Advanced Algorithms I

Todd Davies

January 28, 2016

Introduction

This course unit has two objectives. The first is to introduce the student to a range of advanced algorithms for difficult computational problems, including matching, flow networks and linear programming. The second objective is to outline the mathematical techniques required to analyse the complexity of computational tasks in general. There are two pieces of assessed coursework, and an exam at the end.

Aims

This unit provides an advanced course in algorithms, assuming the student already knows algorithms for common computational tasks, and can reason about the correctness of algorithms and understand the basics of computing the complexity of algorithms and comparing algorithmic performance.

The course focuses on the range of algorithms available for computational tasks, considering the fundamental division of tractable tasks, with linear or polynomial-time algorithms, and tasks that appear to be intractable, in that the only algorithms available are exponential-time in the worst case.

To examine the range of algorithmic behaviour and this fundamental divide, two topics are covered:

- Examining a range of common computational tasks and algorithms available; We shall consider linear and polynomial-time algorithms for string matching tasks and problems that may be interpreted in terms of graphs. For the latter we shall consider the divide between tractable and intractable tasks, showing that it is difficult to determine what range of algorithms is available for any given task.
- Complexity measures and complexity classes: How to compute complexity measures of algorithms, and comparing tasks according to their complexity. Complexity classes of computational tasks, reduction techniques. Deterministic and non-deterministic computation. Polynomial-time classes and non- deterministic polynomial-time classes. Completeness and hardness. The fundamental classes P and NP-complete. NP-complete tasks.

Additional reading

Title	Author	Year
Core Algorithm design: foundations, analysis and internet examples	Goodrich, Michael T. and Roberto Tamassia	2002
Introduction to the theory of computation (3rd edition)	Sipser, Michael	2013

Attribution

These notes are based off of both the course notes (`http://studentnet.cs.manchester.ac.uk/ugt/2015/COMP36111/`) and *Introduction to the theory of computation* by Michael Sipser (the course reading). If you find any errors, then I'd love to hear about them!

Contents

1 Algorithmic Wisdom

The first two lectures of the course describe different types of algorithms, what is computable, where to optimise algorithms, asymptotics and heuristics.

1.1 Different types of algorithms

There are three types of algorithms that are mentioned:

> Note:
>
> You should know all of this from `COMP26120`.

Divide and conquer

These algorithms continually break a problem down into smaller parts, which are easier to solve, until eventually, the problems are trivial and easily solved. This is often used when the data you're operating on is in a recursive datastructure such as a tree. If you're writing an algorithm to find how many nodes there are in a tree, then you could use divide and conquer:

```
int countTreeSize(tree) {
  int size = 1;
  if (tree.left) size += countTreeSize(t
  if (tree.right) size += countTreeSize(
  return size;
}
```

As you can see from the example, divide and conquer algorithms are usually recursive.

The divide and conquer technique can be applied to graphs, but in order to do this, you must keep track of which nodes you've visited with a flag on each node. If we wanted to count the nodes in a graph, we could do:

```
int countGraphSize(graph) {
  if (graph.visited) return 0;
  graph.visited = true;
  int size = 1;
  for (child in graph) {
    size += countGraphSize(child);
  }
  return size;
}
```

Mutual Recursion

Mutual recursion describes an algorithm that operates on data where one type of data can reference another, and the other can reference it. The example given is that of statements and expressions in programming languages; statements contain expressions, and expressions can also contain statements. Parsing such a structure might involve two algorithms that recurse on each other!

Dynamic Programming

Dynamic programming exploits the fact that when some problems are broken down into smaller sub-problems, some of the sub-problems are identical. Dynamic programming algorithms start from the very smallest sub-problems and build up to the final solution, and usually cache results to sub-problems in a table so that work is not done twice.

1.2 Computability

There are many different definitions of computability, including lambda calculus, Turing machines, rewriting rules, random access machines and (many) more. The idea that relates all of these things, is that they all have the same capabilities. That is to say that if you can compute something using one of these ideas, then you can also compute it on all the others too.

There are also a class of 'alternate' computing mechanisms, such as quantum computers and neural computers. These ideas have the potential to compute things that a Turing machine (or its equivalents cannot), but they are significantly harder to build, and functional implementations do not exist yet.

1.3 Asymptotics and optimisation

When you have to get a computer to perform a task, implementing a simple algorithm first is a good idea, since you will at least have something to demonstrate to people, and you will gain a good understanding of the problem at hand. However, simple algorithms are often slow; how should we evolve our implementation to be as fast as we need it to be?

Profiling can tell you where your code is spending most of its time. Sometimes your algorithm will be really fast, and the processor will spend most of its time waiting for IO to give it more data; this is often the case with GPU

computation.

Assuming you find some CPU bottleneck in your code, before you spend hours making it faster, consider whether it is worth the effort. If this part takes up 10% of your runtime, and you make it twice as fast, your program will only run 5% faster. This is an example of the Law of Diminishing Returns.

As well as optimising specific parts of an algorithm, you also should consider its asymptotic run time. An algorithm that runs in $O(n^2)$ time is probably not going to be better than one that runs in $O(nlog(n))$ time. However, this isn't always the case; some algorithms (often ones with good asymptotic run times) take a long time to set up, usually when you have to transform the data into some different datastructure. If you are running your algorithm on a small amount of data, then an algorithm that you can run on your data *as is* might outperform a fancier algorithm that you have to invest more overhead in.

Note:

Sometimes a good solution is to use different algorithms depending on the input. If there are only a few cases that produce worst-case performance, you could even hard-code solutions to those!

The average case runtime of a algorithm is also important. Haskell uses a type checker that runs in $O(2^{n^n})$ time in the worst case, but for every program that isn't made specifically to mess with the compiler, it runs in linear time.

2 Graphs

A graph is a pair $G = (V, E)$ where V is a finite set, and E is a set of pairs between items in V. Elements in V are *vertices* or *nodes*, and elements in E are *edges*.

Different mathematicians have different rules about what exactly can go in a graph. For the purposes of this course, the graphs shown in Figure 1 aren't allowed.

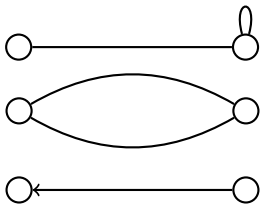


Figure 1: Self loops, duplicate edges and arrows are not allowed.

A **directed graph** is just like a normal graph, except the edges do have arrows. The only mathematical difference is that the set E is a set of ordered pairs. The third diagram in Figure 1 is a directed graph (it is always referred to as a directed graph!).

The *degree* of a node in a graph, is the number of edges that are adjacent to (touching) it. If the graph is directed, then it is the number of edges originating from the node.

A weighted graph is one where each edge is associated with a value representing its weight. The length of a path between nodes is simply the sum of the edge weights connecting the nodes.

2.1 Connectivity

A node a is **reachable** from another node b if there is some sequence of nodes connected by edges that go from the a to b.

A graph where every node is reachable from every other node is **connected**. A strongly connected graph is a **directed graph** where each node is reachable from each other node.

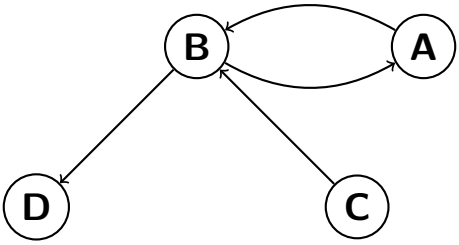


Figure 2: This graph is not strongly connected, but the subgraph with only nodes A and B is. If we were to remove the arrows, then the graph would be connected.

2.2 Representing graphs

We can store graphs in computer memory in two ways:

Adjacency List
There are two different kinds of adjacency list, the first is like so:

```
public class Graph {
  List<Vertex> nodes;
  List<Edge> edges;
```

```
}
public class Edge { Vertex a, b; }
public class Vertex { List<Edge> outl
```

Here, we keep a list of all the nodes, and a list of all the edges. From any edge, we can see what nodes it connects, and from any node, we can see what edges it connects.

The other type of adjacency list is a bit simpler, but less efficient in some cases:

```
public class Graph<T> {
  Set<T, List<T>> adjList;
}
```

Adjacency Matrix

Here, a matrix indicates whether there is an edge between two nodes:

	A	B	C	D
A	0	1	0	0
B	1	0	0	1
C	0	1	0	0
D	0	0	0	0

This adjacency matrix represents the graph in Figure 2. You can see that it is fairly wasteful in terms of memory $O(|E| * |V|)$, though with bit arrays, it has a very low constant overhead.

2.3 Classifying Edges

During a *depth first traversal* (Section 2.4.1) of a graph, we can classify each edge into one of four types. When doing the traversal, we process each edge from left to right (from the viewer's perspective), and we define an edge to be an *ancestor* of another edge if there is a path from the ancestor edge to the descendent edge and the ancestor is traversed first. The four types of edges are:

Tree edges
: These edges lead to a new node during a search. If you remove all the edges from the graph except tree edges, then you get a tree!

Forward edges
: These go from ancestor edges to descendent edges, but are not tree edges (i.e. the descendent node has been visited already).

Cross edges
: These go between edges where no node is the ancestor of the other.

Back edges
: An edge that goes from a descendant to an ancestor.

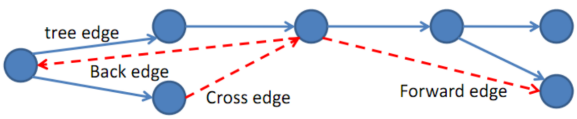


Figure 3: A pictorial illustration of the different edge types.

2.4 Graph Algorithms

You will have covered some of the algorithms featuring here in previous courses, or perhaps seen them in the wild. They are however interesting and useful, so it's worth a recap even if they're not new! I've ordered these in roughly increasing order of mental strain.

2.4.1 Depth first search

Depth first search (DFS) is an algorithm to find a node in a graph starting from another node. It works on both directed and undirected graphs, and runs in $O(|V| + |E|)$ (linear) time. The psudo code looks like this:

> Note:
>
> Note how we mark nodes as having been visited (by adding them to `visitedNodes`). This is so if there is a loop in the graph, the algorithm doesn't run indefinitely.

```
Node dfs(Node haystack, Node needle) {
  Stack<Node> toVisit = new Stack<>();
  Set<Node> visitedNodes = new Set<>();
  toVisit.push(haystack);
  while(!toVisit.isEmpty()) {
    Node node = toVisit.pop();
    if (visitedNodes.contains(node)) continue
    visitedNodes.add(node);
    if(node.equals(needle)) {
      return needle;
    } else {
```

```
            for(Node child : node.children) {
              toVisit.push(child);
            }
          }
        }
        return null;
    }
```

A Breadth First Search is the same, except you use a `Queue` instead of a `Stack`.

> Note:
>
> To see if a graph is connected, do a depth first search as in the example code, but don't stop when you find a needle, only stop when the `toVisit` stack is empty. If `visitedNodes` contains all of the nodes in the graph, then the graph is connected.

Depth first search also lets you find if one node is reachable from another in linear time, and also if a graph is connected in linear time too, with a few modifications.

We can also find out if a directed graph is strongly connected in $O(|V| + |E|)$ time using Tarjan's algorithm, which we'll see later.

2.4.2 Dijkstra's algorithm

Dijkstra's algorithm finds the undirected shortest path between two nodes in a graph. Here is the psudo-code:

Note:

My best advice for learning an algorithm like this, is to get a whiteboard, draw out the problem, and then run the solution manually on the whiteboard. Then you have a pictorial and visual explanation of how the algorithm *really* works.

```
function Dijkstra(Graph, source):
  create vertex set Q

  // Initialization
  for each vertex v in Graph:
    dist[v] = INFINITY
    prev[v] = UNDEFINED
    add v to Q

  // Distance from source to source
  dist[source] = 0

  while Q is not empty:
    u = vertex in Q with min dist[u]
    remove u from Q

  for each neighbour v of u:
    alt = dist[u] + length(u, v)
    // A shorter path to v has been found
    if alt < dist[v]:
      dist[v] = alt
      prev[v] = u
```

```
return dist[], prev[]
```

Listing 1: Dijkstra's algorithm (from Wikipedia)

> Note:
>
> Remember, the maximum value of E is actually V^2, and you can sometimes use that to play with the runtime bounds of graph algorithms (e.g. you could show something is linear in the number of edges, or polynomial in the number of nodes).

If we use a Fibonacci heap, for the priority queue, then the runtime of Dijkstra's algorithm is $O(|E| + |V|log(|V|)$. If we use a normal heap, then the runtime is $O((|E| + |V|)log(|V|))$.

2.4.3 Tarjan's Algorithm

Before we delve into Tarjan's Algorithm, we need to discuss strongly connected components. You will remember from Section 2.1, that a graph is strongly connected if all nodes are reachable from any other node.

A *strongly connected component* is a subset of edges within a graph, where the subset is strongly connected. If a graph has only one strongly connected component, then it is strongly connected.

The psudo code here is from Wikipedia. If you're reading this after week 8 of the `COMP36111` course, then you probably have your own implementation in C.

```
algorithm tarjan is
input: graph G = (V, E)
output: set of strongly connected components

index := 0
S := empty
for each v in V do
  if (v.index is undefined) then
    strongconnect(v)
  end if
end for

function strongconnect(v)
  // Set the depth index for v to the smalles
  v.index := index
  v.lowlink := index
  index := index + 1
  S.push(v)
  v.onStack := true

  // Consider successors of v
  for each (v, w) in E do
    if (w.index is undefined) then
      // Successor w has not yet been visited
      strongconnect(w)
      v.lowlink  := min(v.lowlink, w.lowlink)
    else if (w.onStack) then
      // Successor w is in stack S and hence
      v.lowlink  := min(v.lowlink, w.index)
    end if
  end for
```

```
    // If v is a root node, pop the stack and g
    if (v.lowlink = v.index) then
      start a new strongly connected component
      repeat
        w := S.pop()
        w.onStack := false
        add w to current strongly connected com
      while (w != v)
      output the current strongly connected com
    end if
  end function
```

The algorithm basically applies a depth first search to the graph, and keeps track of two properties (the index and the lowlink) for each node. The index is the logical time at which the node was discovered by the algorithm, and the lowlink is the index of the oldest ancestor that can be reached from the current node.

Once the index has been assigned, it never changes. However, the lowlink is updated during recursive depth first search calls to find back edges between the node and its ancestors.

If any node has its index as the same value as its lowlink, then it is the root of a strongly connected component (possibly only containing itself). Nodes will be popped off the stack until the current node is removed, at which point we have found all the nodes in this component.

If all nodes have not been discovered after a single depth first search, the algorithm is run again on a previously undiscovered node. This ensures that all the strongly connected

components are found, even in graphs that have completely disjoint components.

2.4.4 Graph Colouring

Graph colouring is a problem where we want to assign colours to nodes in a graph, but no adjacent node can have the same colour. A *k-colouring* of a graph G is a function $f : V \to \{0, 1, \ldots, k-1\}$ such that for any edge $(u, v) \in E, f(u) \neq f(v)$, and the output from f is each node's colour.

For some graphs, there is no valid colouring with k colours. A graph is *k-colourable* if there exists a *k-colouring* for that graph.

Of course, we can write a Turing Machine that can solve *k-colourability*. We can encode a graph by writing n as the number of nodes, and the pairs as the edges between them:

$$(n, (u_1, v_1), \ldots, (u_m, v_m))$$

See Section 8.2 for more on this!

2.4.5 Hamiltonian Circuits

You might want to read the later sections about complexity theory before this bit.

A circuit of a graph is a path through the graph where every node is visited at least once. An Eulerian circuit of a graph is where each edge is traversed *exactly* once, and a Hamiltonian circuit is one where each node is visited exactly once.

Note:

Finding if a graph has a Eulerian Circuit is solvable in PTime, since we simply have to check if every node has an even out-degree, which can be done in $O(n^2)$.

We're going to show that finding a Hamiltonian Circuit is a problem in NPTime. We can do this in one of two ways; either to give a nondeterministic Turing machine that will find a path in polynomial time, or we can come up with a deterministic polynomial time algorithm to verify that a given path is Hamiltonian.

You might be thinking *"ummm, how can simply giving a polynomial time verification algorithm show that we're in NPTime?"*; that's what I thought at first. However, we could simply try all combinations of input to the verifier and see if there is one that is Hamiltonian. This would work in deterministic ExpTime, which can be computed in NPTime.

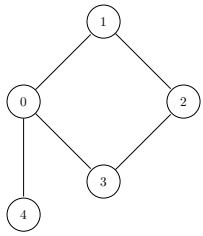
In order to show that trying to find a Hamiltonian path in a graph is NPTime-Complete, we must map it to another NPTime-Complete problem. The course notes (lecture 9) does this by mapping it to 3-SAT like we do for graph colouring in Section 8.2.

We can use the fact that finding Hamiltonian Circuits is NPTime-Complete to show that the Travelling Salesman problem is NP-Hard, and then NP-Complete.

Note:

I've not shown how I computed the path with the matrix, but I did it on a whiteboard by crossing out used paths on the matrix. Giving a full explanation here would take too long now, and its not strictly on the syllabus.

To do this, we construct a $N \times N$ matrix, where 1 corresponds to an edge between nodes and 2 is for no edge. If we had a graph like in Figure 4a, then we could produce a matrix like in Table 4b.



(a) The input graph

	0	1	2	3
0	2	1	2	1
1	1	2	1	2
2	2	1	2	1
3	1	2	1	2
4	1	2	2	2

(b) The resulting matrix; start at 3, move to 2, then 1, then 0 and finally 4.

Figure 4: Transforming the Hamiltonian Cycle problem to the TSP problem.

If the TSP problem finds a path with a total cost of less than or equal to n, then there will be a Hamiltonian path, like in the figure. This means that TSP is `NP-hard`. To show that it is `NP-Complete`, observe that TSP is in `NP`, which is easy, since we can obviously verify that a given path can be taken with a specific cost in polynomial time.

3 Linear Programming

Linear programming is an optimisation problem, where we want to find the 'best' solution to a set of equations. We're going to solve it using the *simplex* method, but before we do, I think it's a good idea to recap some high-school mathematics first. Feel free to skip the next subsection if you're feeling confident with it.

3.1 Background mathematics

An inequality relation is just like a normal equation, except the equals sign is replaced by either $<$, $>$, $\leq$ or $\geq$. When you solve an inequality, you generally want to get all of the similar terms on one side (e.g. move all the variable terms over to one side, and all of the constants to another side). This *mostly* works like a normal equation; you can add and subtract from both sides just like normal, however, if you want to divide or multiply **by a negative quantity**, then you need to **reverse the equality**, for example:

$$\begin{aligned} -2x &> -2 \\ 2x &< 2 \\ x &< 1 \end{aligned}$$

This equality is satisfied whenever x is less than 1. If we have two terms in the equality (something similar to $ax + by \geq c$), it becomes slightly harder to solve. To solve this we can:

- Plot a graph of the line $ax + by = c$.
- Pick a test point (x, y) not on the line, and plot it on the graph.
- If the point (x, y) satisfies the inequality, then shade the opposite side of the line to which the point is on, otherwise, shade the same side.

For example, given $3x + 4y \leq 6$, and choosing the point $(-2, 1)$ we end up with what's in Figure 5.

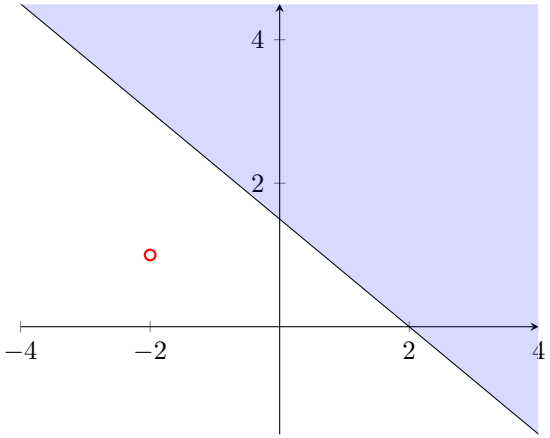


Figure 5: A graph of $3x + 4y \leq 6$, with all the valid values shaded blue.

If we have multiple inequalities, we want to *find the region where all of them are satisfied.* This involves plotting each line, and shading the regions where they're not satisfied, which means the blank bit is the bit we want.

If we have the equations $3x + 4y \leq 6$, $2y - x \leq 2$ and $x \geq 0$, we will get something like in Figure 6.

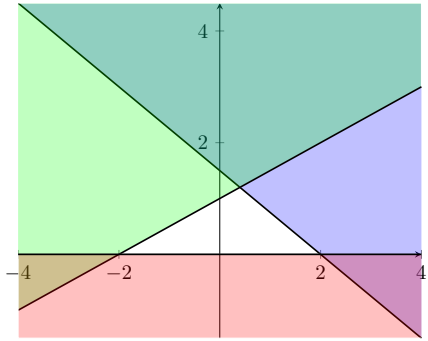


Figure 6: A graph plotting $3x + 4y \leq 6$, $2y - x \leq 2$ and $x \geq 0$, where the points not satisfying the inequalities are shaded in blue, green and red respectively. The clear region satisfies all points.

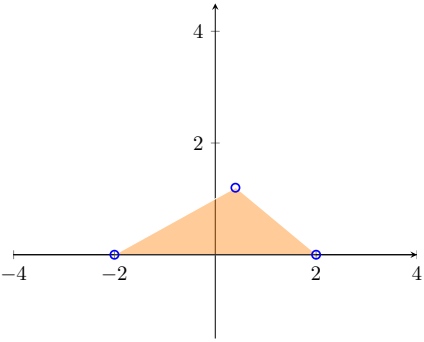


Figure 7: The valid region in the three inequalities from Figure 6. The corner points are $(2, 0), (-2, 0)$ and $(\frac{2}{5}, \frac{6}{5})$.

The corner points are especially important to us, since that's often where the useful numbers are (we'll see more of this later). In order to find the corner points, we solve the two lines that form them. Solving the following equations gives the points in Figure 7.

- $x = 0,\ 3x + 4y = 6$
- $x = 0,\ 2y - x = 2$
- $2y - x = 2,\ 3x + 4y = 6$

So, now we know how to solve these types of problems, lets look at how to decompose a problem statement into a set of equations.

You can buy wood in 11 meter lengths, or 6 meter lengths. A 11 meter piece of wood can be cut into two lengths of 5 meters, and one length of 1 meter. A 6 meter piece of wood is cut into one length of 5 meter, and one length of 1 meter.

If we have room for twenty 1 meter pieces and thirty 5 meter pieces, how many 11, and how many 6 meter lengths should we buy? We cannot go over that amount of pieces, but we can settle for less than that amount.

To answer this, we can make a table to describe it:

Bought Length	Required length	
	5 meter	1 meter
11 meter (x) 6 meter (y)	2 per board 1 per board	1 per board 1 per board
Amount needed	30	20

This decomposes into the following equations:

- $2x + y \leq 30$
- $x + y \leq 20$
- $x \geq 0$, $y \geq 0$

Note:

Notice how our shapes always have a convex hull...

Plotting this on a graph gives us what's in Figure 8, where

we can see a feasible region between $(0, 20), (10, 10), (15, 0)$ and $(0, 0)$. The three corner points are all of the non-zero points (zero eleven meter and zero six meter pieces is an invalid solution), and anywhere on the outer edge of the region is a solution.

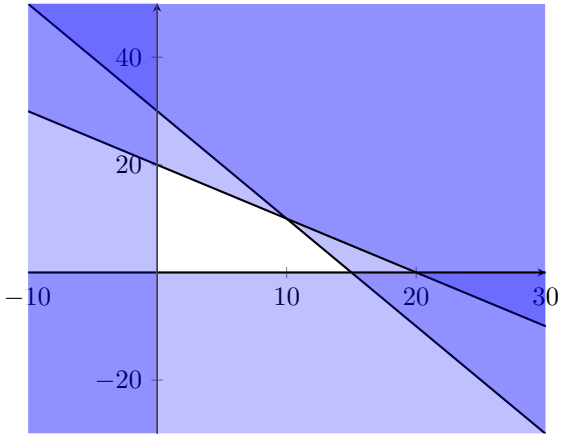


Figure 8: A graph of $2x + y = 30$, $x + y = 20$, $x > 0$, $y > 0$.

3.2 The maximum problem

Sometimes, we want to maximise some particular variable. For example, companies want to maximise their profit. If this is the case, then we can come up with a function for the profit, something like:

$$P = 7x + 3y$$

Where x is the number of 5 meter boards, and y is the number of 1 meter boards. We can factor this equation into our solution, to find the most profitable wood to buy.

If you think about it, the further from the origin we get, the more profit (in this instance) we get. At $(0, 0)$ we have no profit, since we have done nothing, but at $(5, 0)$ we have 35 profit since we have bought five, 5 meter length boards. Since our lines always intersect to produce a region with a convex shape, the points furthest from the origin are the corner points. To figure out how much profit we can make, we just need to find the maximum of these points:

Corner point	**5 meter boards**	**1 meter boards**	**Profit**
$(0, 20)$	20	20	200
$(10, 10)$	30	20	270
$(15, 0)$	30	15	255

Therefore the most profitable choice is to buy 10 11 meter length boards and 10 6 meter length boards.

3.2.1 Matrix form

We can put problems into another form; matrix form, and solve them that way. Here is the example from the Linear programming wikipedia article.

Maximize: $S_1 \cdot x_1 + S_2 \cdot x_2$ (maximize the revenue—revenue is the "objective function")

Subject to:

$$x_1 + x_2 \leq L \quad \text{(limit on total area)}$$
$$F_1 \cdot x_1 + F_2 \cdot x_2 \leq F \quad \text{(limit on fertilizer)}$$
$$P_1 \cdot x_1 + P_2 \cdot x_2 \leq P \quad \text{(limit on insecticide)}$$
$$x_1 \geq 0, x_2 \geq 0 \quad \text{(cannot plant a negative area).}$$

Which in matrix form becomes:

$$\text{maximize} \begin{bmatrix} S_1 & S_2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$
$$\text{subject to} \begin{bmatrix} 1 & 1 \\ F_1 & F_2 \\ P_1 & P_2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \leq \begin{bmatrix} L \\ F \\ P \end{bmatrix}, \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \geq \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

Figure 9: Putting a LP problem into matrix form.

3.3 The Simplex Method

If we have more than two variables, then all of a sudden, working out the optimum on a graph becomes rather hard. Drawing 3d graphs is hard, and visualising n-dimensional graphs quickly becomes infeasible.

The simplex method automates finding the optimum value of any number of variables. This is how it works:

- Draw the feasible region in n dimensional space
- Select a corner of the feasible region
- Choose an edge that is connected to this point that will increase the objective function.
- Go to the next corner on that edge
- Keep doing steps 3 and 4 until you find an optimal solution.

That's pretty far removed from how we see the simplex algorithm working, since we find the optimum by using a tableaux. I've broken down the algorithm into multiple steps with a running example here:

Step 0 The zero'th step, is to examine the input data we're given. This will consist of a function f to be maximised and a set of inequalities that constrain the values we can give to f.

$$
\begin{aligned}
f &= 10x + 12y + 12z \\
3 &\geq 5x + 2y + z \\
2 &\geq x + 2y + 4z
\end{aligned}
$$

N.b. if the inequalities aren't of the form $CONST \geq EXPR$, then rearrange them so that they are.

Step 1 Now we turn each inequality we were given into an equation (replace the inequality sign with an equals basically), and introduce a new variable for each (called a *slack variable*). Also, rearrange f so its equal to zero:

$$
\begin{aligned}
3 &= \quad 5x + 2y + z + m_1 \\
2 &= \quad x + 2y + z + m_2 \\
0 &= -10x - 12y - 12z + f
\end{aligned}
$$

Step 2 Now our problem is in the correct form, we can whack it into a tableaux:

	x	y	z	m_1	m_2	m_3	f	constants
m_1	5	2	1	1	0	0	0	3
m_2	1	2	4	0	1	0	0	2
f	-10	-12	-12	0	0	0	1	0

It should be quite obvious how we go from the equations in step 1 to the tableaux in here. The only stuff you need to *remember* about formatting the tableaux, is that the slack variables go along the left side, f is the bottom row, and to put the constants in the last column.

The values in the bottom row are called **indicators**.

Step 3 We need to find the **pivot cell** in the table. The pivot column is easy to find, its the one with the most negative indicator (in our case, the second or third column, since the indicators there are -12, we're picking column two for no particular reason). Finding the row is a bit harder, you need to divide the constant column by the pivot column and pick the one with the lowest value:

> Note:
>
> **Only choose positive ratios**, and remember that $?/0 = \infty$. Otherwise it goes belly up and it takes you over a day to figure out why (this may or may not have happened to me...).

	x	y	z	m_1	m_2	m_3	f	constants	ratio
m_1	5	2	1	1	0	0	0	3	3/2
m_2	1	2	4	0	1	0	0	2	2/2
f	-10	-12	-12	0	0	0	1	0	N/A

Step 4 Now we do the pivot about the pivot cell:

Step 4a Divide each cell in the pivot's row by the pivot (which makes the pivot equal to 1!).

	x	y	z	m_1	m_2	m_3	f	constants
m_1	5	2	1	1	0	0	0	3
m_2	$\frac{1}{2}$	1	2	0	$\frac{1}{2}$	0	0	1
f	-10	-12	-12	0	0	0	1	0

Step 4b Subtract the pivot row from each other row so that the whole pivot column becomes 0. For example here, we need to make $R1 = R1 - 2(R2), R3 = R3 + 12(R2)$ to make the cell below the pivot 0.

	x	y	z	m_1	m_2	m_3	f	constants
m_1	4	0	-3	1	-1	0	0	1
m_2	$\frac{1}{2}$	1	2	0	$\frac{1}{2}$	0	0	1
f	-4	0	12	0	6	0	1	12

Step 4c Replace the letter on the pivot row with the letter on the pivot column:

	x	y	z	m_1	m_2	m_3	f	constants
m_1	4	0	-3	1	-1	0	0	1
y	$\frac{1}{2}$	1	2	0	$\frac{1}{2}$	0	0	1
f	-4	0	12	0	6	0	1	12

Step 5 Now we have found another solution, to see what it is, read off the letters on the far left and correspond them to the constants on the far right:

$$x = 0, y = 1, z = 0, m_1 = 1, m_2 = 0, f = 12$$

If we sub that in, then we can see that if satisfies the formulae from step 1:

$$\begin{aligned} 3 &= 0 + 2 + 0 + 1 \\ 2 &= 0 + 2 + 0 + 0 \\ 0 &= -0 - 12 - 0 + 12 \end{aligned}$$

Step 6 If there are no negative indicators in the table (bottom row of the tableaux), then we've finished, and our solution is the optimal one. If there are, then we need to go back to step 3 and repeat until there aren't any negative indicators. Since we have a negative indicator in the first column, then we should loop back and do another iteration.

If we run steps 3-6 again (which we should do because there is a negative indicator), we get the optimum, which is:

	x	y	z	m_1	m_2	m_3	f	constants
x	1	0	$\frac{-3}{4}$	$\frac{1}{4}$	$\frac{-1}{4}$	0	0	$\frac{1}{4}$
y	0	1	$\frac{19}{8}$	$\frac{-1}{8}$	$\frac{5}{8}$	0	0	$\frac{7}{8}$
f	0	0	9	1	5	0	0	13

So:

$$f = 13,\ x = 0.25,\ y = 0.825,\ z = 0$$

You can use this tool to check solutions to simplex problems: `http://www.zweigmedia.com/RealWorld/simplex.html`.

3.3.1 Duality

Simplex is great if we're trying to maximise a function, but what if we want to minimise it? Duality does this. Basically, you give it a function and inequalities, it mutates them so they'll fit into simplex, and then you run simplex as normal. So, to apply the duality theorem, you need:

- A function you want to minimize; $f = ax + by + \ldots$
- A set of inequalities of the form '$EXPR \geq CONST$'

Since simplex tries to maximise f, which is the opposite of what we want to do, and our inequalities are the wrong way around, we need to the duality bit now:

First, like the first part of simplex, we put the coefficients in a table. However, this table has a special name; the *coefficient matrix*. It's called a matrix, not a table because we're going to do a transposition on it (and mathematicians transpose matrices, not tables!).

The input:

$$\begin{aligned} g &= 3r + 4s + t \\ r + 2s + t &\geq 4 \\ r + 2s - t &\geq -2 \\ r + s &\geq -1 \\ r \geq 0,\ s \geq 0,\ t &\geq 0 \end{aligned}$$

Would result in a matrix like this:

	r	s	t	constants
	1	2	1	4
	1	2	-1	-2
	1	1	0	-1
g	3	4	1	

So, we put the inequalities in first, then put the minimising function in the last column. Now what we do, is to flip the matrix along its top-left to bottom- right diagonal:

	x	y	z	constants
	1	1	1	3
	2	2	1	4
	1	-1	0	1
f	4	-2	-1	

After we've done that, we get the following equation and inequalities:

$$\begin{aligned} f &= 4x - 2y - z \\ x + y + z &\leq 3 \\ 2x + 2y + z &\leq 4 \\ x - y &\leq 1 \\ x \geq 0, y \geq 0,\ z &\geq 0 \end{aligned}$$

If you do simplex on them (you should try it for practice!), you get:

	x	y	z	m_1	m_2	m_3	f	constants	
w_1	0	0	$\frac{1}{2}$	1	$\frac{-1}{2}$	0	0	1	
y	0	1	$\frac{1}{4}$	0	$\frac{1}{4}$	$\frac{-1}{2}$	0	$\frac{1}{2}$	
x	1	0	$\frac{1}{4}$	0	$\frac{1}{4}$	$\frac{1}{2}$	0	$\frac{3}{2}$	
f	0	0	$\frac{3}{2}$	0	$\frac{1}{2}$	3	1	5	g
				r	s	t			

Therefore the minimum value of g is 5, with the inputs as $r = 0$, $s = \frac{1}{2}$, $t = 3$.

The course reading, sums up the duality theorem like so (in your head, replace 'dual problem' with 'simplex problem'):

> The minimum value of the objective function $g = ar + bs + ct$ in a primal minimization problem is the maximum value of the objective function $f = ax + by + cz$ in the dual problem. Values of r, s and t that minimize g appear in the lat row of the terminal tableau of the dual problem under the slack variables w_1, w_2 and w_3.

3.3.2 Complexity

The Simplex algorithm has an exponential complexity, since in the worst case, adding a new inequality results in having twice as many vertices to traverse as before.

An interesting aspect of the Simplex algorithm, is that although there are polynomial alternatives, in practice it beats all the other algorithms! Is this because we only ever run it on the inputs which are solvable in polynomial time?

4 Integer Programming

Integer programming is just like linear programming, except we are only allowed solutions inside $\mathbb{N}$ (the integers) instead of $\mathbb{R}$ (real numbers). Even though we've narrowed our search space, the new problem turns out to be harder (remember, even though our solutions will be finite and countable, there are still a lot (very exponential) of them)!

We define *mixed integer programming* to be linear programming, where some variables are integers, but some are real numbers. Likewise, *pure integer programming* is when all the variables are integers.

At first, I thought integer programming was a bit pointless; couldn't you just find the linear programming solution, and then round it down to the nearest integer? Well, yes, but there are two issues; firstly, it might not be the best solution, and secondly, you can do cool stuff with integer programming.

For example, what if we wanted to implement multiple choice? Say I want to buy either car A, B, or C, and they all have safety ratings, prices, etc. I obviously can't buy 0.375A, 0.005B and 0.62C; I have to make a decision.

We can encapsulate this in integer programming by having a constraint like this:

$$A + B + C \leq 1$$

Furthermore, we can restrict variables to be binary too, which, if all our variables are binary, actually turns out to

be our old friend the knapsack problem!

In fact, any pure integer programming problem can be expressed as a 0-1 problem, since we can express any decimal number as a binary number.

> Note:
>
> We can also turn scheduling problems such as timetabling or the travelling salesman problem into integer programming problems.

You may have worked out by this (if you remember the lectures from the second half of this course, or if you have read later sections of this document), that if we can turn the 0-1 knapsack problem into an integer programming problem, then integer programming must be in the same complexity class as the knapsack problem; meaning that integer programming is NP hard!

4.1 Solving Integer Programming Problems

It turns out, that *Branch and Bound* is a good way to solve these kinds of problems. If you remember the second year algorithms course, then you might be having a quiet little groan right now. However, I have good news! There's a really good explanation of B&B in the course reading (the Chapter 9 handout you should have picked up in the lecture).

Though it's good, and thorough, it's also a bit boring and dense. Here's my shorter version, broken up into steps:

1. So, we have an optimisation problem just like normal.

We can graph it, and it'll look something like this:

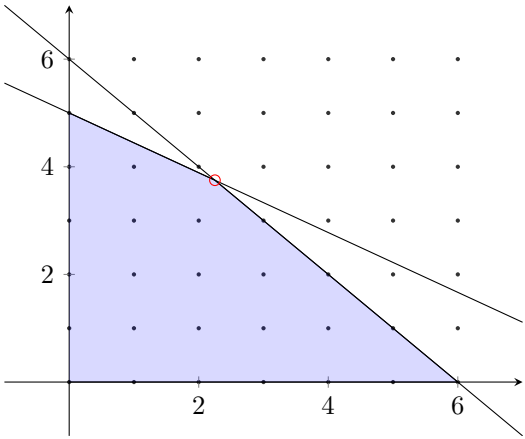


Figure 10: A graph of $y = 6 - x$, $y = \frac{45-5x}{9}$ and $x \geq 0, y \geq 0$. The linear optimum is shown as the red circle.

This of course is a graph of two constraints relating to a maximisation problem. The problem and constraints are here:

$$\begin{aligned} f &= 5x + 8y \\ x + y &\leq 6 \\ 5x + 9y &\leq 45 \end{aligned}$$

If we use Simplex to solve this like a linear programming problem, then we get the optimum as $f(\frac{9}{4}, \frac{15}{4}) = 41.25$, which is obviously not a integral solution (since it has fractions).

One thing to do might be to round off the numbers, trying $(2, 4)$, which is infeasible due to the constraints

and $2, 3$ which gives $f = 34$. However, while the latter solution is feasible, it is *not* the optimum.

Note, in an example like this, you could enumerate all the integral points and find which one is best. This was mentioned in the lectures and is a lot easier than doing Branch and Bound!

2. If we don't have time to enumerate the points, then we need to do branch and bound. We make use of the following observations:

   - The integral solution isn't the same as the linear one for this problem, since the linear solution is not integral.
   - The linear solution provides an upper bound on the integral solution.

   In order to progress, you split the feasible region into half, along where either the x or y axis is equal to the linear optimum. We could split at $x = \frac{9}{4}$ or $y = \frac{15}{4}$, and will do the latter. Since our solutions are integral, we can discard the area between $3 < y < 4$:

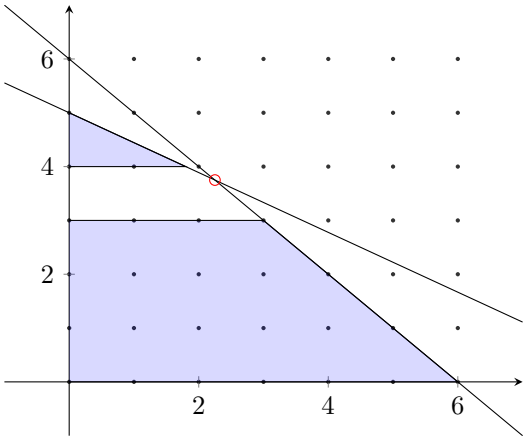


Figure 11: Now, we're showing the two feasible regions after we've split along $y = \frac{15}{4}$

Now, we have two problems; finding the integral solution to the upper region, and the lower region. Then we can simply pick the maximum!

3. To find the integral solution to each sub-region, simply add a new constraint for each; for the top one, we add $y \geq 4$, and for the bottom one, we add $y \leq 3$. Then we recurse to step 1 on both problems.

   However, because I'm nice, I'm going to do the recursion for you here:

   Top recursion

   If we do this, we find that the linear solution for the top problem is $(\frac{9}{5}, 4)$, where $f = 41$. This isn't an integral, but we have gained important information; we can split the top area into two:

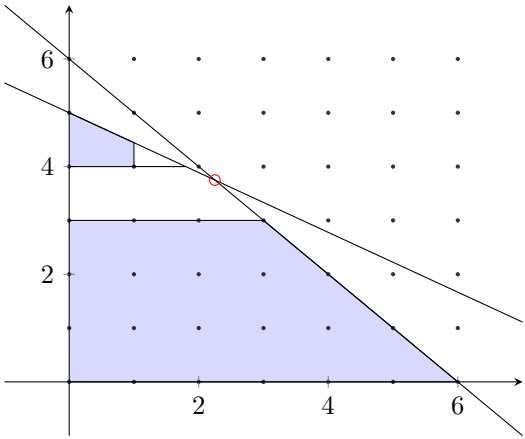


Figure 12: I've split the top area into two new areas where $x \leq 1$ and $x \geq 2$. Ohh wait, there aren't any integral points inside that last one, so it's not shaded :)

If we recurse again, we will find that the top part *still* doesn't have an integral solution, but another level of recursion gives us two integral solutions (this is obvious if you think about it, since we can't recurse any more times due to the number of possible integer points inside the region); $f(1,4) = 37$, and $f(0,5) = 40$.

Bottom recursion

If we recurse on the bottom region, and to simplex on it, we get an integral solution straight away; $f(3,3) = 39$. Since we know that the linear solution is an upper bound on the integral solution (hence we won't find a larger answer inside the rest of the lower region), then we can stop here.

Since the top region's result $f(0,5) = 40$ was better than the bottom region's one $f(3,3) = 39$, and those regions

had inside them all the feasible points, then we have found our solution, $f(0, 5) = 40$.

If you understood all that, then I recommend you read Chapter 9 of the course handout (from section 9.4 on page 287 to the end of section 9.5 on page 292), since it visualises the above as a tree as well. Besides, you've already understood the concept, so it'll be light reading!

5 Flow Matching

Like integer programming, flow matching is a problem that initially seems quite niche, but actually has a lot of applications. In essence, given a weighted graph, a 'start' node and an 'end' node, we want to compute the *maximum flow* through the graph from the source to the target. Imagine each edge is a pipe between nodes, and liquid flows through the pipe. The weight of each edge indicates how much liquid can flow through the graph, and a flow defines how much liquid is flowing through each edge.

While flow matching, at first glance, seems only applicable to things like oil pipelines and plumbing, we can actually map lots of problems onto graphs and weighted edges.

5.1 The stable marriage problem

Given n boys and n girls, and a list for each boy and each girl of who they would marry (this is binary; a yes/no thing), we

want to come up with a $1-1$ pairing between the boys and the girls that produces the maximum number of couples.

If we define that mathematically, then we let $G = (V, W, E)$, where G is a bipartite graph

> Note:
>
> A bipartite graph is one who's nodes can be divided into two disjoint sets, such that every edge in one of the sets connects to one in the other.

, V is the set of boys, W is the set of girls and E is the set of edges. An edge from v to w indicates that v and w both would marry each other.

$E' \subset E$ such that for all $v \in V$, there is at most one $w \in W$, and for all $w \in W$, there is at most one $v \in V$. I.e, E' is a matching between the boys and girls, which is a subset of their preferences for each other.

If every node has one edge (each item in V and W is incident to some $e \in E'$), then the matching is perfect.

We can define the stable marriage problem to return `yes` if the input graph has a perfect matching, and `no` otherwise.

A naive way of solving the problem is to iterate over each possible matching and see if it is perfect:

```
boolean naiveSoln(Set<Node> boys, Set<Node>
  if (boys.isEmpty()) {
    return girls.isEmpty();
  } else {
    Node randomBoy = boys.getRandomElement()
```

```
        for (Edge e : preferences) {
          if (e.from.equals(randomBoy)) {
            if (naiveSoln(
                  boys.remove(e.from),
                  girls.remove(e.to),
                  preferences.remove(e))) {
              return true;
            }
          }
          return false;
        }
      }
    }
```

However, this is exponential in the size of V, since it might try every combination of boy and girl, which is obviously exponential.

5.2 Flow networks and flow matching

A *flow network* is a quintuple, as shown:

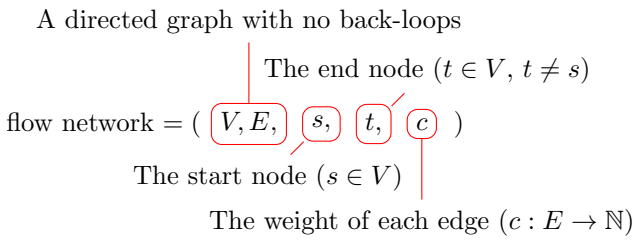


Figure 13: Dissecting the definition of a flow network.

Back-loops are a pair of edges that connect two nodes such

that $[(u, v), (v, u)]$

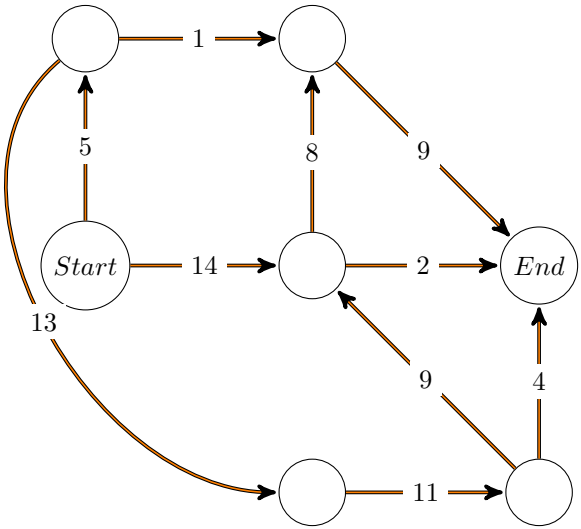


Figure 14: An example of a flow network. We assume that the start and edge nodes have no incoming and outgoing edges respectively.

A **flow** over a flow network is a function $f : E \to \mathbb{R}^+$ that has the following properties:

- The net flow out of a node is zero.
- No edge exceeds its capacity ($f(u, v) \leq c(u, v)$).
- The *value* of f is the net flow out of the start node, or the net flow into the end node.

Our problem is that given a flow network N, we want to compute a flow for N that has the maximum possible value (an optimal flow). If there is more than one possible optimum flow, we don't mind which one is given.

In order to do this, we construct another graph (an *auxiliary directed graph*) from the one we're given. We construct the new graph by including all the nodes, and following this rule for the edges; *include an edge with the value* $c(u, v) - f(u, v)$ *for all* $u, v \in V$.

This gives you the potential flow along each route. If an edge from u to v has a maximum capacity of 6, but is carrying 2, then in the auxiliary graph, there should be an edge (u, v) with a weight of 2, and another from (v, u) with a weight 4 (since the flow can decrease by that much).

5.2.1 The min-cut, max flow idea

Remember, our end goal here is to get the maximum possible flow. The reason we created an auxiliary graph in the first place, was because if there is a route from the start node (s) to the end node (t) in it, then the flow is not optimal.

Using this idea, we can make an algorithm to find the maximum flow:

> Note:
>
> I've made up some extra HashMap methods; increment and decrement. They assume that if the key isn't already associated with a value, the value was 0, and then they increment or decrement the value.

```
1  Map<Edge, Integer> maximumFlow(List<Node> no
2      Node start, Node end, Map<Edge, Integer>
3    // Init the flow to all zeros
4    Map<Edge, Integer> maxFlow = new HashMap<>
```

```
5       for (Edge e : edges) maxFlow.put(e, 0);
6       // Iterate until we have a maximum flow
7       while (true) {
8         // Create the aux graph
9         List<Edge> auxFlow = createAuxFlow(nodes
10        // Find the (possibly empty) path from th
11        List<Edge> path = getPath(auxFlow, start
12        // If there is no path, then we're finish
13        if (path.isEmpty()) break;
14        else {
15          for (Edge e : path) {
16            // Increment each edge in the flow
17            if (edges.contains(e)) maxFlow.increm
18            // If the edge isn't in the flow, dec
19            // the opposite way
20            else maxFlow.decrement(e.reverse());
21          }
22        }
23      }
24      return maxFlow;
25    }
```

If the flow network has integer capacities, then the optimal flow will have integral values! This may be obvious in the above code, since the types are integers, and we only ever increment or decrement.

We can find a path from any two nodes in linear time ($|V| + |E|$), as we do on `Line 11`. If we define the constant C to be the highest edge capacity, then the maximum flow is nC, where n is the number of nodes connected to the start node, which could be $|V|$.

Therefore, the runtime of `maximumFlow` is $O(n(|V|+|E|)C) = O(n(n+m)C)$

5.2.2 Costs in flow networks

What if we assigned every edge an associated cost that is charged per unit of flow? We can add this to the definition of a flow network:

$$\text{flow network} = (\ V, E, s, t, c, \boxed{\gamma}\ )$$

The price per unit flow of each edge ($\gamma : E \to \mathbb{N}$)

We can calculate the total cost of the flow to be:

$$\sum_{e \in E} f(e) \times \gamma(e)$$

What if we want to find the maximum flow with the minimum cost? In fact, it is possible to use the above `maximumFlow` algorithm to solve this, with just a few small modifications; simply have the weight function c take into account γ:

$$c(e) = \begin{cases} \gamma(e) & \text{if } (u,v) \text{is an edge in the aux graph, and in } E \\ -\gamma(e) & \text{if } (u,v) \text{is an edge in the aux graph, and is not} \end{cases}$$

The only change we need to make on our flow finding algorithm is to find the path from the start node to the end node using a minimal length:

```
1   Map<Edge, Integer> maximumFlow(List<Node> no
2       Node start, Node end, Map<Edge, Integer>
3       Map<Edge, Integer> cost) {
4     // Init the flow to all zeros
5     Map<Edge, Integer> maxFlow = new HashMap<>
6     for (Edge e : edges) maxFlow.put(e, 0);
7     // Iterate until we have a maximum flow
8     while (true) {
9       // Create the aux graph
10      List<Edge> auxFlow = createAuxFlow(nodes
11      // Find the (possibly empty) path from th
12      // smallest cost
13      List<Edge> path = getPath(auxFlow, start
14      // If there is no path, then we're finish
15      if (path.isEmpty()) break;
16      else {
17        for (Edge e : path) {
18          // Increment each edge in the flow
19          if (edges.contains(e)) maxFlow.increm
20          // If the edge isn't in the flow, dec
21          // the opposite way
22          else maxFlow.decrement(e.reverse());
23        }
24      }
25    }
26    return maxFlow;
27  }
```

This is the *Busacker-Gowen algorithm.*

5.3 Stable matching as a flow matching algorithm

We can transform a matching problem into a flow matching problem as shown in Figure 15.

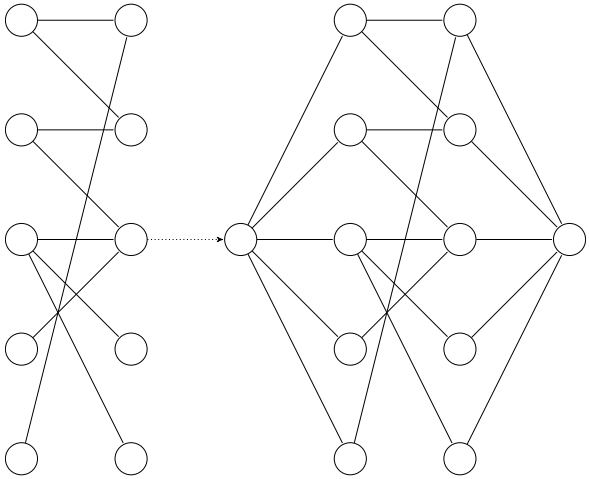


Figure 15: Here, a stable marriage problem (on the left, an edge indicates that both the boy and girl like each other) is mapped to a flow matching problem on the right.

We assume all the edge weights are 1, and a perfect matching will be found iff there is a flow from source to sink (the leftmost node to the rightmost node) with a value n, where n is the number of boys or girls. This runs on $O(n(n+m))$ time!

6 Turing Machines and Complexity Measures

A Turing machine is a theoretical device that implements a model of computation. It operates on a tape of infinite length, which is divided into cells, and can read and write symbols from/to each cell when the *head* of the machine is positioned over the cell. The Turing machine also has internal state, which is finite and determines what actions it takes (reading, writing, moving the tape etc).

In this course, we will be thinking about multi-tape Turing machines. Here, the there are many tapes, numbered $1 \ldots K$. Tape 1 is the input tape and tape K is the output tape. The tapes inbetween the input and output tapes are the work tapes. Any algorithm can be expressed as a multi-tape Turing machine

6.1 Formally defining a Turing Machine

A TM (Turing Machine) can be define as a quintuple, as shown in Figure 16.

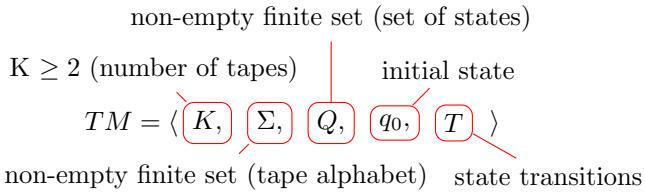


Figure 16: Part of a multi-tape Turing machine definition. We must also define the word *symbol*, which is a mark occupying one cell on the tape. The alphabet that a symbol can be from is $\Sigma \cup \{\text{␣}, \triangleright\}$, where ␣ is a blank cell, and $\triangleright$ is the start symbol (signifying the left edge of the tape).

Up to now, we've defined a TM that is perfectly formed... but does nothing. We need to define how it moves between states in Q. To do that, we define a transition, as in Figure 17.

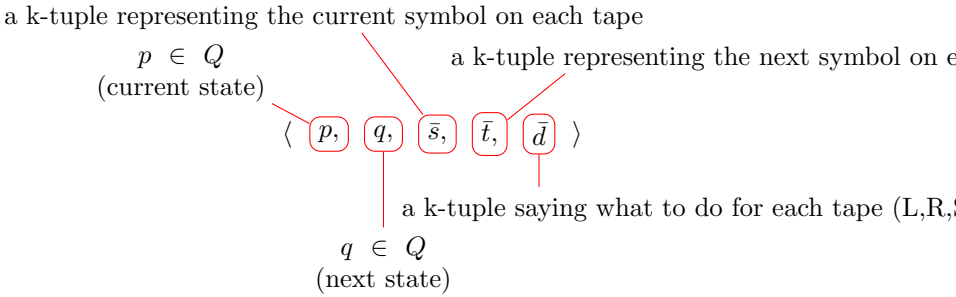


Figure 17: A transition of a Turing Machine

Informally, this transition applies only if the current state is p, and the squares on the tapes confirm to $\bar{s}$, then set the new state to q and write the symbols from $\bar{t}$ to the tapes and move the heads as directed by $\bar{d}$ (Left, Right or Stay).

Now we've nearly finished defining a multi-tape TM, the final points are:

- Since the TM is deterministic, there can be *at most* one transition for every (*state*, $\overline{\textit{current symbols}}$) tuple.
- The tape never moves left past the $\triangleright$ symbol.
- Tape 1 is read only and tape K is write only.

6.1.1 Acceptance, computability and recognisability

We say that a TM M *accepts* an input x if it halts on x. The language *recognised* by M is the set of strings that is accepted by M.

> Note:
>
> Remember, when we say 'Turing Machine', we could mean either a deterministic TM or a non-deterministic one (unless we explicitly said which one we meant of course!).

If a language is recognised by a nondeterministic Turing Machine M, then there is also a deterministic Turing Machine M' that will recognise the same language.

A language is *computable* (aka decidable), if there is a (deterministic) TM that will accept every string in the language, and reject every string not in the language.

Up to now, our definitions of recognisable and computable are pretty similar (if not the same). The difference is, that if an input string **isn't** in a computable language, then the TM will halt (and output 'no'), whereas if the input string isn't in a recognisable language, then the TM *may* halt and

output 'no', or it could just carry on computing for ever. All a Turing Machine does with a recognisable language, is recognise if a string is in the language, it makes no guarantees about string not in the language!

6.1.2 Nailing that TM definition

If you want your Turing Machines very well defined, then this how you do it (alternately, read Chapter 3 of Sipser).

A run of a TM is terminating if it is finite. If the input to a deterministic TM M is x, and the run is finite, we write $M \downarrow x$ otherwise, if the run is infinite, we write $M \uparrow x$.

Let M be a deterministic TM over the alphabet Σ, and its input be $x \in \Sigma^*$ (i.e. the input is zero or more symbols from the alphabet). If $M \downarrow x$ then the output tape of M will contain some string $y \in \Sigma^*$ once M has finished computing. The cool thing, is that we can treat M like a function, lets call it f_M, that operates over $\Sigma^* \to \Sigma^*$:

$$f_M(x) = \begin{cases} y & if M \downarrow x \\ \text{undefined} & if M \uparrow x \end{cases}$$

In this case, M computes the function f_M.

6.1.3 The Universal Machine

If you're that way inclined, you can do lots of cool things with Turing Machines. One such cool thing, is that every TM

is a finite thing (it's just a quintuple as shown in Figure 16), which means we can come up with a way of encoding any Turing Machine into a string over the alphabet Σ and use it as input to another Turing Machine!

If we can do this, then we can construct a **Universal Machine** U. This machine takes two 'arguments', another TM M (encoded of course) and some arbitrary input x. If $M \downarrow x$, then U finishes with y on its output tape. If $M \uparrow x$, then U will never terminate. The input is coded as $M; x$, where ';' is a separator between the Turing Machine we're 'simulating' and the input we'll give it.

6.1.4 Enumerators

Enumerators aren't in the course slides, but they do appear in the reading. Skip this section if you're doing last minute cramming.

An Enumerator is a Turing Machine with an attached printer. Now, I know what you're thinking; *I'm a computer scientist, I hate printers. Why would anybody combine a complicated enough idea like a TM with a printer?!* Enumerators have an important theoretical use though, and don't require us to install drivers like we do for printers, and they certainly don't need to work over WiFi (so this section should be a piece of cake really).

The TM inside an enumerator is able to send a string to the printer whenever it wants to print out a string. If the TM does not halt, then the printer may print an infinite list of strings. The language that is *enumerated* by the

enumerator, is the collection of strings that it eventually prints out. Since there are no rules about repetitions or ordering in the output list of the printer, you might want to think about the enumerated language as a set.

So, as I said before, enumerators *are* useful. Namely a language is Turing recognisable if and only if, there is some enumerator that enumerates it.

Its easy to show that a TM can recognise a language that is enumerated by an enumerator, just build a TM that does this:

1. We have an input word w
2. Run the enumerator E:
   (a) Every time E outputs a new string, compare it to the word w
   (b) If they are equal, then accept the word.

Note that there is the potential for this TM to run for an infinite length of time, if the enumerator generates an infinite list, and w is not in the language it enumerates.

We can also make an enumerator enumerate any language that is recognised by a Turing Machine M:

1. Ignore any input
2. For $i = 1$ to $i = \infty$
   (a) Run the TM M for i steps for all possible strings of length i in the language (Σ^i)
   (b) If any computations accept, then print out the cor-

responding string.

This enumerator will never halt (since it loops infinitely many times over an infinitely large set of input strings), but eventually, it will print out all of the words in the language (even though there will be *a lot* of duplicates!).

6.2 The Halting problem

The Halting problem asks:

> Given a Turing Machine M and a string for its input x, return:
>
> **Yes** if $M \downarrow x$ (M decides x)
>
> **No** otherwise, where M would compute forever

Turing managed to prove that there is no Turing Machine that will decide the Halting problem. The proof is fairly simple, but it takes a while to get your head around it when its written in a mathematical form. Luckily, I found a poem, written in the style of Dr. Seuss, that provides a better textual explanation than I ever could; see Figure 18.

As nice as it is, I think it's unlikely that the poem would be accepted as an answer in an exam, so we'd better learn the formal definition of the Halting problem too.

Suppose we have a TM P that can determine if another TM can halt. We're going to make another TM Q that is defined so:

1. Duplicate the input x TM we're given.

2. Run P on the duplicated input $(x; x)$.

   $P(x; x) = Y$, then Loop

   $P(x; x) = N$, then Halt

Now, if we give Q the input Q, then the embedded P will receive $(Q; Q)$, meaning that if:

$$Q \downarrow Q \implies Q \uparrow Q$$
$$Q \uparrow Q \implies Q \downarrow Q$$

6.3 Complexity

You already know that there are two types of Turing Machine; deterministic and non-deterministic TM's, but now its time to think about the resources that Turing Machines consume. We don't think of TM resources as commodities such as metal, electricity etc, since TM's are abstract, theoretical devices. Instead the two commodities we're interested in with Turing Machines are **time** and **space**.

If we have a TM M, and let $g : \mathbb{N} \to \mathbb{N}$. M runs in time g if for any finite input x, M takes at most $g(|x|)$ steps. We say that M runs in space g if for any finite x, M takes at most $g(|x|)$ squares on its tapes.

If you think about it, this is nearly a formal way of defining the Big-Oh notation. One of the main problems with this, is that TM's might execute at different speeds (maybe one has faster motors for its tape etc). This means that the algorithms we're designing will have certain runtimes on certain machines. In order to mitigate this, we like to define

sets of functions for the Big-Oh notation; M runs in time G if for some $g \in G$, M runs in time g.

This means that the Big-Oh notation can ignore things like constants, and lets us abstract away the details of Turing machines (like exactly how many operations it takes to multiply two numbers).

In fact, possibly the simplest definition of complexity is to let L be a language, and G be a set of functions from $\mathbb{N}$ to $\mathbb{N}$. L is in `TIME/SPACE(G)` if there exists a deterministic TM that decides L in time/space G.

Similarly, if we can find a non-deterministic TM to recognise L, then we say that L is in `NTIME/NSPACE{G}`.

6.3.1 Complexity classes

As you probably know, there are several different complexity classes, as shown in Table 1.

> Note:
>
> Note, you can't have $Time(log(n))$ since reading the input takes n time.

		LogSpace	$Space(log(n))$
PTime	$Time(n)$	PSpace	$Space(n)$
ExpTime	$Time(2^n)$	ExpSpace	$Space(2^n)$
k-ExpTime	$Time(2^{2^{\cdots^2\}^n \leftarrow k \ times}})$	k-ExpSpace	$Space(2^{2^{\cdots^2\}^n \leftarrow k}}$

Table 1: The deterministic complexity classes

To get their non-deterministic counterparts, just stick an 'N' on the front of everything, as in Table 2.

		NLogSpace	$NSpace(l$
NPTime	$NTime(n)$	NPSpace	$NSpace(n$
NExpTime	$NTime(2^n)$	NExpSpace	$NSpace(2$
Nk-ExpTime	$NTime(2^{2^{\ldots^2\}^n \leftarrow k\ times}})$	Nk-ExpSpace	$NSpace(2$

Table 2: The deterministic complexity classes

Obviously, some complexity classes can fit inside each other. Here are some examples:

$$PTime \subseteq ExpTime \subseteq 2 - ExpTime \subseteq \ldots$$

$$LogSpace \subseteq PSpace \subseteq ExpSpace \subseteq \ldots$$

$$NPTime \subseteq NExpTime \subseteq N2 - ExpTime \subseteq \ldots$$

We can also interleave the non-deterministic classes:

$$PTime \subseteq NPTime \subseteq ExpTime \subseteq NExpTime \subseteq \ldots$$

Finally, the Co^* complexity classes are the ones that describe the complementary languages to those described by the others. For example, if we can decide a language L over Σ^* in `NPTime`, then we can decide $\Sigma^* - L$ in `Co-NPTime`. We know that `Time(G) = Co-Time(G)`, and the same for space, but not if `NPTime(G) = Co-NPTime`.

7 Propositional logic

I'm going to assume that you're comfortable with basic propositional logic from other courses. If you did `COMP21111` then you're probably going to find this easy!

Just to jog your memory, here are some PL things you should know:

- A formula is *satisfiable* if there exists an assignment such that the formula evaluates to `T`. E.g. $p \vee q$ is `T` when $p = 1, q = 0$, and therefore is satisfiable. $p \wedge \neg p$ is obviously not satisfiable.
- We can define the problem `Propositional Sat` which will determine whether a given formula is satisfiable or not. Since it is possible to turn any PL formula into one of the 'normal forms', we need only consider formula made completely from negations and disjunctions, since all other formulae can be converted into something like them.
- Since converting a formula into normal form gives us a set of clauses, we need to define the problem `SAT` which when given a set of clauses, will determine whether the whole set of clauses is satisfiable (they must all be satisfiable with the same truth assignment).
- Similarly, `k-SAT` is the problem where each clause has at most k literals.

Note:

Horn satisfiability is actually `PTime-Complete` (see Section 8), since we can convert any deterministic TM into a set of Horn clauses.

- Horn clauses are clauses where there is only 0 or one non-negative literals. In other words, all but at most one literals are negative, e.g. $\neg x \vee \neg y \vee \neg z$ or $\neg x \vee y \vee \neg z$
- Krom clauses have at most two literals, e.g. $x \vee y$.

7.1 DPLL

The Davis-Putnam-Logemann-Loveland algorithm solves `SAT`. Here it is:

```
begin DPLL(clauses)
  if clauses is empty then return Yes
  if clauses contains the empty clause then return
  while clauses contains any unit clause l
    remove l from clauses
    clauses = resolve(, l)
    if clauses is empty then return Yes
    if clauses contains the empty clause then retur
  let l be the first literal of the first clause of
  if DPLL(clauses + l} then return Yes
  if DPLL(clauses + l} then return Yes
  return No

begin resolve(clauses, l)
  for each x in clauses
```

```
    if x contains l, remove x from
    if x contains not l, remove not l from x
```

DPLL tries all the different combinations of truth assignments, and as a result, runs in exponential time. This is close to the best way of solving SAT in practice, but in practice, we don't have non-deterministic Turing machines...

7.2 NPTime SAT

We've shown that SAT is in ExpTime by giving an algorithm for it, but we can also make a non-deterministic algorithm to solve it:

```
begin NdSatTest(clauses)
  If clauses contains FALSE, then return No
  While clauses is not empty:
    Select a literal in clauses, p
    Either:
      Delete every clause containing p
      Delete not p from all the other clauses
    Or:
      Delete every clause containing not p
      Delete p from all the other clauses
    If clauses contains FALSE, then return No
  return Yes
```

This is obviously in NPTime.

7.3 Horn SAT

If we have a set of Horn clauses, how does that affect SAT? Well, we can in fact solve it in $O(n^2)$ time:

```
begin Horn-DPLL(clauses)
  if clauses contains the empty clause then return
  while clauses contains any unit clause l
    remove l from clauses
    clauses = resolve(clauses, l)
    if clauses contains the empty clause then retur
  return Yes

begin resolve(clauses, l)
  for each x in clauses
    if x contains l, remove x from
    if x contains not l, remove not l from x
```

7.4 2-SAT

If we're given Krom clauses, we can also solve that in $O(n^2)$. To do this, you create a graph with $2n$ nodes, one for each literal and one for its negation. For the clauses $(x \vee y), (\neg x \vee z), (y \vee \neg z)$, we create a graph like this:

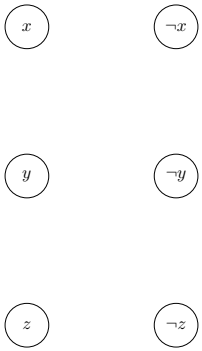


Figure 19: Initial graph for $x, \neg x, y, \neg y, z, \neg z$

For each clause, we insert two edges. If the clause is $(a \vee b)$, we insert $\neg a \to b$ and $\neg b \to a$:

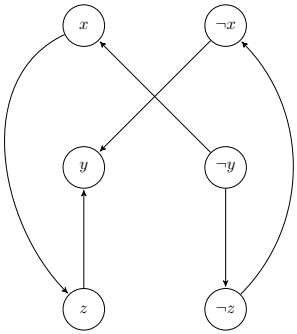


Figure 20: Graph of $x, \neg x, y, \neg y, z, \neg z$, with the appropriate edges.

Then, for each node in the graph, see if there is a path to its inverse. If there is, then the clause is *unsatisfiable*. This runs in $O(n^2)$ time, since finding a path takes $O(n)$ time, and we have to do it for each clause, so we have to do it n times.

8 Reducibility and Hardness

You many already know that we can map some problems onto other problems. In fact, we did this in the previous example with `2-SAT`, where we mapped it onto a graph pathfinding problem.

It turns out that mapping problems onto other problems can be well defined theoretically. if one problem P_1 can be mapped onto P_2, then we say that P_1 is many-one-logspace reducible to P_2.

Formally, let P_1 be over the alphabet Σ_a, and P_2 be over the alphabet Σ_b. P_1 is reducible to P_2 if there exists a function $f : \Sigma_a^* \to \Sigma_b^*$ that is in `SPACE(log n)`, such that $x \in \Sigma_a^*,\ x \in P_1$ iff $f(x) \in P_2$.

In English, this is saying that a problem is reducible to another problem if we can make a function that runs in $log(n)$ space, that converts all instances of the first problem into an equivalent instance of the second problem.

The notation for reducibility is:

> Note:
>
> This is also called a many-one logspace reduction.

$$P_1 \leq_m^{log} P_2$$

Reducibility is helpful for a number of reasons:

- If $P_1 \leq_m^{log} P_2$, then P_2 is at least as hard as P_1.

- Similarly, P_1 is no harder than P_2.
- If anybody comes up with a way of solving P_2 fast, then we can use the same way to solve P_1.

> Note:
>
> This only works for some (most) complexity classes, e.g. `LogSpace`, `NLogSpace`, `PTime`, `NPTime`. Not `Time(n)` or `Time(`n^2`)`.

- **We can use reducibility to show that a problem is inside a complexity class by reducing it to one that you know is in the desired class.**

Reducibility is transitive, so if $P_1 \leq_m^{log} P_2$ and $P_2 \leq_m^{log} P_3$, then $P_1 \leq_m^{log} P_3$. The reason why, is that if:

$$f_1 : \Sigma^*_{P_1} \to \Sigma^*_{P_2}$$
$$f_2 : \Sigma^*_{P_2} \to \Sigma^*_{P_3}$$

We have a Turing Machine M that can compute a function $f_3 = f_2 \cdot f_1$, which essentially is:

$$f_3 : \Sigma^*_{P_1} \to \Sigma^*_{P_3}$$

Lets define M:

```
1  Calculate the first bit of f1(x)
2  Keep a counter to say what bit we've calculat
3  Start a simulation of f2(f1(x)), where the argu
4  If f2 asks to move the read head right:
5    Calculate the next bit of f1(x)
```

```
6       Write it on top of the current bit
7       Update the output bit counter
8     Else, if f_2 asks to move the read head left:
9       Restart the calculation of f_1(x)
10      Continue calculating until we have the desi
11      Write it on top of the current bit
12      Update the bit counter
```

Although M horrifies my programmer-mind at how inefficient it is, I conclude that it does in fact run in logarithmic space (it seems to run in constant space at first, but then you realise that the counter requires $log_2(n)$ bits, and that the functions its emulating, f_1 and f_2, run in log space).

A *many-one polytime* reduction is a weaker notion of reducibility where instead of a function that runs in log space to convert the strings of one problem into the strings of another, we have one that is in `TIME{P}` instead. If this is the case, then the reducibility is mathematically written as:

$$P_1 \leq_m^P P_2$$

Note:

Many-one polynomial reducibility is transitive because any polynomial multiplied by any other polynomial is still a polynomial.

However, *many-one logspace* reducibility trumps its polynomial cousin, since while they're both transitive, the former is

theoretically a bit more useful, and many polytime reducibilities are in fact logspace ones.

A problem $P \in X$ is said to be X-**complete**, when X is a complexity class, when for all problem $Q \in X$, $Q \leq_m^{log} P$. This means that P is at least as hard as all problems X.

If $P \notin X$, then P is not X-complete, but is X-**hard**.

8.1 Cook's theorem

The Cook Levin theorem proves that `SAT` is `NPTime-Complete`. This is important, because certain interesting problems were shown to be `NP-Complete`, and if any of these problems are solved in (deterministic) polynomial time, then all of the problems would be solvable in polynomial time.

The idea behind the Cook Levin theorem, is to turn any Turing Machine M that solves a problem P into a set of clauses that can be solved by `SAT`, and we can do this in deterministic polynomial time. If we can solve `SAT` in better-than `NPTime`, then we can solve P in the same time.

We can also show that `3-SAT` is `NP-Complete`. Obviously `3-SAT` $\in$ `SAT`, but to prove completeness, we need to show that every problem in `SAT` can be converted to a problem in `3-SAT`. We can use the following rules:

- For each input clause of length 1: $\{a\} \rightarrow \{a \vee a \vee a\}$
- For each input clause of length 2: $\{a \vee b\} \rightarrow \{a \vee b \vee a\}$
- For each input clause of length 3, do nothing!

- For each input clause of length $n \geq 3$: $\{a \vee b \vee c \vee d \vee e\} \rightarrow \{a \vee b \vee x\}, \{\neg x \vee c \vee y\}, \{\neg y \vee d \vee e\}$

Hence `3-SAT` is `NP-Complete`. Similarly, we can reduce `3-SAT` to the 0/1 Integer Linear Programming problem and hence (since we know it's in `NP-Time`) prove that ILP is `NP-Complete`.

As was said in Section 4, it is possible to do cool stuff with Integer Programming. In order to convert `3-SAT` clauses to integer programming equations we need to:

- Make an equation for each literal x such that $l_x + l_{\neg x} = 1$, so that every literal is either positive or negative.
- For every clause $\{a \vee b \vee c\}$ in the input, we make an equation with two new variables (u, v) in (unique to the equation): $l_a + l_b + l_c + u + v = 3$. Then if the clause is true (so at least one of its literals is true), then we can set u and v to a value that will make the equation hold.

Note:

N.b., `NP-Complete` is mostly synonymous with `NPTime-Complete`.

If we then solve the ILP programming problem for these equations, then we can find a suitable truth assignment for `3-SAT`, and so we have proved that `ILP(0/1)` is `NPTime-Complete`.

8.2 3-SAT as a graph colouring problem

We can show that *3-colourability* is `NP-Time` hard, because we can convert an instance of `3-SAT` into it. Given a `3-SAT` problem, we compute in log space (since we want a many-one log reduction), a function to create a graph from the satisfiability problem's clauses.

To do this, we create three 'gadgets' which let us represent logical gates in graph form. The gadgets have internal nodes which make them easier to draw, the internal nodes are not often shown in larger sketches, but are when describing the gadgets initially in order to make the properties of the gadget known.

Gadget 1: Equality

We want to make a gadget that will make sure that two nodes have the same colour. Here, a and b are the input and output nodes, while the other two nodes are internal. In any 3 colouring, a and b must be the same colour:

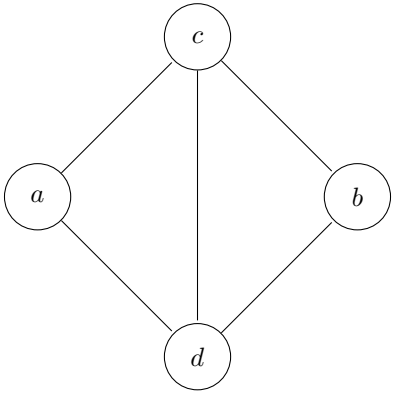


Figure 21: Gadget 1

Gadget 2: XOR Gate

I'm not sure if this is strictly an XOR gate, but that's what it reminds me of. a is either the same colour as b or c, while d is an internal node.

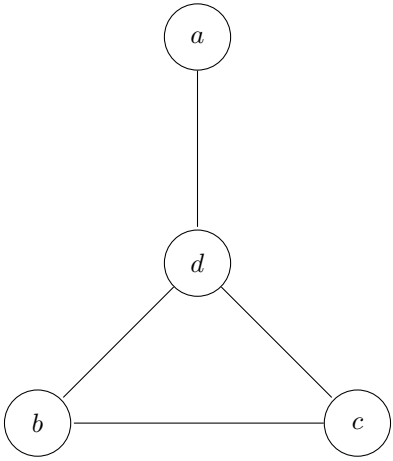


Figure 22: Gadget 2

Note that we can chain this gadget like a normal XOR gate if we attach another Gadget 2 to the c node. Then a will have the same colour as b, or the one of the children of the second gadget.

Gadget 3: OR Gate

This gadget has lots of nodes, though again, only a, b and c are external. Now, a is the same as either b, or c, or both b and c.

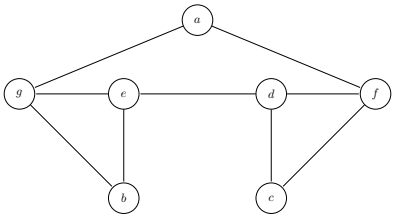


Figure 23: Gadget 3

I tried to encode $(a \vee \neg b), (b \vee c)$ as a colourability problem (shown in Figure **??**) but I did it wrong... Since it takes so long to write the graphs in LaTeX, I thought I'd let you spot the mistake I made ;)

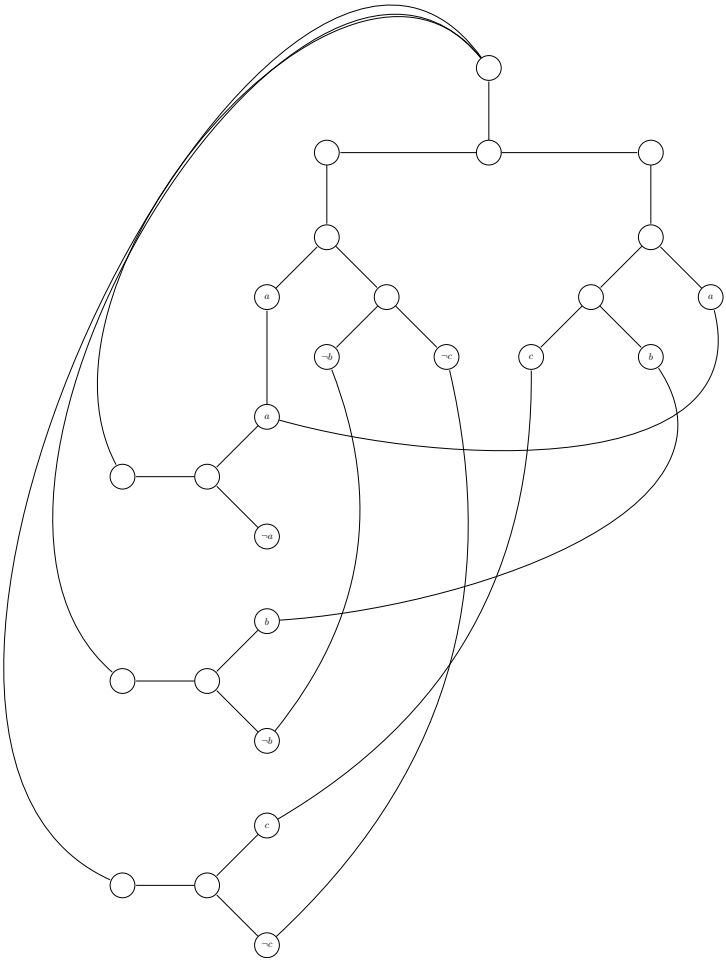


Figure 24: An incorrect encoding in *k-colourability* of the `SAT` problem $(a \vee \neg b), (b \vee c)$. The problem is that I used the wrong gadget (I used the second one instead of the third one) for the top row of nodes. Also, I've encoded a `SAT` problem, not a `3-SAT` problem so that's not really right either. Feel free to fix this with a pull request!

8.3 Space complexity

Note:

Make sure you read over lectures 10 and 11; there's a lot of stuff I've not included here.

Just like time complexity, space complexity is also measured using the Big-Oh notation. `Space(f(n))` is the space (number of tape cells) used by a deterministic Turing Machine, and `NSpace(f(n))` is the same for a nondeterministic Turing Machine.

Savich's theorem describes the relationship between deterministic and non- deterministic space complexity. We know that we can make a deterministic TM simulate a non-deterministic one in exponential time by simulating all branches of the latter's computation tree. Savich's theorem shows that a deterministic TM can simulate a non-deterministic one in the square of the space that would normally be used:

Note:

What does the Computer Scientist say when he goes into space? "OMG, $P = NP$!"

$$NSpace(f(n)) \in Space(f^2(n))$$

This is demonstrated by using directed graph reachability as an example. Given the directed graph G, we can determine if one node is reachable from another using an algorithm like:

```
# For log
import math

graph = {0 : [1,3,4],
         1 : [0,2],
         2 : [1,3],
         3 : [0,2],
         4 : [0]}

def is_reachable(start, end, graph, steps):
  print "is_reachable(%d, %d, graph, %d)" % (
  if steps == 0:
    return start == end or (end in graph[star
  else:
    for adj in graph[start]:
      if is_reachable(start, adj, graph, step
        if is_reachable(adj, end, graph, step
          return True
    return False

print is_reachable(0, 3, graph, int(math.log(
print is_reachable(4, 2, graph, int(math.log(
```

`is_reachable` finds if there is a path between the start and end nodes within 2^{steps} steps. In order to find if the nodes are connected in the graph, we have to give it $log_2(|graph|)$ as the steps parameter. This works because any path from *start* to *end* in 2^x steps must have a midpoint *mid*. If we find a path from *start* to *mid* in 2^{x-1} steps, and a path from *mid* to *end* in the same number, then we'll can put them together and find the complete path.

On a Turing Machine, we implement this by writing a tuple `(start,end,steps)` to the work tape on every step of the recursion. This requires $O(h \times log_2(|V|))$ space, and since we set *steps* to be $log_2(|graph|)$, the total space complexity is $O(log_2(|V|) \times log_2(|V|)) = O(log_2^2(|V|))$.

We can generalise this to any TM; if we visualise the execution of a non- deterministic TM as a graph, we can use the reachability algorithm above to determine whether we can reach an accepting state from the start state and we can do it in $SPACE(O(f(n))^2)$. This gives the result that $NSpace(f(n)) \in Space(f^2(n))$.

We can solve directed reachability with a depth first search in deterministic linear time and space, and have just shown that we can use `num_reachability` to solve it in $Space(log(n)^2)$, but we can in fact do even better space-wise.

Scooping the Loop Snooper

An elementary proof of the undecidability of the halting problem

No program can say what another will do.
Now, I won't just assert that, I'll prove it to you:
I will prove that although you might work til you drop,
you can't predict whether a program will stop.

Imagine we have a procedure called P
that will snoop in the source code of programs to see
there aren't infinite loops that go round and around;
and P prints the word Fine! if no looping is found.

You feed in your code, and the input it needs,
and then P takes them both and it studies and reads
and computes whether things will all end as they should
(as opposed to going loopy the way that they could).

Well, the truth is that P cannot possibly be,
because if you wrote it and gave it to me,