CS 404 Artificial Intelligence - Assignment 4

Alperen Yıldız, Sabancı University, 26758

Spring Term 2023

1 Color Maze Model

Four assumptions of basic search all hold in this case which are:

1. The world is static.
2. The world is discrete/discretizable
3. The world is observable
4. The actions are deterministic

1.1 States

In order to model this as a search problem let us first determine couple elements as a prerequisite:
Before describing the states let us first understand what each character denoting the grid:

1. 0 numbers indicating non-colored squares.
2. 1 numbers indicating squares that have been colored.
3. X characters indicating walls
4. S character symbolizing the cell occupied by the agent

States are, on the other hand, are game boards consisting of any number of characters among 1, 0, and X allowed by the size of the matrix and finally one character S indicating the position of the agent.

1.2 Successor State Function

Each of the successor are obtained by moving the agent in four cardinal directions up until it hits and obstacle, that is either the end of the game board or a wall, and replacing each and every 0 number crossed by the agent to 1.

1.3 Initial State

Initial state is any permutation of any number of X, any number of 0 and one S. One thing to notice in this configuration is that there should be no grids marked by 1, which indicates the agent has not crossed any grid yet.

1.4 Goal Test

State passes the goal test if there is no number 0 within the game board which indicates that the whole maze has been colored by the agent.

1.5 Step Cost Function

The step cost function is simply the number of grids the agent can move before it encounters a blockage which is either an obstacle or the end of the game board.

2 A* Search Implementation

A* implementation consists of a very similar implementation with one major difference: the heuristic function is used alongside of the cost function. The details of the heuristic function is written below

2.1 Heuristic Function

Consider that the game board is a m by n matrix and that m denotes the size of the matrix vertically whereas n denotes the size of the game board in horizontally.

If the agent is on the leftmost part of the game board, decide to move right and crosses no obstacles and no colored grids it will be able to color m-1 grids. If the agent is on the rightmost part of the game board and moves freely to the left with no obstacles it will be able to color m-1 grids once again. Therefore, we can generalize this as the agent being able to color m-1 grids at most if it moves horizontally.

Exact same thing holds with vertical motion, that is if the agent is on the top of the board and moves the bottom and encounters no obstacles it will be able to color n-1 grids and conversely the agent will be able to achieve coloring n-1 grids if it moves up freely while being initially located at the bottom of the board. Therefore we can generalize this as such: The agent can color at most m-1 grids if horizontally and n-1 grids if it moves vertically.

Therefore, the heuristic function should denote the number of grids the agent **could have** colored **minus** the actual number of grids being colored by the move. Thus, if moving to left will result in x grids being colored, the heuristic function should return (n-1) - x as the heuristic of moving in that direction. Since the heuristic function should return positive numbers, we take the theoretical number of maximum grids to be colored as m and n. Therefore the heuristic function can be shown in pseudocode as follows:

Algorithm 1 Heuristic Function

Require: $n > 0 \vee m > 0$
  $MAX\_COLORED\_VERTICAL \leftarrow m$
  $MAX\_COLORED\_HORIZONTAL \leftarrow n$
  $x \leftarrow$ movement results in how many grids to be colored
  if movement is horizontal **then**
    return $MAX\_COLORED\_HORIZONTAL - x$
  else
    return $MAX\_COLORED\_VERTICAL - x$
  end if

Since the heuristic function will never return a value that is greater than the actual cost of coloring the whole maze. This is because it only takes into account at most one line within the maze and regards the rest of the solution starting from each path as zero. Thus, heuristic function will ensure that the A* algorithm will never overlook lower cost paths in favor of more expensive solutions. In order to prove this, let us use mathematical induction:

Let M be an NxN matrix
Let C*: N $\rightarrow$N be the optimal cost of coloring the whole matrix
Let h: N $\rightarrow$ N be the heuristic function

The base case is if the matrix is 1x1 where the cost of coloring it is 0. In this case, the heuristic function will also return 0. Therefore, $f(0) = C^*(0)$ Then to continue with the induction, let us come up with the inductive hypothesis within a (N+1) x (N+1) matrix: $f(N+1) = C^*(N+1)$ assuming the NxN subsection of the matrix has been optimal, and N+1 by N+1 matrix can only be created by adding a margin of 1 to two neighboring sides of the matrix. Since the heuristic function will never overestimate the added margins, induction proves the heuristic function never overestimates the cost and is therefore indeed admissible.

3 Implementation

The algorithm has been implemented in Python programming language alongside with libraries Numpy for matrix implementation and memory_profiler for extracting precise information about the memory usage. UCS implementation is given below as an example.

```
def ucs(filename) -> None:
    """
    Uniform cost search implementation

    :param filename: name of the file
    :raises NoSolutionError: raises an exception if no solution is found
    :returns: None
    """
    closed = list()
    frontier = UCS_Frontier()

    start = Board()
    start.read_file(filename)
    initial_node = Node(
        start,
        cost = 0,
        movement = None,
        parent = None
    )
    frontier.put(initial_node)

    while not frontier.isEmpty():
        n = frontier.pop()

        if n.state.goal_test():
            solution(initial_node, n)
            return

        for s in SUCC(n):
            s.cost += n.cost

            if s.state not in [
                    elem.state for elem in frontier.data_structure.queue] and s.state not in
                    elem.state for elem in closed]:
                frontier.put(s)
```

```
            temp = list()
            while (not frontier.isEmpty()):
                popped = frontier.pop()
                if (s.state.state == popped.state.state and s.cost < popped.cost):
                    popped.cost = s.cost
                    popped.parent = n
                temp.append(popped)

            for elem in temp:
                frontier.put(elem)

        closed.append(n)

    raise NoSolutionError
```

A separate Python script is used to call both algorithms on 15 distinct matrices. The script uses regular expressions and redirection of the output to collect information about the performance metrics. Matrices are divided into 3 groups, namely: easy, medium and hard. The difficulty level is determined by how many successors each matrix is able to produce. More child states produced by each state is theorized to correspond with higher levels of difficulty.

4 Performance

As expected, the A* search was significantly faster and used less memory. This is probably caused by the fact that A* is using a heuristic function which means it has to expand on less nodes. Less nodes decrease the space complexity of the implementation and also means the goal state will be achieved faster which translates to faster execution.

Table 1: Performance Table Per Matrix

Uniform Cost Search

	easy 1	easy 2	easy 3	easy 4	easy 5	medium 1	medium 2	medium 3	medium 4	medium 5	hard 1	hard 2	hard 3	hard 4	hard 5
cells	9.000	9.000	9.000	9.000	18.000	18.000	18.000	17.000	16.000	21.000	19.000	33.000	21.000	20.000	27.000
memory	32.699	32.723	32.918	32.676	32.910	32.555	32.371	33.391	33.484	33.148	33.273	27.527	33.238	33.238	33.363
time	0.026	0.026	0.026	0.026	0.073	0.217	0.485	0.894	16.138	1.796	0.883	26.676	2.214	6.097	2.854
distance	10.000	10.000	10.000	10.000	30.000	108.000	204.000	389.000	2693.000	1565.000	198.000	53234.000	720.000	742.000	1617.000
expanded nodes	1.000	1.000	1.000	1.000	3.000	7.000	13.000	20.000	198.000	34.000	21.000	293.000	41.000	112.000	50.000

A* Search

	easy 1	easy 2	easy 3	easy 4	easy 5	medium 1	medium 2	medium 3	medium 4	medium 5	hard 1	hard 2	hard 3	hard 4	hard 5
cells	9.000	9.000	9.000	9.000	18.000	18.000	18.000	17.000	16.000	21.000	19.000	33.000	21.000	20.000	27.000
memory	32.891	32.594	32.418	32.648	32.504	32.574	33.129	32.398	33.199	32.816	33.078	20.473	33.184	33.109	33.324
time	0.025	0.027	0.025	0.027	0.050	0.182	0.426	0.762	11.316	1.689	0.670	27.395	1.583	6.938	2.743
distance	10.000	10.000	10.000	10.000	30.000	108.000	204.000	389.000	2693.000	1565.000	198.000	53234.000	720.000	742.000	1617.000
expanded nodes	1.000	1.000	1.000	1.000	2.000	6.000	12.000	19.000	152.000	34.000	17.000	306.000	32.000	124.000	49.000

This trend is significantly more visible when we visualize the data using graphs. First of all, in order to check

the assumptions about the difficulty levels, graph 1 has been created. Graph 1 shows the average number of total child nodes in each difficulty level. This shows that a measurable and objective criterion for the difficulty level has been used. The data in graph has been scaled using logarithm function for better visualization.

Secondly, Graph 4 shows that UCS mostly consistently took more time to execute. Since the time complexity of A* algorithm is lower than of uniform cost search, these results are not surprising however this shows two important points. First, this shows that the theoretical construction of the A* algorithm is correct, in that, the heuristic function is in fact designed properly; in other words the heuristic function is admissible in practice as well. If the heuristic function had theoretical or implementation problems, A* algorithm would consistently be outperformed by uniform cost search algorithm since the heuristic function would force it to forsake lower cost paths (or it would not find a solution at all). Secondly, the A* algorithm not only works better in theory but also in practice. Therefore, the translation of both algorithms into Python programming language has been done correctly.

Having said that, the graph 3 also shows a lower space complexity for A* search. This is probably due to the fact that A* search expands on less nodes which means that the Python script has to keep less nodes in memory for both in frontier and closed data structures.

Last but not the least, Graph 2 shows that my hypothesis about correlation between the expanded nodes and time is indeed correct since when their normalized values are plotted in a table they show a stark correspondence. Therefore, A* algorithm would be significantly more scalable in terms of both memory and time consumption.