Algorithms and Data Structures – Ike Glassbrook

Data Structures

Data structures are fundamental in maintaining dynamic sets. In order to analyse and evaluate their construction we generally either use worst case or amortised analysis. Where for a sequence of operations we have costs $c_1, \dots, c_n, \dots$, and we want to put an upper bound on their sum, we can take the most expensive of all the operations between 1 and n, label this C and thus our bound on the cost is Cn. Alternatively, we say that the sequence has amortised cost $\leq C$ if for every $i \in \{1, 2, \dots, n\}$ we have that $\sum_{k=1}^{i} c_k \leq Ci$.

For example, take the data structure of a k-bit integer for which we have the operation increment. If we consider the cost of incrementing it from 0 to $2^k - 1$ in its lifetime, the worst case analysis notes that at worst we have to flip k bits in an operation, so the amortised analysis has it as $O(nk)$. The amortised analysis notes that while the first bit is always flipped, the second bit is flipped only have the time, the third bit a quarter of the time, etc. Thus the aggregate cost is $O(n)$, giving amortised cost $O(1)$.

Another way to consider this is using the banker's method, where we note that certain states of the data structure have a certain degree of entropy which needs to be resolved. For each change from a 0 to a 1 we should note that this will end up being changed back after further increments, so we bank not just a cost for the change, but additionally a cost for the entropy it introduces. Meanwhile, a change from 1 to 0 necessitates no extraneous changes (indeed, it restores the structure to a less complex state), so no cost is incurred here.

A more abstract version of this is the potential method:
Definition 1 *Let $\mathcal{C}$ be the set of all possible configurations of the data structure. The potential function $\varphi : \mathcal{C} \to [0, \infty)$ is the potential value assigned to each state.*

For a potential function φ we set the charge $x_i = c_i + \varphi(i) - \varphi(i-1)$. Telescoping then gives us that $\sum_{i=1}^{n} c_i \leq \varphi(0) + \sum_{i=1}^{n} x_i$. Usually we set $\varphi(0) = 0$ so the amortised cost is less than the maximum charge, so our goal is to determine the asymptotic behaviour of x_n.

The potential function captures the total credit of the data structure at one time. Expensive operations are compensated for by decreasing potential.

The Disjoint-Set Problem

To maintain a dynamic collection of disjoint sets $\mathcal{F} = \{S_1, \dots, S_n\}$, we identify each set by its representative rep(S). This ought to be consistent (if we request rep(S) multiple times without modifying the set, the answer should be the same).

We hold 3 different operations on this set: MAKE-SET(x), FIND-SET(x), and UNION(x, y). We motivate this by its use in Kruskal's algorithm.

There are several ways of going about this. A naive method would work by just identifying rep with an array, although this gives $\Theta(n)$ time for UNION(x). Another method works using singly linked lists, for which union becomes relatively straightforward, although still $\Omega(n)$ in the worst case. A better method works by storing each set as a tree, which makes FIND-SET(x) slightly worse (roughly $\Theta(\log n)$) while improving UNION(x, y) to $O(\log n)$.

As heuristic we can try to use link by height. This allows us to improve our asymptotic complexity to $O(\log n)$. Further, we can then use path compression by introducing additional operations to every use of FIND, pulling up every element scanned to the first level and ensuring that the trees end up of minimum possible depth. This gives us total complexity $O(m \log^* n)$ for m operations on a tree of size n.

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 SEARCH(x), INSERT(x), DELETE(x), 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 necessary in all implementations, so we could theoretically 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.

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. Every leaf is black, and whenever a node is red, this implies that both of its children are black. Furthermore, for any node x, all paths from x to a descendant pass through the same number of black nodes (excluding x). As convention we say that every valued node has 2 children.

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

To see this, first note that the height$(T) \leq 2\text{bh}(T)$ where bh(x) is the number of black nodes found on any path from x to a descendant. This is because we cannot have more than 1 consecutive red node on the path, so for every black node there can be at most 1 red node. Further, if we collapse every red node to its black parent, then each node has ≥ 2 children, the number of null elements in the tree is $n+1$, and so the height is $\log(n+1)$.

To implement a tree obeying these properties, we need the additional operations of restructuring the tree, and recolouring the tree. Rotations transform one BST to another - in a right-rotation the specified element is taken and moved to the right, while its left child becomes its parent, and any of its own children are moved around appropriately.

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 $p[x]$.
- 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), allowing us to get everything in $O(\log n)$.

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, determine T^* 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.

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 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 INSERT(T, x) we insert as normal, then SPLAY(T, x). In DELETE(T, x) we SPLAY(T, x), remove x, then

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

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

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 $e' = \text{rev}(e)$ 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 $|f|$. Thus with b_P the minimum $c_f(e)$ for $e \in P$, if $e \in P$ set $f'(e) = f(e) + b_P$, and if 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)
 1  Set f(e) = 0 for all e ∈ E
 2  C = max_{e∈E} c(e)
 3  Δ = max{2^n | 2^n ≤ C}
 4  while Δ ≥ 1
 5      Compute G_f(Δ)
 6      while there exists s-t path P in G_f(Δ)
 7          f = AUGMENT(f, P)
 8          Update G_f(Δ)
 9      Δ = Δ/2
10  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 $x_1, \dots, x_n$, a linear objective function to be maximised or minimised, a set of $\leq$ constraints upper bounded by some b_i for $i \in \{1, \dots, m\}$, a set of $=$ constraints equal to b_i for $i \in \{m+1, \dots, p\}$, and a set of $\geq$ constraints lower bounded by some b_i for $i \in \{p+1, \dots, q\}$.

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