Informatics 2D Coursework 1: Search and Games answers

Author:
Gagan S. Devagiri

UUN:
s1854008

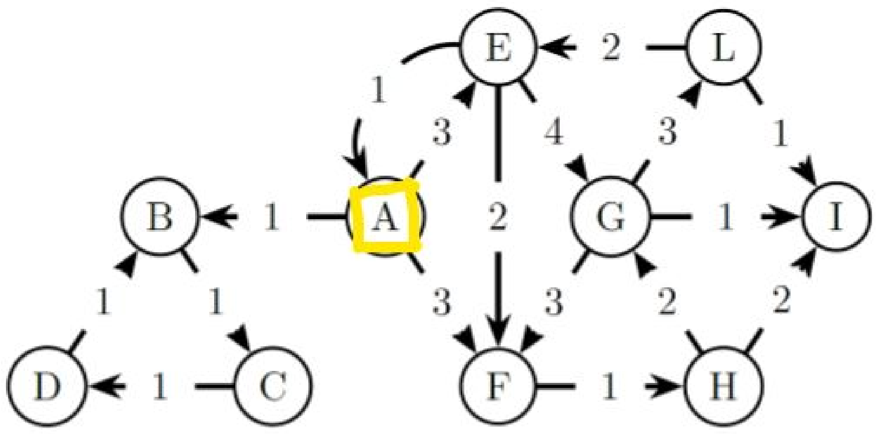
3.4 Uninformed Search - Questions (16%)

3.4.1: (a) Manually perform Breadth-first search with the elimination of the repeated states.

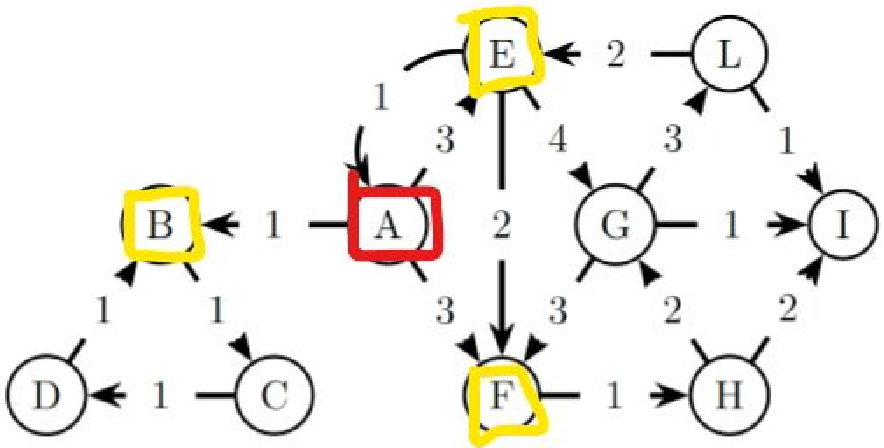
I've shown the step-by-step graph below with the diagram.
A convention I follow is:
- The current nodes in the frontier are yellow
- The nodes in red are explored

Steps:
We have a graph that starts at A and L is the destination node.

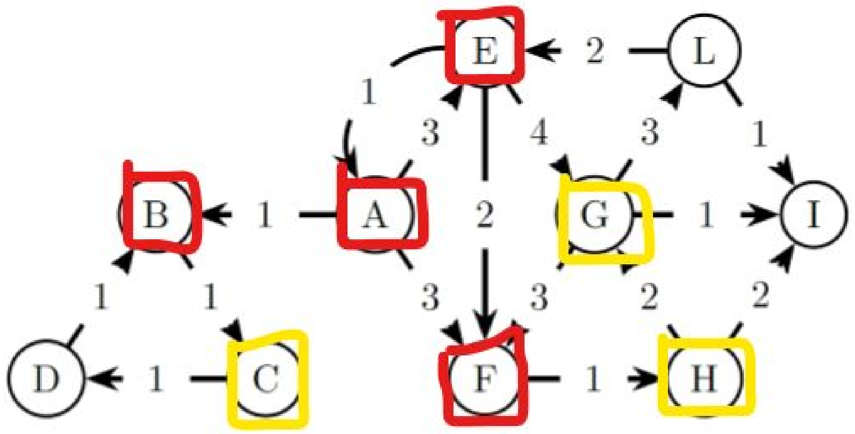
1. The search starts at **A**



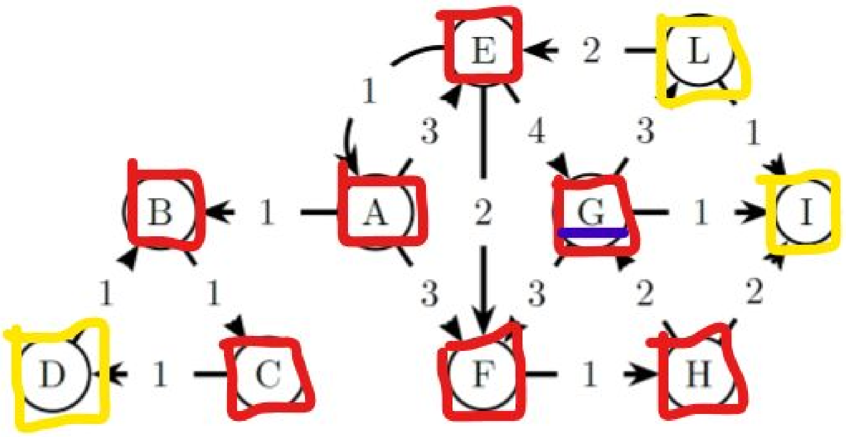
2. the Search puts B, E, F in the frontier



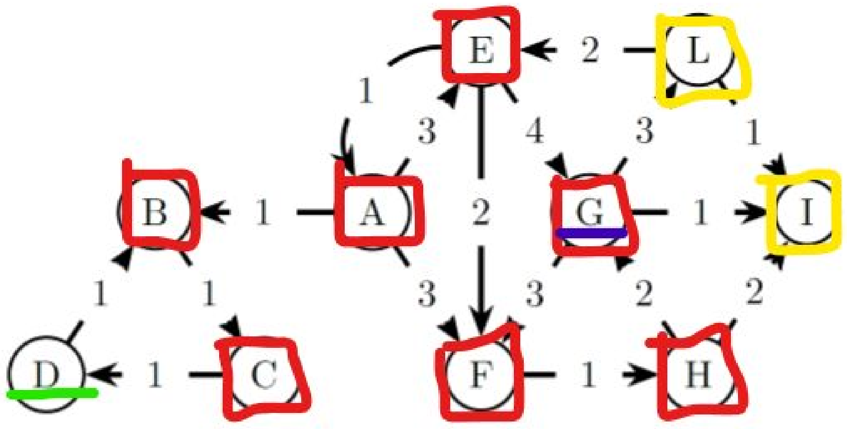
3. The destination node **L** isn't found yet, so the BFS expands again.



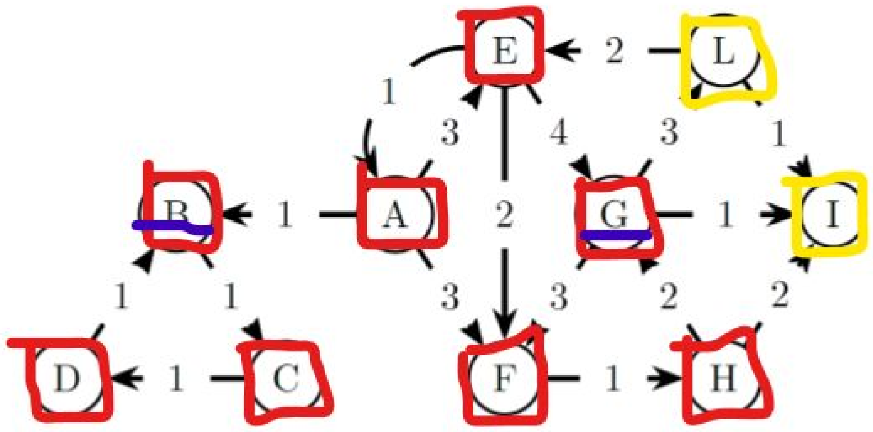
4. This time, at expansion we notice that G was already visited. Thus, BFS doesn't add it to the frontier.



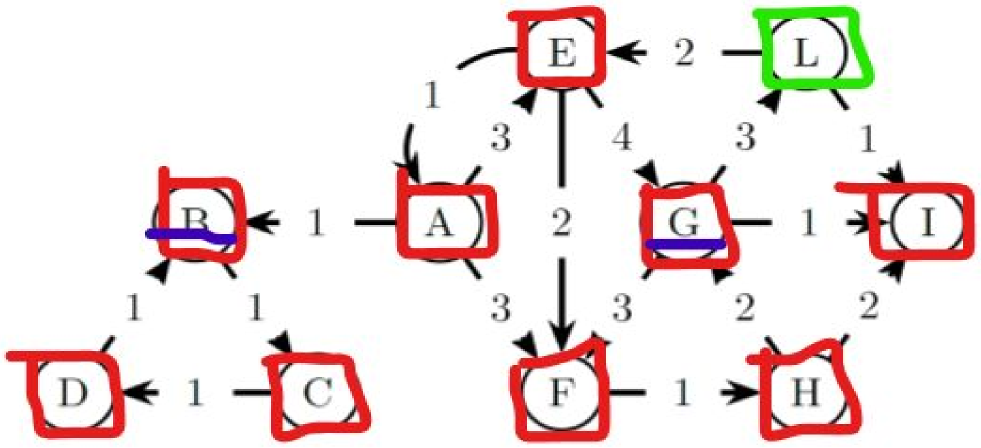
5. Now, we see that the destination node is in the frontier. BFS checks the frontier by taking one node at a time. It starts with **D.**



6. **D** is not the destination node, so it adds it to the explored list and looks at the next node **B** but it won't add it to the frontier as its in the explored list.

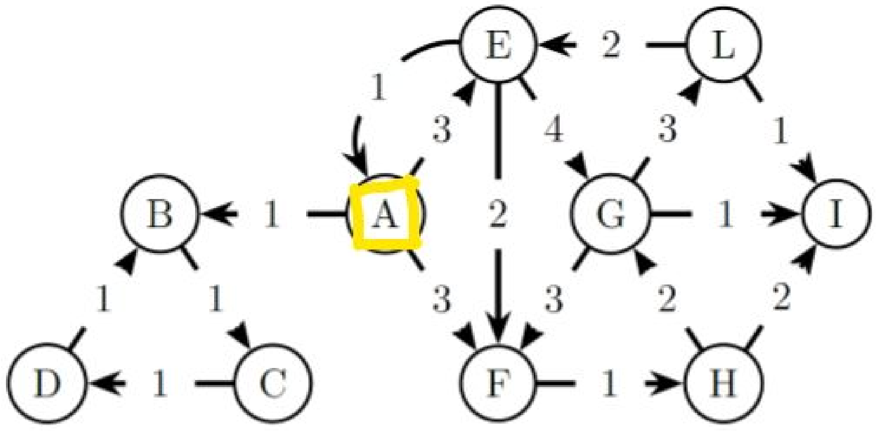


7. The same process goes with **I -** it checks if its the destination node, its not. **I** is added to the explored list and then we finally have **L**, here **L** is checked to see if its the destination node. It is. So we return the solution.

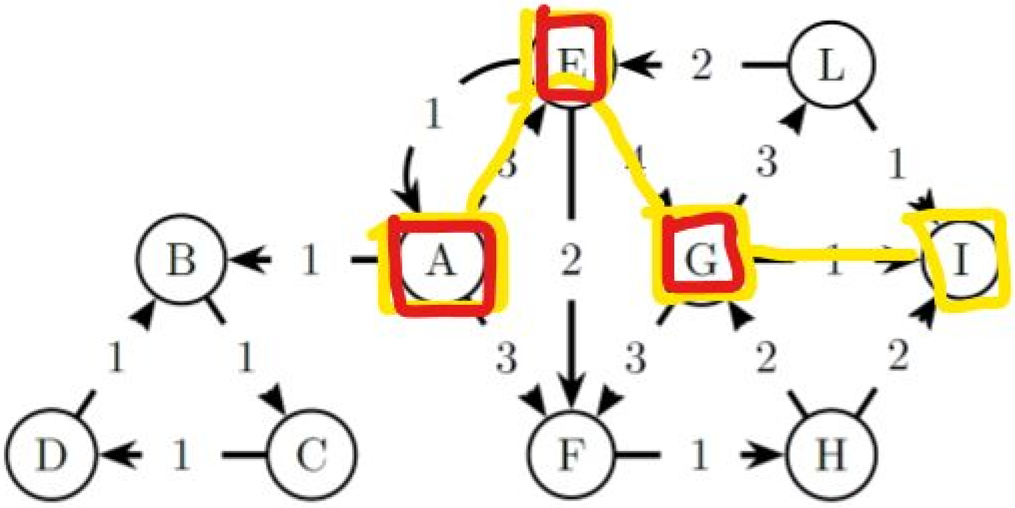


(b) (Manually) perform Depth First Search with elimination of repeated states

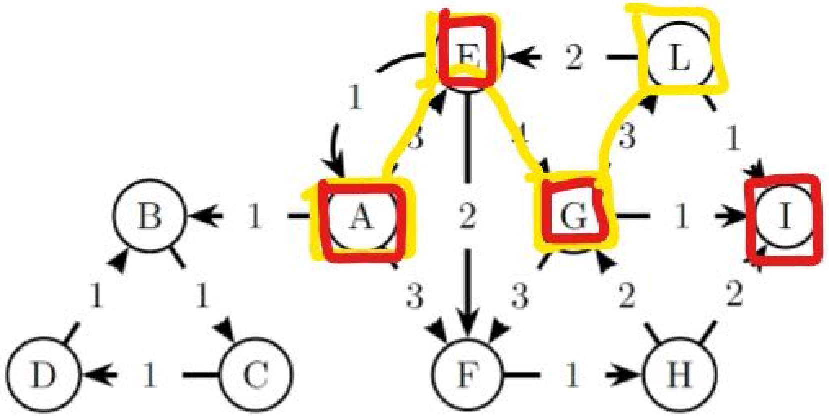
1. We start with the same graph with the starting node as A and search for L. Given graph is lexographic.



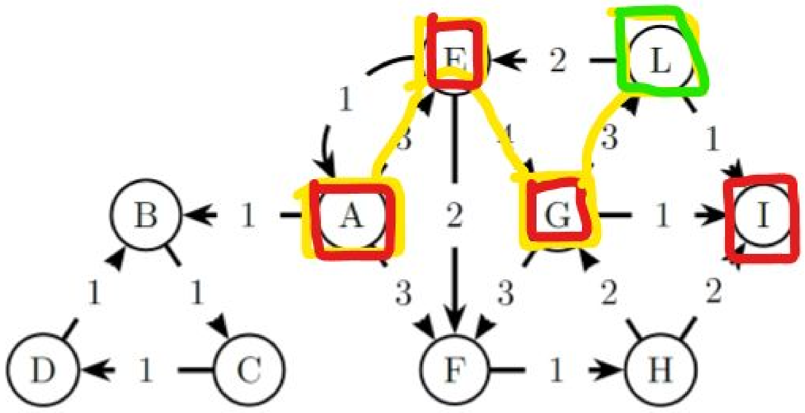
2. First, it expands **E** then goes to **G** then **I**. As it does each step, it adds every visited node in the explored. Like this:



3. We reached a dead-end at **I**, so it recurses back at **G** and goes to the next one. Here, it goes to **L**.



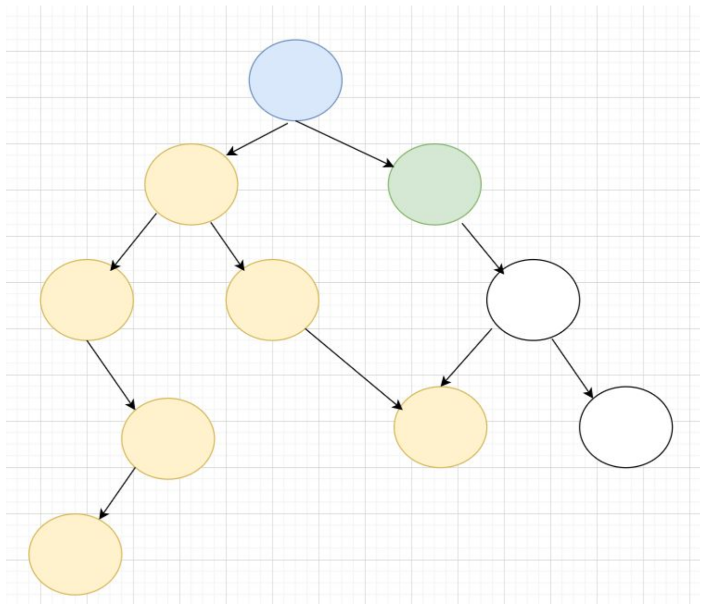
4. We check if **L** is the goal Node. It is, so we return the solution as required.



(c) Draw a graph that favours BFS. Briefly explain why you choose this graph.

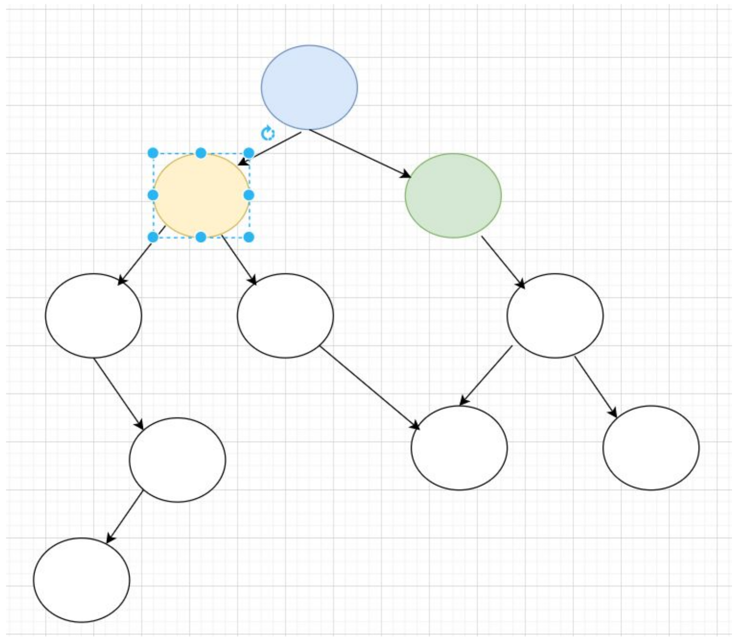
Note: The node labelled blue is the start node. Nodes labelled yellow are the ones that were visited before the destination node labelled.

> DFS for graph 1:



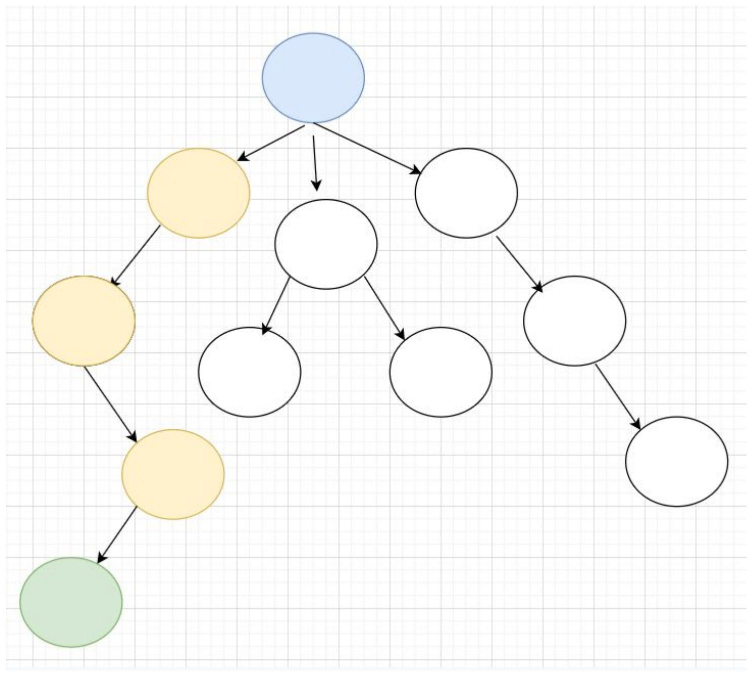
This graph favours BFS because the branching factor is low and the height of the tree is large. We clearly see that the destination node is right next to the starting node, however DFS fails to notice it as it expands the adjacent branch first and looks at all leading paths. We wouldn't miss it we had searched both nodes from the root node. That's why this graph favours BFS.

>BFS on Graph 1:

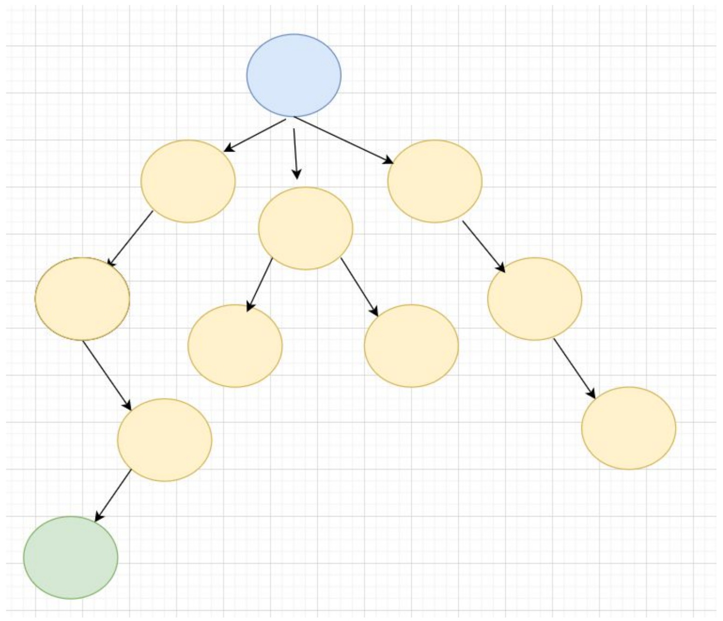


(d) Draw a graph that favours DFS. Briefly explain why you choose this graph.

DFS on graph 2:



BFS on graph 2:



Graph 2 here favours depth-first search. The first node adds to the frontier will be the one on the left. As it continues to expand it, we see that the destination node is found eventually. Here we only had to check 3 other nodes before arriving at the destination node while if had used BFS to find the destination node, as shown in the picture above, we have to explore all 9 other nodes in the graph before reaching the destination node.

(e) Compare BFS and DFS. What are the biggest differences? (Be concise. Character Limit: 750 characters.)

DFS and BFS search are search-algorithms for a graph. We can productively compare them based on four factors: Completeness, time, Space and Optimality along with how it works.
BFS uses a LIFO queue to expand the shallowest unexpanded node first, while DFS uses a FIFO queue to expand the deepest unexpanded node first. Both of them maintain a frontier, an explored empty and have an exponential time complexity. Precisely, its O(b^d) for BFS and O(b^d) for DFS. [1]
DFS has a linear space complexity of O(b*m) and while BFS has O(b^m). Evidently, DFS is preferred when space is an important factor of the search. However, when the goal is at finite depth, with branching factor b, BFS is preferred as it is complete. DFS fails in infinite-depth spaces and spaces with loops.

[1] - b is the branching factor b and d is the depth

2. Iterative Deepening Search

(a) What is the optimal depth for the graph in Figure 2?

Ans. Optimal depth for this graph is 3. It's the optimal depth because the destination node is at depth d. However, any depth greater than 3 would give us a solution closer to the optimal solution, but 3 remains the optimal depth.

3. (b) Compare Depth First Search and Iterative Deepening Search. When would you use Iterative Deepening Search over Depth First Search?

Ans.

Iterative Deepening search (IDS) combines both Breadth-First Search (BFS) and Depth-Limited Search (DLS). Both DFS ***space-complexity*** of **O(b*d)** while IDS has O(d). They both have an exponential ***time complexity*** of **O(b^d)** but IDS has a higher constant factor**.** However, unlike DFS, IDS is guaranteed to find an ***optimal*** when the step costs are identical and IDS is ***complete*** and doesn't get stuck on infinite paths like DFS.

I would use Iterative Deepening Search when the search tree has the same (or similar) branching factor at each level and most of the nodes are at the bottom level. This combined with identical costs at each level is the optimal graph to use IDS over DFS. Another reason I'd use IDS is when finding the destination path is important as IDS is complete and DFS isn't.

4.3 Informed Search - Questions (10 %)

1. Heuristics

(a) Why is the straight line distance a valid heuristic? Name the criteria and why they apply in this case.

Ans. For a heuristic is said to be valid or by showing the heuristic is **admissible** and **consistent**.

A heuristic is ***admissible*** if it never overestimates the cost to reach the goal. We can say that the straight-line heuristic is admissible because the straight line is the shortest path between any two points, so the straight line cannot be an overestimate. Thus, a straight line is an admissible heuristic.

A heuristic h(n) is ***consistent*** if for every node n and every successor node n' of n generated by any action a, the estimated cost of reaching the goal from n is no greater than step cost of getting to n' plus the estimated cost of reaching the goal from n': **h(n) <= c(n, a, n') + h(n')** which holds true for a straight line. Thus we can conclude that straight line distance is valid as its both admissible and consistent.

(b) Describe (1 sentence) three more problems that can be solved with A-Star search and name at least one valid heuristic for each.

Problems that can be solved with A-Star search are:

- A* search can be used to solve continuous motion planning problems in self-driving cars to find a path from point A to point B (Gate to the parking lot)
- A* search can be used to find the problem of parsing using stochastic grammars in Natural Language Processing.
- A* search can be used by social media websites like Facebook or Linkedin to find the shortest path between two people (Degrees of separation)
- A* can also be used in Algorithmic Forex Trading where you buy money in one currency and exchange it into a set of currency to make a profit, leveraging the fact that currency values *may* be independent of each other and subject to change.

2. Best-First Search

(a) What are the differences between the Best-First and A-Star search strategies?

Best-First search:

- Is an instance of the general search algorithm in which a node is selected for expansion based on an evaluation function - **h(n)**. i.e. The algorithm visits the next state based on heuristics function ***f(n) = h*** with lowest heuristic value
- The greedy best-first search is incomplete even in infinite space search.
- Worst case Space-complexity of best-first search is **O(bd)**
- Worst case Time-complexity of best-first search is **O(bd)**

A-Star search:

- Algorithm visits next state based on heuristics **f(n) = h + g**. h is the same as component heuristic and g is the path from the initial state to the current state. A* generally has a better performance than Best-First Search
- Worst case Space-complexity of A* is **O(|V|) = O(b^d)**
- Worst case Time-complexity of A* is **$O(b^d)$**

(b) How would you change your A-Star implementation to be a search instead?

(Change to greed-best first search)

- Greedy best-first search only looks for the local maximum in the search space.
- We can make an assumption that there is a cost function that takes two nodes and finds the cost between them. We'll use it to find the cost between node **n** and node **n'** which can be reached by calling next on **n**

Changes to the function:

- Remove the hrTable function in the argument as greedy best-first search only depends on the current cost between two nodes.
- In the ***sortedFrontierWithCost*** function in my code, we remove the ``**+getHr hrTable (currentBranch !! 0)**`` part of the code.
- In the same function, we replace the cost's argument from g to (**next currentBranch**). So, finally, we have the best first search. A* search is implemented smoothly which makes converting my A* algorithm to Breadth-first search, greedy best-first search easier.

5.4 Connect Four - Questions (15%)

1. Familiarise yourself with the game and its rules.

Done.

2. Given that you can rotate, place a piece and then rotate again, how many possible actions can a player perform during one term?

Ans. You can rotate the board in several ways such as:

- 180 degrees in 4 ways: front & back: horizontal and vertical
- 90 degrees in 2 ways: left, right
- 360 degrees in 4 ways: left, right, up, down

The game has two boards however the goal state from each side is the same. In each board, you can put a token in 16 ways. Thus, there are 16*2 possible ways to place a piece.

So, in total there (4 + 4 + 2 + 4) + 32+(14) = 60 possible actions a player can perform.

4. What are the main challenges for implementing Quadrio over Connect Four with a Twist for alpha-beta pruning? Would such an implementation be practical? Give reasons for your decision.

The first set of problems in implementing the quadrio is to create the infrastructure to support it. Firstly, we should extend our board to support multiple rotations. We currently support 90-degree rotation, this should be extended to 180-degree rotation and 360-degree rotations. Though this appears to be the easier side of problems as most of these rotations give out the same solution and 360-degree rotation is equivalent to no rotation at all. Next, we need to support multiple boards, through the viewpoint of players they both act as one, however, we need to keep track of both the sliders as tokens can later be useful to the players to build up a strategy.

In the picture of alpha-beta pruning: **Yes, such an implementation can be practical.** Quadrio is still a zero-sum game involving two players. However, the search space is now larger.