Constraint Satisfaction
Problems

Chapter 5
Sections 1 – 3
Outline

- Constraint Satisfaction Problems (CSP)
- Backtracking search for CSPs
- Local search for CSPs
Constraint satisfaction problems (CSPs)

- Standard search problem:
  - state is a “black box” – any data structure that supports successor function, heuristic function, and goal test

- 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
Example: Map-Coloring

- Variables: WA, NT, Q, NSW, V, SA, T
- Domains: $D_i = \{\text{red, green, blue}\}$
- Constraints: adjacent regions must have different colors
  
  e.g., WA $\neq$ NT, or (WA,NT) in \{(red,green),(red,blue),(green,red), (green,blue),(blue,red),(blue,green)\}
Example: Map-Coloring

- **Solutions** are **complete** and **consistent** assignments, e.g., WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green
Constraint graph

- **Binary CSP**: each constraint relates two variables
- **Constraint graph**: nodes are variables, arcs are constraints
Varieties of CSPs

- **Discrete variables**
  - **finite domains:**
    - $n$ variables, domain size $d \rightarrow O(d^n)$ complete assignments
    - e.g., Boolean CSPs, incl. ~Boolean satisfiability (NP-complete)
  - **infinite domains:**
    - integers, strings, etc.
    - e.g., job scheduling, variables are start/end days for each job
    - need a constraint language, e.g., $StartJob_1 + 5 \leq StartJob_3$

- **Continuous variables**
  - e.g., start/end times for Hubble Space Telescope observations
  - linear constraints solvable in polynomial time by linear programming
Varieties of constraints

- **Unary** constraints involve a single variable,
  - e.g., SA ≠ green

- **Binary** constraints involve pairs of variables,
  - e.g., SA ≠ WA

- **Higher-order** constraints involve 3 or more variables,
  - e.g., cryptarithmetic column constraints
Example: Task Scheduling

- T1 must be done during T3
- T2 must be achieved before T1 starts
- T2 must overlap with T3
- T4 must start after T1 is complete
Example: Cryptarithmetic

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: \textit{Alldiff}\((F, T, U, W, R, O)\)

\begin{itemize}
    \item \( O + O = R + 10 \cdot X_1 \)
    \item \( X_1 + W + W = U + 10 \cdot X_2 \)
    \item \( X_2 + T + T = O + 10 \cdot X_3 \)
    \item \( X_3 = F, \ T \neq 0, \ F \neq 0 \)
\end{itemize}
Real-world CSPs

- Assignment problems
  - e.g., who teaches what class
- Timetabling problems
  - e.g., which class is offered when and where?
- Transportation scheduling
- Factory scheduling

Notice that many real-world problems involve real-valued variables
Standard search formulation (incremental)

Let's start with the straightforward 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
  - fail if no legal assignments
- **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
   - use depth-first search
3. Path is irrelevant, so can also use complete-state formulation
CSP Search tree size

\[ b = (n - \ell) d \text{ at depth } \ell, \text{ hence } n! \cdot d^n \text{ leaves} \]

Variables: A, B, C, D
Domains: 1, 2, 3

Depth 1: 4 variables x 3 domains
  = 12 states

Depth 2: 3 variables x 3 domains
  = 9 states

Depth 3: 2 variables x 3 domains
  = 6 states

Depth 4: 1 variable x 3 domains
  = 3 states (leaf level)
Backtracking search

- Variable assignments are **commutative**, i.e.,
[ WA = red then NT = green ] same as [ NT = green then WA = red ]

- Only need to consider assignments to a *single* variable at each node
  - Fix an order in which we’ll examine the variables
  - \( b = d \) and there are \( d^n \) leaves

- Depth-first search for CSPs with single-variable assignments is called **backtracking search**
  - Is the basic uninformed algorithm for CSPs
  - Can solve \( n \)-queens for \( n \approx 25 \)
Backtracking search

function BACKTRACKING-SEARCH( csp ) returns a solution, or failure
  return RECURSIVE-BACKTRACKING( {}, csp )

function RECURSIVE-BACKTRACKING( assignment, csp ) returns a solution, or 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 according to 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
Backtracking example
Backtracking example
Backtracking example
Backtracking example
Exercise - paint the town!

- Districts across corners can be colored using the same color.
How would you color this map?
Consider its constraints?
Can you do better than blind search?
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

- Combining these heuristics makes 1000 queens feasible
Forward checking

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

![Map of Australia with state abbreviations]

<table>
<thead>
<tr>
<th></th>
<th>WA</th>
<th>NT</th>
<th>Q</th>
<th>NSW</th>
<th>V</th>
<th>SA</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>WA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Q</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NSW</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>V</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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:

- NT and SA cannot both be blue!

- **Constraint propagation** repeatedly enforces constraints locally
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 $y$.
More on arc consistency

- Arc consistency is based on a very simple concept
  - if we can look at just one constraint and see that \( x=v \) is impossible …
  - obviously we can remove the value \( x=v \) from consideration

- How do we know a value is impossible?
- If the constraint provides *no support* for the value
- e.g. if \( D_x = \{1,4,5\} \) and \( D_y = \{1, 2, 3\} \)
  - then the constraint \( x > y \) provides no support for \( x=1 \)
  - we can remove \( x=1 \) from \( D_x \)
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 $y$

- Arcs are directed, a binary constraint becomes two arcs

- NSW $\Rightarrow$ SA arc originally not consistent, is consistent after deleting blue
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 (re)checked
Arc Consistency Propagation

- When we remove a value from $D_x$, we may get new removals because of it.
- E.g. $D_x = \{1, 4, 5\}$, $D_y = \{1, 2, 3\}$, $D_z = \{2, 3, 4, 5\}$
  - $x > y, \ z > x$
  - As before we can remove 1 from $D_x$, so $D_x = \{4, 5\}$
  - But now there is no support for $D_z = 2, 3, 4$
  - So we can remove those values, $D_z = \{5\}$, so $z=5$
  - Before AC applied to $y-x$, we could not change $D_z$
- This can cause a chain reaction.
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 (re)checked
- Arc consistency detects failure earlier than forward checking
- Can be run as a preprocessor or after each assignment
Arc consistency algorithm AC-3

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) ← REMOVE-FIRST(queue)
    if RM-INCONSISTENT-VALUES(X_i, X_j) then
        for each X_k in NEIGHBORS[X_i] do
            add (X_k, X_i) to queue

function RM-INCONSISTENT-VALUES(X_i, X_j) returns true iff remove a value
removed ← false
for each x in DOMAIN[X_i] do
    if no value y in DOMAIN[X_j] allows (x,y) to satisfy constraint(X_i, X_j)
    then delete x from DOMAIN[X_i]; removed ← true
return removed

- Time complexity: \(O(n^2d^3)\)
Time complexity of AC-3

- CSP has $n^2$ directed arcs
- Each arc $X_i, X_j$ has $d$ possible values. For each value we can reinsert the neighboring arc $X_k, X_i$ at most $d$ times because $X_i$ has $d$ values
- Checking an arc requires at most $d^2$ time

\[
O(n^2 \times d \times d^2) = O(n^2d^3)
\]
Maintaining AC (MAC)

Like any other propagation, we can use AC in search

i.e. search proceeds as follows:
- establish AC at the root
- when AC3 terminates, choose a new variable/value
- re-establish AC given the new variable choice (i.e. maintain AC)
- repeat;
- backtrack if AC gives domain wipe out

The hard part of implementation is undoing effects of AC
Special kinds of Consistency

- Some kinds of constraint lend themselves to special kinds of arc-consistency
- Consider the all-different constraint
  - the named variables must all take different values
  - not a binary constraint
  - can be expressed as $n(n-1)/2$ not-equals constraints
- We can apply (e.g.) AC3 as usual
- But there is a much better option
All Different

- Suppose $D_x = \{2,3\} = D_y$, $D_z = \{1,2,3\}$
- All the constraints $x \neq y$, $y \neq z$, $z \neq x$ are all arc consistent
  - e.g. $x=2$ supports the value $z = 3$
- The single ternary constraint AllDifferent($x,y,z$) is not!
  - We must set $z = 1$
- A special purpose algorithm exists for All-Different to establish GAC in efficient time
  - Special purpose propagation algorithms are vital
K-consistency

- Arc Consistency (2-consistency) can be extended to k-consistency

- 3-consistency (path consistency): any pair of adjacent variables can always be extended to a third neighbor.
  - Catches problem with $D_x$, $D_y$ and $D_z$, as assignment of $D_z = 2$ and $D_x = 3$ will lead to domain wipe out.
  - But is expensive, exponential time

- $n$-consistency means the problem is solvable in linear time
  - As any selection of variables would lead to a solution

- In general, need to strike a balance between consistency and search.
  - This is usually done by experimentation.
Local search for CSPs

- Hill-climbing, simulated annealing typically work with "complete" states, i.e., all variables assigned.

- To apply to CSPs:
  - allow states with unsatisfied constraints
  - operators reassign variable values

- Variable selection: randomly select any conflicted variable

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

- **States**: 4 queens in 4 columns ($4^4 = 256$ states)
- **Actions**: move queen in column
- **Goal test**: no attacks
- **Evaluation**: $h(n) =$ number of attacks

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

- Iterative min-conflicts is usually effective in practice
Midterm test

- Five questions, first hour of class (be on time!)
- Topics to be covered (CSP is not on the midterm):
  - Chapter 2 – Agents
  - Chapter 3 – Uninformed Search
  - Chapter 4 – Informed Search
    - Not including the parts of 4.1 (memory-bounded heuristic search) and 4.5
  - Chapter 6 – Adversarial Search
    - Not including 6.5 (games with chance)
Homework #1

- Due today by 23:59:59 in the IVLE workbin.
- Late policy given on [website](#). Only one submission will be graded, whichever one is latest.
- Your **tagline** is used to generate the ID to identify your agent on the [scoreboard](#).
- If you don’t have an existing account fill out: [https://mysoc.nus.edu.sg/~eform/new](https://mysoc.nus.edu.sg/~eform/new) and send me e-mail ASAP.
Checklist for HW #1

- **Does it compile?**
- Is my code in a single file?
- Did I comment my code so that it’s understandable to the reader?
- Is the main class appropriately named?
- Did I place a unique tagline so I can identify my player on the scoreboard?