Reminder

• CSPs consist of:
  o Variables
  o Domains
  o Constraints: legal tuples of values for subsets of variables

• Goal: complete and consistent assignment

• Example: graph coloring
How hard are CSPs?

• In theory, solving a general CSP is NP-c
  o Obviously in NP
  o Captures graph coloring so NP-hard
• In practice, CSPs are often easy to solve
• Where are the hard problems?
• Identify order parameter to predict problem difficulty
Is this graph 4-colorable?
Is this graph 4-colorable?
Average degree

• Order parameter for graph coloring: average degree $= \frac{2|E|}{|V|}$

• For a random graph, what is the probability of being colorable, as a function of the average degree?

• Should be 1 at $x=0$ and go down to 0
Phase transition

Critical value: avg. degree ~8

Cheeseman et al., IJCAI 1993
Peak in Difficulty

(c) 4-color difficulty

critical value: avg. degree ~8

Cheersman et al., IJCAI 1993
Coincidence?

- Algorithm used: backtracking search with the heuristics we discussed
- Graph coloring is most difficult around the critical value of the order parameter
- In that region problems are neither underconstrained nor overconstrained
Generating hard graphs

• We want to test our CSP solvers with hard problems!
• Example: graph coloring
• First, reduce the graph using operators shown on next slide
• Second, concentrate on graphs with avg. degree around the critical value
Reduction operators

Underconstrained

Before

After

Examples shown for 4-coloring
Reduction operators

Subsumed

Examples shown for 4-coloring
Reduction operators

Connected to (k-1)-clique

![Before](image1)  \Rightarrow  

![After](image2)

Examples shown for 4-coloring
GENERAL FRAMEWORK

- Nogoods = illegal tuples of values for variables
- Sperner system = family of sets s.t. no set is contained in another set
- Construct Sperner system of nogoods by considering only minimized (inclusion-minimal) nogoods
- Order parameter: 
  \[ \beta = \frac{\text{#minimized nogoods}}{\text{#variables}} \]
- Q: How many minimized nogoods in k-graph coloring?
- A: \[ \text{#minimized nogoods} = |E| \cdot k \]
- \[ \frac{\text{#minimized nogoods}}{\text{#variables}} \propto \text{avg. degree} \]
Theoretical prediction

Williams and Hogg, AIJ 1994
CSP example: SAT

• Given a formula in propositional logic, find a satisfying assignment (or prove that none exists)

• Example: \((a \lor b) \land (\neg a \lor \neg b \lor c)\)

• Conjunctive normal form = conjunction of disjunctive clauses

• First established NP-complete problem
  o S. A. Cook. The complexity of theorem proving procedures. STOC 1971
SAT applications

• Electronic design automation, e.g., testing and verification
• AI: automated theorem proving, knowledge base deduction
• Software (from Athanasios): checking if program crashes
### DPLL Algorithm (1962)

<table>
<thead>
<tr>
<th>\neg a \lor b \lor c</th>
</tr>
</thead>
<tbody>
<tr>
<td>a \lor c \lor d</td>
</tr>
<tr>
<td>a \lor c \lor \neg d</td>
</tr>
<tr>
<td>a \lor \neg c \lor d</td>
</tr>
<tr>
<td>a \lor \neg c \lor \neg d</td>
</tr>
<tr>
<td>\neg b \lor \neg c \lor d</td>
</tr>
<tr>
<td>\neg a \lor b \lor \neg c</td>
</tr>
<tr>
<td>\neg a \lor \neg b \lor c</td>
</tr>
</tbody>
</table>
**DPLL algorithm (1962)**

<table>
<thead>
<tr>
<th>\neg a \lor b \lor c</th>
<th>\neg c \lor d</th>
<th>\neg c \lor \neg d</th>
<th>\neg b \lor \neg c \lor d</th>
<th>\neg a \lor \neg b \lor c</th>
</tr>
</thead>
<tbody>
<tr>
<td>c \lor d</td>
<td>c \lor \neg d</td>
<td>\neg c \lor \neg d</td>
<td>\neg b \lor \neg c \lor d</td>
<td>\neg a \lor \neg b \lor c</td>
</tr>
</tbody>
</table>
DPLL algorithm (1962)

\[-a \lor c\]
\[c \lor d\]
\[c \lor \neg d\]
\[\neg c \lor d\]
\[\neg c \lor \neg d\]
\[\neg b \lor \neg c \lor d\]
\[\neg a \lor \neg c\]
\[\neg a \lor \neg b \lor c\]
DPLL algorithm (1962)

\neg a

\neg c \lor \neg d

\neg c \lor \neg d

\neg b \lor \neg c \lor \neg d

\neg a \lor \neg c

\neg a \lor \neg b

\neg c \lor d

\neg b

\neg a

\neg d

\neg c
DPLL algorithm (1962)

- a
- d
- ¬d
- ¬c ∨ d
- ¬c ∨ ¬d
- ¬b ∨ ¬c ∨ d
- ¬a ∨ ¬c
- ¬a ∨ ¬b
DPLL algorithm (1962)

\[
\neg a \lor c \\
c \lor d \\
c \lor \neg d \\
d \\
\neg d \\
\neg b \lor d \\
\neg a \\
\neg a \lor \neg b \lor c
\]
DPLL algorithm (1962)

\[ \neg a \lor b \lor c \]

<table>
<thead>
<tr>
<th>Formula</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>\neg a</td>
<td>T</td>
</tr>
<tr>
<td>\neg b</td>
<td>F</td>
</tr>
<tr>
<td>\neg c</td>
<td>F</td>
</tr>
<tr>
<td>\neg d</td>
<td>T</td>
</tr>
<tr>
<td>\neg e</td>
<td>F</td>
</tr>
<tr>
<td>\neg f</td>
<td>T</td>
</tr>
<tr>
<td>\neg g</td>
<td>F</td>
</tr>
</tbody>
</table>

The DPLL algorithm is a systematic method for solving the satisfiability problem in propositional logic. It uses a tree structure to explore the possible assignments of truth values to variables. The algorithm proceeds by making assignments, and if a contradiction is encountered, it backtracks to a previous assignment and makes a different choice. If the entire tree is explored without finding a contradiction, the formula is satisfiable.
DPLL algorithm (1962)

<table>
<thead>
<tr>
<th>b ∨ c</th>
</tr>
</thead>
<tbody>
<tr>
<td>a ∨ c ∨ d</td>
</tr>
<tr>
<td>a ∨ c ∨ ¬d</td>
</tr>
<tr>
<td>a ∨ ¬c ∨ d</td>
</tr>
<tr>
<td>a ∨ ¬c ∨ ¬d</td>
</tr>
<tr>
<td>¬c ∨ d</td>
</tr>
<tr>
<td>b ∨ ¬c</td>
</tr>
<tr>
<td>c</td>
</tr>
</tbody>
</table>

Diagram:

```
    a
   /\  
  /   \  
 b       b
  \     /  
   \   /    
    c --- c
```

- a: T
- b: F
- c: T
- d: F

Truth values:
- a: T
- b: F
- c: T
- d: F
DPLL algorithm (1962)

| b ∨ c    | a ∨ c ∨ d    | a ∨ c ∨ ¬d    | a ∨ ¬c ∨ d    | ¬c ∨ d  | b ∨ ¬c  | c |

\[
a ∧ c ∧ d
\]

\[
a ∧ c ∧ ¬d
\]

\[
a ∧ ¬c ∧ d
\]

\[
¬c ∨ d
\]

\[
b ∨ c
\]

\[
a ∨ c ∨ d
\]

\[
a ∨ c ∨ ¬d
\]

\[
¬c ∨ d
\]

\[
b ∨ ¬c
\]

\[
c
\]

\[
C=1
\]
DPLL algorithm (1962)

\[\begin{align*}
& b \lor c \\
& a \lor c \lor d \\
& a \lor c \lor \neg d \\
& a \lor \neg c \lor d \\
& a \lor \neg c \lor \neg d \\
& d \\
& b \lor \neg c \\
& c
\end{align*}\]
DPLL algorithm (1962)

• Assign next value
• Erase unsatisfied literals, backtrack when clause becomes empty
• **Unit propagation** = if clause has only one variable left, assign satisfying value
• **Boolean constraint propagation** = iteratively apply unit propagation until there are no unit clauses
Variable ordering for DPLL

• Three design principles for heuristics
• Constrainedness
  o Choose variables that are more constrained
  o Motivation: attack most difficult part of the problem first
  o Short clauses are most constraining: only take them into account
  o Several variants, e.g., most occurrences in short clauses
Variable ordering for DPLL

- Satisfaction
  - Try to find variables that come closest to satisfying the problem
  - Clause of length $k$ rules out $2^{-k}$ of possible assignments; give weight $2^{-k}$ to each clause of length $k$
  - For each literal, calculate weighted sum of clauses that it appears in
  - Gives variable and value ordering
Variable ordering for DPLL

• Simplification
  o Want to simplify the problem as much as possible
  o For each assignment we get a cascade of unit propagations
  o Test all assignments and choose the one that caused the largest cascade
  o Successful variants only probe promising variables (based on other heuristics)
DPLL and Horn clauses

• \[(a \land b \land c) \Rightarrow d\] is equivalent to
  \[\neg(a \land b \land c) \lor d\] is equivalent to
  \[\neg a \lor \neg b \lor \neg c \lor d\] which is a Horn clause

• Formal def: **Horn clause** = clause that has at most one non-negated variable
Theorem. If BCP applied to a set of Horn clauses does not result in contradiction then the set is satisfiable

Proof

- Assume BCP finished
- Remove satisfied clauses and assigned variables from unsatisfied clauses
- Remaining clauses have at least two literals, therefore at least one negated variable
- How do we satisfy the remaining clauses?
- Satisfy remaining clauses by assigning false to all unassigned variables □
DPLL and Horn clauses

• Corollary. Given only Horn clauses, DPLL runs in polynomial time

• Reason: we never take a wrong path in the tree because BCP immediately finds a conflict
Converting CSP to SAT

• SAT is obviously a CSP
• A CSP can also easily be encoded as SAT
  - Clearly a polytime encoding exists because SAT is NP-c
• For each variable X and every \( j \in \text{Dom}(X) \)
  we have a SAT variable \( Z_{X=d} \)
• For example, if \( \text{Dom}(X) = \{1, 2, 3, 4\} \) then
  we have \( Z_{X=1}, Z_{X=2}, Z_{X=3}, Z_{X=4} \)
Converting CSP to SAT

• “At least one value” clause:
  \[ Z_{X=1} \lor Z_{X=2} \lor Z_{X=3} \lor Z_{X=4} \]

• At most one value” clauses:
  \[
  (\neg Z_{X=1} \lor \neg Z_{X=2}) \land (\neg Z_{X=1} \lor \neg Z_{X=3}) \land \\
  (\neg Z_{X=1} \lor \neg Z_{X=4}) \land (\neg Z_{X=2} \lor \neg Z_{X=3}) \land \\
  (\neg Z_{X=2} \lor \neg Z_{X=4}) \land (\neg Z_{X=3} \lor \neg Z_{X=4})
  \]
Converting CSP to SAT

• For every constraint and every tuple that falsifies the constraint, add clause

• For example if constraint is falsified by $(X=1, Y=3)$ add constraint
  \[ \neg Z_{X=1} \lor \neg Z_{Y=3} \]
Linear encoding

- Impose an order on the domain of each variable
- Let $X$ with $\text{Dom}(X) = \{1,\ldots,d\}$
- Add $d-1$ SAT variables $Z_{X \leq i}$ for all $i \in \{1,\ldots,d-1\}$
- Add clauses $[\neg Z_{X \leq i} \lor Z_{X \leq i+1}]$ for all $i$
- Assign $X=i$ by $Z_{X \leq i} = T$, $Z_{X \leq i-1} = F$
- Advantage: BCP automatically assigns $Z_{X \leq k} = T$ for every $k > i$, $Z_{X \leq k} = F$ for every $k < i-1$