Graphs

G(V,E)

V – set of vertices or nodes
E – set of edges connecting the vertices

Edge is represented by (u,v), u,v ∈ V
Directed & Undirected

Undirected Graph

Directed Graph

<table>
<thead>
<tr>
<th>Degree</th>
<th>Number of edges connected to the vertex</th>
</tr>
</thead>
<tbody>
<tr>
<td>In-degree</td>
<td>Number of edges coming in to the vertex</td>
</tr>
<tr>
<td>Out-degree</td>
<td>Number of edges going out of the vertex</td>
</tr>
</tbody>
</table>

Graph Representation

- **Adjacency Lists**
  - Preferred for Sparse Graphs $E << |V|^2$
  - Requires $\Theta(|V| + |E|)$ memory spaces

- **Adjacency Matrix**
  - Preferred for Dense Graphs $E \rightarrow |V|^2$
  - Requires $\Theta(|V|^2)$ memory spaces
  - To determine connectivity between vertices
  - Used by all-pairs shortest-paths algorithms
Adjacency List

1 → 2 → 5 /
2 → 1 → 5 → 3 → 4 /
3 → 2 → 4 /
4 → 2 → 5 → 3 /
5 → 4 → 1 → 2 /

Adjacency Matrix

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Graph Terminology

- **Path**: A path between two nodes is a sequence of intermediate nodes connected by edges.
- **Cycle**: A path in a graph is a cycle if its first and last vertices are identical, no other vertex appears more than once and if it contains at least two distinct edges.
- **Loop**: A loop is an edge with the start and the end nodes are the same.

Graph Terminology

- **Connected Graph**: A graph in which all nodes are connected.
- **Forest**: A graph that has more than one connected component.
- **Acyclic Graph**: A graph that doesn’t contain any cycle.
- **Tree**: Connected Acyclic graph is called a tree.
- **Spanning Tree**: A sub tree of a graph having all the nodes of the graph.
Breadth First Search

- Simple algorithm to search a graph
- Works on both directed and undirected graphs
- Given a source, \( S \in V \), it discovers every other reachable vertex.
- It produces the breadth-first tree with shortest paths between \( S \) and other vertices

BFS

- Breadth First Search, starts at any node (Level 0) of the graph and then scans for the nearest neighbors at Level 1.
- Then it scans for the nodes at Level 2 for all nodes at Level 1
- The algorithm ends when all nodes are visited
- Uses Queue to store the visited nodes
BFS: pseudo-code

BFS(G(V,E),s)

for each vertex u ∈ V[G] – {s}
    color[u] = white
    dist[u] = ∞
end for

color[s] = gray
dist[s] = 0
T=null
Q=null
Add(Q, s)

while Q ≠ null
    u = Delete(Q, s)
    for each v ∈ Adj_List[u]
        if color[v] = white
            color[v] = gray
            dist[v] = dist[u] + 1
            T = T U (u, v)
            Add(Q, v)
        end if
    end for
    color[u] = black
end while

BFS: Example

\[ Q \]
\[ T \]
\[ s, w, s, r, w, t, w, x, r, v, t, u, x, y \]
BFS: Run Time Analysis

- Initialization takes $O(|V|)$ time
- Adding and Deleting an element in Queue takes $O(1)$ time. Total time for queue operations is $O(|V|)$
- In the iteration the adjacency list of every vertex is scanned at most once. Sum of all lengths of all adjacency lists is $\Theta(|E|)$. Total time for scanning the list is $O(|E|)$
- Total Running Time of BFS is $O(|V| + |E|)$

BFS: Applications

- Compute Connected Components of a graph
- Compute a spanning tree of a graph
- Find a cycle in a graph
- Find whether the graph is a forest
- A path between two vertices with minimum number of edges
Depth First Search

- DFS scans for the neighbors at Level 1. When it finds the first neighbor, the algorithm proceeds to scan its neighbors at Level 2.
- Goes as deep into the graph as possible
- The algorithm is recursive and ends when all the nodes are visited
- Uses Stack implicitly

DFS: pseudo-code

DFS(G(V,E))
for each vertex u ∈ V[G]
    color[u] = white
    p[u] = nil
end for

for each vertex u ∈ V[G]
    if color[u] = white
        DFS-Visit(u)
    end if
end for

DFS-Visit(u)

DFS-Visit(u)
for each vertex u ∈ V[G]
    if color[v] = white
        p[v] = u
        DFS-Visit(v)
    end if
end for

color[u] = black
f[u] = time = time + 1
DFS: Example

DFS: Run Time Analysis

- The two loops in DFS procedure takes $O(|V|)$ time, exclusive of time to call DFS-Visit
- DFS-Visit is called only once for each vertex
- DFS-Visit scans the Adjacency list. Sum of all lengths of all adjacency lists is $\Theta(|E|)$. Total time for scanning the list is $O(|E|)$
- Total Running Time of DFS is $O(|V| + |E|)$
DFS: Applications

- Connected Components
- Topological Sort
- Find a cycle in a graph
- Find whether the graph is a forest

DFS vs. BFS

- When a breadth-first search succeeds, it finds a minimum-depth (nearest the root) goal node
- When a depth-first search succeeds, the found goal node is not necessarily minimum depth
- For a large tree, a depth-first search may take an excessively long time to find even a very nearby goal node, where as breadth-first search can find the nearby nodes faster.
Connected Components

**Problem:** Two vertices are in the same component of $G$ if and only if there is some path between them. Find the number of components in a graph.

**Input:** A directed or undirected graph $G$. A start vertex $s$.

**Connected Components pseudo-code**

```
Procedure Connected_Components G(V,E)
Input : G (V,E)
Output : Number of Connected Components
V' ← V;
c ← 0;
while V' ≠ 0 do
    choose u ∈ V';
    T ← all nodes reachable from u (by DFS_Tree);
    V' ← V' - T;
c ← c+1;
G_c ← T;
T ← 0;
```

**Connected Components Example**

**Input**

```
1 ---- 2
3 ---- 4
5 ---- 6
7 ---- 8
```

**Output**

```
1 ---- 2
3 ---- 4
5 ---- 6
7 ---- 8
```

---

**Topological Sort**

- **Problem:** Find a linear ordering of the vertices of $V$ such that for each edge $(i,j)$ in $E$, vertex $i$ is to the left of vertex $j$.
- **Input:** A directed, acyclic graph $G=(V,E)$
Topological Sort pseudo-code

1. TopologicalSort(G)
2. DFS(G)
3. As each vertex finishes, insert it on the front of a linked list
4. Return linked list of vertices

Topological Sort Example

Input

Output
Minimum Cost Spanning Tree

- Find a tree formed from graph edges that connects all the vertices in $G$, at minimum cost

- Kruskal’s Algorithm
- Prim’s Algorithm

Kruskal’s algorithm

- It is a greedy algorithm which sorts the edges in the order of minimum weight and adds them to the tree, till all the nodes are added
- The set formed is a forest and the edges that connect the forests are also included
Kruskal’s algorithm
pseudo-code

Algorithm KruskalMST(G)
T ← 0;
VS ← 0;
for each vertex v ∈ V do
    VS = VS ∪ {v};
sort the edges of E in non-decreasing order of weight
while |VS| > 1 do
    choose (v,w) an edge E of lowest cost;
delete (v,w) from E;
    if v and w are in different sets W1 and W2 in VS do
        W1 = W1 ∪ W2;
        VS = VS - W2;
        T ← T ∪ (v,w);
return T

Kruskal’s Algorithm
Example
Kruskal’s Algorithm Example

CSE 5311

Kruskal’s Algorithm Example

CSE 5311
Kruskal’s Algorithm Example

CSE 5311

Kruskal’s Algorithm Example

CSE 5311
Kruskal’s Algorithm
Run Time Analysis

- Sorting the edges take $O(E \log E)$ time
- Adding the edges to the tree takes $O(E)$ time
- So, the running time for the algorithm is in $O(E \log E)$

Prim’s Algorithm

- Prim’s algorithm for minimum spanning trees has a similar style to Dijkstra’s algorithm for shortest paths
- The set formed is always a tree
- The edge with minimum weight added always connects a node that is not in the tree
Prim’s algorithm pseudo-code

Algorithm PrimMST(G, r)
for each vertex v ∈ V do
  key[u] = infinity
T = NIL
key[r] = 0
Q = V
T = NULL
while Q ≠ NULL do
  u = Extract-Min(Q)
  for each v ∈ Adj[u]
    if v ∈ Q and w(u,v) < key[v]
      key[v] = w(u,v)
      T ← T ∪ (u,v)
  return T

Prim’s Algorithm Example

The minimum spanning tree is shown with bold edges. The total edge cost is 16.
We start with v1, and place this into the tree. The rest of the vertices are not in the tree. The cheapest edge we can add from v1 to the other vertices is (v1, v4).

Now v1 and v4 are in the tree. The cheapest edge we can add is either (v1, v2) or (v4, v3). We choose (v1, v2).
Now v1, v2, and v4 are in the tree. The cheapest edge we can add is (v4, v3).

Now v1, v2, v3, and v4 are in the tree. The cheapest edge we can add is (v4, v7).
Now v1, v2, v3, v4, and v7 are in the tree. The cheapest edge we can add is (v7, v6).

Prim’s Algorithm Example

Now v1, v2, v3, v4, v6 and v7 are in the tree. The cheapest edge we can add is (v7, v5). We are now done. This is optimal.
Prim’s Algorithm Run Time analysis

- The initialization takes $O(V)$ time
- The Extract-Min takes $O(\log V)$ time if it is implemented as binary heap
- For $V$ iterations Extract-Min takes $O(V\log V)$ time
- The key assignment takes $O(\log V)$ time
- For all edges the key assignment takes $O(E \log V)$ time.
- Hence the total time is $O(V \log V + E \log V) = O(E \log V)$. This is same as Kruskal’s algorithm

MCST: Applications

- Used in Networking protocols
  - Multicast Routing
Single Source Shortest Path

Given graph (directed or undirected) \( G = (V,E) \) with weight function \( w: E \to \mathbb{R} \) and a vertex \( s \), find for all vertices in \( V \) the minimum possible weight for path from \( s \) to \( v \).

- BellMan-Ford’s Algorithm
- Dijkstra’s Algorithm

Relaxing an Edge

Relaxing an edge consists of testing whether we can improve the shortest path to \( v \) by going through \( u \).

If there is a shortest path then update Distance[\( v \)] and Predecessor[\( v \)] = \( u \)

\[ \text{Relax}(u,v) \]

If \( d[v] > d[u] + w(u,v) \)

\[ d[v] = d[u] + w(u,v) \]

\[ p[v] = u \]
Bellman-Ford’s Algorithm

The algorithm uses relaxation
The $d[v]$, the weight of the path from source $s$ to destination $v$, is relaxed progressively until it reaches the minimum weight
The algorithm returns true only if the graph does not contain any negative cycles

Bellman-Ford’s Algorithm pseudo-code

BellmanFord(graph (G,w), vertex s)
for each vertex $v \in V$
  $d[v] = \infty$
  $p[v] = \text{NIL}$
  $d[s] = 0$
for $i \leftarrow 1$ to $|V| - 1$
do
  for $(u,v) \in E[G]$
do
    Relax($u,v$)
  for $(u,v) \in E[G]$
do
    if $d[v] > d[u] + w(u,v)$ then
      return false
return true
Bellman-Ford’s Algorithm Example

![Graph with weights showing an example of the Bellman-Ford algorithm.]
Bellman-Ford’s Algorithm Example

```
0  6  8  7
0  7  8  2
0  6  8  7
```

Bellman-Ford’s Algorithm Example

```
0  6  8  7
0  7  8  2
0  6  8  7
```
Bellman-Ford’s Algorithm Example

Bellman-Ford’s Algorithm Example
Bellman-Ford’s Algorithm
Run Time Analysis

- The Bellman-Ford algorithm computes the distance from source s to all other vertices of G.
- Determine that G contains a negative-weight cycle,
- In O(VE) time.

Dijkstra’s Algorithm

- Faster than Bellman-Ford’s Algorithm
- Dijkstra’s algorithm is based on the greedy method
- It starts with the source and builds the spanning tree by adding one edge at a time which gives the shortest path from the source to the vertex not in the spanning tree.
Dijkstra's Algorithm 
 pseudo-code

Dijkstra(G,s)

for each vertex v ∈ V

d[v] = infinity
p[v] = NIL
d[s] = 0
S = NULL
Q = V

while Q ≠ NULL do
u = Extract-Min(Q)
S ← S ∪ {u}
for each v ∈ Adj[u]
    Relax(u,v)

Dijkstra’s Algorithm

- It assumes the path length does not increase when more edges are added to the path. If a node with a negative incident edge were to be added late to the vertex list for which decisions have been made, it could mess up distances for vertices already in the list.

- This is almost identical to that of Prim's algorithm. The only difference here is that the Dijkstra's algorithm stores the distance from the source to current vertex, while Prim's algorithm stores the cost of the minimum-cost edge connecting a vertex in V to \( u \).
Dijkstra’s Algorithm Example

Dijkstra’s Algorithm Example
Dijkstra’s Algorithm
Example

Dijkstra’s Algorithm
Example
Dijkstra’s Algorithm Example

Dijkstra’s Algorithm Example
Dijkstra’s Algorithm Example

Dijkstra’s Algorithm Example
Dijkstra’s Algorithm Example

Dijkstra’s Algorithm Example
Dijkstra’s Algorithm

Example

Run Time Analysis

- The initialisation takes \( O(V) \) time
- There are \( V \) iterations of Extract-Min
- There are \( E \) iterations of distance assignment
- Time = \( V \times T_{\text{Extract-Min}} + E \times T_{\text{dist assign}} \)

<table>
<thead>
<tr>
<th>Q</th>
<th>( T_{\text{EXTARCT-MIN}} )</th>
<th>( T_{\text{DIST-ASSIGN}} )</th>
<th>Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>( O(V) )</td>
<td>1</td>
<td>( O(V^2) )</td>
</tr>
<tr>
<td>Binary Heap</td>
<td>( O(\log V) )</td>
<td>( O(\log V) )</td>
<td>( O(E \log V) )</td>
</tr>
<tr>
<td>Fibonacci Heap</td>
<td>( O(\log V) )</td>
<td>1</td>
<td>( O(E + V \log V) )</td>
</tr>
</tbody>
</table>
SSSP: Applications

- Used in Routing Protocols
  - Dijkstra’s Algorithm is used in Link State Routing
  - Bellman-Ford’s Algorithm is used in Distance Vector Routing

All Pairs Shortest Path

The all-pairs-shortest-path problem is generalization of the single-source-shortest-path problem for all nodes. Problem is to find the shortest paths between all pairs of vertices $V_i, V_j \in V$ such that $i \neq j$. 
Floyd’s Algorithm

- \( d_{s,t}^{(i)} \) – the shortest path from \( s \) to \( t \) containing only vertices \( v_1, ..., v_i \)
- \( d_{s,t}^{(0)} = w(s,t) \)
- \( d_{s,t}^{(k)} = w(s,t) \) if \( k = 0 \)
- \( \min\{d_{s,t}^{(k-1)}, d_{s,k}^{(k-1)} + d_{k,t}^{(k-1)}\} \) if \( k > 0 \)

### Floyd’s Algorithm

**pseudo-code**

\[
\text{FloydWarshall}(\text{matrix } W, \text{ integer } n) \\
D^{(0)} \leftarrow W \\
\text{for } k \leftarrow 1 \text{ to } n \text{ do} \\
\hspace{1em} \text{for } i \leftarrow 1 \text{ to } n \text{ do} \\
\hspace{2em} \text{for } j \leftarrow 1 \text{ to } n \text{ do} \\
\hspace{3em} d_{ij}^{(k)} \leftarrow \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) \\
\text{return } D^{(n)}
\]
Floyd’s Algorithm Example

Adjacency Matrix

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>3</td>
<td>8</td>
<td>∞</td>
<td>-4</td>
</tr>
<tr>
<td>2</td>
<td>∞</td>
<td>0</td>
<td>∞</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>∞</td>
<td>4</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>∞</td>
<td>-5</td>
<td>0</td>
<td>∞</td>
</tr>
<tr>
<td>5</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>6</td>
<td>0</td>
</tr>
</tbody>
</table>

Floyd’s Algorithm Example

Shortest Path Matrix D(1)

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>3</td>
<td>8</td>
<td>∞</td>
<td>-4</td>
</tr>
<tr>
<td>2</td>
<td>∞</td>
<td>0</td>
<td>∞</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>∞</td>
<td>4</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>5</td>
<td>-5</td>
<td>0</td>
<td>-2</td>
</tr>
<tr>
<td>5</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>6</td>
<td>0</td>
</tr>
</tbody>
</table>
Floyd’s Algorithm Example

Shortest Path Matrix $D^{(2)}$

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>3</td>
<td>8</td>
<td>4</td>
<td>-4</td>
</tr>
<tr>
<td>2</td>
<td>$\infty$</td>
<td>0</td>
<td>$\infty$</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>$\infty$</td>
<td>4</td>
<td>0</td>
<td>5</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>5</td>
<td>-5</td>
<td>0</td>
<td>-2</td>
</tr>
<tr>
<td>5</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>6</td>
<td>0</td>
</tr>
</tbody>
</table>

Floyd’s Algorithm Example

Shortest Path Matrix $D^{(3)}$

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>3</td>
<td>8</td>
<td>4</td>
<td>-4</td>
</tr>
<tr>
<td>2</td>
<td>$\infty$</td>
<td>0</td>
<td>$\infty$</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>$\infty$</td>
<td>4</td>
<td>0</td>
<td>5</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>-1</td>
<td>-5</td>
<td>0</td>
<td>-2</td>
</tr>
<tr>
<td>5</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>6</td>
<td>0</td>
</tr>
</tbody>
</table>
Floyd’s Algorithm Example

Shortest Path Matrix $D^{(4)}$

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>3</td>
<td>-1</td>
<td>4</td>
<td>-4</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>0</td>
<td>-4</td>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>4</td>
<td>0</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>-1</td>
<td>-5</td>
<td>0</td>
<td>-2</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>5</td>
<td>1</td>
<td>6</td>
<td>0</td>
</tr>
</tbody>
</table>

Floyd’s Algorithm Example

Shortest Path Matrix $D^{(5)}$

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>3</td>
<td>-1</td>
<td>2</td>
<td>-4</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>0</td>
<td>-4</td>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>4</td>
<td>0</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>-1</td>
<td>-5</td>
<td>0</td>
<td>-2</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>5</td>
<td>1</td>
<td>6</td>
<td>0</td>
</tr>
</tbody>
</table>
### Floyd’s Algorithm

#### Example

The Shortest Path Matrix $D^{(5)}$ is shown below:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>3</td>
<td>-1</td>
<td>2</td>
<td>-4</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>0</td>
<td>-4</td>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>4</td>
<td>0</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>-1</td>
<td>-5</td>
<td>0</td>
<td>-2</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>5</td>
<td>1</td>
<td>6</td>
<td>0</td>
</tr>
</tbody>
</table>

#### Run Time Analysis

- Running Time of $O(|V|^3)$
- Negatively weighed edges may be present.
- Negatively weighted cycles cause problems with the algorithm.
**Negative Cycle Detection**

- Execute Floyd’s Algorithm. If at least one diagonal entry contain a negative value then there is a negative cycle
  - Complexity is $O(V^3)$
- Execute Bellman-Ford Algorithm. If it returns false then the graph contain a negative cycle
  - Complexity is $O(VE)$

**Faster Algorithm for All Source Shortest Path**

- Try using the existing Single Source Shortest Path for all sources
- If Bellman-Ford’s algorithm is again to find the shortest paths for all sources then the complexity is $O(V^2E) = O(V^4)$
- Bellman-Ford’s algorithm is worse than Floyd’s algorithm
Faster Algorithm for All Source Shortest Path

- Use Dijkstra’s Algorithm to compute shortest paths for all sources
- Runs in time $O(V \cdot (E \log V))$ with binary heap and $O(V^2 \cdot (E + V \log V))$ with Fibonacci heaps
- But cannot have negative edges

Reweighting the edges

- A reweighting function $\hat{w}$ should satisfy two properties
  - For all pairs of vertices $u, u \in V$, a path $p$ is a shortest path from $u$ to $v$ using the weight function $w$ if and only if $p$ is also a shortest path from $u$ to $v$ using weight function $\hat{w}$
  - For all edges $(u,v)$ the new weight $\hat{w}(u,v)$ is non-negative
Reweighting the edges

For a weighted, directed graph $G=(V,E)$ with weight function
- $w : E \rightarrow \mathbb{R}$ (weight of edges)
- $h : V \rightarrow \mathbb{R}$ (distance vector calculated using Bellman-Ford’s algorithm)

$$\hat{w}(u,v) = w(u,v) + (h(u) - h(v))$$

Johnson’s Algorithm

1. Add a node $S$ to the graph with zero weight edges connecting to all nodes of the graph
   - It is done to calculate the weighting function using Bellman-Ford’s algorithm. If any one node does not have path to all other nodes then the weight of some nodes will be infinite
Johnson’s Algorithm

2. Execute Bellman-Ford’s algorithm on the new graph
3. If it returns false then there is a negative cycle in the graph and hence abort the algorithm
4. Reweigh all the edges

Johnson’s algorithm

5. For all nodes execute Dijkstra’s algorithm to find the shortest path to all other nodes
6. Recalculate the correct shortest-path weight
   \[ \text{dist}(u)(v) = \hat{g}(u,v) - (h(v) - h(u)) \]
Johnson’s algorithm Example

Original Graph

Graph with the ‘s’ node added with zero weight edges
Johnson’s algorithm Example

Weight function calculated using Bellman Ford’s algorithm

Reweighted non-negative edges
Johnson's algorithm

Example

Run Time analysis

- Find the weighting function $h$ such that $w(u, v) \geq 0$ for all edges by using Bellman-Ford, or determine that a negative-weight cycle exists.
  - Time = $O(VE)$.
- Run Dijkstra's algorithm from each vertex using $w$.
  - Time = $O(VE \log V)$. (when Fibonacci heap is used)
- Reweight each shortest-path length $w(p)$ to produce the shortest-path lengths $w'(p)$ of the original graph.
  - Time = $O(V^2)$.
- Total time = $O(VE \log V)$. Better for sparse graphs.
**Representation of Graph Files**

- **GraphWiz** is a set of graph drawing tools developed by the AT&T Research Labs.
- Two types of file format:
  - `.dot` for directed graphs
  - `.neato` for undirected graphs
- GraphWiz and WinGraphViz are open source software. Contains both executables and libraries.

**.dot file**

```plaintext
/* Nodes = 4 */
/* Edges = 5 */
/* Directed = 1 */
/* Min In-Degree = 2 */
/* Max In-Degree = 4 */
/* Min Out-Degree = 2 */
/* Max Out-Degree = 4 */
/* Weighted = 1 */
/* Min Weight = 3 */
/* Max Weight = 10 */

digraph G {
  size = "7,7";
  1 -> 3 [label="6"];  // Random edge
  2 -> 3 [label="3"];  
  2 -> 2 [label="5"];  
  3 -> 4 [label="7"];  
  3 -> 3 [label="3"];  
}
```
### .neato file

```
/* Nodes = 4 */
/* Edges = 5 */
/* Directed = 0 */
/* Min Degree = 2 */
/* Max Degree = 4 */
/* Weighted = 1 */
/* Min Weight = 3 */
/* Max Weight = 10 */
```

graph G {
    size = "7,7";
    1 -- 3 [label="6"];  
    2 -- 3 [label="3"];  
    2 -- 2 [label="5"];  
    3 -- 4 [label="7"];  
    3 -- 3 [label="3"];  
}
```

### Graphs Algorithms Implementation

![Graph Algorithm Implementation](image-url)
Graph Algorithms Implementation

Graph Algorithms Implementation

Graph Algorithms Implementation
References

- Donald B. Johnson, The Pennsylvania state university, Efficient Algorithms for shortest path in Sparse Networks
- WinGraphViz http://www.research.att.com/sw/tools/graphviz