Single Source Shortest Paths

Given a connected weighted directed graph $G(V, E)$, associated with each edge $\langle u, v\rangle \in E$, there is a weight $w(u, v)$. The *single source shortest paths* (SSSP) problem is to find a shortest path from a given source r to every other vertex $v \in V - \{r\}$. The weight (length) of a path $p = \langle v_0, v_1, \ldots, v_k\rangle$ is the sum of the weights of its constituent edges:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i).$$

The weight of a shortest path from u to v is defined by $\delta(u, v) = \min\{w(p) : p \text{ is a path from } u \text{ to } v\}$.

1 Negative Cycles

In some applications, graphs may contain edges with negative weights. Such edges may result in negative cycles (e.g. Figure 1). If there is a negative weight cycle which is reachable from the source s, the weight $\delta(s, v)$ is *undefined*, where v is a vertex in the cycle.

2 Representation of a shortest path tree

The shortest path tree rooted at s is a directed subgraph $G' = (V', E')$ where $V' \subseteq V$ and $E' \subseteq E$ such that every $v \in V'$ is reachable from s; and for all $v \in V'$, the unique simple path in G' from s to v is a shortest path from s to v in G. In other words, for each vertex $v \in V$, we maintain a predecessor $p(v)$ that is either a vertex or NIL. Thus, the output for a single shortest paths problem is a tree rooted at the source.

3 Initialization

The algorithms introduced here make use of a common initialization that is as follows:

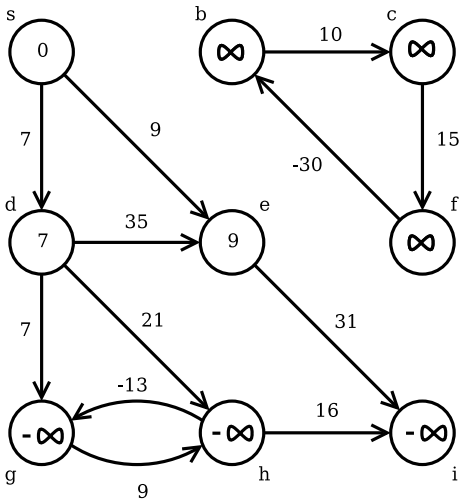

Figure 1: Shortest paths from root vertex s. Negative cycles are formed by g and h, as well as b, c, and f. $\delta(a, g)$, $\delta(a, h)$, and $\delta(a, i)$ are $-\infty$ because of the negative cycle formed by g and h. The weights for b, c and f are ∞, despite of the negative cycle, because they are unreachable from root vertex s.

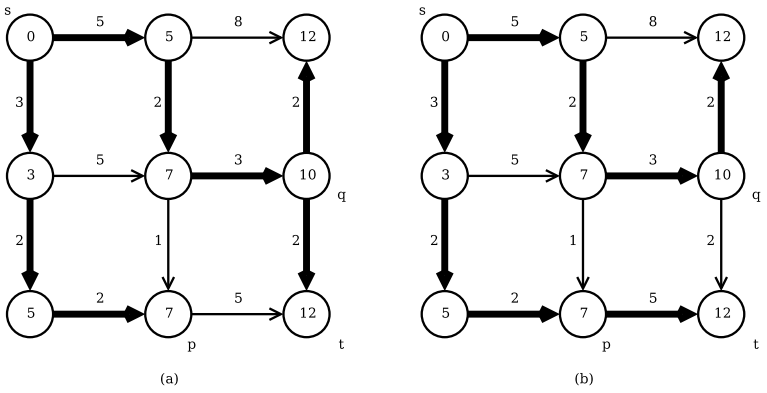

Figure 2: Shortest Path Tree. The shortest path tree rooted at s has its edges in bold. Two variants (a) and (b) are shown for the same graph. In (a) vertex t is reached through the edge (q, t), whereas in (b), edge (p, t) is used instead.

```
INITIALIZE(G, s)
1  d[s] ← 0
2  p[s] ← NIL
3  for all v ∈ V − {s}
4       do d[v] ← ∞
5          p[v] ← NIL
```

4 Shortest Paths and Relaxation

The main technique used by the shortest path algorithms introduced here is relaxation, a method that repeatedly decreases an upper bound on the length of an actual shortest path for each vertex until the upper bound equals the length of the shortest path.

For each vertex v, we maintain an attribute $d[v]$ which is an upper bound on the length of a shortest path from source s to v. $d[v]$ is also called the *shortest path estimate*.

```
RELAX(u, v, w)
1  if d[v] > d[u] + w(u, v)
2     then d[v] ← d[u] + w(u, v)
3          p[v] ← u
```

5 Dijkstra's Algorithm

Dijkstra's algorithm assumes that all the edges in G are non-negative. The algorithm maintains a set S of vertices whose final shortest path weights from the source s have already been determined. That is, for all the vertices $v \in S$, we have $d[v] = \delta(s, v)$.

The algorithm repeatedly selects a vertex $u \in V - S$ with the minimum shortest-path estimate, inserts u to S, and relaxes all the edges leaving u. Also, a priority queue Q that contains all the vertices in $V - S$ is maintained, keyed by their d values. Figure 3 shows the execution of the DIJKSTRA-SHORT algorithm listed below.

```
DIJKSTRA-SHORT(G, w, s)
1  INITIALIZE(G, s)
2  S ← ∅
3  Q ← V
4  while Q ≠ ∅
5        do u ← EXTRACT-MIN(Q)
6           S ← S ∪ {u}
7           for each vertex v ∈ Adj[u]
8                do RELAX(u, v, w)
```

6 The Bellman-Ford Algorithm

Like Dijkstra's algorithm, the Bellman-Ford algorithm uses the technique of relaxation, progressively decreasing an estimate $d[v]$ on the weight of a shortest path from a source s to each other vertex $v \in V$ until it reaches the actual shortest path weight $\delta(s, v)$. The algorithm is capable of detecting negative cycles and returns true if and only if the graph contains no negative cycles that are reachable from the source. If BELLMAN-FORD returns true, then we have $d[u] = \delta(s, v)$ for all vertices v. Also, for all $v \in V$ not reachable from s, $\delta(s, v) = \infty$. Figure 4 shows the execution of the BELLMAN-FORD algorithm listed below on an example graph.

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE(G, s)
2  for i ← 1 to |V| − 1
3       do for each edge (u, v) ∈ E
4               do RELAX(u, v, w)
5  for each edge (u, v) ∈ E
6       do if d[v] > d[u] + w(u, v)
7             then return FALSE
8  return TRUE
```

7 Single-Source Shortest Paths in DAGs

By relaxing the edges in a DAG according to their topological sort of its vertices. We can achieve $\Theta(n + m)$ time complexity.

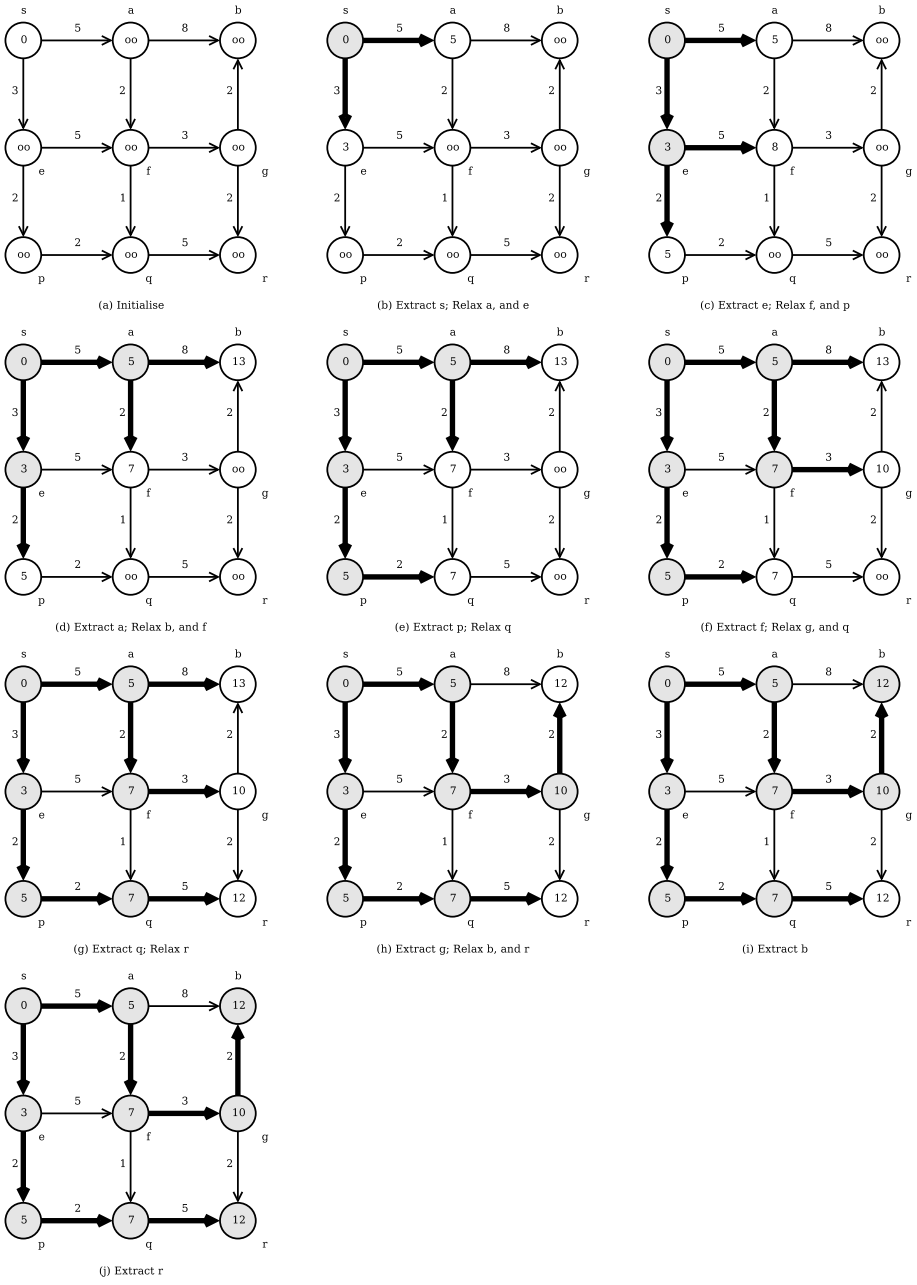

Figure 3: Execution of DIJKSTRA-SHORT. Step (a) shows the state of the graph after initialisation. In step (b), vertex s (the source) is extracted (shaded gray) from Q and the adjacent vertices a and e relaxed. Since $d[a] > d[s]+w(s,a)$, $d[a]$ is updated and $p[a]$ set to s (arrow from s to a is made bold). Ditto for vertex e. In step (c), vertex e is extracted (and shaded) as it is Q's minimum. As before, its adjacent vertices are relaxed. Step (d) illustrates the RELAX mechanism. Here a is extracted and b and f relaxed. In doing so, f's parent is updated to a (formerly e). This execution carries on until Q is empty.

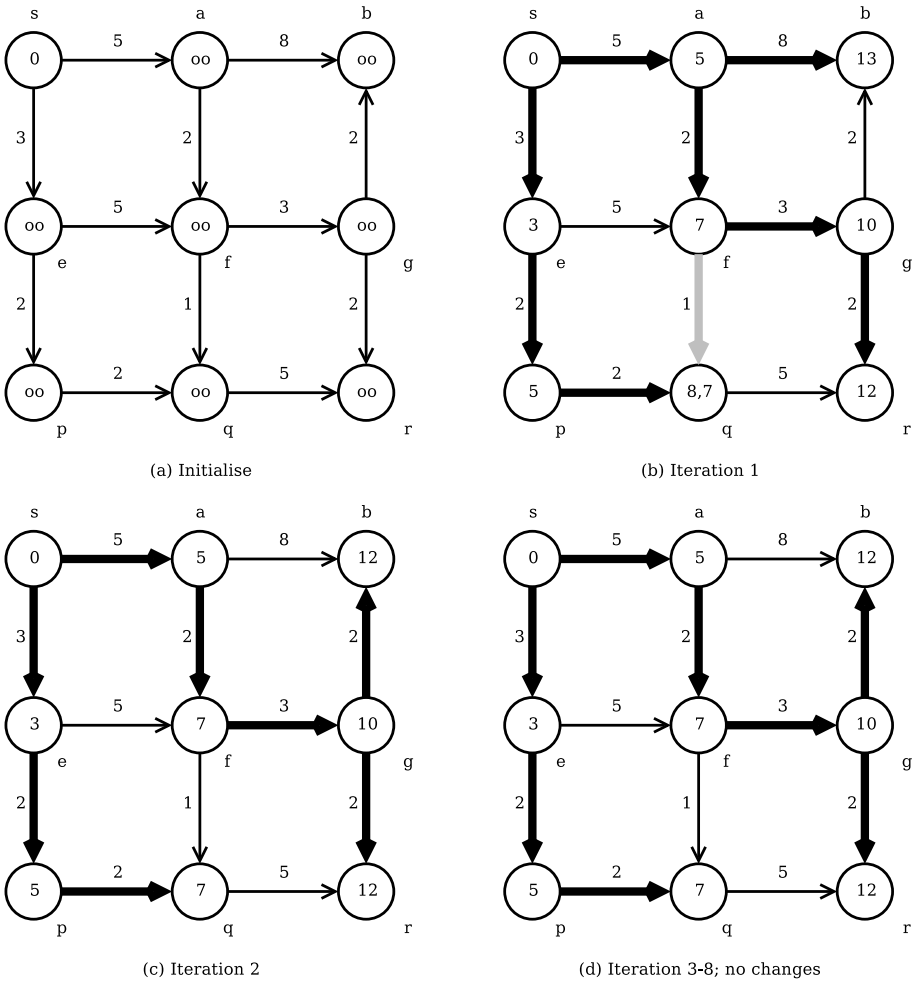

Figure 4: Executing Bellman-Ford. Step (a) shows the state of the graph after initialisation. In each of the $|V| - 1$ iterations, we assume that the edges are visited in a left to right order (s, a), (s, b), (s, e), (a, f), $\ldots (q, r)$. Due to this visiting order, within iteration 1, vertex q is first relaxed through edge (f, q), setting $d[q]$ to 8. Soon after, edge (p, q) is considered and $d[q]$ set to 7. Accordingly, $p[q]$ is also updated.

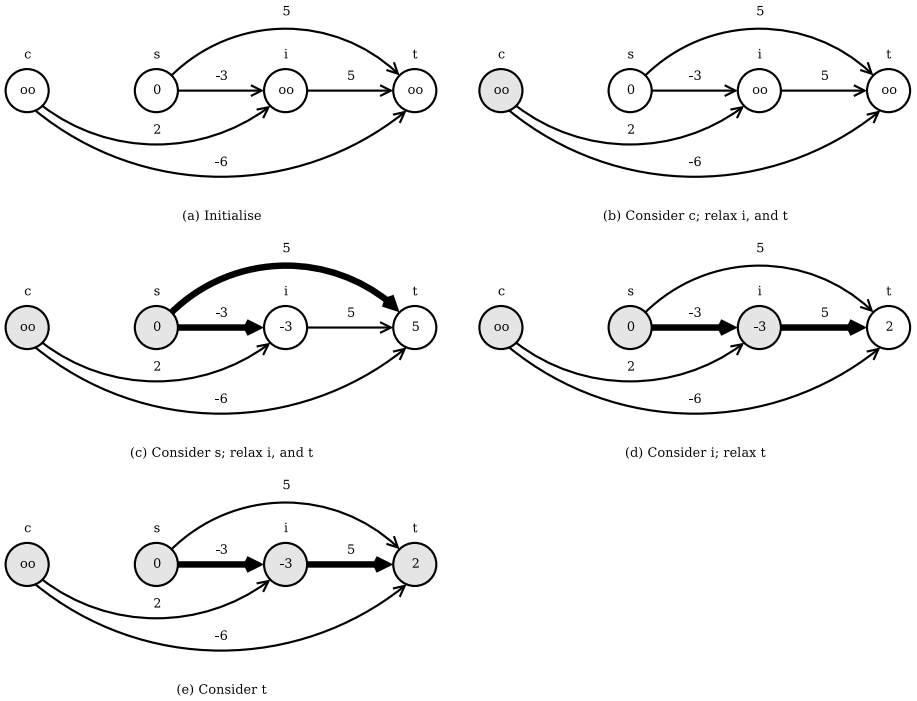

Figure 5: Excution of DAG-Shortest. Step (a) shows the state of the graph after initialisation. Following this, the vertices are considered in topological order: c, s, i, and then t. As each vertex u is considered, vertices in $Adj[u]$ are relaxed.

DAG-Shortest(G, w, s)
1 Topologically sort the vertices of G
2 Initialize(G, s)
3 **for** each vertex u taken in topologically sorted (increasing) order
4 **do for** $v \in Adj[u]$
5 **do** Relax(u, v, w)

Figure 5 shows an example execution of DAG-Shortest on a DAG.

8 Difference constraints and shortest paths

Linear Programming: Given an $m \times n$ matrix A, an m-vector b, and an n-vector c, we wish to find a vector x of n elements that maximise the *objective function* $\sum_{i=1}^{n} c_i x_i$ subject to the m constraints given by $Ax \leq b$.

Approaches to solving linear programming.

- Simplex algorithm
- Ellipsoid algorithm
- Karmarkkar's algorithm

9 Systems of difference constraints

Special linear programming: each row of the linear programming matrix A contains 1 or -1, and all other entries of A are 0s. Thus, the constraints can be expressed as

$$x_j - x_i \leq b_k$$

where $1 \leq i, j \leq n$ and $1 \leq k \leq m$.

For example, a set of four unknowns $\{x_1, x_2, x_3, x_4\}$, and 6 **difference constraints**:

$$\begin{aligned} x_1 - x_2 &\leq 6 \\ x_1 - x_3 &\leq -4 \\ x_2 - x_3 &\leq -10 \\ x_3 - x_4 &\leq 14 \\ x_4 - x_2 &\leq -4 \\ x_4 - x_1 &\leq -10 \end{aligned}$$

... can be represented as ...

$$\begin{pmatrix} 1 & -1 & 0 & 0 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & -1 & 0 \\ 0 & 0 & 1 & -1 \\ 0 & -1 & 0 & 1 \\ -1 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} \leq \begin{pmatrix} 6 \\ -4 \\ -10 \\ 14 \\ -4 \\ -10 \end{pmatrix}$$

Given a system $Ax \leq b$ of difference constraints, the corresponding **constraint graph** is a weighted directed graph $G = (V, E)$ where

$$V = \{v_0, v_1, v_2, \ldots, v_n\}$$

... where n is the number of unknowns, v_0 is a special source vertex created, and v_i corresponds to x_i for $1 \leq i \leq n$. Also, the set of edges ...

$$E = \{\langle v_i, v_j \rangle : x_j - x_i \leq b_k \textit{ is a constraint}\} \cup \{\langle v_0, v_i \rangle : i = 1, \ldots, n\}$$

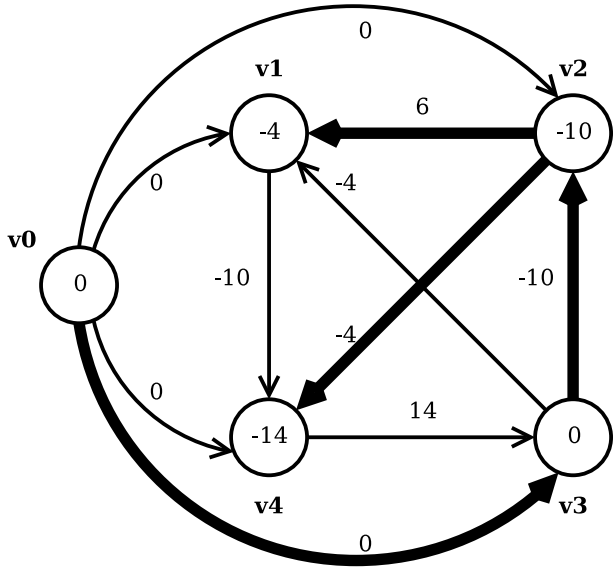

Figure 6: A **constraint graph** constructed from our on going example using simple rules: (1) Each unknown x_i maps onto a vertex v_i. (2) For each constraint $x_j - x_i \leq b_k$, there is an edge (v_i, v_j) with weight b_k. (3) Add a vertex v_0 and have edges from it to all other vertices with weight 0. Next, use one of the SSSP algorithms to solve $d[v_i]$ for $0 \leq i \leq n$; using v_0 as the source.

Theorem: Given a system $Ax \leq b$ of difference constraints, let G be the corresponding constraint graph. If G contains no negative weight cycles, then

$$x = \{\delta(v_0, v_1), \delta(v_0, v_1), \ldots, \delta(v_0, v_n)\}$$

is a feasible solution for the system. Figure 6 shows how this is done for the example with four unknowns above.