More $A^\star$ and Beyond

CS156

Chris Pollett

Sep 17, 2014

Outline

- More $A^\star$
- Beyond Classical Search
- Local Search
- Hill Climbing

Back to A Search

- So much for our Python tutorial... Let's get back to AI.
- Recall in Best First Search we have an evaluation function f from nodes to some ordered set such as the reals, and the node we choose to expand next is always the one on the frontier of least f value.
- For Greedy Best First Search, f was chosen to be some heuristic function $h(n)$ that estimated the cost to a solution
- For Uniform Cost Search, f is chosen to be $g(n) =$ the cost to reach node n.
- For the $A^\star$ algorithm, we chose $f(n) = g(n) + h(n)$. i.e., $f(n)$ is estimated cost of the **total solution**.
- It turns out $A^\star$ search is complete. i.e., given enough resources it will find a solution.
- It is also in some sense **optimal** among best first search algorithms provided h satisfies some constraints.
- Let's look at why the latter fact is true...

Admissible Heuristics; Consistency

- Call an heuristic function h **admissible**, if it never overestimates the cost to a solution.
- For example, if the problem was to get from point A to point B in a city, then the straight-line distance between the two points would be an admissible heuristic.
- If h is admissible then $f(n)$ never overestimates the total cost to a solution.
- A goal is **optimal** if one cannot find a cheaper solution.
- If the nodes one could expand form a tree, then $A^\star$-star will return optimal results provided h is admissible.
- If the nodes one could expand form a directed acyclic graph, then a stronger condition is needed: A heuristic is **consistent** if for every n and every successor n' of n generated by any action a, the estimated cost of reaching n satisfies:
  $h(n) \leq c(n, a, n') + h(n')$.
- Consistency implies admissibility.
- Straight-line distance will satisfy consistency by the triangle inequality.

Proof of optimality

We argue the case where the nodes might form a DAG, the admissible case is similar.

Lemma. Suppose $h(n)$ is a consistent heuristic, then the values of $f(n)$ along any path are nondecreasing.

Proof. Suppose n' is a successor of n, then $g(n') = g(n) + c(n, a, n')$ for some action a and we have:
$f(n') = g(n') + h(n') = g(n) + c(n, a, n') + h(n') \quad (\text{by consistency}) \geq g(n) + h(n$
QED.

Lemma. Whenever $A^\star$ selects a node n for expansion, the optimal path to that node has been found.

Proof. If this were not the case, there would have to be another frontier node n' on the optimal path from the start node to n. Here n' is on the frontier, as the frontier nodes of the graph always separate the unexplored region of the graph from the explored region, and if it was in the explored region we would have selected n' already on the path to n to get a lower solution. Since f is nondecreasing along any path, n' would have lower f-cost that n and would have been selected before n. **QED**.

It follows from these two lemmas that the sequence of nodes expanded by $A^\star$ is in non-decreasing order of f(n). Hence, the first goal node selected for expansion must be optimal because f is the true cost for goal nodes and all later goal nodes will be at least as expensive. (**QED optimality proof**).

Memory bounded heuristic search

- The problem with A^star as presented is that it needs to keep track of all fringe and closed nodes.
- Thus, it tends to run out of space before time.
- We now look at a couple of ways to solve this problem...

IDA^star (Iterative Deepening A^star)

- In IDA^star (Korf 1985) we fix a constant \mu and we modify the expand-node function so that it only adds nodes to the fringe that are of cost less than the current threshold value.
- IDA^star is then:
  Do A^star search for thresholds < \mu, < 2\mu, < 3\mu, ... until you find a solution.
- In general, this will be slower than A^\star but will use less memory.

Recursive Best-First Search (RBFS)

- One might ask can we restrict the memory even further? IDA^\star is a little like the iterative lengthening variant of uniform cost search. So one might ask is there something like an A^\star variant of iterative deepening search (i.e., that would work well when have unit costs).
- Surprise! There is: Recursive Best-First Search (RBFS).
- This algorithm is similar to recursive depth-first-search. Here f_{mbox(limit)} is used to keep track of the best alternative path available from any ancestor of the current node. I.e, this allows us to find the next path to consider if the one we're on doesn't work:

```
function RBF-SEARCH(problem) returns a solution, or failure
    return RBFS(problem, MAKE-NODE(problem.INITIAL_STATE), infty)


function RBFS(problem, node, f_limit)
    returns a solution, or failure and a new f-cost limit

    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    successors := []
    for each action in problem.ACTIONS(node.STATE) do
        add CHILD-NODE(problem, node, action) into successors
    if successors is empty then return failure, infty
    for each s in successors do
        /* update f with value from previous search, if any */
        s.f = max(s.g + s.h, node.f)
    loop do
        best := the lowest f-value node in successors
        if best.f > f_limit then return failure, best.f
        alternative := the second-lowest f-value among successors
        result, best.f := RBFS(problem, best, min(f_limit, alternative))
        if result != failure then return result
```

Simplified memory bounded A* (SMA*) (Russell 1992)

- Do A^star until we run out of memory.
- When we don't have enough memory to add a new node to the fringe, discard from closed or fringe node of worst cost.

Choosing Heuristics

- One way to figure out if a heuristic is good or not, is by looking at its **effective branching factor**, b^star, of the search.
- Let N be the number of nodes generated by A^star for a particular problem and let d denote the solution depth.
- b^star is then solution to the equation:
  N + 1 = 1 + b^star + (b^star)^2 +...+(b^star)^d =\frac((b^star)^(d+1) - 1)(b^star - 1)
- For example, if you find a solution of depth 5 with 52 nodes then after solving for b^star in the above equation, b^star = 1.92.
- The book considers two heuristics for the 8-puzzle:
  - *h_1 := the number of misplaced tiles*
  - *h_2 := the Manhattan distance*
- They did 1200 random configurations of the 8-puzzle.
- They got b^star = 1.5 for h_1, and b^star = 1.3 for h_2. Thus, h_2 is a better heuristic as far as effective branching factor. As it doesn't cost that much more to compute at each node, the lower branching factor will mean one tends to save on space and time.
- We can show that h_1 leq h_2 for all possible boards. So as both are admissible, h_2 will theoretically and empirically give better results.

Generating Heuristics and Picking the Best

- Notice one can the weaken rules of the 8-puzzle, so that h_1 (or h_2) becomes the exact cost of a solution to the appropriately weakened problem.
- Hence, we often say h_1 and h_2 are solutions to a **relaxed version** of the problem.
- ... And we observe exact relaxed solutions for a given problem yield admissible heuristics for original problem.
- So we can come up with heuristics by looking at different problem relaxations.
- It might be the case that h_i sometimes performs better than h_j and sometimes h_j sometimes performs better than h_i.
- Worse yet, suppose you have a list of admissible heuristics h_1,..., h_m. Each performs better than all others in some circumstance.
- What heuristic do we choose?
- Since they were each admissible, we have:
  h(n) = max{ h_1(n), ..., h_m(n) }
  is also an admissible heuristic and performs at the best level among these heuristics.
- Admissible heuristics can also be generated by solutions to sub-problems.
- For example, for the 8-puzzle, one sub-problem is to just get the tiles 1,2,3,4 into the correct position.
- We could make a heuristic from this by estimating the cost of the 8-puzzle to be the cost of the optimal solution to the 1,2,3,4 problem.
- We can collect such solutions to sub-problems into a **disjoint pattern databases**, and use them to get a heuristic cost of a solution.

Beyond Classical Search

- So far this semester we have been looking at search strategies for problem solving agents for a single category of problems.
- We have been looking at problems where the environment is observable, deterministic, and known and where the solution is a sequence of action.
- We now try to relax these assumptions in different ways...

Local Search

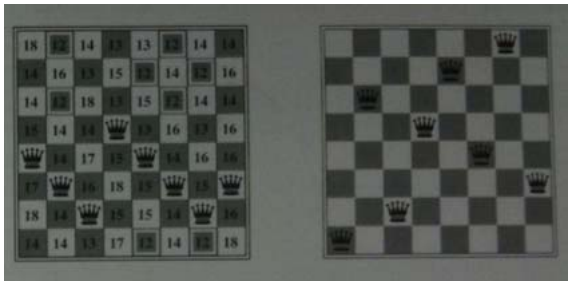
- So far as part of the solution to a problem, we have been interested in the sequence of actions which take us to the goal state.
- For some problems, this sequence of actions is not needed. For example, a solution to the 8-queens problem is eight non-attacking queens on a chess board -- we don't care so much how they were found.
- This property holds of many problems. For example, IC Design, factory-floor layout, job scheduling, network optimization, vehicle routing, portfolio management, etc.
- A **local search** algorithm operates using a single **current node** and generally moves to neighbors of that node. We don't keep track of the path history.
- In addition to just finding goals, local search algorithms are often used in **optimization problems**: problems in which the aim is to find the best state with respect to a **objective function**.
- The search process can be viewed in terms of the **state space landscape**. The landscape has both a "location" (which state we are currently at) and an "elevation" (what the objective function is for this state -- some real number).
- Plotting this scalar-valued function c:S -> RR we can find **local maxima** which would be states so that no neighboring state has as large/small a value of c, we might find **global maxima/minima** which have larger/smaller values than any other values corresponding to states in the state space. There might be other phenomena, like shoulders, saddles, "flat" local maxima, etc.
- We say a local search algorithm is **complete** if it always finds a goal if one exists. We say a local search algorithm is **optimal**, if it can always finds a global minimum/maximum.

Hill Climbing

```
function HILL_CLIMBING(problem) returns a state that is a local maximum
    current := MAKE_NODE(problem, INITIAL-STATE)
    loop do
        neighbor := a highest-valued successor of current
        if(neighbor.value ≤ current.value) return current.state
        current := neighbor
```

- The above is the so-called **hill-climbing** algorithm. It follows the strategy of looking at ones neighbors and always picking the neighbor which moves one in the most upward direction.
- If no such neighbors exist, we return a solution. This will be a local maxima, but not necessarily a global maxima.
- It has no memory of previous moves.
- For example, we could apply this algorithm to the eight queens problem: A state would consist of eight queens on the board, one per column.
- The successor state consist of all states which can be derived by moving one queen to a different square.
- The heuristic cost function h is the number of pairs of queen that are attacking each other.

Hill Climbing



- The left board above has h=17, the right h=1.
- Hill-climbing is sometimes called **greedy-local search**.
- Hill-climbing will usually choose randomly between two sucessors if they have the same cost.
- From a bad solution, hill-climbing often makes quite rapid progress to reach a close to optimal solution. For example, the second board on the right above is reached in five steps from the initial board on the left.
- Unfortunately, it can get stuck on local maxima (the right board above is one). You can also have ridges and plateaux.