Propositional Logic

- A denumerable set of *propositions*, $P$.
- Formulae $\varphi$ over propositions:

\[
\varphi \rightarrow P \mid \varphi \land \varphi \mid \varphi \lor \varphi \mid \neg \varphi \ldots
\]

- 2-valued interpretation $\mathcal{I}$ maps propositions to $\{t, f\}$ (alt. $\{1, 0\}$).
- Semantics:

\[
\begin{align*}
\mathcal{I} \models p & \quad \text{if } \mathcal{I}(p) = t \\
\mathcal{I} \models \neg \varphi & \quad \text{if } \mathcal{I} \not\models \varphi \\
\mathcal{I} \models \varphi_1 \land \varphi_2 & \quad \text{if } \mathcal{I}(\varphi_1) \text{ and } \mathcal{I}(\varphi_2) \\
\vdots
\end{align*}
\]

- An interpretation $\mathcal{I}$ such that $\mathcal{I} \models \varphi$ is called a *model* of $\varphi$.
- Two formulae $\varphi_1$ and $\varphi_2$ are *equivalent* if $\varphi_1$ and $\varphi_2$ have the same set of models.
Normal Forms

- **Negation Normal Form (NNF):**
  - *Literals:* $p \in P$ (positive literal); $\neg p$, where $p \in P$ (negative literal).
  - NNF formulae $\varphi$: Literal, conjunction ($\land$) and disjunction ($\lor$) of $\varphi$.
  - Every propositional formula has an equivalent formula in NNF.

- **Disjunctive Normal Form (DNF):**
  - *Conjunct* $\omega$: Literal (same as in NNF), conjunction ($\land$) of $\omega$.
  - *Formulae* $\varphi$: Conjunct, disjunction ($\lor$) of $\varphi$.
  - Every propositional formula has an equivalent formula in DNF.
  - It is often convenient to represent each conjunct as a set of literals, and a formula as a set of conjuncts.
Normal Forms (contd.)

- Conjunctive Normal Form (CNF):
  - *Clause* $\omega$: Literal (same as in NNF), disjunction ($\lor$) of $\omega$.
  - *Formulae* $\varphi$: Clause, conjunction ($\land$) of $\varphi$.
  - Every propositional formula has an equivalent formula in CNF.
  - It is often convenient to represent each clause as a set of literals, and a formula as a set of clauses.
  - We will use the formula notation and the set notation interchangeably.
A formula is satisfiable if $\exists I$ s.t. $I \models \varphi$.

- An $I$ such that $I \models \varphi$ is called a model of $f$.
- We sometimes call $I$ as a *satisfying assignment*.
- If a formula is not satisfiable, we say it is *unsatisfiable*.
- Determining the satisfiability of a propositional formula is NP-complete.
- Satisfiability of a formula in DNF can be determined in polynomial time.

A formula is a tautology if $\forall I. I \models \varphi$.

- $\varphi$ is a tautology iff $\neg \varphi$ is unsatisfiable. Hence, tautology checking is NP-hard as well.
- Determining whether a CNF formula is a tautology can be done in polynomial time.
Applications of Satisfiability Checking

- Circuit design (equivalence of circuits)
- Model checking and verification (properties of programs and designs)
- Planning (state space exploration)
- Constraint Satisfaction Problems (e.g. optimization problems)
- ...

**Past:** If we suspect a given decision problem $F$ is hard, we show how any algorithm for $F$ can be used (as a subroutine) to solve SAT. This shows the hardness of $F$.

**Present:** Even though the problem is intractable, SAT solvers have become very good at solving large instances efficiently. So to solve $F$, we use SAT as a subroutine!
Binary Decision Diagrams

- Consider a *decision tree*, where
  - Each internal node is a proposition, and has two children, one on a branch labeled 0 and the other labeled 1.
  - Each leaf is labeled 0 or 1.
  - Each proposition appears at most once in any root-to-leaf path.
- Each decision tree represents a formula in DNF.
  - Each path from the root that ends in a leaf labeled 1 represents a conjunct $\omega$. If the path takes the 0-branch from $x$ then $x \in \omega$; if the path takes the 1-branch from $x$, then $\neg x \in \omega$.
- A BDD is the DAG representation of a decision tree which shares common subtrees.
- Each propositional formula has an equivalent BDD.
Consider a total order “≤” over propositions in $P$.

A BDD is said to be ordered (aka OBDD) iff for every pair of propositions $x$ and $y$,
if $x$ is a descendent of $y$ then $x \leq y$.

A BDD is said to be a reduced, ordered BDD (aka ROBDD) if it is an OBDD such that
each internal node has two distinct children.

Given an order “≤”, each formula has a unique ROBDD.
We identify nodes in the BDD uniquely by a natural number.
We use number 0 to represent the 0 leaf; and 1 to represent the 1 leaf.
We maintain a global map \( D: P \times N \times N \rightarrow N - \{0, 1\} \) such that \( D^{-1} \) is a (partial) function. Note: we can use an hash table / associative array to implement this map.
The global map \( D \) has two access functions:
- **lookup**, which implements \( D^{-1} \), and
- **insert**, which, given a triple \((p, i, j) \in (P \times N \times N)\) updates \( D(p, i, j) \) to be defined (if not already defined), and returns \( D(p, i, j) \).
Operations over ROBDDs— 1

Let $s \in N$ be the index of a ROBDD node.

```plaintext
function negate(s):
    switch s
        case 0: return 1
        case 1: return 0
    otherwise:
        let $(x_s, s_0, s_1) = \text{lookup}(s)$
        return insert($x_s$, negate($s_0$), negate($s_1$))
```
Operations over ROBDDs— 2

Let $\oplus$ be “∧” or “∨”, and $s, t \in N$ be indices of ROBDD nodes.

function $\text{bin}_{-}\text{oper}(\oplus, s, t)$:

switch $s$

    case 0: if ($\oplus = \land$) then return 0 else return $t$
    case 1: if ($\oplus = \land$) then return $t$ else return 1
    otherwise:

switch $t$

    case 0: if ($\oplus = \land$) then return 0 else return $s$
    case 1: if ($\oplus = \land$) then return $s$ else return 1
    otherwise: /* Both $s$ and $t$ are non-trivial BDDs */

    let $(x_s, s_0, s_1) = \text{lookup}(s)$ and $(x_t, t_0, t_1) = \text{lookup}(t)$

    if ($x_s \prec x_t$) then
        return $\text{insert}(x_s, \text{bin}_{-}\text{oper}(\oplus, s_0, t), \text{bin}_{-}\text{oper}(\oplus, s_1, t))$
    else if ($x_t \prec x_s$) then
        return $\text{insert}(x_t, \text{bin}_{-}\text{oper}(\oplus, s, t_0), \text{bin}_{-}\text{oper}(\oplus, s, t_1))$
    else /* $x_s = x_t$ */

    return $\text{insert}(x_s, \text{bin}_{-}\text{oper}(\oplus, s_0, t_0), \text{bin}_{-}\text{oper}(\oplus, s_1, t_1))$
Other Operations over BDDs—1

Renaming: rename propositions using map \( \rho \); assume domain and range are non-overlapping:

```markdown
function rename(\( \rho, s \)):
  switch s
    case 0: return 0
    case 1: return 1
  otherwise:
    let \((x_s, s_0, s_1) = lookup(s)\)
    \(r_0 = insert(\rho(x_s), 1, 0)\)
    \(r_1 = insert(\rho(x_s), 0, 1)\)
    \(t_0 = oper(\land, r_0, rename(\rho, s_0))\)
    \(t_1 = oper(\land, r_1, rename(\rho, s_1))\)
    return oper(\lor, t_0, t_1)
```
Project: restrict ROBDD to contain only propositions from a given subset $R$ (i.e. existentially quantify the propositions not in $R$).

```python
function project(R, s):
    switch s
    case 0: return 0
    case 1: return 1
    otherwise:
        let $(x_s, s_0, s_1) = \text{lookup}(s)$
        if $(x_s \not\in R)$ then
            return \text{oper}(\lor, \text{project}(R, s_0), \text{project}(R, s_1))
        else
            return \text{insert}(x_s, \text{project}(R, s_0), \text{project}(R, s_1))
```
Extensions of BDDs

- Multi-Valued DDs (MDDs): internal nodes are no longer binary. Can be used to represent arbitrary relations over finite domains.
  - Such relations can also be represented by ROBDDs by mapping
    - values in a domain $D$ to vectors of Booleans, and
    - variables to vectors of propositions
  - Natural join over relations can be done by conjunction (along with renaming, if necessary).
  - MDDs can be used to represent and compute models of Datalog programs.
- Multi-Terminal BDDs (MTBDDs): Leaves are no longer $0/1$, but range over arbitrary domains.
  - Used for representing functions that map vectors of Booleans to values in other domains (including reals).
Satisifiability Checking

Given a propositional formula $\varphi$, determine if $\varphi$ is satisfiable.

- Sometimes, the problem of determining a satisfying assignment is included as well.
- Via BDDs: construct a ROBDD $s$ for $\varphi$. The formula is satisfiable iff $s \neq 0$.
  Note that, in a ROBDD, any path leading to a 1 leaf spells a satisfying assignment.
- There are other algorithms to perform SAT checking (BDD-based checking is sometimes an overkill).
Notations

- An interpretation, aka *assignment*, denoted by $\sigma$, is a function from propositions to 0, 1.
- $\sigma[x \mapsto v]$ is another assignment $\sigma'$ which is assigns $v$ to $x$, and is same as $\sigma$ otherwise.
- A **CNF** formula $\varphi$ is a set of clauses; a clause $\omega$ is a set of literals.
- $\varphi[x \mapsto v]$ is the smallest formula $\varphi'$ such that:
  - $v = 0$, $x \in \omega$, and $\omega \in \varphi$: then $\omega - \{x\} \in \varphi'$
  - $v = 1$, $\neg x \in \omega$, and $\omega \in \varphi$: then $\omega - \{\neg x\} \in \varphi'$
  - $\omega \in \varphi$ such that $x \notin \omega$ nor $\neg x \notin \omega$: then $\omega \in \varphi'$
Propositional Logic

Davis-Putnam Procedure

- Input: a formula $\varphi$ in **CNF**.
- Repeatedly rewrites $\varphi$ using three important steps:
  - **Resolution**
  - **Unit Propagation**
  - **Pure Literal Elimination**
- Each step maintains the satisfiability of $\varphi$.
- If we end up with an empty formula (i.e., no clauses), then the original formula was satisfiable.
- If at any time we get an empty clause (i.e. no literals in a clause), then the original formula was unsatisfiable.
- Original Davis-Putnam procedure ['58] was refined to Davis-Putnam-Loveland-Logemann (DPLL) procedure ['62], which is the core of modern SAT solvers.
Resolution

- Pick any proposition $x$ such that both $x$ and $\neg x$ appear in the formula.
- For every pair of clauses $\omega_1$ and $\omega_2$ such that $x \in \omega_1$ and $\neg x \in \omega_2$:
  - Add clause $(\omega_1 - \{x\}) \cup (\omega_2 - \{\neg x\})$ to the formula.
- Remove all clauses containing $x$ or $\neg x$ from the formula.

The key idea is to perform every possible resolution with a proposition; we can then eliminate that predicate (and clauses that contain them) from further consideration.
A clause containing only a single literal is said to be a *unit clause*.

Determine the assignment to the proposition in an unit clause:

- A formula with clause \( \{x\} \) is satisfiable only when \( x \mapsto 1 \).
- A formula with clause \( \{\neg x\} \) is satisfiable only when \( x \mapsto 0 \).

Set \( \varphi := \varphi[x \mapsto v] \). That is, propagate that assignment \( x \mapsto v \) to other clauses.

Note that after completing one unit propagation, we may get new unit clauses.
A literal $\ell$ is said to be pure in a CNF formula if $\ell$ appears but its negation does not appear in the formula.
Example: In $\{\{x, \neg y, \neg z\}, \{\neg x, \neg y, z\}\}$: only $\neg y$ is pure.

Let $x$ be a pure (positive) literal. Then if $\sigma$ is a satisfying assignment, then so is $\sigma[x \mapsto 1]$.

Similarly, let $\neg x$ be a pure (negative) literal. Then if $\sigma$ is a satisfying assignment, then so is $\sigma[x \mapsto 0]$.

So we can determine the assignment to propositions in pure literals.

... and propagate that assignment to other clauses:
- If $x$ is a pure positive literal, then $\varphi := \varphi[x \mapsto 1]$.
- If $\neg x$ is a pure negative literal, then $\varphi := \varphi[x \mapsto 0]$. 
DPLL

Uses Davis-Putnam’s Unit Propagation and Pure Literal Elimination steps, and replaces Resolution with backtracking search.

```pseudocode
function DPLL(φ):
    /* Simplify using Unit Propagation and Pure Literal Elimination */
    φ := UPPE(φ)
    if (φ = ∅) return 1 /* Empty Formula: true */
    if (∅ ∈ φ) return 0 /* Empty Clause: false */
    /* Every proposition occurs negatively and positively in φ */
    Pick a proposition x in φ
    return DPLL(φ[x ↦ 0]) || DPLL(φ[x ↦ 1])
```

- Picking x, and considering [x ↦ 0] (and [x ↦ 1]) is known as “Decide” step of DPLL.
- Invoking DPLL on the new formulae (last line) is known as “Deduce” step of DPLL.
Unit Propagation and Pure Literal Elimination

These two can be considered as simplification operations.

function $UPPE(\varphi)$:

\[
\text{while (1):}
\]

- if ($\varphi = \emptyset$) return $\emptyset$ /* Empty Formula: nothing to simplify */
- if ($\emptyset \in \varphi$) return $\{\emptyset\}$ /* Empty Clause: unsatisfiable*/
- if ($\{x\} \in \varphi$) $\varphi := \varphi[x \mapsto 1]$ /* Unit Prop. 1 */
- else if ($\{\neg x\} \in \varphi$) $\varphi := \varphi[x \mapsto 0]$ /* Unit Prop. 2 */
- else if ($x$ is pure in $\varphi$) $\varphi := \varphi[x \mapsto 1]$ /* Pure Literal 1 */
- else if ($\neg x$ is pure in $\varphi$) $\varphi := \varphi[x \mapsto 0]$ /* Pure Literal 2 */
- else /* No more simplification possible */

  return $\varphi$

- Note $UPPE$ is deterministic: there are no choice-points, so no backtracking!
- In contrast, “$DPLL(\varphi[x \mapsto 0]) || DPLL(\varphi[x \mapsto 1])$” is computed by backtracking:
  - Try $DPLL(\varphi[x \mapsto 0])$. If it fails, undo all assignments/simplifications and try $DPLL(\varphi[x \mapsto 1])$. 

Conflict Driven Clause Learning [Marques-Silva&Sakallah '96]

- Consider when $DPLL(\varphi[x \mapsto 0])$ fails.
- We want to identify the root cause of this failure and not subsequently repeat the computations that led to the failure.
- The idea in CDCL is to learn the cause for a failure as a failure clause, and add it to the formula.
- This will force us to consider only those assignments that do not conflict with the failure clause, i.e. cause the same failure again.
- Success of modern SAT solvers is partly due to CDCL.
Propositional Logic

BDD

Satisfiability Checking

Implication Graph Example

\[ \varphi_1 = \omega_1 \land \omega_2 \land \omega_3 \land \omega_4 \land \omega_5 \land \omega_6 \]
\[ = (x_1 \lor x_3 \lor \neg x_2) \land (x_1 \lor \neg x_3) \land (x_2 \lor x_3 \lor x_4) \land 
\neg x_4 \lor \neg x_5) \land (x_2 \lor \neg x_4 \lor \neg x_6) \land (x_5 \lor x_6) \]

\[
\begin{align*}
x_{31} &= 0 @ 3 \\
x_2 &= 0 @ 5 \\
x_1 &= 0 @ 5 \\
x_3 &= 0 @ 5 \\
x_4 &= 1 @ 5 \\
x_{21} &= 0 @ 2 \\
x_5 &= 0 @ 5 \\
x_6 &= 0 @ 5 \\
k &= \ \text{\textit{κ}}
\end{align*}
\]
Propositional Logic

BDD

Satisfiability Checking

DPLL, Trees, and Levels

- Consider augmenting \textit{DPLL} with one extra parameter: \( \ell \), the level number.

- Recursive call to \textit{DPLL} will increment the level number. E.g. the last line of the algorithm will be:
  \[
  \text{return } \text{\textit{DPLL}}(\ell + 1, \varphi[x \mapsto 0]) \parallel \text{\textit{DPLL}}(\ell + 1, \varphi[x \mapsto 1])
  \]

- Consider passing the current level number to the simplification function \textit{UPPE} as well. The level number will never be increased in \textit{UPPE} but used there.

- Every assignment to a proposition will be tagged with the current level number.

- Assignments made in the last line of \textit{DPLL} are special since they start a new level. We will call the first proposition assigned at level \( \ell \) as \( \gamma(\ell) \).
Implication Graph

Rough idea is to keep track of how we obtained the current assignment to a proposition.

- Vertices in the graph are labeled by a triple: \((x, v, \ell)\), where \(x\) is a proposition, \(v\) is its current assignment, and \(\ell\) is the level number when that assignment was generated.
- Consider clause \(\omega_j\), which after a sequence of computation steps becomes a unit clause \(\{x\}\). The value of \(x\) is determined based on \(\omega_j\).
- For every proposition \(y \neq x\) in \(\omega_j\), there is an edge from the vertex labeled with \(y\) to the vertex labeled with \(x\).
- We recognize failure when we remove the last literal from a clause.
- We mark failure with a special vertex “\(\kappa\)” in the graph. All propositions in the failed clause are the antecedents of \(\kappa\) in the graph.
Implication Graph and Failure Clauses

Let $\ell_F$ be the current level (i.e. the one where failure was recognized).

- We walk backwards in the implication graph, starting at $\kappa$ and stopping at the first vertex $(y, v_y, \ell)$ with
  - $\ell < \ell_F$, i.e. assigned at a previous level, or
  - $y = \gamma(\ell_F)$, i.e., the first proposition assigned at the current level.
- Let $\{(y_1, v_{y_1}), (y_2, v_{y_2}), \ldots\}$ be the proposition/valuation pairs of all vertices reached at the end of the above walks.
- Consider the clause $\omega_F$ with literals $\{l_1, l_2, \ldots\}$ such that $l_i = y_i$ if $v_{y_i} = 0$, and $l_i = \neg y_i$ if $v_{y_i} = 1$.
- Any assignment that satisfies $\omega_F$ will not result in the current failure.
- When we backtrack from this failure and try another assignment, we will add $\omega_F$ to the formula. This will prevent the same failure occurring again.
Other Optimizations

- **UIPs:** use *dominators* in the implication graph to generate failure clauses.

- **Non-Chronological Backtracking:** use the levels of literals in the failure clause to find the backtracking points.

- **Randomized Restarts:** cutoff current search and restart. Randomize the selection of proposition selected at decision points.
  - SAT solving is “long tailed”: most instances can be solved quickly, but few take a terribly long time.