Theory Assignment 2

CSE222: *Algorithm Design & Analysis*

Divyajeet Singh (2021529)

April 9, 2023

Problem-1

The police department in the city of Computopia has made all the streets one-way.

(a) The mayor of the city claims that it is possible to legally drive from an intersection to any other intersection. Formulate this problem as a graph theoretic problem and explain why it can be solved in linear time.

(b) Suppose the mayor's claim was found to be wrong. She has now made a weaker claim: *"If you start driving from the town-hall, navigating one-way streets, then no matter where you reach, there is always a way to drive legally back to the town-hall."* Formulate this weaker property as a graph theoretic problem and explain how it can be solved in linear time.

Input

A directed graph $G = (V, E)$ representing the layout of the city. V is the set of all intersections and E is the set of all one-way streets in Computopia.

Note: *Formulation of the problem as a graph theoretic problem is given in the subsequent subsection.*

Output

(**a**) A boolean value TRUE if the mayor's claim holds, else FALSE.

(**b**) A boolean value TRUE the mayor's weaker claim holds, and FALSE otherwise.

Solution

The two sub-parts of the given problem can both be solved in linear time using the concept of strongly connected components in a directed graph. By definition, a component of a graph G is strongly connected if there exists a path from every vertex in the component to every other vertex in the component. The following sections discuss approaches to solving the two sub-problems in detail.

Formulation as a graph theoretic problem

Since the given problem deals with the layout of a city, it can be naturally solved using graphs. Each intersection in the city can be considered as a vertex, $v \in V$, and each one-way street as a directed edge, $(u \rightarrow v) \in E$, where $u, v \in V$. One-way streets naturally represent directed edges as both can only be traversed in one direction.
In this manner we create a graph $G = (V, E)$ based on the city's layout. [1]

[1] For **Problem 1 (b)**, we assume that the town-hall is located at an intersection, $v_t \in V$.

Problem 1 (a)

The Solution Description

This sub-part can be solved in linear time using the popular Kosaraju's Algorithm [2] to find strongly connected components in a directed graph. According to the mayor's claim, we must be able to legally drive from any intersection from any other intersection in the city. This can be achieved if and only if there exists a path between every pair of vertices in G. This, by defintion, is equivalent to G being strongly connected.
Hence, the mayor's claim would be considered true if and only if G contains exactly one stronly connected component that contains all vertices $v \in V$.

The complete Algorithm and Pseudocode

The complete algorithm consists of the following steps:

1. Use Kosaraju's Algorithm to find all the strongly connected components of G. The algorithm makes use of a DFS traversal on G and G^T, the transposed graph of G.

2. Determine if there is exactly one strongly connected component, say S, and if S contains all vertices $v \in V$.

The pseudocode for the complete algorithm is given in Algorithm (1).

Algorithm 1 An algorithm to check the mayor's stronger claim

```
1: procedure CHECK-CLAIM(G = (V, E))
2:     C ← KOSARAJU-ALGORITHM(G)          ▷ Returns a list of all SCCs in G
3:     if |C| = 1 and |V| = |C[0]| then
4:         return TRUE
5:     else
6:         return FALSE
7:     end if
8: end procedure
```

Runtime Analysis of the Algorithm

The algorithm determines the fallacy of the mayor's claim in linear time with respect to the number of intersections and one-way streets in the city.
It makes use of Kosaraju's Algorithm, which finds all SCCs in a graph in $O(|V| + |E|)$ time.

Hence, dominated by the time complexity of Kosaraju's Algorithm, the time complexity of the algorithm is $O(|V| + |E|)$.

[2] Kosaraju's Algorithm was disucssed in Lecture-12 of the course **CSE222** by Dr Debarka Sengupta, Winter 2023.

Proof of Correctness

It can be proven that the algorithm correctly determines if one can legally drive to any intersection from any other intersection. According to the mayor's claim, there must exist a path from every intersection to every other intersection. Hence, the following cases may arise:

1. If G contains more than one SCC, then there exists a vertex $v \in V$ such that v is not reachable from some vertex $u \in V$. Hence, the mayor's claim is false.

2. If G contains exactly one SCC, but the SCC does not contain all vertices in V, then the mayor's claim is false. In this case, there exists a vertex $v \in V$ such that v is not reachable from any other vertex in G.

3. If G contains exactly one SCC containing all vertices in V, then the mayor's claim is true. In this case, there exists a path from every vertex in G to every other vertex in G.

The algorithm handles all the above cases, and hence correctly determines the fallacy of the mayor's claim.

Problem 1 (b)

Assumption: The town-hall is located at an intersection in Computopia.

The Solution Description

This sub-part can be solved in linear time by using Kosaraju's algorithm again. However, in this case, we must check if the town-hall is a part of a strongly component in G.
According to the mayor's weaker claim, all intersections reachable from the town-hall have at least one way to legally drive back to the town-hall. This can be achieved if an only if the town-hall is a part of a strongly connected component in G.
Hence, the mayor's weaker claim would be considered true if and only if the town-hall is a part of a strongly connected component in G.

The complete Algorithm and Pseudocode

The complete algorithm consists of the following steps:

1. Use Kosaraju's Algorithm to find all the strongly connected components of G. The algorithm makes use of a DFS traversal on G and G^T, the transposed graph of G.

2. Determine if the town-hall is a part of a strongly connected component in G.

The pseudocode for the complete algorithm is given in Algorithm (2).

Algorithm 2 An algorithm to check the mayor's weaker claim

```
1: procedure CHECK-CLAIM(G = (V, E), v_t)
2:     C ← KOSARAJU-ALGORITHM(G)          ▷ Returns a list of all SCCs in G
3:     for i ← 1 to |C| do
4:         if v_t ∈ C[i] then
5:             return TRUE
6:         end if
7:     end for
8:     return FALSE
9: end procedure
```

Runtime Analysis of the Algorithm

The algorithm determines the fallacy of the mayor's claim in linear time with respect to the number of intersections and one-way streets in the city.
It again makes use of the $O(|V| + |E|)$ time Kosaraju's Algorithm. The subsequent checking of whether the town-hall v_t is a part of any of the SCCs takes at max $|V|$ operations. This is because the total number of vertices in all the components combined is, at max, $|V|$.

Hence, dominated by the time complexity of Kosaraju's Algorithm, the time complexity of the algorithm is $O(|V| + |E|)$.

Proof of Correctness

It can be proven that the algorithm correctly determines if the mayor's weaker claim is true. According to the mayor's weaker claim, an intersection reachable from the town-hall must have at least one way to legally drive back to the town-hall. Hence, the following cases may arise:

1. If the town-hall v_t is not a part of any of the SCCs, then the mayor's weaker claim is false. In this case, there exists some vertex $v \in V$ such that v is reachable from the town-hall, but there is no way to legally drive back to the town-hall from v.

2. If the town-hall v_t is a part of a SCC, then the mayor's weaker claim is true. By definition, a SCC contains all vertices that are reachable from each other. Hence, there exists a path from every intersection reachable from the town-hall back to the town-hall.

The algorithm handles all the above cases, and hence correctly determines the fallacy of the mayor's weaker claim.

Problem-2

Assume that n is the number of vertices in a graph. Then, given an edge-weighted connected undirected graph $G = (V, E)$ with $n + 20$ edges, design an algorithm that runs in $O(n)$ time and outputs an edge with the smallest weight, connected in a cycle of G. You must give a justification why your algorithm works correctly.

Input

An edge-weighted connected undirected graph $G = (V, E)$ such that $|V| = n$ and $|E| = n + 20$.

Output

An edge (v, w) with the smallest weight, connected in a cycle of G.

Solution

The given problem can be solved in $O(n)$ time using the concept of cut-edges. We make use of the fact that cut-edges, by definition, are not included in any cycle in a graph, and all non-cut-edges are included in some cycle. Hence, we can find the edge with the smallest weight in G that is not a cut-edge.

Terminologies and Definitions

Definition 1 (Cut-Edge) *An edge $(v, w) \in E$ is a cut-edge if removing it from the graph $G = (V, E)$ increases the number of connected components in G. For a connected graph G, removal of a cut-edge disconnects G.*

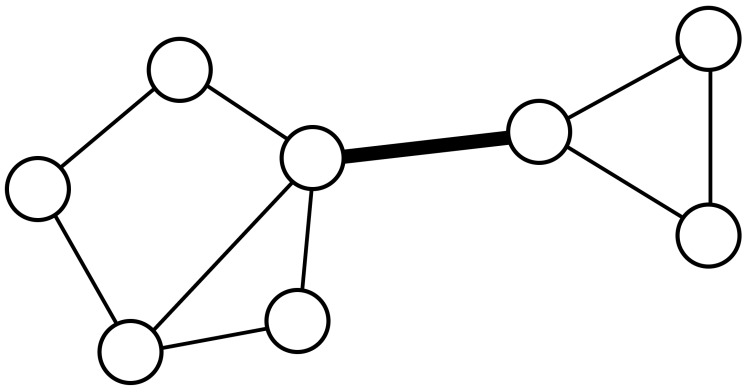

Figure 1: The thick edge is a cut-edge (removing it creates two connected components)

The Solution Description

The solution simply involves finding and marking the cut-edges in G. Once all edges are marked, we can find the edge with the smallest weight in G that is not marked by iterating over E.

Claim 1 *An edge $(v, w) \in E$ is a cut-edge if it is not included in any cycle of G.*

Proof.

> $\Rightarrow$ (If $(v, w) \in E$ is a cut-edge, then it is not included in any cycle in G.)
> We prove the contrapositive of the claim.
> Let $e = (v, w) \in E$ be involved in a cycle. Then, there exists a path from v to w in G that does not contain e, and hence, removing e from G does not disconnect G. So, e is not a cut-edge.
> By contraposition, the forward implication is proved.
>
> $\Leftarrow$ (If $(v, w) \in E$ is not included in any cycle in G, then it is a cut-edge.)
> We prove the contrapositive of the claim.
> Let $e = (v, w) \in E$ be a non-cut-edge. Then, we can safely remove e from G and still have a connected graph, i.e., there still exists a path from v to w in G. So e must be included in some cycle.
> By contraposition, the backward implication is proved.
>
> Hence, the claim is proved. $\square$

The above claim is the key to solving the problem in linear time, as the cut-edges can be identified in G in $O(|V| + |E|)$ time.

The complete Algorithm

The complete algorithm consists of the following steps:

1. Identify all the cut-edges and mark them.
2. Find the edge with the smallest weight in E that is not marked.

Pseudocode for the Find-cut-edges subroutine

To find the cut-edges of G, we use a subroutine FIND-CUT-EDGES(G). The subroutine uses a DFS traversal on the input graph $G = (V, E)$ and returns a hash table C of edges, in which $C[e] = 1$ if and only if e is a cut-edge.

The algorithm uses a DFS traversal on the graph, while maintaining the following data structures:

1. $M[v \in V]$: To keep track of visited vertices.

2. $T_{in}[v \in V]$: To store the time at which a vertex is first visited.

3. $T_{low}[v \in V]$: To store the lowest time of discovery of all vertices reachable from v.

4. $C[e \in E]$: A hash table to mark the cut-edges.

The main idea behind the algorithm is to correctly track and update the discovery times of each vertex v and the lowest discovery time of all its ancestors.

This ensures that if $T_{low}[w] > T_{in}[v]$, then w cannot be reached from any of the ancestors of v. If w was reachable from any of the ancestors of v, then $T_{low}[w] \leq T_{in}[v]$, i.e. it would have already been visited, in which case, $T_{low}[v]$ is updated to $T_{in}[w]$ if it's less than $T_{low}[v]$.
So, $(v, w) \in E$ cannot be a part of a cycle, and hence, must be a cut-edge. [3]

The pseudocode for the subroutine implemented using Tarjan's Algorithm is given in Algorithm (3). [4]

Pseudocode for the complete algorithm

The pseudocode for the complete algorithm is given in Algorithm (4).

Runtime Analysis of the Algorithm

The algorithm makes use of the MARK-CUT-EDGES(G) subroutine, which runs in $O(|V| + |E|)$ time. This is because the subroutine only implements a modified DFS traversal on the graph, with modifications like array lookups and hash table insertions. Since both of these operations take constant time, the total time taken by the subroutine is $O(|V| + |E|)$.

The main algorithm also runs in $O(|V| + |E|)$ time, since it only makes a single call to the subroutine and the subsequent step iterates over the edge-set E while maintaining a minimum edge e_{min}, which takes $\Theta(|E|)$ time.

Hence, the total time taken by the algorithm is $O(|V| + |E|)$.
Since we're given $|V| = n$ and $|E| = n+20$, the overall time complexity of the algorithm is $O(n)$.

[3] A formal proof of correctness of this subroutine is given in the section **Proof of Correctness**.
[4] The subroutine assumes that the graph G is connected and does not contain any self-loops.

Algorithm 3 Tarjan's Algorithm to mark the cut-edges in a connected graph G

```
1: procedure MARK-CUT-EDGES(G = (V, E)):
2:     M[v ∈ V] = T_in[v ∈ V] = T_low[v ∈ V] ← 0
3:     C[e ∈ E] ← 0
4:     t ← 0
5:     procedure DFS(v, p):
6:         M[v] ← 1
7:         T_in[v] = T_low[v] ← t
8:         t ← t + 1
9:         for (v, w) ∈ E do
10:            if w = p then
11:                continue
12:            else if M[w] = 0 then
13:                DFS(w, v)
14:                T_low[v] ← min {T_low[v], T_low[w]}
15:                if T_low[w] > T_in[v] then
16:                    C[(v, w)] ← 1
17:                end if
18:            else
19:                T_low[v] ← min {T_low[v], T_in[w]}
20:            end if
21:        end for
22:    end procedure
23:    DFS(v_0, v_0)                          ▷ v_0 is any arbitrary vertex in V
24:    return C
25: end procedure
```

Algorithm 4 An algorithm to find the least weight edge connected in a cycle of G

```
1: procedure CYCLE-EDGE(G = (V, E)):
2:     C ← MARK-CUT-EDGES(G)
3:     e_min.weight ← ∞
4:     for e ∈ E do
5:         if C[e] = 0 then
6:             if e.weight < e_min.weight then
7:                 e_min ← e
8:             end if
9:         end if
10:    end for
11:    return e_min
12: end procedure
```

Proof of Correctness

The correctness of the algorithm can be proved by a series of the following claims.

Claim 2 *The subroutine correctly identifies all cut-edges in the graph G.*

Proof.

Suppose (v, w) is a cut-edge in G.
By definition, w cannot be reached from any of the ancestors of v. So, $T_{in}[w]$, the discovery time of w, which is equal to $T_{low}[w]$ as it has not been explored yet, must be greater than $T_{in}[v]$, the discovery time of v. All such edges are marked by the subroutine.
Hence, the claim is proved. □

Claim 3 *The subroutine does not mark any non-cut-edge in the graph G.*

Proof.

Suppose (v, w) is a non-cut-edge in G.
By definition, w can be reached from some ancestor of v. So, $T_{low}[w]$, the lowest time of discovery of the vertices reachable from w, must be at most equal to $T_{in}[v]$, the discovery time of v. This is because $T_{low}[w]$ must have been relaxed to $T_{in}[v]$ at some point in the DFS traversal due to presence of a cycle. Hence, the edge (v, w) is not marked by the subroutine.
Hence, the claim is proved. □

Claim 4 *The algorithm correctly identifies the least weight edge connected in a cycle of G.*

Proof.

Suppose e_{min} is the least weight edge connected in a cycle of G.
By definition, e_{min} is not a cut-edge. By Claim (3), e_{min} must not be marked.
Finding the least weight edge among the non-marked edges, hence, identifies the least weight edge connected in a cycle of G.
Hence, the claim is proved. □

Problem3

Suppose that G is a directed acyclic graph with the following features:

- G has a single source s and several sinks $t_1, t_2, \ldots, t_k$.
- Each edge $(v \to w)$ (i.e. an edge directed from v to w) has an associated weight $P[v \to w]$ between 0 and 1.
- For each non-sink vertex v, the total weight of all the edges leaving v is 1, i.e.

$$\sum_{(v \to w) \in E} P[v \to w] = 1 \tag{1}$$

The weights $P[v \to w]$ define a random walk in G from s to some sink t_i. After reaching any non-sink vertex v, the walk follows the edge $(v \to w)$ with probability $P[v \to w]$. All the probabilities are mutually independent.
Describe and analyze an algorithm to compute the probability that this random walk reaches sink $t_i \; \forall \; i \in \{1, 2, \ldots, k\}$.
An example of such a graph is shown in Figure (2). [5]

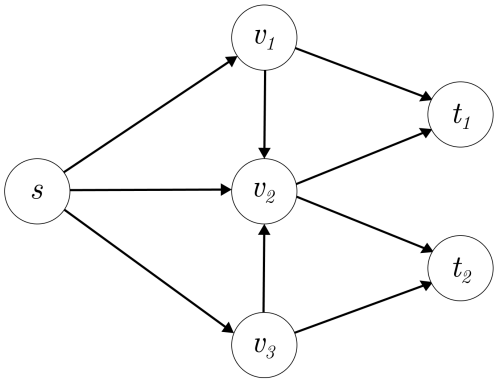

Figure 2: A directed acyclic graph with a single source and several sinks.

Input

A weighted directed acyclic graph $G = (V, E)$ having the above properties.

Output

A sequence of probabilities $P[t_1], P[t_2], \ldots, P[t_k]$, where $P[t_i]$ is the probability that the random walk reaches sink $t_i \; \forall \; i \in \{1, 2, \ldots, k\}$.

[5] The weights are not shown for simplicity.

Solution

The given problem can be efficiently solved by applying dynamic programming on the topologically sorted graph G. It allows for the calculation of the probability of reaching every edge $v \in V$ starting from s.

Notations Used

Notation 1 ($P[v \to w]$) *The weight of the edge $(v \to w) \in E$. This is also the probability of following the edge $(v \to w)$ for the random walk from v.*

Notation 2 ($P[v]$) *The total probability of reaching vertex v from the source s. It also denotes the entry for vertex v in the dynamic programming table.*

Preprocessing

We first obtain a total ordering of the given graph G by topologically sorting it. This can be done using the TOPOLOGICAL-SORT(G) routine covered in class.
The sorting step can be achieved in $\Theta(|V| + |E|)$ time.

Reason for Sorting:
When G is sorted in the aforementioned fashion, then the vertices are ordered such that v appears before w if $(v \to w) \in E$. This allows us to calculate $P[w]$ by using the probabilities $P[v] \; \forall \; (v \to w) \in E$. This is possible because of equation (1) and the fact that the probabilities are mutually independent.

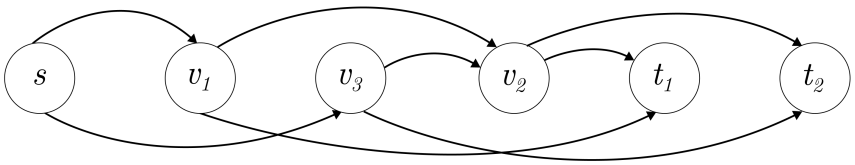

Figure 3: A topologically sorted version of the graph in Figure (2).

Decomposition into Sub-Problems

The given problem of finding $P[t_i] \; \forall \; 1 \leq i \leq k$ is broken down into overlapping sub-problems of the form $P[v]$, where v is not a sink. For each problem $P[v]$, we intend to find the probability of reaching v from s.
Each probability $P[v]$ can be found by adding the probabilities of reaching v from all vertices having an edge directed to v. The existence of $P[u] \; \forall \; (u \to v) \in E$ is guaranteed by the topological ordering.

Hence, this decomposition of the given problem into smaller sub-problems guarantees substructure optimalilty. This is because at any step, we can find the optimal solution using the optimzal solutions obtained in the previous steps.

Recurrence Relation

Let $P[v]$ be the total probability of reaching any vertex v from the source s. Then, the following recurrence relation can be defined:

$$P[v] = \begin{cases} 1 & \text{if } v = s \\ \sum\limits_{(u \to v) \in E} P[u] \times P[u \to v] & \text{otherwise} \end{cases} \tag{2}$$

Using the above recurrence relation, we can find $P[v] \ \forall \ v \in V$ and maintain it in a $|V| \times 1$ dynamic programming table.

Base Case and Solvable Sub-Problems

The base case for the above recurrence relation is $P[s] = 1$. This is because the probability of reaching s from the source is 1, as the random walk starts from s.

The sub-problems that solve the original problem are $P[t_i] \ \forall \ 1 \leq i \leq k$, the probabilities of reaching the sinks t_i from s.

The complete Algorithm and Pseudocode

The complete algorithm consists of the following steps:

1. Topologically sort the graph G to get a total ordering of V.
2. Initialize the dynamic programming table P following the base case.
3. Iterate over the vertices in their topological order and tabulate P using the above recurrence relation.
4. Return $P[t_1], P[t_2], \ldots, P[t_k]$, the total probabilities of reaching the sinks from s.

The pseudocode for the complete algorithm is given in Algorithm (5). [6]

[6]The pseudocode of the algorithm makes use of notations introduced in the section **Notations Used**. Here, $P[v \in V_T]$ denotes a $|V_T| \times 1$ table ordered in the same way as V_T.

Algorithm 5 An algorithm to find the probabilities of reaching the sinks from s

```
1: procedure SINK-PROBABILITY(G = (V, E)):
2:     V_T = TOPOLOGICAL-SORT(G)
3:     P[v ∈ V_T] ← 0
4:     P[s] ← 1
5:     for v ∈ V_T do
6:         for (u → v) ∈ E do
7:             P[v] ← P[v] + P[u] × P[u → v]
8:         end for
9:     end for
10:    return P[t_1], P[t_2], ..., P[t_k]
11: end procedure
```

Runtime Analysis of the Algorithm

The following operations are performed in the algorithm:

1. TOPOLOGICAL-SORT(G): This step takes $\Theta(|V| + |E|)$ time.
2. Initialization of the dynamic programming table P, which takes $\Theta(|V|)$ time.
3. The dynamic programming step takes $\Theta(|V| + |E|)$ time, as it iterates over all vertices, while accessing the edges incident on each vertex. Analogous to a BFS traversal, the total number of iterations of the outer loop is $|V|$ and the inner loop is $|E|$.

Dominated by the sorting step and the dynamic programming step, the time complexity of the algorithm is $\Theta(|V| + |E|)$.

Bibliography

1. **OpenGenius IQ:** Tarjan's Algorithm to find cut-edges in a Graph
2. **USCB Math137A: (Lemma 1**) Cut-edges are not contained in cycles