SC3000: Artificial Intelligence
Lab 1 Report: TDDB Team Armaan & Friends

Members:

Goel Armaan (U2123642H)

Implementation of task2, task3 and task3_alt

Chiam Da Jie (U2121233G)

Implementation of all helper functions, task1_UCS and task1_A*

Guo Yuan Lin (U2122626F)

Researching, testing and implementing heuristics, weight_tuning and task3_weighted_energy

We assert that everyone has contributed equally to the creation of this report.

Date of Submission: October 11, 2023

References

[1] A. Patel, "Heuristics" *Amit's A* Pages*, 2020. [Online]. Available: http://theory.stanford.edu/∼amitp/GameProgramming/Heuristics.html. [Accessed: 03-Oct-2023]

[2] S. Sengupta and B. An, "Exercise 7: Maze Runner." NTU SC1015, Singapore, Mar-2021.

1 Heuristics

Since we have the $\{x, y\}$ coordinates for each node, we can apply various heuristic functions to estimate the distance between a node and the goal to make use of informed search algorithms such as A*. The following coordinate-based functions will be used as heuristics for this assignment.

1.1 Euclidean Distance

Euclidean distance, also known as straight line distance, measures the length of the line that connects any 2 points on the $\{x, y\}$ plane. The formula is

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

1.2 Manhattan Distance

Manhattan distance assumes movement is possible in 4 directions (up, down, left, right) and calculates the number of units any 2 points are away on a standard square grid. The formula is

$$|x_2 - x_1| + |y_2 - y_1|$$

1.3 Chebyshev Distance

Chebyshev Distance assumes diagonal movement is possible and takes the value of the maximum difference over any of the axis values. Instead of adding the 2 differences like in Manhattan, we take only the maximum value here. The forumla is

$$max(|x_2 - x_1|, |y_2 - y_1|)$$

1.4 Octile Distance

Octile Distance is similar to Chebyshev as it also assumes diagonal movement. But instead of just taking the maximum, it also considers the minimum difference with a weight of $\sqrt{2}$. The formula is

$$max(|x_2 - x_1|, |y_2 - y_1|) + \sqrt{2} \cdot min(|x_2 - x_1|, |y_2 - y_1|)$$

1.5 Weight Tuning

From hereon, 'weight tuning' refers to the function implemented in *utils.py*. This function takes in the A* task to be executed along with the optimal result obtained from the UCS implementation of the same. It multiplies the heuristic value by weights ranging from 1 to 0, in steps of 0.01 for all 4 heuristics ($w * h(n)$). It crosschecks whether the distance returned is the same as the UCS result and then returns the weight with the least number of node expansions for each heuristic. The results and discussions of running weight tuning are mentioned in their respective task implementation sections.

2 Tasks

2.1 Task 1

Result:
Shortest path: S → 1363 → 1358 → 1357 → 1356 → 1276 → 1273 → 1277 → 1269 → 1267 → 1268 → 1284 → 1283 → 1282 → 1255 → 1253 → 1260 → 1259 → 1249 → 1246 → 963 → 964 → 962 → 1002 → 952 → 1000 → 998 → 994 → 995 → 996 → 987 → 986 → 979 → 980 → 969 → 977 → 989 → 990 → 991 → 2369 → 2366 → 2340 → 2338 → 2339 → 2333 → 2334 → 2329 → 2029 → 2027 → 2019 → 2022 → 2000 → 1996 → 1997 → 1993 → 1992 → 1989 → 1984 → 2001 → 1900 → 1875 → 1874 → 1965 → 1963 → 1964 → 1923 → 1944 → 1945 → 1938 → 1937 → 1939 → 1935 → 1931 → 1934 → 1673 → 1675 → 1674 → 1837 → 1671 → 1828 → 1825 → 1817 → 1815 → 1634 → 1814 → 1813 → 1632 → 1631 → 1742 → 1741 → 1740 → 1739 → 1591 → 1689 → 1585 → 1584 → 1688 → 1579 → 1679 → 1677 → 104 → 5680 → 5418 → 5431 → 5425 → 5424 → 5422 → 5413 → 5412 → 5411 → 66 → 5392 → 5391 → 5388 → 5291 → 5278 → 5289 → 5290 → 5283 → 5284 → 5280 → T
Shortest distance: 148648.63722140007

Nodes Expanded:
UCS – 5304; A* – 608 using Octile distance with a weight of 1.00

Weight Tuning:

Heuristic	Weight	Expanded	Reduction (%)
Euclidean	1.00	1225	76.9
Manhattan	0.87	626	88.2
Chebyshev	1.00	1958	63.1
Octile	1.00	608	88.5

Discussion:
The relaxed version of the NYC problem is a shortest path problem, which can be solved with Uniform-Cost Search as we have links of varying non-negative costs, and UCS guarantees optimality under this condition. We have used Python's *heapq* to implement the priority queue. The cost function is set to the distance between 2 nodes. We maintain a dictionary of path distances and a set of explored nodes. In each iteration, a node is popped from the queue. If the popped node already exists in the explored set, it is ignored. If the popped node is the goal state, then we stop. Otherwise, all the neighbours of the popped node are added to the priority queue if they do not already exist in the distance dictionary, or if the new distance is less than the existing one. To increase performance, we also implemented an A* version of Task 1 that uses Octile distance as the heuristic as it results in the least number of nodes expanded. The only difference between the UCS and A* versions is that when adding a node to the queue in A*, the priority is the sum of the total path and heuristic distance. $(g(n) + h(n))$. The NYC problem involves movements in all directions, in which case Euclidean should have been the best. But in real life, we do not have straight paths from every node to the other, so a deconstruction of axes is needed, which all the other heuristic functions do. Octile accounts for diagonal movement, and unlike Chebyshev, also assigns a

weight to the axis with the smaller difference, which could indicate why it has performed the best for this scenario. Manhattan also came close, but at a weight less than 1, most likely because it only assumes 4-directional movement.

2.2 Task 2

Result:
Shortest path: S → 1363 → 1358 → 1357 → 1356 → 1276 → 1273 → 1277 → 1269 → 1267 → 1268 → 1284 → 1283 → 1282 → 1255 → 1253 → 1260 → 1259 → 1249 → 1246 → 963 → 964 → 962 → 1002 → 952 → 1000 → 998 → 994 → 995 → 996 → 987 → 988 → 979 → 980 → 969 → 977 → 989 → 990 → 991 → 2465 → 2466 → 2384 → 2382 → 2385 → 2379 → 2380 → 2445 → 2444 → 2405 → 2406 → 2398 → 2395 → 2397 → 2142 → 2141 → 2125 → 2126 → 2082 → 2080 → 2071 → 1979 → 1975 → 1967 → 1966 → 1974 → 1973 → 1971 → 1970 → 1948 → 1937 → 1939 → 1935 → 1931 → 1934 → 1673 → 1675 → 1674 → 1837 → 1671 → 1828 → 1825 → 1817 → 1815 → 1634 → 1814 → 1813 → 1632 → 1631 → 1742 → 1741 → 1740 → 1739 → 1591 → 1689 → 1585 → 1584 → 1688 → 1579 → 1679 → 1677 → 104 → 5680 → 5418 → 5431 → 5425 → 5424 → 5422 → 5413 → 5412 → 5411 → 66 → 5392 → 5391 → 5388 → 5291 → 5278 → 5289 → 5290 → 5283 → 5284 → 5280 → T
Shortest distance: 150335.55441905273
Total energy cost: 259087
Nodes expanded: 23284

Discussion:
For Task 2, we have used UCS again. We still have links of varying non-negative costs, but the conditions under which they are added to the priority queue have changed. Instead of maintaining a dictionary of path distances, we now maintain a dictionary of energy costs. A set of explored nodes is not maintained anymore as we may have to revisit a node since both the distance and the energy cost could change. Although a set of $\{node, distance, cost\}$ pairs could be maintained, we found it to be an overhead as it did not make any improvement to the number of nodes expanded. For each neighbour that is processed, we calculate the total distance and energy up till that point. We only add a neighbour to the queue and update its priority as long as the energy cost is within budget and the neighbour does not exist in the cost dictionary or the new cost is less than the existing one. Since the priority is still based on the distance, we will always expand the shortest nodes first. This ensures that we can get the shortest path within a budget, as we are first prioritising distance, and then optimizing the budget. Initially, we implemented a naive version of this solution, where the only difference from Task 1 was an extra check to ensure new nodes are added to the queue if and only if adding them does not exceed the energy budget. It gave us a path with a distance of 150784 and an energy cost of 287931. Although this distance is very close, it is not the optimal solution.

2.3 Task 3

Result: As expected, the shortest path, distance and total energy cost are the same as the UCS implementation in Task 2. However, the number of nodes expanded has reduced.

Nodes Expanded:
6991 using Manhattan distance with a weight of 0.76; 862 using Manhattan distance with a weight of 0.89 (Non-optimal alternative); 5270 (Weighted-Cost)

Weight Tuning:

Heuristic	Weight	Expanded	Reduction (%)	Weight	Expanded	Reduction (%)
Euclidean	0.15	21967	5.7	–	–	–
Manhattan	0.76	6991	70.0	0.89	862	96.3
Chebyshev	0.05	22949	1.4	–	–	–
Octile	0.11	22466	3.5	–	–	–

Discussion:
Task 3 requires improving the performance of Task 2 using a suitable heuristic. In terms of implementation, the only difference from Task 2 is that when adding a node to the priority queue, the priority is the sum of the distance and the heuristic cost ($g(n) + h(n)$). We have used Manhattan distance with a weight of 0.76 as it has the best performance amongst all the heuristics with 6991 expansions. No other heuristic comes even close as they have almost no performance gains to offer. The Manhattan heuristic works so well for this task that with a weight of 0.89, it can guide the naive version discussed earlier in Task 2 towards the correct solution in just 862 node expansions. This is a 96.3% reduction from the original 23284. This alternate solution does not guarantee optimality for all paths, but it works surprisingly well for the specific problem statement given to us. For Task 1, Manhattan was optimal at a weight of 0.76, but with the energy constraint, it is already optimal at 0.89. Perhaps the Manhattan heuristic is a good measure for cost and is biased towards it, which works out in our favour. We also implemented a version (*task3_weighted_energy*) where $h(n) = 0.1 * energy\_cost$. It was able to find the correct path in 5270 node expansions, but more testing needs to be done to check whether this solution guarantees optimality.

3 Conclusions

In Task 1, we have seen how effective informed search can be when the right heuristic is used. The A* implementation with Octile distance as the heuristic expanded 88.5% fewer nodes than its uninformed UCS counterpart. Task 2 added an energy budget constraint to our shortest path problem. At first, it seemed like a simple additional check would do, but to implement it correctly, we had to flip our approach entirely. This also resulted in a much higher number of nodes being expanded, which is to say that even a simple constraint can make pathfinding more challenging. Yet again, a good heuristic can help improve performance, even when an additional constraint is at play. In Task 3, using Manhattan distance as the heuristic resulted in 87.7% less expansions than UCS. We also witnessed the classic trade-off between optimality and efficiency where a solution that does not guarantee optimality can achieve a solution that is close to optimal in much less time. Sometimes, getting a “good enough” solution quickly rather than the optimal solution slowly might be more desirable.