Overview

- Constraint Satisfaction Problems (CSP) share some common features and have specialized methods
  - View a problem as a set of variables to which we have to assign values that satisfy a number of problem-specific constraints.
  - Constraint solvers, constraint logic programming…
- Algorithms for CSP
  - Backtracking (systematic search)
  - Variable ordering heuristics

Informal Definition of CSP

- CSP = Constraint Satisfaction Problem
- Given \( (V, D, C) \)
  1. \( V \): a finite set of variables
  2. \( D \): a domain of possible values (often finite)
  3. \( C \): a set of constraints that limit the values the variables can take on
- A solution is an assignment of a value to each variable such that all the constraints are satisfied.
- Tasks might be to decide if a solution exists, to find a solution, to find all solutions, or to find the “best solution” according to some metric \( f \).

Example: Path of Length \( k \)

- Given an undirected graph \( G = (V, E) \), does \( G \) have a simple path of length \( k \)?
- Variables: \( x_0, x_1, \ldots, x_k \)
- Domain of variables: \( V \)
- Constraints:
  - (a) all values to \( x_i \) are distinct;
  - (b) \((x_i, x_{i+1})\) is in \( E \).

Example: Vertex Cover of Size \( k \)

- Given an undirected graph \( G = (V, E) \), does \( G \) have a vertex cover of size \( k \)?
- Variables: \( X = \{ x_1, x_2, \ldots, x_k \} \)
- Domain of variables: \( V \)
- Constraints:
  - (a) \( x_i \) is in the vertex cover iff \( x_i \) is true;
  - (b) For each edge \((a, b)\) of \( E \), \( a \) or \( b \) is in \( X \).
Example: Clique of Size k

- Given an undirected graph $G = (V, E)$, does $G$ have a clique of size $k$?
- Variables: $X = \{x_1, x_2, \ldots, x_k\}$
- Domain of variables: $V$
- Constraints:
  - (a) all values to $x_i$ are distinct;
  - (b) $(x_i, x_j)$ is in $E$ for any $i$ and $j$.

Example: Map Coloring

- Color the following map using three colors (red, green, blue) such that no two adjacent regions have the same color.

```
E
D
A
B
C
```

- Variables: $A, B, C, D, E$ all of domain RGB
- Domains: RGB = \{red, green, blue\}
- Constraints: $A \neq B, A \neq C, A \neq E, A \neq D, B \neq C, C \neq D, D \neq E$
- One solution: $A=\text{red}, B=\text{green}, C=\text{blue}, D=\text{green}, E=\text{blue}$

Example: SATisfiability

- Given a set of propositional variables and Boolean formulas, find an assignment of the variables to \{false, true\} that satisfies the formulas.
- Example:
  - Boolean variables = \{A, B, C, D\}
  - Boolean formulas: $A \lor B \lor \neg C, \neg A \lor D, B \lor C \lor D$
  - (the first two equivalent to $C \Rightarrow A \lor B, A \Rightarrow D$)
  - Are satisfied by:
    - $A = \text{false}$
    - $B = \text{true}$
    - $C = \text{false}$
    - $D = \text{false}$

Example: N-queens Example (4 in our case)

- Standard test case in CSP research
- Variables are the rows: $r_1, r_2, r_3, r_4$
- Values are the columns: $\{1, 2, 3, 4\}$
- So, the constraints include:
  - $C_{r_1,r_2} = \{(1,3),(1,4),(2,4),(3,1),(4,1),(4,2)\}$
  - $C_{r_1,r_3} = \{(1,2),(1,4),(2,1),(2,3),(3,2),(3,4),(4,1),(4,3)\}$
  - Etc.
  - What do these constraints mean?

Real-world problems

- Scheduling
- Temporal reasoning
- Building design
- Planning
- Optimization/satisfaction
- Vision
- Graph layout
- Network management
- Natural language processing
- Molecular biology / genomics
- VLSI design

Example: Graph Coloring

- Color the following graph using three colors (red, green, blue) such that no two adjacent regions have the same color.

```
E
D
A
B
C
```
Formal definition of a CSP

A constraint satisfaction problem (CSP) consists of
- a set of variables \( X = \{x_1, x_2, \ldots, x_n\} \)
  - each with an associated domain of values \( \{d_1, d_2, \ldots, d_n\} \).
  - The domains are typically finite.
- a set of constraints \( \{c_1, c_2, \ldots, c_m\} \) where
  - each constraint defines a predicate which is a relation over a particular subset of \( X \).
  - E.g., \( C_i \) involves variables \( \{X_{i1}, X_{i2}, \ldots, X_{ik}\} \) and defines the relation \( R_i \subseteq D_{i1} \times D_{i2} \times \ldots \times D_{ik} \)

Typical Tasks for CSP

- Solutions:
  - Does a solution exist?
  - Find one solution
  - Find all solutions
  - Given a partial instantiation, do any of the above
- Transform the CSP into an equivalent CSP that is easier to solve.

Systematic search: Backtracking (backtrack depth-first search)

- Consider the variables in some order
- Pick an unassigned variable and give it a provisional value such that it is consistent with all of the constraints
- If no such assignment can be made, we’ve reached a dead end and need to backtrack to the previous variable
- Continue this process until a solution is found or we backtrack to the initial variable and have exhausted all possible values.
- DFS: When backtrack, a node is marked as “processed”
- CSP: When backtrack, a node is marked as “undiscovered”

Constraint Solving is Hard

Constraint solving is not possible for general constraints.

Example (Fermat’s Last Theorem):
- \( C: n > 2 \)
- \( C': a^n + b^n = c^n \)

Constraint programming separates constraints into
- basic constraints: complete constraint solving
- non-basic constraints: propagation (incomplete); search needed

Backtrack Search Procedure

```plaintext
BacktrackSearch(A, k)  // A = (a_1, a_2, \ldots, a_k)
if (k > n) {
  if (A is a solution) print(A)
} else {
  compute S_k // the domain of a_k
  while S_k != empty do
    a_k = an element in S_k
    S_k = S_k \ {a_k}
    BacktrackSearch(A, k+1)
}
```
Problems with backtracking

- Thrashing: keep repeating the same failed variable assignments
  - Consistency checking can help
  - Intelligent backtracking schemes can also help
- Inefficiency: can explore areas of the search space that aren’t likely to succeed
  - Variable ordering can help

Improving backtracking efficiency

- General-purpose methods can give huge gains in speed:
  - Which variable should be assigned next?
  - In what order should its values be tried?
  - Can we detect inevitable failure early?
**Most constrained variable**

- Most constrained variable: choose the variable with the fewest legal values
- a.k.a. minimum remaining values (MRV) heuristic

**Least constraining value**

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

**Knapsack problem**

**Problem definition**

**Input:**
- Weight of N items \{w_1, w_2, ..., w_n\}
- Cost of N items \{c_1, c_2, ..., c_n\}
- Knapsack limit S

**Output:**
- Selection for knapsack: \{x_1,x_2,...,x_n\}
  where \(x_i \in \{0,1\}\).

**Sample input:**
- \(w_i=(1,1,2,4,12)\)
- \(c_i=(1,2,2,10,4)\)
- \(S=15\)

**Subset-Sum Problem**

a Simplified Knapsack Problem

**Input:**
- Weight of N items \{w_1, w_2, ..., w_n\}
- Knapsack limit S

**Output:**
- Selection for knapsack: \{x_1,x_2,...,x_n\}
  where \(x_i \in \{0,1\}\).

**Sample input:**
- \(W=(2,3,5,7,11)\)
- \(S=15\)
The Sudoku Puzzle

- Number place
- Main properties
  - NP-complete [Yato 03]
  - Well-formed Sudoku: has 1 solution [Simonis 05]
- Minimal Sudoku
  - In a 9x9 Sudoku: smallest known number of givens is 17 [Royle]
- Symmetrical puzzles
  - Many axes of symmetry
  - Position of the givens, not their values
  - Often makes the puzzle non-minimal
- Level of difficulties
  - Varied ranking systems exist
  - Minimality and difficulty not related

Sudoku as a CSP (9x9 puzzles)

- Variables are the cells
- Domains are sets \{1, ..., 9\}
- Two models
  - Binary constraints: 810 all-different binary constraint between variables
  - Non-binary constraints: 27 all-different 9-ary constraints

Solving Sudoku as a CSP

- Search
  - Builds solutions by enumerating consistent combinations
- Constraint propagation
  - Removes values that do not appear in a solution
  - Example, arc-consistency:

```
  1 2 3 4 5 6 7 8 9
  8 9 7 1 2 3 4 5 6
  5 6 4 8 9 7 1 2 3
  6 7 5 9 8 4 2 3 1
  7 8 6 5 4 9 3 1 2
  3 1 4 9 8 6 2 7 5
  4 5 8 2 9 7 6 7 8
  2 3 9 6 1 5 8 7 4
  9 1 6 8 7 2 3 7 4
```

Search

- Backtrack search
  - Systematically enumerate solutions by instantiating one variable after another
  - Backtracks when a constraint is broken
  - Is sound and complete (guaranteed to find solution if one exists)
- Propagation
  - Cleans up domain of ‘future’ variables, during search, given current instantiations

The way people play

- “Cross-hash,” sweep through lines, columns, and blocks
- Pencil in possible positions of values
- Generally, look for patterns, some intricate, and give them funny names:
  - Naked single, locked pairs, swordfish, medusa, etc.
- ‘Identified’ dozens of strategies
  - Many fall under a unique constraint propagation technique
- But humans do not seem to be able to carry simple inference (i.e., propagation) in a systematic way for more than a few steps

SEND + MORE = MONEY

Assign distinct digits to the letters\n\[
\begin{align*}
S, E, N, D, M, O, R, Y
\end{align*}
\]

\[
\begin{align*}
\text{SEND} \\
\text{+ MORE} \\
\hline \\
\text{MONEY}
\end{align*}
\]

such that \[
\begin{align*}
S & E N D \\
+ M O R E \\
\hline \\
M O N E Y
\end{align*}
\]

holds.
SEND + MORE = MONEY

Assign distinct digits to the letters $S, E, N, D, M, O, R, Y$ such that $\begin{array}{c} S E N D \\ + M O R E \\ \hline = M O N E Y \end{array}$ holds.

Solution

\[
\begin{array}{c}
9 \hspace{2em} 5 \hspace{2em} 6 \hspace{2em} 7 \\
+ \hspace{2em} 1 \hspace{2em} 0 \hspace{2em} 8 \hspace{2em} 5 \\
\hline
1 \hspace{2em} 0 \hspace{2em} 6 \hspace{2em} 5 \hspace{2em} 2 \\
\end{array}
\]

Modeling

Formalize the problem as a CSP:

- Variables: $v_1, \ldots, v_n$
- Domains: $\mathbb{Z}$, integers
- Constraints: $c_1, \ldots, c_m \subseteq \mathbb{Z}^n$
- Problem: Find $a = (v_1, \ldots, v_n) \in \mathbb{Z}^n$ such that $a \in c_i$ for all $1 \leq i \leq m$

A Model for MONEY

- variables: $\{S, E, N, D, M, O, R, Y\}$
- constraints:
  - $c_1 = \{(S, E, N, D, M, O, R, Y) \in \mathbb{Z}^8 \mid 0 \leq S, E, N, D, M, O, R, Y \leq 9\}$
  - $c_2 = \{(S, E, N, D, M, O, R, Y) \in \mathbb{Z}^8 \mid 1000S + 100E + 10N + D + 1000M + 100O + 10R + E = 10000M + 1000O + 100N + 10E + Y\}$

A Model for MONEY (continued)

- more constraints
  - $c_3 = \{(S, E, N, D, M, O, R, Y) \in \mathbb{Z}^8 \mid S \neq 0\}$
  - $c_4 = \{(S, E, N, D, M, O, R, Y) \in \mathbb{Z}^8 \mid M \neq 0\}$
  - $c_5 = \{(S, E, N, D, M, O, R, Y) \in \mathbb{Z}^8 \mid S, E, N, D, M, O, R, Y \text{ all different}\}$

Solution for MONEY

\[
\begin{array}{c}
c_1 = \{(S, E, N, D, M, O, R, Y) \in \mathbb{Z}^8 \mid 0 \leq S, E, N, D, M, O, R, Y \leq 9\}
\end{array}
\]

\[
\begin{array}{c}
c_2 = \{(S, E, N, D, M, O, R, Y) \in \mathbb{Z}^8 \mid 1000S + 100E + 10N + D + 1000M + 100O + 10R + E = 10000M + 1000O + 100N + 10E + Y\}
\end{array}
\]

\[
\begin{array}{c}
c_3 = \{(S, E, N, D, M, O, R, Y) \in \mathbb{Z}^8 \mid S \neq 0\}
\end{array}
\]

\[
\begin{array}{c}
c_4 = \{(S, E, N, D, M, O, R, Y) \in \mathbb{Z}^8 \mid M \neq 0\}
\end{array}
\]

\[
\begin{array}{c}
c_5 = \{(S, E, N, D, M, O, R, Y) \in \mathbb{Z}^8 \mid S, E, N, D, M, O, R, Y \text{ all different}\}
\end{array}
\]

Solution: $(9, 5, 6, 7, 1, 0, 8, 2) \in \mathbb{Z}^8$
Propagate

SEND + MORE
= MONEY

S ∈ {0..9}
E ∈ {0..9}
N ∈ {0..9}
D ∈ {0..9}
M ∈ {0..9}
O ∈ {0..9}
R ∈ {0..9}
Y ∈ {0..9}

0 ≤ S,...,Y ≤ 9
S ≠ 0
M ≠ 0
S,...,Y all different

1000*S + 100*E + 10*N + D + 1000*M + 100*O + 10*R + E = 10000*M + 1000*O + 100*N + 10*E + Y

Propagate

SEND + MORE
= MONEY

S ∈ {1..9}
E ∈ {0..9}
N ∈ {0..9}
D ∈ {0..9}
M ∈ {1..9}
O ∈ {0..9}
R ∈ {0..9}
Y ∈ {0..9}

0 ≤ S,...,Y ≤ 9
S ≠ 0
M ≠ 0
S,...,Y all different

1000*S + 100*E + 10*N + D + 1000*M + 100*O + 10*R + E = 10000*M + 1000*O + 100*N + 10*E + Y

Branch

SEND + MORE
= MONEY

E = 4

S ∈ {9}
E ∈ {4..7}
N ∈ {5..8}
D ∈ {2..8}
M ∈ {1}
O ∈ {0}
R ∈ {2..8}
Y ∈ {2..8}

Propagate

SEND + MORE
= MONEY

S ∈ {9}
E ∈ {4..7}
N ∈ {5..8}
D ∈ {2..8}
M ∈ {1}
O ∈ {0}
R ∈ {2..8}
Y ∈ {2..8}

Propagate

SEND + MORE
= MONEY

S ∈ {9}
E ∈ {5..7}
N ∈ {6..8}
D ∈ {2..8}
M ∈ {1}
O ∈ {0}
R ∈ {2..8}
Y ∈ {2..8}

Branch

SEND + MORE
= MONEY

E = 5

S ∈ {9}
E ∈ {5..7}
N ∈ {6..8}
D ∈ {2..8}
M ∈ {1}
O ∈ {0}
R ∈ {2..8}
Y ∈ {2..8}
Optimization Problem

• Let CSP = \((V, D, C)\)
  1. \(V\): a finite set of variables
  2. \(D\): a domain of possible values (often finite)
  3. \(C\): a set of constraints that limit the values the variables can take on
• Define a numeric function \(f(V)\)
• A solution is an assignment of a value to each variable such that all the constraints are satisfied and \(f(V)\) is minimal (maximal).

Optimization: Longest Paths

What is the longest (simple) path from A to B?

DFS: When backtrack, a node is marked as “processed”
CSP: When backtrack, a node is unmarked as “discovered”

Algorithms for Optimization

Key Components:
• Propagation algorithms: identify propagation algorithms for optimization function
• Branching algorithms: identify branching algorithms that lead to good solutions early
• Exploration algorithms: extend existing exploration algorithms to achieve optimization

Optimization: Example

\(SEND + MOST = MONEY\)
SEND + MOST = MONEY

Assign distinct digits to the letters S, E, N, D, M, O, T, Y such that

\[
\begin{align*}
SEND \\
+ MOST
\end{align*}
\]

= MONEY holds and

MONEY is maximal.

General search strategies

Backtracking

Explore all alternatives
- Solution constructed by stepwise choices
- Decision tree
- Guarantees optimal solution
- Exponential time (slow)

Traveling salesman problem

Input: graph (V,E)

Problem: Find shortest path via all nodes and returning to start node.

Output: path (p_1, p_2, ..., p_n, p_{n+1})

Solution = A-B-C-F-H-E-D-G-A
Length = 22

Branch and Bound

- An enhancement of backtracking
- Applicable to optimization problems
- Uses a lower bound for the value of the objective function for each node (partial solution) so as to:
  - guide the search through state-space
  - rule out certain branches as "unpromising"
Branch and Bound

- The branch-and-bound design strategy is very similar to backtracking in that a state space tree is used to solve a problem.
- The differences are that the branch-and-bound method 1) does not limit us to any particular way of traversing the tree, and 2) is used only for optimization problems.
- A branch-and-bound algorithm computes a number (bound) at a node to determine whether the node is promising.

Branch and Bound

- The number is a bound on the value of the solution that could be obtained by expanding beyond the node.
- If that bound is no better than the value of the best solution found so far, the node is nonpromising. Otherwise, it is promising.
- The backtracking algorithm for the 0-1 Knapsack problem is actually a branch-and-bound algorithm.
- A backtracking algorithm, however, does not exploit the real advantage of using branch-and-bound.

Traveling Salesman Problem

Lower Bound 1

- Finding smallest element in the intercity distance matrix D and multiplying it by number of cities n

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>-3</td>
<td>1</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>b</td>
<td>3</td>
<td>-6</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>c</td>
<td>1</td>
<td>6</td>
<td>-4</td>
<td>2</td>
</tr>
<tr>
<td>d</td>
<td>5</td>
<td>7</td>
<td>4</td>
<td>-3</td>
</tr>
<tr>
<td>e</td>
<td>8</td>
<td>9</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Traveling Salesman Problem

Lower Bound 2

- For each city i, find the sum $s_i$ of the distances from city i to the two nearest cities;
- Compute the sum $s$ of these n numbers;
- Divide the result by 2
- And if all the distances are integers, round up the result to the nearest integer.

Traveling salesman example:

Find a path b goes before c

Bounding

- A bound on a node is a guarantee that any solution obtained from expanding the node will be:
  - Greater than some number (lower bound)
  - Or less than some number (upper bound)
- If we are looking for a maximal optimal (knapsack), then we need an upper bound
  - For example, if the best solution we have found so far has a profit of 12 and the upper bound on a node is 10 then there is no point in expanding the node
  - The node cannot lead to anything better than a 10
Bounding
• Recall that we could either perform a depth-first or a breadth-first search
  – Without bounding, it didn’t matter which one we used because we had to expand the entire tree to find the optimal solution
  – Does it matter with bounding?
  • Hint: think about when you can prune via bounding

We prune a branch (via bounding) when:
  (currentBestSolutionCost >= nodeBound)
• This tells us that we get more pruning if:
  – The currentBestSolution is high
  – And the nodeBound is low
• So we want to find a high solution quickly and we want the lowest possible upper bound
  – One has to factor in the extra computation cost of computing lower upper bounds vs. the expected pruning savings

Traveling Salesman Problem
Lower Bound 3
• For each row, subtract each entry by the minimal entry in that row; store the minimal entry.
• For each column, subtract each entry by the minimal entry in that column; store the minimal entry.
• Use the sum of all the stored entries in the above steps as the lower bound.
Branch and Bound

• Besides using the bound to determine whether a node is promising, we can compare the bounds of promising nodes and visit the children of the one with the best bound.
• This approach is called best-first search with branch-and-bound pruning. The implementation of this approach is a modification of the breadth-first search with branch-and-bound pruning.
**Best-first Search**

- We can implement this search using a priority queue.
- All child nodes are placed in the queue for later processing if they are promising.
- Calculate an integer value for each node that represents the minimum possible distance if we pick that node.
- If the minimum possible distance is greater than the best total so far, don’t expand the branch.

---

**Summary: Branch and bound**

- Branch and Bound is:
  - a general search method.
  - minimize a function \( f(x) \), where \( x \) is restricted to some feasible region.
- To apply branch and bound, one must have:
  - a means of computing a lower (upper) bound on an instance of the optimization problem.
  - a means of dividing the feasible region of a problem to create smaller subproblems.