Assignment 8: Rule Based AI and Search

Neha Devi Shakya (CY-Tech Exchange Student): 12 Hours

Sarvesh Meenowa (CY-Tech Exchange Student): 12 Hours

1. [2p] 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 length ‘r’

a) What is the maximum number of Breadth-First Search (BFS) iterations required to reach the solution in the terms of '*d*' and '*r*'?

BFS (Breadth-First Search) is a traversing algorithm that starts at the starting/source node and travels the graph layer by layer, investigating the child nodes (nodes directly connected to the source node) and then proceeds to child nodes in the next layer.

The graph must be traversed breadthwise, as the name suggests:

1. Traverse horizontally.
2. Visit all the current layer nodes.
3. Move to the next layer.

So, if we consider the worst-case situation, where each node has the highest number of child nodes, each iteration quickly becomes each node. The number of growing nodes thus follows a geometrical development, with the branching factor 'd' serving as the common ratio and the first number of nodes serving as the scale factor a, which in our example is 1:

$$t_n = d * t_{n - 1}$$

where, $$t_1 = a$$

i.e. $$t_n = a * d^{n}$$

In our case, the variable n is r, the length of the graph so $$t_r = a * d^{r}$$

As a result, the sum of this progression (the number of iteriations) can be written as follows:

$$\sum_{l=1}^{r}a * d^{l} = a (\frac{1- d^{r}}{1- d}) = \frac{1- d^{r}}{1- d}$$

b) Suppose that storing each node requires one unit of memory and the search algorithm stores each entire path as a list of nodes. 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*'?

Each node will have the highest number of child nodes in the worst-case scenario. Thus, storing node paths entails storing the geometric progression for each node at step r with k units of memory. The total number of paths stored is determined by the current layer (path length) and the number of nodes at that level (number of unique paths).

For every iteration, d new paths arise. If d = 2, at the first split, one path becomes 2. At the second split, two paths become 4which then becomes 8. The results of these splits are equal to the number of nodes in the layer (d^l where l is the current layer). All these nodes have equal length to the initial node. We can then multiply depth times width to calculate the memory cost.

To calculate the tree's width, we find the number of nodes at the layer that the goal is on. Say $d = 2$ and $r = 3$, then the total amount of nodes will be 15, and the width will be $2^3 = 8$.

```
                o                   ^
            o               o       | total splots is 3 (r)
        o       o       o       o   | path length is 4 nodes (r+1)
      o   o   o   o   o   o   o   o v
```

Since the height is 3, 4 nodes are required to build each path. The total memory required is then $8 * 4 = 32$.

So, in general, we can calculate the maximum amount of memory with:

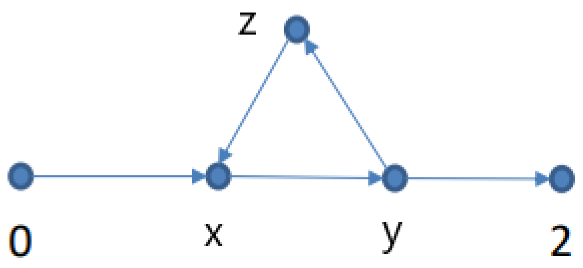
$$(width\ of\ tree) * (height\ of\ tree + 1)$$

In terms of d and r, we can write this as:

$$d^r * (r + 1)$$

---

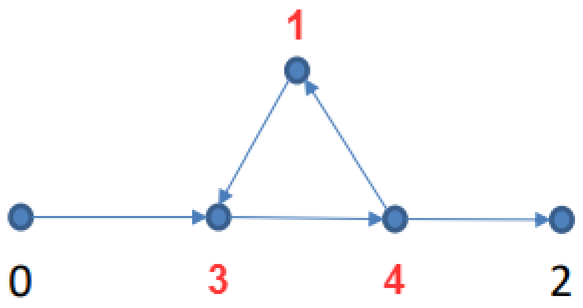
2) [1p] Take the following graph where 0 and 2 are the initial and the goal states. The other nodes are to be labelled by 1, 3 and 4

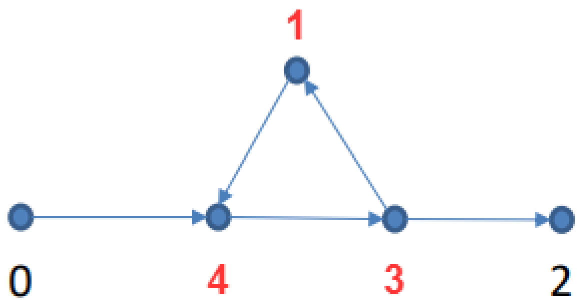


Suppose that we use the Depth First Search (DFS) method, and in the case of a tie, we choose the smaller label. Find all labelling of these three nodes, where DFS will never reach the goal! Discuss how DFS should be modified to avoid this situation?

The algorithm starts by going from 0 to x, the second node. From x, there is only one possible search direction; thus, it searches without regard to labels (this is true for both x and z). However, there are two options from y: proceed to z or 2. Since x is arbitrary, we can now investigate which values of $z \in \{1, 3, 4\}$ render the solution impossible. Since the algorithm prefers labels of smaller size, and since all other searches are independent of labels, it will always choose z if $z = 1$, which will result in an infinite loop.

Thus, there are two scenarios in which the DFS approach will never achieve its purpose and become trapped in an indefinite loop. The following two labellings for the three nodes (x, y, z) pose problems:





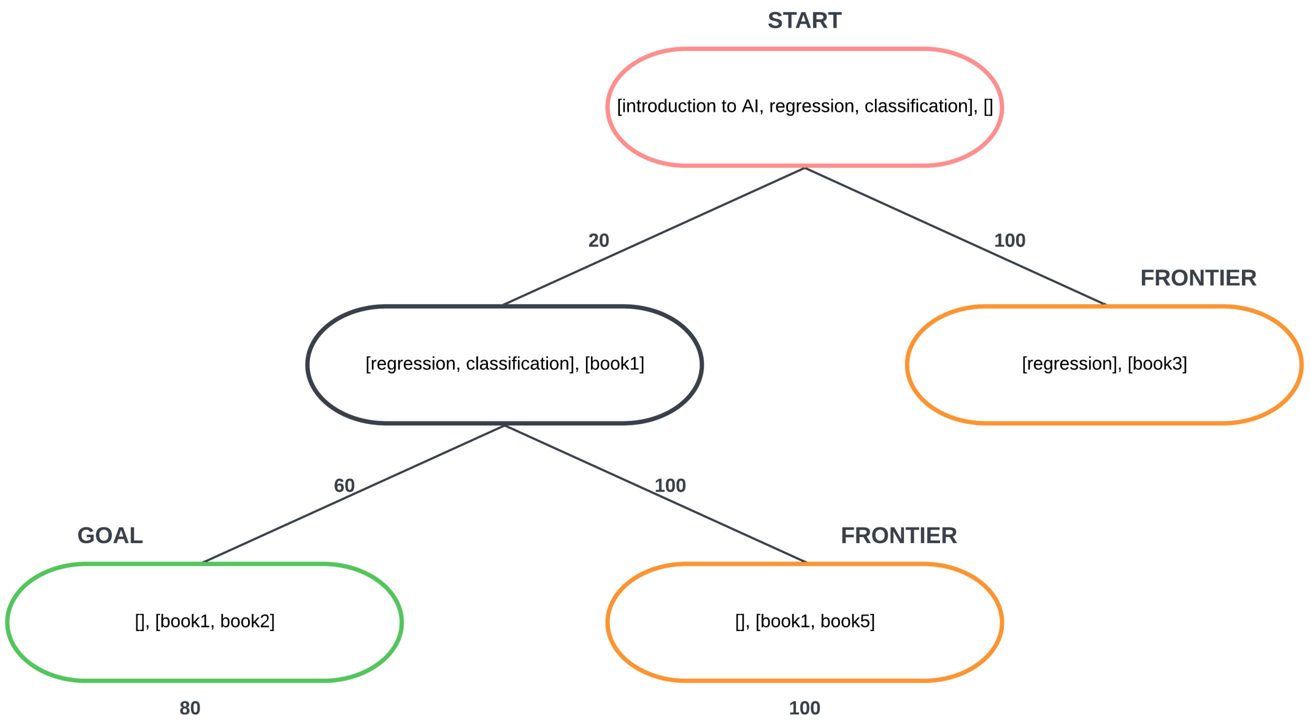
The first option is to remember previously visited nodes and detect a recurring path pattern, and break the path choice at a specified threshold to exit the loop (if possible). The second option is to utilise Iterative Deepening Depth-First Search (IDDFS), a modified version of DFS. This method is a means of executing "DFS" in the style of "BFS," but with increasing depths, so the search cannot go beyond this threshold.

3) [2p] A publisher allows teachers to “build” customised textbooks for their courses by concatenating the text from different books in their catalogue. The catalogue contains the following books, together with the number of pages that the book contains and the topics covered in that book

Books in Catalogue	Number of Pages	Topics Covered
book1	20	[introduction_to_AI]
book2	60	[regression, classification]
book3	100	[introduction_to_AI, search, classification]
book4	80	[machine_learning, neural_networks]
book5	100	[regression, classification]

a) Suppose a teacher requests a customised textbook covering the topics [introduction_to_AI, regression, classification]. The algorithm always selects the leftmost topic when generating child nodes of the current 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

The search space until the first solution is shown below for a lowest-cost-first search. The nodes are written as {Topics Covered, Books in Catalogue}, and the node marked as GOAL is the goal node found.



The "introduction_to_AI" topic is handled initially in the child nodes with "book_1" and "book_3". "book_1" is chosen as it is the cheapest (20 pages). As a result, the other node associated with book 3 is a "frontier" node. Both "book_2" and "book_5" address the last two concepts, "regression" and "classification." Since Book 2 is less expensive (60 pages) than Book 5 (100 pages), it is chosen which is our goal node. As a result, the node containing "book_5" is also considered a "frontier" node.

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

Our goal is to construct a heuristic function that estimates how long it will take to finish the topics in "Topics Covered" given a node. Given {[regression, classification], []} our heuristic function should estimate the time required to cover the topics of regression and classification.

However, an admissible heuristic function should never overestimate the actual cost.

The heuristic function h is admissible if $$h(p) \leq cost(p')$$

As a result, our heuristic function should determine the shortest time to cover the topics in "Topics Covered", ensuring that we never underestimate the actual cost. The straight-line distance in the shortest-path problem is very similar to this.

For all the records in the database, the minimum length/number of topics covered is the quickest way to cover a topic. The heuristic value in a graph attempts to define the significance of this node's value. The number of topics remaining to analyse in a node can tell us if we are on the right track or not. Thus our heuristic value will be the one that allows us to reduce the ultimate cost.

Let b be the set of all books included in the table, pg be the number of pages and allow h to be dependent on the path p. Let $t(p)$ be the set of topics in the last node of path p.

We have a heuristic function

$$h(p) = \begin{cases} min\{pg\} & if\ t(p) \neq \varnothing \\ 0 & otherwise \end{cases}$$

where $min\{pg\}$ is the minimum number of pages in the collection of books b.

Here, h indicates that if at least one topic remains unfinished, the cost is at least the cost of the least expensive book that has not yet been used. The heuristic function is zero only if all topics have been completed, which is permissible because one topic has not been completed, and at least one more book must be added.

4) [3p] Consider the problem of finding a path in the grid shown below from position s to 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. Each square is denoted by the XY coordinate. For example, s is 43 and g is 36. Consider the Manhattan distance as the heuristic. State and motivate any assumptions that you make

a) Write the paths stored and selected in the first five iterations of the A* algorithm, assuming that the algorithm prefers the path stored first in the case of a tie

The table represents the first five iterations of the A* algorithm. Within the possible steps column, the **xy** indicates the selected next node based on the lowest h or in the order stored (starting north going clockwise).

iter	step	possible steps	cost	dist from s	h(p)	dist from s + h(p)
0	43	(*44*, 53, 42)	(4, 6, 6)	0	4	4
1	43 -> 44	(53, 42, 54, *34*)	(6, 6, 6, 4)	3	5	4
2	44 -> 34	(*53*, 42, 54)	(6, 6, 6)	2	2	4
3	43 -> 53	(*42*, 54, 63, 52)	(6, 6, 8, 8)	1	5	6
4	43 -> 42	(*54*, 63, 52, 41, 32)	(6, 8, 8, 8, 6)	1	5	6
5	44 -> 54	(63, 52, 41, *32*, 64)	(8, 8, 8, 6, 8)	1	5	6

The stored paths are as follows:

iter	stored paths	selected step
0	{(43)}	43
1	{(43),(43,44)}	43 -> 44
2	{(43),(43,44),(43,44,34)}	44 -> 34
3	{(43),(43,44),(43,44,34),(43,53)}	43 -> 53
4	{(43),(43,44),(43,44,34),(43,53),(43,42)}	43 -> 42
5	{(43),(43,44),(43,44,34),(43,53),(43,42),(43,44,54)}	44 -> 54

b) Solve this problem using the software in http://qiao.github.io/PathFinding.js/visual/. Use Manhattan distance, no diagonal step and compare A*, BFS and Best-first search. Describe your observations. Explain how each of these methods reaches the solution. Discuss the efficiency of each of the methods for this situation/scenario

The table shows a summary of the numbers exctracted when running the problem.

	number of operations	length
A*	71	10
BFS	362	10
Best-First-Search	48	10

A*

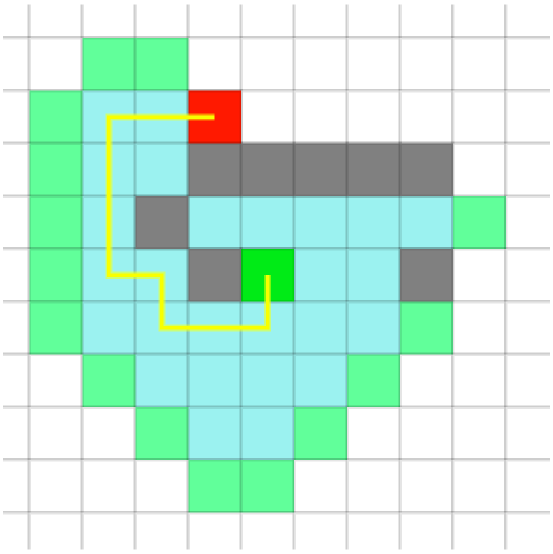
A* knows where the goal is and searches for a way to get there. It "rates" each state based on how it got there and how long it takes to reach the target via "manhattan"-distance. A* overcomes the problem by placing all newly detected nodes in a priority queue. The priority queue is ranked according to the sum of:

'g(x)' is the shortest path to that node. The manhattan distance heuristic for that node, 'h(x)' in our case.

When the values in our priority queue are equal, we can choose to prioritise the node discovered first or the lowest 'h(x)' by convention.

A* is effective because if the start of an optimal path is found, it is likely to continue down that path as the value of 'h(x)' decreases. 'g(x)' grows as 'h(x)' shrinks, while 'f(x)' remains constant if the direction is directed towards the goal. As 'f(x)' is still first in the queue, the path is examined.

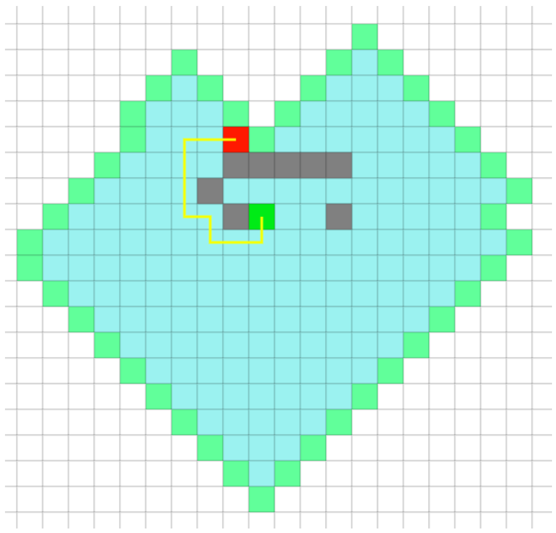
Because it "incorrectly" examines the corner to the upper left of the start, the software reaches the target after '71' operations. We talked over how in greater detail in the iteration example above, but in short, because A* is blind to obstacles, it will try straight-line approaches. If those fail, it will only look at values with a higher 'f(x)'.



Operations needed to find goal: `71`

Breadth First-Search

Breadth first-search **naively** searches for a goal by amassing more states recursively. It works for each state s by visiting all nearby nodes that have not been visited yet, resulting in more operations overall. However, when the search space is large but the objective is thought to be close breadth-first search is considered a good algorithm.



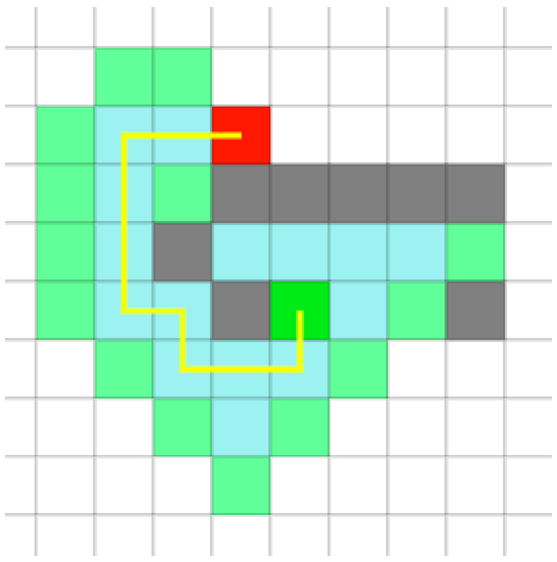
Operations needed to find goal: `362`

Best First-Search

Best-first search is a search method that examines a graph by expanding the most promising node chosen according to a specified rule.

The rule set is the shortest Manhattan distance. We have a similar frontier to A*.

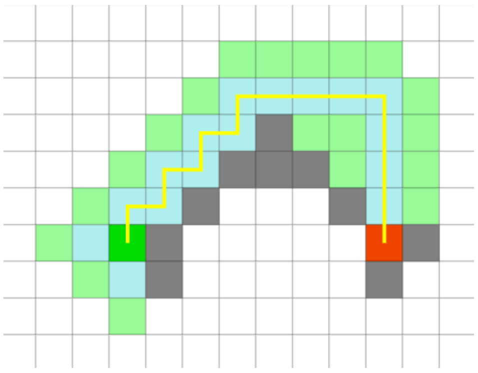
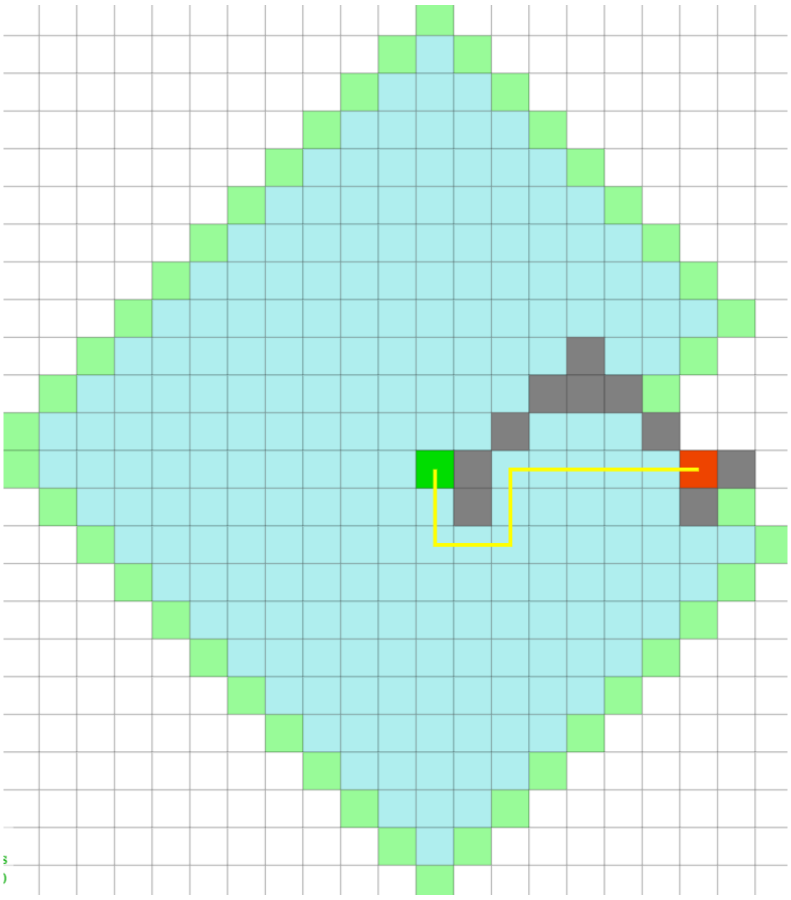
The fundamental difference between Best First-Search and A* is that the former depends exclusively on the heuristic value. That is why, for this problem, it uses fewer operations than A*. It also ends when it discovers a goal (since 'h(x)=0'), whereas A* does not immediately do (it explores 'f(x)': s equal to the target node).



Operations needed to find goal: `48`

c) Using a board like the board in question 4a) or in http://qiao.github.io/PathFinding.js/visual/, describe and draw a situation/scenario where a Breadth-First Search would find a shorter path to the goal compared to a Greedy best-first search. Consider that a piece can move on the grid horizontally or vertically, but not diagonally. Explain why the Breadth-first search finds a shorter path in this case

Below is the solution to the same problem:

Best-First Search	BFS
	
length: 15, number of operations: 55	length: 11, number of operations: 480

Best-First Search finds a path of length 15, although with fewer operations, while BFS keeps its promise of finding the ideal path of 11. The Breadth-First Search algorithm finds a shorter path by scanning the entire environment and sweeping the cases. Best-First-Algorithm utilises a heuristic function called "Manhattan," which directly goes and draws a straight line in the point direction to minimise cost and then goes along the frontiers resulting in a longer path than the Breadth-First Search algorithm.

5) [2p] This question is about the relation between Markov decision processes in Assignment 5 and search algorithms

a) Discuss when and how the generic search problem can be described as a Markov decision process (MDP)

A generic search problem can be described as an MDP in the following way:

- A general search problem's nodes should be described as MDP states.

- The possible actions from the corresponding state in an MDP should be described as the out-arcs from a node in a general search problem. An absorbing state should exist for a node with no out-arcs.
- In an MDP, the reward for an action should be the cost of the corresponding arc of the general search problem. However, the reward of action in an MDP should be the negative cost of the associated arc because it is a cost, not a reward. If an arc has a cost of 1, the reward for the associated action in an MDP should be -1.
- The MDP should be deterministic. Taking action A1 from state S1 should always end up in state S2. As a result, the transition probabilities must always be 0 or 1.

A general search problem can be characterised as a Markov decision process by transforming the problem environment into a space of states, i.e. defining the Markov chain and actions between these states. These actions have a probability of moving from one state to the next. Considering the problem in question 4, going from the node 54 to the node 55 has a probability of 0 because the latter is a wall. This probability is determined by the environment and cannot be changed. The probability of the other actions ranges from 0 to 1.

b) When the search problem can be written as an MDP, what are the advantages and disadvantages of the value iteration algorithm over the A* algorithm?

Advantages	Disadvantages
Value iteration will find the optimal action in each state. We can easily get the optimal path from any state to the goal node by following the optimal action. A* will only find the optimal path from a single start node (and the nodes included in the optimal path). It finds the shortest path, unlike the A* algorithm which finds a ”convincing” short path.	Value iteration needs to perform calculations on all the states in every iteration, which will be heavy if the state space is large, and hence A* star will be much quicker. It has to perform enough iterations to find a decent optimal action, so it has a longer computation time than the A* algorithm.
Finding an admissable heuristic function is not required to use value iteration.	Value iteration will not work if the state space is infinite, as it needs to perform calculations on all states in each iteration.