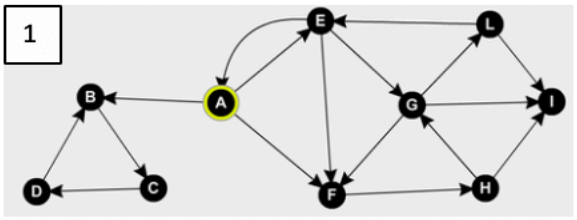
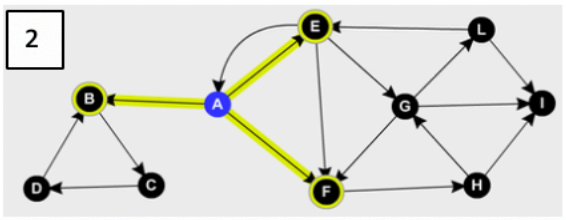
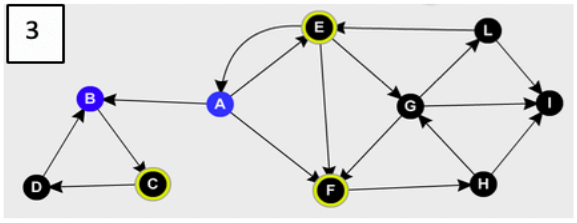
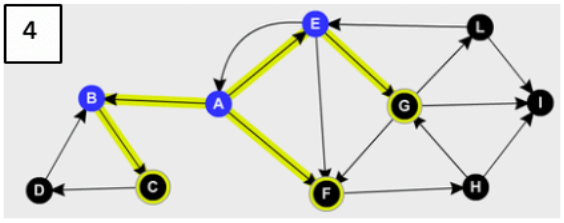
SECTION 3.4: UNINFORMED SEARCH

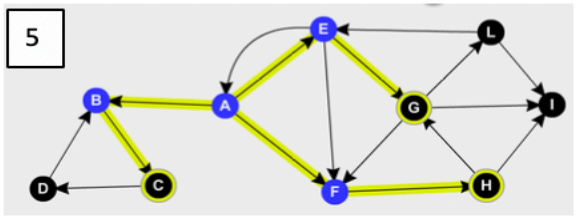
1(a). BFS -- FEEDBACK: Very good

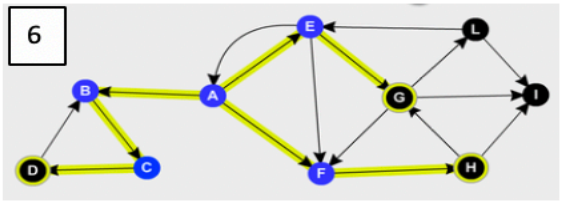


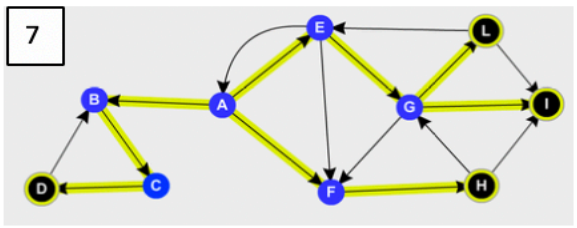


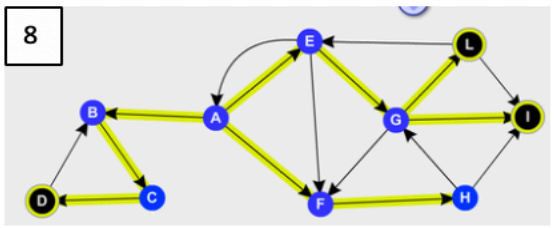


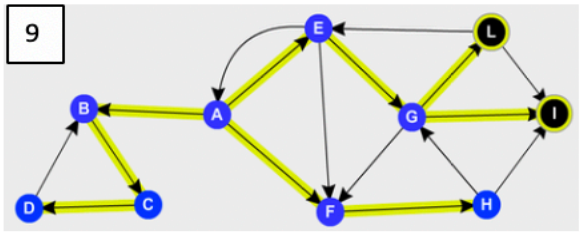


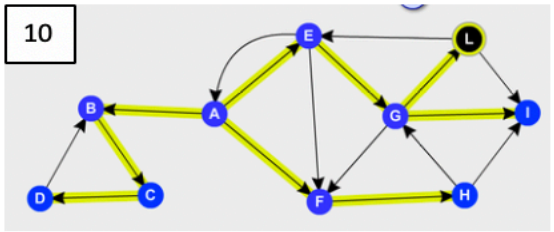


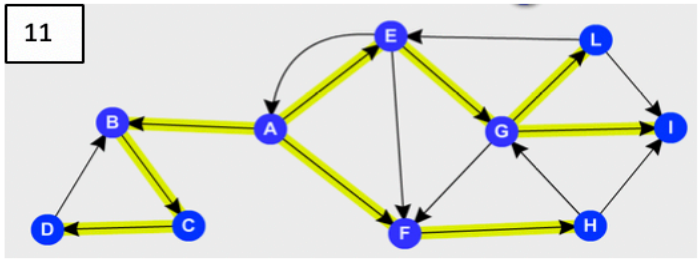








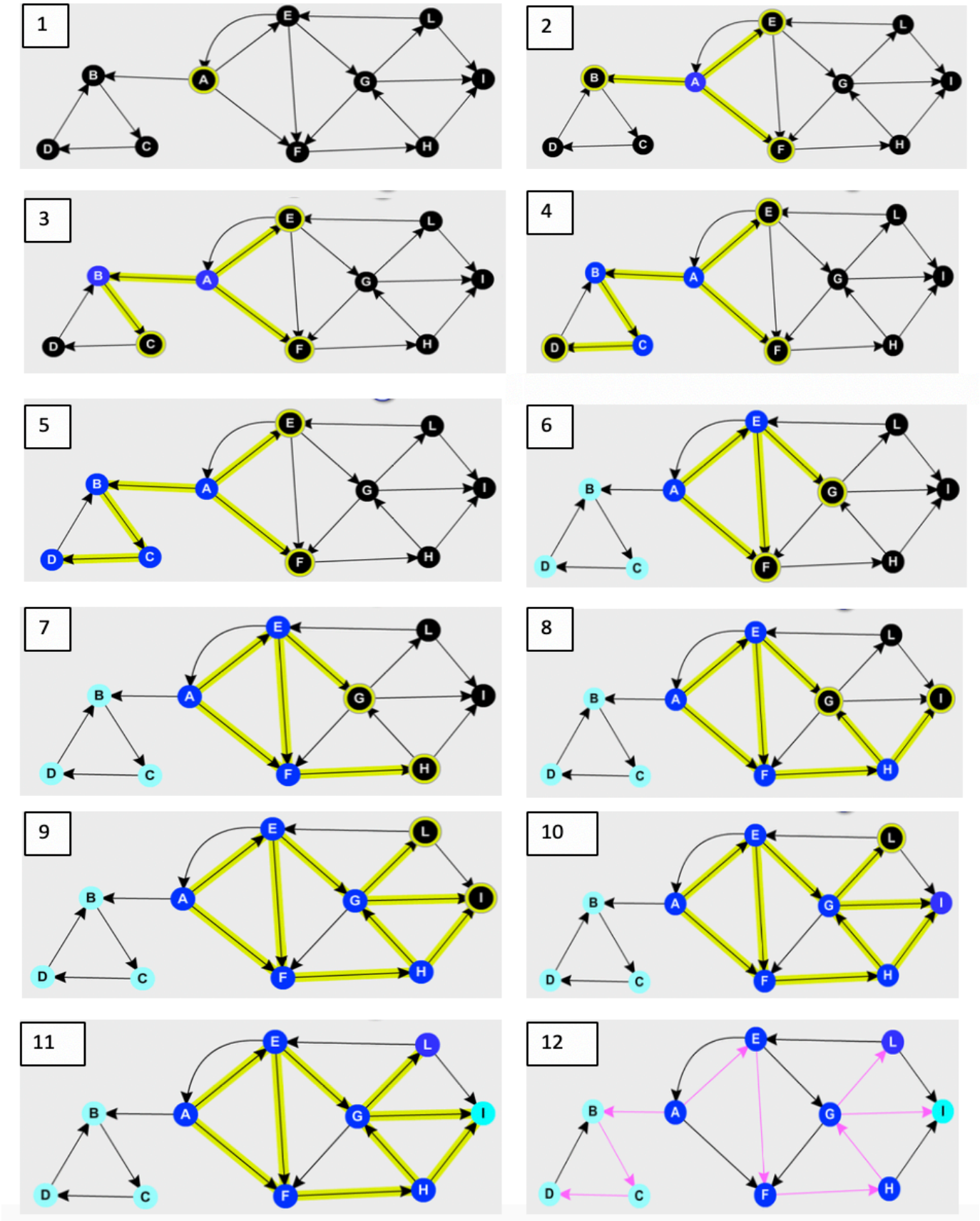




1. A black node indicates an unexplored node.
2. A black node with yellow highlight indicates that it is currently in the frontier.
3. A blue node indicates a node that has been expanded.

- In cases like figure 9, we do not put node ‘B’ in our frontier as it has already been explored before (We know this through the set, *explored*)
- The path we take is: A B E F C G H D I L

1(b). DFS -- FEEDBACK: Very good

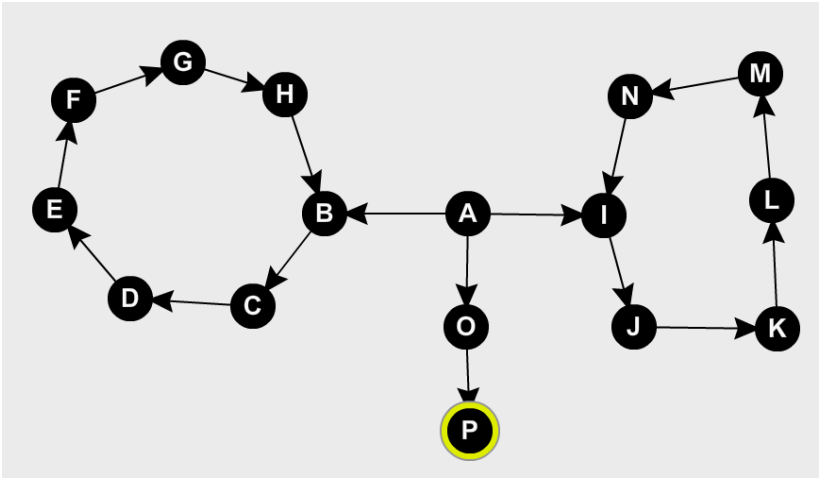


The legend for this is the same as number 1, 2, 3 on previous page.
The light blue nodes are ones that don't lead to a solution.
The last figure (fig 12) shows the paths that we actually take: A B C D E F H G I L

1(c).

-- FEEDBACK: Very good

A graph that favours BFS is one where goal node is closer to starting node. Suppose start node is A and goal node is one with yellow highlight (Node P) and the ordering of the nodes is lexicographic:

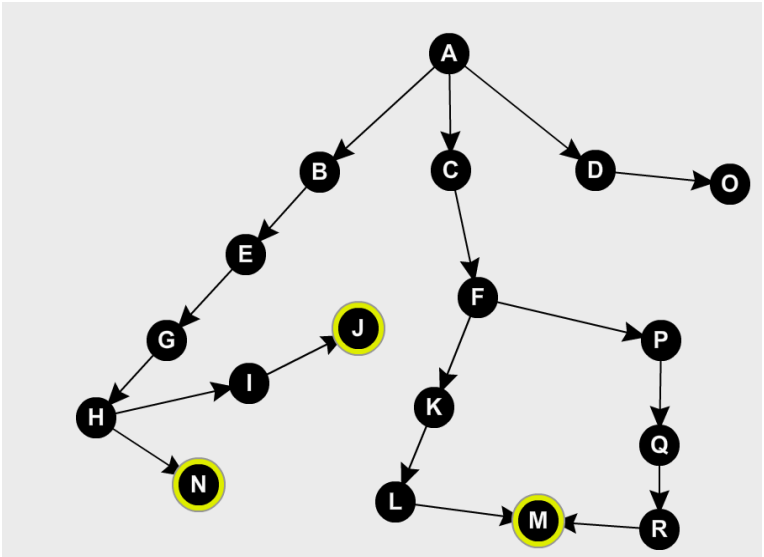


- I chose this graph because goal node is close to start node. If DFS is used, we will explore paths A-B-C-…-H and A-I-J-…- M before exploring A-N-O.

1(d).

-- FEEDBACK: Very good

A graph that favours DFS is one where solutions are frequent but located deep in the graph. Suppose start node is A and goal nodes are ones with yellow highlight and the ordering of the nodes is lexicographic:



- I chose this graph because goal nodes are deep in the graph.

1(e).

-- FEEDBACK: Very good

The main differences between BFS and DFS are:

1. DFS first follows a path from the start to the end node, then another path from start to end, and so on until it founds a goal node or until all nodes are visited, whereas BFS proceeds level by level.
2. To store nodes to our frontier, BFS uses queue data structure, whereas DFS uses stack data structure.
3. Space complexity for DFS is O(bm), which is a clear advantage compared to BFS which is $O(b^d)$. This is because with DFS, once a node has been expanded, it can be removed from memory as soon as all its descendants have been fully explored. In BFS, every node generated is kept in memory.
4. In BFS, a solution is optimal if it is closest to the start node and cost is the same for each possible action. In DFS, a solution is not optimal.

2(a).

-- FEEDBACK: The optimal depth is 5

The optimal depth is 3.

2(b).

-- FEEDBACK: Good, please detail on the scenario where IDS is preferred

- In IDS, we limit the depth of any one path taken by a DFS and gradually increase this limit until a goal is found, ensuring that we never explore any node with distance greater than the limit from the start node. Hence, we are guaranteed an optimal path, whereas in DFS, we are not guaranteed an optimal path.
- This is particularly useful when we have a large search space with a large branching factor, and we want to find an optimal path. With DFS, you may end up searching long paths that contain no solution before finding one. If there is an infinite path, the search might not terminate. With IDS, you can get the optimal path and not search through entire long paths that do not contain a solution. It also avoids being stuck in an infinite path.

SECTION 4.3: INFORMED SEARCH

1(a).

-- FEEDBACK: Very good

Straight line distance is a valid heuristic because it fulfils the criteria, which is that a heuristic is valid if it is an:

1. Admissible heuristic
    a. The straight-line distance never overestimates the cost to reach the goal. In fact, it is always the shortest distance between two points or nodes.
2. Consistent heuristic
    a. The general triangle inequality is satisfied; the SLD between n and n' is no greater than c(n,a,n').

1(b).

-- FEEDBACK: Sort of the same problem just rephrased

1. A delivery robot sending a number of parcels to send to various locations, where each parcel has its own delivery destination, by carrying a parcel (from pick-up location) and deliver it to destination.
    a. Given that the robot can only carry one parcel at a time, a valid heuristic is the sum of distances that the parcels must be carried added to the distance to the closest parcel.
2. Finding the quickest route from a current location to a destination.
    a. A valid heuristic is the minimum of:
        i. The minimum time required to drive directly to destination on local roads and,
        ii. The minimum time required to drive to a freeway on local roads, then drive on freeways to a location nearby the destination, then drive on local roads to the destination.
3. Finding the lowest-cost route from a current location to a destination, where lowest-cost is in terms of fuel and tolls.
    a. Given that the cost function is the distance from current location to destination multiplied by the cost of fuel, a valid heuristic is the total price of tolls paid (if route involves going through tolls).

2(a). ***Differences between (greedy) best-first and A* search:***

-- FEEDBACK: Good job

1. Best-first search visits next state based on the evaluation function f(n) = h(n) where h is the heuristic function. It expands the node with lowest heuristic value; whereas, A* search visits next state based on the evaluation function f(n) = g(n) + h(n), where it takes into account the path cost so far to reach node n, as well as the heuristic function.
2. In general, the path found with best-first search may not be the optimal one, whereas the path found with A* search is guaranteed to be optimal.
3. In general, best-first search is not complete as it can get stuck in infinite loops, whereas A* search is complete (unless there are infinitely many nodes to explore in the search space)

2(b).

-- FEEDBACK: Excellent

I would change the evaluation function to only consider the heuristic function, and disregard the path cost so far to reach node *n.* Therefore, in my code, I would create a new function to sort my search agenda (similar to *sortBranchesByCost*) but this time, *totalCost* do not take into account the path cost.

SECTION 5.4: CONNECT FOUR & QUADRIO

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

-- FEEDBACK: Very good

- There are 3 types of actions that constitute one term: turning, placing a piece and then turning again.
- First rotation gives 4 possible states
- You can then insert a piece in any of the 16 slots.
- Final rotation gives a further 4 possible states.
- Hence, there are 4*16*4=256 possible actions during one term.

3. *Main challenges for implementing Quadrio over Connect Four with a Twist for alpha-beta pruning.*

-- FEEDBACK: Your answer is correct; a similar implementation of alpha-beta pruning for Quadrio is not practical due to the large search space. However, you should have given more details about the differences in the implementation and about the feasibility of running such an implementation on a regular computer: for example, how much time would such a computation require on a normal machine, or what sort of computer architecture would we need to actually be able to run it in a reasonable amount of time; you could have also named some concrete optimizing changes to the code.

- Unlike Connect Four with a Twist, Quadrio has 2 'sides' of the board, so there are actually 2*16=32 possible slots.
- Since the physical board is transparent, this could be difficult to implement in code due to 'line-of-sight'.

Would such an implementation be practical? Give reasons for your decision.

- Such an implementation would not be practical because the search space will be extremely large.
- The branching factor will be much larger because one term can generate up to 256 nodes.