Artificial Intelligence

Programming Project Part 2: Searchclient

Lucian Leahu

DIS Copenhagen Fall

18 November 2019

Team Name: Sejelo

Contributors: Sejal Dua & Angelo Williams

Group Declaration of Work Breakdown

IDEAS: Sejal and Angelo
PROGRAMMING: Sejal and Angelo
BENCHMARKING: Sejal
THEORETICAL: Angelo
REPORT: Sejal and Angelo

Exercise 5: Heuristics

What was your choice of data structure for the frontier?

The data structure we use for the frontier is a priority queue. For best-first search, we always want to choose the state that has the lowest cost according to our heuristic. A priority queue in Java works perfectly for this task since we can pass it our custom heuristic comparator and it will order the states as they are placed into the priority queue. This means that we can just the poll() method to grab the state with the lowest cost.

Provide mathematically precise specification, reasoning, and benchmarks for the heuristics

FIRST HEURISTIC: Manhattan distance

Our first heuristic was quite simple. Our very first thought was that the most important information for our heuristic to use was each box's distance from the goal. Since every level is a grid of squares, Manhattan distance is a simple implementation that would work well. Let x_b, y_b be the x- and y-coordinates of the box respectively. Let x_g, y_g be the coordinates of the corresponding goal. Then,

$$\text{Manhattan distance} = |x_b - x_g| + |y_b - y_g|$$

Our heuristic was the sum of the Manhattan distances between every goal and a corresponding box. Let $numGoals$ be the amount of goals, g_n be the n^{th} goal of the level, and b_n be the goal's corresponding box. Then,

$$\text{h}(n) = \sum_{i=1}^{numGoals} \text{manhattanDist}(g_i, b_i)$$

$$\text{h}(n) = \sum_{i=1}^{numGoals} |x_{b_i} - x_{g_i}| + |y_{b_i} - y_{g_i}|$$

It is important to note that in situations where there were multiple boxes of the same label, the heuristic would use the box that was furthest to the bottom-right of a level. In situations where there were multiple goals of the same label, the heuristic would only account for the most bottom-right goal, ignoring the rest. If the level had multiple boxes and multiple goals all of the same level, both problems would exist in parallel. Another notable limitation of this heuristic is not accounting for the position of the agent.

Table 3: Optimized Implementation #2 - Exercise 2					
Level	Frontier	Time (s)	Memory Used (MB)	Solution length	States Generated
SAD1	BFS	0.029	3.72	19	80
SAD1	DFS	0.025	3.59	27	75
SAD2	BFS	14.467	692.72	19	635190
SAD2	DFS	0.022	3.59	25	86
SAfriendofDFS	BFS	1.185	72.13	8	89112
SAfriendofDFS	DFS	0.031	3.72	60	305
SAfriendofBFS	BFS	0.048	5.12	3	1227
SAfriendofBFS	DFS	35.552	2015.72	981528	2953986
SAFirefly	BFS	18.261	2252.72	60	1961416
SAFirefly	DFS	24.539	4101.72	2517074	4089953
SACrunch	BFS	179.986	3840.72	N/A	6464388
SACrunch	DFS	9.218	4392.28	380992	1023377

Figure 1: Optimized uninformed search implementation benchmarks from project part 1.

Table 1: Heuristic Implementation #1 using best-first search client - Exercise 5					
Level	Evaluation	Time (s)	Memory Used (MB)	Solution length	States Generated
SAD1	A*	0.074	4.00	19	78
SAD1	Greedy	0.072	3.62	21	75
SAD2	A*	41.015	940.72	19	359928
SAD2	Greedy	0.092	4.56	53	284
SAfriendofDFS	A*	2.820	75.93	8	35432
SAfriendofDFS	Greedy	0.046	3.62	10	120
SAfriendofBFS	A*	0.038	3.34	3	50
SAfriendofBFS	Greedy	0.038	3.34	3	30
SAFirefly	A*	2.555	121.30	60	9209
SAFirefly	Greedy	0.195	15.68	99	787
SACrunch	A*	180.000	1317.82	N/A	1888924
SACrunch	Greedy	8.013	554.48	140	166347
SAsoko1_64.lvl	Greedy	0.085	4.84	62	123
SAsoko2_64.lvl	Greedy	0.550	10.84	62	309
SAsoko3_64.lvl	Greedy	172.183	8192.00	N/A	813831

Figure 2: Informed search implementation benchmarks using heuristic 1.

Analyzing the improvements provided by A* and greedy best-first search over BFS and DFS

Figure 1 at the top of the previous page shows the benchmarks after we implemented the second optimization in part 1 of the programming project. It is worth taking a second to draw a comparison between the BFS and DFS search client implementation versus the best-first search client that takes a heuristic-based approach (Figure 2). For levels SAD1 and SAD2, we notice that the time and memory usage is slightly increased due to the preprocessing and greater overhead of the heuristic, but the optimal solution is found and using a minimal number of states. Moving onto SAfriendofDFS and SAfriendofBFS, we notice that the greedy evaluation blows DFS and BFS out of the water. The performance of A* for these levels is comparable to that of BFS except with significantly less states generated. For SAFirefly, we see tremendous improvements across the board. SACrunch still proves to be troublesome for the searchclient, but the A* evaluation burns though memory less quickly and generates less states. The greedy implementation actually solves the level faster than DFS could and using about 1/8 of the memory to find a solution in only 150 moves! SAsoko1_64 and SAsoko2_64 don't prove to be an issue for the greedy informed search implementation, but SAsoko3_64 is too demanding.

SECOND HEURISTIC: Linking boxes to goals

The second heuristic we implemented attempted to address the obvious issues with the first attempt. It still uses the sum of Manhattan distances for box-goal pairs. However, the manner in which box-goal pairs are created is very different. Starting in the upper left, each goal links with the closest box of the same label. Boxes cannot be linked with more than one goal. Additionally, the heuristic also attempts to account for the position of the agent. The distance from the agent to the first box-pair link is added to the heuristic. The final change was to preprocess all of goal positions, since this is standard across every state of a level. Recall the variables used for the first heuristic. Let x_a and x_y be the coordinates of the agent and x_{b_c} and y_{b_c} be the coordinates of the first box linked to a goal. Then,

$$\mathrm{h}(n) = \Big(\sum_{i=1}^{numGoals} \mathrm{manhattanDist}(g_i, b_i) \Big) + \mathrm{manhattanDist}(a, b_c)$$

$$\mathrm{h}(n) = \Big(\sum_{i=1}^{numGoals} |x_{b_i} - x_{g_i}| + |y_{b_i} - y_{g_i}| \Big) + |x_a - x_{b_c}| + |y_a - y_{b_c}|$$

A big issue with this heuristic is that the distance of the agent to the first box-goal pair, which would typically be the 'A' goal, is usually not optimal.

Table 2: Heuristic Implementation #2 using best-first search client - Exercise 5					
Level	Evaluation	Time (s)	Memory Used (MB)	Solution length	States Generated
SAD1	A*	0.055	3.65	19	75
SAD1	Greedy	0.058	3.65	21	52
SAD2	A*	0.119	8.56	19	861
SAD2	Greedy	0.088	3.65	21	78
SAfriendofDFS	A*	1.350	37.27	8	4579
SAfriendofDFS	Greedy	0.066	3.72	8	106
SAfriendofBFS	A*	0.053	3.37	3	40
SAfriendofBFS	Greedy	0.047	3.37	3	56
SAFirefly	A*	5.127	162.50	62	49106
SAFirefly	Greedy	3.192	62.35	240	19353
SACrunch	A*	180.000	1942.99	N/A	2484912
SACrunch	Greedy	25.474	1546.53	334	510164
SAsoko1_64.lvl	Greedy	0.071	4.00	62	123
SAsoko2_64.lvl	Greedy	0.272	9.40	62	309
SAsoko3_64.lvl	Greedy	180.000	668.72	N/A	59024

From the first heuristic to the second, we can see that assigning box-goal pairs may not necessarily always save time due to higher computational costs, but it does result in generating less states, on average. Assigning box-goal pairs also can help the greedy algorithm find the most optimal solution (take a look at SAD2_greedy from Table 1 to Table 2: solution length decreases from 53 to 21). On the other hand, it also results in a less optimal solution (take a look at SACrunch_greedy from Table 1 to Table 2: solution length increases from 140 to 334. The best-first search client also struggles with SAFirefly. We see way less memory usage on the SAsoko levels, however.

THIRD HEURISTIC: Linking agents to box-goal pairs

The third heuristic improves upon the previous by using the distance of the agent to the closest box-goal pair. In other words, the agent has the lowest cost if it is getting closer to the box-goal pair that have the shortest distance between them. The formula for the heuristic is the same as above, except now x_{b_c} and y_{b_c} are the coordinates of the box with the shortest distance to its linked goal.

$$\mathrm{h}(n) = \left(\sum_{i=1}^{numGoals} \mathrm{dist}(g_i, b_i) \right) + \mathrm{dist}(a, b_c)$$

$$\mathrm{h}(n) = \left(\sum_{i=1}^{numGoals} |x_{b_i} - x_{g_i}| + |y_{b_i} - y_{g_i}| \right) + |x_a - x_{b_c}| + |y_a - y_{b_c}|$$

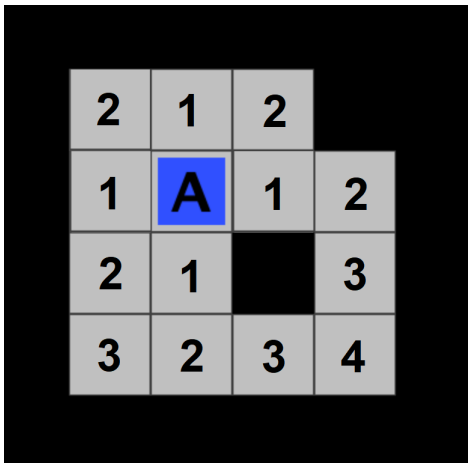
The largest issue facing our heuristic now is from how Manhattan distance is calculated. Our implementation of Manhattan distance completely ignores any walls that are blocking the path between two objects.

Table 3: Heuristic Implementation #3 using best-first search client - Exercise 5					
Level	Evaluation	Time (s)	Memory Used (MB)	Solution length	States Generated
SAD1	A*	0.128	3.65	19	75
SAD1	Greedy	0.117	3.65	21	52
SAD2	A*	1.705	57.85	19	19413
SAD2	Greedy	0.755	53.25	21	4437
SAfriendofDFS	A*	0.702	20.35	8	3401
SAfriendofDFS	Greedy	0.065	4.00	10	145
SAfriendofBFS	A*	0.037	3.37	3	40
SAfriendofBFS	Greedy	0.034	3.37	3	56
SAFirefly	A*	3.126	73.23	72	39157
SAFirefly	Greedy	0.783	31.36	140	2224
SACrunch	A*	180.000	1403.99	N/A	2285375
SACrunch	Greedy	14.836	169.72	190	202358
SAsoko1_64.lvl	Greedy	0.126	4.00	62	123
SAsoko2_64.lvl	Greedy	0.363	9.40	62	309
SAsoko3_64.lvl	Greedy	180.000	758.72	N/A	59024

The changes from the last heuristic to this one affect the performance in variable ways. In some cases we see a decrease in runtime (check out SAsoko1_64 and SAsoko2_64). But most notably is the incredibly large decrease in states generated when evaluating the level using A*. The amount of states generated decreases by almost an entire order of magnitude for most levels of A*. We also observe overall reductions in memory usage.

FOURTH HEURISTIC: Preprocessed distance maps

The fourth heuristic has two major differences from the past iterations. The first change is that now the Manhattan distance calculations account for walls. For every square in the level, a distance map is generated. The distance map is a 2D array containing the distance of every other square to the selected square. See the below graphic for an example of a distance map. These distance maps are generated by running a breadth-first search. States are generated in waves that propagate from the starting tile outwards, with each wave of being one block further away.



The second large change is the amount of preprocessing. With this heuristic, not only are the goal positions preprocessed like with the previous two heuristics, but now the distance map for every square is generated before the search algorithm is even initialized.

$$h(n) = (\sum_{i=1}^{numGoals} \text{dist}(g_i, b_i)) + \text{dist}(a, b_c)$$

Table 4: Heuristic Implementation #4 using best-first search client - Exercise 5					
Level	Evaluation	Time (s)	Memory Used (MB)	Solution length	States Generated
SAD1	A*	0.146	4.19	19	60
SAD1	Greedy	0.124	4.19	19	47
SAD2	A*	0.146	4.19	19	60
SAD2	Greedy	0.110	4.18	19	75
SAfriendofDFS	A*	0.282	13.15	8	1950
SAfriendofDFS	Greedy	0.060	4.56	8	130
SAfriendofBFS	A*	0.067	8.00	3	40
SAfriendofBFS	Greedy	0.074	8.28	3	56
SAFirefly	A*	1.959	193.39	62	29629
SAFirefly	Greedy	1.361	80.13	206	15218
SACrunch	A*	180.000	1929.75	N/A	2916068
SACrunch	Greedy	1.891	201.10	147	37380
SAsoko1_64.lvl	Greedy	0.084	5.37	62	123
SAsoko2_64.lvl	Greedy	N/A	N/A	N/A	N/A
SAsoko3_64.lvl	Greedy	N/A	N/A	N/A	N/A

This implementation really helped to improve our benchmarks in every category. SAD1 and SAD2 have the best benchmarks yet, which is a fantastic indicator. The most significant improvement can be seen in SAfriendofDFS and SAfriendofBFS. Where before the search client was generating 3401 states for SAfriendof DFS A*, it now generates a little over half of that amount with 1950 states generated instead. The client performs better on SAFirefly with A* but compromises performance on SAFirefly with greedy search. Regarding SAsoko2_64 and SAsoko3_64, we struggled with them for this heuristic because we got a Java heap space error as the levels got more difficult.

Summary of Results

Table 5: Optimized Implementation of Uninformed Search vs. Informed Search Heuristic 4

Level	Frontier	Time (s)	Memory Used (MB)	Solution length	States Generated
SAD1	BFS	0.029	3.72	19	80
SAD1	DFS	0.025	3.59	27	75
SAD1	A*	0.146	4.19	19	60
SAD1	Greedy	0.124	4.19	19	47
SAD2	BFS	14.467	692.72	19	635190
SAD2	DFS	0.022	3.59	25	86
SAD2	A*	0.146	4.19	19	60
SAD2	Greedy	0.110	4.18	19	75
SAfriendofDFS	BFS	1.185	72.13	8	89112
SAfriendofDFS	DFS	0.031	3.72	60	305
SAfriendofDFS	A*	0.282	13.15	8	1950
SAfriendofDFS	Greedy	0.060	4.56	8	130
SAfriendofBFS	BFS	0.048	5.12	3	1227
SAfriendofBFS	DFS	35.552	2015.72	981528	2953986
SAfriendofBFS	A*	0.067	8.00	3	40
SAfriendofBFS	Greedy	0.074	8.28	3	56
SAFirefly	BFS	18.261	2252.72	60	1961416
SAFirefly	DFS	24.539	4101.72	2517074	4089953
SAFirefly	A*	1.959	193.39	62	29629
SAFirefly	Greedy	1.361	80.13	206	15218
SACrunch	BFS	179.986	3840.72	N/A	6464388
SACrunch	DFS	9.218	4392.28	380992	1023377
SACrunch	A*	180.000	1929.75	N/A	2916068
SACrunch	Greedy	1.891	201.10	147	37380

Above is a summary of our best results for each frontier. Across the board, the best performance in terms of states generated came from the informed search algorithms. BFS will always find a move optimal solution, but A* and Greedy found a similarly short solution every time. However, when it comes to time spent and memory used, usually one of BFS or DFS were faster on the simpler levels. This is likely because a "dumber" technique can arrive at a solution relatively quickly, whereas no matter how simple a level is, the informed algorithms still do all their preprocessing regardless. Conversely, Greedy performed particularly well on the more complex levels. A major takeaway from this table is that in some situations, such as when trying to solve a simpler level, implementing BFS or DFS may be enough.

Discussion and Further Work

It looks like our best benchmark performance can be seen with heuristic implementation #4. With heuristics, you never really know if "the mess gets worse before it gets better" or if you are just digging yourself deeper and deeper into a ditch. In our case, it doesn't look like we quite cracked how to optimally handle all of the levels; most of our improvements optimized how the best-first search client solved one level but then compromised how it handled different levels.

If we had more time, we would have developed an additional heuristic, a simple implementation of Dijkstra's algorithm for calculating distance.