Graph- algorithms

Sunday, 22 October 2023 7:41 AM

① Topological Ordering of a Directed Acyclic Graph

— is a linear ordering of vertices such that for every directed edge $u \to v$, vertex u comes before v.

— Topological ordering is not possible if the graph is not a DAG.

— In a DAG, we can have multiple orders and, moreover, in each order, vertices may be placed in many ways.

Theory $O(V)$ → visit each v & other. Then, add v at start of list.

alt: Topological ordering using DFS

→ DFS we pick a vertex then recursively call DFS for adjacent vertices. In topological ordering we want to put vertex before adjacent vertex.

DFS → $O(V+E)$

TC → $O(V+E)$

Modification to DFS for Topological:

1) Use a temp stack.
2) Check visited vertex immediately.
3) Recursively call top ordering for all adj vertices, then push it to stack.
4) Finally print contents of stack.

Algo: As vertex is pushed to stack, only when all its adjacent vertices (and their adjacent vertices) have been pushed, we can, safely, be the stack.

$O(V+E)$ time.

$O(V)$ extra space for stack.

```
def topsort(graph, visited):
    stack = []
    visited = [False] * len(graph)
    for vertex in range(len(graph)):
        if vertex not in visited:
            dfs_helper(graph, vertex, visited, stack)
    return stack[::-1]

def dfs_helper(graph, vertex, visited, stack):
    visited[vertex] = True
    for neighbor in graph[vertex]:
        if not visited[neighbor]:
            dfs_helper(graph, neighbor, visited, stack)
    stack.append(vertex)
```

alt: Kahn's Algorithm for Topological Ordering (BFS -based)

Step 1) Compute indegree for each vertex, and initialize queue of zero-indegree nodes.

1.) Pick all vertices with indegree as 0 and add them to queue.

2.) Remove vertex from queue and then:

a) Increment count of visited vertex by 1

b) Decrease indegree by 1 for all neighboring nodes

c) If indegree of neighboring vertex is reduced to zero, add it.

3) Repeat step 2 until queue is empty.

4) If count of visited vertex is not equal to no. of vertex then topological sort is not possible (because cycle).

```
def topsort(graph, vertices):
    indegree = {v: 0 for v in vertices}
    for u in graph:
        for v in graph[u]:
            indegree[v] += 1
    queue = [v for v in vertices if indegree[v] == 0]
    count = 0
    while queue:
        vertex = queue.pop(0)
        count += 1
        for neighbor in graph[vertex]:
            indegree[neighbor] -= 1
            if indegree[neighbor] == 0:
                queue.append(neighbor)
    if count != len(vertices):
        raise ValueError("Graph has a cycle")
    return count
```

Applications of Kahn's Algo

— Cycle Detection

— Maintaining of dependency

— Scheduling tasks

— Data processing

— Circuit $O(V+E)$

② Minimum Spanning Tree

— defined as spanning tree that has minimum weight among all possible spanning tree.

Properties

— no of vertices (V) in graph and spanning tree is same.

— from no of edges in spanning tree, which is equal to one less than total no of vertices ($V-1$).

— Spanning tree should not be disconnected, as it makes it only one single cycle of component and hence tree.

— Spanning tree, should be acyclic, which means there should not be any cycle.

— total cost for weight is defined as sum of edges.

— can be many possible spanning trees.

Algorithms (Both algo designed for undirected graphs only)

Kruskal: Minimum Spanning Tree

— greedy approach (optimal, designs clustering and data analysis)

Prim's: Min Spanning Tree

— greedy (image segmentation, clustering)

Kruskal

— Each edge of graph given in increasing order. Then, it keeps on adding new edges and nodes in MST if the newly added edge does not form a cycle.

— It picks min weighted edge at first and then, weighted edge, etc. and.

— Time: not one step it makes a locally optimal choice in hope of finding an global solution.

Time → sorting edge takes $O(E\log(E))$ time.

→ then, looping, we iterate over all edges and apply find-union algo.

→ takes at most $O(E\alpha(V))$ time.

→ overall time complexity is $O(E\log(E) + E\alpha(V))$

→ since, α is can at most be $O(V)$ → $O(E\log(E)$ and $O(E\log(V)$ are same.

Therefore, overall complexity is $O(E\log(V)$ or $O(E\log(E))$

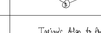
Space → $O(V+E)$

Prim

— adjacency matrix / adjacency list.

Note: why Edges fed to directed?

→ it assumes that all vertices are connected. But in directed graph, every node is not reachable from every other node. So, Prim's fails.



→ no root available from

Why Kruskal fails for directed?

→ it finds strongly connected components in directed graph. Kruskal fails in directed graphs as there are cases where there is no path between vertices in directed graph. Kruskal algo assumes it is acyclic and doesn't have considerations for cycle edges.

③ Kruskal MST Spanning Tree

1) Sort all edges in increasing order of their weight.

2) Initialize: MST = {} , cost = 0

3) In following for every edge E, where MST exists and forms, u,v

(i) adding e to MST does not create a cycle

→ $MST = MST \cup \{e\}$

$cost = cost + e.weight$

4) Return cost

Cluster Edges

MST

$A \leftrightarrow B$ 4

$A \leftrightarrow C$ 2

$B \leftrightarrow C$ 1

$C \leftrightarrow D$ 3

$B \leftrightarrow D$ 10

$D \leftrightarrow E$ 5

$cost = 2+1+3+5 = 11$

Implementation

Complex → create a separate graph for MST

Graph representation for Kruskal's Algo

Class Map:

def __init__(self, vertices):

self.V = vertices

self.graph = []

self.parent = {}

Using Adjacent list: Edge Structure

def add_edge(self, u, v, w):

self.graph.append([u, v, w])

def find(self, i):

if self.parent[i] == i:

return i

return self.find(self.parent[i])

def union(self, x, y):

root_x = self.find(x)

root_y = self.find(y)

self.parent[root_x] = root_y

def kruskal(self):

result = []

self.graph.sort(key=lambda edge: edge[2])

for u, v, w in self.graph:

if self.find(u) != self.find(v):

result.append([u, v, w])

self.union(u, v)

return result

Time

$O(E\log(E) + E\alpha(V))$

$\approx O(E\log(V) + E\alpha(V)) = O(E\log(V))$

Space

$O(E\log(E) + V + E\alpha(V)) = O(E\log(E))$

④ Fast Algorithm for MST

— going of edges that connect two sets of vertices in a graph is called cut in graph theory.

— at every step of Prim's algo, find a cut and pick minimum weight edge from that cut, and include vertex in MST set.

→ Idea is to maintain a two set of vertices that contains vertices already in MST, and other contains vertices not yet included.

At every step, we consider all edges that connect the two sets and pick the minimum edge from these edges.

After picking the edge, we move the endpoint of edge to set containing MST.

Algo:

1. Create a set mstSet that keeps track of vertices already included in MST.
2. Assign a key value to all vertices in input graph, initially all key values as infinite. Assign key value for first vertex as 0 so that it is picked.
3. While mstSet doesn't include all vertices:
   - Pick a vertex u that is not there in mstSet and has min value.
   - Include u in mstSet.
   - Then update key values of adjacent vertices of u.
     - To update distance, compare, if weight through u is smaller than current value.
     - For every vertex v, if the sum of distance vertex u + (edge distance) and weight of u,v is less than distance of v, then update distance value of v.

Note: this uses a priority queue (heap) to extract the min of vertex included in MST.

Adding dist[v] is used to store the shortest distance known so far to vertices.

Time: $O(E^2)$

Space: $O(V)$

→ equivalent to adjacency matrix.

Optimized Approach Time: $O(E\log(V))$ Space: $O(V^2)$

Intuition: 1. Traverse adjacency matrix, with adjacency lists.

2. Then create a priority queue to store vertex and the weight.

3. Get the left most least weight edge.

4. Use queue to globally pick and push the right vertex and its weight.

5. Then, use traversal through the adjacency and check the least weight to connect.

6. At last add edges all vertices, one after one.

⑤ Dijkstra's Algorithm (find shortest distance from source to all other vertices)

Keypoints Algorithm: $O(V^2)$

Idea is to maintain a shortest path tree (SPT) with a given source as root.

Maintain two sets, one set contains:

→ one set contains vertices included in the shortest path tree.

→ other set includes vertices not yet included in shortest path tree.

At every step of algo, find a vertex that is in other set and has a minimum distance from the source.

Algo:

1. Create a set sptSet that keeps track of vertices included in SPT.
2. Assign a distance value to all vertices in input graph. Initially, all distance values as infinite. Assign 0 for the source vertex so that it is picked first.
3. While sptSet doesn't include all vertices:
   - Pick a vertex u that is not there in sptSet & sptSet and has min distance.
   - Include u in sptSet.
   - Then update distance values of adjacent vertices of u.
     - To update distance, compare, if weight through u is smaller than current value.
     - For every vertex v, if the sum of distance vertex u + (edge distance) and weight of u,v is less than distance of v, then update distance value of v.

Note: if we use a binary heap (optimal) to represent the set of vertices included in SPT, it reduces the time complexity from $O(V^2)$ to $O(E\log(V))$.

Adjacency Matrix Representation

— Idea is to traverse all vertices of graph using BFS and use a queue to check the vertex with min distance.

Time complexity of algorithm this method with adjacency matrix is $O(V^2)$.

Adjacency List Representation

— Idea is to traverse all vertices of graph using BFS and use a priority to check the vertex with min distance.

Time complexity of algorithm this method with adjacency list is $O(E\log V)$. $O(E\log V)$ for priority queue.

Adjacent Matrix

— Idea is to traverse all vertices of graph using BFS and use a priority to check the vertex with min distance.

Time complexity of algorithm this method with adjacency matrix is $O(V^2)$. $O(V^2)$ for priority queue.

Adjacency List

1. Create a set sptSet that keeps track of vertices included in SPT.
2. Assign a distance value to all vertices in input graph. Initially, all distance values as infinite. Assign 0 for the source vertex so that it is picked first.
3. While sptSet doesn't include all vertices:
   - Pick a vertex u that is not there in sptSet & sptSet and has min distance.
   - Include u in sptSet.
   - Then update distance values of adjacent vertices of u.
     - To update distance, compare, if weight through u is smaller than current value.
     - For every vertex v, if the sum of distance vertex u + (edge distance) and weight of u,v is less than distance of v, then update distance value of v.

⑥ Bellman-Ford Algorithm

— Same, from Dijkstra's algo but unlike in Dijkstra's graph with negative weight edges. Bellman-Ford also can be found if there is a negative cycle in graph if we can reach to its source from reachable vertex. Bellman-Ford can be used for finding shortest path in graph with negative edge weight.

— Bellman-Ford's idea is to check edges in a single cycle and update distance if edge is less. Because, initially, we unknown and assumed to be infinite, just edge then, the vertex adjacent relaxed. Hence further key identifying a five iterations.

Hence, it's said that Bellman-Ford is a "Dynamic programming".

Principle of Relaxation in Bellman-Ford

— For graph having V vertices and E edges, edges should be relaxed V-1 times to complete the single source shortest path.

→ In order to detect if any cycle, relax all edges one more time, and if shortest path of any edge reduces then we say we have cycle.

Why relaxing all edges (V-1) times (Dynamic Programming Concept Bellman)?

→ In worst case, a shortest path between two vertices can have almost V-1 edges.

→ Iteration, relaxing edges V-1 times guarantees that edge has reached and public shortest path. Looping up to V-1, ensures we have positive, length of shortest path.

In a graph with V vertices.

Algo:

1. Initialize distance array dist[] for each vertex 'v' as dist[v] = infinity.
2. Assume any vertex (say 0) as source and assign dist[0]=0.
3. Relax all edges (u,v,weight) V-1 times as per below condition:
   - dist[v] = min (dist[v], dist[u]+weight)
4. Find value for each vertex, then:
   - Case 1: (this cycle exists) for any edge (u,v,weight)
     - if dist[u] + weight < dist[v]
   - Case 2: (no any cycle) case, 1 false for all edges

Note: if cycle exists all distances are undefined, false case.

→ In worst, disconnected graphs, we can repeat relax step for entire vertex distance initially, so simply initialize with infinite.

Time → connected: $O(V*E)$

disconnected: $O(V*E)$

Average: $O(V*E)$

Worst: $O(V*E)$

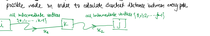
Space → $O(V)$

⑦ Floyd-Warshall Algorithm

— find shortest path between all pairs of nodes in a weighted graph.

— unlike Dijkstra and Bellman-Ford, graphs with both positive and negative edge weights (work for directed/undirected but not with negative cycles)

— uses dynamic programming approach to check, every possible path going from every possible vertex to other to calculate shortest distance between every pair.



Algo:

1. Initialize, initialize matrix, same as input graph matrix.
2. Then update, update matrix by considering all vertices as an intermediate vertex.
3. Idea is to pick all vertices and see, and update shortest path, which contains the first vertex as an intermediate index in the shortest path.
4. While we pick vertex k as intermediate, vertex, we have considered vertices {0, 1, ..., k-1} already.
5. For every pair (i,j), there can possibilities:
   - i, j is not intermediate, vertex, in shortest path from i to j.
   - Keep value dist[i][j] as it is.
   - i, j is our intermediate vertex in shortest path from i to j.
   - Update, dist[i][j] = dist[i][k] + dist[k][j] if dist[i][j] > sum.

Time → $O(V^3)$

Space → $O(V^2)$

How to detect the -ve cycle Floyd-Warshall?

→ Check for each distance from itself and if it comes out to be negative, we will detect negative cycle.

⑧ Johnson's Algorithm

Algorithm

1. Add a new vertex q and add edges from q to all vertices of graph. Assign weight of all edges added as 0.
2. Bellman-Ford algorithm is used to find shortest distance from q to all other vertices. If we find a negative weight cycle, then we report that there is a negative weight cycle in graph.
3. Reweight edges: For each edge (u, v) of original graph, assign new weight to edge as: weight(u, v) + distance(q, u) - distance(q, v).
4. Remove vertex q and run Dijkstra's algorithm for each vertex to find shortest path from vertex to all other vertices.

⑨ Strongly Connected Components (SCC)

— In a directed graph that has a path from every vertex to every other vertex, is said strongly connected. (what only in directed graphs)

eg: 2-strongly connected components {1,2,3,4} and {5,6,7} since there is a path from each vertex to every other vertex.

— Interestingly, in undirected graph relation is similar but vertices are reachable from each other.

— Alternately, strongly-connected relation is equivalent of strongly connected graph.

→ if and only if, for every pair of vertices A and B, there exists a path from A to B and a path B to A.

— Two different connected components mean, a single component that edge is added between a vertex from one component to other.

But this is not case in strongly connected component.

Two strongly connected doesn't become a single strong component if there only one unidirectional edge from one SCC to other SCC.

Kosaraju Algo: Time $O(V+E)$ Space $O(V)$

— for each vertex, 'i' (which is not part of strongly connected component) find vertices which can reach 'i'. Find all SCC containing 'i'.

— Then vertices 'i' and 'j' will be in same SCC if there is directed path from vertex 'i' to 'j' and vice-versa.

Kosaraju Tarjan BFS

Note: It is easy for undirected graph.

— we can do a BFS/DFS starting from a vertex, if BFS/DFS visits all vertices then undirected graph is connected.

— This method works for directed graphs.

eg: 0 → 1 → 2 → 3

if we start BFS/DFS from 0, we can reach all vertices but if we start BFS/DFS from other vertices, we can not reach all vertices.

→ Create reversed 'R' for graph & call BFS/DFS for every vertex of reversed Transpose graph of 'R' graph. Time complexity is $O(V+E)$.

— we can also do BFS of these starting from every vertex. Time: BFS takes $O(V+E)$ and we perform BFS from graph's V and SCC.

Time: $O(V*(V+E))$

— Kosaraju which finds SCC. → if SCC is 1 then graph is strongly connected.

Algo: (use BFS traversal)

1. Initialize, all vertices as not visited.
2. Do a BFS traversal of graph starting from arbitrary vertex. If BFS doesn't visit all vertices, return false.
3. Reverse all edges and find traversal, to create graph.
4. Do a BFS of reversed graph starting from same vertex. If BFS traversal doesn't visit all vertices, then return false otherwise, true.

Time: same as BFS. $O(V+E)$ for graph & reversed graph adjacency list representation.

Instead of BFS, DFS can be, better, and easy since could be recursive.

Tarjan's Algo to find SCC

Based on fact:

— BFS reach properties of BFS tree/forest

— strongly connected components form subtrees of BFS tree.

— if we find head of such subtrees we can store all nodes in that subtree (including head) and that will be SCC.

— Time: $O(V+E)$ which finds SCC in one single DFS traversal.

(Tarjan can be used for cycle detection, finding articulation points, bridges, etc.)

⑩ Prim's Algorithm

Idea: Maintain two disjoint sets of vertices. One set contains vertices included in MST, other contains vertices not yet included.

Every step, we move one vertex from second set to first set.

Algo:

1. Create a set mstSet that keeps track of vertices already included in MST.
2. Assign a key value to all vertices in input graph. Initially, all key values as infinite. Assign key value for first vertex as 0 so that it is picked first.
3. While mstSet doesn't include all vertices:
   - Pick a vertex u that is not there in mstSet & has min value.
   - Include u in mstSet.
   - Then update key values of adjacent vertices of u.
     - To update distance, compare, if weight through u is smaller than current value.
     - For every vertex v, if the sum of distance vertex u + (edge distance) and weight of u,v is less than distance of v, then update distance value of v.

Time: $O(V^2)$

Space: $O(V)$

Adjacency List

1. Create a set mstSet that keeps track of vertices already included in MST.
2. Assign a key value to all vertices in input graph. Initially, all key values as infinite. Assign 0 for the first vertex so that it is picked first.
3. While mstSet doesn't include all vertices:
   - Pick a vertex u that is not there in mstSet & has min value.
   - Include u in mstSet.
   - Then update key values of adjacent vertices of u.
     - To update distance, compare, if weight through u is smaller than current value.
     - For every vertex v, if the sum of distance vertex u + (edge distance) and weight of u,v is less than distance of v, then update distance value of v.

Time: $O(E\log V)$

Space: $O(V+E)$