DAT405 Introduction to Data Science and AI, 2021 –2022,

Assignment 8:

Maële Belmont	12h
Lenia Malki	12h

1) The branching factor 'd' of a directed graph is the ***maximum number of children*** *(outer degree) of a node in the graph. Suppose that the shortest path between the initial state and a goal is of length 'r'.*
→ check this website https://thealgorist.com/Algo/GraphTheory/BFS

a) What is the maximum number of BFS iterations required to reach the solution in terms of 'd' and 'r'?

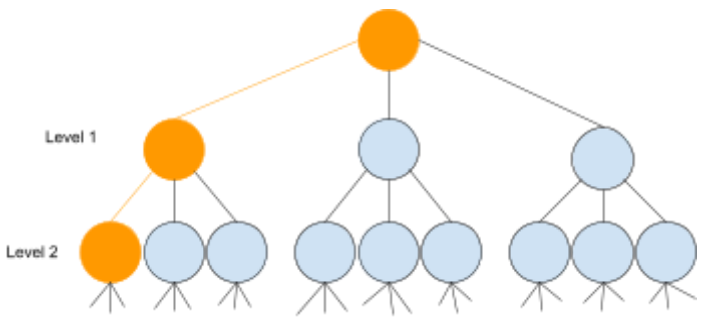
Breadth-first search iterates through each level, starting from the left most node to the right most node. In the worst case scenario, each node at each level has a number of children equal to that of the branching factor, meaning that the only leaf nodes exist at the last level. Thus, we are searching d-nodes at each level n until $n = r$. The maximum number of iterations are thus the sum of d-nodes from n = 0 to n = r as shown below:

$$\sum_{n=0}^{r} d^n$$

b) Suppose that storing each node requires one unit of memory. Hence, storing a path with k nodes requires k units of memory. What is the maximum amount of memory required for BFS in terms of 'd' and 'r'?

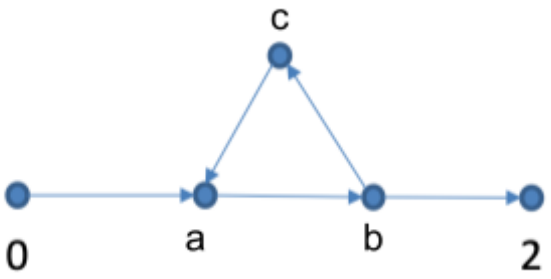
If each node requires 1 memory unit and we imagine the worst case scenario described above in terms of d, the path to a specific node would require $n + 1$ memory units where the extra 1 originates from the root node. This goes for all nodes and thus, we end up with:

$$\sum_{n=0}^{r} (n + 1)d^n$$



Path with 3 nodes for node ending at level 2

2) *Take the following graph where 0 and 2 are respectively the initial and the goal states. The other nodes are to be labelled by 1,3 and 4. Suppose that in case of a tie, the DFS method takes the path with the smallest label of the last node. Show that there exists a labelling of these three nodes, where DFS will never reach the goal! What can be added to DFS to avoid this situation?*



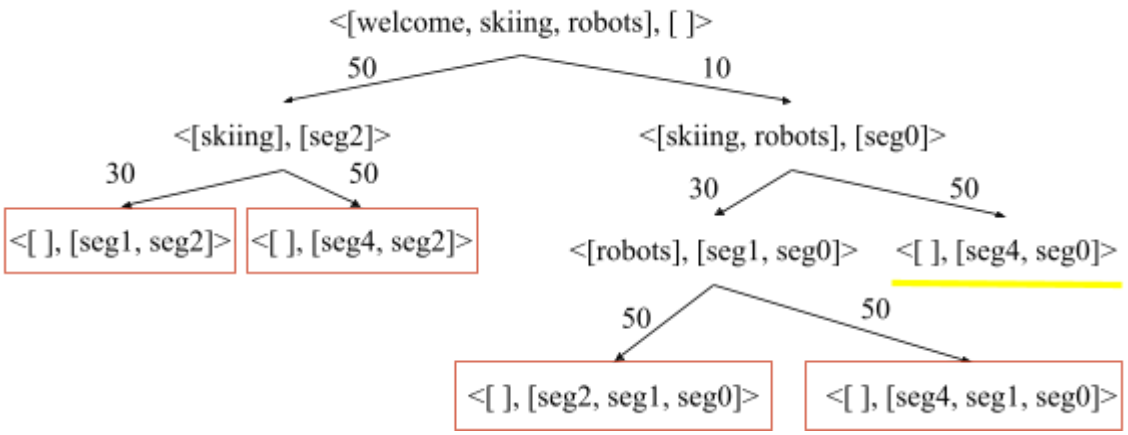
Let's name the unlabeled nodes a, b and c for reference. There is no alternative direction to go but forward from node 0. This has us ending up at node a. Likewise, there is no other path to take but to move forward to node b from node a. Once at node b, we now have two options. Either we move forward to node 2 or node c. We are thus at a tie and the algorithm will choose the path with the smallest label. If we set node c = 1, the algorithm will end up in a loop, never reaching the goal node 2. Labelling of node a and b does not affect the chosen path as the algorithm has to visit these nodes either way. Node a could be labelled 3 or 4, likewise for node b. An infinite loop could be avoided if we were to keep track of visited nodes whenever a tie occurs.

3) This question investigates using graph searching to design video presentations. Suppose there exists a database of video segments, together with their length in seconds and the topics covered, set up as follows:

Segment	Length	Topics covered
seg0	10	[welcome]
seg1	30	[skiing, views]
seg2	50	[welcome, artificial_intelligence, robots]
seg3	40	[graphics, dragons]
seg4	50	[skiing, robots]

We define a node as a pair ⟨*To_Cover, Segs*⟩

- where *Segs* is a list of segments that must be in the presentation and *To_Cover* is a list of topics that also must be covered, but is not covered yet by *Segs*. Hence for a valid node, none of the segments in *Segs* cover any of the topics in *To_Cover*.
- The children of a node are obtained by first selecting a topic from *To_Cover*, adding a segment to *Segs* which covers this topic and finally deleting any topic from *To_Cover covered by this segment*. For example, the children of the node ⟨*[welcome, robots],[]*⟩ , assuming that *welcome* was selected, are ⟨*[],[seg2]*⟩ and ⟨*[robotos],[seg0]*⟩ .
- Thus, each arc adds exactly one segment but can cover one or more topics. Suppose that the cost of the arc is equal to the time of the segment added.
- The goal is to design a presentation that covers all of the topics in a list named *MustCover*. The starting node is ⟨ *MustCover,[]*⟩ , and the goal nodes are of the form ⟨*[], Presentation*⟩ for some list *Presentation*. The cost of the path from a start node to a goal node is the time of the entire presentation. Thus, an optimal presentation is a shortest presentation that covers all the topics in *MustCover*.
    - (a) Suppose that the goal is to cover the topics [*welcome,skiing,robots*] and the algorithm always selects the leftmost topic to find the neighbors for each node. Draw (by hand) the search space as a tree expanded for a lowest-cost-first search until the first solution is found. This should show all nodes expanded, which node is a goal node, and the frontier when the goal was found.



The shortest presentation is the one underlined in yellow, using seg0 and seg4 with a total length of 60. The frontier is indicated by the red rectangles.

- (b) Give a non-trivial heuristic function *h* that is admissible. [*h(n)=0* for all *n* is the trivial heuristic function.]

n = node

$$h(n) = \frac{segment\ length}{number\ of\ topics\ to\ cover\ covered\ by\ the\ segment}\ for\ n\ containing\ a\ topic\ to\ cover$$

$$h(n) = 0\ otherwise$$

Let's demonstrate how this heuristic function works:

For the child node selection, there are two options with the following heuristic values:

Left: $h(< [skiing],\ [seg2] >) = 50/2 = 25$

Right: $h(< [skiing,\ robots\ ],\ [seg0] >) = 10/1 = 10$

→ Since 10 < 25, the node $< [skiing,\ robots\ ],\ [seg0] >$ is the best option.

For the grand-child node selection, there are two options with the following heuristic values:

Left: $h(< [robots\ ],\ [seg1,\ seg0] >) = 30/1 = 30$

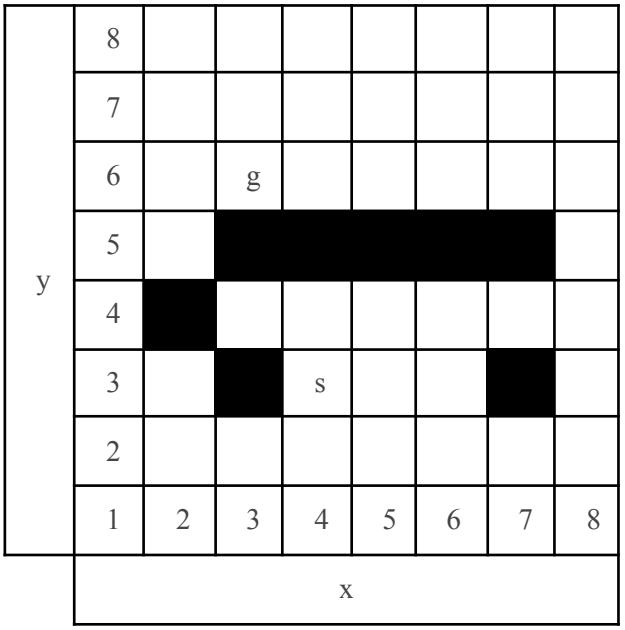
Right: $h(< [\ ],\ [seg4,\ seg0] >) = 50/2 = 25$

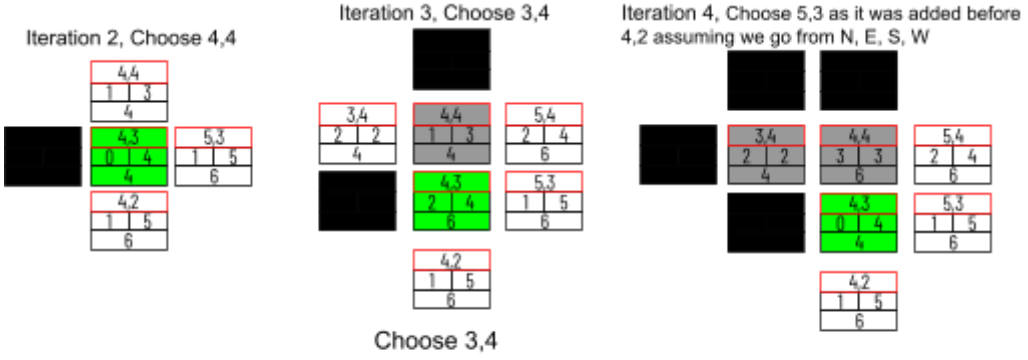
→ Since 25 < 30, the node $h(< [\ ],\ [seg4,\ seg0] >$ is the best option and that is the optimal segments composition (goal).

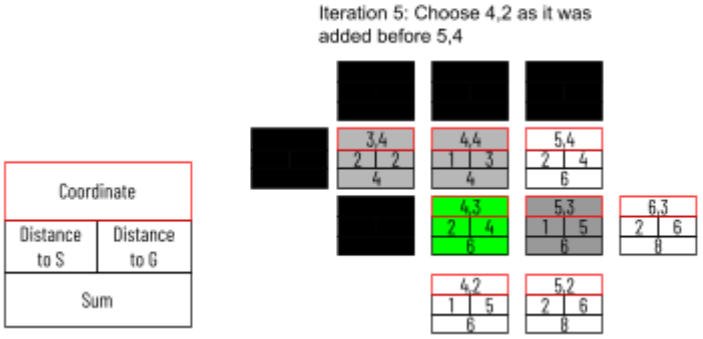
This heuristic function is admissible because the cost estimated to reach the goal is the lowest cost for every node in the path.

4) Consider the problem of finding a path in the grid shown below from the position *s* to the position *g*. A piece can move on the grid horizontally or vertically, one square at a time. No step may be made into a forbidden shaded area. Consider the **Manhattan distance as the heuristic.**

a) Write the paths stored and selected in the first five iterations of the A* algorithm, assuming that in the case of tie the algorithm takes the **shortest path with the smallest lexicographical index**.

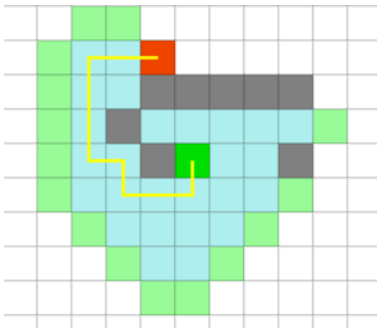
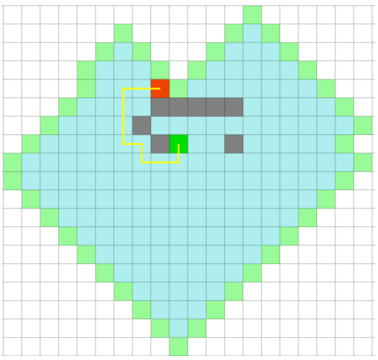
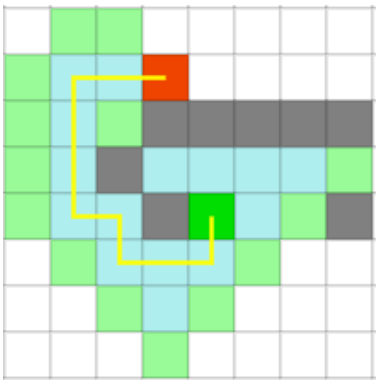






Iteration 1 starts at (x,y) = (4,3) with distance from S = 0 and distance to G = 4. We store nodes in order N, E, S and W. At a tie, the oldest node is chosen.

b) Try to solve this problem by the software in http://qiao.github.io/PathFinding.js/visual/ Use Manhattan distance, no diagonal step and compare A, BFS and Best-First-Search. Write a short description about your observation. How does each of these methods reach the solution? Why? Which one is faster?*

A* Operations: 71 Length: 10	BFS Operations: 364 Length = 10	Best-First-Search Operations: 48 Length = 10
		

BFS is quite insufficient compared to A* and Best-First-Search as seen by the total number of operations made. We can also see that Best-First-Search explores the space the least among the algorithms, resulting in the lowest number of operations. BFS is not optimal to use when there is room for a lot of exploration and thus a lot of paths to iterate through. As the Best-First-Search does not take into account the cost of each path taken, it does not make such extra comparisons as the A' algorithm, resulting in lower operations and a much more greedy algorithm.