Lecture Outline

- Basic SAT algorithms
  - Stochastic local search
  - Davis-Putnam algorithm
  - DPLL algorithm
  - Stålmarck’s method
**Repetition: Resolution/Saturation**

**Saturation Algorithm**

- **INPUT**: CNF formula $F$
- **OUTPUT**: $\{SAT, UNSAT\}$

```plaintext
while (true) do
    $R = resolveAll(F)$
    if $(R \cap F \neq R)$ then $F = F \cup R$
    else break
if ($\bot \in F$) then return UNSAT else return SAT
```

Properties of the saturation algorithm:
- it is sound and complete – always terminates and answers correctly
- has exponential time and space complexity
Question: Can we do better than saturation-based resolution?

- Avoid exponential space complexity
- Improve average-case complexity (for important problem classes)
Stochastic Local Search (SLS)

SAT as an optimization problem: minimize the number of unsatisfied clauses

Start with a complete random assignment \( \alpha \):

\[
\begin{array}{cccccccccccc}
0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\
\end{array}
\]

Repeatedly flip (randomly/heuristically chosen) variables to decrease the number of unsatisfied clauses:

\[
\begin{array}{cccccccccccc}
0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\
\end{array}
\]
Local search algorithms are incomplete: they cannot show unsatisfiability!

Many variants of local search algorithms

Main question: Which variable should be flipped next?
  - select variable from an unsatisfied clause
  - select variable that increases the number of satisfied clauses most

How to avoid local minima?
Maybe[Assignment] \( \text{GSAT}(\text{ClauseSet } S) \) 

\{
\text{for } i = 1 \text{ to } \text{MAXTRIES} \text{ do } \{
\quad \alpha = \text{random-assignment to variables in } S
\text{for } j = 1 \text{ to } \text{MAXFLIPS} \text{ do } \{
\quad \text{if ( } \alpha \text{ satisfies all clauses in } S \text{ ) return } \alpha
\quad x = \text{variable that produces least number of }
\quad \text{unsatisfied clauses when flipped}
\quad \text{flip } x
\}
\}
\text{return Nothing} \quad \text{// no solution found}
\}
SLS: Illustration

[Source: Alan Mackworth, UBC, Canada]
Walksat [2]

- Variant of GSAT
- Try to avoid local minima by introducing "random noise"
  - Select unsatisfied clause $C$ at random
  - If by flipping a variable $x \in C$ no new unsatisfied clauses emerge, flip $x$
  - Otherwise:
    - With probability $p$ select a variable $x \in C$ at random
    - With probability $1 - p$ select a variable that changes as few as possible clauses from satisfied to unsatisfied when flipped
Consider a flip taking $\alpha$ to $\alpha'$. 

- **breakcount**: number of clauses satisfied in $\alpha$, but not in $\alpha'$. 
- **makecount**: number of clauses unsatisfied in $\alpha$, but satisfied in $\alpha'$. 
- **diffscore**: number of unsatisfied clauses in $\alpha$ minus number of clauses unsatisfied in $\alpha'$. 

Typically, **breakcount**, **makecount** and **diffscore** are updated after each flip. 

Question: How can we do this efficiently?
GSAT and Walksat Flip Heuristics

- GSAT: select variable with highest **diffscore**
- Walksat:
  - First randomly select unsatisfied clause $C$
  - If there is a variable with **breakcount** 0 in $C$, select it
  - otherwise with probability $p$ select a random variable from $C$, and with probability $1 - p$ a variable with minimal **breakcount** from $C$
Runtime Comparison Walksat vs. GSAT

Table 4: Comparing an efficient complete method (DP) with local search strategies on circuit synthesis problems. (Timings in seconds.)

<table>
<thead>
<tr>
<th>formula</th>
<th>id</th>
<th>vars</th>
<th>clauses</th>
<th>DP time</th>
<th>GSAT+w time</th>
<th>WSAT time</th>
</tr>
</thead>
<tbody>
<tr>
<td>2bitadd.12</td>
<td>708</td>
<td>1702</td>
<td>*</td>
<td>0.081</td>
<td>0.013</td>
<td></td>
</tr>
<tr>
<td>2bitadd.11</td>
<td>649</td>
<td>1562</td>
<td>*</td>
<td>0.058</td>
<td>0.014</td>
<td></td>
</tr>
<tr>
<td>3bitadd.32</td>
<td>8704</td>
<td>32316</td>
<td>*</td>
<td>94.1</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>3bitadd.31</td>
<td>8432</td>
<td>31310</td>
<td>*</td>
<td>456.6</td>
<td>0.7</td>
<td></td>
</tr>
<tr>
<td>2bitcomp.12</td>
<td>300</td>
<td>730</td>
<td>23096</td>
<td>0.009</td>
<td>0.002</td>
<td></td>
</tr>
<tr>
<td>2bitcomp.5</td>
<td>125</td>
<td>310</td>
<td>1.4</td>
<td>0.009</td>
<td>0.001</td>
<td></td>
</tr>
</tbody>
</table>

Table 5: Comparing DP with local search strategies on circuit diagnosis problems by Larrabee (1989). (Timings in seconds.)

<table>
<thead>
<tr>
<th>formula</th>
<th>id</th>
<th>vars</th>
<th>clauses</th>
<th>DP time</th>
<th>GSAT+w time</th>
<th>WSAT time</th>
</tr>
</thead>
<tbody>
<tr>
<td>ssa7552-038</td>
<td>1501</td>
<td>3575</td>
<td>7</td>
<td>129</td>
<td>2.3</td>
<td></td>
</tr>
<tr>
<td>ssa7552-158</td>
<td>1363</td>
<td>3034</td>
<td>*</td>
<td>90</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>ssa7552-159</td>
<td>1363</td>
<td>3032</td>
<td>*</td>
<td>14</td>
<td>0.8</td>
<td></td>
</tr>
<tr>
<td>ssa7552-160</td>
<td>1391</td>
<td>3126</td>
<td>*</td>
<td>18</td>
<td>1.5</td>
<td></td>
</tr>
</tbody>
</table>

[Source: Selman, Kautz, Cohen Local Search Strategies for Satisfiability Testing, 1993]
Davis-Putnam Algorithm [3]

- Presented in 1960 as a procedure for first-order (predicate) logic
- Procedure to check satisfiability of a formula $F$ in CNF
- Three (deduction) rules:
  1. **Unit propagation**: if there is a unit clause $C = \{l\}$ in $F$, simplify all other clauses containing $l$
  2. **Pure literal elimination**: If a literal $l$ never occurs negated in $F$, add the clause $\{l\}$ to $F$
  3. **Case splitting**: Assume that $F$ is put in the form $(A \lor l) \land (B \lor \lnot l) \land R$, where $A$, $B$, and $R$ are free of $l$. Replace $F$ by the clausification of $(A \lor B) \land R$
- Apply deduction rules (giving priority to rules 1 and 2) until no further rule is applicable
The superiority of the present procedure over those previously available is indicated in part by the fact that a formula on which Gilmore’s routine for the IBM 704 causes the machine to compute for 21 minutes without obtaining a result was worked successfully by hand computation using the present method in 30 minutes.
DPLL Algorithm: Outline

- **DPLL**: Davis-Putnam-Logemann-Loveland [4]
- Algorithmic improvements over DP algorithm
- **Basic idea**: case splitting and simplification
- **Simplification**: unit propagation and pure literal deletion
- **Unit propagation**: 1-clauses (unit clauses) fix variable values: if \( \{x\} \in S \), in order to satisfy \( S \), variable \( x \) must be set to 1.
- **Pure literal deletion**: If variable \( x \) occurs only positively (or only negatively) in \( S \), it may be fixed, i.e. set to 1 (or 0).
Let $F_0 = \{\{x, y\}, \{\neg x, y, \neg z\}, \{\neg x, z, u\}, \{x, \neg u\}\}$.

All clauses containing $y$ may be deleted, as $y$ occurs only positively in $F$. This yields:

$$F_1 = \{\{\neg x, z, u\}, \{x, \neg u\}\}$$

Each solution $\alpha_1$ of $F_1$ can be extended to a solution $\alpha_0$ of $F_0$ by setting $\alpha_0(y) = 1$.

Moreover, if $F_1$ does not possess a solution, then so does $F_0$.

Repeating yields $F_2 = \{\{x, \neg u\}\}$ and $F_3 = \emptyset$, thus $F_0$ is satisfiable.
DPLL Algorithm

```java
boolean DPLL(ClauseSet S)
{
    while (S contains a unit clause \{L\}) {
        delete from S clauses containing L;  // unit-subsumption
        delete \neg L from all clauses in S;   // unit-resolution
    }
    if (\bot \in S) return false;         // empty clause?
    while (S contains a pure literal L)
    {
        delete from S all clauses containing L;
        if (S = \emptyset) return true;     // no clauses?
        choose a literal L occurring in S;   // case-splitting
        if (DPLL(S \cup \{L\}) return true;  // first branch
        else if (DPLL(S \cup \{\neg L\}) return true;  // second branch
        else return false;
    }
}
```
DPLL: Implementation Issues

- How can we implement unit propagation efficiently?
- Which literal \( L \) to use for case splitting?
- How can we efficiently implement the case splitting step?
boolean mDPLL(ClauseSet S, PartialAssignment α) {
    while ((S, α) contains a unit clause \{L\}) {
        add \{L = 1\} to α
    }
    if (a literal is assigned both 0 and 1 in α) return false;
    if (all literals assigned) return true;
    choose a literal L not assigned in α occurring in S;
    if (mDPLL(S, α \cup \{L = 1\}) return true;
    else if (mDPLL(S, α \cup \{L = 0\}) return true;
    else return false;
}

(S, α): clause set S as “seen” under partial assignment α
Stålmarck’s Method [5]

- **Input**: Arbitrary formula $F$ in propositional logic (need not be in CNF, $\Rightarrow$ and $\Leftrightarrow$ also allowed)
- **Goal**: Show unsatisfiability of $F$
- **Preprocessing**: Decompose formula tree into simple equations (triplets) $T$ and a literal equivalence class $R$. ($R \subseteq L^0 \times L^0$ where $L^0 = L \cup \{0, 1\}$, $R$ ‘consistent’)
- **Basic processing steps**: $k$-saturation ($k = 0, 1, \ldots$)
  - 0-saturation: simplification with triplet rules
  - $k$-saturation ($k \geq 1$): case distinction, breadth-first search
- Developed by Gunnar Stålmarck ($\sim 1989$), patented
Decomposition into Triplets

\[ F = ((x \land y) \lor \neg y) \land (z \leftrightarrow y) \]

Formula tree:

Initial equival. class: \{n_1 = 0\}  
(to show unsatisfiability of \( F \))

Triplets:

- \( n_1 = n_2 \land n_4 \)
- \( n_2 = n_3 \lor \neg y \)
- \( n_3 = x \land y \)
- \( n_4 = z \leftrightarrow y \)

Normalized triplets:  
(only \( \land \) and \( \leftrightarrow \))

- \( n_1 = n_2 \land n_4 \)
- \( \neg n_2 = \neg n_3 \land y \)
- \( n_3 = x \land y \)
- \( n_4 = z \leftrightarrow y \)
Stålmarck’s Method: 0-Saturation

Given set of triplets $T$ and literal equivalence class $R$ apply derivation rules (deriving new literal equivalences):

\[
\frac{p = q \land r}{p = 1} \quad \frac{p = q \land r}{p = \neg q} \quad (A) \quad (D)
\]

\[
\frac{r = 1}{q = 1} \quad \frac{p = 0}{r = 0}
\]

\[
\frac{q = 0}{p = 0} \quad (B)
\]

\[
\frac{q = 1}{p = r} \quad (C)
\]

\[
\frac{q = r}{p = q} \quad (E)
\]

\[
\frac{q = \neg r}{p = 0} \quad (F)
\]
Stålmärck’s Method: \( k \)-Saturation

Given formula \( F \), represented as \((T, R)\) (triplets and equiv. rel.)

procedure saturate extends equivalence relation \( R \):

\[
\text{EquivRel saturate}(\text{int } k, \text{ TripletSet } T, \text{ EquivRel } R) \{
\text{if } ( k = 0 ) \text{ return zero-saturate}(T, R)
\text{forall } x \in \text{Var}(T) \text{ not fixed in } R \text{ do }
\text{R}_0 = \text{saturate}(k - 1, T, R \cup \{x = 0\})
\text{R}_1 = \text{saturate}(k - 1, T, R \cup \{x = 1\})
R = R_0 \cap R_1
\}
\text{return } R
\]

(zero-saturate returns all-relation if inconsistency was found)
**k-Saturation: Graphical Illustration**

0-saturation (simplification rules)

1-saturation

2-saturation

merge relations

\[ \rightarrow R_0 \quad \rightarrow R_1 \]

\[ \Rightarrow \text{new } R \]
Summary: Stålmarck’s Algorithm

**Input:** Formula $F$ represented as set of triplets $T$
(with $n_1$ representing top of formula tree)

**Output:** $F$ satisfiable?

```java
boolean stalmarck(TripletSet T)
{
    k = 0;  R = \{n_1 = 0\}
    do {
        R = saturate(k, T, R)
        if ( R = all-relation ) return false
        else if ( R satisfies all triplets T ) return true
        else  k = k + 1
    } 
}
```
URL http://dl.acm.org/citation.cfm?id=1867135.1867203


URL http://doi.acm.org/10.1145/321033.321034
URL http://doi.acm.org/10.1145/368273.368557

URL http://dx.doi.org/10.1023/A:1008725524946