Announcements

- Homework 1: Search
  - due tomorrow

- Project 1: Search
  - due Friday 5pm

- Contest 1: Search – optional but fun
  - due Sunday

- Homework 2: CSPs
  - due Monday
CS 188: Artificial Intelligence

Constraint Satisfaction Problems

Instructor: Anca Dragan
University of California, Berkeley

[These slides adapted from Dan Klein and Pieter Abbeel]
Is A* Optimal?

- What went wrong?
- Actual bad goal cost < estimated good goal cost
- We need estimates to be less than actual costs!
Admissible Heuristics
Idea: Admissibility

Inadmissible (pessimistic) heuristics break optimality by trapping good plans on the fringe.

Admissible (optimistic) heuristics slow down bad plans but never outweigh true costs.
Admissible Heuristics

- A heuristic $h$ is *admissible* (optimistic) if:

$$0 \leq h(n) \leq h^*(n)$$

where $h^*(n)$ is the true cost to a nearest goal.

- **Examples:**

  - ![Diagram 1](image1)
  - ![Diagram 2](image2)

- Coming up with admissible heuristics is most of what’s involved in using A* in practice.
Optimality of A* Tree Search
Optimality of A* Tree Search

Assume:
- A is an optimal goal node
- B is a suboptimal goal node
- h is admissible

Claim:
- A will exit the fringe before B
Optimality of A* Tree Search: Blocking

Proof:
0 Imagine B is on the fringe
0 Some ancestor n of A is on the fringe, too (maybe A!)
0 Claim: n will be expanded before B
    1. f(n) is less or equal to f(A)

\[ f(n) = g(n) + h(n) \quad \text{Definition of f-cost} \]
\[ f(n) \leq g(A) \quad \text{Admissibility of h} \]
\[ g(A) = f(A) \quad h = 0 \text{ at a goal} \]
Optimality of A* Tree Search: Blocking

Proof:
0 Imagine B is on the fringe
0 Some ancestor n of A is on the fringe, too (maybe A!)
0 Claim: n will be expanded before B
  1. \( f(n) \) is less or equal to \( f(A) \)
  2. \( f(A) \) is less than \( f(B) \)

\[
\begin{align*}
g(A) &< g(B) \quad \text{B is suboptimal} \\
f(A) &< f(B) \quad h = 0 \text{ at a goal}
\end{align*}
\]
**Optimality of A* Tree Search: Blocking**

Proof:
- Imagine B is on the fringe
- Some ancestor $n$ of A is on the fringe, too (maybe A!)
- Claim: $n$ will be expanded before B
  1. $f(n)$ is less or equal to $f(A)$
  2. $f(A)$ is less than $f(B)$
  3. $n$ expands before B
- All ancestors of A expand before B
- A expands before B
- A* search is optimal
Video of Demo Contours (Empty) -- UCS
Video of Demo Contours (Empty) -- Greedy
Video of Demo Contours (Empty) – A*
Video of Demo Empty Water Shallow/Deep – Guess Algorithm
Example: 8 Puzzle

- What are the states?
- How many states?
- What are the actions?
- How many successors from the start state?
- What should the costs be?

Admissible heuristics?
8 Puzzle I

- Heuristic: Number of tiles misplaced
- Why is it admissible?
- $h(\text{start}) = 8$
- This is a relaxed-problem heuristic

Start State

Goal State

<table>
<thead>
<tr>
<th>Average nodes expanded when the optimal path has...</th>
<th>4 steps</th>
<th>8 steps</th>
<th>12 steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>UCS</td>
<td>112</td>
<td>6,300</td>
<td>$3.6 \times 10^6$</td>
</tr>
<tr>
<td>TILES</td>
<td>13</td>
<td>39</td>
<td>227</td>
</tr>
</tbody>
</table>

Statistics from Andrew Moore
8 Puzzle II

- What if we had an easier 8-puzzle where any tile could slide any direction at any time, ignoring other tiles?

- Total Manhattan distance

- Why is it admissible?

- \( h(\text{start}) = 3 + 1 + 2 + ... = 18 \)

<table>
<thead>
<tr>
<th></th>
<th>Average nodes expanded when the optimal path has...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>...4 steps</td>
</tr>
<tr>
<td>TILES</td>
<td>13</td>
</tr>
<tr>
<td>MANHATTAN</td>
<td>12</td>
</tr>
</tbody>
</table>
How about using the *actual cost* as a heuristic?

- Would it be admissible?
- Would we save on nodes expanded?
- What’s wrong with it?

With A*: a trade-off between quality of estimate and work per node

- As heuristics get closer to the true cost, you will expand fewer nodes but usually do more work per node to compute the heuristic itself
Graph Search
Tree Search: Extra Work!

- Failure to detect repeated states can cause exponentially more work.
In BFS, for example, we shouldn’t bother expanding the circled nodes (why?)
Graph Search

- **Idea:** never expand a state twice

- **How to implement:**
  - Tree search + set of expanded states ("closed set")
  - Expand the search tree node-by-node, but...
    - Before expanding a node, check to make sure its state has never been expanded before
    - If not new, skip it, if new add to closed set

- **Important:** store the closed set as a set, not a list

- Can graph search wreck completeness? Why/why not?

- How about optimality?
A* Graph Search Gone Wrong?

State space graph

Closed Set: S B C A

Search tree
Consistency of Heuristics

- Main idea: estimated heuristic costs ≤ actual costs
  - Admissibility: heuristic cost ≤ actual cost to goal
    \[ h(A) \leq \text{actual cost from A to G} \]
  - Consistency: heuristic “arc” cost ≤ actual cost for each arc
    \[ h(A) - h(C) \leq \text{cost(A to C)} \]

- Consequences of consistency:
  - The f value along a path never decreases
    \[ h(A) \leq \text{cost(A to C)} + h(C) \]
  - A* graph search is optimal
Optimality of A* Search

- With a admissible heuristic, Tree A* is optimal.
- With a consistent heuristic, Graph A* is optimal.
  - See slides, also video lecture from past years for details.
- With h=0, the same proofs shows that UCS is optimal.
Tree Search Pseudo-Code

function **Tree-Search**(problem, fringe) return a solution, or failure

fringe ← Insert(make-node(initial-state[problem]), fringe)

loop do

  if fringe is empty then return failure

  node ← Remove-Front(fringe)

  if Goal-Test(problem, state[node]) then return node

  for child-node in Expand(state[node], problem) do
    fringe ← Insert(child-node, fringe)
  end

end
function $\text{GRAPH-SEARCH}(\text{problem}, \text{fringe})$ return a solution, or failure

$\text{closed} \leftarrow$ an empty set

$\text{fringe} \leftarrow \text{INSERT}(\text{MAKE-NODE(INITIAL-STATE}[\text{problem}]), \text{fringe})$

loop do

if $\text{fringe}$ is empty then return failure

$\text{node} \leftarrow \text{REMOVE-FRONT}(\text{fringe})$

if $\text{GOAL-TEST}(\text{problem}, \text{STATE}[\text{node}])$ then return node

if $\text{STATE}[\text{node}]$ is not in $\text{closed}$ then

   add $\text{STATE}[\text{node}]$ to $\text{closed}$

   for $\text{child-node}$ in $\text{EXPAND}(\text{STATE}[\text{node}], \text{problem})$ do

      $\text{fringe} \leftarrow \text{INSERT}(\text{child-node}, \text{fringe})$

   end

end
CS 188: Artificial Intelligence

Constraint Satisfaction Problems

Instructor: Anca Dragan

University of California, Berkeley

[These slides adapted from Dan Klein and Pieter Abbeel]
Constraint Satisfaction Problems

N variables

domain D

constraints

states
partial assignment

goal test
complete; satisfies constraints

successor function
assign an unassigned variable
What is Search For?

- Assumptions about the world: a single agent, deterministic actions, fully observed state, discrete state space

- Planning: sequences of actions
  - The path to the goal is the important thing
  - Paths have various costs, depths
  - Heuristics give problem-specific guidance

- Identification: assignments to variables
  - The goal itself is important, not the path
  - All paths at the same depth (for some formulations)
  - CSPs are specialized for identification problems
Constraint Satisfaction Problems

- Standard search problems:
  - State is a “black box”: arbitrary data structure
  - Goal test can be any function over states
  - Successor function can also be anything

- Constraint satisfaction problems (CSPs):
  - A special subset of search problems
  - State is defined by variables $X_i$ with values from a domain $D$ (sometimes $D$ depends on $i$)
  - 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:** $D = \{\text{red, green, blue}\}$

- **Constraints:** adjacent regions must have different colors
  - Implicit: WA $\neq$ NT
  - Explicit: $(\text{WA, NT}) \in \{(\text{red, green}), (\text{red, blue}), \ldots\}$

- **Solutions** are assignments satisfying all constraints:
  $
  \{\text{WA=red, NT=green, Q=red, NSW=green, V=red, SA=blue, T=green}\}$
Example: N-Queens

Formulation 1:

- Variables: $X_{ij}$
- Domains: $\{0, 1\}$
- Constraints

\[
\begin{align*}
\forall i, j, k \quad & (X_{ij}, X_{ik}) \in \{(0, 0), (0, 1), (1, 0)\} \\
\forall i, j, k \quad & (X_{ij}, X_{kj}) \in \{(0, 0), (0, 1), (1, 0)\} \\
\forall i, j, k \quad & (X_{ij}, X_{i+k,j+k}) \in \{(0, 0), (0, 1), (1, 0)\} \\
\forall i, j, k \quad & (X_{ij}, X_{i+k,j-k}) \in \{(0, 0), (0, 1), (1, 0)\} \\
\sum_{i,j} X_{ij} = N
\end{align*}
\]
Example: N-Queens

Formulation 2:
- Variables: $Q_k$
- Domains: \{1, 2, 3, \ldots N\}
- Constraints:
  - Implicit: $\forall i, j$ non-threatening($Q_i, Q_j$)
  - Explicit: $(Q_1, Q_2) \in \{(1, 3), (1, 4), \ldots\}$
    \ldots\ldots
Example: Cryptarithmetic

0 Variables:

\[ F \ T \ U \ W \ R \ O \ X_1 \ X_2 \ X_3 \]

0 Domains:

\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}

0 Constraints:

\[
\text{alldiff}(F, T, U, W, R, O)
\]

\[
O + O = R + 10 \cdot X_1
\]

\[
\ldots
\]
Example: Sudoku

- **Variables:**
  - Each (open) square
- **Domains:**
  - \{1, 2, ..., 9\}
- **Constraints:**
  - 9-way alldiff for each column
  - 9-way alldiff for each row
  - 9-way alldiff for each region
  (or can have a bunch of pairwise inequality constraints)
Solving CSPs
Standard Search Formulation

- Standard search formulation of CSPs

- States defined by the values assigned so far (partial assignments)
  - Initial state: the empty assignment, {} 
  - Successor function: assign a value to an unassigned variable 
  - Goal test: the current assignment is complete and satisfies all constraints

- We’ll start with the straightforward, naïve approach, then improve it
Search Methods

0 What would BFS do?

```{}
{WA=g} {WA=r} ... {NT=g} ...
```
Search Methods

0 What would BFS do?

0 What would DFS do?
  0 let’s see!

0 What problems does naïve search have?
Backtracking Search
Backtracking Search

- Backtracking search is the basic uninformed algorithm for solving CSPs

- Idea 1: One variable at a time
  - Variable assignments are commutative, so fix ordering -> better branching factor!
  - 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 step

- Idea 2: Check constraints as you go
  - I.e. consider only values which do not conflict previous assignments
  - Might have to do some computation to check the constraints
  - “Incremental goal test”

- Depth-first search with these two improvements is called backtracking search (not the best name)

- Can solve n-queens for n ≈ 25
Backtracking Example
Video of Demo Coloring – Backtracking
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

- Backtracking = DFS + variable-ordering + fail-on-violation
- What are the choice points?