Algorithms

Assignment Solutions: Graphs

Name: ____________________________________________________________

Id: ______________________________________________________________

Grade

Good Luck!
1. (i) Draw the BFS tree for the labelled Petersen graph: Start with A and assume that the order among the vertices is A, B, C, D, E, F, G, H, I, J. Classify all the 15 edges in the BFS tree as tree, back, forward, or cross edges.

**Answer:** In the tree below there are nine tree edges (straight and black) and six cross edges (curved and blue). An arrow from vertex X to vertex Y for some X, Y ∈ {A, B, . . . , J} indicates that the edge (X, Y) was explored from X to Y. The children of a vertex are ordered from left to right by their discovery times.

(ii) Draw another BFS tree for the labelled Petersen graph: Start with J and assume that the order among the vertices is J, I, H, G, E, D, C, B, A. Classify all the 15 edges in the BFS tree as tree, back, forward, or cross edges.

**Answer:** In the tree below there are nine tree edges (straight and black) and six cross edges (curved and blue). An arrow from vertex X to vertex Y for some X, Y ∈ {A, B, . . . , J} indicates that the edge (X, Y) was explored from X to Y. The children of a vertex are ordered from left to right by their discovery times.

(iii) The Petersen graph is very symmetric. Support this claim with your trees from the previous two parts.

**Answer:** The structure of any BFS tree is exactly the same regardless of the starting vertex and the order among the other vertices. The root has three children each having two children. The six grandchildren of the root form a cycle in which each one is connected to two of its four cousins that are not siblings.

Note that the above two trees would be perfectly identical if H and I the two children of F would replace their positions.
2. (i) Draw the DFS tree for the labelled Petersen graph: Start with $A$ and assume that the order among the vertices is $A, B, C, D, E, F, G, H, I, J$. Classify all the 15 edges in the DFS tree as tree, back, forward, or cross edges.

**Answer:** In the tree below there are nine tree edges (straight and black) and six back edges (curved and red). An arrow from vertex $X$ to vertex $Y$ for some $X, Y \in \{A, B, \ldots, J\}$ indicates that the edge $(X, Y)$ was explored from $X$ to $Y$.

(ii) Draw another DFS tree for the labelled Petersen graph: Start with $J$ and assume that the order among the vertices is $J, I, H, G, F, E, D, C, B, A$. Classify all the 15 edges in the DFS tree as tree, back, forward, or cross edges.

**Answer:** In the tree below there are nine tree edges (straight and black) and six back edges (curved and red). An arrow from vertex $X$ to vertex $Y$ for some $X, Y \in \{A, B, \ldots, J\}$ indicates that the edge $(X, Y)$ was explored from $X$ to $Y$.

(iii) The Petersen graph is very symmetric. Support this claim with your trees from the previous two parts.

**Answer:** Regardless of the starting vertex and the order among the other vertices, the DFS tree has a path with nine vertices. The tenth vertex is either the child of the ninth and last vertex in the path (creating a path with ten vertices), or the child of the eighth vertex in the path, or the child of the seventh vertex in the path. For a given structure, all the back edges are uniquely determined. See below the three structures.
3. Consider the following weighted Petersen graph:

(a) Below is one of its **minimum** weighted spanning trees whose total weight is 22. The tree is the output of the Kruskal algorithm that considers the edge \((D, I)\) before the edge \((F, H)\) (both have weight 4).

(b) Below is one of its **maximum** weighted spanning trees whose total weight is 42. The tree is the output of the adapted Kruskal algorithm (from problem 4(a)) that considers the edges \((A, B)\) and \((C, H)\) before the edge \((F, H)\) (all three have weight 4).
4. Let $G$ be a weighted connected graph with $n$ vertices and $m$ edges.

(a) Design an efficient algorithm that finds a **maximum weighted spanning tree** in $G$.

What is the complexity of your algorithm as a function of $n$ and $m$?

**Answer 1:** Construct a graph $G'$ that is $G$ with different weights. The weight of any edge $(u, v)$ in $G'$ is the negation of its weight in $G$:

$$w'(u, v) = -w(u, v).$$

Run any minimum spanning tree algorithm on $G'$. The output is a maximum spanning tree in $G$.

The complexity of generating $G'$ is $\Theta(m)$ if $G$ is represented by adjacency lists. Therefore, the complexity of both the Kruskal and the Prim algorithms remain the same.

**Answer 2:** Update the Kruskal algorithm by sorting the edges from the heaviest to the lightest and update the Prim algorithms by maintaining the priority queue in a way that gives higher value more priority. Run the updated versions on $G$ to output a maximum weighted spanning tree.

The complexity of both updated version of the Kruskal and the Prim algorithms remain the same.

**Remark:** The output of either the Kruskal algorithm or the Prim algorithm on $G'$ is the same as the output of the adapted versions of the algorithms on $G$.

(b) Design an efficient algorithm that finds a **minimum** weighted set of edges that intersects all the cycles of $G$. That is, for any cycle of edges in $G$, at least one of the edges belongs to this set.

What is the complexity of your algorithm as a function of $n$ and $m$?

**Answer:** Let $S$ be a set of edges that intersects all the cycles of $G$. Remove these edges from the graph. By definition, the remaining graph has no cycles. Furthermore, the remaining graph is connected because the original graph is connected and there is no reason to add to $S$ an edge that disconnects the graph. Therefore, the remaining graph is a spanning tree of $G$.

The above arguments imply that finding a minimum weighted set $S$ of edges that intersects all the cycles of $G$ is equivalent to finding first a maximum spanning tree $T$ in $G$ and define $S$ as the set of all edges in $G$ that are not in $T$.

If the graph is represented by adjacency lists, then during the run of the maximum weighted spanning tree algorithm, finding all the edges that are not in the spanning tree can be done in $\Theta(m)$. Therefore, the complexity of this algorithm is the same as the complexities of the Kruskal and Prim algorithms.
Example: Consider the following weighted Petersen graph.

Below is the maximum weighted spanning tree from problem 3(b) whose total weight is 42.

Below is a set $S$ of edges that intersects all the cycles of the above weighted Petersen graph whose weight is 12 which is the optimal weight for such a set of edges.
5. The cycle graph \( C_n \) is a graph that has one cycle containing all \( n \) vertices. Describe an efficient algorithm that finds a minimum weighted spanning tree in \( C_n \). The algorithm should be more efficient than the algorithms for general graphs. What is the complexity of your algorithm as a function of \( n \)?

**Algorithm:** Find the heaviest edge in the cycle and omit it. What remains is a path (which is a tree) with \( n \) vertices that spans \( C_n \).

**Correctness:** The algorithm is correct since it is another way to implement the Kruskal algorithm. In the Kruskal algorithm, the edges are considered from the lightest to the heaviest one at a time. An edge is added to the spanning tree as long as it does not close a cycle with previously selected edges. The only cycle in \( C_n \) is the graph itself. Therefore, only the last considered edge would close a cycle and would not be part of the spanning tree. This last edge is the heaviest edge.

**Complexity:** When the graph is represented by adjacency lists, the complexity of finding the heaviest edge is \( \Theta(n) \) because there are exactly \( n \) edges in the cycle. As a result, the complexity of the algorithm is \( \Theta(n) \).

When the graph is represented by an adjacency matrix, the algorithm must examine \( \Omega(n^2) \) entries of the matrix to find the edges of the cycle and then to determine the heaviest edge. Since the algorithm examines at most \( O(n^2) \) entries of the matrix and then find the heaviest in \( \Theta(n) \)-time, it follows that the complexity of the algorithm when the cycle is represented by an adjacency matrix is \( \Theta(n^2) \).
6. Let $G$ be a directed graph that is represented by the adjacency matrix $A$.

- In directed graphs: $A(u, v) = 1$ if the edge $(u \to v)$ exists and $A(u, v) = 0$ otherwise.
- A \textit{tournament} is a directed graph such that for any pair of vertices $u \neq v$, exactly one of the edges $(u \to v)$ or $(v \to u)$ exists.
- An \textit{acyclic} tournament is a tournament with no directed cycles.
- A \textit{source} is a vertex whose in-degree is zero. A \textit{destination} is a vertex whose out-degree is zero.

(a) Describe an algorithm that checks if a directed graph $G$ is a tournament. What is the complexity of your algorithm?

\textbf{Algorithm:} For any pair of indices $1 \leq i < j \leq n$, check if $A(i, j) + A(j, i) = 1$. $G$ is a tournament if and only if the answer is “TRUE” for all pairs of vertices.

\textbf{Correctness:} Since both $A(i, j)$ and $A(j, i)$ could be either 1 or 0, it follows that their sum is either 0, or 1, or 2. If the sum is 1 then exactly one of them is 1, if the sum is 0 then both are 0, and if the sum is 2, then both are 1. By definition, in a tournament all of these sums must be 1.

\textbf{Complexity:} For a graph with $n$ vertices, there are $\binom{n}{2} = \frac{n(n-1)}{2}$ pairs of vertices. Therefore the complexity of the algorithm is $\Theta(n^2)$.

(b) Prove that in an acyclic tournament there is exactly one source and one destination.

\textbf{Proof:} First prove that in any tournament there is at most one source and at most one destination. Then prove that in an acyclic tournament there is at least one source and at least one source. Both claims yield the proof.

(i) Consider first any tournament that is not necessarily acyclic. Assume to the contrary that there are at least two sources and let $u$ and $v$ be two of these sources.

If the edge $(u \to v)$ exists then $v$ is not a source and if the edge $(v \to u)$ exists then $u$ is not a source. Since in a tournament only one of these edges exists, it follows that it cannot be the case that both $u$ and $v$ are sources. A contradiction.

Similar arguments show that there cannot be more than 1 destination.

(ii) Consider now an acyclic tournament. Assume to the contrary that there are no sources in the tournament. Construct the following path with all $n$ vertices starting with an arbitrary vertex $v_0$.

Since $v_0$ is not a source, there exists a vertex, denoted by $v_1$, such that $(v_1 \to v_0)$ is an edge. Let $P_i = (v_1 \to v_0)$ be a path with two vertices.

Assume by induction that for $0 \leq i \leq n - 2$, the path $P_i = (v_i \to v_{i-1} \to \ldots \to v_0)$ with $i + 1$ vertices has been constructed after $i$ rounds. Show how to construct in round $i + 1$ the path $P_{i+1}$.

Since $v_i$ is not a source, there exists a vertex $v$ such that the edge $(v \to v_i)$ exists. Note that $v$ cannot be $v_{i-1}$ because the graph is a tournament. Moreover, $v$ cannot be $v_j$ for some $0 \leq j < i - 1$, since this would close the cycle $(v_j \to v_i \to \ldots v_{j+1} \to v_j)$.

Define $v_{i+1} = v$ and let $P_{i+1} = (v_{i+1} \to v_i \to \ldots \to v_0)$.

After $n - 1$ rounds, this procedure constructs the path $P_{n-1} = v_{n-1} \to \ldots \to v_1 \to v_0$ that includes all the $n$ vertices.

Finally, since $v_{n-1}$ is not a source, it follows that there exists an edge $(v_j \to v_{n-1})$ for some $0 \leq j < n - 1$. This edge would close the cycle $(v_j \to v_{n-1} \to \ldots v_{j+1} \to v_j)$. This is a contradiction because the tournament is acyclic.

Similar arguments show that an acyclic tournament has at least one destination.
(c) Describe an efficient algorithm \((O(n)\text{-time})\) to find the source vertex of an acyclic tournament.

**Algorithm:** If the tournament has only one vertex then trivially this vertex is a source (and also a destination). Assume now that the tournament has \(n \geq 2\) vertices. Denote the \(n\) vertices by \(v_0, v_1, \ldots, v_{n-1}\).

- Let \(s_0 = v_0\) be the first candidate vertex to be the source.
- If \((v_1 \rightarrow v_0)\) is an edge then \(s_1 = v_1\) is the new candidate source and \(v_0\) cannot be the source. Otherwise, \(s_1 = s_0 = v_0\) remains the candidate source and \(v_1\) cannot be a source.
- Assume by induction that for \(1 \leq i < n - 1\), after \(i\) rounds, \(s_i\) is the candidate source and all the other vertices among \(v_0, v_1, \ldots, v_i\) cannot be sources. Show how to define the candidate source \(s_{i+1}\).
- If \((v_{i+1} \rightarrow s_i)\) is an edge then \(s_{i+1} = v_{i+1}\) is the new candidate source and \(s_i\) cannot be the source. Otherwise, \(s_{i+1} = s_i\) remains the candidate source and \(v_{i+1}\) cannot be a source.
- After \(n - 1\) rounds, \(s_{n-1}\) is the candidate source while the other \(n - 1\) vertices cannot be sources. Since an acyclic tournament must have a source, it follows that \(s_{n-1}\) is the source of the tournament.

**Remark:** A similar algorithm can find the destination of the tournament.

**Complexity:** the complexity of each round is \(\Theta(1)\) and therefore the complexity of the algorithm is \(\Theta(n)\).

**Arbitrary tournaments:** Tournaments with cycles can still have a source but some do not. For such tournaments, the algorithm could check if \((s_{n-1} \rightarrow v_i)\) is an edge for all \(0 \leq i \leq n - 1\) such that \(s_{n-1} \neq v_i\). If the answer is always “TRUE”, then \(s_{n-1}\) is the source of the tournament. Otherwise, since the algorithm has already verified that no other vertex could be a source, this tournament does not have a source.
7. Let \( G \) be a graph with \( n \) vertices and \( m \) edges. Describe a greedy algorithm that finds a set \( S \) of vertices with the following two properties:

- \( S \) is independent: There is no edge between any pair of vertices of \( S \).
- \( S \) is dominating: Every vertex that is not in \( S \) has at least one neighbor in \( S \).

What is the complexity of your algorithm if the graph is represented by adjacency lists and what is the complexity of your algorithm if the graph is represented by an adjacency matrix?

**Algorithm:** Initially \( S \) is an empty set. Repeat until \( G \) has no vertices:

- Add an arbitrary vertex \( v \) from \( G \) to the set \( S \).
- Omit from \( G \) the vertex \( v \), all of its neighbors, and all the edges that include the omitted vertices.

**Correctness:**

- \( S \) is an independent set: During the run of the algorithm, when a vertex \( v \) joins the set \( S \) it must be the case that it is not a neighbor of a previously added vertex to \( S \) because these neighbors were omitted from \( G \) before \( v \) is added to \( S \). Moreover, none of \( v \)'s neighbors that are in \( G \) at the time \( v \) joins \( S \) will be added to \( S \) later because they are omitted from \( G \) immediately after \( v \) joins \( S \). Therefore, \( v \) has no neighbors in the output set \( S \). \( S \) is an independent set because this is true for all the vertices in \( S \).
- \( S \) is dominating: Every vertex that is not in \( S \) is omitted from \( G \) because it is a neighbor of at least one of the vertices in \( S \). Otherwise, such a vertex would have been selected to be added to \( S \). Therefore, \( S \) is a dominating set in \( G \).

**Implementation details:** Denote the \( n \) vertices by \( v_0, v_1, \ldots, v_{n-1} \).

- Maintain an array \( B \) of size \( n \). For \( 0 \leq i \leq n - 1 \), during the run of the algorithm, if \( B[i] = S \) then \( v_i \) belongs to \( S \), if \( B[i] = R \) then \( v_i \) was omitted from \( G \), and if \( B[i] = V \) then \( v_i \) is in \( G \).
- Initially, \( B[i] = V \) for all \( 0 \leq i \leq n - 1 \) and let \( p = -1 \).
- Repeat the following procedure until \( B[i] = R \) for all \( p < r \leq n - 1 \):
  - Find the first index \( i > p \) in \( B \) for which \( B[i] = V \).
  - Set \( B[i] = S \) and \( p = i \).
  - For each neighbor \( j \) of \( i \), set \( B[j] = R \). Note that it could be the case that already \( B[j] = R \) but it cannot be the case that \( B[j] = S \).

**Complexity:** When \( G \) is represented by adjacency lists, scanning all the neighbors can be done with complexity \( \Theta(n + m) \) and when \( G \) is represented by an adjacency matrix, scanning all the neighbors can be done with complexity \( \Theta(n^2) \). The rest of the complexity for both data structures is \( \Theta(n) \).

**Observations about the output set \( S \) for some families of graphs:**

- Complete graphs: \( S \) contains only one vertex that dominates the rest of the vertices.
- Null graphs: \( S \) contains all the vertices that are independent.
- Stars: \( S \) could be either the root of the star that dominates its \( n - 1 \) neighbors or all the \( n - 1 \) leaves that are independent and dominate the root.
- Paths: The size of \( S \) can get any value in the range \([n/3], \ldots, [n/2]\).
- Cycles: The size of \( S \) can get any value in the range \([n/3], \ldots, [n/2]\).
The Petersen graph:

In the first round of the algorithm, $A$ is added to $S$ and the vertices $A, B, E, F$ are omitted from the graph. The remaining graph is a cycle with six vertices. Due to symmetry, the remaining graph is always a cycle with six vertices regardless of which vertex is selected by the algorithm.

In the second round of the algorithm, $C$ is added to $S$ and the vertices $C, D, H$ are omitted from the graph. The remaining graph is a path with three vertices. Due to symmetry, the remaining graph is always a path with three vertices regardless of which two vertices are selected by the algorithm.

At this stage, there are two options. If $G$ is added to $S$, then the algorithm terminates with the set $S = \{A, C, G\}$. If instead $I$ (respectively, $J$) is added to $S$, then in the fourth and last round $J$ (respectively, $I$) is added to $S$. For both options, the algorithm terminates with the set $S = \{A, C, I, J\}$.

The unlabelled Petersen graph: The algorithm can output only two types of sets:

- A set containing the three neighbors of a particular vertex.
- A set containing the four neighbors of three disjoint edges that do not share a vertex and do not have another edge that connects two of the six endpoints of these edges.

In the above example, $S = \{A, C, G\}$ contains all the neighbors of $B$ and $S = \{A, C, I, J\}$ contains all the neighbors of the edge $(B, G)$ or the edge $(D, E)$ or the edge $(F, H)$.

The labelled Petersen graph: There are ten possible outputs for the first option in which $S$ contains three vertices one per each vertex. There are five possible outputs for the second option in which $S$ contains four vertices one per each set of three disjoint edges. The total number of possible outputs is 15.
8. A leaf in a graph $G$ (not necessarily a tree) is a vertex that has exactly one neighbor. Let $G$ be a graph with $n$ vertices and $m$ edges. Describe an efficient algorithm that counts the number of leaves in $G$.

What is the complexity of your algorithm if the graph is represented by adjacency lists and what is the complexity of your algorithm if the graph is represented by an adjacency matrix?

Algorithm: For each vertex, check if its degree is exactly 1. Count the number of such vertices.

Complexity:

- $G$ is represented by adjacency lists: For each vertex, it is enough to examine at most two of its neighbors to determine if its degree is 1. Therefore, it takes $\Theta(1)$ time to verify if a vertex is a leaf. Hence, the complexity of this algorithm is $\Theta(n)$.

- $G$ is represented by an adjacency matrix: For each vertex, in the worst case, its entire row in the matrix must be examined to see if it contains only one entry that equals 1. Therefore, the complexity for each vertex is $\Theta(n)$ and for all the vertices the complexity is $\Theta(n^2)$.

To show that any algorithm must have complexity $\Omega(n^2)$ when the graph is represented by an adjacency matrix consider the following instance. Assume that it is given that in the graph each vertex is either a leaf or an isolated vertex with no neighbors. As a result, the entire rows of $n - 1$ vertices must be examined to distinguish between the cases of a null graph and a graph whose single edge is between the last two examined vertices.