Algorithm Design and Analysis
Assignment 2
Deadline: April 15, 2024

1. (25 points) Let $G = (V, E)$ be an undirected connected graph.

   (a) Prove that you can sort the vertices $(v_1, \ldots, v_n)$ such that, for any $k \in \{1, \ldots, n\}$, the subgraph induced by the vertex set $V_k = \{v_1, \ldots, v_k\}$ is connected.
   Answer:
   To prove this statement, we can use an iterative process based on a property of trees obtained from the graph G. Specifically, we can iteratively choose vertices from G to maintain connectivity in the growing subset. Here is a structured proof:
   1. **Initialize the sequence:** Begin by selecting any vertex $v_1 \in V$ to start the sequence. Since G is connected, v_1 can reach any other vertex in G. Initialize $V_1 = \{v_1\}$.
   2. **Constructing the sequence iteratively:** Suppose we have constructed a sequence $(v_1, \ldots, v_k)$ for some $k < n$, where the induced subgraph $G[V_k]$ is connected (where $V_k = \{v_1, \ldots, v_k\}$). We need to show how to pick v_{k+1} to maintain this connectivity property for $V_{k+1} = V_k \cup \{v_{k+1}\}$.
   3. **Selecting the next vertex:** Since G is connected and $G[V_k]$ is connected, there exists at least one vertex $u \in V \setminus V_k$ such that u has an edge connecting it to some vertex in V_k. Choose such a vertex u to be v_{k+1}. Specifically, select v_{k+1} such that there exists an edge $\{v_{k+1}, v_i\}$ where $v_i \in V_k$.
   4. **Connectivity of the extended subgraph:** By the choice of v_{k+1}, $G[V_{k+1}]$ remains connected. The new vertex v_{k+1} is directly connected to at least one vertex in the previously connected subgraph $G[V_k]$, thus preserving connectivity. Therefore, the subgraph induced by $V_{k+1} = V_k \cup \{v_{k+1}\}$ is connected.
   5. **Completion of the process:** Repeat the above step until all vertices of G are exhausted. Since we add one vertex at a time and maintain connectivity at each step, by the time we reach $k = n$, we have constructed a sequence $(v_1, \ldots, v_n)$ where each $G[V_k]$ for $k = 1, \ldots, n$ is connected.

   Thus, we have constructed a sequence of vertices such that the subgraph induced by the first k vertices is always connected for any $k \in \{1, \ldots, n\}$. This concludes the proof.

   (b) Does the claim hold for directed and strongly connected graphs? That is, given a directed graph $G = (V, E)$ that is strongly connected, can we sort the vertices $(v_1, \ldots, v_n)$ such that the subgraph induced by $V_k = \{v_1, \ldots, v_k\}$ is strongly connected for any $k \in \{1, \ldots, n\}$? If so, prove it; if not, provide a counterexample.
   Answer:
   The claim that holds for undirected connected graphs does not necessarily extend

to directed graphs, even if they are strongly connected. In the directed case, the analogous statement would be that one can always find an ordering of vertices such that each leading subsequence induces a subgraph that is also strongly connected. We will explore whether this is possible with a counterexample.

Counterexample:

Consider a directed graph $G = (V, E)$ where $V = \{v_1, v_2, v_3\}$ and $E = \{(v_1, v_2), (v_2, v_3), (v_3, v_1)\}$. This graph forms a directed cycle, and it is strongly connected because there is a directed path from every vertex to every other vertex.

Now, let's attempt to construct a vertex sequence $(v_1, \ldots, v_n)$ that maintains the strong connectivity in every induced subgraph $G[V_k]$ for $k \in \{1, 2, 3\}$.

1. **Starting with any vertex:** Suppose we start with v_1, so $V_1 = \{v_1\}$. The induced subgraph $G[V_1]$ trivially contains only the vertex v_1 and no edges, so it is vacuously strongly connected.

2. **Adding a second vertex:** Let's add either v_2 or v_3 next: - If we choose v_2 next, then $V_2 = \{v_1, v_2\}$ and $E_2 = \{(v_1, v_2)\}$. This subgraph is not strongly connected because there is no path from v_2 back to v_1. - If we choose v_3 next, then $V_2 = \{v_1, v_3\}$ and $E_2 = \emptyset$ (since there is no direct edge from v_1 to v_3 or vice versa in the original set E). This subgraph is also not strongly connected.

3. **Adding the third vertex:** Regardless of the second choice, adding the third vertex completes the original graph, which is strongly connected. However, the problem arises with ensuring that all earlier subgraphs (specifically with 2 vertices) are strongly connected.

From this analysis, we can see that in this simple case of a directed cycle, it is impossible to order the vertices in such a way that every subsequence up to n forms a strongly connected subgraph. Thus, the claim that works for undirected connected graphs does not hold universally for directed, strongly connected graphs.

This counterexample shows that the desired property is not generally achievable in directed graphs, even if the entire graph is strongly connected.

2. (25 points) Let $G = (V, E)$ be a directed graph and s, t be two vertices. An (s, t)-*Hamiltonian path* is a path from s to t that visits each vertex exactly once. Design a polynomial time algorithm that, given a *directed acyclic graph* $G = (V, E)$ and two vertices s, t as inputs, decides if G contains an (s, t)-Hamiltonian path. Prove the correctness of your algorithm, and analyze its running time.

   Remark: We believe this problem does not admit polynomial-time algorithms on general graphs (that is not necessarily acyclic). In the last chapter of this course, we will prove that this problem is NP-complete.

Answer:
To determine if there exists an (s,t)-Hamiltonian path in a directed acyclic graph (DAG) $G = (V,E)$, we can utilize dynamic programming. The key insight for our approach stems from the properties of DAGs, notably that they have a topological ordering.

Algorithm Description:

1. **Topological Sort:** Compute a topological ordering of the vertices in G. Let this ordering be $v_1, v_2, \ldots, v_n$ where $s = v_1$ and $t = v_n$ (relabel if necessary). If s and t cannot be positioned as such in any topological sort (i.e., if s appears after t), then there is no Hamiltonian path from s to t and we can stop here.

2. **Dynamic Programming Setup:** Define $dp[v]$ as **true** if there exists a Hamiltonian path from s to vertex v in the subgraph induced by $\{v_1, \ldots, v\}$ (according to the topological order), and **false** otherwise.

3. **Initialization:** Set $dp[s] = true$.
4. **Fill the DP Table:** For each vertex v_i (for $i = 2$ to n) in the topological order:

   - Set $dp[v_i] = false$ initially.
   - For each vertex u such that there is a directed edge from u to v_i (i.e., $(u, v_i) \in E$): If $dp[u] = true$, set $dp[v_i] = true$.
   - This means that v_i can be reached via a Hamiltonian path if there's a predecessor u in the topological order which can reach v_i and itself can be reached by a Hamiltonian path from s.

5. **Check for Hamiltonian Path from s to t:** The result is given by $dp[t]$.

Proof of Correctness:
The topological sort ensures that we process each vertex in a way that respects the directed edges. Each $dp[v]$ effectively captures whether there exists a path from s to v covering all intermediate vertices exactly once, by building upon previously established paths to its predecessors.If $dp[t] = true$, by the construction of our DP table, there exists a sequence of vertices leading from s to t that uses all vertices exactly once, as any true value in $dp[v]$ relies on having a valid path to some predecessor u that itself has a valid Hamiltonian path from s.

Time Complexity:

- Topological sorting of the graph takes $O(|V| + |E|)$.
- Initializing the DP table takes $O(|V|)$.
- Filling the DP table requires checking each edge once, which also takes $O(|V|+|E|)$.

Thus, the total time complexity is $O(|V| + |E|)$.

3. (25 points) We have seen in the class that Dijkstra algorithm fails if the graph contains negatively-weighted edges. Consider the following variant of Dijkstra algorithm. Given a directed weighted graph $G = (V, E, w)$ where $w(u, v)$ may be negative, find an integer W such that $w(u, v) + W > 0$ for each edge $(u, v) \in E$, and define the new weight of (u, v) as $w'(u, v) = w(u, v) + W$. Now, $G' = (V, E, w')$ is a positively weighted graph where the weight of each edge is increased by W. Implement Dijkstra algorithm on $G' = (V, E, w')$, and a shortest path from s to every vertex u in G' is also a shortest path in G.

   (a) (10 points) Does this algorithm work for directed acyclic graphs? If so, prove it; if not, provide a counterexample.

   (b) (15 points) A *directed grid* is a directed weighted graphs $G = (V, E, w)$ where the set of vertices is given by $V = \{v_{ij} \mid i = 1, \dots, m; j = 1, \dots, n\}$ ($m, n \in \mathbb{Z}^+$ are two parameters) and the set of directed edges is given by

$$E = \left(\bigcup_{i=1,\dots,m-1; j=1,\dots,n} \left\{ (v_{ij}, v_{(i+1)j}) \right\} \right) \cup \left(\bigcup_{i=1,\dots,m; j=1,\dots,n-1} \left\{ (v_{ij}, v_{i(j+1)}) \right\} \right).$$

   That is, the vertices form a $m \times n$ grid. There is a *directed* edge *from* every vertex *to* the vertex "right above" it, and there is a *directed* edge *from* every vertex *to* the vertex "to the right of" it (unless the vertex is "on the boundary"). The weight of each edge is specified as input and can be negative.

   Does the algorithm work for directed grids? If so, prove it; if not, provide a counterexample.

Answer:
(a): Directed Acyclic Graphs (DAGs)

Question Context:

The proposed variant of the Dijkstra algorithm involves increasing each edge weight by a constant W (let's say $W = 3$) so that all edge weights become positive, making the graph suitable for Dijkstra's algorithm, which fails with negative edge weights. The idea is to run Dijkstra's algorithm on this adjusted graph G'.

Analysis:

Adding a constant W such that $w'(u, v) = w(u, v) + W$ for all edges (u, v) is meant to transform all weights into positive values. However, this transformation does not preserve the relative path costs between different paths when they contain different numbers of edges.

Example and Explanation:

Consider a graph with vertices and edges such that:

- $w(A, B) = 5$
- $w(A, C) = 1$
- $w(C, B) = 2$

In the original graph G, the shortest path from A to B is via C, totaling a weight of $1 + 2 = 3$. With $W = 3$, the transformed weights become:

- $w'(A, B) = 5 + 3 = 8$
- $w'(A, C) = 1 + 3 = 4$
- $w'(C, B) = 2 + 3 = 5$

Calculating the new path weights:

- Direct path from A to B: 8
- Path from A to B via C: $4 + 5 = 9$

In G', the path directly from A to B is now seen as shorter than the path via C, which contradicts the shortest path in the original graph G. This demonstrates that adding the same constant W to each edge can result in different total increases in path weights due to varying numbers of edges in different paths, thereby changing the shortest path structure.

Key Issue:

An additional issue arises with the practical application of Dijkstra's algorithm in G'. Dijkstra's algorithm does not inherently count or store the number of edges used in each path. It only calculates the cumulative weights based on the updated graph G' and selects the minimum weight paths accordingly. Because the algorithm does not track the number of edges per path, converting back the shortest path distances from G' to those of G by subtracting the weighted sum $W\times$ number of edges fails. The algorithm simply cannot differentiate if changes in path lengths are due to fewer edges or actually shorter distances.

Conclusion:

The approach of uniformly adjusting all edge weights by the same constant W and applying Dijkstra's algorithm does not preserve the original shortest path structure in a DAG when negative weights are involved. This methodology can inadvertently favor paths with fewer edges, potentially misrepresenting the true shortest path dynamics of the original graph. For correctly handling graphs with negative edge weights, algorithms specifically designed to accommodate negative weights, such as the Bellman-Ford algorithm, should be used.

(b): Directed Grids

Question Context:

We are analyzing a directed grid where each vertex connects to the vertex right above it and to the right, unless at boundary conditions. The grid allows only upward or rightward movements, creating structured paths where from any vertex s to another vertex t, the number of edges traversed is consistent across all possible paths. This constant number of edges is a unique feature compared to more arbitrary graphs, influencing the path weight analysis when using the modified Dijkstra's algorithm.

Key Insight:

In directed grids, especially from one vertex s to another vertex t, the number of edges in any path is constant due to the grid's structured nature. Each path from s to t will always have the same number of steps (edges), equal to the sum of the differences in their row and column indices.

Analysis:

Given this consistent number of edges across all paths:

1. **Choosing W:** Select W such that $w(u, v) + W > 0$ for every edge (u, v). This transforms every edge weight in G to be positive in G'.

2. **Transformed Path Weights:** For any path P from s to t in G, its transformed weight in G' is $w'(P) = w(P) + k \times W$, where k is the number of edges in P. Since k is constant for any path from s to t, the path P that was the shortest in G will still be the shortest in G' when the additive constant $k \times W$ is the same for all paths.

3. **Correctness of Modified Dijkstra's Algorithm:** Implementing Dijkstra's algorithm on G' to find the shortest path from s will effectively yield the correct shortest path distance in the original graph G once $W \times k$ is subtracted from the result.

Conclusion:

In the specific case of directed grids, because all paths between any two vertices contain the same number of edges, the transformation that turns all edge weights positive and the subsequent use of Dijkstra's algorithm on the modified graph G' correctly identifies the shortest path distances from s to every other vertex u in G. Thus, the modified algorithm works effectively for directed grids under the given conditions.

4. (25 points) Given a directed graph $G = (V, E)$ where each vertex can be viewed as a port. Consider that you are a salesman, and you plan to travel the graph. Whenever you reach a port v, it earns you a profit of p_v dollars, and it cost you c_{uv} if you travel from u to v. For any directed cycle in the graph, we can define a profit-to-cost ratio to be

$$r(C) = \frac{\sum_{(u,v)\in C} p_v}{\sum_{(u,v)\in C} c_{uv}}.$$

As a salesman, you want to design an algorithm to find the best cycle to travel with the largest profit-to-cost ratio. Let r^* be the maximum profit-to-cost ratio in the graph.

(a) If we guess a ratio r, can we determine whether $r^* > r$ or $r^* < r$ efficiently? Design an algorithm to determine this. Prove the correctness of your algorithm, and analyze its running time.

(b) Based on the guessing approach, given a desired accuracy $\epsilon > 0$, design an efficient algorithm to output a good-enough cycle, where $r(C) \geq r^* - \epsilon$. Justify the correctness and analyze the running time in terms of $|V|$, ϵ, and $R = \max_{(u,v)\in E}(p_u/c_{uv})$.

(Hint: You may want to construct another graph with appropriate edge weights, and use an appropriate shortest path algorithm.)

Answer:

(a): Checking a Given Ratio r

To determine if there exists a cycle in the graph with a profit-to-cost ratio greater than or equal to a given r, we can reformulate the problem into a single-source shortest path problem on a modified graph. Here's how:

1. **Transform the graph** G into a new graph G_r where each edge $(u, v) \in E$ has a new weight $w_{uv}(r) = c_{uv} \cdot r - p_v$,which is derived from rearranging the terms of the equation $p_v - r \cdot c_{uv} \geq 0$ that should hold for a profitable cycle under the ratio r.

2. **Cycle Detection:** In the modified graph G_r, finding a cycle with weight less than or equal to zero implies the existence of a cycle in the original graph where the profit-to-cost ratio is at least r. This translates to finding a negative cycle in G_r, because a negative total weight in G_r indicates a total ratio in the original graph that exceeds r.

3. **Algorithm for Detection:** Use the Bellman-Ford algorithm, which is well-suited for this purpose as it can handle graphs with negative weight edges and can detect negative cycles. The Bellman-Ford algorithm runs in $O(|V| \cdot |E|)$ time.

- If the Bellman-Ford algorithm detects a negative cycle in G_r, then $r^* \geq r$.

- If no negative cycle is detected, then $r^* < r$.

Correctness and Running Time: The correctness of this method relies on the construction of G_r. The transformation $w_{uv}(r) = c_{uv} \cdot r - p_v$ ensures that the weights in G_r

directly correspond to the "profitability" of traversing edge (u, v) under the conjectured ratio r. If the sum of weights around any cycle in G_r is negative, it means the actual sum of profits minus r times the costs in the original graph is positive, confirming that $r^* \geq r$.

The running time of the Bellman-Ford algorithm is $O(|V| \cdot |E|)$, which is polynomial in terms of the size of the graph, making this method efficient.

(b): Finding the Best Approximate Cycle

To find the best approximate cycle within an ϵ of the optimal ratio r^*, we can employ a binary search approach on r using the technique described in part (a).

1. **Initialize:** Set an interval $[r_{\text{min}}, r_{\text{max}}]$ where you expect r^* to lie. A reasonable choice could be $r_{\text{min}} = 0$ and $r_{\text{max}} = R$, the maximum p_v/c_{uv} observed in the graph.

2. **Binary Search:** While $(r_{\text{max}} - r_{\text{min}}) > \epsilon$, perform the following steps:

- Compute $r_{\text{mid}} = (r_{\text{min}} + r_{\text{max}})/2$.

- Use the algorithm from part (a) to check if there's a cycle with ratio at least r_{mid}.

- If such a cycle exists, update $r_{\text{min}} = r_{\text{mid}}$; otherwise, update $r_{\text{max}} = r_{\text{mid}}$.

3. **Result:** The value at r_{min} (or r_{max}) will be your best approximation of r^* within an error margin of ϵ.

Correctness and Running Time: The binary search refines the interval containing r^* by half every iteration. The total number of iterations required to reach an accuracy of ϵ is approximately $\log_2((R-0)/\epsilon)$, which means the overall complexity is $O(|V| \cdot |E| \cdot \log(R/\epsilon))$, assuming that the ratio ranges are well-bounded and log is taken base 2.

This algorithm efficiently approximates the maximum profit-to-cost ratio to within an arbitrary precision specified by ϵ, making it both versatile and powerful for real-world applications where exact calculations might be computationally infeasible.

5. How long does it take you to finish the assignment (including thinking and discussion)? Give a score (1,2,3,4,5) to the difficulty. Do you have any collaborators? Please write down their names here.

   This assignment costs me almost the entire day to finish,with much more thinking time than last assignment.From my perspective,questions above are progressive.Specifically, question 1 and question 2 are close to the contents in class.Question 3 and 4 require me to understand the situation and abstract the examination content.I will give 4 to the difficulty.Though hard,these questions benefit me a lot.And thanks for your preparation and correction.