ADA Assignment 2

Anirudh S. Kumar
Roll Number - 2021517
IIIT - Delhi
anirudh21517@iiitd.ac.in

Vartika
Roll Number - 2021571
IIIT - Delhi
vartika21571@iiitd.ac.in

April 8, 2023

1 City of Computopia

1.1 Problem

The police department in the city of Computopia has made all the streets one-way. But the mayor of the city still claims that it is possible to legally drive from one intersection to any other intersection.

(**a**) Formulate this problem as a graph theoretic problem and explain why it can be solved in linear time.

(**b**) Suppose it was found that the mayor's claim was wrong. She has now made a weaker claim: "If you start driving from 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.

1.2 Solution Part (a)

1 Formulation as a graph problem

Consider the initial city as a graph $G(V, E)$ with intersections as nodes and streets as edges between them as shown in Fig 1.

More formally, any intersection $\mathbf{int}_i \in V(G)$, and $(\mathbf{int}_i, \mathbf{int}_j) \in E(G)$ if there is a road connecting intersections $\mathbf{int}_i$ and $\mathbf{int}_j$

Now, the police turning all the streets to one-way means converting all the undirected edges into directed ones. as shown in Fig 2

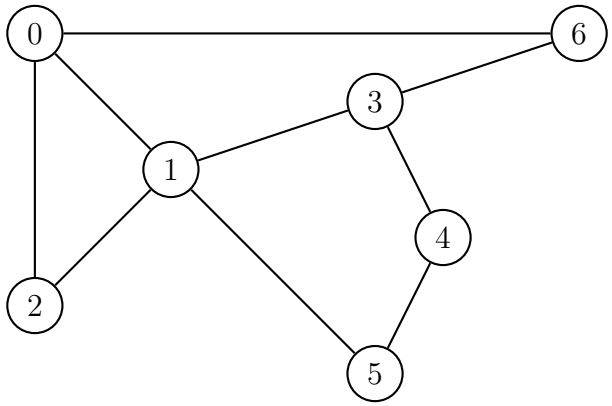

Figure 1: two-way streets

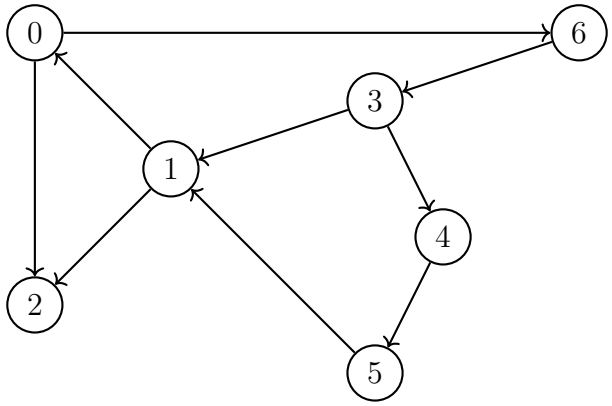

Figure 2: One-way streets instead of 2-way

2 Linear time solution

Algorithm 1: All intersections reachable

```
1  procedure connectedGraph(G) → boolean
       /* Returns true if mayor's claim is true, i.e. graph is
       strongly connected. Otherwise, returns false                    */
2      visited ← boolean array of size |V(G)|, and initialize all to false
3      pick any arbitrary node int_i ∈ V(G)
4      perform DFS with int_i as the starting node
5      set visited[j] = vis_j = true if int_j is visited in the DFS ∀ int_j ∈ V(G)
6      for vis_i in visited do
7          if vis_i = false then
8              return false // In case any node is not visited
9          end
10     end
11     return true
12 end
```

The main idea behind the algorithm is to check if the graph is strongly connected (implying that all intersections are reachable). It takes $\mathcal{O}(|V| + |E|)$ operations to perform DFS on a

graph, and $\mathcal{O}(|V|)$ further operations to check if all nodes have been visited. Therefore, the overall time complexity of the algorithm is $\mathcal{O}(|V| + |E|)$, i.e. linear.

1.3 Solution Part (b)

1 Formulation as a graph problem

The formulation is the same as part (a) when converting all the streets into one-way roads.

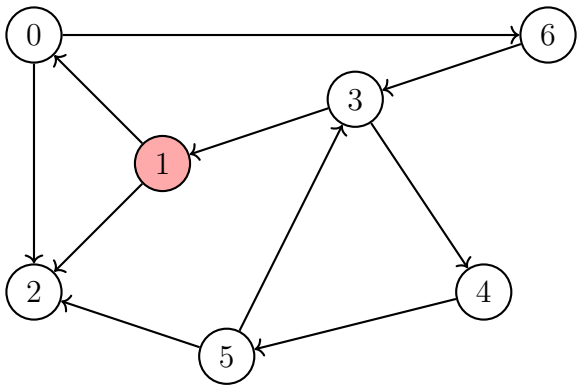

Figure 3: Cycle in a directed graph, Node 1 is the starting node or town hall

2 Linear time solution

Algorithm 2: All intersections reachable

```
1  procedure connectedGraph(G) → boolean
       /* Returns true if there is a path from town hall to itself, false otherwise    */
2      Convert G into a condensation graph using Kosaraju's algorithm.
3      Let the new graph be C(V, E)
4      Let U ∈ V(C) be of the form U = {u_1, u_2, ..., u_k} where u_i ∈ V(G)
5      Let T ∈ V(C) be the condensation node which contains the town-hall
6      for U ∈ V(C) do
7          for u ∈ U do
8              set label(u) = U
9          end
10     end
11     for (u_1, u_2) ∈ E(G) do
12         if label(u_1) = T ∧ label(u_2) ≠ T then
13             return false
14         end
15     end
16     return true
17 end
```

Kosaraju's algorithm takes $\mathcal{O}(|V| + |E|)$ to form the condensation graph, and then further $\mathcal{O}(|E|)$ operations are performed to check all the edges. Therefore, the overall time complexity of the algorithm is $\mathcal{O}(|V| + |E|)$, i.e. linear.

2 Smallest weight in a cycle

2.1 Problem

Given an edge-weighted connected undirected graph $G = (V, E)$ with $n + 20$ edges. Design an algorithm that runs in $\mathcal{O}(n)$-time and outputs an edge with the smallest weight contained in a cycle of G. You must give a justification for why your algorithm works correctly

2.2 Solution

1 Linear Time Algorithm

Algorithm 3: Smallest weight contained in a cycle of G

```
   // global variables
 1 min_w ← ∞
 2 visited ← boolean array of size |V(G)|, and initialize all to false
 3 curr_min ← ∞
 4
 5 procedure parentDFS(v)
 6 |   set vis_v = true
 7 |
 8 |   for p in Adj(v) do
 9 |   |   if vis_p = false then
10 |   |   |   parent(w) = v
11 |   |   |   curr_min = min(curr_min, (p, v)) // (u,v) is the weight of the
12 |   |   |   Run parentDFS(p)
13 |   |   end
14 |   |
15 |   |   if vis_p = true ∧ parent(p) ≠ v then
16 |   |   |   min_w = min(min_w, curr_min)
17 |   |   |   curr_min ← ∞
18 |   |   end
19 |   end
20 end
21
22 procedure smallestCycleEdge(G) → int
   |   /* Returns smallest weight contained in a cycle of G.
   |   Returns ∞ if there is no cycle in G                          */
23 |   consider any arbitrary vertex u as the starting node
24 |   Run parentDFS(u)
25 |   return min_w
26 end
```

The main idea behind Algorithm 3 is to keep track of current minimum edge, and then update the global minimum edge when it detects a loop.

2 Correctness Proof

Assumption: A cycle contains 3 or more vertices, i.e. no self-loops and edge being considered as loop.

Any undirected connected graph $G = (V, E)$ can be cyclic or acyclic.

Case 1: Acyclic - In this case, it will never be possible for a vertex $v \in V(G)$ to be both visited and have its parent not be the current node, i.e. $u \in V(G)$.

1. If v is not visited, then its parent will be set to u and marked visited.
2. If v is visited, its parent must be u. Assume by contradiction that v is visited, and its parent is $w \in V(G)$. This implies a cycle since we know a path from u to w (the graph is known to be connected). We have just encountered the edge $(u, v) \because v \in \text{Adj}(u)$, and $\text{parent}(v) = w$, so there is also an edge (v, w), thus completing the cycle.

Thus, the value of min_w remains ∞ implying there is no cycle in the graph and therefore no minimum edge

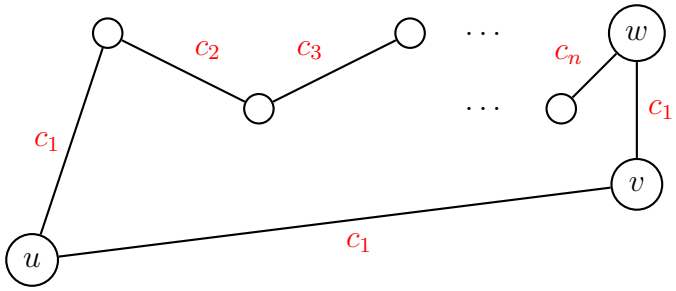

Figure 4: Cycle condition

Case 2: Cyclic If there is a cycle, there will always be a case where v is visited and $\text{parent}(v) \neq u$. There will always be a node u with a path from it to itself, as shown in Fig 3. We always maintain the minimum edge along the cycle and update it with min_w when we reach the cycle's starting point, giving us the smallest edge weight in all cycles of G.

Also, the fact that we only need to call `parentDFS()` once stems from the fact that the graph is connected, and the DFS Tree of any one node will contain all the graph nodes.

3 Time Complexity

In `parentDFS(`u`)`, we visit a node at max 2 times, once when it is marked visited and once when we go to check if it is visited and the parent is not u. Therefore, time complexity of the algorithm is linear, i.e. $\mathcal{O}(|V| + |E|)$

3 Probability of reaching sink

3.1 Problem

Suppose that G be a directed acyclic graph with the following features

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

$$\sum_{(v \to w) \in E} Pr(v \to w) = 1$$

The weights $Pr(v \to w)$ define a random walk in G from the source s to some sink t_i ; after reaching any non-sink vertex v, the walk follows the edge $v \to w$ with probability $Pr(v \to w)$.

All the probabilities are mutually independent. Describe and analyze an algorithm to compute the probability that this random walk reaches sink t_i for every $i \in 1, \ldots, k$. You can assume that an arithmetic operation takes $\mathcal{O}(1)$ time.

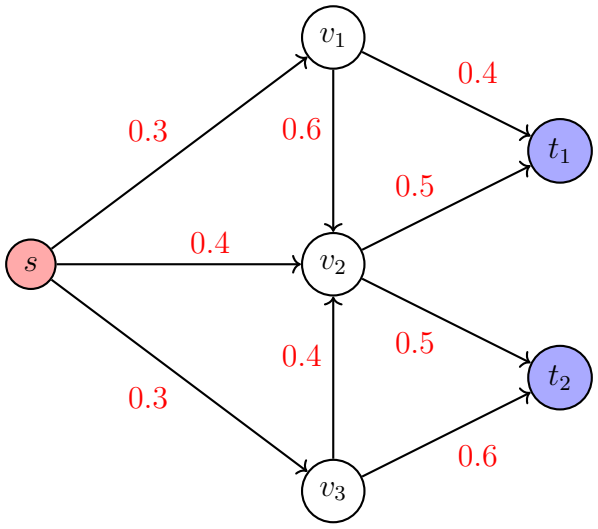

Figure 5: An Example Network Flow Graph. Probabilities are mentioned on every edge. The source vertex is coloured red, and sinks are coloured blue

3.2 Solution

1 Sub-problem

Probability of reaching the sink t_i from vertex v, denoted as $P(v, t_i)$.

2 Recurrence Relation

Base Case: $P(t_i, t_i) = 1$, and $P(v, t_i) = 0 \ \forall v \neq t_i$

Recurrence Relation :-

$$P(v, t_i) = \sum_{(v \to w) \in E} Pr(v \to w) \times P(w, t_i)$$

In other words, the probability of reaching from v to t_i is the sum of the probability of going from v to w times the probability of reaching from w to t_i

3 Subproblem which gives solution

$P(s, t_i)$, which gives the probability of reaching sink t_i from s

4 Algorithm

Algorithm 4: Probability of reaching sinks

```
1  procedure computeProb(G, u, v, P)
2      if u = v = t_i then
3          return
4      end
5      curr_val ← 0 // for storing current value of P(u, v)
6      for w ∈ Adj(u) do
7          if P(w, v) = −1 then
8              Run computeProb(G, w, v, P)
9          end
10         curr_val+= Pr(u → w) × P(w, v)
11     end
12     P(u, v) = curr_val
13 end
14
15 procedure pSinks(G) → array
       /* Returns an array pathProb where pathProb[i] is probability of reaching sink t_i
          from source s.  Assuming the graph has n sinks t_1, t_2, ..., t_n                 */
16     P ← 2-D array to store value of P(u, v) ∀u, v ∈ V(G)
17     pathProb ← array to store value of P(s, t_i)
18     Initialize all values in pathProb to 0 and in P to -1
19
20     for i = 1 to n do
21         P(t_i, t_i) = 1
22     end
23     for i = 1 to n do
24         Run computeProb(G, s, t_i, P)
25         pathProbs[i] = P(s, t_i)
26     end
27     return pathProbs
28 end
```

5 Time Complexity

For every sink t_i, We will traverse through the entire graph in a DFS-type manner and then backtrack to fill the values of the edges already visited, as can be seen in the recursive nature of the algorithm. Therefore, the time complexity for finding $P(s, t_i)$ is $\mathcal{O}(|V| + |E|)$

If there are n such sinks, then the algorithm will be repeated n times for each sink. Therefore, total time complexity of the algorithm is $\mathcal{O}(n(|V| + |E|))$