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 of length r.**

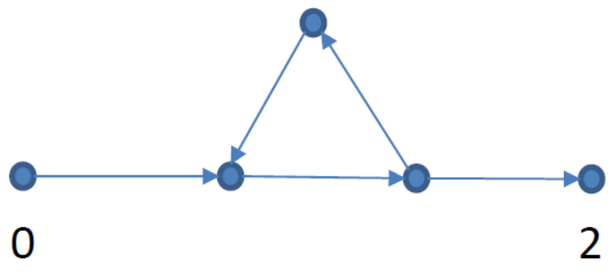
   a) **What is the maximum number of Breadth-First Search (BFS) iterations required to reach the solution in terms of d and r?**

   Assuming the worst-case scenario, where each node has d child nodes, the total number of nodes in the tree grows exponentially with d^r. If we then consider the behavior of the Breadth-First search algorithm, which visits each node in the frontier, one per iteration, at every level of the tree, it becomes quite obvious the total number of iterations required to reach the solution also scales with d^r. This is assuming the goal is located at the last node of the tree.

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

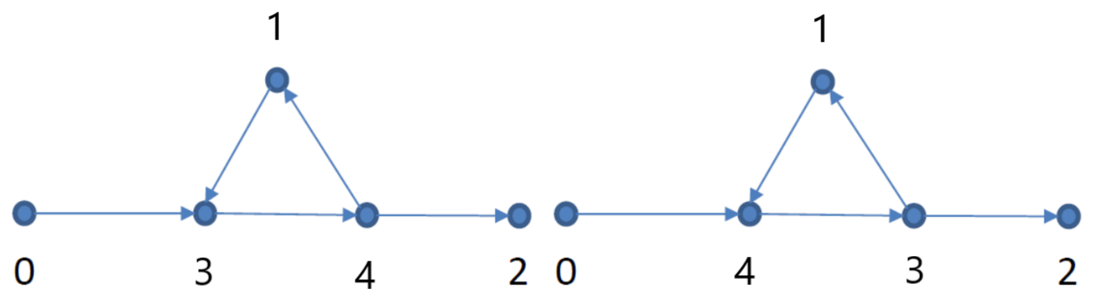
   The way the algorithm searches the tree, it needs to keep the entire frontier in memory at each level of the tree. Hence, at the first level, it requires d units of memory. The nodes at this level have d child nodes each, resulting in a frontier of d^2 nodes and so on. Following this logic, the final layer, which contains the largest frontier, requires d^r units of memory, which corresponds to the memory complexity of the algorithm.

2) **[1p] Take the following graph where 0 and 2 are respectively the initial and the goal states. The other nodes are to be labeled 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 labeling of these three nodes, where DFS will never reach the goal! Discuss how DFS should be modified to avoid this situation?

One situation where the Depth-First search algorithm won’t ever reach the goal node for the graph above is whenever the node at the top of the triangle has a label lower than that of the goal node. The following scenarios illustrate this idea:



One solution to this might be to exclude already visited nodes from ties. This would cause the algorithm to visit the goal node after returning to the node in the triangle’s right corner the second time around. However, this could required a substantial amount of memory if the graph is large.

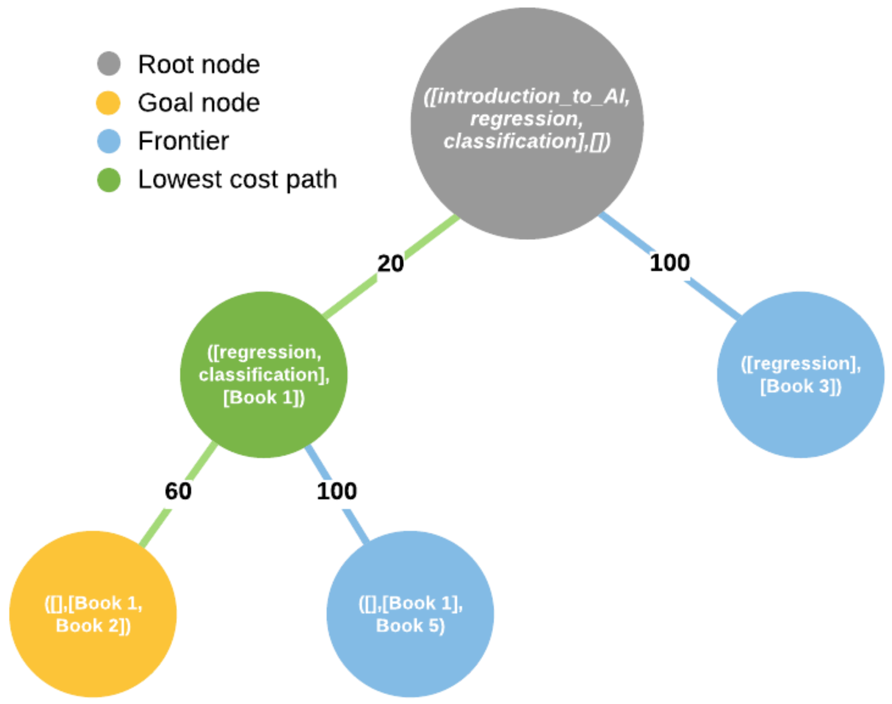
3) **[2p] A publisher allows teachers to “build” customized textbooks for their courses by concatenating the text from different books in their catalog. The catalog contains the following books, together with the number of pages that the book contains, and the topics covered in that book:**

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]

Suppose we define a node to be a pair (Topics, Books) where Books is a list of the books that will make up a customized textbook and Topics is a list of topics that must be covered by the customized textbook but are not already covered by Books. Therefore, a node is only valid if none of the books in Books covers any of the topics in Topics. Child nodes are obtained by selecting a topic from Topics, then selecting a book that covers this topic, then adding this book to Books, and finally removing all of the topics that are covered by this book from the Topics list. For example, if we have the node ([introduction_to_AI, classification], []) and we select the topic ‘introduction_to_AI’ then the child nodes will be ([], [book3]) and ([classification], [book1]). Thus, each arc in the graph adds one book which covers one or more topics. Suppose that the cost of an arc is equal to the number of pages in the selected book. The goal is to design a customized textbook that covers all of the topics requested by the teacher, i.e., the topics in the list Topics. The start node is (Topics, []) and the goal nodes have the form ([], CustomisedTextbook), where CustomisedTextbook is a list of books selected from the catalog. The cost of the path from the start node to a goal node is equal to the total number of pages in the customized

textbook, and an optimal customized textbook is one that covers all of the requested topics (i.e., all of the topics in Topics) with the fewest pages.

a) **Suppose a teacher requests a customized textbook that covers the topics [introduction_to_AI, regression, classification] and that 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. Indicate which node is a goal node, and which node(s) are at the frontier when the goal is found.**

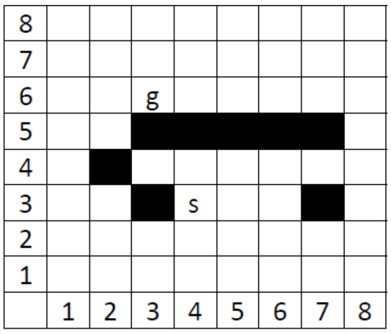


As the graph suggests, the optimal (most concise) textbook covering the topics: *introduction to AI*, *regression,* and *classification* is constructed by combining books 1 and 2, yielding a textbook of only 80 pages in total.

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

For a function to qualify as heuristic, it must be less or equal to the actual cost of the shortest path from node n to a goal. One such function might be the number of pages in a book divided by the number of topics in that book that are left to be covered. This heuristic function acts as a measure of the underlying goal of this algorithm, to minimize the number of pages of a complete book, by solving the much simpler problem of how to maximize value (topics covered) per page in the final book.

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 in the case of a tie the algorithm prefers the path stored first.

i) first iteration

Visited list			
node	**s-score (cost)**	**f-score (cost + heurisitc)**	**previous**

Unvisited list			
node	**s-score (cost)**	**f-score (cost + heuristic)**	**previous**
42	0	4	none

ii) For the second iteration, we pick the node with the lowest f score in the unvisited list which in this case is node 36.

Visited list			
node	**s-score (cost)**	**f-score (cost + heuristic)**	**previous**
36	0	4	none

Unvisited list			
node	**s-score (cost)**	**f-score (cost + heuristic)**	**previous**
42	1	5	36
33	∞	∞	36
44	1	3	36
53	1	5	36

iii) At the third iteration, the algorithm visits node 44 as it has the lowest f-score in the frontier.

Visited list			
node	**s-score (cost)**	**f-score (cost + heuristic)**	**previous**
s	0	4	none
44	1	3	36

Unvisited list			
node	**s-score (cost)**	**f-score (cost + heuristic)**	**previous**
s	2	4	44
34	2	2	44
45	2	∞	44
54	2	4	44

iv) During the fourth iteration, the algorithm visits node 34 as it has the lowest f-score in the frontier.

Visited list			
node	**s-score (cost)**	**f-score (cost + heuristic)**	**previous**
s	0	4	none
44	1	3	36
34	2	2	44
42	1	5	36

Unvisited list			
node	**s-score (cost)**	**f-score (cost + heuristic)**	**previous**
33	∞	∞	44
24	∞	∞	44
35	∞	∞	44

44	3	3	44

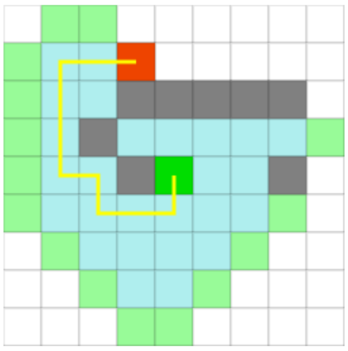
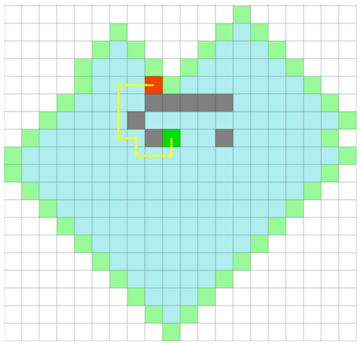
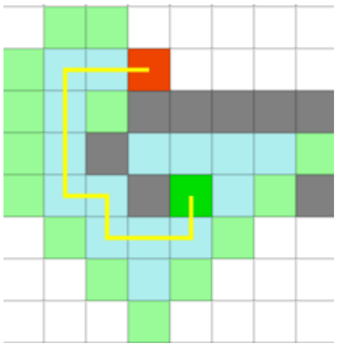
v) Since 44 has the lowest f-score it goes to the lowest f-score that is oldest, 42

Visited list			
node	**g-score (cost)**	**f-score (cost + heuristic)**	**previous**
s	0	4	none
44	1	3	36
34	2	2	44
42	1	5	36

Unvisited list			
node	**g-score (cost)**	**f-score (cost + heuristic)**	**previous**
41	3	6	42
32	3	4	42
s	3	4	42
52	3	6	42

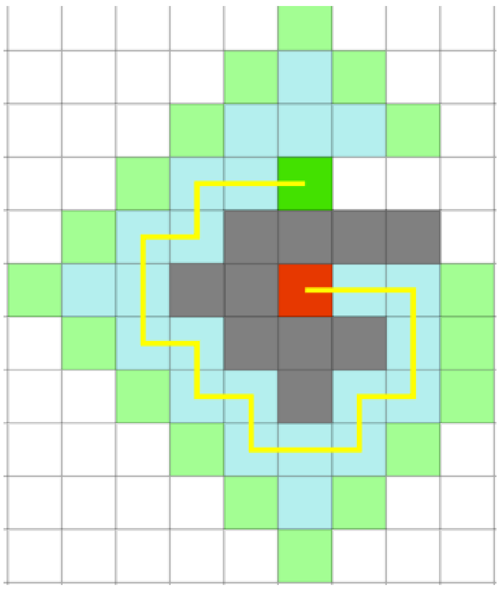
After five iterations the path is {s,42,32}

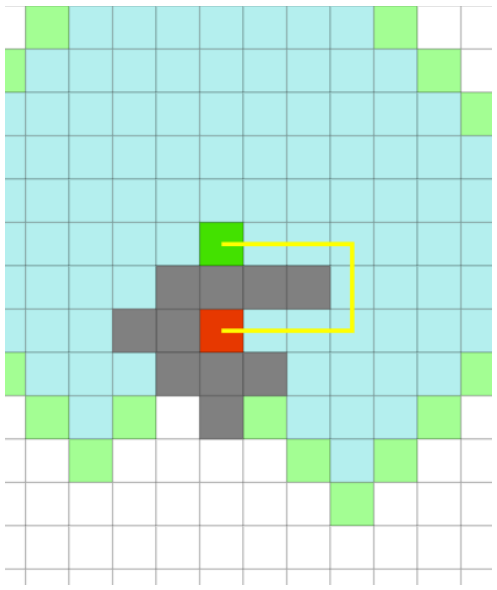
b) Solve this problem using the software in http://qiao.github.io/PathFinding.js/visual/ Use Manhattan distance, no diagonal step, and compare A*, BFS, and Best-first search. Describe your observations. Explain how each of these methods reaches the solution. Discuss the efficiency of each of the methods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

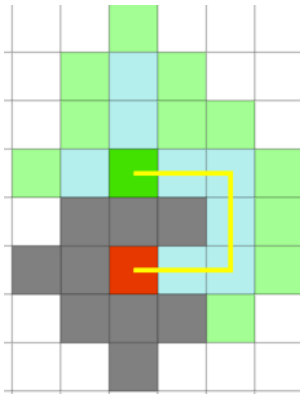
A*: length = 10, iteration = 71
BFS: length = 10, iteration = 364
Best-first search : length = 10, iteration = 48
The above figures show A*, Breadth-First seachand Best-First Search in the mentioned order. As one might notice, the Best-First search algorithm outperform the others for this particular scenario. However, the A* algorithm also does a decent job of locating the goal node in a relatively small number of iterations compared to the Breadth-First algorithm.
The two most similar algorithms, A*, and Best-First search, only differ in how they decide which state's to visit next. The latter uses a (greedy) cost function $f(n) = h$ where h denotes the heuristic, being Manhattan distance, while the latter uses $f(n) = h + g$, with $g$being the cost of the path from the initial state to the particular state n. These types of search algorithms are often referred to as informed search as they, compared to Breadth-First search, use a heuristic measure to indicate a sense of direction towards the goal. This is the main reason why these two algorithms perform much better than Breadth-First search, which exhaustively searches the grid without the additional information provided from a heuristic function.
As to why Greedy Best-First search outperforms the A* algorithm is most likely explained by how the grid is designed. To be more specific, as the A* algorithm tries to walk around the left corner its cost function grows with each step, causing it to eventually abandon this path and explore other nodes that were previously added to the frontier. The Greedy Best-First algorithm on the other hand doesn't care if the path it's on grows very long. All it cares about is the distance to the goal is decreasing, hence it keeps going where A* didn't.

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





The above images show Greedy Best-First and Breadth-First search used on the same grid layout. As one sees, here the latter mentioned algorithm finds the better, optimal, solution. The reason the Greedy Best-First algorithm has this behaviour is it always resort to picking the, for the moment, most optimal path, being the one closest to the goal. In this scenario, taking the left path might seem better from the heuristic function's point of view. However, even though it's obviously a suboptimal solution, the greedy algorithm never sees this as it discard the path for a seemingly better one, preventing it from discovering the more optimal path. To illustrate this, removing one block from the right path allows the greedy algorithm to find the optimal solution, as seen in the image below.



5) **[2p] This question is about the relation of 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).**

      The aim of a general search problem is to find a, preferably optimal, path between a start and a goal node. The search finishes once this path is found, or no paths exist and all nodes have been visited. The optimal solution is then defined as the path which minimizes the total cost of traversing the weighted edges of this graph.
      One defining property of the Markov Decision Process is that every state is associated with a set of actions that are performed according to some probability, making it non-deterministic. This poses an issue, as the general search problem is deterministic, given a start and a goal node one expects to find the same optimal path over different runs. Consequently, for such a problem to be described by a Markov Decision Process the probability of its actions must be constrained such that it's always 1. Hence, a search problem is only a special case of a Markov Decision Process.

   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?**

      The Value Iteration algorithm works by iteratively calculating and updating the value function for all states until convergence. This obviously becomes a performance and memory issue as the state and action space grows. The A* algorithm on the other hand often reaches the optimal solution after only visiting a small portion of the state space allowing it to scale a lot better than the Value Iteration algorithm.