CS 771
Artificial Intelligence
Informed Search
Outline

• Review limitations of uninformed search methods

• Informed (or heuristic) search
  – Uses problem-specific heuristics to improve efficiency
  – Best-first, A* (and if needed for memory limits, RBFS, SMA*)
  – Techniques for generating heuristics
  – A* is optimal with admissible (tree)/consistent (graph) heuristics
  – A* is quick and easy to code, and often works *very* well

• Heuristics
  – A structured way to add “smartness” to your solution
  – Provide *significant* speed-ups in practice
  – Still have worst-case exponential time complexity
Limitations of uninformed search

• Search Space Size makes search tedious
  – **Combinatorial Explosion**

• For example, 8-puzzle
  – Avg. solution cost is about 22 steps
  – branching factor ~ 3
  – Exhaustive search to depth 22:
    • $3.1 \times 10^{10}$ states
  – E.g., d=12, IDS expands 3.6 million states on average

• 24 puzzle has $10^{24}$ states (much worse)
Recall tree search...

This “strategy” is what differentiates different search algorithms.

```plaintext
function TREE-SEARCH(problem) returns a solution, or failure
    initialize the frontier using the initial state of problem
    loop do
        if the frontier is empty then return failure
        choose a leaf node and remove it from the frontier
        if the node contains a goal state then return the corresponding solution
        expand the chosen node, adding the resulting nodes to the frontier
```
Heuristic search

• **Idea**
  
  – use an evaluation function $f(n)$ as an estimate of node quality
  
  – Evaluation function $f(n)$ consists of two quantities
    
    • A function $g(n)$ that captures known path cost so far to node $n$
    
    • A **heuristic function** $h(n)$ is an estimate of (optimal) cost to goal from node $n$
  
  – The **evaluation function** $f(n) = g(n) + h(n)$ is estimate of total cost to goal through node $n$
  
  – $f(n)$ provides an **estimate** for the total cost
  
  – Expand the node $n$ with smallest $f(n)$

• **Implementation**
  
  – Order the nodes in frontier by increasing estimated cost

• **Search efficiency depends on heuristic quality!**
  
  – The better your heuristic, the faster your search!
Heuristic function

• **Heuristic**
  – **Definition**: a commonsense rule (or set of rules) intended to increase the probability of solving some problem
  – Same linguistic root as “Eureka” = “I have found it”
  – “using rules of thumb to find answers”

• **Heuristic function** $h(n)$
  – Estimate of (optimal) remaining cost from node $n$ to *goal*
  – Defined using only the *state* of node $n$
  – $h(n) = 0$ if $n$ is a goal node
  – **Example**: straight line distance from node $n$ to Bucharest
    • Note that this is not the true state-space distance
    • It is an estimate – actual state-space distance can be higher
  – Provides problem-specific knowledge to the search algorithm
Heuristic functions for 8-puzzle

• 8-puzzle
  – Avg. solution cost is about 22 steps
  – branching factor ~ 3
  – Exhaustive search to depth 22:
    • $3.1 \times 10^{10}$ states.
  – A good heuristic function can reduce the search process.

• Two commonly used heuristics
  – $h_1 = \text{the number of misplaced tiles}$
    • $h_1(s)=8$
  – $h_2 = \text{the sum of the distances of the tiles from their goal positions (Manhattan distance)}$.
    • $h_2(s)=3+1+2+2+2+3+3+2=18$
Romania with straight-line distance
Relationship of Search Algorithms

• Component of evaluation function
  – \( g(n) \) = known cost so far to reach \( n \)
  – \( h(n) \) = estimated (optimal) cost from \( n \) to goal
  – \( f(n) = g(n) + h(n) \) = estimated (optimal) total cost of path through \( n \) to goal

• Two extreme cases
  – Uniform Cost search sorts frontier by \( f(n) = g(n) \)
  – Greedy Best First search sorts frontier by \( f(n) = h(n) \)

• Combination of both
  – A* search sorts frontier by \( f(n) = g(n) + h(n) \)
    • Optimal for admissible/consistent heuristics
    • Generally the preferred heuristic search

• Memory-efficient versions of A* are available
  – RBFS, SMA*
Greedy best-first search

- $h(n) = \text{estimate of cost from } n \text{ to } \text{goal}$
  - e.g., $h(n) = \text{straight-line distance from } n \text{ to Bucharest}$

- Greedy best-first search expands the node that appears to be closest to goal
  - *Priority queue sort function* = $h(n)$
Greedy best-first search example

**Start state**: Arad  **Goal state**: Bucharest
Greedy best-first search example
Greedy best-first search example
Greedy best-first search example
Greedy best-first search example
Optimal Path
Properties of greedy best-first search

- **Complete?**
  - Tree version can get stuck in loops (Iasi to Fagaras)
  - Graph version is complete in finite spaces
- **Time?**
  - $O(b^m)$
  - A good heuristic can give **dramatic** improvement
- **Space?**
  - $O(b^m)$
  - Keeps all nodes in memory
- **Optimal?**
  - No
  - E.g., **Arad** $\rightarrow$ **Sibiu** $\rightarrow$ **Rimnicu Vilcea** $\rightarrow$ **Pitesti** $\rightarrow$ **Bucharest** is shorter!
A* search

- **Idea:** avoid paths that are already expensive
  - Generally the preferred simple heuristic search
  - Optimal if heuristic is: admissible(tree)/consistent(graph)
- **Evaluation function** $f(n) = g(n) + h(n)$
  - $g(n) =$ known path cost so far to node $n$
  - $h(n) =$ estimate of (optimal) cost to goal from node $n$
  - $f(n) = g(n) + h(n) =$ estimate of total cost to goal through node $n$
- **Priority queue sort function**
  - Based on $f(n)$
Admissible heuristics

- A heuristic $h(n)$ is admissible if for every node $n$
  \[ h(n) \leq h^*(n) \]
  - where $h^*(n)$ is the true cost to reach the goal state from $n$

- An admissible heuristic never overestimates the cost to reach the goal, i.e., it is optimistic (or at least, never pessimistic)
  - Example: $h_{SLD}(n)$ (never overestimates actual road distance)

- Theorem: If $h(n)$ is admissible, A* using TREE-SEARCH is optimal
Admissible heuristics

E.g., for the 8-puzzle:

• $h_1(n)$ = number of misplaced tiles
• $h_2(n)$ = total Manhattan distance
  (i.e., no. of squares from desired location of each tile)

• $h_1(S) = ?$
• $h_2(S) = ?$
Admissible heuristics

E.g., for the 8-puzzle:

- \( h_1(n) = \) number of misplaced tiles
- \( h_2(n) = \) total Manhattan distance
  (i.e., no. of squares from desired location of each tile)

\[ h_1(S) = 8 \]
\[ h_2(S) = 3+1+2+2+2+3+3+2 = 18 \]
Consistent heuristics (consistent => admissible)

- A heuristic is **consistent** if for every node $n$, every successor $n'$ of $n$ generated by any action $a$,
  \[ h(n) \leq c(n,a,n') + h(n') \]

- If $h$ is consistent, we have
  \[ f(n') = g(n') + h(n') \quad \text{(by def.)} \]
  \[ = g(n) + c(n,a,n') + h(n') \quad \text{(g(n')=g(n)+c(n,a,n'))} \]
  \[ \geq g(n) + h(n) = f(n) \quad \text{(consistency)} \]
  \[ f(n') \geq f(n) \]

- i.e., $f(n)$ is non-decreasing along any path

- Theorem:
  If $h(n)$ is consistent, $A^*$ using $\text{GRAPH-SEARCH}$ is optimal

**It’s the triangle inequality!**

keeps all checked nodes in memory to avoid repeated states
Admissible (Tree Search) vs. Consistent (Graph Search)

• Why two different conditions?
  – In graph search you often find a long cheap path to a node after a short expensive one, so you might have to update all of its descendants to use the new cheaper path cost so far
  – A consistent heuristic avoids this problem (it can’t happen)
  – Consistent is slightly stronger than admissible
  – Almost all admissible heuristics are also consistent

• Could we do optimal graph search with an admissible heuristic?
  – Yes, but you would have to do additional work to update descendants when a cheaper path to a node is found
  – A consistent heuristic avoids this problem
A* search example
A* search example
A* search example
A* search example
A* search example
Contours of A* Search

- A* expands nodes in order of increasing $f$ value
- Gradually adds "$f$-contours" of nodes
- Contour $i$ has all nodes with $f = f_i$, where $f_i < f_{i+1}$
Properties of A*

• **Complete?**
  - Yes
  - unless there are infinitely many nodes with \( f \leq f(G) \)
  - This can’t happen if step-cost \( \geq \varepsilon > 0 \)

• **Time/Space?**
  - Exponential \( O(b^d) \)
    - except if: \( | h(n) - h^*(n) | \leq O(\log h^*(n)) \)

• **Optimal?**
  - Yes
  - Tree-Search, admissible heuristic
  - Graph-Search, consistent heuristic

• **Optimally Efficient?**
  - Yes
  - no optimal algorithm with same heuristic is guaranteed to expand fewer nodes
Optimality of A* (proof)

Suppose some suboptimal goal $G_2$ has been generated and is in the frontier. Let $n$ be an unexpanded node in the frontier such that $n$ is on a shortest path to an optimal goal $G$.

We want to prove:

$$f(n) < f(G_2)$$

(then A* will prefer $n$ over $G_2$)

- $f(G_2) = g(G_2)$ since $h(G_2) = 0$
- $f(G) = g(G)$ since $h(G) = 0$
- $g(G_2) > g(G)$ since $G_2$ is suboptimal
- $f(G_2) > f(G)$ from above
- $h(n) \leq h^*(n)$ since $h$ is admissible (under-estimate)
- $g(n) + h(n) \leq g(n) + h^*(n)$ from above
- $f(n) \leq f(G)$ since $g(n) + h(n) = f(n)$ and $g(n) + h^*(n) = f(G)$
- $f(n) < f(G2)$ from above
Memory Bounded Heuristic Search: Recursive Best First Search (RBFS)

• How can we solve the memory problem for A* search?

• **Idea:** Try something like best-first search, but let’s not forget everything about the branches we have partially explored
  – It is similar to recursive depth search BUT
  – rather than continuing indefinitely down the current path keep track of the best alternative path from any ancestor of current node
    • If current node cost exceeds the best alternative path from any of its ancestors, recursion unwinds back to the alternative path

• We remember the best f(n) value we have found so far in the branch we are deleting.
RBFS:

- RBFS changes its mind very often in practice.
- This is because the $f=g+h$ become more accurate (less optimistic) as we approach the goal.
- Hence, higher level nodes have smaller $f$-values and will be explored first.
- Problem: We should keep in memory whatever we can.

**best alternative over frontier nodes, which are not children: i.e. do I want to back up?**
Simple Memory Bounded A* (SMA*)

- This is like A*, but when memory is full we delete the worst node (largest f-value).
- Like RBFS, we remember the best descendent in the branch we delete.
- If there is a tie (equal f-values) we delete the oldest nodes first.
- SMA* finds the optimal *reachable* solution given the memory constraint.
- Time can still be exponential.

A Solution is not reachable if a single path from root to goal does not fit into memory.
Memory Bounded A* Search

• The Memory Bounded A* Search is the best of the search algorithms we have seen so far
  – It uses all its memory and uses smart heuristics to first descend into promising branches of the search-tree

• If memory not a problem, then plain A* search is easy to code and performs well
Heuristic functions

- **8-puzzle**
  - Avg. solution cost is about 22 steps
  - Branching factor ~ 3
  - Exhaustive search to depth 22:
    - $3.1 \times 10^{10}$ states
  - A good heuristic function can reduce the search process

- **Two commonly used heuristics**
  - $h_1 = \text{the number of misplaced tiles}$
    - $h_1(s)=8$
  - $h_2 = \text{the sum of the axis-parallel distances of the tiles from their goal positions (Manhattan distance)}$.
    - $h_2(s)=3+1+2+2+2+3+3+2=18$
Dominance

• If $h_2(n) \geq h_1(n)$ for all $n$ (both admissible) then $h_2$ dominates $h_1$
  – $h_2$ is always better for search than $h_1$
  – $h_2$ guarantees to expand no more nodes than does $h_1$
  – $h_2$ almost always expands fewer nodes than does $h_1$

• Typical 8-puzzle search costs (average number of nodes expanded):
  – $d=12$  
    IDS = 3,644,035 nodes
    $A^*(h_1) = 227$ nodes
    $A^*(h_2) = 73$ nodes
  – $d=24$  
    IDS = too many nodes
    $A^*(h_1) = 39,135$ nodes
    $A^*(h_2) = 1,641$ nodes
Why is dominance always better?

• Suppose the optimal evaluation function \( f(n) = g(n) + h(n) \) cost is \( C^* \)
  – All nodes \( n \), having value \( f(n) < C^* \) will surely be expanded
  – All nodes \( n \) having value \( h(n) < C^*- g(n) \) will surely be expanded

• Suppose \( h_2(n) \geq h_1(n) \) for all \( n \) (both admissible)
  then \( h_2 \) dominates \( h_1 \)
  – \( g(n) \) is same in both the cases the set of nodes expanded are
    • \( A_1 = \{ n : h_1 (n) < C^* \} \)
    • \( A_2 = \{ n : h_2 (n) < C^* \} \)
  – Take any node \( n \) in \( A_2 \)
    • \( C^*>h_2(n)>h_1(n) \)
    • \( n \) is in \( A_1 \)
Effective branching factor: $b^*$

- Let A* generate $N$ nodes to find a goal at depth $d$
  - $b^*$ is the branching factor that a uniform tree of depth $d$ would have in order to contain $N+1$ nodes

\[
N + 1 = 1 + b^* + (b^*)^2 + \ldots + (b^*)^d
\]

\[
N + 1 = ((b^*)^{d+1} - 1)/(b^* - 1)
\]

\[
N \approx (b^*)^d \Rightarrow b^* \approx \sqrt[d]{N}
\]

- For sufficiently hard problems, the measure $b^*$ usually is fairly constant across different problem instances

- A good guide to the heuristic’s overall usefulness
- A good way to compare different heuristics
Effectiveness of different heuristics

<table>
<thead>
<tr>
<th>d</th>
<th>IDS</th>
<th>$A^*(h_1)$</th>
<th>$A^*(h_2)$</th>
<th>IDS</th>
<th>$A^*(h_1)$</th>
<th>$A^*(h_2)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>10</td>
<td>6</td>
<td>6</td>
<td>2.45</td>
<td>1.79</td>
<td>1.79</td>
</tr>
<tr>
<td>4</td>
<td>112</td>
<td>13</td>
<td>12</td>
<td>2.87</td>
<td>1.48</td>
<td>1.45</td>
</tr>
<tr>
<td>6</td>
<td>680</td>
<td>20</td>
<td>18</td>
<td>2.73</td>
<td>1.34</td>
<td>1.30</td>
</tr>
<tr>
<td>8</td>
<td>6384</td>
<td>39</td>
<td>25</td>
<td>2.80</td>
<td>1.33</td>
<td>1.24</td>
</tr>
<tr>
<td>10</td>
<td>47127</td>
<td>93</td>
<td>39</td>
<td>2.79</td>
<td>1.38</td>
<td>1.22</td>
</tr>
<tr>
<td>12</td>
<td>3644035</td>
<td>227</td>
<td>73</td>
<td>2.78</td>
<td>1.42</td>
<td>1.24</td>
</tr>
<tr>
<td>14</td>
<td>539</td>
<td>113</td>
<td>113</td>
<td>–</td>
<td>1.44</td>
<td>1.23</td>
</tr>
<tr>
<td>16</td>
<td>–</td>
<td>1301</td>
<td>211</td>
<td>–</td>
<td>1.45</td>
<td>1.25</td>
</tr>
<tr>
<td>18</td>
<td>–</td>
<td>3056</td>
<td>363</td>
<td>–</td>
<td>1.46</td>
<td>1.26</td>
</tr>
<tr>
<td>20</td>
<td>–</td>
<td>7276</td>
<td>676</td>
<td>–</td>
<td>1.47</td>
<td>1.27</td>
</tr>
<tr>
<td>22</td>
<td>–</td>
<td>18094</td>
<td>1219</td>
<td>–</td>
<td>1.48</td>
<td>1.28</td>
</tr>
<tr>
<td>24</td>
<td>–</td>
<td>39135</td>
<td>1641</td>
<td>–</td>
<td>1.48</td>
<td>1.26</td>
</tr>
</tbody>
</table>

- Results averaged over random instances of the 8-puzzle
Inventing heuristics

- \( h(n) = \max\{ h_1(n), h_2(n), \ldots, h_k(n) \} \)
  - Assume all \( h \) functions are admissible
  - E.g., \( h_1(n) = \# \) of misplaced tiles
  - E.g., \( h_2(n) = \) Manhattan distance, etc.
  - \( \max \) chooses least optimistic heuristic (most accurate) at each node

- \( h(n) = w_1 h_1(n) + w_2 h_2(n) + \ldots + w_k h_k(n) \)
  - A convex combination of features
    - Weighted sum of \( h(n) \)'s, where weights sum to 1
  - Weights learned via repeated puzzle-solving
  - Try to identify which features are predictive of path cost
Summary

• Uninformed search methods have uses, also severe limitations

• Heuristics are a structured way to add “smartness” to your search

• Informed (or heuristic) search uses problem-specific heuristics to improve efficiency
  – Best-first, A* (and if needed for memory limits, RBFS, SMA*)
  – Techniques for generating heuristics
  – A* is optimal with admissible (tree)/consistent (graph) heuristics

• Can provide significant speed-ups in practice
  – E.g., on 8-puzzle, speed-up is dramatic
  – Still have worst-case exponential time complexity