

# Algorithms and Programming Techniques

## COMP3121 UNSW

Hussain Nawaz  
hussain.nwz000@gmail.com

2022T2

## Contents

<b>1</b>	<b>Introduction - Revision</b>	<b>2</b>
1.1	Rates of Growth . . . . .	2
1.2	Assumed Data Structures . . . . .	3
1.3	Algorithms . . . . .	4
1.4	Non-Comparison Sort . . . . .	5
1.5	Graphs . . . . .	6
<b>2</b>	<b>Algorithm Analysis</b>	<b>6</b>
2.1	Stable Matching Problem . . . . .	6
<b>3</b>	<b>Divide-and-Conquer Method</b>	<b>7</b>
3.1	Counting Inversions . . . . .	7
3.2	The Master Theorem . . . . .	7
3.3	Integer Multiplication and Addition . . . . .	8
<b>4</b>	<b>The Greedy Method</b>	<b>8</b>
4.1	When greed pays off - foundations of the Greedy Method . . . . .	8
4.2	Activity Selection problem . . . . .	8
4.3	Job Lateness . . . . .	9
4.4	Huffman Codes . . . . .	9
4.5	Tsunami Warning . . . . .	9
4.6	Minimum Spanning Trees . . . . .	12
<b>5</b>	<b>Network Flow Algorithms</b>	<b>13</b>
5.1	Flow networks . . . . .	13
5.2	Ford – Fulkerson Algorithm . . . . .	14
5.3	Speeding Up Max Flow . . . . .	15
5.4	Strategies for Alternative Max Flow Problems . . . . .	16
5.5	Max Flow Application: Movie Rental . . . . .	16

# 1 Introduction - Revision

## 1.1 Rates of Growth

**The Problem** To analyse algorithms, we need a way to compare two functions representing the runtime of each algorithm. However, comparing values directly presents a few issues.

- Outliers may affect the results.
- One algorithm may be slower for a set period of time, but catch up after a while.
- The runtime of an algorithm may vary depending on the implementation or the architecture of the machine where, some instructions are faster than others.

**Asymptotic Growth** For algorithms, we often prefer to refer to them in terms of their asymptotic growth in runtime, with relation to the input size.

A function that quadruples with every extra input will always have a greater runtime than one that increases linearly with each new input, for some large enough input size.

**Big O Notation** We say  $f(n) = O(g(n))$  if there exists a positive constants  $C, N$  such that

$$0 \leq f(n) \leq Cg(n) \quad \forall n \geq N.$$

We may refer to  $g(n)$  to be the asymptotic upper bound for  $f(n)$ .

**Big Omega Notation** We say  $f(n) = \Omega(g(n))$  if there exists positive constants  $c, N$  such that

$$0 \leq cg(n) \leq f(n) \quad \forall n \geq N.$$

Then,  $g(n)$  is said to be an asymptotic lower bound for  $f(n)$ . It is useful to say that a problem is at least  $\Omega(g(n))$ .

**Landau Notation**  $f(n) = \Omega(g(n))$  if and only if  $g(n) = O(f(n))$ .

There are strict version of Big *O* and Big *Omega* notations; these are little *o* and little  $\omega$  respectively.

We say  $f(n) = \Theta(g(n))$  if

$$f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n)).$$

That is, both  $f$  and  $g$  have the same asymptotic growth.

**Logarithms** Logarithms are defined so that for  $a, b > 0$  where  $n \neq 1$ , let

$$n = \log_a b \Leftrightarrow a^n = b.$$

They have the following properties:

- $a^{\log_a n} = n$

- $\log_a(mn) = \log_a(m) + \log_a(n)$
- $\log_a(n^k) = k \log_a(n)$

By the change of base,

$$\log_a(x) = \frac{\log_b(x)}{\log_b(a)}.$$

As such, the denominator is constant in terms of  $x$  and so all log bases are equivalent under Big Theta notation.

## 1.2 Assumed Data Structures

**Arrays** We assume static arrays (though, it is possible to extend to dynamic arrays).

- We assume random-access in  $O(1)$
- Insert / delete  $O(n)$
- Search:  $O(n)$  -  $\log n$  if sorted

**Linked Lists** We assume the linked lists are doubly-linked since the  $2\times$  overhead is negligible.

- Accessing next / previous:  $O(1)$
- Insert / delete to head or tail:  $O(1)$
- Search:  $O(1)$ .

**Stacks** Last in, first out.

- Accessing top:  $O(1)$
- Insert / delete from top:  $O(1)$

**Queue** First in, first out.

- Access front:  $O(1)$
- Insert front:  $O(1)$
- Delete front:  $O(1)$

**Hash Tables** Store values by their hashed keys. Ideally, no two keys will hash to the same value, however this may not be guaranteed.

- Search is expected to be  $O(1)$  however, in the worst case, we expect to have to search through all the values in  $O(n)$ .
- Insertion is expected to be  $O(1)$  however, in the worst case, we expect to have to search through all the values in  $O(n)$ .
- Deletion follows the same pattern of  $O(1)$  expectation and  $O(n)$  worst case.

**Binary Search Trees** We store (comparable) keys (or key-value pairs) in a binary tree, where each node has at most two children, the left and right. The value of a node must be greater than the values of all its children in its left sub-tree and greater than those in the right sub-tree.

- In the best case, the height of the tree is  $h = \log_2 n$ . Such a tree is *balanced*. in the worst case however, the tree is a long chain of height  $n$ .
- The average search is  $\log n$ . If the tree is self-balancing then search and other operations are guaranteed to be  $\log n$ .

**Binary Heap** A max-heap is such that the value of a node is greater than or, equal to the value of all its children. A min-heap follows the same principle but in reverse.

- Finding the maximum involves finding the top item in  $O(1)$
- Deleting the top item will also require re-balancing.
- Re-balancing the heap requires  $\log n$  time.
- Insertion also requires  $\log n$  time.

## 1.3 Algorithms

**Linear Search** Given an array  $A$  of  $n$  integers. We may determine if a value is inside  $A$ , by searching through the array linearly. This occurs in  $O(n)$ .

**Sorted Arrays - Binary Search** Given a sorted array  $A$  of  $n$  integers that are sorted by value. We may determine if a value is inside  $A$ , by binary search. We pick the midpoint of the  $A$ . If the midpoint  $m$  is the value desired, we return. Otherwise, we continue to recursively search through the array by halving the search space. If the  $m$  is greater than the value we desire, then we search the right sub-array of the array; otherwise, we search the left sub-array of the array

**Decision Problems and Optimisation Problems** Decision problems are of the form

Given some parameters  $X$ , can you ...

Optimisation problems are of the form

What is the smallest  $X$  for which you can ...

**Comparison Sorting** We are given an array of  $n$  items. We may sort the array in  $O(n \log n)$  at best.

**Asymptotic Lower Bound on Comparison Sorting** There are  $n!$  permutations of the array, of which there is 1 correct sort. If we do  $k$  comparisons, then we are able to check  $2^k$  permutations of results from these comparisons and so, can at most, distinguish  $2^k$  permutation.

Thus, we need to perform comparisons such that  $2^k \geq n!$ . That is,  $k \geq \log_2(n!)$ . Therefore,

$$k \geq \frac{n}{2} \log_2 \left( \frac{n}{2} \right)$$

## 1.4 Non-Comparison Sort

**Counting Sort** Create another array  $B$  of size  $k$  where  $k$  is the maximum size of an element. We iterate through the array and count the number of times each element occurs, iterating the index of  $B$  that is equivalent to the value of our current element.

The time complexity is  $O(n + k)$  where  $O(k)$  space is required.

**Bucket Sort** Distribute the items into buckets  $1, \dots, k$  such that each bucket contains a range of items such that all items in bucket 1 are less than all the items in bucket 2 and so on. We may sort within that bucket with any algorithm and then concatenate all the buckets together.

**Radix Sort** We assume an array  $A$  of  $n$  keys, of  $k$  symbols.

We bucket the keys by their first symbol. Then, within each bucket we also classify those items by the next symbol recursively. This happens in  $O(nk)$ .

**LSD Radix Sort** Sort all keys by their last symbol, then sort all keys by their second last symbol, and so on. The sorting algorithm at each stage must be stable.

The time is  $O(nk)$  with space complexity of  $O(n + k)$ .

## 1.5 Graphs

# 2 Algorithm Analysis

## 2.1 Stable Matching Problem

**Hospital's Stable Matching Problem** Suppose there exists  $n$  doctors and hospital. Each hospital wants to hire exactly one new doctor and the hospital ranks the priority of the doctors they would prefer and each doctor also lists their preferences for the hospitals they'd like to go to.

For a stable match, we would like to create an allocation where no group would be happy to trades their allocations with each other.

**Naive Solution** The naive method computes all possible allocations and then picks one that is stable.

This runs in  $O(n!)$  time which is not preferable.

### Fun Fact about Factorials - Sterling's Approximation

$$n! \approx \left(\frac{n}{e}\right)^n$$

### Gale - Shapely Algorithm - Assumptions

- Produces pairs in stages, with possible revisions.
- A hospital which has not been paired, will be called *free*.
- Hospitals will offer jobs to doctors, who will decide whether to accept a job or not. Doctors may renege.
- All hospitals start off as free.

**Gale - Shapely - Solution** While there exists a free hospital which has not offered jobs to all doctors, pick a free  $h$  and have it offer a job to the highest doctor  $d$  on its list to whom they have not offered a job yet.

If no one has offered  $d$  a job yet, they will always accept the pair  $h, d$ .

Otherwise, if they're already on the pair  $h', d$  then if  $h$  is a higher preference than  $h'$  the doctor will renege  $h'$  for  $h$  to form  $(h, d)$ . Otherwise, the hospital will seek another person to offer a job.

**Proving Termination in  $n^2$**  In all rounds of the Gale-Shapely algorithm, the hospital will offer a job to one doctor. They may offer a job to each doctor only once. Thus, the hospital may only make  $n$  offers. There are  $n$  hospitals each making  $n$  offers so, the algorithm must terminate in  $n^2$  rounds or less.

**Proving Correctness of Gale-Shapely** If the while loop has terminated with  $h$  still free, that implies that  $h$  has offered a job to everyone. Since the only way to get declined is if another, more appealing hospital has offered a job to other doctors.

If there are no more doctors available, then  $n$  other hospitals must have offered a job to  $n$  doctors. However this would imply there are  $n+1$  hospitals in total, which is a contradiction.

Hospitals will be happy as they start with their highest preference first and, doctors will move up to accept their most preferred offer. Thus, there cannot exist a circumstance where both a hospital and doctor would like to swap.

## 3 Divide-and-Conquer Method

### 3.1 Counting Inversions

**Brute Force** Brute force will take  $O(n^2)$  time. Any method of looking at entries one by one will have at worst, a quadratic time complexity since there can be  $\frac{n^2}{2}$  inversions.

**Divide and Conquer Method** We can split the array into two equal parts  $A_{hi}$  and  $A_{lo}$ .

Then, for all of the elements in  $A_{hi}$  and the total inversions is the number of elements in  $A_{lo}$  that are less than the elements of  $A_{hi}$ .

We may sort both  $A_{hi}$  and  $A_{lo}$ . Then, we seek to merge the arrays together. Every time we pull an element from  $A$ , we add the number of elements remaining in  $A_{lo}$  to the total number of inversions as, all of those elements are greater than our current element from  $A_{hi}$  but to the left of our current element.

**Time Complexity of Divide and Conquer Method** The divide and conquer method will have the same time complexity as merge sort and thus runs in  $\Theta(n)$ .

### 3.2 The Master Theorem

**Setup Master Theorem** Let  $a \geq 1$  be an integer and  $b > 1$  be a real number,  $f(n) > 0$  be a non-decreasing function defined on the positive integers. Then,  $T(n)$  is the solution of the recurrence

$$T(n) = aT\left(\frac{n}{b}\right) + f(n).$$

Then, we define the critical exponent  $C^* = \log_b(a)$  and the critical polynomial  $n^{C^*}$ .

#### Master Theorem

1. If  $f(n) = O(n^{C^*})$  for some  $\epsilon > 0$  then,  $T(n) = \Theta(n^{C^*})$ .
2. If  $f(n) = \Theta(n^{C^*})$  then,  $T(n) = \Theta(n^{C^*}) \log n$ .
3. If  $f(n) = \Omega(n^{C^*+\epsilon})$  for some  $\epsilon > 0$  and, for some  $c < 1$ , and some  $n_0$ ,

$$af\left(\frac{n}{b}\right) \leq cf(n)$$

holds for all  $n > n_0$  then,  $T(n) = \Theta(f(n))$ .

If the conditions above do not hold then, the master theorem is not applicable.

### 3.3 Integer Multiplication and Addition

**Notation and Basic Methodology** We let  $n$  be the number of bits in the integer. Addition occurs by moving from the least to most significant bit, adding each bit at a time in  $\Theta(n)$ . Multiplication is much of the same but, in  $O(n^2)$ .

**The Karatsuba Trick** This happens in  $O(n^{\log_2 3})$ .

## 4 The Greedy Method

### 4.1 When greed pays off - foundations of the Greedy Method

**What is a Greedy Problem** A greedy algorithm will divide a problem into stages and rather than exhaustively searching through all combinations of options in all stages, it only considers the choice that is the best for the current stage.

The idea is that the search space is reduced however, it is not necessarily given that the solution is found using a greedy point.

**Proofs of Greedy Algorithm** There are two main methods of proof.

1. **Greedy Stays Ahead:** This proves that at every stage, no other algorithm can do better than the proposed algorithm.
2. **Exchange Argument:** Consider an optimal solution and gradually transform it to the solution found by the proposed algorithm without making it any worse.

These methods are analogous to proof by induction and contradiction respectively.

### 4.2 Activity Selection problem

**Problem Statement** There is a list of  $n$  activities with starting times  $s_i$ . and finishing time  $f_i$ . Schedule the activities such that no two activities overlap. Maximise for the total number of activities.

**Solution** Among the activities that do not conflict with the previously chosen activities, choose the activity with the earliest end-time. Ties may be broken arbitrarily.

**Proof** Correctness is proved with the *exchange argument* to show that any optimal solution can be transformed into our greedy solution.

1. Find the first place at which the optimal solution violates the greedy choice.



2. Replace the activity chose with the greedy choice. Clearly, the number of activities is the same. Also, we know there are no conflicts that have been created because, we are working left-to-right and have chosen the first conflict. So, no conflict happens before the start of the greedy's choice. Also, the finish time of greedy is no greater than the finish time of the optimal solution. As such, there is no conflict on the right-side of the greedy choice either.

**Complexity** We can sort using the finishing time as the key in  $n \log n$ . Then, loop through all the activities linearly for a total time of  $O(n \log n)$ .

### 4.3 Job Lateness

**Problem** At a start time  $T_0$  and a list of  $n$  jobs with duration times  $t_i$  and deadlines  $t_i$ . Assume that only one job can be done at a time and all jobs need to be completed

**Solution** Ignore the duration. Then, choose jobs in terms of ascending deadlines.

**Proof** Consider an optimal solution. We say  $i, j$  is an inversion if  $i$  is scheduled before  $j$  but,  $j$  is scheduled after  $i$ .

### 4.4 Huffman Codes

**Array Merging Problem** Are given  $n$  sorted arrays of different sizes. May only merge 2 at a time.

**Huffman Code** Given a set of symbols, encode these symbols into a binary string that can be decoded unambiguously.

**Naive Solution: Ascii** Designate each symbol as a character and assign a unique integer to that character. Given  $n$  characters, you require  $\lceil n \log n \rceil$  bits. Wastes lots of space.

### 4.5 Tsunami Warning

**Situation** There are  $n$  radio towers to broadcast tsunami warnings. You are given the  $(x, y)$  coordinates of each tower and its radius of range. When a tower is activated, all towers within that radius will also activate, causing other towers to activate andso on.

**Task** Design and algorithm to ind the fewest number of towers needed to be equipped

**Attempt 1 - Bad** Find activated tower with the largest radius. Place a sensor at this tower. Then find and remoe all activated towers. Repeat.

**Attempt 2 - Bad** Find unactivated tower with largest number of towers in range. If none, place a sensor at the left-most tower. Repeat.

**Motivation** Consider the towers as nodes on a directed graph where an edge  $a_{ij}$  implies  $i$  causes an activation of  $j$ . Observe that the relation is symmetric.

Suppose that a tower  $a$  causes  $b$  to be activated and vice-versa. Then, we never want to activate  $a$  and  $b$ . Consequently, we never want to activate more than two elements in a cycle.

Let  $S$  be a subset of towers such that activating any tower in  $S$  causes the activation of all towers in  $S$ . As such, we may treat  $S$  as a singular unit; a super-tower.

**Strongly Connected Components** Given a directed graph  $G = (V, E)$  and a vertex  $v$ , the strongly connected components of  $G$  containing  $v$  consists of all vertices  $u \in V$  such that there is a path in  $G$  from  $v$  to  $u$  and a path from  $u$  to  $v$ . We denote this as  $C_v$ . In terms of our problem, strongly connected components are maximal super-towers.

**Finding Strongly Connected Components** Given a graph  $G = V, E$ , create  $G' = (V, E')$  so that  $E'$  has all edges reversed from  $E$ .)

We claim that  $u \in C_v$  iff and only if  $u$  is reachable from  $v$  and  $v$  is reachable from  $u$ . Equivalently,  $u$  is reachable from  $v$  in both  $G$  and  $G'$ .

**All Strongly Connected Components Time Complexity** Observe that this may require a BFS for each component, which runs in  $O(V + E)$ . Doing so for all  $V$  vertices leads to a total of  $O(V(V + E))$  time.

**Kosaraju's Algorithm** Kosaraju's and Tarjan's algorithm can find all strongly connected components in linear time as  $O(V + E)$ . See this in CLRS section 22.5.

**Condensation Graph** Clearly each  $v \in V$  exists in only one strongly connected component. That is, the strongly connected components form a partition of  $V$ . Let  $C_G$  be the condensation graph of  $G$ . That is,

$$\sum_G = (C_G, E^*)$$

where,

$$E^* = \{(C_{u_1}, C_{u_2}) : (u_1, u_2) \in E, C_{u_1} \neq C_{u_2}\}.$$

The vertices of  $\sum_G$  are strongly connected components of  $G$ . The edges of  $\sum_G$  correspond to those edges of  $G$  that are not within a strongly connected component with duplicated ignored.

## Returning To Tsunami Problem with Condensation Graphs

1. Find the condensation graph. This gives a set of *super tower*, for which we know what others get activated. Observe that we want our super-towers to be maximal.
2. Now need to decide what super towers to put towers on. Observe that the condensation graph is a directed, acyclic and anti-symmetric graph.

**Topological Sorting** Let  $G = (V, E)$  be a directed graph with  $n = |V|$ . A topological sort of  $G$  is a linear ordering of its vertices  $\sigma : V \rightarrow \{1, \dots, n\}$  such that if there is an edge  $v, w \in E$  then  $v$  preceded  $w$  in the order. That is,  $\sigma(v) < \sigma(w)$ .

Observe that this may only occur on an acyclic graph, hence the need for the condensation graph. Also, the topological sort may not be unique.

This algorithm may be done in  $O(V + E)$ . Recall that linear algorithms like this are equivalent to reading the graph. As such, they may be done *for free*.

## Dijkstra - Shortest Single Path

**Algorithm Outline** Create a boolean array  $S$  with each index  $i$  covering if  $i$  has had its shortest path weight found.

For each vertex  $v$  maintain a  $d_v$  equal to the weight of the shortest know path  $s \rightarrow v$ . Initially  $d_s = 0$  and  $d_v = \infty$ .

## Correctness

**Claim** Suppose that  $v$  is the next vertex to be added to  $S$  then  $d_v$  is the length of the shortest path from  $s$  to  $v$ .

TODO: Finish this off

**Notation for following** Let  $n = |V|, m = |E|$ . Let the vertices be labelled  $1, \dots, n$  with source  $n$ .

**Array Solution** Store  $d_i$  in array of length  $n$ . At each stage

1. Perform linear search of  $d$  ignoring vertices already in  $S$ . Select vertex  $v$  with smallest  $d[v]$  to be added to  $S$ .
2. For each outgoing edge from  $v$  to some  $z \in V \setminus S$ , update  $d[z]$  if necessary.]

At each  $n$  steps, we perform a linear scan on array of length  $n$  and also update  $d$  in constant time at most once for each edge.

Thus, the complexity is  $O(n^2 + m)$ . IN a simply graph, this can be simplified to  $O(n^2)$ . This solution is okay for dense graphs but struggles when it comes to sparse graphs.

**Motivation for Better Structure** We need to support

1. Finding  $v \in V \setminus S$  with smallest  $d_v$
2. For each outgoing edge  $w, z$  update  $d_z$  if needed.

The first is a a slow linear search. However we can skip over vertices already in  $S$ . This can be implemented by deleting the  $d_v$  from the data structure all together.

For this, we may use a min-heap priority queue as it has fast deletion and access of the minimum element. However, the standard heap doesn't allow updating values.

**Augmented Heap** The heap may be represented as an array of size  $n$  where the left child of  $A[j]$  is stored in  $A[2j]$  and the right child in  $A[2j + 1]$ .

Changing  $d[i]$  is now  $O \log n$ . It requires a lookup of ] TODO::w

## 4.6 Minimum Spanning Trees

**Definition** A minimum spanning tree (mst)  $T$  of a connected graph is a sub-graph of  $G$  which is a tree that contains all vertices of  $G$  and minimises the sum of all edges.

**Lemma** Let  $S$  be a non-empty proper subset of all vertices of  $G$ . Assume that  $e = (u, v)$  is an edge such that  $u \in S$  and  $v \notin S$  and  $e$  is of minimal length of all edges with this property. Then,  $e$  must belong in all msts of  $G$ .

**Kruskal's Algorithm** Sort all the edges  $E$  in increasing order by weight. Then, starting from lowest weight to highest, if adding an edge will not result in a cycle, then add it to the graph. Otherwise, discard the edge and return it.

**Uniqueness of Kruskal** Consider where all weights are distinctive. Consider edge  $e = (u, v)$  added during Kruskal's algorithm and let  $F$  be the forest in its state before adding  $e$ .

Let  $S$  be the set of vertices reachable from  $u \in F$ . Then clearly  $u \in S, v \notin S$ .

The original graph does not contain any edges shorter than  $e$  with one ending in  $S$  and the other not in  $S$ . If it existed, then it would have already been added in during an earlier iteration of the algorithm.

**Implementing** The sorting is a solved problem. However, finding cycles is more difficult. For this, we use a union-find.

**Union-find** The union-find handles disjoint sets and has at least the following operations.

1. Creation of the set returns a structure where all vertices are placed into distinct singleton sets.
2. Find: Returns the label of the set to which the given element belongs. Returns in  $O(1)$
3. Union: Takes two sets and merges them together into the union of the sets. Takes linear time but, by amortized analysis we see that the first  $k$  operations run in a total of  $O(k \log k)$ . Note that this is NOT equivalent to an average case analysis but rather, the worst-case of an aggregate of operations.

**Additional Data Structures** The simplest union-find requires

1. An array  $A$  such that  $A[i] = j$  means that  $i$  belongs to the set with representative  $j$ .
2. Array  $B$  where  $B[i]$  contains the number of elements in the set with representative  $i$ .
3. Array  $L$  where  $L[i]$  contains pointers to the head and tail of a linked list containing elements of the set with representative  $i$ .

**Union-Find** Given sets  $I, J$  with representatives  $i, j$ , the union is as follows:

1. Assume  $B[i] \geq B[j]$  without loss of generality.
2. For each  $m \in J$  update  $A[m] = i$ .
3. Update  $B[i] = B[i] + B[j]$ .
4. Append  $L[j]$  to  $L[i]$  and replace  $L[j]$  with an empty list.

Observe that the new  $B[i]$  is atleast twice the old value of  $B[j]$  since  $B[i] \geq B[j]$ . That is, the number of elements in  $B[A[m]]$  changes, it

## 5 Network Flow Algorithms

### 5.1 Flow networks

**Definition: Flow Network** A flow network  $G = (V, E)$  is a directed graph where each edge  $e = (u, v) \in E$  has a positive integer capacity  $c(u, v) > 0$ .

There are two special vertices. A source  $s$  and sink  $t$ . There are no outgoing edges for a sink and likewise, no incoming edges for a source.

**Flow** A flow in  $G$  is a function  $f : E \rightarrow [0, \infty)$  such that  $f(u, v) > 0$ . The function  $f$  must satisfy

1. **Capacity Constraints:**  $f(u, v) \leq c(u, v)$ , for all  $e(u, v) \in E$ .
2. **Flow Conservation:** For all  $v \in V \setminus \{(s, t)\}$ , we require

$$\sum_{(u,v) \in E} f(u, v) = \sum_{(v,w) \in E} f(v, w).$$

That is, the incoming flow must be equal to the outgoing flow.

**Value of Flow** The value of flow is defined by

$$|f| = \sum_{v:(s,v) \in E} f(s, v) = \sum_{v:(v,t) \in E} f(v, t).$$

That is, the flow leaving the source or, flow arriving at sink.

**Goal of Flow Network** Given a flow network, the aim is to find a flow of maximal value.

**Integrality Theorem** If all capacities are integers, then there exists a flow of maximum value such that  $f(u, v)$  is an integer for all edges  $(u, v) \in E$ .

**Greedy Does not Work** The obvious greedy solution would be to send flow arbitrarily, one-unit at a time. However, this may converge to some local maximum that is not reflective of the true global maximum.

**Residual Flow Network** Given a flow network, the *residual flow network* is the network is the network made up of the leftover capacities.

**New Edges** Suppose there is an edge  $e(v, w)$  with capacity  $c_1$  and flow  $f_1$  units and edge  $e(w, v)$  with  $c_2, f_2$  capacity and flow respectively.

The forward edge  $v \rightarrow w$  allows  $c_1 - f_1$  additional units of flow. We can also send  $f_2$  units to cancel the flow to the reverse edge.

Thus, we create edges  $v \rightarrow w$  with a capacity of  $c_1 - f_1 + f_2$  for the forward edge and,  $c_2 - f_2 + f_1$  on the backwards edge.

**Augmenting Path** An *augmenting path* is a path from  $s \rightarrow t$  in the residual flow network.

The capacity of an augmenting path is the capacity of its *bottleneck* edge. That is, the edge of smallest capacity.

We should then send that amount of flow along the augmenting path, recalculating the flow and the residual capacities for each edge used.

**Recalculating Augmented Path** Suppose we have an augmenting path of capacity  $f$  that includes an edge  $v \rightarrow w$ . Then,

1. Cancel up to  $f$  units of flow being sent from  $w \rightarrow v$
2. Add the remainder of those  $f$  units to the flow being sent from  $v \rightarrow w$ . That is, reverse the flow.
3. Increase the residual capacity from  $w \rightarrow v$  by  $f$  and decrease the residual capacity from  $v \rightarrow w$  by  $f$ .

## 5.2 Ford – Fulkerson Algorithm

### Ford - Fulkerson Method

1. Initialise flow  $f$  to 0.
2. While there exists an augmenting path  $p$  in the residual network  $G_f$ , augment flow along  $p$ .
3. The final flow is  $f$ .

That is, keep adding flow through augmenting paths for as long as it is possible.

**Proving Termination** If all capacities are integers then, each augmenting path increases the flow through the network by at least one unit. However, the total flow is finite. It cannot be larger than the sum of all the capacities.

**Cut** A cut in a flow network is any partition of the vertices of the underlying graph into two subsets  $S$  and  $T$  such that:

1.  $S \cup T = V$
2.  $S \cap T = \emptyset$
3.  $s \in S, t \in T$ .

**Capacity of Cut** The capacity  $c(S, T)$  of a cut  $(S, T)$  is the sum of capacities of all edges leaving  $S$  and entering  $T$ . That is,

$$c(S, T) = \sum_{(u,v) \in E} \{c(u, v) : u \in S, v \in T\}.$$

**Flow of Cut** The flow through a cut is the net flow from the source side, to the sink side,  $S \rightarrow T$ .

**Lemma: Value of Flow of Cut** The flow of the cut, is no more than the capacity of the cut. That is,  $|f| \leq c(S, T)$ .

**Max Flow Min Cut Theorem** The maximal amount of flow in a flow network is equal to the capacity of the cut of minimal capacity.

**Time Complexity: Ford Fulkerson** The number of augmenting paths can be up to the value of the max flow  $|f|$ . Each augmenting path takes  $O(V + R)$  (by a DFS). In any sensibly flow network,  $V \leq E + 1$ , so we may shorten this as  $O(E)$ . If this was not the case, we would have isolated vertices, which are redundant.

Therefore, the total time complexity is  $O(E|f|)$ .

**The Capacity Issue** Suppose that all capacities are  $\leq C$ . Then, the length of the input (the bits required to encode it) is  $V + E \log C$ . However, the value of the maximum flow can be as large as  $VC$ . Therefore,  $O(E|f|)$  can be exponential. This is very cringe.

## 5.3 Speeding Up Max Flow

**Edmonds-Karp Algorithm** This algorithm improves upon the Ford-Fulkerson algorithm by using a BFS instead, to find the augmenting path. That is, it chooses the augmenting path consisting of the fewest edges.

At each step, we find next augmenting path in  $O(V + E) = O(E)$ . However, choosing paths on length may lead to prioritising edges with small edge length, before edges with large capacity. This does not matter.

See CLRS for a proof that the number of augmenting paths is  $EV$ . Thus, the total time complexity is  $O(VE^2)$ .

**Faster Algorithms** Dinic's algorithm is faster as it can be done in  $O(V^2E)$  and Preflow-Push in  $O(V^3)$ . However, these are not allowed within the context of this course.

**Note on Practicality** In practice, max-flow algorithms based on augmenting paths will run a lot better on average, than the worst-case complexity.

## 5.4 Strategies for Alternative Max Flow Problems

**Networks with Multiple Sources** If there are multiple sources and sinks, the problem is reducible by adding a *super-source* and *super-sink* to the network where the super-source is connected to all sources with an edge of infinite capacity. Likewise, all the sinks connect to the super-sink with infinite capacity.

**Networks with Vertex Capacities** Suppose that each vertex  $v_i$  has a capacity  $C(v_i)$  which limits the total throughput of the flow coming in and out of the vertex. That is,

$$\sum_{e(u,v) \in E} c(u,v) = \sum_{e(v,w) \in E} f(v,w) \leq C(v).$$

This can be handled by placing an edge which bottlenecks the flow in and out of the vertex, by splitting the vertex  $v$  into  $v_{\text{in}}, v_{\text{out}}$ .

Then, attach all of  $v$ 's incoming edges to  $v_{\text{in}}$  and all of its outgoing edges from  $v_{\text{out}}$ . Then, create an edge to connect  $v_{\text{in}}, v_{\text{out}}$  such that

$$c^*(v_{\text{in}}, v_{\text{out}}) = C(v).$$

## 5.5 Max Flow Application: Movie Rental

**Problem** Suppose that you have  $k$  movies with  $m_i$  copies of the  $i$ -th movie. There are also  $n$  customers who all have a subset of the  $k$  movies they want to see. However, each customer can only see up to 5 movies at a time.

The task is to design an algorithm running in polynomial time of  $n$  and  $k$  that dispatches the largest number of movies.

**Creating a Flow Network** Construct a flow network with a super-source  $s$  and super-sink  $t$ . Also add the following:

1. A vertex  $u_i$  for each  $i$ -th customer.
2. A vertex  $v_j$  for each  $j$ -th movie.
3. For all  $i$  customers, an edge from  $s$  to  $u_i$  with capacity 5.
4. For all customers,  $i$ , for all movies  $j$  that they want to see, create an edge  $u_i \rightarrow v_j$  with capacity 1.
5. For each movie  $j$ , an edge from  $v_j$  to  $t$  with capacity  $m_j$ .



**Interpreting the Flow Network** Any path in this network from  $s \rightarrow t$  will be of the form  $s \rightarrow u \rightarrow v \rightarrow t$ . Each edge  $u \rightarrow v$  has capacity 1. This can be thought of as a customer  $u$  loaning a movie  $v$ .

Observe that the flows exhibit preference and as such, each customer will only rent movies they want to see.

By flow-conservation, the amount sent along  $s \rightarrow u$  is equal to the flow from  $u$  to all  $v$ . Thus, the capacity constraint ensures that this doesn't exceed 5. Similarly, no more than  $m_j$  of each movie is rented out.

**Time Complexity** Recall that Edmonds-Karp can find edges in  $O(VE^2)$  time. We have  $n+k+2$  vertices and up to  $nk+n+k$  edges. Therefore, as required, the total time complexity is in polynomial time

$$O((n+k+2)(nk+n+k)^2) \leq O(n^3k^3).$$

However, recall that Ford-Fulkerson can find max flow in  $E|f|$  time, and the max flow is  $5n$ , this problem can be solved in

$$O(E|f|) = O(n(nk+n+k)) = O(n^2k).$$