Heuristic Search Methods

Kris Beevers
Intro to AI 9/15/03
Ch. 4.1-4.2

Overview

- “Informed” (**heuristic**) algorithms (as opposed to “uninformed” ones like BFS, DFS, etc.)
- Use problem-specific knowledge beyond the definition of the problem itself
- General approach: **best-first search**. Select node for expansion based on an evaluation function $f(n)$
  - Usually, “best-first” means pick the node with lowest $f(n)$
  - Note that “best-first” is inaccurate: if we really knew the lowest-cost node it wouldn’t be a search at all! Instead we pick the node that *appears* the best based on the evaluation function
- Searches we will study include **Greedy** searches (best means “closest to goal”) and **A*** (and related) searches (best means “lowest total estimated cost”)
- Key concept: **heuristic function** (a heuristic is a “rule of thumb”):

  $$h(n) = \text{estimated cost of cheapest path from node } n \text{ to the goal node}$$

  - Example: might estimate the cost of the shortest path from Troy to Syracuse as the straight-line distance
  - Assume that if n is goal node, $h(n) = 0$

Properties of Heuristics

- **Admissibility**: a heuristic $h(n)$ is admissible if it *never overestimates* the cost to the goal from node n; i.e. it is *always optimistic*
- **Consistency** or **monotonicity**: a heuristic $h(n)$ is consistent if for any nodes A and B, $h(B) \geq h(A) + c(A, B)$

- Intuitively, this says that our heuristic will become more accurate (less optimistic) as we approach the goal
- This is just a form of the *triangle inequality*—a heuristic is consistent iff it satisfies the triangle inequality

- Example: assume $h(n)$ is admissible and that it says we are 10 from the goal. The actual cost to the goal must be more—we are *at least* 10 from the goal.
- Suppose we then take a step of cost 1. If our heuristic is consistent, it cannot say we are closer than 9 to the goal. If our heuristic was admissible but not consistent, it could say we were 2 from the goal.
- **Consistency $\Rightarrow$ Admissiblity**

Greedy Search

- Algorithm:
  - Put the root node on a queue Q
  - Repeat:
    - if Q is empty, return failure
    - remove the node N *with the lowest* $h(\cdot)$ *value* from Q
    - if N is the goal, return success
    - add children of N to Q
- Just uses the heuristic function $f(n) = h(n)$
- Problems:
  - Susceptible to false starts (i.e. might end up expanding more nodes than necessary); like DFS, will tend to follow one solution all the way to the end (even if it isn't the best)
  - Not complete on infinite depth search trees
  - Not optimal
  - Time/space complexity: $O(b^m)$ (remember m is maximum depth of search tree, b is branching factor)

A* Search

- Let

$$\begin{aligned} g(n) &= \text{cost to reach node } n \\ h(n) &= \text{estimated cost from } n \text{ to the goal} \end{aligned}$$

- A* minimizes the *total solution cost*, using

$$f(n) = g(n) + h(n)$$

- Expand node with *lowest* $f(\cdot)$
- Note that if $h(n) = 0, \forall n$, we get uniform cost search!

Queue Implementation

- Put the root node on a queue Q
- Repeat:
  - if Q is empty, return failure
  - remove the node N *with the lowest* $f(\cdot) = g(\cdot) + h(\cdot)$ *value* from Q
  - if N is the goal, return success
  - add children of N to Q

OPEN /CLOSED List Implementation

This implementation avoids repeated states:

- Put the root node on OPEN
- Repeat:
  - if OPEN is empty, fail
  - remove the node N *with the lowest* $f(\cdot) = g(\cdot) + h(\cdot)$ *value* from OPEN
  - put N on CLOSED
  - if N is a goal, return success
  - expand N and compute $f(\cdot)$ for its successors
  - for successors not already on OPEN or CLOSED , add to OPEN
  - for those already on OPEN or CLOSED , if the new $f(\cdot)$ is smaller than that they currently have, use this instead; if any items on CLOSED are updated, put them back on OPEN

Properties of A*

- A* is complete
- If (and only if) $h(n)$ is consistent:
  - A* is optimal
  - A* is *optimally efficient*: it is guaranteed to expand fewer nodes than any other search algorithm, given that heuristic
- Time/space complexity: generally still $O(b^d)$

Show A* Example "Animation"

Proof of Optimality of A*

Theorem 1. *Given a graph in which*

- *each node has a finite number of successors; and*
- *arcs in the graph have a cost greater than some positive* ϵ

and a heuristic function $h(n)$ *that is admissible, A* is optimal.*

Proof. We first introduce the following lemma:

Lemma 2. *At every step of the A* algorithm, there is always a node* n *on* OPEN *with the following properties:*

- n *is on an optimal path to the goal*
- *A* has found an optimal path to* n
- $f(n) \leq f^*$, *where* f^* *is the optimal cost to the goal*

Proof. We prove this by induction, making use of the admissibility of $h(n)$:

- Base case: at the beginning, S is on the optimal path and is on OPEN and A* has found this path. Also, because $h(n)$ is admissible, $h(S) \leq c^*(S, G)$, so $f(S) \leq f^*$.
- Inductive step:
  - if n is not expanded, the conditions still hold
  - if n is expanded, then
    - all its successors will be placed on OPEN and (at least) one will be on the optimal path

- * we have found the optimal path to this node, because otherwise, there would be a better path to the goal, contradicting the assumption that the optimal path goes through n
  - * $f(n) \le f^*$ because:
    - · $f(n) = g(n) + h(n)$
    - · because of our optimality assumption, $g(n) = g^*(n)$
    - · because of admissibility, $h(n) \le c^*(n, G)$
    - · so, $f(n) \le g^*(n) + c^*(n, G) = f^*(n) = f^*$

□

Continuing: since it explores the graph in a breadth-first manner, and since each arc cost $> \epsilon$, A* must terminate (because all nodes on OPEN must eventually exceed f^*). A* terminates on an optimal path, because:

- if we reached a suboptimal goal g', then $f(g') < f(n)$
- but from the lemma, $f(n) \le f^*$
- if g' is a suboptimal goal, $f(g') > f^*$
- immediately, we have a contradiction: $f(g') < f^*$ and $f(g') > f^*$

So, A* is optimal. □

More About Heuristics

- Example heuristics: 8-puzzle example
  - Number of tiles out of place
  - Number of swaps needed
  - Manhattan distance
- Generating heuristics from *relaxed versions* of the problem. E.g. in the 8-puzzle, where the "real" problem states that a tile can move from A to B iff $Adjacent(A, B) \cap Blank(B)$, might relax as follows:
  - Can *always* move from A to B (i.e. number of tiles out of place heuristic)
  - Can move from A to B iff $Adjacent(A, B)$ (i.e. manhattan distance)
  - Can move from A to B iff $Blank(B)$ (i.e. number of swaps)
- For two admissible heuristics h_1 and h_2, h_2 **dominates** h_1 if $h_2(n) \ge h_1(n)$ for all nodes n. A* with h_1 will expand *at least as many* nodes as h_2

- Consider this: you could have a heuristic that calculated the right answer by doing a search! But, even if the number of nodes in the "real" search decreases, the computation time doesn't. It is important to maintain a balance between the accuracy of a heuristic and its computational cost.
- Other ideas for creating heuristic functions:
  - Statistical heuristics: collect statistics and use them; gives up admissiblity but is still likely to succeed
  - Learn weightings for hand-picked features
  - For a group of admissible heuristics where no one dominates any other, take the maximum!

Memory Bounded A* Searches

IDA*: Iterative Deepening A*

Like iterative deepening, except use $f(\cdot)$ as cutoff. Use slide.

SMA*: Simplified Memory-bounded A*

Use slide.

- Uses as much memory as available
- Avoids repeated states as far as memory allows
- Complete and optimal if memory is sufficient to store the shallowest solution path
- Optimally efficient if memory is sufficient to store the entire tree