Lecture 5
Constraint Satisfaction Problems (CSPs)
Review: Informed Search

⇒ **Heuristic functions**, often denoted $h(x)$, are estimated **cost-to-go** from the current state to a goal state

- **Admissible heuristic**: $h(x)$ is an underestimate of the optimal cost
- **Consistent heuristic**: $h(x_G) = 0$ and $h(n) \leq c(n, a, n') + h(n')$
  - $c(n, a, n')$ is the actual cost from $n$ to $n'$ following action $a$
  - A consistent heuristic is also admissible but the reverse is not true

⇒ **Greedy best-first** and **A** * are similar to **uniform-cost search**

⇒ Only difference is how the **evaluation function** $f(x)$ is chosen

- **Greedy best-first search** uses only estimated cost-to-go, i.e., $f(x) = h(x)$
- **A** * uses cost-to-come + estimated cost-to-go, i.e., $f(x) = g(x) + h(x)$

⇒ Dijkstra’s is basically uniform-cost that does not stop at goal

<table>
<thead>
<tr>
<th></th>
<th>Complete?</th>
<th>Optimal?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greedy best-first</td>
<td>Yes, for graph search on finite state space</td>
<td>No</td>
</tr>
</tbody>
</table>
| A* best-first    | Yes, for finite state space | Tree search: with admissible $h(x)$
|                  |           | Graph search: with consistent $h(x)$ |
Review: Beyond Classical Search

- **Local search**: for solving problems with large state space and many locally optimal solutions, generally without the need to find a path

- **Hill-climbing**: starting at a random $x_I$, pick most promising successor as next state to expand. Stop when no progress can be made
  - Allow random restart and sideway moves to improve performance
  - **Local beam search**: like hill-climbing but work with $k$ states at each iteration

- **Simulated annealing**: randomly pick a successor, if improvement is obtained, always take the step, otherwise take a locally “bad” step with probability affected by the step itself and an annealing schedule

- **Genetic algorithms**: select promising pairs of individuals from a population using a fitness function, then do crossover and mutation to obtain a new generation

- Good to know: search in continuous domain (gradient descent/ascent), handling non-deterministic actions with partial observations
Constraint Satisfaction Problems: Examples


source: http://www.hiusa.org/
Constraint Satisfaction Problems: Definition

- A set of variables: \( X = \{x_1, \ldots, x_n\} \)
- A set of domains for the variables: \( \mathcal{D} = \{D_1, \ldots, D_n\} \)
- A set of constraints over the variables: \( \mathcal{C} = \{c_1, \ldots, c_m\} \)
- State: an assignment of values to a subset of \( X \)
- A consistent assignment is a state that satisfies all constraints
- A complete assignment: an assignment of values to all of \( X \)
- Solution: a consistent and complete assignment
Cryptarithmetic

⇒ Variables: $x_T, x_W, x_O, x_F, x_U, x_R, c_1, c_2$

⇒ Domains: $x_i \in \{0, \ldots, 9\}, c_j \in \{0,1\}$

⇒ Constraints

$⇒ x_O + x_O = x_R + 10c_1$

$⇒ x_W + x_W + c_1 = x_U + 10c_2$

$⇒ x_T + x_T + c_2 = x_O + 10x_F$

$⇒ x_i \neq x_j$

$⇒ x_T \neq 0, x_F \neq 0$
Eight-Queens

⇒ Variables: $X = \{x_{ij}\}, 1 \leq i, j \leq 8$

⇒ Domains: $\mathcal{D} = \{D_{ij}\}, D_{ij} = \{0,1\}$

⇒ Constraints

⇒ Total number of queens: $\sum x_{ij} = 8$

⇒ No two queens should attack

⇒ Rows and columns: $\sum_i x_{ij} \leq 1, \sum_j x_{ij} \leq 1$

⇒ Diagonals: $\sum_{k=1}^8 x_{k,k} \leq 1, \sum_{k=2}^8 x_{k+1,k} \leq 1, ...$

⇒ Is this unique?
Another CSP Formulation for Eight-Queens

⇒ Variables: $X = \{x_1, \ldots, x_8\}$
⇒ Domains: $D = \{D_1, \ldots, D_8\}, D_i = \{1, \ldots, 8\}$
⇒ Constraints
  ⇒ No two queens on same row: $x_i \neq x_j$
  ⇒ No diagonal attack: $|x_i - x_j| \neq |i - j|$
⇒ One can be very creative in formulating CSPs for the same problem
⇒ Can you provide another CSP formulation for 8-queens?
⇒ Some additional CSP examples can be found at http://www.csplib.org
The Map Coloring Problem

⇒ Task: color a map (say, the contiguous part of Australia)

⇒ Variables: $X = \{x_{WA}, x_{NT}, x_{SA}, x_{Q}, x_{NSW}, x_{V}\}$

⇒ Domains: $D_i = \{\text{red, green, blue}\}$

⇒ Constraints: $x_i \neq x_j$ if $i, j$ are adjacent

⇒ Easy if we use many colors, but
  ⇒ Too many colors will be difficult to tell apart
  ⇒ Printing many colors is more costly

⇒ Interesting fact about map coloring

**Four-Color Theorem.** Given a separation of the plane into contiguous regions, no more than four colors are need to color the regions so that no two adjacent regions share the same color.

⇒ The proof is done by computer

(Appel & Haken, 1976)
The **adjacency relationship** of regions in map coloring may be represented using a **planar graph** – a graph that can be drawn in the plane without any two edges crossing each other.
Solving CSP as a Standard Search Problem

- **States**: an assignment of some or all variables
- **Initial state**: an empty assignment
- **Action**: choose an unassigned variable and pick a value for the variable
  - Fail when no assignment action is consistent
- **Goal test**: the current assignment is complete and consistent

- Branching factor: (average) size of the domains, say $|D|$
- Search depth: the number of variables, $n$
- Search tree size: $n! \cdot |D|^n$
- Worst case time Complexity: $(nm) \cdot n! \cdot |D|^n$
- Can reduce to $(nm) \cdot |D|^n$ by fixing the order of variable assignment
  - $x_1 = a, x_2 = b$ is the same as $x_2 = b, x_1 = a$
Backtracking Search

Search with backtracking: essentially DFS with single variable assignment at each node expansion
Basic (Recursive) Backtracking

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

⇒ We can obtain additional efficiency through
⇒ Carefully choosing the order of variables to assign values
⇒ Carefully choosing the order of values to be assigned to a variable
⇒ If we must fail, then fail early
Choosing the Order of Variable Assignment

⇒ Most constrained variable heuristic
  ⇒ Variables with the least number of valid values
  ⇒ Also known as minimum remaining values heuristic
  ⇒ Example: next variable for

⇒ Best choice is $x_{SA}$ with a single color available
Choosing the Order of Variable Assignment

- **Most constrained variable** heuristic
  - Variables with the least number of valid values
  - Also known as *minimum remaining values* heuristic

- **Degree** heuristic
  - Degree of a graph node is the **number of neighbors** of that node
  - Sometimes all variables have same choices of values
  - Pick one with the largest degree, with the hope to make later selections “harder”, thus fail early
  - Example: which one to choose in \( \{x_{WA}, x_{NT}, x_{SA}, x_{Q}, x_{NSW}, x_{V}\} \)?
Choosing Values for a Selected Variable

⇒ Once a variable is decided, we need to pick among many possible values for that variable

⇒ **Least-constraining-value** heuristic
  
  ⇒ Pick the value that allow the neighboring variables the most choices of values
  
  ⇒ Example: suppose we pick $x_Q$ as the next variable to assign a value

![Diagram](image)

Allow $x_{SA}$ to be *green*
Early Detection of Failure

⇒ If we can detect failure early, large branch can be truncated quickly, speeding up the search

⇒ Basically, apply inference to reduce the search space and detect inconsistency

⇒ **Forward checking**: keep track of valid values and terminate search when no valid value is left. E.g.,

\[
\begin{array}{ccccccc}
W_A & Q & N_T & N_S & W_A & N_S & V & S_A \\
\hline
\text{valid values} & & & & & & &
\end{array}
\]
Constraint Propagation

⇒ Forward checking propagates information from assigned variable to unassigned variable, but does not provide early detection for all failures

⇒ The general method to do this is through constraint propagation

⇒ Basically, constraint propagation repeatedly enforces constraints locally

  ⇒ Arc consistency checks two variables
  ⇒ Path consistency checks three variables
  ⇒ $k$-consistency checks $k$ variables
Arc Consistency

⇒ The arc $x \rightarrow y$ is consistent if for every value of $x$ there is some valid value of $y$.

⇒ E.g., given

- $x_{SA} \rightarrow x_{NSW}$ is consistent: for $x_{SA} = \text{blue}$, we may let $x_{NSW} = \text{red}$.

- $x_{NSW} \rightarrow x_{SA}$ is inconsistent – why?
  ⇒ If we let $x_{NSW} = \text{blue}$, then there is no valid value for $x_{SA}$

⇒ When checking consistency, throw out values that cause inconsistencies

⇒ If $x$ loses a value, all $z \rightarrow x$ arcs must be rechecked
  ⇒ Note that $x \rightarrow z$ arcs do not need to be rechecked
Arc Consistency for Early Failure Detection

⇒ Arc consistency detects error earlier
⇒ Run before or after each assignment
AC-3 Arc Consistency Algorithm (Mackworth)

function AC-3(csp) returns false if an inconsistency is found and true otherwise
inputs: csp, a binary CSP with components (X, D, C)
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 REVISE(csp, X_i, X_j) then
        if size of D_i = 0 then return false
        for each X_k in X_i.NEIGHBORS - {X_j} do
            add (X_k, X_i) to queue
    return true

function REVISE(csp, X_i, X_j) returns true iff we revise the domain of X_i
revised ← false
for each x in D_i do
    if no value y in D_j allows (x,y) to satisfy the constraint between X_i and X_j then
        delete x from D_i
        revised ← true
return revised
Path Consistency and $k$-Consistency

Path consistency: $x_i, x_j$ are path consistent with respect to $x_m$ if for every assignment $x_i = a, x_j = b$ consistent with the constraint on \{x_i, x_j\}, there is an assignment $x_m = c$, that satisfies the constraints on \{x_i, x_m\} and \{x_m, x_j\}

$k$-consistency: generalization of arc and path consistency

Note that all these heuristic checks are, after all, heuristics

- If a problem is really hard, all these heuristics are likely to fail
- So, take these with a grain of salt and do not apply blindly

All these heuristic checks amounts to inference
Computational Complexity of CSP

- CSP contains many challenging problems including 3SAT
  - A set of \( n \) binary variables, \( x_1, \ldots, x_n \)
  - A set of \( m \) disjunctive clauses containing up to three variables, for example, \( c_1 = x_1 \lor \neg x_3 \lor x_7, \ldots, c_m \)
  - Question: is there an assignment to \( \{x_1, \ldots, x_n\} \) so that \( c_1 \land \cdots \land c_m = \text{true} \)?

- The 3SAT problem is \textbf{NP-hard} – no polynomial time algorithm is believed to exist for such problems

- Not all hope is lost! Some CSPs can be quite “easy”
  - E.g., when variable dependencies can be represented on a \textbf{tree}:

- Map coloring for trees is fairly easy due to the lack of circular constraints
- Arc consistency only need to be checked in one direction
- When checking \( x \rightarrow y \), \( y \) has not been checked, few restrictions over \( x \)
To Put it All Together

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

function BACKTRACK(assignment, csp) returns a solution, or failure
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-Variable(csp)  // Order variables
    for each value in ORDER-Domain-VALUES(var, assignment, csp) do  // Order values
        if value is consistent with assignment then
            add \{var = value\} to assignment
            inferences ← INFERENCE(csp, var, value)  // Check consistency
            if inferences ≠ failure then
                add inferences to assignment
                result ← BACKTRACK(assignment, csp)  // DFS backtracking
                if result ≠ failure then
                    return result
                return failure
            remove \{var = value\} and inferences from assignment
    return failure
```
Additional Resources and Exercises

⇒ Write a CSP formulation of a “4x4” Sudoku
⇒ *Think of a CSP problem you have in real life (e.g., daily schedule) and how you may formulate