Constraint Satisfaction Problems

• The constraint network model
  – Variables, domains, constraints, constraint graph, solutions
• Examples:
  – graph-coloring, 8-queen, cryptarithmetic, crossword puzzles, vision problems, scheduling, design
• The search space and naive backtracking,
• The constraint graph

Maybe:
• Consistency enforcing algorithms
  – arc-consistency, AC-1, AC-3

Next time:
• Backtracking strategies
  – Forward-checking, dynamic variable orderings
• Special case: solving tree problems
• Local search for CSPs
Constraint satisfaction problems (CSPs)

Standard search problem:
- **state** is a “black box”—any old data structure
  that supports goal test, eval, successor

CSP:
- **state** is defined by **variables** $X_i$ with **values** from **domain** $D_i$
  
  **goal test** is a set of **constraints** specifying
  allowable combinations of values for subsets of variables

Simple example of a **formal representation language**

Allows useful **general-purpose** algorithms with more power
than standard search algorithms
Constraint Satisfaction

**Example: map coloring**

Variables - countries (A, B, C, etc.)
Values - colors (e.g., red, green, yellow)
Constraints:  $A \neq B$, $A \neq D$, $D \neq E$, etc.
Constraint Satisfaction

Example: map coloring

Variables - countries (A,B,C,etc.)
Values - colors (e.g., red, green, yellow)
Constraints: \( A \neq B, A \neq D, D \neq E, \) etc.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>red</td>
<td>green</td>
</tr>
<tr>
<td>red</td>
<td>yellow</td>
</tr>
<tr>
<td>green</td>
<td>red</td>
</tr>
<tr>
<td>green</td>
<td>yellow</td>
</tr>
<tr>
<td>yellow</td>
<td>green</td>
</tr>
<tr>
<td>yellow</td>
<td>red</td>
</tr>
</tbody>
</table>
Constraint Satisfaction

Example: map coloring

Variables - countries (A, B, C, etc.)
Values - colors (e.g., red, green, yellow)
Constraints: $A \neq B, A \neq D, D \neq E, \ etc.$

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>red</td>
<td>green</td>
</tr>
<tr>
<td>red</td>
<td>yellow</td>
</tr>
<tr>
<td>green</td>
<td>red</td>
</tr>
<tr>
<td>green</td>
<td>yellow</td>
</tr>
<tr>
<td>yellow</td>
<td>green</td>
</tr>
<tr>
<td>yellow</td>
<td>red</td>
</tr>
</tbody>
</table>
Example: Map-Coloring

Variables \( WA, NT, Q, NSW, V, SA, T \)

Domains \( D_i = \{\text{red}, \text{green}, \text{blue}\} \)

Constraints: adjacent regions must have different colors
  
  e.g., \( WA \neq NT \) (if the language allows this), or
  
  \( (WA, NT) \in \{(\text{red}, \text{green}), (\text{red}, \text{blue}), (\text{green}, \text{red}), (\text{green}, \text{blue}), \ldots\} \)
Example: Map-Coloring contd.

**Solutions** are assignments satisfying all constraints, e.g.,
\[
\{WA = \text{red}, NT = \text{green}, Q = \text{red}, NSW = \text{green}, V = \text{red}, SA = \text{blue}, T = \text{green}\}
\]
**Constraint graph**

*Binary CSP*: each constraint relates at most two variables

*Constraint graph*: nodes are variables, arcs show constraints

General-purpose CSP algorithms use the graph structure to speed up search. E.g., Tasmania is an independent subproblem!
Sudoku

Each row, column and major block must be all different

“Well posed” if it has unique solution: 27 constraints
Sudoku

Each row, column and major block must be alldifferent

“Well posed” if it has unique solution: 27 constraints
Sudoku

Each row, column and major block must be all different

“Well posed” if it has unique solution: 27 constraints
Sudoku

Each row, column and major block must be alldifferent

“Well posed” if it has unique solution: 27 constraints
Each row, column and major block must be all different

“Well posed” if it has unique solution: 27 constraints
Each row, column and major block must be alldifferent.

“Well posed” if it has unique solution: 27 constraints.
Sudoku

Each row, column and major block must be alldifferent

“Well posed” if it has unique solution: 27 constraints
Sudoku

Each row, column and major block must be alldifferent

“Well posed” if it has unique solution: 27 constraints
Each row, column and major block must be alldifferent

“Well posed” if it has unique solution: 27 constraints
Sudoku

Each row, column and major block must be alldifferent

“Well posed” if it has unique solution: 27 constraints
Sudoku

Each row, column and major block must be alldifferent

“Well posed” if it has unique solution: 27 constraints

Variables: 81 slots

Domains = \{1,2,3,4,5,6,7,8,9\}

Constraints:
- 27 not-equal

Constraint propagation
A network of binary constraints

- Variables $X_1, \ldots, X_n$
- Domains $D_1, \ldots, D_n$
  - sets of discrete values,
- Binary constraints $R_{ij}$
  - represent the list of allowed pairs of values; $R_{ij} \subseteq D_i \cap D_j$
- Constraint graph:
  - A node for each variable and an arc for each constraint
- Solution:
  - An assignment of a value to each variable such that no constraint is violated.
- A network of constraints represents the relation of all solutions.
  $\text{Solutions} = \{ (X_1, \ldots, X_n) \mid (X_i, X_j) \in R_{ij}, X_i \in D_i, X_j \in D_j \}$
Varieties of constraints

• **Unary** constraints involve a single variable,
  – e.g., SA \neq \text{green}

• **Binary** constraints involve pairs of variables,
  – e.g., SA \neq \text{WA}

• **Higher-order** constraints involve 3 or more variables,
  – e.g., cryptarithmetic column constraints
Cryptarithmetic

• Each letter represents a different digit
• They should satisfy the addition constraint

\[
\begin{array}{cccc}
& & T & W & O \\
+ & T & W & O \\
\hline
& F & O & U & R \\
\end{array}
\]
Cryptarithmetic

• Each letter represents a different digit
• They should satisfy the addition constraint

\[
\begin{array}{c}
T \quad W \quad O \\
+ \quad T \quad W \quad O \\
\hline
F \quad O \quad U \quad R
\end{array}
\]

Variables: \( F, T, U, W, R, O, X_1, X_2, X_3 \)
Domains: \( \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \)
Constraints
\( \text{alldiff}(F, T, U, W, R, O) \)
\( O + O = R + 10 \cdot X_1, \text{ etc.} \)
Examples

• Cryptarithmetic: SEND+MORE = MONEY
• n-Queens
• Crossword puzzles
• Graph coloring
• Vision problems
• Scheduling
  – Assignment (who teaches what); timetable (where & when)
  – Transportation scheduling, factory assembly, etc.
Example 1: The 4-queen problem

Place 4 Queens on a chess board of 4x4 such that no two queens reside in the same row, column or diagonal.

- **Variables**: each row is a variable.
- **Domains**: $D_i = \{1, 2, 3, 4\}$.
- **Constraints**: There are $\binom{4}{2} = 6$ constraints involved:

The standard CSP formulation of the problem:

1. $X_1$
2. $X_2$
3. $X_3$
4. $X_4$
Example 1: The 4-queen problem

Place 4 Queens on a chess board of 4x4 such that no two queens reside in the same row, column or diagonal.

Standard CSP formulation of the problem:
- **Variables**: each row is a variable.

- **Domains**: \( D_i = \{1,2,3,4\} \).

- **Constraints**: There are \( \binom{4}{2} = 6 \) constraints involved:

  \[ R_{12} = \{(1,3)(1,4)(2,4)(3,1)(4,1)(4,2)\} \]
Example 1: The 4-queen problem

Place 4 Queens on a chess board of 4x4 such that no two queens reside in the same row, column or diagonal.

- **Domains:** \( D_i = \{1,2,3,4\} \).

- **Constraints:** There are \( \binom{4}{2} = 6 \) constraints involved:

  \[
  R_{12} = \{(1,3)(1,4)(2,4)(3,1)(4,1)(4,2)\}
  \]

  \[
  R_{13} = \{(1,2)(1,4)(2,1)(2,3)(3,2)(3,4)(4,1)(4,3)\}
  \]

Standard CSP formulation of the problem:

- **Variables:** each row is a variable.

\[
\begin{array}{cccc}
X_1 & X_2 & X_3 & X_4 \\
\end{array}
\]
Example 1: The 4-queen problem

Place 4 Queens on a chess board of 4x4 such that no two queens reside in the same row, column or diagonal.

Standard CSP formulation of the problem:
- **Variables**: each row is a variable.

- **Domains**: \( D_i = \{1,2,3,4\} \).

- **Constraints**: There are \( \binom{4}{2} = 6 \) constraints involved:
  
  \[
  R_{12} = \{(1,3)(1,4)(2,4)(3,1)(4,1)(4,2)\}
  \]
  
  \[
  R_{13} = \{(1,2)(1,4)(2,1)(2,3)(3,2)(3,4)(4,1)(4,3)\}
  \]
  
  \[
  R_{14} = \{(1,2)(1,3)(2,1)(2,3)(2,4)(3,1)(3,2)(3,4)(4,2)(4,3)\}
  \]
  
  \[
  R_{23} = \{(1,3)(1,4)(2,4)(3,1)(4,1)(4,2)\}
  \]
  
  \[
  R_{24} = \{(1,2)(1,4)(2,1)(2,3)(3,2)(3,4)(4,1)(4,3)\}
  \]
  
  \[
  R_{34} = \{(1,3)(1,4)(2,4)(3,1)(4,1)(4,2)\}
  \]
Example 1: The 4-queen problem

Place 4 Queens on a chess board of 4x4 such that no two queens reside in the same row, column or diagonal.

Standard CSP formulation of the problem:

- **Variables**: each row is a variable.

- **Domains**: \( D_i = \{1,2,3,4\} \).

- **Constraints**: There are \( \binom{4}{2} = 6 \) constraints involved:

  \[
  \begin{align*}
  R_{12} &= \{(1,3)(1,4)(2,4)(3,1)(4,1)(4,2)\} \\
  R_{13} &= \{(1,2)(1,4)(2,1)(2,3)(3,2)(3,4)(4,1)(4,3)\} \\
  R_{14} &= \{(1,2)(1,3)(2,1)(2,3)(2,4)(3,1)(3,2)(3,4)(4,2)(4,3)\} \\
  R_{23} &= \{(1,3)(1,4)(2,4)(3,1)(4,1)(4,2)\} \\
  R_{24} &= \{(1,2)(1,4)(2,1)(2,3)(3,2)(3,4)(4,1)(4,3)\} \\
  R_{34} &= \{(1,3)(1,4)(2,4)(3,1)(4,1)(4,2)\}
  \end{align*}
  \]

- **Constraint Graph**:

- \( X_1 \)

- \( X_2 \)

- \( X_3 \)

- \( X_4 \)
Example 1: The 4-queen problem

Place 4 Queens on a chess board of 4x4 such that no two queens reside in the same row, column or diagonal.

- **Domains:** $D_i = \{1,2,3,4\}$.

- **Constraints:** There are $\binom{4}{2} = 6$ constraints involved:
  
  $R_{12} = \{(1,3)(1,4)(2,4)(3,1)(4,1)(4,2)\}$
  
  $R_{13} = \{(1,2)(1,4)(2,1)(2,3)(3,2)(3,4)(4,1)(4,3)\}$
  
  $R_{14} = \{(1,2)(1,3)(2,1)(2,3)(2,4)(3,1)(3,2)(3,4)(4,2)(4,3)\}$
  
  $R_{23} = \{(1,3)(1,4)(2,4)(3,1)(4,1)(4,2)\}$
  
  $R_{24} = \{(1,2)(1,4)(2,1)(2,3)(3,2)(3,4)(4,1)(4,3)\}$
  
  $R_{34} = \{(1,3)(1,4)(2,4)(3,1)(4,1)(4,2)\}$

Standard CSP formulation of the problem:

- **Variables:** each row is a variable.

- **Constraint Graph:**

  ![Constraint Graph](image_url)
Standard search formulation (incremental)

Let’s start with the straightforward, dumb approach, then fix it

States are defined by the values assigned so far

◊ **Initial state**: the empty assignment, \{\}

◊ **Successor function**: assign a value to an unassigned variable that does not conflict with current assignment.
  \[\Rightarrow\text{ fail if no legal assignments (not fixable!)}\]

◊ **Goal test**: the current assignment is complete

1) This is the same for all CSPs!
2) Every solution appears at depth \(n\) with \(n\) variables
  \[\Rightarrow\text{ use depth-first search}\]
3) Path is irrelevant, so can also use complete-state formulation
4) \(b = (n - \ell)d\) at depth \(\ell\), hence \(n!d^n\) leaves!!!!
Backtracking search

Variable assignments are commutative, i.e.,

\[ [WA = \text{red} \text{ then } NT = \text{green}] \text{ same as } [NT = \text{green} \text{ then } WA = \text{red}] \]

Only need to consider assignments to a single variable at each node

\[ \Rightarrow b = d \text{ and there are } d^n \text{ leaves} \]

Depth-first search for CSPs with single-variable assignments is called backtracking search

Backtracking search is the basic uninformed algorithm for CSPs

Can solve \( n \)-queens for \( n \approx 25 \)
The search space
The search space

\[ X_1, \ldots, X_n \]

- **Definition**: given an ordering of the variables
  - **a state**:
    - is an assignment to a subset of variables that is consistent.
  - **Operators**:
    - add an assignment to the next variable that does not violate any constraint.
  - **Goal state**:
    - a consistent assignment to all the variables.
Backtracking example
Backtracking example
Backtracking example
Backtracking example
Backtracking search

function Backtracking-Search(csp) returns solution/failure
    return Recursive-Backtracking({}, csp)

function Recursive-Backtracking(assignment, csp) returns soln/failure
    if assignment is complete then return assignment
    var ← Select-Unassigned-Variable(Variables[csp], assignment, csp)
    for each value in Order-Domain-Values(var, assignment, csp) do
        if value is consistent with assignment given Constraints[csp] then
            add {var = value} to assignment
            result ← Recursive-Backtracking(assignment, csp)
            if result ≠ failure then return result
        remove {var = value} from assignment
    return failure
Dependence on variable ordering

Example 2: Given the following constraint network:

Variables: $Z, X, Y$

Domains: $D_Z = \{5, 2\} \quad D_X = \{2, 4\} \quad D_Y = \{5, 2\}$

Constraints: $R_{ZX} \equiv Z \text{ divides } X, \quad R_{ZY} \equiv Z \text{ divides } Y,$

With the ordering $d = \{Z, X, Y\}$, the search space explored is:

With the ordering $d = \{X, Z, Y\}$, the search space explored is:
Recap

• Constraint Satisfaction problem (e.g., map coloring)
  – Variables: (regions of a map)
  – Domains: values that the variables can take (colors)
  – Constraints: Restrictions on values that can be assigned simultaneously
• Backtracking search
  – a state is an assignment to a subset of variables that is consistent.
  – Operators: add an assignment to the next variable that does not violate any constraint.
  – DFS in this state-space
    • Select an unassigned variable and assign a value to it!!
Improving backtracking efficiency

General-purpose methods can give huge gains in speed:

1. Which variable should be assigned next?
2. In what order should its values be tried?
3. Can we detect inevitable failure early?
4. Can we take advantage of problem structure?
Minimum remaining values (MRV):
choose the variable with the fewest legal values

Which variable to choose next?
Degree heuristic

Tie-breaker among MRV variables

Degree heuristic:
choose the variable with the most constraints on remaining variables

Which variable to choose next?
Least constraining value

Given a variable, choose the least constraining value:
the one that rules out the fewest values in the remaining variables

Combining these heuristics makes 1000 queens feasible

In What order should a variable’s values be tried?
Can we detect failures early?

Forward checking

Idea: Keep track of remaining legal values for unassigned variables. Terminate search when any variable has no legal values.
Forward checking

Idea: Keep track of remaining legal values for unassigned variables. Terminate search when any variable has no legal values.
Forward checking

Idea: Keep track of remaining legal values for unassigned variables. Terminate search when any variable has no legal values.
Forward checking

Idea: Keep track of remaining legal values for unassigned variables. Terminate search when any variable has no legal values.
Constraint propagation

Forward checking propagates information from assigned to unassigned variables, but doesn’t provide early detection for all failures:

\[\text{WA} \rightarrow \text{NT} \rightarrow \text{Q} \rightarrow \text{NSW} \rightarrow \text{V} \rightarrow \text{SA} \rightarrow \text{T}\]

\(\text{NT}\) and \(\text{SA}\) cannot both be blue!

Constraint propagation repeatedly enforces constraints locally

Also called consistency enforcement techniques
Consistency enforcement

Consistency enforcement techniques

- Arc-consistency (Waltz, 1972)
- Path-consistency (Montanari 1974, Mackworth 1977)
- I-consistency (Freuder 1982)
- Transform the network into smaller and smaller networks.
Arc consistency

Simplest form of propagation makes each arc consistent

$X \rightarrow Y$ is consistent iff

for every value $x$ of $X$ there is some allowed $y$
Arc consistency

Simplest form of propagation makes each arc consistent

$X \rightarrow Y$ is consistent iff for every value $x$ of $X$ there is some allowed $y$
Arc consistency

Simplest form of propagation makes each arc consistent

\( X \rightarrow Y \) is consistent iff

for every value \( x \) of \( X \) there is some allowed \( y \)

If \( X \) loses a value, neighbors of \( X \) need to be rechecked
Arc consistency

Simplest form of propagation makes each arc consistent.

\[ X \rightarrow Y \] is consistent iff for every value \( x \) of \( X \) there is some allowed \( y \).

If \( X \) loses a value, neighbors of \( X \) need to be rechecked.

Arc consistency detects failure earlier than forward checking.

Can be run as a preprocessor or after each assignment.
Arc consistency algorithm

function AC-3 (csp) returns the CSP, possibly with reduced domains
inputs: csp, a binary CSP with variables \{X_1, X_2, \ldots, X_n\}
local variables: queue, a queue of arcs, initially all the arcs in csp

while queue is not empty do
    \((X_i, X_j) \leftarrow \text{REMOVE-FIRST}(queue)\)
    if \text{REMOVE-INCONSISTENT-VALUES}(X_i, X_j) then
        for each \(X_k\) in \text{NEIGHBORS}[X_i] do
            add \((X_k, X_i)\) to queue

function \text{REMOVE-INCONSISTENT-VALUES}(X_i, X_j) returns true iff succeeds
removed \leftarrow false
for each \(x\) in \text{DOMAIN}[X_i] do
    if no value \(y\) in \text{DOMAIN}[X_j] allows \((x,y)\) to satisfy the constraint \(X_i \leftarrow X_j\)
        then delete \(x\) from \text{DOMAIN}[X_i]; \ removed \leftarrow true
return removed

\(O(n^2d^3)\), can be reduced to \(O(n^2d^2)\) (but detecting all is NP-hard)
Arc-consistency

1 ≤ X, Y, Z, T ≤ 3
X < Y
Y = Z
T < Z
X ≤ T

\[ X \rightarrow Y \text{ is consistent iff for every value } x \text{ of } X \text{ there is some allowed } y \]
Arc-consistency

\[ 1 \leq X, Y, Z, T \leq 3 \]
\[ X < Y \]
\[ Y = Z \]
\[ T < Z \]
\[ X \leq T \]

\[ X \rightarrow Y \text{ is consistent iff} \]
\[ \text{for every value } x \text{ of } X \text{ there is some allowed } y \]
Arc-consistency

1 ≤ X, Y, Z, T ≤ 3
X < Y
Y = Z
T < Z
X ≤ T

X → Y is consistent iff
for every value x of X there is some allowed y
Arc-consistency

$1 \leq X, Y, Z, T \leq 3$

$X < Y$

$Y = Z$

$T < Z$

$X \leq T$
Arc-consistency

1 ≤ X, Y, Z, T ≤ 3
X < Y
Y = Z
T < Z
X ≤ T

X → Y is consistent iff for every value x of X there is some allowed y.
Arc-consistency

1 ≤ X, Y, Z, T ≤ 3
X < Y
Y = Z
T < Z
X ≤ T

X → Y is consistent iff for every value x of X there is some allowed y
Arc-consistency

\[ 1 \leq X, Y, Z, T \leq 3 \]

- \( X < Y \)
- \( Y = Z \)
- \( T < Z \)
- \( X \leq T \)

\( X \rightarrow Y \) is consistent iff for every value \( x \) of \( X \) there is some allowed \( y \)
Arc-consistency

1 \leq X, Y, Z, T \leq 3
X < Y
Y = Z
T < Z
X \leq T

• Incorporated into backtracking search
Problem structure

Tasmania and mainland are independent subproblems

Identifiable as connected components of constraint graph
Problem structure contd.

Suppose each subproblem has $c$ variables out of $n$ total

Worst-case solution cost is $\frac{n}{c} \cdot d^c$, linear in $n$

E.g., $n = 80$, $d = 2$, $c = 20$

$2^{80} = 4$ billion years at 10 million nodes/sec

$4 \cdot 2^{20} = 0.4$ seconds at 10 million nodes/sec
Tree-structured CSPs

Theorem: if the constraint graph has no loops, the CSP can be solved in $O(nd^2)$ time.

Compare to general CSPs, where worst-case time is $O(d^n)$.

This property also applies to logical and probabilistic reasoning: an important example of the relation between syntactic restrictions and the complexity of reasoning.
Algorithm for tree-structured CSPs

1. Choose a variable as root, order variables from root to leaves such that every node’s parent precedes it in the ordering

2. For $j$ from $n$ down to 2, apply $\text{REMOVEINCONSISTENT}(\text{Parent}(X_j), X_j)$

3. For $j$ from 1 to $n$, assign $X_j$ consistently with $\text{Parent}(X_j)$
**Nearly tree-structured CSPs**

**Conditioning:** instantiate a variable, prune its neighbors’ domains

**Cutset conditioning:** instantiate (in all ways) a set of variables such that the remaining constraint graph is a tree

Cutset size $c \Rightarrow$ runtime $O(d^c \cdot (n - c)d^2)$, very fast for small $c$
Hill-climbing, simulated annealing typically work with "complete" states, i.e., all variables assigned

To apply to CSPs:
  - allow states with unsatisfied constraints
  - operators \texttt{reassign} variable values

Variable selection: randomly select any conflicted variable

Value selection by \texttt{min-conflicts} heuristic:
  - choose value that violates the fewest constraints
  - i.e., hillclimb with $h(n) = \text{total number of violated constraints}$
Example: 4-Queens

States: 4 queens in 4 columns \( (4^4 = 256 \text{ states}) \)

Operators: move queen in column

Goal test: no attacks

Evaluation: \( h(n) = \text{number of attacks} \)
Performance of min-conflicts

Given random initial state, can solve $n$-queens in almost constant time for arbitrary $n$ with high probability (e.g., $n = 10,000,000$)

The same appears to be true for any randomly-generated CSP except in a narrow range of the ratio

$$R = \frac{\text{number of constraints}}{\text{number of variables}}$$
Summary

CSPs are a special kind of problem:
  states defined by values of a fixed set of variables
  goal test defined by constraints on variable values

Backtracking = depth-first search with one variable assigned per node

Variable ordering and value selection heuristics help significantly

Forward checking prevents assignments that guarantee later failure

Constraint propagation (e.g., arc consistency) does additional work to constrain values and detect inconsistencies

The CSP representation allows analysis of problem structure

Tree-structured CSPs can be solved in linear time

Iterative min-conflicts is usually effective in practice
Propositional Satisfiability

Example: party problem

• If Alex goes, then Becky goes:  \( A \rightarrow B \) (or, \( \neg A \lor B \))
• If Chris goes, then Alex goes:  \( C \rightarrow A \) (or, \( \neg C \lor A \))
• Query:
  Is it possible that Chris goes to the party but Becky does not?

\[ \varphi = \{ \neg A \lor B, \neg C \lor A, \neg B, C \} \]

is the proposition satisfiable?
Unit Propagation

• Arc-consistency for cnfs.

• Involve a single clause and a single literal

• Example: \((A, \neg B, C) \land (B) \rightarrow (A, C)\)
Look-ahead for SAT

(Davis-Putnam, Logeman and Laveland, 1962)

\[
\text{DPLL}(\varphi)
\]

Input: A cnf theory \( \varphi \)
Output: A decision of whether \( \varphi \) is satisfiable.

1. \( \text{Unit\_propagate}(\varphi) \);
2. If the empty clause is generated, return(\text{false});
3. Else, if all variables are assigned, return(\text{true});
4. Else
5. \( Q = \text{some unassigned variable} \);
6. return( DPLL( \varphi \land Q) \lor \\
   \quad \text{DPLL}(\varphi \land \neg Q) )

Figure 5.13: The DPLL Procedure
Look-ahead for SAT: DPLL

element: \((\neg A \lor B) \land (\neg C \lor A) \land (A \lor B \lor D) \land (C)\)

(Davis-Putnam, Logeman and Laveland, 1962)
GSAT – local search for SAT
(Selman, Levesque and Mitchell, 1992)

1. For i=1 to MaxTries
2. Select a random assignment A
3. For j=1 to MaxFlips
4. if A satisfies all constraint, return A
5. else flip a variable to maximize the score
6. (number of satisfied constraints; if no variable
7. assignment increases the score, flip at random)
8. end
9. end

Greatly improves hill-climbing by adding restarts and sideway moves
WalkSAT
(Selman, Kautz and Cohen, 1994)

Adds random walk to GSAT:
With probability $p$
  random walk – flip a variable in some unsatisfied constraint
With probability $1-p$
  perform a hill-climbing step

Randomized hill-climbing often solves large and hard satisfiable problems
More Stochastic Search

• Simulated annealing:
  – A method for overcoming local minimas
  – Allows bad moves with some probability:
    • With some probability related to a temperature parameter T the next move is picked randomly.
  – Theoretically, with a slow enough cooling schedule, this algorithm will find the optimal solution. But so will searching randomly.

• Breakout method (Morris, 1990): adjust the weights of the violated constraints