Introduction to Search

- Search is one of the most powerful approaches to problem solving in AI
- Search is a universal problem solving mechanism that
  - Systematically explores the **alternatives**
  - Finds the sequence of steps **towards a solution**
- Problem Space Hypothesis All goal-oriented symbolic activities occur in a problem space
  - Search in a problem space is claimed to be a completely general model of intelligence

Introduction to Search:
Popular Classical Problem Domains

- 8-puzzle
- Tower of Hanoi
- Missionaries and Cannibals
- Water Jug
- Vacuum World
- Wumpus World
- Block World
- Travelling Salesperson
- Maze
- Crossword Puzzle
- Crypt-arithmetic
- Wheel of Fortune
- Chess, Bridge, etc

Knowledge & Problem Types

Vacuum World Domain as an illustration

- Let the world consist of only **2 locations** - Left and Right Box
- Each location may contain **dirt**
- The agent may be in **either box**
- There are **8 possible states**
- The agent can have **3 possible actions** - Left, Right and Suck

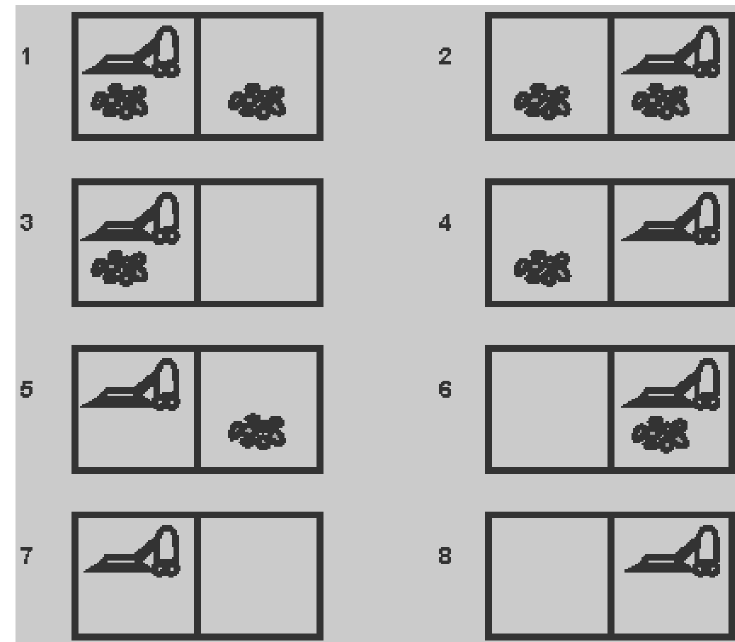


Fig.4.1: The 8 possible states of a Vacuum World

Problem Formulation

- **Initial State:** State know to be currently in.
  - From the current state, **what state is next?**
- *Querying* possible next states:

```
[ NEXT_STATES ] = successors( CURRENT_STATE )
```

- *Moving* to the next state:

```
CURRENT_STATE = operator ( CURRENT_STATE,
NEXT_STATE )
```

- **State space:** Set of all states reachable from the *initial state* by any sequence of actions.
- **Path:** A sequence of states in the *state space*.

Problem Formulation (Contd)

- **Path cost:** Expense to incur when getting from the *initial state* to another (probably non-neighboring) state.
- **Path cost function:** Function to determine the cost of a *path*.
- **Goal test function:** Function to determine if the current state is the final (i.e., goal) state.

- Problem solving = Initial state + Operators + Goal test function + Path cost function

Problem-solving process

Solving a problem:

- Is a solution possible?
- If a solution is found, Is such a solution a good one?
- Was it expensive to find this solution?

Search Cost: Expense to incur when finding a specific solution.

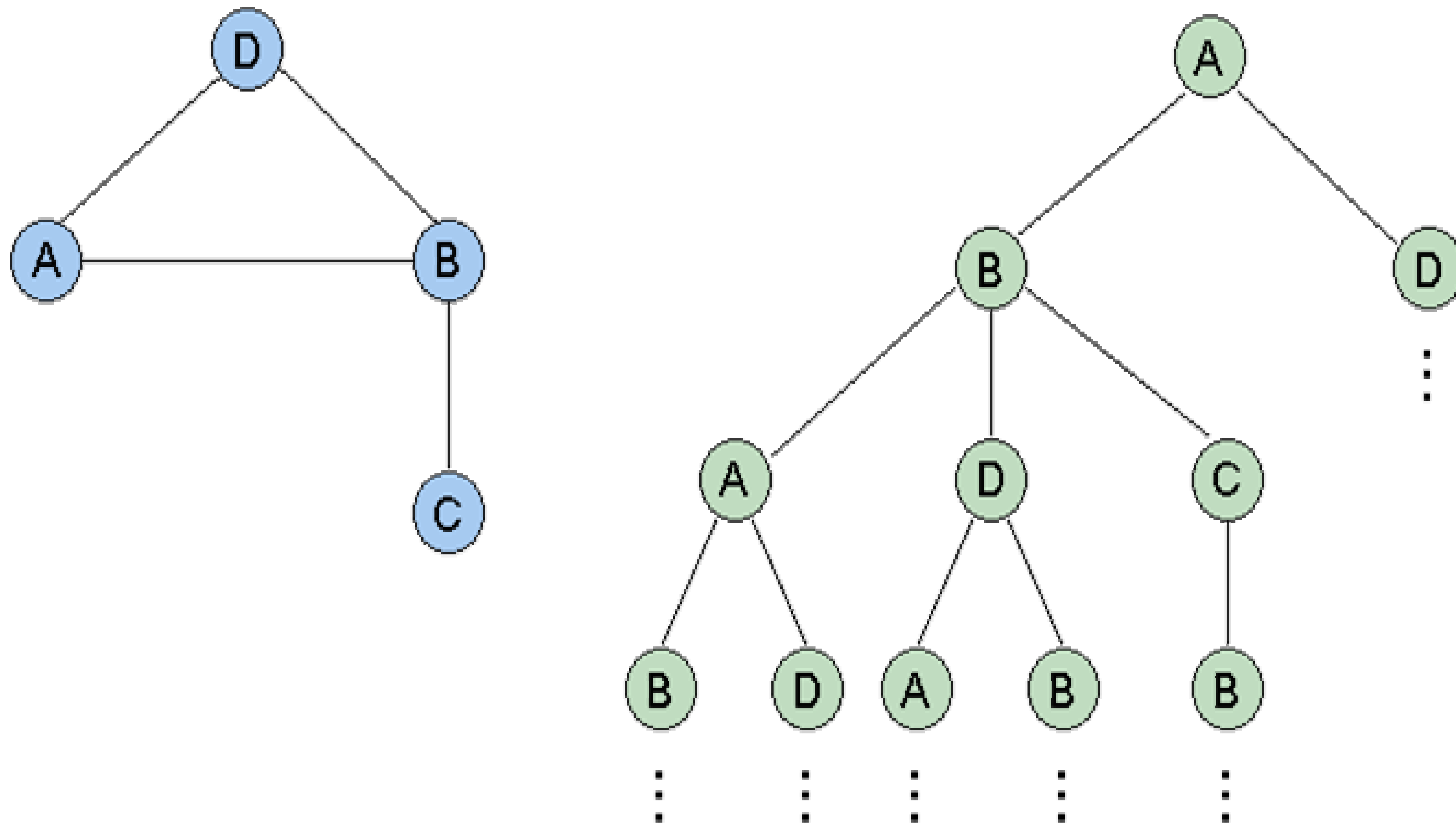
Total Cost: Cost resulting from adding the *search cost* and the *path cost*.

There is a trade-off between these two costs. Given a solution, How much resources (e.g., time, memory) does it take to find this solution? (i.e., search cost), versus How expensive is to carry out the solution? (i.e., path cost)

Defining the Search Problem

- Starting from an initial state, the aim of a search is *to maintain and extend a set of partial solution sequences of states*.
- Essentially, searches choose one option and put aside other options for later consideration.
- The choice of which state to try first is determined by a *search strategy*.
- Search strategies result in the building of *search trees*.
- What are the differences between a *search tree* and a *state space*?

  On the one hand, a state space is the network of all state relationships in a problem, while a tree search is a network of all possible paths that result from the dependencies of a state space.



Example of a *state space* and its corresponding *search tree*.

A **Node:** Represents a state from the state space.
The **Root:** First node in a search tree. It represents the initial state in the state space.

Knowledge & Problem Types

State Space for Vacuum World as a SS & MS problem

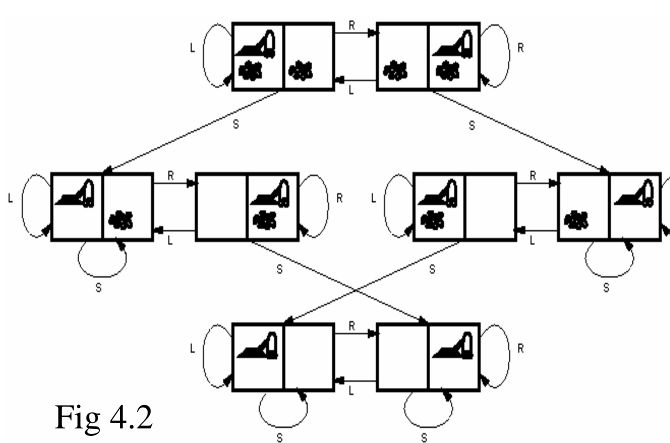


Fig 4.2

Single State Search Space ↖

Multiple-State Search Space ↗

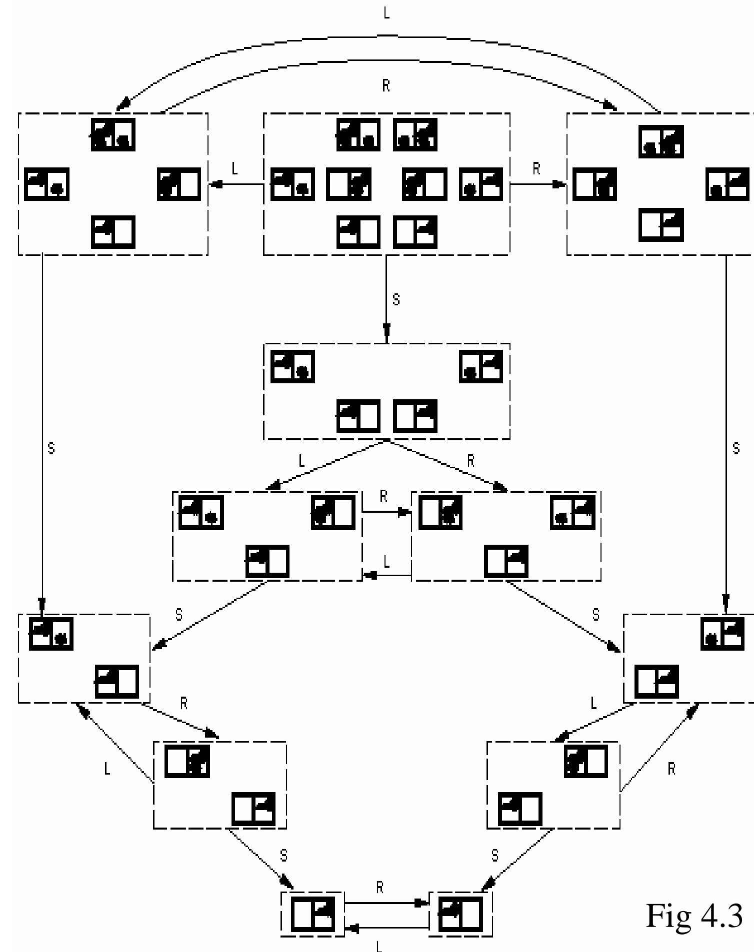
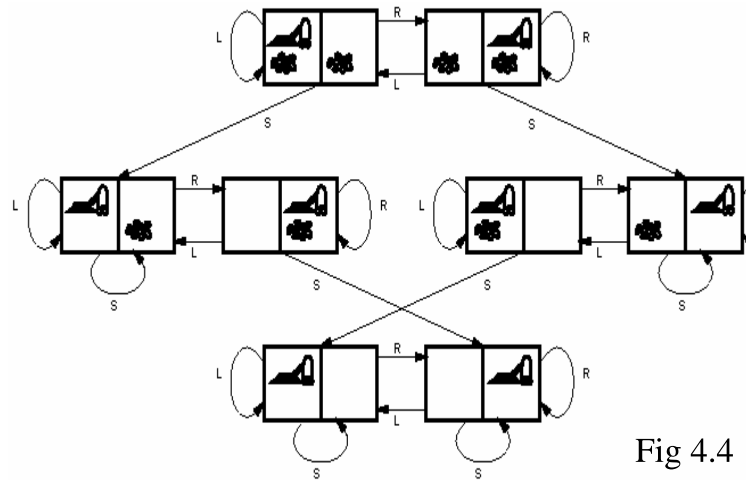


Fig 4.3

Toy Problems
(1) Vacuum World as a Single-state problem



- **Initial State**: one of the 8 states shown above.
- **Operators** move Left, move Right, Suck.
- **Goal Test**: no dirt in any square.
- **Path cost**: each action costs 1.

Toy Problems

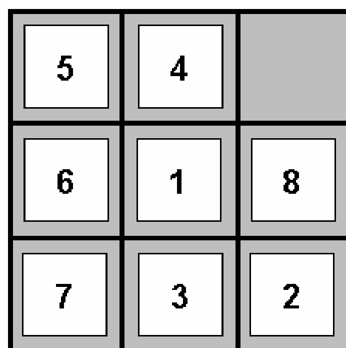
(2) Vacuum World as a Multiple-state problem

Fig 4.5

- **State sets**: subsets of state 1-8 shown in Fig 4.1 or Fig 4.2
- **Operators** move Left, move Right, Suck.
- **Goal Test**: all states in state set have no dirt.
- **Path cost**: each action costs 1.

Toy Problems

(3) 8-puzzle problem



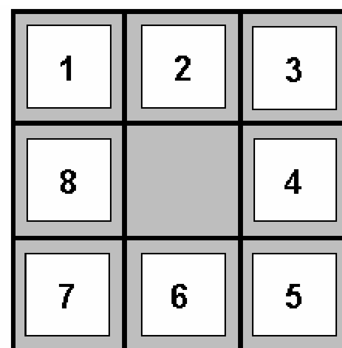


Fig 4.6

- **Initial State**: The location of each of the 8 tiles in one of the nine squares
- **Operators**: blank moves (1) Left (2) Right (3) Up (4) Down
- **Goal Test**: state matches the goal configuration
- **Path cost**: each step costs 1, total path cost = no. of steps

Toy Problems
(4) 8-queens problem

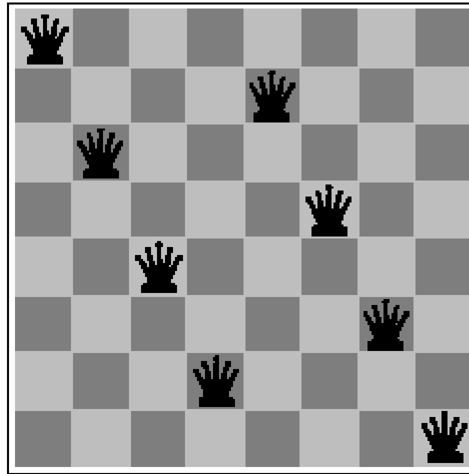


Fig 4.7

- **Initial State**: Any arrangement of 0 to 8 queens on board.
- **Operators**: add a queen to any square.
- **Goal Test**: 8 queens on board, none attacked.

Toy Problems

(5) Crypt arithmetic

```
 FORTY          Solution:  29786      F=2, O=9, R=7, etc
+  TEN                       850
+  TEN                       850
 -----                     -----
 SIXTY                     31486
```

Fig 4.8

- **Initial State**: a crypt arithmetic puzzle with some letters replaced by digits.
- **Operators**: replace all occurrences of a letter with a non-repeating digit.
- **Goal Test**: puzzle contains only digits, and represents a correct sum.

Real-world Problems

- **Route Finding** - computer networks, automated travel advisory systems, airline travel planning.
- **VLSI Layout** - A typical VLSI chip can have as many as a million gates, and the positioning and connections of every gate are crucial to the successful operation of the chip.
- **Robot Navigation -** rescue operations
- **Mars Pathfinder** - search for Martians or signs of intelligent life forms
- **Time/Exam Tables**

Problem solving

- A problem consists of a triple {Initial-state, operators, goal-state}
  - operators:
    - » Move-blank-up,
    - » move-blank-down,
    - » move-blank-left
    - » move-blank-right
  - Initial state- unique
  - goal-state- partial description
- A Solution consists of
  - A unique goal state
  - A sequence of operators that transform the initial state into the goal state.
    - » (move-blank-down move-blank-right)

Initial State

1	2	3
4		6
7	5	8

1	2	3
4	5	6
7		8

1	2	3
4	5	6
7	8	

Goal State

Farmer Wolf Goat and Cabbage problem

- A farmer has a wolf, a goat, and a cabbage on the east side of the river.
- He wants to get them on the west side of the river
- He has a boat in which he and only one other thing may fi
- The wolf will eat the goat if they are left together unattended
- The goat will eat the cabbage if they are left together unattended
- State representation- (f, w, g, c)
  - f: the side of the farmer (east or west)
  - w:the side of the wolf
  - g: the side of the goat
  - c: the side of the cabbage

Water Jug Problem

- You have a 4-gallon and a 3-gallon water jug
- You have a faucet with an unlimited amount of water
- You need to get exactly 2 gallons in 4-gallon jug

- State representation: (x, y)
  - x: Contents of four gallon
  - y: Contents of three gallon
- Initial state: (0, 0)
- Goal state (2, n)
- Operators
  - Fill 3-gallon from faucet, fill 4-gallon from faucet
  - Fill 3-gallon from 4-gallon , fill 4-gallon from 3-gallon
  - Empty 3-gallon into 4-gallon, empty 4-gallon into 3-gallon
  - Dump 3-gallon down drain, dump 4-gallon down drain

General Search Problem

General Search Problem

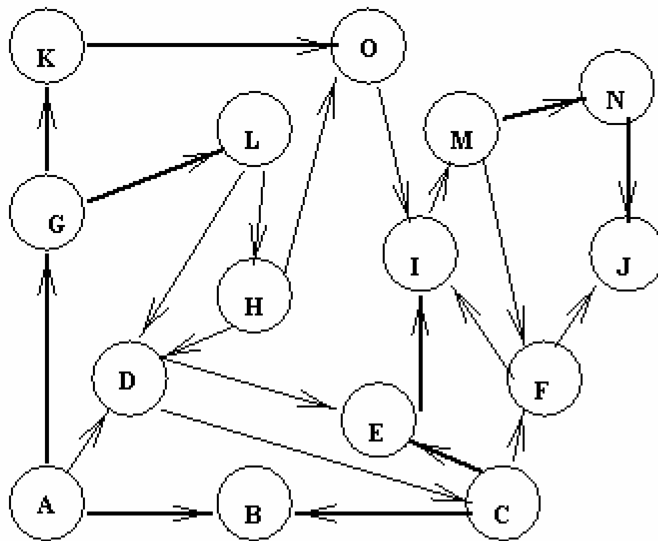


Fig 4.9

Search tree representation:

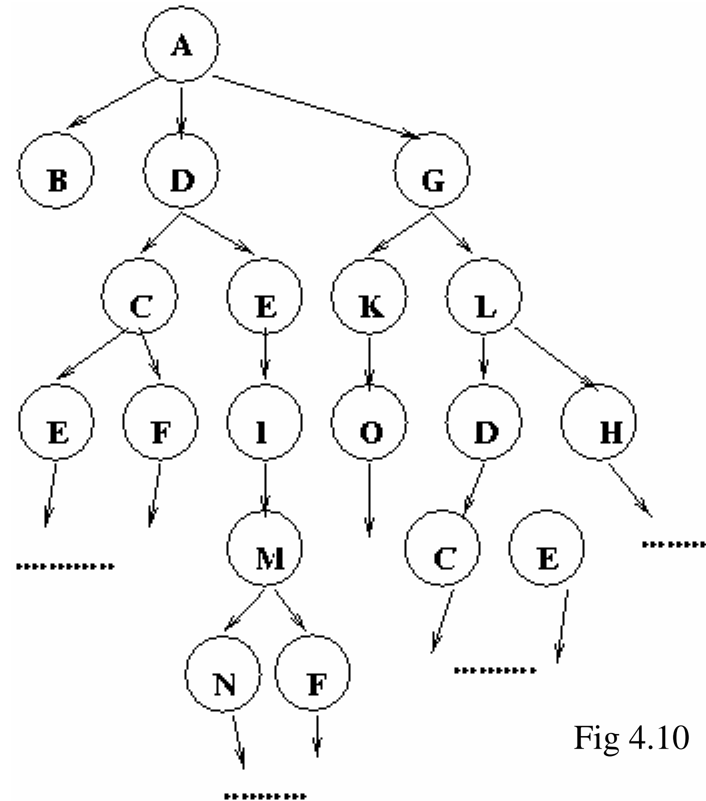


Fig 4.10

Criteria for Evaluating Search Strategies

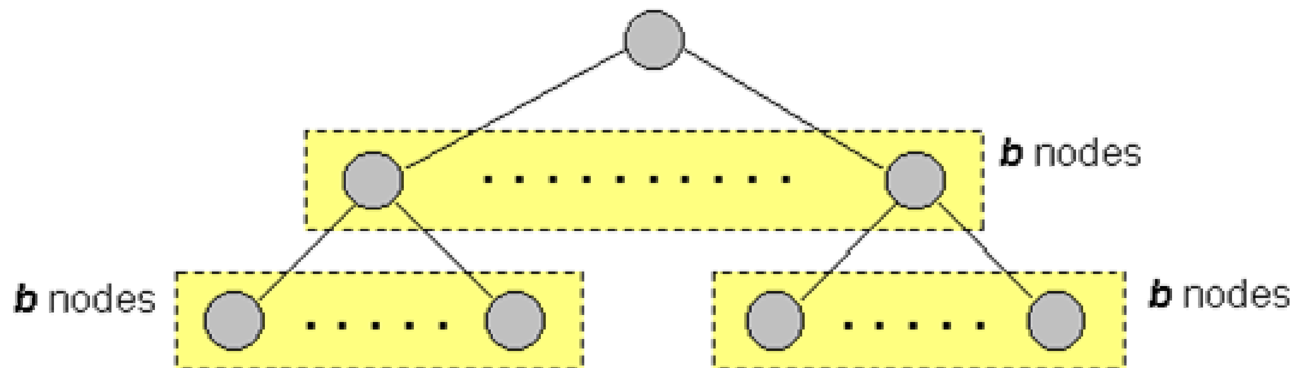
Each of the search strategies are evaluated based on:

- **Completeness**: is the strategy **guaranteed** to find a **solution** when there is one?
- **Time complexity**: how **long** does it take to find a solution
- **Space complexity**: how much **memory** does it need to perform the search?
- **Optimality**: does the strategy find the **highest-quality solution** when there are **several solutions**?

Branching factor: The amount of nodes that are expanded at any given node in a hypothetical search tree.

Assuming a branching factor of ***b***, the number of nodes that are expanded at a depth ***d*** is given by the formula $1 + b + b^2 + b^3 + ... + b^d$.

The branching factor is used for defining the time and space complexity of different strategies.



Search tree of depth ***d***=2 and branching factor ***b***

Uninformed Searches

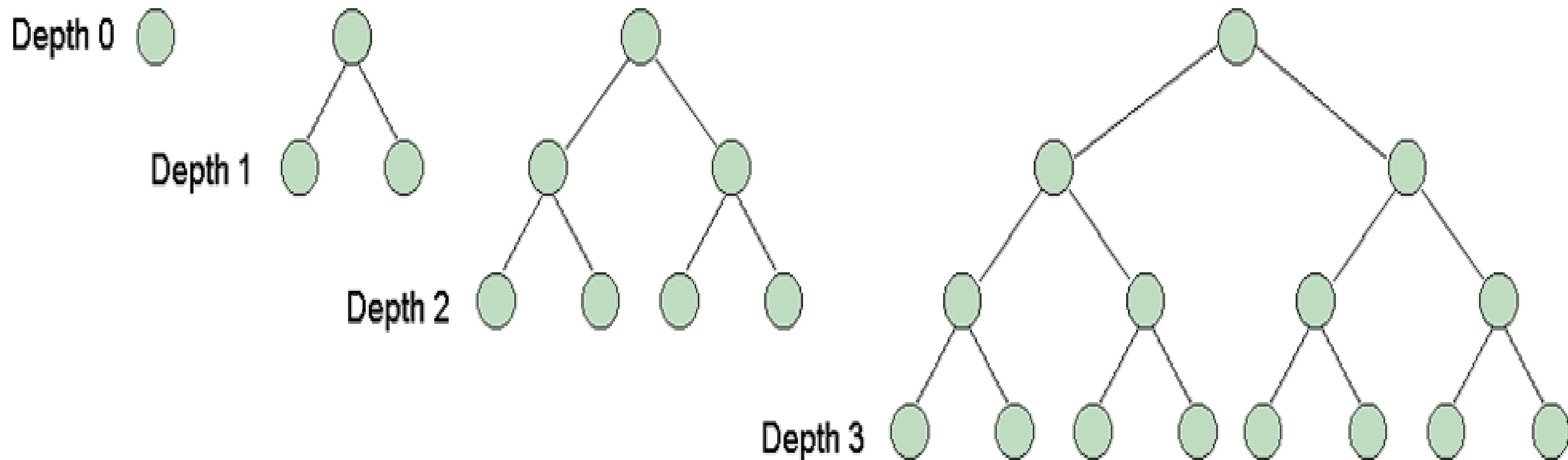
Uninformed searches, differently from informed searches, do not use state information to decide the order on which nodes are expanded. Instead, they rely on the arbitrary order on which nodes are found.

There are six uninformed searches divided into three main groups:

1. Breadth-First search
   Uniform Cost search
2. Depth-First search
   Depth-Limited search
   Iterative Deepening search
3. Bidirectional search

Breadth-First Search

1. Nodes are expanded by levels.
2. All expanded nodes are maintained in memory.



BS1. Breadth-First Search (cont)

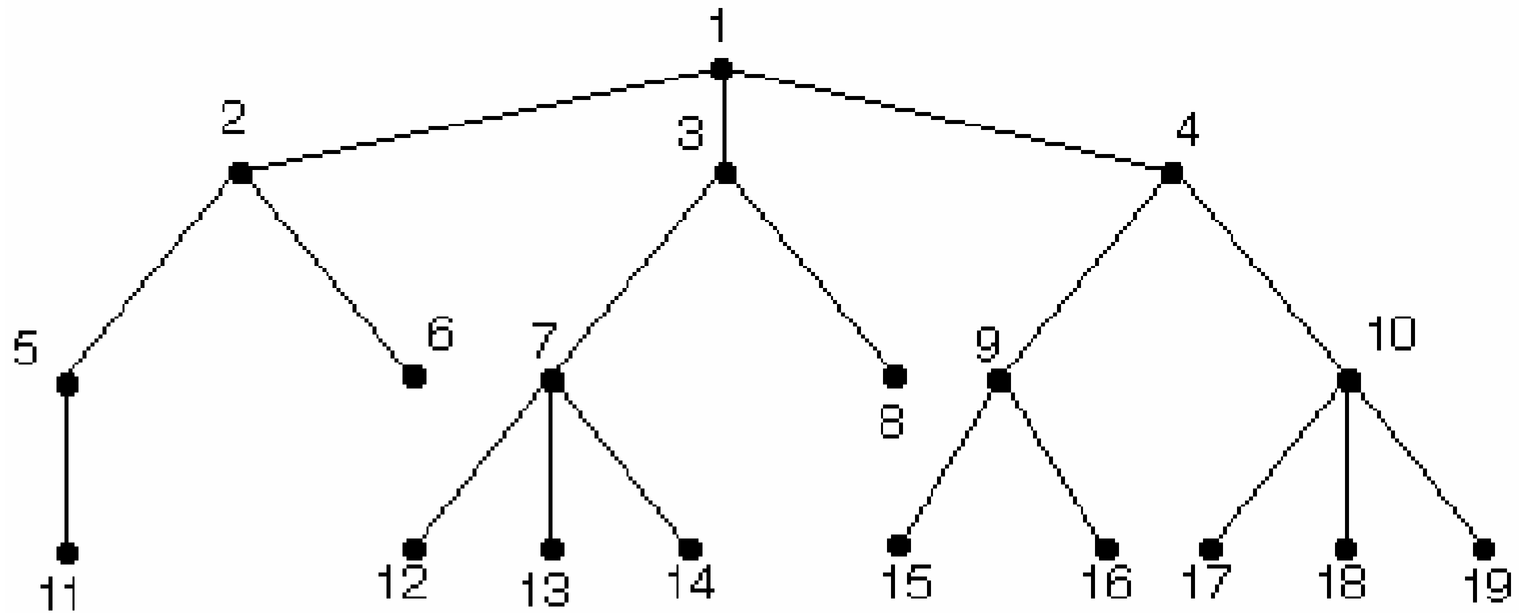
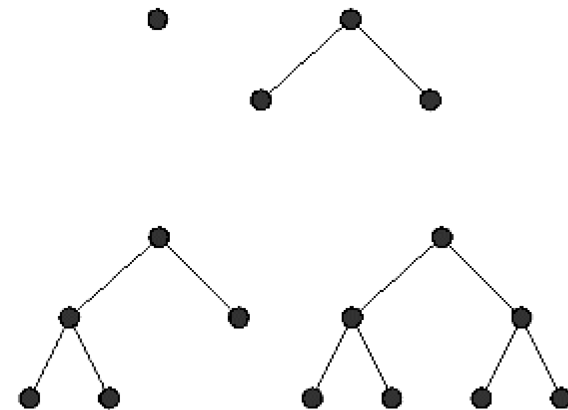


Fig 4.13 Breadth-first Tree Search (Numbers refer to order visited in search)

```
% call breadthfirst procedure, Solution is returned
% as a reverse path from Goal State to Start State

solve(StartState, Solution) :-
                    breadthfirst([[StartState]], Solution).

% first describe terminating case
% Current Node (Frontier of Search) is  the Goal Node

breadthfirst([[Current|Path]|_], [Current|Path]) :-
                                        goal(Current).

% Now describe the recursive case
% The level we're at doesn't contain the goal node, so
% Extend the level in all possible ways at one

breadthfirst([FirstPath|RestPaths], Solution) :-
```

BS1. Breadth-First Search

- One of the simplest search strategy
- **Time** and **Space complexity**
- Cannot be use to solve any but the **smallest problem**, see next page for a simulation.



- **Completeness**: Yes
- **Time complexity**: b^d
- **Space complexity**: b^d
- **Optimality**: Yes

(*b - branching factor, d - depth)*

Fig 4.11 Breadth-first search tress after 0, 1, 2, and 3 node expansions (b=2, d=2)

BS1. Breadth-First Search (cont)

Depth	Nodes	Time	Memory
0	1	1 millisecond	100 bytes
2	111	.1 seconds	11 kilobytes
4	11,111	11 seconds	1 megabyte
6	10^6	18 minutes	111 megabytes
8	10^8	31 hours	11 gigabytes
10	10^{10}	128 days	1 terabyte
12	10^{12}	35 years	111 terabytes
14	10^{14}	3500 years	11,111 terabytes

Fig 4.12

- Time and Memory requirements for a breadth-first search.
- The figures shown assume (1) branching factor b=10; (2) 1000 nodes/second; (3) 100 bytes/node

BS2. Uniform Cost Search

- BFS finds the ***shallowest*** goal state.
- Uniform cost search modifies the BFS by **expanding ONLY the lowest cost node** (as measured by the path cost *g(n)*)
- The **cost of a path** must **never decrease** as we traverse the path,
  ie. no negative cost should in the problem domain
- **Completeness**: Yes
- **Time complexity**: b^d
- **Space complexity**: b^d
- **Optimality**: Yes

BS2. Uniform Cost Search (cont)

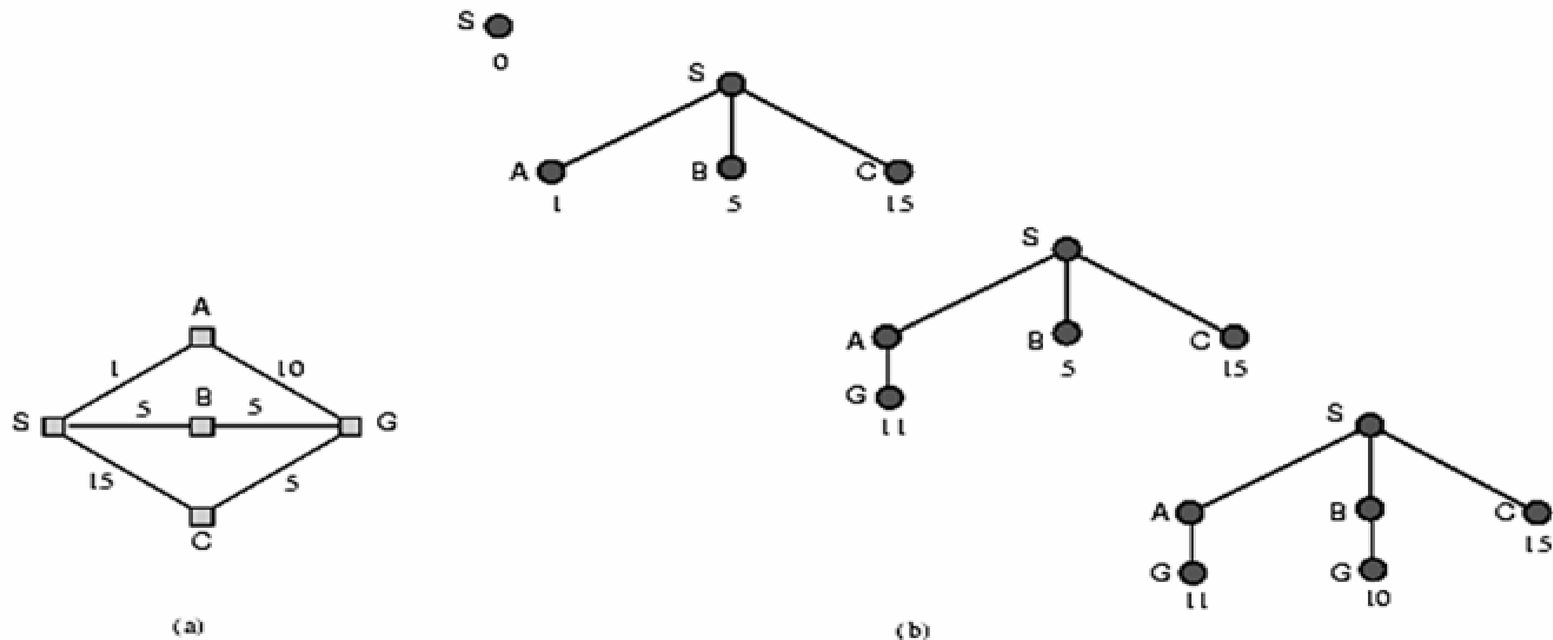


Fig 4.14

- A route finding problem. **(a)** The state space, showing the cost for each operator. **(b)** Progression of the search. Each node is labeled with a numeric path cost *g(n)*. At the final step, the goal node with g=10 is selected

BS3. Depth-First Search

- DFS always **expands one** of the nodes at the deepest level of the tree.
- The search **only go back** once it **hits a dead end** (a nongoal node with no expansion)
- DFS have **modest memory requirements**, it only needs to store a single path from root to a leaf node.
- Using the sample simulation from Fig 4.12, at depth d=12, **DFS** only requires **12 kilobytes** instead of **111 terabytes** for a **BFS** approach.
- For **problems that have many solutions**, **DFS** may actually be faster than BFS, because it has a good chance of finding a solution after exploring only a small portion of the whole space.

BS3. Depth-First Search (cont))

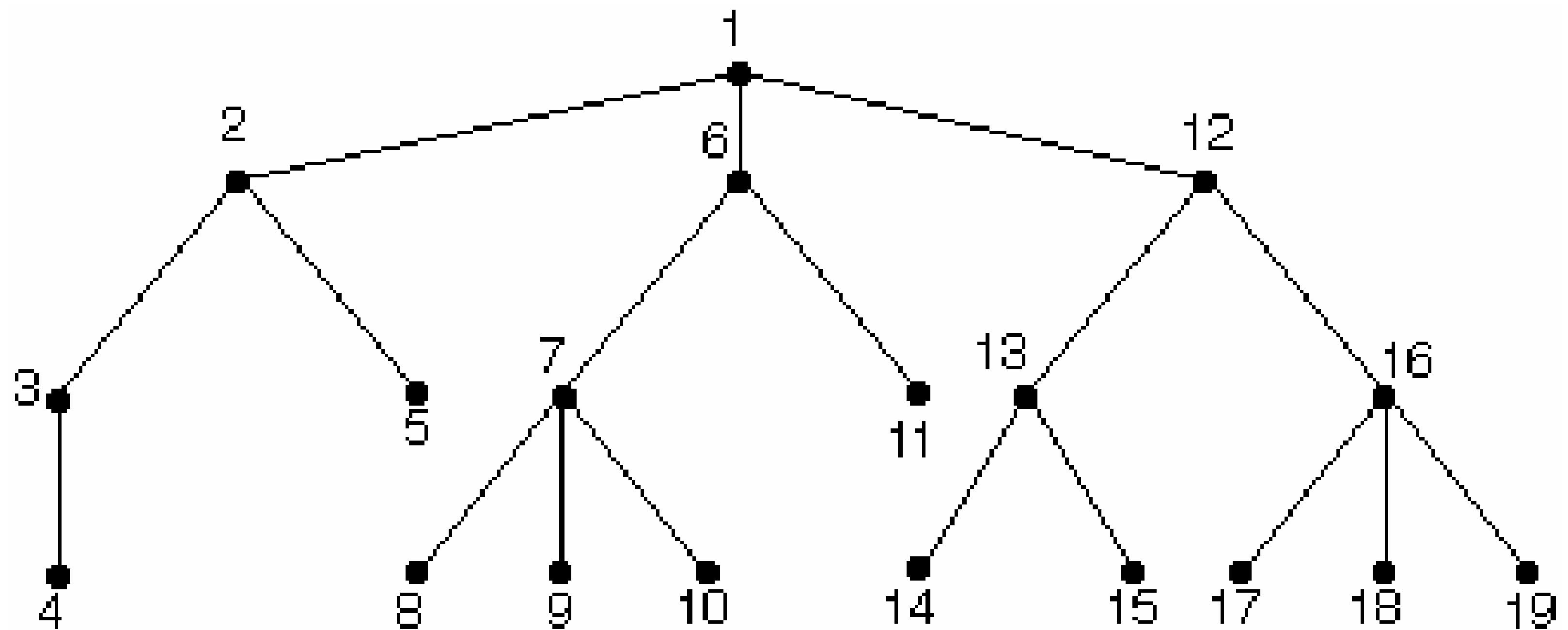


Fig 4.15 Depth-first Tree Search

BS3. Depth-First Search (cont)

- One problem with DFS is that it can get **stuck** going down the wrong path.
- Many problems have **very deep** or even **infinite** search trees.
- DFS should be **avoided** for search trees with **large** or **infinite maximum depths**.
- It is common to implement a **DFS** with a **recursive function** that calls itself on each of its children in turn.

- **Completeness**: No
- **Time complexity**: b^m
- **Space complexity**: bm
- **Optimality**: No *(b-branching factor, m-max depth of tree)*

BS4. Depth-Limited Search

- "**Practical**" DFS
- DLS avoids the pitfalls of DFS by imposing a cutoff on the **maximum depth** of a path.
- However, if we choose a depth limit that is too small, then DLS is not even complete.
- The time and space complexity of DLS is similar to DFS.

- **Completeness**: Yes, if $l >= d$
- **Time complexity**: b^l
- **Space complexity**: bl
- **Optimality**: No *(b-branching factor, l-depth limit)*

BS4. Depth-Limited Search (cont)

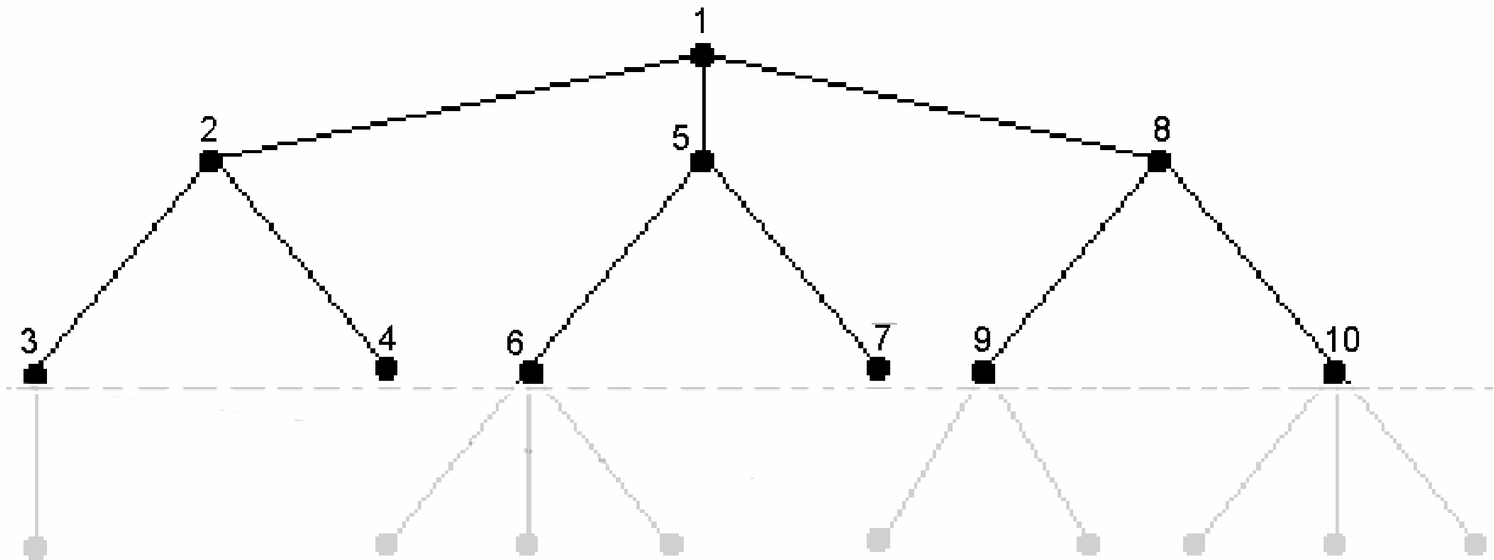


Fig 4.16

- Depth-first search trees for binary search tree. Same problem as Fig 4.15
- Depth limit, $dl = 2$

BS5. Iterative Deepening Search

- The **hard part** about DLS is **picking a good limit**.

- IDS is a strategy that sidesteps the issue of choosing the best depth limit by **trying all possible depth limits**: first depth 0, then depth 1, the depth 2, and so on.

- In effect, it combines the **benefits of DFS and BFS**.

- It is **optimal** and **complete**, like BFS and has **modest memory requirements** of DFS.

BS5. Iterative Deepening Search (cont)

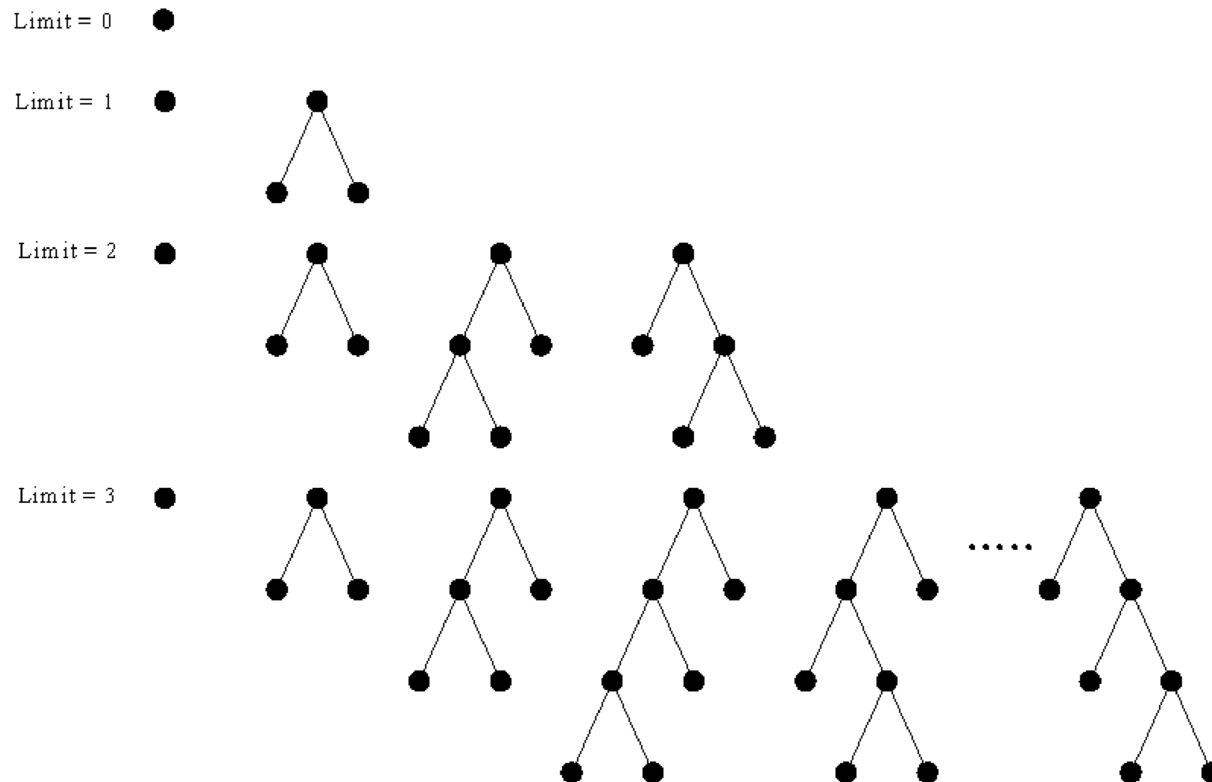


Fig 4.17 Four iterations of iterative deepening search on a binary tree

BS5. Iterative Deepening Search (cont)

- IDS may seem wasteful, because so many states are expanded multiple times.
- For most problems, however, the **overhead** of this multiple expansion is actually **rather small**.
- **IDS** is the **preferred** search method when there is a **large search space** and the **depth** of the solution is **not known**.

- **Completeness**: Yes
- **Time complexity**: b^d
- **Space complexity**: bd
- **Optimality**: Yes

BS6. Bi-directional Search

- **Search forward** from the Initial state
- And **search backwards** from the Goal state..
- Stop when two meets in the **middle**.

- **Completeness**: Yes
- **Time complexity**: $b^{d/2}$
- **Space complexity**: $b^{d/2}$
- **Optimality**: Yes

BS6. Bi-directional Search (cont)

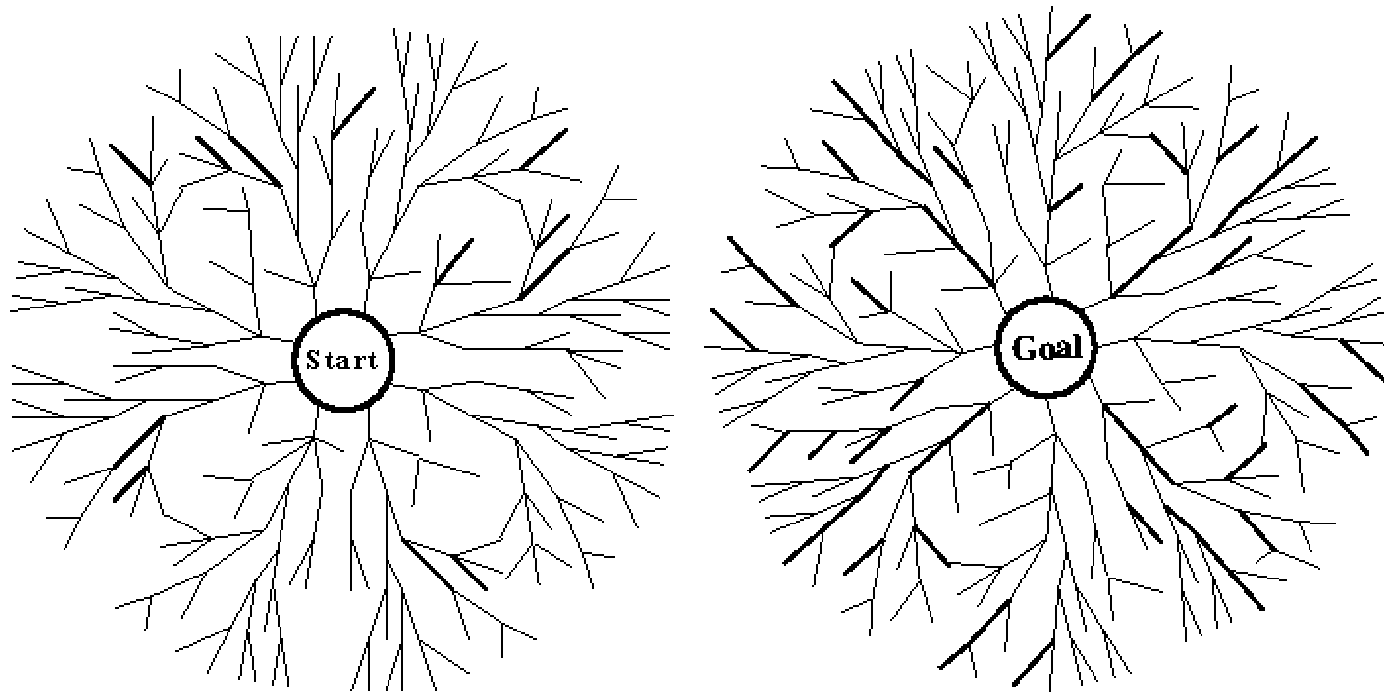


Fig 4.18 A schematics view of a bi-directional BFS that is about to succeed, when a branch from the start node meets a branch from the goal node

Comparing Blind Search Strategies

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Time	b^d	b^d	b^m	b^l	b^d	$b^{d/2}$
Space	b^d	b^d	bm	bl	bd	$b^{d/2}$
Optimal?	Yes	Yes	No	No	Yes	Yes
Complete?	Yes	Yes	No	Yes, if $l \geq d$	Yes	Yes

Fig 4.19

- Comparison of 6 search strategies in terms of the 4 evaluation criteria set forth in "Criteria for Evaluating Search Strategies"
- *b* - branching factor; *d* is the depth of the solution; *m* is the maximum depth of the search tree; *l* is the depth limit