Graph Algorithms : Summary

1. Basic concepts

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.

2. Representation

a. **adjacency matrix**
b. **adjacency lists**

3. Topological sort

Topological sort: an ordering of the vertices in a directed acyclic graph, such that:

If there is a path from ***u*** to ***v***, then ***v*** appears after ***u*** in the ordering.

Types of graphs: **directed**, **acyclic**

Degree of a node U: the number of edges (U,V) - outgoing edges

Indegree of a node U: the number of edges (V,U) - incoming edges

Algorithm

1. Initialize sorted list to be empty, and a counter to 0
2. Compute the indegrees of all nodes
3. Store all nodes with indegree 0 in a queue

4. While the queue is not empty
   a. **get** a node U and put it in the sorted list. Increment the counter.
   b. For all edges (U,V) decrement the indegree of V, and **put** V in the queue if the updated indegree is 0.
5. If counter is not equal to the number of nodes, there is a cycle.

Complexity

The number of operations is O(|E| + |V|), |V| - number of vertices, |E| - number of edges.
How many operations are needed to compute the indegrees?
Depends on the representation:

Adjacency lists: O(|E|)
Matrix: $O(|V|^2)$

4. Single source shortest path algorithm for directed unweighted graphs

Algorithm for computing the distance for a vertex s to all other vertices:

1. Initialize

   D_table $_{s,0}$ = 0 (distance from s to itself = 0)
   D_table $_{s,1}$ = 0 (s is the starting vertex)
   D_table $_{i,j}$ = -1 i ≠s

2. Store **s** in a queue
3. While there are vertices in the queue:
   1. Read a vertex **v** from the queue
   2. For all adjacent vertices **w**:

      If D_table $_{w,0}$ = -1 (distance not computed)

      D_table $_{w,0}$ ← D_table $_{v,0}$ + 1 (i.e. Distance $_{w}$ is (distance to **v**) + 1)
      D_table $_{w,1}$ ← **v** (i.e. Path$_{w}$ = **v**)
      Append **w** to the queue

Complexity:

Matrix representation: $O(V^2)$
Adjacency lists: O(E + V)

5. Single source shortest path algorithm for directed weighted graphs: Dijkstra's algorithm

Similar to the single source shortest path algorithm for unweighted graphs. 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:

s – starting node
DT – Distance Table,
PQ – priority queue, the priority of a node is equal to the distance from **s** to that node

Initialize DT(s,0) = 0, DT(s,1) = 0, all remaining DT(j,k) = -1

1. Store **s** in PQ 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) + (distance(v,w))

      i.e. **new_distance = DT(v,0) + distance(v,w)**

      If distance to **w** not computed (**DT(w,0) = -1**)

      store new distance in table : **DT(w,0)= new_distance**
      append **w** in PQ with priority **new_distance**
      make path to **w** equal to **v**, i.e. DT(w,1) = v

else
if old distance > new distance, i.e. **DT(w,0) > new_distance**

Update old_distance = new_distance, i.e. **DT(w,0) = new_distance**
Update the **priority** of **w** in PQ

(this is done by updating the priority of an element in the queue - **decreaseKey** operation. Complexity O(logV))

Update path to **w** to be **v**, i.e. **DT(w,1) = 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).

6. Spanning trees (for unweighted graphs)

Similar to finding the shortest path in unweighted graphs

Data structures needed:

A table (an array) T with size = number of vertices, where T_i = parent of vertex v_i
Adjacency lists
A queue of vertices to be processed

Algorithm

1. Choose a vertex **u** and store it in the queue. Set a counter = 0, and T_u = 0 (**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** from the queue.
For each $\mathbf{u_k}$ in the adjacency list of **v** do:

If $\mathbf{T_k}$ is empty,

$\mathbf{T_k = v}$
counter = counter + 1
store $\mathbf{u_k}$ in the queue

Complexity: O(E + V) - we process all edges and all nodes

7. Minimal spanning trees (weighted graphs) Prim's algorithm

The algorithm is similar to finding the shortest paths in 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 **T** with number of rows = number of vertices, and three columns:

  $\mathbf{T_{i,1}}$ **= True** if the vertex has been fixed in the tree, **False** otherwise. This is necessary because the graph is not directed and without this information we may enter a cycle.
  $\mathbf{T_{i,2}}$ = the length of the edge from the chosen parent (stored in the third column of the table) to the vertex $\mathbf{v_i}$,
  $\mathbf{T_{i,3}}$ = parent of vertex $\mathbf{v_i}$

- 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.

Algorithm:

1. Initialize first column to **False**, select a vertex **s** and store it in the priority queue with priority = 0, set $\mathbf{T_{s,2} = 0, T_{s,3} = root}$
   (It does not matter which vertex is chosen, because all vertices have to be in the tree.)
2. While there are vertices in the queue:
   a. **DeleteMin** a vertex **v** from the queue and set $\mathbf{T_{v,1} = True}$
   b. For all adjacent vertices **w**:
      - If $\mathbf{T_{w,1} = T}$ do nothing
      - If $\mathbf{T_{w,2}}$ is empty:
        - $\mathbf{T_{w,2}}$ = weight of edge **(v,w)** *(stored in the adjacency list)*
        - $\mathbf{T_{w,3} = v}$ *(this is the parent)*
        - append **w** in the queue with priority = weight of **(v,w)**
      - If $\mathbf{T_{w,2}}$ > weight of **(v,w)**
        - Update $\mathbf{T_{w,2}}$ = 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(logV))
        - Update $\mathbf{T_{w,3} = v}$

At the end of the algorithm, the tree would be represented in the table with its edges

$$\mathbf{\{(T_{i,3}\ ,\ v_i\ )\ |\ i = 1,\ 2,\ ...,V\}}$$

Complexity: O(Elog(V))
All edges have to be examined, and in the worst case each edge might cause updating of the priority queue. If adjacency matrix is used, the complexity would increase to $O(V^2)$

8. Minimal spanning trees (weighted graphs) Kruskal's algorithm

Kruskal's algorithm works with tree forests and the set of edges.

Algorithm:

1. Initially we build V trees consisting of one vertex only - each vertex is a tree of its own. The edges are stored in a priority queue with priority - the weight.
2. While the number of distinct trees is greater than one:

   DeleteMin an edge **(u,v)** from the priority queue.
   a. If **u** and **v** belong to one and the same tree, do nothing.
   b. If **u** and **v** belong to different trees, link the trees by the edge

Complexity of Kruskal's algorithm

A detailed analysis will show O(V) + O(Elog(E)) + O(Elog(V)).

- We need O(V) operations to build the initial forest with |V| trees each containing one node.
- The edges are stored in a priority queue and each time the smallest edge is retrieved, hence we need O(Elog(E)) operations to process the edges.
- Finally, the disjoint set operations are implemented by a tree with V nodes, hence we need O(Elog(V)) operations (a comparison is performed for each edge in the worst case).

Disregarding the lower term O(V) we get O(E (log(V) + log(E)).
At the worst case $E = O(V^2)$. Hence $\log(E) = O(\log(V^2)) = O(2\log(V)) = O(\log(V)$.
Thus we get complexity O(Elog(V)). On the other hand, V = O(E), hence we can reduce the complexity expression to 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)

9. Scheduling Networks

Activity: a task, has duration and prerequisites
Event: a point in time, characterized by the completion of one or several activities

A model of a project:

Directed simple acyclic graph
Simple graph means: each pair of nodes connected by at most one edge

Nodes: events (Source: starting event, Sink: terminating event
Edges: activities

Dummy activities with duration 0: used to make the graph simple if necessary.

Purpose of the model:

- find the earliest occurrence time of an event
- find the earliest completion time of an activity
- find the slack of an activity: how much the activity can be delayed without delaying the project
- find the critical activities: must be completed on time in order not to delay the project

We use topological sorting to determine the above characteristics of the project.

The earliest occurrence time of an event
EOT(j)= max(earliest completion times of all activities preceding the event)
EOT(source) = 0

The earliest completion time of an activity
ECT(j,k) is equal to EOT(j) + duration(j,k)

Algorithm to compute EOT and ECT

1. Sort the nodes topologically
2. For each event ***j***: initialize EOT(j) = 0
3. For each event ***i*** in topological order

For each event j adjacent from i:

ECT(i,j) = EOT(i)+ duration (i,j)
EOT(j) = max(EOT(j) , ECT(i,j))

Algorithm to find the slack of activities: latest occurrence time LOT(j)

LOT(sink) = EOT(sink)
For each non-sink event ***i*** in the reverse topological order:

LOT(i) = min(LOT(j) – duration(i,j))
For each activity (i,j)

Slack(i,j) = LOT(j) – ECT(i,j)

Critical activities: slack(j) = 0
Critical path in the project: all edges are activities with slack = 0

Complexity: O(V + E) with adjacency lists, $O(V^2)$ with adjacency matrix

10. Breadth-first search in a graph

BFS algorithm

1. Store source vertex **S** in a queue and mark as processed
2. While queue is not empty

Read vertex **v**from the queue
For all neighbors **w**:

If **w** is not processed

Mark as processed
Append in the queue
Record the parent of **w** to be **v**

We can use a distance table to mark vertices as processed by storing the distance to the source vertex.

Complexity

In step 1 we read a node from the queue. Each node is stored only once – if it is not processed. Thus the reading step will be performed O(V) times.

In step 2 we examine all neighbors, i.e. we examine all edges of the currently read node. Since the graph is not oriented, we will examine 2*E edges, where E is the number of the edges in the graph.

Hence the complexity of BFS is O(V+ 2*E)

If we use adjacency matrix instead of adjacency lists, the complexity will be $O(V^2)$

Attention: a typical error in determining the complexity is to multiply the number of the nodes by the number of the edges. Why we should not multiply the number of the edges by the number of the nodes?

11. Depth-first search in a graph

For a selected vertex **A** let **B1, B2, .. Bm** be adjacent vertices.
For all adjacent vertices, if **Bk**is not visited, visit **Bk** and all vertices reachable from **Bk**, before proceeding with **Bk+1.**

General algorithm:

Procedure dfs(s)

mark all vertices in the graph as not reached
invoke scan(s)

Procedure scan(s)

mark and visit **s**
for each neighbor **w** of **s**

if the neighbor is not reached

invoke scan(**w**)

Implementation

a. **Recursive implementation of depth-first search**

DepthFirst(Vertex v)

visit(v)
for each neighbor w of v

if (w is not visited)

add edge (v,w) to tree T
DepthFirst(w)

Visit(v)

mark v as visited

b. **Non-recursive implementation**

The non-recursive implementation uses a stack

Initialization:

mark all vertices as unvisited,

```
visit(s)

while the stack is not empty:

    pop (v,w)
    if w is not visited

        add (v,w) to tree T
        visit(w)

visit(v)

    mark v as visited
    for each edge (v,w)

        push(v,w) in the stack
```

12. Graph Connectivity

Connected graph: An undirected graph is said to be connected if there is a path between any two nodes

Biconnected graph: There are no vertices whose removal will disconnect the graph.

Articulation point (cut-vertex): A node whose removal disconnects the graph

Bridge: An edge whose removal disconnects the graph

Any edge in a graph, that does not lie on a cycle, is a bridge.

Strongly connected graph: A directed graph where for any pair of nodes there is a path from each one to the other

Unilaterally connected graph: A directed graph where for any pair of nodes at least one of the nodes is reachable from the other

Weakly connected graph: A directed graph where the underlying undirected graph is connected

Each strongly connected graph is also unilaterally connected. Each unilaterally connected graph is also weakly connected.

Back to Contents page

Created by Lydia Sinapova