CSCE 4613 – Artificial Intelligence
Fall 2019
Assignment 1

Carson Molder
cbmolder@uark.edu
010755069

Tyler Tracy
tgtracy@uark.edu
010805685

1 Theoretical Questions

Background Questions

Search algorithm completeness Search algorithm *completeness* is a measurement of search algorithm performance that is defined as its capability of guaranteeing a solution if one exists. If there is at least one solution to the search problem or goal, a *complete* search algorithm is guaranteed to find at least one of the solutions.

Search algorithm optimality Search algorithm *optimality* is a measurement of search algorithm performance that is its capability of finding an optimal solution. Given a solution exists, an *optimal* search algorithm will always find, out of all possible solutions, the one that is the most optimal.

Depth-first search (DFS) advantages One advantage of DFS over BFS is that it takes less space. BFS has an exponential space complexity $O(b^s)$, where b is the branching factor of the graph and s is the depth of the shallowest solution. Meanwhile, DFS has a smaller space complexity of $O(bm)$, where b is the branching factor of the graph and m is the number of tiers/layers of the graph.

DFS's space complexity is better because it only needs to store one path (of maximum possible length m) at a time as it searches that path from the root to the lower nodes. Once it is done exploring a path, it goes back up and only needs to track the adjacent, unsearched nodes along the path it just explored (of maximum possible number b for each tier of the tree). This means, in the worst case, it needs to store b nodes times m edges, or $O(bm)$.

Meanwhile, BFS has to track every node across every layer of the tree it searches until it finds a solution. For the worst case, for the first layer this would be 1 node, for the second b nodes, for the third b^2 nodes, and so on until the s^{th} layer with b^s nodes. The lower polynomials cancel out in big-O notation, leaving an exponential space complexity of $O(b^s)$.

Breadth-first search (BFS) advantages One advantage of BFS over DFS is that, unlike DFS, it is a complete search algorithm. If a solution exists, it must be at a *finite* depth m somewhere inside the tree, regardless of if the tree is infinitely deep. Meanwhile, DFS may traverse an infinitely-long branch with no solution and never backtrack to another branch with a solution.

Another advantage of BFS over DFS is that it can be an optimal search algorithm. DFS will always find the leftmost solution on the tree first, regardless of the cost of doing so, since it explores branch-by-branch. Meanwhile, BFS will always find the shallowest solution first, since it explorers tier-by-tier. This means, if the costs of all edges are equal (or at least a non-decreasing function of depth), BFS is guaranteed to find the least-costly solution, whereas DFS is not guaranteed to do so because there could be a shallower solution on another branch besides the ones it has traversed before finding the first solution.

Because of BFS's optimality, BFS is better suited than DFS for problems involving path length optimization. It will generally find the optimal distance between two nodes. For example, BFS would be better than DFS for minimizing the distance between two cities on a map with cities as the nodes and highways as the edges on a search tree.

Uninformed Search

Choice of search In the graph from Question 2, where a is the start node and f is the goal node, DFS is preferred due to requiring less space. Both BFS and DFS arrive at the goal node f in six nodes. Since the number of nodes searched is the same, the algorithm that takes less space is preferred. This assumes that DFS tracks the nodes it has visited and does not visit them again if they reappear in the search.

Sequence of paths BFS explores the following sequence of paths: (a, b, e, c, d, f). DFS explores this sequence of paths: (a, b, c, d, e, f). Figure 1 shows the search tree for the graph from question 2.

A*-Tree Search

Figure 2 is an example of an acyclic directed graph and a heuristic function, where an A*-tree search does not find an optimal path. A is the starting node, and G is the goal node. The heuristic value for each node is in brackets.

A*-tree search did not find the optimal path for this graph because its heuristic is *not admissible*. One example that proves this is at node C. The heuristic estimates the distance

Figure 1: The search tree for the graph in Question 2.

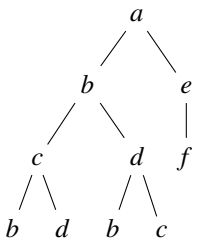
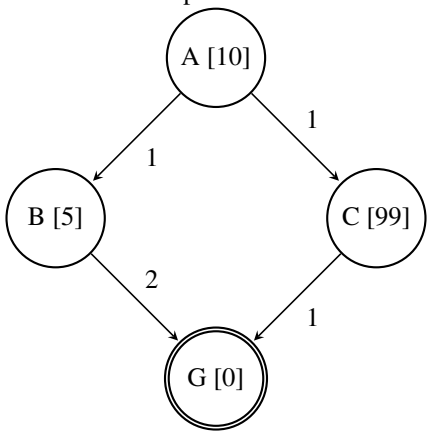


Figure 2: A directed graph and heuristic for which A*-tree search does not return an optimal solution.



Optimal path from A to G: $< G, C, A >$ with length 2
A*-tree path from A to G: $< G, B, A >$ with length 3

to G is 99, yet the actual path length from C to G is 1. This is a vast overestimation, and the A*-tree search algorithm pushes this path to the bottom of its priority queue due to the overestimating heuristic.

A*-Graph Search

Figure 3 is an example of an acyclic directed graph and an *admissible* heuristic function, where A*-graph search does not find an optimal path. A is the starting node, and G is the goal node. The heuristic value for each node is in brackets.

A*-graph search has a greater requirement for optimality than A*-tree search. A*-graph search not only requires its heuristic to be admissible but also *consistent*.

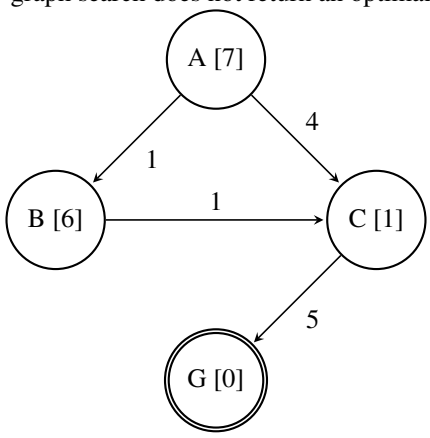
A consistent heuristic, for all node on a graph, always has an estimate that is at most the heuristic distance from any neighboring node to the goal plus the distance to that neighboring node. Mathematically, a consistent heuristic $h(n)$ has the following property for all nodes *n* with neighbors *p*:

$$h(n) \leq distance(n, p) + h(p).$$

Also, for a heuristic to be consistent, the heuristic estimate at the goal node must be zero. For Figure 3, this is the case.

However, looking at nodes B and C, the consistency of the heuristic used in Figure 3 falls apart. The heuristic value $h(B)$ equals 6, which is greater than the value $distance(B, C)+h(C)$, which equals $1+1 = 2$. Since these

Figure 3: A directed graph and *admissible* heuristic for which A*-graph search does not return an optimal solution.



Optimal path from A to G: $< A, B, C, G >$ with length 7
A*-graph path from A to G: $< A, C, G >$ with length 9

two nodes fail the definition, the entire heuristic is inconsistent, meaning an A*-graph search can no longer guarantee an optimal solution for the graph in Figure 3.

The key property of A*-graph search that makes it fail when A*-tree search would not is that A*-graph search remembers and refuses to revisit nodes it has already visited. For Figure 3, it would visit the path $< A, C >$ before visiting the paths $< A, B >$ and $< A, B, C >$. When the search arrives at the path $< A, B, C >$, it checks to see if it has already visited node C and it sees that it has. Therefore, it does not explore that path even though it is the only way it can find the optimal path.

A consistent heuristic avoids this pitfall because it ensures the A* search takes the most optimal path first, instead of taking a non-optimal first path and be forced to revisit a node again if it were to find the optimal path.

2 Search Algorithm Implementation

Programming

The program was written in the Python programming language. The entire source code is contained in the file $search.py$.

Traffic Time!

Instead of finding the path with the shortest distance, now the task is to find the path with the shortest travel time at a given moment based on current traffic. The assumption that drivers never drive faster than the speed limit gives a lower bound to travel time, which is helpful for developing a more effective heuristic.

An *admissible* heuristic never overestimates the cost to the goal node. In this scenario, the cost is travel time. Therefore, any admissible heuristic for this scenario would tend to underestimate the travel time to a destination.

The shortest distance between a starting location and destination is simply the Euclidean (straight-line) distance between these points. If every road has a speed limit, then

taking the maximum speed limit out of every road on the map/graph would give the greatest possible speed a driver could drive. Using these two bounds would always return a travel time that never overestimates the real value, because such a travel time requires a driver to drive the shortest distance possible at the greatest legal speed possible.

To formulate this admissible heuristic $h(n)$, let *n* be the node in question, *g* be the goal node for the search, $EuclideanDist(n)$ be the Euclidean distance between node/intersection *n* and node/intersection *g*, and SL_{max} be the maximum speed limit out of every road on the map. Since speed is distance over time, time can be represented as distance over speed. Therefore, one admissible heuristic $h(n)$ that never overestimates travel time is

$$h(n) = \frac{EuclideanDist(n)}{SL_{max}}.$$

Real-World Routing

A traveller may not leave their starting location at the exact time their path routing is generated, meaning that the "current" traffic data used to generate their route is outdated. Also, traffic can change while the traveller is en route to their destination, meaning their route chosen earlier may no longer have the shortest travel time.

The admissible heuristic used earlier is agnostic to the traffic conditions, meaning it does not need to be changed to account for real-time traffic data. However, the weights (travel times) between nodes/intersections would need to update. This could be done by letting the navigation program download new traffic data from the Internet at a certain time interval—for example, every 30 seconds—and use this data to reassign its graph's weights and recheck for the quickest route using a search algorithm like A*.