DAT405 Assignment 8: Rule-based AI – Group 4

Student 1 - Dimitrios Koutsakis (8 hrs)
Student 2 - Bingcheng Chen (8 hrs)

May 20, 2023

Problem 1

(a) In BFS, each level of the search tree represents one iteration, and BFS explores all nodes at each level before moving to the next level. Since we are given that the shortest path between the initial state and a goal is of length 'r'. In the worst case scenario, before reaching the goal, each node explored has 'd' number of children nodes and the goal node is the last children node explored.

At the first level, there is only the initial state. At the second level, there can be at most 'd' nodes. At the third level, there can be at most 'd^2' nodes , and so on.

Therefore, the maximum number of iterations required in BFS can be calculated by summing up the number of nodes at each level.

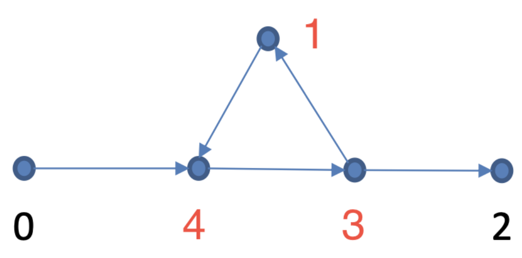
$$\begin{aligned} Maximum\ number\ of\ iterations\ required &= 1 + d + d^2 + d^3 + ... + d^r \\ &= \frac{d^{r+1} - 1}{d - 1} \end{aligned}$$

(b) At the first level, since there is only the initial state, thus the unit of memory needed is 1. At the second level, there can be at most 'd' nodes explored, and for each path it needs 2 units of memory. At the third level, there can be at most 'd^2' paths , and the units of memory needed for each path is 3.

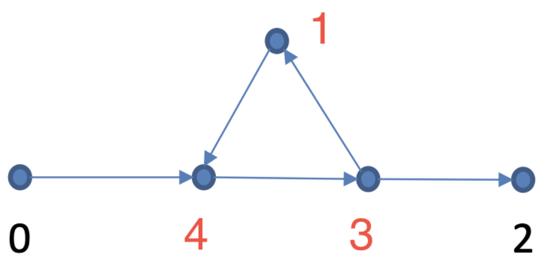
Therefore, the maximum amount of memory required for BFS can be calculated as follow:

$$Maximum\ amount\ of\ memory\ required = 1 \times d^0 + 2 \times d^1 + 3 \times d^2 + 4 \times d^3 + ... + (r + 1) \times d^r$$

Problem 2



a) scenario 1.



b) scenario 2.

Figure 1: DFS will get trapped into a loop and never reach to the goal.

DFS explores the depth of a tree first, this means DFS tends to traverse along one path until it reaches the end or encounters a dead end before backtracking to explore other paths. One main problems associated with DFS is that it could get trapped in infinite loops if there are cycles in the graph. In our case, since we chose the smaller label when confronting with a tie, thus in two scenarios from Figure 1 DFS will get trapped into a loop and never reach to the goal.

To avoid getting trapped in infinite loops, we can keep track of visited nodes.

- Step1: Initialize an empty set to store visited nodes.
- Step2: Start DFS from the initial node, when visiting a node, mark it as visited. Before exploring a neighbor node, check if it has already been visited. If it has, skip exploring that node and move on to the next unvisited neighbor.
- Step3: Continue until all nodes are visited.

Problem 3

(a) The search space as a tree expanded for a lowest-cost-first search can be seen in Figure 7, the optimal customised textbook is [book1, book2].

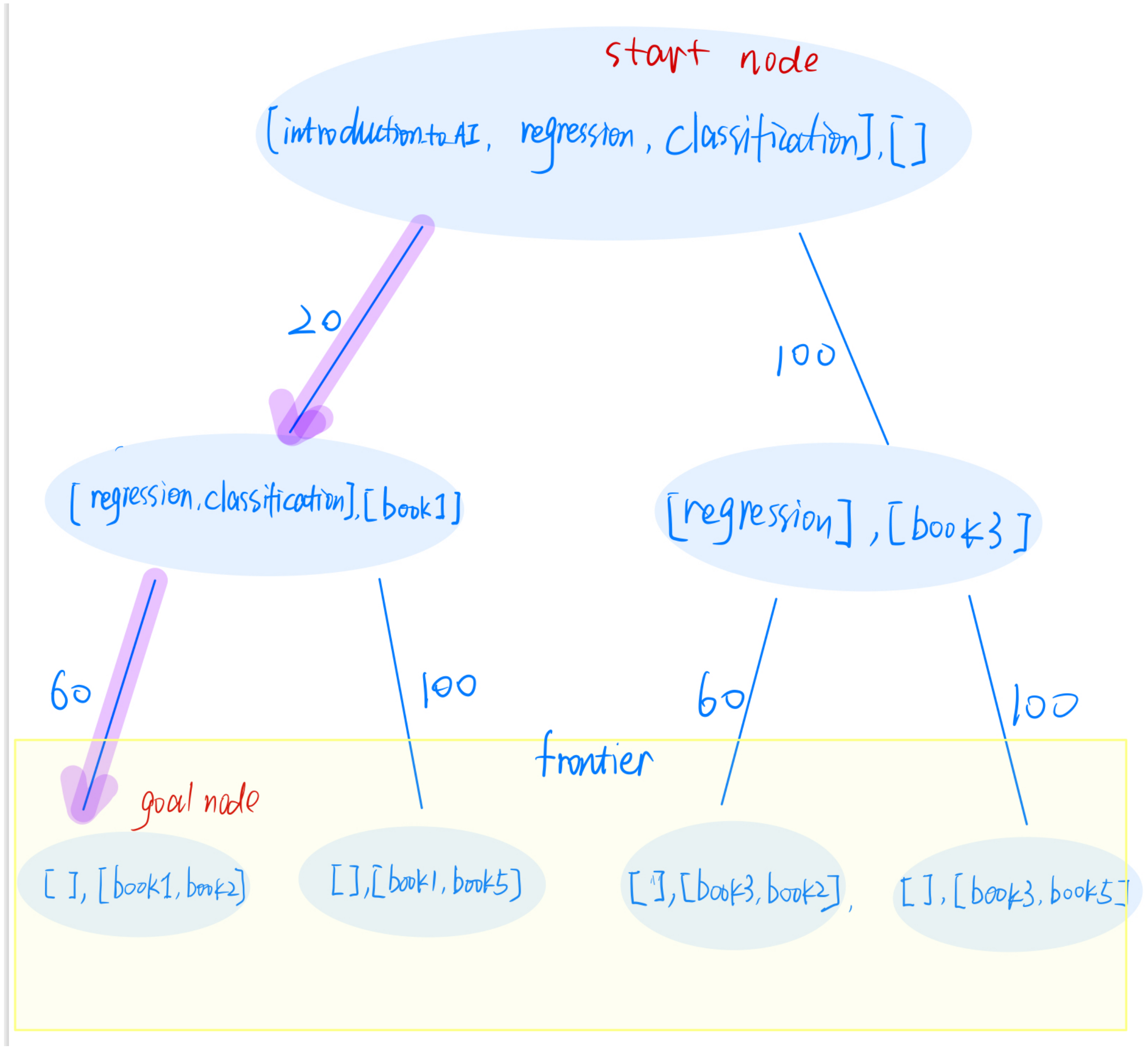


Figure 2: The search tree.

(b)

The heuristic function indicates how close a node is to the goal, and for an admissible heuristic function , h(n) should always less than or equal to the actual cost of a lowest-cost path from node n to a goal, Our goal is to cover all the topics in the list. Therefore an adequate heuristic value could be the size of the uncovered topics list in the node. In that case the smaller the size of the list the closer we are to the goal while the goal node has heuristic value 0.

Problem 4

(a) Assumptions:
- Use Manhattan distance as the G cost and H cost, no diagonal step.
- The top left corner of each node is the node's G cost.
- The top right corner of each node is the node's H cost.
- The center value of each is the node's F cost, Note that F cost is the sum of G cost and F cost.
- In the case of tie the algorithm prefers the path stored first.
- Initialize two lists: the open list and the closed list. The open list contains nodes that are to be explored, while the closed list contains nodes that have already been visited.

- Step 1: choose to explore node (4,3).
  open list = [(4,2),(5,3),(4,4)]
  closed list = [(4,3)]

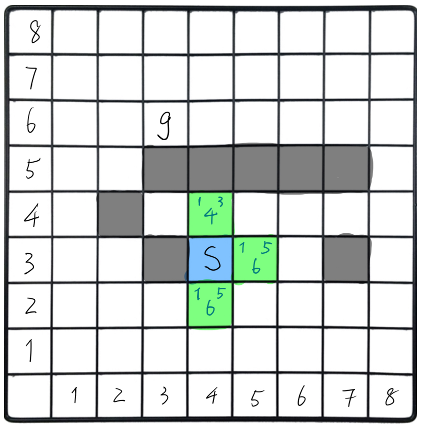


Figure 3: step 1

- Step 2: choose to explore node (4,4).
  open list = [(4,2),(5,3),(3,4),(5,4)]
  closed list = [(4,3),(4,4)]

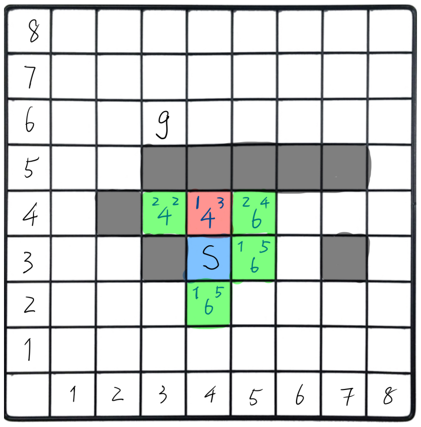


Figure 4: step 2

- Step 3: choose to explore node (3,4).
  open list = [(4,2),(5,3),(5,4)]
  closed list = [(4,3),(4,4),(3,4)]

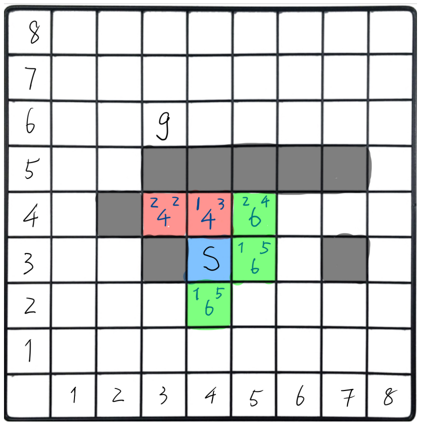


Figure 5: step 3

- Step 4: since there is a tie, choose to explore the first least cost node, which is (4,2)
  open list = [(5,3),(5,4),(3,2),(4,1),(5,2)]
  closed list = [(4,3),(4,4),(3,4),(4,2)]

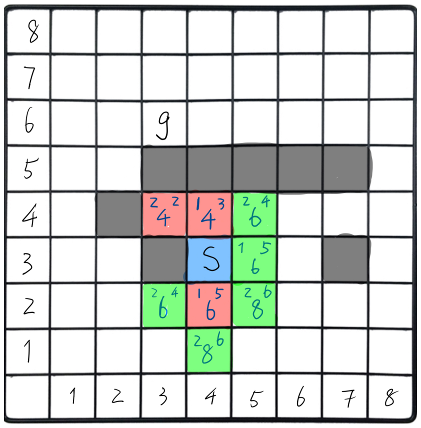


Figure 6: step 4

- Step 5: There is a tie, choose to explore the first least cost node, which is (5,3)
  open list = [(5,4),(3,2),(4,1),(5,2),(6,3)]
  closed list = [(4,3),(4,4),(3,4),(4,2),(5,3)]

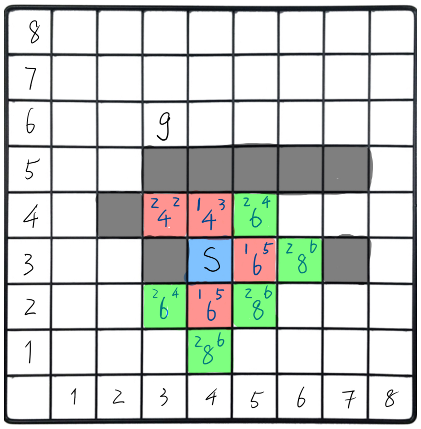
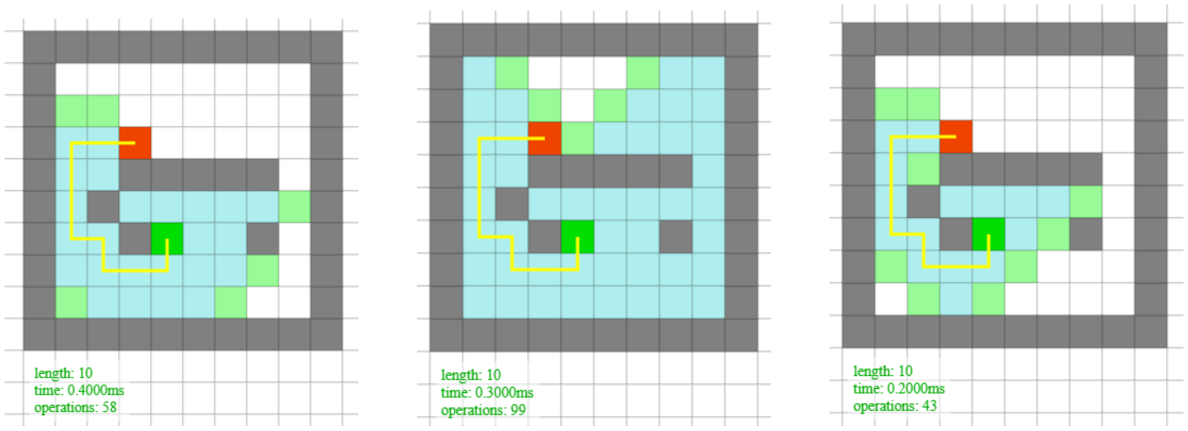


Figure 7: step 5

(b)



A* BFS Best-first Search

Figure 8: Comparisson of A*, BFS, Best-first Search with Manhattan distance heuristic

A*, BFS, and Best-first search are all search algorithms, but they differ in exploration strategies:

- BFS explores nodes in a breadth-first manner, it explores all nodes at each level before moving to the next level.
- Best-first search explores nodes based on an evaluation function that estimates how close a node is to the goal, for each step it chooses to explore the most promising node.
- The A* algorithm combines the benefits of both Dijkstra's algorithm and greedy best-first search.

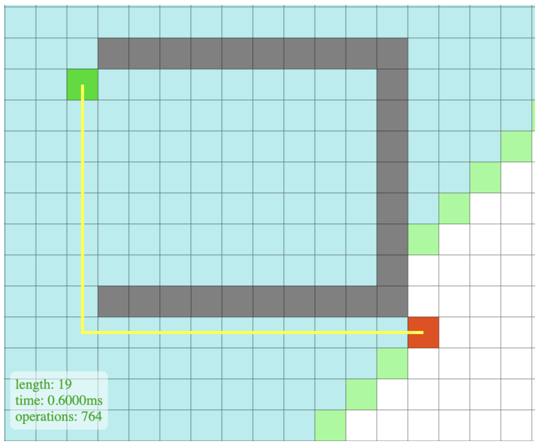
In this case, all three algorithms found the goal at the length of 10, However, by comparing the three algorithms we found that Best-first Search performed the best requiring only 43 iteration to find the shortest path to the goal, followed by A* that needed 58 iterations and finally BFS with 99 iterations.

This was expected since algorithms that utilise heuristics to evaluate each step and perform the locally optimal move each time have an advantage in such open layouts since they don't diverge from the goal. On the other hand BFS works by searching each neighboring node from a certain depth making it cover a larger area before it finds the best path.

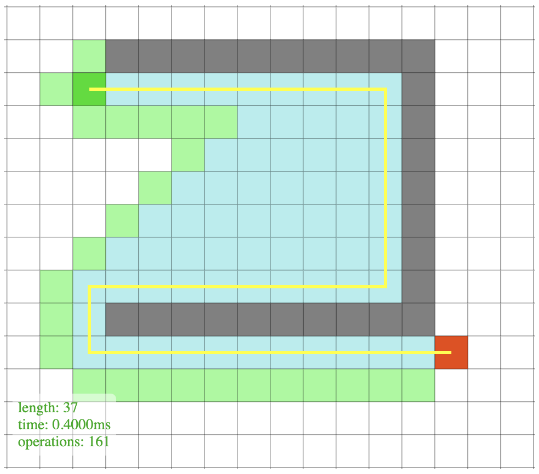
(c)

Greedy best-first search explores nodes based on an evaluation function that estimates how close a node is to the goal. but it can be misled by the evaluation function and may prioritize nodes that seem closer to the goal but actually lead to longer paths. In order to find a layout that favours BFS over Best-first Search we had to get rid of some of the openness of the map by placing barriers, especially next to the goal node. The reason that we want to trick the Best-first Search to take the paths that lead as close the the goal as possible and at the last step to find a barrier requiring to backtrack and find another solution. As a result BFS found the goal at the length of 19 as opposed to the 37 found by Greedy Best-first Search as is shown in Figure 9.

However, BFS explores all nodes at each level before moving to the next level. It guarantees that the shortest path from the start node to any other node will be found first. This makes it ideal for finding the shortest path in an unweighted graph, although it may require exploring a larger number of steps. In our case, BFS uses 764 iterations to find the goal while the greedy best-first search uses 161 iterations.


length: 19
time: 0.6000ms
operations: 764

(a) Breadth-first search, length = 19


length: 37
time: 0.4000ms
operations: 161

(b) Greedy best-first search, length = 37

Figure 9: Map layout where BFS performs better than Best-first Search