CS404-Assignment1

Alper Kaan Odabaşoğlu

March 2023

1 Python Implementation Of Color Maze Puzzle Solution

https://colab.research.google.com/drive/1FhLoE7AQlGIT_TxQrRFin_w32jwg7nqA?usp=sharing

2 Model as Search Problem

A problem can be modeled as a search problem by examining 5 properties which are States, Successor states, initial state, goal state and the path cost. In the subsections of this section, states of color maze puzzle will be identified, legal actions that can be taken by agent will be emphasized, successor state function for gathering successor states of particular state will be defined. Besides, initial state, goal test function for goal state and the step cost function for gathering arc cost for particular state transition will be discussed.

2.1 States

States are the configurations of snapshots of the maze field with 1 to N colored cells (if there is N cells that can be reachable by the agent) that can be achieved by the legal actions of the agent.
So a state can include;

1-Current Position of Agent ('S')
2-Positions of Walls ('X')
3-Uncolored cells ('0')
4-Colored cells ('C')

2.1.1 State Examples

0	0	0	0	0	X	0
0	X	X	X	0	X	0
0	X	X	X	0	X	0
0	X	X	X	0	X	0
0	X	X	X	0	X	0
S	0	0	0	0	0	0

State Instance 1

S	0	0	0	0	X	0
C	X	X	X	0	X	0
C	X	X	X	0	X	0
C	X	X	X	0	X	0
C	X	X	X	0	X	0
C	0	0	0	0	0	0

State Instance 2

As we can see these are 2 state examples for color maze puzzle. 'X' represents the positioning

of the walls in maze, '0' represents the uncolored cells that are eligible to be colored, 'C' represents colored cells and 'S' represents the position of agent in the field. Actually, the second state in this example is a successor state (reachable from state 1 doing a legal action) of state 1.

2.1.2 Legal Actions

Legal actions of the agent are selecting a direction (left, right, up, down) and if the next cell in that direction is not a wall, color all the cells in that direction up to encountering a wall.
Input:
CurrentState: Current state of the field agent in

```
LegalActionFinder(CurrentState)
    Set = empty set
    row, col = getCoordinates of Agent ('S')
    if(row-1 ≥ 0 and state[row-1][col] is not Wall)
        add action "up" to Set
    if(row+1 ≤ len(state) - 1 and state[row+1][col] is not Wall)
        add action "down" to Set
    if(col-1 ≥ 0 and state[row][col-1] is not Wall)
        add action "left" to Set
    if(col+1 ≤ len(state[0]) - 1 and state[row][col+1] is not Wall)
        add action "right" to Set
    return the Set
```

Output:
Set: the set that contains actions can be performed by agent in current state

2.2 Successor State Function

Successor state function is characterized by the state that the agent in and the possible legal actions that can be taken by the agent. This function returns feasible states that can be reached by the legal actions taken on the state agent in. So, if our agent is in the state s and the legal action set that can be implemented on this state is L, it returns an array S including the snapshot of states that is gathered by performing each l (legal action) in L (legal action set) on s.

So we can say that the function successor takes one input, which are the state that the agent is currently in (s) and it can use another function called LegalActionFinder(CurrentState) to find the legal actions can be performed by the agent in given state

Input:
s: Current state of field Agent in

```
Succ(s)
    R = empty return array
    t = deep copy of s
    L = LegalActionFinder(s)
    for each action a in L
        s' = state gathered from performing a on t
        add s' to R
    return R
```

Output:
R: Set that contains all states gathered by implying legal actions on the initial state s.

2.3 Initial State

The Initial state for this problem is a snapshot of field instance which has a colored cell that the agent starts in on the field.

2.4 Goal Test

For this problem, the goal test is basically based on checking whether all the cells that are eligible to be colored, colored or not. So, this test basically looks for the field if all the cells that the agent can reach is colored. This can be represented as GoalTest(state) function.

Input:
state: Current state of field Agent in
GoalTest(state)
    s = deep copy state
    change agents position on field from 'S' to 'C' which is colored cell
    if s == goal state return True else False

output:
Bool value that is True if the state is goal state, false o.w.

Goal state in this part represents the state where all the cells that can be eligible to be colored is colored. It is gathered by coloring all the cells that can be colored in initial state given to the problem. In other words changing all the '0's and 'S' value with 'C' which represents the colored phenomena.

2.5 Step Cost Function

In general step cost function is characterized by the state that agent is in, action that is taken by the agent and the intended state. So basically, step cost function measures how many units of cost is taken by transition to one state to another with particular action.

For this problem, step cost function is characterized by how many cells that the agent is covered in particular action. For acquiring the cost of the action, one can look at the displacement of the agent. Thus, agents coordinates could be taken into consideration. First, initial positions are gathered and then depends on the displacement in row and column, the function can return the cost as a positive value ($cost \geq \epsilon > 0$).

Input:
initState: state before agent apply the action
finalState: final state after agent apply the action on initial state
CostFinder(initState, finalState)
    row1 and col1 = initState coordinates of agent
    row2 and col2 = finalState coordinates of agent
    if row1-row2 != 0
        return absolute value of row1-row2
    else
        return absolute value of col1-col2

Output: positive displacement value of agent which represents number of colored cells for a particular state transition which is the cost

3 Extension of Search Model

The previous model indicated above is enough for Uniformed-Search Algorithms such as Uniformed Cost Search (UCS) algorithm. However, it should be extended for the heuristic based algorithms such as A* Search Algorithm that we will cover below. In the A* Search, instead of performing actions based on g(n) which represents the real cost value to reaching state kept in node n (current node) from initial state, the algorithm finds a heuristic function h(n), sum it up with the real cost value to reaching state kept in node n (current node) from initial state g(n) to extract a f(n) value from them. This f(n) = h(n) + g(n) where n is a particular node, indicating the estimation of the cost from current state the agent in kept in node n to the goal state.

In UCS algorithm, the program always choose and get the node with lower g(n) which is real

cost value of initial state to destination. Whereas the A* search algorithm always choose the nodes with lower f values. Nodes that are remarked above are keeping meta-data such as state, cost, parent state that the current state gathered from, heuristic value and the f value. So the search problem specified above should be extended such that a relatively qualified heuristic function adopted to the problem.

3.1 Heuristic Function Selection

There can be multiple heuristic functions for particular search problem. However, some of them ensures the optimality and selecting one among these can give better performance than them. As we discussed in the lecture admissiblity implies optimality. Thus, selecting a heuristic function which is admissible can imply optimality and increase the performance of the program.

For the color maze puzzle that we are working on, the heuristic function determined as a function which gets a particular state and returns the number of non-colored cells (count of '0' cells) from the domain of this state.

3.1.1 Heuristic Function Pseudo-code

Input:
s: Current state of field Agent in

Heuristic(s)
    count = number of uncolored cells in s
    return count

Output:
count: Number of uncolored cells for this particular state

3.2 Proof Of Admissibility

As a condition for heuristic function, admissibility ensures that heuristic function do not overestimate the real cost to reach to the goal state from a given node. For instance, if reaching to a goal state requires real cost h*(n), the return value of heuristic function h(n) obliged to be less than or equal to the h*(n) where n is the current node agent be in.
h(n): estimated value by the heuristic function from node n's state to goal state
h*(n): real cost needed to reach goal state from node n's state
Admissibility ensures that $h(n) \leq h^*(n)$.

For color maze puzzle our heuristic function is admissible since it does not overestimate the real cost to reach to the goal state from the current state agent in. As we always return the count of uncolored states for particular state the agent should at least trace that many block for reaching the goal state. For instance if there is 10 uncolored cell in our maze, the agent should at least travel 10 many steps to color all the cells and finalize the mission.
In more general notion, if there is a path from node n to goal state g and if we have k many cells left to be colored for reaching goal state from this node n, in the best scenario the agent colors k many cells to reach the goal state. Because our heuristic value is $h(n) = k$, and the real cost of reaching the destination state is $h^*(n) \geq k$ (least and best value for h*(n) is k), the h(n) will always be less than or equal to h*(n). Therefore $h(n) \leq h^*(n)$ and h is an admissible function.

4 Performance Analysis

4.1 Results Table

4.1.1 UCS Results

Instance Number	Number Of Cells	Difficulty	Total Distance Traveled	Total Number Of Expanded Nodes	CPU time (seconds)	Memory Consumption (MB)
1	54	Easy	53	21	0.002061256	9.76322e-5
2	46	Easy	45	32	0.002093331	9.76345e-5
3	44	Easy	43	18	0.001131368	9.76361e-5
4	48	Easy	49	39	0.002645683	9.76377e-5
5	33	Easy	32	20	0.001295304	9.76398e-5
6	63	Normal	64	157	0.016436656	9.76398e-5
7	52	Normal	67	436	0.065095567	9.76398e-5
8	54	Normal	86	1546	0.453041808	9.77216e-5
9	55	Normal	66	468	0.087996188	9.77218e-5
10	61	Normal	79	2307	0.936163139	9.77218e-5
11	60	Hard	69	7575	16.93339905	0.000106115
12	63	Hard	74	3588	2.740861599	0.000102416
13	59	Hard	73	7394	12.93957383	0.000104955
14	53	Hard	69	7730	18.12724033	0.000104994
15	67	Hard	79	5751	5.789493807	0.000102154

CPU: INTEL CORE I5 9300-H CPU @ 2.40 GHZ
RAM: 16GB
OS: Windows 11 Home Single Language 64bit
COMPILER: VS Code

4.1.2 A* Search

Instance Number	Number Of Cells	Difficulty	Total Distance Traveled	Total Number Of Expanded Nodes	CPU time (seconds)	Memory Consumption (MB)
1	54	Easy	53	10	0.00123116	0.0001007
2	46	Easy	45	11	0.00116377	0.0001006
3	44	Easy	43	9	0.00099711	0.0001006
4	48	Easy	49	19	0.00219415	0.0001006
5	33	Easy	32	9	0.00093080	0.0001006
6	63	Normal	64	48	0.00681514	0.0001006
7	52	Normal	67	303	0.07196520	0.0001007
8	54	Normal	86	1431	0.79698765	0.0001005
9	55	Normal	66	272	0.07117969	9.97371e-5
10	61	Normal	79	1310	0.70118212	9.97273e-5
11	60	Hard	69	2019	2.43401420	9.97195e-5
12	63	Hard	74	1376	1.03118039	9.97195e-5
13	59	Hard	73	3776	7.31682831	9.97307e-5
14	53	Hard	69	5522	17.5265689	0.00010257
15	67	Hard	79	2511	2.54592020	0.00010172

CPU: INTEL CORE I5 9300-H CPU @ 2.40 GHZ
RAM: 16GB
OS: Windows 11 Home Single Language 64bit
COMPILER: VS Code

Both of the CPU Time as well as Memory Consumption acquired by runing each instances 30 times and getting average of this 30 results.

4.2 Notion Behind the Difficulty Levels

The 3 difficulty levels (easy, normal and hard) are arranged based on 3 different options respectively. The first option emphasizes that the agent can straightly go to the destination state without recoloring any cells that it colored previously. So there is a direct path to the goal state for the agent in the first option. This option is valid for all easy benchmark instances. Thus, in easy benchmarks, the agent can go to goal state without coloring previously colored cells. The scientific notion behind the first option is to minimize node extension count for our both algorithm and facilitate the coloring process for the agent. In the second option, besides the straight path, agent can trace back the colored cells to reach the destination state. In other words, agent should re-pass the cells that it passed over before to reach the goal state. The scientific reason behind this option is to increase the extended node count with re-passing choice to diminish the convenience rate. This option is valid for the normal benchmark instances. In all normal benchmark instances, agent can re-pass the previously colored cells to color all cells. In the final option which follows a more comprehensive approach, the agent can both re-trace the cells that it previously passed and keep track of the cyclic paths. The scientific contextualization of generating this option is increasing the extension count more with cyclic paths and re-tracing choice.This option is valid for the hard difficulty level instances.

4.2.1 Easy Benchmark Example

S	0	0	0	0	0	0	0	0	0
X	X	X	X	X	X	X	X	X	0
0	0	0	0	0	0	0	0	X	0
0	X	X	X	X	X	X	0	X	0
0	X	X	X	X	X	X	0	X	0
0	X	0	0	0	X	X	0	X	0
0	X	X	X	0	X	X	0	X	0
0	X	X	X	0	0	0	0	X	0
0	X	X	X	X	X	X	X	X	0
0	0	0	0	0	0	0	0	0	0

As you can see, there is just one direct path that the agent can trace to reach the destination state in easy benchmarks.

4.2.2 Normal Benchmark Example

S	0	0	0	0	0	0	0	0	0
X	X	X	X	X	X	X	X	X	0
0	X	0	X	0	0	0	X	X	0
0	X	0	0	0	0	X	0	0	0
0	X	0	X	X	0	X	0	0	0
0	X	0	X	X	0	X	0	0	0
0	X	0	X	X	0	X	0	0	0
0	X	0	X	0	0	X	0	0	0
0	X	0	X	0	0	0	0	0	0
0	0	0	X	X	X	X	X	0	0

As you can see in normal benchmark instances, the agent should go back the same way it previously covered to reach the destination state. So there is no direct route to the goal state. Thus, the extended node count is more than easy instances (it trace back the same route more than one time).

4.2.3 Hard Benchmark Example

0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	X	X	0
X	X	X	X	0	0	0	0	0	0
X	X	X	X	0	X	X	X	X	X
0	0	0	0	0	X	X	X	X	X
0	0	0	0	0	0	0	0	0	S
X	X	X	X	0	X	X	X	X	X
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	X	X	X	0
X	X	X	X	X	0	0	0	0	0

Table 1: Matrix H2

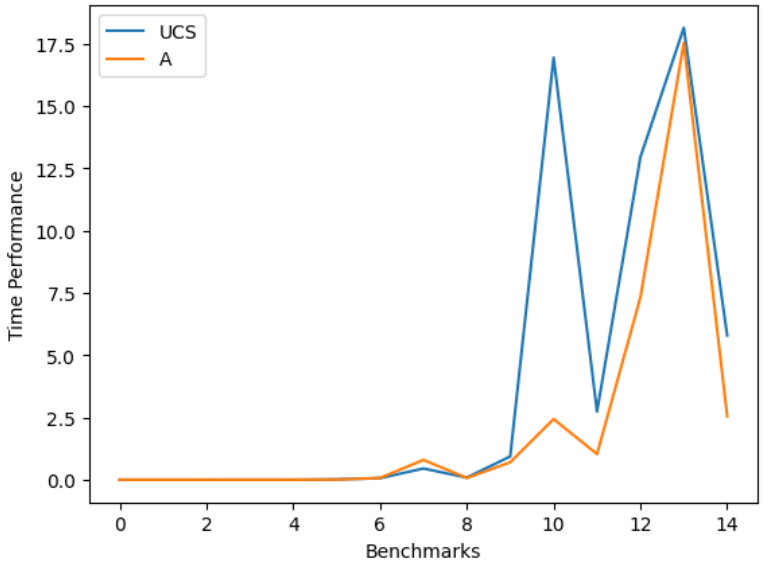
As it can be seen from this hard instance, there are some cyclic paths (cycles) to be colored as well as trace-back paths (paths to be re-passed again) exist. Thus the expansion count of node is increased more.

4.3 Comparison of A* Search and UCS

4.3.1 Scalibility of Algorithms

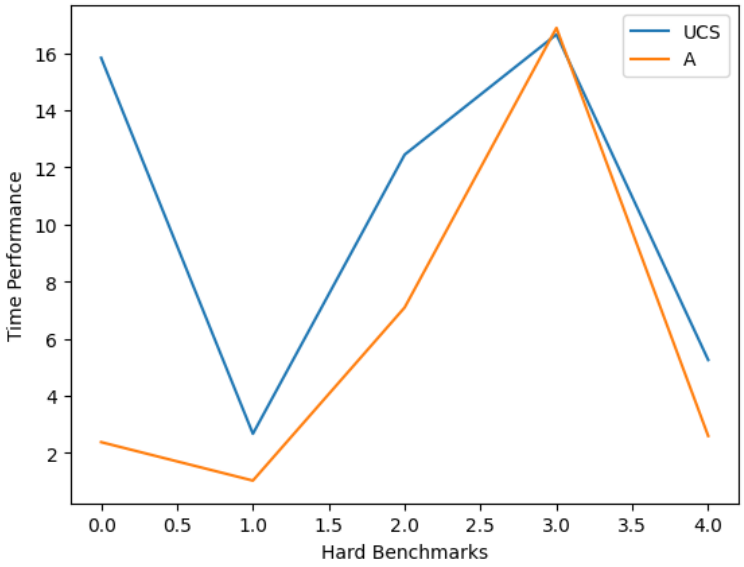
In terms of time, as we can observe from the table, even though the algorithms give scalable results for easy instances in terms of CPU time consumption, when we increase the hardness, we can see that the time consumption increase a lot. For instance, when we compare the best average time consumption of UCS which is 3^{rd} instance with the worst average time consumption which is 14^{th} instance in table, we can see that the time consumption increases more than 16000 times. Even though both time consumption taken from 10x10 grid maze, there is 16000 time more consumption in one of them. Same argument is valid for A* Search as well since the best average time which is 5^{th} instance is 18829.5755 times less than the worst average time consumption which belongs to the 14^{th} instance. Considering these results, even in the same dimensions, when the difficulty level as well as node extension count increases, the time complexity increases too much. As we discussed in the lecture, if the heuristic function is not admissible, both A* search and UCS have exponential time complexity. However, in our example, our heuristic function is admissible as we proved in previous sections. So, the efficiency of A* search is relatively better in terms of time complexity of UCS as the node expansion count is diminished (The decrease in expanded node count can be seen from the table passing from UCS to A* search). Nevertheless, in terms of scalibility, even though A* gives relatively better results for these instances, when we increase the dimensions into NxN from 10x10, both of the algorithms seems to be not scalable for particular input instances.

The plot of time consumption indicates this scalability in terms of time well.



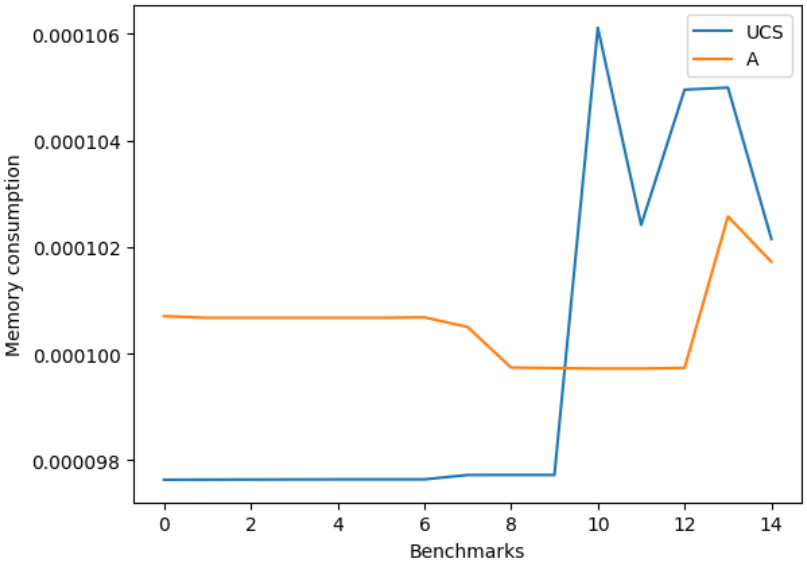
As it can be observed, the time consumption increases rapidly when the instances going through the hard difficulty instances. This is just for the illustration of 10x10 grid mazes. However, when the cell count increased to NxN, the time taken will also increase a lot where the issue of scalability will rises. Apart from that, as we can see, in the instance number 10, A* consumes nearly 8 times less time. Which will arise relatively high performance for bigger instances such as NxN mazes.

The surprising result in terms of time can be seen in the 14^{th} instance where the average time consumption are very similar in terms of magnitude of seconds. When we run each hard instance 30 times and plot the average time consumptions:



We can observe that A* can give higher result for this instance as well. This is surprising as we willing to observe better results for A* compared to UCS in general. The reason behind this abnormally for our expectation can be the structure of instance and the implemented python code. As it is indicated in the table the expanded node count do not decrease a lot for instance 14 compared to other hard difficulty instances.Probably in this instance, the general structure of our heuristic function do not led to the reopening of previously closed nodes. Also, more for loops in python implementation for A* search can increase the time consumption when the decrement of expansion rate of node is not high. Apart from this instance, in general, the A* search algorithm with an admissible heuristic function gives better results compared to UCS in terms of CPU time consumption due to the decrease of expanded node count and increased efficiency in terms of decreasing the re-tracing phenomena for paths.

In terms of memory, fluctuations can be observed from instance to instance. However, the memory consumption is relatively stable compared to the CPU time consumption. Even though it is not directly appropriate to interpret the space complexity by just looking at the 10x10 grid maze instances, we can say that, in general, the space complexity related with and proportional to the maximum number of nodes that can be stored in Frontier (priority queue). From the observations, we can see that the A* search algorithm follows a more stable memory consumption path compared to the increasing behavior of UCS.



As we can see, there is a sudden increase going through the hard difficulty instances in UCS. However, the A* is relatively stable compared to UCS. This can be related to the decreasing node expansion count performance of the heuristic function. There are some decreases in terms of memory consumption going through hard instances. This is surprising, but probably it is related with the compilation time of the code. Also up to the 8^{th} instance, A* seems to be more than the UCS. However, it is probably because the expanded node counts are less in this range for benchmark instances. Thus, in this rage, using more variable for A* may be affecting this.
In terms of scalability, both of the algorithms give relatively better results for 10x10 instances. However, when the search space increased to NxN grid maze, both algorithms can become non-scalable. Nevertheless, the A* search algorithm can perform way better in bigger instances since it decrease the expanded node count a lot for some instances such as the hard instances in table.

4.3.2 How They Explore the Search Space

Both of the algorithms keep the states from search space in a node and use priority queue to select the next state. However as we discussed in the previous sections as well as in lecture, UCS choose the least real cost node that can be chosen and A* choose least f-value (estimate of cost to getting to the goal state passing through a certain node) node. So, different from UCS, A* algorithm take the heuristic value into account and explore search space with this estimate value in addition to the real cost value to reaching this node. In general, they expand nodes based on this rule. However expansion counts differ from one algorithm to other as we can see from the table. The A* algorithm expand less nodes due to the impact of heuristic function. As we discussed in the lecture, the UCS algorithm expand the search space with consecutive circles, however, the A* search algorithm expand the search space elliptical latitudinally towards the target state, discovering less redundant nodes.