

The first parameter of a `StartTuple(label,type)` call is a header information which is written, either a string (which should not contain any white spaces) or an integer number (which represents some kind of index).

The second parameter is the **type** of the tuple. If zero, the tuple is a structured object, and the next operation must be another call to `StartTuple()`. Otherwise, the tuple represents a vector or a constant. If the type k is one, the entire vector is written to a single line. Finally, if $k > 1$, the entries are written in batches of k numbers.

The third optional parameter denotes the maximum length of an entry if written to file. This parameter is needed for formatting the output only.

A call `MakeItem<T>(x,l)` writes a value x of type T into a field of width l. In case of floats, one can use the context method `SetExternalPrecision()` to control the formatting. Finally, `MakeNoItem(l)` writes an asterisk `*` which represents undefined values. All items are aligned to the right-hand side.

18.2 Import of General Data Objects

Include file: `fileImport.h`
Synopsis:

```
enum TBaseType {
    TYPE_NODE_INDEX,    TYPE_ARC_INDEX,     TYPE_FLOAT_VALUE,
```

```
    TYPE_CAP_VALUE,     TYPE_INDEX,         TYPE_ORIENTATION,
    TYPE_INT,           TYPE_BOOL
};

enum TArrayDim {
    DIM_GRAPH_NODES,    DIM_GRAPH_ARCS,     DIM_ARCS_TWICE,
    DIM_ALL_NODES,      DIM_LAYOUT_NODES,   DIM_SINGLETON
};

class goblinImport
{
    goblinImport(char*,goblinController&
                           = goblinDefaultContext);

    char*       Scan(char* = NULL);
    bool        Seek(char*);
    bool        Head();
    bool        Tail();
    bool        Eof();

    TNode*          GetTNodeTuple(unsigned long);
    TArc*           GetTArcTuple(unsigned long);
    TCap*           GetTCapTuple(unsigned long);
    TFloat*         GetTFloatTuple(unsigned long);
    TIndex*         GetTIndexTuple(unsigned long);
    char*           GetCharTuple(unsigned long);
    bool            Constant();
    unsigned long   Length();

    size_t      AllocateTuple(TBaseType,TArrayDim);
    void        ReadTupleValues(TBaseType,size_t);

    template <class TEntry>     TEntry* GetTuple();

    template <class TToken>     TToken ReadTuple(
            const TTokenTable listOfParameters[],
            TToken endToken,TToken undefToken)
    }
```

Only a few comments are needed regarded the import of data objects: The most basic method is `Scan()` which reads a string separated by white spaces and parentheses, called **token** in what follows. Note that an opening parenthesis may not be followed by a white space. If string argument is passed to `Scan()`, the method checks if this tring equals the scanned token and throws an `ERParse` exception otherwise. If no argument is passed, a pointer to the read token is returned.

The method `Seek()` scans the input, searching for the string which has been passed as argument. It returns `true` if the string has been found in the context, and `false` otherwise.

The methods `Head()` and `Tail()` can be used to determine the position of the last read token within its tuple. Accordingly, `EOF()` detects the end of an object definition which should coincide with the file end.

For each base type used in GOBLIN, a special method exists which reads a complete tuple. These methods take a parameter which specifies the desired length of the tuple, and the input is accepted if either the actual length matches this parameter value or if the actual length is one. This fact is used to read constant graph labelings more economically.

The method `Length()` returns the length of the last read tuple and, accordingly, `Constant()` decides whether the last read tuple has length 1.

18.3 Import of Graph Objects

Include file: `fileImport.h`
Synopsis:

```
class goblinImport
```

```
{
    TOptDefTokens       ReadDefPar();
    TOptLayoutTokens    ReadLayoutPar();
    TOptRegTokens       ReadRegister();
}
```

18.4 File Format for Graph Objects

The general file format for graph objects is as follows:

< *graph object* >:=

(< *class label* >
< *definition* >
< *objectives* >
< *geometry* >
< *layout* >
< *solutions* >
< *configuration* >
)[CR/LF]

where

< *class label* >:=

`graph` | `dense_graph` | `digraph` | `dense_digraph` |
`bigraph` | `dense_bigraph` | `balanced_fnw` | `mixed_graph`

Usually, the information associated with some node or arc is stored by a file record. Instead of this, GOBLIN stores **vectors**, that are lists of numbers which represent a specific node or arc labeling. Many fields in the file format can be filled either with such a vector or with a single value which then denotes a constant labeling.

This may be inconvenient for reading and editing the files by hand, but a lot of information is immaterial for concrete problems. In that sense, the GOBLIN file format keeps the file sizes small. Some items merely keep place for future extensions of GOBLIN.

In what follows, a term $< arc >^x$ can be replaced either by a single arc index or by a list of arc indices with exact length x. Corresponding terms are used for node indices, booleans, capacities and floating numbers.

18.4.1 Definition

$< definition >$:=

```
(definition
 (nodes  < n1 > < n2 > < n3 >)
 [(arcs  < m = number of arcs >)
 (incidences
  (inc0          < arcs incident with node 0 >)
  (inc1          < arcs incident with node 1 >)
  .
  .
  (inc< n − 1 >   < arcs incident with node n − 1 >)
 )]
 (ucap          < capacity >^m)
 (lcap          < capacity >^m)
 (demand        < capacity >^n)
 (directed      < boolean >^m)
)
```

The definition part essentially describes the feasibility region of a network programming problem. For concrete classes, the following items can be omitted:

- For bipartite graphs, the cardinality of both partitions is specified by the numbers $n1$ and $n2$, and the total number of nodes is $n := n1+n2$. Otherwise, the number of graph nodes is $n := n1$. The number $n3$ denotes interpolation points which are needed for the graph layout sometimes. In what follows, some vectors have length $n^* := n + n3$.
- Incidence lists are specified for sparse graphs only. In dense graphs, the incidences are determined by the arc indices implicitly.
- A list of arc directions are specified for mixed graphs only. Otherwise, this field is filled with a constant 0 or 1.

The incidence lists must be disjoint and cover the integers $0, 1, \ldots, 2m - 2, 2m - 1$. The node whose incidence list contains the integer a is the start node of the arc `a`, and the node whose incidence list contains the integer `a^1` is the end node. As mentioned earlier, an even index $2i$ denotes a forward arc, $2i + 1$ is the corresponding backward arc.

18.4.2 Objectives

$< objectives >$:=

```
(objectives
 (commodities  < c = number of commodities >)
 [(bound        < float >^c)
 (length
  (comm0        < float >^m)
  (comm1        < float >^m)
  .
  .
  (comm< c − 1 >      < float >^m)
 )]
)
```

An **objective function** is a cost vector on the arc set of a graph, essentially a set of arc length labels. A **network programming problem with side constraints** asks for a certain subgraph such that for each objective the total length does not exeed a respective bound or which minimizes the maximal objective.

This part has been added to support such problems at least by an adequate file format. So far, no algorithms and no internal data structures for problems with multiple objectives are available in GOBLIN, and this part should look like

```
(objectives
 (commodities 1)
 (bound       *)
 (length
  (comm0       < float >^m)
 )
)
```

18.4.3 Geometry

< *geometry* >:=

```
(geometry
 (metrics      < type of metrics >)
 (dim          < d = dimension of the embedding>)
 [(coordinates
  (axis0        < float >^n*)
  (axis1        < float >^n*)
  .
  .
  (axis< d − 1 >     < float >^n*)
 )]
)
```

This information becomes important if one needs to solve geometrical problems, but is also used for the graphical display.

The field < *type of metrics* > denotes the method by which length labels are computed internally and overwrites the context variable `methGeometry`. If this parameter is zero, the length labels are specified in the objectives part. Otherwise, GOBLIN takes the geometric embedding specified here and computes the distances with respect to the specified metric.

In the current release, the dimension d must be either 0 or 2, that is, a graph either has a plane embedding or is not embedded at all.

18.4.4 Layout

< *layout* >:=

```
(layout
 (model        < layout model >)
 (align        < node >^m)
 (thread       < node >^n*)
 (exteriorArc  < arc >)
)
```

This information is needed only for the graphical display. Reading the value of < *layout model* > overwrites the corresponding context variable. Even more, `SetLayoutParameters()` is called with this value and effectively sets all default values for this layout model. The configuration part is used to customize the layout model.

If you do not want any graphical output, or if the pure geometric embedding is satisfactory, the dimension $n3$ should be zero, and the layout part should look as follows:

```
(layout
 (model        6)
 (align        *)
 (thread       *)
)
```

The displayed order of tuples is realized by the file export interface. When reading from file, the order is immaterial and tuples can be omitted instead of passing default values.

18.4.5 Potential Solutions

< *solutions* >:=

```
(solutions
 (label        < float >^n)
 (predecessor  < arc >^n)
 (subgraph     < float >^m)
 (potential    < float >^n)
 (nodeColour   < node >^n)
```

(edgeColour $< arc >^{2m}$)
)

This part keeps the computational results and corresponds to the internal data structures discussed in Chapter 13. If an object is imported from file, the internal data structures are initialized with the external data. This can be used for post-optimization procedures.

Some care is recommended when a graph object is exported: All internal data structures which are not needed any longer should be deleted explicitly before file export. If possible, subgraphs should be converted to predecessor labels. There are methods available for the conversion of paths, trees and matchings, see Section 11.2.1 for details.

The displayed order of tuples is realized by the file export interface. When reading from file, the order is immaterial and tuples can be omitted instead of passing default values.

18.4.6 Configuration

$< configuration >$:=

(configure
{-$< context\ parameter >$ $< integer >$}*
)

This part may keep any kind of context parameters: logging, method selection as well as layout information. When a graph object is imported from file, the method `goblinImport::ReadConfiguration()` is called, and the information from file overwrites the respective context variables. The method

```
goblinExport::WriteConfiguration(goblinController&,
                                 TConfig = CONF_DIFF)
```

allows to write the configuration of the specified controller object to file. If the optional parameter is `CONF_DIFF`, the values of the configuration parameters are compared with the default context, and only differing values are written to the output file. Alternatively, `CONF_FULL` can be specified to write a complete set of parameters. During graph export, the method

```
goblinExport::WriteConfiguration(goblinDataObject*)
```

is used which calls the graph method `ConfigDisplay()` and then writes the resulting configuration.

18.5 File Format for Linear Programs

The native file format for linear programs and mixed integer problems consists of a GOBLIN specific header followed by the problem definition and some basis:

$< mip\ object >$:=

(mixed_integer
(rows $< integer >$)
(columns $< integer >$)
(size $< integer >$)
(pivot { * | $< integer >$ $< integer >$ {0|1}})
(rowvis $< boolean >^k$)
(colvis $< boolean >^l$)
$< configuration >$
)

$< mps\ problem >$

$< mps\ basis >$

Here $< mps\ problem >$ denotes the full description of a mixed integer linear program in CPLEX MPS format, and $< mps\ basis >$ denotes a respective basis. The fields in the header are as follows:

- `rows` specifies the number k of structural restrictions.
- `columns` specifies the number l of variables.
- `size` denotes the number of non-zero matrix coefficients.
- `pivot` specifies a potential pivot element, listing the row index, the column index and if the lower (0) or upper bound (1) is achieved after the pivot step. Alternatively, an asterisk indicates that no pivot element is defined.
- `rowviz` and `colviz` are currently not in use and must be set to 1 constantly.
- The $< configuration >$ part is formatted as in graph objects files.

18.6 Canvas and Text Form

Include file: `abstractMixedGraph.h`
Synopsis:

```
class goblinDataObject
{

    void    Export2XFig(char*);
    void    Export2Tk(char*);
    void    Export2Ascii(char*);

};
```

In principle, every data object can be exported into some user readable form. The method prototypes are listed above and are, so far, implemented for graph objects (canvas and text forms) and mixed integer problems (only text form).

The text form provided by `Export2Ascii()` is used by the GOBLET browser. The exact format for mixed integer problems is described in Section 16.1.8. For mixed graphs, a node oriented format is generated which lists the node attributes and all node incidences. An incidence record possibly starts with a mark `P` to indicate the predecessor arc and with a mark `B` to indicate backward arcs. All constant arc labellings are listed at the end of the file.

Graph can also be written to some canvas formats. The method `Export2Tk()` generates a Tcl/Tk canvas and is needed by the GOBLET browser again. The method `Export2XFig()` generates a canvas format which can be processed by the `xfig` drawing program and the `transfig` filter software. By the latter tool, one can obtain a series of other canvas and bitmap formats. More details about the GOBLIN graph layout functionality can be found in Section 14.6.

18.7 Support of Standard File Formats

We have already mentioned that MPS file can be read and written from C++ level. The GOBLIN library does not support additional graph and lp formats directly, but there are GOSH scripts `import.tk` and `export.tk` which can be used to read and write DIMACS and TSPLIB problems. Solutions can be exported, but not imported into the GOSH interpreter. For example, you may input at the GOSH prompt the following:

Example:

```
source tcl/import.tk
goblinImport G sample.tsp tsp
G tsp
source tcl/export.tk
```

```
goblinExport G sample.tour tour
```

This sequence would load the filter precedures, read a problem in TSPLIB format from the file `sample.tsp`, compute a tour and save this tour to the file `sample.tour` which is again in TSPLIB format.

Do not confuse the Tcl/Tk canvasses which have been discussed in the last section with the Tcl library graph objects which can be generated from script level.

18.7.1 Import Filters

Type	Description
`gob`	GOBLIN native format
`edge`	DIMACS generic format for undirected graphs
`max`	DIMACS max-flow instance
`min`	DIMACS min-cost flow instance
`asn`	DIMACS assignment problem instance
`geom`	DIMACS geometric matching instance
`tsp`	TSPLIB symmetric TSP instance
`atsp`	TSPLIB asymetric TSP instance
`stp`	Steinlib instance
`mps`	MPS linear program (standard and CPLEX)
`bas`	MPS basis

18.7.2 Export Filters

Type	Description
`gob`	GOBLIN native format
`tcl`	Tcl library graph Object
`edge`	DIMACS generic format for undirected graphs
`max`	DIMACS max-flow instance
`min`	DIMACS min-cost flow instance
`asn`	DIMACS assignment problem instance
`flow`	DIMACS flow labels
`geom`	DIMACS geometric matching instance
`match`	DIMACS matching solution
`tsp`	TSPLIB symmetric TSP instance
`atsp`	TSPLIB asymetric TSP instance
`tour`	TSPLIB solution
`mps`	Standard MPS linear program
`cplex`	CPLEX MPS linear program
`lp`	CPLEX LP format
`bas`	MPS basis

Chapter 19

Exception Handling

Include file: `globals.h`
Synopsis:

```
class ERGoblin                              {};

class ERIO          : protected ERGoblin    {};
class ERFile        : protected ERIO        {};
class ERParse       : protected ERIO        {};

class ERInternal    : protected ERGoblin    {};

class ERRejected    : protected ERGoblin    {};
class ERRange       : protected ERRejected  {};
class ERCheck       : protected ERRejected  {};
```

Throughout this document, we have described the exceptions which are thrown by the various methods. On the other hand, we did not list any declarations of exceptions. Instead of this, we formulate the general policy which exceptions should be used in which circumstances:

An exception `ERInternal` indicates that a data structure has been corrupted by an error prone method. The calling context is asked to destruct this object. This error class is a dummy. That is, such exceptions may be thrown, but should not occur in a method signature. Hence, instead of the GOBLIN exceptions, an `unexpected` exception is thrown which usually causes the termination of a program.

We mention that `absobj.h` defines macros `InternalError(scope,event)` and `InternalError1(scope)` which write some debug information including file and line information and then raise an internal error. The first macro takes two strings, the second reads the event description from `CT.logBuffer`. Use these macros consequently, but be aware that they can be applied from data object methods only.

An exception `ERRange` is returned if an array index exeeds the limits. Occasionally, another data structure has been corrupted by the calling context before and the calling context cannot handle the exception. In that sense, `ERRange` may also denote an internal error.

An exception `ERRejected` indicates that a method failed its task, but leaves consistent data structures. This does not mean that the method undoes all object manipulations which probably would result in very inefficient code.

It is impossible to formalize the notion of consistency from this general point of view, but only when the concrete algorithm or data structure has been specified.

For example, the method `abstractGraph::ExtractCycles()` translates 2-factors from the subgraph data structure into predecessor labels. If the subgraph is not a 2-factor, the method will use the predecessor labels as well, but later call the method `ReleasePredecessors()` to guarantee consistency.

On the other hand, the method `abstractGraph::ExtractTree(TNode x)` would return some spanning tree via the predecessor labels even if the subgraph contains cycles. Nevertheless, an exception `ERCheck` is returned to indicate the special situation. If the calling context considers this an error, it may release the predecessor labels from its own.

The detection of GOBLIN errors heavily depends on the presence of the pragma `_FAILSAVE_` which is defined in the file `config.h`. If this pragma is undefined, no error messages are generated, and no errors are detected. This substantially increases the performance and decreases the binary size of problem solvers.

Note that GOBLIN may throw an exception `ERCheck` even if the pragma `_FAILSAVE_` is undefined. Hence, if algorithms work correctly, the definition of `_FAILSAVE_` does not change the functionality of a problem solver, and should be omitted in the final version.

Part V

GOBLIN Executables

Chapter 20

The GOSH Interpreter

The `gosh` interpreter is based on the Tcl/Tk libraries which are the outcome of one of the most successful open source projects. The Tcl interpreter can process complex scripts, but can also be used interactively. Without much effort, it allows to construct adequate user interfaces for any kind of mathematical software.

GOSH extends the Tcl/Tk scripting language to graph objects in a natural way. Although Tcl is a rather traditional language, the windowing commands in Tk and the GOSH graph commands support some of the ideas of object orientation.

The interpreter is called by the console command `gosh` and then starts in the interactive mode. If the name of a script is passed as a parameter, this script is evaluated. A script `example.gosh` can also be evaluated by typing `source example.gosh` in the interactive mode.

Note that the Tcl interface of the GOBLIN library does not support all of the library functions, but mainly those which were useful for the graph browser GOBLET. Note also that the Tcl interface does not check the parameter lists of a GOSH command exhaustively. Inappropriate parameters are detected by the library functions, and instructive error reports are available by the log file in addition to the Tcl return value.

If you have built the shared object `libgoblin.so`, this dynamic library can be imported to an existing `tcl` interpreter by the command

```
load $libgoblin goblin
```

where `$libgoblin` stands for the complete path to the shared object. So far, this shared object does not form part of the system installation.

20.1 GOSH Ressources

There are two files which are important when using the GOSH shell, namely the **transscript** and the **configuration file**. Both files are located in the user root directory.

The transscript file `gosh.log` is an important source of information since most GOSH commands do not return instructive error messages. It can be flushed explicitly by the command `goblin restart`.

The configuration file `.goshrc` is read during the initialization of the `gosh` interpreter and whenever an object is read from file, this overwrites the default configuration parameters with some user dependent settings. The format is the same as described in Section 18.4.6 for the graph object files. The current context variable settings may be saved to `.goshrc` by the command `goblin export settings`.

20.2 Context Variables

All configuration parameters discussed in Chapter 14 can be manipulated by GOSH scripts. The variable name in GOSH differs from the C++ variable name just by the prefix `goblin`. For example, the Tcl variable `goblinMethSolve` matches the C++ variable `methSolve`. Note that all configuration parameters are global Tcl variables. If you want to access `goblinMethSolve` within a procedure, you have to declare this variable by `global goblinMethSolve`.

20.3 Root Command

After its initilization, a GOSH interpreter provides only one new command compared with Tcl/Tk. There is a many-to-one correspondence between GOSH interpreters and GOBLIN controller objects. All options of the root command `goblin` manipulate the controller or generate a new GOSH command and, by that, a new object.

Example:

```
goblin sparse digraph G 10
G generate arcs 20
```

The first command generates a directed graph with 10 nodes whose Tcl name is `G`. Initially, this graph does not contain any arcs. Hence the second command is used to generate 20 random arcs for `G`. If you want to generate bipartite graphs, specify the number of nodes in each component.

Message	Parameters	Effects
`restart`		Reset logging and tracing module
`configure`		Set some context flags
`read`	Object name, file name	Read graph object from file

`mixed graph`
`sparse graph`
`sparse digraph`
`sparse bigraph`
`dense graph`
`dense digraph`
`dense bigraph`
	Object name, number of nodes	Generate graph object

`ilp`
`lp`
`mip`
	Object name, number of rows, number of variables	Generate (mixed integer) linear program

`export`	`tk` | `xfig` | `goblet`, input file name, output file name	Read data object from file and translate it to a canvas. Uses a separate context
`export ascii`	input file name, output file name, optional integer	Similar, but export to a text based form. Mainly used for linear programs
`export settings`		Write configuration file
`echo`	`-nonewline`, string	Write string to goblin transscript

20.3.1 Ressource Management

Message	Return value
`size`	Current heap size
`maxsize`	Maximum heap size
`#allocs`	Total number of mallocs
`#fragments`	Current number memory fragments
`#objects`	Number of currently managed objects
`#timers`	Number of managed timers

20.3.2 Thread Support

The GOSH shell is all but thread-safe, and the thread support is intended for the GOBLET browser only. The browser utilizes a master thread for the graphical interface and one slave thread for the computations. Both threads (interpreters) share the GOBLIN controller and occasionally some graph objects. The slave interpreter uses an `alias` for the graph object which can be traced by the master but should not be edited during computations. All listed messages start by `goblin solver ...`

The master thread can try to terminate the slave thread by the command `stop` and wait for termination by testing `goblin solver idle` which returns `false` if the computation is still running. Note that only some solver routines support this termination mechanism. Eventually, the solver thread returns some information before exiting by using the `return` and `throw` commands. The information is received on master side by the command `result`.

Message	Parameter	Description
`thread`	Script name	Evaluate script in an own thread of execution
`alias`	Object name, object handle	Assign a Tcl command name to an existing graph object
`return`	Return code	Set return value of a thread
`throw`	Return code	Set return value of a thread and signal an error
`result`		Acknowledge the return code of a thread
`stop`		Try to terminate the solver thread
`idle`		Check if the solver thread is active

20.3.3 Messenger Access

Just as the context, ther is one messenger object shared by all GOSH shells. The explicit access to the messenger is restricted to the methods described in Section 15.2. Note that posting a message from Tcl level is implemented by the `goblin echo` command. All listed messages start by `goblin messenger ...`

The messenger does not keep all raised messages but only the most recent ones. The buffer size is just large enough to fill a screen. Complete and persistent information is provided by the transscript file.

Message	Operation / Return value
`restart`	Flush the message queue
`reset`	Reset the iterator to the first queued message
`eof`	Are there unread messages?
`void`	Is the queue empty?
`skip`	Move iterator to the next message
`text`	Message text
`class`	Message class ID
`handle`	Originators handle
`level`	Nesting level
`hidden`	Is message marked as hidden?
`filename`	Name of the most recent trace file
`blocked`	Is the solver thread currently waiting at trace point?
`unblock`	Free the solver thread from waiting at trace point

20.3.4 Accessing Timers

The commands listed here wrap the functionality described in Section 17.2. All messages start by the prefix `goblin timer ...` and the timer index which must be ranged in `0` to `[expr [goblin #timers]-1]`.

Message	Operation / Return value
`reset`	Reset the timer
`enable`	Enable the timer
`disable`	Disable the timer
`label`	Return the label
`acc`	Return the accumulated times
`prev`	Return the previous timer value
`max`	Return the maximum timer value
`min`	Return the minimum timer value
`av`	Return the average timer value

The running time of timer j relative to the previous cycle of timer i is retrieved by the command `goblin timer $i child $j`.

20.4 General Object Messages

All commands other than the `goblin` root command are associated with data objects to which messages can be sent. Messages may manipulate the addressed object, generate new objects from existing or call a solver routine. In many cases, the correspondence between the Tcl message and the signature of the GOBLIN C++ method called is obvious. A detailed documentation of the Tcl commands is therefore omitted.

Message	Parameters	Description
`delete`		Delete object and Tcl command
`trace`		Write trace object
`handle`		Return object handle
`master`		Register this object as the master object
`is`	`graph` | `mip` | `sparse` | `undirected` | `directed` | `bipartite` | `balanced`	Evaluate object type
`set name`	file name	Assign a file name

The above messages apply to all GOBLIN data objects. In the following, we list the messages for special classes of data objects. Currently, graph objects and linear problems are covered by the Tcl wrapper. The Tcl interpreter adopts the graph polymorphism from the core library.

20.5 Graph Retrieval Messages

Message	Parameter	Description
`write`	File name	Write object to file
`#nodes`		Return number of nodes
`#arcs`		Return number of arcs
`#artificial`		Return number of bend nodes
`source`		Return the default source node
`target`		Return the default target node
`root`		Return the default root node
`cardinality`		Return subgraph cardinality
`weight`		Return subgraph weight
`length`		Return total length of predecessor arcs
`max`	`ucap` | `lcap` | `length` | `demand` | `cx` | `cy`	Return maximum label
`is`	`planar`	Perform planarity test
`constant`	`ucap` | `lcap` | `length` | `demand`	Is this a constant labeling?
`adjacency`	Start node, end node	Return an adjacency or `*`

20.6 Graph Manipulation Messages

Message	Parameters	Description
`node`	`insert`	Insert graph node
`arc`	`insert`, head, tail	Insert arc
`generate`	`arcs`, number of arcs	Generate random arcs
	`eulerian`, number of arcs	Generate random cycle
	`regular`, node degree	Generate random regular graph
	`length`, `ucap`, `lcap`, `geometry`,	Generate random node and arc labels
	`parallels`	Split arcs so that every arc has capacity ≤ 1

Message	Parameters	Description
`extract`	`tree`, root node	Check if the subgraph forms a rooted tree. Generate predecessor labels
	`trees`	Check if the subgraph splits into trees. Generate predecessor labels
	`path`, start node s and end node t	Check if s and t are in the same connected component of the subgraph. Generate predecessor labels for some st-path
	`cycles`	Check if the subgraph forms a 2-factor. Generate predecessor labels. Return the number of cycles
	`matching`	Check if the subgraph forms a 1-matching. Generate predecessor labels
	`edgecover`	Check if the subgraph forms a (maximum cardinality) 1-matching. Return a (minimum size) edge cover by the predecessor labels
	`cut`	Generate colours which separate the nodes with finite and infinite distance labels
	`bipartition`	Generate colours which separate the nodes with odd and even distance labels
	`colours`	Generate node colours equivalent (not equal) with the node partition
`delete`	`subgraph`, `labels`, `predecessors`, `colours`, `potentials`, `partition`	Delete the specified data structure
`set`	`ucap` | `lcap` | `length` | `demand`	Assign a constant labeling
	`source` | `target` | `root`, node index	Assign special nodes

20.7 Sparse Graphs and Planarity

Message	Parameter	Description
`planarize`		Check if the graph is planar and occasionally compute a combinatorial embedding
`outerplanar`		For planar graph objects: If possible, refine the present combinatorial embedding to an outerplanar embedding. Return an arc on the exterior
`exterior`	arc index	For planar graph objects: Set the exterior face to the left hand side of the specified arc. Adjust the first incidence of the exterior nodes

20.8 Graph Layout Messages

The following messages apply to every graph object `G` with the prefix

```
G layout ...
```

and manipulate the node coordinates. rtificial nodes (which are only used for layout purposes) are also added, deleted or shifted. Most methods allow to specify `-spacing` followed by the desired minimum distance between two nodes. For grid drawings, the keyword `-grid` is synonymous. Whenever `-dx` and `-dy` are available, `-spacing` can also be used.

Message	Options	Description
`scale`	bounding box (minX maxX minY maxY)	Scale geometric embedding to the specified size. When max ¡ min, the drawing is mirrored
`strip`		Shift the geometric embedding so that the upper left corner of the bounding box becomes the origin
`align`	`-spacing`	Reroute arcs so that parallel arcs and loops can be distinguished
`tree`	`-dx`, `-dy`, `-left`, `-right`	Embedding guided by the predecessor arcs. A tree or forest is drawn, and the nodes are aligned atop of its successors as specified
`circular`	`-spacing`, `-colours`, `-predecessors`, `-outerplanar`	Embedding of the graph on a cycle. Use an option to control the node order
`orthogonal`	`-grid`, `-small`	Embedding of the graph in a grid. The small node option applies to 2-connected graphs with maximum degree 4 or less
`fdp`	`-spacing`, `-preserve`, `-unrestricted`	Force directed layout. Using the preserve option, nodes are shifted without modifying the edge crossing properties
`layered`	`-dx`, `-dy`	Embedding guided by the node colours
`plane`	`-grid`, `-convex`, `-basis`	For planar graph objects: Straight line drawing of the current embedding and the specified basis arc. Convex drawing requires 3-connectivity
`visibility`	`-grid`, `-giotto`, `-raw`	For planar graph objects: Visibility representation or an follow-up giotto drawing
`equilateral`	`-spacing`	For 2-connected outerplanar graphs: Draw every interior face as a regular polygone

20.9 Graph Node and Arc Messages

The most significant difference between the C++ library functions and the GOSH message concerns the nodes and arcs of a graph. All messages which address the node 3 and the arc 7 of a graph `G` start

```
G node 3 ...
```

and

```
G arc 7 ...
```

respectively. This applies to all messages listed in Table 20.9.1 and Table 20.9.2. All arc indices range between 0 and $2m-1$, and arc directions are specified by the least significant bit. On the other hand, arc insertion messages return the new arc index without this additional bit.

Example:

```
G spath $s
set a [expr 2*[G arc insert $u $v]]
G arc $a set length [expr -[G node $u potential]
                          +[G node $v potential]]
```

would generate a new arc with start node `u` and end node `v`. The new arc is initialized with zero reduced length so that it can replace one of the arcs in the shortest path tree which was computed before.

In order to specify the drawing of a graph arc `a`, first add an alignment point by the command

```
G arc $a set align $x $y
```

where `x` and `y` are the coordinates of the alignment point (This denotes the point where the arc labels are printed). Then interpolation points are successively defined by

```
G arc $a interpolate $x $y
```

where `x` and `y` are the coordinates again. New interpolation points are placed at the end of the list. If an arc is deleted, its alignment point and all interpolation point are deleted recursively. If a node is deleted, all incident arcs are deleted recursively.

Example:

```
for {set a 0} {$a<[G #arcs]} {incr a} {
    set a2 [expr 2*$a]
    if {[G arc $a2 head] == [G arc $a2 tail]} {
        set $x0 [G node [G arc $a2 head] cx]
        set $y0 [G node [G arc $a2 head] cy]
        G arc $a2 set align $x0 [expr $y0-10]
        G arc $a2 interpolate [expr $x0-10] [expr $y0-10]
        G arc $a2 interpolate $x0 [expr $y0-20]
        G arc $a2 interpolate [expr $x0+10] [expr $y0-10]
    }
}
```

checks the graph for loops which cannot be displayed without interpolation points. For every loop, an alignment point for the arc label and three interpolation points for a spline drawing are defined.

Whenever undefined or infinite labels are needed they are replaced by an asterisk `*`.

20.9.1 Node Based Messages

Message	Description
`delete`	Delete node
`demand`	Return the node demand
`cx`	Return the x-coordiante
`cy`	Return the y-coordinate
`colour`	Return the node colour or `*`
`degree`	Return the subgraph degree
`distance`	Return the distance label or `*`
`potential`	Return the node potential
`predecessor`	Return the predecessor arc or `*`
`first`	Return some outgoing arc or `*`
`thread`	Return the next bend node index or `*`
`hidden`	Check if the node is displayed
`set thread`	For artificial nodes: Insert new bend nodes after that node with the given coordinates
`set`	Manipulate one of the listed node ressources

20.9.2 Arc Based Messages

Message	Description
`delete`	Delete arc
`contract`	Contract arc
`straight`	Release all bend node of this arc
`ucap`	Return the upper capacity bound or `*`
`lcap`	Return the lower capacity bound
`length`	Return the arc length
`subgraph`	Return the subgraph label
`orientation`	Return the orientation
`head`	Return the start node
`tail`	Return the end node
`right`	Return a further arc with the same start node
`align`	Return the label alignment point index or `*`
`set align`	Generate an alignment point with the given coordinates
`hidden`	Check if the arc is displayed
`set`	Manipulate one of the listed arc ressources

20.10 Graph Optimization Messages

The Tcl API of the C++ problem solver methods on script level is obvious:

Example:

```
goblin read G "example.gob"
G set demand 2
G set ucap *
G maxmatch
G write "example.rst"
```

computes a 2-matching of the graph whereas

```
G set demand 2
G set ucap 1
G maxmatch
```

determines a 2-factor.

Note that a matching solver is defined for undirected graphs only while network flow methods can be accessed with digraphs only. The Tables 20.12, 20.13, 20.14 and 20.15 list all messages which are restricted to special classes.

In order to simplify contributions by other authors, some solver messages are available from script level for every graph object even if there are no solver methods for mixed graphs yet. This applies for the tree packing and the Chinese postman solver.

Message	Parameter	Description
`spath`	Root node	Compute a shortest path tree and return its length
`connected`	Order of Connectivity	Check for vertex connectivity
`econnected`	Order of Connectivity	Check for egde connectivity
`sconnected`	Order of Connectivity	Check for strong connectivity
`seconnected`	Order of Connectivity	Check for strong edge connectivity
`colouring`	Accepted number of colours (optional)	Compute a node colouring
`edgecolouring`	Accepted number of colours (optional)	Compute an edge colouring
`cliques`	Accepted number of cliques (optional)	Compute a cliques cover
`clique`		Compute a maximum clique and return its cardinality
`vertexcover`		Compute a vertex cover and return its cardinality
`stable`		Compute a maximum stable set and return its cardinality
`eulerian`		Compute an Euler cycle if one exists. Return if the Graph is Eulerian
`stnumbering`		Compute an *st*-numbering if the graph is 2-connected
`topsort`		Compute a topological ordering or return a node on a cycle
`critical`		Compute a critical path and return its end node
`mintree`	`-root` Root node (optional)	Compute a minimum spanning arborescence and return its length
`mintree`	`-max`	Compute a maximum spanning arborescence
`mintree`	`-cycle`	Compute a minimum 1-cycle tree
`tsp`	Root node (optional)	Compute an Hamiltonian cycle and return its length
`steiner`	Root node	Compute a minimum steiner tree and return its length
`treepacking`	Root node	Compute a maximum packing of arborescences
`maxcut`		Compute a cut of maximum capacity and return this capacity
`postman`		Compute a minimum Eulerian supergraph and return its weight

20.11 Derived Graph Constructors

Message	Description
`linegraph`	Generate line graph
`linegraph -planar`	Generate a planar line graph
`truncate`	Replace the verices by cycles
`complement`	Generate complementary graph
`underlying`	Generate underlying graph
`dualgraph`	Generate the dual graph of a plane graph
`spread`	Generate an outerplanar representaion of a plane graph
`induced subgraph`	Subgraph induced by a specified node colour
`induced orientation`	Orientation induced by the node colours
`induced bigraph`	Bigraph induced by two specified node colours
`transitive`	Generate transitive closure
`intransitive`	Generate intransitive reduction
`contraction`	Contract every node colour into a single node
`nodesplitting`	Generate node splitting
`orientation`	Generate complete orientation
`distances`	Generate distance graph

20.12 Messages for Undirected Graphs

Message	Parameters	Description
`subgraph`	Object name	Export subgraph into a separate object
`metric`	Object name	Generate metric closure
`tiling`	Object name, number of rows, number of columns	Generate graph which consists of several copies of the addressed graph
`maxmatch`		Compute maximum matching and return its cardinality
`mincmatch`		Compute perfect matching of minimum weight, return this weight or `*`
`edgecover`		Compute an edge cover of minimum weight and return this weight
`tjoin`		Compute minimum t-join and return its weight or `*`

20.13 Messages for Directed Graphs

Message	Parameters	Description
`subgraph`	Object name	Export subgraph into a separate object
`splitgraph`		Generate a balanced version of the network flow problem
`maxflow`	Source, target	Compute a maximum *st*-flow and return the flow value
`mincflow`	Source, target	Compute a maximum *st*-flow of minimum weight and return this weight
`circulation`		Compute an admissible circulation or *b*-flow
`minccirc`		Compute an admissible circulation or *b*-flow of minimum weight and return this weight

20.14 Messages for Bipartite Graphs

Message	Parameters	Description
`#outer`,		Cardinality of the left hand component
`#inner`,		Cardinality of the right hand component
`node`	index, `swap`	Move node the other component

20.15 Messages for Balanced Flow Networks

Message	Parameters	Description
`maxbalflow`	Source	Compute a maximum balanced *st*-flow, return the flow value
`mincbalflow`	Source	Compute a maximum balanced *st*-flow of minimum weight and return this weight

20.16 Linear Programming

20.16.1 Instance Manipulation Messages

Message	Parameters	Operation
`read`	`bas`, `basis`, `mps` or `problem`, file name	Read MIP instance or basis
`maximize`		Mark as maximization problem
`minimize`		Mark as minimization problem
`invert`		Invert the object vector and sense
`nullify`		Dismiss the objective vector
`resize`	Number of rows, number of columns, number of non-zero coefficients	Reallocate MIP instance with the specified dimensions
`strip`		Reallocate MIP instance within a minimum of memory
`set`	`coeff` or `coefficient`, row index, column index, float value	Set a coefficient in the restriction matrix
	`index`, row index, variable index, `upper` or `lower`	Specify a basis restriction
`reset`		Reset basis to the lower variable range restrictions

20.16.2 Instance Retrieval Messages and Basis Access

Message	Parameter	Operation / Return Value
`write`	`lp`, `mps`, `cplex`, `bas` or `basis`, file name	Write instance or basis to file
`#rows` or `#restrictions`		The number of restrictions
`#columns` or `#variables`		The number of variables
`orientation`		`row` or `column`
`direction`		`maximize` or `minimize`
`coeff` or `coefficient`	row index, variable index	A coefficient of the restriction matrix
`tableau`	`coeff` or `coefficient`, row index, column index	A tableau coefficient
`inverse`	`coeff` or `coefficient`, row index, column index	A basis inverse coefficient
`feasible`	`primal` or `dual`	Is the current basis feasible?
`pivot`	`veriable` or `column`	Return the pivot column
	`row` or `restriction`	Return the pivot row
	`direction`	Return `upper` or `lower`
`objective`	`primal` or `dual`	Return the objective value
`row index`	restriction label	Corresponding index or `*`
`column index`	variable label	Corresponding index or `*`

20.16.3 Row and Column Based Messages

All messages which address the restriction 3 and the variable 7 of a mixed integer problem `X` start

```
X row 3 ...
```

and

```
X column 7 ...
```

respectively. You may also use the keywords `restriction` instead of `row` and the keyword `variable` instead of `column`. This syntax applies to all messages listed in Table 20.16.4 and Table 20.16.5. All column indices range between 0 and $l-1$. The row indices range between 0 and $k-1$ respectively $k+l-1$ depending on whether variable range restrictions are included. Here, k and l denote the effective dimensions returned by `[X #restrictions]` and `[X #variables]` respectively.

Whenever infinite labels are needed they are replaced by an asterisk `*`. Lower bounds cannot be set to $+\infty$, upper bounds are never $-\infty$ which makes the procedure unique.

Example:

```
if {[X row $i type]=="non_basic"} {
    set k [X column $j index]
    catch {X pivot $i $j upper}
}

puts [X row $i type]
```

would check if the ith row is in the current basis and occasionally try to exchange the current basis row k matched with variable j with i. If the pivoting is successful, that is, if a basis structure can be maintained, the output is `upper`.

20.16.4 Row Based Messages

Message	Parameter	Operation / Return Value
`insert`	Upper and lower bound (or `*`), variable type	Add a variable (column)
`cancel`		Effectively deletes the restriction
`ubound`		Upper right-hand side bound
`lbound`		Lower right-hand side bound
`label`		The restriction label
`type`		The restriction type
`index`		The variable associated with the restriction in basis or `*`
`value`	`upper` or `lower`	The (dual) variable value
`slack`	`upper` or `lower`	The slack
`set`	`ubound`, `lbound` or `label`, ressource value	Change one of the listed ressources

20.16.5 Column Based Messages

Message	Parameter	Operation / Return Value
`insert`	Upper and lower bound (or `*`), variable type	Add a variable (column)
`cancel`		Effectively deletes the variable
`urange`		The upper variable bound
`lrange`		The lower variable bound
`cost`		The cost coefficient
`type`		The variable type
`label`		The variable name
`index`		The basis row associated with the variable
`value`		The (primal) variable value
`mark`	`float`, `int` or `integer`	Set variable type
`set`	`urange`, `lrange`, `cost` or `label`, ressource value	Change one of the listed ressources

20.16.6 Optimization Messages

Message	Parameter	Operation
`solve`	`lp`, `primal` or `dual`	Solve linear relaxation
	`mixed` or `mip`	Solve mixed integer problem
`start`	`primal` or `dual`	Determine feasible solution of the linear relaxation
`pivot`	Variable index, incoming row index, `upper` or `lower`	Move from one basis to another

Chapter 21

Solver Applications

One may argue that explicit solver programs are immaterial by the existence of the GOSH interpreter. But the overhead for tracing and the graphical display is obvious, and the compilation of efficient solvers does not require a Tcl/Tk installation.

All GOBLIN executables support the runtime configuration as described in Section 14.8. That is, one can control the logging and tracing functionality (including the graphical display) from the console.

21.1 Solver Applications

The last argument passed to a problem solver is the input file name, say `xyz`. The solver expects a file `xyz.gob` which consists of a graph definition (see Section 18.4 for the file formats). Do not specify the extension `.gob` explicitly!

The computational results are written to a file `xyz.rst`, and the logging information is written to a file `xyz.log`. By default, the output is the entire graph definition which can be read by the program `gobshow` to display the results. One can produce a more economic output by using the options `-sh` and `-silent`. The first option forces the solver to write only the relevant data structure (subgraph, predecessor labels, etc.) to file, the second option suppresses the writing of a log file completely.

Note that the main routines do not support any error handling yet. In case of trouble, consult the log file. The return value indicates the existence of a feasible solution rather than internal errors. If the log file does not give evidence of what has gone wrong, please contact the author.

21.1.1 Matching Problems

The program `optmatch` is the GOBLIN solver for all kinds of matching problems. The input graph may be any undirected graph, either sparse or complete. For bipartite graphs, specialized methods are used.

If one specifies `-w`, either a perfect matching of minimal costs is computed or the program shows that no perfect matching exists. If this option is omitted, the objective is a maximal or minimum deficiency matching. For example,

```
optmatch -w samples/optmatch2
```

would return the 2-factor depicted in Figure 13.10 since all node demands defined in the input file are 2.

The node demands are specified in the input file. If you want to distinguish upper and lower bounds on the node degrees, you may use the option `-deg`. Then the solver expects two additional input files whose names differ from the graph definition file only by the extensions `.adg` respectively `.bdg`. The first file consists of the lower degree bounds, the second consists of the upper degree bounds. The formats are the same as for the graph definition.

If you want to solve a geometrical problem, you must set the `metrics` in the input file to a value other than zero (see Section 11.1.3 for the details).

21.1.2 Network Flow Problems

The program `optflow` is the GOBLIN solver for all kinds of network flow problems. The input graph must be a directed graph, either sparse or complete.

There are two ways to use this solver: One may use the `-div` option and specify a source s and a target node t. The solver will try to find a pseudo-flow such that all divergences are zero, except for s and t. The divergence of s is maximized, and the divergence of t is minimized simultaneously. This solver requires that

- all lower arc capacities are zero,
- all node demands are zero, except for the nodes s and t,
- all arc length labels are non-negative.

For example,

```
optflow -div 0 7 samples/maxflow4
```

would return the $(0, 7)$-flow depicted in Figure 13.5, and a minimum $(0, 7)$-cut likewise.

If no source node and no target node are specified, the solver will determine a pseudo-flow such that all divergences match the node demands, called a b-flow. If the `-w` option is used, the solver returns a b-flow with minimum costs. This solver requires that

- all lower capacity bounds are non-negative,
- the node demands sum up to zero.

If the maximum value of an st-flow is known a priori, one can assign the demands of s and t accordingly such that the second solver applies. For example,

```
optflow samples/maxflow4
```

determines a maximum $(0, 7)$-flow due to the node demands specified in the file.

21.1.3 Minimum Spanning Tree Problems

The program `mintree` is the GOBLIN solver for minimum spanning tree and 1-tree problems. The input graph must be a graph or a digraph object, either sparse or complete.

One may specify a root node r by the option `-r`. In that case, the predecessors will form a rooted tree or, for 1-trees, a directed cycle through r plus several node disjoint arborescences with their root nodes on the cycle.

If the input graph is undirected and no root node is specified, a subgraph is returned which consists of the tree arcs. The 1-tree solver is enabled by the parameter `-1`. For example,

```
mintree -r 9 -1 samples/mintree1
```

would return the 1-tree depicted in Figure 21.1.

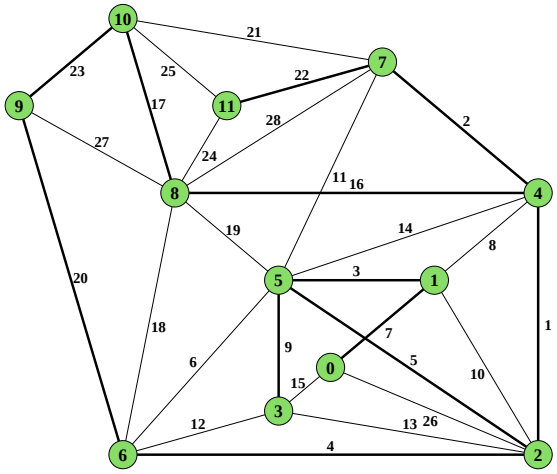


Figure 21.1: A Minimum 1-Cycle Tree

21.1.4 Shortest Path Problems

The program `gsearch` is the GOBLIN solver for shortest path problems. The input graph must be a graph or a digraph object, either sparse or complete.

One has to specify a root node s by using the `-s` option. The output are the predecessor labels which determine a shortest path tree rooted at s. If the complete output form is used, the distance labels are also returned.

One may optionally specify a target node by using the `-t` option. In that case, the Dijkstra label setting method may halt once the target has been reached.

Note that all shortest path algorithms require that no negative length cycles exist, and some methods that the length labels are even non-negative. If the input graph and the method configured are incompatible, this will be reported in the log file. For example,

```
gsearch -s 0 samples/gsearch1
```

would return the shortest path tree depicted in Figure 13.1.

21.1.5 Chinese Postman Problems

The program `postman` is the solver for Chinese postamn problems. The input file must denote a *sparse* graph object, either directed or undirected. No mixed or bipartite graphs are allowed. The output is an Eulerian supergraph with minimum costs. For example,

```
postman samples/postman1
```

would return the graph depicted in Figure 13.12.

21.1.6 Other Solvers

Table 2.1 lists some more problem solvers some of which are experimental. For this reason we omit a documentation of these programs, but refer to the source files of the main routine which easily exhibit how the solvers apply.

21.2 Linear Programming

The last argument passed to `lpsolve` is the LP instance name, say `xyz`. The solver expects an input file `xyz.mps` which contains a linear program in CPLEX MPS format. Do not specify the extension `.mps` explicitly!

The optimal basis is written to a file named `xyz.bas`, and the logging information is written to `xyz.log`. If the option `-b` is given, the start basis is read from `xyz.bas` and overwritten with the final basis. If `-f` is specified, the computation stops with a suboptimal but primal or dual feasible basis depending on which method is configured in `methLP`. The option `-silent` suppresses the writing of a log file.

21.3 Random Instance Generators

These tools can be used to generate random graph objects. The last argument passed to an instance generator is the output file name, say `xyz`. In any case, the solver writes a file `xyz.gob`, but never a log file. All tools work in a very similar way, and Table 21.1 describes the command line options. By default, no random arc labels and no parallel arcs are generated.

Option	Description
`-n`	Number of nodes
`-m`	Number of arcs, only for sparse objects
`-dns`	Complete graph
`-euler`	Generate Eulerian graph
`-regular`	Generate regular graph

Table 21.1: Instance Generator Options

21.3.1 Random Digraphs

The tool `rgraph` generates directed graphs. The option `-euler` can be used to obtain Eulerian digraphs. For example,

```
rdigraph -n 5 -m 6 -randUCap 1 -randLCap 1 example1
```

would generate a flow network with 5 nodes, 6 arcs and random upper and lower capacity bounds, and

```
rdigraph -n 5 -m 22 -euler -randParallels 1 example2
```

would generate an Eulerian digraph with 5 nodes and 22 arcs. Note that the `-randParallels 1` option cannot be omitted here since a simple digraph on 5 nodes may only consist of 20 arcs.

21.3.2 Random Bigraphs

The tool `rbigraph` generates bipartite graphs. The option `-regular` can be used to obtain regular bigraphs. In that case, the `-n` and the `-m` option are immaterial. Otherwise the size of both partitions is passed by the `-n` option. For example,

```
rbigraph -n 3 4 -m 5 -randLength 1 example3
```

would generate a bigraph with 5 arcs, 3 outer nodes, 4 inner nodes and random length labels. On the other hand,

```
rbigraph -regular 3 2 example4
```

would generate a 2-regular bigraph with 6 arcs, 3 outer nodes and 3 inner nodes. That is, the `-regular` option replaces or overrides the `-n` and the `-m` option.

21.3.3 Random Graphs

The tool `rgraph` generates undirected non-bipartite graphs. There are two additional options `-euler` and `-regular` to obtain Eulerian and regular graphs respectively. For example,

```
rgraph -n 5 -m 6 example5
```

would generate a sparse graph with 5 nodes and 6 edges, whereas

```
rgraph -n 5 -randGeometry 1 -dns -seed 77 example6
```

would generate a complete graph with 5 nodes and 10 and a random embedding into plane. The random generator is initialized with a special seed.

21.4 Graphical Display

Every problem solver has the capability to produce graphical information if the tracing module is configured that way. But sometimes it is more convenient to display a graph directly. This is achieved by the program `gobshow`. Note that the file extension must be specified. For example,

```
gobshow -arcLabels 4 samples/optasgn1.gob
```

would show the graph defined in the file `optasgn1.gob`, especially its length labels. This program should be redundant in view of the existence of the GOBLET graph browser. Since the compilation of the GOSH interpreter is the most difficult part of the GOBLIN installation, it may be useful in case of trouble.

Part VI

Appendix

Chapter 22

Computational Results

22.1 Symmetric TSP

All computations were performed with the GOBLET graph browser 2.7.1 on an Athlon XP 1800 PC with 256 MB RAM and SuSE Linux 7.3. and with gcc optimization level -O5. The test problems are all from the TSPLIB:

```
http://www.iwr.uni-heidelberg.de/groups
                /comopt/software/TSPLIB95/
```

The following methods have been tested here:

- SGO: The fast version of the 1-tree subgradient optimization with local search enabled. This method has produced the most heuristic tours.
- SGO2: Exhaustive 1-tree subgradient optimization with local search enabled.
- CAND: Branch and bound on a candidate graph with local search enabled and with `methCandidates=0`.
- CND2: As before but with `methRelaxTSP2=2`.
- EXH: Branch and bound on the entire graph with local search disabled.
- EXH2: As before but with `methRelaxTSP2=2`.

The initial tours were obtained from random tours with local search enabled. Note that the candidate graph generation also includes such random tours. The performance of the available construction heuristics is not tested.

With a few exceptions (marked by an asterisk), the branch and bound has not been restricted in terms of running times or memory usage. Practically, one would interrupt the candidate search after a certain number of branching steps.

Instance	Opt	Method	Root	Found Gap	Time	Branch
burma14	3323	SGO	13	3323	0s	
ulysses16	6859	SGO	14	6859	1s	
gr17	2085	SGO	0	2085	1s	
gr21	2707	SGO	0	2707	0s	
ulysses22	7013	SGO	14	7013	1s	
gr24	1272	SGO	0	1272	1s	
fri26	937	SGO	0	937	1s	
bayg29	1610	EXH	4	1610	2s	8
bays29	2020	EXH	4	2020	2s	12
dantzig42	699	SGO	27	[697, 699]	3s	
dantzig42	699	EXH	27	699	3s	8
swiss42	1273	SGO	4	[1272, 1273]	3s	
swiss42	1273	EXH	4	1273	3s	8
hk48	11461	SGO	40	[11445, 11461]	4s	
hk48	11461	EXH	40	11461	1s	12
gr48	5046	SGO	40	[4959, 5055]	4s	
gr48	5046	CAND	40	[4959, 5046]	8s	1614
gr48	5046	EXH	37	5046	201s	816
eil51	426	SGO	21	[423, 432]	3s	
eil51	426	CAND	21	[423, 426]	7s	1830
eil51	426	EXH	21	426	32s	228
berlin52	7542	SGO	42	7542	3s	

Instance	Opt	Method	Root	Found Gap	Time	Branch
brazil58	25395	SGO	30	[25355, 25395]	8s	
brazil58	25395	EXH	30	25395	17s	56
st70	675	SGO	16	[671, 684]	6s	
st70	675	CAND	16	[671, 675]	11s	2844
st70	675	EXH	16	675	18s	64
eil76	538	SGO	6	[537, 543]	7s	
eil76	538	CAND	6	[537, 538]	19s	3324
eil76	538	EXH	6	538	2s	8
pr76	108159	SGO	39	[105120,108879]	12s	
pr76	108159	CAND	39	[105120,108159]	345s	6158
pr76	108159	EXH*	39	[106509,108159]	1516s	1000
gr96	55209	SGO	79	[54570, 55462]	34s	
gr96	55209	CAND	79	[54570, 55209]	45s	4664
gr96	55209	EXH	79	55209	715s	760
rat99	1211	SGO	63	[1206, 1220]	14s	
rat99	1211	CAND	63	[1206, 1211]	15s	4544
rat99	1211	EXH	63	1211	36s	62
rd100	7910	SGO	15	[7898, 8046]	17s	
rd100	7910	SGO2	15	[7900, 8046]	29s	
rd100	7910	CAND	15	[7900, 7910]	16s	4612
rd100	7910	EXH	15	7910	9s	18
kroA100	21282	SGO	86	[20937, 21583]	20s	
kroA100	21282	CAND	86	[20937, 21282]	61s	4946
kroA100	21282	EXH	86	21282	5016s	5180
kroB100	22141	SGO	53	[21834, 23698]	15s	
kroB100	22141	CAND	53	[21834, 22141]	51s	4914
kroB100	22141	EXH	53	22141	1338s	1274
kroC100	20749	SGO	49	[20473, 20812]	17s	
kroC100	20749	CAND	49	[20473, 20749]	21s	4706
kroC100	20749	EXH	49	20749	1422s	1402
kroD100	21294	SGO	45	[21142, 21493]	18s	
kroD100	21294	CAND	45	[21142, 21294]	19s	4674
kroD100	21294	EXH	45	21294	148s	156
kroE100	22068	SGO	71	[21800, 22141]	36s	
kroE100	22068	CAND	71	[21800, 22068]	61s	5100
kroE100	22068	EXH	71	22068	1195s	1142

Instance	Opt	Method	Root	Found Gap	Time	Branch
eil101	629	SGO	41	[628, 647]	13s	
eil101	629	CAND	51	[628, 629]	43s	5112
eil101	629	EXH	51	629	98s	196
lin105	14379	SGO	103	[14371, 14379]	29s	
lin105	14379	EXH	103	14379	5s	6
pr107	44303	SGO2	86	[44116, 44744]	144s	
pr107	44303	CND2	86	[44116, 44438]	92s	5210
pr107	44303	EXH2	86	44303	25s	8
gr120	6942	SGO	17	[6912, 7082]	21s	
gr120	6942	CAND	17	[6912, 6942]	104s	6614
gr120	6942	EXH	17	6942	611s	446
pr124	59030	SGO	59	[58068, 59076]	26s	
pr124	59030	CAND	59	[58068, 59030]	12s	1534
pr124	59030	EXH	59	59030	1489s	786
bier127	118282	SGO	93	[117431,118580]	39s	
bier127	118282	CAND	93	[117431,118282]	28s	6500
bier127	118282	EXH	93	118282	112s	66
ch130	6110	SGO	81	[6075, 6216]	29s	
ch130	6110	SGO2	81	[6076, 6216]	45s	
ch130	6110	CAND	81	[6076, 6110]	109s	7338
ch130	6110	EXH	81	6110	4428s	3052
pr136	96772	SGO	34	[95720, 98650]	40s	
pr136	96772	SGO2	34	[95935, 98650]	177s	
pr136	96772	CAND	34	[95935, 96772]	2757s	23861
gr137	69853	SGO	87	[69120, 70240]	55s	
gr137	69853	CAND	87	[69120, 69853]	78s	7462
gr137	69853	EXH	87	69853	2894s	1508
pr144	58537	SGO	29	[58190, 59113]	25s	
pr144	58537	CAND	29	[58190, 58537]	19s	3670
pr144	58537	EXH	29	58537	818s	324
ch150	6528	SGO	39	[6490, 6610]	40s	
ch150	6528	CAND	39	[6490, 6528]	422s	10930
ch150	6528	EXH	39	6528	6318s	3470
kroA150	26524	SGO	112	[26265, 26725]	49s	
kroA150	26524	SGO2	112	[26299, 26725]	92s	
kroA150	26524	CAND	112	[26299, 26525]	264s	9428
kroA150	26524	EXH	112	26524	22472s	

Instance	Opt	Method	Root	Found Gap	Time	Branch
kroB150	26130	SGO2	68	[25733, 26678]	164s	
kroB150	26130	CAND	68	[25733, 26130]	905s	13410
pr152	73682	SGO2	120	[73209, 74279]	223s	
pr152	73682	CAND	120	[73209, 73682]	52s	8356
pr152	73682	EXH	120	73682	18547s	1136
u159	42080	SGO	86	[41925, 42168]	56s	
u159	42080	CAND	86	[41925, 42080]	132s	9352
u159	42080	EXH	86	42080	430s	204
si175	21407	SGO2	1	[21375, 21426]	278s	
brg180	1950	SGO2	111	[1950, 2020]	513s	
brg180	1950	CND2	111	1950	248s	10478
rat195	2323	SGO2	43	[2300, 2379]	297s	
rat195	2323	CAND	43	[2300, 2323]	2145s	23784
d198	15780	SGO2	167	[15712, 15825]	450s	
d198	15780	CAND	167	[15712, 15780]	802s	15196
kroA200	29368	SGO	40	[29065, 30043]	104s	
kroA200	29368	CAND	40	[29065, 29368]	17015s	91520
kroB200	29437	SGO	57	[29165, 30364]	87s	

22.2 Asymmetric TSP

All computations were performed with the GOBLET graph browser 2.7.2 on an Athlon XP 1800 PC with 256 MB RAM and SuSE Linux 10.0 and without gcc optimization. The test problems are all from the TSPLIB:

```
http://www.iwr.uni-heidelberg.de/groups
                /comopt/software/TSPLIB95/
```

The following methods have been tested here:

- SGO: The fast version of the 1-tree subgradient optimization with local search enabled. This method has produced the most heuristic tours.
- SGO2: Exhaustive 1-tree subgradient optimization with local search enabled.
- CAND: Branch and bound on a candidate graph with local search enabled and with `methCandidates=0`.
- CAND: Branch and bound on the entire graph with local search disabled. For difficult problems, the number of branch nodes has been restricted to 1000 so that no optimality proof is obtained but the lower bound is improved.

Instance	Opt	Method	Root	Found Gap	Time	Branch
br17	39	SGO	7	39	1s	
ftv33	1286	SGO	11	1286	0s	
ftv35	1473	SGO	11	[1456, 1484]	4s	
ftv35	1473	EXH	11	1473	11s	23
ftv38	1530	SGO	7	[1512, 1541]	2s	
ftv38	1530	CAND	7	[1514, 1530]	5s	284
ftv38	1530	EXH	7	1530	30s	56
p43	5620	SGO2	40	[5611, 5629]	29s	
p43	5620	CAND	40	[5611, 5620]	1629s	5000
p43	5620	EXH	40	[5614, 5620]	1133s	100
ftv44	1613	SGO	17	[1581, 1708]	3s	
ftv44	1613	CAND	17	[1583, 1634]	35s	596
ftv44	1613	EXH	17	1613	175s	164
ftv47	1776	SGO	17	[1748, 1932]	4s	
ftv47	1776	CAND	17	[1748, 1776]	27s	542
ftv47	1776	EXH	17	1776	156s	190
ry48p	14422	SGO2	40	[14290, 14429]	21s	
ry48p	14422	EXH	40	14422	75s	46
ft53	6905	SGO2	52	6905	10s	
ftv55	1608	SGO2	30	[1584, 1758]	9s	
ftv55	1608	CAND	30	[1584, 1608]	31s	612
ftv55	1608	EXH	30	1608	883s	882
ftv64	1839	SGO2	20	[1808, 1958]	26s	
ftv64	1839	CAND	20	[1808, 1839]	30s	738
ftv64	1839	EXH	20	1839	5816s	3996

Instance	Opt	Method	Root	Found Gap	Time	Branch
ft70	38673	SGO	47	[38632, 38793]	22s	
ft70	38673	CAND	47	[38632, 38694]	14s	708
ft70	38673	EXH	47	38673	43s	16
ftv70	1950	SGO	70	[1907, 2176]	7s	
ftv70	1950	CAND	70	[1907, 1973]	2181s	12804
ftv70	1950	CAND	70	[1908, 1950]	335s	2510
ftv70	1950	EXH	70	[1928, 1950]	2589s	1000
kro124p	36230	SGO	90	[35974, 39278]	27s	
kro124p	36230	SGO2	90	[35998, 39278]	64s	
kro124p	36230	CAND	90	[35999, 36230]	207s	1496
kro124p	36230	EXH	90	36230	427s	52
ftv170	2755	SGO	123	[2682, 2932]	40s	
ftv170	2755	SGO2	123	[2707, 2932]	182s	
ftv170	2755	CAND	123	[2707, 2780]	4622s	10000
ftv170	2755	CAND	123	[2707, 2772]	4516s	10000
ftv170	2755	CAND	123	[2707, 2755]	6031s	10000

22.3 Min-Cost Flow

All computations were performed with the GOBLET graph browser 2.5.1 on an Athlon XP 1800 PC with 256 MB RAM and SuSE Linux 7.3. The test sets are NETGEN problems taken from

```
http://elib.zib.de/pub/Packages/mp-testdata
                        /mincost/netg/index.html
```

The tested methods are the cost scaling algorithm (**CS**, `methMinCCirc=1`) and the network simplex method (**NW**, `methMinCCirc=5`). The columns for the respective solution times with `gcc` optimization level `-O5` and the pragmas `_LOGGING_` and `_FAILSAVE_` unset.

Instance	Nds	Arcs	Cap	Len	Objective	CS	NW
big5	5000	80101	10000	1000	15817090	198s	9s
big6	5000	60092	10000	1000	15864843	165s	7s
big7	5000	40105	10000	1000	13970599	138s	6s
cap1	1000	10000	500000	10000	2572055650	7s	1s
cap2	1000	30000	1199995	10000	868553404	14s	1s
cap3	1000	40000	1199995	10000	835752895	23s	1s
cap4	5000	30000	600000	10000	6572052044	107s	1s
cap5	5000	40000	600000	10000	4596714758	130s	2s
cap6	5000	49999	600000	120756	3905503120	130s	2s
cap7	5000	60000	600000	10000	3514982153	142s	2s
cap8	10000	40000	1000000	10000	13836268653	473s	4s
cap9	10000	50000	1000000	10000	12273727410	389s	5s
transp1	800	10028	200000	9997	258178684	9s	0s
transp2	800	20000	200000	10000	147794030	16s	1s
transp3	800	30000	200000	10000	93015638	24s	1s
transp4	800	40002	200000	10000	75304321	37s	1s
transp5	1000	20049	200000	10000	176263777	22s	1s
transp6	800	40002	200000	10000	124416104	34s	1s
transp7	1000	40025	200000	10000	96121936	34s	2s
transp8	1000	50055	200000	10000	92366438	51s	2s
transp9	400	10000	200000	10000	158058350	6s	0s
transp10	400	19969	200000	10000	94008769	13s	1s
transp11	600	10020	200000	9997	220335437	9s	0s
transp12	600	20000	200000	10000	126443694	15s	0s
transp13	600	30000	200000	10000	110331273	25s	1s
transp14	600	40000	200000	10000	85534936	28s	1s

Instance	Nds	Arcs	Cap	Len	Objective	CS	NW
stndrd1	200	1308	100000	9998	196587626	1s	0s
stndrd2	200	1511	100000	9998	194072029	1s	0s
stndrd3	200	2000	100000	9998	159442947	2s	1s
stndrd4	200	2200	100000	9998	138936551	2s	1s
stndrd5	200	2900	100000	9997	102950805	1s	1s
stndrd6	300	3174	150000	9996	191968577	2s	0s
stndrd7	300	4519	150000	9998	172742047	3s	0s
stndrd8	300	5168	150000	9997	164468452	4s	1s
stndrd9	300	6075	150000	9996	144994180	4s	0s
stndrd10	300	6320	150000	9996	148675665	4s	0s
stndrd16	400	1306	400000	10000	6815524469	1s	0s
stndrd17	400	2443	400000	10000	2646770386	1s	1s
stndrd18	400	1306	400000	10000	6663684919	1s	0s
stndrd19	400	2443	400000	10000	2618979806	1s	0s
stndrd20	400	1400	400000	10000	6708097873	1s	0s
stndrd21	400	2836	400000	10000	2631027973	2s	1s
stndrd22	400	1416	400000	10000	6621515104	2s	0s
stndrd23	400	2836	400000	10000	2630071408	1s	1s
stndrd24	400	1382	400000	10000	6829799687	2s	0s
stndrd25	400	2676	400000	10000	6396423129	2s	1s
stndrd26	400	1382	400000	10000	5297702923	1s	0s
stndrd27	400	2676	400000	10000	4863992745	1s	0s
stndrd28	1000	2900	1000000	9998	11599233408	6s	0s
stndrd29	1000	3400	1000000	9997	11700773092	6s	0s
stndrd30	1000	4400	1000000	9997	8782721260	6s	0s
stndrd31	1000	4800	1000000	9998	8577913734	6s	1s
stndrd32	1500	4342	1500000	9997	17996365110	13s	0s
stndrd33	1500	4385	1500000	9995	18424893900	13s	1s
stndrd34	1500	5107	1500000	9998	14596094907	11s	0s
stndrd35	1500	5730	1500000	9997	14350903861	13s	1s
stndrd36	8000	15000	4000000	10000	87957673940	329s	2s
stndrd37	5000	23000	4000000	10000	35607266430	149s	2s
stndrd38	3000	35000	2000000	10000	7265734372	84s	2s
stndrd39	5000	15000	4000000	10000	48660418428	145s	2s
stndrd40	3000	23000	2000000	10000	11068572024	62s	2s
stndrd45	4000	20000	5000	-50000	-1864582590629	16s	15s
stndrd50	350	4500	300000	100	4024557	2s	0s

22.4 Non-Weighted Matching

All computations were performed with the GOBLET graph browser 2.3 on a Pentium III/850 MHz notebook with 256 MB RAM and SuSE Linux 7.3. The test problems `r10000` to `r30000` are random graphs while `reg3` is a 3-regular random graph and `tiling1`, `tiling2` are tilings with different base graphs. The following methods have been tested here:

- "Phase": The phase ordered augmentation algorithm. We report the running times and the number of phases which occur.
- "Cancel": The cycle canceling method. We report the running times and the number of odd cycles after the call of `CancelEven`.

Note that the respective numbers of phases and odd cycles are much less than the worst-case bounds may suggest.

Instance	Nodes	Arcs	Type	Phase	Cancel	Objective
r10000	10000	10000	1-factor	2s (13)	1s (0)	3932
			2-factor	1s (7)	1s (0)	6815
r15000	10000	15000	1-factor	4s (24)	4s (0)	4634
			2-factor	4s (22)	4s (0)	8488
r20000	10000	20000	1-factor	2s (15)	4s (6)	4896
			2-factor	2s (15)	4s (6)	9407
r25000	10000	25000	1-factor	1s (12)	4s (8)	4963
			2-factor	2s (11)	4s (6)	9755
r30000	10000	30000	1-factor	1s (9)	4s (2)	4989
			2-factor	2s (9)	4s (2)	9903
reg3	10000	15000	1-factor	1s (10)	2s (0)	5000
			2-factor	1s (11)	2s (2)	10000
tiling1	10166	30361	1-factor	2s (3)	1s (2144)	5083
			2-factor	1s (5)	2s (34)	10166
tiling2	9941	29540	1-factor	1s (2)	2s (70)	4970
			2-factor	2s (3)	3s (132)	9941
Average	10013	21863	1-factor	1.8s (11)	4.6s (279)	
			2-factor	1.8s (10.4)	3s (23)	

22.5 Weighted Matching

All computations were performed with the GOBLET graph browser 2.2 on a Pentium III/850 MHz notebook with 256 MB RAM and SuSE Linux 7.3. The test problems are from TSPLIB and defined on complete graphs. The instances `pr1002` and `u1060` are geometric while `si1032` is defined by a matrix. The problem `rnd1000` is a matrix problem with random length labels equally distributed in the interval $[0, 49999]$.

The following methods have been tested here:

- "heuristic": The problem is solved on a sparse subgraph only where `methCandidates=10`.
- "candidates": The fractional matching problem is solved on a candidate graph with `methCandidates=10` converted into a optimal fractional matching on the entire graph, and then converted into a optimal integral solution.
- "exhaustive": The matching solver is applied to the complete graph directly, that is with `methCandidates=-1`.

The results indicate that the candidate graph is constructed slowly, but provides excellent solutions. The price&repair strategy for the fractional matching problem cannot reach the performance of price&repair methods for the 1-matching problem. The running times of the price&repair method strongly depend on the performance of the primal-dual method since the number of expensive PD-operations does not decrease via candidate search. Note the significant differences in the running times for the geometric and the matrix problems.

Instance	Type	Method	Objective	Time	Dual	Expand
pr1002	1-factor	heuristics	112630	84s	583	189
pr1002	1-factor	candidates	112630	1263s	250	120
pr1002	1-factor	complete	112630	5224s	248	119
pr1002	2-factor	heuristics	244062	104s	419	87
pr1002	2-factor	candidates	244062	2029s	428	106
pr1002	2-factor	complete	244062	5435s	428	106
si1032	1-factor	heuristics	45448	39s	6	0
si1032	1-factor	candidates	45448	60s	6	0
si1032	1-factor	complete	45448	1019s	7	0
si1032	2-factor	heuristics	91940	59s	65	10
si1032	2-factor	candidates	91939	562s	93	8
si1032	2-factor	complete	91939	2163s	76	10
u1060	1-factor	heuristics	100651	98s	590	130
u1060	1-factor	candidates	100356	1898s	519	89
u1060	1-factor	complete	100356	3461s	519	80
u1060	2-factor	heuristics	210931	123s	409	77
u1060	2-factor	candidates	210931	1439s	405	94
u1060	2-factor	complete	210931	5372s	410	89
rnd1000	1-factor	heuristics	41284	56s	0	0
rnd1000	1-factor	candidates	41284	937s	69	0
rnd1000	1-factor	complete	41284	2628s	69	0
rnd1000	2-factor	heuristics	103401	77s	42	4
rnd1000	2-factor	candidates	103282	274s	22	0
rnd1000	2-factor	complete	103282	2752s	24	0

22.6 Cliques and Node Colouring

All computations were performed with the GOBLET graph browser 2.1d/2.2a/2.3c on a Pentium III/850 MHz notebook with 256 MB RAM, SuSE Linux 7.2/7.3 and without any code optimization. The test sets are from Michael Tricks graph colouring page

`http://mat.gsia.cmu.edu/COLOR/instances.html`

All computation times were restricted to 5 minutes (exceptions are marked with an asterisk *). Note that node k-colourings and k-clique covers have

been computed for a series of fixed, decreasing k. A successful k-colouring usually takes less than one second, times for negative results mainly depend on the branch and bound configuration.

For the series `fpsol*`, `inithx*` and `le450*`, the k-colour enumeration scheme requires too much computer memory to obtain the optimal colouring. Moreover, the computation times for cliques and clique covers are dominated by the explicit construction of complementary graphs. Results are therefore omitted.

Instance	Nodes	Arcs	Clique	Colour	Stable	Cover
anna	138	586	11	11	80	80
david	87	812	11	11	36	36
homer	561	3258	13	13	341	341
huck	74	602	11	11	27	27
jean	80	508	10	10	38	38
DSJC125.1	125	736	4	[5, 6]	[32, 48]	[32, 48]
DSJC125.5	125	3891	10	[10,21]	10	[10, 20]
DSJC125.9	125	6961	[32,46]	[32,46]	4	[4, 6]
flat300_20	300	21375	[10,40]	[10,41]	[12, 43]	[12, 44]
fpsol2.i.1	496	11654	[45,65]	[45,65]	307	307
games120	120	1276	9	9	[22, 24]	[22, 24]
le450_5a	450	5714	[5, 8]	[5, 8]	[78,141]	[78,141]
miles250	128	774	8	8	44	44
miles500	128	2340	20	20	18	[18, 19]
miles750	128	4226	31	31	12	12
miles1000	128	6432	42	42	8	8
miles1500	128	10396	73	73	5	5
mulsol.i.1	197	3925	49*	49	100	100
mulsol.i.2	188	3885	31	31	90	90
mulsol.i.3	184	3916	31	31	86	86
mulsol.i.4	185	3946	31	31	86	86
mulsol.i.5	186	3973	31	31	88	88
myciel3	11	20	2	4	5	6
myciel4	23	71	2	5	11	12
myciel5	47	236	2	[4, 6]	23	24
myciel6	95	755	2	[4, 7]	47	48
myciel7	191	2360	2	[4, 8]	95	96
queen5_5	25	320	5	5	5	5
queen6_6	36	580	6	7	6	6
queen7_7	49	952	7	7*	7	7
queen8_8	64	1456	8	[8,10]*	8	8
queen9_9	81	2112	9	[9,11]*	9	9
queen8_12	96	2736	12	[12,14]	8	8
queen10_10	100	2940	10	[10,13]*	10	10
queen11_11	121	3960	11	[11,15]	11	11
queen12_12	144	5192	12	[12,17]	12	12
queen13_13	169	6656	13	[13,17]*	13	13
queen14_14	196	8372	14	[14,20]	14	14
queen15_15	225	10360	15	[15,21]	15	15
queen16_16	256	12640	16	[16,22]	16	16
school1	385	19095	14	14	[40, 48]	[40, 47]
school1_nsh	352	14612	[14,17]	[14,17]	[37, 47]	[37, 47]
zeroin.i.1	211	4100	49	49	120	120
zeroin.i.2	211	3541	30	30	127	127
zeroin.i.3	206	3540	30	30	123	123

Index

2-edge connected component, 119
T-join, 135
b-flow, 124, 127
    ϵ-optimal, 128
r-tree, 118
st-flow, 127
    (ν)-optimal, 127
    extreme, 127
    maximum, 122
st-numbering, 120
st-orientations, 62
st-path
    eligible, 112
1-matching, 95
2-factor, 94

arc incidences, 91
artificial nodes, 93
auxiliary variables, 174

backward arcs, 40
balanced network search (BNS), 131
balanced pseudo-flow, 132
basic timer, 188
basis, 177
basis arc, 121
basis row, 177
bipolar digraphs, 60
block, 119
blossom, 99
    base, 99
branch and bound, 81
    branch node, 81
    left successor, 82
    right successor, 82
branch tree, 85
breakpoint, 156
bridge, 119

canonical element, 74
canonically ordered partition, 121
child timers, 188
Chinese postman problem (CPP), 135
circulation, 124
circulations, 127
clique, 141
clique cover, 140
code module, 190
combinatorial embedding, 120
combinatorially embedded, 49
complementary graph, 60
complementary pairs, 39, 40

complete orientation, 62
concrete classes, 43
configuration file, 207
contact nodes, 121
container
    dynamic, 72
    static, 72
copy constructor, 147
cut edges, 121
cut node, 119
cycle free solution, 130
cycle space, 136

DAG, 115
Dantzig rule, 130
data structures, 71
default constructor, 147
degenerate pivot steps, 129
dense implementation, 46
dictionary, 76
directed dual graphs, 60
double depth first search, 55
dual graph, 60
dual update, 56

ear decomposition, 119
edge connectivity number, 125
elementary operation, 39, 74
eligible arc, 112
Eulerian cycle, 135
excess scaling, 124
exterior face, 121

faces, 50
first-in first-out principle, 73
flow value, 122
force directed, 106
forward arcs, 40

general position, 108
geometric embedding
    dimension, 93
geometric optimization instances, 103
GIOTTO, 107
global timers, 188
GOBLIN file
    token, 194
    tuple, 193
      type, 193
    vector, 196
graph
    Eulerian, 135
graph drawing, 103

hamiltonian cycle, 137
hash function, 75
hash table, 75
    collisions, 75

implementation classes, 43
index set
    ivalidation, 80
induced subgraph, 64
inner nodes, 39
internally triconnected, 121
invalidated, 101

Kandinski, 107

last-in first-out principle, 73
layout model, 103
Layout models, 103
line graph, 58
linear program, 174

maximum cut, 142
metric closure, 61
modified length labels, 56
multiple partial pricing, 130

network programming problem with side constraints, 198
network programming problems, 11
node
    balanced, 122
node adjacencies, 91
node capacities, 125
node colouring, 87, 140
    active node, 87
    dominated node, 87
node incidence
    predecessor, 48
    successor, 44
node splitting, 62

objective function, 198
odd cycle canceling problem, 57
open ear decomposition, 120
outer nodes, 39
outerplanar graph, 121

partial pricing, 130
persistent, 46
persistent object, 46
personal installation, 17
phase of the Dinic method, 52, 56
pivot arc, 129
pivot cycle, 129
planar, 49
planar graph, 120
planar line graphs, 59
pricing rule, 130
primal algorithms, 127
priority, 73
problem relaxation, 81
problem variables, 81
Proportional growth, 107
pseudo-flow, 95
push and relabel method
    active node, 123

regions, 50
return arc, 120
root node, 81

SAP algorithm, 127
segments, 120
semaphores, 167
shortest path tree, 112
shrinking family
    real items, 75
    virtual items, 75
sparse implementation, 46
stable set, 141
Steiner nodes, 142
Steiner tree, 142
strong component, 119
strongly connected node pair, 119
strongly feasible spanning tree structures, 129

structural restrictions, 174
subgraph, 95
    cardinality, 95
    infeasible, 95
    non-optimal, 95
    weight, 95
successor, 44
system installation, 17

templates, 72
terminals, 142
thread safe, 167
topological erasure, 52
tracing point, 156
transitive arcs, 64
transscript, 207
traveling salesman problem (TSP), 137
triangular graph, 24, 66
triangulations, 121

union-find process, 74

valid path, 131
value
    st-flow, 127
variable range restrictions, 174
vertex connectivity number, 125
vertex cover, 141
Visibility representations, 107
Voronoi regions, 113

weight, 142