Names: Michael Radoian, Michael Le, Alexander Varshavsky
Intro to Artificial Intelligence
Professor McMahon
10/18/20

Assignment 1: Heuristic Search Report

3. Optimizations

Implement a customized priority queue to improve search performance. The default contains() method of the priority queue class in the Java SDK has a time complexity of O(n). By introducing a customized class (SearchPriorityQueue) with a HashSet, we can improve the time complexity of the contains() method from linear to constant time. The tradeoff would be redundancy in cell stored, but the space complexity would still be upper-bounded in linear time. However, our current implementation can be optimized further. If we are under the assumption that the current grid is unlikely to change or will remain static, then precomputation will improve the time complexity of our search algorithms drastically. In addition, our implementation can take advantage of caching to reduce redundant calculations i.e. storing heuristic values in our states.

Furthermore, our implementation can combine other optimal search algorithms with A* Search. For example, combining A* Search with Beam search to limit the number of values stored in the open-list by using a min and max heap to store f-values and reducing memory usage as a result. Another example would be to combine A* Search with the Jump Point Search algorithm that. This combination allows states to skip to other states on the grid instead of limiting state *s* to expanding only neighboring states. Looking at states farther away can reduce the number of calculations and node expansions needed to reach the path goal.

4. Heuristics

1) Diagonal Distance

```
public static double diagonalHeuristic(Cell s, Cell goal) {
    Coords sCoords = s.getCoordinates();
    Coords goalCoords = goal.getCoordinates();
    double dx = Math.abs(sCoords.getColumn() - goalCoords.getColumn());
    double dy = Math.abs(sCoords.getRow() - goalCoords.getRow());
    return .25 * (dx + dy) + ((.25 * Math.sqrt(2)) - 2 * .25) * Math.min(dx, dy);
}
```

Adjusted diagonal distance finds the shortest path using eight degrees of motion. Since this mimics the search's movement, this would be very useful if there is a chunk of fewer blocked cells in the course's way. In order for it to be admissible, we use the constants .25 and (.25 * sqrt(2)). .25 is the smallest horizontal distance and (.25*sqrt(2)) is the shortest diagonal distance. These values can also account for the difference in moving diagonally vs horizontally and vertically.

2) Adjusted Manhattan Distance (best)

```
public static double manhattanHeuristic(Cell s, Cell goal) {
    Coords sCoords = s.getCoordinates();
    Coords goalCoords = goal.getCoordinates();
    double dx = Math.abs(sCoords.getColumn() - goalCoords.getColumn());
    double dy = Math.abs(sCoords.getRow() - goalCoords.getRow());
    return .25 *(dx+dy);
}
```

Adjusted Manhattan distance finds the shortest path using four degrees of motion. Since this is the largest estimate that still underestimates the actual cost, it expands the least amount of nodes and is the fastest admissible heuristic. It also avoids the sqrt() function which can help save time over a large number of computations. Similarly, to the other heuristics we multiply it by .25 to simulate taking a highway, which is the shortest possible path on the grid.

3) Adjusted Euclidean Distance

```
public static double euclideanHeuristic(Cell s, Cell goal) {
    Coords sCoords = s.getCoordinates();
    Coords goalCoords = goal.getCoordinates();
    double dx = Math.abs(sCoords.getColumn() - goalCoords.getColumn());
    double dy = Math.abs(sCoords.getRow() - goalCoords.getRow());
    return .25*Math.sqrt(dx*dx+dy*dy);
}
```

Adjusted Euclidean distance is a straight-line path to the goal. Adjusted Euclidean would be helpful if our search was not a grid and had a full range of motion. Since this is the shortest of the admissible heuristics, it will expand the most nodes and will not be as efficient. We multiplied the Euclidean distance by 0.25 to simulate taking a straight line highway directly to the goal.

4) Euclidean Distance Squared

```
public static double euclideanSquaredHeuristic(Cell s, Cell goal) {
    Coords sCoords = s.getCoordinates();
    Coords goalCoords = goal.getCoordinates();
    double dx = Math.abs(sCoords.getColumn() - goalCoords.getColumn());
    double dy = Math.abs(sCoords.getRow() - goalCoords.getRow());
    return .25*(dx*dx+dy*dy);
}
```

Adjusted Euclidean distance squared removes the sqrt() function. It is an inadmissible heuristic and is extremely fast and greedy. It also saves time by avoiding the sqrt(), which can

add up after searching large amounts of nodes. The heuristic finishes searching for most paths in under 2 ms. However, the path it returns can be up to 50 units higher than the shortest route. Using Adjusted Euclidean distance

5) Chebyshev

```
public static double chebyshevHeuristic(Cell s, Cell goal) {
    Coords sCoords = s.getCoordinates();
    Coords goalCoords = goal.getCoordinates();
    double dx = Math.abs(sCoords.getColumn() - goalCoords.getColumn());
    double dy = Math.abs(sCoords.getRow() - goalCoords.getRow());
    return (.25 * (dx + dy) + ((.25 ) - 2 * .25) * Math.min(dx, dy));
}
```

Chebyshev is very similar to diagonal because it uses eight degrees of motion to find the goal's distance. However, unlike diagonal, the cost to travel diagonally vs. horizontally and vertically is the same. Chebyshev would be a useful search if there was no terrain on the map or a large chunk of our map had no obstacles. It is similar in speed to diagonal, but runs slightly slower because it does not consider diagonal travel weight.

5. Experimental Evaluation of Uniform-Cost, A* Search, and Weighted A* Search

Uniform-Cost & A* Search

	Average time	Average path length difference from optimal	Average # of Nodes	Memory Requirements
Uniform-Cost	115.06 ms	0	11870.92	22774.32 kb
A* diagonal	87.82 ms	0	8121.28	20946.94 kb
A* Adjusted Euclidean	95.23	0	8958.28	26728.22
A* Adjusted Euclidean Squared	1.26 ms	59.09	119.62	1023.84
A*Chebyshev	90.3 ms	0	8565.18	20028.4
A* Adjusted Manhattan	70.68 ms	0	7447.68	19210.9

Weighted A* with a weight 1.1

	Average time	Average path length f(time)	Average # of Nodes	Memory Requirements
Diagonal	87.28 ms	0	10162.78	47840.22 kb
Adjusted Euclidean	83.38 ms	0	10286.60	49077.42 kb
Adjusted Euclidean Squared	1.52 ms	53.51	128.80	1022.24 kb
Chebyshev	83.48 ms	0	10458.04	52067.28 kb
AdjustedManhattan	80.08 ms	0	9626.94	46029.20 kb

Weighted A* with a weight 2.4

	Average time	Average path length f(time)	Average # of Nodes	Memory Requirements
Diagonal	41.44 ms	1.21	3669.94	17650.32 kb
Adjusted Euclidean	48 ms	1.18	4008.24	18723.26 kb
Adjusted Euclidean Squared	1.4 ms	50.6	112.02	971.78 kb
Chebyshev	59.1 ms	.91	4787.08	26046.46 kb
Adjusted Manhattan	21.1 ms	2.18	2321.12	13013.60 kb

6. Relative Performance of Different Heuristics on Uniform-cost, A*, and Weighted A*

Uniform Cost Search is the slowest method of finding the shortest path and is the worst of the three algorithms in terms of speed. It has no heuristic, so it may go down paths that do not lead to the optimal route when it is searching. This method is not optimal when looking for the shortest path between a start and endpoint. However, if looking for the shortest path from one to all other points, this is the most optimal. For example, the average number of nodes expanded over the 50 benchmarks for Uniform Cost Search was 11870 while the same metric was at most 9000 nodes for A* Search and the 5 chosen heuristics.

A* search can vary in its results depending on the heuristic. However, when using an admissible heuristic, it is one of the fastest ways to get the shortest path. Out of all the heuristics, Adjusted Manhattan gives back the best results. Adjusted Manhattan explores the least amount of nodes and data. Euclidean Squared is an ultra greedy heuristic that almost never finds the correct path, but calculates its path extremely fast. Out of the five heuristics, the ranking of speed is as followed:

Euclidean Squared > Adjusted Manhattan > Adjusted Euclidean > Adjusted Diagonal > Chebyshev

For Weighted A* Search, we chose weights w1 = 1.1 and w2 = 2.4. W1 kept the A* searches to find the shortest path and marginally improved the algorithm's speed. W1 would be helpful if accuracy was important, but for this assignment time is valued more than accuracy. W2 improved the search speed by up to 50% while finding the shortest path within two units higher than the optimal route. This weight would be useful if the search prioritized speed over accuracy.

7. Experimental Evaluation of Sequential A* Search

Sequential A* Search

	Average time	Average cost difference from optimal	Average # of Nodes	Memory Requirements
n = 5 heuristics	362.06 ms	0	9770.94	27682.94 kb

A* Search

	Average time	Average cost difference from optimal	Average # of Nodes	Memory Requirements
Uniform-Cost	115.06 ms	0	11870.92	22774.32 kb
A* Diagonal	87.82 ms	0	8121.28	20946.94 kb

A* Adjusted Euclidean	95.23 ms	0	8958.28	26728.22 kb
A* Adjusted Euclidean squared	1.26 ms	59.09	119.62	1023.84 kb
A*Chebyshev	90.3 ms	0	8565.18	20028.4 kb
A* Adjusted Manhattan	70.68 ms	0	7447.68	19210.9 kb

8. Relative Performance of Sequential A* Search Implementation

In the second part of question 9, we proved that Sequential A* Search guarantees that the solution is bounded by the $w_1 * w_2$ sub-optimality factor. Since the algorithm terminates as soon as there is a solution from any one of the *n* heuristic searches, the solution will be within this bound. And since the heuristics are expanded in a round-robin fashion, the total number of cells (or states) expanded is the number of heuristics *n* times the minimum number of expansions across all heuristic searches. This would make sense because if any of the inadmissible searches gets to the solution faster than the admissible/anchor search, then Sequential A* Search is more efficient than regular A* Search. This captures the idea that inadmissible heuristics can still provide valuable information to the traditional A* search to improve performance.

However, in terms of relative performance, we were unable to match our implementation with the expected result from the above analysis. The average cost difference over 50 benchmarks for Sequential A* Search was 0 indicating the results were accurate, but the results for execution time, memory usage, and expanded nodes showed our implementation was flawed. The computation time and solution quality should have both improved. The total number of cells expanded for Sequential A* Search should be less than or equal to A* search, and therefore the time and space complexity would be better. In the above experimentation results, Sequential A* Search was 5x slower than admissible A* Search, expanded roughly 20% more nodes, and used 50% more memory.

9. Proof of Sequential A* Search Properties

1. Prove that for any state s that $\text{Key}(s, 0) \leq \text{Key}(u, 0)\ \forall u \in \text{OPEN}_0$, it holds that $\text{Key}(s, 0) \leq w1 * g^*(s_{goal})$. In other words, show that the anchor key of any state s when it is the minimum anchor key of an interaction in Sequential A* Search is bounded by w_1 times the cost of the optimal path to the goal ($g^*(s_{goal})$).
    a. Does such a state always exist in the queue? Yes, we will always find a state such that s_i is in OPEN_0 because s_0 is put into OPEN_0 when the algorithm is initialized. Afterwards, every time a state on the most optimal path is expanded and removed from OPEN_0, it is guaranteed that the next state in the most optimal path, a neighbor of the current state, will be added to OPEN_0. However, s_{goal} does not exist in the queue, but by then the algorithm would have already terminated because the path is found if s_{goal} is reached. The guaranteed existence of state s in the queue implies that the OPEN_0 queue will never be empty.
    b. First, we prove that the g_0-value for any state s_i is equal to w_1 times the cost of the optimal path to the start. The g_0-value for any state s_i is always less than or equal

to the g_0-value of the previous state in the optimal path to the goal plus the cost to travel. This would make sense because if s_i had a better g_0-value, then state s_i would not be on the optimal path. The conditional logic in ExpandState(s, *i*) confirms this by making sure the neighboring state always has a g_0-value equal to its parent plus the cost to travel. Additionally, the answer to the question "does such a state always exist in the queue" proves that the previous state has always been expanded. Since the g_0 value of the preceding state with the admissible heuristic is the same as $w_1 * g^*(s_{i-1})$, it can be substituted in. The cost to travel between s_{i-1} and s_i can be factored in as well, so now we can prove that g_0-value for any state s_i is equal to $w_1 * g^*(s_i)$, or the optimal distance to the start.

c. Now we take the formula for calculating the minKey_0 of any state s_i and plug in our equality from step b in place of the g_0-value for that particular state. Since the heuristic h_0 is admissible, it can also be substituted in with the optimal cost from traveling from state s_i to the goal s_{goal}. The term $g^*(s_i)$, or the optimal cost from the start to s_i, and the term $c^*(s_i, \text{goal})$, or the optimal cost from s_i to the goal, can be combined to create a new term $g^*(s_{goal})$ since s -> s_i -> s_{goal}.
   i. $\text{Key}(s_i, 0) \leq g_0(s_i) + w_1 * h_0(s_i)$
   ii. $\text{Key}(s_i, 0) \leq w_1(g_0(s_i) + c^*(s_i, s_{goal}))$
   iii. $\text{Key}(s_i, 0) \leq w_1(g^*(s_{goal}))$

d. Therefore, for any state s that $\text{Key}(s, 0) \leq \text{Key}(u, 0)\ \forall u \in \text{OPEN}_0$, it holds that $\text{Key}(s, 0) \leq w1 * g^*(s_{goal})$.

2. Prove that when the Sequential Heuristic A* terminates in the i^{th} search, that $g_i(s_{goal}) \leq w_1 * w_2 * c^*(s_{goal})$. In other words, prove that the solution cost obtained by the algorithm is bounded by a w1 * w2 sub-optimality factor.

a. The algorithm will terminate if $g_i(s_{goal}) \leq \text{OPEN}_i.\text{Minkey}()$, $\text{OPEN}_i.\text{Minkey}() \leq w_2 * \text{OPEN}_0.\text{Minkey}()$, and $g_i(s_{goal})$ is not infinity (in other words has been set). The algorithm will also terminate if $\text{OPEN}_i.\text{Minkey}() > w_2 * \text{OPEN}_0.\text{Minkey}()$, $g_0(s_{goal}) \leq \text{OPEN}_0.\text{Minkey}()$, and $g_0(s_{goal})$ is not infinity (in other words has been set) as shown in the pseudocode.

b. $\text{OPEN}_i.\text{Minkey}()$ can be substituted in with $w_2 * \text{OPEN}_0.\text{MinKey}()$ because the second termination statement indicates that $w_2 * \text{OPEN}_0.\text{MinKey}()$ is upper bounded by $\text{OPEN}_1.\text{MinKey}()$:
   i. $g_i(s_{goal}) \leq \text{OPEN}_i.\text{Minkey}()$
   ii. $g_i(s_{goal}) \leq w_2 * \text{OPEN}_0.\text{Minkey}()$

c. $\text{OPEN}_0.\text{MinKey}()$ can be substituted in $w_1 * g^*(s_{goal})$ as proven in our first proof.
   i. $g_i(s_{goal}) \leq w_2 * \text{OPEN}_0.\text{Minkey}()$
   ii. $g_i(s_{goal}) \leq w_2 * w_1 * c^*(s_{goal})$

d. Therefore, it is proven that Sequential Heuristic A* terminates in the i^{th} search s.t. $g_i(s_{goal}) \leq w_1 * w_2 * c^*(s_{goal})$