

CSCE 411

Graph Algorithms

1. Ryan is heading to a birthday party, and along the way he needs to pick up a birthday cake and a present. Of course, as a computer scientist, he wants to find the fastest route to get those two things and then get to the party. You are given:
 - An undirected weighted graph G where the vertices of G are locations and the weight of an edge $\{u, v\}$ represents the number of minutes it takes to get between u and v .
 - A vertex $s \in V(G)$ representing Ryan's starting location
 - A vertex $t \in V(G)$ representing the location of the party
 - A set $C \subseteq V(G)$ representing the locations that sell cake.
 - A set $P \subseteq V(G)$ representing the locations that sell acceptable presents.

Give an efficient algorithm to find the fastest route that Ryan can take to acquire a present and a cake and then get to the party (note that the present and cake can be acquired in either order). Analyze the runtime of your algorithm.

Solution: We construct a graph G' based on G . Informally, G' has 4 layers, each based on a copy of G . The first layer represents when Ryan has neither a cake nor a present, the next represents when Ryan has a present but not a cake, the next represents when Ryan has a cake but not a present, and the last represents when Ryan has a present and a cake. It is possible to move from the no cake layers to the cake layers only at vertices in C and it is possible to move from the no present layers to the present layers only at vertices in P . Our goal is to find the shortest path from s in the first layer to t in the final layer.

Formally, we define $G' = (V', E')$ based on $G = (V, E)$ as follows:

- $V' := \{v_N, v_P, v_C, v_{PC} : v \in V\}$, where v_N, v_P, v_C, v_{PC} are newly created vertices
- E' contains the following edges:
 - For every edge $\{u, v\} \in E$, we have edges $\{u_N, v_N\}, \{u_P, v_P\}, \{u_C, v_C\}$, and $\{u_{PC}, v_{PC}\}$ each with the same weight as $\{u, v\}$
 - For every vertex $v \in P$, we have edges $\{v_N, v_P\}$ and $\{v_C, v_{PC}\}$, each with weight 0.
 - For every vertex $v \in C$, we have edges $\{v_N, v_C\}$ and $\{v_P, v_{PC}\}$, each with weight 0.

We note that $|V'| = 4n$ and $|E'| = 4m + 2|C| + 2|P| \leq 4(m + n)$. Since each vertex and edge of G' can be constructed in constant time, we can construct our new graph in $\Theta(m + n)$ time.

To find the shortest path with a present and cake, we run Dijkstra's algorithm on G' , starting from s_N , and find the shortest path to t_{PC} . Our construction guarantees that any path from s_N to t_{PC} must pass through a present vertex and a cake vertex, while also ensuring that every $s - t$ path which does pass through a present vertex and a cake vertex corresponds to an $s_N - t_{PC}$ path of the same length in G' . The runtime of Dijkstra's on G' is $\Theta(|E'| + |V'| \log |V'|) = \Theta(4(m + n) + 4n \log(4n)) = \Theta(m + n \log n)$, and so this is the overall runtime of our algorithm as well.

2. You want to get together with a childhood friend who lives in a different state. In order to meet up as soon as possible, you decide to meet at an intermediate city. How can you choose an intermediate city that lets the two of you meet as soon as possible, assuming that you both leave right now?

You are given a directed weighted graph G where the vertices are cities and the weight of an edge (u, v) is the time to get from u to v . You are also given a city x which is your location and a city y which is your friend's. Give an efficient algorithm to find the intermediate city that will allow the two of you to meet as soon as possible. Analyze the runtime of your algorithm.

Solution: We run Dijkstra's algorithm twice, once starting from x and once starting from y . This gives us for each vertex $v \in V$ both $\text{dist}(x, v)$ and $\text{dist}(y, v)$. Then we return the vertex v which minimizes $\max(\text{dist}(x, v), \text{dist}(y, v))$, the time it takes for both you and your friend to reach the vertex.

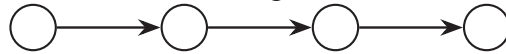
Running Dijkstra's twice takes $\Theta(m + n \log n)$ time. Iterating through all the vertices to find the one which minimizes the maximum of the two distances takes $\Theta(n)$ time. Thus our overall runtime is $\Theta(m + n \log n)$.

3. We say that a pair of vertices u and v in a directed graph G are incomparable if G has neither a $u \rightarrow v$ path nor a $v \rightarrow u$ path.

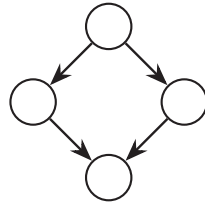
- (a) Give an example of a 4-vertex directed graph with no incomparable pair. Give an example of a 4-vertex directed graph with an incomparable pair where each vertex in the pair has at least one incoming edge and at least one outgoing edge.

Solution:

The graph below has no incomparable vertices: for any pair of vertices the one on the left has a path to the one on the right.



The two middle vertices on the graph below each have an incoming and an outgoing edge, but they are incomparable.



- (b) Give an $O(m + n)$ time algorithm to determine if a directed acyclic graph has an incomparable pair.

Solution: We give the following algorithm:

```

INCOMPARABLE( $G = (V, E)$ )
  Use DFS to compute a topological sort  $v_1, \dots, v_n$  of  $G$ 
  For  $i$  from 1 to  $n$ :
    If  $(v_i, v_{i+1}) \notin E$ :
      Return True
  Return False

```

We need to prove that $\text{INCOMPARABLE}(G)$ returns True if and only if G has an incomparable pair.

- First suppose that G has an incomparable pair, x, y . Let i be the index of x in the topological sort computed by the algorithm and let j be the index of y in the sort (i.e. $v_i = x$ and $v_j = y$). Assume without loss of generality that $i < j$. Since v_i and v_j are incomparable, there is no $v_i \rightarrow v_j$ path in G . In particular, $v_i, v_{i+1}, \dots, v_{j-1}, v_j$ is not a $v_i \rightarrow v_j$ path. So there must be some $i \leq k < \ell \leq j$ such that $(v_k, v_\ell) \notin E$. Therefore, the algorithm will find a missing edge in its for loop and return True.
- Now suppose that G has no incomparable pairs. Let i be an arbitrary integer with $1 \leq i < n$. By the definition of a topological sort, G cannot contain a $v_{i+1} \rightarrow v_i$ path. Since v_i and v_{i+1} are not incomparable, this means that there must be a $v_i \rightarrow v_{i+1}$ path in G . If this path used an edge to a vertex after v_j , it would have to have an edge going from a later vertex in the topological sort back to an earlier one in order to get to j , which is impossible by the definition of a topological sort. Therefore, the $v_i \rightarrow v_j$ path must be just the single edge (v_i, v_j) .

We note that DFS computes a topological sort in $O(m + n)$ time, and iterating through the sort to check for edges between adjacent vertices takes $O(n)$ time, so INCOMPARABLE runs in $O(m + n)$ time.

- (c) Give an $O(m + n)$ time algorithm to determine if an arbitrary directed graph has an incomparable pair.

Solution: We can use Tarjan's algorithm to find all strongly connected components and construct the meta-graph $\text{meta}(G)$ in $O(m + n)$ time. Recall that each vertex of $\text{meta}(G)$ corresponds to a strongly connected component in G . We run the algorithm from part (b) on $\text{meta}(G)$. We note that there is a path between u and v in G if and only if there is a path between their components in $\text{meta}(G)$.

Thus, if $\text{meta}(G)$ has an incomparable pair C_1, C_2 , then any $u \in C_1, v \in C_2$ will form an incomparable pair in G . If $\text{meta}(G)$ has no incomparable pair, then neither does G .