Graphs(connectedness) **May 29, 2025**

Homework 8

Spring 2025 *COSC 31: Algorithms*

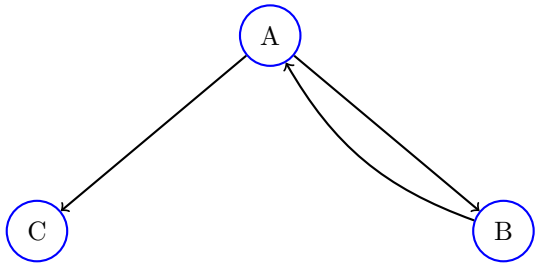
1. **Kosaraju's algorithm**
   Let $G = (V, E)$ be a directed graph with $V = \{1, 2, \ldots, n\}$, and k be the number of SCCs of G. One way to represent the SCCs of G is with an array S of length n such that, for all $i, j \in \{1, 2, \ldots, n\}$, $S[i] \in \{1, 2, \ldots, k\}$ and $S[i] = S[j]$ if and only if vertices i and j are in the same strongly connected component of G.

   Write an algorithm of $O(n+m)$ time and space complexity that, given the adjacency list representation of a directed graph $G = (V, E)$ with $V = \{1, 2, \ldots, n\}$, returns (k, S) where k is the number of SCCs of G and S is an array representing the SCCs of G.

2. **Simpler SCC** Dr. Murkha claims that the algorithm for strongly connected components would be simpler if it used the original (instead of the transpose) graph in the second depth-first search and scanned the vertices in order of increasing finishing times. Does this simpler algorithm always produce correct answers? Justify your answer.

   Answer

   No, the simpler algorithm will not always produce correct answers. Consider the graph below:

   The edges are $A \to B$, $B \to A$, and $A \to C$. So $\{A, B\}$ is one strongly connected component (SCC), and $\{C\}$ is another.

   Suppose we run the first DFS starting at A. It visits B and C from A. The finish times will be: $B.f = 3$, $C.f = 5$, and $A.f = 6$. Now, the simpler algorithm proposes to run a second DFS on the original graph, using the order of increasing finishing times: B, C, A. Starting from B, the DFS explores A then C, and returns the SCC $\{A, B, C\}$. This incorrectly groups all three vertices into one SCC: $\{A, B, C\}$ because there is no path from C to A or B.

   The correct algorithm uses the transpose graph and scans vertices in decreasing finish time, which correctly identifies SCCs: $\{A, B\}$ and $\{C\}$.

3. **Component Graph**
   Write an algorithm of $\Theta(n + m)$ time and space complexity that, given

   - the adjacency list A of a directed graph $G = (V, E)$ with $V = \{1, 2, \ldots, n\}$, and
   - the array S that represents the SCCs of G,

Algorithm 1 Kosaraju's Algorithm

```
Function Kosaraju(G = (V, E)):
    foreach u ∈ V do
        u.visited ← false
    A ← empty stack
    foreach u ∈ V do
        if u.visited = false then
            DFSOne(u, A)
    G^T ← Transpose(G)

    foreach u ∈ V in G^T do
        u.visited ← false
    S ← empty array
    k ← 0

    while A ≠⊥ do
        u ← pop from stack A
        if u.visited = false then
            k ← k + 1
            DFS2(u, k, S)
    return (k, S)
Function DFSOne(u, A):
    u.visited ← true
    foreach v ∈ Adj[u] do
        if v.visited = false then
            DFSOne(v, A)
    push u onto stack A
Function Transpose(G = (V, E)):
    G^T ← new graph with vertices V
    foreach u ∈ V do
        foreach v ∈ Adj[u] do
            add edge (v → u) to G^T
    return G^T
Function DFS2(u, k, S):
    u.visited ← true
    S[u] ← k
    append u to newSCC
    foreach v ∈ Adj[u] do
        if v.visited = false then
            DFS2(v, S)
```

Time Complexity: The algorithm runs in $O(n+m)$ time, where n is the number of vertices and m is the number of edges. The first DFS traversal takes $O(n+m)$ time, the transposition of the graph also takes $O(n+m)$ time, and the second DFS traversal also takes $O(n+m)$ time. Thus, the overall time complexity is linear in terms of the size of the graph.

Space Complexity: The space complexity is $O(n+m)$ due to the storage of the adjacency list for the transposed graph and the stack used in the DFS traversals.

returns the adjacency list of the graph GSCC , which I define below exactly as in class:$G^{SCC} = (V^{SCC}, E^{SCC})$, where $V^{SCC} = \{C | C \text{ is an SCC of } G\}$ and $E^{SCC} = \{(C, C') | C \neq C', \text{and } \exists u \in C, u' \in C', (u, u') \in E\}$

Answer

Algorithm 2 Component Graph Construction

Function `ComponentGraph`(A, S)**:**
- Let $n \leftarrow$ length of A
- Let $k \leftarrow 1 + \max(S)$
- Initialize $A^{SCC}[1 \dots k]$ as empty sets
- **for** $u \leftarrow 1$ **to** n **do**
  - **foreach** $v \in A[u]$ **do**
    - **if** $S[u] \neq S[v]$ **then**
      - Add $S[v]$ to $A^{SCC}[S[u]]$
- **for** $i \leftarrow 1$ **to** k **do**
  - Convert $A^{SCC}[i]$ from set to list
- **return** A^{SCC}

Time Complexity: The algorithm iterates through each vertex u and its adjacency list $A[u]$. Since each edge is processed exactly once, the total time complexity is $\Theta(n + m)$, where n is the number of vertices and m is the number of edges. The space complexity is also $\Theta(n + m)$ due to the torage of the adjacency list for the component graph.
Space Complexity: The space complexity is $\Theta(n + m)$, as we store the adjacency list for the component graph, which contains at most n vertices and m edges.

4. Semiconnectivity
A directed graph $G = (V, E)$ is semiconnected if, for all pairs of vertices $u, v \in V$, there is a path from u to v or there is a path from v to u(or both). Give an efficient algorithm that, given a directed graph $G = (V, E)$, outputs whether or not it is semiconnected. Argue the correctness of your algorithm and analyze its time complexity to be $\Theta(V + E)$

Answer

Algorithm 3 Semiconnectivity Check

Function `Semiconnected`$(G(V, E))$**:**
- $(k, S) \leftarrow$ FKosaraju$G$ // **SCC ID array; from q1**
- $G^{SCC} \leftarrow$ FComponentGraphG, S // **from q3**
- Do topological sort on G^{SCC}
- **for** $i \leftarrow 1$ **to** $k - 1$ **do**
  - $C_i \leftarrow$ the i^{th} SCC in the topological order
  - $C_{i+1} \leftarrow$ the $(i + 1)^{th}$ SCC in the topological order
  - **if** $C_{i+1} \notin A^{SCC}[C_i]$ **then**
    - **return false**
- **return true**

Correctness: The algorithm first identifies SCCs because semiconnectivity is preserved within SCCs. To check for semiconnectivity of the SCCs, we construct the component graph G^{SCC} and perform a topological sort. We then verify that each SCC can reach the next one in the topological order. If any SCC cannot reach the next, the graph is not semiconnected.

Time Complexity: The algorithm runs in $\Theta(V + E)$ time. The Kosaraju's algorithm for finding SCCs runs in $\Theta(V + E)$, and constructing the component graph also takes $\Theta(V + E)$. The topological sort of the component graph takes $\Theta(V + E)$ as well. Thus, the overall time complexity is $\Theta(V + E)$.

Space Complexity: The space complexity is also $\Theta(V+E)$ due to the storage of the adjacency list for the component graph and the SCC ID array.

5. **All Pairs Shortest Paths**

Let Gbe a digraph, with edge weights given by$w : E(G) \to \mathbb{R}$. Assume that $V(G) = \{1, 2, \ldots, n\}$. In class we study the Bellman-Ford algorithm that computes shortest paths to all vertices from a given source vertex $s \in V(G)$. Suppose instead that we wish to compute the shortest paths from every vertex to every vertex, i.e., we want to compute $\delta(i, j)$ for all $i, j \in V(G)$, where $\delta(i, j)$ is the weight of a shortest path from i to j. One solution would be to run Bellman-Ford n times, once for each $s \in V(G)$, but this solution would take V $\times$O(nm) = O(n2m) time. Since m can be as high as n2, the solution can be as expensive as O(n4). Can we do better? The answer is yes: we can design a O(n3)-time algorithm. How? Well, read on.

If $i_1, i_2, \ldots, i_k$ is a path p in a graph G, we call $i_2, i_3, \ldots, i_k - 1$, which are all but the start and end vertices on path p, the intermediate vertices on p. For example, if p is $3, 6, 2, 1, 4$, the intermediate vertices are $6, 2, 1$. For another example, if p is $3, 6, 2, 4, 1, 4$, then the intermediate vertices are $6, 2, 4, 1$ (notice that even though 4 is the end vertex, it is counted as an intermediate vertex since it also occurs as a non-start and a non-end vertex).

(a) Write a recurrence for $d^k_{i,j}$, including base cases(s).

Answer

$$d^k_{i,j} = \begin{cases} 0 & \text{if } k = 0 \text{ and } i = j \\ w(i, j) & \text{if } k = 0 \text{ and } i \neq j \\ \infty & \text{if } P^k_{i,j} = \emptyset \\ -\infty & \text{if path in } P^k_{i,j} \text{ has a negative weight cycle} \\ \min(d^{k-1}_{i,j}, d^{k-1}_{i,k} + d^{k-1}_{k,j}) & \text{otherwise} \end{cases}$$

(b) Use this recurrence to write an algorithm that, given an adjacency matrix Afor a weighted digraph G, returns the following two quantities, for all $i, j \in V(G)$:

- $\delta(i, j)$
- $\pi(i, j)$, where $\pi(i, j)$ is the last vertex before j on a shortest path from i to j if $i \neq j$ and $\delta(i, j) \notin \{-\infty, \infty\}$; and $\pi(i, j) = nil$ otherwise.

Answer

Algorithm 4 Floyd-Warshall using $O(n^3)$ space

```
Function AllPairsShortestPaths(A):
    // Input:  A[i][j] is the weight of edge (i, j) or ∞ if no edge
    // Step 1:  Initialize d[0...n, 1...n, 1...n] and π[1...n, 1...n]
1   for i ← 1 to n do
2       for j ← 1 to n do
3           if i = j then
4               d[0, i, j] ← 0,   π[i, j] ← nil
5           else if A[i, j] ≠ ∞ then
6               d[0, i, j] ← A[i, j],   π[i, j] ← i
7           else
8               d[0, i, j] ← ∞,   π[i, j] ← nil
    // Step 2:  Apply recurrence for k = 1 to n
9   for k ← 1 to n do
10      for i ← 1 to n do
11          for j ← 1 to n do
12              if d[k − 1, i, k] + d[k − 1, k, j] < d[k − 1, i, j] then
13                  d[k, i, j] ← d[k − 1, i, k] + d[k − 1, k, j]
                    π[i, j] ← π[k, j]
14              else
15                  d[k, i, j] ← d[k − 1, i, j]
16  return d[n], π
```

(c) State how to modify the above algorithm so that it still runs in $O(n^3)$ time, but takes only $O(n^2)$ space. Your algorithm should of course remain correct, run in $O(n^3)$ time and take $O(n^2)$ space,

Answer

To reduce the space complexity to $O(n^2)$ while preserving the $O(n^3)$ time complexity, we modify the algorithm as follows:

Algorithm 5 Floyd-Warshall with $O(n^2)$ space

```
Function AllPairsShortestPathsSpaceOptimized(A):
    Initialize d[1...n, 1...n] and π[1...n, 1...n] for i ← 1 to n do
        for j ← 1 to n do
            if i = j then
                d[i, j] ← 0,   π[i, j] ← nil
            else if A[i][j] ≠ ∞ then
                d[i, j] ← A[i, j],   π[i, j] ← i
            else
                d[i, j] ← ∞,   π[i, j] ← nil
    for k ← 1 to n do
        for i ← 1 to n do
            for j ← 1 to n do
                if d[i, k] + d[k, j] < d[i, j] then
                    d[i, j] ← d[i, k] + d[k, j]
                    π[i][j] ← π[k, j]
    return d, π
```

(d) Write the invariant for only the outermost loop in your algorithm for Part (c). You don't need to prove your invariant correct.

Answer

- $1 \leq k \leq n + 1$
- $\forall i, j \in V, i \neq jd[i, j]$ contains the minimum weight of a path from i to j whose intermediate vertices are in the set $\{1, 2, \ldots, k - 1\}$

(e) Write a method that, given $i, j \in V(G)$, uses the information computed by your algorithm to output a shortest path from i to j if $\delta(i, j) \notin \{-\infty, \infty\}$.

Answer

To achieve, we can the $\pi(i, j)$ returned from part c rescontruct the path from i to j.

Algorithm 6 GetShortestPath(i, j, π)

```
Function GetShortestPath(i, j, π):
    if π[i][j] = nil then
        return "No path from i to j"
    path ← empty list
    current ← j
    while current ≠ i do
        insert current at the beginning of path
        current ← π[i][current]
        if current = nil then
            return "No path from i to j"
    insert i at the beginning of path
    return path
```

(f) Write a method that outputs whether G has a negative weight simple cycle and, if it does, it outputs a simple cycle in G. Your method is not required to do anything more than what I asked for (for instance, it can stop and return as soon as it is able to answer the question asked).

Answer

Algorithm 7 DetectNegativeCycle(d, π)

```
Function DetectNegativeCycle(d, π):
    for v ← 1 to n do
        if d[v][v] < 0 then
            visited ← empty set
            cycle ← empty list
            current ← v
            while current ∉ visited do
                add current to visited
                current ← π[v][current]
                if current = nil then
                    return "No path traceable to form a cycle"
            start ← current
            cycle ← list with start
            current ← π[v][start]
            while current ≠ start do
                insert current at the beginning of cycle
                current ← π[v][current]
            insert start at the beginning of cycle
            return "Negative-weight cycle found: ", cycle
    return "No negative-weight cycle found."
```

Explanation: The algorithm checks each vertex v for a negative weight cycle by examining the diagonal of the distance matrix $d[v, v]$ and uses the predecessor matrix π to trace back the cycle. If a negative cycle is found, it constructs the cycle by following the predecessors until it loops back to the starting vertex.