Family Name: ..... **Other Names:** .....

Student ID: ..... **Signature** .....

COMP 261 : Test 2

6 May 2025

Instructions

- Time allowed: **50 minutes**
- Write your name, student ID, and sign the top of the front page.
- Write your student ID on the top of every other page.
- Attempt **all** the questions. There are 50 marks in total.
- Write your answers in this test paper and hand in all sheets.
- If you think a question is unclear, ask for clarification.
- This test contributes 20% of your final grade.
- This is a closed book test.
- You may use dictionaries and calculators.
- You may write notes and working on this paper, but make sure your answers are clear.

Questions	**Marks**	
1. Adjacency Matrix and Adjacency List	[13]	
2. Shortest Paths	[15]	
3. Strongly Connected Components	[10]	
4. Articulation Points	[12]	
	TOTAL:	

Question 1. Adjacency Matrix and Adjacency List **[13 marks]**

Suppose we have a simple undirected graph with N nodes, E edges, and a maximum degree of Δ.

(a) **[2 marks]** What is the Big-O time cost of finding all neighbours of a given node using *Adjacency Matrix* and *Adjacency List*?

Adjacency Matrix: Adjacency List:

(b) **[2 marks]** Given node indices i and j, and we want to check if there is an edge between node i and node j, what is the Big-O time cost for both *Adjacency Matrix* and *Adjacency List*?

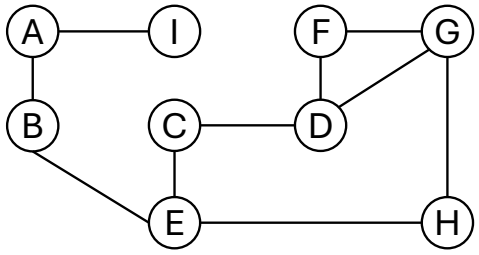
Adjacency Matrix: Adjacency List:

(c) **[1 mark]** True or false: if the graph is very sparse, then *Adjacency Matrix* has lower *space* complexity than *Adjacency List*. **Hint:** A sparse graph does not have many edges relative to the number of nodes in the graph.

(d) **[1 mark]** True or false: when traversing a graph using depth first search, *Adjacency List* has lower time complexity than *Adjacency Matrix*.

(e) **[1 mark]** Briefly explain why an *Adjacency Matrix* is not well suited for a multi-graph.

In the graph below, the nodes are labelled as A, B, ..., I. The edges are all undirected.



Please note that we use the labels (A-I), instead of integers (0-8), to represent the nodes in the following questions.

(f) **[5 marks]** Fill in the *Adjacency Matrix* of the above graph. You can use the number 1 to indicate edges.

	A	B	C	D	E	F	G	H	I
A									
B									
C									
D									
E									
F									
G									
H									
I									

(g) **[1 mark]** What is the maximum degree of the above graph?

Question 2. Shortest Paths **[15 marks]**

(a) **[6 marks]** Both of the shortest path algorithms in the lectures (Dijkstra's algorithm and A*) construct a map of backpointers as they search. The map records the edge by which the search got to each node it visited.

Suppose the search algorithm produced the following backpointers Map:

- nodes are represented by capital letters (like G)
- edges are represented by pairs of letters (like A-B, the edge from node A to node B).

Backpointers:

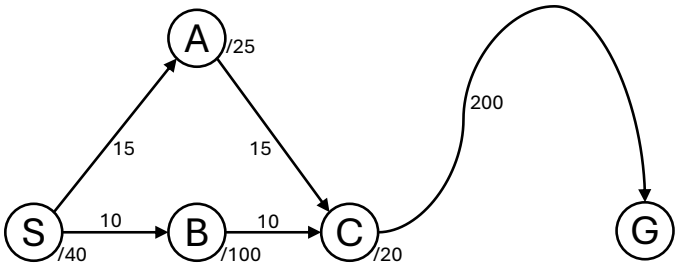
A → S-A	D → F-D	G → I-G	J → S-J
B → K-B	E → H-E	H → J-H	K → D-K
C → A-C	F → S-F	I → B-I	L → J-L

Using this map, reconstruct the path of nodes from the start node S to the goal node G.

S G

(b) A consistent (monotonic) heuristic used by A* ensures that when we visit a node, it must be the best path to that node. This question is in three parts:

1) **[3 marks]** Generally explain what issue occurs in A* search with an inconsistent heuristic.
2) **[2 marks]** In the following graph, identify which node has the inconsistent estimate, and briefly describe how it is inconsistent.
3) **[4 marks]** In the following graph, explain how the inconsistent heuristic affects the A* algorithm. You don't need to explain every step of the A* algorithm, but your description should include relevant examples of when nodes are added to the fringe and when nodes are visited to support your explanation (**tip:** if you are unsure, you can still get partial marks for writing out some steps to the A* algorithm, which may also help you identify the problem with the inconsistent heuristic). Use the following format: $\langle node, prevNode, costSoFar, estimatedTotalCost \rangle$, where the fringe is a priority queue prioritises based on *estimatedTotalCost*.

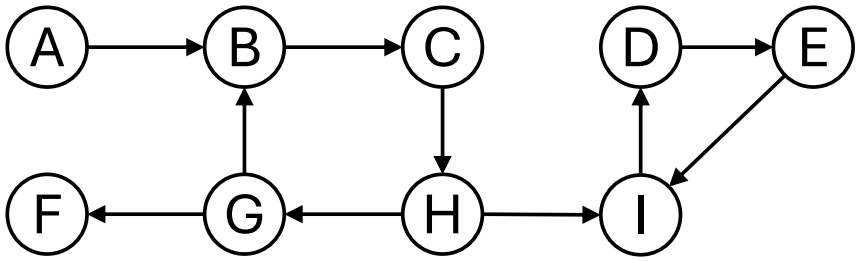


Note that the number by the edge is the edge length. The number with the forward slash by each node is the node's estimated remaining distance to the goal.

Provide your answer to the three questions in the following box:

Question 3. Strongly Connected Components. **[10 marks]**

Suppose we are using Kosaraju's algorithm described in lectures to search for the strongly connected components in the graph below.



The algorithm is given on the facing page for your reference.

The start node is A. The neighbours of a node are enumerated in *alphabetical order* (i.e., when a node has multiple neighbours, you select the neighbouring node based on which comes first in the alphabet - e.g., from H, G is visted before I when traversing outgoing neighbours).

Alphabet: A B C D E F G H I

Show how it works by the following:

List the nodes in the order they are visited:

List the nodes in the nodeList in the order they are added:

In the following table, write the corresponding component number beneath the node label:

A	B	C	D	E	F	G	H	I

Rewrite each component as a list of nodes. The nodes in each component must be listed in the order their component numbers are assigned.

```
Kosuraja(graph):
        for each node in graph:
                node.component ← −1   // initialize nodes to not be in a component
        componentNum ← 0
        nodeList ← empty list;
        visited ← empty set

        for each node in graph:
           if node is not visited then
                   // traverse graph from node forward along edges, adding nodes to nodeList in post−order
                   ForwardVisit(node, nodeList, visited )



        for each node in nodeList in reverse order:
           if node.component = −1 then
                   // traverse graph from node backward along edges, marking nodes with the component number
                   BackwardVisit(node, componentNum)
                   componentNum++

// Search forward from node, putting node on nodeList after visiting everything it can get to.
ForwardVisit(node, nodeList, visited ):
        if node is not in visited then
                add node to visited .
                for each neighbour in node.outNeighbours:
                        ForwardVisit(neighbour, nodeList, visited )
                add node to nodeList

// Search backwards from node, marking all the nodes that can get to it as the same component
BackwardVisit(node, componentNum):
        if node.component = −1 then
                node.component ← componentNum
                for each backNeighbour in node.inNeighbours:
                        BackwardVisit(backNeighbour, componentNum).
```

Question 4. Articulation Points **[12 marks]**

Find the articulation points using the Recursive Articulation Point Algorithm described in lectures. The start node is A, whose depth and reachBack are set to 0.

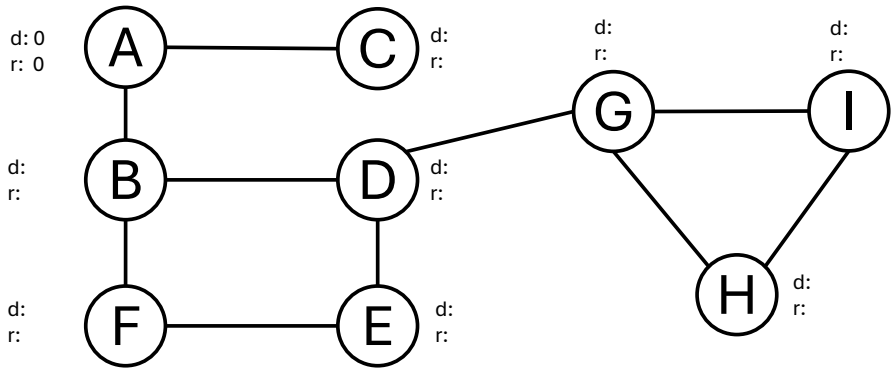
- Calculate the depth (d) of each node.
- Calculate the reachBack (r) value of each node.
- List the articulation points in the order they are identified.

The algorithm is given on the facing page for your reference.

The start node is A.

The neighbours of a node are enumerated in *alphabetical order* (i.e., when a node has multiple neighbours, you select the neighbouring node based on which comes first in the alphabet).

Alphabet: A B C D E F G H I



List the Articulation Points in the order they are identified:

```
FindArticulationPoints (graph, start ):
    for each node:
        node.depth ← −1
    articulationPoints ← emptyset
    start .depth ← 0
    numSubtrees ← 0
    for each neighbour of start
        if neighbour.depth = −1 then
            RecursiveArtPts(neighbour, 1, start )
            numSubtrees ++
    if numSubtrees > 1 then add start to articulationPoints

RecursiveArtPts(node, depth, fromNode):
    node.depth ← depth
    reachBack ← depth
    for each neighbour of node other than fromNode
        if neighbour.depth ≠ −1 then
            reachBack ← min(neighbour.depth, reachBack)
        else
            childReach ← RecursiveArtPts(neighbour, depth + 1, node)
            if childReach ≥ depth then add node to articulationPoints
            reachBack ← min(childReach, reachBack )
    return reachBack
```

* * * * * * * * * * * * * * *

SPARE PAGE FOR EXTRA ANSWERS

Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.