Algorithms and Data Structures Notes

William Guest

January 2023

1 Amortized Analysis

Consider a sequence of operations in a data structure, with costs $c_1, \ldots, c_n, \ldots$. A worst case cost analysis would give an upper bound of

$$C = \max_{i=1,\ldots,n} c_1, \ldots, c_n$$

Hence the total cost for the first n operations is then $\leq Cn$. But some operations may be expensive and infrequent whilst others may be cheap and frequent.

Def 1 (Amortised Analysis)**.** For a sequence of operations $c_1, \ldots, c_n$, we say that they have amortised cost C if for every $i = 1, 2, \ldots, n$ we have that

$$c_1 + \ldots, +c_1 \leq Ci$$

Another way to consider amortised analysis is

$$\max_i \frac{c_1 + \ldots + c_n}{i}$$

We can think of the amortized cost as the average cost of each operation.

1.1 Aggregate Method

In aggregate anaylsis, we determine an upper bound for the cost of n operation, $T(n)$. The amortised cost is therefore $\frac{T(n)}{n}$, even when there are several types of operations in the sequence.

1.2 Accounting Method

Also known as the "Banker's method", we assign "charges" to different operations, with some operations charged more or less than what they actually cost. We call the amount we charge an operation the **amortised cost**. When an operation's amortised cost exceeds the actual cost, this difference is known as the **credit**. This credit can help to pay for later operations.

In order to analyse the average cost per operation, we need to choose the amortised cost of each operation such that the total amortised cost is greater than or equal to the total actual cost, (the amortized cost is an upper bound on the actual cost) i.e. for **all** sequences of n operations,

$$\sum_{i=1}^{n} \hat{c_i} \geq \sum_{i=1}^{n} c_i$$

where $\hat{c_i}$ is the amortised cost, and c_i is the actual cost of the ith operation. This leads to the credit invariant

$$\sum_{i=1}^{n} \hat{c_i} - \sum_{i=1}^{n} c_i \geq 0$$

meaning that we must always have non-negative credit during the operation.

1.3 Potential Method

Instead of representing prepaid work with "credit", which is stored with specific objects in the data structure, the potential method represents the prepaid work as "potential", which can also be released to pay for future operations. This potential is associated with the entire data structure, rather than specific objects within the data structure.

The potential is represented with a potential function, ϕ which maps a data structure to a real number. We start with the initial data structure D_0. For $i = 1, \ldots, n$, c_i is the actual cost and D_i is the data structure obtained from applying the ith operation to D_{i-1}. The amortised cost $\hat{c_i}$ is hence defined as

$$\hat{c_i} = c_i + \phi(D_i) - \phi(D_{i-1})$$

Hence the total amortized cost telescopes,

$$\sum_{i=1}^{n} \hat{c_i} = \sum_{i=1}^{n} c_i + \phi(D_0) - \phi(D_n)$$

We then need to define ϕ such that $\phi(D_0) \geq \phi(D_n)$ so that the total amortised cost is an upper bound on the total actual cost. We usually define $\phi(D_0) = 0$, so we just need to show that for all n

$$\phi(D_n) \geq 0$$

A general rule of thumb is that more costly data structures should have a higher potential function.

2 Union Find

A disjoint-set data structure maintains a collection $F = \{S_1, \dots S_k\}$ of **disjoint** dynamic sets. We identify each set by a representative $rep(S)$, which is some member in the set. We want to define the following operations:

- MAKE-SET(x) creates a new set whose only member and representative is x. (Note that as sets are disjoint, they can't be in any other set).
- UNION(x, y) unites the disjoint sets that contain x and y. The representative of this new set is any member of $S_x \cup S_y$, typically $rep(S_x)$ or $rep(S_y)$. Sets S_x and S_y are removed from F and $S_x \cup S_y$ is added to F.
- FIND-SET(x) returns a pointer to the representative of the set containing x

This algorithm is used in Kruskal's Algorithm for Minimum Spanning Trees

```
1: $A = \emptyset$
2: for $v \in V$
3:     MAKE-SET(v)
4: Sort E into increasing order by weight w
5: for $(u, v) \in E$
6:     if FIND-SET(u) $\neq$ FIND-SET(v)
7:         $A = A \cup \{(u, v)\}$
8:         UNION$(u, v)$
```

There are three ways of implementing union find.

2.1 Array Implementation

Suppose the elements are $1, \dots, n$ We maintain the representatives of each set in an array R.

- MAKE-SET(x) - $R[x] = x$, time = $O(1)$
- UNION(x, y) - scan R; if $R[i] = R[x], setR[i] := R[y]$, time = $O(n)$
- FIND-SET(x) - return R[x], time = $O(1)$

2.2 Linked List Implementation

In the linked list implementation, each set is represented by its own linked list. The object for each set has a *head*, pointing to the first element in the list and a *tail*, pointing to the last object. Each object in the list contains a member of a set, a pointer to the next item in the list and a pointer to the set object. The representative of the set is the first object in the list.

- MAKE-SET(x) - Just create the set objects etc, time = $O(1)$

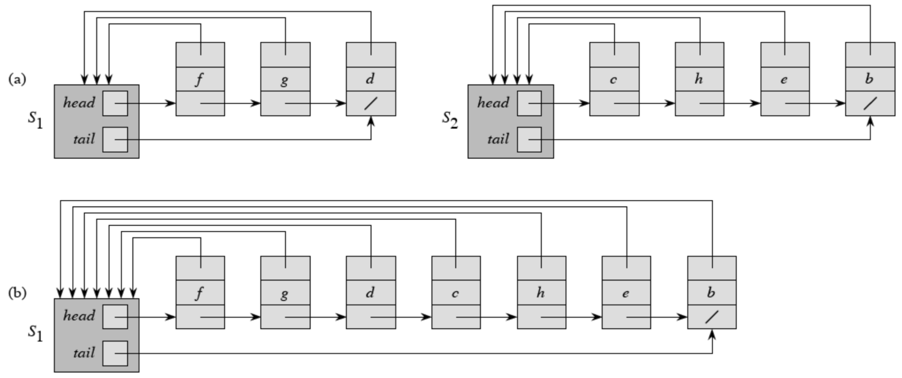

Figure 1: Before: S_1 and S_2; After $S_1 := S_1 \cup S_2$

- UNION(x, y) - We append the tail pointer for x's list to the head of y's list. We also need to change the set object pointer for each of the elements in y, time = $O(|S_y|)$
- FIND-SET(x) - return the follow the pointer to the set object, which points to the representative of the list, time = $O(1)$

We can find a heuristic for UNION, we should always append the smaller tree to the longer tree. We do this by keeping track of the height of each tree in the root. Since each head is updated at most $log(n)$ times, the time taken is $O(m + nlogn)$ for a sequence of m operations on n elements.

2.3 Disjoint Set Forest

In the disjoint set forest implementation, we represent sets by rooted trees, with each node containing one element and each tree representing one set. The root of each tree is the representative of each set (and is hence its own parent).

- MAKE-SET(x) - Create a tree with one node, time = $O(1)$
- UNION(x, y) - The root of x's tree points to the root of y's (or vice versa), time = $O(treeheight)$
- FIND-SET(x) - Follow parent pointers until we reach the root, time = $O(treeheight)$

On this implementation, this may be no faster than the linked-list implementation (we may just create a chain of n nodes). However using **union by rank** and **path compression**, we can get an almost linear runtime.

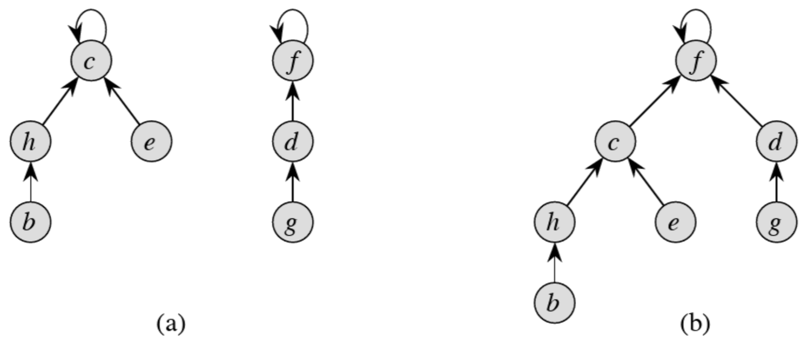

Figure 2: Before: S_1 and S_2; After $S_1 := S_1 \cup S_2$

For each node we maintain a *rank*, an upper bound on the height of the tree. We make the root with the smaller rank point to the root with the larger rank during a $UNION$ operation, incrementing the rank of the top root if the two ranks are the same.

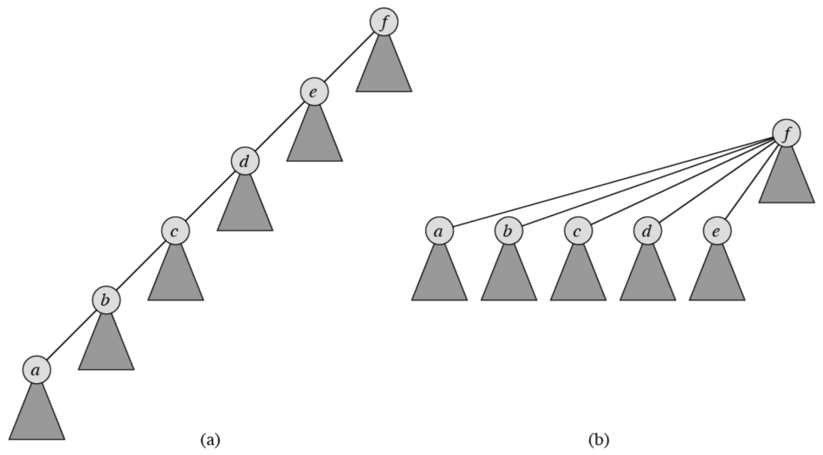

Figure 3: Before and after a FIND-SET operation

During any invocation of FIND-SET(x), all nodes found whilst traversing up the tree then have their parents redirected to the root (the representative). In this way we flatten long chains in trees.

Using both these heuristics, the runtime is now $O(m\ log^*n)$

3 Binary Trees

3.1 Binary Search Trees

A binary search tree is a dynamic data structure composed of nodes. Each nose has a value, $val[x]$, a left and right child, $left[x]$ and $right[x]$, and a parent, $p[x]$. If there is no child, then we set the respective pointer to NIL.

Binary search trees have the BST Property, for every node x in T,

- $val[x]$ is greater than all values in the left subtree of x
- $val[x]$ is less than all values in the right subtree of x.

BSTs support several operations:

- INSERT(T, z) - Inserts node z in T, time = $O(treeheight)$
- DELETE(x) - Deletes the node x from its tree.
  - If x has no children, then this is immediate
  - If x has one child, then we promote this child in place of x
  - If x has two children, find the SUCCESSOR of x, y. Swap x and y, and then delete x

  time = $O(treeheight)$
- SEARCH(T, v) - Returns x such that $val[x] - v$, if there is such a node, NIL otherwise, time = $O(treeheight)$
- MIN(T) - Returns $xwith$ minimum $val[x]$, time = $O(treeheight)$
- MAX(T) - Returns $xwith$ maximum $val[x]$, time = $O(treeheight)$
- PREDECESSOR - Returns y such that y has the largest such value smaller than $val[x]$, time = $O(treeheight)$
- SUCCESSOR(x) - Returns y such that y has the smallest such value greater than $val[x]$, time = $O(treeheight)$

A binary tree is balanced if its height is $O(log\ n)$, hence all operations on a binary tree take time $O(log\ n)$.

3.2 Rotations

Rotations can be used to transform one BST into another. They can be performed in $O(1)$ time. Note that rotations also preserve the BST property.

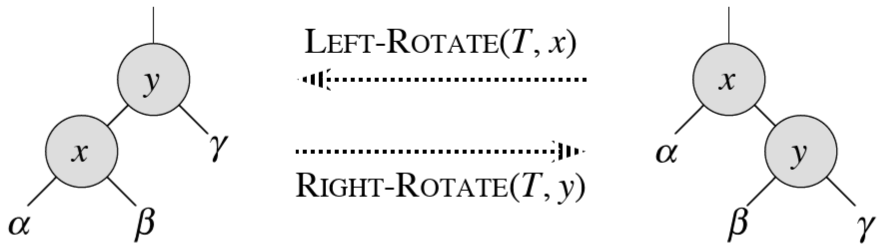

Figure 4: Left and Right rotations

3.3 Red-Black Trees

Red-Black trees are BSTs that also satisfy the following properties:

1. Every node is either red or black
2. The root is black
3. Every leaf (NIL) is black
4. If a node is red, then both its children are black
5. For each node all simple paths from the node to descendant leaves contain the same number of black nodes. We call this number the **black height** of x

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

This can be proven be showing that :

- height$(T) \leq 2bh(T)$
  - Consider an arbitrary path P from root to leaf
  - The number of red nodes in $P \leq$ the number of black nodes in P
- $bh(T) \leq log\ (n+1)$
  - This can be shown by induction on the height of the tee
  - We can also collapse every red node to its black parent, since every value-bearing node has ≥ 2 children and the height of the collapsed tree is $bh(T)$, then we know that $2^{bh(T)} \leq \#$ of $NILs$ in collapsed tree $= n+1$

Hence BST operations on a red-black tree take $O(log\ n)$ time.

3.3.1 Insertions

To insert into a red-black tree, we insert a red node into the tree. This ensures that the number of black nodes on a path is maintained. Note that this may mean that there are some red-red violations between the inserted node and its parent. To resolve this, we use rotations to move this violation upwards until it reaches the root.

There are three cases when resolving red-red violations:

Case 1 Uncle[x] is red

- In this case, we recolour the parent and uncle of x black, and the grandparent *red*.
- Note if the grandparent is the root, then we keep the root colour black.
- We also need to recurse on the grandparent of x as we may have created a new red-red violation

Case 2 Uncle[x] is black, the triple $(x, p[x], p[p[x]])$ is not aligned

- In this case, we rotate x with its parent
- This turns the case into case 3, we recurse on the old parent of x

Case 3 Uncle[x] is black, the triple $(x, p[x], p[p[x]])$ is aligned

- In this case, we rotate the parent of x with the grandparent of x.
- We colour the old grandparent of x red, and the old parent of x, black.

Since case 3 occurs at most one, case 2 occurs at most once, and case 1 occurs at most the height of the tree, it takes $O(log\ n)$ time to resolve red-red violations.

3.4 Statically Optimal Trees

Let $S = \{x_1, x_2, \ldots, x_n\}$ be the set of values in a BST tree. In a sequence of m searches, x_i is accessed mp_i times (so $p_1 + \ldots + p_n = 1$). The lookup cost of x_i is $1 + \text{depth}(x_i)$. A **statically optimal binary search tree** is the tree that has the minimum aggregate lookup cost among all BSTs on n items.

There is an $O(n^2)$ time dynamic programming algorithm for constructing statically optimal BSTs, and an $O(n\ log\ n)$ time algorithm that acheives a lookup cost of at most $\frac{3}{2}$ times the optimal lookup cost. Note that both of these methods require that the access probabilities are known in advance.

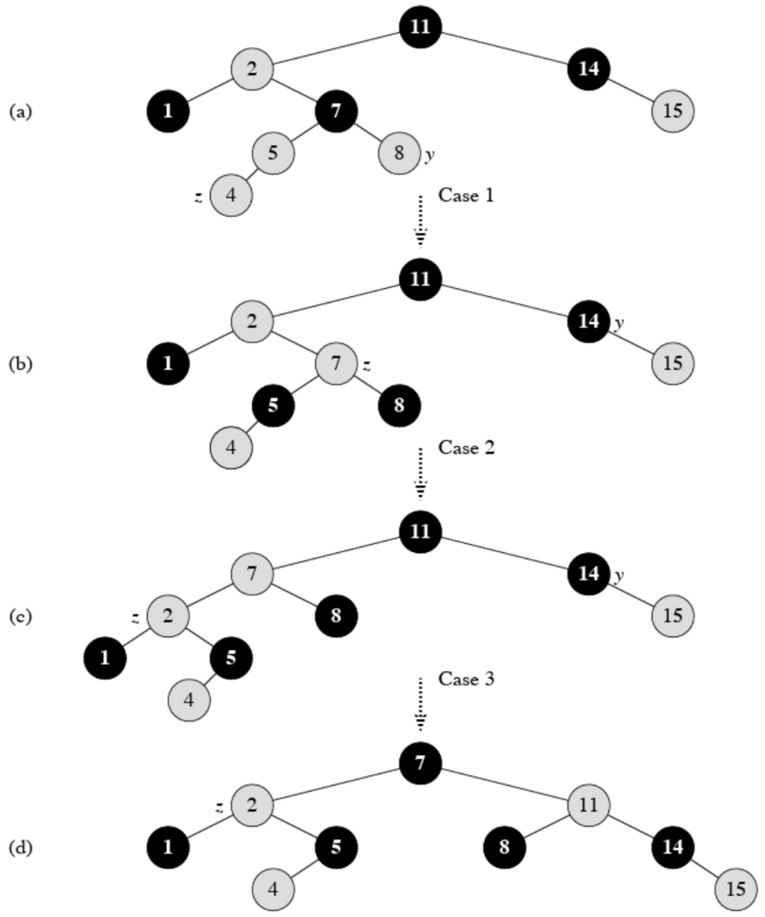

Figure 5: All three cases of a resolving red-red violations

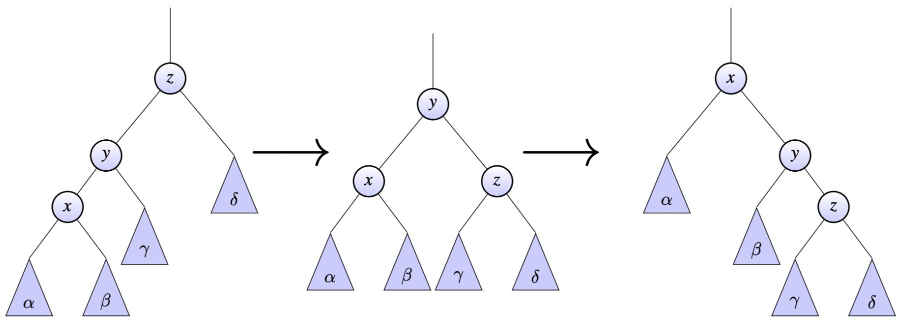

Figure 6: Zig-zig

3.5 Splay Trees

Splay trees are self-adjusting BSTs, that adapt to the access sequence. They do not store any additional data, and support BST operations in $O(log\ n)$ amortised time. The principle of splay trees is that when we access an item x, we move it to the root of the tree via rotations. In the long run, frequently accessed items are closer to the top of the tree, so that they don't take as much time to find.

The splay procedure, SPLAY(T, x) is as follows, recursing up until x is at the root

- If x has a parent y but no grandparent
  - rotate x with y
- If the triple $(x, p[x] = y, p[p[x]] = z)$ is aligned (zig-zig)
  - rotate y with z
  - rotate x with y
- If the triple $(x, p[x] = y, p[p[x]] = z)$ is not aligned (zig-zag)
  - rotate x with y
  - rotate x with z

We may also need to modify BST operations to support splaying a tree:

- SEARCH(T, x) - Perform SPLAY(T, x)
- INSERT(T, x) - Insert x into T as usual and then perform SPLAY(T, x)

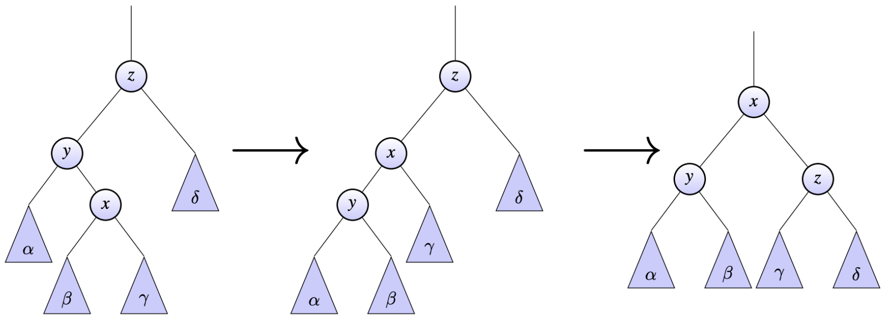

Figure 7: Zig-zag

- DELETE(T, x) - Perform SPLAY(T, x), then remove x from the root. This splits the tree into $T_{<x}$ and $T_{>x}$. Find $w = max(T_{<x})$, perform SPLAY$(T_{<x}, w)$, and then join $T_{>x}$ on w, which we can do as w has no right child.

The amortised cost of SPLAY is $Olog\ n$. This analysis can be performed by the potential function method. It can then be shown that for n INSERTS and m SEARCH/DELETE operations, the total cost is $O((m+n)logn)$.

4 Flow Networks

Def 2. A **flow network** is a directed graph $G = (V, E)$ in which each edge $(u, v) \in E$ has a non-negative capacity $c(u, v) \geq 0$. Furthermore if $(u, v) \in E$, then $(v, u) \notin E$. We distinguish two nodes in G, the **source** s and the **sink** t, such that for each vertex $v \in V$, the flwo network contains a path $s \rightsquigarrow v \rightsquigarrow t$

Def 3. A **flow** in G is a real-valued function $f : V \times V \to \mathbb{R}$ that satisfies

Capacity Constraint For all $u, v \in V$, we require

$$0 \leq f(u, v) \leq c(u, v)$$

Flow Constraint For all $u \in V - \{s, t\}$ we require

$$\sum_{v \in V} f(u, v) = \sum_{v \in V} f(u, v)$$

Note that when $(u, v) \notin E$, then $f(u, v) = 0$

The **value** of f is defined as

$$|f| = \sum_{e \text{ out of } s} f(e)$$

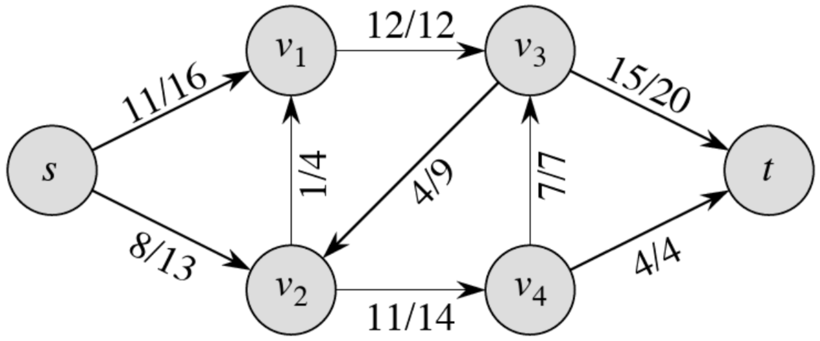

Figure 8: Example Flow Network

4.1 Minimum Cut

Def 4. An **s-t cut** is a partition of V into A, B such that $s \in A$ and $t \in B$.

The **capacity** of an s-t cut (A, B) is defined as

$$\text{cap}(A, B) = \sum_{e \text{ out of } A} c(e)$$

The minimum cut of a flow network is an s-t cut with minimum capacity.

Def 5. Weak Duality If (G, s, t, c) is a flow network, then

$$\text{value of max flow} \leq \text{capacity of min s-t cut}$$

i.e. for any flow f and any s-t cut (A, B)

$$|f| \leq \text{cap}(A, B)$$

We can prove weak duality by observing that due to flow conservation,

$$|f| = \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ into } A} f(e)$$

and hence

$$\begin{aligned} |f| &\leq \sum_{e \text{ out of } A} f(e) \\ &\leq \sum_{e \text{ out of } A} c(e) \\ &= c(A, B) \end{aligned}$$

As a corollary result, if f satisfies $|f| = \text{cap}(A, B)$, then f is a maximum flow. Hence to certify f is a maximum flow, it suffices to find a corresponding s-t cut.

4.2 Residual Graphs

Given a flow network G and a flow f, the residual network G_f consists of edges with capacities that represent how we can change the flow on edges of G. An edge of the flow network can admit an amount of additional flow equal to the edge's capacity minus the flow on that edge. Since a network may also decrease the flow on particular edges, we need backward edges on G_f with a capacity equal to the flow on the original edges, i.e. to build G_f,

- G_f has the same edges as G
- For each $e = (u, v)$ with flow $f(e)$ and capacity $c(e)$
  - If $f(e) < c(e)$ - add forward edge (u, v) with capacity $c_f((u, v)) = c(e) - f(e)$
  - If $f(e) > 0$ - add backward edge (v, u) with capacity $c_f((v, u)) = f(e)$

4.3 Augmenting Paths

Def 6. Given a flow network $G = (V, E)$ and a flow f, an **augmenting path** p is a simple path from s to t in the residual network G_f. By definition of the residual network, we can increase the flow on an edge in G, (u, v) by up to $c_f(u, v)$

We can also define the bottleneck capacity, b_P, of an augmenting path and a flow f in G as the the minimum residual capacity $c_f(e)$ over $e \in P$. We can therefore augment f by b_P units of flow.

AUGMENT(f, P) : construct flow f' on G For each $e \in E$:

- if $e \in P$ set $f'(e) = f_e + b_P$
- if $rev(e) \in P$ set $f'(e) = f_e - b_P$
- otherwise, set $f'(e) = f(e)$

If f is a flow in G and P is an augmenting path in G_f, then

- $f' =$ AUGMENT(f, P) is a valid flow in G
- $|f'| = |f| + b_P$

Hence calling AUGMENT on a graph always increases the flow.

Theorem 2. The **Max-Flow Min-Cut** theorem states that if f is a flow in G, with source s and sink t, then the following are equivalent:

1. f is a maximum flow in G
2. The residual network G_f has no augmenting paths
3. $|f| = \text{cap}(S, T)$ for some cut (S, T) of G

We can prove (1) $\rightarrow$ (2) by observing that if the residual graph did have an augmenting path P, then we can increase the flow in G by b_P, meaning that f can't be a maximum flow.

Suppose G_f has no augmenting paths, meaning we can split V into the nodes reachable from s, A and all other nodes, B. Clearly $s \in A$ and $t \in B$. Consider a pair of vertices $u \in A$, $v \in B$. If $(u,v) \in E$, then we must have $f(u,v) = c(u,v)$ as otherwise $(u,v) \in E_f$, meaning v would be reachable from s. Similarly, if $(v,u) \in E$, $f(v,u) = 0$, as otherwise $c_f(u,v) = f(v,u)$ would be positive, meaning $(u,v) \in E_f$, and hence v would be reachable from s. If neither (u,v) nor (v,u) is in E, then $f(u,v) = f(v,u) = 0$. Hence

$$\begin{aligned} f(A,B) &= \sum_{u\in A}\sum_{v\in B} f(u,v) - \sum_{v\in B}\sum_{u\in A} f(v,u) \\ &= \sum_{u\in A}\sum_{v\in B} c(u,v) - \sum_{v\in B}\sum_{u\in A} 0 \\ &= \mathrm{cap}(A,B) \end{aligned}$$

Since (A,B) is a cut of the network, $|f| = f(A,B) = c(A,B)$ thus proving (2) $\rightarrow$ (3).

We can finally prove (3) $\rightarrow$ (1) by weak duality.

4.4 Ford-Fulkerson

```
FORD-FULKERSON(G,s,t,c)
1: Set $f(e) = 0$ for all $e \in E$
2: $G_f$ := residual graph of $G$ wrt $f$
3: **while** there exists s-t path $P$ in $G_f$
4:     $f$ := AUGMENT$(f, P)$
5:     Update $G_f$
```

We also know that the Ford-Fulkerson algorithm returns a maximum flow as upon termination G_f has no s-t path, and is therefore disconnected. This means that we can split V into (A,B), where A are all the vertices reachable from s. (A,B) is a cut of G with $cap(A,B) = |f|$, which by weak duality implies f is a max-flow .

We can only be certain that this algorithm terminates when the capacities are integers. Since the flow increases by at least 1 at each iteration, the total time is $O(m|f^*|)$, where m is the number of edges in G, $|f^*|$ is the value of the maximum flow.

In order to improve the runtime of the Ford-Fulkerson Algorithm, we want to find the largest bottleneck capacity each time. A good heuristic is the capacity-scaling algorithm. This algorithm uses a parameter Δ which starts large and decreases by a factor of 2. We only consider a subgraph of G_f that contains edges with a capacity $\geq \Delta$. In this way, we increase the flow by at least half the max bottleneck capacity every while loop.

```
CAPACITY-SCALING($G, s, t, c$)
1: Set $f(e) = 0$ for all $e \in E$, $C = \max_{e \in E} c(e)$
2: $\Delta :=$ largest power of 2 less than or equal to C
3: **while** $\Delta \geq 1$
4:     $G_f :=$ residual graph of $G$ wrt $f$
5:     **while** there exists s-t path $P$ in $G_f$
6:         $f :=$ AUGMENT$(f, P)$
7:         Update $G_f$
8:     $\Delta := \Delta/2$
9: return $f$
```

Since every path in $G_f(\Delta)$ is a path in G_f, and $G_f(\Delta) = G_f$, we know that CAPACITY-SCALING returns a maximum flow.

Let $C =$ maximum capacity edge in G. The outer loop of the algorithm runs $1 + log\ C$ times.

Lemma 1. After the end of a Δ-scaling phase, $|f^*| \leq |f| + m\Delta$, where f^* is the maximum flow.

This is true because after the Δ-scaling phase, there is no s-t path in $G_f(\Delta)$, and we can therefore partition V into (A, B), where A is all the vertices reachable from s, $B = V \setminus A$. Hence

- For each $(u, v) \in E$ with $u \in A$, $v \in B$
  - $c(e) < f(e) + \Delta$
- For each $(u, v) \in E$ with $u \in B$, $v \in A$
  - $f(e) < \Delta$

as otherwise there would be an $s - t$ path in $G_f(\Delta)$. Therefore

$$\begin{aligned} |f| &= \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ into} A} f(e) \\ &\geq \sum_{e \text{ out of } A} (c(e) - \Delta) - \sum_{e \text{ into} A} \Delta \\ &\geq \text{cap}(A, B) - \Delta m \\ &\geq |f^*| - \Delta m \end{aligned}$$

Hence by this lemma, we can see that the inner loop is executed $\leq 2m$ times. This is because after the previous execution of the inner loop, by the lemma we have $|f^*| \leq |f| + 2\Delta m$ and each augmentation in a Δ-phase increases f by at least Δ.

Since the outer loop takes $O(1 + log\ C)$, the inner loop takes $O(m)$ and it takes $O(m)$ time to find an s-t path, the total runtime is $O(m^2(1 + log\ C))$

4.5 Edmonds-Karp

```
EDMONDS-KARP$(G, s, t, c)$
1: Set $f(e) = 0$ for all $e \in E$
2: $G_f$ := residual graph wrt $f$
3: while There exists s-t path in $G_f$
4:     $P$ := s-t path with fewest edges
5:     $f$ := AUGMENT$(f, P)$
6:     Update $G_f$
7: Return $f$
```

The idea behind that Edmonds-Karp Algorithm is that edges don't become bottlenecks too often.

We can show that the runtime of this algorithm is $O(nm^2)$ by examining two claims, (not proven here). Consider the ith time that the loop is executed. Let f_i be the flow at the beginning of the loop, G_i the residual graph wrt f_i and $d_i(u, v)$ be the distance between u and v in G_i

Claim 1 For any vertex $v \in V$, $d_i(s, v)$ is non-decreasing with i, i.e.

$$d_1(s, v) \leq d_2(s, v) \leq \ldots$$

Claim 2 If e is the bottleneck edge at time i, it will not be used in an augmenting path again until $d_i(s, t)$ increases

Using these two claims, we can see that the distance between s and t can increase at most n times. For each value $d = distance(s, t)$ in G_f, an edge becomes a bottleneck at most once, so there are at most m augmentations for $d \in [1, n]$. Since the time taken to find P uses BFS, which is $O(m)$, the total time taken is $O(nm^2)$.

4.6 Bipartite Matchings

Def 7. Given an undirected graph $G = (V, E)$ a **matching** is a subset of edges $M \subseteq E$ such that for all $v \in V$, at most one edge of M is incident on v. If an edge is incident on v in a matching M, then this vertex is **matched**, otherwise it is **unmatched**. A **maximum matching** is a matching of maximum cardinality.

Given an instance $G = (L \cup R, E)$ of a **bipartite** matching problem, we can create an instance $G' = (V', E')$ of MAXFLOW by adding a source node s that connects to every node in L, a sink node that is connected to every node in R, and by setting all edge capacities to 1. G has a maximal matching equal to k

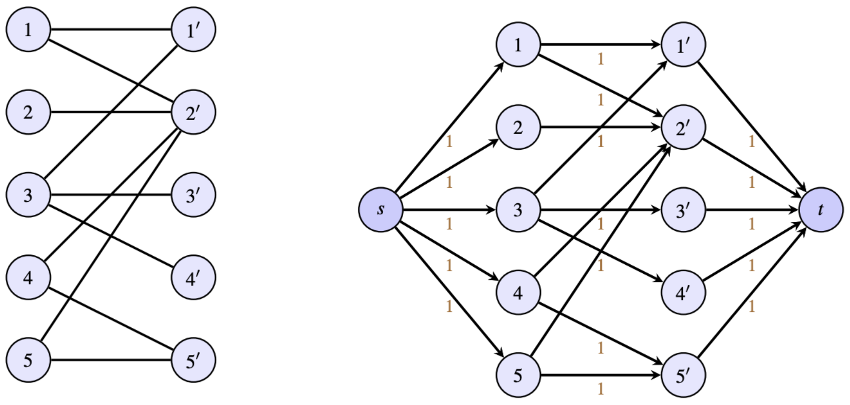

Figure 9: Constructing a flow network from a bipartite graph

iff the max flow on G' is equal to k.

A matching is called **perfect** if every vertex is matched. We can use this algorithm to find the perfect matching, if it exists. A certificate for when a graph G does not have a perfect matching is when there is a set of vertices in L that has a number greater than that of the set of neighbouring vertices, i.e. there is some $S \in L$ such that

$$|N(S)| \leq |S|$$

However, the converse statement also holds

Theorem 3 (Hall's Theorem)**.** A bipartite graph $G = (L \cup R, E)$ has a perfect matching iff for every set $S \subseteq L$ it holds that $|N(S)| \geq |S|$

To prove the converse direction, suppose that for every $S \subseteq L$, $|N(S)| \geq |S|$. Consider the constructed flow network G', with an arbitrary s-t cut (A, B). Let $A_L = A \cap L$, $A_R = |A \cap R|$. Hence

$$\begin{aligned}
\text{cap}(A, B) &= \sum_{e \text{ out of } A} \\
&\geq |L \setminus A_L| + |N(A_L) \setminus A_R| + |A_R| \\
&\geq |L| - |A_L| + |N(A_L)| - |A_R| + |A_R| \\
&\geq |L| - |A_L| + |A_L| - |A_R| + |A_R| && \text{(by assumption)} \\
&= |L| \\
&= n
\end{aligned}$$

By the max-flow min-cut theorem, since the minimum capacity is at least n, then the max flow is n, hence G' has a maximum flow, and so G has a perfect matching.

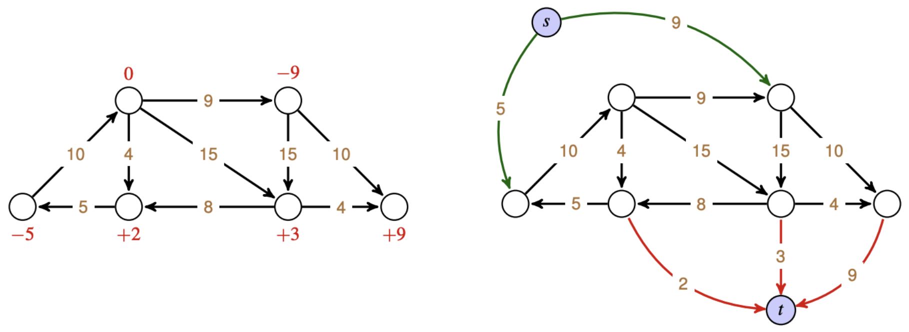

Figure 10: Constructing a flow network from a circulation instance

4.7 Circulations

Def 8. Given a directed graph $G = (V, E)$, capacity bounds $c : E \to \mathbb{Z}^+$ and a demand function $d : V \to \mathbb{Z}$ the **circulation problem** is to find a **circulation** $f : E \to \mathbb{R}^+$ such that for each edge $e \in E$

$$0 \leq f(e) \leq c(e) \qquad \text{(Capacity Constraint)}$$

and for each $v \in V$

$$\sum_{e \text{ into } v} f(e) - \sum_{e \text{ out of } v} f(e) = d(v) \qquad \text{(Flow Conservation)}$$

Note that for any circulation, by flow conservation, we must have that

$$\sum_{v \,:\, d(v)>0} d(v) = \sum_{v \,:\, d(v)<0} |d(v)|$$

Given a circulation instance (G, c, d), we can construct a flow network (G', s, t, c) by

- Adding a source s and sink t
- For each v with $d(v) < 0$, add edge (s, v) with capacity $-d(v)$
- For each v with $d(v) > 0$, add edge (v, t) with capacity $d(v)$

We can see that G has a circulation iff G' has max flow equal to

$$D = \sum_{v \,:\, d(v)>0} d(v) = \sum_{v \,:\, d(v)<0} |d(v)|$$

In this way, the sink and source nodes mimic the demand functions of the circulation.

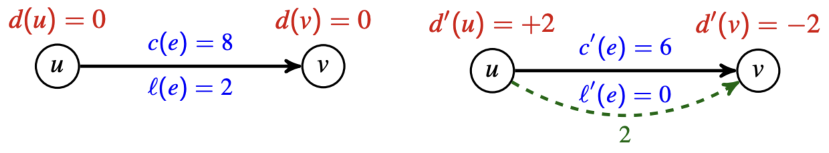

Figure 11: Removing lower bound constraints

Suppose instead that in addition t capacity bounds, we also have s lower bounds for the circulation, $l : E \to \mathbb{Z}^+$. We can still format this as a network flow problem. We do this by constructing the circulation as before, and for each edge $e = (u, v)$ with $l(e) > 0$

- $d'(u) := d(u) + l(e)$
- $d'(v) := d'(v) - l(e)$
- $c(e) := c(e) - l(e)$
- $l(e) := 0$

Meaning that we act as if these lower bounds are being satisfied, with the capacities and the demand of v being lowered and the demand of u being raised.

5 Minimum Cost Perfect Matchings

Given an undirected bipartite graph $G = (X \cup Y, E)$, and a cost function $c : E \to \mathbb{R}^+$, how can we find a perfect matching M such that $c(M)$ is minimised? This algorithm can't be achieved using network flows, as network flows encode **constraints** for problems, however this problem deals with **preferences**.

The structure of the algorithm assumes at any stage there is a matching of size i. We then find an augmenting path that will produce a matching of size $i + 1$. We use the cheapest augmenting path at each stage so that the larger matching will have the minimum cost.

Recall the construction of the residual graph used for finding augmenting paths, let M be a matching. We also add the source s and the sink t nodes. We add edges (s, x) for all nodes $x \in X$ that are unmatched and edges (y, t) for all nodes $y \in Y$ that are unmatched. This residual graph is denoted by G_M. Note that wrt to a flow f,

- If $(x, y) \in M$, then $(y, x) \in G_M$ (as we can reduce this flow to 0)
- If $(x, y) \notin M$, then $(x, y) \in G_M$ (as we can increase this flow).

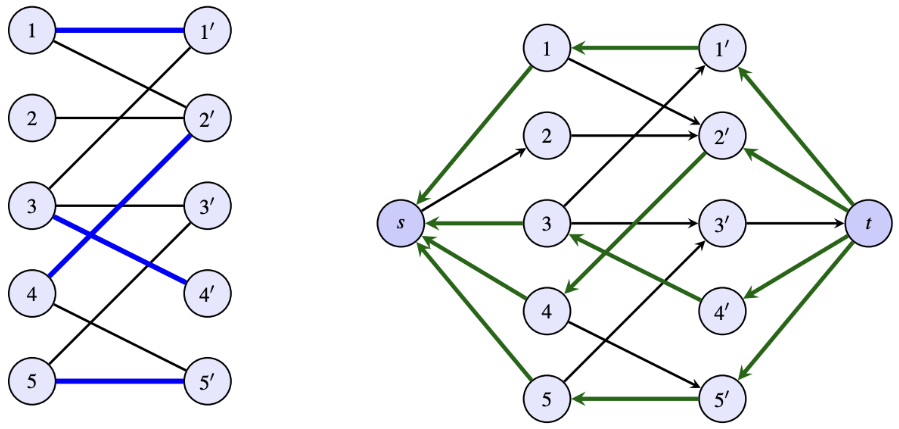

Figure 12: Residual graph of a matching

Note that any augmenting path in G_M starts at s and alternates between matched and unmatched edges. When we start at s, we go to an unmatched node. We must then use an unmatched edge, as the only edges that go from X to Y in G_M are those where $(x, y) \notin M$. Likewise, we must then use an matched edge to go from Y to X. Finally we go from an unmatched vertex to t. Hence when we take the symmetric difference of any matching M and an augmenting path P, we must result in a matching $M' = M \oplus P$ that has one more matching than M. This is because we can think of the action of using an edge from X to Y as being equivalent to including it in the new matching M', and using an edge from Y to X as being equivalent to discounting it from M'. Since we go from X to Y once more than we go from Y to X, we must have increased the number of edges by 1.

The cost of this alternating path is hence each $c(x, y)$ as we add edges (x, y) to M minus each $c(y, x)$ as we remove these edges from M. We also define the cost of each edge in M to reflect this:

- If $(x, y) \in M$ then cost of (x, y) in G_M is $-c(x, y)$
- If $(x, y) \notin M$ then cost of (x, y) in G_M is $c(x, y)$
- All edges from s or into t have cost 0

Hence on this definition,

$$c(M') = c(M) + c(P)$$

We can see that this algorithm may fail to produce a well-defined output when there exist negative cycles in the residual graph G_M.

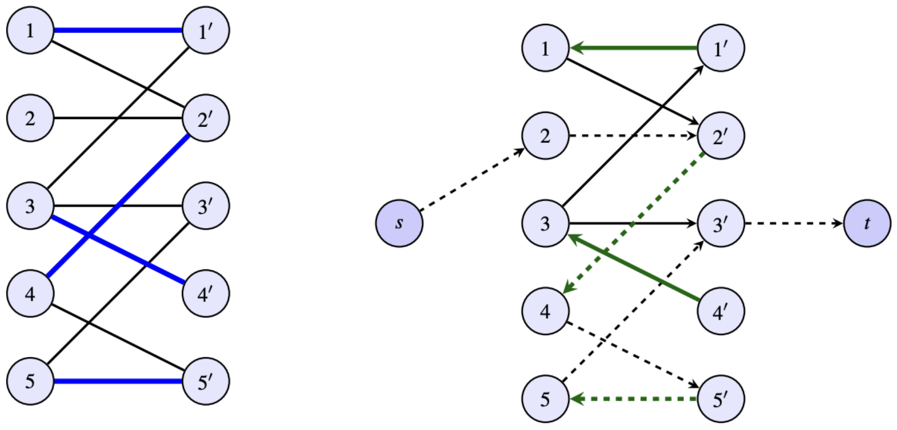

Figure 13: An augmenting path alternates between unmatched and matching nodes

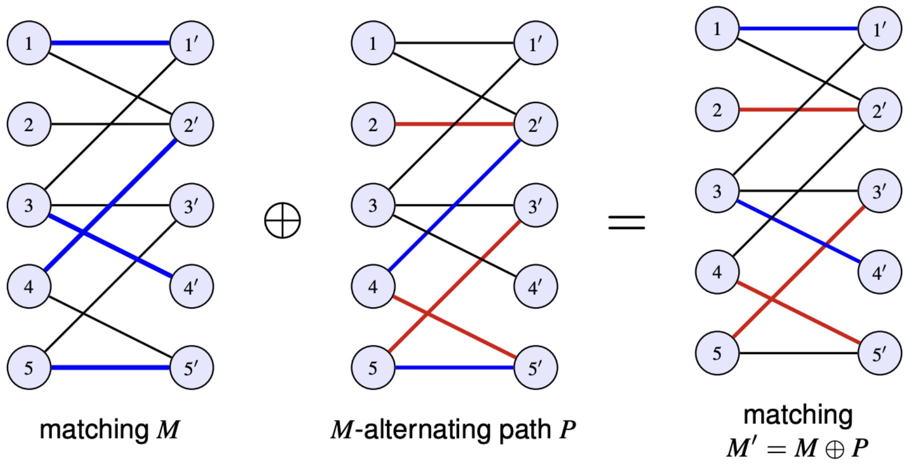

Figure 14: Constructing a new matching of a larger size

6 Linear Programming

Def 9. A **linear program** is an optimisation problem over real-valued variables. We maximise or minimise a linear **objective** function subject to linear **constraints**. Note that these constraints are $\leq, \geq, =$, but not $>, <$

$$\begin{aligned} \text{maximise} \quad & 2x + 5y \\ \text{subject to} \quad & 3x + 5y \leq 15 \\ & x + y \leq 4 \\ & -x - 10y \geq -20 \\ & x, y \geq 0 \end{aligned}$$

A linear program is **feasible** if there is a solution satisfying all the constraints. It is **bounded** if the optimal value is finite.

Theorem 4 (Fundamental Theorem of Linear Programming)**.** For an LP, exactly one holds:

- The LP is infeasible
- The LP is feasible but unbounded
- The LP is bounded, feasible and there is a solution that achieves OPT, the optimum value of the objective function.

To solve LPs, we can consider the feasible set, all the points that satisfy the constraints. The optimal value must then be at an intersection point, hence if we find the values of the objective function at these extreme points, we can then find the optimal value for the objective function.

6.1 Dual Linear Programs

Def 10. The **dual** of a linear program (called the **primal**) is a closely related LP whose feasible and optimal solutions provide information about the primal LP.

For example, if the primal LP is a maximisation problem, then the dual gives upper bounds on the optimal value (and similarly lower bounds on minimisation problems).

As we have seen, a general LP consists of n real variable, an objective function and several constraints of the form $\geq, \leq=$. In order to write the dual of a linear program, we must first write the linear program in standard max form. This consists of

- Replacing all min functions with max functions
  - min $x_1 - 2x_2$ becomes

  - max $-x_1 + 2x_2$

- Replacing all $=$ constraints with $\leq$ *and* $\geq$ constraints
  - $x + y = 1$ becomes
  - $x + y \geq 1$, $x + y \leq 1$
- Replacing all $\geq$ constraints with $\leq$ constraints
  - $x + y \geq 1$ becomes
  - $-x - y \leq -1$
- Making all variables non-negative by writing them as the difference between a non-negative variable and a non-positive variable
  - $x \leq 1$ becomes
  - $x^+ - x^- \leq 1$ and $x^+, x^- \geq 0$

We now have a primal LP of the form

$$\begin{array}{rll} \text{maximise} & c_1x_1 + c_2x_2 + \ldots + c_nx_n & \\ \text{subject to} & a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n & \leq b_1 \\ & a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n & \leq b_2 \\ & & \vdots \\ & a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n & \leq b_n \\ & x_1, x_2, \ldots, x_n & \geq 0 \end{array}$$

We then construct the dual by

- Multiplying the ith constraint with $y_i \geq 0$
  - $y_i(\sum_{j=1}^n a_{ij}x_j) \leq b_iy_i$
- Summing these new inequalities to obtain $L \leq$ BOUND
  - BOUND $= \sum_{i=1}^m b_iy_i$
  - $L = \sum_{i=1}^m y_i(\sum_{j=1}^n a_{ij}x_j) = \sum_{j=1}^n x_j \sum_{i=1}^m a_{ij}y_i$
- Ensuring that the objective function $\leq L$ by imposing constraints on each y_i
  - $c_j \leq \sum_{i=1}^m a_{ij}y_i$ for each $j = 1, \ldots, n$
- Optimising each y_i to find the minimum value of BOUND.
  - min $\sum_{i=1}^m b_iy_i$

Hence the dual LP can now be written as

$$\begin{array}{rll}
\text{minimise} & b_1y_1 + b_2y_2 + \ldots + b_my_m & \\
\text{subject to} & a_{11}y_1 + a_{21}y_2 + \ldots + a_{m1}y_m & \geq c_1 \\
& a_{12}y_1 + a_{22}y_2 + \ldots + a_{m2}y_m & \geq c_2 \\
& & \vdots \\
& a_{1n}x_1 + a_{2n}x_2 + \ldots + a_{mn}y_m & \geq c_n \\
& y_1, y_2, \ldots, y_m & \geq 0
\end{array}$$

Note that the b_i's and the c_i's have switched, the rows of the a_{ij}'s have become columns and instead of n variables and m constraints, there are m variables and n constraints. This allows us to capture (standard) LPs in terms of matrices. The primal LP has now become

$$\begin{array}{rll}
\text{maximise} & \mathbf{c}^T\mathbf{x} & \\
\text{subject to} & A\mathbf{x} & \leq \mathbf{b} \\
& \mathbf{x} & \geq 0
\end{array}$$

And the dual now becomes:

$$\begin{array}{rll}
\text{minimise} & \mathbf{b}^T\mathbf{y} & \\
\text{subject to} & A^T\mathbf{y} & \geq \mathbf{c} \\
& \mathbf{y} & \geq 0
\end{array}$$

Theorem 5 (Weak LP Duality)**.** Let $\mathbf{x}$ be in the feasible region of the primal LP and $\mathbf{y}$ be in the feasible region of the dual LP. Then

$$\mathbf{c}^T\mathbf{x} \leq \mathbf{b}^T\mathbf{y}$$

We can see this is true as

$$\begin{aligned}
\mathbf{c}^T\mathbf{x} &= \mathbf{x}^T\mathbf{c} \\
&\leq \mathbf{x}^TA^T\mathbf{y} \\
&= (A\mathbf{x})^T\mathbf{y} \\
&\leq \mathbf{b}^T\mathbf{y}
\end{aligned}$$

Hence if $\mathbf{c}^T\mathbf{x} = \mathbf{b}^T\mathbf{y}$, then these objective functions must have reached their optimum.

Theorem 6 (Strong LP Duality)**.** Suppose either of the primal or dual LP is bounded and feasible. Then both of them are, and the optimum value of the primal is equal to the optimal value of the dual

6.2 Solving LPs

To solve the LP

$$\begin{array}{rll} \text{maximise} & \mathbf{c}^T\mathbf{x} & \\ \text{subject to} & A\mathbf{x} & \leq \mathbf{b} \\ & \mathbf{x} & \geq 0 \end{array}$$

- Enumerate all vertices of the polyhedron $P = \{\mathbf{x} \in \mathbb{R}^n \mid A\mathbf{x} \leq \mathbf{b}\}$
- Find the vertex with the maximum value of the objective function
- Check that the LP is bounded
  - Use the dual LP and check that it is feasible

6.3 Applications of LPs

6.3.1 Max Flows

We can formulate flow networks as LPs, as we are trying to maximise the flow on each edge whilst being constrained by the capacities on each edge.

Given a flow network $G = (V, E, s, t)$ and a capacity function $c : E \to \mathbb{Z}^+$ create the LP:

$$\begin{array}{rlll} \text{maximise} & \displaystyle\sum_{e \text{ out of } s} f_e & & \\ \text{subject to} & f_e & \geq 0 & \text{(for each } e \in E) \\ & f_e & \leq c(e) & \text{(for each } e \in E) \\ & \displaystyle\sum_{e \text{ out of } v} f_e - \sum_{e \text{ into } v} f_e & = 0 & \end{array}$$

where f_e is the flow on each edge in the network.

6.3.2 Circulations

We can also formulate circulation, however as this is a feasibility problem instead of an optimisation problem, there is no objective function. However, this may be a problem as we need to find a starting point for the algorithms available to solve LPs. Hence we introduce nonnegative slack variables for the demands at each of the vertices, if the objective function at its minimum manages to be 0, then there is a circulation

$$\begin{array}{lllr} \text{minimise} & \displaystyle\sum_{v\in V} q_v^+ + q_v^- & & \\ \text{subject to} & l(e) & \leq f_e & \text{(for each } e \in E) \\ & f_e & \leq c(e) & \text{(for each } e \in E) \\ & \displaystyle\sum_{e \text{ out of } v} f_e - \sum_{e \text{ into } v} f_e & = d(v) + q_v^+ - q_v^- & \text{(for each } v \in V) \\ & q_v^+, q_v^- \geq 0 & & \text{(for each } v \in V) \end{array}$$

Now to find the initial starting point, we set $f_e = l(e)$ for all $e \in E$. Let A_v be the excess flow for each node, i.e.

$$A_v = \sum_{e \text{ into } v} f_e - \sum_{e \text{ out of } v} f_e$$

If $A_v \geq 0$, set $(q_v^+, q_v^-) = (A_v, 0)$, otherwise set $(q_v^+, q_v^-) = (0, -A_v)$

6.3.3 Min-Cost Circulation

We can also formulate min-cost circulations more directly, given $G = (V, E)$ a demand function $d : V \to \mathbb{Z}$, upper and lower capacity bounds $c, l : E \to \mathbb{Z}$ and a cost function $k : E \to \mathbb{Z}$, we can form the LP

$$\begin{array}{lllr} \text{minimise} & \displaystyle\sum_{e\in E} k(e) f_e & & \\ \text{subject to} & l(e) & \leq f_e & \text{(for each } e \in E) \\ & f_e & \leq c(e) & \text{(for each } e \in E) \\ & \displaystyle\sum_{e \text{ out of } v} f_e - \sum_{e \text{ into } v} f_e & = d(v) & \text{(for each } v \in V) \end{array}$$

which will output the minimum cost circulation $\{f_e\}_{e\in E}$ with min cost $\sum_{e\in E} k(e) f_e$

6.3.4 Min-Cost Perfect Matching

Given $G = (L \cup R, E)$ and a cost function $c : E \to \mathbb{Z}$, we can form an LP by setting variables x_e to be 1 if e belongs to the matching, and 0 otherwise.

$$\begin{array}{lllr} \text{minimise} & \displaystyle\sum_{e\in E} c(e) x_e & & \\ \text{subject to} & \displaystyle\sum_{e \text{ incident to } v} x_e & = 1 & \text{(for each } v \in L \cup R) \\ & x_e & \in \{0, 1\} & \text{(for each } e \in E) \end{array}$$

Note that this is an instance of an **Integer Linear Program** (ILP).

6.3.5 Bipartite Perfect Matching

Bipartite perfect matching can be given as an ILP, but another solution is to relax the constraints for the LP so that they may be non-integral.

Given $G = (V, E)$ and cost function $c : E \to \mathbb{Z}$ the LP can be written as

$$\begin{aligned} \text{minimise} \quad & \sum_{e \in E} c(e) x_e & \\ \text{subject to} \quad & \sum_{e \text{ incident to } v} x_e = 1 & \text{(for each } v \in V) \\ & x_e & \\ geq0 & & \text{(for each } e \in E) \end{aligned}$$

For bipartite graphs this will always give an integer solution (the same is not true for general graphs).

To prove this claim, suppose for the sake of contradiction that $E' = \{e \mid 0 < x_e < 1\}$, for an extreme point of the LP, x, is non-empty. By the constraints $\sum_{e \text{ incident to } v} x_e = 1$ and $x_e \geq 0$, we know that every vertex v has a degree of either 0 or ≥ 2. Hence we know that G has a cycle of even length (as G is bipartite).

Call the edges of the cycle $C = \{e_1, e_2, \ldots, e_{2k}\}$. Define $\epsilon = \min\{x_{e_1}, x_{e_2}, \ldots, x_{e_{2k}}\}$, i.e. the smallest edge in the cycle. We can then define y by

$$y_e = \begin{cases} x_{e_i} + \epsilon & \text{if } e = e_i \in C \text{ and } i \text{ is even} \\ x_{e_i} - \epsilon & \text{if } e = e_i \in C \text{ and } i \text{ is odd} \\ x_e & \text{otherwise} \end{cases}$$

and similarly z by

$$z_e = \begin{cases} x_{e_i} - \epsilon & \text{if } e = e_i \in C \text{ and } i \text{ is even} \\ x_{e_i} + \epsilon & \text{if } e = e_i \in C \text{ and } i \text{ is odd} \\ x_e & \text{otherwise} \end{cases}$$

We have that $x = \frac{1}{2}(y + z)$, but y and z are still feasible, contradicting the assumption that x is an extreme point.

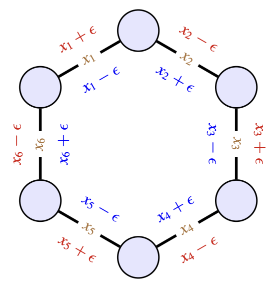

Figure 15: We can reduce and increase by ϵ in both directions without violating the constraint $x_e \geq 0$

6.4 Zero-Sum Games

7 NP Hardness

Def 11. A **decision problem** is a problem where the answer is either YES or NO

Def 12. P is the class of decision problems such that for each decision problem Π in P there is a polynomial-time algorithm A such that for every instance x of Π

- if $\Pi(x) = \text{YES}$, then $A(x) = \text{YES}$
- if $\Pi(x) = \text{NO}$, then $A(x) = \text{NO}$

Def 13. NP is the class of decision problems such that for each decision problem Π in NP there is a polynomial-time algorithm A and a constant $c > 0$ such that for every instance x of Π

- if $\Pi(x) = \text{YES}$, then there is a certificate y with $|y| \leq |x|^c$ such that $A(x, y) = \text{YES}$
- if $\Pi(x) = \text{NO}$, then for all certificates y, $A(x, y) = \text{NO}$

A decision problem in NP can hence be solved by non-deterministically guessing all certificates y with $|y| \leq |x|^c$ and returning whether any of these certificates give $A(x, y) = \text{YES}$.

7.1 Reductions

Def 14. A **reduction** from a problem A to problem B is a polynomial time algorithm that takes an instance x of A and produces an instance $y = f(x)$ of B such that

- $|y|$ is at most a polynomial in $|x|$

- $A(x) = \text{YES}$ iff $B(y) = \text{YES}$

If f exists, then we write $A \leq B$.

Intuitively this means that if B can be decided in polynomial-time, then A can also be solved in polynomial-time

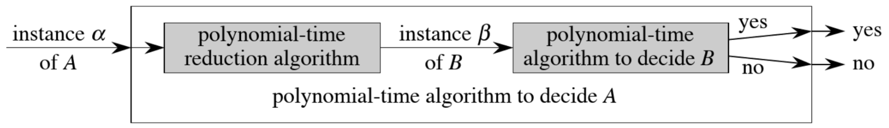
instance α of A
polynomial-time reduction algorithm
instance β of B
polynomial-time algorithm to decide B
yes
no
yes
no
polynomial-time algorithm to decide A

Figure 16: If B can be decided in polynomial-time, then so can A, $A \leq B$

Additionally, **Karp reductions** are reductions that use only **one** call to B, whereas **Turing reductions** allow for **multiple** calls to B.

Reductions are also transitive, if $A \leq B$ and $B \leq C$, then $A \leq C$.

8 Approximation Algorithms

9 Fixed Parameter Algorithms