OME Algorithms

Single-source shortest paths

input: graph G, source vertex s ▷

$d[s] \leftarrow 0$
$\text{PARENT}[s] \leftarrow \text{None}$
for $v \leftarrow V - \{s\}$:
  $d[v] \leftarrow \infty$
  $\text{PARENT}[v] \leftarrow \text{None}$
end

Algorithm 1: Relaxation Technique Initialisation
$O(n) - n$ vertices

input: from vertex u, to vertex v, weights w ▷

if $d[v] > d[u] + w(u, v)$ **then**
  $d[v] \leftarrow d[u] + w(u, v)$
  $\text{PARENT}[v] = u$
end

Algorithm 2: Relaxation Technique Relax Edge
$O(1)$ except if we perform additional steps

input: graph G, weights w, source vertex s
output: cycle present c

relaxation_init(G, s) ▷ Algorithm 1
$(V, E) \leftarrow G$
$n \leftarrow |V|$
for $i \leftarrow 1$ to $(n - 1)$ **do** ▷ nodes indexed from 1, s will already be correct
  for $u, v \leftarrow E$ **do** ▷ consider in any arbitrary order
    relaxation_relax(u, v, w) ▷ Algorithm 2
  end
end
$c \leftarrow \bot$
for $u, v \leftarrow E$ **do**
  if $d[v] > d[u] + w(u, v)$ **then**
    $c \leftarrow \top$ ▷ negative cycle reachable from s
  end
end

Algorithm 3: Bellman-Ford Algorithm
$\Theta(nm) - n$ vertices, m edges

input: graph G, weights w, source vertex s
output: cycle present c

```
relaxation_init(G, s)                                   ▷ Algorithm 1
(V, E, Adj) ← G
Q ← Queue()                                             ▷ create a new empty queue
Q ↞ s
c ←⊥
while not Q.empty do
    u ← Q.dequeue()
    for v ← Adj[u] do
        relaxation_relax(u, v, w)                       ▷ Algorithm 2
        if [edge was relaxed] then
            if v not in Q then
                Q ↞ v
            end
        end
    end
    if [negative cycle present] do                      ▷ algorithm has been omitted
        c ← ⊤
        break
    end
end
```

Algorithm 4: Bellman-Ford Algorithm with FIFO Queue
$O(nm) - n$ vertices, m edges

input: graph G, weights w, source vertex s
assumption: all edge weights are non-negative

```
1  (V, E, Adj) ← G
2  relaxation_init(G, s)                          ▷ Algorithm 1
3  S ← ∅                                          ▷ keep track of nodes that we know d[v] is δ(s, v)
4  Q ← PriorityQueue(V)                           ▷ write all nodes excluding s into a priority queue
5  while not Q.empty do
6     u ← Q.extract_min()                         ▷ find u with minimum d[.] value in Q and remove it from the queue
7     S ← S | {u}                                 ▷ mark this vertex as visited
8     for v ← Adj[u] do
9        relaxation_relax(u, v, w)                ▷ Algorithm 2
10       if [d[v] has reduced] then for modified Dijkstra's algorithm
11          Q ↞ v
12          S ← S − {u}                           ▷ mark this vertex as unvisited
13       end
14    end
15 end
```

Algorithm 5: Dijkstra's Shortest-paths Algorithm
$O(\min\{n^2, m \log n\}) - n$ vertices, m edges
Unordered List: $O(n^2) - n$ vertices
Ordered List: $O(mn) - n$ vertices, m edges
Heap: $O(m \log n) - n$ vertices, m edges

Directed acyclic graph shortest paths

input: graph G, weights w, source vertex s

```
(Adj) ← G
V ← topological_sort(G)
relaxation_init(G, s)
for u ← V do
    for v ← Adj[u] do
        relaxation_relax(u, v, w)
    end
end
```

Algorithm 6: Directed Acyclic Graph Shortest Paths Algorithm

$O(n + m) - n$ vertices, m edges

All-pairs shortest paths

If all edge weights are non-negative, we repeatedly use Dijkstra's algorithm. No method with better (worst-case) running time is known.

For negative edge weights, we can use:

- Bellman-Ford, $O(n^2m)$, $\Theta(n^4)$ if $m = \Theta(n^2)$ (Algorithm 3)
- Floyd-Warshall algorithm, $\Theta(n^3)$
- Johnson's algorithm, $O(nm \log n)$

input: graph G, weights w
output: shortest paths M

$(V, E) \leftarrow G$
$G' \leftarrow \text{copy}(G)$ ▷ create a clone of the graph
$(V', E') \leftarrow G'$
$V' \leftarrow V \mid \{s\}$ ▷ create a new node s
$E' \leftarrow E \mid \{(s, v) \mid v \in V\}$ ▷ create new edge from s to v
for $v \leftarrow V$ **do**
  $w(s, v) \leftarrow 0$ ▷ set the edge weights to 0
end
$c \leftarrow$ bellman_ford(G', w, s) ▷ Algorithm 3
if c **then**
  terminate ▷ there is a negative cycle so we do not continue
end
for $v \leftarrow V$ **do**
  $h[v] \leftarrow \delta(s, v)$ ▷ copy computed $\delta(s, v)$ from Bellman Ford algorithm to our distance array
end
$w' \leftarrow w$ ▷ clone the weights
for $u, v \leftarrow E$ **do**
  $w'(u, v) = w(u, v) + h[u] - h[v]$
end
$M \leftarrow ()$ ▷ empty matrix
for $u \leftarrow V$ **do**
  dijkstra(G, w', u) ▷ Algorithm 5, compute $\delta(u, v)$ for all $v \in V$
  for $v \leftarrow V$ **do**
    $M_{uv} \leftarrow \delta(u, v) - [h[u] - h[v]]$
  end
end

Algorithm 7: Johnson's Algorithm

$$O(\min\{n^3, nm \log n\}) - n \text{ vertices}, m \text{ vertices}$$
$$\underbrace{O(nm)}_{\text{bellman-ford}} + \underbrace{n \cdot O(\min\{n^2, m \log n\})}_{n \text{ dijkstra's}} + \underbrace{O(n^2)}_{\text{other computation}}$$

Maximum-flow problem

input: graph $G\ (V, E, c, s, t)$
output: maximum flow f

$f \leftarrow$ zero flow ▷ $f(u, v) = 0$ for each (u, v) in E
while $\top$ **do**
  $G_f \leftarrow$ [construct residual network]
  $P \leftarrow$ [find augmenting path in G_f using depth-first search]
  if P **then**
    $f \leftarrow f_p^f$ ▷ path augmentation
  else
    break
  end
end

Algorithm 8: Ford-Fulkerson Method (DFS)

input: graph $G\ (V, E, l, u, s, t)$
output: maximum flow f

$f \leftarrow$ zero flow ▷ $f(u, v) = 0$ for each (u, v) in E
while $\top$ **do**
  $G_f \leftarrow$ [construct residual network]
  $P \leftarrow$ [find shortest augmenting path in G_f using breadth-first search]
  if P **then**
    $f \leftarrow f_p^f$ ▷ path augmentation
  else
    break
  end
end

Algorithm 9: Edmonds-Karp Method (BFS)