

CMPSCI-683 Homework Assignment #1: Search

Patrick Pegus II

February 9, 2016

Problem 1. Iterative lengthening search is an iterative analogue of uniform-cost search. The basic idea is to use increasing limits on path cost. If a node is generated whose path cost exceeds the current limit, it is immediately discarded. For each new iteration, the limit is set to the lowest path cost of any node discarded in the previous iteration.

1. Show that this algorithm is optimal for general path costs.

Let modified uniform-cost path search (MUCS) be uniform-cost search (UCS) altered so that it does the following.

- (a) Discards any generated nodes exceeding the given limit.
- (b) Upon failure it returns the lowest path cost of any discarded node or failure if no node was discarded.

The function signature of MUCS is:

```
function MUCS(problem, limit)return (solution, failure, cost)
end function
```

Lemma 1. *MUCS is optimal.*

Proof. Let p be the path returned by MUCS from start state S to goal state G . Let C^* be the optimal path cost from S to G . Now suppose that $\text{cost}(p) > C^*$. Since, like UCS, MUCS expands nodes in order of their optimal path cost, there is only one way this could have happened— MUCS must have discarded a generated node representing path p' that is an optimal path or a subpath of an optimal path $p' + p''$. But any node discarded by MUCS must have a path cost greater than the limit l . Since step costs are positive, $\text{cost}(p' + p'') > \text{cost}(p') > l \geq \text{cost}(p) > C^*$. Therefore, the discarded node could not be an optimal path or a subpath of an optimal path. \square

Now define iterative lengthening search (ILS) as follows:

```
function ILS(problem)return (solution, failure)
   $s \leftarrow \text{NULL}$ 
   $f \leftarrow \text{NULL}$ 
   $c \leftarrow 0$ 
  while  $s$  is NULL and  $f$  is NULL do
```

```

     $s, f, c \leftarrow \text{MUCS}(\text{problem}, c)$ 
  end while return  $s, f$ 
end function

```

Theorem 1. *ILS is optimal.*

Proof. Since ILS only returns solutions returned by MUCS and MUCS is optimal, ILS is optimal. \square

2. Consider a uniform tree with branching factor b , solution depth d , and unit step costs (each action costs one unit). How many iterations will iterative lengthening require?

ILS will expand the start state in the first iteration, all states at depth 1 the second iteration, and all states at depth d at iteration $d + 1$.

3. Now consider the case where each step cost is a real number from the interval $[e, 1]$ for some $0 < e < 1$. How many iterations are required in the worst case? Try to derive the best estimate you can.

A worst case example is each step cost along the path to the solution costs 1 and another infinitely long path exists having uniform step cost of e . In that case ILS will expand the start state the first iteration, all states reachable by a path of cost e the second iteration, $2e$ the third iteration, and the solution having path cost d at iteration $1 + \lceil \frac{d}{e} \rceil$.

Problem 2. Suppose you were doing an A^* search and there were a number of different ways that you could compute an admissible heuristic value. Suppose that some were very cheap to compute but also very inaccurate estimators while others were very expensive to compute but were very accurate estimators. How would you determine which computation to use for a given application? Be as precise as possible. Describe intuitively a case in which a fast inaccurate heuristic would be beneficial and a case in which a more time-consuming accurate heuristic would be beneficial.

Let i be the fast inaccurate heuristic and a be the expensive accurate heuristic. We could say that the average time and space complexity of A^* is $(1 + h) b^{1 + \lceil \frac{C^*}{e} \rceil}$ where h is the average time to compute the heuristic of a given state, b is the effective branching factor achieved with the heuristic, C^* is the optimal solution cost, and e is the average step cost. Therefore, if C^* is much larger than e , b should be reduced as much as possible by choosing a . Alternatively, if C^* known to be not much greater than e , then i may be acceptable. However, if a can be cached and many searches must be computed within the domain, then the cost of computing the a cache can be amortized over all searches making the average time complexity near $b^{1 + \lceil \frac{C^*}{e} \rceil}$.

Special properties of step cost function can also influence the heuristic choice. Suppose paths leading to an optimal solution have decreasing step costs and others do not. Then the search space would be self-pruning for A^* even if the given heuristic was 0 for all states. Therefore, i would be a good choice, although a greedy search might be a better choice. Conversely, a would be required in the reverse situation.

Problem 3. Give an example of a search space in which A* would expand a node with a g-value that is greater than the cost of the optimal path from the initial state to that state. To do that, draw a small graph, designate one node as an initial state and one node as the goal state. Assign path costs to edges and an admissible heuristic estimate to each node.

1. Show the graph and trace the execution of A* until it terminates.

In Figure 1, node representing path (I, C) with cost 5 is expanded before the node representing path (I, A, C) with cost 4. The execution trace is shown in Table 1.

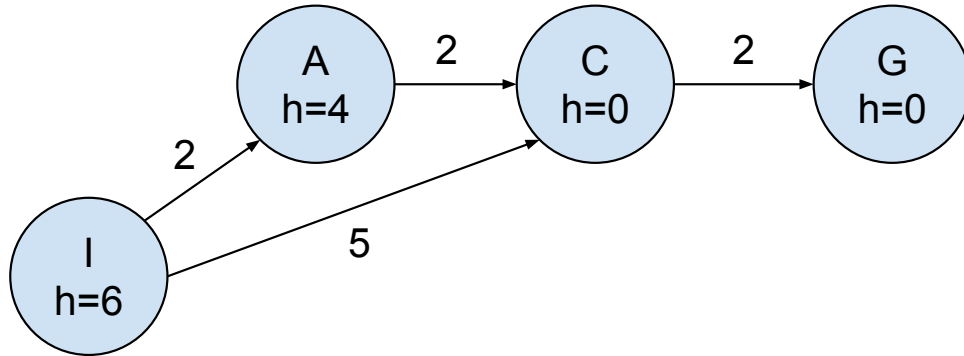


Figure 1: I and G are the initial and goal states, respectively.

Expansion	Open List
1	$[(I; 0 + 6)]$
2	$[(I, C; 5 + 0), (I, A; 2 + 4)]$
3	$[(I, A; 2 + 4), (I, C, G; 7 + 0)]$
4	$[(I, A, C; 4 + 0), (I, C, G; 7 + 0)]$
5	$[(I, A, C, G; 6 + 0), (I, C, G; 7 + 0)]$

Table 1: Execution trace of A* on the graph in Figure 1. Nodes are given symbolically as $(path; g + h)$.

2. Is your heuristic function consistent? Is it generally possible to create such an example when the heuristic is consistent? Explain your answer.

My heuristic is inconsistent. $h(A) = 4 \not\leq \text{cost}(A, C) + h(C) = 2$.

In general it is impossible to create an example with a consistent heuristic. Let i , n , and n' be nodes. Let a_1 , a_2 , a_3 be actions. Let $\text{cost}(i, a_1, n) + \text{cost}(n, a_3, n') < \text{cost}(i, a_2, n')$, but $\text{cost}(i, a_1, n) + h(n) > \text{cost}(i, a_2, n') + h(n')$, which creates the situation described. Since $\text{cost}(i, a_1, n) + \text{cost}(n, a_3, n') < \text{cost}(i, a_2, n')$ and all costs are non-negative, $\text{cost}(i, a_1, n) < \text{cost}(i, a_2, n')$. Then for $\text{cost}(i, a_1, n) + h(n) > \text{cost}(i, a_2, n') + h(n')$, $h(n) > h(n')$. This is not possible by the definition of consistency $h(n) \leq \text{cost}(n, a_3, n') + h(n')$, when costs are non-negative.

3. In general, will A* still find the optimal solution when this happens (i.e., a node is expanded with a g-value greater than the shortest distance from the initial state)? State clearly the assumptions you make about handling duplicate states.

A* will still find the optimal solution if no duplicate checking is implemented, or if a node representing a path from the start state I to some other state S with cost C is not placed on the open list only if there exists some other node from I to S with cost $C' \leq C$ that was already placed on the open list.

Problem 4. In this exercise, we consider the choice between two admissible heuristic functions, one of which is always at least as accurate as the other. More precisely, for a fixed goal, let h_1 and h_2 be two admissible heuristic functions such that $h_1(n) \leq h_2(n)$ for every node n in the search tree. Now recall that the A* algorithm expands nodes according to their f values, but the order in which nodes of equal f values are expanded can vary, depending on the implementation. For a given heuristic function h , we say that an ordering of the nodes in the search tree is **h -legal** if expanding the nodes in that order is consistent with A* search using h . Let N denote the set of nodes expanded by an A* search using h_1 (the “less accurate” heuristic function). Prove that there is always some search ordering that is h_2 -legal that expands only a (not necessarily strict) subset of the nodes in N . That is, prove that it is always possible that h_2 leads to a smaller search space than h_1 . (Note that the question asks you to find some search ordering such that certain properties hold. Hint: It is not possible, in general, to show that these properties hold for an arbitrary search ordering.)

Let $r(n)$ be the remaining cost from node n to the optimal solution. Let an implementation of A*, a , break f ties by imposing a descending ordering of the function $o(n) = r(n) - h(n)$. Therefore, in the event of f ties, a will expand the greatest underestimates of the true remaining cost first. Now consider nodes n_1, \dots, n_m where $f(n_i) = f(n_j)$, $o(n_1) > \dots > o(n_m)$, and n_m is on the solution path. a using h_1 must expand n_1, \dots, n_{m-1} , but a using h_2 may not expand them since $f_{h_1}(n_i) < f_{h_2}(n_i)$ —meaning they are no longer tied with the node on the solution path. This is certainly true in the case where n_m is a solution.

Problem 5. A knight moves on a chessboard two squares up, down, left, or right followed by one square in one of the two directions perpendicular to the first part of the move (i.e., the move is L-shaped). Suppose the knight is on an unbounded board at square (0,0) and we wish to move it to square (x,y) in the smallest number of moves. (For example, to move from (0,0) to (1,1) requires two moves. The knight can move to board locations with negative coordinates.)

1. Explain how to decide whether the required number of moves is even or odd without constructing a solution.

If $x \bmod 2 \neq y \bmod 2$, then the number of moves is odd. Otherwise, it is even.

2. Design an admissible heuristic function for estimating the minimum number of moves required; it should be as accurate as you can make it. Prove rigorously that your heuristic is admissible.

The heuristic returns a lookup in a pattern database if it exists, otherwise a calculated underestimate. The pattern database contains the exact path cost to move from $(0, 0)$ to (x, y) where x and y are integers less than or equal to 3. These path costs were found by uniform cost search (UCS). Therefore, they do not over estimate the path cost, since UCS is optimal. Let (x_g, y_g) and (x, y) be the coordinates of the goal and the current position, respectively. A lookup key for the pattern database is $(|x_g - x|, |y_g - y|)$. The differences shift the problem to the origin. The absolute values translate the position to the first quadrant, which does not change path costs, since the problem is symmetric.

The calculated heuristic is achieved by relaxing the problem and finding the admissible solution to that relaxed problem, making it an admissible heuristic to the original one. Instead of restricting the knight's movement to "two squares up, down, left, or right followed by one square in one of the two directions perpendicular to the first part of the move", it is allowed to move three times. Each time the knight may move up, down, left, or right one square. Therefore, the optimal path cost is at least $\lfloor \frac{m}{3} \rfloor$, where m is the Manhattan distance.

3. Implement A* and use it to solve the problem using your heuristic. Create two scatter plots showing (a) the number of nodes expanded as a function of solution length, and (b) computation time as a function of solution length for a set of randomly generated problem instances.

Figure 2 shows both nodes expanded and computation time as a function of solution length. As expected, nodes expanded is highly correlated with computation time.

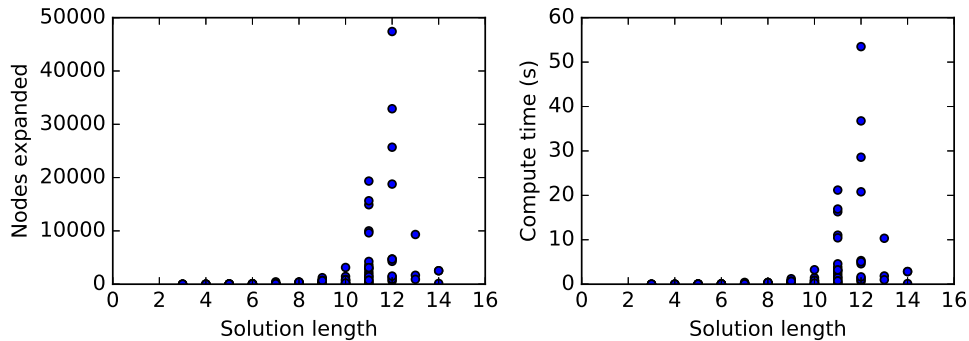


Figure 2: Nodes expanded and compute time as a function of solution length during 100 A* searches of randomly chosen goals in the 40-unit square.

Problem 6. The Traveling Salesman Problem (TSP) is one of the most intensively studied computational problems. You can read about the history and applications of TSP algorithms at <http://www.tsp.gatech.edu/>. A TSP can be solved using heuristic search. One possible heuristic is the minimum spanning tree (MST), which can be used to estimate the cost of completing the tour, given that a partial tour has already been constructed. The MST cost of a set of cities is the smallest sum of the link costs of any tree that connects all the cities.

1. Formulate the TSP as a search problem and show that the MST heuristic can be derived from a relaxed version of the TSP.

States Given a particular starting city, a state is any partial or complete tour beginning with that city.

Initial state The salesperson begins the tour in a randomly selected initial city from the set of cities to be visited.

Actions The actions consist of choosing which city to visit next from the set of unvisited cities, if that set's size is greater than two. Otherwise, choose the last unvisited city and complete the tour by continuing to the first visited city.

Successors The successors of a partial tour are all tours reachable by visiting one more unvisited city.

Path The path is the sequence of visited cities.

Step cost The distance between the last city in parent partial tour and the last city in its successor tour.

Path cost The cost of a path is the sum of all distances between cities on the tour.

Goal test The goal state consists of a path containing all cities with no repeated cities except the first and last city are the same.

2. Write a problem generator for instances of TSP where cities are represented by random points in the unit square (i.e., each coordinate is between 0 and 1). The number of cities to be generated should be a parameter.

Please see the `generateProblem` function in `p6.py`.

3. Find an efficient algorithm in the literature for constructing the MST, and use it with A* search to solve instances of the TSP (The two common algorithms are Prim's algorithm and Kruskal's algorithm); measure and plot the number of nodes expanded and computation time as a function of the number of cities for randomly generated problem instances. (Use problem instances that your algorithm can solve in less than a few minutes.)

The portion of Prim's algorithm necessary to return the weight of the MST is implemented in function `primMstWeight` in `p6.py`. Nodes expanded and computation time as a function of the number of cities is given in Figure 3.

6. Implement and test a hill-climbing method to solve TSPs. Compare the results with the optimal solutions obtained using A*.

Please see the `localSearch` function in `p6.py`. The implementation creates successors of a complete tour by swapping any pair of cities that do not begin or end the tour. Instead of choosing the best successor of all possible swaps, it chooses the first successor that results in a shorter tour. Figure 4 shows a comparison between A* and the local search over 50 problems each containing up to 12 cities. Accuracy is calculated as the length of the tour found by A* divided by the one found by the local search. Speedup is the A* compute time divided by the local search compute time. The accuracy of the

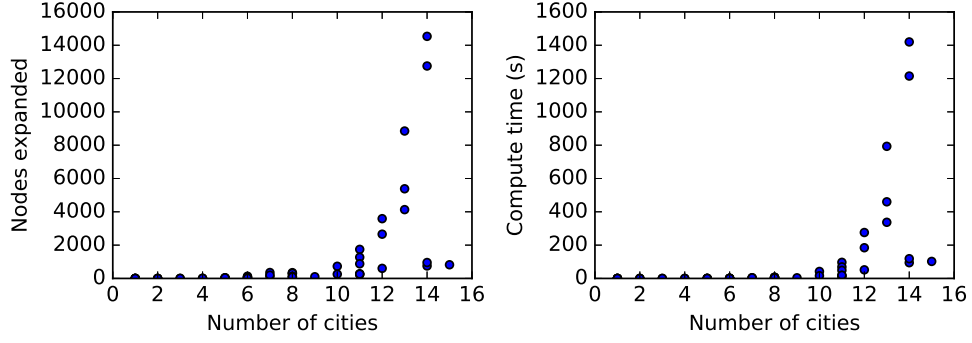


Figure 3: Nodes expanded and compute time as a function of the number of cities to be toured during 50 A* searches. The number of cities is a uniform random number between 1 and 15.

hill-climbing search decreased with problem size, but different methods may mitigate this effect while retaining much of the speedup.

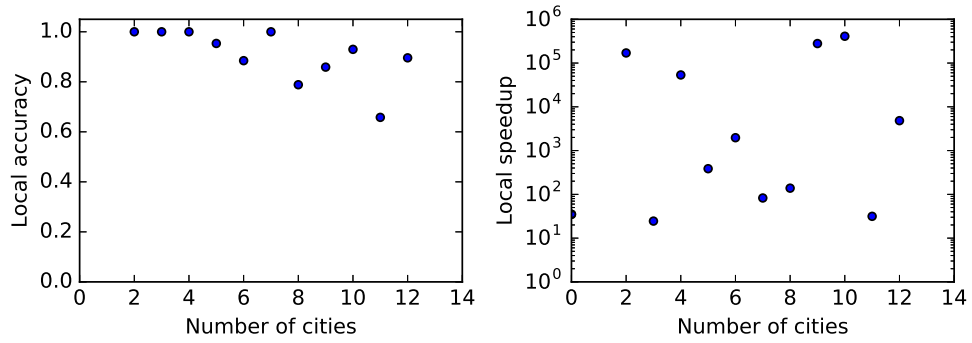


Figure 4: A comparison of A* and local search.