Informed Search – SMA*

- **advantages**
  - complete if depth of shallowest goal node ≤ max size of PQ
  - optimal if length of shallowest optimal solution path ≤ max size of PQ (returns best reachable solution otherwise)
  - “forgets” subtrees only when necessary (i.e. out of memory)
  - stores how worthwhile each forgotten subtree is, so only re-expands if all other paths turn out to be worse

- **disadvantages**
  - can essentially thrash if there are many candidate paths but only a few fit in memory – repeatedly re-expands subtrees as each path extension juggles the ordering of the paths (h(n) tends to be more accurate closer to the goal)

Observation

Once again, a tradeoff between time and space.
- solutions to address memory limitations can lead to a great deal of repeated work

Memory limitations can make an otherwise tractable problem intractable with respect to computation time.
- A* tends to run out of memory but can solve problems that cause SMA* to thrash (given unlimited memory)

Informed Search Roundup

- greedy best-first – expand nodes with minimal h(n)
  - not optimal
  - often efficient in practice
- A* - expand nodes with minimal g(n)+h(n)
  - complete and optimal if h(n) is admissible (tree search) or consistent (graph search)
  - space complexity often prohibitive
- bounded-memory (IDA*, RBFS, SMA*)
  - complete and optimal
  - linear space
  - can solve things A* cannot because A* runs out of memory
  - may suffer from excessive node re-expansion, rendering a problem that A* could solve (with enough memory) intractable
  - IDA* is practical in many cases with step costs, but can be poor otherwise

Heuristic Functions

But what about h(n)?

The performance of all informed search algorithms depends on the quality of the heuristic function h(n).

Good heuristics are problem-specific.
Choosing a Heuristic

- key properties (necessary for A* to be complete and optimal)
  - admissible – never overestimates the cost of the solution
  - consistent – \( h(n) \leq \text{cost}(n \rightarrow n') + h(n') \) i.e. the estimate of total solution cost can't decrease as you move along

...but there may be many admissible and consistent heuristics for a given problem.
- e.g. \( h(n) = 0 \) is always admissible and consistent, but not terribly useful

Choosing a Good Heuristic

- what if no one heuristic dominates?
  - can use \( h(n) = \max \{ h_1(n), h_2(n), \ldots, h_m(n) \} \)
    - if each heuristic is admissible, solution cost is at least as high as the most expensive
    - tradeoff is additional time to evaluate multiple heuristics

Evaluating Heuristics

- the running time and size of the solution space depends on \( b \), the branching factor
  - heuristics can be evaluated based on the effective branching factor

  effective branching factor \( b^* \) = branching factor in a tree with \( N+1 \) nodes and all leaves at depth \( d \)
  \[ b^* = 1 + b^* + (b^*)^2 + \ldots + (b^*)^d \]
  - \( N \) = number of nodes expanded during the search
  - \( d \) = depth of solution

- observations
  - \( b^* \) varies somewhat across instances of a problem, but tends to be fairly constant for sufficiently hard problems
  - can be measured experimentally using a small set of problem instances
  - the closer \( b^* \) is to 1, the better
Calculating \( b^* \)

- need to solve \( N+1 = 1 + b^* + (b^*)^2 + \ldots + (b^*)^d \) for \( b^* \)
- approximate
  - \( b^* \approx N^{(1/d)} \)
  - based on the idea that \((b^*)^d\) dominates the sum
  - fast to calculate, but less accurate
- solve numerically
  - use binary search
    - choose high and low estimates for \( b^* \)
    - average the estimates to get a guess for \( b^* \)
    - calculate \( N' \) using the guess for \( b^* \)
    - if \( N' \) is sufficiently close to \( N \), use the guess as \( b^* \)
    - otherwise, modify high or low estimate based on whether \( N' \) is bigger or smaller than \( N \)
  - as accurate as you want, but longer to calculate and subject to floating-point overflow problems for large \( d \)

Example – 8-puzzle

Are these heuristics good?
- both heuristics can significantly underestimate the true solution cost – but there is still a significant benefit over uninformed search

<table>
<thead>
<tr>
<th>algorithm / heuristic</th>
<th>( N ) (d=22)</th>
<th>( b^* )</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDS</td>
<td>~9,175,089,381</td>
<td>~2.78</td>
</tr>
<tr>
<td>A* with ( h_1 )</td>
<td>~17,167</td>
<td>~1.48</td>
</tr>
<tr>
<td>A* with ( h_2 )</td>
<td>~778</td>
<td>~1.26</td>
</tr>
</tbody>
</table>

- notes
  - average branching factor for the 8-puzzle is 2.67
  - average solution cost for randomly-generated puzzle is \( \sim 22 \) steps
  - actual number of distinct reachable states = 181,440

Coming Up With Heuristics

Strategies –
- a brilliant idea may occur to you
- solve a relaxed problem
- utilize pattern databases
- generate automatically
- learn a heuristic function

Example – 8-puzzle

\( h_1 \): number of misplaced tiles
  - admissible?
    - yes, any out-of-place tile must be moved at least once to be in its proper position (and each move only moves one tile)

\( h_2 \): sum of Manhattan distances of tiles from their goal positions
  - admissible?
    - yes, each move can move at most one tile one step closer to its proper position

Does one of these heuristics dominate?
- \( h_2 \), because the distance of a tile from its goal position is at least 1 for each misplaced tile
A Brilliant Idea May Occur to You

- brilliant ideas are more likely if you have a deep understanding of the problem

Solve a Relaxed Problem

- observations
  - any optimal solution to the original problem is a solution to the relaxed problem
  - relaxed problem may have better solutions
  - thus, solving the relaxed problem yields an admissible (and also consistent) heuristic for the original problem

- this is a useful tactic for a heuristic if the relaxed problem is sufficiently easy to solve
  - i.e. can be solved without search
    - e.g. computed outright
    - e.g. solved with a greedy algorithm

- observations
  - any optimal solution to the original problem is a solution to the relaxed problem
  - relaxed problem may have better solutions
  - thus, solving the relaxed problem yields an admissible (and also consistent) heuristic for the original problem

- this is a useful tactic for a heuristic if the relaxed problem is sufficiently easy to solve
  - i.e. can be solved without search
    - e.g. computed outright
    - e.g. solved with a greedy algorithm

Solve a Relaxed Problem

- relaxed = fewer constraints (and hopefully easier to solve)

- examples – 8-puzzle
  - the constraints control movement of tiles
    - can only move one spot at a time
    - can only move into an empty spot
  - allow a tile to move directly to any spot
    - number of steps in optimal solution = number of out-of-place tiles ($h_1$)
  - allow a tile to move to any adjacent spot, even if already occupied by another tile
    - number of steps in optimal solution = sum of Manhattan distances between tiles and their goal positions ($h_2$)

Example

- admissible heuristic for a maze?
  - what are the constraints?
Example

• admissible heuristic for missionaries and cannibals?
  – what are the constraints?
    – constraint: boat capacity
      • send everyone over at once so \( h(n) = 1 \)
      • admissible? yes, need at least one trip to get a person from the left side to the right (and at least two if the boat is currently on the right side)
    – constraint: cannibals can't outnumber missionaries
      • for left \(\rightarrow\) right trips, send two people in the boat (max number)
      • for right \(\rightarrow\) left trips, send one person in the boat (min number)
      • this amounts to a net increase of one on the right side for each roundtrip, so \( h(n) = 2 \times \) number of people on the left side – 1
        –1 because the last trip doesn't need to return the boat to the left side
      • admissible? yes, can't have net increase on right side of more than one per roundtrip; if boat starts on right, need at least one additional trip

Example

• admissible heuristics for the Rubik's cube?
  – number of incorrect cubies / 8
    • divide by 8 because one turn moves four corner and four edge cubies
  – sum of the minimum number of moves to correctly position and orient each cubie / 8
    • can be precomputed
  – max { sum of minimum number of moves to position and orient edge cubies / 4, sum of minimum number of moves to position and orient corner cubies / 4 }
  – sum of minimum number of moves to position and orient edge cubies / 4
    • lower value for \( h(n) \) but faster calculation

Utilize Pattern Databases

Pattern database stores exact cost for solving subproblems. (idea is that the subproblems are small enough to solve – both in time and space)

• admissible
  – have to at least solve the subproblem to solve the whole problem

Utilize Pattern Databases

• combining multiple pattern databases
  – \( h(n) = \max \{ h_1(n), h_2(n), \ldots, h_m(n) \} \)
    • each represents an admissible heuristic, so solution cost is at least as high as the highest
    • can sum \( h_1(n) + h_2(n) + \ldots + h_m(n) \) only if subproblems are independent (disjoint pattern database)
  – e.g. partition tiles into disjoint sets (e.g. 1-4 and 5-8) count only steps involving tiles in the current set when solving a subproblem (undercounts moves to solve a subproblem but prevents double-counting when patterns are combined)
  – take \( h(n) = \) steps to get 1-4 arranged + steps to get 5-8 arranged
Example

- pattern databases for Rubik's cube [Korf 1997]
  - store minimum number of moves to get all elements in the pattern to the correct position

  - corner cubie DB (88 million entries)
  - two edge cubie DBs (43 million entries each)
  - total space required: ~82MB (max value 11 for corners, 10 for edges → 4 bits to store each)

Utilize Pattern Databases

Constructing pattern databases –

- search backwards from goal, recording cost for each state encountered
- need an optimal search

Other Options

- generate heuristics automatically
  - express the problem in a way that allows automatic generation of relaxed problems
  - derive heuristics from relaxed problems

Example

- pattern databases for Rubik's cube [Korf 1997]

- heuristic
  - \( h(n) = \max \{ h_{\text{corners}}(n), h_{e_1}(n), h_{e_2}(n) \} \)
Automatic Generation of Heuristics

- **e.g. STRIPS representation for 8-puzzle**
  - **states** are a conjunction of predicates
    - e.g. \( \text{on}(x_1, p_1), \text{on}(x_2, p_2), \ldots, \text{on}(x_8, p_8), \text{clear}(p_9) \)
  - **rules** specify changes to predicate lists
    - e.g. \( \text{move}(x, p_1, p_2) \) moves tile \( x \) from position \( p_1 \) to \( p_2 \)
      - preconditions: \( \text{on}(x, p_1), \text{clear}(p_2), \text{adj}(p_1, p_2) \)
      - add list: \( \text{on}(x, p_2), \text{clear}(p_1) \)
      - delete list: \( \text{on}(x, p_1), \text{clear}(p_2) \)
  - **relaxation** involves dropping predicates from the preconditions
    - e.g. remove \( \text{clear}(p_2) \)
    - e.g. remove \( \text{adj}(p_1, p_2) \)
    - e.g. remove \( \text{clear}(p_2), \text{adj}(p_1, p_2) \)

Other Options

- **learn a heuristic function**
  - based on actual solution costs discovered during the search