Algorithms and Programming Techniques
COMP3121 UNSW

Hussain Nawaz
hussain.nwz000@gmail.com

2022T2

Contents

1 Introduction - Revision **3**
1.1 Rates of Growth . . . 3
1.2 Assumed Data Structures . . . 4
1.3 Algorithms . . . 5
1.4 Non-Comparison Sort . . . 6
1.5 Graphs . . . 7

2 Algorithm Analysis **7**
2.1 Stable Matching Problem . . . 7

3 Divide-and-Conquer Method **8**
3.1 Counting Inversions . . . 8
3.2 The Master Theorem . . . 8
3.3 Integer Multiplication and Addition . . . 9

4 The Greedy Method **9**
4.1 When greed pays off - foundations of the Greedy Method . . . 9
4.2 Activity Selection problem . . . 9
4.3 Job Lateness . . . 10
4.4 Huffman Codes . . . 10
4.5 Tsunami Warning . . . 10
4.6 Minimum Spanning Trees . . . 13

5 Network Flow Algorithms **14**
5.1 Flow networks . . . 14
5.2 Ford – Fulkerson Algorithm . . . 15
5.3 Speeding Up Max Flow . . . 16
5.4 Strategies for Alternative Max Flow Problems . . . 17
5.5 Applications of Max Flow Problems . . . 17

5.5.1 Max Flow Application: Movie Rental . . . 17
5.5.2 Flow Application: Cargo Allocation . . . 18
5.5.3 Disjoint Paths . . . 19
5.5.4 Bipartite Matching . . . 19
5.5.5 Job Center Problem . . . 20

6 Dynamic Programming Method **20**
6.1 Introduction . . . 20
6.2 Longest Increasing Subsequence . . . 21
6.3 Actvitiy Selection . . . 22
6.4 Making change . . . 23
6.5 Knapsack Problem . . . 24
6.5.1 Knapsack: Duplicates Allowed . . . 24
6.5.2 Knapsack: Duplicates Allowed . . . 24
6.6 Balanced Partition . . . 25
6.7 Multiplying Chains of Matrices . . . 26
6.8 Longest Common Subsequence . . . 27
6.8.1 Two Sequences . . . 27
6.8.2 Three Sequences . . . 28
6.8.3 Shortest Common Supersequence . . . 28
6.9 String Matching . . . 28
6.10 Maximising an Expression . . . 29
6.11 Shortest Path in Directed, Acyclic Graph . . . 30
6.12 Assembly line scheduling . . . 30
6.13 Single Source Shortest Paths . . . 31
6.13.1 Bellman-Ford . . . 32

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 ω 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 i $\log n$.

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 $1, \ldots, k$ 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 aspect 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 exists 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 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 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 and 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,

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

   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 strogly 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 reachible 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.

Kosaraku'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 stronlgly 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 $\sum_G$ correspend 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. Observer 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 $\sigma : V \to \{1, \ldots, n\}$ 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 acylic 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*.

Djikstra - 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 $s \to v$. Initially $d_s = 0$ and $d_v = \infty$.

Correctness

Claim Suppose that $v$is the next vextex to be added to S then d_v d_v is teh lenght 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 $1, \ldots, n$ with source n.

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

1. Perfor linear search of d ignoring vertices already in S. Select vertext v with smallest $d[v]$ to be added to S.
2. For each outgoing edge from v to some $z \in V \backslash S$, 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 simiplified 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 $v \in V \backslash S$ with smallest d_v
2. For each outdoing 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 theod_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 wwhere 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 connexted 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 subsert of all vertices of G. Assume that $e = (u, v)$ is an ege 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 $e = (u, v)$ add during Kruskal's algorithmn 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 grpah 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 and 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 unino-find handles disjoint sets and has atleast the following operations.

1. Creation of the set returns a sstructure 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 toghether 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 it NOT equivalent to an average case analysis but rather, the worst-case of an aggregate of operations.

Additional Data Structures The simiplest union-find requires

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

Union-Find Given sets I, J with represtatives 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 $f : E \to [0, \infty)$ such that $f(u, v) > 0$. The function f must satisfy

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

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

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 arbitrarity, 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 Suppoe there is en 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 $v \to w$ 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 $v \to w$ 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 $s \to t$ 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 $v \to w$. Then,

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

Lemma: Value of Flow of Cut T he 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 maximual 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 comlexity 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, this are not allowed within the context of this course.

Note on Practicality In practice, max-flow algorithms based on augmenting paths will run alot 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-sink* and *super-sink* to the network where the super-source is connected to all sources with an edge of infinite capacity. Likewike, all the sinks connect to the super-sink with infinite capacity.

Networks with Vertex Capacities Suppose that each vertex v_i has a capcity $C(v_i)$ 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_{in} and all of its out going edges from v_{out}. Then, create an edge to connect $v_{\text{in}}, v_{\text{out}}$ such that

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

5.5 Applications of Max Flow Problems

5.5.1 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 ot 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 and edge $u_i \to v_j$ with capacity 1.
5. For each movie j, an edge from v_j to t with capacity m_j.

Interperting the Flow Network Any path in this network from $s \to t$ will be of the form $s \to u \to v \to t$. Each edge $u \to v$ 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 $s \to u$ 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

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

5.5.2 Flow Application: Cargo Allocation

Problem The storage of a ship is in the form of a rectangular grid of cells with n rows and m columns. Some of the cells are supported with pillars and cannot be used for storage and as such, have zero capacity.

Every cell in row i and column j has a capacity of $C(i,j)$. The capacity of the total weight of each row i must not exceed $C_r(i)$. Similarly, the capacity of the weight of any colum j must exceed $C_c(j)$.

Task Design an algorithm in polynomial time of n and m that allocates the maximum possible weight of cargo to the ship without violating capacity.

Flow Network Construct a flow network with a source s and sink t. Also add the following:

1. A vertex r_i for each row i,
2. A vertex c_j for each column j,
3. An edge $s \to v_i$, $\forall i$,
4. For each cell $c_{i,j}$ that is not a pillar, an edge $r_i \to c_j$ with capacity $C(i,j)$.
5. An edge from $c_j \to t$ with capacity $C_c(j)$, for all j.

Interperting the Flow Network Consider a flow path $s \to t$. For some i, j, this path traverses through with form

$$s \to r_i \to c_j \to t.$$

The value of this flow can be thought of as the weight of the cargo on cell i, j. The capacity of edge $s \to r_i$ ensures non-violation of the row, while $c_j \to t$'s capacity ensures non-violation of the column capacity. Finally, the weight of $r_i \to c_j$ will be to avoid violation of the cell's capacity.

5.5.3 Disjoint Paths

Problem Given a directed G with n vertices and m edges. r of the vertices are painted red, b are blue and the remaining $n - r - b$ are black. Red vertices have only outgoing edges, while blue vertices have only incoming edges.

Task Design an algorithm in polynomial time of n and m to determine the largest possible number of vertex-disjoint (non-intersecting) paths in this graph that starts at a red vertex and finishes at a blue vertex.

Flow Network Trivially, the red-vertices are sources and the blue-vertices are sinks. As such, we apply the super-source and super-sink to the flow network. Also,

1. For all black vertices v, create a vertex $v_{\text{in}}, v_{\text{out}}$, joined by an edge of capacity 1. The capacity ensures no vertex is used twice.
2. Each edge of the original graph $u \to v$ becomes an edge $u_{\text{in}} \to v_{\text{out}}$ with capacity 1.

5.5.4 Bipartite Matching

Problem A bipartite grpah is one where vertices can be seperated into two disjoint sets such that no two vertices in the saim set are adjacent. The number of matching pairs in a maximum matching, may be referred to as the size of this graph.

Conversion to Max Flow Let A, B be the two disjoint sets of G. Create vertices s, t as source and sink. Then,

1. Create an edge from $s \to u, \forall u \in A$ with capacity 1.
2. Create an edge from $s \to v, \forall u \in B$ with capacity 1.
3. Orient existing edges from $A \to B$. That is, $\forall e(u, v)$, create that edge in the flow graph.

Recall that for all $e(v, w)$ in the flow network with capacity c that carries f units of flow, we record two edges in the residual graph. That is, the edges:

- An edge $v \to w$ with capacity $c - f$,
- An edge $v \to w$ with capacity f.

Since all capacities are 1, we only denote the direction of the edge in the residual graph.

5.5.5 Job Center Problem

Problem There are k recognised qualifications. n unemployed people hold at most k of these qualifications. There are also m job openings that require a subset of these qualifications.

Task Design a problem in polynomial time of n, m, k that allocates as many people as possible to the job openings for which they are qualified. Workers take at most one job. Each job employs at most one worker.

Creating The Graph Create an unweighted, undirected graph with vertices $a_1, \ldots, a_n$ and $b_1, \ldots, b_m$, that represent each worker and job respectively.

Create an edge $a_i \to b_j, forall i, j$, if the worker a_i is qualified for job b_j.

The resultant graph is bipartite and the problem is reduces to a maximum matching problem for a bipartite graph.

Time Complexity For all n workers, we need to iterate up to k qualifications for all m jobs. Thus, creation of the graph can be done in $O(nmk)$ time.

Recall that the tightest bound is of form $O(E|f|)$. Consider that

$$E \leq nm + n + m, \text{ and } |f| \leq \min(n, m).$$

Thus, the total time complexity is $O(nm(k + \min(m, n)))$, in polynomial time as required.

6 Dynamic Programming Method

6.1 Introduction

Dynamic programming Dynamic Programming is a method of solving problems that can be broken down into subproblems and solved recursively.

CLRS: Four-Step Method

1. Characterize the structure of the optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution; typically this is done bottom-up.
4. Construct an optimal solution from computed information.

Overlapping Subproblems Property We must choose subproblems in a way such that the same subproblem occurs several times in the recursion tree. When a subproblem is solved, we store the result so that subsequent instances of the same subproblem can be answered by a lookup table.

Considerations for Putting it Together

- Is the original problem a subproblem or a combination of subproblems.
- Order that the subproblems occur in.
- How many subproblems are there? How long does each take?

6.2 Longest Increasing Subsequence

Problem Given a sequence of n real numbers $A[1..n]$, find a subsequence (not necessarily contiguous), where the values are strictly increasing.

Subproblem: Prefixing Lengths A natural choice is, for all $i \in [1, n]$, find the length of the longest increasing subsequence of $A[1..i]$ that ends at $A[i]$. Call this $Q(i)$.

We try to solve $Q(i)$ by extending some sequence $Q(j)$ for $j \leq i$.

We assume that we have solved for all $j < i$. Then, look for a $j < i$ such that $A[j] < A[i]$. Pick a sequence m to get a maximal result and extend it with $A[i]$.

We stop at a base case of $i = 1$, which is the trivial case.

The recurrence we define is

$$Q(i) = \max\{Q(j) : j < i, A[j] < A[i]\} + 1.$$

The longest subsequence will correspond to the largest $Q(i)$.

Time Complexity The n-th entry may require us to look at $n - 1$ entries. We do this n times, which means the prefixing takes $O(n^2)$ time in total. The final solution being extracted takes $O(n)$ and is therefore negligible.

Doing Better This can be done in $n \log n$. Finding such an algorithm is left as an exercise left to the reader.

Correctness We claim that truncating the optimal solution for $Q(i)$ will produce an optimal solution for $Q(m)$. I can't be bothered writing out why.

Getting Sequence Not Length To get the sequence, create an array of *predecessors*, P. When finding $Q[i]$, let $P[i] = j$ where j is the index chosen according to the definition of $Q[i]$.

When backtracking through the array, when the max is found, start from the max, then follow the predecessors until one with no predecessors is found.

6.3 Actvitiy Selection

Problem Input is a list of n activities with starting and finishing times s_i, f_i. Find the maximal duration of a subset of activities, where no two activities take place at the same time.

This is a variation of a problem from the greedy chapter. The greedy approach will not work for maximal duration.

Setup Sort the activities by their finishing times such that

$$f_1 \leq f_2 \leq \cdots \leq f_n.$$

Subproblem: Maximal Duration Let $P(i)$ be the subproblem of finding the duration $t(i)$ of a subsequence $\sigma(i)$, of the first i acitivities that

1. consists of non-overlapping edges,
2. ends with activity i,
3. is of maximal duration.

$P(i)$ is solved by appending activity i to some sequence

$$t(i) = \max\{t(i) : j < i, f_j < s_i\} + f_i - s_i.$$

There is a base case $P(i) = f_1 - s_2$.

Best Solution The best time is $\max(P)$.

Time Complexity

1. Sorting takes $O(n \log n)$.
2. We loop through n elements, searching $n-1$ elements for the best t which leads to the complexity for all subproblems to be $O(n^2)$.
3. Finding the best t is $O(n)$.

Thus, the overall time is $O(n^2)$.

Proof of Correctness of Subproblem Let the optimal solution for $P(i)$ be given by $\{k_j\}_{j=1}^{m}$, where $k_m = i$. The requirement is that $\sigma' = \{k_j\}_1^{m-1}$ is optimal solution to $P(k_{m-1})$. This is proved by contradiction.

Suppose that instead $P(k_{m-1})$ is solved by a sequence τ' with a duration larger than σ'.

Then, extend τ' with activity i. Therefore, τ_i will trivially have a larger duration than σ'. This however contradicts the assumption of σ as the sequence solving i. Hence, the optimal solution is achieved by extending the $P(j)$ of largest duration that supports extension of i, where $j < i$.

Sequence of Activities To find the largest sequence, we wish not just to store $t(i)$ with each $P(i)$ but, also the predecessor j where $P(i)$ extends $P(j)$.

6.4 Making change

Problem You are given n types of coin denominations of integer values

$$\{v_i\}_{i=1}^{n}, \text{ where } v_1 < v_2, \ldots, < v_n.$$

Suppose $v_1 = 1$ and there is an unlimited amount of each v_i. This allows for it to always be possible to create change.

Task Make change for a given integer amount C, with the minimal number of coins.

Greedy Setup: Works Greedily take as many coins of the largest possible denomination v_n that is possible. Then, do the same for v_{n-1} and so on.

This approach works for all real-world currencies but, not for all sequences. Consider the denominations $\{1, 10, 15\}$ with $C = 20$. Greedy would choose $\{15, 1, 1, 1, 1, 1\}$ but the optimal is $\{10, 10\}$.

Subproblem We will work in a bottom-up direction, finding the optimal solution for all values up to C.

Solving a Subproblem Suppose that we have found the solutions for all amounts $j < i$ and want to find a solution for i.

Consider all coins v_k that are a part of the solution for amount i. Make up the remaining amount $i - v_k$ with the previously computed optimal solution.

Of all the optimal solutions (being computed recursively), find the pick the one that uses the fewest number of coins.

Suppose that we choose the coin m. The optimal solution $P(i)$ is found by adding a coin of denomination v_m to $P(i - v_m)$.

Base Case Suppose that $C = 0$. Then trivially, the solution has no coins.

Time Complexity Each of the C subproblems can be solved in $O(n)$ time. As such, the time complexity is just $O(nC)$. This is *not* polynomial time in terms of m, as C may be unrestricted.

Proving Optimal Solution Consider an optimal solution for some i. Suppose this contains a coin of denomination v_m, for some $1 \leq m \leq n$. Remobving this coin must leave an optimal solution for $i - v_m$ by the same cut and paste argument.

By consider all coins (of value up to i), we can pick an m for which the optimal solution $i - v_m$ uses the fewest coins.

Finding Combinations of Coins If the solution also finding what combination of coins is used, then each $P(i)$ should also store with it, a pointer to pred(i), which is the optimal solution for $i - v_k$, alongside the value of v_k.

Then, when answering the question, we can just do a backtrace.

Notation A k that minimises $P(i - v_k)$ is denoted as

$$\text{argminopt}(i - v_k).$$

6.5 Knapsack Problem

6.5.1 Knapsack: Duplicates Allowed

Problem There are n items. All items of kind i are identical with weight w_i. All weights are integers. We can take any number of items of each kind, to fill a Knapsack of capacity C.

The task is to choose a combination of items of maximal total capacity, which fit the capacity of the Knapsack.

Subproblem The problem is effectively a reflavoured version of the making change problem.

We assume that we have solved the problem for all $j < i$. Call this value $P(j)$. Then, consider all item types where the k-th element has a weight w_k. If we add this item, then the rest of the bag can be fixed with an allocation matching the optimal solution for $i - w_k$. Choose the m that maximises the total value of $P(i - w_m)$. Then, the optimal solution for i is $w_m + P(i - w_m)$.

There is a base case where $\text{opt}(0) = 0$.

Time Complexity There are up to C subproblems, each of which can be solved in $O(n)$ time. As such, the total overall time is $O(nC)$. Again, this is not polynomial in terms of the length of the input.

6.5.2 Knapsack: Duplicates Allowed

Flaws in Previous Algorithm The previous algorithmn is not able to discern if a value v_k has already been used. This breaks the rule of duplicates allowed.

Storing Values is not Enough Suppose that we follow the previous approach but store the values of what v_k was used at that step. This is not enough. There are two main issues:

1. The optimal soltion for $i - w_k$ is not necessarily unique. Thus, we may record some set of items, including k when, an equal selection of items is possible without k.

2. Even if all optimal solutions for $i - w_k$ use k, its still possible that there is a suboptimal solution for $i - w_k$ that does not use k.

The reason that this fails is due to the lack of the optimal substructure property.

New Way to Solve Subproblem For all total weights i, find the optimal solution using only the first k items. Then,

- If k is used in the solution then, there is $i - w_k$ weight left to fill using the first $k-1$ items.
- If k is not used in the solution then, we must fill all i units of weight with the first $k-1$ items.

New Subproblem For $0 \leq i \leq C$ and $0 \leq k \leq n$, let $P(i,k)$ be the optimal solution for determining $\text{opt}(i,k)$. That is, the maximum value that can be achieved up to i units of weight, only using the first k items where, $m(i,k)$ is the (largest) index of an item in such a collection.

We have the recurrence for $i > 0$ and $0 \leq k \leq n$, that

$$\text{opt}(i,k) = \max(\text{opt}(i,k-1), opt(i-w_k, k-1), +v_k),$$

with $m(i,k) = m(i,k-1)$ in the first case and k in the second.

There is a base case with $\text{opt}(i,k)$ if i or k is zero and $m(i,k)$ is undefined.

Order Matters When we get to $P(i,k)$, there is an expectation that $P(i,k-1)$ and $P(i-w_k, k-1)$ has already been solved. This is guaranteed if the subproblems are solved in increasing order of k first and then, increasing capacity of i.

Overall Solution and Time Complexity The final solution is just $\text{opt}(C,n)$.

Each problem can be solved in constant time and, there are $O(nC)$ subproblems. Thus the overall time complexity is $O(nC)$

6.6 Balanced Partition

Problem The input is a set of n positive integers $x_i \in S$. The task is to partition these integers into two subset S_1, S_2 with sums $\sum_1, \sum_2$ respectively, such that $|\sum_1 - \sum_2|$ is minimised.

Reframing the Condition Without loss of generatlity, assume $\sum_1 \geq \sum_2$ and let $\sum$ be the sum of all elements in the set. Then,

$$\sum_1 + \sum_2 = \sum.$$

As such, it follows

$$\sum_1 - \sum_2 = 2\left(\frac{\sum}{2} - \sum_2\right).$$

That is, we want to find a subset S_2 with a sum as close to $\frac{\sum}{2}$ as possible, without exceeding it.

Reframing as Knapsack For all $x_i \in S$, create an item with both weight and value equal to x_i. Consider the knapsack problem (with no duplicates) as specified as above with capacity of $\frac{\sum}{2}$.

All items that are put in this knapsack can be used as S_1 while the remaining items can be a part of S_2.

6.7 Multiplying Chains of Matrices

Matrix Multiplication The product of any two matrices A, B AB exists if the number of columns in A is equal to the number of rows of B. If A is $m \times n$ and B is $n \times p$ then, AB is $m \times p$.

Each element of AB $ab_{i,j}$ is dot product of the i-th row of A with the j-th column of B. Hence, a naive solution would hvae $m \times n \times p$ multiplications.

The order in which matrices are multiplied can drastically change the amount of multiplications required.

Problem The input is a sequence of matrices $A_1, A_2, \ldots, A_n$ where A_i is of dimension $s_{i-1} \times S_i$.

The task is to find the order of multiplication that minimises the number of multiplications required.

Brute Force The brute force solution is of form $\Omega(2^n)$. The number of groupings $T(n)$ is called the Catalan numbers.

Dynamic Approach There are many redundancies in the brute force approach as many of the same prefixes are solved more than once.

A natural place to start, is just solve for the prefixes. However, this is not enough. We need to solve for all contiguous subsequences $A_{i+1}, \ldots, A_j$.

Recurrences The recurrence will consider all of the possible ways to place the outermost multiplication, splitting the chain into the product of elements $(A_{i+1}, \ldots A_k)(A_{k+1}, \ldots, A_j)$. We assume that we have solved the shortest way to multiply the sequence on the left and, on the right.

The base-case is for a subsequence of length one where no multiplications are required.

Notation for the Subproblems Let $P(i, j)$ be the problem of determining $\text{opt}(i, j)$, which is the fewest multiplications needed to compute the product $A_{i+1}\times, \ldots, A_j$. Then, for all $j - 1 > i$,

$$\text{opt}(i, j) = \min\{\text{opt}(i, k) + s_i s_k s_j + \text{opt}(k, j) : i < k < j\}.$$

Order Matters Solving $P(i, j)$ requires solving $P(i, k)$ and $P(k, j)$ for all $i < k < j$. This is guaranteed if, we solve in order of chain length.

Final Solution and Time Complexity The overall solution is $\text{opt}(0, n)$. There are $O(n^2)$ subproblems. Each of these requires $O(n)$ checks to find the best point of split. Thus, the time complexity is $O(n^3)$ as an upper bound.

However, there are many short chains and, very few short chains and, the time complexity can be theoretically tightened. This is left as an exercise to the reader.

Finding Actual Bracketing To find the actual location of brackets, we need to record the splitting point that is used to obtain it. Then, we can just backtrack from the final solution.

6.8 Longest Common Subsequence

6.8.1 Two Sequences

Problem Consider two subsequences

$$S = \langle a_1, a_2, \ldots, a_n \rangle \qquad \text{and,} \qquad S* = \langle b_1, b_2, \ldots, b_m \rangle$$

The task is to find the longest common subsequence of S and S^*.

Subsequences A subsequence s of another sequence S is a sequence of ordered elements of S obtained by removing some elements of S while preserving order.

This may be used as a measure of similarity between two sequences. This can be particularly useful for genetic analysis.

Prefixes Consider S_i, S_j^* which are just the first i and j elements of S, S^*. Consider the last symbol of both, a_i, b_j. If $a_i = b_j = c$ then, the longest common subsequence is found by appending c to the solution for S_{i-1}, S_{j-1}^*.

Subproblem For all $0 \leq i \leq n$ and $0 \leq j \leq m$, let $P(i, j)$ be the subproblem of determinig $\text{opt}(i, j)$. That is, the length of the longest common subsequence of the truncated sequences

$$S_i = \langle a_1, a_2, \ldots, a_i \rangle \quad \text{and, } S_j^* = \langle b_1, b_2, \ldots, b_j \rangle.$$

We have that

$$\text{opt}(i, j) = \begin{cases} \text{opt}(i-1, j-1) + 1 & \text{if } a_i = b_j \\ \max(opt(i-1, j), opt(i, j-1)) & \text{if } a_i \neq b_j \end{cases}.$$

The base cases are $\text{opt}(i, 0) = \text{opt}(0, j) = 0$.

Order Matters Solving $P(i, j)$ requires having solved $P(i, j-1), P(i-1, j)$. As such, we can proceed in a bottom up manner, by solving all subproblems $P(i, 0)$, then all $P(i, 1)$ until we reach $P(i, j)$.

Overall Solution and Time Complexity The overall solution is $O(mn)$. There are mn subproblems, each of which an be solved in constant time.

6.8.2 Three Sequences

Problem Consider three sequences S, S^*, S' with elements a_i, b_i, c_i of length l, m, n respectively.

Subproblems For all i, j, k let $P(i, j, k)$ be the subproblem of determining the longest common sequence of the truncates sequences

$$S_i = \langle a_1, a_2, \ldots, a_i \rangle \quad \text{and, } S_j^* = \langle b_1, b_2, \ldots, b_j \rangle \quad \text{and, } S'_k = \langle c_1, c_2, \ldots, c_k \rangle$$

Then, for all i, j, k

$$\text{opt}(i, j, k) = \begin{cases} opt(i-1, j-1, k-1) + 1 & \text{if } a_i = b_j = c_k \\ \max\Big(\text{opt}(i-1, j, k), \text{opt}(i, j-1, k), \text{opt}(i, j, k-1)\Big) & \text{otherwise.} \end{cases}.$$

Overall Solution and Time Complexity We solve $\text{opt}(l, m, n)$ There is a total of lmn subproblems, each of which can be solved in constant time. The time complexity is thus $O(lmn)$.

6.8.3 Shortest Common Supersequence

Problem Consider two sequences s, s' with n, m elements of form a_i, b_i respectively.

The task is to find a shortest common supersequence of S and S'. That is, thne shortest possible sequuence of S, S' such that s, s' are subsequences of the supersequence.

Solution Find the longer common subsequence of s, s'. Then, add back differing elements in the right places, in any arbitrary, compatible order.

6.9 String Matching

Problem You are given two strings of length A, B of length n, m. You can perform the following operations on the strings:

- insert a character for cost c_I,
- delete a character for cost c_D,
- replace a character for cost c_R.

Task The task is to find the lowest cost total transformation of A into B.

Levenshtein Distance This is calculated when all costs to insert, delete and replace are equal. As such, the Levenshtein distance is the minimum number of operations needed.

Consider Prefixes Consider prefixes $A_i = A[1..i]$, $B_j = B[1..j]$. To transform $A_i \to B_j$, the following options exist

- Delete $A[i]$ and tranform $A[1..i-1] \to B[1..j]$,
- Transform $A[1..i] \to B[1..j-1]$ and append $B[j]$
- Transform $A[1..i-1] \to B[1..j-1]$. If neccesary, replace $A[i]$ with $B[j]$.

If $i = 0$ or $j = 0$, we only insert or delete respectively.

Subproblems For all $0 \leq i \leq n$ and $0 \leq j \leq m$, let $P(i, j)$ be the problem of determining $\text{opt}(i, j)$. That is, the minimium cost of transforming the squeuce $A[1..i]$ into the sequence $B[1..j]$.

Then,

$$\text{opt}(i,j) = \min \begin{cases} \text{opt}(i-1,j) + c_D \\ \text{opt}(i,j-1) + c_I \\ \begin{cases} \text{opt}(i-1,j-1) & \text{if } A[i] = B[j] \\ \text{opt}(i-1,j-1) + c_R & \text{if } A[i] \neq B[j] \end{cases} \end{cases}.$$

There are also base cases where $\text{opt}(i, 0) = ic_D$ and $\text{opt}(0, j) = jC_I$.

Overall Solution The overall solution is $\text{opt}(n, m)$, leading to a total of mn subproblems. Each subproblem can be solved in constant time. Thus, the total time complexity is $O(mn)$.

6.10 Maximising an Expression

Problem Consider a seqeunce of numbers with operations $+, -, \times$ in between.

The task is to place brackets in such a way that the resulting expression has maximal value.

Link to Matrix Multiplication This problem is effectively the same as the *matrix chain multipliatition* problem. It is not just enough to solve for prefixes $A[1..i]$ but, all subsequences.

Cases with Operations Suppose that the sequence is of form $A \circ B$ with $\circ in\{+, -, \times\}$. We want to choose between whether A, B are maximised or minimised, depending on the value of $\circ$:

- $\circ = +$: A, B are maximised.
- $\circ = \times$: A, B are maximised.
- $\circ = -$: A is maximised, B is minimised.

Therefore, it it not enough to just care for the maximum values but, also the minimum values.

Solution Left as an exercise to the reader.

6.11 Shortest Path in Directed, Acyclic Graph

Notation: Direct Acyclic Graphs Define a DAG to be a directed acyclic graph.

Definition: Topological Sort Recall that a *topological sort* of a DAG is a ordering of vertices where all the edges point *left to right*.

Also note that:

- This can occur only if, the graph is acyclic.
- The solution of a topological sort may not be unique.
- A topological sort can be found in linear $O(|V| + |E|)$ time.

Problem Given a DAG G, find the shortest path from s to t.

Shortest Path: Standard Approach If all edge weights are positive, we can use Dijkstra's algorithm in $O(m \log n)$. For directed acyclic graphs however, this may be done in $O(n + m)$ time.

Subproblems For all $t \in V$, let $P(t)$ be the problem of determining $\text{opt}(t)$. That is, the length of the shortest path from s to t.

Then, for all $t \neq s$,

$$\text{opt}(t) = \min\{\text{opt}(v) + w(v, t) : (v, t) \in E\}.$$

The trivial base case is where $\text{opt}(s) = 0$.

Overall Solution The overall solution is given by $\text{opt}(t)$ where at each step, a predecessor must be stored. There are n total subproblmes. However, we only do this for all m edges and, each edge only gets used once. Thus, the real bound is $O(m)$.

Order of Problems $\text{opt}(t)$ relies on solving all $\text{opt}(v)$ where there is an edge from v to t. As such, we can solve in the topological order from left to right.

6.12 Assembly line scheduling

Problem You are given two assembly lines, each of n workstations. The k-th workstation performs the k-th job, out of n.

- To bring a new job to the start of the assembly line i takes s_i units of time.
- To retrieve a finished product from the end of assembly line i takes f_i units of time.

- On the assembly line i, the k-th job takes $a_{i,k}$ units of time to complete.
- To move a product from station k on assembly line i to station $k+1$ takes $t_{i,k}$ units of time.
- There is no time required to continue from station k to station $k+1$ on the same assembly line.

The task is to find the fastest way to assemble a product, switching lines as necessary.

Topological Sort Denote vertices (i, j) to be the k-th workstation on assembly line i. The problem is effectively requires finding the shortest path from start node to end. Trivially, the graph is directed and acyclic, as the product only moves forwards.

The topological sort is trivially

$$\text{start}, (1,1), (2,1), (1,2), (2,2), \ldots, (1,n), (2,n), \text{end}.$$

Hence, we can use dynamic programming over the topological sort to find the shortest path.

Time Complexity There are $2n+2$ vertices and $4n$ edges. Thus, the DP should take $O(n)$ time. Compare this to dijkstra's algorithm, which takes $O(n \log n)$.

Recurrence For $i \in \{1,2\}$ and $1 \leq k \leq n$, let $P(i,k)$ be the problem of determining $\text{opt}(i,k)$. That is, the minimal time taken to complete the first k jobs, with the k-th job being performed on assembly line i.

Then,

$$\begin{aligned}\text{opt}(1,k) &= \min(\text{opt}(1,k-1), \text{opt}(2,k-1) + t_{2,k-1}) + a_{1,k} \\ \text{opt}(2,k) &= \min(\text{opt}(2,k-1), \text{opt}(2,k-1) + t_{1,k-1}) + a_{2,k}.\end{aligned}$$

The base cases are where $\text{opt}(1,1) = s_1 + a_{1,1}$ and $\text{opt}(2,1 = s_2 + a_{2,1})$.

6.13 Single Source Shortest Paths

Problem There is a directed weighted graph $G = (V, E)$ with edge weights e. Edges may have negative weight but, cycles must have positive total weight. Also, let $n = |V|, m = |E|$.

Let $s \in V$. Find the shortest path from s to all other vertices t.

Dijkstra Fails Allowing negative weights means that the greedy strategy does not work. Observe though, that cycles must not have negative weight. This is because there is otherwise no finite shortest path as it may endlessly loop over the cycle.

Property: No Cycles and Uniqueness of Vertex Observe that for all $t \in V$, there exists a shortest path $s \to t$ without cycles.

This is true as any cycles in the graph are of positive weight as as such, can be removed to make the weight of the path no worse.

As such, it follows that all shortest paths $s \to t$ can contain any vertex at most once and as such, have at most $n - 1$ edges.

6.13.1 Bellman-Ford

For all vertices t, we find the weight of the shortest path $s \to t$ that consists of at i edges for all $i \in [1, n - 1] \in \mathbb{N}$.