OME: Optimisation Methods

Important Notes About These Notes

These notes were written by me, Mark Ormesher, during my revision for Computer Science exams at King's College London. I am publishing them here to help my classmates and other students in the years below me. To the best of my knowledge they are correct.

These notes are **not endorsed** by King's College London, the module lecturers, or any member of College staff. These notes are **not checked** by any qualified individual, and I offer no guarantee as to how accurate or complete these notes are. They are offered for free, as-is, and I accept no responsibility for how you choose to use them.

These notes were relevant for my year (2016/17) but **the content for your course may be different**. Check your lecture slides, syllabi, tutorial guides, etc.

These notes were produced for my own personal use (it's part of how I study and revise). That means that some annotations may by irrelevant to you, and **some topics might be skipped** entirely.

Feel free to **share** these notes, however please only share a link to the repo (see the link below), not individual files.

Notes are originally from **https://github.com/markormesher/CS-Notes**.

Contents

Single-Source Shortest Paths

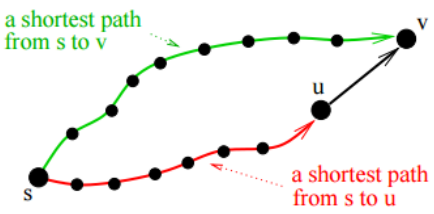
Objective: given a weighted directed graph, find the **minimum-weight paths** from a single vertex s to **all other vertices**. Algorithms that solve this problem can also be adapted to solve **point-to-point** problems.

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.
  - i.e. $\delta(u, v)$ is the minium weight of a path from u to v.
  - 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 $\langle b, c, d \rangle$.
  - This holds because if there was a shorter path from b to d then it would also form part of the part of the shortest path from a to e.
- 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 in this case are not required to compute anything else).

Why not look for the **shortest simple paths** by avoiding cycles? It's a valid question, but it's NP-hard and we do not cover it in this section. *See more: NP-Hard Problems, page 43*.

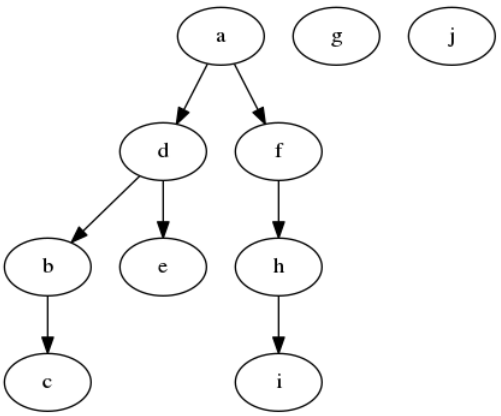
Shortest-Path Trees

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 in which 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 parent of v.

Initialisation

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

```
1 fun INITIALISE(G, s):
2     d[s] = 0
3     parent[s] = nil
4     for each vertex v in V − {s}, do:
5         d[v] = inf
6         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**. This is especially effective if not all vertices are reachable from s.

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 5*).

```
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-source 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

    return TRUE // no negative cycle, d[] now holds shortest path weights
```

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.

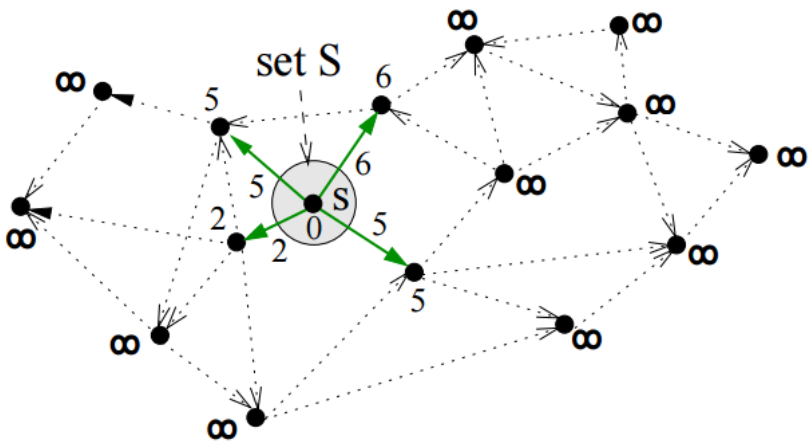
1. For each $v \in S$:
   (a) v is in the current tree.
   (b) $d[v]$ is the shortest-path weight from s to v, i.e. $d[v] = \delta(s, v)$.
   (c) The path from s to v in the current tree is a shortest path from s to v.
2. For each $z \notin S$ such that there **is an edge** (v, z) for some vertex $v \in S$:
   (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 tree is the shortest path tree 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

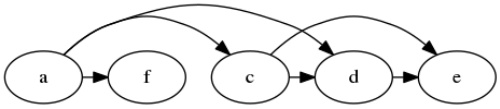
- We have selected vertex u to add to S, so we must prove that at the end of the iteration $d[u] = \delta(s, u)$.
- We already know that $d[u] \geq \delta(s, u)$ because it is a basic property of the relaxation operation, so we must prove $d[u] \leq \delta(s, u)$.
- Select any path R from s to u and 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 below (where all edges point from left to right).



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 17*), 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

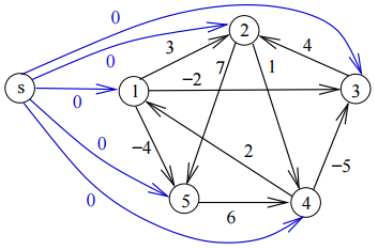
We must change the weight of edges to **remove negatives without changing shortest paths**. Adding some large value to each edge won't work because the shortest paths may change (because the path length becomes a major component in its weight).

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** $(u, v) \in E$.
- $s \in E$ 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 underlying 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.
- **Min-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 **simultaneous flow of multiple commodities** that satisfies some specified **global objective**.

Flow Formalisation

We assume that if $(u, v) \in E$ then $(v, u) \in E$. Edges with zero capacity are added when required, but are 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

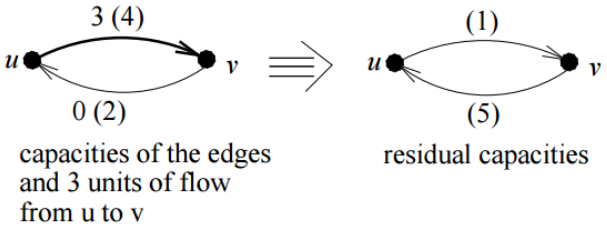
Maximum flow problems are solved by continually adding flow to a network. The amount that can be added (i.e. the network's remaining capacity) is the **residual capacity**.

Residual Capacity

If f is a flow defined in the network $G = (V, E, c, s, t)$, then residual capacities 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 $f(u, v) = 3$. The residual capacities $c_f(u, v) = 1$ 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 $c(S',T') \not< min\{c(S,T) : (S,T) \text{ is a cut in } G\}$, 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** $(u, v) \in E$.
- $d(v)$ 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'.

Min-Cost Flow Problems

Min-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

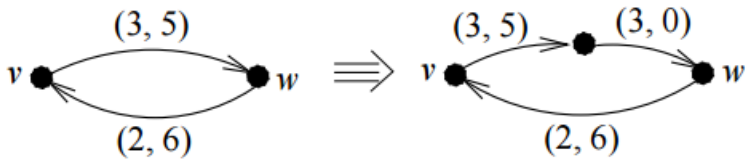
- $G = (V, E, u, c, d)$ is a **flow network** G with vertices V and edges E.
  - $|V| = n$, $|E| = m$.
- $u(v, w)$ is the **capacity of the edge** $(v, w) \in E$.
- $c(v, w)$ is the **cost of the edge** $(v, w) \in E$ **per unit of flow**.
- $d(v)$ is the initial supply/demand at the vertex v.
  - $d(v) > 0$ 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 total cost 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:

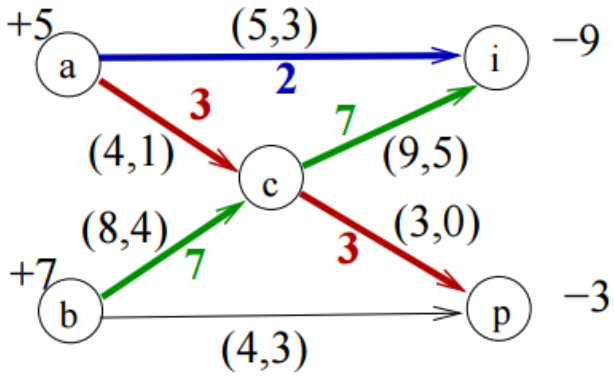


Sending Flow Along Cheapest Paths

A naïve approach is to find the cheapest paths between supply and demand. Formally:

- Take a node v with remaining supply.
- Find a cheapest path to a node w with remaining demand.
- Send as much flow as possible from v to w along this path.

For example:

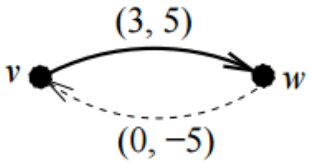


The paths identified are $\langle a, c, p \rangle$, $\langle a, i \rangle$ and $\langle b, c, i \rangle$

The cost of this flow is $(2 * 3) + (3 * 1) + (3 * 0) + (7 * 5) + (7 * 4) = 72$. This is **not optimal** - sometimes we need to **re-route** flow to get the optimal cost.

Residual Network

- Edges exist in both directions between vertices, but $u(v, w) = 0$ and/or $u(w, v) = 0$.
- If $u(v, w) > 0$ then $c(v, w) \geq 0$ and $u(w, v) = 0$.
- $c(w, v) = -c(v, w)$



Let f be a flow in a network $G = (V, E, u, c, d)$.

The **residual network** of G with respect to f is $G_f = (V, E_f, u_f, c, d_f)$, where:

- **Capacity**: for each edge (v, w) where $u(v, w) > 0$ and $u(w, v) = 0$:
  - $u_f(v, w) = u(v, w) - f(v, w)$ *(i.e. the remaining flow)*
  - $u_f(w, v) = f(v, w)$ *(i.e. the flow that can be reversed)*

- **Supply/Demand**: for each vertex $v \in V$:
  - $d_f(v) = d(v) - \sum_{(v,x)\in E} f(v,w) + \sum_{(z,v)\in E} f(z,v)$
  - *(i.e. the original supply/demand, minus flow going out, plus flow going in)*
- **Edges**: E_f is the set of edges with positive, non-zero residual capacities.

Successive Shortest Path Algorithm

The correct, optimal solution to this style of problem is to send flow along the **cheapest paths in the residual network** that is created after each action, rather than using just the original network (as in the earlier approach).

```
fun SUCCESSIVE_SP(G)
    f = zero flow

    // initial residual network and nodes with supply
    G_f = residual network of G
    S = v in V, where d_f(v) > 0

    while (S is not empty):
        G_f = residual network of G

        // find a vertex with supply
        v = select any vertex from S

        // find a vertex with demand
        T = shortest—paths tree in G_f from v
        if (T contains a vertex w with d_f(w) < 0):

            P = path in T from v to w

            // find the path's flow bottleneck
            q = MIN( d_f(v), —d_f(w), min edge—capacity in P )

            // compute new flow and combine it with existing flow
            f_P = q units of flow along path P
            f = f + f_P

            // remove v from S if all residual supply was used
            if (d_f(v) = 0):
                S.remove(v)

        else:
            // remove v from S if no supply can be sent
            S.remove(v)
```

Correctness and Runtime

- At the end of each iteration, f has the minimum cost of all flows which satisfy the same supply/demand quantities satisfied by f. This can be proved by induction, but such a proof is not covered here.
- The cost of residual edges can be negative, but there can be no negative cycles.
- **Dijkstra's algorithm** can be used to compute the shortest paths, even with negative residual edges: appropriate $h(v)$ values can be maintained in each iteration to allow the re-weighting system from Johnson's algorithm to create non-negative edges.
- The running time of **each iteration** is bounded by Dijkstra's algorithm, $O(min\{m \cdot log_2(n), n^2\})$.
- If all input edge capacities, supplies and demands are integral, the **number of iterations** is at most $\sum\{d(v) : v \in V, d(v) > 0\}$ *(i.e. the sum of supply)*.
- The number of iterations **can be exponential** in the number of nodes, but more complex algorithms exist which run in **polynomial** time. The best known is $O(m^2 \cdot log_2(n)^2)$.

Multi-Commodity Flow Problems

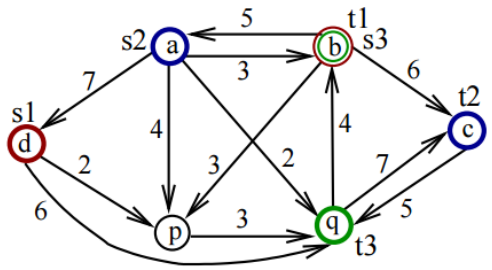
This variation adds multiple commodities to the flow network. The objective is to find a **simultaneous flow of all commodities** that satisfies some specified **global objective**.

Notation

- $G = (V, E, u)$ is a **flow network** G with vertices V and edges E.
  - $|V| = n$, $|E| = m$.
  - $u(v, w)$ is the **capacity of the edge** $(v, w) \in E$.
- $C = \langle (s_1, t_1, d_1), ..., (s_k, t_k, d_k) \rangle$ specifies k commodities where, for $q = 1, 2, ...k$:
  - $s_q \in V$ and $t_q \in V$ are the source and target vertices of commodity q.
  - d_q (> 0) is the demand of this commodity.

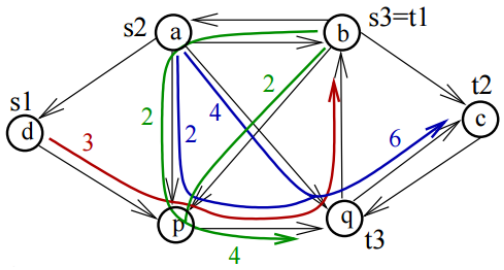
For some multi-commodity flow problems, the flows selected do not need to satisfy capacity constraints.

Example



Commodity (k)	Source (s_k)	Target (t_k)	Demand (d_k)
1	d	b	3
2	a	c	6
3	b	q	4

The following is one flow that satisfies these demands. Note that it does not satisfy the capacity constraints given on the previous diagram.



Flows of Commodities

- A simultaneous flow of the specified commodities consists of flows of individual commodities $\langle f_1, ..., f_k \rangle$.
- A flow f_q of commodity q is a function $f_q : E \mapsto \mathbb{R}$ where $f_q(x, y)$ is the flow of commodity q along the edge $(x, y) \in E$.
  - f_q must satisfy **flow conservation constraints**.
  - **Capacity constraints** may not be present.
- Flow conservation for commodity q:
  - The net flow of q outgoing from the source s_q is d_q.
  - The net flow of q incoming to the target t_q is d_q.
  - The net flow outgoing from all other vertices $v \in V - \{s_q, t_q\}$ is 0.

Types of Multi-Commodity Flow Problems

Feasibility

Total flow is defined as $f(v, w) = \sum_{q=1}^{k} f_q(v, w)$.

The objective is to design a simultaneous flow $f = \langle f_1, ..., f_k \rangle$ of all commodities that satisfies **capacity constraints**.

Formally, for each edge $(v, w) \in E$, the total flow $f(v, w) \leq u(v, w)$.

Min-Cost

Edge costs are introduced to the flow network, such that $c(v, w)$ is the cost of sending one unit of flow along the edge $(v, w) \in E$. The total cost of a flow $f = \langle f_1, ..., f_k \rangle$ is defined as the sum of the cost of all flows along all edges.

Total cost: $\sum_{(v,w)\in E} c(v, w) \cdot [f_1(v, w) + ... + f_k(v, w)]$.

The objective is to design a simultaneous flow that satisfies **capacity constraints** and **minimises total cost**.

Min-Congestion

The congestion of a flow on an edge $(v, w) \in E$ is defined as $f(v, w)/u(v, w)$ *(i.e. total flow divided by capacity)*.

The objective is to design a simultaneous flow of all commodities that **minimises the maximum congestion**.

Note: the optimal (minimum) congestion may be greater than 100%.

Computational Approaches

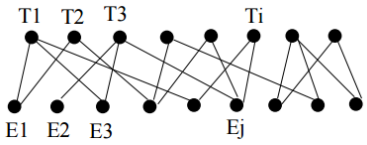
- **Specialised algorithms**: start with some initial simultaneous flow, and iteratively keep improving it by re-routing some of the flow of each commodity via better paths.
- **Mathematical programming**: express the problem as a linear programming problem (or other suitable model) and use appropriate general solving algorithms.

Maximum Bipartite Matching

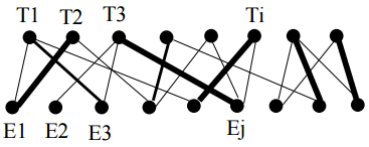
A **bipartite graph** is a graph with nodes that can be divided into two **disjoint sets**, such that all edges have **one end in each set**.

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

Bipartite graph G:



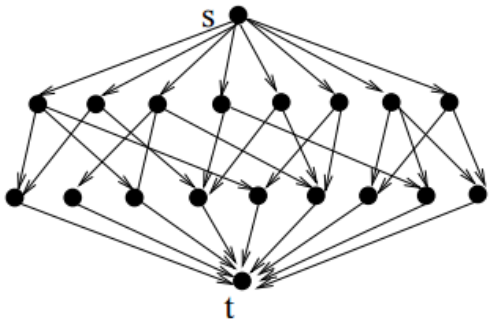
Bipartite matching M:



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 22*.



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

Linear programming (LP) can be used to express and solve a wide range of problems, including network flow problems.

General Form

The general form of an LP is as follows:

- n **decision variables**, $x_1, x_2, ..., x_n \in \mathbb{R}$.
- m **constraints**.
  - These must be linear equations containing decision variables and (optionally) constant values.
  - Equalities, inequalities and bounds are all acceptable.
- An optional **objective** to minimise or maximise the value of some linear equation containing decision variables and (optionally) constant values.

The **input** to such a problem is the set of values for constants, bounds, constraints, etc. (i.e. all numbers apart from the decision variables). The **output** from such a problem is a value assignment for each decision variable that satisfies the objective, or a report that no assignment exists.

The objective may not be specified - if this is the case, the goal is to find **any feasible assignment** without optimising towards a given objective.

Solution Types

Several **commercial** (CPLEX, MINOS, GIPALS) and **open-access** (GLPK, LP_SOLVE) packages exist for solving LPs, but they are not covered within the scope of this course (check the FC2 notes if you really want to).

Modelling Max-Flow Problems

- Create decision variables $f_{x,y}$ to represent the flow on each edge for all $(x, y) \in E$.
- Create constraints for edge capacity and flow conservation:
  - Non-negative flow: $0 \leq f_{x,y}$.
  - Edge capacity: $f_{x,y} \leq u(x, y)$.
  - Flow conservation: $\sum f_{x,v} - \sum f_{v,y} = 0$ for all $v \in V$.
    i.e. the net flow of each vertex is zero.
- Create the objective to maximise $\sum_{(x,y)\in E} f_{x,y}$.

Modelling Min-Cost Problems

- Create decision variables $f_{x,y}$ to represent the flow on each edge for all $(x, y) \in E$.
- Use constants $c_{(x,y)}$ to represent the cost per unit of flow along each edge.
- Create constraints for edge capacity and flow conservation:
  - Non-negative flow: $0 \leq f_{x,y}$.
  - Edge capacity: $f_{x,y} \leq u(x, y)$.
  - Flow conservation: $\sum f_{x,v} - \sum f_{v,y} = d(v)$ for all $v \in V$.
    $d(v)$ is positive if $v = s$ *(source)*
    $d(v)$ is negative if $v = t$ *(target)*
    $d(v) = 0$ otherwise *(transitional)*
- Create the objective to minimise $\sum_{(x,y)\in E} c_{(x,y)} \cdot f_{x,y}$.

Modelling Multi-Commodity Problems

- Create decision variables $f^{(i)}_{x,y}$ to represent the flow along each commodity i on each edge (x, y).
- Create constraints for edge capacity and flow conservation:
  - Non-negative flow: $0 \leq f_{x,y}$.
  - Edge capacity **if they are being observed**: $f_{x,y} \leq u(x, y)$.
  - Flow conservation: $\sum f^{(i)}_{x,v} - \sum f^{(i)}_{v,y} = d^{(i)}(v)$ for all $v \in V$ and $0 \leq i < k$.
  - Note: $d^{(i)}(v)$ is the net flow of the commodity i at vertex v.
    $d^{(i)}(v) = +d_i$ if $v = s_i$ *(source)*
    $d^{(i)}(v) = -d_i$ if $v = t_i$ *(target)*
    $d^{(i)}(v) = 0$ otherwise *(transitional)*
- Leave the objective blank to find any feasible flow, or create one for min-cost using cost constants.

Modelling Min-Congestion Multi-Commodity Problems

- Create decision variables $f^{(i)}_{x,y}$ to represent the flow along each commodity i on each edge (x, y).
- Create constraints for edge capacity and flow conservation:
  - Non-negative flow: $0 \leq f_{x,y}$.
  - **Ignore** edge capacity constraints.
  - Flow conservation: $\sum f^{(i)}_{x,v} - \sum f^{(i)}_{v,y} = d^{(i)}(v)$ for all $v \in V$ and $0 \leq i < k$.
  - Note: $d^{(i)}(v)$ is the net flow of the commodity i at vertex v.
    $d^{(i)}(v) = +d_i$ if $v = s_i$ *(source)*
    $d^{(i)}(v) = -d_i$ if $v = t_i$ *(target)*
    $d^{(i)}(v) = 0$ otherwise *(transitional)*

The objective is to minimise the maximum edge congestion, but there's a problem here: that **isn't a linear equation**, and therefore cannot be part of an LP.

The solution is to introduce a new decision variable, λ, that is an **upper-bound** on edge congestion. This is achieved by creating new constraints in the following form:

$$\frac{\sum_{i=1}^{k} f_{x,y}^{(i)}}{u(x,y)} \leq \lambda \text{ for each } (x,y) \in E$$

This states that the sum of all commodity flows along an edge divide by that edge's capacity (i.e. it's congestion) must be smaller than or equal to λ for each edge, therefore making λ an upper-bound.

This should be re-written as:

$$\sum_{i=1}^{k} f_{x,y}^{(i)} - \lambda \cdot u(x,y) \leq 0 \text{ for each } (x,y) \in E$$

Each of these constraints is linear, and therefore the problem can be solved by setting the objective to simply minimise λ.

Adding Constraints

LPs are especially useful for network flow problems when **non-standard** constraints are introduced. For example, the following constraints cannot be handled by the normal successive shortest path algorithms, but can be easily handled by LPs.

'Both edges out of b must have equal flow.'

Assuming b has edges to i and j, this can be achieved by creating one extra constraint:

$$f_{b,i} - f_{b,j} = 0$$

'The flow on edge (a,b) must not exceed the flow on edge (c,d).'

$$f_{a,b} - f_{c,d} \leq 0$$

'There is a budget of 50 for all flow out of y.'

$$\sum_{(x,y) \in E} f_{x,y} \cdot c_{x,y} \leq 50$$

'Flow along (a, b) leaks by 5%.'

We update flow conservation for vertex b, because it will receive less flow than was sent:

$$... + 0.95 \cdot f_{a,b} + ...$$

Adding Constraints with Binary/Integer Variables

Binary/integer variables allow us to encode yes/no decision into an LP. However, they move away from 'pure' linear programming and into integer or mixed-integer programming, which is more complex:

- Linear programming (LP): polynomial solutions are known.
- Mixed-integer programming (MIP): complexity grows with the number of integers, making this an NP-hard problem.
  - A small number of integers is ok-ish.
  - Large, commercial packages can handle up to a few dozen integers.

'Flow along (a, b) is either 0 or fully saturating.'

We add the variable $x \in \{0, 1\}$ to denote 'edge (a, b) is used', and add the constraint

$$f_{a,b} - x \cdot u_{a,b} = 0$$

Either the edge is used ($x = 1$) and the flow equals the capacity, or the edge is unused ($x = 0$) and the flow equals zero.

'Commodity 2 can only use one edge outgoing from i.'

We add the variable $x_j \in \{0, 1\}$ for each edge (i, j), and the following constraints:

$$\sum_{(i,j) \in E} x_j \leq 1$$

$$f^{(2)}_{i,j} \leq x_j \cdot u(i, j) \text{ for each } (i, j) \in E$$

These state that the sum of all 'use this edge' variables can be at most one (so only zero or one edges can be used) and that the capacity limit for each edge outgoing from i is the given capacity multiplied by its 'use this edge' binary variable (meaning either all of them or all but one of them will be zero).

'There is a fixed cost of 5 for using any edge.'

We add the variable $x_{a,b} \in \{0, 1\}$ for each edge (a, b), and the following constraints:

$$f_{a,b} \leq x_{a,b} \cdot u(a, b) \text{ for each } (a, b) \in E$$

This creates a 'this edge is used' binary variable for each edge, using them to effectively turn capacity on and off for each edge, as appropriate. The objective function must then be modified to add 5 for each used edge:

$$\text{minimise } ... + \sum_{(a,b) \in E} 5 \cdot x_{a,b}$$

Multiple Objectives

Sometimes we want to consider multiple objectives (such as 'of all maximising flows, find the one with the minimum cost'). There are several ways to approach this:

Strict Priorities (Nested Solutions)

In this case, we need to find all solutions that meet one objective, then solution(s) within that set that meet a second. For example: 'of all maximising flows, find the one with the minimum cost'. This can be achieved as follows:

- Solve the LP for the first objective function, $h_1()$, to find its optimal value, h_1^*.
- Add the constraint $h_1(...) = h_1^*$.
- Solve the LP again for the second objective function, $h_2()$.
- This process can be repeated for more layers of strictly prioritised objectives.

Equally Important Priorities

When two or more priorities are equally important, we can optimise for the **sum** of the objective functions.

$$h(...) = h_1(...) + h_2(...) + ...$$

Biased Priorities

When two of more priorities have different levels of importance, we can optimise for the **weighted sum** of the objective functions.

$$h(...) = 1 \cdot h_1(...) + 0.5 \cdot h_2(...)$$

This states the the first objective function is twice as important as the second.

Convert Objectives into Constraints

For some problems it may be possible to keep one 'main' objective and make the others into constraints. For example, consider the following two statements:

- 'Minimise the travel time and minimise the travel distance, where the first objective is twice as important as the second.'
- 'Minimise the travel time, and constrain the travel distance to a maximum of 200 miles.'

NP-Hard Problems

Combinatorial Optimisation Problems

Many problems can be modelled as combinatorial optimisation problems, including almost everything in this module. Instances of these problems are given in the form (R, C), where:

- R: a finite set of **combinatorial configurations** (specificed in some implicit way, not as a list of all combinations).
- C: a **cost function**, such that $C : R \mapsto \mathbb{R}$, that assigns real-number costs to any given configuration (specified as a procedure or algorithm).

The problem is to find a configuration $r \in R$, such that $C(r)$ is as small as possible.

Example: Single-Source Single-Destination Shortest Simple Path Problem

- A single configuration, r: a simple path in the graph from s to t.
- R: all possible simple paths, specificed as the graph G and vertices s and t.
- $C(r)$: the sum of the cost of all edges in r.

Example: Multi-Commodity Unsplittable Minimum-Congestion Flow Problem

- A single configuration, r: a sequence of k paths, $P_1, ..., P_k$ where P_q is a simple path for sending commodity q from s_q to t_q.
- R: all possible sequences of simple paths, specificed as the graph G and vertices s_q and t_q for each $q \in [0..k-1]$.
- $C(r)$: the maximum congestion for any edge used in any path in r.

Example: Travelling Salesman Problem

- A single configuration, r: a Hamiltonian tour of all cities, $1..n$.
- R: all possible Hamiltonian tour of all cities, $1..n$.
- $C(r)$: the cost of all edges used in r.

Example: Weighted Graph-Bisection Problem

Sketch: given an undirected graph with weighted edges and an even number of vertices, partition the vertices into two sets of equal size such that the total weight of edges between sets is minimised.

Applications: circuit design, where edge weight expresses the preference for keeping components close together.

- A single configuration, r: a partitioning of vertices into two equal-sized sets.
- R: all possible partitionings.
- $C(r)$: the cost of all edges passing from one set to the other.

Problem Classes

- A **polynomial-time** algorithm has a worst-case running time of $O(n^k)$ for an input of size n and some constant k.
  - Examples: $O(n)$, $O(n^2)$, $O(log_2 n)$, $O(nm)$
- An **exponential-time** algorithm has worst-case running times of $O(2^{dn})$ and $\Omega(2^{cn})$ for an input of size n and some constants $0 < c \leq d$.
  - Examples: $O(2^n)$, $\Theta(n2^n)$
  - Compare $\Theta(n^2)$ and $\Theta(2^n)$:
    - For $n = 100$...
    - $\Theta(n^2) = \Theta(10^4)$ operations (a few seconds of computation)
    - $\Theta(2^n) \approx \Theta(10^{30})$ operations (around 40 trillion years of computation)

P Problems

Algorithms exist with which these problems can be solved in **polynomial-time**. These problems are considered to be *computationally easy*, or *tractable*.

For **example**: the single-source, single-destination shortest simple path problem with non-negative edge weights can be solved with Dijkstra's single-source shortest-paths algorithm, which runs in $O(m \cdot log_2(n))$ with a heap-based implementation.

For **example**: the minimum-cost flow problem can be solved by an algorithm with a running time of $O(m^2 \cdot (log_2(n))^2)$.

NP Problems

A combinatorial optimisation problem is an NP problem if a **verification algorithm** exists that can check in polynomial-time (respective to $|I|$) whether a given configuration X for a problem instance I is valid and not greater than some cost bound, K.

For **example**: the single-source single-destination shortest simple path problem can be verified in polynomial time by checking that the path is a valid path from s to t, and that it's cost is $\leq K$.

For **example**: the travelling salesman problem can be verified in polynomial time by checking that the path is a valid Hamiltonian cycle, and that it's cost is $\leq K$.

Note that the existence of a polynomial-time verification algorithm does not imply the existence of a polynomial-time solution algorithm. There are NP problems for which it is currently believed that no such solution algorithm is possible.

NP-Hard Problems

A problem Q is NP-hard if the existence of a polynomial-time solution algorithm for Q implies that polynomial-time solutions exist for **all** NP problems.

NP-Complete Problems

A problem is NP-complete if is **both NP and NP-hard**.

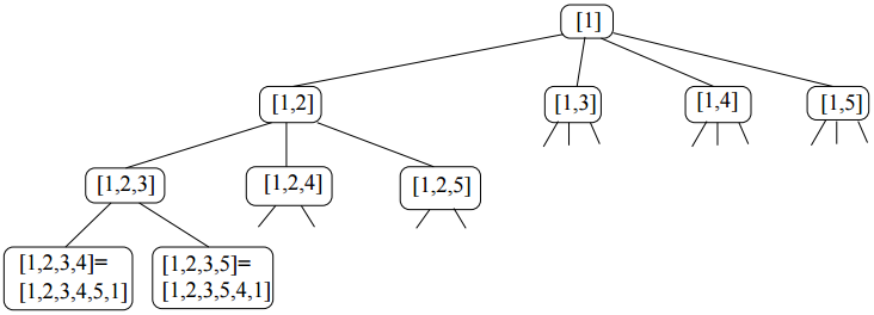
Solution Apporaches

- **Exhaustive search**: try every combination. This will find optimal solutions, but is usually prohibitively slow.
- **Branch-and-bound**: the same as exhaustive search, but uses techniques to avoid trying combinations that will not be valid or optimal. This will also find optimal solutions, but is just as slow as exhaustive search in the worst case.
  - *See more: Branch-and-Bound, page 46*.
- **Approximate algorithms**: compute some 'good' solution that is not necessarily optimal, but has some guarantee on 'goodness' (i.e. the difference between the computed solution and the optimal solution). These are fast - usually polynomial.
- **Heuristics**: compute some 'good' solution, but with no guarantee on how good it is. These are fast.
- **Meta-heuristics**: the same as heuristics, but using domain-agnostic heuristics that can be applied to many different problems.
  - *See more: Simulated Annealing, page 48*.
  - *See more: Genetic Algorithms, page 52*.

Branch-and-Bound

Exhaustive Search

Many approaches use a heirarchical partitioning of possible configuartions, creating a **search tree** with **complete configurations at the leaves** and **partial configurations** as internal nodes. For example, the diagram below shows the beginning of the search space for the travelling salesman problem:



The number of leaves in the search tree for an n-city problem is $(n-1)!$.

Partial Configurations and Branding Procedure

A partial configuration is a **not-fully-specificed** configuration that **can be extended** to a number of possibly-complete configurations.

Executing the branching part of a branch-and-bound approach requires two things:

- The **initial partial configuration**: this is a 'root' configuration that can be extended to any other configuration (often, it is a 'blank' configuration).
- A **branching procedure**: this takes a partial configuration P and produces a number of possibly-partial configurations $P_1, P_2, ..., P_k$.
  - This procedure should not 'skip' or 'lose' any possible configurations.
  - Each configuration (excluding the root) should be the 'child' of exactly one 'parent' configuration (this maintains the tree property).

Bounding Procedure

For any given partial configuration P, the bounding procedure produces a **lower-bound** on the cost of any configuration that could be obtained by extending P. This lower-bound can be used to limit the search:

- If the lower-bound cost of configurations extending from P is k and another solution with a cost $\leq k$ has already been found, there is no need to continue trying extensions of P.

- In some problem types, the bounding procedure may return ∞ to indicate that no valid extensions can be produced from P, meaning that no further extensions of P should be tried.

General Description

The general approach is very much the same as straightforward greedy best-first search.

```
fun BRANCH_AND_BOUND():
    bestConfig = null
    minCostSeen = inf

    Q = priority queue for pairs (partial config, lower—bound)

    P = initial partial config
    lowerBound = BOUND(P)
    Q.insert(P, lowerBound)

    while (Q is not empty) do:
        P, lowerBound = Q.removeMin()

        // prune this branch if it can't give a cheaper solution
        if (lowerBound >= minCostSeen):
            break loop

        P1...Pk = BRANCH(P)
        for each Pi in P1...Pk, do:
            // update our best if this beats it
            if (Pi is a complete config):
                if (COST(Pi) < minCostSeen):
                    minCostSeen = COST(Pi)
                    bestConfig = Pi
            else:
                // add this to the queue if it might give a cheaper solution
                lowerBound = BOUND(Pi)
                if (lowerBound < minCostSeen):
                    Q.insert(Pi, lowerBound)

    return minCostSeen and bestConfig
```

Simulated Annealing

Simulated annealing is a **meta-heuristic** or **general heuristic** for combinatorial optimisation problems, which means it can be applied to any instance instance of a problem.

The approach is similar to greedy best-first search and relies on a neighbourhood function.

Neighbourhoods

For any given configuration $c \in R$, the neighbourhood of this configuration is a well-defined set of similar configurations that can be obtained by a simple modification: $N(c) = \{c_1, c_2, ..., c_k\}$.

For a given problem, neighbourhood configurations can be defined in multiple ways. A **generation mechanism** for a given neighbourhood structure N is a procedure for which a given configuration $c \in R$ returns **one random configuration** from $N(c)$.

More complex neighbourhood structures will require more complex generation procedures.

Examples for Symmetric Travelling Salesman Problem

- $N_1(c)$: the set of tours other than c which can be obtained from c by swapping the order of two consecutive cities.
  - The size of this neighbourhood is $\Theta(|c|)$.
  - This neighbourhood is not very useful.
- $N_2(c)$: the set of tours that can be obtained by replacing two non-consecutive edges in c with two new edges (the **2-change** modification).
  - The size of this neighbourhood is $\Theta(|c|^2)$
- $N_3(c)$: the set of tours that can be obtained by replacing three non-consecutive edges in c with three new edges (the **3-change** modification).
  - The size of this neighbourhood is $\Theta(|c|^3)$

Example for Weighted Graph Bisection Problem

- $N_1(c)$: the set of partitions created by swapping a node from L with a node from R.
  - The size of this neighbourhood is $\Theta((\frac{n}{2})^2)$.

Side Note: Local Search

Local search is a **non-backtracking iterative improvement heuristic** that maintains a current configuration and only progresses along a path of strictly-cheaper neighbour configurations.

The basic algorithm is as follows:

```
fun KEEP_PICKING_BETTER_CONFIG():
    config = initial config

    while (config has more neighbours), do:
        newConfig = GENERATE(config)
        if (COST(newConfig) < COST(config)):
            config = newConfig

    if (config is complete):
        return config
    else:
        return FAIL
```

Local Optima

Such a search may terminate at a **local optimum**: a situation where all neighbours of a configuration are more expensive, but either a solution or a **global optimum** has not yet been reached.

Key Idea of Simulated Annealing

The heart of the simulated annealing algorithm is to **give worse neighbourhood functions a chance, but decrease that chance as computation progresses**.

Algorithm

The algorithm works in **phases**, each of which has a **temperature** (control parameter) and **length** (explained below).

```
fun SIMULATED_ANNEALING():
    k = 0 // phase counter

    // Ck: the control (temperature) in phase k
    // Lk: the length of phase k

    // implementation— and domain—specific parameters
    config = initialConfig
    C0 = initialControl
    L0 = initialLength

    repeat:
        // we are now in phase k

        for len from 1 to Lk, do:
            newConfig = GENERATE(config)

            // always take better configs
            if (COST(newConfig) <= COST(CONFIG)):
                config = newConfig

            // sometimes take worse configs
            else if (ODDS(Ck, config, newConfig) > random(0, 1)):
                config = newConfig

        k = k + 1
        UPDATE_LENGTH(Lk)
        UPDATE_CONTROL(Ck)

    until stopping criteria is met
```

Odds of Taking a Worse Configuration

Odds are calcualted in the secenario when $COST(newConfig) > COST(config)$. Considering the cost of the new configuration as x, the cost of the current configuration as a, and the temperature C_k as c, the odds are defined as:

$$ODDS(...) = exp(\frac{COST(config) - COST(newConfig)}{C_k}) = e^{(a-x)/c} = P_c(x)$$

$a - x$ is guaranteed to be negative, and therefore $(a - x)/c$ will also be negative (because c is positive). Raising to the power of a negative number will give a number in the range $[0, 1)$ (i.e. including 0, but exlcuding 1).

The worse configuration is accepted if $P_c(x) < r$, where r is the random number selected from $[0, 1)$. The probability of accepting a worse configuration is therefore $P_c(x)$.

Control and Length Parameters

- The control parameter (a.k.a. temperature) C_k determines how likely it is that a worse configuration will be chosen. A higher value makes selecting a worse configuration more likely.
  - This starts relatively high and decreases as the computation continues.
  - The value of C_0 should be linked to the initial configuration.
  - One good idea is to build C_0 based on the worst-case increase in a configuration's cost.
  - Generally, $C_{k+1} = \alpha C_k$ where $\alpha < 1$ (typically $0.8 \leq \alpha \leq 0.99$).
- The length parameter L_k determines how many iterations should be done in each phase.
  - This value usually increases as computation continues.
  - It should be relative to the size of a the neighbourhood of a configuration.
  - The initial value is often computed by a fast, specialised heuristic.

Stopping Criteria

The algorithm has no specific 'stop here' point, because the stopping critera can depend on the problem, the objective, the resources available, etc. This way, the algorithm can easily handle 'I need a rough answer in 60 seconds' as well as 'We need the best answer you can find with two weeks of computing time'.

Typical criteria include stopping when C_k is below a certain level, when a given number of rounds has not produced an improvement, or when a specific execution time limit is reached.

Running Time

The running time of the whole algorithm depends on the stopping critera, as mentioned above.

The running time of one iteration depends on the generation function, cost functions, and storage of current configuration.

Genetic Algorithms

Genetic algorithms are another type of **meta-heuristic**, like simulated annealing. As the name suggests they are inspired by **genetic evolution** and use the same terminoligy:

- An **individual** is a given configuration.
- The **fitness function** is analogous to the cost function.
  - Note: higher fitness is better, whereas lower cost was better for simulated annealing.
- The algorithm maintains a **population** of **individuals** (a collection of configurations).
  - Note: simulated annealing maintains only one configuration.

Crossover Operation

A key function within genetic algorithms is the crossover operation: it takes **two individuals** as an input and produces **one or more new individuals** called **offspring** or **children**.

The crossover function is often the hardest part of a genetic algorithm to produce, because all outputs must be valid configurations and some problem classes do not easily fit into this model.

1-, 2- and n-Point Crossovers

When an individual can be represented as a binary string, an easy crossover function exists:

- **1-point**: generate a random index i within the length of the strings, then combine $parent_1[0..i]$ with $parent_2[i+1..n]$ to create the offspring.
  - You could also create two offspring by combining the leftover chunks from each parent.
- **2-point**: generate two random indexes, then combine the first and third chunk of one parent with the second chunk of another.
- **n-point**: generate $n-1$ random indexes and combine alternating chunks.

Not all problems lend themselves to this method, because randomly spliced strings may not yield valid offspring.

Mutations

A small number of new individuals also come from mutations, which are **small, random changes** made to exisiting individuals. This is the only method of creating new individuals in simulated annealing.

Fitness Function

This measures how 'good' an individual is, such that **higher is better**. A common way to define the fitness function of an individual h in a problem P is as follows:

$$fitness(h) = C_{max}(P) - C(h) + r$$

where C is the cost function, $C_{max}(P)$ is the highest cost, and r is some non-negative constant.

r is used so that the highest-cost individual still has a non-zero configuration. This is used because some versions of the algorithm use a fitness score of 0 to signal 'this configuration is worthless and should not be used', whereas we want to signal 'this configuration is worse than all the others, but might still be useful'.

The Algorithm

The algorithm itself has three parameters that can be used to tune execution:

- k is the population size in each generation (usually around 100).
- r is the fraction of the population that should be replaced by crossover (usually around 0.75).
  - $1 - r$ is therefore the fraction from one generation that will survive into the next. This is called **survival of the fittest**.
- q is the mutation rate (usually around 0.05).

```
fun GENETIC_ALGO(k, r, q):

    // initial popuation
    p = k random individuals

    repeat:
        nextGen = empty set

        // in this code we use roulette selection, but other
        // selection methods exist

        // calculate each individual's roulette selection probability
        fitnessSum = sum of FITNESS(h) for all h in p
        for each h in p, do:
            prob[h] = FITNESS(h) / fitnessSum

        // crossover step
        select (r * k) pairs from p using roulette selection
        for each pair (p1, p2) in selection, do:
            nextGen.add(CROSSOVER(p1, p2))

        // survival of the fittest
        // optional: always allow the fittest individual to survive
        select ((1 — r) * k) individuals from p using roulette selection
        for each h in selection, do:
            nextGen.add(h)

        // nextGen.size = k at this point

        // mutations
        select (q * k) individuals from nextGen, uniformly at random
        for each h in selection, do:
            nextGen.replace(h, MUTATE(h))

        p = nextGen

    until stopping criteria is met

    return fittest individual from p
```

Selection of Individuals

In the code above, individuals are selected according to $prob[]$, which is an example of roulette selection. This is just one of a few possible selection methods:

- **Routlette selection**: individuals are selected with probability proportional to their fitness score divided by the sum of fitness scores for all individuals.

  - For example, if three individuals have fitnesses of 4, 8 and 3, their respective probabilities will be $\frac{4}{15}$, $\frac{8}{15}$ and $\frac{3}{15}$.
  - More details can be found in the AIP notes from last term.

- **Rank selection**: this uses a similar mechanism to roulette selection, but instead of using an individuals fitness divided by the sum of fitnesses, it uses the individual's rank divided by the sum of all ranks.
  - For example, if the same three individuals have fitnesses of 4, 8 and 3, their respective probabilities will be $\frac{2}{6}$, $\frac{3}{6}$ and $\frac{1}{6}$.

- **Tournament selection**: a small number q of individuals are selected at random, then the fittest of these individuals is selected.
  - q is typically around 3 or 4.
  - Each individual's probability is determined by the number of groups it could appear in and the number of those groups it would 'win'.

Stopping Criteria

This algorithm also has no specific 'stop here' point, because the stopping condition can depend on objective, available resources, etc.

Typical stopping criteria include:

- When a given fitness has been reached.
  - This runs the risk of running forever, if the target fitness is not possible.
- When a set number of rounds have been completed.
- When a set time limit has been reached.
- When improvements plateau for a set number of rounds.