Intro to AI Assignment 1 — Heuristic Search

Kenneth Bambridge — kmb394

February 22, 2017

Contents

1 Introduction

In this project several best first search algorithms were explored, notably the A* family of algorithms. The algorithms were implemented and visualized in a GUI. Then another script was written to measure the running time of the algorithms. This data was aggregated and several heuristics and modifiers to the heuristics were compared across different metrics including runtime, memory used, and number of visited nodes.

2 Algorithms

The algorithms explored in this project were best-first-search algorithms, i.e. the algorithms choose the next node to explore by choosing the next best node according to a computed evaluation function.

2.1 Uniform Cost Search

The simplest algorithm explored is uniform cost search, whose evaluation function is a cumulative sum of the costs to get to each node visited, defined as $f(n) = g(n)$. There are no heuristics used. We expect this algorithm to perform fairly poorly as it is $O(b^{d+1})$ time and space complexity where b is the branching factor and d is the depth of the goal.

2.2 A*

A* adds a heuristic to the evaluation function, $f(n) = g(n) + h(n)$. The heuristic represents some prior knowledge of the environment, therefore it is an *informed* search algorithm. If the heuristic is *admissible*, or it never overestimates the cost to the goal node from any node, then the solution will be optimal. If not, then there is a chance that the algorithm returns a suboptimal path. Furthermore, if the heuristic is *consistent*, or it satisfies the triangular inequality $h(n) \leq c(n, n') + h(n')$. A* is optimally efficient in this case, meaning it will only select nodes that will bring it closer to the optimal path.

2.3 Weighted A*

Weighted A* adds a weight w_1 multiplier to the heuristic in the evaluation function, $f(n) = g(n) + w_1 * h(n)$. This addition as we will see can drastically speed up the algorithm, at a w_1 cost to optimality.

2.4 Sequential A*

Sequential A* allows the use of multiple heuristics with an anchor heuristic which, if admissible, will guarantee the return the optimal path. It runs through the non-anchor heuristics in a round-robin fashion, only expanding nodes from the non-anchor heuristics if the keys are less then the anchor.

2.4.1 Proof that Sequential A* is w_1 suboptimal

Given that the g value of any state s expanded by the Weighted A* algorithm is at most w_1 suboptimal for an admissible and consistent heuristic, consider a state s_i in the $OPEN_0$ queue that is on the least cost path to s_{goal}. During the initialization loop starting at line 13, all of the $OPEN_i$ fringes are populated with the start node, which is trivially on the shortest path to s_{goal}. For the iterations starting at line 19, the only time the $OPEN_0$ data structure is inserted or updated is on line 11, where it is inserted with the value $g_0(s) + w_1 * h_0(s)$ for all neighbors where their g values are less then the existing keys in the fringe.

Since during every `ExpandState` call, all neighbors are considered for the node that was originally on the shortest path (the start node), there must also be a node s_n considered every time $OPEN_0$ is updated that is on the shortest path to s_{goal}. This node's g value is also updated if it is less then the existing value. The key of this value is $g(s_n) + w_1 * h(s_n)$. The heuristic is admissible, so $h(s_n) \leq c^*(s_n) \implies w1 * h(s_n) \leq w1 * c^*(s_n)$. Since s_n is on the shortest path to goal, $g(s_n)$ represents the shortest path from $s_{start} \rightarrow s_n$, therefore $g(s_n) + w_1 * h(s_n) \leq w_1 * c * (s_n)$. Since this 'minimum' node exists for the initialization and is maintained in every loop, it exists for the duration of the algorithm.

The same is true for the Weighted A* algorithm above, there is always a node that is on the shortest path in the fringe for similar reasons. The start node is on the shortest path, and a shortest path node must be added every time. Since it is known that the node on the optimal path exists and it's key value must be less then $w_1 * c * (s)$, then the minimum key must also be less then that value in $OPEN_0$ in A* sequential.

2.4.2 Proof that Sequential A* is $w_1 * w_2$ subomptimal

The program can exit in one of two ways, either with the anchor search or via an non-anchor heuristic. If all the non-anchor heuristics have minimum keys greater than the anchor, the anchor will be run and the goal returned if $g_0(s_{goal})$ is less then the minimum key in $OPEN_0$ and infinity. In this case the output is w_1-suboptimal as proven above.

When the program exits with a non-anchor heuristic, it is run when the minimum key is less then $w_2 * OPEN_0.\text{Minkey} < w_2 * w_1 * c^*(s_{goal})$. Therefore when the algorithm exits in this way the solutions is $w_1 * w_2$-suboptimal.

2.5 Integrated A*

Integrated A* shares g and *parent* values across all the heuristics, while still keeping it within the $w_1 * w_2$ bound.

2.5.1 Proof i

No state expanded more than twice.

1. When a state is expanded, it is inserted into one of the CLOSED data structures. (a) In the inadmissible branch, it is inserted into $CLOSED_{inad}$ (line 36). (b) In the admissible branch, it is inserted into $CLOSED_{anchor}$ (line 44).
2. A state is never inserted into $OPEN_i, \forall i$ while it is in the $CLOSED_{anchor}$ data structure (line 12).
3. A state is never inserted into $OPEN_i, \forall i \neq 0$ while it is in the $CLOSED_{inad}$ data structure (line 14).
4. A state can only be expanded after being inserted into an $OPEN_i$ data structure (line 29, 34, 42).
5. When a state is expanded, the state is removed from $OPEN_i, \forall i$ (line 4).
6. By combining (1a), (2), (4), and (5), a state will be expanded at most once in the inadmissible branch (line 35).
7. By combining (1b), (2), (4), and (5), a state will be expanded at most once in the anchor branch (line 43).

Since a non-start state will only be expanded at most once in each branch, therefore no state is expanded more than twice other than the start node.

2.5.2 Proof ii

State expanded in the anchor search is never re-expanded.

1. When the anchor search expands a state, the state is added to $CLOSED_{anchor}$ (line 44).
2. Only the anchor search branch can expand nodes in $OPEN_0$ (line 42).
3. The anchor search branch only expands nodes in $OPEN_0$ (lines 42–44).
4. A state is never inserted into $OPEN_i, \forall i$ while it is in the $CLOSED_{anchor}$ data structure (line 12).
5. After a state is expanded in the anchor search, it is not in $OPEN_i, \forall i$ (line 4).
6. A state can only be expanded while in an $OPEN_i$ data structure (line 29, 34, 42).
7. Therefore, a state expanded in the anchor search is never re-expanded.

2.5.3 Proof iii

A state expanded in an inadmissible search can only be re-expanded in the anchor search if its g-value is lowered.

1. A state expanded in an inadmissible can add the the state to $OPEN_0$ (line 13).
2. Whenever a key is added to an inadmissible $OPEN_i$, then it's key must be less then or equal to the corresponding admissible heuristic (line 16)
3. If a state s_i was expanded in an inadmissible heuristic, then it was removed from $OPEN_0$ (line 4).
4. The g value is retained in a data structure, but the node is only re-added to $OPEN_0$ if the check in line 10 is satisfied for $n \in \text{succ}(s_i)$, which for which it must be a lower g value.

3 Heuristics

3.1 Chebychev Distance

The Chebychev distance is the maximum distance of the minimum distance between to spaces in either the horizontal or vertical direction. This heuristic is not admissible because of highways.

3.1.1 Admissible Version

It is admissible when divided by 4 because even in the case when there is only highway movement from the start to the goal, the heuristic will not overestimate the cost to get to the goal.

3.2 Manhattan Distance

The Manhattan distance is the distance required by only making horizontal or vertical movements. Manhattan distance represents a less direct path then the diagonal distance for 2d grids that allow 8 directional movement. I would expect this to underperform compared to the diagonal distance but nonetheless be good guidance. It is not admissible because if there is a valid highway movement on the shortest path the heuristic will return a greater value then the true cost. It is consistent in the equal cost case because it satisfies the triangular inequality.

3.2.1 Admissible Version

If Manhattan distance is divided by 4, then it is admissible for similar reasons to the Chebychev distance.

3.3 Diagonal Distance

The Diagonal distance is the minimum distance when diagonal movement is allowed. This heuristic is not admissible for similar reasons to the Manhattan and Chebychev distance. This heuristic in the absence of any terrain modifications the best heuristic for 8-directional movement defined in this environment because it calculates the exact distance in number of blocks needed to be traveled to get to and from two spaces on a 2d grid. It dominates any of the other heuristics while maintaining admissibility in the equal-cost case. It is consistent in the equal-cost case.

3.3.1 Admissible Version

With the addition of spaces with sped up movement, the minimum cost to get to and from two nodes would be a path consisting of all of the fastest spaces in that grid. Therefore the potential minimum cost to travel between two nodes would be a path consisting of all of the minimum cost space, resulting in $h(n) = d(n) * s_m$ where s_m denotes the minimum cost space. In our case the minimum cost is traveling between two highways vertically or horizontally which is 0.25, making our best admissible heuristic $h(n) = d(n) * 0.25$.

3.4 Euclidian Distance

Euclidean distance does worse than diagonal distance for 8 directional movement and it can be illustrated when we look at a 2d grid with equal movement costs. For any two start and goal pairs, the diagonal distance is at least as long as the euclidian distance, them being equal when the start and goal are equal distance away in the horizontal and vertical directions. That is, diagonal distance dominates euclidian distance in this situation. Both Euclidean and Diagonal distance share the same inadmissibility problem in this case so one would expect the diagonal to outperform the euclidean distance for this problem. Euclidian distance would be consistent in an equal-cost case because Euclidian geometry satisfies the triangular inequality.

3.5 Favor Highways Modification

This is a modification that can be applied to all the previous heuristics. If any node is a highway, reduce it's heuristic value by multiplying by the cost reduction of the highway, so the heuristics value becomes $h(n_{highway}) = h(n) * .25$.

3.6 Favor Highways Smart Modification

One shortcoming of the favor highways (3.5) modification is it will keep on the highway far beyond it needs to go in the direction of travel to the goal node. This heuristic tries to fix this shortcoming.

If the start node is a highway and there is a highway in one of the directions (vertically or horizontally) that needs to be traversed to get to the goal node, inflate the value of that heuristic by reducing it's value scaled by the cost savings of traveling on a highway while it's in a desired direction of travel.

- Highway movement – 0.25 per direction of movement (only one direction because it is horizontal or vertical)
- Rough Highway movement – 0.5 per direction of movement

This heuristic combined with diagonal distance is probably the optimal configuration given what is known about the map. This is because it bests diagonal distance if there is a highway on the way. To illustrate this consider the smallest case where instead of traveling diagonally for $\sqrt{2}$ movement, take the Manhattan distance along a (non-rough) highway in the direction of movement and then normal movement. The cost of this move is $1.25 < \sqrt{2} = 1.41$. If the highway is rough, however, then it will slightly underperform. Considering there are much fewer rough squares, and even in the situation was within a 31×31 square of 50% chance of being rough, we can show that it is still worth it for greater than 1 moves because:

$$.5 * .25n + .5 * .5 * n + .5n + .5 * 2n = 1.875n < 1.41 * .5n + 2.82 * .5n = 2.115n. \tag{1}$$

4 Implementation Notes

A combination of functional programming and object-oriented programming was used to program this assignment. On the functional programming side, techniques like currying, first-class function types, and immutable types (like tuples) were used. For OOP, a inheritance was used for the fringe so I could swap different data structures in and out, a class was made called `Space` which represented one space in the grid, and the GUI framework used called `Tkinter` was used in an object-oriented fashion. Here are some notes on the optimizations and strategies used to implement various parts of the project.

4.1 Map Generation

The map was generated using functions from `numpy` and a class was written named `Space` to store terrain data for individual locations on the map. The functionality of 2d arrays in `numpy` including the ability to choose random elements from an a 2d array and functions to input and output to files was useful for map generation. See `maps.py`, `grid.py`, and `gen-map.py` for details.

4.2 Algorithm Implementation

The algorithm functions were written as generators since it provided the functionality of an iterator which made step by step animation possible without tightly coupling the code with the GUI. See `search.py` and `fringe-binheap.py` for relevant code.

4.2.1 Organization

A general `best-first-search` was made that serves as the control flow for the algorithms `uniform-cost-search` and `a-star` searches. An optional parameter was added to `a-star` to support weighted A* which defaults to 1 if omitted. Since they were implemented as generators, `a-star-sequential` is essentially multiple `a-star` searches looped together. A separate control flow was needed to be implemented for `a-star-integrated` because of the shared data structures.

4.2.2 Heuristic Implementation

The heuristics were implemented as normal functions that were decorated with modifications like `make-admissible` and `favor-highways`. See `heuristics.py`.

4.2.3 Optimizations

Data structure changed from heap to binary heap The fringe for the first phase of the project was implemented as a heap using the library heapq. This worked well because it has $O(1)$ pop and $O(\log n)$ insert, but it had $O(n)$ removal which drastically slowed down Integrated A*. Then the fringe was switched to a binary tree using the library pqdict. This sped up Integrated A* by about 50%.

Memoization was used to temporarily store cost and neighbor values Staying true to the functional programming paradigm, the `neighbors` function which calculates the neighbors of a Space was being called many times in Sequential and Integrated A* and was slowing down the algorithms. To speed this up a decorator called `memo` that wrapped the function and stored the values of neighbors in a dictionary so they can be quickly indexed next time they are needed. While this does increase the memory usage of the program it cut the cost of some of the longer running algorithms like `uniform-cost-search` and `a-star-sequential` by a significant amount.

4.3 GUI Visualizer

The GUI visualizer was implemented using Tkinter, a python binding to Tk GUI toolkit. Drawing is done on the canvas after every step of the algorithm, and there is a command line interface to start the visualizer. Run `python visualizer.py -h` for details.

5 Benchmarks

Here we explore the data gathered from the benchmarks.

5.1 Algorithm Comparison

Algorithm	Run Time	Path Length	Path Cost/Optimal	Nodes Expanded	Memory
ucs	0.44028228	155.38	1.0	11532.96	99846963.2
a$_s tar$	0.38645439875	145.03625	1.11695158894	3314.47375	114677667.84
a$_s tar_w$	0.1450405375	135.738	1.3500551616	1148.09325	54550267.904
a$_s tar_s eq$	0.305271811111	133.5225	1.24528362448	2827.02125	91410478.08
a$_s tar_i nt$	0.469058569167	139.679027778	1.22433737248	1717.33777778	100941314.276

5.1.1 Uniform Cost Search, A*, and Weighted A*

Here as expected Uniform Cost Search grossly underperforms any o the weighted heuristic search algorithms. A* with no weight provides a 14% improvement to runtime for an 11% optimality decrease and a 10% increase in memory on average across all heuristics, while weighted A* improves running time by over 50% and using about 50% less memory when considering the weights `[1.25, 2, 3, 4, 10]`. The path cost to optimal ratio is sacrificed as expected.

Uniform Cost Search underperforms relative to heuristic methods because without any information to guide the search, UCS will search in many directions trying to find the shortest path to goal. A* improves this drastically with a good heuristic, but we run the risk of sacrificing optimality. When adding a weight to the heuristic, meaning we will trust the heuristic, we can speed up the algorithm considerably, while continuing to incur suboptimality based on the weight.

Benchmarks data for UCS and A* can be viewed at the link a-star-data

5.1.2 Sequential A*

Sequential A* has too many repeats to be a very useful algorithm. It sounds useful to be able to use many heuristics, but when nodes are expanded up to $n+1$ times there is too much wasted time and memory, as can be seen in table in 5.1. It is most likely more useful to use regular A* with a weight unless there are a many unknowns in the environment and many heuristics are needed to navigate. This algorithm may also benefit from multiprocessing if many heuristics are used.

Benchmarks on Sequential A* were done with the diagonal admissible heuristic and can be viewed at the link a-star-sequential-data

5.1.3 Integrated A*

Integrated A* fixes the shortcoming of Sequential A* and uses significantly less time and memory by ensuring that each node is only expanded at most twice. It performed with similar runtime of A* weighted but with significantly less nodes expanded.

Benchmarks on Integrated A* were done with the diagonal admissible heuristic and can be viewed at the link a-star-integrated-data

5.2 Comparison of Heuristics

Heuristic	Average Run Time	Average Path Length	Average Path Cost/Optimal	Average Nodes Expande
c	0.374780437333	137.485416667	1.24690690773	2250.56883333
m	0.366348710917	137.886833333	1.24188847064	2175.31866667
d	0.370725274551	136.717692308	1.23857789732	2206.68358974
e	0.371076276667	137.725	1.23505278036	2213.63891667
a	0.394651551859	136.905320513	1.22770583835	2447.39064103
f	0.334366948095	139.89297619	1.22583237857	2050.11083333
s	0.384309394048	138.234642857	1.22931499065	2020.49904762
ca	0.564171526667	142.796666667	1.15570152271	5322.23666667
cf	0.354675792564	143.895897436	1.20718287276	2253.57897436
cs	0.457599452051	141.206410256	1.21455534781	2198.1674359
ma	0.390045656667	138.286666667	1.17083489097	3665.09333333
mf	0.351161372564	144.816923077	1.19702862632	2221.05282051
ms	0.453145243846	142.072307692	1.20376428616	2166.78974359
da	0.388789434014	136.679931973	1.23248309149	2317.07857143
df	0.3633373448	139.147333333	1.21801188452	2226.14226667
ds	0.4168981804	137.672533333	1.22141550476	2197.35573333
ea	0.516981246667	140.676666667	1.12249569734	4740.13333333
ef	0.352814044615	144.525641026	1.19451987746	2233.38871795
es	0.454803085897	141.748717949	1.20134913067	2177.67589744

This table is not too informative because the data is diluted with many repeats of the heuristics in A* Sequential and Integrated. Another comparison is made below with only heuristics from vanilla A* and A* weighted. However, it shows the stark contrast between admissible and inadmissible heuristics for this problem. Run time is roughly doubled for every admissible heuristic ('a' denotes admissible) except for the diagonal admissible heuristic because it was used as the anchor for Sequential and Integrated A*.

5.3 Comparison of Heuristics in only A*

Heuristic	Average Run Time	Average Path Length	Average Path Cost/Optimal	Average Nodes Expande
c	0.229672435833	134.873333333	1.37658574196	1912.565
m	0.145355171667	138.8875	1.326401371	1160.06333333
d	0.1734462875	138.120833333	1.28378668343	1420.7325
e	0.192630829167	137.269166667	1.25804446822	1543.26583333
a	0.4844878925	140.56	1.14244991684	4549.92416667
f	0.0890828925	147.2075	1.27970913632	556.518333333
s	0.1111560275	143.6975	1.28244046974	519.283333333
ca	0.564171526667	142.796666667	1.15570152271	5322.23666667
cf	0.116243703333	139.386666667	1.3898778766	811.62
cs	0.14415533	136.816666667	1.39913224789	771.46
ma	0.390045656667	138.286666667	1.17083489097	3665.09333333
mf	0.0705562433333	151.36	1.25787267291	388.78
ms	0.0862506233333	148.073333333	1.2588484464	363.55
da	0.46675314	140.48	1.12076755634	4472.23333333
df	0.0774906433333	150.51	1.24582705813	476.526666667
ds	0.106415586667	146.033333333	1.24432975973	437.053333333
ea	0.516981246667	140.676666667	1.12249569734	4740.13333333
ef	0.09204098	147.573333333	1.22525893764	549.146666667
es	0.10780257	143.866666667	1.22745142495	505.07

This table again shows the how much the admissibility trait has on the running time of this algorithm. This also shows a surprising discovery, contradictory to what was thought before the experiment. It was predicted that diagonal distance would be the best admissible/consistent heuristic, but it is in fact the Manhattan distance with lower runtime, node expansions, and memory used. This most likely has to do with the nature of the environment, and how the highways create a large incentive to move horizontally and vertically, therefore Manhattan distance provides the most accurate pathfinding tool.

Another note is that the 'smart' version of the favor highways heuristic did not outperform the 'dumb' version from a resource use perspective. While the 'smart' version did expand less nodes, the added computation time to figure out if there was a highway surrounding the current node took up enough computation time to offset it.

5.4 Sorted w_1 aggregate comparison

w1	Average Run Time	Average Path Length	Average Path Cost/Optimal	Average Nodes Expanded	M
1	0.64881806625	145.579375	1.09774147231	4398.5796875	13
1.25	0.488736113437	142.293125	1.16539966011	3136.67125	11
2	0.387449830313	138.218125	1.22590144961	2457.32625	98
3	0.271779624375	136.94875	1.25189729453	1577.2953125	80
4	0.150248898438	133.60375	1.31779988353	700.9165625	63
10	0.073125094375	123.991875	1.46471433229	217.75375	33

This table shows how when weight increases, runtime decreases and path cost increases. The largest gain per weight was observed for the initial jump from 1 $\rightarrow$ 1.25. There were still significant decreases in running time, path length, node expansion, and memory usage for the max weight of 10.

5.5 Sorted w_2 aggregate comparison

w2	Average Run Time	Average Path Length	Average Path Cost/Optimal	Average Nodes Expanded	M
1	0.70620277625	141.275833333	1.11808998975	4527.39291667	13
1.25	0.604753754167	141.296666667	1.12048473196	3928.98416667	12
2	0.388317910833	138.226666667	1.18500084199	2468.64791667	10
3	0.232930965	134.724583333	1.27366319717	1195.88375	79
4	0.190250043333	131.439166667	1.35576806177	755.338333333	60
10	0.20053569125	132.641666667	1.35585616822	756.83	64

The second weight parameter did not seem to have as much of an effect, but that would depend on the inadmissible heuristics used. Still a similar expected trend.

5.6 Sorted w_1, w_2 aggregate comparison

w1	w2	Average Run Time	Average Path Length	Average Path Cost/Optimal	Average Nodes Expand
1	1	1.3055408175	155.38	1.0	9036.2275
1	1.25	1.1389432575	152.9125	1.01194370557	8045.7325
1	2	0.68917947	144.6575	1.09097550097	4727.845
1	3	0.4598987025	141.6325	1.14160070729	2687.46
1	4	0.4142671725	139.925	1.15126467549	2119.78
1	10	0.4098063125	140.055	1.15224401128	1942.645
1.25	1	1.253215495	155.25	1.00005340031	7930.8
1.25	1.25	0.984279645	152.775	1.01016845442	6615.73
1.25	2	0.526259755	141.785	1.13478463985	3376.235
1.25	3	0.2297721625	137.1025	1.24099863731	1049.7775
1.25	4	0.18435012	135.09	1.24509090812	698.785
1.25	10	0.19218153	134.4675	1.24430597796	670.725
2	1	0.9244946275	148.8	1.01285013602	5811.0325
2	1.25	0.7677099025	147.3225	1.02837134961	4954.0025
2	2	0.5055669275	141.2125	1.13280726147	3507.6975
2	3	0.2198643875	133.5575	1.31621655376	1146.23
2	4	0.14399206	130.5975	1.35538093183	495.5825
2	10	0.14890624	130.455	1.35687075781	489.5375
3	1	0.5164951325	143.0825	1.03455743722	3180.225
3	1.25	0.4909332325	143.14	1.04867683576	2846.475
3	2	0.372200435	143.065	1.11014227766	2238.93
3	3	0.241194325	136.72	1.24854860221	1411.41
3	4	0.14406234	129.43	1.44394523617	482.3825
3	10	0.14746969	129.335	1.44365748996	478.2125
4	1	0.2027698275	133.18	1.18064786977	1090.1275
4	1.25	0.20226664	134.89	1.18428582081	974.42
4	2	0.1668906725	137.6675	1.2252091785	795.7725
4	3	0.1628069325	135.305	1.27533102767	699.935
4	4	0.145378455	129.36	1.46040809638	478.5825
4	10	0.151667295	129.58	1.46100545421	479.4475
10	1	0.0347007575	111.9625	1.48043109517	115.945
10	1.25	0.0443898475	116.74	1.43946222558	137.545
10	2	0.069810205	120.9725	1.4160861935	165.4075
10	3	0.08404928	124.03	1.41928365478	180.49
10	4	0.1094501125	124.2325	1.47851852265	256.9175
10	10	0.15318308	131.9575	1.47705331813	480.4125