Assignment 8: Rule-based AI

Group 6:

Name	Contribution
1. Himanshu Chuphal (guschuhi@student.gu.se)	12 H
2. Claudio Aguilar Aguilar(claagu@student.chalmers.se)	12 H

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’?

[Answer]

The maximum number of Breadth First Search (BFS) iterations required to reach the solution in terms of ‘d’ and ‘r’:

$\sum_{i=0}^{r} d^i$ iterations

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’ ?

[Answer]

It would require probably d^r units of memory in order to store the maximum queue length, i.e. the length of the longest row in the graph.

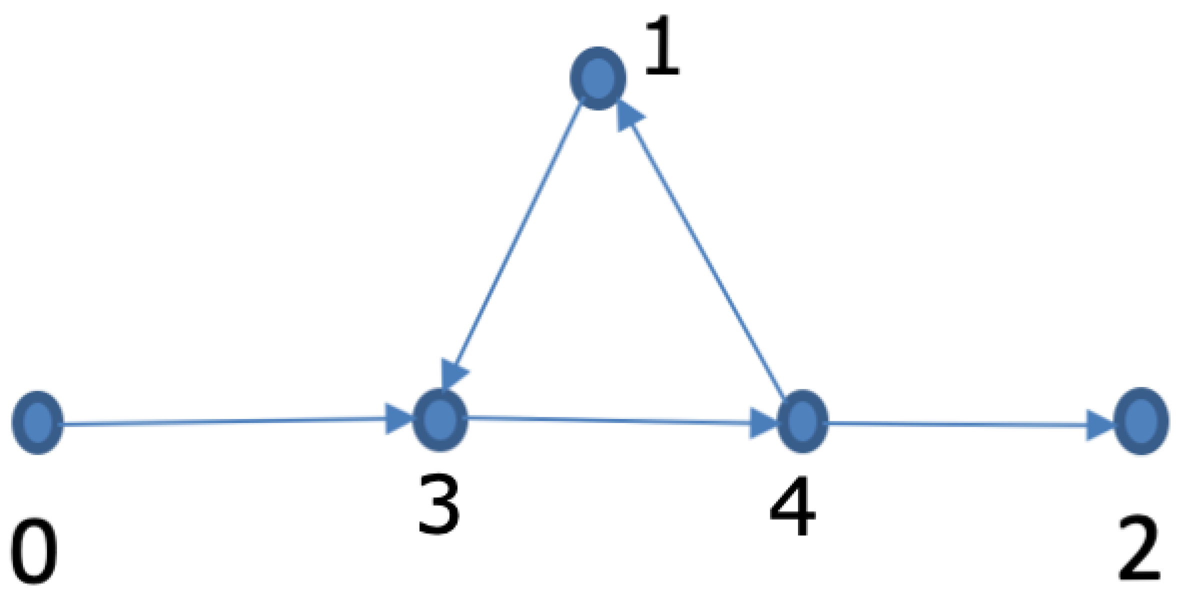
Question 2)

[1p] 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 we use the Depth First Search (DFS) method and in the case of a tie, we chose the smaller label. Find all labelling of these three nodes, where DFS will never reach to the goal! Discuss how DFS should be modified to avoid this situation?

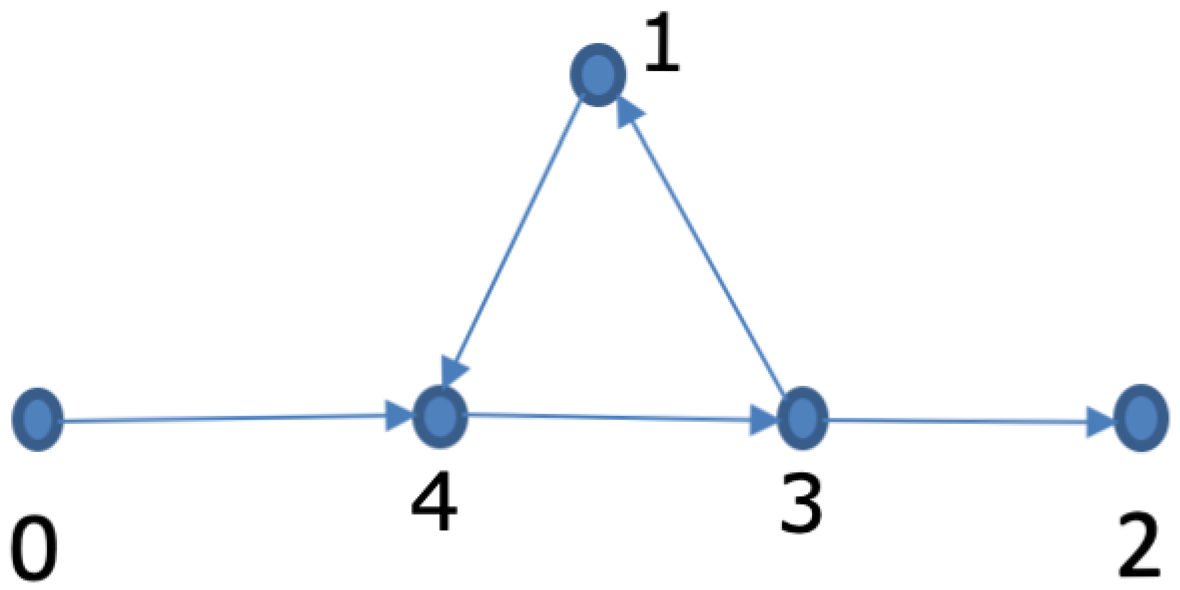
[Answer]

All labelling of the three nodes where DFS will never reach the goal:

look either like this:



or like this:



As long as the top label is a '1' the algorithm will always select that direction since it is the smallest lable(smaller than the goal label). Causing the algorithm to get stuck in an infinite loop and never reach to the goal.

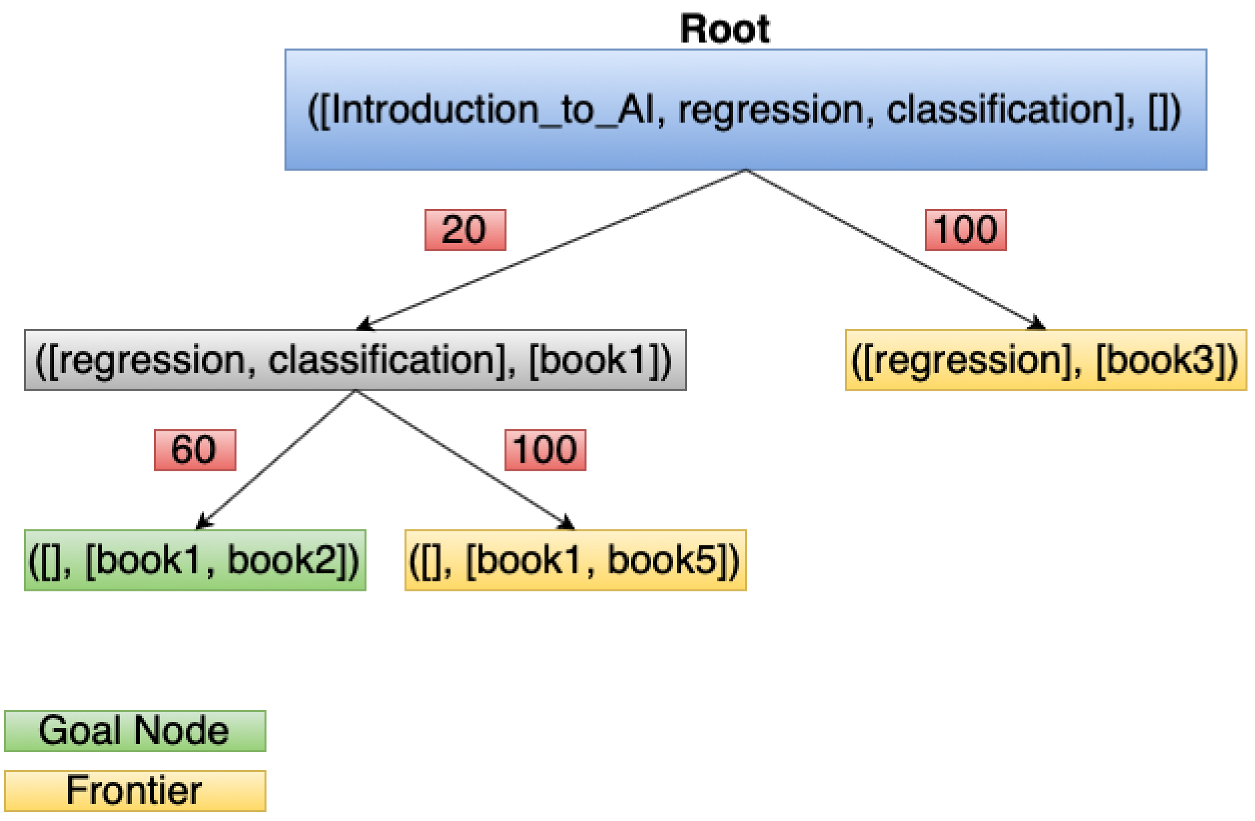
To avoid this situation we need to define two terms called the frontier- and the explored list. The frontier is a list with nodes we know about but have not yet visited. The explored list is a list of visited nodes. The frontier is used to track which node we will explore next. The explored list is used to prevent us from searching nodes we have already visited, which in this case would be helpful since it would prevent us from visiting the top node repeatedly, thus preventing the algorithm to get stuck in an infinite loop.

Question 3)

a) Suppose a teacher requests a customised 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.

[Answer]

Since we implement a lowest-cost-first search and the algorithm always selects the leftmost topic when generating child nodes, our path was defined as:



Goal node is in green color and frontier in yellow

From the root node:

The topic Introduction_to_AI is both covered by book 1 and book 3, but since book 1 is cheaper(less pages) we added that book instead and added book 3 to the frontier. From here(the gray area), we have to choose between book 2 or book 5 since they both cover the topics "regression" and "classification". Since book 2 is cheaper we went that way and added book 2 which means that book 5 is added to the frontier, and the goal node(green area) is found since we have covered all the topics.

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

[Answer] h here the estimated movement cost to move from that given node to the final destination, which is referred to as the heuristic. In the given case, probable heuristic function is h(n) as the total cost from the start node to the final destination, using which the frontier is sorted with lowest h(n) to start with and will be less than the final cost to find the destination node but eventually will be same as the cost for the final goal.

<u>So in our case a non-trivial heuristic function would be from node:</u>

([Introduction to AI, regression, classification], []), where we first off look for which books that cover these topics and combine them to find the cheapest path(less pages) in order to not overestimate the cost of reaching the goal, because then the heuristic would not be admissible.

<u>Books that covers the topics:</u>

"Introduction to AI"

- book1 (20 pages)
- book3 (100 pages)

"Regression"

- book2 (60 pages)
- book5 (100 pages)

"Classification"

- book2 (60 pages)
- book3 (100 pages)
- book5 (100 pages)

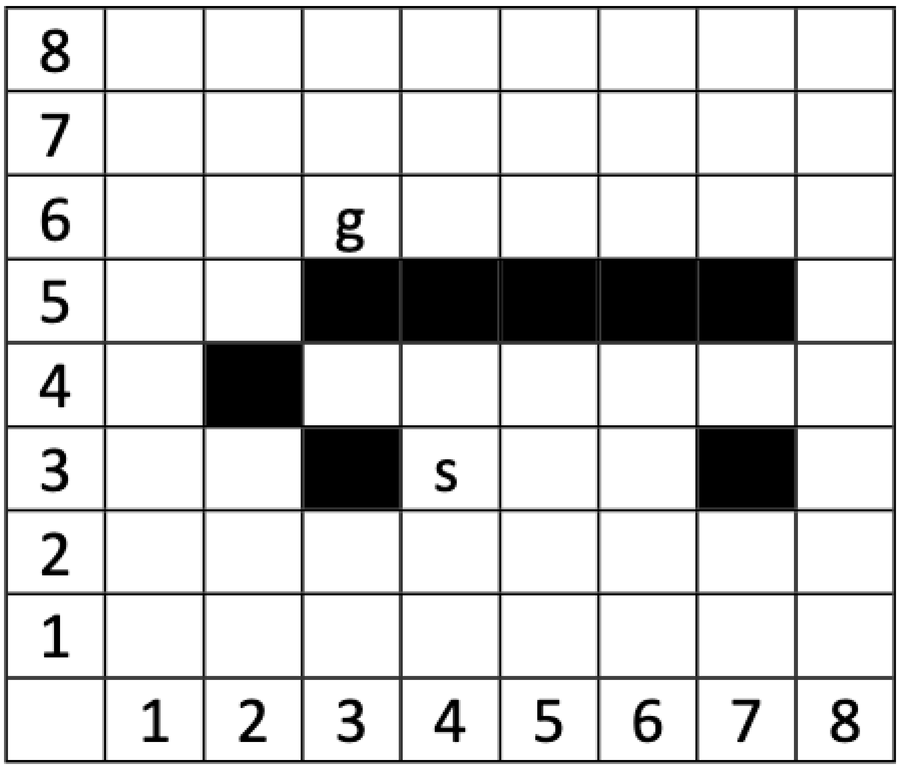
Now from here we can list the different book combinations that covers all the topics and find the cheapest path:

1. book1 + book2 = 20 + 60 = 80 pages
2. book1 + book5 = 20 + 100 = 120 pages
3. book3 + book2 = 100 + 60 = 160 pages
4. book3 + book5 = 100 + 100 = 200 pages

We can see that combination number 1 gives us the cheapest path and therefore becomes our heuristic.

Question 4)

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 prefers the path stored first



Reasoning behind the iteration process:

At every node we visit we check both horizontally and vertically for our options on where to move. We then pick the path(node) based on the shortest path to the goal node and distance from the start node which is the F-cost. In case of a tie, we compare the different paths heuristics(H-cost) which is the distance to the goal node, and if there is a tie there aswell we then pick the path that was stored first in the frontier list, which is a list of known paths that we have discovered while moving around in the map but have not yet visited since they were less interesting(high F-cost values).

When we have picked the path we move to that node and store the previous node on a visited list in order to not waste computation by revisiting it again. From there we check our options again on where to move, but know we have more knowledge in the form of a frontier-list and visited list that gets updated at each iteration in order to always pick the shortest known path.

This is then iterated until we find the goal node, but in this problem we only present the first five iterations since it was what the problem asked for.

<u>Start Position in the grid:</u>

(4,3) S(Start)

<u>Iteration 1 (At (4,3)):</u>

Frontier

(x,y)	previous node	G-cost(distance from starting node)	H-cost(distance from end node)	F-cost(G-cost+H-cost)
(4,4)	(4,3)	1	3	4
(4,2)	(4,3)	1	5	6
(5,3)	(4,3)	1	5	6

Visited nodes

(x,y)	previous node	G-cost(distance from starting node)
(4,3)	S(Start)	0

<u>Iteration 2 (At (4,4)):</u>

Frontier

(x,y)	previous node	G-cost(distance from starting node)	H-cost(distance from end node)	F-cost(G-cost+H-cost)
(3,4)	(4,4)	2	2	4
(5,4)	(4,4)	2	4	6
(4,2)	(4,3)	1	5	6
(5,3)	(4,3)	1	5	6

Visited nodes

(x,y)	previous node	G-cost(distance from starting node)
(4,3)	S(Start)	0
(4,4)	(4,3)	1

<u>Iteration 3 (At (3,4)):</u>

We can only move to node(4,4) but we have already visited that node so we avoid visiting it again.

Frontier

(x,y)	previous node	G-cost(distance from starting node)	H-cost(distance from end node)	F-cost(G-cost+H-cost)
(5,4)	(4,4)	2	4	6
(4,2)	(4,3)	1	5	6
(5,3)	(4,3)	1	5	6

Visited nodes

(x,y)	previous node	G-cost(distance from starting node)
(4,3)	S(Start)	0
(4,4)	(4,3)	1
(3,4)	(4,4)	2

Iteration 4 (At (5,4)):

Frontier

(x,y)	previous node	G-cost(distance from starting node)	H-cost(distance from end node)	F-cost(G-cost+H-cost)
(4,2)	(4,3)	1	5	6
(5,3)	(5,4)	1	5	6
(6,4)	(5,4)	3	5	8

Visited nodes

(x,y)	previous node	G-cost(distance from starting node)
(4,3)	S(Start)	0
(4,4)	(4,3)	1
(3,4)	(4,4)	2
(5,4)	(4,4)	2

Iteration 5 (At (4,2)):

Frontier

(x,y)	previous node	G-cost(distance from starting node)	H-cost(distance from end node)	F-cost(G-cost+H-cost)
(3,2)	(4,2)	2	4	6
(5,3)	(5,4)	1	5	6
(6,4)	(5,4)	3	5	8
(4,1)	(4,2)	2	6	8
(5,2)	(4,2)	2	6	8

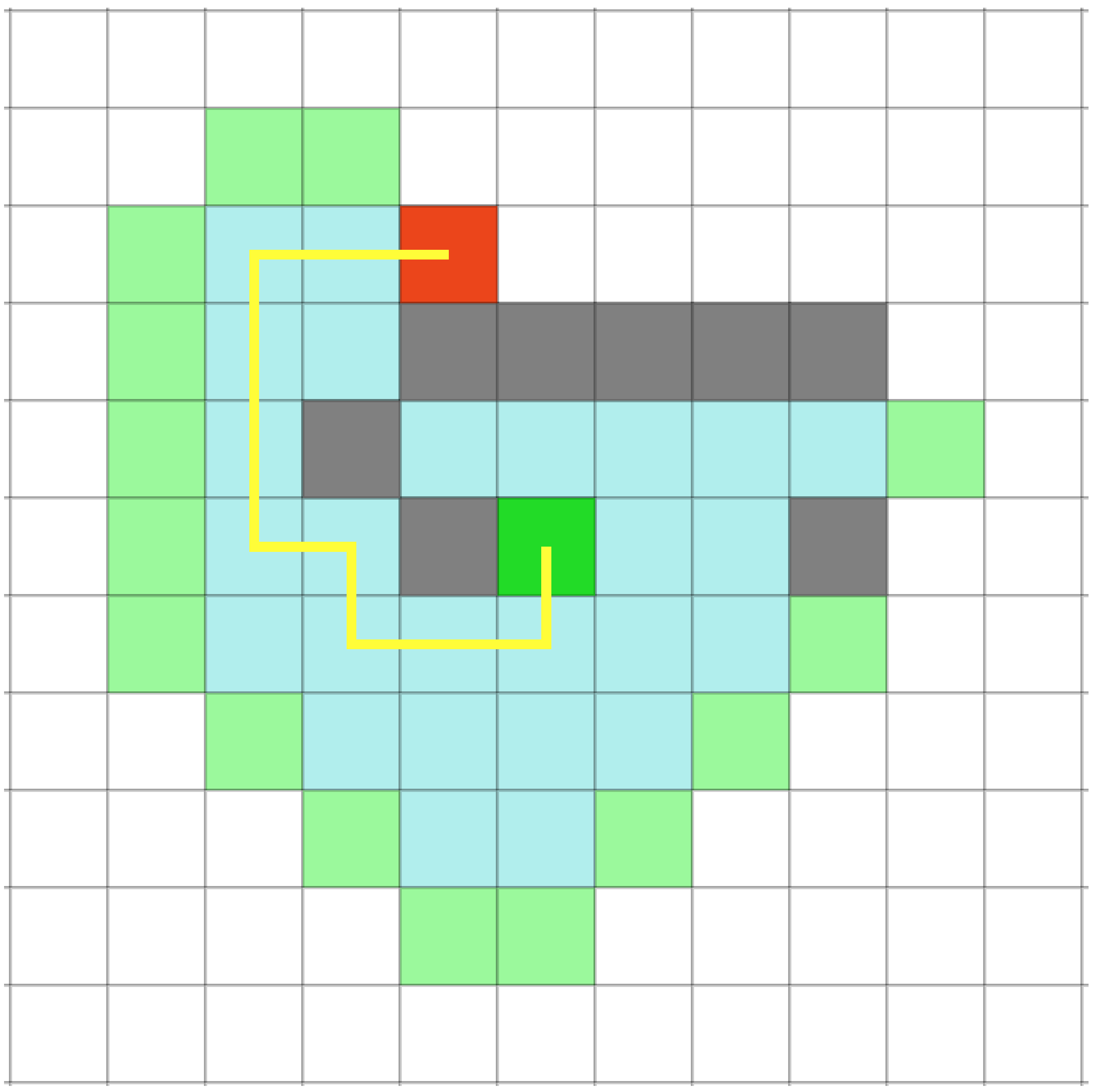
Visited nodes

(x,y)	previous node	G-cost(distance from starting node)
(4,3)	S(Start)	0
(4,4)	(4,3)	1
(3,4)	(4,4)	2
(5,4)	(4,4)	2
(4,2)	(4,3)	1

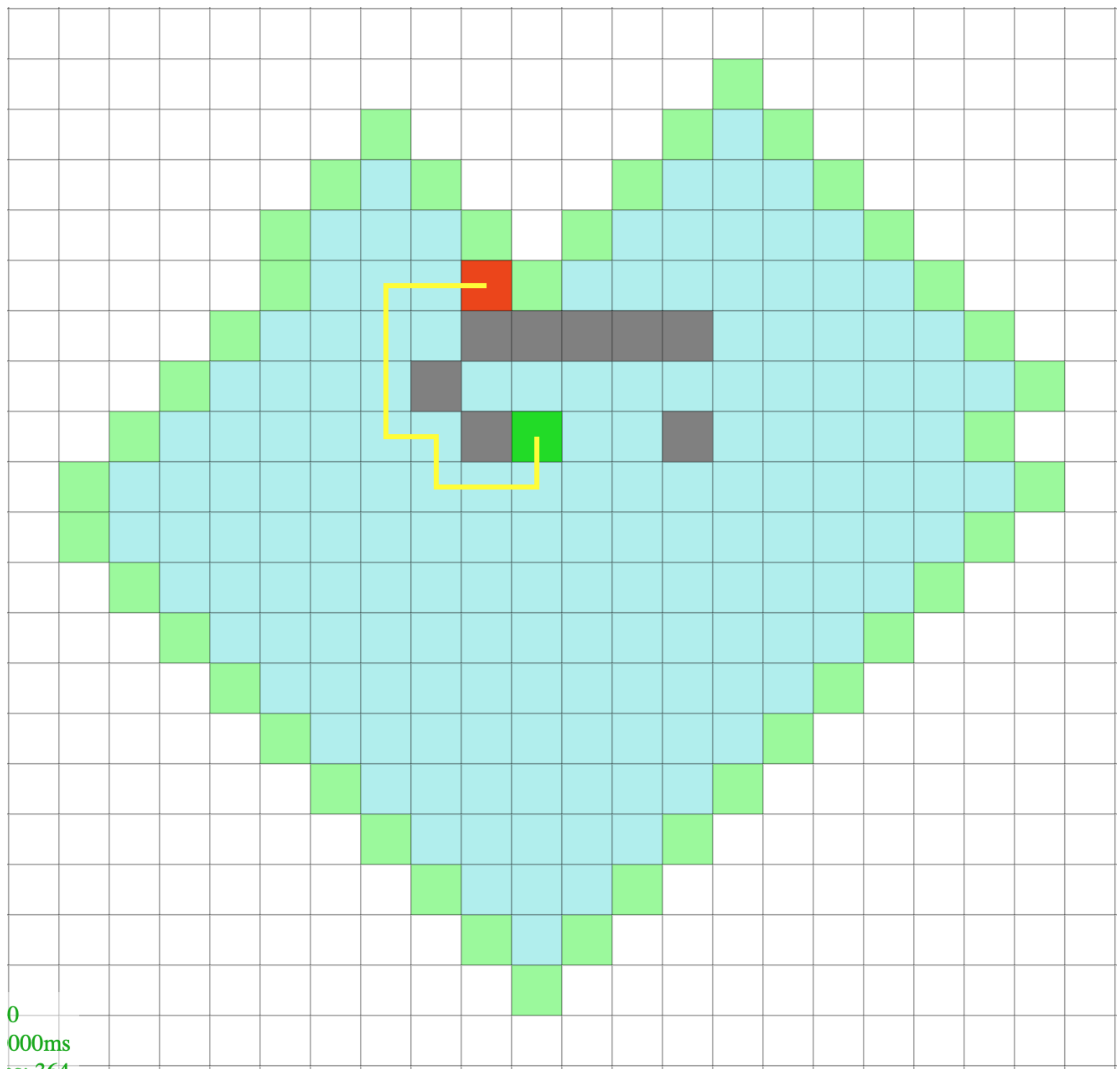
b) Solve this problem using the software in [http://qiao.github.io/PathFinding.js/visual/Use (http://qiao.github.io/PathFinding.js/visual/Use)](http://qiao.github.io/PathFinding.js/visual/Use) Manhattan distance, no diagonal step and compare A*, BFS and Best-first search. Describe your observations.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

[Answer]

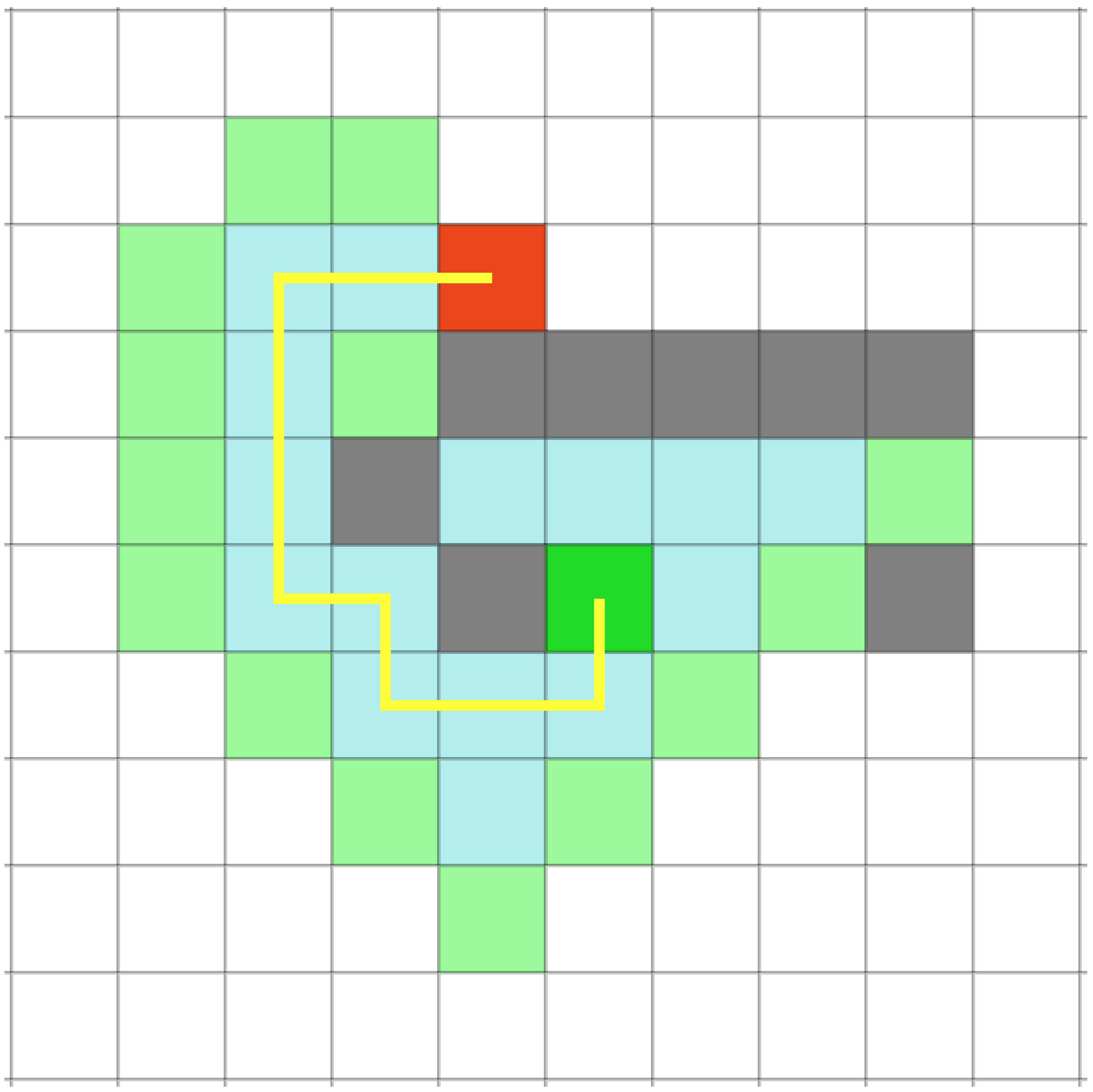
A*:



Breadth-first search(BFS):



Best-first search:



By observing the images, we can see that the most efficient searching algorithm for this particular problem is the **Best-First search** algorithm since it observes the least amount of nodes in order to find the goal node, and thus uses less time and computational memory.

How A* reaches the goal node is explained in subquestion a). A pretty efficient algorithm but not the most efficient for this particular problem.

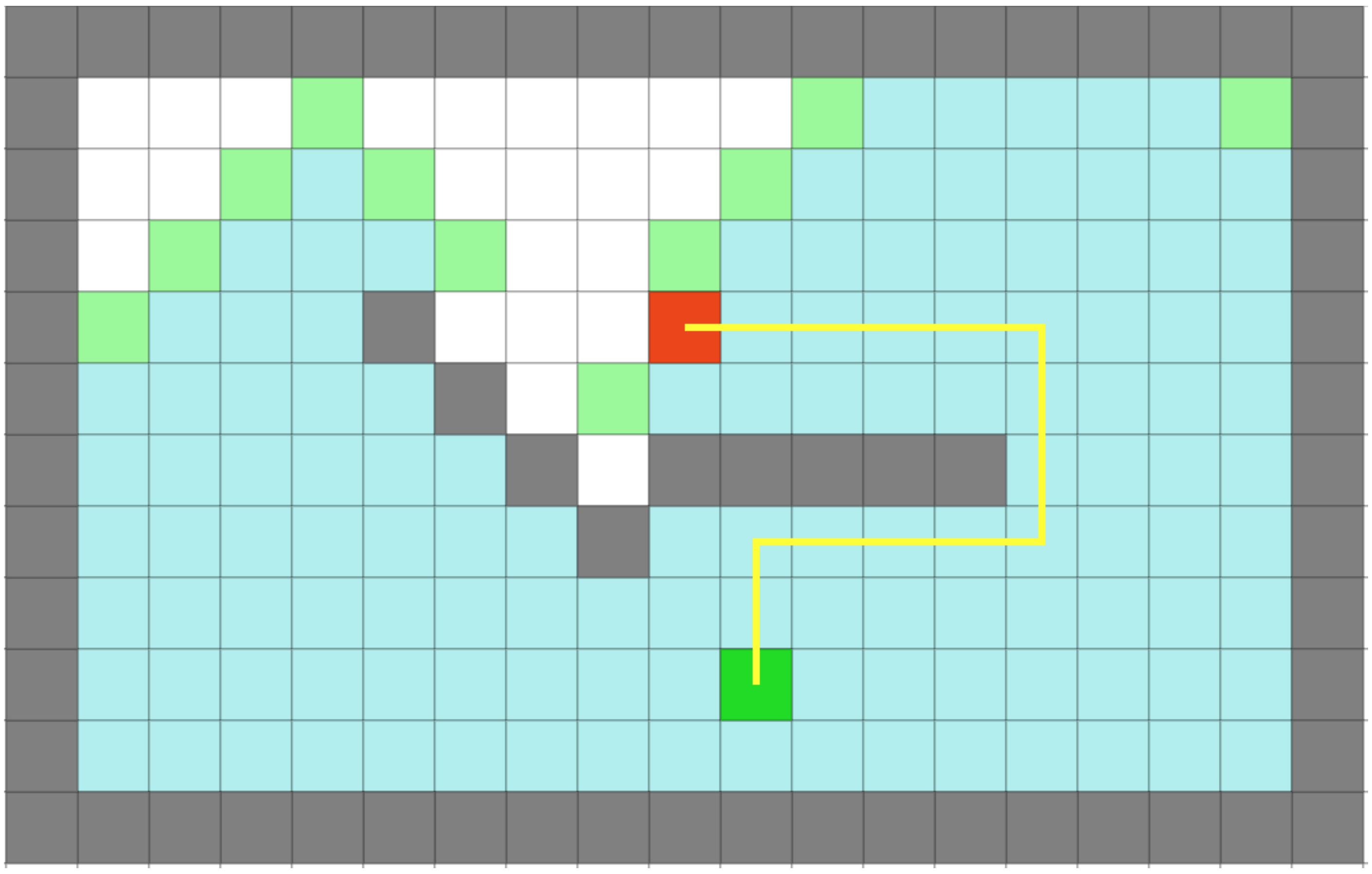
In BFS, we add every adjacent node to the current node we are at to a queue and visit them one by one while adding their neighbours(the ones that is not already in the queue) to the queue as well. We do this until we find the goal node. Less efficient since it takes more observations to reach the goal node, which means more time and computational memory. This is probably due to the walls/obstacles structure in this problem which makes the BFS go in the opposite direction to the goal node and has no help from heuristic function as compared to A* and Best-First search.

Best-first search works similar to the A* algorithm but does not take into account the G-cost which is the distance from the start node. Instead it only considers the heuristic(h-cost) which is the distance from the goal node. For this particular problem it was the most efficient algorithm.

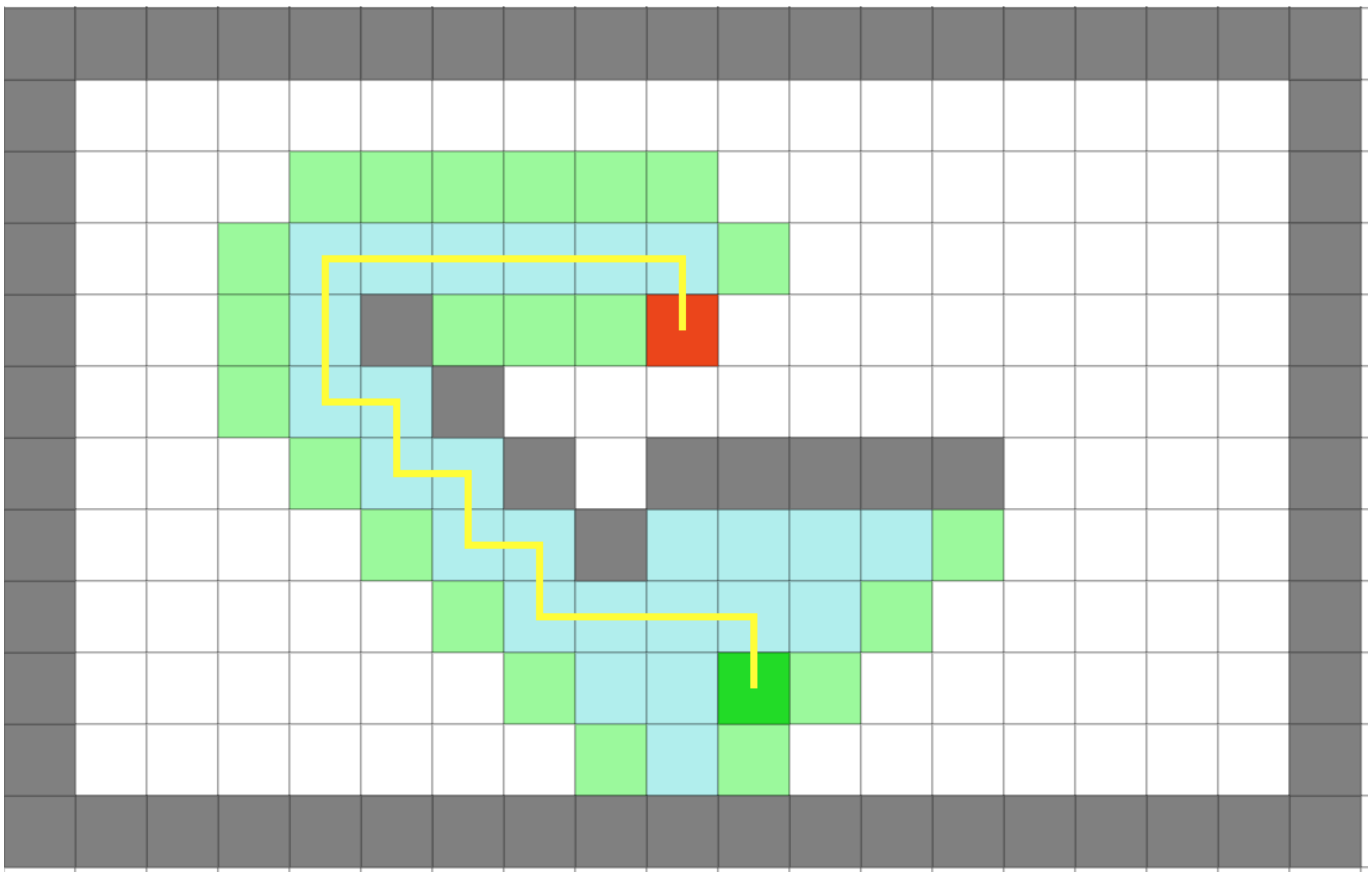
c)Using a board like the board used in question 4a) or in [http://qiao.github.io/PathFinding.js/visual/ (http://qiao.github.io/PathFinding.js/visual/)](http://qiao.github.io/PathFinding.js/visual/), describe and draw a situation/scenario where Breadth-first search would find a shorter path to the goal 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.

[Answer]

Breadth-first search(BFS):



Best-first search:



Here, we have a situation where BFS finds a shorter path than Best-first search. This is because Best-first search only looks for the best heuristic(shortest distance to the goal node). In this situation, it checks both the nodes at the right side and the nodes at the left side, and the further it goes to the right it recognizes that the distance to the goal node (heuristic) increases to a value '7' that is higher than all the other explored nodes and decides to go with the left side of the source beacause of smaller the h-cost(less than 7).

BFS on the other hand always finds the shortest path since it does explore more nodes and therby chooses the shortest path to the goal node, but it requires more observations and becomes much slower than Best-first search.

Hence, Best-first search is much faster than BFS but it cannot guarantee that it will find the shortest path which BFS will do.

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

a)Give a definition of a Markov decision process (MDP). Explain what a policy is and how it relates to such a process.

[Answer] Markov Decision Process (MDP) is a mathematical framework used for modeling decision-making problems where the outcomes are partly random and partly controllable. In Markov Decision Processes, the agent acts according to the state it is in and the agent's actions also influences the next states, while receiving a certain reward depending on the effect of the action performed. The agent selects an action based on a policy, i.e. the probability distribution of actions and rewards, given the states. In other words, the agent tries to maximize the total rewards by choosing the action that are most likely to lead to the highest future reward for the state the agent is in.

The policy the solution of MDP is a mapping from S to a. It indicates the action 'a' to be taken while in state S. A policy is the thought process behind picking an action. In practice, it is a probability distribution assigned to the set of actions. Highly rewarding actions will have a high probability and vice versa.

A MDP consists of five parts: the specific decision times, the state space of the environment/system, the available actions for the decision maker, the rewards, and the transition probabilities between the states.

- Decision epochs: $t = 1, 2, \ldots, T$, where $T \leq \infty$
- State space: $S = \{s_1, s_2, \ldots, s_N\}$ of the underlying environment :: set of possible world states s: a set of tokens that represent every state that the agent can be in.
- Action space $A = \{a_1, a_2, \ldots, a_K\}$ available to the decision maker at each decision epoch :: is set of all possible actions. A(s) defines the set of actions that can be taken being in state s
- Reward functions $R_t = r(a_t, s_t, s_{t+1})$ for the current state and action, and the resulting next state
- Transition probabilities $p(s'|s, a)$ that taking action a in state s will lead to state s' ::(sometimes called Transition Model) gives an action's effect in a state. In particular, T(S, a, S') defines a transition T where being in state S and taking an action 'a' takes us to state S' (S and S' may be same). For stochastic actions (noisy, non-deterministic) we also define a probability P(S'|S,a) which represents the probability of reaching a state S' if action 'a' is taken in state S. Note Markov property states that the effects of an action taken in a state depend only on that state and not on the prior history.

At a given decision epoch t and system state s_t, the decions maker, or *agent*, chooses an action a_t, the system jumps to a new state s_{t+1} according to the transition probability $p(s_{t+1}|s_t, a_t)$, and the agent receives a reward $r_t(s_t, a_t, s_{t+1})$. This process is then repeated for a finite or infinite number of times.

A *decision policy* is a function $\pi : s \to a$, that gives instructions on what action to choose in each state. A policy can either be *deterministic*, meaning that the action is given for each state, or *randomized* meaning that there is a probability distribution over the set of possible actions. Given a specific policy π we can then compute the the *expected total reward* when starting in a given state $s_1 \in S$, which is also known as the *value* for that state,

$$V^{\pi}(s_1) = E\left[\sum_{t=1}^{T} r(s_t, a_t, s_{t+1}) \Big| s_1\right] = \sum_{t=1}^{T} r(s_t, a_t, s_{t+1}) p(s_{t+1}|a_t, s_t)$$

where $a_t = \pi(s_t)$. To ensure convergence and to control how much credit to give to future rewards, it is common to introduce a *discount factor* $\gamma \in [0, 1]$. For instance, if you think all future rewards should count equally, you would use $\gamma = 1$, while if you only care less about future rewards you would use $\gamma < 1$. The expected total *discounted* reward becomes

$$V^{\pi}(s_1) = \sum_{t=1}^{T} \gamma^{t-1} r(s_t, a_t, s_{t+1}) p(s_{t+1}|s_t, a_t)$$

Now, to find the *optimal* policy we want to find the policy π^* that gives the highest total reward $V^{\pi^*}(s)$ for all $s \in S$. That is

$$V^{\pi^*}(s) \geq V^{\pi}(s), s \in S$$

The problem of finding the optimal policy is a *dynamic programming problem*. It turns out that a solution to the optimal policy problem in this context is the *Bellman equation*. The Bellman equation is given by

$$V(s) = \max_{a \in A} \left\{ \sum_{s' \in S} p(s' | s, a)(r(s, a, s') + \gamma V(s')) \right\}$$

if π is a policy such that V^{π} fulfills the Bellman equation, then π is an optimal policy

b) Discuss when and how the generic search problem can satisfy the criteria of a MDP. Explain in detail how the elements of an MDP, such as the reward function R(s'|s, a) should be defined in such a case.

[Answer]

MDPs can formulated with a reward function R(s), R(s, a) that depends on the action taken or R(s, a, s') that depends on the action taken and outcome state.As explained in the previous section, a real valued reward function r(s,a) in MDP is a real-valued reward function. R(s) indicates the reward for simply being in the state S. R(S,a) indicates the reward for being in a state S and taking an action 'a'. R(S,a,S') indicates the reward for being in a state S, taking an action 'a' and ending up in a state S'

The Bellman equation for reward function R(s'|s, a) is given by

$$V(s) = \max_{a \in A} \left\{ \sum_{s' \in S} p(s' | s, a)(r(s, a, s') + \gamma V(s')) \right\}$$

c)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?

[Answer] In value iteration, we start with a random value function and then find a new (improved) value function in an iterative process, until reaching the optimal value function. We can derive easily the optimal policy from the optimal value function. This process is based on the optimality Bellman operator. The value iteration is based on greedy approach where value iteration is done until algorithm converge then policy has to be defined using this value function, on the other hand A* Search algorithm is one of the best and popular technique used in path-finding and graph traversals.

The advantages and disadvantages of the value iteration algorithm over the A *algorithm is based on the space and time complexity of each method. The value iteration is not strongly polynomial, where as in A* algorithm, in the worse case time complexity is O(E), where E is the number of edges in the graph and required auxiliary space in worst case is O(V), where V is the total number of vertices.

Although being the best path finding algorithm around, A *Search Algorithm doesn't produce the shortest path always, as it relies heavily on heuristics. A* performs a little worse since it is more indecisive on which path to follow, and sorting is based on distance (from source) and heuristic(from goal).

The other advantage of A* search algorithm is that many games and web-based maps use this algorithm to find the shortest path very efficiently. It is more prefered in one source and one destination problems with unweighted as well as weighted graphs.