Amortised Analysis

In order to analyse data structures, it's insufficient to characterise their corresponding algorithms in terms of their worst case performance without context. While this is fine to do when analysing an algorithm over variables that can be arbitrarily valued, algorithms for data structures can be analysed with more context (i.e. some knowledge of the previous operations that were performed on the data structure). To do this, we perform amortised analysis, where instead of determining the worst case performance for each data structure operation, we get an upper bound on the average cost of an arbitrary sequence of operations.

Note that although this is an average over operations, this is distinct from average-case analysis, where one finds the expected performance given a probability distribution over inputs, as the upper bound is over all possible sequences.

There are three methods of performing amortised analysis:
- Aggregate analysis, where we get an upper bound on the total cost of a sequence of n operations, then divide by n.
- The accounting method, where we amortise the cost of each operation by modifying it, such that for any sequence of operations the sum of modified costs is no lower than the sum of real costs. If chosen deliberately this should simplify analysis.
- The potential method, where we define a non-negative potential function φ on the states of the data structure, measuring the 'entropy' of the structure at each point, allowing us to amortise the cost of operations by how they alter the entropy (do they make other operations easier or more complex to perform?).

Arguably, the accounting method is a more general means of performing aggregate analysis, by simplifying the problem to one that can be dealt with through aggregate methods, and the potential method is itself a general version of the accounting method.

More precisely, we want a C such that for any n, sequence $c_1, \ldots, c_n$, we have

$$C \geq \frac{1}{n}\sum_{k=1}^{n} c_k.$$

In certain cases we can immediately do aggregate analysis, and just observe a bound algebraically. This is especially easy when we only need to consider one or two operations with a straightforward or no algebraic relationship with one another. In other situations when we have more operations that have the possibility of interacting differently with one another it can be far more difficult to do this and still get a reasonable bound.

In more difficult cases, we might want to transform the sequence to $\widehat{c}_1, \ldots, \widehat{c}_n$ such that

$$\sum_{k=1}^{n} \widehat{c}_k \geq \sum_{k=1}^{n} c_k.$$

In order for this to work for an arbitrary sequence of operations, we redefine the cost of each operation by subtracting from high cost operations some degree of credit, which we pass on as additional charges to more frequent low cost operations. To do this one needs to make an argument that the charges account for the credit, but once this argument is made usually the amortised cost is $O(\max \widehat{c}_k)$.

As an example, take a stack with the operations $\textsc{Push}(S, x)$, $\textsc{Pop}(S)$, and $\textsc{Multipop}(S, m)$ (to pop up to m elements at once). Push and Pop are both of cost 1, while Multipop has cost $\min(n, m)$ where S is of size n. For each Push we can add an additional charge of 2, in exchange for a credit of 1 for each Pop, and a credit of $\min(n, m)$ for each Multipop. This makes Push the only operation for which a cost is actually assigned, giving $O(1)$ amortised cost.

In other cases, it's easier to conceptualise the amortisation not as credit for prepaid charges, but as the cost of each operation in addition to the degree by which it 'complicates' the state of the data structure. Strictly, with the set of configurations of the data structure being $\mathcal{C}$, we have $\varphi : \mathcal{C} \to [0, \infty)$, and then define the modified costs for a sequence of costs $c_1, \ldots, c_n$, states $s_0, \ldots, s_n$ by

$$\widehat{c}_k = c_k + \varphi(s_k) - \varphi(s_{k-1}),$$

which gives

$$\sum_{k=1}^{n} \widehat{c}_k = \varphi(s_n) - \varphi(s_0) + \sum_{k=1}^{n} c_k.$$

Thus as we take $\varphi(s_0) = 0$ by default, this gives an upper bound on the aggregate cost, and for any appropriate measure of the entropy of the data structure, we only need to understand how the entropy is changed by a new operation. Without working through details, an example of a φ for an m-bit number might be the number of digits equal to 1.

Disjoint sets

A commonly required data structure is one that represents a collection of disjoint sets, used for various applications such as Kruskal's algorithm for finding the minimum spanning tree of a graph, and the unification algorithm (**confirm how this is used?**).

Strictly, we want to store and update $\mathcal{F} = \{S_1, \ldots, S_n\}$ where for $i \neq j$, $S_i \cap S_j = \varnothing$. To do this we identify each set by its representative $\mathrm{rep}(S)$, which ought to be consistent (if we request $\mathrm{rep}(S)$ multiple times without modifying the set, we would expect the same answer). By default we hold 3 different operations: $\textsc{Make-Set}(x)$, $\textsc{Find-Set}(x)$, and $\textsc{Union}(x, y)$.

There are a wealth of different options for implementing this data structure. One immediate approach is to store each set as a linked list of elements. To speed up Find-Set, we need each object to point to the list head, making the operation constant. To perform Union, we will always need to iterate through all elements of one of the sets to update their head pointer, but we can prevent the need to do both by storing a pointer to the tail of each list. Either way Union is $\Theta(n)$, and our amortised cost is $\Theta(n)$.

An improvement to this can be made by always preferring that the new representative after the union is the one of the larger set, and thus we only need to iterate over the smaller set. To do this we need an additional store of the size of each set, which has no effect on performance. This allows us to get $O(\log n)$ amortised cost, by observing that each object's pointer is updated $O(\log n)$ times across n union operations.

An alternative to the linked list approach is to use trees to represent each set instead. This simplifies Union to two Find-Set operations from which you have one root inserted as the child of another, although this only improves performance if Find-Set is reasonably fast, which it isn't without improvements (currently worst-case $\Omega(n)$). We apply two heuristics which give our desired improvements:
- *Union by rank*, where we store the rank of each set (or an upper bound on it, at least), and ensure that the tree with a lower rank becomes the child of the other one.
- *Path compression*, where Find-Set involves updating the pointer of each element traversed to the root.

On its own, union by rank guarantees that rank will only be increased if both sets have equal rank, meaning for n elements the rank is at most $\log n$, so Find-Set and Union take $O(\log n)$ time.

Combining union by rank with path compression gives an extremely good bound of $O(\log^* n)$ amortised performance. The proof of this follows from a potential function defined by the sum of the potentials of each set element, noting that Union and Find operations will only decrease the entropy of the structure, and if they do it will be by a significant amount. This bound is so good that in practice it's constant.

Binary Search Trees

Binary search trees are used to maintain a dynamic set of orderable elements. Note they are similar but distinct from max heaps, as an inorder search returns their ordering, while for a max heap we only have a partial ordering from each parent to each child.

Typically we endow a BST with the operations $\textsc{Search}(x)$, $\textsc{Insert}(x)$, $\textsc{Delete}(x)$, $\textsc{Successor}(x)$. These all behave as expected – note that Successor returns the next largest value in the entire tree. All of these have running time $O(h)$ where h is the height of the tree.

Ideally we should just have that the tree is balanced, giving $O(\log n)$ amortised complexity. This of course isn't guaranteed in all implementations, so without extra work we get $O(n)$ complexity in a worst case. The key problem to deal with is one where we insert elements in order, which has the potential to create a tree with a single long branch.

Red-Black Trees

One way of ensuring that trees are balanced is using Red-Black trees. These are BSTs which identify each node as either red or black, with the following conditions:
- The root is black;
- Every leaf (NIL) is black;
- If a node x is red, $\mathrm{left}[x]$ and $\mathrm{right}[x]$ are both black;
- For each node x, all paths from x to any descendant pass through the same number of black nodes.

As convention we say that every valued node has 2 children. The above allows us to write without loss of generality the *black-height* for x, $\mathrm{bh}(x)$, as the number of black nodes on a path from x to a leaf (not including x).

Theorem 1 *A red-black tree T with n items has height $\leq 2\log n$*

Firstly, any subtree rooted at node x has at least $2^{\mathrm{bh}(x)}$ nodes, as collapsing each red node into its parent guarantees a tree of height $\mathrm{bh}(x)$ for which every node has ≥ 2 children. Further, at least half of the nodes on any path must be black, so $\mathrm{height}(x) \leq 2\mathrm{bh}(x)$, and thus with n nodes we have $\mathrm{height}(x) \leq 2\log n$.

To implement a tree obeying these properties, we need to use rotations to restructure the tree while maintaining the BST properties. Given a subtree with root y, $\mathrm{left}[y] = x$, with α, β the descendent trees of x and γ the right descendent tree of y, a right rotation about y moves x to the root, $\mathrm{right}[x] = y$, and β the left subtree of y. A left rotation behaves identically in the symmetric way.

In order to insert into a RB tree, we insert the element as normal, then colour it red so as to initially preserve the black height. This gives a few cases where there is a red-red violation:
- We could have that the uncle element is red, in which case we just recolour the parent and uncle, then recolour their parent. Then we've moved the problem from $p[x]$ to $p[p[x]]$, and can repeat.
- We could have that the uncle element is black and the triple x, $p[x]$, $p[p[x]]$ occur in a zig-zag. Then we rotate x with $p[x]$ to bring us to the next case.
- We could have that the triple occurs in a straight line, in which case we rotate $p[x]$ with $p[p[x]]$ and recolour.

In total this gives us $O(\log n)$ time (as case 2 and 3 occur at most once, because case 2 leads to case 3 which leads to termination), allowing us to get everything in $O(\log n)$. Deletions are more complicated, although also $O(\log n)$, but not directly considered in this course.

In certain cases, we can do better by considering the likelihood of particular elements being accessed. Given a distribution over the frequency of elements being accessed, we write T^* as the statically optimal BST which gives the minimum aggregate look-up cost.

We have from Knuth that there is an $O(n^2)$ DP algorithm to solve this, and from Mehlhorn that there is an $O(n \log n)$ algorithm which achieves at most a factor of $3/2$ over the optimal. Both require knowing the distribution beforehand however.

Splay trees

We introduce Splay trees as self-adjusting BSTs. The key idea is that when accessing an item x, move it to the root via a sequence of rotations. Thereby a tree develops for which the most accessed items stay near the root, and the least access items are the furthest away.

Until x is the root of T, if x has a parent y but no grandparent, then we rotate x with y. Otherwise if the triple x, $p[x]$, $p[p[x]]$ is aligned, rotate $p[x]$ with $p[p[x]]$ and then x with $p[x]$, and if it is not aligned rotate x with $p[x]$ then x with $p[p[x]]$.

The main point to notice here is that in the zig-zig procedure (where x, $p[x]$, and $p[p[x]]$ are aligned) ensures that surrounding vertices are brought up at the same time as the vertex being searched for. If a double rotation was used instead, there would be no wider improvement made to the tree, meaning certain sequences of searches would be $\Omega(n)$ amortised rather than $O(\log n)$.

We need to make some modifications to the other BST operations – in $\textsc{Insert}(T, x)$ we insert as normal, then $\textsc{Splay}(T, x)$. In $\textsc{Delete}(T, x)$ we $\textsc{Splay}(T, x)$, remove x, then take the furthest right element on the left $w = \max(T_{<x})$, $\textsc{Splay}(T_{<x}, w)$, and join $T_{<x}$ and $T_{>x}$.

Via $\varphi(T) = \sum_{x \in T} \log |T_x|$, we track the potential of a state as the sum of element ranks. This eventually, after far more technical detail than I want to write, gives that an empty Splay tree has $O((m+n)\log n)$ cost after n Insert and m Delete/Search operations. To replicate the proof, note that $\log |T_x|$ is the rank of x, and then determined the amortised cost in terms of $r(x)$ and $r'(x)$ for $\textsc{Splay}(T, x)$.

In fact, the same proof method can be used with each element's contribution to the rank weighted by its access frequency to show that the cost of Splay trees is within a constant factor of the cost of an optimal static tree for any distribution. Note that whether the same is true for an optimal dynamic tree (one where the tree is permitted to change during operations) is open.

Fixed Parameter Algorithms

While it is generally very difficult to get solutions for NP-hard problems, and usually we would prefer to just find approximate solutions, in certain instances we can make problems far easier by solving them for fixed parameters. For example, we might notice that for certain exponential time algorithms, they behave quite well for certain parameters bounded to a reasonable size.

Definition 1 *A parametrisation of a decision problem $\mathcal{P}$ is a computable function p that maps each input instance x to an integer value k.*
In general a parameter can be either explicit or implicit, it need not be computable in reasonable time (or at all). For example, we can take for VertexCover the parameter k, which is explicit, or for HamCycle the parameter which is the size of the minimum vertex cover.

A decision problem $\mathcal{P}$ if fixed-parameter tractable with respect to parameter p if there is an algorithm $\mathcal{A}$ which decides an instance x in polynomial time in $|x|$ holding k constant (provided the constant k contributes is computable).

With this framework, consider an algorithm for VertexCover where we take an edge (u, v) in G, then recurse on both $G' = G \setminus \{u\}$, and $G' = G \setminus \{v\}$, $k' = k - 1$. This gives time complexity $O(2^k |E|)$, so for small k this is very viable.

Kernelisation is a method to obtain FPT algorithms whereby the size of the instance is iteratively reduced, until the final instance is of size dependent on the parameter, and we can brute force it.

For VertexCover we can reduce by saying if a vertex is isolated we remove it from the graph, and if it has $> k$ neighbours we recurse on $(G \setminus \{v\}, k-1)$. Once neither rule applies, if G' has $> k^2$ edges, then reject, and otherwise (G', k') is a kernel which can be brute forced, yielding a $O(k^{2k}|E|)$ algorithm.

Linear Programming

Definition 2 (Linear Programs) *A linear program is a problem that can be expressed of the form:*

$$\begin{aligned} &\max\ \boldsymbol{c}^\top \boldsymbol{x} \\ &\textit{subject to } A\boldsymbol{x} \leq \boldsymbol{b}, \\ &\boldsymbol{x} \geq \boldsymbol{0}. \end{aligned} \tag{1}$$

For some $\boldsymbol{c} \in \mathbb{R}^n$, $\boldsymbol{b} \in \mathbb{R}^m$, $A \in \mathbb{R}^{m \times n}$.

LPs are a special case of optimisation problems, and are both generally easier to solve, but also occasionally useful for more general problems. At the same time, often problems cannot be formulated precisely in terms of an LP (which would put the problem in P, as will be demonstrated shortly), but rather require the additional restriction that $\boldsymbol{x}$ is integer-valued (and solving integer LPs is NP-hard).

Definition 3 (The Dual) *The dual to LP (1) is*

$$\begin{aligned} &\min\ \boldsymbol{b}^\top \boldsymbol{y} \\ &\textit{subject to } A^\top \boldsymbol{y} \geq \boldsymbol{c}, \\ &\boldsymbol{y} \geq \boldsymbol{0}. \end{aligned} \tag{2}$$

It's worth noting that through negation of A, $\boldsymbol{b}$, and $\boldsymbol{c}$ we get an equivalent maximisation LP, and consequently everything said about duals is essentially symmetric in either direction.

Lemma 2 (Weak LP duality) *Let $\boldsymbol{x}$ be any feasible solution to the primal LP (1), and $\boldsymbol{y}$ any feasible solution to the dual LP (2). Then*

$$\boldsymbol{c}^\top \boldsymbol{x} \leq \boldsymbol{b}^\top \boldsymbol{y}$$

To see this, note that $A\boldsymbol{x} \leq \boldsymbol{b}$, so for any $\boldsymbol{y} \geq 0$, $\boldsymbol{y}^\top A\boldsymbol{x} \leq \boldsymbol{b}^\top \boldsymbol{y}$. Further, $\boldsymbol{y}^\top A \boldsymbol{x} = \boldsymbol{x}^\top A^\top \boldsymbol{y}$, so as $\boldsymbol{y}$ is feasible and $\boldsymbol{x} \geq 0$, $\boldsymbol{y}^\top A \boldsymbol{x} \geq \boldsymbol{c}^\top \boldsymbol{x}$. This gives the desired inequality.

Theorem 3 (LP duality) *For the primal-pair of LPs (1) and (2), exactly one of the four cases occurs:*
- *Both (1) and (2) are infeasible.*
- *(1) is unbounded and (2) is infeasible.*
- *(2) is unbounded and (1) is infeasible.*
- *There exist solutions $\boldsymbol{x}$ and $\boldsymbol{y}$ to (1) and (2) respectively, and $\boldsymbol{c}^\top \boldsymbol{x} = \boldsymbol{b}^\top \boldsymbol{y}$.*

To show this, we introduce the simplex method of solving LPs, and then construct from the termination of the simplex method a dual solution equal to the primal.

Firstly, we convert (1) into slack form, by introducing the non-negative variables $\boldsymbol{\varepsilon} \in \mathbb{R}^n$ and augmenting each $\boldsymbol{a}_i^\top \boldsymbol{x} \leq b_i$ to $\boldsymbol{a}_i^\top \boldsymbol{x} + \varepsilon_i = b_i$. We then rewrite the LP as

$$\begin{aligned} z &= \boldsymbol{c}^\top \boldsymbol{x} \\ \boldsymbol{\varepsilon} &= \boldsymbol{b} - A\boldsymbol{x}, \end{aligned}$$

labelling variables on the LHS 'basic', and variables on the RHS 'non-basic'. Beginning by setting all non-basic variables to 0, at each iteration we take a non-basic variable u which occurs with positive coefficient in the first equation, and increase it until a basic variable v is made negative.

Thus provided the solution is bounded, the iteration concludes with $v = 0$, and we rewrite the system of equations by setting u to be basic and v non-basic (push u to the LHS of v's equation, then substitute wherever u occurs elsewhere). This gives an equivalent linear system, with all variables non-negative and thus feasible. This operation is known as a *pivot*.

We repeat this process until either the solution is determined to be unbounded, or all coefficients of the first equation are negative (so the objective is trivially maximised). Note that provided we break ties in a sensible manner (e.g. select the variable with the lower index) this will always terminate.

To briefly remark on initialisation: we can determine from the start if a problem is feasible by constructing an LP with an additional variable with its negation maximised, tracking the distance of the LP from the feasible region. This will always be feasible with some small manipulations, and so we can run the main simplex logic, returning feasibility if the optimal value is 0, infeasible otherwise.

Ultimately, if the solution is feasible and bounded we terminate with A', $\boldsymbol{b}'$, $\boldsymbol{c}'$, and the sets B and N specifying which variables are basic and non-basic. We can then construct a dual solution as follows:

$$y_i = \begin{cases} -c'_{n+i} & \text{if } (n+i) \in N \\ 0 & \text{otherwise.} \end{cases}$$

This can then be shown to be both feasible and equal to the optimal primal solution via algebra, and weak duality gives that this must then be an optimal dual solution.

The simplex method as given above is in EXP, although both useful for the above proof, and generally more practical than other methods. At the same time, we do have a couple different algorithms to solve LPs in polynomial time. The Ellipsoid method does this by noting that using duality allows us to reduce LPs to feasibility checks, and while we've used the Simplex method above to do feasibility checks, it is in fact possible to do so with minimisation methods for convex functions, such as the ellipsoid method, which approximates a solution with accuracy of any $\varepsilon > 0$ in polynomial time. Thus we get that solving LPs are in P.

Zero-sum games

Definition 4 (Zero-sum game) *A zero-sum strategic form game is defined by an $m \times n$ matrix $A = (a_{ij})$, where given a row-column choice i, j by the row and column players respectively, the row player receives payoff a_{ij}, and the column player $-a_{ij}$. A mixed strategy is a probability distribution over a player's choices.*

Given mixed strategies $\boldsymbol{p}$, $\boldsymbol{q}$ over rows and columns respectively, the expected payoff of the row player is $\boldsymbol{p}^\top A\boldsymbol{q}$, which is the negation of the payoff for the column player.

Definition 5 (Nash equilibrium) *For a zero-sum strategic form game defined by $A = (a_{ij})$, a pair of mixed strategies $(\boldsymbol{p}, \boldsymbol{q})$ is a Nash equilibrium if for every mixed strategy $\boldsymbol{r}$ on rows, $\boldsymbol{s}$ on columns,*

$$\begin{aligned} \boldsymbol{p}^\top A\boldsymbol{q} &\geq \boldsymbol{r}^\top A \boldsymbol{q} \\ \boldsymbol{p}^\top A\boldsymbol{q} &\leq \boldsymbol{p}^\top A \boldsymbol{s}. \end{aligned}$$

Nash equilibria are stable in the sense that even knowing the other player's strategy beforehand, neither player would gain from changing their strategy.

Given that the row player chooses $\boldsymbol{p}$, the column player will choose $\boldsymbol{q}$ minimising $\boldsymbol{p}^\top A\boldsymbol{q}$, so the row player should always choose $\boldsymbol{p}$ maximising $\min_{\boldsymbol{q}} \boldsymbol{p}^\top A\boldsymbol{q}$. In the opposite way, the column player should always choose $\boldsymbol{q}$ minimising $\max_{\boldsymbol{p}} \boldsymbol{p}^\top A\boldsymbol{q}$.

Theorem 4 (Von Neumann's theorem) *Given a zero-sum game defined by $A = (a_{ij})$,*

$$\max_{\boldsymbol{p}} \min_{\boldsymbol{q}} \boldsymbol{p}^\top A\boldsymbol{q} = \min_{\boldsymbol{q}} \max_{\boldsymbol{p}} \boldsymbol{p}^\top A\boldsymbol{q}.$$

Firstly, note that $\min_{\boldsymbol{q}} \boldsymbol{p}^\top A \boldsymbol{q} = \min_j \boldsymbol{p}^\top \boldsymbol{a}_{(\cdot)j}$, so we can write the maximisation problem $\max_{\boldsymbol{p}} \min_{\boldsymbol{q}} \boldsymbol{p}^\top A\boldsymbol{q}$ as

$$\begin{aligned} &\max\ z \\ &\text{s.t. } \mathbf{1}_n z - A^\top \boldsymbol{p} \leq \mathbf{0}_n \\ &\quad\ \ \mathbf{1}_m^\top \boldsymbol{p} \leq 1 \\ &\quad\ \ z \geq 0,\ \boldsymbol{p} \geq 0, \end{aligned}$$

which has dual

$$\begin{aligned} &\min\ w \\ &\text{s.t. } \mathbf{1}_n w - A\boldsymbol{q} \geq \mathbf{0}_n \\ &\quad\ \ \mathbf{1}_n^\top \boldsymbol{q} \geq 1 \\ &\quad\ \ w \geq 0,\ \boldsymbol{q} \geq 0. \end{aligned}$$

Noting that this corresponds to the minimisation problem $\min_{\boldsymbol{q}} \max_{\boldsymbol{p}} \boldsymbol{p}^\top A\boldsymbol{q}$, by LP duality we get Von Neumann's equality.

Minimum Cost Perfect Matchings

Definition 6 (Perfect Matchings) *Given a bipartite graph $(L \cup R, E)$ with $E \subseteq L \times R$ and a cost function $c : E \to [0, \infty)$, a perfect matching is a subset $M \subseteq E$ such that for each $v \in L \cup R$ there is exactly one $e \in M$ incident to v.*

Further, the cost of a perfect matching is defined as

$$\mathrm{cost}(M) = \sum_{e \in M} c(e)$$

We can formulate this as an LP:

$$\begin{aligned} \min\ & \sum_{e \in E} c(e) y_e \\ \text{subject to } & \sum_{e \sim u} y_e \leq 1 && \text{for } u \in L \\ & \sum_{e \sim v} y_e \geq 1 && \text{for } v \in R \\ & \boldsymbol{y} \geq \boldsymbol{0} \end{aligned}$$

Note that this is only equivalent when we restrict $\boldsymbol{y}$ to be integer valued, in which case as $|L| = |R|$ both inequalities turn into equalities. Considering the relaxed LP we get the following dual:

$$\begin{aligned} \max\ & \sum_{v \in R} x_v - \sum_{u \in L} x_u \\ \text{subject to } & x_v - x_u \leq c(u, v) && \text{for } (u, v) \in E \\ & \boldsymbol{x} \geq \boldsymbol{0} \end{aligned}$$

Our goal is then to write an algorithm which maintains a solution to the dual on a subgraph S while constructing a feasible solution to the primal. Once the solution to the dual is extended to the entire graph we then have a solution.

We begin with $M = \varnothing$, $S = \varnothing$, corresponding to $\boldsymbol{y} = \boldsymbol{0}$, $\boldsymbol{x} = \boldsymbol{0}$ a solution for the empty graph. While M is not a perfect matching:

1. We compute $c_{\boldsymbol{x}}(u, v) = c(u, v) + x_u - x_v$, then construct the graph $G_M = (X \cup Y, E \setminus M \cup \mathrm{rev}(M))$ with additional edges from s to unmatched vertices in L and from unmatched vertices in R to t.
2. We run Dijkstra on G_M to find a min-cost path from with respect to $d(e) = c_{\boldsymbol{x}}(e)$ if $e \notin M$, $d(\mathrm{rev}(e)) = -c_{\boldsymbol{x}}(e)$ if $e \in M$, and calculate the distance to each vertex $\delta(v)$ for $v \in L \cup R$. Thus $\min_{v \in R} \delta(v)$ gives a shortest path, beginning and ending with unmatched vertices labelled u_{start} and v_{end} respectively.
3. $\boldsymbol{x}$ is updated via $\boldsymbol{x}' = \boldsymbol{x} + \boldsymbol{\delta}$. This preserves feasibility while maintaining that it maximises the objective function on S'.
4. $M' = (P \setminus M) \cup (M \setminus P)$, $S' = S \cup \{u_{\text{start}}, v_{\text{end}}\}$ increasing the set of incident vertices by 2, thus increasing the set cardinality by 1.

To see that we preserve feasibility, note that for each $(u, v) \notin M$, we have

$$\begin{aligned} x'_v - x'_u &= x_v - x_u + \delta(v) - \delta(u) \\ &\leq x_v - x_u + d(u, v) \\ &= c(u, v), \end{aligned}$$

with equality if $(u, v) \in P$, as then the path to v via (u, v) is minimal, so $\delta(v) = \delta(u) + d(u, v)$. For each $(u, v) \notin M$, we first note that as M is a matching there is at most one backwards edge in G_M incident to any $u \in L$, and this is (v, u), so $\delta(u) = \delta(v) + d(v, u) = \delta(v)$ as $c(u, v) = x_v - x_u$ for $(u, v) \in M$. Thus

$$\begin{aligned} x'_v - x'_u &= x_v - x_u + \delta(v) - \delta(u) \\ &= x_v - x_u \\ &= c(u, v). \end{aligned}$$

Furthermore we preserve optimality as follows, noting that $\delta(u_{\text{start}}) = 0$ and $\delta(v_{\text{end}})$ is the cost of the shortest path.

$$\begin{aligned} \sum_{v \in R \cap S'} x'_v - \sum_{u \in L \cap S'} x'_u &= \sum_{v \in R \cap S'} x_v + \delta(v) - \sum_{u \in L \cap S'} x_u + \delta(u) \\ &= \sum_{v \in R \cap S} x_v - \sum_{u \in L \cap S} x_u + \sum_{(u,v) \in M} (\delta(v) - \delta(u)) + \delta(v_{\text{end}}) - \delta(u_{\text{start}}) \\ &= \mathrm{OPT}_S + \delta(v_{\text{end}}) \\ &= \sum_{e \in M} c(e) + \Bigg(\sum_{(u,v) \in P \setminus M} (c(u, v) + x_u - x_v) \\ &\qquad - \sum_{(u,v) \in \mathrm{rev}(P) \cap M} (c(u, v) + x_u - x_v) \Bigg) \\ &= \sum_{e \in M'} c(e) + x_{u_{\text{start}}} - x_{v_{\text{end}}} \\ &= \sum_{e \in M'} c(e) \end{aligned}$$

As M is a min-cost perfect matching on S, thus any augmenting path corresponds to a perfect matching on S'. **Not sure how to continue proving here**.

Consequently we have shown that the algorithm terminates with an optimal solution to the dual, and thus an optimal solution to the primal, which is a relaxed form of the initial problem. Note that our solution is in fact integral however, so we have proven that there is an integral optimal solution equivalent to the relaxed solution, and this is the one we return.

Approximation Algorithms

We say that algorithm $\mathcal{A}$ is a ρ-approximation algorithm for a minimisation problem $\mathcal{P}$ if for every instance x of $\mathcal{P}$, $\mathcal{A}(x) \leq \rho \cdot \mathrm{OPT}(x)$. In the opposite way, algorithm $\mathcal{A}$ is a ρ-approximation algorithm for a maximisation problem $\mathcal{P}$ if for every instance x of $\mathcal{P}$ we have $\mathcal{A}(x) \geq \rho \cdot \mathrm{OPT}(x)$.

There are 3 main approximation techniques:
- Combinatorial algorithms, which use counting methods to find a separate bound.
- LP rounding, wherein we express the problem using an integer linear program, then relax the integrality constraints and solve the standard LP, then round and argue that the integral solution is not far from the optimal.
- Randomisation, wherein we use probabilistic arguments to allow us to make more general choices and argue from probability that they give good results.

In particular we can use approximation algorithms to focus on the optimisation form of several NP-complete problems considered in the block to the right. For example, while we might have difficulty finding if there is a vertex cover of size $\leq k$, we can approximate the minimum vertex cover by some reasonable factor to help with this. Elsewhere, while SAT is difficult to perform, we can approximate the maximum number of clauses that can be satisfied at any one time for a given formula.

$\textsc{MinVertexCover}(G)$
$C = \varnothing$
$M = \varnothing$
while there is $(u, v) \in E$ with $u, v \notin C$
  $C := C \cup \{u, v\}$
  $M := M \cup \{(u, v)\}$
return C

The above provides a vertex cover of size C. With M the set of edges that trigger the while loop (noting no edge can trigger it twice), we have $|C| = 2|M|$, and as no vertex occurs twice in M, for each edge in M a vertex must be added to the cover, so $|M| \leq |C^*|$ and thus $|C| \leq 2|C^*|$.

Another method of constructing the above algorithm is using LP rounding. We get the following relaxed LP for the problem, from removing the specification that $\boldsymbol{x}$ is integer-valued.

$$\begin{aligned} \min\ & \sum_{v \in V} x_v \\ \text{s.t. } & x_u + x_v \geq 1 && \text{for } (u, v) \in E \\ & \boldsymbol{x} \geq \boldsymbol{0} \end{aligned}$$

If we construct a vertex cover such that $v \in C$ iff $x_v \geq 1/2$, then we are never more than doubling, and so we get a $|C| \leq 2|C^*|$, and LPs are solvable in polynomial time so this is a relatively fast procedure.

In more complex scenarios we may need to work on constructing a solution where we round according to a random $T \sim U(0, 1)$, and then use the behaviour of the random construction to demonstrate either that any optimal solution is integer-valued in the first place, or argue otherwise that the solution is approximate.

Using the third method of randomisation, we can get an approximation to MaxSat by taking a random assignment and outputting the number of satisfied clauses under the assignment. The expected number of clauses satisfied is $\geq |C|/2$, so $\mathbb{P}(|C| - X \geq (|C|+1)/2) \leq |C|/(|C|+1)$, meaning running repeatedly should relatively quickly give a $1/2$ approximation with very high probability (tending to 1).

Another means of getting an approximation is to reformulate MaxSat as an integer LP, then relax it and use the optimal solution to form a random distribution over assignments.

NOTE: Have currently missed off details regarding TSP approximations – consider looking at this further.

Flow Networks

Flow networks are an abstraction to capture networks where edges capture some sort of traffic, and nodes act as switches passing traffic. Formally, it is a tuple (G, s, t, c) where $G = (V, E)$ is a directed graph, we have a source $s \in V$, a sink $t \in V$, and a capacity function $c : E \to \mathbb{Z}_{\geq 0}$.

For simplicity we assume that there are no anti-parallel edges (edges (v, u) and (u, v)), that no edge enters s, and no edge leaves t.

An s-t cut is a partition of V into two sets A, B such that $s \in A$, $t \in B$. The capacity of an s-t cut is the sum of the capacities of edges exiting A. The minimum cut problem is that of finding an s-t cut of minimum capacity.

A flow is an assignment $f : E \to \mathbb{R}_{\geq 0}$ where for each $e \in E$, $f(e) \leq c(e)$, and for each $v \in V \setminus \{s, t\}$, the sum of $f(e)$ for e going into v is the sum of $f(e)$ for e out of v. The value of f is the sum of $f(e)$ for e out of s. The maximum flow problem is that of maximising the flow value.

With (G, s, t, c) a flow network, then for a flow f, s-t cut (A, B), then

$$\begin{aligned} |f| &= \sum_{e \text{ out of } s} f(e) \\ &= \sum_{v \in A \setminus \{s\}} \Bigg(\sum_{e \text{ out of } v} f(e) - \sum_{e \text{ into } v} f(e) \Bigg) + \sum_{e \text{ out of } s} f(e) \\ &= \sum_{v \in A} \Bigg(\sum_{e \text{ out of } v} f(e) - \sum_{e \text{ into } v} f(e) \Bigg) \\ &= \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ into } A} f(e) \end{aligned}$$

Hence $|f| \leq c(A, B)$ (the capacity of the (A, B) cut). Thus we get weak duality, that if $|f| = c(A, B)$, then f is maximum flow. Note we also get this from identifying that max flow and min cut are each integer LPs, and together a primal dual pair.

We have the idea of a residual graph for a greedy method of maximising flow. With respect to a specific flow, we construct $G_f = (V, E_f)$, where for each $e \in E$, if $f(e) < c(e)$ we have a forwards edge e to E_f with $c_f(e) = c(e) - f(e)$, and if $0 < f(e)$, then we had a backwards edge $e' = \mathrm{rev}(e)$ with $c_f(e') = f(e)$. Thus we can trace from t the flow back to s, and the forward edges represent lost flow which could move through that edge.

Constructed as such, we have that each path from s to t in the residual graph represents unused flow, and indeed we can pump $b_P = \min_{e \in P} c_f(e)$ units of flow along this path to create f'. Note that in certain instances this involves requisitioning flow along some edge to provide it elsewhere, so if for $e \in E$, $\mathrm{rev}(e) \in P$, we need to reduce the flow along e by b_P.

The Ford-Fulkerson method is developed along these lines:

$\textsc{Ford-Fulkerson-Method}(G, s, t, c)$
for $e \in E$
  $f(e) = 0$
$G_f = G$
while there exists a path P from s to t in G_f
  Select P
  $b_P = \min_{e \in E} c_f(e)$
  Augment f along P by b_P
  Update G_f
return f

For each iteration, we increase $|f|$ by the bottleneck capacity of P, which for integer capacities is at least 1, so it takes at most $|f^*|$ iterations to complete the while loop. Further, the execution of the loop takes $O(|E|)$ time, as either breadth-first or depth-first search are $O(|V| + |E_f|) = O(|E|)$. If capacities are irrational, without more precise choice of paths it's possible to reach a cycle where the bottleneck capacity approaches 0 asymptotically. Thus we get initially a bound of $O(|E| \cdot |f^*|)$ (and as we don't know $|f^*|$ in most cases, perhaps the more helpful bound is $O(|E|C)$ with $C = \max_e c(e)$).

Augmentation of f in this way is actually a special case of a more general notion. In general we write for f a flow on G, f' a flow on G_f, $f \uparrow f' = (f + f' - (f' \circ \mathrm{rev})) \cdot \mathbb{1}_E$, which is a flow on G satisfying $|f \uparrow f'| = |f| + |f'|$. The special case here is of taking $f_P = b_P \cdot \mathbb{1}_P$.

Upon termination, G_f has no s-t path, so it is disconnected. The corresponding cut (A, B) has $c(A, B) = |f|$, so by weak duality the algorithm returns the max flow. This is true because for any forward edge crossing (A, B), $f(e) = c(e)$ (as otherwise we would have a forward edge across the cut), and for any backwards edge crossing (A, B) we have $f(e) = 0$ (as otherwise, again, we get a forward edge across the cut). Thus $|f| = c(A, B)$.

Refinements to the Ford-Fulkerson method

The bulk of time lost in the Ford-Fulkerson method is with respect to the choice of path at each stage. If we can find an improvement on this, while it may increase the time taken to find a path, we can guarantee termination within a shorter time.

One method of doing this is to modify the residual graph so as to guarantee that we select a path with bottleneck of the same order as the maximum path. To do this we add an additional parameter for constructing the residual graph, Δ, where $G_f(\Delta)$ is the subgraph of G_f with edge capacities $\geq \Delta$. Thus any path P in $G_f(\Delta)$ has $b_P \geq \Delta$, and so at the end of each Δ-scaling phase we have $|f^*| < |f| + m\Delta$, and thus $O(m)$ path searches (each $O(m)$) per phase, of which there are $\log C$, so $O(m^2 \log C)$ complexity.

$\textsc{Capacity-Scaling}(G, s, t, c)$
Set $f(e) = 0$ for all $e \in E$
$C = \max_{e \in E} c(e)$
$\Delta = \max\{2^n \mid 2^n \leq C\}$
while $\Delta \geq 1$
  Compute $G_f(\Delta)$
  while there exists s-t path P in $G_f(\Delta)$
    $f = \textsc{Augment}(f, P)$
    Update $G_f(\Delta)$
  $\Delta = \Delta/2$
return f

It turns out however that, provided we always select the shortest path returned by BFS, we can remove runtime dependence on C entirely in our analysis. Assume that there is some flow augmentation on G_f, and $v \in V \setminus \{s, t\}$ is the vertex minimising $\delta_{f'}(s, v)$ such that $\delta_{f'}(s, v) < \delta_f(s, v)$. Then with $p = s \rightsquigarrow u \to v$ a shortest path, we have $\delta_f(s, u) \leq \delta_{f'}(s, u) = \delta_{f'}(s, v) - 1$, and so if $(u, v) \in E_f$ then

$$\begin{aligned} \delta_f(s, v) &\leq \delta_f(s, u) + 1 \\ &\leq \delta_{f'}(s, v). \end{aligned}$$

Thus the augmentation introduces (u, v), meaning flow must have just been increased on a path from s to u via v, and further this must be a shortest path, creating a contradiction.

From here, we then describe each edge $e \in E$ as critical in an augmentation if $b_P = c_f(e)$. Once an edge is critical it is removed from the graph, and only returned if $\mathrm{rev}(e)$ is on an augmenting path, which requires the augmenting path distance to have increased by ≥ 2. Thus e appears in at most $|V|/2$ paths (one for every two distances), so we get that there are $O(|V||E|)$ augmentations, each taking $O(|E|)$ time, so $O(|V||E|^2)$ time.

In summary, we get
- Ford-Fulkerson runs in time $O(mC)$ by default.
- Capacity-Scaling runs in time $O(m^2 \log C)$.
- Edmonds-Karp runs in time $O(m^2 n)$.
- State of the art algorithms are $O(mn)$.

Applications

Max-flow solutions to flow networks are quite flexible, and many other problems can be reduced to Max-Flow. An alternative problem mentioned in the course is that of circulation with demands, where we have a directed graph to which capacity bounds are assigned to each edge, and capacity bounds to each vertex. The goal is to determine if there is a circulation for which the flow going through each vertex is equal to the demand, and each edge is within capacity. We can reduce this to a flow network by introducing s with an edge to every vertex with negative demand, and t with an edge from each vertex with a positive demand. We have a circulation iff all edges from s and to t are at capacity.

Another problem is that of bipartite matchings, which has a similarly straightfoward reduction from Max-Flow. Given an undirected bipartite graph $(L \cup R, E)$ with $E \subseteq L \times R$, we say that an $M \subseteq E$ is a matching iff no two edges in M share an endpoint. The maximum cardinality matching can be found by introducing s with edges to L, t with edges from R, then setting every edge to have capacity 1. The max-flow is equal to the maximum cardinality matching.

A further problem is more specifically that of perfect matchings, where we want to determine if there is a matching of cardinality $|L| = |R|$, and further provide a proof if not. The first part of this is straightforward via Max-Flow, although the second part requires some work.

Writing for any set of vertices S, $N(S) = \{e \in E : \exists v \in S\,.\, v \sim e\}$, we have the following theorem.

Theorem 5 (Hall's theorem) *A bipartite graph $G = (L \cup R, E)$ has a perfect matching iff for every $S \subseteq L$, $|N(S)| \geq |S|$.*

The forward direction is immediate. For the backwards direction we construct a flow network, and note that if for every $S \subseteq L$, $|N(S)| \geq |S|$, then for any min cut (A, B) we write $A_L = A \cap L$, $A_R = A \cap R$, and get the following

$$\begin{aligned} \mathrm{cap}(A, B) &= |\{e \in E : \exists u \in A,\ v \in B\,.\, e = (u, v)\}| \\ &= |L \setminus A_L| + |N(A_L) \setminus A_R| + |A_R| \\ &= |L| - |A_L| + |N(A_L)| \\ &\geq |L| \end{aligned}$$

so by weak duality we get that the flow is a perfect matching.

Thus it's sufficient to show a certificate $S \subseteq L$ such that $|N(S)| < |S|$ to demonstrate that there is no perfect matching, and this can be found by constructing a flow network where all edges from L to R have infinite capacity, meaning the min cut produces this certificate.

There is an additional, more difficult problem of performing the same with the goal of minimising the cost of the perfect matching (each edge assigned its own cost). This is more easily dealt with using LPs than directly through flows.

Intractability

Definition 7 *For a decision problem L, $L \in$ NP if there exists a non-deterministic Turing machine computing L in time polynomial in its input size.*

Definition 8 (Reducibility) *For languages $L_1, L_2 \subseteq \{0,1\}^*$, we say that L_1 is polynomial-time Karp reducible to L_2, $L_1 \leq_p L_2$ if there is some computable function $f : \{0,1\}^* \to \{0,1\}^*$ such that $x \in L_1$ iff $f(x) \in L_2$ (alternatively written, $L_1 = f^{-1}(L_2)$).*

While this is the most common notion of reducibility, there are other notions:
- Truth-table reducibility, allowing us to call the TM for L_2 a constant number of times, so we have computable $\boldsymbol{f} : \{0,1\}^* \to (\{0,1\}^*)^k$, and pass the resultant $\mathbb{1}_{L_2} \circ \boldsymbol{f}$ into the fixed k-ary formula φ.
- Cook / Turing reducibility, allowing us to call the TM for L_2 a polynomial number of times during execution.
- Log-space reducibility, where $f(x)$ must have length polynomially bounded in $|x|$, and one can compute both whether index i is in $f(x)$ and whether $f(x)_i = 1$ in logarithmic time.

Definition 9 (NP-hardness) *A language $L \subseteq \{0,1\}^*$ is NP-hard if for all $L' \in$ NP, $L' \leq_p L$.*

Further, we say that L is NP-complete if $L \in$ NP. Note that if any NP-hard L were to have a polynomial-time algorithm, then NP = P.

We introduce the following NP-complete decision problems:
- $\textsc{Clique}(G, k)$, determining whether there is a clique of size $\geq k$ (a clique is a set $S \subseteq V$ where every pair is connected by an edge).
- $\textsc{IndSet}(G, k)$, determining whether there is an independent set of size $\geq k$ (an independent set $S \subseteq V$ has no pairs connected by an edge).
- $\textsc{VertexCover}(G, k)$, determining whether there is a vertex cover of size $\leq k$ (a vertex cover is a set $S \subseteq V$ where each $e \in E$ is incident to some $v \in S$).
- $\mathrm{SAT}(\varphi)$, determining whether the CNF formula φ has a variable assignment $\mathcal{A}$ for which $\mathcal{A} \vDash \varphi$.
- $\mathrm{3SAT}(\varphi)$, the restriction of SAT to CNF formulas for which each clause has ≤ 3 variables.
- $\textsc{LargeCut}(G, M)$, determining whether G has a cut of size $\geq M$.
- $\textsc{HamCycle}(G)$, determining whether G has a Hamiltonian cycle, one which goes through every vertex exactly once.
- $\textsc{3Col}(G)$, determining whether there exists an assignment $c : V \to \{\text{red}, \text{green}, \text{blue}\}$ such that for each $(u, v) \in E$, $c(u) \neq c(v)$.
- $\textsc{4Col}(G)$, determining the same for 4 colours.
- $\textsc{SetCover}(U, \mathcal{F}, k)$, determining for a set U, collection $\mathcal{F}$ of subsets of U, whether we can choose k sets from $\mathcal{F}$ with union U.
- $\textsc{HittingSet}(U, \mathcal{F}, k)$, determining if there is $U' \subseteq U$ with $|U'| \leq k$ such that $U' \cap S_i \neq \varnothing$ for all i (U' 'hits' all S_i).
- $\textsc{SubsetSum}(\{a_1, \ldots, a_n\}, K)$, determining if there is $I \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in I} a_i = K$.
- $\textsc{Partition}(\{a_1, \ldots, a_n\})$, determining whether there is $I \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in I} a_i = \sum_{i \notin I} a_i$.

Within the above, the first few problems concerning graphs are particularly closely related to one another. We can reduce IndSet to Clique by setting $E' = (V \times V) \setminus E$ (and vice-versa). We can also reduce IndSet to VertexCover by noting that $S \subseteq V$ is a vertex cover iff $V \setminus S$ is an independent set, so $f(G, k) = f(G, n-k)$ (and vice-versa to get $\textsc{VertexCover} \leq_p \textsc{IndSet}$).

Theorem 6 (Cook-Levin theorem) *Both SAT and 3SAT are NP-complete.*

Roughly, the proof of SAT being NP-complete is done through constructing a polynomial-size CNF which verifies that a sequence of snapshots of a polynomial-time TM is accepting. If this TM is the polynomial verifier for a language in $L \in$ NP, thus we have that $x \in L$ iff this formula is satisfiable. From here the reduction to 3SAT can be done through algebraic manipulation of formulae.

Using this theorem allows us to then prove that all the above problems are in fact NP-hard (and all are clearly NP). This is because we can reduce 3SAT to VertexCover. To do this, given a formula φ with n variables and m clauses (which without loss of generality we assume each have exactly 3 variables, which ends up fine with some algebraic manipulation), for each variable x we include a variable gadget composed of x, $\neg x$, and the edge $(x, \neg x)$; for each clause $\{x, y, z\}$ we include a clause gadget which is a clique of 3 vertices, each one with an edge to their corresponding vertex in their variable gadget.

Thus we get that for any vertex cover of the resulting graph, each variable gadget has at least one variable, and each clause gadget has at least two variables, so the vertex cover has size $\geq n + 2m$. If there was a vertex cover of size $\leq n + 2m$ (equivalently, equal to $n + 2m$), then we take the vertex in each variable gadget and assign it 1. For each clause gadget there is a vertex not in the cover, and so that variable must have been assigned 1 (and so included in the cover within its gadget), meaning all clauses are satisfied so we have satisfiability.

On the other hand, if φ is satisfiable, then take the satisfying assignment and include every variable assigned true in its variable gadget in the vertex cover, meaning we must include every instance of a variable assigned false in a clause gadget in the vertex cover. As we have that φ is satisfiable, we always have that ≤ 2 variables need to be included in the vertex cover in each clause gadget, so thus we get a vertex cover of size $\leq n + 2m$.