cs 5800 - hw10

Yijing Xiao

November 2021

1 Exercise 22.1-5

For the adjacency-matrix representation of a graph $G = (V, E)$, we assume that the vertices are numbered $1, 2, ..., |V|$ in some arbitrary manner. Then the adjacency-matrix representation of a graph G consists of a $|V| \times |V|$ matrix $A = (a_{ij})$ such that

$$f(n) = \begin{cases} 1, & \textbf{if } (i,j) \in E \\ 0, & \textbf{otherwise } n \end{cases}$$

We have algorithm: .

---

for $i = 0$ to V **do**
    for $j = 0$ to V **do**
        $G^2[i][j] \leftarrow 0$
        for $k = 0$ to V **do**
            if $G[i][k] == 1$ and $G[k][j] == 1$ **then**
                $G^2[i][j] = 1$
            end if
        end for
    end for
end for
return G^2

The G^2 may be computed in V^3 times, so the running time is $O(V^3)$.

2 Exercise 22.2-6

Let $G = (V, E)$ be the first graph and $G' = (V, E_\pi)$ be the second graph, and let 1 be the source vertex.

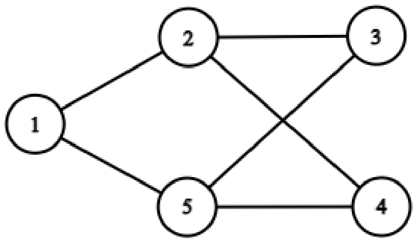

Figure 1: first graph

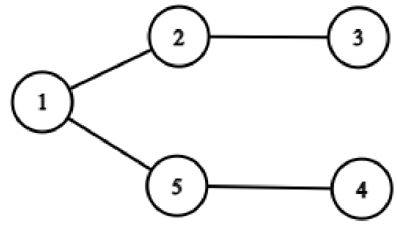

Figure 2: second graph

The breadth-first search systematically explores the edges of G to "discover" every vertex that is reachable from 1. Suppose that 2 proceeds 5 in the adjacency list of 1.We could see that E_π will never be produced by running breadth-first search on G.

- If 5 precedes 2 in the $Adj[1]$. We will dequeue 5 before 2, so $3.\pi$ and $4.\pi$ are both 5, which is not true.
- If 2 preceded 5 in the $Adj[1]$. We will dequeue 2 before 5, so $3.\pi$ and $4.\pi$ are both 2, which again is not true.

Therefore, the set of edges E_π cannot be produced by running breadth-first search on G no matter how the vertices are ordered in each adjacency list.

3 Exercise 22.2-7

Color the vertices of graph of rivalries by two colors, "babyface" and "heel". We use BFS of each connected component to get the distance value for each vertex.

1) Initialize graph $G(V, E)$ source vertex's color is "babyface".
2) Assign all wrestlers whose distance is even to be "babyface" color, and all wrestlers whose distance is odd to be "heel" color.
3) Verify each vertex's color. If vertex doesn't have any color, then its adjacency vertex will be colored by a different color. Otherwise, vertex has color and its color same with its adjacency vertex color, then it means that it is not possible to find a designation.

We have algorithm to verify each vertex's color:

.

```
for each vertex u ∈ G.V − {s} do
    if u.color == UNCOLORED then
        u.color ← "heel"
        if adjacency vertex of u color is "heel" then
            adjacency vertex of u color ← "babyface"
        else
            adjacency vertex of u color ← "heel"
        end if
        for neighbor of adjacency vertex of u do
            if neighbor.color == UNCOLORED then
                neighbor.color = adjacency vertex of u color
            else
                if neighbor.color ≠ adjacency vertex of u color then
                    return FALSE
                end if
            end if
        end for
    end if
end for
return TRUE
```

Time analysis: this algorithm will take $O(n + r)$ time for the BFS where n is the number of vertex and r is the number of edge in graph G.

4 Exercise 22.3-7

Algorithm 1 Rewrite DFS using a stack to eliminate recursion

```
function DFS(G)
    for each vertex u ∈ G.V do
        u.color = WHITE
        u.π = NIL
    end for
    time ← 0
    for each vertex u ∈ G.V do
        if u.color == WHITE then
            DFS-VISIT(G, u)
        end if
    end for
end function
```

```
function DFS-VISIT(G, u)
    for each vertex u ∈ G.V do
        u.color = WHITE
        u.π = NIL
    end for
    stack.PUSH(u)
    while stack is not empty do
        x = stack.TOP
        flag = TRUE
        for each v ∈ G : Adj[x] do                          ▷ explore edge (x, v)
            if v.color == WHITE then
                v.π = x
                flag = FALSE
                time = time + 1
                v.d = time
                v.color = GRAY
                stack.PUSH(v)
            end if
        end for
        if flag == TRUE then        ▷ if all neighboring vertex of the element at the top
            y = stack.POP              ▷ of stack are not white (all marked as GRAY or
            y.color = BLACK                               ▷ BLACK), then the stack pop
            time = time + 1
            y.f = time
        end if
    end while
end function
```

5 Exercise 22.3-10

We can define four edge types in the depth-first forest G_π produced by a depth-first search on G : Tree edge, Back edge, Forward edge, and Cross edge. The DFS algorithm has enough information to classify some edges as it encounters them. The key idea is that when we first explore an edge (u, v), the color of vertex v tells us something about the edge:

- WHITE indicates a tree edge,
- GRAY indicates a back edge, and
- Black indicates a forward or cross edge

The first case is immediate form the specification of the algorithm. For the second case, observe that the GRAY vertices always from a linear chain of descendants corresponding to the stack of active. DFS-VISIT invocations. Exploration

always proceeds from the deepest GRAY vertex, so an edge that reaches another GRAY vertex has reached an ancestor. The third case handles the remaining possibility: edge (u, v) is a forward edge if $u.d < v.d$ and a cross edge if $u.d > v.d$.

Algorithm 2 Prints out every edge in the directed graph G with its type

```
function DFS(G)
    for each vertex u ∈ G.V do
        u.color = WHITE
        u.π = NIL
    end for
    time ← 0
    for each vertex u ∈ G.V do
        if u.color == WHITE then
            DFS-VISIT-PRINT(G, u)
        end if
    end for
end function

function DFS-VISIT-PRINT(G, u)
    time = time + 1                          ▷ white vertex u has just been discovered
    u.d = time
    u.color = GRAY
    for each v ∈ G.Adj[u] do                 ▷ explore edge (u, v)
        if v.color == WHITE then
            print "(u, v) is a tree edge"
            v.π = x
            DFS-VISIT-PRINT(G, v)
        else
            if v.color == GRAY then
                print "(u, v) is a back edge"
            else
                if v.d > u.d then
                    print "(u, v) is a forward edge"
                else
                    print "(u, v) is a cross edge"
                end if
            end if
        end if
    end for
end function
```

An undirected graph may entail some ambiguity in how we classify edges, since (u, v) and (v, u) are really the same edge. In such a case, we classify the edge as the first type in the classification list that applies. Equivalently, we classsify the edge according to whichever of (u, v) or (v, u) the search encounters first. For

DFS-VISIT-PRINT algorithm, if G is undirected we don't need to make any modifications.

6 Exercise 22.3-12

On the basic of DFS, we could add a numerical identifier cc of connected components to count how many connected components. The input graph G may be undirected or directed. The variable time is a global that used for time stamping and DFS-VISIT procedures to assign values to the cc attributes of vertices.

Algorithm 3 Identify the connected components of G

```
function DFS(G)
    for each vertex u ∈ G.V do
        u.color = WHITE
        u.π = NIL
    end for
    time ← 0
    k ← 1
    for each vertex u ∈ G.V do
        if u.color == WHITE then
            u.cc = k
            k = k + 1
            DFS-VISIT(G, u)
        end if
    end for
end function

function DFS-VISIT(G, u)
    time = time + 1                         ▷ white vertex u has just been discovered
    u.d = time
    u.color = GRAY
    for each v ∈ G.Adj[u] do                ▷ explore edge (u, v)
        v.cc = u.cc
        if v.color == WHITE then
            v.π = x
            DFS-VISIT(G, v)
        end if
    end for
    u.color = BLACK                         ▷ blacken u; it is finished
    time = time + 1
    u.f = time
end function
```

7 Exercise 22.4-5

Algorithm 4 Topological sorting to find a vertex of in-degree 0

```
function TOPOLOGICAL-SORT(G)
    for each vertex u ∈ G.V do                          ▷ initialize in-degree
        u.in − degree = 0
    end for
    for each vertex u ∈ G.V do                          ▷ compute in-degree
        for each vertex v ∈ G.Adj[u] do
            v.in − degree = v.in − degree + 1
        end for
    end for
    Q ← ∅                                               ▷ initialize queue
    for each vertex u ∈ G.V do
        if u.in − degree == 0 then
            ENQUEUE(Q, u)
        end if
    end for
    while Q is not ∅ do
        u = DEQUEUE(Q)
        Output u
        for each vertex v ∈ G.Adj[u] do
            v.in − degree = v.in − degree − 1
            if v.in − degree == 0 then
                ENQUEUE(Q, v)
            end if
        end for
    end while
    for each vertex u ∈ G.V do                          ▷ check for cycles
        if u.in − degree ≠ 0 then
            report there is a cycle
        end if
    end for
end function
```

To find and output vertices of in-degree 0, we compute all vertices in-degree by making pass through all the edges and incrementing the in-degree of each vertex an edge enters. If there are no cycles, all vertices are output. Otherwise, not all vertices will be output because some in-degree never become 0.

8 All paths from s to t must have a common vertex

Thinking of any breadth-first spanning tree T of graph G with s as the root. In T, each node v has a level, which is the number of edges in a shortest path from s to v. Since every path from s to t has at least $l = 1 + \frac{|V|}{2}$, node t occurs at a level $\geq l$. Thinking of level 1 through $\frac{|v|}{2}$, the total number of nodes in level 1 through $\frac{|v|}{2}$ is at most $|v| - 2$ (Since node s and t doesn't in these level). If each of these level has 2 or more nodes, then total number of nodes in G will exceed $|V|$. Therefore, there must be a level in the range 1 through $\frac{|v|}{2}$ containing just one node v which is a common vertex of s to t. We have algorithm:

- Construct a BFS of G, run it starting from node s. For each level i, construct the list $L[i]$ of the node
- Find a level j where $1 \leq j \leq \frac{|V|}{2}$ such that $L[j]$ has only one node v
- Output node v

For the running time, the step 1 takes $O(V+E)$ time since we construct a BFS of G. Step 2 takes $O(V)$ time and step 3 takes $O(1)$ time. Therefore, the algorithm runs in $O(V+E)$ time.

9 Exercise 23.1-3

Algorithm 5 Growing a minimum spanning tree

```
function CENTRIC-MST(G, w)
    A ← ∅
    while A does not from a spanning tree do
        find an edge(u, v) that is safe for A
        A = A ∪ {(u, v)}
    end while
    return A
end function
```

In this CENTRIC-MST, prior to each iteration, A is a subset of some minimum spanning tree. At each step, we determine an edge (u, v) that we can add to A without violating this invariant, and we can add it safely to A while maintaining the invariant. Assume we've already get a set of edges A, let's get rid of an edge (u, v), then draw a cut. In this case, our strategy is to choose a light edge because the cut respects A and (u, v) is a light edge for this cut.

10 Exercise 23.2-2

Algorithm 6 Prim's algorithm runs in $O(v^2)$ time

```
function MST-PRIM(G, w, r)
    for each vertex u ∈ G.V do
        u.key = ∞
        u.π = NIL
    end for
    r.key = 0
    Q = G.V
    while Q is not ∅ do
        u = EXTRACT-MIN(Q)
        for each vertex v ∈ G.Adj[u] do
            if MATRIX[u][v] == 1 and v ∈ Q and w(u, v) < v.key then
                v.π = u
                v.key = w(u, v)
            end if
        end for
    end while
end function
```

11 Exercise 23.2-4

The running time of Kruskal's algorithm for a graph $G = (u, v)$ depends on how we implement the disjoint-set data structure. We assume that we use the disjoint-set-forest implementation of Book Section 21.3 with the union-by-rank and path-compression heuristic, since it is the asymptotically fastest implementation known. The total running time is $O((V + E)\alpha(V))$ where α is very slowly growing function defined in Book Section 21.4. Assume graph G is connected, we have $|E| \leq |V| - 1$, and so the disjoint-set operations take $O(E\alpha(V))$ time. Moreover, since $\alpha(|V|) = O(lgV) = O(lgE)$, the total running time of Kruskal's algorithm is $O(ElgE)$.

If alledge weights are integers in the range from 1 to $|V|$, then we could sort the edges in $O(V + E)$ time using counting sort. Since we assume the graph G is connected, $V = O(E)$ and so the sorting time is reduced to $O(E)$. The total running time would be $O(V + E + E\alpha(V)) = O(W\alpha(V))$ since $V = O(E)$ and $E = O(E\alpha(V))$.

If the edge weights are integers in the range from 1 to w for some constant w, then we still apply counting sort to sort edges. The sorting time would take $O(E + w) = O(E)$ time since w is a constant. The total running time is $O(E\alpha(V))$.

12 Exercise 23.2-5

The running time of Prim's algorithm depends on how we implement the min-priority queue Q. We can improve the asymptotic running time of Prim's algorithm by using Fibonacci heaps. Book Chapter 19 shows that if a Fibonacci heap holds $|V|$ elements, an EXTRACT-MIN operation takes $O(lgV)$ amortized time and a DECREASE-KEY operation takes $O(1)$ amortized time. Therefore, if we use a Fibonacci heap to implement the min-priority queue Q, the running time of Prim's algorithm improves to $O(E + VlgV)$.

If the edge weights are integers in the range from 1 to w for some constant w, we can implement the queue as an array $Q[0...w + 1]$. EXTRACT-MIN scan the array and find the first non-empty slot in $O(1)$ time, and DECREASE-MIN also takes $O(1)$ time. The total running time is $O(E)$.