Graph Algorithms Midterm Exam – 28/10/2025

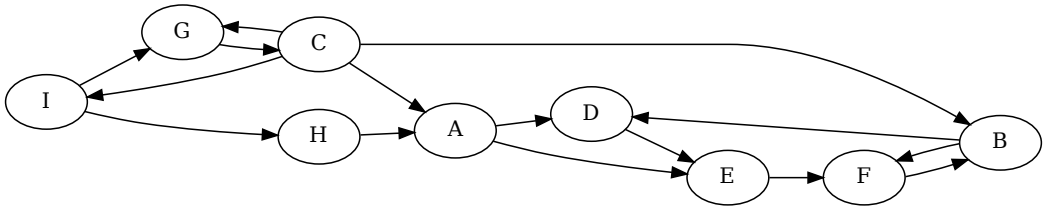
Guidelines

- You may use all algorithms and theorems we have seen in class (including the TDs) without proof. Other than that, you need to provide full justifications for all your answers.
- When asked to design an algorithm, you may do so in **pseudocode** or in **natural language**. Using a programming language (e.g. Python) is **unnecessary** and strongly discouraged.
- Provide proofs of correctness for your algorithms.
- Provide complexity analyses for your algorithms.
- Upper bounds on the running times of your algorithms must be given in terms of n (for adjacency matrix representation) or n, m (for adjacency list representation).
- **Algorithmic efficiency** is a major criterion in grading your answer. However, sub-optimal but correct algorithms may receive partial credit (depending on performance).
- An appendix with a reminder of some of the algorithms we have seen in class is given in the end.
- In your analysis, you may assume that arithmetic operations take $O(1)$ time, **except for exercise 5**, where you need to supply an upper bound on the cost of the arithmetic operations of your algorithm.
- **Write clearly!**
- **Don't panic!**
- **Good luck!**

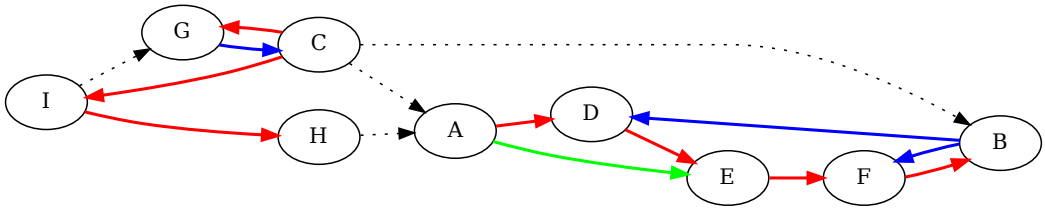
1 DFS and SCC algorithm execution (5 points)

Consider the digraph given below, where we assume that all vertices and adjacency lists are ordered alphabetically. This exercise relies on the DFS algorithm, a reminder of which is given in the appendix.

1. Compute the output of DFS on this digraph. You should give a table with the discovery and finish times of all vertices. Furthermore, you should give a list of tree arcs, a list of backward arcs, a list of forward arcs, and a list of cross arcs. (3 points)
2. As you recall, executing DFS is the first step of the algorithm we saw in class for computing strongly connected components. Show how the execution of this SCC algorithm continues on the given digraph and the strongly connected components it outputs. (2 points)



Solution:



Times:

Vertex	Discovery	Finish
A	1	10
B	5	6
C	11	18
D	2	9
E	3	8
F	4	7
G	12	13
H	15	16
I	14	17

We will now execute DFS using ordering C,I,H,G,A,D,E,F,B, on the graph obtained by reversing all arcs. This gives the components: (C,G,I); (H); (A); (D,B,F,E); where vertices are ordered in order of their discovery time.

2 Disconnecting by Removing (3 points)

For each of the following statements, indicate if it is true or false and give an appropriate justification (proof or counter-example).

1. If $G = (V, E)$ is an undirected connected graph, $v \in V$ a vertex, then $G - v$ has at most 2 connected components.
2. If $G = (V, E)$ is an undirected connected graph, $e \in E$ an edge, then $G - e$ has at most 2 connected components.
3. If $G = (V, A)$ is a strongly connected directed graph, $e \in A$ an arc, then $G - e$ has at most 2 strongly connected components.
4. If $G = (V, E)$ is an undirected graph and $e \in E$ an edge contained in a cycle, then e is not a bridge.
5. If $G = (V, E)$ is an undirected graph and $v \in V$ a vertex contained in a cycle, then v is not an articulation point.
6. If $G = (V, E)$ is an undirected graph, $v_1, v_2 \in V$ are articulation points, and v_1, v_2 are adjacent, then $v_1 v_2$ is a bridge.

In all cases, $G - v$ or $G - e$ indicates the graph obtained after removing v or e respectively from G. Recall that a bridge is an edge e such that $G - e$ has more components that G, and an articulation point is a vertex v such that $G - v$ has more components than G. (0.5 points per question)

Solution:

1. False. For example, consider a star $K_{1,3}$ with three leaves attached to a central vertex.
2. True. To see this, observe that adding an edge to a disconnected graph may only decrease the number of components by 1, because the edge has two endpoints, so can touch at most two components.
3. False. Consider a directed path $1 \to 2 \to 3 \to 4 \to 5$ and the are $5 \to 1$. Removing this arc turns a strongly connected digraph into a digraph with 5 strongly connected components.
4. True. If $e = xy$ is in a cycle, then there is a path $x \to y$ avoiding e. Hence, any path in the original graph which used e can be turned into a path in the new graph by using the other edges of the cycle instead of e.
5. False. Take a cycle and attach a leaf to one of its vertices. That vertex is now an articulation point.
6. False. As before, take a cycle and attach leaves to two consecutive vertices of the cycle. The cycle edge connecting them is not a bridge.

3 2-Kings (4 points)

A 2-king in a directed graph $G = (V, A)$ is a vertex v such that for all other vertices u there is a path of length at most 2 from v to u. In other words, v 2-dominates a vertex u if $vu \in A$ or there exists w such that $vw, wu \in A$; and v is a 2-king if it 2-dominates every other vertex.

1. Give an efficient algorithm that finds a 2-king in a given digraph (or decides correctly that none exists), assuming the digraph is given in adjacency matrix form. (1 point)

2. Same as before, but assume that the digraph is given in adjacency list form. (1 point)

3. Suppose that we are given as input a digraph $G = (V, A)$ where for each $u, v \in V$ exactly one of the arcs uv or vu is in A (in adjacency matrix form). Give an algorithm for finding a 2-king in this case which is more efficient than your general algorithm. Prove that a 2-king always exists for this class of digraphs. (2 points)

Solution:

1. We would like to compute the out-degree of every vertex in the graph G^2, which is the graph obtained by G if we add arcs between all pairs x, y such that there is a path of length at most 2 from x to y in G. For the algorithm that computes G^2 see exercise 3 of TD1. In adjacency matrix form this can be done in time $O(n^\omega)$, where ω is the matrix multiplication constant.

2. Similarly, exercise 3 of TD1 gives an algorithm with complexity upper-bounded by $O(nm)$.

3. We observe that in this case the vertex of maximum out-degree is a 2-king. **Proof:** Suppose x has maximum out-degree but there exists y such that $xy \notin A$ and also there is no path of length 2 from x to y. Because for each pair of vertices have an arc we observe that (i) y has an arc to every outneighbor of x (otherwise we would have a path of length 2) (ii) y has an arc to x. But then, y has strictly larger out-degree than x, contradiction.

   Given the above, the algorithm simply finds the vertex of maximum out-degree, which can be done in $O(n^2)$ time in adjacency matrix form.

4 Connectivity-Preserving Orientation (4 points)

We are given a connected undirected graph $G = (V, E)$. An orientation of G is a digraph obtained by replacing every edge $uv \in E$ with either the arc uv or the arc vu (but not both). A connectivity-preserving orientation is an orientation with the property that the resulting digraph is strongly connected.

1. Give an example of a connected graph G on 5 vertices where no connectivity-preserving orientation exists. (1 point)

2. Give an algorithm that takes as input a connected undirected graph G in adjacency list form and outputs a connectivity-preserving orientation or correctly decides that none exists. (3 points)

Solution:

1. A path on 5 vertices: any orientation produces a DAG, which is not strongly connected.

2. We prove the following claim: G admits a connectivity-preserving orientation if and only if G contains no bridge. As we have seen an algorithm for computing all bridges (TD4) in linear time, using the articulation-point finding algorithm we saw in class, we can then execute this to decide if an orientation exists.

Proof of claim: If G has a bridge e, such that $G - e$ has components C_1, C_2, then G admits no connectivity-preserving orientation, as no matter how we orient e, it will be impossible to go from one component to another in some direction.

If G has no bridge, we can obtain a connectivity-preserving orientation as follows: perform DFS from some vertex s, orient all tree edges away from s (i.e. in the direction they were used in DFS) and all backward edges towards the ancestor (recall that no forward or cross edges exist, as G is undirected).

We now observe that every vertex x has a path to all its descendants (since we oriented tree edges away from the root), hence in the oriented graph there is a path from s to all vertices. What remains is the converse: there is a directed path from every vertex to s. If we establish this, the orientation gives a strongly connected graph.

We will show that if x is the parent of y, then our orientation has a directed path from y to x. Applying this claim inductively then implies that all vertices can reach all their ancestors, therefore all vertices can reach s, as desired.

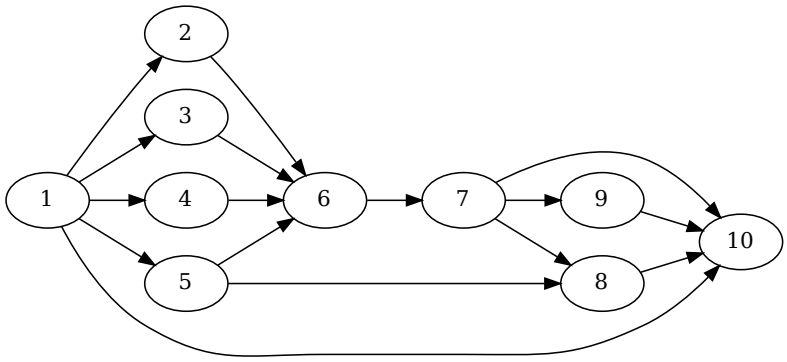
Observe now that xy is not a bridge in G (by assumption). So, G contains an edge between some descendant of y and the rest of the graph. Since G is undirected, this edge is a backward edge, that is, its other endpoint is a proper ancestor of y, hence, an ancestor of x. We have already seen that there is a path from all ancestors of x to x, hence, this backward edge together with the tree edges forms a $y \to x$ path.

Observe that the above imply a linear-time algorithm for orienting the graph: run DFS, orient tree edges away from the root and backward edges toward the ancestor.

5 Most critical arc (4 points)

We are given as input a DAG $G = (V, A)$, in adjacency list form, with the vertex set topologically ordered so that $V = \{1, \ldots, n\}$. The **most critical arc** of G is defined as the arc $ij \in A$ such that removing ij from G **minimizes** the number of distinct paths from 1 to n.

1. What is the most critical arc of the graph below? Why? (1 point)
2. Design an efficient algorithm that takes as input a DAG G topologically ordered $\{1, \ldots, n\}$ and outputs the most critical arc of G (or one of the most critical arcs, if there is a tie). (3 points)



Solution:

The most critical arc is $6 \to 7$, as there are 12 paths crossing it, but removing it leaves a graph with only 2 paths from 1 to 10.

We execute the algorithm of TD4 which computes for each i the number of paths from 1 to i, call this $P_1[i]$. By inverting the graph and using the reverse topological ordering (or by slightly modifying the algorithm) we compute for each j the number of paths from j to n, call this $P_2[j]$.

We now go through each arc $ij \in A$ and observe that if we remove ij from G we will destroy $P_1[i] \cdot P_2[j]$ paths, as this is the number of paths $1 \to n$ that use the arc ij. We therefore compute this number for each $ij \in A$ and select that arc which maximizes this value. Removing it will minimize the number of $1 \to n$ paths in the resulting DAG.

As we have seen in TD4, the above can be done in $O(n + m)$ arithmetic operations. However, arithmetic operations are not $O(1)$ time, as the numbers involved can be exponential in n. For the first part of the algorithm, we perform addition on numbers with $O(n)$ bits, so this adds a factor of $O(n)$ to the complexity giving time $O(n(m + n))$ for the first part.

For the second part, we perform m multiplications of two numbers with $O(n)$ bits. Multiplying two integers with b bits can be done in $O(b^2)$ time, using the method we learned in primary school. It can also be done in $O(b \log^{O(1)} b)$ time, using more advanced techniques (FFT). Using the primary school algorithm is considered a correct answer here, since we have not discussed faster algorithms. This adds an n^2 factor to the running time. So, the total running time is $O(n^2(m + n))$.

Note: an alternative algorithm would be to try every arc, remove it, and execute the algorithm of TD4. This would give $O(mn(m + n))$ time (m executions of an $O(n(m + n))$ time algorithms. This is worse than the multiplication-based algorithm given above, as $mn > n^2$ (except for some boring cases, like graphs with many isolated vertices), so this algorithm, though correct, does not receive full points.

A Algorithms

A.1 DFS

procedure DFS(G)
  for $v \in V \setminus \{s\}$ **do** ▷ initialize
    Color v White, Parent of $v \leftarrow$ NULL
  end for
  $t = 0$ ▷ universal time variable
  for $v \in V$ **do**
    if v is White **then**
      DFS-Visit(G,v)
    end if
  end for
  procedure DFS-VISIT(G,u)
    $t++$
    $u.d = t$, Color u Gray
    for $v \in N(u)$ **do**
      if v is White **then**
        Set Parent of v to be u
        DFS-Visit(G,v)
      end if
    end for
    $t++$
    $u.f = t$, Color u Black
  end procedure
end procedure

A.2 BFS

for $v \in V \setminus \{s\}$ **do** ▷ initialize
  Color v White, Parent of $v \leftarrow$ NULL, $\mathrm{dist}(s, v) \leftarrow \infty$
end for
$\mathrm{dist}(s, s) \leftarrow 0$, Parent of $s \leftarrow$ NULL, Color s Gray
Enqueue(s) ▷ end of initialization
while Queue not empty **do** ▷ main loop
  $v \leftarrow$ Dequeue
  for each $u \in N(v)$ **do**
    if u is White **then**
      $\mathrm{dist}(s, u) \leftarrow \mathrm{dist}(s, v) + 1$, v becomes parent of u
      Color u Gray
      Enqueue(u)
    end if
  end for
  Color v Black
end while