s1803764

INF2D – Reasoning & Agents
Coursework 1 written questions

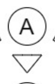
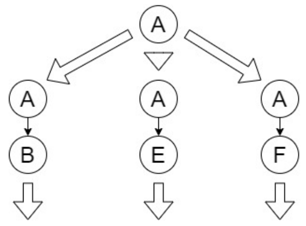
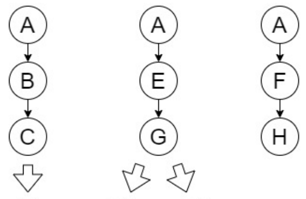
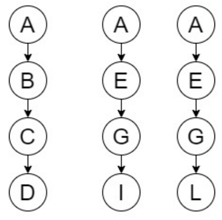
3.4) UNINFORMED SEARCH

Lexicographic

$\therefore \textit{We can assume} \quad A = S_i \ (\textit{initial state})$
$\quad \textit{and that we must explore new nodes in a lexicographic fashion.}$

1) BFS and DFS

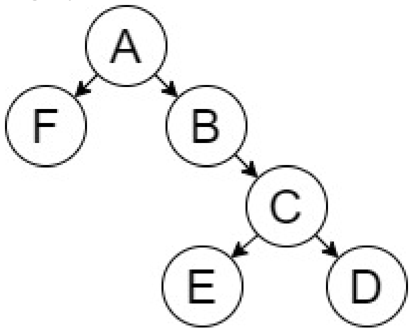
a) Step-by-step sequence of how Breadth First Search is applied to the graph in figure 2:

	Branch search agenda	New explored nodes	Queue data structure	Description
1		A		Queue is initialized. We start search by visiting node A, and mark it as visited.
2		B,E,F	F E B	We then check for unvisited adjacent nodes from node A (B,E, F) mark them as visited and enqueue them.
3		C,G,H	H G C	Now node A is left with no unvisited adjacent nodes so one by one the nodes within the queue (B,E,F) are dequeued and expanded from (where possible), thus creating a new queue of unexplored nodes (H G C).
4		D,I,L	L I D	Now nodes B,E,F, are left with no unvisited adjacent nodes so one by one the nodes within the queue (C,G,H) are dequeued and expanded from (where possible), thus creating a new queue of unexplored nodes (L I D).
5	All the nodes in the graph have been traversed so there is nowhere else to expand, thereby the algorithm terminates.		Back of queue $\longrightarrow$ Front of queue	At this stage, we are left with no unvisited nodes, but as per the algorithm we keep on dequeuing in order to get all unvisited nodes. When queue gets emptied the program is over.

b) Step-by-step sequence of how Depth First Search is applied to the graph in figure 2:

	Branch search agenda	New explored nodes	Stack data structure	Description
1	A	A	A	Initialize the stack. Mark node A as visited and put it on the stack.
2	$A \to B$	B	B A	Explore any adjacent, unvisited node from A (B,E,F). So we mark node B as visited and put it on the stack.
3	$A \to B \to C$	C	C B A	Explore any adjacent, unvisited node from B (C). So we mark node C as visited and put it on the stack.
4	$A \to B \to C \to D$	D	D C B A	Explore any adjacent, unvisited node from C (D). So we mark node D as visited and put it on the stack.
5	$A \to E$	E	E A	No adjacent nodes to explore from node D, node C, or node B, so these are popped off the stack. We now explore any adjacent unvisited node from A (E,F). So we mark node E as visited and put it on the stack.
6	$A \to E \to F$	F	F E A	Explore any adjacent, unvisited node from E (F). So we mark node F as visited and put it on the stack.
7	$A \to E \to F \to H$	H	H F E A	Explore any adjacent, unvisited node from F (H). So we mark node H as visited and put it on the stack.
8	$A \to E \to F \to H \to G$	G	G H F E A	Explore any adjacent, unvisited node from H (G,I). So we mark node G as visited and put it on the stack.
9	$A \to E \to F \to H \to G \to I$	I	I G H F E A	Explore any adjacent, unvisited node from G (I,L). So we mark node I as visited and put it on the stack.
10	$A \to E \to F \to H \to G \to L$	L	L G H F E A	No adjacent, unvisited nodes to explore from node I, so this is popped off the stack. We now explore any adjacent, unvisited nodes from G (L). So we mark node L as visited and put it on the stack.
11	All the nodes in the graph have been traversed so there is nowhere else to expand, thereby the algorithm terminates.		Top of queue $\uparrow$ Bottom of queue	We keep popping off the stack until it is empty since there are no more adjacent, unvisited nodes left to explore. At this point the program terminates.

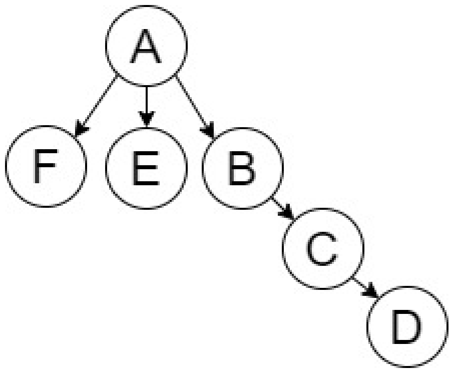
c) A graph that favours BFS over DFS: **(Let start node = A & goal node = F)**



This graph favours BFS as upon first expansion of the start node (A) it finds the goal node (F). Whereas DFS would expand the right branch first going all the way to node D, backtracking to node E, and then finally backtracking all the way to node F.

BFS would have to explore 3 nodes.
DFS would have to explore all 6 nodes.

d) A graph that favours DFS over BFS: **(Let start node = A & goal node = D)**



This graph favours DFS as it would expand the right branch first all the way to the goal node (D). Whereas BFS would first expand all adjacent, unvisited nodes of A (F,E, and B) and then expand B to find C, and then finally reach the goal node (D) upon expansion of node C.

DFS would have to explore 4 nodes.
BFS would have to explore all 6 nodes.

e) Compare BFS & DFS

BFS	**DFS**
Proceeds search level by level.	Proceeds search by following a path until it cannot expand the path no more and then backtracks to the last adjacent, unvisited node.
Uses queue data structure for storing nodes.	Uses stack data structure for storing nodes.
More suitable for finding nodes which are closer to the given source.	More suitable for finding nodes which are farther away from the given source.
Is not memory efficient.	Is memory efficient.
Optimal for finding the shortest path.	Not optimal for finding the shortest path.
Not suitable for decision making trees used in games.	More suitable for decision making trees used in games.

2) Iterative Deepening Search

a) Optimal depth for the graph in figure 2:

Depth = # edges

We can define the optimal depth of an IDS as the most efficient (with regards to space and time) depth in order to find the goal/terminating node.

After analyzing the graph I found that the optimal depth must be 3, as this is the minimum possible path length to reach any node on the graph and thereby reduces the exploration of unnecessary branches, and the number of nodes needed to be stored on the stack.

b) Compare DFS & IDS

DFS	IDS
DFS may explore the entire graph before finding the goal node.	Iterative deepening will only explore the entire graph before finding the goal node if the distance between the start and end node is the maximum in the graph.

When you would use IDS over DFS

i) Iterative deepening is useful for when the goal node is close to the start node (contrary to DFS) especially for large graphs with long branches.

ii) Iterative deepening can also be useful in order to find the optimal path in a graph (contrary to DFS).

4.3) INFORMED SEARCH

1) Heuristics

a) Why straight-line distance is a valid heuristic

Straight-line distance is a valid heuristic as it informs the program which node is closer to the goal node. This is particularly useful and relevant as it makes node expansion and choosing an optimal path more efficient.

b) Problems that can be solved with A* Search and their heuristics

Problems	**Relevant heuristic**
Distance-based route-planning (maps)	Manhattan distance
Cost-based route-planning (ie. CityMapper)	Monetary transport cost
Game playing (ie. Chess)	Score (ie. Piece count)

2) Compare Best-First and A* Search

a) Differences between Best-First and A* search strategies

Best-First	**A***
Visits next node based on heuristics function $f(n) = h$ with lowest heuristic value. It does not consider the cost of the path to that next node.	Visits next node based on overall cost function $f(n) = h+g$ with the lowest sum of heuristic and path cost.
Not optimal	Optimal
Not complete	Complete
Uses less memory	Uses more memory

b) How I would change my A* implementation to be a Best-First search instead

i) Base node expansions on the minimum respective heuristic values alone.

ii) If there are no adjacent, unvisited nodes from the current node to be expanded from then the search terminates.

5.4) CONNECT FOUR

<u>Quadrio assumptions:</u>

- Can only place counters into the board from the top
- Possible moves to do in any given turn:
  P, RP, PR, RPR

  Where P = counter placement & R = 90° board rotation left/right

2) Number of possible actions a player can perform during 1 turn

Possible moves: P, RP, PR, RPR
Possible counter placements: Col. 1, Col. 2, Col. 3, Col. 4
Possible rotations: 90° left, 90° right

$$\therefore \#P = 4 \quad \& \quad \#R = 2$$

$$\begin{aligned}\therefore Max.\#\,possible\ actions &= \#P + \#RP + \#PR + \#RPR \\ &= 4 + 2*4 + 4*2 + 2*4*2 \\ &= 36\end{aligned}$$

3) Main challenges for implementing Quadrio over Connect Four with a Twist using Alpha-Beta pruning

For Connect Four:
$Max.\#\,possible\ actions = \#P + \#PR = 4 + 4 = 8$

For any given turn in Connect Four with a Twist a player can do 8 possible actions, in contrast to Quadrio in which a player can do 36 possible actions (over 4 times more).

This thereby implies the total amount of possible games/outcomes in Quadrio is exponentially larger, in effect, meaning that implementing Quadrio with Minimax would have an exponentially larger space and time complexity. However, upon implementation of Alpha-Beta pruning there would a significant reduction in the number of outcomes needed to traverse by Minimax, by the process of pruning unnecessary branches that cannot affect the final outcome.

So, in final, with regards to choosing an algorithm to implement Quadrio, I think using Alpha-Beta pruning would be a practical choice as it is particularly useful for evaluating games with a vast amount of outcomes due to its pruning capabilities.