Uninformed search

- Depth-first
- Depth-limited
- Iterative deepening
- Breadth-first
- Bidirectional search

- None of these searches take into account how close you are to the goal.

- (Q) What if search cost is not the number of moves?
Function $\text{UNIFORM-COST-SEARCH}(\text{problem})$ returns a solution, or failure

- $\text{node} \leftarrow$ a node with $\text{State} = \text{problem}.\text{Initial-State}$, $\text{Path-Cost} = 0$
- $\text{frontier} \leftarrow$ a priority queue ordered by $\text{Path-Cost}$, with $\text{node}$ as the only element
- $\text{explored} \leftarrow$ an empty set

Loop do

- If $\text{Empty?}(\text{frontier})$ then return failure
- $\text{node} \leftarrow \text{Pop}(\text{frontier})$ /* chooses the lowest-cost node in $\text{frontier}$ */
- If $\text{problem.Goal-Test}(\text{node}.\text{State})$ then return $\text{Solution}(\text{node})$
- Add $\text{node}.\text{State}$ to $\text{explored}$
- For each $\text{action}$ in $\text{problem_ACTIONS}(\text{node}.\text{State})$ do
  - $\text{child} \leftarrow \text{Child-Node}(\text{problem, node, action})$
  - If $\text{child}.\text{State}$ is not in $\text{explored}$ or $\text{frontier}$ then
    - Insert($\text{child}.\text{frontier}$)
  - Else if $\text{child}.\text{State}$ is in $\text{frontier}$ with higher $\text{Path-Cost}$ then
    - Replace that $\text{frontier}$ node with $\text{child}$

Figure 3.13 Uniform-cost search on a graph. The algorithm is identical to the general graph search algorithm in Figure ??, except for the use of a priority queue and the addition of an extra check in case a shorter path to a frontier state is discovered. The data structure for $\text{frontier}$ needs to support efficient membership testing, so it should combine the capabilities of a priority queue and a hash table.
Uniform Cost Search

• (Q) What data structures are needed?
  • Frontier
  • Explored set
  • Move costs
  • $g$ cost for each node as we put it in explored set
Uniform cost search

• (Q) Perform uniform cost search from A to F
Priority Queue

• (Q) How is a priority queue implemented?

  • Typically as a min heap
    • $O(\log n)$ access where $n$ is size of heap
    • Dijkstra's algorithm $O(n \log n)$ instead of $O(n)$

  • Can use array of bins in some circumstances
    • Each bin stores elements with particular value
    • $O(1)$ access instead of $O(\log n)$
A* search

• We want the path to have the lowest possible cost.

• Estimate the total cost with the cost of the path so far plus an estimate of the cost to get to the goal:
  • \( f(n) = g(n) + h(n) \)

• (Note that uniform cost search just uses \( g(n) \))
A* search

(Q) Perform A* search from A to F using the $h$ values below:

- $h(A) = 21$
- $h(B) = 30$
- $h(C) = 22$
- $h(D) = 12$
- $h(E) = 19$
- $h(F) = 0$
A* search contours

- A* expands towards the goal rather than in concentric circles.
Conditions for Optimality

• Admissibility
  • h(n) never overestimates the true cost, c(n)
    • h(n) \leq c(n)

• Consistency (monotonicity)
  • “A heuristic h(n) is consistent if, for every node n and every successor n’ generated by any action a, the estimated cost of reaching the goal from n is no greater than the step cost of getting to n’ plus the estimated cost of reaching the goal from n’.”
    • h(n) \leq c(n,a,n’) + h(n’)


Sliding Puzzle

• Find sequence of moves to get from start to goal state.

• (Q) For this problem: What are
  • states?
  • transition model?
  • path cost?
Sliding Puzzle Heuristics

- $h_1(n) =$ number of misplaced tiles
- $h_2(n) =$ Sum of Manhattan distances to goal

Start State

Goal State
## Search Costs

### NODES GENERATED

<table>
<thead>
<tr>
<th>depth</th>
<th>IDS</th>
<th>A*(h1)</th>
<th>A*(h2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>680</td>
<td>20</td>
<td>18</td>
</tr>
<tr>
<td>12</td>
<td>3644035</td>
<td>227</td>
<td>73</td>
</tr>
<tr>
<td>16</td>
<td>1301</td>
<td>211</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>39135</td>
<td>1641</td>
<td></td>
</tr>
</tbody>
</table>

Effective branch factors at depth 12:

- **IDS:** 2.78
- **A*(h1):** 1.42
- **A*(h2):** 1.22
Heuristics by Relaxation

• "Think of a search problem as a graph where the nodes are states and the edges are actions. The problem is to find a path connecting the initial state to a goal state. There are two ways we can relax a problem to make it easier: by adding more edges to the graph making it strictly easier to find a path, or by grouping multiple nodes together, forming an abstraction of the state space that has fewer states, and thus is easier to search."
Relaxation of sliding puzzle

• We can generate heuristic functions by "relaxing" the original problem description.

• Conditions of sliding puzzle
  • (a) A tile can move from A to B if they are adjacent
  • (b) A tile can move from A to B if B is blank
  • (c) A tile can move from A to B

• Removing condition (a) gives $h_2$
• Removing condition (c) gives $h_1$
Max heuristics

• Heuristics can be combined by taking the max:
  • $H(n) = \max\{ h_1(n) \ldots h_m(n) \}$
Heuristics for Subproblems

- We can make an admissible heuristic that solves part of the problem.
- Alternatively, we can solve a number of problems on the way to the full solution (likely not optimally).
Pattern Databases

• Store exact solution cost for every possible subproblem. Example: Cost to get 1234 in place, rather than all numbers.

• Disjoint pattern databases: Store multiple subproblems in which solution moves cannot overlap.
  • For 15 puzzle, 10000-fold speedup over Manhattan distance.
  • For 24 puzzle, a million-fold speedup over Manhattan distance.
Limited space variants

• Iterative Deepening A*
  
  • Essentially, same idea as iterative deepening, but use \( f (g+h) \) instead of depth.
  
  • One big question is what to add to the limit value each time.
  
  • (Q) Suggest ways how to do this
Limited space variants

• RBFS – recursive best first search
  • Keep track of best f-value from any ancestor of current node.
  • If current node exceeds this limit, the recursion unwinds back to alternate path.
  • As recursion unwinds, replace f-value at each node with best value of its children.
Limited space variants

• SMA* - simple memory bounded A*
  • Expand best leaf until memory full
  • To add next node, drop node with worst f value
  • If there is a tie, drop oldest of the tied nodes
  • It will actually regenerate nodes if needed