

# Intro to AI Project 1

Tim Raymaker and Mike Rizzo

July 17, 2019

## 1 Part 1: Understanding the Methods

### 1a

The agent will move toward the cell with the lowest  $f$ -value. The  $f$ -value is determined from the sum of the cost from the start, or  $g$ -value, and the heuristic value to the goal, or  $h$ -value. For the first move, each possible direction is one tile away from the start, meaning that the  $g$ -value for each move will be the same, and differences in the  $h$ -value will be the deciding factor on where to move. Since the agent does not know that cells D3 and E4 are blocked, it will move to the east, the direction that leads to the cell with the lowest  $h$ -value and therefore the lowest  $f$ -value.

### 1b

The answer has been divided into four cases:

1. Case 1: In an infinite grid, the agent can find the target and the target is reachable. Yes, a path can be found and in finite time.

The agent will go about expanding reachable nodes that decrease in  $f$ -value over time. This will lead it closer and closer to the target until eventually it will encounter the target and stop. Assuming the target is not moving, it will be a finite distance away and therefore only a finite number of cells with generally decreasing  $f$ -values will be expanded between the agent and the target.

2. Case 2: In an infinite grid, The agent can tell there is no solution when the target is unreachable. A solution cannot be found; this case does not exist.

The agent will expand all reachable nodes and even though the values of  $f$  will be generally decreasing, if the target is unreachable, the agent

will resort to cells with higher and higher  $f$ -values which in an infinite grid are also infinite and will thus never terminate.

3. Case 3: In a finite grid, the agent makes an upper-bounded number of moves when the target is reachable. Yes, an upper-bound  $n^2$  exists. In a theoretical worst case scenario, the agent must replan  $n$  times in order to find the goal, one replan for each tile on the map. With each replan it is forced to expand every node on the grid, meaning that each replan makes  $n$  moves. In this case, the worst possible, the number of moves is upper-bounded by  $n^2$ .

4. Case 4: In a finite grid, the agent makes an upper-bounded number of moves when the target is unreachable. Unlike case 2, an upper-bound exists.

It is in fact the same bound as case 3 and follows the same reasoning. The agent will have to investigate  $n$  cells in the finite grid with each of  $n$  replans before determining that the target is unreachable, for  $n^2$  total moves.

## 2 Part 2: The Effects of Ties

Tiebreaking was important in the planning phase of the A\* search. Low- $g$  tiebreaking meant prioritizing tied neighbors that were closer to the start in the heap. High- $g$  meant the opposite. From our data low- $g$  was able to find very slightly shorter paths in slightly shorter times than high- $g$  was. This seems counterintuitive, given that in the example problem a high- $g$  search will drastically outperform low- $g$  by cutting straight to the goal where low- $g$  explores other nodes, but it is worth considering that high- $g$ 's worst case was often worse than low- $g$ 's. High- $g$  searches favored diving toward the goal, which for an uncluttered map is optimal, but our map had enough obstacles that every search was significantly hampered. High- $g$ , in its attempt to 'greedily' find a solution as fast as possible, sometimes ended up trapped in dead end areas with no way forward, being forced to backtrack significantly. Low- $g$ , on the other hand, performed a slower but safer search. In this example low- $g$  is more akin to BFS while high- $g$  played a similar role to DFS. Of course, the average and best case scenarios of high- $g$  are much better than those of low- $g$ . With more maps and more data, perhaps a higher density of average cases would have shown high- $g$  to perform better overall.

On average, the runtime for favoring low  $g$  was 5.84 seconds. Favoring high  $g$  was nearly identical, being slower on average only by a hundredth of a second at 5.86 seconds. Investigating further, we see the medians are still

only fractions of a second different with. Though our average runtimes were nearly identical, using  $g$  to tiebreak would likely be useful for performance on an empty map as it would help limit expansions. In the maps provided, the amount of obstacles in the way means that there is relatively little gain with manipulating  $g$ . With more maps and more data, perhaps a higher density of average cases would have shown that tie-breaking in favor of high  $g$ -values will perform better overall.

### 3 Part 3: Forward vs. Backward

The Repeated Forward A\* search and Repeated Backward A\* search were both able to find possible solutions, where solutions existed, but their exact paths differed in most cases. The difference comes from the path planning and replanning. Because Repeated Backward A\* plans from the goal, different blocked and unblocked cells are found at each iteration. Neither method's path was consistently superior to that of the other. This is likely because that although they solved the same maze, each one had different information available to them as they could only see cells near them. As such, their differing starting locations led them down different paths. Neither method's path was consistently superior to that of the other, but the backwards paths tended to have worse runtimes, perhaps due to the fact that we only animated the projected paths of A\* in the first quadrant. Although the forward searches would have longer projected paths, the amount of replans for the backward searches was likely higher in the first quadrant as the searches neared their goal and the amount of available moves became more limited.

### 4 Part 4: Heuristics in the Adaptive A\* (i)

The "Manhattan Distance" can be described as the number of moves required to reach a goal given no obstacles. More formally it is the function  $d(a, b) = (|x_a - x_b|) + (|y_a - y_b|)$  where  $a$  is the current state and  $b$  is the next state.

A Consistent Heuristic must satisfy the inequality  $h(N) \leq c(N, P) + h(P)$  and initial condition  $h(G) = 0$  where  $N$  is the current node,  $P$  is any descendent of  $N$ , and  $G$  is the target or goal. The proof that the Manhattan Distance follows:

1. For the initial condition,  $d(G, G) = (|x_G - x_G|) + (|y_G - y_G|) = 0$  which makes sense because the distance from the goal to itself is 0. So  $h(G) = 0$  is satisfied.

2. For the inequality,  $P$  can only take on one of four directions each a distance of 1 away from its parent  $N$ , making  $c(N, P)$  always equal to 1.
3. Now we have  $h(N) \leq 1 + h(P)$ , we will then substitute  $h(N)$  and  $h(P)$
4.  $(|x_S - x_N|) + (|y_S - y_N|) \leq 1 + (|x_S - x_P|) + (|y_S - y_P|)$ , where we now see that the coordinates of  $P$  can be replaced with  $N$ 's coordinates plus 1 based off of premise 2.
5.  $(|x_S - x_N|) + (|y_S - y_N|) \leq 1 + 1 + (|x_S - x_N|) + (|y_S - y_N|)$  and with some algebra
6.  $0 \leq 2$

Therefore, it makes sense that the Manhattan Distance will be consistent because the consistent inequality holds true.

In Adaptive A\*,  $h(n)$  can sometime change to  $g(s_{goal}) - g(s)$  when an a previously expanded node is encountered. Any value that has not been expanded will remain unchanged, making its  $h(n)$  still the Manhattan Distance and still consistent. The new heuristic is consistent because although its based off of olpath costs, a new path will be calculated that will be equal to or longer than the previous path and so the respective  $g$ -values will be less than or equal to their new path values.

## 5 Part 5: Heuristics in the Adaptive A\* (ii)

Our Adaptive A\* method was able to find shorter paths than Repeat Forward A\* on average, but it also took a longer time. This is likely due to it attempting to avoid previous projected paths and move in new directions. Our A\* recalculated the heuristic value of every tile that was expanded by the previous A\* search upon replanning, increasing their cost by using  $h(n) = g(s_{goal}) - g(s)$  and incentivizing the algorithm to explore alternatives. By doing so it was sometimes able to find a better path at the cost of runtime. Ties were broken randomly and each search weighed in favor of larger  $g$  values.

## 6 Part 6: Memory Issues

### 6a Improvements to Our Implementation

We currently use Python's tuples to record  $(x, y)$  coordinate information. According to Python's "sizeof" function, for one, 2-integer tuple, the memory consumption is 40 bytes. We could reduce this by storing the coordinates in the minimum number of bits it takes to store two integer values between  $0 \rightarrow 100$ . This would be  $\log_2(101)$  or 7 bits per value and 14 total. We could then also keep the tree pointers on two bits with 00, 01, 10, 11 corresponding to whether the neighbor is N, S, E, W of the current node. We can do this because the neighbors are only in discrete directions. Altogether, we could store grid information, excluding the storage of  $f(n)$ ,  $g(n)$ , and  $h(n)$  values, at 16 bits a cell.

### 6b Calculations

$(1,001 \times 1,001) = 1002001$  cells

At  $\log_2(1001) * 2$  or 20 bits per coordinate and 2 for the parent we get 22 bits per cell:

$22 * 1002001 = 22044022$  bits or roughly 2.8MB

4MB is 32000000 bits  
 $32000000 = 2\log_2(x) + 2x$   
 $x = 16000000$  cells  
 $\sqrt{16000000}$  or 4000 X 4000 grid

## 7 Data, Methods, and Observations

Data was collected for Repeat Forward A\* with Tie-breaking in favor of Low  $g$ -values, Repeat Forward A\* with Tie-breaking in favor of High  $g$ -values, Repeat Backward A\* with Tie-breaking in favor of Low  $g$ -values, Repeat Backward A\* with Tie-breaking in favor of High  $g$ -values, Adaptive A\* with Tie-breaking in favor of High  $g$ -values algorithms. The searches were each performed on the 50 maps generated in Part 0. The start and end goal were standardized to  $(0, 0)$  and  $(100, 100)$ . Of the 50, 26 had solutions. The 24 without solutions are marked in red and their runtimes and paths are not included in the data because animation conflicted with reporting significant and comparable runtimes and paths. For each of the 5 algorithms, we have calculated and provided a median and mean path and runtime.

Sources of error can be attributed to small sample size and random choices in cases of  $g$ -values also tying. If there were more time, we would have like to address the small sample size by generating more graphs that tested specific edge cases as well as graphs at varying amounts of blockages. As well, the randomization in the implementation can lead to variance in the paths taken and nodes expanded even by the same search on the same map. Finally, optimization differences between different hardware and software may also have played a role in the uniformity of our data. We acknowledge these issues and have answered the as best as possible.

map #	RFAS, low g	RFAS, high g	RBAS, low g	RBAS, high g	Adaptive, high g
0	7.7213574	9.1950901	6.7969897	7.8623861	9.0152847
2	6.9196187	6.9712262	7.6465468	5.5816107	5.6690023
3	3.9915893	4.3840804	6.7751229	7.1500973	4.2392989
4	5.9092012	6.0295224	11.0584462	8.4783229	7.0102482
8	4.6046704	12.5949367	11.3128128	13.7844876	8.4654507
9	6.8428818	5.7294314	7.6365113	6.9002095	8.309093
10	4.7868589	3.042356	10.605645	3.366994	3.0213504
11	3.7711137	3.361756	5.7730264	5.2442304	4.0409993
13	5.1367127	4.8565318	4.7986285	5.2731823	5.2888487
14	4.7563143	4.0845828	3.2024566	4.5925566	6.0809099
16	4.8462931	3.5783031	6.7569728	7.5016737	5.1775221
17	8.1781014	5.9725671	6.3026338	8.0480699	6.5414467
21	5.3414607	6.6613533	10.2345253	4.3040543	6.1683117
26	7.1164372	5.8671252	17.3251894	12.5306752	6.0977053
27	10.260051	7.6755199	7.6325636	13.9176276	8.506881
28	6.9538869	7.150394	11.333109	5.1050541	8.8734808
29	2.8755443	5.5396908	7.9331223	4.9588877	8.0512628
31	6.1890968	4.4748292	4.1596228	3.8154154	8.1954607
36	5.3203536	6.2204821	4.4558545	10.1051347	6.3781885
38	4.5374294	4.4214935	5.267704	5.9583476	2.8418502
39	5.2069622	4.6570976	4.033652	14.0452184	6.0060953
40	6.912583	7.1265559	2.7502696	10.3243313	7.8313355
42	7.9304773	6.647907	7.701683	8.7141308	6.3253358
45	5.6581125	5.6563307	5.6652659	5.8762738	6.642757
48	5.4622862	6.4065723	7.2156794	4.1108994	5.4881515
49	4.8295453	4.2993716	7.9657949	3.7306057	4.8967266
Median Values	5.40187345	5.7982783	7.00633455	6.42927855	6.24682375
Mean Values	5.848420742	5.869427196	7.397685712	7.356941423	6.352422985

Figure 1: Our Run Time Data

map #	RFAS, low g	RFAS, high g	RBAS, low g	RBAS, high g	Adaptive, high g
0	230	248	280	228	240
2	240	282	262	250	242
3	230	260	244	224	232
4	232	258	256	244	250
8	240	260	278	228	254
9	288	294	256	242	280
10	248	250	320	240	244
11	222	234	234	264	230
13	228	230	274	268	246
14	226	250	222	232	248
16	276	218	232	246	230
17	238	254	238	236	256
21	224	244	282	224	226
26	238	228	266	252	230
27	302	266	266	266	250
28	240	260	230	216	240
29	240	230	246	224	218
31	228	220	262	254	244
36	260	242	302	310	256
38	266	262	228	234	254
39	250	224	242	242	244
40	228	238	222	244	244
42	258	240	314	324	248
45	288	264	232	232	288
48	224	234	242	242	232
49	252	228	222	254	282
Median	240	246	251	242	244
Mean	246	246.8461538	255.8461538	246.9230769	246.4615385

Figure 2: Our Path Data