Graph Algorithmics Sheet 2023

I. Graph Traversal: BFS

Queue q
Dictionary prev, dist
Set visited
q.enqueue(s)
visited.add(s)
dist[s] = 0
while not q.isEmpty() do
  x = q.dequeue()
  for y in Nout(x) do
    if y not in visited then
      q.enqueue(y)
      visited.add(y)
      dist[y] = dist[y] + 1
      prev[y] = x
    endif
  endfor
endwhile
accesible = visited

O(m)

Pr-Correctness: ① all vertices returned are ACCESSIBLE
② the algo finishes
③ all accesible vert. are ret.

Graph Traversal: DFS

Stack stack
Dict prev
Set visited
stack.push(s)
visited.add(s)
prev[s] = null
while not stack.isEmpty() do:
  x = stack.pop()
  for y in Nout(x) do:
    if y not in visited then
      stack.push(y)
      visited.add(y)
      prev[y] = x
    endif
  endfor
endwhile
accesible = visited

O(n+m)
O(n)

II. Strongly Connected Comps

Kosaraju Algorithm:
- set of vertices
- DFS for each vertex
- doesn't visit the prev found

Reflexive-Tranzitive Closure
- = the accesibility relation in that graph; Notation: x ν y
  x acc. from y, y acc from x =>
- defines the partitioning of the set
- cycle => single SCC

Tarjan's Algorithm SCC
- performs DFS, computes for each vertex the earliest ancestor vertex (closest to root) directly reachable from that vertex or a descendant in the DFS tree (LOWLINK)
- SCC => SCC representative = their own lowlink

III. Minimum Cost Walk (Dyn Prog)

Input:
G-graph
s-source vertex
t-target vertex

O(m×m)

Output:
minCost: min cost s→t
path: path that achieves the minimum cost

- define a table minCost[1..n, 1..m]: store the min cost from s to each vertex i
- define a table prev[1..n, 1..m]: store the prev vertex in the optimal path to i
- initialize minCost[s] = 0
- initialize prev[s] = null

for (each) vertex i in G\{s} do
  minCost[i] = infinity
for vertex u in G, starting with s do
  for neighbour v of u in G do:
    cost = minCost[u] + weight(u,v)
    if cost < minCost[v] then:
      minCost[v] = cost
      prev[v] = u

if minCost[t] is infinity then
  return "No path exists"
//construct the path s→t using table
path = []
current = t
while current is not null do
  prepend current to path
  current = prev[current]
return minCost[t], path

Proof of Correctness: ① at each stage, dist and prev correspond to existing walks (relaxation)
② the algorithm finishes
③ when the algorithm finishes, dist[x] = d(s,x) for all vert.

IV. Minimum Cost Walk: BELLMAN-FORD

Input:
G: directed graph with costs
s,t: start, target vertices
Output:
dist: map from each acc. vertex to s (cost-wise)
prev[]
Algorithm:
for x in X do:
  dist[x] = ∞
dist[s] = 0
changed = true
while changed do:
  changed = false
  for (x,y) in Edges do:
    if dist[y] > dist[x] + cost(x,y) then:
      dist[y] = dist[x] + cost(x,y)
      prev[y] = x
      changed = true
    endif
  endfor
endwhile

V. Minimum Cost Path: DIJKSTRA

Proof Of Correctness (Costs ≥ 0)
We claim that, when a vertex is dequeued from the priority queue, its dist = the cost of min cost walk from the start to it.

DIJKSTRA Algorithm

Input:
- G: directed graph with costs
- s,t: start, target

Output:
- dist[]
- prev[]

Algorithm:
Priority Queue q
Dictionary prev
Dictionary dist
q.enqueue(s,0)
dist[s] = 0
found = false
while not q.isEmpty() and not found do:
  x = q.dequeue()
  (element with min value of priority)
  for y in Nout(x) do
    if y not in dist.keys() or dist[x] + cost(x,y) < dist[y] then:
      dist[y] = dist[x] + cost(x,y)
      q.enqueue(y, dist[y])
      prev[y] = x
    if x == t then
      found = true
endwhile

Kinda like BFS, but with a Priority Queue

VI. Minimum Cost Walk: FLOYD-WARSHALL

- based on dynamic programming
- the recursion starts like for the matrix multiplication algorithm

Algorithm:
for i = 0 to n-1 do
  for j = 0 to n-1 do
    if i == j then
      w[i,j] = 0
    else if (i,j) is edge in G then
      w[i,j] = cost(i,j)
      f[i,j] = j
    else
      w[i,j] = infinity

O(n³)

for k = 0 to n-1 do
  for i = 0 to n-1 do
    for j = 0 to n-1 do
      if w[i,j] > w[i,k] + w[k,j] then:
        w[i,j] = w[i,k] + w[k,j]
        f[i,j] = f[i,k]
      endif
    endfor
  endfor
endfor

VII. Matrix Multiplication

O(n³ log n) / O(n³)

VIII. Minimum Spanning Tree: PRIM's Algorithm

Input:
- G: directed graph with costs

Output:
- edges: collection of edges

Algorithm:
PriorityQueue q O((n+m) log n)
Dictionary prev
Dictionary dist
edges = ∅
choose s in X arbitrarily
vertices = {s}
for x in N(s) do
  dist[x] = cost(x,s)
  prev[x] = s priority
  q.enqueue(x, d[x])
while not q.isEmpty() do
  x = q.dequeue()
  if x ∉ vertices then
    edges.add({x, prev[x]})
    vertices.add(x)
    for y in N(x) do
      if y not in dist.keys() or cost(x,y) < dist[y]:
        dist[y] = cost(x,y)
        q.enqueue(y, dist[y])
        prev[y] = x
    endif
  endwhile

IX. Minimum Spanning Tree: KRUSKAL

Input: O(m²) / O(m log m)
G-graph
Output:
T-min spanning tree
Algorithm:
- sort the edges of G in nondecreasing order of their weights
- initialize an empty set T to store the edges of MST
- initialize a disjoint-set data structure to keep track of the connected comps

for vertex v in G do
  MakeSet(v) // create a set for each vertex
for edge (u,v) in sorted set of edges do
  if Find(u) != Find(v):
    Add (u,v) to T
    Union (u,v)
  endif
endfor
return T

↳ if u and v are in dif connected components, add the edge u,v to the MST and then merge the connected components of u & v

X. Flows

Prerequisites:
- a directed graph G
- s,t - source, destination
- each edge (x,y) has a positive capacity cap(x,y)

A flow can be established through the graph. A flow is an assignment of a flow value to each edge, s.t.:
a) for each edge, 0 ≤ flow(x,y) ≤ cap(x,y)
b) for each edge sans s and t, the inbound flow = outbound flow

Cut = partitioning the vertices into two sets: one containing the source and one containing the destination
Net flow across the cut = total "left to right" flow

Capacity of the flow = only left to right capacity

X.1. Naive Algorithm

- a flow of zero → valid flow; we can increase the flow while keeping it valid by using the approach:
- find a path s→t of only non saturated edges
- compute the capacity of the path as the smallest residual capacities (res. cap. = cap(edge) - flow)
- increase the flow on all edges on that path with that value

X.2. FORD-FULKERSON

Theorem: the value of the max flow = the capacity of the min cut

Algorithm:
① start with zero flow
② for the current flow, construct a residual graph ⊂ same vertices as G, but
- for each non-saturated edge of the original graph, put an edge with the same dir. and remaining cap.
- for each edge with ≠0 flow, put an edge in the opp. dir with a cap = flow
↳ flow can be ↑ in the reverse dir of the forward flow.
③ find a path s→t in residual graph
④ compute capacity of path from ③
⑤ update the flow:
- for → edges, ↑ flow to the cap of path
- for ← edges, ↓ flow on the corresp. forward edges by the same value
⑥ repeat steps 2-5 until no path can be found in the residual graph from s→t

maximum length path → topo + dyn pr.
nr of distinct paths → by Dijkstra, BFS
cycle of minim. length → Floyd-Warshall
shortest cycle of strictly negative → Bellman-Ford

1. DAG & Topological Sorting

① function isDAG(graph):
```
visitedNodes = emptySet()
inProgressNodes = emptySet
for vertex in Graph:
    if vertex not in visitedNodes:
        if !DFS(Graph, vertex, visitedNodes, inProgN)
            return false
            // a cycle was found
return true
function DFS(g, v, visited, inProg):
    for neighbour in g.neighbour(v):
        if !DFS(g, neighbour, visited, inProgressNodes)
            return false
        else if neighbour in inProgressNodes:
            return false
    inProgressNodes.remove(node)
    return true
```

DAG, DFS based ↑

```
function isDAG(graph):
    inDegrees = createEmptyMap()
    zeroInDegreeNodes = createEmptyList()
    for node in Graph:
        inDegrees[node] = 0
    for each edge(u,v) in Graph:
        inDegrees[v] += 1
    for each node in Graph:
        if inDegrees[node] == 0
            zeroInDegreeNodes.add(Node)
    while zeroInDegrees is not empty:
        node = zeroInDegrees.removeFirst()
        for each neighbour in node.neighbours:
            inDegree[neighbour] -= 1
            if inDegree[neighbour] == 0
                zeroInDegreeNodes.add(neighbour)
    for each node in Graph:
        if inDegree[node] != 0
            return false
    return true
```

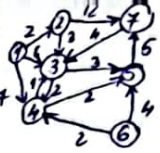
DAG, predecessor counting

O(n+m)

⑪ Topological Sorting

```
function topologicalSort(graph):
    sortedOrder = []
    visited = set()
    for node in Graph:
        if node not in visited:
            dfs(node, visited, sortedOrder)
    return sortedOrder
function DFS(node, visited, sortedOrder):
    visited.add(node)
    for neighbour in node.neighbours:
        if neighbour not in visited
            dfs()
    sortedOrder.insert(0, node)
    prepend
```

```
function topologicalSort(graph):
    sortedOrder = []
    inDegree = Dictionary
    prereq = Dictionary
    for activity, (priority, prereqs) in Graph:
        inDegree[activity] = 0
        prereqs[activity] = prereqs
        for prereq in prereqs:
            inDegree[prereq] += 1
    priorityQueue = dequeue()
    // enqueue act. with 0 prereqs
    for activity, degree in inDegree:
        if degree == 0
            priorityQueue.append(activity)
    while priorityQueue:
        activity = priorityQueue.pop()
        sortedOrder.append(activity)
        // decrement inDegree of dep act.
        for dependentAct in prereqs[act]:
            inDegree[dependentAct] -= 1
            if inDegree[dependentAct] == 0:
                priorityQueue.append(depAct)
    endfor
endwhile
if len(sortedOrder) != len(graph):
    return [] // cycles
return SortedOrder
```



		d							node						
		1	2	3	4	5	6	7	1	2	3	4	5	6	7
	7	∞	∞	∞	∞	∞	∞	0	-	-	-	-	-	-	-
7	2,5	∞	12	∞	∞	5	∞	0	-	7	-	-	7	-	-
5	2,3,4,6	∞	12	8	7	5	9	0	-	7	5	5	7	5	-
4	1,2,3,6	14	12	8	7	5	9	0	4	7	5	5	7	5	-
3	1,2,6	14	11	8	7	5	9	0	4	3	5	5	7	5	-
6	1,2	14	11	8	7	5	9	0	4	3	5	5	7	5	-
2	1	13	11	8	7	5	9	0	2	3	5	5	4	5	-
1		13	11	8	7	5	9	0	2	3	5	5	7	5	

1: 1→2→3→5→7 (C:13)
2: 2→3→5→7 (C:11)
3: 3→5→7 (C:8)
4: 4→5→7 (C:7)
5: 5→7 (C:5)
6: 4→5→7 (C:9)
7: 7 (C:0)