CS 440: Artificial Intelligence
Project 1
Heuristic Search using Information from Many Heuristics

Allen Tung
Julian Liu
February 5, 2017

Our visualization was designed in Visual Studio, using the Windows Presentation Foundation (WPF). The backend was written in C#, managing the algorithms and map generation. Specific application behavior is defined in the XAML codebehind files, and data is stored in a viewmodel, somewhat following the Model - View - ViewModel behavior as dictated by Microsoft. Observable data on the front end is represented as properties in the Viewmodel, and are bound together. When the viewmodel is updated, the observable data on the front end updates as well. The map is represented by rectangles, with different colors and borders to distinguish different kinds of cells. Rivers are denoted by an aqua border, black nodes are impassable, white nodes are freely traversable, and gray nodes are difficult to traverse. When the path is calculated, a series of buttons are generated along the path, which update the f, g, and h values of the UI when clicked.

There is only one window for the user interface, consisting of a side panel to display the relevant A* algorithm and values, and to give the user the options to select or generate maps, calculate new runtimes and a new path, and change the specifications of each algorithm. When all the algorithm options have been selected, the user may click the “Calculate Path” button to generate buttons along the path from start to end node which will update the ui’s f, g, and h values upon click. When a new map is selected, the world will be reloaded, and the map will refresh to reflect the new data.

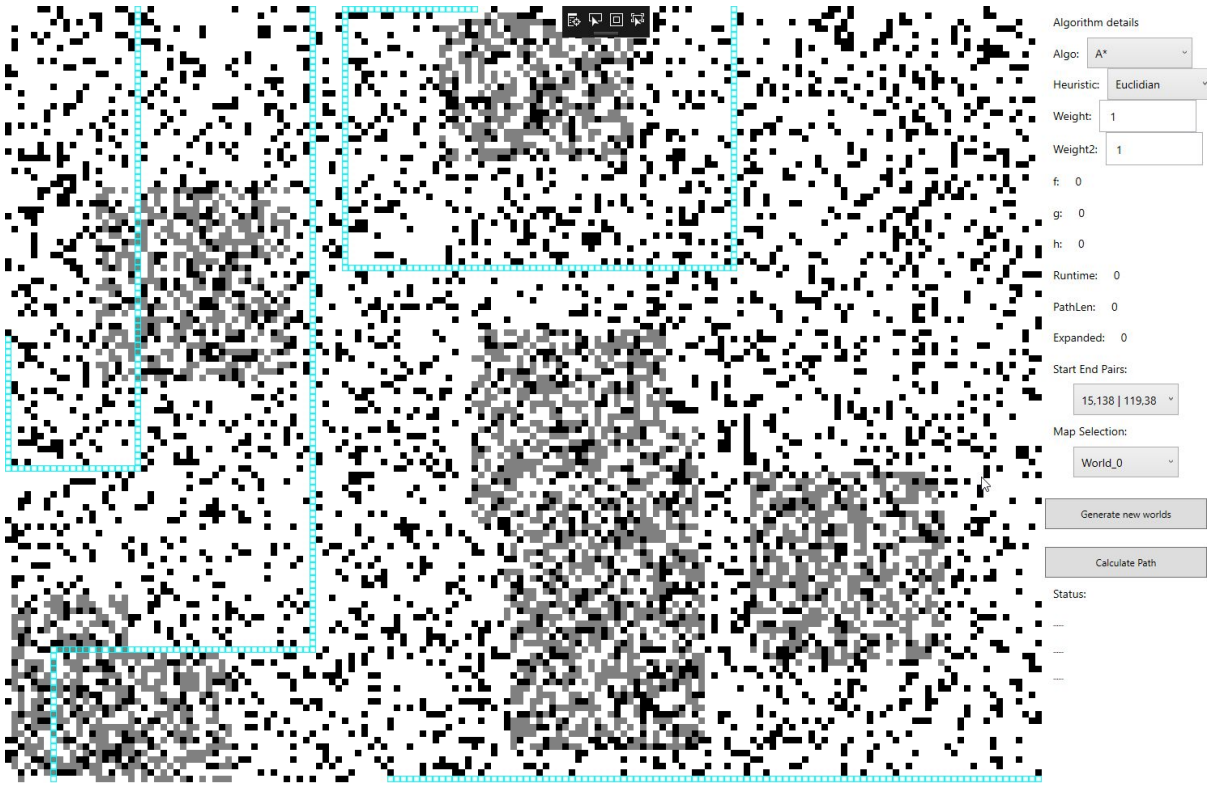

Figure 1. Basic UI and map example.

We choose to represent the map as a 2D array of integers to describe the properties of the cell. Since we are aware of the array size, dynamic allocation will not be an issue. Expanding nodes would involve memory accesses of array locations in a block around the desired cell,

which could be inefficient in terms of memory accesses, depending on the architecture, which could be a source of optimization in the future. We keep a priority queue for potential nodes to be expanded, and dequeue as necessary. Using a priority queue is superior to other datastructures such as arrays or linked lists, because the cost of dequeueing the minimum cost node is O(1). In an array or list, an O(n) or at least O(logn) search would need to be done to find the minimum cost node. Since we need to use this dequeue operation frequently, a priority queue is ideal.

To store the details pertinent to each node, we keep an array of 'worldnode' structures, which keep a reference to a parent as well as f, g, h values. These are only generated as the algorithm proceeds, and the array structure allows for constant time accesses to any desired node. Although expanding neighbors will be slightly inefficient in terms of cache accesses since the matrix is stored in row major form, the O(1) access time for any world node makes this an improvement over a linked list format, for example.

To compute the final path from goal to start, we follow the parent references from the end node, in an implicit linked list. As we follow this path, we calculate the path length as well; this is done for convenience on the front end, since the realtime cost of this operation is low.

When designing the heuristics, we propose first Euclidian distance, the straight line distance from the current point to the goal, as the most naiive heuristic. The shortest possible path will always be greater to or equal to the straight line distance, and thus, is generally a consistent heuristic. However, there is always a computationally intensive square root necessary to compute the distance, and in a grid world where only 8 directions of movement are possible, there must be a better choice than Euclidian distance. Further, we do not consider highways or impassable terrain or difficult to traverse terrain, all of which would affect the real cost of travel.

We attempt to use the Manhattan distance, the distance between any point and the goal while travelling only in right angled axes. This heuristic is also proven to be admissible on 4-way connective grids, however, we must consider that we have 8-way connectivity. Also, there are highways in the world, which means that we must consider this inadmissible without scaling.

If we use Manhattan distances while assuming every node traversed is a highway, we will obtain a consistent heuristic. We choose to use this as our anchor heuristic in all algorithms.

We propose next the diagonal distance, also known as the Octile distance, in which we consider the diagonal path as well in our heuristic. In this case, we assume the diagonal cost is the square root of 2, as discussed in the assignment itself. This seems to follow the actual graph costs most closely, and it seems to be the most optimal heuristic. The assumptions made here are that each tile is freely traversable. However, this is potentially not the case- there could even be a highway connecting a start goal node pair directly, in a straight line. In this case, if we are to make this admissible, we must scale the heuristic, and assume that we travel solely on highways. This way, there is no possible more efficient route to the goal.

Although this heuristic should prove to be admissible, we choose to use the manhattan distance over this one as our anchor heuristic, because it is simpler to calculate.

Another heuristic is similar to the Octile distance, called the Chebyshev distance, which considers all paths as equivalent in cost, whether they are diagonal or not. We do not expect this to perform as well as Octile distance, being that it is less close to the actual conditions of the map.

The major overhead in performance is when unnecessary nodes are generated and expanded. When the heuristic used performs well, the search is guided to generate as few nodes as possible to proceed from start to finish. Following this line of thought, the uninformed Uniform Cost Search performs the most poorly, expanding nodes without any intelligent guide. It stands to reason that using this heuristic causes the algorithm to perform magnitudes worse than any other search.

When performing an ordinary A* search, several optimizations are done to improve performance. First, we do not ever expand a node that is impassible, or even place it in the fringe. When encountered, we immediately place it into the closed list. There is often a tradeoff between memory and runtime, and we often choose to optimize runtime over space. For example, storing each worldnode in an array and only generating nodes as we proceed results in a rather sparse array, but we obtain a O(1) access time as a result.

A chart of average performance data is included at the end. Of the first three algorithms, Uniform cost performs the worst, in terms of nodes expanded. Its runtime is also much higher than A* and weighted A*. This is consistent with our assumptions, because uniform cost searches are not guided by any heuristic. They will expand uniformly around the start node, until they reach the end node. The resulting path length will not be optimal either. In the averages for each algorithm, the heuristics used are also different, with A* and Weighted A* averaging several inadmissable heuristics in their results. Sequential and Integrated A* innately use all heuristics to guide the search, resulting, in a more optimized path length. Sequential A* produces the most optimal path length, expanding the fewest nodes.

In the averages of the heuristics, the Manhattan heuristic is the fastest and expanded the fewest nodes. However, it produces the longest path, which limits its usefulness as a heuristic. The Manhattan/4 and Chebyshev/4 heuristics compete for the shortest path length, and although the Chebyshev/4 has a .4 shorter path length on average, the Manhattan/4 heuristic wins in the number of nodes expanded, and in runtime.
When we examine the averages for each heuristic in just the A* algorithm, we find the same results as in the overall average- except this time, the Manhattan/4 and Chebyshev/4 heuristics produce the same average path length for the benchmarks, indicating that they are both consistent heuristics. However, the Manhattan/4 heuristic wins again, having 1440 fewer expansions on average, and a 2.7 second faster runtime.

When we weight the algorithms with 1.25 and 2 scale weights, the algorithms all perform on average about 2x faster than ordinary A*, with only a 1% increase to the path length for the

Manhattan/4 and Chebyshev/4 heuristics. These results are very favorable, and indicate that it is worth giving up a little optimality in the path for a 100% decrease in runtime.

When using the Sequential and Incremental Heuristics, it is possible to obtain attributes of any of the heuristics used in the search. When tuning the w1 and w2 values, the results will be affected by the selected heuristic as an anchor, and the quality of each individual heuristic. As the w1 value increases, the algorithm will favor the heuristic more heavily, as though in a weighted A* search. This will reduce the quality of the algorithm in different amounts depending on the heuristic, up to a 30% reduction in quality for the euclidian heuristic for example. Since each search uses heuristics in a round robin fashion, increasing the w1 value will tend to drastically reduce the quality of the algorithm.

As the w2 value is increased, the algorithm will have more freedom to use heuristics other than the anchor heuristic. In several cases, this will cause the algorithm to follow strange, inoptimal paths to the goal, but the path will be computed faster. In the averages, weights were not tuned for the most optimal results, and yet the integrated search runs at a speed comparable to weighted A*, with an average path length about 8 units shorter. Sequential A* has the shortest average path length, although with a hefty run time, it also has the least expanded nodes on average. It is most comparable to the results of the anchor heuristics when used with A*, most probably as a result of the weights defined. The Sequential and Integrated versions have more potential to be optimized in an autotuner, where various combinations of weights could be tested against a condition to optimize the tradeoff between run time and path length. In our implementations, we frequently sacrifice memory in favor of run time. Further, C# uses a managed garbage collection system, in which we do not have readily available access to lower level data management, and as such, we do not explicitly examine the bytes. Rather, we focus on the number of nodes expanded, as this is a more applicable measure of memory usage, since nodes will need to be generated for every expansion.

Averages	Cost	Number of nodes	Milliseconds	Bytes
Algorithm	**PathLength**	**Expanded**	**RunTime**	**Memory**
A*	112.295746	74878.51556	2707.617778	13552955.72
Weighted A*	126.9553872	40598.93053	1547.202105	13999602.81
Uniform Cost	114.2852468	163853.808	4139.896	14083203.07
Sequential A*	97.7325215	7017.8	8832.4	14060448.44
Integrated A*	118.9158697	221454.46	1561.76	13793494.8
Heuristic	**PathLength**	**Expanded**	**RunTime**	**Memory**
Euclidian	138.4334984	520.46	63.92	13691905.84
Manhattan	160.5713267	338.04	28.26	13850213.6
Manhattan/4	96.8046718	5582.01	3301.83	14051158.72
Octal/4	125.6418406	2994.73	909.95	14157216.96
Chebyshev/4	96.4204407	7782.42	5841.13	13706298.28
A*	**PathLength**	**Expanded**	**RunTime**	**Memory**
Euclidian	106.6705144	1554.8	238.6	12469489.44
Manhattan	130.7021908	816.96	92.72	12788688.8
Manhattan/4	95.7627552	8407.24	6355.36	13416712
Octal/4	106.1000196	4554.48	1917.6	13882429.28
Chebyshev/4	95.7627552	9887.32	9097.56	13799191.04
Weighted A*	**PathLength**	**Expanded**	**RunTime**	**Memory**
Euclidian	149.0211597	175.68	5.693333333	14099377.97
Manhattan	170.5277053	178.4	6.773333333	14204055.2
Manhattan/4	97.15197733	4640.266667	2283.986667	14262640.96
Octal/4	132.1557809	2474.813333	574.0666667	14248812.85
Chebyshev/4	96.6396692	7080.786667	4755.653333	13675334.03
Uniform Cost	**PathLength**	**Expanded**	**RunTime**	**Memory**
	114.2852468	163853.808	4139.896	14083203.07
Sequential A*	**PathLength**	**Expanded**	**RunTime**	**Memory**
	97.7325215	7017.8	8832.4	14060448.44
Integrated A*	**PathLength**	**Expanded**	**RunTime**	**Memory**

	118.9158697	221454.46	1561.76	13793494.8

All data:
https://docs.google.com/spreadsheets/d/1QAIULwyDeULz3m21S5eJX5QuG38L7q2OMlmbs3Sz1m4/edit?usp=sharing

Sunday, February 19, 2017 8:33 PM

project 1 I part2.

1. It can terminate at line 19 for no results, line 32 for Anchor search, or line 24 for inadmissible search.

So if the anchor search terminates at line 32
then it's the case that $key(s,0) \leq key(u,0) \; \forall u \in OPEN_0$.
From the given property we have

$$g_0(s) \leq w_1 * C^*(s)$$
$$\leq w_1 * w_2 * C^*(s) \quad \text{since } w_2 \geq 1$$

If from an inadmissable search in line 24
then from lines 21-24 we have that

$$g_i(s) \leq w_2 * OPEN_0.\text{Minkey}()$$
$$\leq w_2 * w_1 * C^*(s) \quad \text{from previous proof problem.}$$

Therefor, if the anchor or an inadmissable search terminate it is bounded by $w_1 * w_2$ factor of the optimal solution cost.

Finally if it terminates with out a solution in line 19
From the previous proof question we know that $OPEN_0.\text{Minkey}() \leq w_1 * C^*(s)$
So $OPEN_0.\text{Minkey}() \geq \infty \Rightarrow C^*(s) \geq \infty$. So there are no finite cost solution. □

Project 1 phase 2 j

Sunday, February 19, 2017 9:15 PM

i. no state is expanded more than twice.
ii, a state expanded in the anchor search is never re expansed
iii a state expanded in an in admissable search can only be re expanded in the anchor search if it's g value is lowered.

Pf. A state can only be expanded in lines 34, 42, as it is selected as the top state of OPEN;
If S is selected for expansion in line 34, the very next line is the ExpandState call, therefor Removing this state from Open; $i \in 0 \ldots n$. Also after the expandstate call S is inserted to closed; in line 36. Now the only state other than the start state can be put in Open; where $i \neq 0$ in line 17. If S has already been expanded in any inadmissable searches, or S in closed; where $i \neq 0$, the line 14 checks ensures that S is not inserted again in OPEN; where $i \neq 0$. Therefor inadmissable searches can only expand a state once.
In the case where a State S is expanded in an anchor state it is also removed from all OPEN; and inserted to Closed$_0$. Thus S can only be expanded again in an inadmissible search or in anchor search. If it is re inserted to any OPEN, in lines 13 or 17. but line 12 garentees this will never happen as S is in Closed$_0$. Therefor (ii) is true. This also implies that S can be expanded at most once in the anchor search and once in an inadmissible search, therefor (i) is true.
to prove (i,ii) A state S, that has been expanded in an inadmissible search, can only be re-expanded if it is inserted into Open$_0$. A state can only be inserted in Open$_0$ it at line 10 the if conditional is true. So only if S's g value is lower than it's previous g value. Thus if S is a state such that its g value was not lowered after expansion in any inadmissible search, will never make line 10's condition true. and will not be inserted to OPEN$_0$. and thus can never be expanded in the Anchor search. □