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). $\textsc{Push}$ and $\textsc{Pop}$ are both of cost 1, while $\textsc{Multipop}$ has cost $\min(n, m)$ where S is of size n. For each $\textsc{Push}$ we can add an additional charge of 2, in exchange for a credit of 1 for each $\textsc{Pop}$, and a credit of $\min(n, m)$ for each $\textsc{Multipop}$. This makes $\textsc{Push}$ 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 $\textsc{Find-Set}$, we need each object to point to the list head, making the operation constant. To perform $\textsc{Union}$, 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 $\textsc{Union}$ 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 $\textsc{Union}$ to two $\textsc{Find-Set}$ operations from which you have one root inserted as the child of another, although this only improves performance if $\textsc{Find-Set}$ 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 $\textsc{Find-Set}$ 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 $\textsc{Find-Set}$ and $\textsc{Union}$ 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 $\textsc{Union}$ and $\textsc{Find}$ 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 $\textsc{Successor}$ 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, left[x] and 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, 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, 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 $\textsc{Insert}$ and m $\textsc{Delete}$/$\textsc{Search}$ 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.

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 $c : E \to \mathbb{R}_{\geq 0}$.

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

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

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.

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.

Take a path from s to t in the residual graph - this uses the edges along which flow was lost, ensuring we can make an improvement to $|f|$. Then the minimum $c_f(e)$ for $e \in P$, if $e \in P$ set $f'(e) = f(e) + b_P$, and if $\mathrm{rev}(e) \in P$ set $f'(e) = f(e) - b_P$. This constructs a new valid flow.

The **Ford-Fulkerson** algorithm does precisely this. It sets $f = 0$ initially, constructs the residual graph G_f, then applies the above augmentation to f, updates G_f, then repeats until there is no s-t path in G_f. For each iteration, we increase $|f|$ by the bottleneck capacity of P, which is at least 1, so it takes at most $|f^*|$ iterations to complete the while loop. Further, the execution of the loop takes $O(m)$ time. Note that this only terminates necessarily because the capacities are integers (**why?**).

Upon termination, G_f has no s-t path, so disconnected. The corresponding cut (A, B) has $c(A, B) = |f|$, so 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 $|f| = c(A, B)$.

In order to choose the s-t path P to reduce the time complexity, we want to use the P with maximum bottleneck capacity, so to increase $|f|$ by as much as we can at each iteration. One option would be to adapt Dijkstra's algorithm, and the other would be to do a binary search on the value of b_P. With $\Delta \geq 1$, let $G_f(\Delta)$ be the subgraph of G_f with edge capacities $\geq \Delta$. We use the following algorithm:

```
Capacity-Scaling(G, s, t, c)
Set f(e) = 0 for all e ∈ E
C = max_{e∈E} c(e)
Δ = max{2^n | 2^n ≤ C}
while Δ ≥ 1
    Compute G_f(Δ)
    while there exists s-t path P in G_f(Δ)
        f = Augment(f, P)
        Update G_f(Δ)
    Δ = Δ/2
return f
```

It turns out that we can actually remove runtime dependence on C entirely however, by the **Edmonds-Karp** algorithm which just selects the path with the fewest edges at each point of the iteration.

Make sure to analyse the runtime of all the above

In summary, we get

- **Ford-Fulkerson** runs in time $O(mnC)$.
- **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 $O(mn)$.

Given an undirected bipartite graph $(X \cup Y, E)$, (X, Y) denotes the bipartition of the vertex set of G. Note any edge is of the form (x, y) for some $x \in X$, $y \in Y$. With a cost function $c : E \to [0, \infty)$, for a set $S \subseteq E$ we write $c(S)$ as the sum cost of S, and we want to find the minimum cost perfect matching (a matching where every vertex appears exactly once).

Linear Programming

A linear programming problem is one involving maximisation or minimisation of a cost function subject to constraints formed of weak inequalities and equality. The maximum flow problem is itself a linear programming problem.

We say that a linear program is feasible if there is a solution, infeasible otherwise. The fundamental theorem of linear programming states that for each LP, either the LP is infeasible, it is feasible but unbounded, or it is bounded, feasible, and there is a solution.

To solve LPs naively we initially draw out the feasible set, then calculate the cost manually at each intersection point. To do this more systematically, we can try to pick multipliers for each equality to solve a different LP. This ends up being the dual LP.

Written precisely: a general LP consists of n real variables $x_1, \ldots, x_n$, a linear objective function to be maximised or minimised, a set of $\leq$ constraints upper bounded by some b_i for $i \in \{1, \ldots, m\}$, a set of $=$ constraints equal to b_i for $i \in \{m+1, \ldots, p\}$, and a set of $\geq$ constraints lower bounded by some b_i for $i \in \{p+1, \ldots, q\}$.

We convert initially to maximum standard form by multiplying the objective function by -1, replacing equality constraints by two inequality constraints, convert each inequality constraint to a $\leq$ inequality, and make all variables non-negative by writing each x as $x^+ - x^-$ for both positive.

Having got the maximum standard form, we multiply each constraint by $y_i \geq 0$, sum the inequalities to obtain

$$\sum_{i=1}^{m} y_i \left(\sum_{j=1}^{n} a_{ij} x_j \right) \leq \sum_{i=1}^{m} b_i y_i$$

and we ensure that the objective function $\sum_{i=1}^{n} c_i x_i \leq L$ by having $c_j \leq \sum_{i=1}^{m} a_{ij} y_i$, and then our goal is just to minimise the upper bound in the y_is

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.