Elementary Graph Algorithms: Summary

Definition: A graph is a collection (nonempty set) of vertices and edges

A path from vertex x to vertex y: a list of vertices in which successive vertices are connected by edges

Connected graph: There is a path between each two vertices

Simple path: No vertex is repeated

Cycle: Simple path except that the first vertex is equal to the last

Loop: An edge that connects the vertex with itself

Tree: A graph with no cycles

Spanning tree of a graph: a subgraph that contains all the vertices, and no cycles

Complete graphs: Graphs with all edges present – each vertex is connected to all other vertices.

Weighted graphs – weights are assigned to each edge (e.g. road map with distances)

Directed graphs: The edges are oriented, they have a beginning and an end.

A tree with N vertices has N-1 edges.
A graph with less than N-1 edges is not connected.

Algorithms

1. Topological Sort

Types of graphs:
   - The graphs should be directed: otherwise for any edge (u,v) there would be a path from u to v and also from v to u, and hence they cannot be ordered.
   - The graphs should be acyclic: otherwise for any two vertices u and v on a cycle u would precede v and v would precede u.

Algorithm:
1. Compute the indegrees of all vertices
2. Find a vertex U with indegree 0 and print it (store it in the ordering)
   If there is no such vertex then there is a cycle and the vertices cannot be ordered.
   Stop.
3. Remove U and all its edges (U,V) from the graph.
4. Update the indegrees of the remaining vertices.
5. Repeat steps 2 through 4 while there are vertices to be processed.

Complexity O(E + V)
   We examine all edges (O(E)), and we store in the queue each vertex only once (O(V)).
2. Shortest path for unweighted graphs

Data structures needed:
  Adjacency lists to represent the graph
  A table with rows for each vertex and two columns:

  1 - Distance from source vertex
  2 - Path - the name of the vertex where we have come from to reach that vertex

  A queue to store the adjacent nodes to be processed

Algorithm:

1. Store s in a queue with distance = 0
2. While there are vertices in the queue:

   1. Read a vertex \( v \) from the queue
   2. For all adjacent vertices:
      If distance = * (not computed)
      Distance = (distance to \( v \)) + 1
      Path = \( v \)
      Append to the queue

Complexity \( O(E + V) \)
  We examine all edges (\( O(E) \)), and we store in the queue each vertex only once (\( O(V) \)).
3. Shortest Path in weighted graphs - Dijkstra's Algorithm

Differences:

a. The adjacency lists contain in addition the weights of the edges
b. Instead of ordinary queue, a priority queue is used (the distances being the priorities) and the vertex with the least distance is selected for processing
c. In the table, we add the length of the new edge to the currently stored distance.
d. Distances are subjected to adjustments - if the newly computed distance is smaller.

Algorithm:

1. Store s in a queue with distance = 0
2. While there are vertices in the queue:
   1. DeleteMin a vertex v from the queue
   2. For all adjacent vertices w:
      - Compute new distance = (distance to v) + (d(v,w))
      - If distance = * (not computed)
        - store new distance in table
        - append v in queue
        - path = v
      - If old distance > new distance
        - Update old distance = new distance
          (this is done by updating the priority of an element in the queue - decreaseKey operation. Complexity O(logV))
        - Update path = v

Complexity O(ElogV + VlogV) = O((E + V)log(V))

Each vertex is stored only once in the queue - max elements = V
The deleteMin operation is O( VlogV )
The decreaseKey operation is logV ( a search in the binary heap). It might be performed for each examined edge - O(ElogV).
4. **Algorithm to find a spanning tree** (unweighted graph)

**Data structures** needed:
- A table (an array) $T$ with size = number of vertices, where $T_i = \text{parent of vertex } v_i$
- Adjacency lists
- A queue of vertices to be processed

1. Choose a vertex $u$ and store it in the queue. Set a counter $= 0$, and $Tu = r$ (u would be the root of the tree)
2. While the queue is not empty and counter $< |V| - 1$ do the following:
   - Read a vertex $v_j$ from the queue.
   - For each $u_k$ in the adjacency list of $v_j$ do the following
     - If $Tk$ is empty,
       - $Tk = v_j$.
       - counter $= \text{counter} + 1$
       - store $u_k$ in the queue

Complexity: $O(|E| + |V|)$ - we process all edges and all nodes
5. **Minimum Spanning Tree - Prim's algorithm (weighted graph)**

The algorithm is similar to finding the shortest paths in a weighted graphs. The difference is that we record in the table the length of the current edge, not the length of the path.

**Data structures** needed:
- A table with number of rows = number of vertices, and three columns:
  - \( T_{i,1} = T \) if the adjacency list of the vertex has been processed, \( F \) otherwise
  - This is necessary because the graph is not directed and without this information we may enter a cycle.
  - \( T_{i,2} = \) the length of the edge from the parent to the vertex \( v_i \),
  - \( T_{i,3} = \) parent of vertex \( v_i \)
- Adjacency lists
- A priority queue of vertices to be processed.
  - The priority of each vertex is determined by the weight of edge that links the vertex to its parent. The priority may change if we change the parent.

It does not matter which vertex is chosen, because all vertices have to be in the tree.

**Algorithm:**

1. Initialize first column to \( F \), select a vertex \( s \) and store it in the priority queue with priority = 0, set \( T_{s,1} = 0, T_{s,2} = \) root

2. While there are vertices in the queue:
   3. **DeleteMin** a vertex \( v \) from the queue and set \( T_{v,1} = T \)
   4. For all adjacent vertices \( w \):
      - If \( T_{w,1} = T \) do nothing
      - If \( T_{w,2} \) is empty:
        - \( T_{w,2} = \) weight of edge \((v,w)\)  // stored in the adjacency list
        - \( T_{w,3} = v \)  // this is the parent
        - Append \( w \) in the queue with priority = weight of \((v,w)\)
      - If \( T_{w,2} > \) weight of \((v,w)\)
        - Update \( T_{w,2} = \) weight of edge \((v,w)\)
        - Update the priority of \( w \)
          (this is done by updating the priority of an element in the queue - **decreaseKey** operation. Complexity \( O(\log V) \))
        - Update \( T_{w,3} = v \)

At the end of the algorithm, the tree would be represented in the table with its edges \( \{(T_{i,3}, v_i) \mid i = 1, 2, \ldots |V|\} \)

**Complexity** \( O(|E|\log(|V|)) \)
6. Minimum Spanning Tree – Kruskal's algorithm

Kruskal's algorithm works with tree forests and the set of edges. Each vertex belongs to only one tree in the forest.

Initially we build |V| trees consisting of one vertex only - each vertex is a tree of its own. The basic operation of the algorithm is to choose an edge (u,v) from the set of edges with minimal weight (and remove the edge from the set). This is implemented by storing the edges in a priority queue, with priority = the weight of the edge.

1. If u and v belong to one and the same tree then we do nothing.
2. If u and v belong to different trees, we link the trees by the edge

Complexity of Kruskal's algorithm O(Elog(E)).

For sparse trees Kruskal's algorithm is better - since it is guided by the edges. For dense trees Prim's algorithm is better - the process is limited by the number of the processed vertices (first column in the table to be F in order to process, otherwise we skip the vertex)