Constraint Satisfaction and Backtrack Search

Chapter 13
CS3030 Algorithms

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: x0, x1, …, xk
• Domain of variables: V
• Constraints:
  – (a) all values to xi are distinct;
  – (b) (xi, xi+1) is in E.
• Note: If k = |V| – 1, the path is called Hamiltonian Path.

Example: Vertex Cover of Size k

• Given an undirected graph G = (V, E), does G have a vertex cover of size k?
• Variables: X = { x1, x2, …, xk }
• Domain of variables: V
• Constraints:
  – (a) all values to xi are distinct;
  – (b) For each edge (a, b) of E, a or b is in X.

Example: Vertex Cover of Size k

VERTEX COVER: Given a graph G = (V, E) and an integer k, is there a subset of vertices S ⊆ V such that |S| ≤ k, and for each edge, at least one of its endpoints is in S?

Ex. Is there a vertex cover of size = 4? Yes.
Ex. Is there a vertex cover of size = 3? No.
Example: Vertex Cover of Size k

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

Example: Clique of Size k

- Given an undirected graph \( G = (V, E) \), does \( G \) have a clique of size \( k \)? That is, does \( V \) have a subset \( S \) of size \( k \) such that for each pair \( (x, y) \) of points in \( S \), \( (x, y) \) is an edge of \( E \)?
- 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.

Example: Map Coloring

- 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: Map Coloring

- Variables: \( A, B, C, D, E \) of any planar map
- 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: \( A=\text{red}, B=\text{green}, C=\text{blue}, D=\text{green}, E=\text{blue} \)
Map Coloring

- Color a map so no two adjacent regions share the same color

- Two-coloring is easy (if possible)
- Three-coloring is very hard
- Four-coloring is possible but hard (the decision problem is easy)
- Five-coloring is complicated but fast

Planarity

**Def.** A graph is planar if it can be embedded in the plane in such a way that no two edges cross.

**Applications:** VLSI circuit design, computer graphics.

**Kuratowski's Theorem.** An undirected graph $G$ is non-planar iff it contains a subgraph homeomorphic to $K_5$ or $K_{3,3}$.

Planarity Testing

**Planarity testing.** [Hopcroft-Tarjan 1974]. $O(n)$. simple planar graph can have at most $3n$ edges

**Remark.** Many intractable graph problems can be solved in poly-time if the graph is planar; many tractable graph problems can be solved faster if the graph is planar.

Three-Coloring

- No known polynomial time solution
- Obvious approach:
  - Try all possible combinations in $O(3^n)$ time using Backtrack Search

Four-Coloring

- **Four-coloring theorem:** any planar map can be colored with at most four colors.
- First proposed by Francis Guthrie in 1852
- Studied extensively for 124 years
- Proven by Kenneth Appel and Wolfgang Haken in 1976
- Requires a computer to exhaustively examined a set of 1,936 specially selected maps
Four-Coloring Algorithms
• The four-coloring theorem doesn’t give you a good way to find a four-coloring
• Try all possible combinations in O(4^n) time

N-queens Example (4 in our case)
• Standard test case in CSP research
• Variables are the rows: r1, r2, r3, r4
• Values are the columns: {1, 2, 3, 4}
• So, the constraints include:
  – C r1,r2 = {(1,3),(1,4),(2,4),(3,1),(4,1),(4,2)}
  – C r1,r3 = {(1,2),(1,4),(2,3),(2,3),(3,2),(3,4), (4,1),(4,3)}
  – Etc.
  – What do these constraints mean?

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 ∨ B ∨ ~C, ~A ∨ D, B ∨ C ∨ D
  – (the first two equivalent to C → A ∨ B, A → D)
  – Are satisfied by
    A = false
    B = true
    C = false
    D = false

CSP is a good model for many Real-world problems
• Scheduling
• Temporal reasoning
• Building design
• Planning
• Optimization/satisfaction
• Vision
• Graph layout
• Network management
• Natural language processing
• Molecular biology / genomics
• VLSI design

Formal definition of a CSP
A constraint satisfaction problem (CSP) consists of
• a set of variables X = {x1, x2, … x_n}
  – each with an associated domain of values {d1, d2, … d_n}.
  – The domains are typically finite.
• a set of constraints {c1, c2, … 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} … X_{ik}} and defines the relation R_i ⊆ D_{i1} x D_{i2} x … 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.

Constraint Solving is Hard

Constraint solving is not possible for general constraints.
Example (Fermat’s Last Theorem):

\[ C: \quad n > 2 \]
\[ C': \quad a^n + b^n = c^n \]

Link: [Wiles’ proof of Fermat’s Last Theorem](https://en.m.wikipedia.org/wiki/Wiles%27_proof_of_Fermat%27s_Last_Theorem)

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

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.

Backtrack Search Procedure

```
BacktrackSearch(A, k)  {  // A = (a1, a2, …, an)
  if (k > n) {
    if (A is a solution) print(A)
  } else {
    compute Sk // the domain of ak
    while Sk != empty do
      ak = an element in Sk
      Sk = Sk – {ak}
      BacktrackSearch(A, k+1)
  }
```

Backtracking example
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

Most constraining variable

• Tie-breaker among most constrained variables
• Most constraining variable:
  – choose the variable with the most constraints on remaining variables
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 \( \{w_1, w_2, \ldots, w_n\} \)
- Cost of N items \( \{c_1, c_2, \ldots, c_n\} \)
- Knapsack limit S

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

Sample input:
\[
\begin{align*}
w_i &= \{1,1,2,4,12\} \\
c_i &= \{1,2,2,10,4\} \\
S &= 15
\end{align*}
\]

Subset-Sum Problem

*a Simplified Knapsack Problem*

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

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

Sample input:
\[
\begin{align*}
W &= \{2,3,5,7,11\} \\
S &= 15
\end{align*}
\]

The Sudoku Puzzle

- **Number place**
  - Main properties
    - NP-complete
    - Well-formed Sudoku: has 1 solution
    - Minimal Sudoku
      - In a 9x9 Sudoku: smallest known number of givens is 17
      - 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

- Variables are the cells
- Domains are sets \( \{1, \ldots, 9\} \)
- 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:

<table>
<thead>
<tr>
<th>2</th>
<th>5</th>
<th>3</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>9</td>
<td>6</td>
</tr>
</tbody>
</table>

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
S, E, N, D, M, O, R, Y

such that

\[
\begin{align*}
\text{SEND} + \text{MORE} &= \text{MONEY} \\
9 & 5 & 6 & 7 \\
+ & 1 & 0 & 8 & 5 \\
\hline
1 & 0 & 6 & 5 & 2
\end{align*}
\]

holds.

Modeling

Formalize the problem as a CSP:

• Variables: \( v_1, v_2, \ldots, v_n \)
• Domains: \( \mathbb{Z} \), integers
• Constraints: \( c_1, \ldots, c_m \in \mathbb{Z}^n \)
• 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:**
  \[
  c_1 = \{(S, E, N, D, M, O, R, Y) \in \mathbb{Z}^8 | 0 \leq S, \ldots, Y \leq 9 \}
  \\
  c_2 = \{(S, E, N, D, M, O, R, Y) \in \mathbb{Z}^8 | \\
  1000S + 100E + 10N + D + 1000M + 100O + 10R + E = 10000M + 1000O + 100N + 10E + Y \}
  \]

### Solution for **MONEY**

- More constraints
  \[
  c_3 = \{(S, E, N, D, M, O, R, Y) \in \mathbb{Z}^8 | S \neq 0 \}
  \\
  c_4 = \{(S, E, N, D, M, O, R, Y) \in \mathbb{Z}^8 | M \neq 0 \}
  \\
  c_5 = \{(S, E, N, D, M, O, R, Y) \in \mathbb{Z}^8 | S, \ldots, Y \text{ all different} \}
  \]

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

---

Propagate

\[
\begin{align*}
S &\in \{0, \ldots, 9\} \\
E &\in \{0, \ldots, 9\} \\
N &\in \{0, \ldots, 9\} \\
D &\in \{0, \ldots, 9\} \\
M &\in \{0, \ldots, 9\} \\
O &\in \{0, \ldots, 9\} \\
R &\in \{0, \ldots, 9\} \\
Y &\in \{0, \ldots, 9\} \\
\end{align*}
\]

\[
1000S + 100E + 10N + D + 1000M + 100O + 10R + E = 10000M + 1000O + 100N + 10E + Y
\]
Propagate

Propagate

Branch

Branch

Propagate

Propagate

Branch

Branch

Complete Search Tree

Complete Search Tree
Knight’s Legal Move
Can only move 1 x 2
...Or 2 x 1

A Knight’s Tour
In this problem:
...Every single square, once only!

Solving The Tour
Similar problems were studied 4000 years ago
Modern chess variety studied by various mathematicians
Today, computers can use backtracking to find solutions

One Possible Solution

Magic Knight’s Tour
Some tours can be semi-magic
An example of a semi-magic tour

There are no fully magic tours

Different Chess Boards
Define Knight’s Tour as CSP

- Variables: each cell on the board is a variable
- Domains: \{1..(n*m)\} // i represents the ith step in the move.
- Constraints: legal knight moves. i.e., if (x,y) = i, i<n*m, and (x’, y’) = i+1, then (x’, y’) is in \{(x-2,y-1), (x-2,y+1), (x-1,y-2), (x-1,y+2), (x+1,y-2), (x+1,y+2), (x+2,y-1), (x+2,y+1)\}.

Variables: n=3, m=4

\[
\begin{array}{cccc}
 v_1 & v_2 & v_3 & v_4 \\
 v_5 & v_6 & v_7 & v_8 \\
 v_9 & v_{10} & v_{11} & v_{12} \\
\end{array}
\]

A solution:

\[
\begin{array}{cccc}
 1 & 4 & 7 & 10 \\
 8 & 11 & 2 & 3 \\
 5 & 6 & 9 & 12 \\
\end{array}
\]

The Knight’s Graph

- Related to the Hamilton Path/Cycle problem

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” (black)
CSP: When backtrack, a node is unmarked as “discovered” (white)

Optimization: Example

\[
\text{SEND} + \text{MOST} = \text{MONEY}
\]
SEND + MOST = MONEY

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

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

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 (TSP)**

**Input:** graph (V,E)

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

**Traveling salesman problem**

**Input:** graph (V,E)

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

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

**Branch and Bound**

- It’s an enhancement of backtracking, but also works for other search techniques.
- Applicable to optimization problems
- Uses a lower (upper) bound for the min (max) value of the objective function for each node (partial solution) so as to:
  - guide the search through state-space by selecting most promising branches
  - rule out certain branches as “useless”
Traveling Salesman Problem

**Lower Bound 1:** Finding smallest element in the intercity distance matrix $D$ and multiplying it by number of cities $n$

Before we choose any edge, the shortest distance is 1 and the tour must have at least $1 \times 5$.

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

After choosing (a, c), the remaining shortest distance is 2 and the tour must have at least $1 + 4 \times 2 = 9$.

After choosing (a, c) and (c, e), the remaining shortest distance is 3 and the tour must have at least $1 + 2 + 3 \times 3 = 12$.

**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.

Before any edge is chosen, lowerbound2 = \[
\frac{[1+3]+[3+6]+[1+2]+[3+4]+[2+3]+[2+3]}{2} = 14
\]

Traveling salesman example:

Find a path where b goes before c

\[
b = \frac{[1+3]+[3+6]+[1+2]+[3+4]+[2+3]+[2+3]}{2} = 14
\]

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 (can be used in a breadth-first search.) 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 bound is an estimate 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 **useless**. Otherwise, it is promising.
- The backtracking algorithm for the 0-1 Knapsack problem is actually a branch-and-bound algorithm.

**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

**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.

```
<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>c</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>d</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>e</td>
<td>6</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>
```

\[ lb = 1 + 3 + 1 + 3 + 2 + 1 = 13 \]

```
<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>c</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>d</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>e</td>
<td>6</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>
```

\[ lb = 13 \]

```
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.