Algorithms

Carl Turner

3rd July 2013

Abstract

This is a set of course notes from *Algorithms*, a course taught in Lent Term 2010 at the Centre for Mathematical Sciences, Cambridge. The course was delivered by Sean Lip, Freddie Manners and Ludwig Schmidt. The course homepage is http://www.srcf.ucam.org/algorithms/ - contact details may be easily found on this page. You can contact the author of this set of notes at courses@suchideas.com; the notes may be found at SuchIdeas.com.

Contents

Introduction

The field of algorithms is a fascinating area of study, and a relatively young one at that. Many people find some of the ideas that come from theoretical computer science to be intriguing, and more than once an individual's interest in a particular problem has led to the discovery of a totally new technique. In fact, new discoveries are still frequent, and there is much we do not know, including some issues we will touch upon here.

This course is meant to give a broad outline of the subject, placing particular emphasis on the techniques used in designing algorithms. The format is largely problem-driven: we pick a particular problem, then attempt to solve it, or more often a generalized version thereof. There are exercises in the material which should definitely attempt if you want to really come to understand the material thoroughly.

You will likely find the course homepage very useful if you are keen to learn more about this field. There are lots of resources there which you can use to accompany these notes, or indeed replace them - the videos of the lecture course can be found there. Different people work in different ways: you may like to watch the lectures, then read through the notes afterwards to consolidate the information, or you may like to watch the lectures for the topics you feel less confident about. Besides the lectures, you can also find *example sheets* (the Cambridge name for problem sheets) with problems of a wide range of difficulty, and several extension challenges that were given for students. You should attempt the example sheets after covering the relevant material. You may also like to have a go at the extension exercises; in many cases example solutions from students are given, if you would like to see how other people go about these problems. There is advice on which questions to attempt when at the end of the relevant lectures, as well as above the lecture videos.

Thanks go to the chief lecturer Sean Lip for his commitment to open education, in particular working with me to make these notes good quality and allowing me to distribute them freely, and for making the proofreading (relatively) painless; also, to Matvey Soloviev for some of the diagrams in these notes.

List of Lectures

LECTURE 1: INTRODUCTION

1 Basics

We think of an *algorithm* as a finite sequence of instructions that solves a problem. Often, it is given some input and has to produce an output.

> **Example.** Given a continuous function $F : [0,1] \to \mathbb{R}$, with $F(0) < 0$ and $F(1) > 0$, find an $x \in [0,1]$ such that $F(x) = 0$, with an error at most 10^{-9}.

Remark. When we ask for an answer with error at most ϵ, what we mean is that we want to find some x *nearby* to a true answer. So we need there to be x_0 with $F(x_0) = 0$ and $|x - x_0| < \epsilon$. We can think of this $\epsilon = 10^{-9}$ as the desired precision of the answer.

Note that this is *not* the same as wanting some x with $|F(x) - 0| \leq 10^{-9}$.

Idea: One very useful recurring theme in computer science is the idea of a *binary search*, or in a continuous situation like this, *bisection*.

The key idea is that we want to eliminate as much of the interval as possible at each step. So a sensible approach to solving this would be simply to look in the middle of the interval, that is $x = 0.5$, and see which of the following three cases arises:

- $F(0.5) > 0$ - then we can look more closely at F on the smaller interval $[0, 0.5]$, because $F(0) < 0$ and $F(0.5) > 0$;
- $F(0.5) < 0$ - then we can look more closely at F on $[0.5, 1]$;
- $F(0.5) = 0$ - then we are done.

But in the first two cases, we have just reduced the problem to itself but with different input, so we can start again. This gives rise to the following algorithm:

Algorithm 1 Root-finding by bisection

```
L := 0
R := 1
WHILE ( R-L > 10^-9 )
{
    M := (L+R)/2
    IF ( F(M) = 0 ) THEN
        RETURN M
    ELSE IF ( F(M) > 0 )
        SET R := M
    ELSE
        SET L := M
}
RETURN L
```

There are a few things worth noting about this way of writing down algorithms, especially if the reader is not familiar with so-called *pseudocode*.

(**i**) The notation `:=` corresponds to an *assignment*, so `L := 0` sets the variable `L` in computer memory to the value 0. This is because `=` is reserved for testing for equality.

(**ii**) The `RETURN` statement just states that whatever follows the keyword should be given as the answer.

(**iii**) The `WHILE` loop simply runs the code contained in the braces, `{ }`, while the condition stated holds.

Note that when we eventually return `L`, this is fairly arbitrary, because we *know*[1] there is a root x_0 in $[L, R]$, so the distance from any point in this interval to x_0 is at most 10^{-9}.

Remark. We should think about the speed of this algorithm - we are requiring quite a high level of precision here, so might this not take a long time? The answer is actually no; we can see this by observing that the size of the search interval halves at each iteration. So we need at most say $\log_2 \frac{1}{10^{-9}} \approxeq 30$ iterations to reach this precision. When we consider that a typical computer will carry out 10^8 to 10^9 basic operations[2] per second, we see that this is very fast indeed.

Problem 1.1. Given N pairs (v_i, w_i) representing the *values* and *weights* of books, pick a set S of 6 books such that the *goodness*

$$g(S) = \frac{\sum_{i \in S} v_i}{\sum_{i \in S} w_i}$$

is maximized.

Assume $N = 500$ and v_i and w_i are integers in the range $[1, 10000]$.

Note that in this problem, we are asked for a way of locating an optimal subset of a larger set, a very common type of optimization problem. There is a standard way of obtaining an answer to such a problem, referred to as the *brute force* approach (for obvious reasons!), as follows:

Solution 1 (Brute force). Try all $\binom{500}{6} \approxeq 2 \times 10^{13}$ sets, and select the one with the best goodness.

This has the merit of obviously being a correct answer; but equally obviously there is a serious problem; even if we assume the computer can formulate a set and test its goodness in a single step, we are looking at a runtime of maybe 10^4 seconds, which is around 3 hours. In some sense, we want a *faster* solution.

However, it is not entirely clear how we should go about measuring objectively the speed of an algorithm. There are several problems, one of the most problematic being that we cannot currently compare algorithms independently of the computers they run on. In fact, we would ideally like a measure of the speed which was independent of the implementation, operating system and so on.

But first, we must decide how to get an objective measure of speed.

[1] Formally, this is a consequence of the Intermediate Value Theorem.

[2] These are the instructions in *assembly languages*, which typically include instructions like addition, multiplication, assignment and comparison tests. Programs can easily have hundreds of thousands or millions of these low-level instructions.

Idea: Consider the number of *basic operations* performed.

Observe that this is necessarily dependent on input size. In fact, the dependence on input size gives a natural way of thinking about how well-written an algorithm is - a bad algorithm 'gets slower more quickly'. This motivates the following key definition[3]:

Definition. The *time complexity* (often, we just say *complexity*) of an algorithm is a function $f(n)$ that gives the number of elementary operations needed to process an input of size n.

By the standards of this definition, what we are interested in is not the behaviour of f for any particular values of n, or indeed for small n - instead, we want to know the type of behaviour f exhibits as n grows (its asymptotic behaviour).

A very useful notation for talking about behaviour of functions in the limit of large n is usually called *big O notation*, defined by the following:

Definition. We write $f(n) = O(g(n))$ if there are constants C and n_0 such that

$$f(n) \leq Cg(n) \quad \forall n > n_0$$

We say f is 'big O' of g.

Remark. A more abstract way of viewing definitions such as this arises from thinking of the statement 'f is big O of g' as really meaning 'f is in the class of functions dominated by g'. In this formalism, you would write $f \in O(g)$ instead of $f = O(g)$. Then, for instance, $O(n^2) \subset O(n^3)$. One reason this is better is the asymmetry of writing something like $n + O(n^2) = O(n^3)$. More rigorously, one would write $n + O(n^2) \subset O(n^3)$ where the left-hand side is defined as $\{n \mapsto n + f(n) : f \in O(n^2)\}$.

This is simply saying that f is bounded above by some constant multiple of g for all sufficiently large n, which corresponds very closely to what we want to know about the behaviour of some time complexity f - we do not, in theory, care much about the constant factor (since this in general will depend on the specific computer, language, libraries used, and so on) or about small cases.

Of course, in practice, these do matter, since some algorithms, whilst having better asymptotic performance, have such a large constant factor that they are impractical for most imaginable input; and similarly, some algorithms with worse asymptotic performance will be used in preference because of simplicity, or smaller constant factors. (An example of this is the widespread simplex algorithm for solving linear programming problems, which has a worst-case time complexity which is *exponential*, but is often used in preference to *polynomial* algorithms - though this is partly due to the fact that the simplex algorithm only runs in exponential time on a very small subset of problems.)

[3]The definitions of ideas from complexity analysis in this course are intentionally fairly loose, being included so that we can develop a practical understanding of the comparative speed of different ways of solving the same problem. Formally, we should define 'elementary operations', and the size of the input. (The latter is actually usually viewed in terms of the *number of bits of input*, but we shall not be careful about ensuring this is actually what we are using.)

Example. We can use this notation to greatly simplify expressions of complexity; for example

$$3n^2 + 100n + 5 = O\left(3n^2\right) = O\left(n^2\right)$$

Note that this loses a lot of information, but allows us to make crude (but in practice, very useful) comparisons of the efficiency of algorithms.

Exercise 1.2. Find the values of $f(n) = \log n, n, n\log n, n^2, n^3, 2^n, n!$ for each of $n = 10, 10^2, \cdots, 10^6$. Now find the largest $n^\star$ for which $f(n^\star) \leq 10^{10}$.

This exercise (particularly the second part) shows that the first 5 complexities grow much more slowly than the last two (for which we cannot go beyond double figures in $n^\star$ before exceeding the 10^{10} limit). This is borne out even more strikingly for larger upper bounds, where one can also see that any polynomial power n^k is eventually massively exceeded by the exponential and factorial expressions[4].

Definition. An algorithm is *efficient* if $f(n) = O(p(n))$ for some polynomial $p(n)$ - the algorithm is said to run in *polynomial time*.

In this course, we will almost always consider the *worst-case* time complexity - that is, we always use a function $f(n)$ which gives the *maximum* possible number of elementary operations needed over all inputs of size n.

Remark. Several other types of analysis are possible, including *average-case* time complexity (the expected time taken for a randomly generated legal input of size n) and something more sophisticated called *amortized analysis*, which gives a guaranteed worst running time per operation over a worst-case sequence of operations - as a result, is it only useful in problems with some persistent state (typically data structures like binary trees) where a particular call may take a long time, but on average later calls will be more rapid, an effect which we would like to see reflected in the complexity analysis.

Now note that applying these ideas to our brute-force solution above, and treating N as the only variable, we see

$$\binom{N}{6} = \frac{N(N-1)\cdots(N-5)}{6!} = O\left(N^6\right)$$

so technically our solution is efficient; however, for an input as large as $N = 500$, we have already seen that this is really quite slow.

(We could let the number of books B we want be a variable too; then the algorithm would be $O\left(\frac{1}{B!}N^B\right)$ which can be very bad for some combinations of N, B - for example, if we always take $N = 2B$, this is considerably worse than 2^B.)

So the question is - is there a considerably faster solution? Here is one idea:

Solution 2? Find $\frac{v_i}{w_i}$ for each book, and choose the books with the six largest goodness ratios.

[4] One way of thinking about this is that $e^x = 1 + x + \frac{1}{2}x^2 + \frac{1}{3!}x^3 + \cdots + \frac{1}{n!}x^n + \cdots$, so all polynomials are eventually exceeded by some term in an exponential series.

Complexity:

- Calculating v_i/w_i for each book, $O(N)$
- Picking the best six books[5], $O(N)$
- Total, $O(N)$

So this certainly seems to be a much faster algorithm - in fact, it is blindingly fast - but there is one small problem. It does not actually work!

Example. For a counterexample, consider the small case for $N = 3$ where we want to pick the best 2 books, and we have

$$\begin{aligned}(v_i, w_i) &= (1,1) \to 1 \\ &\quad (3,1) \to 3 \\ &\quad (4,3) \to \frac{4}{3}\end{aligned}$$

Then our suggested solution chooses $(3,1)$ and $(4,3)$, giving a goodness of $7/4$. But taking $(1,1)$ and $(3,1)$ gives $4/2 = 2 > 7/4$ so the given solution is suboptimal. (It is easy to extend this to the $(500,6)$ case - try it!)

Exercise 1.3. Can you come up with a correct algorithm for solving this problem?

The reader should attempt this exercise before continuing, since a correct solution is given at the beginning of the next lecture.

The algorithms given below were suggested by students taking the course:

Exercise 1.4. All of the following algorithms are efficient, but which are correct? Give a proof or counterexample for each algorithm.

(i) *The Mohrmann-Maas algorithm*: Start with any seven books. Compute all goodness values for each of the seven possible sets of 6 out of the 7, and find the best set. The one book which is not in that set is cast out entirely and never seen again. Repeat for each of the other 493 books (by choosing one, adding it to your current set to get a 7-set, and casting out the book that doesn't contribute to the set with maximum goodness). The set of 6 left at the end will be the best one.

(ii) *Zhu's algorithm*: Find the goodness of each book. Pick the best one. Then find the combined goodness of this best book and another book. Pick the second book which yields the highest combined goodness. Then find the combined goodness of these two books and another book. And so on, until you get to six books.

(iii) *Bell's algorithm*: Find the sum of all values and the sum of all weights for all N books. Try removing each book in turn and see which set of $N-1$ books has the highest goodness. Remove the book not in this set. Repeat until there are only six books left.

[5] We can do this by running through the list once, and keeping a list of the 'best 6 so far', a standard technique for doing this - alternatively, we could just sort the list (which we can do slightly more slowly in $O(N \log N)$ as we shall see later - 1.7) and pick the first 6 items.

You should be able to attempt questions 1 to 4 on Example Sheet 1 after this lecture. You might also like to try the Books Problem challenge, as well as the 'Solutions' to the Books Problem challenge as suggested in the last two exercises. If you are interested in programming, you can also attempt Problem 1-1 on ACOS.

Lecture 2: Recursion

Before we continue on to the next fundamental topic in this course in section 1.1, we will consider a (correct!) solution to the books problem from the previous lecture.

Idea: It is difficult to consider many different combinations of books and their goodnesses, because we cannot use information about individual books to efficiently deduce information about all of the sets they make. But if we can rephrase the problem to ask about some condition on the existence of a set of sufficiently high goodness which we can check rapidly, then maybe we can perform a binary search to find the answer.

Solution 3. Consider the related decision problem $P(\lambda)$, which may be stated as

> Given a threshold value λ, can I achieve $g(S) \geq \lambda$?

The answer is true if and only if there is some set S with

$$\sum_{i \in S} (v_i - \lambda w_i) \geq 0$$

So we just need to find the 6 books with the largest $v_i - \lambda w_i$ and check if these have a sum at least 0; this then tells us precisely the result of $P(\lambda)$. *Complexity*: $O(N \log N)$ if we sort, or $O(N)$ if we use a running buffer as before, for small fixed target numbers of books[6].

This is the algorithm for determining $P(\lambda)$, and the complexity of this for one specific λ.

Then we can binary search on λ, as $P(\lambda)$ is equivalent to $g^\star \geq \lambda$. Initially, we can take $g^\star \in [0, {}^{60000}/_{6}] = [0, 10000]$ (which actually would allow for $v_i = 0$). Now, how fine must we make our search, to guarantee we obtain a unique result? Well, consider two fitnesses, ${}^{a}/_{b}$ and ${}^{c}/_{d}$. Their difference is

$$\frac{a}{b} - \frac{c}{d} = \frac{ad - bc}{bd}$$

and we know $b, d \leq 6 \cdot 10000$.

So we can see that the possible values of $g^\star$ are at least $\frac{1}{60000^2}$ apart, which gives us the terminating condition. In turn, this gives an upper bound on the runtime of the binary search: we will need, for the given numbers, around

$$\log_2 \left(\frac{10000}{\left({}^{1}/_{60000}\right)^2} \right) \approxeq 46$$

iterations. *Complexity*: $O(46N \log N) = O(N \log N)$.

Finally, when $g^\star$ has been determined, running $P(g^\star)$ once gives us the desired set of books.

Remark. The complexity analysis is not entirely complete; the constant factor clearly depends significantly on the ranges of the values and weights and the number of books required. If we let values be

[6] Whilst running buffers are in general $O(N)$ for a fixed number of target books, the hidden constant factor grows fairly raipdly when we increase the number of items to be chosen. (You may like to check this.) You may be interested in http://en.wikipedia.org/wiki/Selection_algorithms#Selecting_k_smallest_or_largest_elements, which describes various algorithms for solving this problem with better complexity than sorting the whole list or keeping a running buffer.

integers $v_i \in [1, V]$ and we let weights be integers $w_i \in [1, W]$, and look for B books, then we have complexity

$$\begin{aligned} O\left(\log_2\left[\frac{V}{(1/BW)^2}\right] N \log N\right) &= O\left(N \log N \left[\log\left(VB^2W^2\right)\right]\right) \\ &= O\left(N \log N \log\left(VBW\right)\right) \end{aligned}$$

where we have dropped the base on the logarithm. (Why is this legitimate?)

This is clearly an efficient solution from all points of view. The only things we might be worried about are

(**i**) the dependence of the running time on V and W;

(**ii**) the need for v_i and w_i to be integers.

The first point is a common feature of approaches like this, where we use mathematical properties of the problem to come up with a clever solution - we introduce new dependencies. This actually is reasonably acceptable in most physical or real-world applications, since there are upper bounds imposed effectively by common sense. Similarly, we can actually use rational v_i and w_i if we multiply through by lowest common denominators (and there is unlikely to be a situation where we need true irrational values of weights) - the problem here is that V and W are similarly increased. Fortunately, since the dependence on V and W is only logarithmic, this is not very problematic.

In general, this solution is actually very fast for the scale of problem we were asked to solve, and fairly easy to implement. It gives a good example of the essentially two-stage process of *algorithm analysis*:

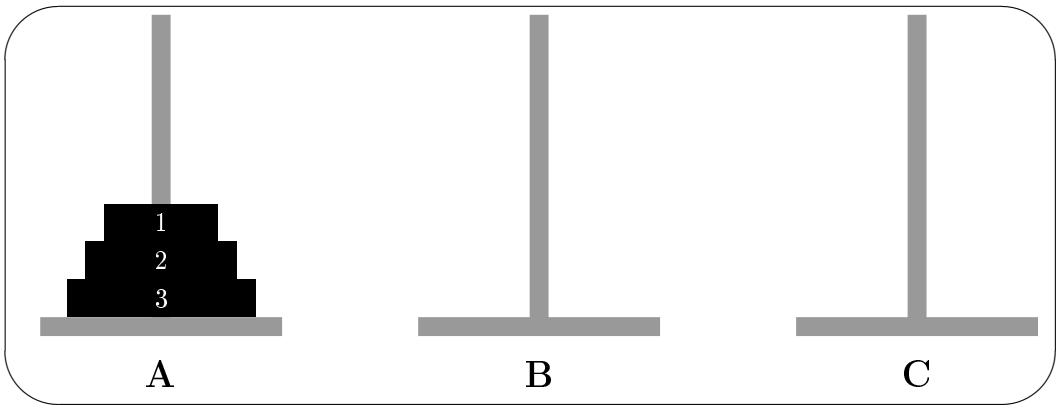
(**i**) Prove correctness.

(**ii**) Evaluate complexity.

1.1 Recursion

The 'books' example shows how large, difficult problems can be solved efficiently by breaking them into subproblems of a known or at least simpler nature - in the above case, a binary search problem (which forms the structure of the solution) and the main subproblem (answering the decision problem $P(\lambda)$). In this section, we look at a different way of reducing problems.

Problem 1.5 (The Tower of Hanoi)**.** In this well-known problem, we are given three pegs, A, B and C, and N discs stacked in order of size on peg A. For example, with $N = 3$ we begin with the layout below. The task is to move all of the discs from A to C, one at a time, and respecting the rule that *larger discs may not be placed on top of smaller discs.*



Clearly, at some point we need to get the largest disc from the bottom of A onto the bottom of C. But to do this, we have to get rid of all of the other smaller discs first.

Idea: Solve the problem *recursively*, by moving everything except the largest disc from A to B; then moving the largest disc from A to C; and finally moving the remaining discs from B to C.

The first and third steps here are effectively solving the problem for $N-1$ discs. This leads to the following pseudocode algorithm:

Algorithm 2 Solve the Tower of Hanoi problem

```
SolveHanoi(N, start, end):

    IF ( N > 1 ) THEN
        Let ‘other’ be the peg other than ‘start’ and ‘end’
        SolveHanoi(N-1, start, other)
        PRINT Move disc ‘N’ from ‘start’ to ‘end’
        SolveHanoi(N-1, other, end)
    ELSE
        PRINT Move disc 1 from ‘start’ to ‘end’
```

Remark.

(**i**) The `PRINT` command just outputs the text after it to the screen, with `'N'`, `'start'` and `'end'` replaced by their respective values.

(**ii**) To solve the problem, we call `SolveHanoi(N, A, C)`.

We can easily check that this is a valid answer by noting that

- we only move discs when we know there is no disc on top of them;
- we only move disc N onto a peg when all the smaller discs are out of the way.

This shows clearly the characteristic structure of a so-called *recursive* solution. We have:

(**i**) a function called from within its own body, the *recursive call*; and

(**ii**) a *terminating condition* or *base case*, which prevents an infinite descent.

To calculate the time complexity, it is convenient to define T_N to be the number of moves made by this algorithm; clearly[7] the only real work done is in the moves (or rather, the print instructions) which take $O(1)$, so the algorithm is $O(T_N)$.

Now $T_1 = 1$, and $T_N = 2T_{N-1} + 1$, and solving the recurrence relation gives

$$T_N = 2^N - 1$$

and hence the algorithm is exponential with base 2 - that is, the time complexity is $O\left(2^N\right)$.

This may not be fast, but it is in fact optimal, as may be shown:

Exercise 1.6. Show that any solution to the Tower of Hanoi problem takes at least $2^N - 1$ moves. *Hint*: Working inductively, you can show that the number of required moves, R_N, obeys the same recurrence as T_N.

We now move on to look at a more complicated problem which admits an efficient recursive solution.

Problem 1.7. Given N numbers, sort them quickly.

For a benchmark, we consider the naïve solution given by implementing what a human would probably usually do:

Solution 1. Find the smallest item in the list, append it to the new list, and repeat. *Complexity*: The obvious implementation is $O\left(N^2\right)$, as may be seen by creating the list where the smallest item is always in the 'last place the algorithm looks'.

A cleverer approach creates a recursive solution by noting that sorting a smaller list is 'easier' than sorting a longer list.

Solution 2 (Merge sort). If we were to halve the list, and sort the two halves recursively, then we would be left with the simpler problem of combining two sorted lists into one.

The base case is a list of 0 or 1 elements, as in either case the list is already sorted.

Schematically, leaving aside the subproblem of working out how to `Merge` two lists, we have the following algorithm (using |`L`| to denote the size of a list):

[7] Actually, in implementation, a function call - like the two recursive calls here - involves some overhead. We can avoid this complication by assuming such an overhead takes constant time, and then noting that for every move, there are at most two recursive calls, so the overhead is absorbed into the constant factor.

Algorithm 3 Merge sort

```
MergeSort(L):

    IF ( |L| > 1 ) THEN
        L1 := left half of L
        L2 := right half of L
        S1 := MergeSort(L1)
        S2 := MergeSort(L2)
        A := Merge(S1, S2)
        RETURN A
    ELSE
        RETURN L
```

Again, the characteristic structure of a recursive solution is clearly shown by this algorithm. The only tasks remaining are to write `Merge` and work out the overall complexity, since this is clearly a correct solution.

`Merge` is actually fairly straightforward - at any given point, we either add the next element from the first list or the second list, so all we need to do is to step through the lists simultaneously:

Algorithm 4 Merge sort, `Merge` sub-problem

```
Merge(L1, L2):

    M := empty list
    WHILE |L1| ≠ 0 or |L2| ≠ 0

        X := smaller of first elements of L1 and L2 (L1's if they are equal)
        Remove X from its list
        Append X to M
    RETURN M
```

Remark. In implementation, we would probably not actually remove elements from the lists to be merged, since this might be expensive; we would just keep variables recording how far through each list we have reached so far. However, we cannot easily do `Merge` actually *in place* (without a third list, `M`) so for now we shall settle for the above algorithm.

Complexity

The interesting part of the merge-sort problem is how to go about calculating its complexity. The usual approach is to begin with an analysis of any subroutines used - in our case, this is the `Merge` procedure.

`Merge(·,·)` does a constant amount of work per list element, so the time complexity is $O(|L_1| + |L_2|)$.

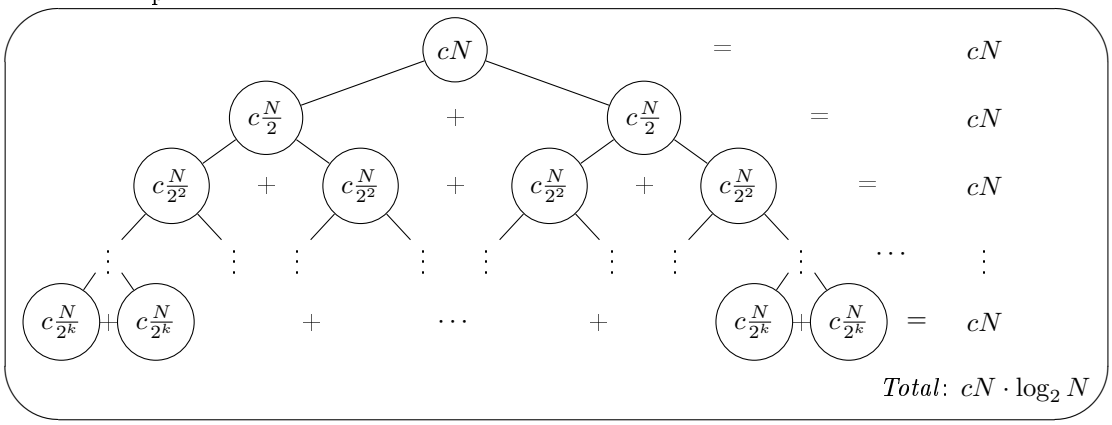
Now let `MergeSort` takes time T_N to sort a list of length N. Then

$$T_N = T_{\lceil N/2 \rceil} + T_{\lfloor N/2 \rfloor} + cN$$

for some constant c, since the procedure is called recursively with lists of size as given (roughly halving the input size) and some $O(N)$ work is done.

As this suggests, this will become fiddly to analyze precisely when N is not a multiple of 2, so here we will assume $N = 2^k$ for some k. It is reasonably clear that actually the algorithm will not perform much worse than indicated (in terms of complexity) in the other cases, but for a fuller analysis the reader is directed to the book *Introductions to Algorithms*[8] or any of the many places online where the general case is considered.

In this simpler case, however, we have simply $T_N = 2T_{N/2} + cN$. This scheme can be solved using a so-called *recursion tree*. The idea is that we calculate the time spent in total at each level of recursion, and add them up afterwards:



The final line comes from noting there are $\log_2 N$ levels in the tree, since then the final nodes all have $N/2^k = 1$ item.

Hence we have the result

$$T_N = O(N \log N)$$

Note that for large N we have $N \log N \ll N^2$ so that this is a significant improvement on the simple algorithm given in the first solution.

In fact, it is not very difficult to show this is precisely the best complexity that can be achieved in the general case of a sort where the keys to be compared can take on continuous values.

Exercise 1.8. Show that any sorting algorithm takes at least $O(N \log N)$ time in the worst case. *Hint*: Every time we make a comparison, we pick a branch of a binary tree (called the decision tree) - how many leaves must the tree have? (See the handout later in the course for definitions relating to trees.)

Remark. By contrast, when the entries to be sorted can only take on a (small) finite set of values, there are actually more efficient solutions - the curious reader can look up *counting sort* and *radix sort*. Also, *bucket sort* (and more generally *distribution sort*) might be of interest.

In the next lecture, we will generalize the recursion tree methods to a useful theorem that will make these calculations less tedious in the future.

[8] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein *Introduction to Algorithms*. MIT Press 2009.

Lecture 3: Recursion Continued

Many recursive algorithms are based on the *divide and conquer* technique used in merge sort, and as a result have associated recurrences of the form

$$T_N = aT_{N/b} + f(N)$$

where a and b are constants. In merge-sort, $a = 2$, $b = 2$ and $f(N) = cN$.

Whenever $af(N/b) \propto f(N)$ we have the following theorem:

Theorem 1.9 (The Master Theorem)**.** *Suppose* $T_N = aT_{N/b} + f(N)$ *and* $af(N/b) = kf(N)$ *for some constants* a, b, k*. Then*

(i) if $k < 1$, $T_N = O(f(N))$

(ii) if $k = 1$, $T_N = O(f(N)\log N)$

(iii) if $k > 1$, $T_N = O\left(N^{\log_b a}\right)$

Remark. Note this statement of the Master Theorem gives only upper bounds - although you can see that our analysis below allows us to calculate T_N explicitly (where N is a multiple of b) - and assumes f has a very particular form. It is actually possible to give accompanying lower bounds, even with weaker restrictions on f. The Wikipedia article at http://en.wikipedia.org/wiki/Master_theorem may be of interest.

To prove this, we basically imagine drawing a general version of the recursion tree from the previous lecture.

Proof. From the recursion tree, once again assuming that N is a power of b, we see

$$\begin{aligned} T_N &= f(N) + af(N/b) + a^2 f(N/b^2) + \cdots + a^{\log_b N} f(1) \\ &= f(N)\left[1 + k + k^2 + \cdots + k^{\log_b a}\right] \end{aligned}$$

Then we can simply analyze this geometric series to find the answer:

(i) if $k < 1$, then the series is $O(1)$ - as may be seen by the closed form of the sum - so $T_N = O(f(N))$.

(ii) if $k = 1$, then all terms in the series are equal, and there are $\log_b N$ of them, so $T_N = O(f(N)\log N)$.

(iii) if $k > 1$, then the sum is dominated by its largest term, so

$$\begin{aligned} T_N &= O\left(f(N)k^{\log_b N}\right) \\ &= O\left(a^{\log_b N} f(1)\right) \\ &= O\left(a^{\log_b N}\right) \\ &= O\left(N^{\log_b a}\right) \end{aligned}$$

□

This can then be applied to easily deduce a few results we already know:

Example (Merge sort)**.** $T_N = 2T_{N/2} + cN$.

We have $a = 2$, $b = 2$ and $f(N) = cN$. Since $2c \cdot \frac{N}{2} = cN$ we have $k = 1$, and then

$$T_N = O(N \log N)$$

This followed directly from the second case.

Example (Binary search)**.** $T_N = T_{N/2} + c$.

This time, $a = 1$ and $b = 2$, whilst $f(N) = c$. Hence $ac = kc$, so $k = 1$. Then

$$T_N = O(\log N)$$

This was another instance of the second case. Finally, we see an application of the third:

Example (Towers of Hanoi)**.** $T_N = 2T_{N-1} + 1$.

This is not initially in the desired form, because the recurrence is defined by a linear change. We can, however, still apply the master theorem with a suitable transformation.

If we let $N = \log_b M$, then we can define

$$\begin{aligned} U_M &= T_{\log_b M} \\ &= 2T_{\log_b M/b} + 1 \\ &= 2U_{M/b} + 1 \end{aligned}$$

Here $a = 2$ and we have some arbitrary b. Hence $a \cdot 1 = k \cdot 1$ implies $k = 2$, and

$$U_M = O\left(M^{\log_b 2}\right)$$

which implies

$$\begin{aligned} T_N &= U_M \\ &= O\left(b^{N \log_b 2}\right) \\ &= O\left(2^N\right) \end{aligned}$$

As a final note on recursion, we look at another classic example.

Problem 1.10. Write a program to play Noughts and Crosses (Tic-Tac-Toe). Your program should take as input a (legal) position, then output the outcome of the game and an optimal move, if applicable. Assume both players play perfectly.

We are asked to classify a fairly reasonable number of possible boards - fewer than $3^9 = 19683$ as each square can be in at most 3 states: winning, losing or drawing[9].

A configuration is *losing* if all moves from it lead to winning positions (for the other player), and it is *winning* if some move from it leads to a losing position (for the other player). Otherwise, it is a draw.

Remark. One way to view this is as a game tree, where the root of the tree is a blank board, and every node has child nodes for every possible move after it. We are then exploring various branches of the tree, using what is called a *minimax* algorithm.

This leads to the following recursive solution:

Solution. Given a board and the player whose turn it is (say O), check all possible next moves for O recursively, to see if they are wins or losses or draws for X. Then apply the above rules.

This can be implemented by something like the algorithm below, which defines a function `SolveNC` which takes a board position and the symbol of the current player, and returns a pair with either `WIN`, `LOSE` or `DRAW` in the first place, and the best cell to choose in the second place.

You may like to try to come up with the algorithm yourself by following the same two key steps as before:

- Identify what recursive calls need to be made.
- Identify the base cases.

A typical solution is provided below. We write `S` for the symbol of the player whose turn it is, and $\bar{\texttt{S}}$ for the other player's symbol.

Algorithm 5 Solving Noughts and Crosses

```
SolveNC(position P, symbol S):

    IF ( There is a row of three S̄ )
        RETURN (LOSE, -)
    IF ( There is no empty cell )
        RETURN (DRAW, -)
    (bestStatus, bestMove) := (LOSE, first empty cell in P)
    FOR empty cell C in P
        Q := P with cell C filled with S
        (status, move) := SolveNC(Q, S̄)
        IF (status = LOSE)
            RETURN (WIN, C)
        IF (status = DRAW)
            (bestStatus, bestMove) := (status, move)
    RETURN (bestStatus, bestMove)
```

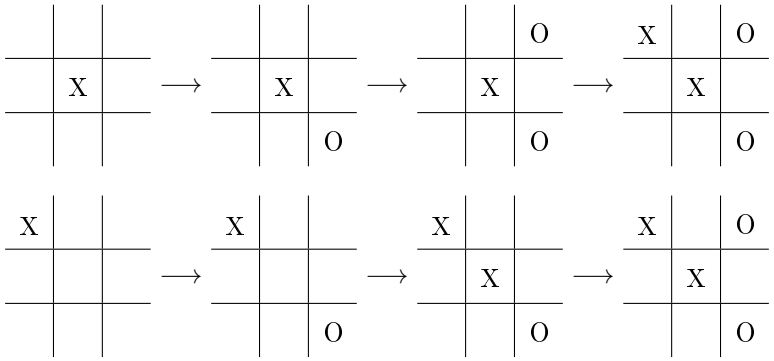
[9] We are classifying states from the perspective of whichever player is to move next - so if O is guaranteed a win, and it is X's turn, then the board is in a *losing* state.

Remark. When the command `RETURN` is made, the assumption is that execution stops there. This avoids wrapping large amounts of code in nested `ELSE` clauses.

This algorithm is more complicated than the previous examples for two main reasons. Firstly, it has two base cases. Secondly, it iterates over some variable number of recursive calls.

The basic idea, however, is still the same, and it is quite easy to see how the algorithm reflects the earlier description of it.

However, there is a problem with this algorithm - namely inefficiency due to redundancy. The algorithm will examine (given an empty board) something on the order of $9! = 362880$, which is over ten times larger than 3^9, because it will examine the same state repeatedly if it arrives at it from a different sequence of moves. For example, the possibilities for the right-hand configuration here are explored at least twice:



The obvious solution to this problem is the key to efficient recursion in many problems:

Memoization. Store the results of recursive calls so that if the same call if made again, we just look it up and return it straight away.

Schematically, this would amount to defining a table (two-dimensional array) `MEMO[P][S]` containing `(result, move)` pairs. By default, let `MEMO[P][S]` have the value `NULL` where `NULL` is a value indicating that the variable has not yet been assigned a value. We would end up having something like the following:

Algorithm 6 Solving Noughts and Crosses with memoization

```
SolveNCMemoized(position P, symbol S):
    IF (MEMO[P][S] = (NULL, NULL))
        MEMO[P][S] := SolveNC(P, S)
    RETURN MEMO[P][S]
SolveNC(P, S):
    See previous solution
```

Remark. In this case, we could actually further reduce runtime by taking advantage of the inherent symmetry of the board. We might do this by defining a *canonical* version of the board, so that we always rotate/reflect the board to some obey some standard rules before looking it up in `MEMO` and then rotate/reflect the board back to return the correct best cell. One example of such a rule would be - denoting the cells by *E, X* and *O* according to whether they are empty or have a symbol in - to write

the board as a string of nine characters, so that the above final state would be $XEOEXEEEO$; then we could rotate and reflect the board to get this string into the earliest possible position alphabetically (lexicographically), and work with that board instead.

One perhaps less obvious symmetry arises from the fact that the outcome and best move for O given a board position P is the same as the outcome and best move for X given $\bar{P}$ (which is P with the X and O characters swapped). This allows us to just store a one-dimensional `MEMO[P]` array by always assuming that it is, say, X's turn, inverting the board if necessary.

Exercise 1.11. Improve the memoized noughts and crosses algorithm as follows:

(**i**) Devise a routine that checks whether a board is a valid position, and determines which player's turn it is. (Assume one particular player always starts.)

(**ii**) Modify the solution algorithm to take advantage of the rotational, reflective, and player-swapping symmetries.

Lecture 4: Dynamic Programming

The idea of *memoization* that we saw at the end of the previous lecture actually leads to another way of solving problems with some naturally recursive solution - we could build up the memoization table iteratively, filling it in cell by cell according to the recursive rule, and then reading off the final answer. We introduce this idea, called *dynamic programming*, or DP for short, using an example.

1.2 Dynamic Programming

Problem 1.12. Given N coins with positive integer values $a_1, \cdots, a_N$, can we find a subset with total value K? If so, what is the minimum number of coins needed, and what is an example of a minimal set (that is, one with the minimum number of coins)?

As ever, we have a simple brute-force solution:

Solution 1. Try all 2^N subsets. This is clearly impractical for large values of N.

The dynamic programming solution relies on the following observation:

Idea: We either use or do not use the Nth coin. Therefore, we can solve the problem (N, K) (referring to the problem of using $a_1, \cdots, a_N$ to make a total K) if and only if we can solve at least one of $(N-1, K)$ and $(N-1, K-a_N)$.

This leads to the idea of defining a Boolean[10] array, `C[i][j]`, indicating whether we can make the sum j using just the first i coins $a_1, \cdots, a_i$. For now, we will concentrate on the first part of the problem, answering the yes/no question about the value of `C[N][K]`.

Solution 2. We then have the recurrence relation

`C[i][j] = 1` $\iff$ `C[i-1][j-a`$_\texttt{i}$`] = 1 or C[i-1][j] = 1`

This has an obvious base case

`C[0][j] = 1` $\iff$ `j = 0`

Here, `C[N][K]` can be calculated 'top-down' with recursion and memoization as in previous lectures:

Exercise 1.13. Write down a recursive algorithm to calculate `C[N][K]`.

Alternatively, we can do this 'bottom-up' by calculating answers in order of increasing i. This is then an *iterative* solution known as dynamic programming.

It would be implemented by something like the following algorithm (note that we are using the standard assumption that arrays have indices beginning at 0).

[10] A Boolean value is just one that takes the values yes and no; or equivalently true and false or 1 and 0. Named for the mathematician and philosopher George Boole (1815-1864).

Algorithm 7 Making change

```
Define a Boolean array C[N+1][K+1]
C[0][0] := 1
FOR j = 1 to K:
    C[0][j] := 0
FOR i = 1 to N:
    FOR j = 0 to K:
        IF ( (j - a_i ≥ 0 AND C[i-1][j-a_i] = 1) OR (C[i-1][j] = 1) )
            C[i][j] := 1
        ELSE
            C[i][j] := 0
RETURN C[N][K]
```

Complexity: It is straightforward to see from this that the algorithm takes $O(NK)$ time, by simply inspecting all of the loops. Also, it takes $O(NK)$ space, in the form of the array `C`.

It is worth noting the differences between this solution and the recursive solution. The key features of a dynamic programming algorithm are the following:

(**i**) *State space.* This is encoded in the table which the algorithm works with; here, it is $\{0, \cdots, N\} \times \{0, \cdots, K\}$.

(**ii**) *Recurrence relation.* This implicitly comes with the necessary base cases.

(**iii**) *Time ordering.* It is very important that there is some definite order in which we can parse through the table so that we only ever need to refer to values already calculated, and that we do actually use such an order. (It is possible to construct tables which cannot be filled in at all like this.)

Now we must work out how to use the structure of the above solution in order to find first the minimum number of coins necessary for a solution, and then to find such a set.

The first part is achieved fairly easily by considering a new array, `D[i][j]`, giving the minimum number of coins needed to make a sum of j using the first i coins, or ∞ otherwise. We can then simply observe again that either a_i is used, or it is not, and we just need to work out which of the two possibilities is better:

Solution. The recurrence obeyed by `D[i][j]` is simply

```
D[i][j] = min( D[i-1][j-a_i] + 1, D[i-1][j] )
```

The base case is

`D[0][j]` = 0 if `j` $= 0$.
`D[0][j]` = ∞ otherwise.

Then `D[N][K]` is the desired answer.

An example of a DP table for the case $N = 4$, $K = 5$ with $a_i = \{2, 1, 3, 2\}$:

i \ j	0	1	2	3	4	5
0	(0)	∞	∞	∞	∞	∞
1	0	∞	(1)	∞	∞	∞
2	0	1	(1)	∞	∞	∞
3	0	1	1	1	2	(2)
4	0	1	1	1	2	(2)

Remark. The practice, depending on how we implement the algorithm (and with what data types), we may simply store a very large number to represent ∞, or possibly a meaningless value like -1 (and then treat this as a special case in comparisons).

The next challenge is to use this to construct an optimal set.

Two possible ways of building up a valid minimal set are shown, the boxed values taking the third and fourth coins, and the (bracketed) values taking the first and third. How would we go about finding these cells? Well, starting from any given point, we either came from the cell immediately above (i.e. we did not select the current coin) or we came from the cell above and somewhere to the left, corresponding to selecting the current coin. We can just go through the table, selecting an option which satisfies the recurrence every time, and we will have an optimal set. An option which satisfies the recurrence is one which gave the smaller value for the current cell.

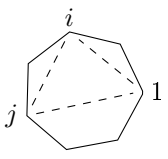
Solution to minimal set problem. Trace back through the table beginning at `D[N][K]`. From cell (i, j), find the smaller of `D[i-1][j]` and `D[i-1][j-a`$_i$`] + 1`. Move to the corresponding cell, including coin i only if the latter (i.e. `D[i-1][j-a`$_i$`] + 1)` gives the smaller result.

Remark. In the given example, the first step has some ambiguity (this is because in the optimal solution, we can take either one of the 2s); it does not matter which way we go at this point. Both solutions will be optimal.

It is important to understand that in a DP solution, subproblems are processed at most once (like recursion with memoization). Hence DP is useful for problems where there are many overlapping subproblems, most of which need to be explored. It eliminates the recursive overhead in these cases (indeed, there is often a limit on the size of the 'stack' which is used when nested functions calls are made). However, it does generally evaluate nearly all possible states, even when it may not be necessary to do so.

Problem 1.14. How many triangulations are there of a convex N-gon?

This problem is equivalent to asking how many sets of $N - 3$ non-intersecting edges there are[11].



This has a naturally recursive feel, since adding some interior diagonal divides the N-gon into two smaller shapes which are also convex. This might lead to the following algorithm:

[11] In fact, the sequence of numbers generated by answering this question are called the *Catalan numbers*.

Solution 1? Let `dp[n]` be the number of triangulations of a convex n-gon.

Recurrence: We argue that vertex 1 must lie in some triangle, so we consider every triangle specified by its vertices $(1, i, j)$, and then subdivide the resulting shapes.

$$\texttt{dp[n]} = \sum_{2 \le i < j \le n} \texttt{dp[i]dp[j - i + 1]dp[n + 1 - j + 1]}$$

Base cases: `dp[3]` and `dp[2]` are both 1. (The latter case is there so that if we pick say $i = 2$, we do not try to subdivide the degenerate polygon which is the line between vertices 1 and 2.)

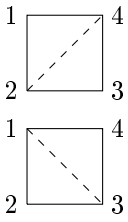
State space: $\{2, 3, 4, \cdots\}$

Complexity: $O\left(N^3\right)$ time, $O\left(N\right)$ space.

The only thing omitted from our analysis is a proof of correctness.

Exercise 1.15. Does the algorithm work? Give a proof or explain what is wrong as appropriate.

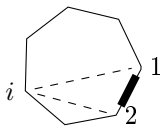
There is actually a good reason for our reticence in giving an answer - the algorithm is incorrect! The problem is an error typical of recursive counting problems - namely double counting. Consider the simple case of a square - there are obviously 2 triangulations. The case where the edge does not pass through the 1 node is handled correctly, as shown.



However, the other case is double-counted, as both triangles $(1, 2, 3)$ and $(1, 3, 4)$ give rise to the same situation. So our algorithm gets the answer wrong.

Idea: Instead of working with vertices then, which have the ambiguity that they can belong to more than one triangle, why not consider edges?

Solution 2. Each edge is used exactly once in each triangulation.



Recurrence: The edge between points 1 and 2 is connected to exactly one vertex, so:

$$\begin{aligned} \texttt{dp[n]} &= \sum_{3 \le i \le n} \texttt{dp[i - 2 + 1]dp[n + 1 - i + 1]} \\ &= \sum_{3 \le i \le n} \texttt{dp[i - 1]dp[n - i + 2]} \end{aligned}$$

Base cases: As before.

Complexity: Interestingly, we have in fact also improved complexity in finding this solution; it is now $O\left(N^2\right)$ in time and still uses $O\left(N\right)$ space.

The Catalan numbers C_n give the number of ways of subdividing a polygon with $n + 2$ sides. You may like to attempt the following exercise:

Exercise 1.16. Show that the Catalan numbers also satisfy

$$C_{n+1} = \frac{2\,(2n+1)}{n+2} C_n, \quad C_0 = 1$$

and derive a closed-form expression for the values in terms of factorials or binomial coefficients.

Lecture 5: Dynamic Programming Continued

In this lecture, we look first at an application of DP to another problem, and then move on to briefly consider ways of improving DP algorithms.

Dynamic Programming on Trees

Informally, an *undirected graph* $G = (V, E)$ consists of a set V of *vertices* or *nodes* and a set E of *edges* joining pairs of nodes.

A *tree* is a special type of graph - a *connected acyclic graph*. This simply means that there is always a route between any two nodes (the graph is *connected*) and there is no cycle in the graph (the graph is *acyclic*). We are also assuming the graph is *simple*, so that at most one edge is allowed between any pair of nodes.

We will look in more detail at graphs in the following lectures - after this lecture, there is a handout containing useful reference information.

Properties of trees:

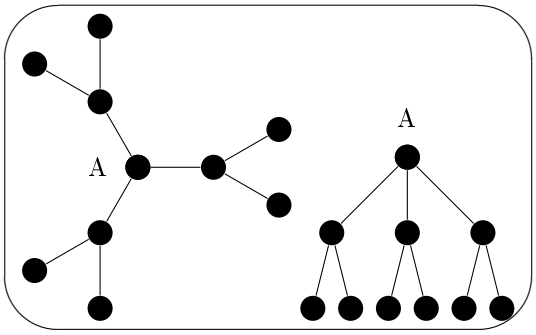
(**i**) There is always one more vertex than edge: $|E| = |V| - 1$

(**ii**) There is a unique path between any two nodes.

(**iii**) Adding an edge to a tree creates a cycle (a tree is *maximal acyclic*).

(**iv**) Removing an edge disconnects the tree (a tree is *minimal connected*).

The second property allows us to 'root' a tree by choosing a particular node, the *root*, and letting the tree 'hang down' from that node, so all other nodes are on branches below it.



Note that in a rooted tree, nodes that are a distance d from the root end up at level d of the rooted tree. Also, all nodes in a tree have a unique parent, except for the root - but the parent might be different for a different choice of the root.

Many problems involving trees are amenable to DP or recursive solutions, because the hierarchical structure inherent to a rooted tree lends itself to these techniques.

Problem 1.17 (Employee Party Problem)**.** Given a rooted tree representing a company hierarchy, what is the maximum number of employees I can invite to a party such that I don't invite both an employee and his immediate boss?

Much like in the example of making change, we have a situation where, for each employee, we choose whether or not to select them, and this affects what we can choose later. Inspired by this similarity, we might come up with the following solution:

Solution 1. Let `dp[i]` be the maximum number of people that can be invited from the subtree rooted at employee i (we almost always number nodes sequentially for convenience).

Recurrence: At this stage, the choice is whether or not we invite employee i, so we wish to make the optimal choice between including i and excluding all i's children, and excluding i and considering all their children. (Note that we may not want to include all their children, or indeed i's grandchildren; but this does not matter, as we simply use the optimal values we have already calculated for these subtrees). Hence

$$\texttt{dp[i]} = \max\left(1 + \sum_{k \text{ grandchild of } i} \texttt{dp[k]}, \sum_{j \text{ child of } i} \texttt{dp[j]}\right)$$

The base cases for this are actually implicit in this definition - if i has no children or grandchildren, the empty sums evaluate to 0, and the optimal choice is still correctly made.

This solution works perfectly well. The only thing worth noting is that we are exploring most levels of the tree twice, to find nodes as both grandchildren and children, and iterating over grandchildren may be slightly awkward programmatically. There is, however, a way around this.

The reason we have the need to explore two generations is because the child nodes have no information to give us about the specific cases where they are or are not included - they only store information for the optimal choice. Thus by storing that additional information, rather than disposing of it in the $\max(\cdot,\cdot)$ statement, we can simplify our algorithm a little.

Solution 2. Let `dp[i][0]` be the maximum number of people that can be invited from i's subtree given i is *not* invited, and `dp[i][1]` be the same value given i *is* invited.

Recurrence: We now have two cases to consider, but they only involve the children of `i`:

$$\begin{aligned}\texttt{dp[i][0]} &= \sum_{j \text{ child of } i} \max(\texttt{dp[j][0]}, \texttt{dp[j][1]}) \\ \texttt{dp[i][1]} &= 1 + \sum_{j \text{ child of } i} \max(\texttt{dp[j][0]})\end{aligned}$$

Again, there is no need to give explicit base cases.

In this case, the answer must be calculated as $\max(\texttt{dp[root][0]}, \texttt{dp[root][1]})$.

Complexity: Suppose the tree has N nodes; then there are $2N$ states to examine. A naïve analysis would say that the number of children of each node is $O(N)$, and there are $O(N)$ nodes to examine, so the algorithm is $O(N^2)$. However, a more appropriate approach is to note that each *edge* of the tree is examined twice (that is, we pass from parent to child twice, once for each sum), so the time complexity is

$$O(N) + O(\text{number of edges}) = O(N)$$

since for a tree, $|E| = N - 1$.

Exercise 1.18.

(i) How would you go about finding the set of employees to invite?

(**ii**) What if you were a member of the organization (not necessarily the boss), and you knew you wanted to invite yourself?

Optimizing DP: Some Techniques

The following are some useful rules of thumb when solving dynamic programming problems, or implementing ideas for solutions - it is often possible to improve on the first approach that comes to mind.

Keep only what you need. Recall that in the coin-change example, `dp[i][j]` only depends on `dp[i-1][·]`. This means that if we do not need to trace back through the tree (i.e. if we do not need to give a set of coins, but only need the number of coins in it), then we will only ever look at the current row and the previous row. Therefore, we need only store these two rows, which reduces space complexity, which was obviously on the order of NK, to $O(K)$. This is frequently a very useful technique when the dimensions of the problem (here, N and K) grow large, as they might do when processing large data sets.

Use fast matrix multiplication. DP algorithms can often be implemented as the multiplication of matrices together. This arises in problems like the following:

Exercise 1.19. (See Example Sheet 1, Question 6.) Let x_n be the number of ways to tile a $2 \times n$ room using only 1×2 dominos. Show that x_n obeys the following recurrence, which could be used for a dynamic programming solution:

$$x_n = x_{n-1} + x_{n-2} \text{ for } n \geq 2, \quad x_0 = 1, \quad x_1 = 1$$

Example 1.20. This problem can be written as

$$\begin{pmatrix} x_{n+1} \\ x_n \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} x_n \\ x_{n-1} \end{pmatrix}$$
$$\mathbf{x}_n = P\mathbf{x}_{n-1}$$

where $\mathbf{x}_n = \begin{pmatrix} x_{n+1} \\ x_n \end{pmatrix}$ is a vector. This obviously has the solution

$$\mathbf{x}_n = P^n \mathbf{x}_0 = P^n \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

so if we could calculate P^n rapidly, this would solvethe problem more efficiently. (Note also that this approach also automatically saves space by discarding intermediate stages of the calculation.)

If one is raising some matrix M to a large power n, it is possible to do much better than $O(n)$ algorithms (omitting the dependence on the dimensions of M):

Exercise 1.21. (See also Example Sheet 1, Question 4.)

(**i**) Assuming that you can do basic operations on arbitrary numbers in $O(1)$ time, show how you can calculation a^n in $O(\log n)$ time, where $n \geq 0$ is a positive interger. *Hint*: Consider the binary representation of n.

(**ii**) Using a similar method, how quickly can you compute the nth power of a $k \times k$ matrix?

In fact, it is possible to find ways to multiply two large square matrices more rapidly than in the obvious implementation. Probaby most famous is the *Strassen algorithm* which gives a complexity of $O\left(k^{2.8}\right)$ when multiplying together two $k \times k$ matrices, for $k = 2^m$ a power of two.

Remark. An alternative way to raise a matrix M to a large power rapidly is to find its eigenvalues and eigenvectors, and diagonalize it, if possible. Then we have $M = PDP^{01}$ where D is diagonal, and we can calculate

$$M^k = \left(PDP^{-1}\right)\left(PDP^1\right)\cdots\left(PDP^{-1}\right) = PD^kP^{-1}$$

and since D is diagonal it is very easy to calculate D^k:

$$D = \begin{pmatrix} \lambda_1 & & & \\ & \lambda_2 & & \\ & & \ddots & \\ & & & \lambda_m \end{pmatrix} \quad \Longrightarrow \quad D = \begin{pmatrix} \lambda_1^k & & & \\ & \lambda_2^k & & \\ & & \ddots & \\ & & & \lambda_m^k \end{pmatrix}$$

Then we can use our fast *scalar* exponentiation to compute D^k.

This method also allows us to deduce closed-form solutions to these recurrences - if you know how to diagonalize a matrix, you may like to attempt the following exercise:

Exercise 1.22. Using this method, show that

$$M = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$$

can be written

$$M = \begin{pmatrix} \Phi & -\frac{1}{\Phi} \\ 1 & 1 \end{pmatrix}\begin{pmatrix} \Phi & 0 \\ 0 & -\frac{1}{\Phi} \end{pmatrix}\begin{pmatrix} \Phi & -\frac{1}{\Phi} \\ 1 & 1 \end{pmatrix}^{-1}$$

where $\Phi = \frac{1+\sqrt{5}}{2}$ is the golden ratio, and deduce that the closed form of x_n is

$$x_n = \frac{\Phi \cdot \Phi^n + \frac{1}{\Phi} \cdot \left(-\frac{1}{\Phi}\right)^n}{\sqrt{5}} = \frac{\Phi^{n+1} - \left(-\frac{1}{\Phi}\right)^{n+1}}{\sqrt{5}}$$

Transforming a permutation problem to a subset problem. This is best seen by considering a classic problem, known as the *Travelling Salesman Problem*, or *TSP*.

Problem 1.23 (Travelling Salesman)**.** There are N towns, and each pair of towns is connected by a road of known length. Find the minimal length of a tour that visits each town exactly once.

The brute-force approach is this case has a truly awful complexity:

Solution 1. Without loss of generality, pick a start point C, and try all tours beginning from here. There are $(N-1)!$ permutations of the other towns, and calculating the length of each corresponding tour takes $O(N)$ time. Hence the complexity is $O(N!)$.

To come up with a dynamic programming approach here, we need to find some way of reducing the problem to a (time-ordered) stage by stage process.

The first way one might think of is to find the best way of visiting town 1; then town 1 and town 2; then towns $1, 2, 3$; ... However, this is useless, because the previous stages are of next to no use in calculating the next! (To see why this is, imagine that all the towns $1, 2, 3, 4, 10, 5, 6, 7, 8, 9$ arearranged in that order along a straight line - knowing that visiting towns 1 through 9 in that order is optimal is no use in finding when we should visit 10.)

To remedy this, we might generalize the basic idea as follows: we find the best return-leg of a tour through arbitrary sets of points, and with various arbitrary starting points. This avoids the problem of privileging some particular sequence as we did before. Then we can say that the best route back to, say, town 1 through some selection of m towns starting at i is

> the minimum length (over all towns j not yet visited) of a tour going first from i to j, and then from j back to 1 via the remaining $m-1$ towns.

Solution 2. Let `dp[B][i]` be the minimum length of the end of a tour from i to 1, given that we have *already visited* all the towns in the set[12] B exactly once. We have $N \cdot 2^N$ states.

Recurrence: We are going to be taking the minimum over all towns j we have yet to visit - so $j \notin B$ - and finding the optimal tour given that we have now also visited j. Hence

$$\texttt{dp[B][i]} = \min_{j \notin B} \left(\texttt{dp[B} \cup \{\texttt{j}\}\texttt{][j]} + \texttt{length(i, j)}\right)$$

Base cases: Clearly, for any $i \neq 1$, `dp[all towns][i]` is just `length(1, i)`. In fact, this also works for $i = 1$, because then the length is 0, as it should be.

Answer: The answer will be given by `dp[{1},1]`.

Time ordering: It is important to verify this algorithm has some valid time-ordering - in fact, we can see it does, because `dp[B][·]` only ever depends on entries `dp[B'][·]` where $B' \supsetneq B$ contains more entries than B. So if we first consider all sets B containing n towns, then $n-1$, then $n-2$ and so on, we get a valid time ordering.

Complexity: We have $2^N N$ states, and each takes $O(N)$ time to process. Hence the time complexity of this approach is $O\left(2^N N^2\right)$.

Remark. This second solution is significantly better than the first, but it is still exponential-time. Finding a polynomial-time solution, or even proving that there is no such algorithm, is a major open

[12] We can formalize this using the idea of a *bitmask* for efficiency when N is reasonably small - we write B as an N-bit binary number, where the jth bit from the right is equal to 1 if we have already visited town j. For example, $B = 01100101_2$ would indicate we have visited towns 1, 3, 6 and 7, and there are $N = 8$ towns in total. Note that we are mapping town 1 to the right-most bit, so this is not zero-indexed.

problem, though it is true that arbitrarily accurate polynomial-time algorithms exist for the Euclidean version of the problem[13]. This is closely related to the so-called *P vs NP* problem, one of the Clay Institute's $1m prizes.

You should be able to attempt questions 5 to 8 on Example Sheet 1 after this lecture. If you are interested in programming, you can also attempt Problems 1-2 and 1-3 on ACOS.

[13] There is a *polynomial-time approximation* scheme, or *PTAS*, for the TSP given a set of points and distances obeying all the rules of Euclidean space - the problem called the *Euclidean* or *planar TSP*. It gives a tour of length at most $(1+\epsilon)\,L$ where L is the optimal length in polynomial time - for fixed ϵ. (See http://www.cs.princeton.edu/ arora/publist.html). Unfortunately, the dependence on ϵ in general makes these schemes very expensive, so one can look for an 'efficient' or even 'fully' polynomial-time approximation scheme. Often, the special case $\epsilon = 1$, giving the *2-approximation*, is particularly easy to calculate, and the *metric* TSP (where we require only the triangle inequality) admits a fairly simple and fast 2-approximation.

Handout: Graph Terminology

An **undirected graph** $G = (V, E)$ comprises a set V of **vertices** (or **nodes**) and a set E of **edges**. The elements of E are unordered pairs of vertices and are denoted ij, where $i, j \in V$. (The element ij of E represents an edge between the nodes i and j.) We write $V(G)$ for the set of vertices of G, and $E(G)$ for the set of edges of G.

In a **directed graph**, the edges are one-way (so the elements of E are *ordered* pairs).

In this course we will be dealing with **simple** graphs. This means that there are no edges of the form ii and that there are no repeated edges. (For a directed graph it is acceptable to have one edge from a to b and another edge from b to a.)

For an undirected graph, we say that the nodes i and j are **neighbours** if the edge ij exists. (In this case, i and j are also said to be **adjacent**.)

The edge e is said to be **incident** to vertex i if i is one of the end-points of e.

$G' = (V', E')$ is a **subgraph** of $G = (V, E)$ if $V' \subset V$, $E' \subset E$ and every edge in E' connects two vertices in V'. The subgraph **induced** by $V' \subset V$ is obtained as follows: take the set of vertices to be V', and the set of edges to be $E' = \{e \in E : e = ij \text{ for some } i, j \in V'\}$.

A **path** from vertex a to vertex b is a sequence of edges ac_1, c_1c_2, ..., $c_{n-1}c_n$, c_nb. The path is **simple** if it doesn't use a vertex twice. We use the notation $a \rightsquigarrow b$ to denote a path from a to b.

A **cycle** is a path from vertex a to itself. If vertex a is used exactly twice (once at the beginning and once at the end) and the other vertices are used exactly once, the cycle is **simple**.

A graph is **acyclic** if it contains no cycles.

An undirected graph G is **connected** if for any $i, j \in V(G)$ there is a path from i to j. Otherwise it is **disconnected**. A **connected component** is a maximal connected subgraph of G.

If G is connected, and removing $i \in V(G)$ (and all edges incident to i) causes G to become disconnected, then i is called an **articulation point** (or **cutvertex**).

For an undirected graph, the **degree** of a vertex i is the number of vertices j such that $ij \in E$. For a directed graph, the **out-degree** of a vertex i is the number of vertices j such that $ij \in E$, and the **in-degree** of a vertex i is the number of vertices j such that $ji \in E$.

An undirected graph is a **tree** if it is connected and acyclic. Trees have the following properties:

(**i**) There is a unique path between any two nodes.

(**ii**) The number of edges in a tree is exactly one less than the number of nodes.

(**iii**) Adding any edge to a tree produces a cycle. Removing any edge from a tree disconnects it.

(**iv**) Any tree has at least one **leaf**, *i.e.* a node of degree 1.

A **forest** is a disjoint union of trees.

$T = (V', E')$ is a **spanning tree** of an undirected graph $G = (V, E)$ if $V' = V$, $E' \subset E$ and T is a tree.

An undirected graph G is **bipartite** if we can write $V(G) = A \cup B$ where A, B are disjoint, no two vertices in A are connected by an edge, and no two vertices in B are connected by an edge. We write $A \amalg B$ to indicate that the vertices fall into two *disjoint* sets, so that the vertices are indexed according to whether they lie in A or B.

(It is worth keeping the preceding handout to hand for reference when first getting to grips with graphs.)

2 Graph Algorithms

Before we begin discussing the many applications of graphs in problem-solving, it is important to establish a few conventions we will adhere to.

(**i**) We consider only *simple* graphs, with no self-loops, and no repeated edges.

(**ii**) We write $n = |V|$ and $m = |E|$ for the numbers of vertices and edges respectively, and label the nodes $1, \cdots, n$, except where we wish to avoid confusion.

It is hopefully reasonably clear that this course does not go into detail on the implementation of algorithms in code - similarly, we are by and large not too concerned with precisely how we represent information we want to store in computer memory. This is the field of *data structures*, which is also a rich area of computer science, but not the focus of this course.

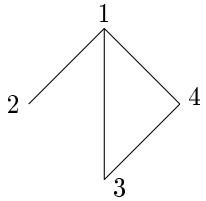
However, the following table gives a summary[14] of three possible methods for storing the structure of a graph with n vertices and m edges.

	Space	Time	
		Check edge exists	Iterate over a node's neighbours
Edge list	$O(m)$	$O(m)$	$O(m)$
Adjacency matrix	$O(n^2)$	$O(1)$	$O(n)$
Adjacency list	$O(m)$	$O(n)$	(optimal)

(**i**) *Edge list*: We simply store all of the edges in the graph in a single list. (We could also introduce an ordering of some sort.)

(**ii**) *Adjacency matrix*: A two dimensional $n \times n$ array where each entry a_{ij} is either a 0 or 1, according to whether the corresponding edge ij exists. For example:

$$(a_{ij}) = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix}$$



(**iii**) *Adjacency list*: A list of n further lists, where the ith list stores all of the neighbours of node i.

Remark. In the case of a directed graph, the entries in the edge list become ordered pairs.

[14] Note that this table does not consider optimizations, which may take advantage of the special structure of some class of graphs; be adaptive; or involve caching frequent queries, for example. It is also far from considering all important graph operations - we might also think about adding or removing nodes and edges, for instance.

There is no 'best' graph data structure, since the answer will depend not only on the information being stored, but also what the programmer wants to do with it. For example, in 'sparse' graphs[15], it may well be preferable to use adjacency lists, whereas otherwise an adjacency matrix is more efficient for checking if edges exist, and so on.

Often, purpose-built data structures can be devised for the special cases of graphs that commonly arise in practice. For example, in a tree, one often distinguishes between 'parent' and 'child' nodes, and might maintain these separately, so that for each node, we have a list of its children and a reference to its parent; perhaps the children only need to store information about their parent node, and not vice versa (as in a so-called *spaghetti stack*); and so on.

Similarly, there is often more information than the existence of edges to be stored - we will see later that the idea of a *weighted graph* (where edges have some values attached to them) frequently crops up (for example, in the travelling salesman problem from the previous lecture).

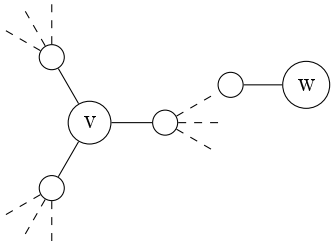
We will leave all of these problems aside for the moment, and concentrate on how efficiently we can write algorithms at a higher level, without paying attention to the underlying structures.

2.1 Search Algorithms

Problem 2.1 (Shortest path)**.** Given two nodes $v, w \in V(G)$, is there a path from v to w? If so, what is the shortest such path? (Assume G is undirected.)

Intuitively, we want to *explore* the graph G, by walking along branches from the v to see if we can find w.

If we try this, then we need to keep track of which nodes we have already seen, to avoid infinite loops.



Idea: Mark unseen nodes as **W**hite, and processed nodes as **B**lack. Also, when we start processing child nodes, we want to make sure we don't start processing them again before we are done, so we mark them **G**rey. Initially, all nodes are **W**hite.

This gives rise to the following generic algorithm for searching through a graph:

Algorithm 8 Generic search

```
Colour all nodes White
Pick a starting node and colour it G
WHILE ( Grey nodes exist )

    Pick a Grey node i
    Mark all of its White neighbours Grey
    Process i (if necessary)
    Mark i as Black
```

---

[15] A *sparse graph* is one where the number of edges is significantly less than the maximum $\frac{1}{2}n(n-1)$. There is no universally accepted definition. One can pick some k with $1 < k < 2$ and then define the sparse graphs to be a class with $m = O(n^k)$.

To prove correctness, we note first that a node can only go **White** → **Grey** → **Black**, and every iteration of the loop at least one node changes colour (namely `i`), so this cannot loop infinitely.

Also, the algorithm only finishes when all nodes are either **White** or **Black**. Any node which is **White** cannot be a neighbour of a node which is connected to the starting node (otherwise it would have been marked **Grey**). Any node which is **Black** must be connected to the starting node. Therefore, there is a path from the starting node v to some other node w if and only if it is marked **Black** when this algorithm is finished, so our algorithm addresses the first part of our problem.

Actually, we get an added bonus once we have this algorithm - we have a way to find *connected components*, the disjoint portions of the graph which are not linked by edges.

Algorithm 9 Connected components

```
Define arrays cc[·] and colour[·] of size n
Let numCC := 0
FOR each node i:
    colour[i] := White
FOR each node i (from 1 to n):
    IF ( colour[i] = White )
        numCC := numCC + 1
        cc[i] := numCC
        colour[i] = Grey
        WHILE (Grey nodes exist)
            Pick a Grey node j
            Colour all its White neighbours Grey
            cc[j] := numCC
            colour[j] := Black
```

When this algorithm terminates, `numCC` contains the total number of connected components, and `cc[i]` gives the label of the connected component of `i`.

Remark. The first `cc[i] := numCC` line is technically unnecessary, as it would be labelled as such by the following search loop because it is **Grey**, but it is included for clarity.

Note that in the above algorithms, the only really ambiguous step is to pick a **Grey** node. There are two common approaches:

- *breadth-first search (BFS)*: Pick the node which was found the *earliest*. This means that all children of the start node are processed first; then the nodes a distance 2 from the start node; and so on.
- *depth-first search (DFS)*: Pick the node which was found the *latest*. This means that nodes on a long, spindly chain leading away from the node are explored first.

Both have many applications, and solve different problems. First, we consider breadth-first search.

Breadth-First Search

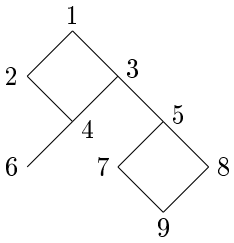
Algorithm 10 Breadth-First Search

```
Let Q be a list for Grey nodes
Pick a starting node and colour it Grey
Add it to Q
WHILE ( Q is not empty )
    Let i := First element of Q
    Mark all of its White neighbours Grey, and add them to the end of Q
    Process i (if necessary)
    Mark i as Black, and remove it from Q
```

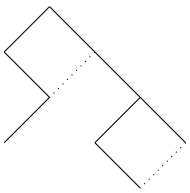
This is using `Q` as a *queue*, which can be done with an $O(1)$ 'push' operation that adds elements to the end, and an $O(1)$ 'pop' operation that removes elements from the front. We say it is a *FIFO*, or *first-in, first-out*, data structure.

Complexity: We process each node exactly once. Each time we do so, we change its colour and scan its neighbours. So using an adjacency matrix, the time complexity is certainly $O(n^2)$. But in fact, each edge is scanned at most twice (once from each endpoint), and a constant amount of work is done per neighbour found - hence using an adjacency list gives complexity $O(n+m)$. Therefore, this is advantageous when the graph is *sparse*, i.e. $m \ll \frac{1}{2}n^2$.

The *traversal order* of a graph search algorithm refers to the sequence in which nodes are visited, and as noted above, this is significantly different for BFS and DFS. The diagram shown assumes that the start node is 1, and the algorithm runs through children from left to right - the numbers indicate the order in which the nodes are visited.



Note that the sequence in which points are visited naturally generates a tree (or more generally, a forest, if there is more than one connected component). This is called the *BFS forest* of the graph, given the start nodes selected. If the graph is connected, this is one way to generate a *spanning tree* for the graph. (We will see later that the problem of finding a *minimum* spanning tree arises in certain applications.)



Exercise 2.2. Solve the second part of Problem 2.1 - i.e. show how to find a shortest path.

The solution is given at the start of the next lecture.

Solution to shortest path problem. BFS from v. Nodes are visited in 'layers' of increasing distance from v. Hence we can assign a distance label d_i to each node i before queuing.

Initially, $d_v = 0$, and $d_i = \infty$ for all other i.

For a grey node x, colour **White** neighbours **Grey** and tag them with $d_y = d_x + 1$.

This terminates with all distances stored in the d_i, with an ∞ indicating that there is no connection. This can be proved by induction on the distance of a node from v (see the following exercise).

To reconstruct the shortest path afterwards, we need only maintain a predecessor array $p[\cdot]$ so that $p[y] = x$ means that y was found from x. Reading backwards from the target node w will give the shortest path.

Complexity: The same as the corresponding BFS, $O(n+m)$.

Exercise 2.3. Prove that d_i does in fact give the shortest distance to the node i.

Remark. We will see a more complicated version of this problem next lecture, when we consider the problem where different edges have different costs.

Depth-First Search

Algorithm 11 Depth-First Search

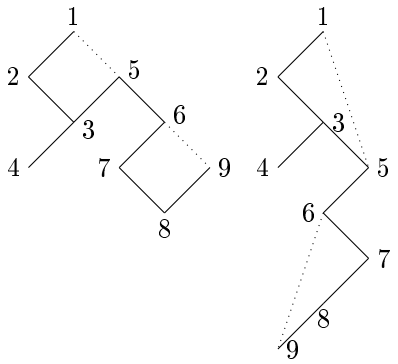
```
Let S be a list for Grey nodes
Pick a starting node and colour it Grey
Add it to S
WHILE ( S is not empty )

    Let i := Last element of S
    Mark i as Black, and remove it from S
    Mark all of i's White neighbours Grey, and add them to the end of S
    Process i (if necessary)
```

This time we are using `S` as a *stack*, which can be done with an $O(1)$ 'push' operation that adds elements to the end, and an $O(1)$ 'pop' operation which removes elements, also from the end. We say a stack is a *LIFO*, or *last-in, first-out*, data structure.

Complexity: This is the same as for BFS, with complexity $O(n+m)$.

Again, looking at the traversal order for the same graph as the BFS, we see we naturally generate a *DFS forest*, or in this case tree. In fact, this structure has some special properties which can be useful.



Applications of DFS

Definition 2.4. For a graph G with DFS forest F,

(i) a *tree edge* is an edge of G also in F; and

(ii) a *back edge* is an edge from a node in G back to one of its proper ancestors in F.

The back edges are shown in the above figure as the dotted lines.

A 'cross edge' is an edge not in F which links two branches of F. However:

Lemma 2.5. *Let F be a DFS forest of G. Then every edge in G is either a tree edge or a back edge with respect to F.*

Proof. Let $e \in E(G)$ and write $e = uv$. Without loss of generality, assume u is coloured **Grey** before v. Then consider the time when u is coloured **Grey** but v is still **White**.

Since u and v are connected, the part of the DFS starting from u will visit v before returning to u.

Hence there is a path $u \rightsquigarrow v$ consisting of tree edges, so u is an ancestor of v. If u is a parent of v, e is a tree edge. Otherwise, it is a back edge. □

It is not immediately obvious what application this could have. However, if we have a problem where we want to think about string of nodes which have few connections with each other, it is clear that the branches in a DFS tree have some sort of 'separability' properties.

Exercise 2.6. Design an algorithm to test for cycles in an undirected graph G. If G has cycles your algorithm should output any one of them; otherwise it should indicate that G has no cycles. (From Example Sheet 2, Question 2.)

Problem 2.7. Given a network of computers, find all computers which, if removed, would disconnect the network (assuming the network is currently connected).

Formally, this is asking us to find the *articulation points* of a connected graph G. As usual, we have a simple brute-force solution:

Solution 1. Remove each node one at a time from G, and check if the resulting graph is connected (i.e. if it has exactly one connected component).

Complexity: $O(n(n+m))$. (Recall we can use BFS or DFS to identify connected components.)

However, using our DFS tree, we can actually do considerably better.

Solution 2. Let T be a DFS tree of G - note that one must exist, since G is assumed to be a connected graph. We root T at the node chosen as the start point for the corresponding search.

The reason for considering T is that it has no cross edges - so no two branches of the tree are connected to each other at any point other than where they diverge. So a branch is totally isolated except for its connection to the branch point, and any back edges from the branch to nodes higher up. Therefore, a node v is an articulation point if and only if

> v is the root of T, and has at least two children

or

> v is not the root of T, and some descendant of v is not connected to any ancestor of v in T (that is, via a back edge).

Let depth(v) be v's depth in T (i.e. its distance from the root), and let $H[v]$ be the highest (least deep) level of T which can be reached from v by going up *at most one back edge* from somewhere in v's subtree. Then consider some node v other than the root, and write u for the parent of v. There are three possible cases:

- v and its subtree are isolated - that is, connected only to the rest of the tree via u - and so $H[v] = \text{depth}(v)$;
- v is connected via a back edge vw to a node higher than any of its descendants can reach via a single back edge, so $H[v] = \text{depth}(w)$; or
- one of more of the descendants are connected, via back edges, to points higher than can be reached directly from v - and then $H[v] = H[w]$, where w is the child of v with a subtree connected to this back edge.

Hence we have

$$H[v] = \min \begin{cases} \text{depth}(v) & \\ H[w] & \text{for tree edges } vw \text{ where } w \neq u \\ \text{depth}(w) & \text{for back edges } vw \end{cases}$$

Now if v (other than the root) is an articulation point, then removing v would disconnect some child w of v from the rest of the tree. Therefore, there are no back edges from within w's subtree reaching any ancestors of v, and we must have $H[w] \geq \text{depth}(v)$. Conversely, if w is a child of v and $H[w] \geq \text{depth}(v)$, then w's subtree must be disconnected from the rest of the tree when v is removed.

So to check if v is an articulation point, we need only check if $H[w] \geq \text{depth}(v)$ for each child w of v.

We can calculate $H[\cdot]$ recursively (going *up* the tree) if we know what the depth of each node is. But depth(i) is easily found by maintaining a 'current depth' variable during the DFS, beginning at 0, and giving each unprocessed child of i the depth (depth(i) + 1).

Complexity: This runs in $O(n+m)$ time; since G is connected, $n \leq m+1$ so the the algorithm is more simply $O(m)$.

Exercise 2.8. Write the code to calculate $H[\cdot]$, including the calculation of $\text{depth}(\cdot)$. You should check it works for a few small cases.

You should be able to attempt questions 1 to 3 on Example Sheet 2 after this lecture. If you are interested in programming, you can also attempt Problem 2-1 on ACOS.

2.2 Shortest Path Algorithms

The obvious generalization of the shortest path problem from lecture 6 comes from having edges of different 'lengths'. Formally, we need some way of attaching numbers to the edges. We do this by introducing *weights*:

Definition 2.9. A *weighted graph* is a graph $G = (V, E)$ together with a function $w : E \to \mathbb{R}$. We write $w(i, j) = \infty$ if $ij \notin E$.

Similarly, to formalize the statement of the 'shortest path' problem, and to fix some notation, we make the following definition:

Definition 2.10. A path

$$x = x_0, x_1, \cdots, x_m = y$$

is a *shortest path* from x to y if it minimizes

$$\sum_{i=0}^{m-1} w(x_i, x_{i+1})$$

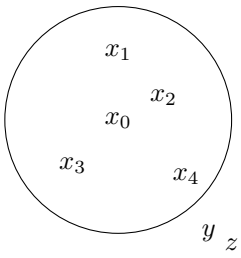
We write $d(x, y)$ for the length of a shortest path from x to y.

We can then state the shortest path problem very concisely:

Problem 2.11 (Generalized shortest path)**.** Given a weighted graph G, and two nodes x and y, find $d(x, y)$. Assume $w(i, j) > 0$ for all i, j.

The last condition will be crucial in determining how we go about solving this problem, as we shall see below[16].

With this positiveness condition, there is an obvious way to generalize our previous solution, of exploring in order of increasing depth:



Idea: Visit nodes in order of increasing distance from x.

In the following discussion, we will write x_k for the kth furthest node from x. We say $x_0 = x$. The question then becomes: how do we find x_k given $x_0, \cdots, x_{k-1}$?

Well, schematically, consider the diagram shown. It seems obvious that if y is the next closest node, then the shortest path from $x \rightsquigarrow y$ must come straight out of the circle, without visiting any other node first. Formally:

[16] Strictly, we could allow $w(i, j) = 0$ as well, but that is just equivalent to the same problem with $i = j$ identified as the same node.

Lemma 2.12. *A shortest path $x \rightsquigarrow x_k$ has the form $x, a_1, \cdots, a_m, x_k$ where $a_i \in \{x_0, \cdots, x_{k-1}\}$.*

Proof. Suppose $a_i \notin \{x_0, x_1, \cdots, x_{k-1}\}$. Then a_i is closer to x than x_k is - a contradiction to the definition of x_k. □

So define a function $f_k(z)$ as

> the length of the shortest path $x \rightsquigarrow z$ that does not leave $\{x_0, \cdots, x_{k-1}, z\}$.

By the above lemma, this has the property that $f_k(x_k) = d(x, x_k)$.

In fact, by conditioning on the last node visited before z (which must be one of $x_0, \cdots, x_{k-1}$), we have

$$f_k(z) = \min_{0 \le l < k} \{d(x, x_l) + w(x_l, z)\}$$

Note that

$$f_k(x_k) = d(x, x_k) \le d(x, z) \le f_k(z)$$

for all $z \in V \setminus \{x_0, \cdots, x_{k-1}\}$ as nodes in this set are at least as far away as x_k.

Solution. Iterate over k, maintaining an array `F[i]` storing $f_k(i)$, and another array `D[i]` storing $d(x, i)$. Then by the above note, at each stage, the node minimizing `F[i]` is x_k. Also, by the previous equation, the array `F[j]` can be updated by noting that the shortest path is either the one previously found, or one going on the shortest path to i (of length `F[i] = D[i]`) and then directly to j.

Complexity: There are n values of k to iterate over, and at each stage there are $O(n)$ nodes to consider for x_k. The only other work which is done is in iterating over the neighbours of each node, and as we saw for the search algorithms, each edge is considered at most twice, giving rise to $O(m \cdot 1)$ work. So the time complexity of the algorithm is $O(n \cdot n + m) = O(n^2 + m)$.

This is called *Dijkstra's algorithm* and in pseudocode is as follows:

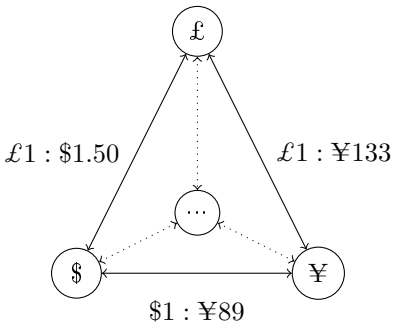
Algorithm 12 Dijkstra's algorithm

```
F[i] := { 0    if i = x
        { ∞    otherwise
FOR k from 0 to n-1:
    Find i in V\{x_0, ··· , x_{k-1}} that minimizes F[i]
    x_i := i
    D[i] := F[i]
    FOR ij an edge:
        F[j] := min( F[j], D[i]+w(i,j) )
```

Remark. If $m \ll n^2$, so the graph is not dense, then we can actually do better, reaching a complexity of $O((n+m)\log n)$. See the handout following this lecture for details, on page 47.

Up to now, we have been solely concerned with the case of positive (or at least non-negative) edge weights. The following problem illustrates one situation where the more general case, involving negative weights, could arise.

Problem 2.13 (Arbitrage)**.** Given a set of currencies, and a set of exchange rates, find the greatest amount of currency B that may be obtained for a single unit of currency A.



This problem is different to anything we have seen before because it is inherently *multiplicative* - so the overall rate $R = r_1 r_2 \cdots r_k$ where we make exchanges with rates r_i. But we can easily transform it to an additive problem by taking the logarithm of all of the rates - then

$$\log R = \log r_1 + \log r_2 + \cdots + \log r_k$$

Since log is strictly increasing, we still want to find the maximum sum of logarithms.

In fact, under the following transformation, the problem becomes a lot more familiar:

(**i**) let the nodes be the set of currencies, $V = \{\text{currencies}\}$.

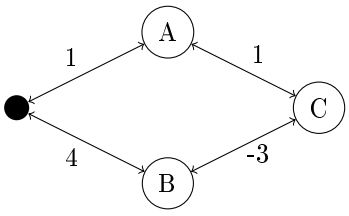
(**ii**) let the edges represent the exchanges, $E = \{\text{exchanges}\}$.

(**iii**) let the edge weights represent exchange rates by

$$w(i,j) = -\log(\text{exchange rate } i \to j)$$

where we negate the logarithm so that better value exchange rates correspond to *smaller* (i.e. closer to $-\infty$) weights. Note that the edge weights are actually *anti-symmetric* here; $w(i,j) = -w(j,i)$.

Here, we want to find a path with the smallest sum of weights, as $R = \exp\left(\sum \log r_{ij}\right) = \exp\left(-\sum -\log r_{ij}\right)$. This is precisely the shortest path problem, but with negative weights allowed.



It is important to see exactly why Dijkstra doesn't work in this situation. It is clear what goes wrong with the arguments above - Lemma 2.12 cannot hold in general, because of situations like that pictured - the second iteration (for $k = 1$) concludes that the closest point to the root (black) is A with distance 1, and calculates the current shortest path to C, i.e. `F[C]`, as being 2. On the next iteration, it then confirms this as the shortest distance to C, which is incorrect, as going to C via B instead has length just 1.

The first thing we need to consider is whether or not there actually *is* a shortest path. Clearly, there wouldn't be if there was some cycle of conversions we could repeatedly carry out, decreasing the total weight each time. In fact, the absence of such a cycle is also precisely the necessary condition:

Lemma 2.14. *Shortest paths exist $\iff$ there is no negative weight cycle. (Assume the graph is strongly connected - i.e. that it is possible to get from any node to any other node in both directions.)*

Proof. Suppose there is a negative weight cycle $z \rightsquigarrow z$. Then paths

$$x \to \underbrace{z \rightsquigarrow z \rightsquigarrow \cdots \rightsquigarrow z}_{n} \to y$$

have arbitrarily small (negative) length as $n \to \infty$.

Suppose there is no negative weight cycle. Then it is sufficient to consider *acyclic* paths, as adding a cycle will not result in a strictly shorter path. But acyclic paths can have at most $n-1$ edges between n nodes - so there are finitely many of them, and hence a minimum exists. □

So how can we find a shortest path, assuming one exists? Our previous attempt, Dijkstra, fails because we didn't allow for the possibility of shorter paths being found later on, via nodes that are themselves apparently further away. This is because the assumption that new paths have to originate from some specific set of nodes, the $\{x_j\}$, is now invalid. But there is nothing wrong with our basic idea of trying to *extend* old paths to make more efficient routes - we simply need to allow for finding improved routes to *all* nodes, not just those outside the magic set $\{x_j\}$.

The only issue, therefore, is whether or not our algorithm will halt with a correct final state. But we know from the above that a shortest path has at most $n-1$ edges, so if we gradually increase the number of edges we are considering, we can be sure that route lengths will never improve after this many steps.

Let us write $d(x, y, k)$ to be the length of the shortest path $x \to y$ with at most k edges.

Solution. Run a DP on $d(x, y, k)$.

Recurrence: A path with at most k edges is only an improvement on a path of at most $k-1$ edges if there is a *shorter* path reaching a neighbouring node in at most $k-1$ edges that goes on to reach this node via the kth edge. Hence

$$d(x, y, k) = \min \begin{cases} d(x, y, k-1) \\ \min_{z \in V} \{d(x, z, k-1) + w(z, y)\} \end{cases}$$

Base case: Obviously, as with Dijkstra,

$$d(x, y, 0) = \begin{cases} 0 & \text{if } x = y \\ \infty & \text{otherwise} \end{cases}$$

This is easily implemented in a similar way to our previous DP algorithms. Let `D[y][k]` store $d(x, y, k)$.

Algorithm 13 Bellman-Ford algorithm

```
D[y][0] = { 0    if x = y
          { ∞    otherwise
FOR k from 1 to n-1:
    FOR z ∈ V:
        D[z][k] := D[z][k-1]
    FOR ij an edge:
        D[j][k] := min( D[j][k], D[i][k-1] + w(i,j) )
```

Complexity: Implemented this way we have an $O(n(n+m)) = O(nm)$ algorithm (assuming that $n < m$, as holds for a reasonably well-connected graph).

Remark. We could use a pair of one-dimensional tables by discarding the rows in `D` corresponding to optimal paths with at most $k-2$ nodes, since this information is always carried over into the next row if it remains relevant.

Note that as with Dijkstra, we can also add a predecessor array to actually retrieve the shortest paths from x to all other nodes after running Bellman-Ford. An interesting extension to the problem we have discussed is the *all pairs shortest path problem*, in which we are asked to find the lengths of the shortest paths between *every pair* of nodes. The Floyd-Warshall algorithm solves this problem, but we will not discuss this here.

You should be able to attempt question 4 on Example Sheet 2 after this lecture. If you are interested in programming, you can also attempt Problem 2-2 on ACOS.

Handout: Dijkstra with Heaps

In lectures, it was stated that the runtime of Dijkstra's algorithm could be 'improved' from $O(n^2)$ to $O((n+m)\log n)$ (noting that this is actually worse if the graph is dense). Here we show how to do this.

Priority Queues

The new data structure we will need is the *min-priority queue*. Informally, this is a gadget that we keep throwing new objects into, which we can interrogate at any point to ask it to spit out the smallest one. By 'smallest' we mean that every object comes with a *key* k, and these keys can be compared to determine the objects' size ordering. More formally, we expect the gadget to support the following operations:

- `EmptyQ()` - returns an empty Queue
- `Insert(Q, x, k)` - inserts object x with key k into Q
- `ExtractMin(Q)` - returns the minimal object in Q and removes it from Q
- `DecreaseKey(Q, x, k)` - adjusts object x's key to k, which is smaller than its previous value

It's not too hard to think of a data structure that does the job; for instance, a list of (x,k) pairs sorted by k. Then the operations `Insert` and `DecreaseKey` run in $O(n)$ time, where n is the number of objects in the queue. This may be good enough for some things, but for our purposes we want all these operations to run in $O(\log n)$ time. So we must find another data structure.

Binary Heaps

Definition

A *binary tree* is a rooted tree where each node has at most two children. An *almost complete* binary tree is one where depths of leaves differ by at most one, and the deepest leaves are all on the left. (Hence there is a uniquely shaped binary tree of each size.) A *min binary heap* is an almost complete binary tree with keys and objects stored at the nodes, such that a node's key is greater than or equal to its parent's. The name 'heap' is appropriate: we have described a sort of spreading mound of objects with the smallest ones at the top. Note also that the maximum depth of the heap is $\sim \log_2 n$, so provided our operations just move 'up and down' the tree, they should run in $O(\log n)$ time as required.

Representing Binary Heaps

In practice, we represent a binary heap by numbering the nodes $1, \cdots, n$ in a top-down, left-to-right fashion, and thereby storing the nodes in an array. Then the following operations allow us to navigate the tree:

$$\begin{aligned} \texttt{Parent}(m) &= \left\lfloor \frac{m}{2} \right\rfloor \\ \texttt{LeftChild}(m) &= 2m \\ \texttt{RightChild}(m) &= 2m+1 \end{aligned}$$

If the left/right child doesn't exist, you get a value greater than n. We assume that the array is always big enough, and that we have the variable n stored somewhere sensible.

Implementing Priority Queues

It turns out we can indeed use heaps to implement an efficient priority queue. `EmptyQ` is trivial:

```
EmptyQ():
    n := 0
    RETURN a new array
```

`DecreaseKey` turns out not to be too bad:

```
DecreaseKey(Q, x, k):
    Set x's key to k
    WHILE Parent(x) != 0 AND k < Parent(x)'s key:
        SWAP x and Parent(x)
```

where we write `SWAP` to mean swap keys and objects. You can check that this works; i.e. given that the heap conditions are satisfied beforehand, they will be afterwards as well.

We can now do `Insert` with a cunning trick. We can just put the new node at the end (i.e. bottom right) of the tree, and run `DecreaseKey` until it's in the right place.

```
Insert(Q, x, k):
    n := n + 1
    Q[n] := (x, k)
    DecreaseKey(Q, x, k)
```

We do something similar to implement `ExtractMin`: first we write an `IncreaseKey` procedure, which is slightly more fiddly than `DecreaseKey`.

```
IncreaseKey(Q, x, k):
    SET x's key to k
    WHILE x is not a leaf:
        c := a child of x with minimal key
        SWAP x and c
```

Then, we can do the same trick in reverse for `ExtractMin`:

```
ExtractMin(Q):
    (x, k) := Q[1]
    SWAP Q[1] and Q[n]
    n := n-1
    IncreaseKey(Q, 1, Q[1]'s key)
    RETURN (x, k)
```

Again, you can check these implementations work and run in worst-case $O(\log n)$ time.

Dijkstra using Heaps

How can we use this to speed up Dijkstra? The slow bit is when we iterate over all nodes to find a node that minimizes the magic function f_k. So if we keep a queue of all nodes keyed by their value of f_k, we can hopefully do this in $O(\log n)$ time. Of course, we'll have to keep updating the queue as f_k changes, which will take some time as well.

Here goes:

Algorithm 14 Fast Dijkstra

```
SQ := EmptyQ()
Insert(Q, 0, 0)
FOR i from 1 to n-1:
    Insert(Q, i, ∞)
FOR k from 1 to n:
    (i, d) := ExtractMin(Q)
    D[i] := d
    FOR all edges ij:
        IF d + w(i,j) < j's key
            DecreaseKey(Q, j, d + w(i,j))
```

Complexity: We do n `ExtractMin`'s and m `DecreaseKey`'s, so the algorithm runs in time $O((n+m)\log n)$ as claimed.

Remark. As an aside, we can actually speed this up further using *Fibonacci heaps*, which can do the `DecreaseKey` operation in constant time on average. That reduces the runtime to $O(n \log n + m)$.

Exercise 2.15. Prove that $O(n \log n + m)$ is optimal for a Dijkstra implementation.

Hint: You may use the fact that a sort cannot be performed faster that $O(n \log n)$.

2.3 Minimum Spanning Trees (MSTs)

Imagine we have a city full of computing centres that need to be linked together, or a country with a electricity substation that has to be connected to all its cities - where should we lay the cable to minimize the amount of cable we must use?

This type of problem is a restricted (Euclidean) version of a much more general problem, which we can formalize as follows:

Problem 2.16. Given a connected weighted graph $G = (V, E)$ with positive edge weights, find a subset $T \subseteq E$ such that (V, T) is connected and the sum of the weights of the edges of T is minimized.

Remark. T must be a (spanning) tree[17]: if it was not, then it would contain a cycle, and we could remove any edge to obtain another spanning set of edges with a strictly smaller total weight - a contradiction.

We can easily dismiss the brute force solution of trying all sets of $n-1$ edges as being woefully inefficient for substantial n and m, since there are $\binom{m}{n-1}$ of these.

So what other approaches could we adopt? A natural idea would be to gradually build up a tree using the least expensive edges available:

Solution? Start with $T = \emptyset$. Iterate through edges in increasing order of cost, adding an edge to T if and only if it doesn't create a cycle.

Complexity: Sorting edges is $O(m \log m)$, and checking if an edge creates a cycle can be done in $O(n+m)$ - see Exercise 2.6. Hence overall, this is

$$O(m(n+m) + m \log m) = O(m^2)$$

The problem is that we have no idea whether or not this actually works.

One way of rephrasing the approach our putative solution takes is to say that we add an edge to T (proceeding in order of increasing weight) if and only if *it joins two different subtrees*. (Since we start with $T = \emptyset$, we initially take every node to be in its own tree.) We need to show that if we do this, then T remains a subset of some MST.

Definition 2.17. A *cut* is a partition $V = S \coprod (V \backslash S)$ into two disjoint sets with neither being the empty set ($S \neq \emptyset, V$). (The notation $A \coprod B$ just means the *disjoint union* of A and B, where we label elements according to which set they belong to.)

We say an edge e *straddles* the cut if e has one endpoint in S and one in $V \backslash S$.

We therefore would like to prove the following lemma:

[17] Formally, since T is just a collection of edges, what we are really saying is that (V, T) forms a tree. This abuse of notation is fairly harmless, since it is clear what the set of nodes is.

Lemma 2.18 (Cut Lemma)**.** *Let T be a set of edges in some MST. Let $S \coprod (V \backslash S)$ be a cut, and suppose T contains no edge straddling the cut.*

Let e be the cheapest edge straddling the cut - then $T \cup \{e\}$ is also part of some MST.

To prove this, assuming $T \subseteq M$ for M some MST, we need to show that e is one valid way of joining the two sides of the cut - so that there is some M' which straddles the cut using e.

Proof. Let M be an MST containing T, and write $e = uv$ for $u \in S$ and $v \in V \backslash S$.

Now in M, there is a unique $u \rightsquigarrow v$ path - call this P. Since u and v are separated by the cut, there is an edge $f \in P$ that straddles the cut.

Define $M' = (M \backslash \{f\}) \cup \{e\}$. Removing f splits M into two subtrees, one containing u and one containing v, and therefore adding e makes M' another spanning tree.

But $\text{weight}(e) \leq \text{weight}(f)$ by definition of e, and hence

$$\text{weight}(M') \leq \text{weight}(M)$$

so as M is an MST, M' must be too. Therefore, $T \cup \{e\} \subseteq M'$, a minimum spanning tree. □

So we are done! And this gives us immediately:

Corollary 2.19. *The above solution is correct.*

We call it *Kruskal's algorithm.*

Remark. Using a *disjoint set* data structure, we can bring the complexity of Kruskal's algorithm down to $O(m \log n)$ (or better if the edges are already sorted by weight, or can be sorted in better than $O(m \log n) = O(m \log m)$ time). See Kleinberg & Tardos, Chapter 4, for details[18].

From the cut lemma, we can actually deduce another similar solution:

Alternative Solution (Jarník-Prim). Start with $S = \{v\}$ for some $v \in V$. Keep growing S using the cheapest edge between S and $V \backslash S$ until $S = V$. The edges used constitute the MST T.

This solution, often called the *Jarník-Prim* or just *Prim's* algorithm, can be easily proved correct using Lemma 2.18.

Complexity: A simple analysis concludes this is $O(nm)$. But we shouldn't have to examine all edges at each stage.

An improvement can be made by maintaining 2 arrays, `c[i]` containing the node in S closest to i, and `l[i]` containing the cost of reaching i from S, namely $\text{weight}(\texttt{i}, \texttt{c}[\texttt{i}])$. Initially, `c[i]` is v for all i and `l[i]` is $\text{weight}(i, c[i])$.

At each step, find $j \notin S$ with the minimum `l[j]` and then update the arrays as follows:

[18] J. Kleinberg, E. Tardos *Algorithm Design.* Pearson Education 2005.

```
c[j] := j
l[j] := 0
FOR each node k adjacent to j:

    IF ( weight(k,j) < l[k] )
        l[k] := weight(k,j)
        c[k] := j
```

Complexity: We do $O(n)$ iterations, and each time we find the nearest missing node in $O(n)$ steps and update its neighbours in $O(\deg j)$ where $\deg j$ is the degree of j. So the total complexity is

$$O\left(n \cdot n + \sum_{j \in V} \deg j\right) = O\left(n^2 + m\right)$$

Remark. Again, we can 'improve' this to $O(m \log n)$ time by using a priority queue (see the handout on optimizations for Dijkstra).

Before moving on from MSTs, we address a few related problems.

Problem 2.20. Solve the minimum spanning graph problem (Problem 2.16) if zero and negative edge weights are allowed.

Clearly, T need no longer be a tree - there is no reason not to include *all* zero- and negative-weight edges in it! In fact, we can solve this problem easily as follows:

Solution. Put all edges with negative or zero weight into the set T, and then modify Kruskal to run on the resulting graph.

Exercise 2.21. Write down pseudocode showing exactly how we would implement this variant of Kruskal.

Both algorithms (Kruskal and Jarník-Prim) are examples of *greedy algorithms* (see the next lecture) - this basically means they choose the best available option at each stage. This makes the problem perhaps deceptively easy to solve.

But many variants of the MST problem are actually NP-hard[19]; for example,

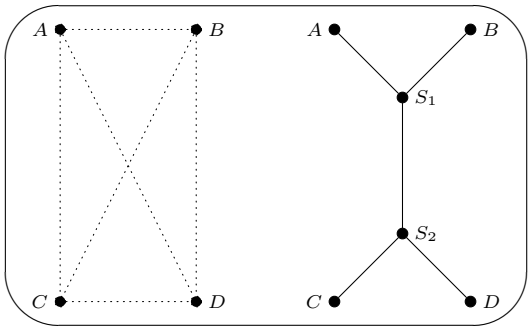
(**i**) finding the minimum subtree spanning at least k nodes.

(**ii**) finding the spanning tree with the most leaves.

(**iii**) finding the minimum *Steiner tree* containing the initial nodes in V. When we build a Steiner tree, we are allowed to introduce new nodes at intermediate points to reduce the total length of the edges needed to network all the original ones together - we assume we have some structure like the Euclidean plane that allows us to calculate the lengths of as-yet non-existent edges. For example, if we want to distribute power to n nodes, and want to minimize the cable which we lay, we would try to find the Steiner tree for these nodes.

[19] This puts them in one particular class of 'hard' problems that no-one knows how to solve efficiently.

Example 2.22. Solution of a four-point Steiner tree problem, with initial points A, B, C, D, leading to the creation of two new points, S_1 and S_2:



Remark. Finding the minimum spanning tree of a connected graph in the general case of a *metric* Steiner tree (i.e. where the edge weights obey the triangle inequality) actually gives a 2-approximation - that is, the total length of the edges used in an MST is never any more than twice that of the optimal Steiner tree.

For the Euclidean Steiner tree problem, the MST is at most a factor $2/\sqrt{3} \approxeq 1.15$ worse than the optimal solution.

You should be able to attempt questions 5 and 6 on Example Sheet 2 after this lecture.

3 Greedy Algorithms

As we saw last time, sometimes it is possible to make choices which are in some sense 'optimal' at each stage, and end up with a solution which is also optimal overall. However, it is obviously quite possible that this does not work. (Recall, for example, the suggested solutions to the 'Books' problem from the first lecture.)

In this lecture, we will briefly discuss a few problems with greedy solutions, hopefully picking up a few useful techniques for problem-solving.

Problem 3.1. Given N events with start and finish times (s_i, t_i), with $s_i < t_i$, what is the maximum number of events I can attend, assuming I can attend at most one at any given time?

Solution 1. Try all 2^N subsets. This has a painful time complexity, and becomes impractical for large N.

We could view this as a degenerate or trivial greedy algorithm - it has one stage, and makes the optimal choice by evaluating the entire space!

So let us consider making more immediate decisions - what criterion could be used to pick out a particular element? One natural approach is to assume that short events, being less likely to have clashes, should be preferred:

Solution 2? Pick the shortest event, and remove all clashes. Repeat.

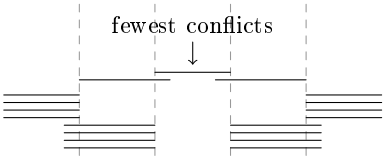
However, this is clearly wrong, as the diagram shows.

The reason is that there is no particular reason why short events should have only a few conflicts. So instead, why not directly choose the events with the fewest clashes?

Solution 3? Pick the event with the fewest conflicts, and remove all clashing events. Repeat.

This is also wrong, though it is perhaps not so immediately obvious what a counterexample would be.

We can, however, construct one by considering elaborate alternating sequences of events $ABABA$ such that the optimal solution picks A, A, A and the algorithm picks B, B.



Indeed, in the pictured situation, it is possible to select 4 disjoint events from the given diagram, but the algorithm selects just 3.

Remark. Note how intricate the counterexample can be; we should be wary of only using small cases to investigate greedy algorithms.

Another attempt to solve the problem would be to pick earlier events first, and always attend the next available event.

Solution 4? Pick the event that starts first, and remove all clashes. Repeat.

This is also too optimistic. ————————

Let us step back a minute from coming up with greedy solutions more or less at random, and adopt a more traditional approach.

Specifically, let's consider a dynamic-programming-style approach. We would like to know the maximum number of *complete* events that can be attended up to some point. Let's write $M(t)$ to denote this, where t is some point in time. Clearly, M can only change its value at the end of an event. Hence we can do the following:

Solution 5. Run a dynamic programming algorithm on $M(t_i)$.

Recurrence: At the time that event i ends, either we did attend event i or we didn't. Hence

$$M(t_i) = \max\left(1 + M(s_i), M(\text{largest } f_j \leq f_i)\right)$$

This involves $M(s_i)$, which we have not calculated in our DP. But because M only changes it value at the f_j,

$$M(s_i) = M(\text{largest } f_j \leq s_i)$$

so if we choose our labels so that $f_1 \leq f_2 \leq \cdots \leq f_N$, then

$$M(f_i) = \max\left(1 + M(\text{largest } f_j \leq s_i), M(f_{i-1})\right)$$

Complexity: We have N entries to calculate, and for each one we need to find the the largest f_j below some other point which can be done easily in $O(N)$, so this takes $O\left(N^2\right)$ time. (Note that sorting f_i takes $O(N \log N)$ which is dominated by N^2.)

Improvement: We can actually binary search through the f_j to find the latest one no larger than s_i, reducing this to $O(N \log N)$. Alternatively, we could also sort the s_i and then pre-calculate the largest $f_j \leq s_i$ for all i - we can store this in `LFJ[i]`, the 'latest f_j' preceding i. We can do this by iterating simultaneously through both lists, just as we did when discussing the `Merge` procedure for merge sorting - we keep increasing i one step at a time, checking whether we pass another f_j or not. If we do, then we step through the f_j until we find the latest one $f_{j^\star}$ falling before s_i; then `LFJ[i] := j`*; otherwise, `LFJ[i] := LFJ[i-1]`.

Now what is interesting about finding this solution is that it actually suggests a new approach for a greedy algorithm - what we're interested in throughout the above iterative approach is choosing events with the most convenient *finishing* times.

Solution 6. Pick the event that finishes first, and remove all clashes. Repeat.

Proof. Suppose some optimal set S of events doesn't contain $E_1 = [s_1, f_1]$ (where again we write $f_1 = \min_i f_i$).

Then if S doesn't contain an event that clashes with E_1, S would be obviously be improved by adding E_1 - contradiction.

If it does contain a clash, we can replace the clashing event by E_1, and the resulting set is just as good, so we have another optimal set.

Hence there is an optimal set S' containing the event which finishes earliest. Therefore, we can include this event - then we are forced to remove clashes, and thereafter, we can repeat our argument again, since E_1 does not overlap with any remaining events. (Note that as S' is an optimal solution containing the chosen event, it does not contain any of the clashes which were removed, so this removal is legitimate.) □

Implementation: To actually implement this intelligently, we should sort the s_i and then the f_i separately. Then we can repeatedly find the smallest f_i - in $O(1)$ time - and select the event i, and remove elements with $s_j < f_i$ (using the sorted list).

Complexity: $O(N \log N + N) = O(N \log N)$.

Example 3.2. Consider the events

$$A : [0, 3] \quad B : [1, 2] \quad C : [1, 3] \quad D : [2, 4] \quad E : [2, 5] \quad F : [3, 4] \quad G : [4, 7] \quad H : [5, 6] \quad I : [6, 9]$$

Sorting these by start time gives $(A, B, C, D, E, F, G, H, I)$ whilst sorting them by finish time gives $(B, A, C, D, F, E, H, G, I)$.

Our last algorithm iterates through this last list:

- Select B. Finish time is 2. Events with $s_j < 2$ are A, B, C.
- Remove A, remove C. Select D. Finish time is 4. Events with $s_j < 4$ are D, E, F.
- Remove F, remove E. Select H. Finish time is 6. Events with $s_j < 6$ are G, H.
- Remove G. Select I. Done.

Hence our selected set is B, D, H, I.

A greedy algorithm like this is correct when it satisfies what is called the *greedy choice property*: successively picking locally optimal solutions leads to globally optimal ones. Like this final example, they tend to be fairly easy to describe and code, and also tend to be efficient (note that we automatically achieved the same complexity with the sixth attempt as with the iterative approach of the fifth). However, the obvious problem is that they are often wrong.

Therefore, it is particularly important to have good techniques to prove the correctness of greedy algorithms. We have seen 2 ways (though they are essentially equivalent) so far.

- An *exchange argument*, as in solution 6. Take any solution S not including the greedy choice g and show that we can modify it to obtain a solution $S' \ni g$ which is at least as good.
- As in lecture 9, when deducing that Kruskal's algorithm worked - directly show that making the greedy choices one at a time never leads to a loss in optimality of a solution to a restricted problem which eventually coincides with the original problem.

We have derived several greedy algorithms so far in this course, most notably

- Dijkstra (lecture 8, on page 43);
- Kruskal's MST algorithm (lecture 9, on page 50);
- (Jarník-)Prim's MST algorithm (lecture 9, on page 51).

The employee party problem (lecture 5) also has a greedy solution, as you may like to verify:

Exercise 3.3. Show that the employee party problem (Problem 1.17) can be solved by picking all leaves, removing the nodes they were attached to, and repeating this process on the remaining trees.

Hint: An exchange argument can be used.

Recall also that in the same 'Books' problem which we had several incorrect greedy solutions for, we came up with a correct algorithm in lecture 2 that solved the subproblem of optimizing $\sum_{i \in S} v_i - \lambda w_i$ over S by a greedy algorithm. This technique (of combining a greedy algorithm with a binary search) is an often useful approach that is frequently *not* discussed in textbooks.

Problem 3.4. Solve the following problem as quickly as you can:

> Given a list of N positive integers, partition it into $K \leq N$ contiguous subsets (i.e. sublists) such that the maximum sum of elements in any sublist λ is minimized.

(For example, the best we can do with $\overline{3,3}, \overline{4,2,1}, \overline{2,5}$ where $K = 3$ is $\lambda = 7$, as shown.)

You should be able to attempt questions 7 and 8 on Example Sheet 2 after this lecture.
If you are interested in programming, you can also attempt Problem 2-3 on ACOS.

4 Matchings and Network Flow

We are now moving on to discuss a very different type of problem to what we have considered so far: the problem of *matchings*.

4.1 The Stable Marriage[20] Problem

Problem 4.1. There are n students $a, b, c, \cdots$ and n supervisors $A, B, C, \cdots$. Each student submits a preference list of supervisors, and each supervisor submits a preference list of students. Match students to supervisors so there is no *instability* - that is, no student a assigned to a supervisor A could be partnered with some other supervisor B resulting in both a and B being happier.

This problem can be thought of as trying to avoid encouraging an 'adulterous' situation, where it is in the interest of two parties to break away from their assigned partners (though in this case we are more concerned about bored Ph.D. students and their supervisors).

This is the kind of problem which seems like it may admit a simple greedy solution, so let us try to find one:

Solution 1? Take each student in turn, and give him his favourite supervisor (among those that are not already assigned).

Unfortunately, this is wrong:

Preferences (preferred items list first):	$A : ba$	$a : AB$	Matching for A:	$\boxed{a}$
	$B : ab$	$b : AB$	B:	$\boxed{b}$

The generated matching is unstable.

Solution 2? Start with a matching M. While it contains an instability, swap the affected partners and repeat. Terminate when the matching is stable.

Whilst clearly if this does terminate, it gives a stable matching, we have no guarantee it will terminate! The progression of the algorithm on the initial matching $a-A$, $b-B$, $c-C$ is illustrated, with the instability highlighted at each stage:

Preferences:	$A : bac$	$a : ACB$	Matching for A:	$\boxed{a}$	$\boxed{b}$	c	$\boxed{c}$	$\boxed{a}$	$\cdots$
	$B : cab$	$b : CAB$	B:	$\boxed{b}$	a	$\boxed{a}$	b	$\boxed{b}$	$\cdots$
	$C : abc$	$c : CAB$	C:	c	$\boxed{c}$	$\boxed{b}$	$\boxed{a}$	c	$\cdots$

So as before, when faced with a problem intractable by an obvious greedy algorithm, why not attempt:

Solution 3? Recursion? But this needs an exponential number of states; it is very difficult to efficiently encode the idea of an *instability* into a recursive algorithm's state structure.

[20] The problem was named in the days before gay marriage was something that would ever cross anyone's minds; a progressive variant is mentioned at the end of this lecture!

So what can we do? Well, there is another type of greedy algorithm we have not yet tried, which is in some ways a combination of the first two solutions:

Solution 4? Every iteration, take all unmatched students, and let them apply to their favourite supervisor who they have not yet seen; then for each supervisor, let them choose his favourite student between those who have just applied to him and his current student (if he has one).

There are two questions which we need to answer for an algorithm such as this:

(**i**) *Will this terminate?* Clearly, in each iteration, at least one student approaches a 'new' supervisor; since there are only n^2 possible proposals, this algorithm has at most n^2 iterations in total, and hence must terminate.

(**ii**) *Is the matching produced stable?* Well, suppose (without loss of generality) that $a-A$ and $b-B$ are two pairs in the matching, and that $a-B$ is an instability. Then a prefers B to A, whilst B prefers a to b.

Then since a prefers B to A, he must have approached B earlier and got rejected at some point. But this is impossible, since once a supervisor has selected a student, they will only ever change that matching if the supervisor is petitioned by a student they prefer (and they remain at all times matched with some student) - so B would never have selected b, but instead would either have retained a or chosen a student preferable to both at some point.

Remark. You may be worried about the possibility that a student has no supervisor but has been rejected by everyone already. But that means that the student has applied to all n supervisors. However, as we've just seen, a supervisor never gives up a student unless they accept a new one - so if a student ever applies to a supervisor, then from then on that supervisor will always have a student. So the n supervisors must each have exactly one of the other $n-1$ students matched to them. This is an obvious contradiction.

Hence we do indeed have a correct solution, which seems reasonably fast:

Algorithm 15 Stable Marriage algorithm (sometimes called the *Gale-Shapley* algorithm)

```
WHILE some student has no supervisor:
    FOR each unmatched student i:
        Let i approach their favourite unseen supervisor
    FOR each supervisor J who received an application:
        Let k be the most preferable student who applied to J
        IF (J currently has no student)
            J accepts k
        ELSE IF (J prefers k to their current student)
            The current student is discarded
            J accepts k
        ELSE
            J keeps their current student
RETURN (student,supervisor) pairs
```

Complexity: As it stands, we have at most n^2 iterations, with each iteration being $O(n)$. So this takes $O(n^3)$ time.

But we can actually improve this by considering the proposals independently:

Algorithm 16 Better Stable Marriage algorithm

```
WHILE some student i has no supervisor:

    Let i approach his favourite unseen supervisor A
    IF ( A prefers i to his current student or has no student )
        Match A and i (possibly throwing out A's current student)

RETURN (student,supervisor) pairs
```

Complexity: There are still at most n^2 iterations, by the same logic as before applied more directly, but now the looped code is $O(1)$ because we assume the preference list of each student is sorted, or at least that we can look up their jth favourite supervisor in constant time.

What is interesting about a solution to this problem is considering what the students and supervisors make of it - how fairly does it treat them all? We answer this question in the following two theorems.

Theorem 4.2. *Under this algorithm, each student receives the best possible supervisor (in the sense that there is no stable matching in which they have a more desirable supervisor).*

This may seem surprising, as the algorithm appears to make students subject to the whims of the supervisors. However, the power students wield in 'voting with their feet' (so to speak) means they actually do very well out of this process.

Proving this theorem is not actually very difficult:

Proof. Call a supervisor 'possible' for a student if there is some stable matching in which they are paired. We are claiming no student is ever rejected by a possible supervisor.

Clearly this holds at the start of the algorithm - let us proceed inductively, assuming that the algorithm so far respects the induction hypothesis that 'no student has been rejected by a possible supervisor'.

Suppose that a has just been rejected by A in favour of b. Then we know that b prefers A to all other supervisors possible for b (by the induction hypothesis, any supervisors that rejected b were impossible). Also, A prefers b to a.

But then $b - A$ is obviously an instability in any possible matching with the pairing $a - A$! Therefore, such a matching M is not stable.

Hence, if a student is rejected by a supervisor, that supervisor is not possible for a supervisor. So because students apply to supervisors in order of preference, they each get the best possible supervisor. □

Interestingly, because students give a definite ranking to all the supervisors, this clearly specifies exactly one matching.

Corollary. *The above algorithm always produces the same matching.*

So, given that students do well, it is tempting to conclude that supervisors do not. Indeed:

Theorem 4.3. *In the above algorithm, each supervisor ends up with the worst possible student (i.e. no stable matching could contain a worse student).*

Proof. Imagine that A prefers a to b, and that b is possible for A. Now suppose that the stable matching the above algorithm gives contains the pair $a - A$. Then A is the best possible supervisor for a.

Also, since b is possible for A, there is a stable matching M containing $b - A$ (and $a - C$, for some other C).

But then we can see that $a - A$ is an instability (because A prefers a to b, and A is the best possible supervisor for a), and hence M is not stable, which is a contradiction. □

We have shown that our algorithm is *student-optimal* and *supervisor-pessimal*.

Generalizations

As with the MST problem, it is easy to come up with apparently closely related problems to which the answers are surprisingly elusive. There are four examples below.

(**i**) Is there a stable matching algorithm that is 'fair' for both sides? (For example, where all parties have an equal probability to be matched with their optimal partner.)

(**ii**) In the worst case, how many stable matchings are there between n students and n supervisors? (This is an open problem.)

(**iii**) The *stable family problem* (Knuth): organize $3n$ players - n men, n women and n dogs - families of three, with one member from each group, so that there is no *blocking triple* - three players each preferring one another to their assigned family members. (This is NP-complete.)

(**iv**) The *stable roommates problem*: Organize $2n$ people into pairs to share rooms with no instabilities (i.e. a *progressive* stable marriage problem not excluding homosexual couples!). This is not always possible (as can be seen from taking $A : BCD$, $B : CAD$, $C : ABD$ and any preferences for D - whoever D is matched with will have an instability) but it turns out there is an efficient $O\left(n^2\right)$ algorithm[21] for determining whether a solution exists, and if so finding it.

You should be able to attempt question 1 on Example Sheet 3 after this lecture. If you are interested in programming, you can also attempt Problem 3-1 on ACOS

[21] Irving, Robert W. (1985), 'An efficient algorithm for the "stable roommates" problem', *Journal of Algorithms* 6 (4): 577–595

Lecture 12: Maximum Bipartite Matching

There are actually other types of matching problems, most notably those where the number of each group is not the same. In this lecture, we set aside issues of preference or value, and simply consider how many pairs it is possible to make, given two groups with only some partnerships allowed.

4.2 Maximum Matching in Bipartite Graphs

The formalism we use is probably the obvious one: we draw a graph with two columns of nodes, with no edges joining two nodes in the same column - a *bipartite* graph. (Here, edges represent *possible* partnerships.) The problem then becomes:

Problem 4.4. Given a bipartite graph $G = (A \coprod B, E)$, find a maximum matching. (A *matching of size* k is a set of k edges, each joining a node in A to a node in B, such that each node is incident to at most one edge. A *maximum matching* is a matching of maximum size.)

Remark. We can assume G is connected, since each connected component can just be treated in isolation. (You may like to check this.)

We will use the following definition to keep our arguments concise.

Definition 4.5. A node is *used* with respect to a matching M if it is part of some edge in M. Otherwise, it is *unused*. We use the terms similarly for edges.

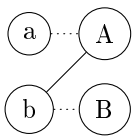
So the question is: how do we maximize the number of used edges?

A natural approach is to adopt the method used by the second (greedy) algorithm we saw for the stable marriage problem: begin with some matching M, and then modify it to improve it. When used in the stable marriage problem, this failed, because it could lead to infinite loops, as we had no well-defined aspect of the matching which we were actually quantitatively improving.

This time, however, we can easily measure our improvement by recording and improving the size of the matching. What's more, since there is obviously some upper limit on the size of the matching, any algorithm that always improves this will terminate - we only need to check that it cannot stop at a sub-optimal matching. The question then becomes:

> What can we do with a matching to improve it?

There is one trivial case, namely where we have a pair of unused nodes linked by an unused edge - in this case, we simply include it, and improve the matching.



But it is not too hard to come up with another situation where we can clearly improve the matching. Imagine we have a set of four nodes like those pictured, where we could match two pairs, but we have matched just one pair, excluding the other two. It would obviously be better to 'flip' the system, removing the offending edge, and inserting the two new ones.

It is also clear how this can be extended to more edges. This all motivates the following definition:

Definition 4.6. An *alternating path* (with respect to a matching M) is a simple path in G whose edges alternate between used and unused. (Recall that a *simple* path is one which does not use the same vertex twice.)

An *augmenting path* (with respect to M) is an alternating path whose first and last nodes are unused with respect to M.

Idea: Keep 'flipping' augmenting paths.

To come up with a way to find an augmenting path, it is useful to think of how we can explore a bipartite graph G. We know two basic approaches to writing graph search algorithms, namely BFS and DFS. We shall work with BFS here.

Solution. We modify BFS as follows:

Pick an unused $a_0 \in A$ and find all its neighbours $b_i \in B$. Each b_i is either already matched, or unused. If it is matched, then find the unique node $a_i \in A$ which is currently matched to $b_i \in B$, then find all the neighbours of a_i, and so on, until we find an unused node in B.

More precisely, let

$$\begin{aligned}
A_1 &= \{a_0\} \\
B_1 &= \{b \in B : a_0 b \text{ is an unused edge}\} \\
A_2 &= \{a \in A \backslash A_1 : ab \text{ is a used edge for some } b \in B_1\} \\
\cdots &\quad \cdots \\
B_r &= \left\{ b \in B \backslash \bigcup_{i=1}^{r-1} B_i : ab \text{ is an unused edge for some } a \in A_r \right\} \\
A_{r+1} &= \left\{ a \in A \backslash \bigcup_{i=1}^{r} A_i : ab \text{ is a used edge for some } b \in B_r \right\}
\end{aligned}$$

Once we find an unused node in some B_r, we have found an augmenting path. We can then flip all of its edges (between used and unused) and repeat.

Complexity: Notice that once a node is used, it remains used - and each augmenting path increases the number of used nodes by 2. Hence there are at most $\left\lfloor \frac{n}{2} \right\rfloor$ augmentations. Finding a path is equivalent to running a BFS, which is $O(n+m) = O(m)$ (assuming the graph is connected) so as we do this for each $a \in A$, we have an $O\left(m \cdot n \cdot \left\lfloor \frac{n}{2} \right\rfloor\right) = O\left(n^2 m\right)$ algorithm.

Remark. We can actually bring this down to $O(nm)$ by the prudent step of setting

$$A_1 = \{a \in A : a \text{ is unused}\}$$

So all we need to do to show that this algorithm works is to establish the following result:

Theorem 4.7. *If there are no augmenting paths, the matching is maximum.*

The converse to this theorem is obviously true, and was the justification for our above suggested solution. The theorem itself, however, is not *a priori* obvious, though again, the proof is not difficult to follow:

Proof. Suppose M is not a maximum matching, but that there are no augmenting paths. Let M' be a maximum matching (so $|M| < |M'|$) and consider the *symmetric difference*

$$\begin{aligned} M_\Delta &= (M \backslash M') \cup (M' \backslash M) \\ &= \{\text{edges in exactly one of the two matchings}\} \end{aligned}$$

Each node in M_Δ has degree at most 2, because each node in M and M' has degree at most 1. Therefore, M_Δ consists only of *simple* paths and cycles. Further, in all paths and cycles, the edges alternate between being in M and in M'.

Now each cycle is obviously of even length as G is bipartite, and so contains equal numbers of edges from both matchings.

But $|M'| > |M|$, so there must be some simple path in M_Δ with more edges in M' than in M. By the alternating membership of edges, any such path must start and end with edges from the same matching (otherwise it has even length), which must be M'. Then this is an augmenting path with respect to M - a contradiction, as we assumed there were no such paths. □

Our solution works!

Another question we can ask about a bipartite graph is how many nodes we may select given relationships that exclude certain pairs of nodes. For example, recall that in the employee party problem, we disallowed the inclusion of two nodes when they were linked by an edge. (Note that a tree is always bipartite, because whenever two nodes have an edge between them, one has even depth and the other has odd depth.)

This is naturally phrased in this language of independent sets:

Definition 4.8. An *independent set* of a graph G is a set of nodes $S \subset V(G)$ such that no two nodes in S are joined by an edge in G.

Problem 4.9. Given a bipartite graph G, find a maximum independent set of G.

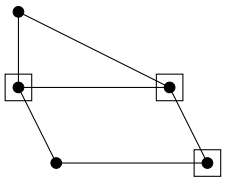
It is interesting to note the strong relationship of this problem to *vertex covers*.

Definition 4.10. A *vertex cover* of a graph G is a set of nodes $K \subset V(G)$ such that each edge of G has at least one endpoint in K.

We can then pose the following problem:

Problem 4.11. Given a bipartite graph G, find a minimum vertex cover.

In ths diagram shown, for example, the nodes in squares form a vertex cover, and the other nodes form an independent set. This relationship is actually totally general: in fact, this means that these two problems are exactly equivalent. We need to prove the following key theorem.



Theorem 4.12. *S is an independent set if and only if $V \backslash S$ is a vertex cover.*

Proof.

$\implies$: Consider some edge $e \in E$. If neither of e's endpoints is in $V \backslash S$, then they are both in S, which is a contradiction.

$\impliedby$: Take two nodes $a, b \in S$. If ab is an edge in G, then it is not covered by $V \backslash S$, so $V \backslash S$ cannot be a vertex cover, again a contradiction.

Hence these conditions are equivalent.

□

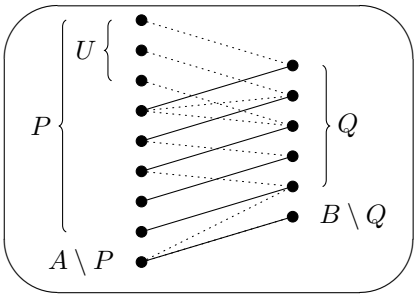
Remark. Note that this theorem holds for *all* graphs, not just bipartite ones.

What is striking, however, about the independent set/vertex cover problems is that they actually have a strong relationship to matchings in a bipartite graph.

Theorem 4.13 (König)**.** *For a bipartite graph G, the size of a maximum matching equals the size of a minimum vertex cover.*

Proof. We can see that $|M| \leq |K|$ for any matching M and vertex cover K, as for any edge $e \in M$, at least one of e's endpoints must be in K, and no two edges in M share an endpoint.

So if $M^\star$ is a maximum matching, and $K^\star$ a minimum vertex cover, we have $|M^\star| \leq |K^\star|$. Then if we can find a vertex cover K such that $|K| \leq |M^\star|$, we have $|M^\star| = |K^\star|$ and we are done.



So let $M^\star$ be a maximum matching, and let U be the set of unused nodes in A. Let W be the set of all nodes reachable from U via alternating paths; then let $P = W \cap A$ and $Q = W \cap B$.

There can't be an edge from P to $B \backslash Q$ (because otherwise the endpoint in B would be in W, and hence not in $B \backslash Q$). So $K = (A \backslash P) \cup Q$ is a vertex cover.

Note that all nodes in $A \backslash P$ are used, by definition of U, as are all nodes in Q (otherwise there would be an augmenting path). Also, there can't be a used edge between $A \backslash P$ and Q (because otherwise the endpoint in A is in P). So every node in K corresponds uniquely to an edge in $M^\star$. Therefore, $|K| \leq |M^\star|$, and by the above we are done. □

This surprising result also allows us to find a maximum independent set explicitly.

Exercise 4.14 (Rooks problem)**.** What is the largest number of rooks we can put on an $N \times N$ chessboard so that no two are attacking each other (that is, no two are in the same row or column)? Now find an algorithm to solve this problem given that there are some holes in the board where we can't place rooks (but over which they can still attack).

You should be able to attempt question 2 on Example Sheet 3 after this lecture. You might also like to try the Rooks Problem challenge, as suggested in the last exercise. If you are interested in programming, you can also attempt Problem 3-2 on ACOS.

Lecture 13: Maximum Flow

In this lecture, we move on to consider an apparently unrelated type of problem, more like the shortest path problems we dealt with previously. We are interested in the properties of networks, specifically *flow networks*. We will then see how this relates to the matching problems we dealt with previously.

4.3 Maximum Flow

Definition 4.15. A *flow network* $N = (V, E, c, s, t)$ is a weighted, directed graph (V, E) with two specified nodes, a *source* $s \in V$ and a *sink* $t \in V$. The weights given by $c : E \to \mathbb{N}$ are called *capacities*. (We will also consider non-integer capacities later, $c : E \to \mathbb{R}^+$.)

For the purposes of intuition, this is probably best thought of as a network of tubes of various widths, linking the *source* to the *sink*, with different sizes of pipe being able to carry different amounts of water along the routes to the sink.

With this in mind, we can imagine controlling the passage of water through the points in the network to realize various different *flows*. This is formalized as follows:

Definition 4.16. A *flow* f is a function $E \to \mathbb{N}$ with the following properties:

(**i**) for all edges $ij \in E$, the flow lies between 0 and the total capacity of the edge:

$$0 \leq f(i,j) \leq c(i,j)$$

(**ii**) for all vertices $v \in V \setminus \{s,t\}$ (excluding the source and the sink), the amount of flow entering the node equals the amount leaving:

$$\sum_{jv \in E} f(j,v) = \sum_{vi \in E} f(v,i)$$

The *value* of a flow is the total (net) amount of flow leaving the source, or equivalently (by conservation of flow, i.e. the second property), the amount of flow entering the sink,

$$|f| = \sum_{si \in E} f(s,i) - \underbrace{\sum_{js \in E} f(j,s)}_{\text{"back flow"}}$$

Remark. If i and j are not joined by an edge ($ij \notin E$), we write $f(i,j) = c(i,j) = 0$ for obvious reasons.

The obvious optimization problem is in fact one which comes up naturally in some contexts. For example, we might have road capacities, and want to find the most efficient way to get traffic through the system; or we might have electric load capacities, and want to get the maximum possible power

through an electricity grid; and so on. Also, as we will see over the coming lectures, some other interesting types of question can be recast as so-called *max flow* problems.

For now, however, we shall deal with the general abstract case:

Problem 4.17. Given a network N, find a flow in N with the maximum value.

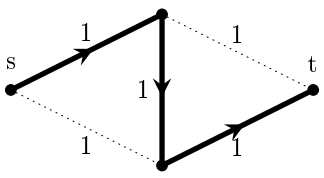
Inspired by our previous success in finding maximum matchings by repeatedly improving the estimated flow, we might try to find a sure-fire way of improving the network, and repeatedly loop that until we run out of 'augmentations', at which point we stop and hope we have a maximum flow. Thanks to the integer capacity constraint, we can always augment by an integer amount, so there is no danger of slow convergence through arbitrary rationals (or reals). We will actually prove this by using the fact that one integer-based algorithm we will exhibit returns a maximum flow: see Corollaries 4.24 and 4.25 below.

This time, our idea of an augmenting path is actually simpler that that for a maximum matching:

Definition 4.18. An *augmenting path* p in N is a path $(v_0, \cdots, v_n)$ with $v_0 = s$, $v_n = t$ and $c(v_{i-1}, v_i) > 0$ for all $1 \leq i \leq n$.

The first attempt at a solution, then, would be

Solution 1? Find an augmenting path p in N and decrease the capacities along p by $|p|$ to get a new network N', and increase the flow value by $|p|$. Then repeat this on N'.



Unfortunately, this does not work, as a bad choice of the first augmenting path will squeeze out the possibility of some others ever being used, as shown in the diagram - if we augment along the bold path, then we can only ever achieve a maximum flow of one. However, if it was possible to somehow reclaim the edges after making an augmentation, then finding another augmenting path allowing us to reverse the mistake we made would still represent a definite improvement in the total flow.

With this in mind, we shall not form the new network N' by simply removing the used capacity; instead, we shall add back edges to allow our changes to be reversed, defining the *residual network*:

Definition 4.19. Given a network N and flow f, the *residual network* N_f is $N_f = (V, E', c', s, t)$ with

$$E' = \underbrace{(E \backslash \{ij \in E : f(i,j) = c(i,j)\})}_{\text{removing full edges}} \cup \underbrace{\{ij \in V^2 : f(j,i) > 0\}}_{\text{adding back edges}}$$

and new capacities

$$c'(i,j) = c(i,j) - f(i,j) + f(j,i)$$

With this new approach, we can derive a new solution:

Solution 2? Find an augmenting path, augment along it, form the residual network, and repeat.

We shall prove, via the three following theorems, that this solution is in fact correct. It is called the *Ford-Fulkerson (FF)* algorithm.

Algorithm 17 Ford-Fulkerson

```
FOR each ij ∈ N:
    f(i,j) := 0
WHILE there is an augmenting path p in N_f:
    Increase f along p by |p|
```

First, we establish formally the termination requirement:

Theorem 4.20. *FF terminates on any network N.*

Proof. Let $f^\star$ be a maximum flow in N. By the definition of a flow,

$$|f^\star| \leq \sum_{ij \in E} c(i,j) < \infty$$

Since all capacities and flows are non-negative integers, $|p| \geq 1$ for all augmenting paths p. Thus FF terminates after at most $|f^\star|$ iterations. □

Complexity: The complexity implied by this is $O(m\,|f^\star|)$, where $|f^\star|$ is the value of a maximum flow, which is not necessarily polynomial in the edge capacities[22]. A different analysis will be carried out in the next lecture.

To prove that FF gives a maximum flow, we need some surprising additional theory which we have already touched upon before.

Definition 4.21. A *cut* S is a set $S \subset V$ with $s \in S$ and $t \notin S$. The *capacity* of the cut is the total capacity of all edges crossing the cut,

$$c(S) = \sum_{i \in S} \sum_{j \notin S} c(i,j)$$

The important property that a cut in a flow network has is that, because any flow $s \rightsquigarrow t$ must cross over the cut, the total value of the flow is bounded by the size of the cut:

[22] This may seem incorrect, but it is not. When we say an algorithm is *polynomial* in some input data, we mean it is polynomial in the *number of bits* necessary to specify it - so a linear-time algorithm will take twice as long to run when twice as many bits are needed. So something is polynomial in the size of a graph if if is polynomial in n, because n indicates the number of bits needed to describe all vertices. However, for every extra bit allowed to specify the edge capacities, we double the maximum *value* of an edge capacity, and hence the stated maximum run-time of this algorithm. This is actually exponential in the number of bits needed! We call an algorithm which is polynomial in the number of bits *and* the value expressed by those bits *pseudo-polynomial*.

Theorem 4.22. *For all cuts S and flows f, $|f| \leq c(S)$.*

Proof. From the definitions, we have

$$\sum_{j \in V} [f(v,j) - f(j,v)] = 0$$

for all $v \in S \backslash \{s\}$, and also

$$\sum_{j \in V} [f(s,j) - f(j,s)] = |f|$$

from which it follows that (as the value of the flow is precisely the total flow crossing the cut)

$$\begin{aligned} |f| &= \sum_{i \in S} \sum_{j \in V} [f(i,j) - f(j,i)] \\ &= \underbrace{\sum_{i \in S} \sum_{j \in S} [f(i,j) - f(j,i)]}_{0} + \sum_{i \in S} \sum_{j \notin S} \left[\underbrace{f(i,j)}_{\leq c(i,j)} - \underbrace{f(j,i)}_{\geq 0} \right] \\ &\leq c(S) \end{aligned}$$

□

Then it is sufficient to establish is that the flow returned by FF attains the capacity of some cut. This is indeed true:

Theorem 4.23. *Let f be the flow returned by FF. Then there is a cut S with $|f| = c(S)$.*

Proof. Let S be the set of nodes reachable from s in N_f. Since FF terminated, there is no path $s \rightsquigarrow t$, so $t \notin S$ and S is a cut.

For all i in S and j not in S, there is no edge ij in N_f. By the definition of the residual network, $f(i,j) = c(i,j)$ and $f(j,i) = 0$. Hence

$$\begin{aligned} |f| &= \sum_{i \in S} \sum_{j \notin S} [f(i,j) - f(j,i)] \\ &= \sum_{i \in S} \sum_{j \notin S} c(i,j) \\ &= c(S) \end{aligned}$$

□

Corollary 4.24. *FF returns a maximum flow.*

So our solution is correct! What is more, this gives us another very useful result for free:

Corollary 4.25 (Max-flow min-cut theorem)**.** *In every network N,*

$$|f_{max}| = \min_{S\ a\ cut} c(S)$$

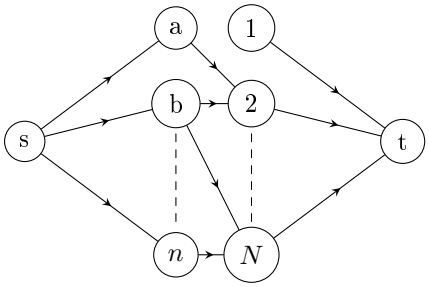
This means that we can solve so-called *min cut* problems by solving the dual *max flow* problem.

Exercise 4.26. Can you think of a way to handle a maximum flow problem with the additional constraint that at most m_i capacity can flow through vertex i?

All of these tools are very useful in solving practical problems. For now, we shall see one example of the use of max-flow approach, by solving a problem from the previous lecture.

Example 4.27 (Rooks problem)**.** What is the largest number of rooks we can put on an $N \times N$ chessboard so that no two are attacking each other (no two are in the same row or column), given that there are some holes in the board where we can't place rooks (but over which they can still attack)?

We can rewrite this problem as a maximum bipartite matching problem, with the rows on the left and the columns on the right. Then we link rows and columns if and only if there is no hole at the corresponding cell on the board, and the problem becomes to select as many (row, column) pairs as possible.



But we can turn this into a network flow problem. Create a source node on the left connected to each of the rows, and a sink node on the right connected to each of the columns. Next, orient all edges from left to right, and give them capacity 1. Then maximum flows in this derived network correspond exactly to maximum bipartite matchings in the original graph.

With a graph $G = (A \coprod B, E)$, maximum bipartite matching using the FF algorithm will then give the solution in $O(nm)$, because the max flow $|f|$ is bounded above by $|A| = O(n)$.

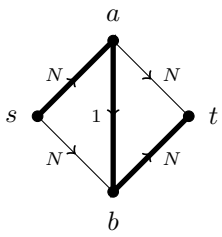
You should be able to attempt questions 3 to 8 on Example Sheet 3 after this lecture. If you are interested in programming, you can also attempt Problem 3-3 on ACOS.

Lecture 14: More Efficient Maximum Flow Algorithms

Recall that the Ford-Fulkerson algorithm had time complexity $O(m|f|)$, and that we assumed we had integer capacities in showing that FF terminated.

FF is in fact not entirely well specified as it stands, because we have not said how we *find an augmenting path*.

Doing this stupidly can lead to a highly inefficient algorithm. For example, consider the directed graph pictured. The maximum flow is $2N$ and this can be achieved in 2 augmentations. But if the augmenting paths use the edge ab or ba every time, we will take the maximum $2N$ iterations.



A reasonable measure to adopt is to ensure that we take the largest augmentation available each time.

Algorithm 4.28. *In each step, choose an augmenting path with maximum capacity. We can use a variation of Dijkstra's algorithm to do this.*

Exercise 4.29. Write down pseudocode for the modified Dijkstra algorithm - define the array `d[i]` to be the maximum flow that can be passed along a single path from the source s to the node i in the residual network N_f. Initially, we take `d[s]` $= \infty$.

So we can find each augmenting path in $O(m \log n)$ time - but how many times do we need to do this? The problem is that, even though we are now being sure to do the maximum amount of work possible at each step, we do not have any bound on how many such augmentations make up a flow. To obtain a bound, we will prove the following lemma:

Lemma 4.30. *Any flow f in a network N can be decomposed into at most m parts, each of which is either an $s \rightsquigarrow t$ path or a cycle.*

Proof. We prove this by constructing the set of paths and cycles. Call this set S; initially, $S = \emptyset$.

While there is a positive flow from s to t, find a path $s \rightsquigarrow t$ carrying positive flow, put it in S (decreasing flows along the corresponding edges in N appropriately) and repeat, until there is no positive flow from s to t.

While there is an edge with positive flow, follow it around until a cycle is found (there must be a cycle, by conservation of flow, since all $s \rightsquigarrow t$ flow has been removed). Put this cycle in S, decrease the flow to compensate, and repeat.

At the end of this process, S has size at most m because in each step we reduce the flow along some edge to 0, and we never increase the flow along any edge. □

This result now allows us to give a complexity analysis of Algorithm 4.28, and hence of the general case of FF.

Complexity of Improved Ford-Fulkerson

Suppose our current residual network admits a maximum flow of value $|f|$. By the above lemma, this flow is carried by at most m augmenting paths, so the capacity of a largest-capacity path is at least

$|f|/m$, and after augmenting along this path, the maximum flow in the residual network is now at most $\left(1 - \frac{1}{m}\right)|f|$. So after T augmentations, the maximum flow in the residual network is at most

$$\left(1 - \frac{1}{m}\right)^T |f|$$

This inequality allows us to calculate the number of iterations T required to bring the residual network's maximum flow down to 1, from which point we need at most one more iteration.

$$\begin{aligned}
\left(1 - \frac{1}{m}\right)^{T-1} |f| &\leq 1 \\
\left(1 - \frac{1}{m}\right)^{T-1} &\leq \frac{1}{|f|} \\
(T-1)\log\left(1 - \frac{1}{m}\right) &\leq \log\frac{1}{|f|}
\end{aligned}$$

The number of iterations, therefore, is at most

$$\begin{aligned}
\frac{\log 1/|f|}{\log\left(1 - 1/m\right)} + 1 &= \frac{-\log|f|}{-\frac{1}{m} - O\left(1/m^2\right)} \\
&= O\left(m \log|f|\right)
\end{aligned}$$

Hence the modified FF algorithm as a whole runs in

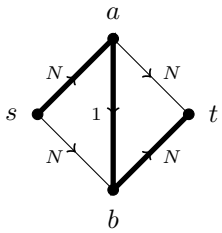
$$O\left(m^2 \log n \log|f|\right)$$

This still depends on the value of the maximum flow, but at least the dependency is now logarithmic rather than linear.

Eliminating Dependence on Flow Values

The analysis of the above algorithm became rather awkward because the algorithm depended inherently on (and indeed selected routes based on) the capacities of the edges, and this seemed to force us to include dependence on $|f|$ in our complexities.

The question then becomes: what other criterion could we use to choose our augmenting paths intelligently?



Recall that in the example at the start of the lecture, the problem was that we passed flow along a long path with low capacity. We have already tried avoiding low capacity paths. The other obvious graph traversal technique which would avoid this problem is to augment along the *shortest* $s \rightsquigarrow t$ path each time, which we can do using another modified form of BFS. Selecting augmenting paths in this way is called the *Edmonds-Karp* algorithm.

Algorithm 4.31 (Edmonds-Karp)**.** *At each step, choose an augmenting path in the residual network with the fewest number of edges.*

Remark. We will use the notation $d(i,j)$ to mean the minimum number of edges in an $i \rightsquigarrow j$ path in the following discussion.

In our modified Ford-Fulkerson algorithm, we selected paths with the largest capacities, and we gradually decreased the capacity remaining until the algorithm terminated. Now, we are selecting paths with the shortest lengths - and consequently, we are increasing the length of $s \rightsquigarrow t$ paths. So in order to prove that this terminates, we want to prove a result like the following:

Lemma 4.32. *Under the Edmonds-Karp algorithm, $d(s,i)$ and $d(i,t)$ in the residual network are non-decreasing.*

Proof. Suppose that after some augmentation, some node moves closer to s. Let i be one of the closest such nodes. After the augmentation, there is, under our assumption, a shortest $s \rightsquigarrow i$ path in the new residual network, so let j be the immediate predecessor of i on such a path.

Again by assumption, we know j did not move closer to s - hence the edge ji must be a *new* edge in the residual network, added during the augmentation; hence the augmentation passed flow from i to j, as this is the only reason an edge would be created.

Therefore, before the augmentation, $d(s,i) < d(s,j)$, because of the way we select paths - we would not push flow along $i \to j$ unless $i \to j$ was part of a shortest path $s \rightsquigarrow t$.

But during the augmentation, we know $d(s,j)$ did not decrease, and $d(s,i)$ did, so this inequality still holds:

$$d(s,i) < d(s,j)$$

This is a clear contradiction to j being a predecessor to i on a shortest path from $s \rightsquigarrow i$, so therefore the distances were in fact non-decreasing.

A similar argument applied to the distances to the sink t gives the required result. You can verify this as an exercise. □

Now what we know about an augmentation is that we always augment along the $s \rightsquigarrow t$ path by the largest possible amount - hence, there is always some edge on the path which limits the augmentation. We say this edge is *saturated* or *critical*. This, along with the strictly increasing distance property, is enough to give us a useful upper bound on the number of iterations required:

Lemma 4.33. *Edmonds-Karp requires at most $\frac{mn}{2}$ iterations.*

Proof. As noted above, in any augmenting path, we saturate some edge ij. Then we cannot saturate this edge again (or indeed use it at all) unless we send flow back along ji at some point.

An augmentation from i to j will only occur if

$$d(s,i) < d(s,j)$$

An augmentation from j to i will only occur if

$$d(s,j) < d(s,i)$$

Therefore, between any two successive saturations of ij, $d(s,i)$ must increase by at least 2, as these values are all non-decreasing. Hence ij can be saturated at most $n/2$ times, since $d(s,i)$ cannot exceed $n-1$.

But this holds for all edges, so in total we saturate at most $\frac{mn}{2}$ edges, and thus we need at most this many augmenting paths, as claimed. □

Complexity: Since each BFS is $O(m)$, the overall complexity of Edmonds-Karp is

$$O(m^2 n)$$

Perhaps the most impressive property of Edmonds-Karp, given how we arrived at it, is that it will work perfectly even for arbitrary positive capacities - not just for any non-integer values, but irrational values (assuming they are stored correctly). This is because the proofs above do not rely in any way on the properties of capacities.

Remark. These algorithms are not the best possible. For example, the *Dinitz blocking flow* algorithm, which effectively works by using BFS techniques to find multiple augmenting paths simultaneously, runs in $O(n^2 m)$, and in $O(n\sqrt{m})$ for graphs with edges all of capacity 1. A further modification to this algorithm brings it down to $O(nm \log n)$ (an example of a smart data structure - in this case *dynamic trees* - improving performance). Also, there is a family of fast *push-relabel* algorithms.

You might like to try the Plane Scheduling challenge before watching or reading the next lecture, as a solution will be presented there.

Lecture 15: Applications of Maximum Flow

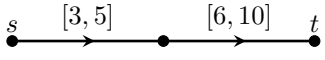
We now consider some extensions and applications of maximum flow.

In the following, we will write $c_{ij} = c(i,j)$ and $f_{ij} = f(i,j)$ to avoid drowning in brackets.

Lower Capacity Bounds

The first generalization we consider is a fairly natural one:

Problem 4.34. Suppose that, in addition to the edge capacities c_{ij}, each edge also has a lower capacity bound $l_{ij} \geq 0$, so that we require $f_{ij} \in [l_{ij}, c_{ij}]$ for all edges ij. Find a maximum flow, if one exists.

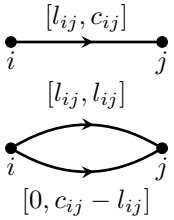


The first thing to be clear about is that there is absolutely no guarantee that a solution (or a *feasible flow*) will actually exist. This was not a problem when $l_{ij} = 0$ for all edges, because the 0 flow was always feasible.

In fact, however, once we have a feasible flow, we can just use the 'augmenting paths' approach to get a maximum flow, so long as we note that the residual capacity of back edges is now $f_{ij} - l_{ij}$ rather than just f_{ij}.

So the only new work we need to do is to try to find a feasible flow in the network. There is an approach in *circulation theory*, which is essentially network flow theory with lower bounds and cost per unit flow on edges. Whilst we will consider minimum-cost flows in the last lecture, we want to avoid introducing new theory to address this question.

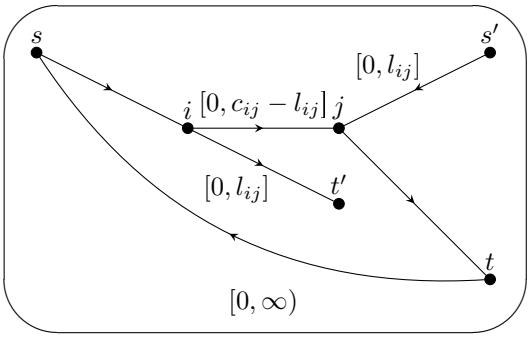
This is fortunately reasonably easy to do, though not necessarily easy to come up with.



The key idea is to make a small transformation as shown here - we transform every edge into two edges, one of which has a fixed flow, and the other of which has no lower bound.

Using these ideas, we can gradually build up the following solution:

Solution. Modify the original network initially by transforming the bounds on each edge from $[l_{ij}, c_{ij}]$ to $[0, c_{ij} - l_{ij}]$. Now there is a feasible flow if and only it is possible to have a flow where l_{ij} units of flow leave i and l_{ij} units enter j in addition to any other value along the new ij edge.



But this is exactly equivalent to having a source introducing l_{ij} units of flow at j, and a sink removing l_{ij} units of flow at i.

So if we introduce precisely such a new source s' and sink t', connected by edges $[0, l_{ij}]$ to j and i respectively, then if there is a maximum flow which saturates these edges, there is a feasible flow.

The only problem is that we now have a network with two sources and two sinks - but this is easily rectified. Since we don't actually care about the value of our feasible flow, we can just consider s and t to be ordinary nodes, and allow arbitrarily large amounts of flow to pass from t back to s again (in the feasible flow, this will simply be the value of the flow).

So our transformation works as follows: make the source and sink the new nodes s' and t', and for each edge ij, replace it with

- an edge $s'j$ with $f_{s'j} \in [0, l_{ij}]$
- an edge it' with $f_{it'} \in [0, l_{ij}]$
- an edge ij with $f_{ij} \in [0, c_{ij} - l_{ij}]$

Also, add an edge ts with $f_{ts} \in [0, \infty)$. (This does not break our maximum flow algorithms, since the total maximum flow is still bounded.)

Then find a maximum flow $s' \rightsquigarrow t'$. Flow will go from s' to j to t to s to i to t'. If the value of the maximum flow is $\sum_{ij \in E} l_{ij}$, then we have a feasible flow - otherwise, there is no feasible flow.

From this point on, we are free to augment more or less as usual.

Vertex Covers

Recall König's theorem (Theorem 4.13): for a bipartite graph G, the size of a maximum matching equals the size of a minimum vertex cover.

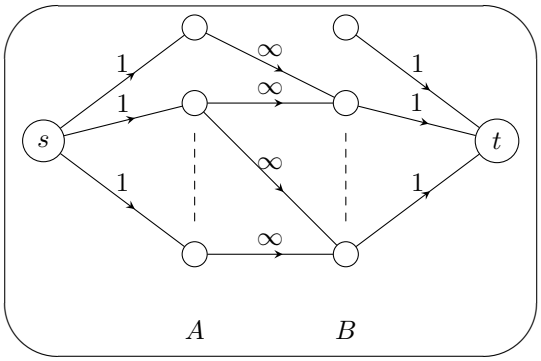
Also, recall the max-flow min-cut theorem (Corollary 4.25): the size of a maximum flow corresponds to the capacity of a minimum cut.

Idea: We know that a maximum matching corresponds to a maximum flow. So can we relate vertex covers to cuts?

The answer is in the affirmative - in fact, this will give us an alternative proof of König's theorem:

Proof. Clearly the size of a maximum matching is at most the size of a minimum vertex cover, as we noted in the original proof.

So suppose $G = (A \coprod B, E)$ is bipartite, and consider the network N shown:



Importantly, note that the infinite capacities do not affect the solution for the maximum flow. So this maximum flow in N corresponds to a maximum matching in G. Let S be the corresponding minimum cut. (Clearly, this must be of finite value, because e.g. the cut with all edges of the form si has a finite capacity $|A|$.)

Now let $K = \{i \in A : i \notin S\} \cup \{i \in B : i \in S\}$. Suppose $ij \in G$, with $i \in A$ and $j \in B$. The only way K can fail to cover ij is if $i \in S$ and $j \notin S$. But this means ij is saturated, which is impossible as $c_{ij} = \infty$ (which is why we made that choice).

So K is a vertex cover - what is more, it has size equal to the cut S's capacity: you can see this by first noting all edges straddling the cut must be of capacity one, since the minimum cut has finite capacity. You may like to construct an example to see this. Hence:

$$\text{the size of the maximum matching} = \text{the value of the maximum flow} = \text{the capacity of the minimum cut} \geq \text{the size of the minimum vertex cover}$$

and hence we are done. □

Unusual Graph Transformations

The final application of graph theory which we will consider in this lecture is to a specific problem which it is not obvious should be phrased in the language of graphs - or at least not in the way you might expect. It is useful to see how problems can be gradually transformed into a format we know how to deal with.

Problem 4.35 (Plane Scheduling)**.** Given a set of flights with origins, destinations, departure times and arrival times, what is the minimum number of planes needed to make all the flights?

The slightly misleading element of this problem is that physical intuition seems to demand that nodes be the airports and edges the flights, which is entirely unhelpful.

Instead, what we need is a graph encoding information about *compatible* flights - that is, ones which a single plane can make. Influenced by the success of graph techniques on maximum matchings, it seems a good idea to make nodes correspond to flights, and edges to compatible flights.

Solution. Construct a graph $G = (V, E)$ where V is the set of flights, and $ij \in E$ iff the same plane can make both flight i and *then* flight j (so i occurs *before* j).

Clearly, due to the time ordering, G contains no cycles. What we want to find is the minimum number of directed paths that can cover all of the nodes in G (that is, so that every node lies in some path).

The next stage in this solution is not at all immediately obvious. Let us recall the analogy we drew above with compatibility in bipartite graphs. We are trying to match up as many pairs of flights as possible (in order to reduce the number of planes needed). So can we restate the problem of making these 'pairings' in a form that we have seen before? The answer is yes - we are trying to match up finishing times of earlier flights with the start times of later flights (recall the problem of maximizing the number of events one person could attend in the greedy algorithms lecture, on page 10). If we create a new graph to reflect this, we can easily make it bipartite.

So we take G and form a new graph G' by splitting each node i into *two* nodes i_1 and i_2. Then, we draw the edge $i_1 j_2$ if and only if there was originally a connecting edge, i.e. $ij \in E$.

Now a path in G corresponds to a sequence of edges

$$a_1 b_2, b_1 c_2, c_1 d_2, \cdots$$

in the new graph G'. We want to find the minimum number of paths needed to cover G - that is, we want to use as many edges of the form $i_1 j_2$ as possible. But now G' is clearly a bipartite graph, and want we want to do is precisely to find a maximum matching in it! This we have all the equipment we need for, so we are done.

Remark. Note that each time we add an edge to the matching, we decrease the number of planes needed by 1, so the size of the matching is equal to

$$n - \{\text{the corresponding number of paths}\}$$

where n is the number of flights. In particular, the size of a maximum matching is

$$n - \{\text{the minimum number of paths}\}$$

So by König's theorem, and the relationship between vertex covers and independent sets,

$$\begin{aligned} \text{the size of a maximum matching} &= \text{the size of a minimum vertex cover} \\ &= n - \text{the size of a maximum independent set} \end{aligned}$$

Hence the minimum number of planes needed is the maximum number of flights no two of which can be made by the same plane - a very down-to-earth result to derive so abstractly!

Lecture 16: Minimum-Cost Flow

The final type of problem which we will consider is an important variation on the theme of the network flow problem.

4.4 Minimum-Cost Flow

Problem 4.36 (Assignment Problem)**.** There are n workers and n jobs. Assigning worker i to job j incurs a cost a_{ij}. Match workers to jobs such that the total cost is minimized.

We know how to rephrase matching problems like this in terms of network flows: we form the corresponding bipartite graph, and attach a source on one side and a sink on the other. In this case, with unit capacities, we would be looking for some flow of value n. This is obviously trivial with this problem in the sense that all edges exist (although we could effectively have $a_{ij} = \infty$, instead of actually removing the edge ij - then if we get a minimum cost matching with cost infinity, we know it is impossible to make a matching without these forbidden edges). Note that it is implicit that the edges attached to the source and sink have cost 0.

Where this obviously differs is that it is advantageous to use some particular edges rather than others, even with all other things being equal (note that if we were to look for a minimum cost flow of value 1, this is a shortest path problem - you may like to check this, noting that we can send non-integer flow along the edges too). Therefore, this is a special case of the following problem.

Problem 4.37. Given a network N, whose edges have a cost per unit flow a_{ij} and an (integer) capacity c_{ij}, find a *minimum-cost* flow of a given value v.

There are actually many ways to solve this problem; the approach we shall adopt is to extend ideas we have already seen.

If we were to use similar ideas to our solution for the maximum flow problem, we would proceed to find augmenting paths until we reach the target value v; the obvious way of doing this is greedy:

Idea: Find minimum-cost augmenting paths from s to t, to value v.

To be clear about what we mean by this:

Definition 4.38. The *cost* of an augmenting path p is the sum of the costs of its edges. This gives the net cost incurred in pushing one unit of flow along P. Note that we can have edges of negative cost, since these will necessarily arise when we introduce *back edges*.

It is reasonable to expect, however, that this alone might not be sufficient to guarantee finding an overall minimum cost flow - that is, when we get a flow of value v, we might not have one of minimum cost.

It would therefore be useful to be able to improve the performance of a flow of *fixed* value by finding routes along which we can alter flow without changing the *net* flow but changing the net cost. This motivates the following definition.

Definition 4.39. An *augmenting cycle* C is a (directed) cycle whose edges all have positive capacity. The *cost* of C is the sum of the costs of its (directed) edges.

Clearly, we would improve our flow if we could remove all negative-cost augmenting cycles by augmenting along them, in much the same way as when we exploited back edges when first coming up with the maximum flow algorithms. As with the maximum flow arguments, this only works if our repeated augmentations converge to the optimum solution.

Theorem 4.40. *A flow f of value v has minimum cost (among all flows of the same value) if and only if there is no negative-cost cycle in the residual graph N_f.*

Proof.

$\Longrightarrow$: Suppose N_f has a negative-cost cycle. Then we can augment f around this cycle to get a cheaper flow of the same value.

$\Longleftarrow$: Suppose f is not a minimum-cost flow. Then there is a cheaper flow g of the same value v. The flow $g - f$ has value 0, and hence can be decomposed into a union of augmenting cycles, by Lemma 4.30, and obviously at least one must have negative cost, since $g - f$ is of negative cost. Also, the cycles must be augmenting in f because all of the edges exist in N_f (i.e. they have some capacity remaining).

□

This looks promising. In fact, there is another result we have as a consequence:

Theorem 4.41. *Let f be a minimum-cost flow of value v, and let P be a minimum-cost augmenting path in N_f of value δ. Then augmenting f along P by δ gives a minimum-cost flow g of value $v + \delta$.*

Proof. Suppose this did not hold. Then by the previous theorem, g admits a negative-cost cycle C. Since f admitted no negative-cost cycles, there must be some edge $ij \in P$ on our path with $ji \in C$. Then

$$(P \setminus \{ij\}) \cup (C \setminus \{ji\})$$

or some subset thereof is a cheaper augmenting path than P, which is a contradiction. □

Hence we have the following solution, called the *successive shortest paths* algorithm, which is very much like a generalized Ford-Fulkerson algorithm:

Solution. Start with $f = 0$. Keep finding negative-cost augmenting cycles, augmenting along them, until there are no more. At the end, we have a minimum-cost zero flow. Then, by repeatedly

finding minimum-cost augmenting paths $s \rightsquigarrow t$ (using Bellman-Ford), we will either run out of augmenting paths (in which case there are no flows of value v at all) or we will be able to reach v (since we do not *have* to saturate an edge when we augment!) and hence we are done.

Complexity: Ignoring the first stage (i.e. assuming there are no negative cost cycles with respect to a_{ij}, i.e. that the zero flow is minimum cost, which is often a reasonable assumption) the time complexity - recalling that v is an integer - is

$$O(nmv)$$

because at most v augmenting paths must be found as we always augment by at least value 1, and as we know from analyzing Bellman-Ford we can do this in $O(nm)$ time.

There are alternative approaches to this problem.

For example, the *cycle-cancelling* algorithm uses the first theorem only, and works by first establishing any feasible solution of the desired value, and then finding negative cycles. Note that if we assume the zero flow is minimum cost, this unnecessarily complicates matters (compared to the successive shortest path algorithm). There is a variation on the cycle-cancelling algorithm called the *minimum mean(-value) cycle cancelling* algorithm.

There is also a primal-dual algorithm called *cost-scaling* which is similar to our successive shortest path approach, which augments along all shortest paths simultaneously. This generalizes the push-relabel scheme of maximum flow algorithms.

This problem can also be solved via techniques in linear programming.