OME: Optimisation Methods

Contents

Single-Source Shortest Paths

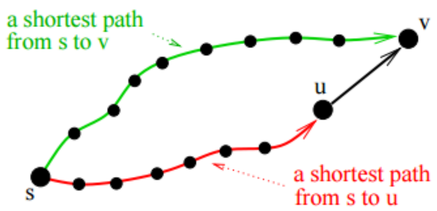
Objective: given a weighted directed graph, find the **minimum-weight paths** from a single vertex s to **all other vertices**. Algorithms that solve this problem can also be adapted to solve **point-to-point** problems.

Notation

- $G = (V, E)$ is a **directed graph** G with vertices V and edges E.
  - Vertices may also be referred to as nodes or sites.
  - $|V| = n$, $|E| = m$.
- $w(u, v)$ is the **weight of the edge** (u, v).
- $s \in E$ is the **source** vertex.
- $p = \langle v_1, v_2, v_3, ..., v_k \rangle$ is a **path** from vertex v_1 to v_k via v_2, v_3, etc.
- $w(p) = w(v_1, v_2) + w(v_2, v_3) + ... + w(v_{k-1}, v_k)$ is the **weight of a path**, equal to the sum of the weights of all edges in the path.
- $\delta(u, v)$ is the **minimum of** $w(p)$ for all p from u to v, or $+\infty$ if no such p exists.
  - Multiple minimum-weight paths may exist.
  - A shortest path from u to v is any path $p = \langle u, ..., v \rangle$ such that $w(p) = \delta(u, v)$.

Useful Facts

- A sub-path of a shortest path is also a shortest path.
  - If a shortest path from a to e is $\langle a, b, c, d, e \rangle$, a shortest path from b to d is $\langle b, c, d \rangle$.
- For each edge $(u, v) \in E$, it holds that $\delta(s, v) \leq \delta(s, u) + w(u, v)$.
  - This states that the weight of the shortest path between s and v is **at most** the weight from s to u, plus the weight of the edge from u to v.
  - This is an example of the **triangle inequality** and is shown in the following diagram:



Negative Weights

Negative weights can be useful/essential in solving a problem - the most obvious example is that of finding the **maximum-weight** paths. Rather than create new algorithms, it is possible to negate every edge and then find the minimum-weight paths as normal.

The **minimum-weight paths for negated edges** are equivalent to the **maximum-weight paths for non-negated edges**.

Exchange Rate Example

Taking the example of a graph showing exchange rates between currencies, the most profitable path **maximises the product** of its edge weights. This is unsuitable, because the algorithms we study **minimise the sum** of edges along a path.

This can be resolved by taking the negative logarithm of every edge. If $\gamma(u, v)$ is the exchange rate from u to v, $w(u, v) = -ln(\gamma(u, v))$. [1] This works because the sum of logarithms of a set of numbers is equal to the logarithm of their products [2].

However, we **cannot avoid negative weights** now: if $\gamma(u, v) > 1$ then $w(u, v) < 0$.

Negative Cycles

If a negative cycle exists on a path from s to v, repeated journeys around this cycle will continually reduce the weight of the path. Therefore, if a negative cycle exists on a path from s to v, $w(s, v) = -\infty$.

We consider only shortest-paths algorithms that:

- Compute the shortest paths for graphs where **no negative cycles are reachable** from the source.
- **Correctly detect when negative cycles are reachable** from the source and are not required to compute anything else.

Why not look for the **shortest simple paths** (i.e. avoid cycles). It's a valid question, but it's NP-hard and we do not cover it in this section. *See more: Optimisations for NP-Hard Problems, page 32*.

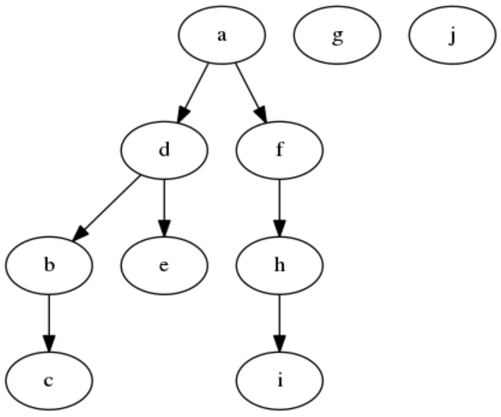
Shortest-Path Trees

For each vertex v reachable from the source s, we want to find any one of the minimum-weight paths that exist. If there are no negative cycles, these paths can be easily represented with a **shortest-paths tree**. Such a tree contains exactly one shortest path from s to each v reachable from s.

[1] Any fixed logarithm base greater than 1 would work.
[2] $ln(a) + ln(b) + ln(c) = ln(abc)$

The tree T can be represented by an array $parent[v], v \in V$ in $\Theta(n)$ space. $parent[v]$ is the parent of the vertex v in the tree T. An optional array $d[]$ can store the weight of the shortest paths to each vertex. For example, from the source vertex a:

v	a	b	c	d	e	f	g	h	i	j
$parent[v]$	nil	d	b	a	d	a	nil	f	h	nil
$d[v]$	0	6	9	2	4	3	∞	5	6	∞



A disadvantage of this representation is that only one path for each reachable vertex can be stored. Listing every path explicitly leads to $O(n^2)$ space requirements.

Relaxation Technique

This is the most common approach for solving single-source shortest paths problems. It uses an **initialisation** step followed by a sequence of **relaxation** steps. Algorithms following this pattern differ in their **termination condition(s)** and the **order that edges are relaxed**.

At each vertex v, the following is maintained:

- $d[v]$ is the shortest-path **estimate** for reaching v from the source s. It is an upper-bound.
- $parent[v]$ is the current predecessor of v.

Initialisation

This stage sets the upper-bound for all non-source vertices at infinity and creates no parent relationships.

```
fun INITIALISE(G, s):
    d[s] = 0
    parent[s] = nil
    for each vertex v in V − {s}, do:
        d[v] = inf
        parent[v] = nil
```

Initialisation is a $\Theta(n)$ operation. For large graphs this can be too slow, so an alternative is to only initialise s at the start of the algorithm and then **initialise other vertices as they are discovered**.

Relaxation

Applied to an edge (u, v), this stage checks whether a path to v via u would have a lower weight than the current estimate of reaching v, $d[v]$. If so, $d[v]$ is updated with the weight of the path via u and the parent relationship is updated so that $parent[v] = u$. This process stems from the triangle inequality (*see more: Useful Facts, page 4*).

```
fun RELAX(u, v, w):
    if (d[v] > d[u] + w(u, v)):
        d[v] = d[u] + w(u, v)
        parent[v] = u
```

Relaxation is a $\Theta(1)$ operation.

Relaxation Properties

- For every vertex v, throughout the computation...
  - $d[v]$ will only **decrease**.
  - $d[v]$ is either ∞ or the weight of some path from s to v.
  - $d[v] \geq \delta(s, v)$.
- An **effective relaxation operation** is defined as an operation that decreases $d[v]$.
- During computation, there is an edge $(u, v) \in E$ such that $d[v] > d[u] + w(u, v)$ (i.e. there is an effective relaxation operation) if and only if there is a vertex x such that $d[x] > \delta(s, x)$.
  - From this, it follows that if no effective relaxation operations remain then all shortest paths have been found.
- If there **is no negative cycle** reachable from s...
  - There will be a **finite number of effective relaxation operations**.
  - The $parent$ pointers will form a tree rooted at s.
  - When no effective relaxation operations remain, then for each vertex v, $d[v] = \delta(s, v)$ and $parent[v]$ points to v's predecessor on a shortest path from s.
- If there **is a negative cycle** reachable from s...
  - There will always be an effective relaxation operation that can be applied.
  - The $parent$ pointers will eventually form a cycle. This suggests that negative cycles in the graph can be detected by periodically checking $parent$ for cycles.

Checking for Cycles

If the $parent$ array contains a cycle then the graph contains a negative cycle, so computation should indicate this and terminate.

A cycle can be detected by following parent pointers from all vertices, marking vertices as visited during traversal, and reducing work by avoiding re-visiting any vertices. On each traversal, if a vertex that was already seen is encountered then a cycle exists.

```
fun CHECK—CYCLE(parent):
    visited = [false, false, ...]
    for each v in V, do:
        if (!visited[v]):
            seen = { }
            while (true):
                visited[v] = true
                seen.add(v)

                next = parent[v]
                if (next == nil):
                    break

                if (seen.contains(next)):
                    return CYCLE

                if (visited[next]):
                    break

                v = next

    return NO_CYCLE
```

Bellman-Ford Algorithm

The Bellman-Ford algorithm is a **relaxation-based algorithm** for finding shortest single-sources paths of a graph. It **permits negative-weight edges**.

```
fun BELLMAN—FORD(G, w, s):

    // run initialisation and relaxation:
    INITIALISE(G, s)
    for i from 1 to n − 1, do:
        for each edge (u, v) in G, do:
            RELAX(u, v, w)

    // determine outcome:
    for each edge (u, v) in G, do:
        if (d[v] > d[u] + w(u, v)):
            return FALSE // negative cycle reachable from s
        else:
            return TRUE // no negative cycle, d[] now holds shortest path weight
```

Correctness

If a **negative cycle is reachable** from s then the algorithm will always **correctly** return `FALSE` because an effective relaxation will always be possible.

If **no negative cycles are reachable** from s then the algorithm will always **correctly** return `TRUE` because at the end of the first section no more effective relaxations will be possible. This is guaranteed, because of two lemmas:

- If the shortest simple path from s to v is of length k, then after k iterations of the first loop $d[v] = \delta(s, v)$.
  - Assume the k-length path is $\langle s, x_1, x_2, ..., v\rangle$.
  - After the first iteration $d[x_1] = \delta(s, x_1)$.
  - After the second iteration $d[x_2] = \delta(s, x_2)$.
  - ...
  - After k iterations $d[k] = \delta(s, k)$.
- The shortest path from s to any vertex contains at most $n-1$ edges, because the longest non-cyclic path will visit every vertex once.

Running Time

This algorithm **runs the** $\Theta(1)$ **relaxation step exactly** $n - 1$ **times** for each edge in the first section, then in the second section it checks to see if any further relaxations are possible. If more effective relaxations are possible, a negative cycle must exist and `FALSE` is returned; if no effective relaxations are possible then the shortest path weights have been found and `TRUE` is returned.

The running time is $\Theta(nm)$. The worst case for **any** single-source shortest paths problem with negative weights is $\Omega(nm)$.

Improving the Running Time

The $\Theta(nm)$ running time comes from the first section of the algorithm, so we focus our efforts there:

- We could try to **decrease the number of iterations** of the outer loop by terminating it early in certain conditions:
  - Not every graph will require $n - 1$ iterations to find the solution. If **no effective relaxation operations took place** in a given iteration the loop can **terminate early** because the shortest paths were already computed.
  - Some iteration may create a cycle in the $parent$ array. It can be checked periodically (after every iteration?); if a $parent$ **cycle is found** then a negative loop is reachable from s and the entire algorithm can **terminate early** with `FALSE`.
- We could try to **reduce the work done in each iteration** of the outer loop by considering only edges that will give effective relaxations.
  - A vertex u is **active** if its outgoing edges have not been relaxed since the last time $d[u]$ was decreased.
  - We perform relaxations only on the edges out of active vertices.
  - We store active vertices in a **FIFO** queue.

Optimisations: Early Termination and Queue of Active Vertices

This optimisation keeps track of active vertices and **only considers relaxations of their outgoing edges**, as described above.

```
fun BELLMAN—FORD—FIFO(G, w, s):

    // initialisation
    INITIALISE(G, s)
    Q = empty queue // active vertices
    Q.enqueue(s)

    // relaxation of active vertices
    while (Q is not empty), do:
        u = Q.dequeue()
        for each v in adj[u], do:
            RELAX(u, v, w)

        if (CHECK—CYCLE(parent)):
            return FALSE

    return TRUE
```

The relaxation method is augmented to **enqueue the vertex** v if $d[v]$ is updated when considering the edge (u, v) (i.e. if it is 'active').

```
fun RELAX(u, v, w):
    if (d[v] > d[u] + w(u, v)):
        d[v] = d[u] + w(u, v)
        parent[v] = u

        // enqueue v if it was updated
        if (v is not in Q):
            Q.enqueue(v)
```

Note that a mechanism for detecting negative cycles has been added, because Q will never be empty if a negative cycle is reachable from s. This could be in the form of checking for cycles in $parent$ after each set of relaxations.

The running time is now $O(nm)$, but still $\Theta(nm)$ in the worst case.

Dijkstra's Algorithm

- Critical assumption: **all weights are non-negative**.
  - As a consequence, there are also no negative cycles.
- We assume that all vertices are reachable from s.

A **set** S is maintained of all vertices for which $d[v] = \delta(s, v)$ (i.e. vertices that are 'finished'). A **min-priority queue** Q is maintained of all non-finished vertices and their current path weight estimate.

```
fun DIJKSTRA(G, w, s):
    INITIALISE(G, s)
    S = []
    Q = V

    while (Q is not empty), do:
        u = Q.removeMin()
        S.add(u)
        for each v in adj[u], do:
            RELAX(u, v)
```

The relaxation method is augmented to **update the min-priority queue** value of the vertex v if $d[v]$ is updated when considering the edge (u, v).

```
fun RELAX(u, v, w):
    if (d[v] > d[u] + w(u, v)):
        d[v] = d[u] + w(u, v)
        parent[v] = u
        Q.decreaseKey(v, d[v])
```

Running Time

- The data structures will be instantiated $\Theta(1)$ times.
- The main **`while`** loop will run $\Theta(n)$ times, because all vertices start in the queue, each vertex is removed, and no vertex is replaced.
- A total of $\Theta(n)$ calls to `Q.removeMin()` will be made.
- A total of $\Theta(m)$ relaxations will take place, each of which may include a call to `Q.decreaseKey()`.

It is clear that the priority queue implementation will be the main factor in the algorithm's running time. There are three implementations to consider, each with different running times:

- **Unsorted array**:
  - Instantiation is $\Theta(n)$.
  - `removeMin()` requires an $O(n)$ search and will be done $\Theta(n)$ times.
  - `decreaseKey()` requires a $\Theta(1)$ update and will be done $O(m)$ times.
  - The total running time is $\Theta(n) + O(n^2) + \Theta(m) = O(n^2)$.
- **Sorted array**:
  - Instantiation is $\Theta(n)$.
  - `removeMin()` requires a $\Theta(1)$ removal and will be done $\Theta(n)$ times (this assumes that values are not shifted to fill the gap created).
  - `decreaseKey()` requires an $O(n)$ re-ordering and will be done $O(m)$ times.
  - The total running time is $\Theta(n) + \Theta(1) + O(mn) = O(mn)$.
    - Note: $n - 1 \leq m \leq n(n-1)$, so the total is $O(n^3)$ for a dense graph.
- **Heap**:
  - Instantiation is $\Theta(n)$.
  - `removeMin()` requires an $O(log_2(n))$ removal and will be done $\Theta(n)$ times.
  - `decreaseKey()` requires an $O(log_2(n))$ re-ordering and will be done $O(m)$ times.
  - The total running time is $\Theta(n) + O(n \cdot log_2(n)) + O(m \cdot log_2(n))) = O(m \cdot log_2(n))$.

If the input graph is **dense**, such that $m = \Omega(n^2/log_2(n))$, then the **unsorted array** implementation of the priority queue gives a better worst-case running time. If the graph is **not dense**, the **heap implementation** gives a better worst-case running time.

For most applications the input graphs are **not dense**, so Dijkstra's algorithm is **assumed to use a heap-based priority queue**.

Correctness - Invariants

We establish **invariants** for the algorithm, based on each vertex's membership in S or Q.

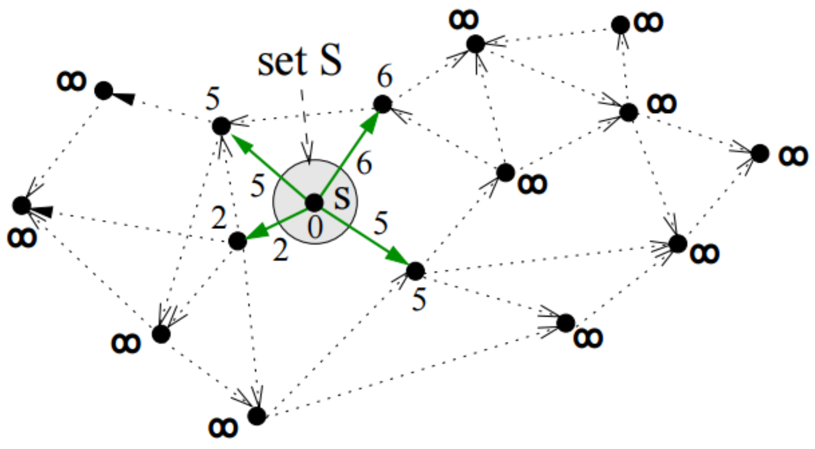
1. For each $v \in S$:
   (a) v is in the current tree.
   (b) $d[v]$ is the shortest-path weight from s to v, i.e. $d[v] = \delta(s, v)$.
   (c) The path from s to v in the current tree is a shortest path from s to v.
2. For each $z \notin S$ such that there **is an edge** (v, z) for some vertex $v \in S$:
   (a) z is a leaf in the current tree.
   (b) The path from s to z in the current tree is a shortest path from s to v where all intermediate nodes are in S.
   (c) $d[z]$ is the weight of this path (the tree path from s to z).
3. For each $y \notin S$ such that there **is no edge** (v, y) for some vertex $v \in S$:
   (a) y is not in the current tree.
   (b) $parent[y] = nil$
   (c) $d[y] = \infty$

At the end of the algorithm's execution $S = V$, so only the first set of invariants apply for each vertex. These state that $d[v] = \delta(s, v)$ for all vertices v and that the current three is the shortest path three from s to all other vertices. Therefore, if these invariants hold, the algorithm is correct.

Correctness - Induction on Invariants

Basis step: prove that the invariants hold for the first iteration ($i = 0$).

At the first iteration, s is taken from Q, added to S, and all outgoing edges are relaxed. This produces the state pictured below, for which all invariants hold.



Inductive step: prove that if the invariants hold for iteration i, they also hold for $i + 1$.

- Assume that the invariants hold at the end of some iteration i (possibly the first iteration, described above).
- Let u denote the vertex selected in the iteration $i + 1$.
- Show that the invariants hold at the end of iteration $i + 1$ when u has moved from Q to S.

Proving Invariant 1b

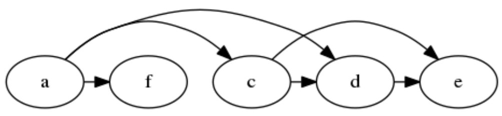
- We have selected vertex u to add to S, so we must prove that at the end of the iteration $d[u] = \delta(s, u)$.
- We already know that $d[u] \geq \delta(s, u)$ because it is a basic property of the relaxation operation, so we must prove $d[u] \leq \delta(s, u)$.
- Select any path R from s to u and show that $w(R) \geq d[u]$.
  - Split R into $R1$ and $R2$, such that $R1$ ends at the first vertex y outside of S.
  - $w(R) = w(R1) + w(R2) \geq d[y] + w(R2) \geq d[u] + w(R2) \geq d[u]$
  - Therefore, no path from s to u has a smaller weight than $d[u]$, so $d[y] \leq \delta(s, u)$.
- This works because u has the lowest d-value, therefore $d[y] \geq d[u]$, and all edge weights are non-negative, therefore $d[y] + w(R2) \geq d[u]$.

Directed Acyclic Graphs (DAGs)

The objective is the same as before: find the **minimum-weight paths** from a single vertex s to **all other vertices**. Dijkstra's algorithm won't work because negative-edges are possible; the Bellman-Ford algorithm could work in $O(mn)$, but the restricted graph format (DAG) allows for a $\Theta(n + m)$ algorithm.

Topological Order

The algorithm relies on creating a **topological order** for the vertices in the graph. A topological order rearranges vertices (without changing edges) so that all edges point in the same direction, as pictured:



Once the topological order has been created, the algorithm is simple:

```
fun DAG—SHORTEST—PATHS(G, w, s):
    TOPO—SORT(G)
    INITIALISE(G)
    for each vertex u in topological order, do:
        for each vertex v in adj[u], do:
            RELAX(u, v, w)
```

The running time is made up of one $\Theta(n+m)$ topological order generation, one $\Theta(n)$ initialisation and $\Theta(m)$ relaxations, giving $\Theta(n+m)$.

The algorithm works because when a vertex is considered all of its incoming edges have already been relaxed, therefore $d[u] = \delta(s, u)$ when u is considered.

Geographic Networks

When looking for a **single destination** d, one approach is to run Dijkstra's algorithm from the source and stop when d is considered (i.e. removed from Q). This may require checking the entire network, wasting a lot of effort.

Another approach is to **run two Dijkstra computations**, one searching 'forwards' from the source and another searching 'backwards' from the destination. When a shortest path from s to d is found both computations can stop. This **bi-directional** search can result in a smaller part of the networking being examined, but is very tricky to implement with an optimal stopping condition.

When we have a geographic network we know the **coordinates** of each vertex. With this extra information we can compute the **straight-line distance between vertices**, which is a lower-bound on the shortest paths between them. Straight-line distances to the destination can be used to **re-weight** edges.

Re-Weighting Edges

Graph edges can be re-weighted so that edges leading geographically *towards* d become cheaper and edges leading *away* from d get more expensive. In a similar fashion to Johnson's Algorithm (*see more: Johnson's Algorithm, page 16*), edges can re-weighted as follows:

$$\hat{w}(u,v) = w(u,v) - straight\_line(u,d) + straight\_line(v,d)$$

Values in $\hat{w}$ are always ≥ 0, satisfying the requirements of Dijkstra's algorithm.

Note that this is a **heuristic** that will improve on the average-case running time, but **will not improve the worst-case** scenario.

All-Pairs Shortest Paths

Objective: given a weighted directed graph, find the **minimum-weight path** from **all vertices to all other vertices**. This will create $n(n-1)$ paths and weights.

The output is two $n * n$ matrices:

- D, such that $D[i,j] = \delta(i,j)$.
- P, such that $P[i,j]$ holds the parent of j in a path from i to j.

For simplicity, we will only consider computation of the first matrix.

Existing Algorithms

- If all edge weights are **non-negative** we can run **Dijkstra's algorithm** from each vertex.
  - $n \cdot O(min\{m \cdot log_2(n), n^2\}) = O(min\{mn \cdot log_2(n), n^3\})$.
  - This is the best worst-case running time that is known.
- In the general case where negative edges are allowed we can run the **Bellman-Ford** algorithm from each vertex.
  - $n \cdot O(mn) = O(mn^2)$, up to $O(n^4)$.

Floyd-Warshall Algorithm

This algorithm runs in $\Theta(n^3)$ and is not covered in this course.

Johnson's Algorithm

This algorithm runs in $O(min\{mn \cdot log_2(n), n^3\})$, making it **suitable for sparse graphs** but **unsuitable for dense graphs**.

It works by **re-weighting edges** to remove negatives, then applying **Dijkstra's algorithm** at every vertex.

Re-Weighting Edges

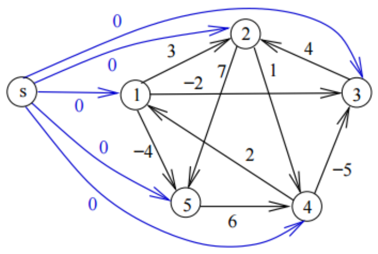
We must change the weight of edges to **remove negatives without changing shortest paths**. Adding some large value to each edge won't work because the shortest paths may change (because the path length becomes a major component in its weight).

Johnson's algorithm re-weights edges by first assigning a value $h(v)$ to each vertex v and then creating an updated set of weights $\hat{w}$ such that $\hat{w}(i,j) = w(i,j) + h(i) - h(j)$. That is, each edge weight is increased by its source vertex h-value and decreased by its destination vertex h-value.

For any path the h-values of all intermediate nodes **cancel each other out**. For a path $P = \langle v_1, v_2, ..., v_k \rangle$ its new weight is calculated as $\hat{w}(P) = w(P) + h(v_1) - h(v_k)$. All paths between v_1 and v_k are offset by the same amount regardless of their length, so shortest paths are not affected.

Calculating h-values

The **Bellman-Ford** algorithm can be used to calculate the h-value for each vertex. An 'imaginary vertex' s is created with a 0-weight edge to all vertices of the graph, and then the shortest path weights are found from s. Once complete, $h(v) = d[v] = \delta(s, v)$ for all vertices in the 'real' graph.



Full Algorithm

```
fun JOHNSON(G, w):
    D = output matrix

    // add imaginary vertex
    V' = V + { s }
    E' = E + { (s, v) for all v in V }
    w(s, v) = 0 for all v in V
    G' = (V', E')

    // detect negative cycle or find h—values
    if (BELLMAN—FORD(G', w, s) == false):
        return false // negative cycle
    else:
        // re—weight edges
        for each vertex v in V, do:
            h(n) = d[v]
        for each edge (u, v) in E, do:
            w'(u, v) = w(u, v) + h(u) — h(v)

        // run Dijkstra and record (adjusted) result in D
        for each vertex u in V, do:
            d' = DIJKSTRA(G, w', u)
            for each vertex v in d', do:
                D[u, v] = d'[v] — h(u) + h(v)

    return D
```

Running Time

The running time is made up of one iteration of the Bellman-Ford algorithm and n iterations of Dijkstra's algorithm.

$$O(mn) + n \cdot O(min\{m \cdot log_2(n), n^2\}) = O(min\{mn \cdot log_2(n), n^3\})$$

Correctness

- **Detecting negative cycles**: we know that the Bellman-Ford algorithm can do this correctly and that the 'imaginary vertex' s cannot create a cycle (because it has no incoming edges), so therefore this algorithm can correctly detect negative cycles.
- **Computing shortest path weights**: we know that Dijkstra's algorithm can do this correctly and we apply it at every vertex, so therefore this algorithm can correctly compute the shortest path weights.

Network Flow Problems

Notation

- $G = (V, E, c, s, t)$ is a **flow network** G with vertices V and edges E.
  - Vertices may also be referred to as nodes or sites.
  - $|V| = n$, $|E| = m$.
- $c(u, v)$ is the **capacity of the edge** (u, v).
- $s \in E$ is the **source** vertex.
- $t \in E$ is the **sink/destination** vertex.
- $s \neq t$.

Types of Flow Problem

- **Max-Flow**: find a **maximum flow** from the **source** to the **sink** of a flow network. A maximum flow sends the **maximum amount of the underling commodity** from the source to the sink without exceeding the capacity of any edge.
- **Flow Feasibility** (Transshipment Problem): find a flow which satisfies edge capacities *and* the specified supply/demand values at various vertices.
  - This can be reduced to the maximum flow problem.
- **Minimum Cost Flow**: find a **maximum flow** or a **supply/demand satisfying flow** that minimises the cost incurred by using each edge.
- **Multi-Commodity Flow**: find a **maximum flow**.

Flow Formalisation

We assume that if $(u, v) \in E$ then $(v, u) \in E$. Edges with zero capacity are added when required, but usually not shown on diagrams.

A **flow is a function** $f : E \mapsto \mathbb{R}$ where $f(u, v) \geq 0$ is the flow on the edge (u, v) with the following properties:

- **Capacity constraints**: for each edge $(u, v) \in E$, $0 \leq F(u, v) \leq c(u, v)$.
- **Flow conservation**: for each vertex $v \in V - \{s, t\}$, the total flow in to v is equal to the total flow out of v (i.e. the net flow through v is zero).
- **Single-direction flow**: for each edge $(u, v) \in E$, if $f(u, v) > 0$ then $f(v, u) = 0$.

The **value of a flow** is the net flow into the sink (which is the same as the net flow out of the source, because of flow conservation):

$$|f| = \sum_{(x,t)\in E} f(x,t) - \sum_{(t,z)\in E} f(t,z)$$

$$|f| = \sum_{(s,x)\in E} f(s,x) - \sum_{(z,s)\in E} f(z,s)$$

For a given flow f, if $f(u,v) = c(u,v)$ then f is said to **saturate** the edge (u,v) and the edge (u,v) is said to be a **saturated edge**.

From Flows to Paths

Given a network $G = (V, E, c, s, t)$ and a flow $f : E \mapsto \mathbb{R}$, the flow f can be **decomposed** into at most m paths from s to t, where $m = |E|$.

Algorithm: **select and remove maximal** $\langle s, ..., t\rangle$ **flow paths** from f until f is empty. Each path removes all remaining flow from at least one edge, so at most m paths are created.

Each path can be found in $O(n)$ and there are $O(m)$ paths, so a flow can be decomposed into paths in $O(mn)$. This is less than the time needed to find a maximum flow, so this is okay.

Note that this algorithm does not work when **flow cycles** exist, but can be modified to accommodate them.

Max-Flow Problems

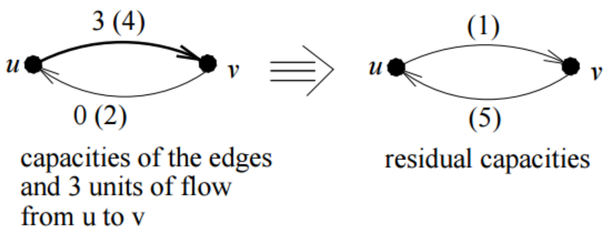
Maximum flow problems are solved by continually adding flow to a network. The amount that can be added (i.e. the network's remaining capacity) is the **residual capacity**.

Residual Capacity

If f is a flow defined in the network $G = (V, E, c, s, t)$, then residual edges c_f are defined as follows:

- $c_f(u, v) = c(u, v) - f(u, v)$ if $f(u, v) \geq 0$ and $f(v, u) = 0$.
  - When there is flow from u to v, the residual capacity of (u, v) decreases by the amount of that flow.
- $c_f(u, v) = c(u, v) + f(u, v)$ if $f(u, v) = 0$ and $f(v, u) \geq 0$.
  - Where there is flow from v to u, the residual capacity of (u, v) increases by the amount of that flow.

In the example below, $c(u, v) = 4$, $c(v, u) = 2$ and $f(u, v) = 2$. The residual edges $c_f(u, v) = 1$ and $c_f(v, u) = 5$ are created.



This shows that we can send 1 more unit of flow from u to v, or we can send 5 unit from v to u by 'sending back' the 3 that are already being sent in the other direction, plus saturating the edge (v, u).

If $c_f(u, v) > 0$ then (u, v) is a **residual edge**.

The **residual network** of G induced by the flow f is the flow network $G_f(V, E_f, c_f, s, t)$ where c_f are the residual capacities and E_f is the set of residual edges.

Adding Flow Using the Residual Network

If f is a flow in the network G and f' is a flow in the residual network G_f, then f and f' can be combined to generate a new flow h in G.

The new flow $h = f \bigoplus f'$ in G is defined as follows:

For each (u, v) such that $f(u, v) \geq 0$ and $f(v, u) = 0$:

- If $f'(u,v) \geq 0$ and $f'(v,u) = 0$, then:
  - $h(u,v) = f(u,v) + f'(u,v)$
  - $h(v,u) = 0$
  - i.e. when the flows go in the **same direction** just add them together and leave the opposite direction as zero.
- If $f'(u,v) = 0$ and $f'(v,u) > 0$, then:
  - $h(u,v) = max\{f(u,v) - f'(v,u), 0\}$
  - $h(v,u) = max\{f'(v,u) - f(u,v), 0\}$
  - i.e. when the flows go in **opposite directions**, cancel out the smaller flow and add whatever is left.

The value of h is defined as $|h| = |f| + |f'|$.

General Approach for Finding Maximum Flow

- Start with f as a zero flow (i.e. $f(u,v) = 0$ for all $(u,v) \in E$).
- Loop forever:
  - Construct the residual network G_f.
  - Find a non-zero flow f' in G_f.
    - If no such flow exists, exit loop.
  - $f = f \bigoplus f'$ *(combine the flows)*
- Return f as the maximum flow.

Most maximum flow algorithms use this approach.

Side Note: Cuts

A cut (S,T) in a network divides the nodes into **two disjoint groups** S and T such that $S \subseteq V$ and $T = V - S$. The source and sink nodes are in different groups, such that $s \in S$ and $t \in T$.

The **capacity** of a cut is the sum of capacities of all edges from S to T:

$$c(S,T) = \sum_{(u,v) \in E: u \in S, v \in T} c(u,v)$$

The **net flow** across a cut is the sum of net flows leaving S minus the sum of net flows coming in to S:

$$f(S,T) = \sum_{(u,v)\in E: u\in S, v\in T} f(u,v) - \sum_{(x,y)\in E: x\in T, y\in S} f(x,y)$$

By the conservation of flow property, for any cut (S,T) on a network with some flow f, it holds that $f(S,T) = |f|$.

For any cut (S,T) on a network with some flow f, it also holds that $|f| = f(S,T) \leq c(S,T)$. This is intuitive: the net flow across a cut **cannot exceed the capacity** of that cut.

Therefore, the **maximum flow in a network is not greater than the minimum capacity of a cut**:

$$max\{|f| : f \text{ is a flow in } G\} \leq min\{c(S,T) : (S,T) \text{ is a cut in } G\}$$

Theorem One: Max-Flow Min-Cut Theorem

The maximum value of a flow is equal to the minimum value of a cut.

$$max\{|f| : f \text{ is a flow in } G\} = min\{c(S,T) : (S,T) \text{ is a cut in } G\}$$

Theorem Two

For a flow f in G, the following conditions are equivalent *(i.e. all are true, or all are false)*.

(a) f is a maximum flow in G.

(b) There is no augmenting path in the residual network G_f.

(c) There exists a cut (S', T') in G such that $f(S',T') = c(S',T')$.

Theorem two can be shown by proving that $(a) \implies (b) \implies (c) \implies (a)$.

- $(a) \implies (c)$: if f is a maximum flow, there is a cut (S',T') such that $f(S',T') = c(S',T')$.
  - We know that $|f| = f(S',T')$.
  - We know that $f(S',T') = c(S',T')$ from (c).
  - We know that $c(S',T') \geq min\{c(S,T) : (S,T) \text{ is a cut in } G\}$, because it cannot be smaller than the smallest.
  - We know that $c(S',T') \nless min\{c(S,T) : (S,T) \text{ is a cut in } G\}$, because no flow can be greater than the minimum cut.

  - Therefore, $|f| = f(S', T') = c(S', T') = min\{c(S,T) : (S,T) \text{ is a cut in } G\}$.

- $(a) \implies (b)$: if f is a maximum flow, there is no augmenting path in G_f.
  - If there is an augmenting path then f cannot be a maximum flow, because the path could be used to add more flow.

- $(b) \implies (c)$: if there is no augmenting path in G_f, a cut (S', T') exists such that $f(S', T') = c(S', T')$.
  - Consider a set S', constructed so that no augmenting path exists.
  - All edges from S' to T' are full saturated (because no path exists).
  - All edges from T' to S' have zero flow (because no path exists).
  - Therefore, $|f| = f(S', T') = c(S', T')$.

- $(c) \implies (a)$: if a cut (S', T') exists such that $f(S', T') = c(S', T')$, then f is a maximum flow.
  - $f(S', T') = c(S', T') = |f|$ (from the statement).
  - There is no flow greater than $c(S', T')$, so f is a maximum flow.

Ford-Fulkerson Algorithm

The Ford-Fulkerson algorithm **follows the general pattern** above - start with a zero flow and a residual network, find an augmenting path in the residual network, apply the flow from that path, then repeat until no more augmenting paths can be found.

Correctness: the algorithm terminates when there is no augmenting path in the residual network (second cut theorem, condition b), which means that the flow found must be a maximum (second cut theorem, condition a).

The **running time** is $O(m)$ for one iteration: constructing G_f takes $\Theta(m)$ (or (n) incrementally), searching for a path takes $O(m)$, and updating the flow takes $O(n)$.

The **number of iterations** depends on the selection strategy used to find augmenting paths. Assuming all edge capacities are integral:

- Capacities never become non-integral, so the current total flow in G increases by some integer (i.e. by at least 1).
- Therefore, if f^* denotes a maximum flow, the number of iterations is $O(|f^*|)$.
- The **running time of the whole algorithm** is therefore $O(|f*|m)$.

Edmonds-Karp Algorithm

This algorithm is the Ford-Fulkerson algorithm with the following **selection strategy**: in each iteration, select the **shortest augmenting path** (counting by number of edges, not capacities).

The **running time** can be explored as follows:

- Again, the running time is $O(m)$ for one iteration.
- The number of iteration is $O(nm)$:
  - Let q be the number of iterations, and let $k_1, k_2, ..., k_q$ be the augmenting path lengths selected in iterations $1, 2, ..., q$.
  - We know that $1 \leq k_1 \leq k_2 \leq ... \leq k_q \leq n - 1$.
  - A path of the same length appears at most m times in $\langle k_1, k_2, ..., k_q \rangle$.
  - Therefore, $q \leq nm$.
- The overall running time is therefore $O(nm^2)$, **independent** of the maximum flow value.

Flow Feasibility Problems

This variation adds **supply and demand** at various vertices, such that:

- $G = (V, E, c, d)$ is a **flow network** G with vertices V and edges E.
- $c(u, v)$ is the **capacity of the edge** (u, v).
- $d(v)$ is the initial supply/demand at the vertex v.
  - $d(v) > 0$ indicates a supply of $d(v)$ units at v.
  - $d(v) < 0$ indicates a demand for $d(v)$ units at v.
  - $d(v) = 0$ indicates a 'transitional' vertex.
- We assume that supply matches demand, i.e. $\sum_{v \in V} d(v) = 0$.

The objective is to find a flow f that 'moves' all supply to meet all demand. This means that each vertex should have a net flow of zero, i.e. for each vertex v,

$$\sum_{(v,x) \in E} f(v, x) - \sum_{z,v} f(z, v) = d(v)$$

Reduction to Maximum Flow Problem

For a given $G = (V, E, c, d)$ for a flow feasibility problem, $G' = (V', E', c', s, t)$ can be constructed to solve it as a maximum flow problem:

- Create an **artificial source vertex** s with edges **to** all 'supply' vertices v, each with capacity equal to the supply, $d(v)$.
- Create an **artificial sink vertex** t with edges **from** all 'demand' vertices u, each with capacity equal to the negative of the demand, $-d(u)$.

Formally:

- $V' = V + \{s, t\}$
- $E' = E + \{(s, v) : v \in V, d(v) > 0\} + \{(u, t) : u \in V, d(u) < 0\}$
- $c'(u, v) = c(u, v)$
- $c'(s, v) = d(v)$
- $c'(u, t) = -d(u)$

A maximum flow in G' saturates all outgoing edges from s (and therefore saturates all incoming edges to t) if and only if there is a feasible flow in G.

- If a maximum flow f' in G' saturates all edges outgoing from s, then remove the artificial vertices s and t to get a feasible flow for G.
- If f is a feasible flow in G, then saturate all edges outgoing from s and all edges incoming to t to get a maximum flow in G'.

Minimum Cost Flow Problems

Minimum cost flow problems add **costs** to each edge and attempt to find a flow that **satisfies supply/demand** or **maximises flow** (depending on the question type) whilst **minimising cost**.

Notation

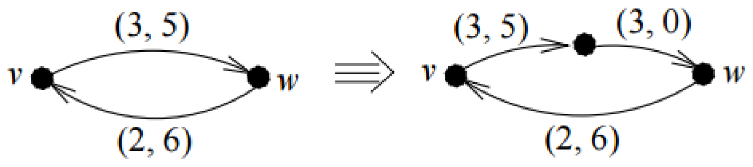
- $G = (V, E, u, c, d)$ is a **flow network** G with vertices V and edges E.
  - $|V| = n$, $|E| = m$.
- $u(u, v)$ is the **capacity of the edge** (u, v).
- $c(u, v)$ is the **cost of the edge** (u, v) per unit of flow.
- $d(v)$ is the initial supply/demand at the vertex v.
  - $d(v) > 0$ indicates a supply of $d(v)$ units at v.
  - $d(v) < 0$ indicates a demand for $d(v)$ units at v.
  - $d(v) = 0$ indicates a 'transitional' vertex.
- Edges are typically drawn with labels in the form $(capacity, cost)$.
- We assume that supply matches demand, i.e. $\sum_{v \in V} d(v) = 0$.

The **cost of a flow** f is the sum of the costs of all used edges:

$$\text{cost of } f = \sum_{(v,w) \in E} c(v, w) f(v, w)$$

Usual capacity constraints, flow conservation constraints and net flow definitions apply.

For convenience, the following **assumption** can be made: if $(w, v) \in E$ then $(v, w) \in E$, but at least one of $u(w, v)$ and $u(v, w)$ is 0. Where this does not hold, one edge can be split as followed without changing the result:



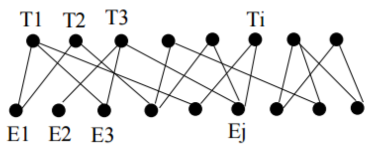
Multi-Commodity Flow Problems

Maximum Bipartite Matching

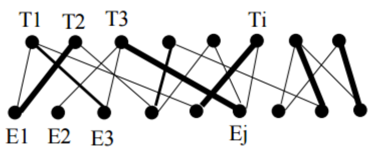
A **bipartite graph** is a graph with nodes that can be divided into two **disjoint sets**, such that all edges have **one end in each set**.

A **bipartite matching** is a subset of edges M such that every node belong to at most one edge. A **maximum bipartite matching** maximises $|M|$.

Bipartite graph G:



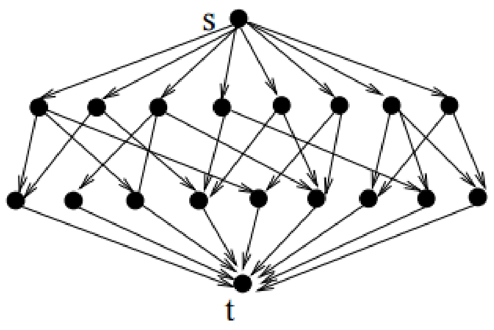
Bipartite matching M:



This approach is useful when considering the **allocation of resources**.

Computing a Maximum Matching

- Label the two disjoint sets of nodes as A and B.
- Make the edges between A and B directed, from A to B.
- Create pseudo-nodes s and t.
- Create edges from s to all nodes in A.
- Create edges from all nodes in B to t.
- Set the capacity of all edges to 1.
- Find the maximum flow from s to t.
  - Edges selected for the flow are part of the matching.
  - The size of the maximum flow is the size of the matching.
  - *See more: Max-Flow Problems, page 21*.



The runtime of such an approach is $O(nm)$, because the maximum value of the flow is $n/2$ and cost per iteration is $O(m)$.

Linear Programming

TODO: Lecture 5.

Optimisations for NP-Hard Problems

TODO: Lectures 6, 7, 8.