CHAPTER 1

SORTING AND TRIES

1.0 Algorithm Analysis

The time complexity of an algorithm can be thought of as a function of the input size. For example, if we are given an input of size n, the time complexity denotes how long it takes for the algorithm to run. It is either the worst case or the average case of the algorithm.

The worst case analysis is the one specified most commonly. It gives a guarantee of the algorithm's performance. A key factor is the asymptotic behaviour- this indicates what will happen as the input size grows. It is generally expressed using the 'Big Oh' notation. In some cases, space complexity can also be significant.

Big-Oh notation

In 'Big-Oh' notation, we say that $f(n) = O(g(n))$ if f and g grow at approximately the same rate. Here, the domain of f and g is the natural numbers. Formally, we say that $f(n) = O(g(n))$ if there exists a real constant c and an integer constant N such that $|f(n)| \leq |c \cdot g(n)|$ for all $n \geq N$. Usually, we use this notation to better understand the function f; f is typically a complicated function while g is a well-understood function.

When using the 'Big-Oh' notation, we use the tightest g that we can. For example, selection sort has worst-case complexity $O(n^2)$. This also means that selection sort is $O(n^3)$, $O(n^4)$, $O(n^2 \log n)$, etc. but we do not say that any of these are the worst-case complexity. However, we cannot state that selection sort is $O(n \log n)$.

The logarithmic function

We say that $x = \log_a n$ if $n = a^x$. The following are few standard properties of the function:

- $\log_a m \cdot n = \log_a m + \log_a n$;
- $\log_a m/n = \log_a m - \log_a n$;
- $\log_a n^c = c \log_a n$.

The logarithmic function grows very slowly. For example, $\log_2 1000000 \approx 20$. For this reason, $O(n^c \log_2 n)$ is only a little worse than $O(n^c)$.

Using the standard properties of the logarithmic function, we can show that logarithms to different bases related by a constant factor. So, let $x = \log_a n$. In that case, $n = a^x$. Now, if we take $\log_b$ of both sides, we get

$$\log_b n = \log_b a^x = x \log_b a = (\log_a n) \cdot (\log_b a).$$

Since b and a are constants, $\log_b a$ is also a constant. Therefore, $\log_b n$ and $\log_a n$ are related by a constant factor. In terms of 'Big-Oh' notation, this implies that $O(\log_a n) = O(\log_b n)$. For this reason, we specify both of these as just $O(\log n)$.

Polynomial-time algorithms

A polynomial-time algorithm runs in $O(n^c)$. The execution time of algorithms with various complexity functions is shown below:

	1 000	10 000	100 000	1 000 000
n	10^{-6} secs	10^{-5} secs	10^{-4} secs	10^{-3} secs
$n \log_2 n$	10^{-5} secs	1.3×10^{-4} secs	1.6×10^{-3} secs	0.02 secs
n^2	0.001 secs	0.1 secs	10 secs	16 mins
n^3	1 sec	16 mins	11 days	32 years

Table 1.1: The execution time of polynomial-time algorithms with various complexity functions provided that we can complete 10^9 operations per second.

Clearly, the higher the power, the longer it takes for the algorithm to be solved. Nonetheless, the table above denotes the worst-case behaviour- it is possible for the average case behaviour might be much faster.

Exponential-time algorithms

An exponential-time algorithm runs in $O(c^n)$. The execution time of algorithms with various complexity functions is shown below:

	10	20	30	40
2^n	10^{-6} secs	0.001 secs	1.1 secs	18 mins
$n!$	0.004 secs	77 years	$\star$	$\star$

Table 1.2: The execution time of exponential-time algorithms with various complexity functions provided that we can complete 10^9 operations per second.

A star ($\star$) indicates that it takes longer than the age of the Earth for the algorithm to terminate. Looking at the two tables, we can say that polynomial-time algorithms are potentially useful in practice, while exponential-time algorithms are potential useless in practice.

1.1 Abstract Data Types

Stack

A stack is an abstract data type that holds a collection of objects. The objects are accessed using a last-in-first-out (LIFO) policy. The basic operations defined on a stack are:

- `create`, which creates an empty stack;
- `isEmpty`, which checks if the stack is empty;
- `push`, which inserts a new item on the top of the stack; and
- `pop`, which removes the item from the top of the stack.

We can represent a stack as an array. Here, the bottom of the stack is 'anchored' to one end of the array. All the operations are $O(1)$ in that case. We can also represent a stack as a linked list. Again, all the operations are $O(1)$.

Queue

A queue is an abstract data type that too holds a collection of objects. The objects are accessed using a first-in-first-out (FIFO) policy. The basic operations defined on a queue are:

- `create`, which creates an empty queue;
- `isEmpty`, which checks if the queue is empty;
- `insert`, which inserts a new item at the back of the queue; and
- `delete`, which removes the item at the front of the queue.

We can represent a queue as an array. We need to wrap the queue around the array so that all the operations are $O(1)$. We can also use a linked list to represent a queue. Again, all the operations are $O(1)$.

Priority Queue

A priority queue is an abstract data type that is very similar to a queue. Instead of holding the elements in the given order, each element has a priority. When removing an element, we remove the one with the highest priority first. We can achieve this by keeping the queue sorted at all times, for example.

The basic operations defined on a priority queue are the same as a queue: `create`, `isEmpty`, `insert` and `delete`.

If we represent a priority queue as an unordered list, then the operation `insert` is $O(1)$ while the operation `delete` is $O(n)$. On the other hand, if we represent it as an ordered list, then the operation `delete` is $O(n)$ while the operation `insert` is $O(1)$. If we store it as a heap, then both `insert` and `delete` are both $O(\log n)$. In all cases, the other operations `create` and `isEmpty` are $O(1)$.

1.2 Sorting algorithms

We have seen before that naive sorting algorithms such as selection sort, insertion sort and bubble sort are $O(n^2)$ in worst (or average) case. There are also clever sorting algorithms such as heap sort, merge sort and quick sort that are $O(n \log n)$ in worst (or average) case. In practice, the fastest sorting algorithm is quicksort, even though its worse case is $O(n^2)$. This is because, in practice, the worst case is not encountered often.

The best runtime of these algorithms is $O(n \log n)$. In fact, it is not possible for us to use a comparison-based sorting algorithm with runtime better than $O(n \log n)$.

Comparison-based sorting

The algorithms we have above involve comparing elements to establish the order of the sorted list. For this reason, they are called comparison-based sorting algorithms.

An execution of a comparison-based sorting algorithm can be represented as a decision tree. In the decision tree, we have a node that compares two elements- this leads to two branches depending on which of the node is bigger. An node from the tree is shown below.

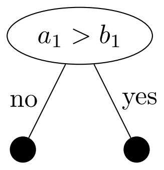


Figure 1.1: A node in a decision tree where we compare the elements a_1 and b_1 and proceed differently depending on which element is bigger.

In the decision tree, every node can be associated with a permutation of the list, each representing a different step in the sorting procedure (including those that do not sort the list). The leaf nodes represent a sorted list.

If we are given an unsorted list of size n, then any of the $n!$ permutations of the list could be the sorted version of the list. Therefore, we need at least $n!$ distinct leaves, each leading to a different permutation of the unsorted list. We could have more leaves, but in that case there will be a permutation associated with more than one leaf.

This implies that the worst-case complexity of a sorting-based algorithm is $O(h)$, where h is the height of the decision tree. An execution is a path from the root to the leaf node. Moreover, we need to perform an operation to go from one node to another, so we need to perform h operations in the worst-case.

If we have a binary tree of height h, the it has at most $2^{h+1} - 1$ leaf nodes. We can prove this by induction:

- If $h = 1$, then we have one node, which is a leaf. In that case, we have $n = 1 \leq 2^{h+1} - 1 = 2^2 - 1 = 3$.
- Now, assume that for some height h, every tree is of size $n_h \leq 2^{h+1} - 1$. Then, consider a tree of height $h + 1$. It has at most $2n_h$ more nodes compared to a tree of height h. Therefore,

  $$n_{h+1} \leq 2 \cdot n_h \leq 2 \cdot (2^{h+1} - 1) = 2^{h+2} - 2 \leq 2^{h+2} - 1.$$

  So, by induction, we find that a binary tree of height h has at most $2^{h+1} - 1$ leaf nodes.

A decision tree is a binary tree (since every node has two branches- 'yes' or 'no'). Therefore, we find that

$$n! \leq 2^{h+1} - 1 \leq 2^{h+1}.$$

Taking $\log_2$ of both sides, we find that

$$\begin{aligned} h + 1 &\geq \log_2(n!) \\ &\geq \log_2(n/2)^{n/2} \\ &= (n/2)\log_2(n/2) \\ &= (n/2)\log_2 n - (n/2)\log_2 2 \\ &= (n/2)\log_2 n - n/2. \end{aligned}$$

This implies that h is $O(n \log n)$. Here, we use the fact that $n! \geq \log_2(n/2)^{n/2}$. We can see this as follows:

$$\begin{aligned} n! &= n \cdot (n-1) \cdots (n/2) \cdot (n/2 - 1) \cdots 1 \\ &\geq n \cdot (n-1) \cdots (n/2) \\ &\geq (n/2) \cdot (n/2) \cdots (n/2) \\ &= (n/2)^{n/2}. \end{aligned}$$

Radix Sorting

We have proven that no sorting algorithm based on comparison can be better than $O(n \log n)$ in the worst case. So, if we want to improve on this worst-case bound, we have to create a method based on something other than comparisons.

Radix sorting uses a different approach to achieve $O(n)$ complexity. However, the algorithm has to exploit the structure of the items being sorted- it may be less versatile. Moreover, in practice, it is faster than $O(n \log n)$ algorithms only for very large n.

This algorithm breaks down a number into smaller chunks with fewer digits, sorts them out in place and continues sorting a more significant chunk. We consider sorting integers, but by transforming them into binary.

So, assume that every integer in the list can be written as a bit-sequence of length m. Moreover, let b be a chosen factor of m. Here, m and b are constants for a particular instance. We can label each item from 0 to $m-1$, with bit 0 being the least significant/leftmost bit.

The algorithm iterates m/b times. In each iteration, it places an element in one of the 2^b lists (or buckets), and joins each list to give the list to be used in the next iteration. The buckets correspond to a value between 0 and 2^b-1. During the i-th iteration, the bits from position $b \cdot (i-1)$ to $b \cdot i - 1$ are used to find the corresponding bucket.

Since the buckets get concatenated starting from the bucket representing the smallest value and the sorting is in place, we end up with a sorted list after all the loop terminates.

Now, we apply the radix sorting algorithm to sort the list below.

$$\boxed{15} \quad \boxed{43} \quad \boxed{5} \quad \boxed{27} \quad \boxed{60} \quad \boxed{18} \quad \boxed{26} \quad \boxed{2}$$

First, we convert it into binary. The longest bit string is of length 6. So, we have $m=6$. We choose $b=2$. The list is shown below as bit strings.

$$\boxed{00\ 11\ \underline{11}}\boxed{10\ 10\ \underline{11}}\boxed{00\ 01\ \underline{01}}\boxed{00\ 10\ \underline{11}}\boxed{11\ 11\ \underline{00}}\boxed{01\ 00\ \underline{10}}\boxed{01\ 10\ \underline{10}}\boxed{00\ 00\ \underline{10}}$$

In this case, we have $2^2 = 4$ buckets: 00, 01, 10 and 11. In the first iteration, we add these numbers into the corresponding buckets based on positions $2\cdot(1-1) = 0$ to $2 \cdot 1 - 1 = 1$.

00	60=111100		
01	5=000101		
10	18=010010	26=011010	2=000010
11	15=001111	43=101011	27=001011

We now concatenate the buckets to get the list we will use in the next iteration.

$$\boxed{60} \quad \boxed{5} \quad \boxed{18} \quad \boxed{26} \quad \boxed{2} \quad \boxed{15} \quad \boxed{43} \quad \boxed{27}$$

The binary representation of the list is now the following.

$$\boxed{11\ \underline{11}\ 00}\boxed{00\ \underline{01}\ 01}\boxed{01\ \underline{00}\ 10}\boxed{01\ \underline{10}\ 10}\boxed{00\ \underline{00}\ 10}\boxed{00\ \underline{11}\ 11}\boxed{10\ \underline{10}\ 11}\boxed{01\ \underline{10}\ 11}$$

We are now in the second iteration. So, we are considering the positions $2 \cdot (2-1) = 2$ to $2 \cdot 2 - 1 = 3$, as highlighted in blue above. We place them in the four buckets depending on the value within the highlighted position, and that gives us the following list of buckets.

00	18=010010	2=00 00 10	
01	5=000101		
10	26=011010	43=101011	27=011011
11	60=111100	15=001111	

Note that the value 26 comes before 27 since 26 came before 27 in the list at the start of this iteration. This happens because we are sorting in place. Concatenating this, we get the following list.

$$\boxed{18} \quad \boxed{2} \quad \boxed{5} \quad \boxed{26} \quad \boxed{43} \quad \boxed{27} \quad \boxed{60} \quad \boxed{15}$$

The list in binary, with the highlighted positions 4 and 5, is shown below.

$$\boxed{\underline{01}\ 00\ 10}\boxed{\underline{00}\ 00\ 10}\boxed{\underline{00}\ 01\ 01}\boxed{\underline{01}\ 10\ 10}\boxed{\underline{10}\ 10\ 11}\boxed{\underline{01}\ 10\ 11}\boxed{\underline{11}\ 11\ 00}\boxed{\underline{00}\ 11\ 11}$$

Now, we are in the final iteration. In total, there were $m/b = 6/2 = 3$ iterations. We add each element in the list into the bucket.

00	2=000010	5=000101	
01	18=010010	26=011010	27= 01 1011
10	43=101011		
11	60=111100		

We then concatenate the list.

$$\boxed{2} \quad \boxed{5} \quad \boxed{15} \quad \boxed{18} \quad \boxed{26} \quad \boxed{27} \quad \boxed{43} \quad \boxed{60}$$

This gives us the sorted list.

The pseudocode for radix sorting is given below.

```
List<int> radixSort(List<int> list, int m, int b):
    // the number of iterations
    final numIterations = m/b
    // the number of buckets
    final numBuckets = 2**b

    // initialise the buckets
    List<List<int>> buckets = List.generate(i => [])


    for int i in range(1, numIterations+1):
        // clear the buckets
        for List<int> bucket in buckets:
            buckets.clear()

        // add each number to the right bucket
        for int value in list:
            int k = value.toBitString()[b*(i-1):b*i].toInt(2)
            buckets[k].add(value)

        // concatentate the buckets at the end
        list = buckets.reduce(concatenate)
    return list
```

Next, we show that radix sorting is correct. That is, if we have x and y in a list with $x < y$, then we need to show that x precedes y in the final sequence. Suppose that j is the last iteration for which x and y differ (i.e. every bit after j is the same in x and y- this could be the final bit in the case that the first bit is different). Since $x < y$, the relevant bits of x must be smaller than those of y. Therefore, x goes to an 'earlier' bucket that y- after concatenation, x precedes y in the sequence after this iteration. Since any bit after this is the same in both x and y, they will both end up at the same bucket. Since the algorithm is in-place, x will still precede y at the end. Therefore, the final list will be sorted.

Finally, we consider the complexity of radix sort. The number of iterations is m/b and the number of buckets is 2^b. In each iteration, we scan the sequence and add them to one of the buckets- this takes $O(n)$ time. The buckets then get concatenated- this takes $O(2^b)$ time. So, the overall complexity is $O(m/b \cdot (n + 2^b))$. Since m and b are constants, this simplifies to $O(n)$.

When choosing the value b, there is a time-space trade-off. If we choose a large value of b, the multiplicative constant m/b will be small. So, the algorithm will become faster. But, we will need an array of size 2^b- this increases the space requirements exponentially.

1.3 Tries

A trie is used to store items that can be interpreted as a sequence of bits or characters. There is a multi-way branch at each node where each branch has an associated symbol and no two siblings have the same symbol. The branch taken at level i during a search is determined by the i-th element of the value (i-th bit, i-th character, etc.). Tracing a path from the root to a node spells out the value of the item. A trie can be used to store items with string values, e.g. words in a dictionary. An example is of a trie is given below.

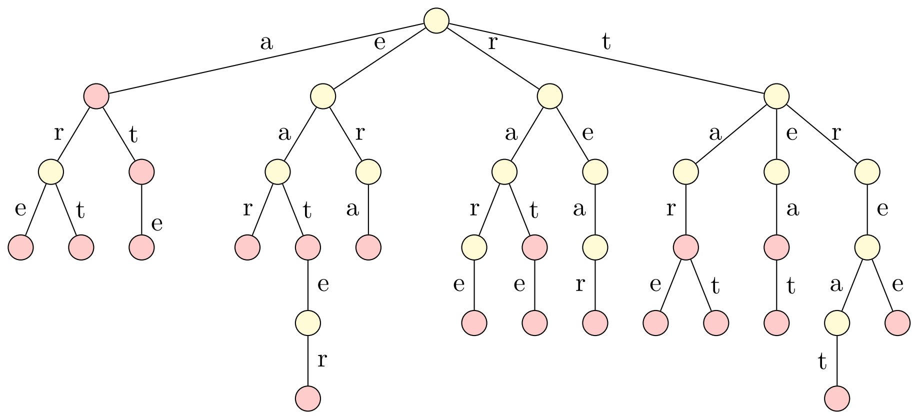


Figure 1.2: A Trie

We follow a path to spell out a word from left to right. A red node represents values that are words, while a yellow node represents intermediates. So, `tree` is a word but `trea` is not.

The search algorithm in a trie is given below:

```
bool search(Trie trie, String word):
    Node node = trie.root

    for char letter in word:
        // if the node has the required child,
        if node.hasChild(letter):
            // move to the next character
            node = node.getChild(letter)
        // otherwise, the word isn't present
        else:
            return false

    // for the trie to have a word, it actually needs to be a word
    return child.isWord
```

The insertion algorithm in a trie is given below:

```
void insert(Trie trie, String word):
    Node node = trie.root
    for char letter in word:
        // if the node has the required child,
        if node.hasChild(letter):
            // move to the next character
            node = node.getChild(letter)
        else:
            // create a node for that character,
            Node child = Node(letter)
            // attach it to the node,
            node.addChild(child)
            // and move to the next character
            node = child

    // the final character represents a word
    node.isWord = true
```

The deletion algorithm in a trie is given below:

```
bool delete(Trie trie, String word):
    Node node = trie.root

    for char letter in word:
        // if the node has the required child,
        if node.hasChild(letter):
            // move to the next character
            node = node.getChild(letter)
        // otherwise, the word isn't present
        else:
            return false

    // we can only delete a word
    if not node.isWord:
        return false

    // the node no longer represents a word
    node.isWord = false

    // while a node can be deleted,
    while node != trie.root and node.hasChild and not node.isWord:
        // delete it
        node.remove()
        node = node.parent

    // the word was present in the trie
    return true
```

Most trie operations are independent of the number of items present. They are essentially linear in the length of the string.

There are various possible implementations of a trie, e.g. arrays or linked lists. In a linked list implementation, a node stores the character it represents, along with pointers to the next child of the parent node (called its sibling) and the (first) child of this node. The code can be optimised by sorting the children nodes using the lexicographic order of the character.

CHAPTER 2

STRING AND TEXT ALGORITHMS

2.1 Text Compression

Text compression is a special case of data compression. It aims to save disk space and transmission time. Text compression, unlike image, audio or video, must be lossless, i.e. the original file must be recoverable without error. So, the original file can be converted into a compressed file using a compression algorithm, and it must also be possible to transform the compression file back into the original file unchanged, using a decompression algorithm.

There are two main approaches to text compression- statistical and dictionary. The compression ratio is the value x/y, where x is the size of the compressed file and y is the size of the original file. For example, if we compress a 10MB file into a 2MB file, the compression ratio is $2/10 = 0.2$. The percentage of space saved is $(1 - 0.2) \cdot 100\% = 80\%$. It is usual to have the space saving to be 40% to 60%.

Huffman Encoding

This is a classical statistical method of text compression. In this approach, a fixed (ASCII) code is replaced by a variable length code. Every character is represented by a unique codeword (a bit string). This approach compresses the file because we represent frequently occurring character with shorter codewords.

The code has the prefix property. That is, no codeword is a prefix of another. This allows us to decompress the text unambiguously- it is clear when to stop reading a bit string and how to interpret it.

This approach is based on the Huffman tree, which is a proper binary tree. Each character is represented by a leaf node. The codeword for a character is given by the path from the root to the appropriate leaf (left=0 and right=1). The prefix property follows from this. Following the bit string is equivalent to traversing the binary tree. Since all the characters are leaf nodes, there is no path that can correspond to more than one character. Moreover, we know we have reached the end when we encounter a leaf node.

We will now illustrate how to create the Huffman tree. So, assume that we have the following character frequency table for some text.

Space	E	A	T	I	S	R	O	N	U	H	C	D
15	11	9	8	7	7	7	6	4	3	2	1	1

Table 2.1: A character frequency table.

We shall construct a Huffman tree for these characters based on their frequency. We start by placing the characters (along with their frequency) in the tree, as leaf nodes.

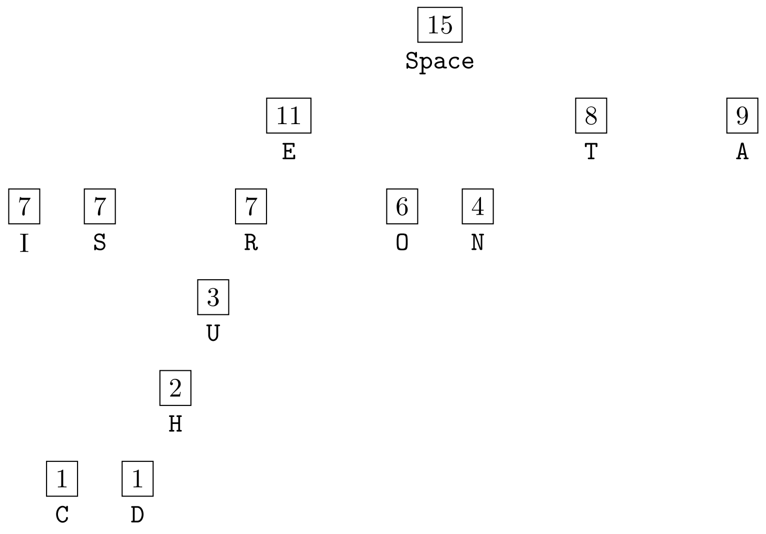
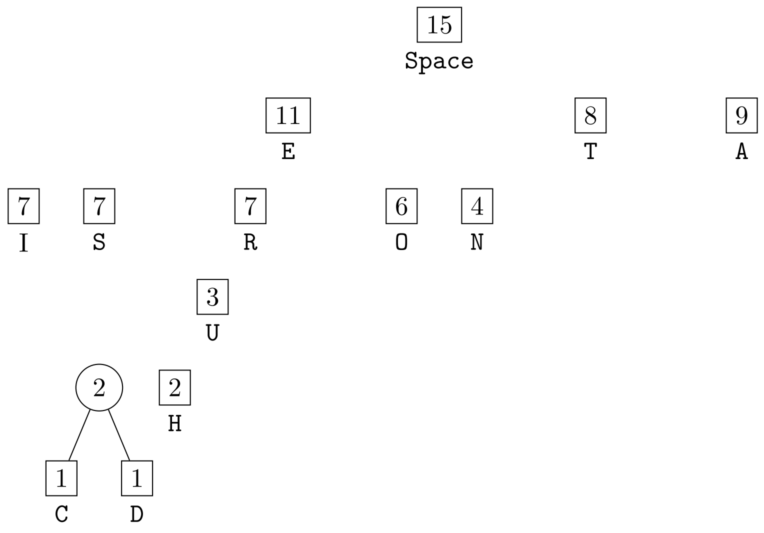
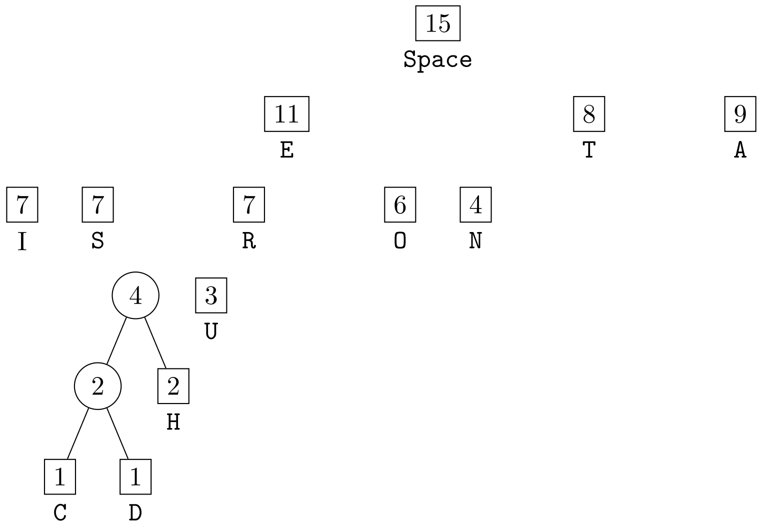


Figure 2.1: A Huffman tree to be formed for the character frequency tree above.

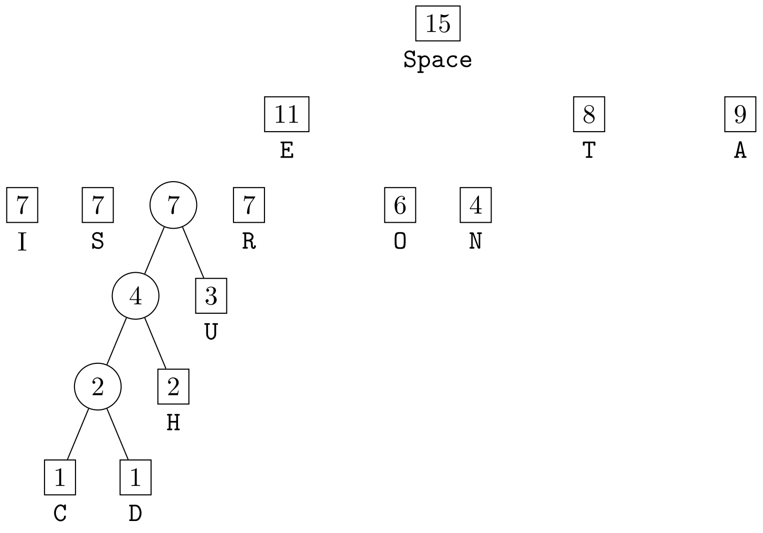
Next, we add a new parent to nodes of the smallest weights. The weight of this parent is equal to the sum of the weights of the child nodes.



In this case, we connect `C` and `D` with a parent node of weight 2. Now, we connect the new node with H to get a node of weight 4.



Now, we can either connect the parent node of weight 4 with `U`, or we can connect `N` with `U`. It does not matter which one we choose- we will just end up with a different compression algorithm in that case.



In this case, we will connect `U` to the parent node.

We can continue on connecting nodes until we end up with no parentless nodes.

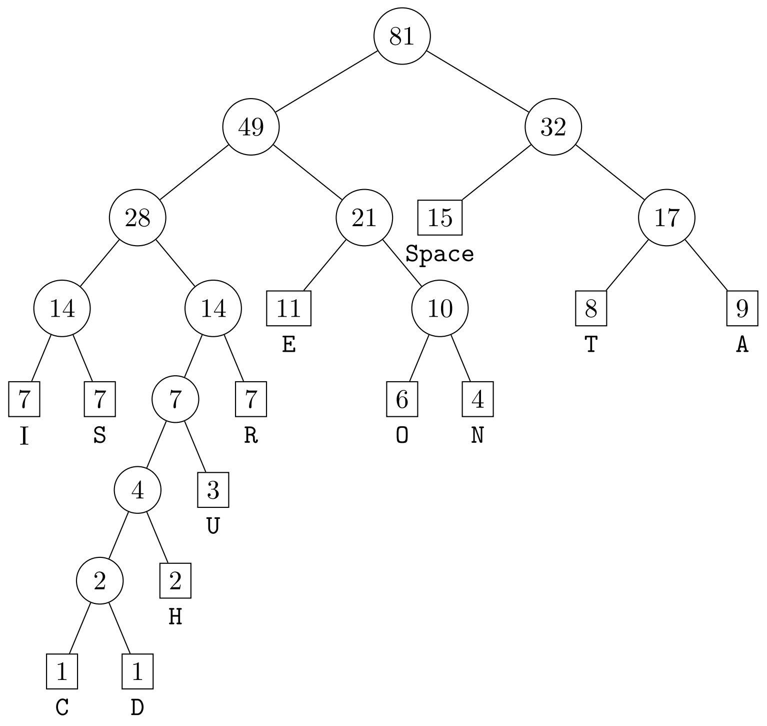


Figure 2.2: The Huffman tree for the character frequency given above.

Using the tree, we can construct Huffman code for each character. Going to the left corresponds to a `0` while going to the right corresponds to a `1`. This gives us the following table.

Space	E	A	T	I	S	R
10	010	111	110	0000	0001	0011
	O	N	U	H	C	D
	0110	0111	00101	001001	0010000	0010001

Table 2.2: The Huffman code for each character.

Clearly, the more frequently occurring values have a shorter code. Also, no code is a prefix of another. For example, if we read `111010`, it can only correspond to `AE`.

The pseudocode for constructing the Huffman tree is given below.

```
Tree huffman(Map<char, int> frequency):
    // the list of all the nodes in the tree
    List<Node> nodes = []
    // add each character node to the list
    for char key in frequency:
        Node node = CharNode(key)
        node.weight = frequency[key]
        nodes.add(node)

    Node node = null

    // until there's only one thing left in the list
    while not nodes.length > 1:
        // remove the two smallest values from the list
        Node left = nodes.removeSmallest(node => node.weight)
        Node right = nodes.removeSmallest(node => node.weight)

        // create the parent node
        node = Node(left, right)

        // set the weight to the sum of left and right weight
        node.weight = left.weight + right.weight

        // add it to the list
        nodes.add(node)

    // the root of the tree is the remaining node
    return Tree(nodes[0])
```

Huffman encoding is an optimal algorithm. To see this, we consider the weighted path length (WPL) of a tree T. This is the sum of the weight multiplied by the distance from the root product. For example, in the tree above, the WPL is

$$7 \times 4 + 7 \times 4 + 1 \times 7 + 1 \times 7 + 2 \times 6 + \cdots + 9 \times 3 = 279.$$

Huffman tree has minimum WPL over all binary trees with the given leaf weights. They need not be unique, though all Huffman trees for a given set of frequencies have the same WPL.

The weighted path length is the number of bits in the compressed file. This is because every character has a length (corresponding to the length of the edges) along with a frequency. So, the total number of bits is equal to the sum of this product for every node, i.e. the weighted path length. The Huffman tree has the minimum WPL, so this algorithm minimises the file size as much as possible.

Now, we analyse the algorithm. Assume that we have a text of length n containing m distinct characters. It takes $O(n)$ time to find the frequency. It takes $O(m \log m)$ time to construct the code, for example, using a (min) heap to store the parentless nodes and their weights. Initially, we need to build the heap- this takes $O(m)$ time. An iteration takes $O(\log m)$ time- we find and remove the two minimum weights, and insert a new weight. There are $m - 1$ iterations before the heap is left with one element. Overall, the algorithm is $O(n + m \log m)$. The number of characters m is essentially a constant, so it is really $O(n)$.

We also need to consider compression and decompression. Compression uses a code table (an array of codes indexed by a character). It takes $O(m \log m)$ to build the table as we have m characters with paths of length $\leq \log m$ since it is a binary tree. It takes $O(n)$ to compress- there are n characters in the text, so we lookup the code n times. Overall, it takes $O(n \log m) + O(n)$ time. Decompression uses the tree directly, so it is $O(n \log m)$ as we traverse n times a path of length $\leq \log m$.

In order to decompress, we must keep some representation of the Huffman tree along with the compressed file. An alternative to storing the tree is using a fixed set of frequencies based on typical values for text. But, this will usually reduce the compression ratio. We can also use adaptive Huffman coding. Here, the (same) tree is built and adapted by the compressor and the decompressor as characters get encoded and decoded. This slows the compression and the decompression, but not by much if done in a clever way.

LZW Encoding

LZW is a dictionary-based method. The dictionary is a collection of strings, each with a codeword that represents it. The codeword is a bit pattern, but it can be interpreted as a non-negative integer as well. Whenever a codeword is outputted during compression, what is written to the compression file is the bit pattern. It is a number of bits determined by the current codeword length. So, at any point, all bit patterns are the same length.

The dictionary is built dynamically during compression (and decompression). Initially, the dictionary contains all possible strings of length 1. Throughout, the dictionary is closed under prefixes. So, if the string s is represented in the dictionary, so is every prefix of s. Therefore, a trie is an ideal representation of the dictionary. Every node in the trie represents a 'word' in the dictionary. This makes trie an effective data structure to use during the algorithm.

At any given point during compression (or decompression), there is a current codeword length k. So, there are exactly 2^k distinct codewords available. This limits the size of the dictionary. However, the codeword length can be incremented as necessay, thereby doubling the number of available codewords. The initial value of k should be large enough to encode all strings of length 1.

The pseudocode for LZW compression is given below.

```
String lzwCompress(String string, int k, Map<String, Code> dict,
    Code code):
    // the index in the string
    int i = 0

    // the compressed file
    String bits = ""

    // until we've not looked at all the characters,
    while string.length < i:
        // find out how many letters we can find in the dictionary
        int j = 1
        while string[i:i+j] in dict:
            j++

        // add the string to the dictionary
        String s = string[i:i+j]
        dict[s+string[i]] = code.next()

        // add the code to the text
        String bits = dict[string[i:i+j]].toString(length=k)
        bits += s

        // move past the matched characters
        i += j
    return bits
```

Also, the pseudocode for LZW decompression is given below.

```
String lzwDecompress(String bits, int k, Map<Code, String> dict,
    Code code):
    // read the first k bits from the compressed file
    String string = dict[bits[:k]]
    // the index of the text
    int i = k

    // until there are bits left to read,
    while i < bits.length:
        // read the old string
        String oldS = string[-k:]

        // increase the codeword length if required
        if dict.isFull:
            k++

        // interpret the bit to string and add it to the file
        String s = dict[bits[i:i+k].toCode()]
        string += s

        // add the new string
        dict[code.next()] = oldS + s[0]

        // move past the code we just read
        i += k

    return string
```

Since tries are trees, when we find the longest string present starting at $s[i]$, we just need to go down a branch instead of searching a new string from the start. This means that the tries implementation is very efficient.

There are many variants to LZW compression. We can have a constant codeword length, where the dictionary has fixed capacity. When it is full, we stop adding more codewords. The one we saw above is the dynamic version, where we start with the shortest reasonable codeword length. When the dictionary becomes full, we add 1 to the current codeword length- it doubles the number of codewords. It does not affect the sequence of codewords already present. We may specify a maximum codeword length, as increasing the size indefinitely may become counter-productive. Another variant is the LRU version, where the current string replaces the least recently used string in the dictionary when the dictionary is full and the codeword length is maximal.

We will illustrate the algorithm on the following text: `GACGATACGATACG`. Assuming 2 bits per each character, the file size is 28 bits. There are 4 different characters present, so the initial codeword length $k = 2$. The initial dictionary is `A:00`, `C:01`, `G:10` and `T:11`.

We start from the beginning of the text. `G` is the longest string we can find in the dictionary starting there. We keep track of the codeword for `G`- 10. We add to the dictionary the longest string and the next character, i.e. `GA`. It gets code 100. We have now increased the codeword length from 2 to 3.

We are now at position 2. The longest string here is `A`- its codeword is 000. We add to dictionary `AC`- its codeword is 101. We can continue this process until we reach the final position. This process is summarised in the table below.

position	longest string	b	add to dictionary	code
1	`G`	10	`GA`	100
2	`A`	000	`AC`	101
3	`C`	001	`CG`	110
4	`GA`	100	`GAT`	111
6	`T`	011	`TA`	1000
7	`AC`	0101	`ACG`	1001
9	`GAT`	0111	`GATA`	1010
12	`ACG`	1001	-	-

Table 2.3: The LZW compression table for the text `GACGATACGATACG`

The compressed file is the concatenation of the codewords b. So, it is 100000011 00011010101111001. This has file size of 26 bits.

The decompression algorithm also builds the same dictionary as the compression algorithm, but it is one step out of phase. The implementation of the LZW decompression algorithm is given below.

We will illustrate the decompression algorithm by decompressing the bit string above. We start with the same dictionary at the start and the same codeword length. So, the codeword length is 2 and the dictionary is `A:0`, `C:1`, `G:2` and `T:3`. Note that the bit strings are in decimal here- they do not need to be.

First, we search for a code. The codeword length is 2, so we have the bit 10. This corresponds to `G`. At this point, we do not add anything to the dictionary. We keep track of the old string.

The dictionary is full, so we increment the size- the codeword length is now 3. We then look at the next 3 bits- we find 000. This corresponds to the character `A`. So, we keep track of `A`. We also add to the dictionary the old string and the first character of the new string- `GA`. It gets the code 4.

We can continue this process and decode the string. The following table summarises the process.

position	old string	dictionary code	string	add to dictionary	code
1	-	10	G	-	-
3	G	000	A	GA	100
6	A	001	C	AC	101
9	C	100	GA	CG	110
12	GA	011	T	GAT	111
15	T	0101	AC	TA	1000
19	AC	0111	GAT	ACG	1001
23	GAT	1001	ACG	GATA	1010

Table 2.4: The LZW decompression table for the encoded text 100000011000110101011111001.

Concatenating the string, we get back the original text: `GACGATACGATACG`.

During decompression, it is possible to encounter a codeword that is not (yet) in the dictionary. This is possible because decompression is 'out of phase' with compression. But, in that case, it is possible to deduce what the string it must represent- it is the old string at this point and the first character of the new string.

The complexity of both compression and decompression is $O(n)$, where n is the length of the text. The algorithm essentially involves just one pass through the text. We do not need to search for longest strings in the dictionary.

2.2 String Distance

Before discussing string distance, we will consider some string notations. If a string s is of length m, then $s[i]$ is the $i+1$-th element of the string if $0 \leq i < m$. Also, for $-m \leq i < 0$, $s[i]$ refers to $m-i$-th element of the string. The slice $s[i:j]$ refers to the substring from $s[i]$ (included) to $s[j]$ (excluded). If either value is missing, then we start from the start or we finish in the end respectively.

The j-th prefix of a string is the first j characters, i.e. $s[:j]$. The 0-th prefix $s[:0]$ is the empty string ϵ. Also, the j-th suffix is the last j characters, i.e. $s[-j:]$. Like in the case of prefixes, the 0-th suffix $s[0:]$ is the empty string.

We can look at two strings s and t and consider the distance between them. This is the smallest number of basic operations that we need to perform in order to transform s into t. There are 3 basic operations- insertion of a character (e.g. `red` $\to$ `read`), deletion of a character (e.g. `heat` $\to$ `eat`) and substitution of a character (e.g. `dead` $\to$ `bead`).

For example, consider the strings s = `abadcdb` and t = `acbacacb`. The following is an alignment between s and t.

s	a	-	b	a	d	c	d	-	b
t	a	c	b	a	-	c	a	c	b

Table 2.5: The alignment of the two strings `abadcdb` and `acbacacb`. The mismatches are shown in red.

An alignment illustrates how we can transform the string s into t- we add a `c`; we remove a `d`; we substitute a `d` with an `a`; and we add a `c`. Therefore, the distance between s and t is less than or equal to 4. It turns out that it is not possible for the distance to be 3 or lower- we will prove this later.

The string distance algorithm uses dynamic programming. The problem is solved by building up solutions to subproblems of ever increasing size. This is often called the tabular method since we build a table of relevant values. Eventually, one of the values in the table gives the required answer.

In the dynamic programming algorithm, let s and t be the two strings whose distance we want to compute. Let $d(i,j)$ be the distance between the i-th prefix of s and the j-th prefix of t. The distance between s and t is then $d(m,n)$, where m is the length of s and n the length of t.

The recurrence relation defining the distance is given by the distance between the shorter prefixes- $d(i-1,j-1)$ (which accounts for equal characters or substitution), $d(i,j-1)$ (which accounts for deletion of a character from t) and $d(i-1,j)$ (which accounts for deletion of a character from s).

The base case here is that $d(i,0) = i$ and $d(j,0) = j$. This reflects the fact that the distance from the empty string to a string of length k is k- we need to delete k elements from the string. In an optimal alignment of the i-th prefix of s with the j-th prefix of t, the last position of the alignment must either be a match, substitution, insertion or deletion. If there is a match, then we do not need to increment the distance. Otherwise, we increment the distance by 1. The recurrence relation is therefore given as

$$d(i,j) = \begin{cases} d(i-1,j-1) & s[i-1] = t[j-1] \\ 1 + \min(d(i-1,j-1), d(i,j-1), d(i-1,j)) & \text{otherwise.} \end{cases}$$

The dynamic programming algorithm for string distance follows immediately from the formula. We fill in the entries of an $m \times n$ table row by row and column by column. The algorithm has both time and space complexity $O(mn)$, as determined by the size of the table. We can easily reduce the space complexity to $O(m+n)$ by just keeping track of the previous column. To get the optimal alignment, we can use a traceback in the table. It is less obvious how this can be done using $O(m+n)$ space, but it turns out that this is still possible.

We illustrate this with an example. Assume we want to find the distance between string `acbacacb` and `abadcdb`. We initialise the table by the base case.

	ϵ	a	c	b	a	c	a	c	b
ϵ	0	1	2	3	4	5	6	7	8
a	1								
b	2								
a	3								
d	4								
c	5								
d	6								
b	7								

For example, the entry $(0,3)$ tells us that the distance between the empty string ϵ and the prefix `acb` is 3. We now fill the next row.

	ϵ	a	c	b	a	c	a	c	b
ϵ	0	1	2	3	4	5	6	7	8
a	1	0	1	2	3	4	5	6	7
b	2								
a	3								
d	4								
c	5								
d	6								
b	7								

The entry $(1,1)$ is a 0- since $s[0] = t[0]$, we take the entry at $(0,0)$. On the other hand, the entry $(1,2)$ is a 1- since $s[0] \neq t[1]$, we increment by 1 the smallest of $(1,1)$, $(0,1)$ and $(0,2)$, which is $(1,1)$.

We then move onto the next row.

	ϵ	a	c	b	a	c	a	c	b
ϵ	0	1	2	3	4	5	6	7	8
a	1	0	1	2	3	4	5	6	7
b	2	1	1	1	2	3	4	5	6
a	3								
d	4								
c	5								
d	6								
b	7								

Here, the entry $(2, 1)$ is a 1- since $s[1] \neq t[0]$, we add 1 to the lowest distance from the closest 3. Also, the entry $(2, 3)$ is a 1- since $s[1] = t[2]$, we just take the value in $(1, 2)$. We can continue this process and fill out the rest of the table.

	ϵ	a	c	b	a	c	a	c	b
ϵ	0	1	2	3	4	5	6	7	8
a	1	0	1	2	3	4	5	6	7
b	2	1	1	1	2	3	4	5	6
a	3	2	2	2	1	2	3	4	5
d	4	3	3	3	2	2	3	4	5
c	5	4	3	4	3	2	3	3	4
d	6	5	4	4	4	3	3	4	4
b	7	6	5	4	5	4	4	4	4

The entry at the final row, final column is 4. So, the distance between the strings s and t is 4.

The pseudocode for this algorithm is given below.

```
int stringDistance(String word1, String word2):
    // initialise the distance (with base case)
    List<List<int>> distance = List.generate(word1.length+1,
        (i) => List.generate(word2.length+1, (j) => i+j))

    // for each entry in the table,
    for int i in range(0, word1.length+1):
        for int j in range(0, word2.length+1):
            // if the values match, take the diagonal distance
            if word1[i-1] == word2[j-1]:
                distance[i, j] = distance[i-1, j-1]
            // otherwise, add 1 to the minimum entry
            else:
                distance[i, j] = 1 + min(distance[i-1, j-1],
    distance[i, j-1], distance[i-1, j])

    // return the last cell
    return distance[word1.length, word2.length]
```

To compute the optimal alignment between s and t, we enter the traceback phase. This traces a path in the table from the bottom right to the top left. We take a path from one entry to a previous one based on what led to the value of this node, i.e. whether there was a match and we took the diagonal, or whether the minimum value was above, etc.

A vertical step in the table refers to a deletion from s; a horizontal step is an insertion in s; and diagonal steps are matches (if the distance does not change) or substitutions. The traceback is not necessarily unique since more than one of the entries could have the minimum value. This implies that there can be more than one optimal alignment.

In the above table, we construct the following table.

	ϵ	a	c	b	a	c	a	c	b
ϵ	0	1	2	3	4	5	6	7	8
a	1	0	1	2	3	4	5	6	7
b	2	1	1	1	2	3	4	5	6
a	3	2	2	2	1	2	3	4	5
d	4	3	3	3	2	2	3	4	5
c	5	4	3	4	3	2	3	3	4
d	6	5	4	4	4	3	3	4	4
b	7	6	5	4	5	4	4	4	4

This gives us the following traceback.

	d	h	d	d	d	h	d	v	d
s	a	-	b	a	d	-	c	d	b
t	a	c	b	a	c	a	c	-	b

Table 2.6: An optimal traceback of the dynamic programming algorithm. The value d refers to a diagonal step; h a horizontal step; and v a vertical step.

Below is another example, where we compute the distance between the strings `saturday` and `sunday`.

	ϵ	s	u	n	d	a	y
ϵ	0	1	2	3	4	5	6
s	1	0	1	2	3	4	5
a	2	1	1	2	3	3	4
t	3	2	2	2	3	4	4
u	4	3	2	3	3	4	5
r	5	4	3	3	4	4	5
d	6	5	4	4	3	4	5
a	7	6	5	5	4	3	4
y	8	7	6	6	5	4	3

The distance is therefore 3. We can traceback to find the optimal path.

	ϵ	s	u	n	d	a	y
ϵ	0	1	2	3	4	5	6
s	1	0	1	2	3	4	5
a	2	1	1	2	3	3	4
t	3	2	2	2	3	4	4
u	4	3	2	3	3	4	5
r	5	4	3	3	4	4	5
d	6	5	4	4	3	4	5
a	7	6	5	5	4	3	4
y	8	7	6	6	5	4	3

So, the optimal traceback is provided below.

	d	v	v	d	d	d	d	d
s	s	a	t	u	r	d	a	y
t	s	-	-	u	n	d	a	y

Table 2.7: Optimal traceback of the dynamic programming algorithm.

2.3 String and pattern searching

String and pattern searching is where we search a (long) text for a (short) string. There are many variants, such as exact or approximate matches. Moreover, we can return the first occurence or all the occurences; we can search in one text for many strings; and we can search in many texts for one string.

Brute Force

Given a text t (of length n) and a string s (of length m), we are trying to find the position of the first occurence (if any) of s in t. The naive brute force algorithm (also known as exhaustive search, as we simply test all the possible solutions) sets the current starting position in the text to be zero. It then compares the text and string characters left-to-right until the entire string is matches, or we have a mismatch. If there is a complete match, then we return the right index. Otherwise, we advance the starting position by 1 and repeat.

The pseudocode for this algorithm is given below.

```
int bruteForce(String string, String substring):
    int m = string.length
    int n = substring.length

    // matching string[start:] to substring
    int start = 0
    // the index in string
    int i = 0
    // the index in substring
    int j = 0

    // until there is no possible alignment
    while start <= m-n:
        // if the values match, consider the next character
        if string[i] == substring[j]:
            i++
            j++

            // if we had a complete match, return the start index
            if j == n:
                return start
        // otherwise, restart
        else:
            // increment the starting position
            start++
            // move to string[start]
            i = start
            // move to substring[0]
            j = 0

    // we couldn't find the substring
    return -1
```

We illustrate the algorithm with an example as well. In the figure below, we try to find `ababaca` in the string `bacbababababacaab`.

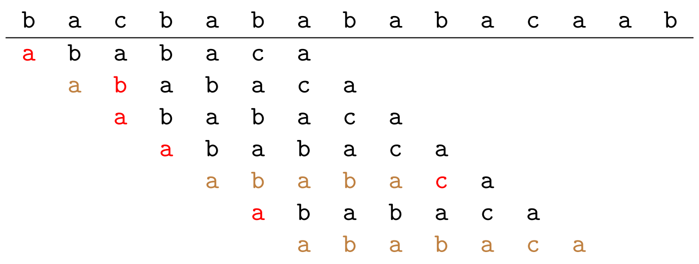


Figure 2.3: Brute Force Search of the string `ababaca` in the text `bacbababababacaab`. Red means a mismatch, while brown means a match.

We start checking from the beginning of the text. If there is a mismatch (i.e. the value we are checking in the string does not match the value in the text), then we restart, checking from the next value. We continue this until we find a match, or we have reached the end of the string.

We now analyse the algorithm. In the worst case, we reach the last character in the string, i.e. $s = \underbrace{\texttt{aa}\cdots\texttt{ab}}_{\text{length } m}$ and $t = \underbrace{\texttt{aa}\cdots\texttt{ab}}_{\text{length } n}$. We have m character comparisons need at each $n-(m+1)$ positions in the text before we find the pattern. So, the complexity is $O(mn)$. In practice, the number of comparisons from each point will be small. Often, it just takes 1 comparison to find a mismatch. So, we can expect $O(n)$ on average.

KMP Algorithm

Instead of using the brute force, we can use KMP algorithm for time complexity $O(m+n)$ in the worst case. It is an on-line algorithm, i.e. it removes the need to back up in the text. It involves pre-processing the string to build a border table. A border table is an array b with entry $b[j]$ for each position j of the string. If we have a mismatch at position j in the string, we remain on the current text character and do not back up. The border table tells us which string character should next be compared with the current text character.

A substring of string s is a sequence of consecutive characters of s. If s has length n, then $s[i:j]$ is a substring for valid i and j. A prefix of s is a substring that begins at position 0, i.e. $s[:j]$ for some j. A suffix of s is a substring that ends at position $n-1$, i.e. $s[i:]$ for some i. A border of a string is a substring that is both a prefix and a suffix, but not the entire string. For example, if we have the string $s =$ `acacgatacac`, then `ac` and `acac` are borders. In this case, the longest border is `acac`. Many strings do not have a border- their longest border is the empty string ϵ.

The KMP algorithm requires the border table of the string pattern. In the border table b, the entry $b[j]$ contains the length of the longest border of $s[:j]$. For example, the border table for `ababaca` is given below.

a	b	a	b	a	c	a
0	0	0	1	2	3	0

Table 2.8: The border table for the string `ababaca`.

The first two is 0 since the empty string and a character has no border. Since the string `aba` has a border of size 1, the fourth entry is 1. Also, the longest border of `ababa` is `aba`. This is allowed even though the middle `a` is part of both the prefix and the suffix. So, the second last entry is 3.

We shall now illustrate how the KMP is better than the brute force. Assume that we are at a mistmatch, like in the case below.

a	g	a	g	t	c	a	t	a	a	c	g	a
a	g	a	g	g								

In the brute-force case, we would advance the starting position by 1 and restart the search.

a	g	a	g	t	c	a	t	a	a	c	g	a
a	g	a	g	g								
	a	g	a	g	g							

However, the KMP algorithm moves the string along until we have a match before the string, like below.

a	g	a	g	t	c	a	t	a	a	c	g	a
a	g	a	g	g								
		a	g	a	g	g						

This way, we do not have to re-check the characters we have already checked before.

The first match directly corresponds to the longest border of the string. When we move the string, we require the start of the string to match with the end of the mismatch. So, we are looking for the (longest) prefix and suffix of the string that match- this is the longest border of the string up to (but excluding) the mismatch.

If we cannot move the string *s* along to get a match, then we move the string all the way to the end and set the starting position to be the mismatched character. If this was the first iteration, doing this would not move the string along. So, we do the same as brute force in this case- we move the starting position by 1.

The pseudocode for the KMP algorithm is given below.

```
int kmp(String string, String substring):
    int m = string.length
    int n = substring.length
    // the index in string
    int i = 0
    // the index in substring
    int j = 0

    List<int> borderTable = _createBorderTable(substring)

    // until we cannot find a match,
    while j <= n:
        // if the values match, consider the next character
        if string[i] == substring[j]:
            i++
            j++
            // return if end of substring
            if j == n:
                return i-j
        // otherwise,
        else:
            // if there is a border, go to end of border prefix
            if borderTable[j] > 0:
                j = borderTable[j]
            // if this is substring[0], increment string index
            else if j == 0:
                i++
            // otherwise, search the substring from start
            else:
                j = 0

    // the substring wasn't found
    return -1
```

We illustrate the algorithm by searching `ababaca` in the text `bacbababababacaab` in the figure below.

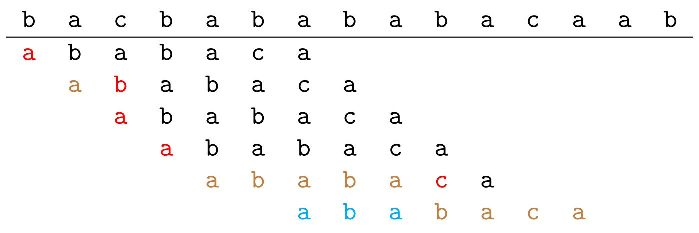


Figure 2.4: KMP search of the string `ababaca` in the text `bacbababababacaab`. Red means a mismatch, while brown means a match. Light blue represents the letters that did not need to be compared.

In the first iteration, there is a mismatch. Since this is the first character in the substring, we just move along by one alignment. In the next iteration, there is a mismatch in the second character. The value in the border table for this character is 0, so we still move along by one alignment. In the fifth iteration, there is a match until the sixth character. The value in the border table for this character is 3, so we move the string by 2 alignments. The character we search in the string remains the same as we do not need to check the first 3 characters in the substring. In the following iteration, we have a complete match.

Now, we consider the complexity of the KMP algorithm. We will consider the values of i and $k = i - j$ during the iteration. The loop runs for $i \leq n$, and since j is always non-negative, so $k \leq n$ as well. In each iteration, either i or k is incremented, and neither is decremented. Therefore, KMP is $O(n)$ in the worst case.

We also need to consider creating the border table. The naive method requires $O(j^2)$ to evaluate $b[j]$. Therefore, the algorithm is $O(m^3)$ overall. However, a more efficient method requires just $O(m)$ steps in total and involves a subtle application of the KMP algorithm. Therefore, the algorithm is $O(m+n)$.

Boyer-Moore

The Boyer-Moore algorithm is almost always faster than brute force or KMP. There are variants used in many applications. Typically, many text characters are skipped without even being checked. The string is scanned right-to-left. The text character involved in a mismatch is used to decide the next comparison.

For example, if we want to search for `pill` in `the caterpillar`, the process would run as follows.

```
t h e   c a t e r p i l l a r
-----------------------------
p i l l
        p i l l
                p i l l
                  p i l l
```

Figure 2.5: BM search of the string `caterpillar` in the text `the caterpillar`. Red means a mismatch, while brown means a match. Light blue represents a fixed letter when realigning the text.

We search the string from right to left- the `l` does not match the space, so we try to align a space character in that position. There is no space in `pill`, so we move it past the space. We search again- the `l` does not match the `l`. There is no `e` in `pill`, so we move past the character. In the next iteration, the `l` does match the `l`, but we then run into a mismatch. So, we try to ensure the `i` matches, so we shift the text by one. Then, we end up with a match. Clearly, the process avoids checking a lot of alignments.

In summary, the string is scanned from right to left. The text character involved in a mismatch is used to decide the next comparison. It involves pre-processing the string to record the position of the last occurrence of each character in the alphabet. Thus, the alphabet must be fixed in advance of the search. The character may not occur in the string, in which case we write it as a -1.

We shall now consider the 3 types of possible mismatch environments that BM considers with an example. We will search `acaa` in `abccaabacaababa`. The first possibility is that the mismatched character in the string is present in the substring, and we have not yet gone past the last occurrence in this iteration. For example, we can be in the following situation.

```
a b c c a a b a c a a b a b a
-----------------------------
a c a a
    a c a a
```

Here, we align with the mismatched character in the string with the last occurrence of the character in the substring. Another possibility is that the mismatched character is present in the substring, but we have gone past the character in this iteration. For example, we can be in the following situation.

```
a b c c a a b a c a a b a b a
-----------------------------
    a c a a
      a c a a
```

Here, we move the substring by 1 character and start searching again. The final possibility is that the mismatched character is not present in the substring. For example, we can be in the following situation.

```
a b c c a a b a c a a b a b a
-----------------------------
        a c a a
              a c a a
```

Here, we re-align the substring just past the mismatched character.

The pseudocode for the algorithm is given below.

```
int bm(String string, String substring):
    int m = string.length
    int n = substring.length

    // matching string[start:] to substring
    int start = 0
    // the index in string
    int i = m-1
    // the index in substring
    int j = m-1

    Map<String, int> lastOcurrence = _lastOcurrence(substring)

    // until there is no possible alignment
    while start <= m-n:
        // if the values match, consider the previous character
        if string[i] == substring[j]:
            i--
            j--

            // if we had a complete match , return the start index
            if j == -1:
                return start

        // otherwise ,
        else:
            int value = lastOcurrence[substring[i]]
            // if the character doesn't occur , then align after i
            if value == null:
                start = 1 + i
            // if we are past value , increment the alignment
            else if j > value:
                start ++
            // otherwise , fix substring[value] to this index
            else:
                start = 1 - value

            // update i and j
            i = m - 1 + start
            j = m - 1

    // we couldn't find the substring
    return -1
```

The worst case of the algorithm is no better than $O(mn)$. For example, we can have $s = \underbrace{\texttt{ba...aa}}_{\text{length } m}$ and $t = \underbrace{\texttt{aa...aaaa...aa}}_{\text{length } n}$. There are m character comparisons needed at each $n-(m+1)$ positions in the text before the string is found. There is an extended version that is $O(m+n)$- it uses the good suffix rule.

We will now illustrate the BM algorithm by searching `ababaca` in the string `bacbababacaab`. We start by checking the initial alignment right to left. The `a` matches with the `a`, but the `c` does not match the `b`. The string has a `b` before the `c`, so we align the text with the closest `b`. We then start searching again. The `a`'s match, but the `b` does not match with the `c`. So, we align the text with the closest `b` before the `c` and restart. We continue like this until there is a complete match. This is presented pictorially below.

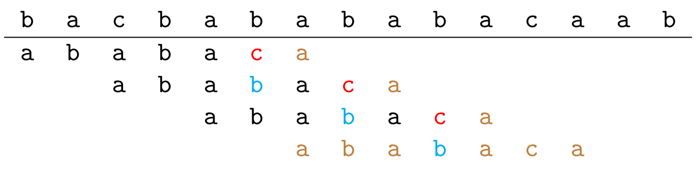


Figure 2.6: BM search of the string `ababaca` in the text `bacbababacaab`. Red means a mismatch, while brown means a match. Light blue represents a fixed letter when realigning the text.

CHAPTER 3

GRAPH AND GRAPH ALGORITHMS

3.1 Introduction to Graphs

Undirected graphs

An *undirected graph* G is defined to be the tuple (V, E), where V is a finite set of vertices (the vertex set), and E is the set of edges, where every edge is a subset of V of size 2. Pictorially, we can represent each vertex as a point and an edge as a line connecting two points. For example, if $V = \{a, b, c, x, y, z\}$ and $E = \{\{a, x\}, \{a, y\}, \{a, z\}, \{b, x\}, \{b, y\}, \{b, z\}, \{c, x\}, \{c, y\}, \{c, z\}\}$, then it can be represented as the following graph:

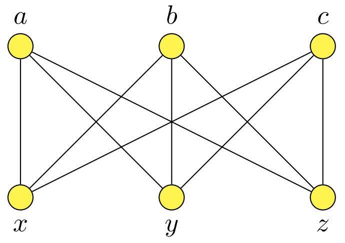


Figure 3.1: An undirected graph

A graph can have multiple pictorial representations. For example, the graph above can also be represented as the following graph:

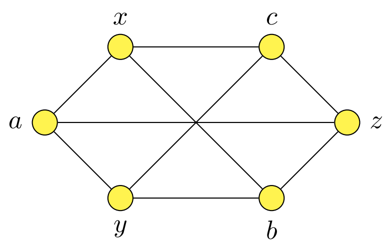


Figure 3.2: Another representation of the undirected graph above

We say two vertices are *adjacent* if there is an edge between them. Otherwise, the two vertices are *non-adjacent.* For example, in the graph above, vertices a and z are adjacent. We say a vertex is *incident* of an edge if it is part of the edge. A *path* traverses through edges and goes from one vertex to another. Its length is the number of edges present. For example, $a \to x \to b \to y \to c$ is a path of length 4. A *cycle* is a path that starts and ends at the same node. For example, $a \to x \to b \to y \to a$ is a cycle of length 4. The *degree* of a vertex is the number of edges it is incident to. For example, all the vertices above have degree 3.

A *connected* graph if there is a path between any pair of vertices. For example, the following is a connected graph.

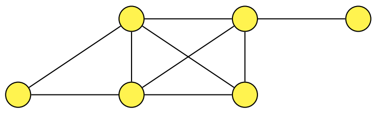


Figure 3.3: A connected graph

If a graph is not connected, it is composed of two or more connected components, like in the case below.

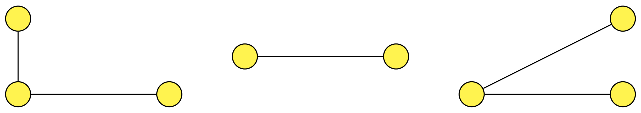


Figure 3.4: An unconnected graph with connected components

A graph is a *tree* if it is connected and doesn't have any cycles. For example, the following is a tree.

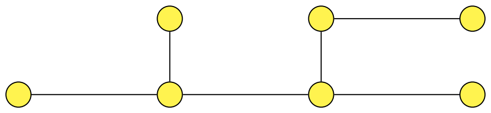


Figure 3.5: A Tree

A tree with n vertices has precisely $n-1$ edges. Since it is connected, we must have at least $n-1$ edges (since we can reach from one vertex to another). Also, since it is not cyclic, we must have at most $n-1$ edges (if we have another edge, then we must have a vertex in at least two edges, and since it is connected, we can find a cycle from that vertex to itself).

A graph is a *forest* if it is not cyclic and all its components are trees. For example, Figure 3.1 is a forest. A graph is *complete* or (*clique*) if every pair of vertices is joined by an edge. For example, the figure below is a clique on 4 vertices.

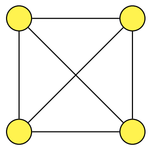


Figure 3.6: The clique on 4 vertices K_4

A graph is *bipartile* if the vertices are in two distinct sets U and W, and every edge joins a vertex in U to a vertex in W. For example, the following is a partition of a complete bipartile graph on 6 vertices.

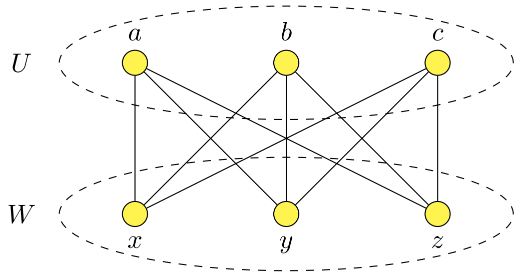


Figure 3.7: A complete bipartile graph $K_{3,3}$

Bipartile graphs do not need to be complete, but the one above is.

Directed graphs

A *directed graph* is like undirected graphs, but the edges have a direction. So, we have a finite set of vertices V and a finite set of edges E, but every edge is an ordered pair (x, y) of vertices. This corresponds to having an edge from x to y.

An example of a directed graph is given below.

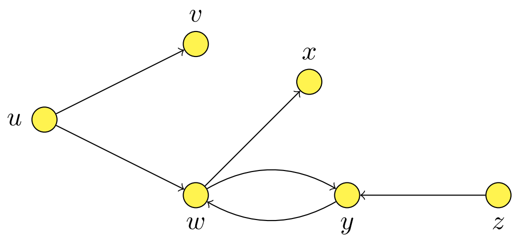


Figure 3.8: A directed graph

The terminology for directed graphs is slightly different to the ones we discussed above. In the particular case of the directed graph above, we say that:

- the vertex u is *adjacent to* v, and the vertex v is *adjacent from* u;
- the vertex y has *in-degree* 2 and *out-degree* 1;
- $u \to w \to x$ is a valid *path*, but $x \to w \to u$ is not a path;
- $w \to y \to w$ is a valid *cycle*.

3.2 Graph representations

We can represent an undirected graph using an *adjacency matrix*. This matrix contains one row and one column for each vertex. The value of row i and column j is a 1 if the i-th and the j-th vertices are adjacent; otherwise, the value is 0. We can also represent them as *adjacency lists*. We store a list for each vertex. The list i contains an entry for j if the vertices i and j are adjacent.

For example, consider the undirected graph below.

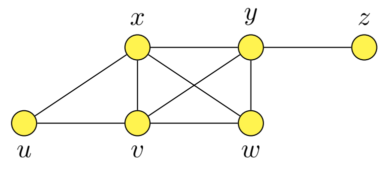


Figure 3.9: An undirected graph

We can represent it as the following adjacency matrix.

	u	v	w	x	y	z
u	0	1	0	1	0	0
v	1	0	1	1	1	0
w	0	1	0	1	1	0
x	1	1	1	0	1	0
y	0	1	1	1	0	1
z	0	0	0	0	1	0

Table 3.1: The adjacency matrix representation for the undirected graph above

Also, we can represent it as the following adjacency list.

u	$[v, x]$
v	$[u, w, x, y]$
w	$[v, x, y]$
x	$[u, v, w, y]$
y	$[v, w, x, z]$
z	$[y]$

Table 3.2: The adjacency list representation for the undirected graph above

In the case of directed graphs, we can still use both adjacency matrices and adjacency lists. However, in an adjacency matrix, the entry in row i and column j is a 1 if there is an edge from i to j; it is 0 otherwise. In an adjacency list, the list for vertex i contains a vertex j if there is an edge from i to j.

For example, consider the directed graph below.

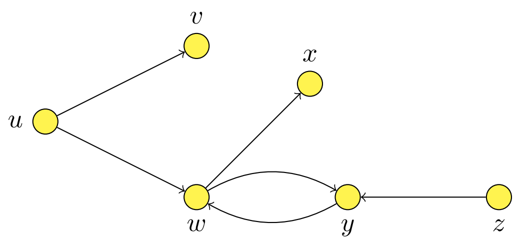


Figure 3.10: A directed graph

We can represent it as the following adjacency matrix.

	u	v	w	x	y	z
u	0	1	1	0	0	0
v	0	0	0	0	0	0
w	0	0	0	1	1	0
x	0	0	0	0	0	0
y	0	0	1	0	0	0
z	0	0	0	0	1	0

Table 3.3: The adjacency matrix representation for the directed graph above

Also, we can represent it as the following adjacency list.

u	$[v, w]$
v	
w	$[x, y]$
x	
y	$[w]$
z	$[y]$

Table 3.4: The adjacency list representation for the directed graph above

In the case of an undirected graph, the adjacency list features an edge $\{a, b\}$ twice- once as going from a to b, and another time as going from b to a. On the other hand, in the case of a directed graph, the adjacency list features an edge (a, b) precisely once- it is only present in the list corresponding to the vertex a. In both cases, an adjacency matrix is an $|V| \times |V|$ array, while an adjacency list contains $|V|$ arrays which are themselves arrays containing vertices.

To implement this, we define classes for the entries of the adjacency lists, the vertices (as a linked list representation), and graphs (which includes the size of the graph and an array of vertices). The array allows for efficient access using the index of a vertex- we can then access it in constant time.

3.3 Graph search and traversal

Graph search and traversal is a systematic way to explore a graph, starting from some vertex. A search visits all the vertices by travelling along the edges. So, the traversal is efficient if it explores the graph in $O(|V| + |E|)$ time.

Depth First Search

Depth first search starts at a particular vertex, follows an edge from this vertex and continues on following some edge from that vertex until there is no edge to follow. It then backtracks and tries to take a different edge to find unvisited vertex until there are no more edges. Then, it starts the procedure all over again on an unvisited vertex (if the graph is not connected).

We illustrate this with an example below. So, consider the following graph with a starting vertex.

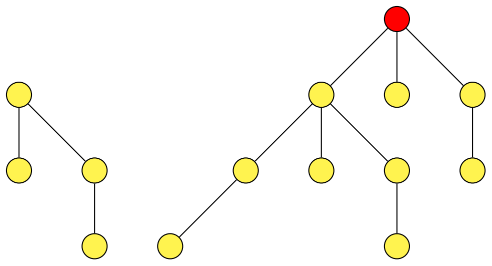


Figure 3.11: An undirected graph to be traversed via depth-first search.

We will label visited vertices in red. We take an edge and discover another vertex.

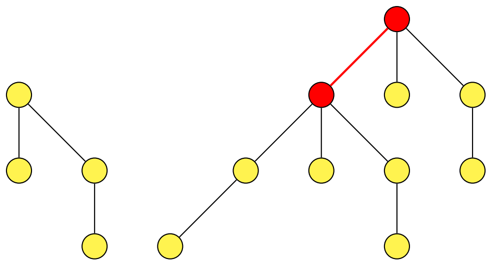
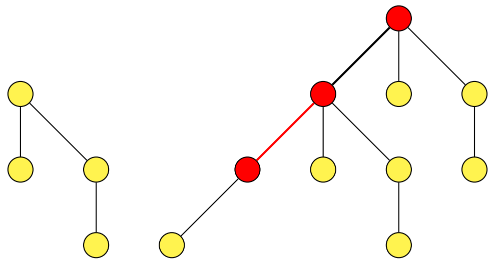
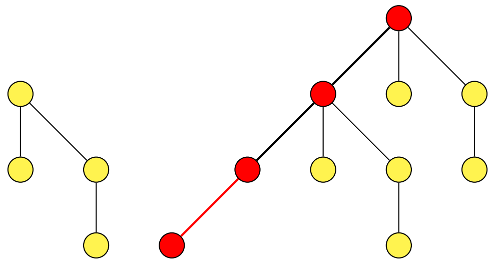


Figure 3.12: An undirected graph to be traversed via depth-first search.

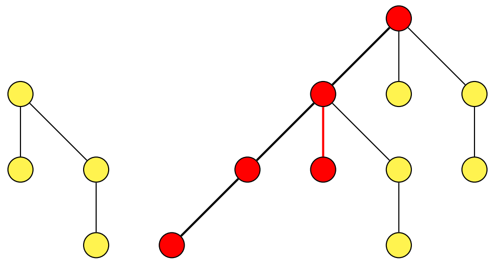
The edge we just took is in red. At this point, we have two visited vertices, and the lower one is the current vertex. We take another edge from this vertex and discover another unvisited vertex.



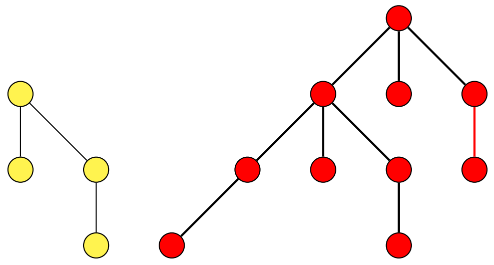
The visited edges are thicker. We have no discovered three vertices. There is only one edge we can take there, so we take it.



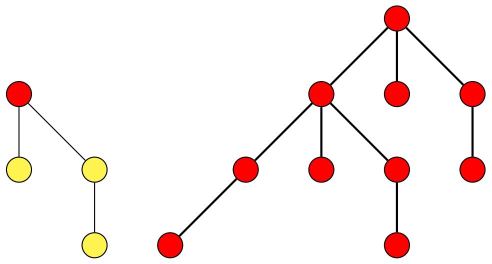
At this point, there is no edge we can take from the current vertex. So, we backtrack and go back two edges to find a new edge to take.



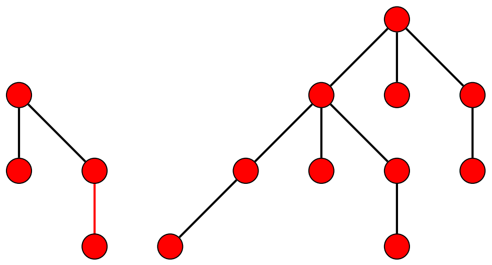
We can then continue on, finding new vertices and backtracking until we have completely visited all the vertices in the right connected component.



Since we have visited all the edges in the right connected component, we randomly choose an edge that has not yet been visited.



We find all the vertices we can visit from this vertex like we did at the start.



We have now visited all the vertices, so the depth first search is complete.

The edges we traverse during this procedure form a *spanning tree* (or a forest if the graph is not connected). A spanning tree of a graph is a tree composed of all the vertices and some (or perhaps all) of the edges of the graph. Spanning trees that are found through a depth-first search traversal are called *depth-first spanning trees*.

We will show how to form a spanning tree in the example below. Here, we have selected a vertex to start the depth-first search at.

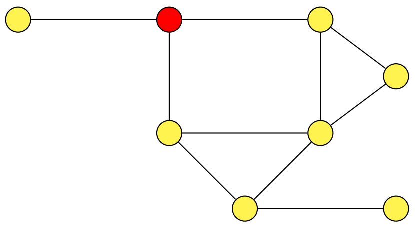
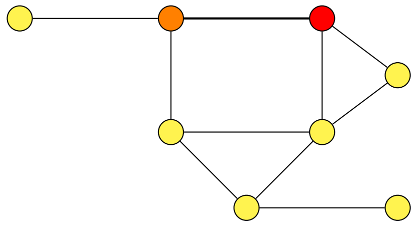
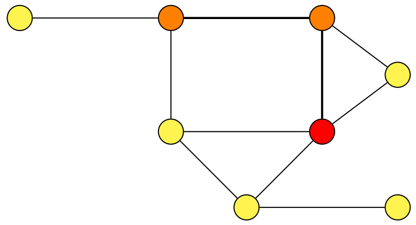


Figure 3.13: An undirected graph being visited by depth-first search.

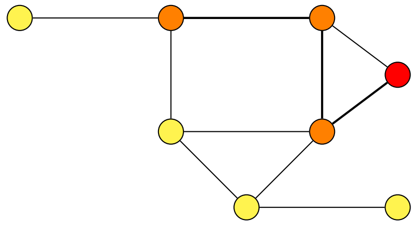
The vertex in red is the current vertex for the depth-first search (in order to find the minimum spanning tree). Like before, we will take some edge to find unvisited vertices.



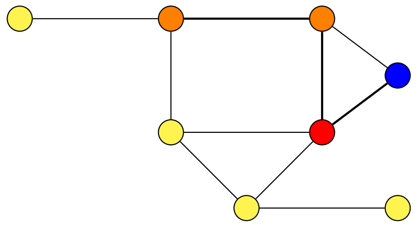
The vertices in orange are previously visited vertices. Thick edges are visited. We continue the depth-first search algorithm and find another unvisited vertex from the current vertex.



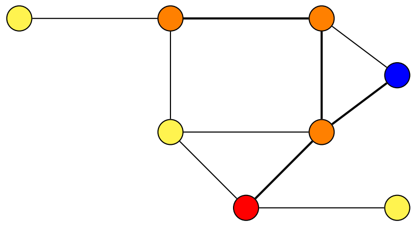
From the current vertex, we again find another unvisited vertex.



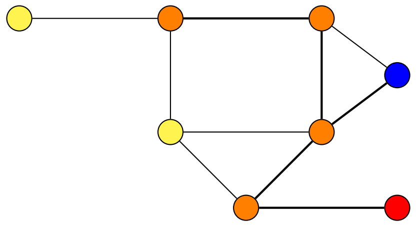
There is only one edge that we haven't taken that this vertex is incident to. It turns out that taking this edge doesn't find us an unvisited vertex, so we have exhausted this vertex. We then backtrack and consider the previous vertex.



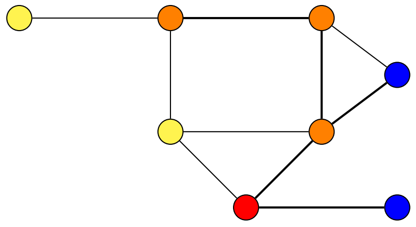
The vertices in blue are vertices such that all of their edges have been completely explored- these vertices have been exhausted and we do not need to consider any edges from this vertex from this point. From this vertex, there is an edge that takes us to an unvisited vertex, so we take it.



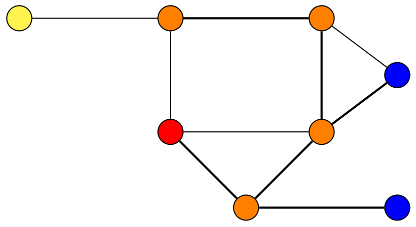
From this edge, we can find another unvisited vertex.



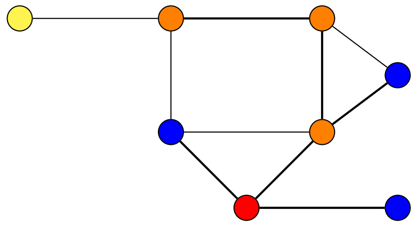
There is no edge we haven't taken that is incident to the current vertex. So, we mark it as exhausted and backtrack.



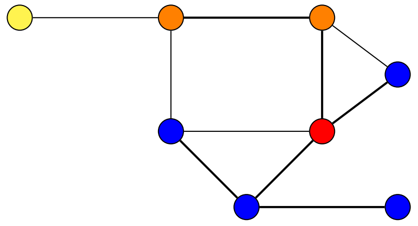
We can find an edge from here that takes us to an unvisited vertex, so we take it.



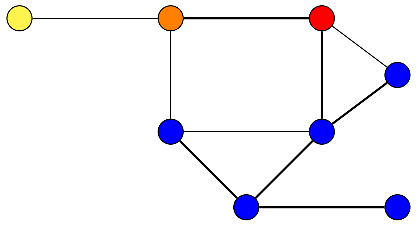
There are two edges from this vertex that we haven't taken, but both of them lead to vertices we have already visited. So, we mark this vertex as exhausted and backtrack.



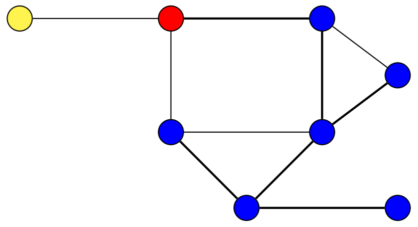
There is no edge we haven't visited through the current edge, so we mark it as exhausted and backtrack.



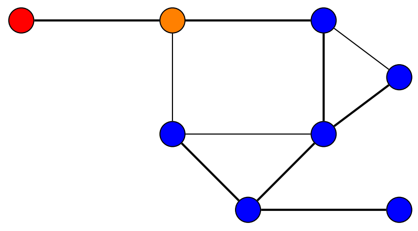
There is only one edge from here that we haven't considered, and it leads to a visited vertex. So, we mark this vertex as exhausted and backtrack again.



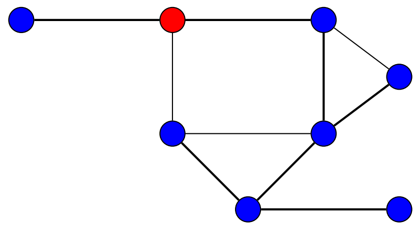
Like before, there is no edge we can take to find an unvisited vertex. So, we mark this vertex as exhausted and backtrack again.



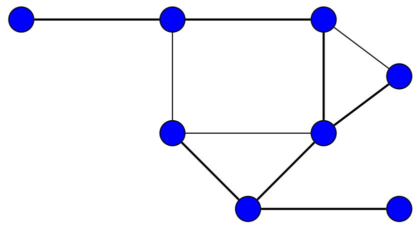
Finally, we can find an edge from the current vertex that goes to an unvisited vertex.



There is no edge we can take from here. So, we mark this vertex as exhausted and backtrack.



There is no edge we can take from here to take us to an unvisited vertex, so we mark this as exhausted.



We cannot backtrack since this was the first edge, so we have found the minimum spanning tree.

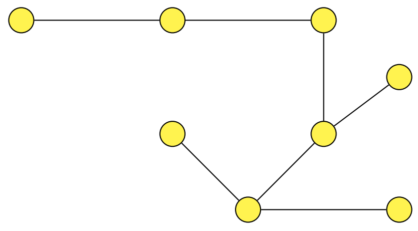


Figure 3.14: A spanning tree of the graph above, found using depth-first search.

We made many choices when selecting the edges. This implies that there is no unique depth-first spanning tree.

The following is the implementation of depth first search.

```
void visit(Vertex vertex1):
    // this vertex has now been visited
    vertex1.visited = true

    // for a vertex that is incident to vertex1
    for Vertex vertex2 in vertex1.incidentVertices:
        int i = node.vertexIndex

        // if this vertex hasn't been visited, visit it
        if not vertex2.visited:
            visit(vertex2)

void dfs(Graph graph):
    // visit a vertex if it has not been visited
    for Vertex vertex in vertices:
        if not vertex.visited:
            visit(vertex)
```

We now analyse the complexity of depth first search using adjacency lists. We visit each vertex once, and each element in the adjacency list is processed. So, the overall complexity is $O(|V|+|E|)$. We can adapt it using the adjacency matrix representation, but then it is $O(|V|^2)$ since we need to look at every entry of the adjacency matrix.

There are many applications of depth-first search. It can be used to determine if a graph is connected; to identify the connected components of a graph; to determine if a graph contains a cycle; and to determine if a graph is bipartile.

Breadth First Search

In depth-first search, we saw that we keep traversing in the same path until we cannot go any further. On the other hand, in breadth-first search, we visit all the edges of a particular vertex before visiting any vertices adjacent to them; the search fans out as widely as possible at each vertex. At every point, we keep a queue of vertices so we know the order in which we visit the vertices.

We show how this algorithm works by example. So, we start with an empty queue and one randomly chosen vertex as visited.

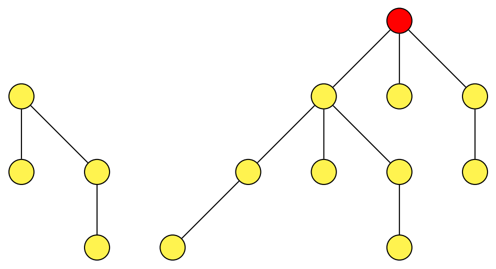


Figure 3.15: An undirected graph to be traversed via breadth-first search.

The visited vertices are in red, and the current edges are red. We find three edges here. So, we add the three vertices to a queue.

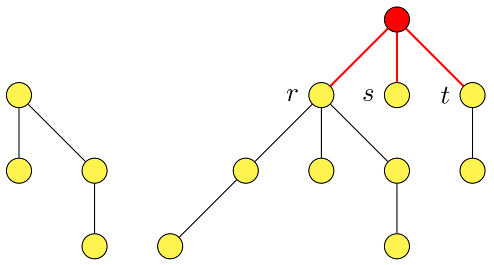
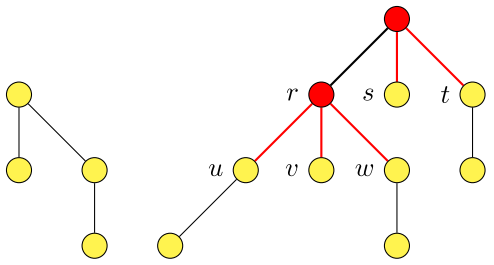
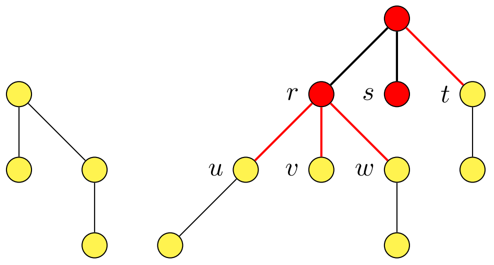


Figure 3.16: An undirected graph to be traversed via depth-first search.

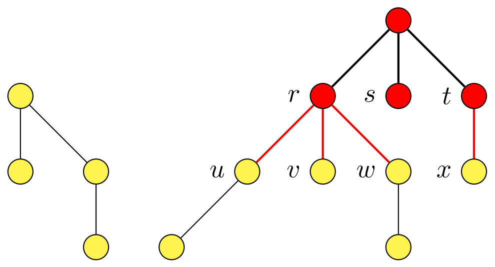
The queue is now $\langle r, s, t\rangle$. We remove the first element from the queue and visit that vertex.



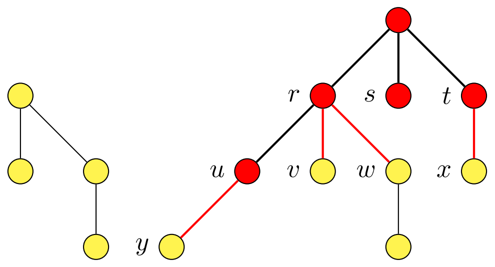
The visited edges are thicker. We now find 3 more vertices, so we add them to the queue as well. The queue is $\langle s, t, u, v, w\rangle$. We remove the first element from the queue and visit that vertex.



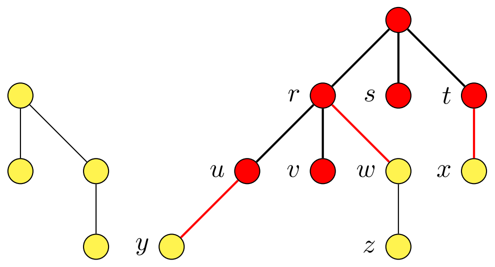
There are no edges from s that are to be visited, so the queue remains $\langle t, u, v, w\rangle$. We remove t from the queue and visit that vertex.



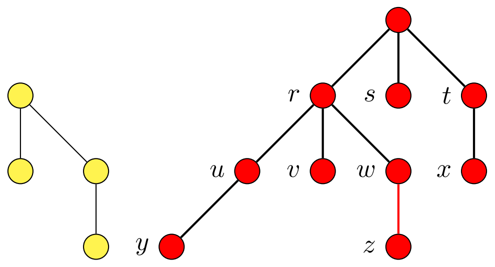
We add x to the queue now, so the queue is $\langle u, v, w, x \rangle$. We remove u now and visit that vertex.



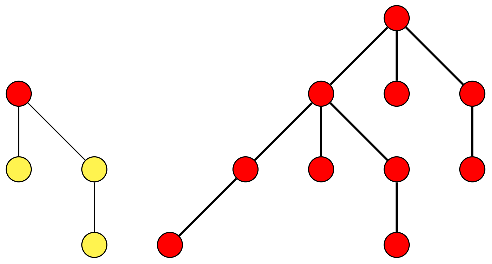
We add y to the queue now. The queue is then $\langle v, w, x, y \rangle$. So, we remove v and visit it.



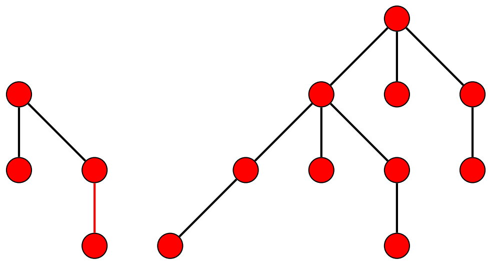
There is no vertex to add here, so the queue remains $\langle w, x, y \rangle$. We can continue this process and find all the remaining vertices.



There are still unvisited vertices, so we select an unvisited vertex.



We continue this process and find all the vertices like above.



Like in the case of a depth-first search, the edges traversed form a spanning tree (or a forest). We illustrate this with an example- we will traverse the following graph using breadth-first search.

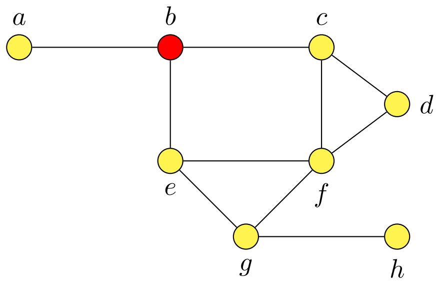
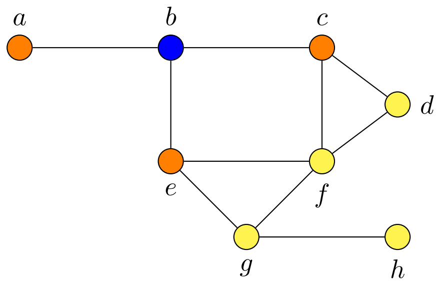
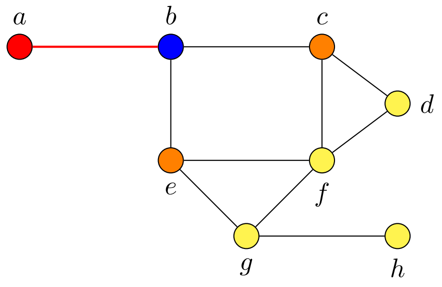


Figure 3.17: An undirected graph being visited by breadth-first search.

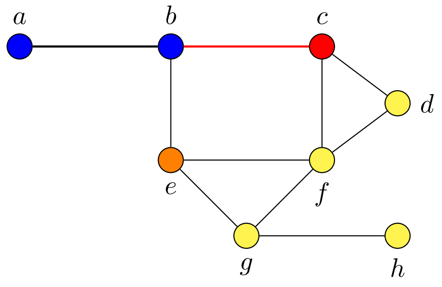
The vertex in red is the current vertex for the breadth-first search (in order to find the spanning tree). At this point, the queue is $\langle b \rangle$. So, we remove b from the queue and visit it.



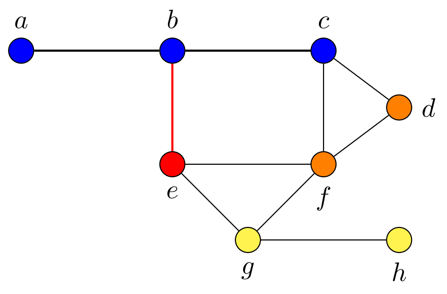
The vertices in blue are (completely) visited vertices, and orange vertices are those in the queue. At this point, the queue is $\langle a, c, e \rangle$. So, we remove a from the queue and visit it.



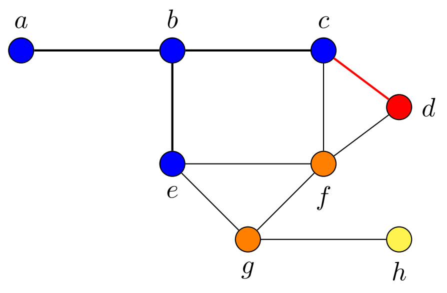
The red edge is the current edge. We do not discover any new vertex, so the queue remains $\langle c, e \rangle$. Next, we remove c from the queue and visit it.



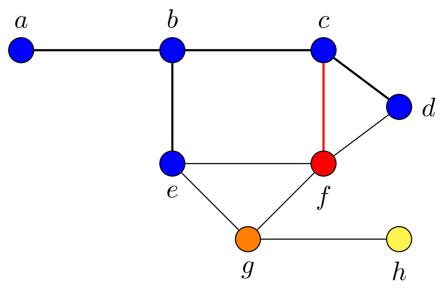
Thick edges are visited. We add the vertices d and f to the queue, and remove e from the queue.



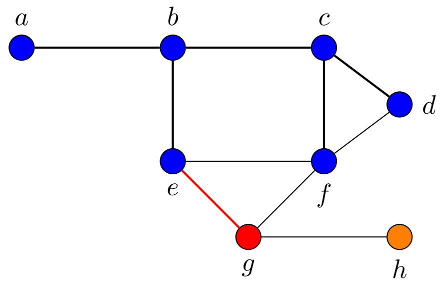
At this point, the queue is $\langle d, f \rangle$. We add g to the queue, and remove d.



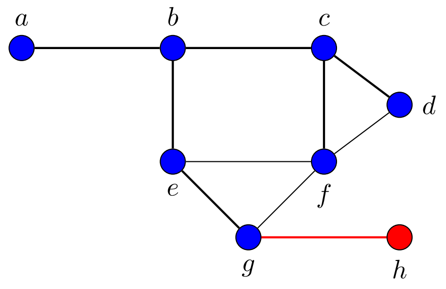
We do not discover any new vertices, so the queue remains $\langle f, g \rangle$. So, the next vertex we visit is f.



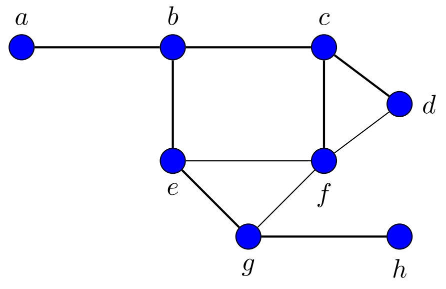
Still, we do not find any new vertex, so the queue remains $\langle g \rangle$. We then remove g.



Now, we find h, so the queue becomes $\langle h \rangle$. We visit h next.



We do not find any new vertex, so h is visited as well. The queue remains empty.



We have now finished the traversal. The thick edges are part of the spanning tree, which is shown below.

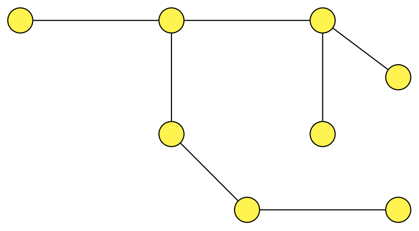


Figure 3.18: A spanning tree of the graph above, found using breadth-first search.

The following is the implementation of breadth first search using adjacency lists. It features all the code we had in order to implement adjacency lists as well.

```
void visit(Queue<Vertex> queue):
    // until the queue is empty
    while not queue.isEmpty:
        // remove a vertex from the queue
        Vertex vertex1 = queue.delete()
        // it will now be visited
        vertex1.visited = true

        // for a vertex that is incident to vertex1,
        for Vertex vertex2 in v.incidentNodes:
            int i = node.vertexIndex

            // if this vertex hasn't been visited, visit it
            if not vertex2.visited:
                queue.add(vertex2)

void bfs(Graph graph):
    // initialise the queue
    Queue<Vertex> queue = Queue()

    // if it has not been visited,
    for Vertex vertex in graph.vertices:
        if not vertex.visited:
            // add it to the queue and visit it
            queue.add(vertex)
            visit(queue)
```

We now consider the complexity of breadth-first search using adjacency lists. We visit every vertex and queue it exactly once. Also, the adjacency list is also traversed once. Therefore, the overall complexity is $O(|V| + |E|)$ using an adjacency list. If we instead used an adjacency matrix, the complexity is $O(|V|^2)$.

An application of breadth-first search to find the distance between two vertices. The distance is the number of edges in the shortest path between the two vertices. We can assign the distance from a vertex to itself as 0. Then, we can carry out a breadth-first search from the vertex, incrementing distance by 1 as we find a new vertex. We stop when we have reached the other vertex.

We illustrate this by an example. We are visiting the blue vertex from the red vertex.

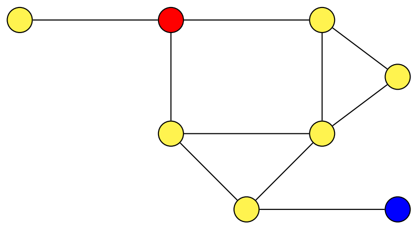
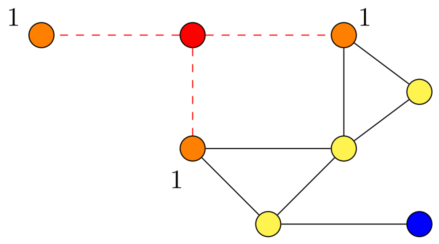
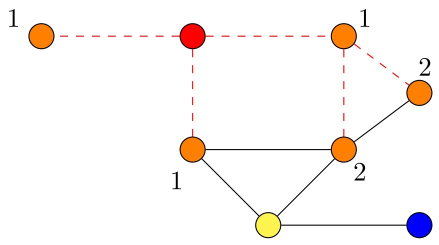


Figure 3.19: An undirected graph, where we want to find the distance between the red and the blue vertices.

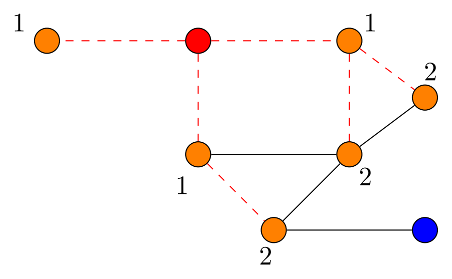
From the red edge, we can find three vertices. We assign these three vertices distance 1.



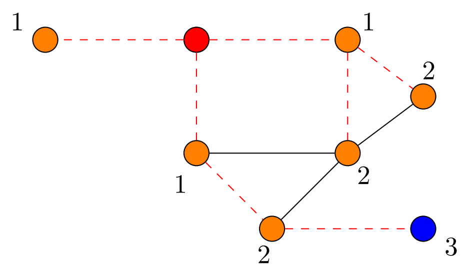
The dotted lines represent the shortest path we can take from a vertex to the original, red vertex. We now find vertices from the top right vertex (assuming it was the first element on the queue). These vertices have distance 2.



From the middle left vertex, we can assign the distance of the bottom left vertex as 2. We process this vertex before the middle right vertex since it was added to the queue later- that is why the vertex gets assigned distance 2 (correctly), and not 3.



We can finally assign distance to the blue vertex- it is 3.



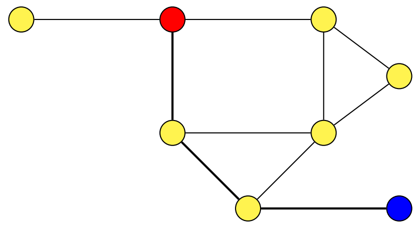


Figure 3.20: Shortest path between the red and the blue vertices, shown in bold.

The algorithm has now terminated. The shortest path from the red vertex to the blue vertex is shown below.

3.4 Weighted Graphs

In a weighted graph, each edge has an weight that is strictly positive. The graph can be both directed or undirected. The weight may represent length, cost, capacity, etc. If an edge is not part of the graph, we assume that its weight is infinity.

An example of a weighted graph is given below.

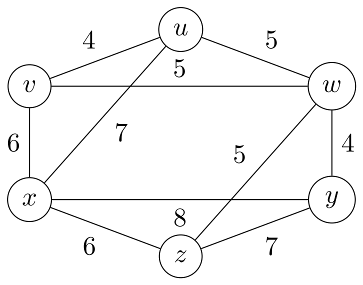


Figure 3.21: A weighted graph.

We can represent a weighted graph as both adjacency matrix and adjacency list. An adjacency matrix becomes the weight matrix, while an adjacency list stores the weight within the array, as a tuple corresponding to a particular edge.

So, the weighted graph above can be represented as follows.

	u	v	w	x	y	z
u	0	4	5	7	0	0
v	4	0	5	6	0	0
w	5	5	0	0	4	5
x	7	6	0	0	8	6
y	0	0	4	8	0	7
z	0	0	5	6	7	0

Table 3.5: The adjacency matrix representation of the weighted graph above

Moreover, the adjacency list is given below.

u	$[(v, 4), (w, 5), (x, 7)]$
v	$[(u, 4), (w, 5), (x, 6)]$
w	$[(u, 5), (v, 5), (y, 4), (z, 5)]$
x	$[(u, 7), (v, 6), (y, 8), (z, 6)]$
y	$[(w, 4), (x, 8), (z, 7)]$
z	$[(w, 5), (x, 6), (y, 7)]$

Table 3.6: The adjacency list representation for the weighted graph above

Shortest Path

Given a weighted graph and two vertices u and v, the shortest path algorithm computes the shortest path between u and v with respect to their weight. We say that the length of the path is the sum of the weights of its edges.

Dijkstra's algorithm finds the shortest path between a vertex u and all the others. It is based on maintaining a set S containing all the vertices for which the shortest path with u is currently known. Initially, S only contains the vertex u. Eventually, S contains all the vertices, i.e. the shortest path to all the vertices from u is known.

Each vertex v has a label $d(v)$ indicating the length of the shortest path between u and v, passing only through the vertices in S. S gets updated as the algorithm runs, so we change $d(v)$ as we add more elements. If there is no path, we set $d(v)$ to infinity. If v is in S, then $d(v)$ is the length of the shortest path between u and v. Moreover, if v is in S and w is not in S, then the length of the shortest path between u and w is at least that between u and v. This means that the weight of the edge between u and w is at least $d(v)$.

In Dijkstra's algorithm, we add to S the vertex v not in S such that $d(v)$ is minimum at each step. This ensures that at any point, the shortest path for vertices not in S to u have length greater than or equal to that for all vertices in S.

After adding a vertex v to S, we carry out *edge relaxation* operations. This means we update the length $d(w)$ for all vertices w that are still not in S. We previously have computed $d(w)$ based on all the vertices in S. Since we have added a new vertex to S, we need to ensure that if there is a shorter distance going from the new vertex, then this is reflected in the distance. In fact, the updated distance is given by

$$d(w) = \min(d(w), d(v) + wt(e)),$$

where $wt(e)$ is the weight of the edge e that is incident to the new vertex v.

Using this formula, we can construct the pseudocode for the Dijkstra's algorithm.

```
Map<Vertex, int> dijkstra(Graph graph, Vertex v1):
    // the set of vertices for which we know the optimal distance
    Set<Vertex> S = {v1}
    // the map which gives the corresponding weight for a vertex
    Map<Vertex, int> weights = []

    // initialise the weight for each vertex
    for Vertex v2 in graph.vertices:
        weights[v2] = graph.edge(v1, v2).weight

    // until S has all the vertices,
    while not S.containsAll(graph.vertices):
        // find the edge with smallest weight from a vertex
        // not in S to one in S
        Vertex v3 = _minVertex(weights, S)
        // add it to S
        S.add(v3)
        // relax all the edges not in S using the new vertex
        for Vertex v4 in graph.vertices:
            if v4 not in S and v4.incidentTo(v3):
                weights[v4] = min(weights[v4], weights[v3] +
                graph.edge(v3, v4).weight)

    return weights
```

The function `_minVertex` finds the vertex not in S such that the edge from the vertex to some vertex in S has the minimum weight.

We will now analyse the algorithm. Assume that we have n vertices and m edges. Using an unordered array to store the lengths, it takes $O(n)$ to initialise the lengths. Finding the minimum vertex is $O(n^2)$ overall- each time, it takes $O(n)$ to find the minimum, and we find it $n-1$ times. The relaxation is $O(m)$ overall- each edge is considered once, and the updating length takes $O(1)$. Therefore, the complexity of Dijkstra's algorithm is $O(n^2+m)$. Since the number of edges is at most $n(n-1)$, this is equivalent to $O(n^2)$.

Instead of using an unordered array, if we used a heap to store the lengths, it still takes $O(n)$ to initialise the lengths and create a heap. Find the minimum is now $O(n \log n)$- each time, it takes $O(\log n)$ to find the minimum, and we find it $n-1$ times. The relaxation is $O(m \log n)$ overall- each edge is considered once, and the updating takes $O(\log n)$. This updates a certain value in the heap (not necessarily the root). We would also need to keep track of each vertex distance in the heap, so care must be taken. Therefore, the complexity is $O((m+n)\log n)$. Assuming a graph has more edges than vertices, we can simplify the complexity to $O(m \log n)$. A graph with n vertices could have $O(n^2)$ edges, so the complexity may be $O(n^2 \log n)$- this is not better than $O(n^2)$ using an unordered array.

We shall now illustrate Dijkstra's algorithm with an example. The graph below is a directed weighted graph, and we aim to find the minimum distance from u.

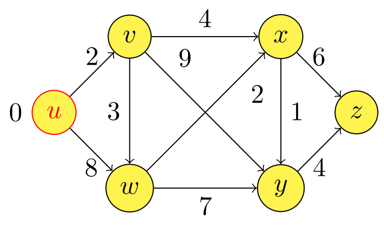
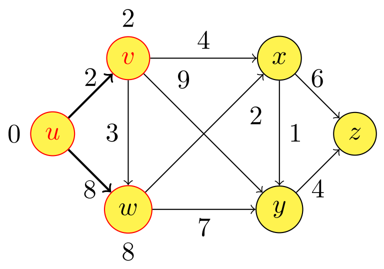
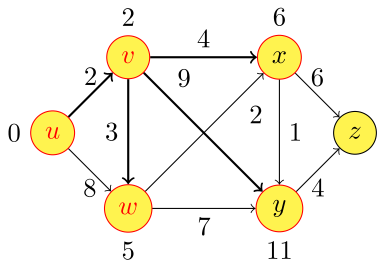


Figure 3.22: A weighted graph whose distances (from u) are to be found using Dijkstra's algorithm.

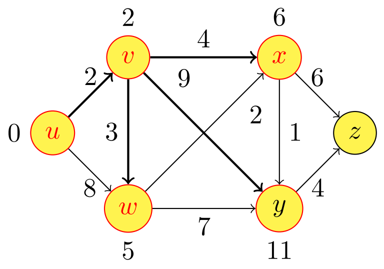
Vertices whose shortest path to u is known are labelled with red text. First, we set the distance of u from u to be 0. Any distance we do not know is ∞. Considering vertices going from $S = \{u\}$ to those not in S, we choose the edge connected with the smallest distance- it is v.



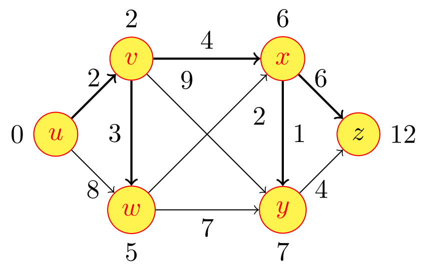
Vertices for which we know a path (not necessarily the shortest path) from u are shown with a red border. Thick edges represent the path from u to that edge, if known. At this point, $S = \{u, v\}$. We relax the edges with respect to the vertex v.



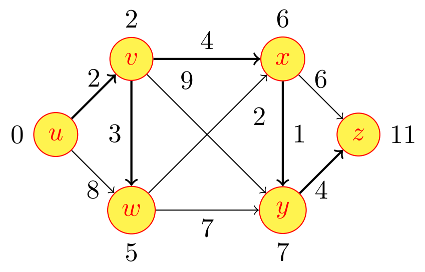
Now, we know the distance from u to x and y. Also, we have found a shorter distance from u to w, going through v. The smallest weight from an element not in S is w, so we add that to S. We then relax the vertices with respect to w.



We do not change any path since going through w only increases the weight. Also, we do not discover the distance to z. Here, the smallest-weighted vertex is x. So, we relax the vertices again with respect to x.



We can find a path with smaller weight from x to y, and we have finally found a non-infinite distance from u to z. Now, the smallest weighted vertex not in S is y. So, we add it to S and relax the edges with respect to y.



At this point, the only weight we can change is of z. It turns out that the weight going through y is smaller, so we change that weight. There is only one element not in S, so we add it into S via the edge y to z. All the edges are now in S, so the algorithm terminates. We have found the shortest distance and the path going from u to a particular vertex.

We illustrate another example for unweighted graphs. We will find the distance from u to every vertex.

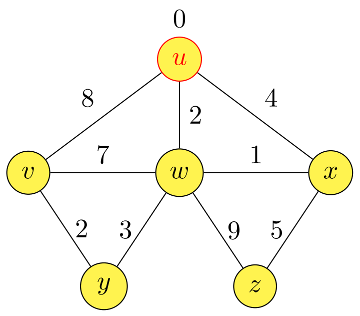
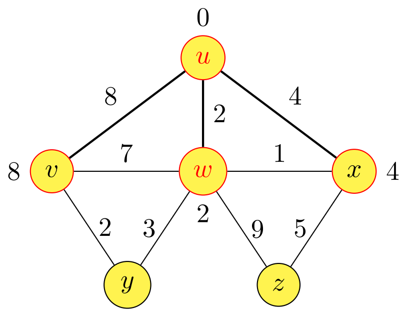
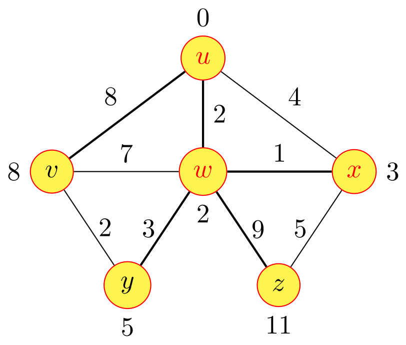


Figure 3.23: A weighted graph whose distances (from u) are to be found using Dijkstra's algorithm.

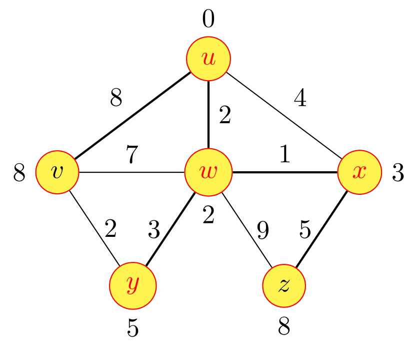
Vertices whose shortest path to u is known are labelled with red text. At the start, we only know the distance from u to u- it is 0. We relax the edges with respect to u to find the distance from u to the other vertices.



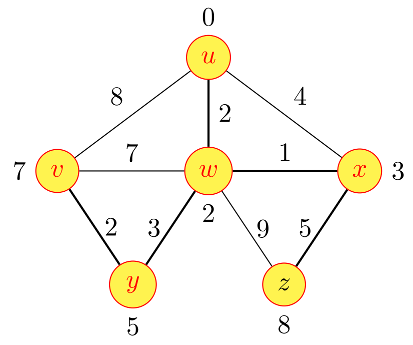
Vertices for which we know a path from u are shown with a red border. Thick edges represent the path from u to that edge, if known. The smallest-weighted vertex not in S is w. So, we add w to S, and relax with respect to it.



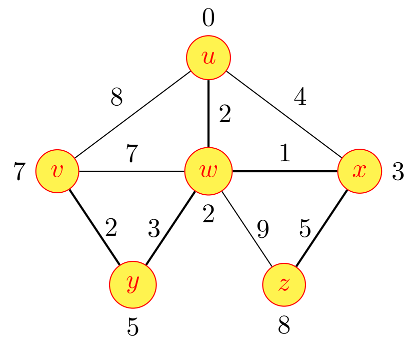
We now know the distance from u to y and z. Also, the distance from u to x is updated- it is now 3. We did not change the distance from u to v because the path through w is not better. Now, the smallest distance from an element not in S is x. So, we relax the edges with respect to x.



Following the relaxation, we have found a shorter path to z. Now, the smallest distance of an element not in S is from y. So, we relax the edges with respect to y.



We have now found a shorter path from u to v. The next vertex to be added to S is v. So, we relax the edges with respect to v.



At this point, the only weight that could change is of z. This does not change by the relaxation. Now, we add the final vertex z into S. We have now added all the vertices into S, so we have found the minimum distance from u to all the vertices.

3.5 Minimum Spanning Trees

A spanning tree is a subgraph (a subset of edges) which is both a tree and spans every vertex. It is obtained from a connected graph by deleting edges. The weight of a spanning tree is the sum of the weights of its edges.

For example, in the weighted graph below, we can remove the red edges and one of the three dashed edges to get a spanning tree.

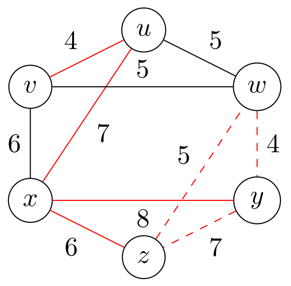


Figure 3.24: A weighted graph whose edges are to be deleted to form a spanning tree. The red edges can be removed, and so can one of the three dashed edges.

Two of the possible spanning trees are given below.

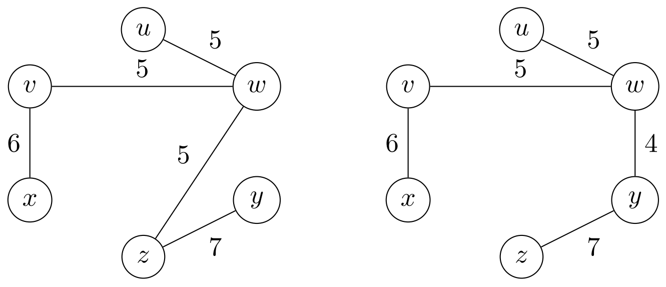


Figure 3.25: Two distinct spanning trees of the graph above.

The weight of the left tree is 28, while the weight of the right tree is 27. In this section, we will consider an algorithm that computes the minimum weight spanning tree. For the graph above, a minimum spanning tree is given below.

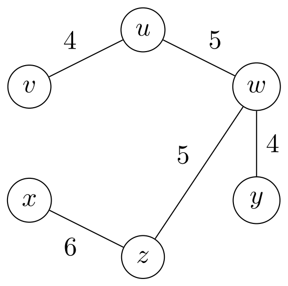


Figure 3.26: A minimum spanning tree of the graph above.

The weight of the graph above is 24. There may not be a unique spanning tree, but all the minimum spanning weights will have the same weight.

Computing the minimum weight is an example of a problem in combinatorial optimisation. We find the 'best' way of doing something among a (large) nbumber of candidates. It can always be solved by exhaustive search, at least in theory. In practice, however, this may be infeasible since it is typically an exponential-time algorithm. For example, the clique of size n K_n has n^{n-2} spanning trees. A much more efficient algorithm may be possible, and this is true in the case of minimum weight spanning trees.

Prim-Jarnik

The Prim-Jarnik algorithm is a minimum spanning tree algorithm. It is an example of a greedy algorithm, i.e. it makes a sequence of decisions that are locally optimal to come up with a globally optimal solution.

The pseudocode for the algorithm is given below.

```
List<Edge> primJarnik(Graph graph):
    // the tree vertices
    List<Vertex> tv = graph.vertices[:1]
    // the non-tree vertices
    List<Vertex> ntv = graph.vertices[1:]
    // the edges in the minimum spanning tree
    List<Edge> spanTree = []

    // until there are no more non-tree vertices
    while ntv.length > 0:
        // find the edge from a non-tree vertex to a
        // tree vertex with smallest weight
         Edge edge = _minEdge(tv, ntv)
         // make it a tree vertex
         tv.add(ntv.remove(edge.from))
         // add it to the minimum spanning tree list
         spanTree.add(edge)

    return spanTree
```

The function `_minEdge` finds the non-tree vertex such that the edge from the vertex to a tree vertex has the minimum weight.

We now analyse the algorithm. The vertices can be added to the list of tree vertices `tv` or the list of non-tree vertices `ntv` in $O(n)$ time overall. The outer loop is executed $n-1$ times. Initially, we have $n-1$ ntv and each iteration turns one `ntv` into a `tv`. The inner loop checks all the edges from a tree-vertex to a non-tree vertex, and there can be $O(n^2)$ of these. Overall, the algorithm is $O(n^3)$.

We illustrate the algorithm with an example. Consider the following graph.

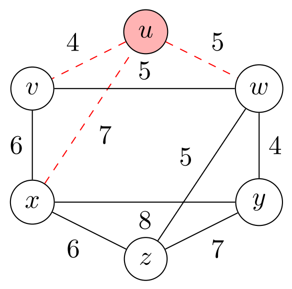
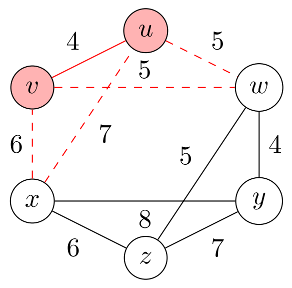
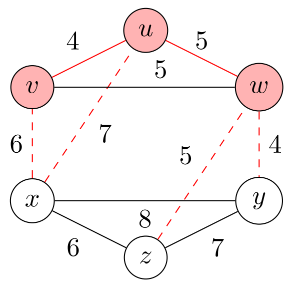


Figure 3.27: A weighted graph to which we apply the Prim-Jarnik algorithm.

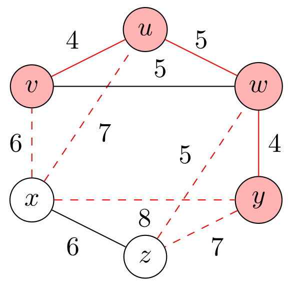
The orange vertices represent those part of the MST, i.e. they are tree vertices. The dashed edges are the ones that are being considered to be added, i.e. they connect a non-tree vertex to a tree vertex. We take the edge with the smallest weight, so we add the edge connecting u and v to the spaning tree.



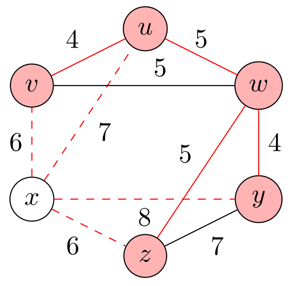
The edges in red are part of the MST. Now, there are two edges with the smallest weight 5- we choose to add the one connecting u and w.



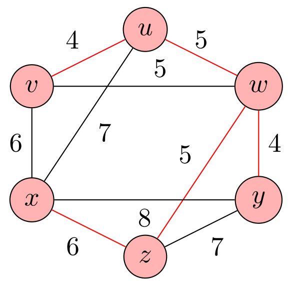
Now, we choose the edge with weight 4- the one that connects w to y.



Next, we add the edge connecting w and z.



Finally, we connect u and x.



Now, every vertex is a tree vertex. We have the following spanning tree.

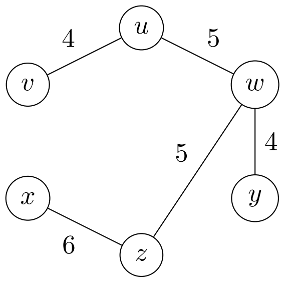


Figure 3.28: A minimum spanning tree of the graph above with weight 24.

Dijkstra’s Refinement

We saw before that Prim-Jarnik algorithm was $O(n^3)$. Dijkstra’s refinment to Prim-Jarnik’s algorithm preprocesses the vertices so that the internal loop of $O(n^2)$ becomes linear.

This version introduces an attribute `bestTV` for each non-tree vertex. This variable is set to the tree vertex such that the weight of the edge connecting the tree and the non-tree vertex is minimised. It is possible that there are no edges that connect the non-tree vertex to a tree vertex. In that case, we choose an arbitrary edge and set the weight to be infinite.

The pseudocode for Dijkstra’s refinement is shown below.

```
List<Edge> dijkstraRefinement(Graph graph):
    // the tree vertices
    List<Vertex> tv = graph.vertices[:1]
    // the non-tree vertices
    List<Vertex> ntv = graph.vertices[1:]
    // the edges in the minimum spanning tree
    List<Edge> spanTree = []

    // initialise the best tree-vertex -> it is the first vertex
    for Vertex v1 in ntv:
        v1.bestTV = graph.vertices[0]

    // until there are no more non-tree vertices,
    while ntv.length > 0:
        // find the edge from a non-tree vertex to a
        // tree vertex with smallest weight
        Edge edge = _minEdge(ntv)
        // make it a tree vertex
         tv.add(tv.remove(edge.from))
         // add it to the minimum spanning tree list
         spanTree.add(edge)

         // update the best tree vertices using the new vertex
         for Vertex v2 in ntv:
             if graph.weight(v2, v2.bestTV) > graph.weight(v2,
             edge.from):
                 v2.bestTV = edge.from

    return spanTree
```

We now analyse the algorithm formally. The initialisation takes $O(n)$ time. The outer loop runs $n-1$ times. The first part of the inner loop is $O(n)$- we find the minimal `ntv`. Also, the second part of the inner loop is $O(n)$- we just loop through the non-tree vertices. Therefore, the algorithm is $O(n^2)$.

We shall now illustrate the algorithm. We will use the same graph as above, but start with vertex z.

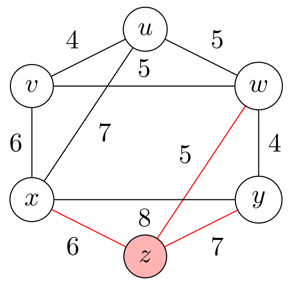
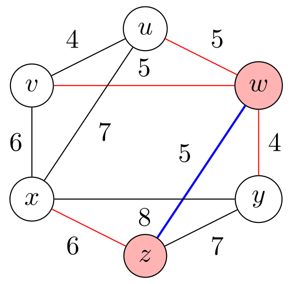
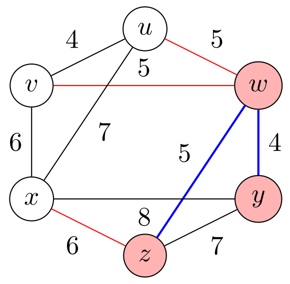


Figure 3.29: A weighted graph to which we apply the Dijkstra's refinement.

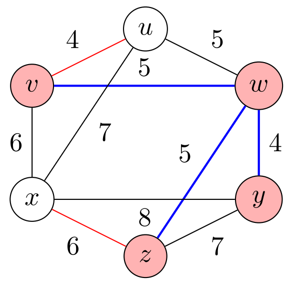
The orange vertices represent those part of the MST. The red edges are the best edges connecting a non-tree vertex to a tree vertex (if it exists). Although the best vertex of v is z, its weight is infinite, since there is no edge connecting them. We take the minimal edge from the three- it is between z and w.



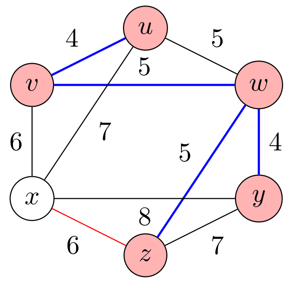
The blue edges are part of the MST. We have found a better edge from y to a tree vertex. Also, we have more vertices that have a non-infinite weight. The next edge we add is from w to y.



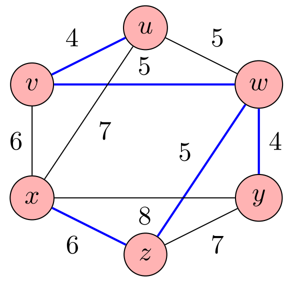
We update the best tree vertex for each of the non-tree vertices. The next edge we choose is from w to v.



We continue on and add the edge connecting u and v.



Finally, we add the edge connecting x and z.



Now, all the edges are tree vertices. The algorithm now terminates, and we have found a minimum spanning tree.

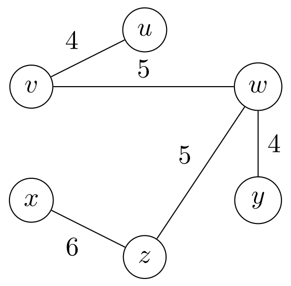


Figure 3.30: A minimum spanning tree of the graph above with weight 24.

This is a different minimum spanning tree to the one we saw above, but they both have weight 24.

3.6 Topological Ordering

A direct acyclic graph (DAG) is a directed graph that is not cyclic. A topological order on a DAG is a labelling of the vertices $1, \ldots, n$ such that if $(u, v) \in E$, then label$(u) <$ label(v).

A directed graph D has a topological order if and only if it is a DAG. Clearly, it is impossible if D has a cycle. A source is a vertex of in-degree 0, while a sink is a vertex of out-degree 0. A DAG has at least one source and at least one sink. If a DAG has no source or sink, then we can build a cycle. If there is no source or sink, we can always keep adding vertices to the start or the end of a path. But, since there are a finite number of vertices, we must encounter at least one vertex twice. In that case, we have a cycle. Therefore, the graph isn't acyclic. This forms a basis of a topological sorting algorithm.

The following is an example of a DAG.

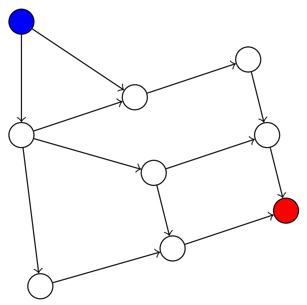
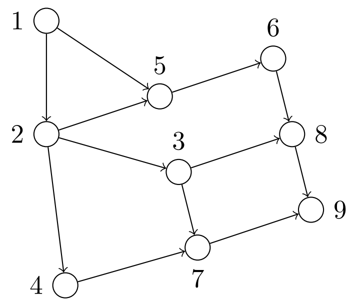


Figure 3.31: A directed acyclic graph. The source is in blue, and the sink is in red.

In general, a DAG can have more than one sink and source vertices. The following is a topological ordering on the DAG.



We will now go over the topological ordering algorithm. It adds two integer attributes to each vertex in a graph- a label in the topological order, and a count that initially equals the in-degree of the vertex. It gets decremented every time we label a vertex. At a given point, it equals the number of incoming edges from vertices that have not been labelled.

We require the label of a vertex to be greater than that of all its incoming vertices. So, if all vertices have incoming edges have been labelled, we can just label this vertex with a greater value. When the attribute becomes zero, we add the vertex to a queue to be labelled. Any source vertex can be added to the queue immediately.

The pseudocode for topological sorting algorithm is given below.

```
List<Vertex> topSort(Graph graph):
    // initialise the count
    for Vertex v1 in graph.vertices:
        v1.count = v1.inDegree

    // the queue containing all the source vertices
    Queue<Vertex> sourceQueue = <>

    // add to the source queue all the source vertices
    for Vertex v2 in graph.vertices:
        if v2.count == 0:
            sourceQueue.add(v2)

    // the vertices in their topologically sorted order
    List<Vertex> vertices = []

    // until the source queue is empty,
    while not sourceQueue.isEmpty:
        // remove the first element
        Vertex v3 = sourceQueue.delete()
        label[v3] = nextLabel ++

        // lower the count for all the incident vertices
        for Vertex v4 in graph.incidentVertices:
            v4.count--

            // if it is now a sink, add it to the queue
            if v4.count == 0:
                sourceQueue.add(v4)
```

We illustrate the algorithm on the DAG below.

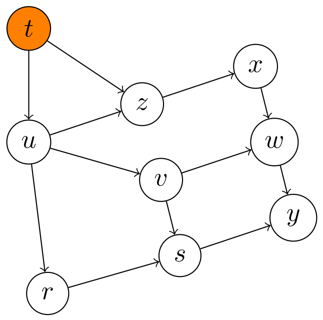
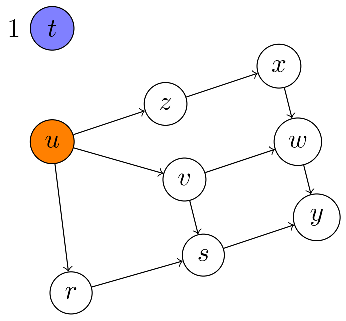
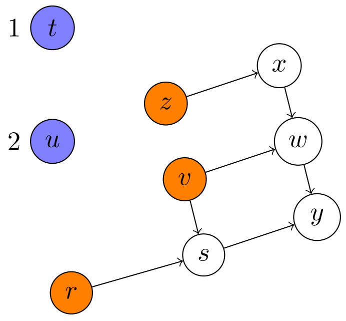


Figure 3.32: A DAG to be topologically sorted.

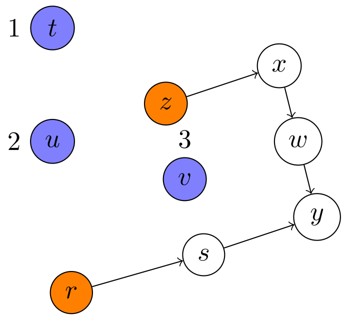
The queued vertices are in red. At the start, the source vertex t is in the queue. We delete the vertex from the queue and lower the count of the adjacent vertices. The vertex s is labelled 1.



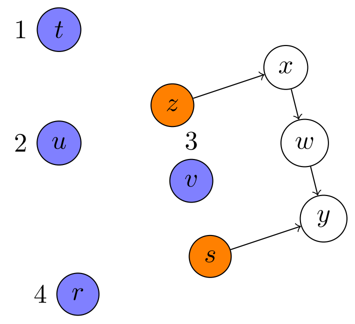
The labelled vertices are in blue. At this point, only the vertex u has count 0. So, we remove it from the queue, label it 2 and decrement the count of further vertices.



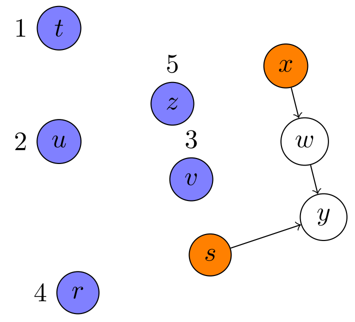
Now, the queue is $\langle v, r, z \rangle$. The order we add these three elements may give rise to a different topological ordering. In this case, we first delete v and label it 3.



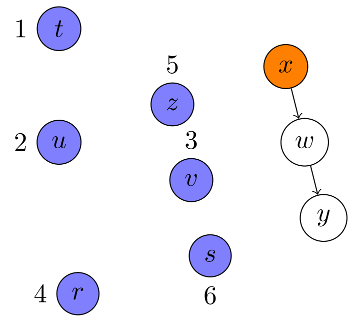
Next, we remove the vertex r and label it 4.



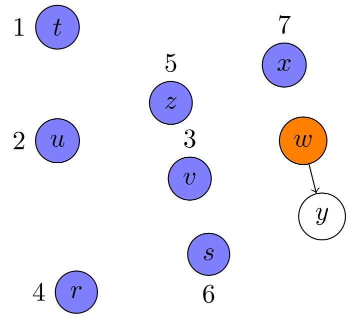
At this point, the queue is $\langle z, s\rangle$. Since z was encountered earlier, it is important that we use the queue data type. The vertex s must be labelled a higher number compared to z for it to be a valid topological ordering. So, we remove z and label it 5.



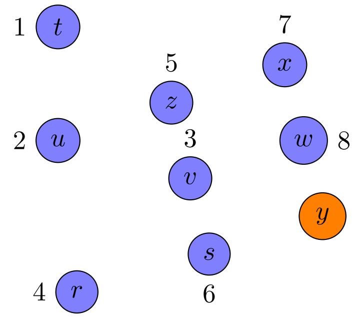
Now, the queue is $\langle s, x\rangle$. So, we remove s and label it 6.



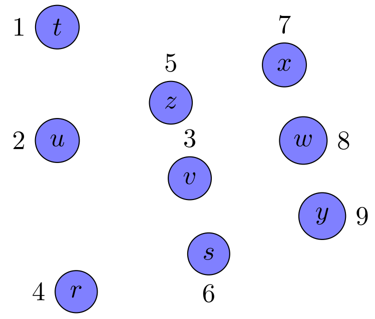
Then, the queue is $\langle x\rangle$. We remove it and label it 7.



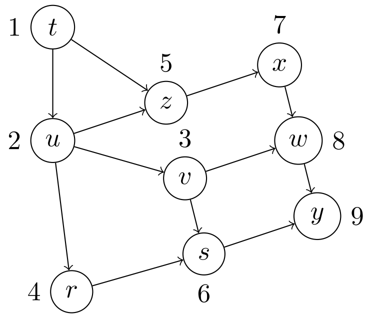
Next, the queue is $\langle w \rangle$. We remove it and label it 8.



The queue is now $\langle y \rangle$. We remove it and label it 9.



There are no further edges to label, so we have sorted the DAG topologically.



In this algorithm, a vertex is given a label only when the number of incoming edges from unlabelled vertices is zero. All the predecessor vertices must already be labelled with smaller numbers. It is dependent on the FIFO property of the queue.

We now analyse the algorithm. First, assume we are using the adjacency list representation. To find the in-degree of each vertex, it takes $O(|V| + |E|)$- we set the count for each vertex and we encounter an edge precisely once (since the graph is directed). The main loop is executed $|V|$ times. Each time, one adjacency list is scanned, that of the vertex being labelled. So, the same list is never scanned twice. Every list is scanned again once. Overall, the algorithm is $O(|V| + |E|)$.

Instead, if we used adjacency matrix representation, finding the in-degree of each vertex is $O(|V|^2)$. The main loop is executed $|V|$ times. Within the loop, one row is scanned, so the runtime is $O(|V|)$. In that case, the overall algorithm is $O(|V|^2)$.

An application of topological sorting is deadlock detection. It determines whether a directed graph contains a cycle. We can adapt the topological sorting algorithm and check whether the source queue becomes empty before all the vertices get labelled- this indicates that there is a cycle. On the other hand, if all the vertices can be labelled, then the graph is acyclic.

Instead, we could also adapt the depth-first search algorithm. When a vertex `u` is visited, we check whether there is an edge from `u` to a vertex `v` which is on the current path from the current starting vertex. The existence of such a vertex indicates that we have a cycle.

CHAPTER 4

INTRODUCTION TO NP-COMPLETENESS

4.1 Polynomial and exponential algorithms

We have seen algorithms for a wide range of problems so far, giving us a spectrum of worst-case complexity functions. For example, searching a sorted list has $O(\log n)$ runtime. We can find the maximum value in an unsorted list in $O(n)$ time. Comparison based sorting are $O(n \log n)$. The distance between two strings has runtime $O(n^2)$ (where both strings have length n). We can find the shortest path in $O(n^2)$ time for a weighted graph with n vertices. In each case, the algorithm associated is a polynomial-time algorithm, i.e. it is $O(n^c)$ for some constant c.

Given an undirected graph G, we can ask whether G admits an Euler cycle. An Eulerian cycle is a cycle that traverses each edge exactly once. A path is labelled below- the path goes from vertex 1 to 8.

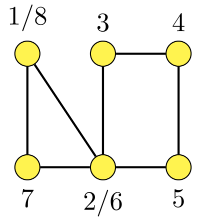


Figure 4.1: A traversal of the graph that results in an Eulerian cycle.

We take an edge precisely once (but have visited two vertices twice). There is a theorem that states that a connected undirected graph has an Euler cycle if and only if each vertex has an even degree. Therefore, we can test whether a graph has an Eulerian cycle by checking the in-degree of each vertex. So, if we represent the graph as an adjacency matrix, the algorithm is $O(|V|^2)$, while if we use an adjacency list, it is $O(|V| + |E|)$.

Given an undirected graph G, we can also ask whether G admits a Hamiltonian cycle. A Hamiltonian cycle is a cycle that visits each vertex precisely once. For example, the graph below has a Hamiltonian cycle shown in red.

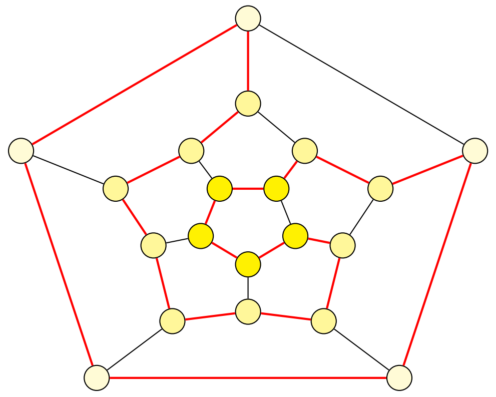


Figure 4.2: A traversal of the graph that results in a Hamiltonian cycle. The traversal is shown in red.

Superficially, the problem seems similar to the Euler cycle problem. However, in an algorithmic sense, they are very different. In fact, nobody has found a polynomial-time algorithm for Hamiltonian cycle. Also, nobody has shown that there does not exist a polynomial-time algorithm to find a Hamiltonian cycle.

One way of finding a Hamiltonian cycle is to generate all the permutations of the vertices. We then check to see if it is a cycle, i.e. the corresponding edges are present. If we have n vertices, then there are $n!$ permutations. For each permutation σ, it takes $O(n^2)$ to check whether σ is a Hamiltonian cycle (assuming G is represented by adjacency lists). In the worst case, to check an edge is present, we have to traverse the adjacency list of length $n-1$, and there are n edges to check. So, the worst-case number of operations in the brute force approach to find a Hamiltonian cycle using adjacency lists is $O(n^2 n!)$. This is an example of an exponential algorithm.

The table below shows the running time of algorithms with various complexities.

	40	50	60	70
n	.00003 secs	.0004 secs	.0005 secs	.0006 secs
n^2	.0009 secs	.0016 secs	.0025 secs	.0036 secs
n^3	.027 secs	.064 secs	.125 secs	.216 secs
n^5	24.3 secs	1.7 mins	5.2 mins	13.0 mins
2^n	17.9 mins	12.7 days	35.7 years	366 cents
3^n	6.5 years	2855 cents	2×10^8 cents	1.3×10^{13} cents
$n!$	8.4×10^{16} cents	2.6×10^{32} cents	9.6×10^{48} cents	2.6×10^{66} cents

Table 4.1: The running time of algorithms with various complexities (assuming 10^9 operations per second).

Clearly, as n grows, the distinction between polynomial and exponential time algorithms becomes dramatic.

This behaviour still applies even with increases in computing power. Assume that for each time complexity, we have a size N of the largest instance solvable in an hour on a normal computer. The table below shows how the complexity changes as the computer gets faster.

	current	100 times faster	1000 times faster
n	N_1	$100 \cdot N_1$	$1000 \cdot N_1$
n^2	N_2	$10 \cdot N_2$	$31.6 \cdot N_2$
n^3	N_3	$4.64 \cdot N_3$	$10 \cdot N_3$
n^5	N_4	$2.5 \cdot N_4$	$3.98 \cdot N_4$
2^n	N_5	$N_5 + 6.64$	$N_5 + 9.97$
3^n	N_6	$N_6 + 4.19$	$N_6 + 6.29$
$n!$	N_7	$\leq N_7 + 1$	$\leq N_7 + 1$

Table 4.2: Maximum improvement in computation with increasing computing power.

Again, we can see that for polynomial-time algorithms, there is a significant improvement. However, for the exponential algorithms, there is little to no increase.

In summary, exponential-time algorithms are ‘bad’ in general. If we increase the processor speed, there is no significant change in this slow behaviour when the input size is large. On the other hand, polynomial-time algorithms are ‘good’ in general. Often, polynomial-time algorithms require some insight, while exponential-time algorithms are variations on exhaustive search. A problem is polynomial-time solvable if it admits a polynomial-time algorithm.

4.2 NP-Complete Problems

There is a set of problems, called NP-complete, for which no polynomial-time algorithm is known. However, if one of them is solvable in polynomial time, then they all are. On the other hand, we also do not know whether they cannot be solved in polynomial time. But, if one of them is not solvable in polynomial time, then they all aren't solvable.

There are two different causes of intractability. Either the polynomial time is not sufficient in order to discover a solution, or the solution itself is so large that exponential time is needed to output it. We will focus on the first case. There are intractability proofs for case 1. In fact, some problems have been shown to be undecidable, i.e. there cannot be an algorithm that could solve the problem. Also, some decidable problems have been shown to be intractable, i.e. they can be solved but not in polynomial time.

The roadblock problem is a decidable problem that is not intractable. There are two players- A and B, along with a network of roads, comprising intersections connected by roads. Each road is coloured either black, blue or green. Some intersections are marked either 'A wins' or 'B wins'. A player has a fleet of cars at intersections, with at most one per intersection. Player A begins, and subsequently players make moves in turn. They move one of their cars on one or more roads of the same colour. A car may not stop at or pass over an intersection which already has a car. The problem is to decide, for a starting configuration, whether A can win, regardless of what moves B makes.

In summary, there are two classes of problems- polynomial-time solvable problems and intractable problems. We do not know where NP complete problems belong.

A problem is usually characterised by parameters. Typically, there are infinitely many instances for a given problem. A problem instance is created by giving these parameters values. For example, the Hamiltonian cycle is NP-complete. The problem is:

- **Name**: Hamiltonian Cycle (HC)
- **Instance**: a graph G
- **Question**: does G contain a cycle that visits each vertex exactly once?

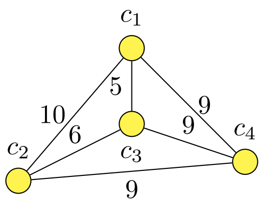
This is an example of a decision problem- the answer is `true` or `false`. So, every instance is either a 'yes-instance' or a 'no-instance'.

Another NP-complete problem is the Travelling Salesman, shown below:

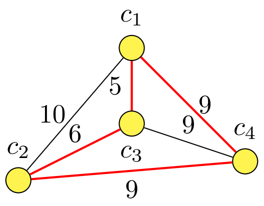
- **Name**: Travelling Salesman Decision Problem (TSDP)
- **Instance**: a set of n cities and integer distance $d(i,j)$ between each pair of cities i,j, and a target integer K
- **Question**: is there a permutation $p_1, p_2, \ldots, p_{n-1}, p_n$ of the cities such that
$$d(p_1,p_2) + d(p_2,p_3) + \cdots + d(p_{n-1},p_n), d(p_n,p_1) \leq K.$$

That is, can we find a path to travel from one city back to itself (going through all the cities) such that the total distance travelled is less than K- such a cycle is called a travelling salesman tour.

For example, consider the following graph representation of the cities.



There is a travelling salesman tour of length 29 when we traverse through the following edges.

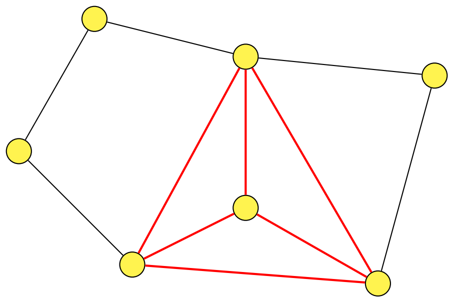


However, there is no tour of length less than 29. It has been shown that the TSDP is NP-complete.

Another NP-complete problem is the clique problem:

- **Name**: Clique Problem (CP)
- **Instance**: a graph G and a target integer K
- **Question**: does G contain a clique of size K? That is, does G have a set of K vertices for which there is an edge between all the pairs.

For example, the graph below has a clique of size 4, as shown in red.

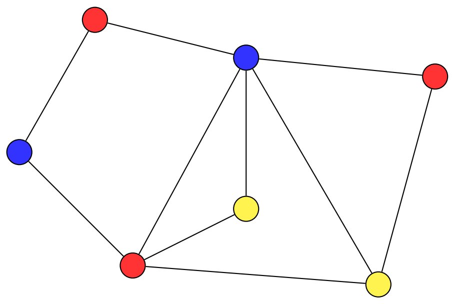


However, the graph does not have a clique of size 5.

The Graph Colouring Problem is also NP-complete. The problem is:

- **Name**: Graph Colouring Problem (GCP)
- **Instance**: a graph G and a target integer K
- **Question**: can one of the K colours be attached to each vertex of G such that the adjacent vertices always have different colours?

For example, the graph below can be coloured using 3 colours, as shown below.



However, there is no colouring using 2 colours.

Finally, we look at the satisfiability problem:

- **Name**: Satisfiability (SAT)
- **Instance**: Boolean expression B in conjunctive normal form (CNF). A CNF is of the form $C_1 \wedge C_2 \wedge \cdots \wedge C_n$, where each C_i is a clause. A clause is of the form $l_1 \vee l_2 \vee \cdots \vee l_m$, where each l_j is a literal. Finally, a literal is a variable x (which can either be `true` or `false`) and its negation $\neg x$.
- **Question**: Is B satisfiable? That is, can we assign the variables `true` or `false` so that the expression B is true.

For example, the following boolean expression is in CNF:

$$(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3 \vee \neg x_4) \wedge (\neg x_2 \vee x_4) \wedge (x_2 \vee \neg x_3 \vee x_4).$$

Also, the expression above is satisfiable if we set $x_1 = \texttt{true}, x_2 = \texttt{false}, x_3 = \texttt{true}, x_4 = \texttt{true}$. The satisfiability problem is NP-complete.

The problems we have looked at have been decision problems- they return `true` or `false`. However, we are typically interested in optimisation problem, where we actually have the maximum or the minimum value. For example, the travelling salesman optimisation problem (TSOP) finds the minimum length of a tour. We also have search problems, where we find some appropriate optimal structure. For example, the travelling salesman search problem (TSSP) finds the minimum length tour.

NP-completeness primarily deals with decision problems, but corresponding to each instance of an optimisation or a search problem is a family of instances of a decision problem by setting the 'target' values. Almost invariably, an optimisation or a search problem can be solved in polynomial time if and only if the corresponding decision problem can.

4.3 P and NP

The class P is the set of all decision problems that can be solved in polynomial time. There are many problems in P, e.g.

- Is there a path of length $\leq K$ from vertex u to vertex v in a graph G?
- Is there a spanning tree of weight $\leq K$ in a graph G?
- Is a graph G bipartile?
- Is a graph G connected?
- Does a directed graph D contain a cycle?
- Does a text t contain an occurrence of a string s?
- For two strings s and t, is their distance $d(s,t) \leq K$?

P is often extended to include search and optimisation problems.

The class NP is the set of decision problems that can be solved in non-deterministic polynomial time. A non-deterministic algorithm can make non-deterministic choices, so it can give different answers depending on when we run it. Hence, it is apparently more powerful than a normal deterministic algorithm.

Clearly, P is contained within NP- a deterministic algorithm is just a special case of a non-deterministic one. But, the containment may not be strict. Currently, there is no problem known to be in NP but not in P.

A non-deterministic algorithm (NDA) have an extra operation- they can make a non-deterministic choice. A NDA has many possible executions depending on which values were chosen during the process. We say that NDA 'solves' a decision problem Π if for a yes-instance I of Π, there is some execution that returns `true`, and for any no-instance I of Π, there is no execution that returns `true`. Moreover, the NDA 'solves' a decision problem Π in polynomial time if for a yes-instance I of Π, there is some execution that returns `true` which uses a number of steps bounded by a polynomial in the input, and for a no-instance, there is no execution that returns `true`.

Clearly, NDAs are not useful in practice since they do not always return the right answer. However, they are a useful mathematical concept for defining the classes of NP and NP-complete problems.

We will now provide a NDA that 'solves' the graph colouring problem.

```
bool graphColouring(Graph graph, int k):
    // choose one of the k colours for each vertex
    for Vertex v in graph:
        v.colour = Colour.random(k)

    // check whether vertices with an edge are coloured different
    for Edge edge in graph.edges:
        if edge.to.colour == edge.from.colour:
            return false
    return true
```

In line 4, we make a non-deterministic choice for the colour of a vertex from the set of k colours. We then check whether the coloured graph satisfies the graph colouring problem. The operations take polynomial time for every execution of the algorithm.

In general, a NDA can be viewed as having a guessing stage (which is non-deterministic), and a checking stage (which is determinstic). If the problem is in NP, then we need the checking stage to run in polynomial time.

4.4 Polynomial-time reductions

A polynomial-time reduction (PTR) is a mapping f from a decision problem Π_1 to a decision problem Π_2 such that for every instance I_1 of Π_1, the instance $f(I_1)$ of Π_2 can be constructed in polynomial time. Moreover, $f(I_1)$ is a yes-instance of Π_2 if and only if I_1 is a yes-instance of Π_1. We denote this as $\Pi_1 \propto \Pi_2$.

Polynomial-time reductions are transitive. That is, if we have decision problems Π_1, Π_2, Π_3 such that $\Pi_1 \propto \Pi_2$ and $\Pi_2 \propto \Pi_3$, then $\Pi_1 \propto \Pi_3$. By definition, there exist PTRs f from Π_1 to Π_2 and g from Π_2 to Π_3. Now, for any instance I_1 of Π_1, we can create an instance $f(I_1)$ of Π_2 in polynomial time. By definition, $f(I_1)$ has the same answer as I_1. Moreover, g is a PTR as well, so $g(f(I_1))$ is an instance of Π_3 that we can construct from $f(I_1)$ in polynomial time. By definition, $g(f(I_1))$ has the same answer as $f(I_1)$. Therefore, the composition $g \circ f$ is a polynomial time reduction from Π_1 to Π_3 since $(g \circ f)(I_1)$ has the same answer as I_1.

If we have $\Pi_1 \propto \Pi_2$ and $\Pi_2 \in P$, then $\Pi_1 \in P$ as well. To solve an instance of Π_1, we can reduce it to an instance of Π_2 in polynomial-time. Then, we can solve Π_2 in polynomial-time and return the result. The entire process takes polynomial time, so $\Pi_1 \in P$.

Roughly speaking, $\Pi_1 \propto \Pi_2$ means that Π_1 is 'no harder' than Π_2. That is, if we can solve Π_2, then we can solve Π_1 without much more effort- we just need to perform a polynomial time reduction. It is also possible that Π_2 is harder to solve than Π_1. We may only map the instances of Π_1 into those in Π_2 that are 'easy' to solve.

We can reduce the Hamiltonian cycle problem to the travelling salesman problem. For instance, assume that we are given the following instance of the Hamiltonian cycle problem.

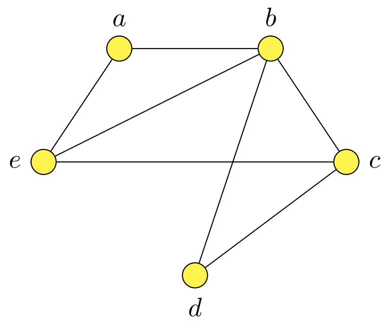


Figure 4.3: An instance of the Hamiltonian Cycle problem.

We will construct a TSDP corresponding to this graph. So, let the vertices be the cities in the problem. Moreover, for two cities x, y, we set $d(x, y) = 1$ if there is an edge $\{x, y\}$ in the graph above, and $d(x, y) = 2$ otherwise. Then, we can set K to be the number of vertices. This gives us the following graph.

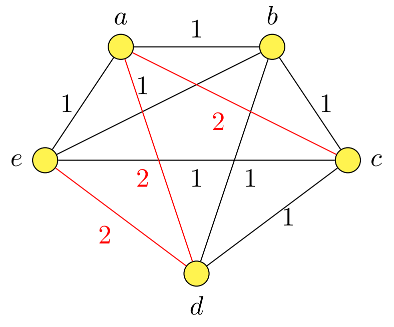


Figure 4.4: An instance of the Travelling Salesman Problem

All these operations can be performed in polynomial-time. Moreover, the graph has a travelling salesman tour of length $\leq K$ if and only if the original graph has a Hamiltonian cycle. The tour includes $|V|$ edges, so we cannot take any edges with weight 2. So, if the travelling salesman problem is in P, then the Hamiltonian cycle problem is in P as well. Equivalently, if we can show that the Hamiltonian cycle problem is not in P, then the travelling salesman problem is not in P either.

We now define NP-completeness. We say that a decision problem Π is NP-complete if Π is in NP, and for every problem Π' in NP, the problem Π' reduces to Π in polynomial-time. If Π is NP-complete and Π is in P as well, then we have P = NP. Every problem in NP can be solved in polynomial time by reduction to Π. Supposing P $\neq$ NP, if Π is NP-complete, then Π is not in P.

To prove that a problem is NP-complete, it is not feasible to describe a reduction from every problem in NP. However, if we know that a problem Π_1 is NP-complete, then to show that another problem Π_2 is NP-complete, we need to show that Π_2 is in NP and that there exists a polynomial-time reduction from Π_1 to Π_2. Since Π_1 is NP-complete, we know that for any problem Π in NP, we have $\Pi \propto \Pi_1$. We also know that $\Pi_1 \propto \Pi_2$. Since $\propto$ is transitive, it follows that $\Pi \propto \Pi_2$. Therefore, Π_2 is NP-complete.

We know that the satisfiability problem (SAT) is NP-complete. The proof consists of a generic polynomial-time reduction to SAT from an abstract definition of a general problem in the class NP. The generic reduction could be instantiated to give an actual reduction for each individual NP problem. So, to prove that a decision problem Π is NP-complete, we need to show that Π is in NP and that there is some polynomial-time that reduces SAT to Π.

We will use this technique to show that the Clique problem (CP) is NP-complete. To prove CP is NP-complete, we show that CP is in NP and that there exists a polynomial-time reduction from SAT to CP. The following is an NDA that solves the CP.

```
bool clique(Graph graph, int k):
    List<Vertex> vertices = []
    // choose k vertices from the graph
    for int i in range(0, k):
        vertices.add(graph.vertices.random())

    // check every distinct pair of vertices has an edge
    for Vertex v1 in vertices:
        for Vertex v2 in vertices:
            if v1 != v2 and not graph.hasEdge(v1, v2):
                return false
    return true
```

Therefore, the problem is in NP.

Next, we show that there is a polynomial-time reduction from SAT to CP. So, assume that we are given an instance B of SAT. We construct an instance (G, K) of CP. We set K to be the number of clauses of B. The vertices of G are (l, C), where l is a literal in clause C. We add an edge $\{(l, C), (m, D)\}$ if $l \neq \neg m$ and $C \neq D$. That is, we add an edge if distinct literals from different clauses can be satisfied simultaneously. This is a polynomial-time algorithm since the construction takes $O(n^2)$- in the worst case, we need to compare every literal with every other literal when constructing the edges.

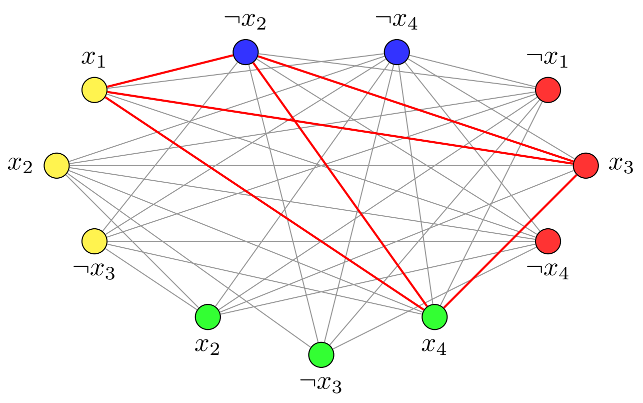
Now, if B has a satisfying assignment, then if we choose a `true` literal in each clause, the corresponding vertices form a clique of size K in G. On the other hand, if G has a clique of size K, then assigning each literal associated with a vertex in the clique to be `true` yields a satisfying assignment for B.

We only have edges between literals in distinct clauses, so if we have a clique of size K, then it must include precisely one literal from each clause. Moreover, we can satisfy all the literals in the clique simultaneously, since a clause is a disjunction of the literals and we can satisfy one of the literals. Moreover, this satisfies B since B is the conjunction of the clauses, and we satisfy all the clauses.

We will illustrate this with an example. Assume that we are given the following CNF.

$$(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_3 \vee \neg x_4) \wedge (\neg x_2 \vee x_4) \wedge (x_2 \vee \neg x_3 \vee x_4)$$

There are 4 vertices, so $K = 4$. The graph G consists of (l, C), where l is a literal in clause C. We can join them if they are from different clauses and can be satisfied together. So, this gives us the following graph.



Highlighted in red is the clique of size 4. So, setting $x_1 = \texttt{true}, \neg x_2 = \texttt{true}, x_3 = \texttt{true}, x_4 = \texttt{true}$ satisfies B.

Problem Restrictions

A restriction of a problem consists of a subset of the instances of the original problem. If a restriction of a given decision problem Π is NP-complete, then so is Π. However, given an NP-complete problem Π, a restriction of Π might be NP-complete.

For example, a clique restricted to cubic graphs is in P. By definition, a cubic graph is a graph in which every vertex belongs to 3 edges. A largest clique has size at most 4, so exhaustive search is $O(n^4)$. On the other hand, if we restrict graph colouring to cubic graphs, the problem remains NP-complete.

The K-colouring problem is a restriction of GCP for a fixed number K of colours. A 2-colouring problem is in P, since it reduces to checking the graph is bipartile. On the other hand, 3-colouring is NP-complete.

Similarly, K-SAT is a restriction of SAT in which every clause contains exactly K literals. 2-SAT is in P, but 3-SAT is NP-complete. We can prove this by showing that SAT can be reduced to 3-SAT. So, assume we are given an instance B of SAT. We will construct an instance B' of 3-SAT. For each clause C of B, we construct a number of clauses of B', as shown below:

- If $C = l_1$, we introduce 2 addition variables x_1 and x_2 and add the clauses: $(l_1 \vee x_1 \vee x_2), (l_1 \vee x_1 \vee \neg x_2), (l_1 \vee \neg x_1 \vee x_2)$ and $(l_1 \vee \neg x_1 \vee \neg x_2)$ to B'. The conjunction of these clauses can be satisfied if and only if l_1 is $\texttt{true}$.

- If $C = (l_1 \vee l_2)$, then we introduce a variable y and add the clauses $(l_1 \vee l_2 \vee \neg y)$ and $(l_1 \vee l_2 \vee \neg y)$. The conjunction of these clauses can be satisfied if and only if $l_1 \vee l_2 = \texttt{true}$.

- If $C = (l_1 \vee l_2 \vee l_3)$, then we can just add the clause to B'.

- If $C = (l_1 \vee \cdots \vee l_k)$ (for $k > 3$), then we introduce $k-3$ additional variables $z_1, \ldots, z_{k-3}$ and add the clauses: $(l_1 \vee l_2 \vee z_1), (\neg z_1 \vee l_3 \vee z_2), (\neg z_2 \vee l_4 \vee$

$z_3), \ldots, (\neg z_{k-4} \vee l_{k-2} \vee z_{k-3}), (\neg z_{k-3} \vee l_{k-1} \vee l_k)$. Again, all the clauses hold if and only if C holds.

If we have a problem that is NP-complete, we can try restricting a problem if only a particular class of problems are of interest, given that the restriction is in P. We can seek an exponential-time algorithm that improves on exhaustive search. Moreover, for an optimisation problem, we could settle for an approximation algorithm that runs in polynomial time, especially if it gives a provably good result (within some factor of the optimal solution). For a decision problem, we can settle for a probabilistic algorithm that gives the correct answer with high probability.

CHAPTER 5

AN INTRODUCTION TO COMPUTABILITY

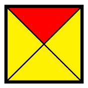
5.1 Computation and Decidability

A computer takes some input x, acts on the input as a black box, and returns an output $f(x)$. In this section, we consider what the black box can do. It comes a function that maps an input to an output.

Computability concerns which functions can be computed. It is a formal way of answering 'what problems can be solved by a computer', or alternatively 'what problems cannot be solved by a computer'.

To answer such questions, we require a formal definition, i.e. the definition of what a computer is. Alternatively, we ask what an algorithm is if we view a computer as a device that can execute an algorithm.

There are some problems that cannot be solved by a computer, even with unbounded time. For example, consider the tiling problem. A tile is a 1×1 square that is divided into 4 triangles by its diagonals, with each triangle given a colour. Moreover, each tile has a fixed orientation, i.e. we cannot rotate them. For example, a tile is:



We now define the tiling problem.

- **Name**: the tiling problem.
- **Instance**: a finite set S of tile descriptions.
- **Question**: can any finite area, of any size, be completely covered using only tiles of types in S, so that adjacent tile colours match?

We illustrate this problem with an example. So, assume that we can use the following tiles.

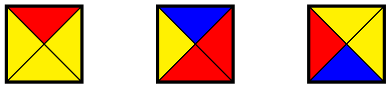


Figure 5.1: The available tiles.

Using these three tiles, we can make a 5×5 square as follows.

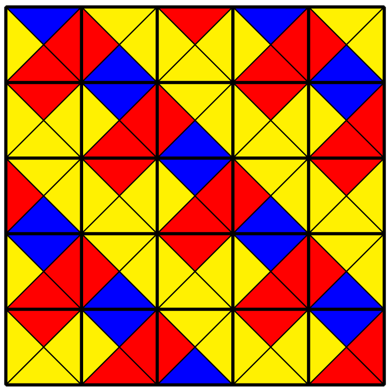


Figure 5.2: A 5×5 square made using the tiles above.

We can further extend them to an 8×5 square, by adding the bottom 3 rows to the end of the 5×5 square, as shown below.

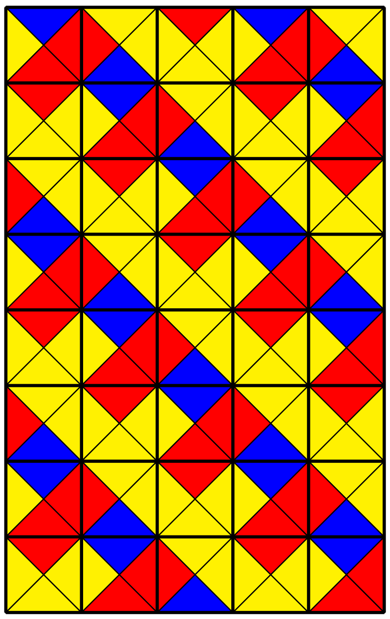
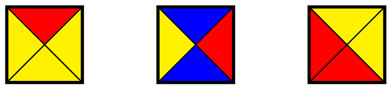


Figure 5.3: An 8×5 rectangle made using the tiles above.

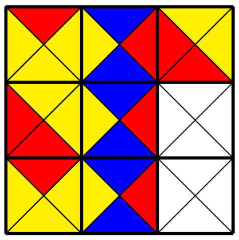
In fact, we can keep adding the bottom 3 rows to the rectangle at the bottom. There is a similar way to extend it to the rows, meaning that these tiles can be used to tile any finite area.

Now, assume we have the following 3 tiles.

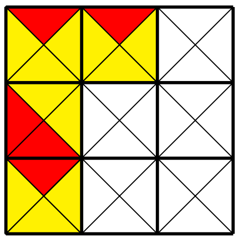


(1) (2) (3)

It is now impossible to tile a 3×3 square. We need to start from a tile that matches another tile in the right orientation, so we choose the tile (1) and stack tile (3) on top of it. We can only follow it with tile (1) at the top, and the first column is complete. Now, tile (1, 2) can be either tile (1) or tile (2)- we choose tile (2) first. In that case, the bottom two tiles in the second column must also be tile (2). Next, tile (2, 2) can only be tile (3). But then, we are stuck- there is no tile with top and bottom triangle red. Diagramatically, we are at the following state.



We had a choice when choosing tile (1, 2). If we instead choose tile (1), then we are stuck already- the tile we need needs to have both top and left yellow. Diagrammatically, we are at the following state.



So, the three tiles cannot be used to tile every finite area.

In general, there is no algorithm that decides this problem. That is, if we have an algorithm A that could solve this problem, then either A gives the wrong answer for some set of tiles, or A does not terminate. The issue here is that we have to check all finite areas, and there are infinitely many of these. For certain sets of tiles that can tile any area, there is no repeated pattern like we saw in the first case. So,the algorithm would really have to check all finite areas for it to be correct- such an algorithm cannot terminate.

A problem Π that admits no algorithm is called non-computable or unsolvable. If Π is a decision problem and does not admit an algorithm, then it is called undecidable. The Tiling problem is undecidable.

Another undecidable problem is the post's correspondence problem (PCP).

- **Name**: Post's correspondence Problem
- **Instance**: two finite sequences of words $X_1, \ldots, X_n$ and $Y_1, \ldots, Y_n$ over the same alphabet.
- **Question**: Does there exist a sequence $i_1, i_2, \ldots, i_r$ of integers chosen from $\{1, \ldots, n\}$ such that $X_{i_1} X_{i_2} \ldots X_{i_r} = Y_{i_1} Y_{i_2} \ldots Y_{i_r}$. That is, concatenating the X_{ij}'s and the Y_{ij}'s gives the same result.

For example, consider the case when $n = 5$ and we have the following sequence of words.

a b b	a	b a b	b a b a	a b a
b b a b	a a	a b	a a	a

Then, the sequence $2, 1, 1, 4, 1, 5$ gives the correspondence, as shown below.

a	a	b	b	a	b	b	b	a	b	a	a	b	b	a	b	a
a	a	b	b	a	b	b	b	a	b	a	a	b	b	a	b	a

Now, assume we have the following sequence of words.

b b	a	b a b	b a b a	a b a
b a b	a a	a b	a a	a

Then, there is no way to satisfy the correspondence condition. We can start with word (2) or (5) since the others would lead to a mismatch. If we start with (2), then we can only follow it with (2)- using (5) would lead to the `b` mismatching with an `a`. We can only continue on with (2), so we end up in a situation like this.

a	a	a			
a	a	a	a	a	a

There is no way that the two strings will match, so this will not work.

Instead, if we started with word (5), then we can only follow it with word (1). However, now we cannot use words (1), (3) or (4) since `b` and `a` would mismatch. So, we are stuck in the following state.

a	b	a	b	b
a	b	a	b	

So, there is no way to satisfy the correspondence condition with these 5 strings.

Next, we consider the halting problem.

- **Name**: The Halting Problem (HP)
- **Instance**: A program X and an input S.
- **Question**: Does the program X halt on S?

That is, the algorithm returns `true` if X halts when run on S, and instead returns `false` if X enters an infinite loop when run with input S. We will prove that there cannot be an algorithm that decides the Halting problem.

First, consider the following program.

```
void program1(int n):
    if n == 1:
        while true:
            pass
```

Clearly, this program terminates if and only if $n \neq 1$. Now, consider the following program.

```
void program2(int n):
    while n != 1:
        if n % 2 == 0:
            n = n/2
        else:
            n = 3*n + 1
```

If we call the program with $n = 7$, we get the sequence: $7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1$. So, the program above terminates for $n = 7$. In general, we do not know whether the program terminates for all values of n.

We shall now prove that HP is undecidable. So, assume for a contradiction that HP is decidable. In that case, we have an algorithm `halting` that decides HP. Now, define the following function `opposite`:

```
bool opposite(Program program):
    if halting(program, program.toString()):
        while true:
            pass
    else:
        return true
```

Now, run the program `opposite` on the program `opposite` itself. If the program halts when it is given itself, then the program loops. On the other hand, if the program does not halt when it runs on itself, then the program halts. Both of these statements are contradictions, so the programs `halting` and `opposite` cannot exist. Therefore, HP is undecidable.

We can prove that a problem is undecidable by reduction. Suppose that we can reduce any instance I of Π_1 into an instance J of Π_2 such that I has a yes-answer for Π_1 if and only if J has yes-answer for Π_2. Unlike PTRs, this algorithm need not be constructed in polynomial time.

If Π_1 is undecidable and we can perform such a reduction, then Π_2 is undecidable. Suppose for a contradiction that Π_2 is decidable. Then, we can use the reduction to decide Π_1. We can take an instance of Π_1, and convert it into an instance of Π_2. Since Π_2 is decidable, we can solve this problem. Therefore, the instance of Π_1 can also be solved since it has the same answer. Since Π_1 is undecidable, this is a contradiction. So, Π_2 must be undecidable.

5.2 Finite State Automata

We shall now look at different models of computation to give a mathematical representation of a computer. We will look at three classical models of computation, of increasing power. The models are described below.

- Finite state automata (FSA) are simple machines with a fixed amount of memory. They have very limited problem-solving ability, but they are still useful.
- Pushdown automata (PDA) are simple machines with an unlimited amount of memory that behaves like a stack.
- Turing machines (TM) are simple machines with unlimited memory that can be used arbitrarily. They essentially have the same power as a typical computer.

A deterministic finite-state automata (DFA) are simple machines with limited memory that recognise input on a read-only tape. It either recognises/accepts an input (which is made of characters from an alphabet), or does not recognise it. A DFA is made up of a finite input alphabet Σ, a finite set of states Q, an initial/start state $q_0 \in Q$ and a set of accepting states $F \subseteq Q$, along with a transition relation $T \subseteq (Q \times \Sigma) \times Q$, where $((q, a), q') \in T$ means that at state q, if we read an a, we go to q'. After we read the input, the input is recognised if and only if we end up at an accepting state. Determinism means that if $((q, a_1), q_1), ((q, a_2), q_2) \in T$, then either $a_1 \neq a_2$ or $q_1 = q_2$. That is, for any state and action, there is at most one move- we do not have a choice.

An example of a DFA is given below.

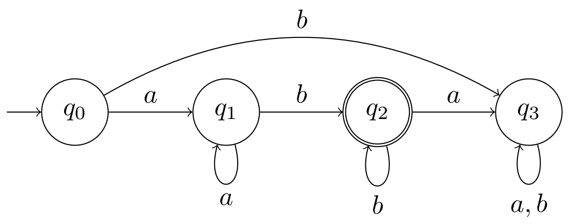


Figure 5.4: A DFA

Here, the alphabet $\Sigma = \{a, b\}$, and the states $Q = \{q_0, q_1, q_2, q_3\}$, the initial state is q_0 (marked by the arrow), and there is only one accepting state- q_2 (marked by a double circle). The transition relation is shown by edges connecting the states, e.g. $((q_0, a), q_1) \in T$. This DFA recognises the strings *ab* and *aaabb*, but it does not recognise *aabba*.

A DFA defines a language. It determines whether the string on the input tape belongs to that language. In other words, it solves a decision problem. In the case above, we find that *ab* and *aaabb* belong to that language, but *aabba* does not. In fact, the DFA recognises precisely the strings starting with an a, followed by as many a's, and then a b, followed by as many b's.

The following DFA recognises strings that contain two consecutive a's.

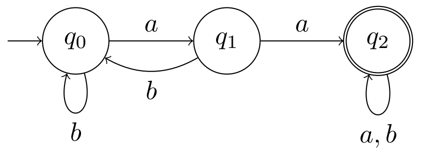


Figure 5.5: A DFA that recognises strings containing two consecutive a's.

At the start, it is possible for the string to have as many b's as possible (so, $((q_0, b), q_0) \in T$). As soon as we see 2 consecutive a, we're done- it doesn't matter what else we see (so, $((q_2, a), q_2), ((q_2, b), q_2) \in T$). If we see an a then a b, we restart (so, $((q_1, b), q_0) \in T$).

We can easily recognise the complement language, i.e. the set of strings that do not contain two consecutive a's. We make an accepting state non-accepting, and a non-accepting state an accepting one. The DFA is shown below.

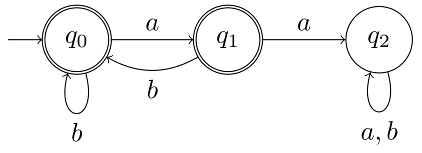


Figure 5.6: A DFA that recognises strings not containing two consecutive a's.

The following FSA recognises strings that start and end with a b.

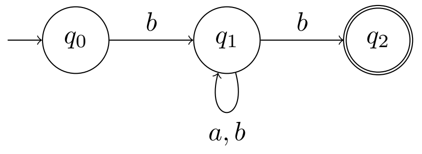


Figure 5.7: An NFA that recognises strings that start and end with b.

However, it is not a DFA since $((q_1, b), q_1), ((q_1, b), q_2) \in T$- we have a choice when we see a b in q_1- we can either stay at q_1 or go to q_2. This is a non-deterministic finite state automaton (NFA). We say that an NFA recognises a string if there is some way for the string to be accepted.

A DFA is an NFA by definition. However, every NFA can also be converted into a DFA. Therefore, non-determinism, in this case, does not expand the class of languages that can be recognised by a FSA.

We can reduce a NFA to a DFA using the subset construction. States of the DFA are sets of states of the NFA. So, construction can cause a blow-up in the number of states. In the worst case, we can go from N states to 2^N states. However, typically not all sets of the states are required in the DFA. For example, in the case above, the following is the DFA corresponding to it.

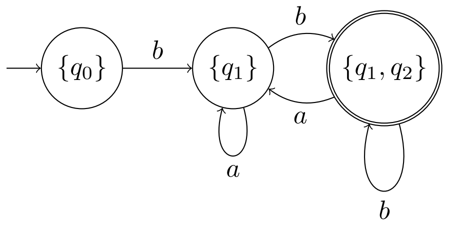


Figure 5.8: A DFA that recognises strings that start and end with b.

It only requires 3 states.

The languages that can be recognised by DFAs are called the regular languages. A regular language (over an alphabet Σ) can be specified by a regular expression over Σ. Here,

- The empty string ϵ is a regular expression;
- σ is a regular expression for all $\sigma \in \Sigma$;
- If R and S are regular expressions, then their concatenation RS is a regular expression;
- If R and S are regular expressions, then their union $R|S$ is also a regular expression;
- If R is a regular expression, then its closure R^* is a regular expression. R^* denotes 0 or more copies of R;
- If R is a regular expression, then (R) is an regular expression as well. This is used to override precedence between operators.

The order of precedence is closure, then concatenation, then union. We can use brackets to override this order. So, suppose $\Sigma = \{a, b, c, d\}$. Then, $R = (ac|a^*b)d$ means $((ac)|((a^*)b))d$. The corresponding language is

$$\{acd, bd, abd, aabd, aaabd, \dots\}.$$

There are other operations as well, such as complement $\neg x$, which is equivalent to the union of all characters in Σ except x. We can also use ? to denote any character from Σ- it is equivalent to the union of all the characters.

We now formally define concatenation, union and closure. So, let $L(R)$ and $L(S)$ be the languages corresponding to the regular expression R and S respectively. Then, the concatenation is defined as

$$L(RS) = \{rs \mid r \in R, s \in S\}.$$

The union is defined as

$$L(R|S) = L(R) \cup L(S).$$

Finally, the closure is defined as

$$L(R^*) = L(R^0) \cup L(R^1) \cup L(R^2) \cup \cdots,$$

where $L(R^0) = \{\epsilon\}$ and $L(R^{i+1}) = L(RR^i)$. Importantly,

$$L(R^*) \neq \{r^* \mid r \in L(R)\}.$$

In fact, this language cannot be recognised by any DFA for certain R. Essentially, for such a language, we would need a memory to remember which string in $r \in L(R)$ is repeated, and there could be an unbounded number of elements in $L(R)$.

Now, consider the language $(aa^*bb^*)^*$. This is the language composed of zero or more sequences which consist of a non-zero number of a's followed by a non-zero number of b's. The following DFA recognises the language.

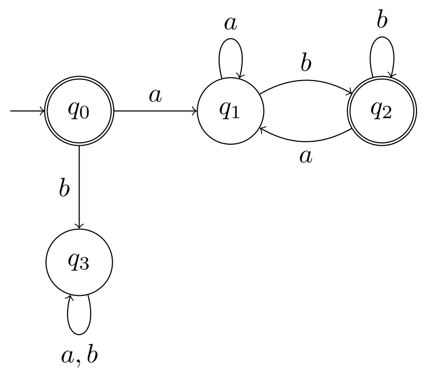


Figure 5.9: A DFA that recognises $(aa^*bb^*)^*$.

We accept the empty string, so q_0 is an accepting state. If we start with a b, then the string cannot be accepted. If we start with an a, then we can have as many a's after as we want, but the string can only get accepted if we see a b. After a b, if we see an a, then we can only accept the string if it is then followed by a b, and so on.

A DFA cannot recognise the language

$$L_1 = \{(a^m b^n)^* \mid m > 0, n > 0\}.$$

The problem is that the DFA would need to remember the m and the n to check that a string is in the language. But, there are infinitely many values for m and n. Hence, a DFA would need to have an infinitely many states. This is not possible, so a DFA cannot recognise L_1. Similarly, a DFA cannot recognise

$$L_2 = \{a^n b^n \mid n > 0\}.$$

The languages that can be recognised by a DFA are called regular. The languages L_1 and L_2 are not regular.

Now, we look at some more DFAs. The DFA below recognises strings of the form aa^*bb^*.

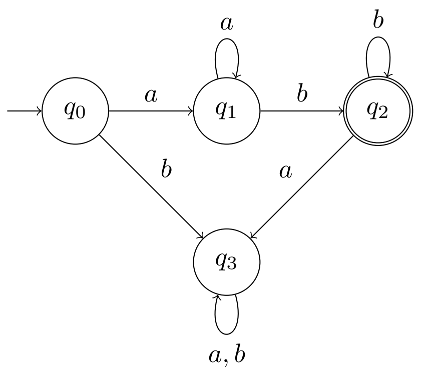


Figure 5.10: A DFA that recognises aa^*bb^*.

Moreover, the NFA below recognises $(a|b)^*aa(a|b)^*$- it has 2 consecutive a's.

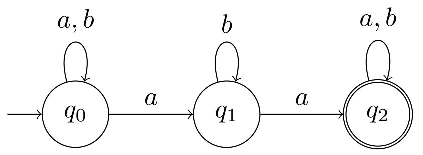


Figure 5.11: An NFA that recognises $(a|b)^*aa(a|b)^*$.

The following DFA also recognises strings that contain consecutive a's.

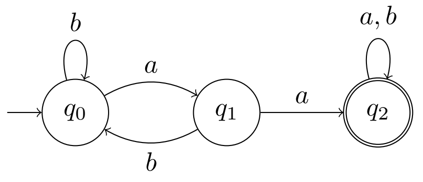


Figure 5.12: A DFA that recognises $(a|b)^*aa(a|b)^*$.

The following DFA recognises strings that do not contain consecutive a's- the regular expression is $a|b^*(abb^*)^*$.

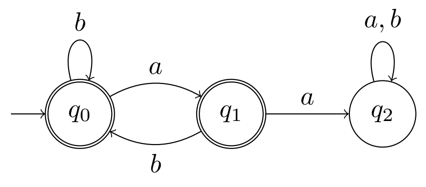


Figure 5.13: A DFA that recognises $a|b^*(abb^*)^*$.

The following NFA recognises strings that start and end with a b.

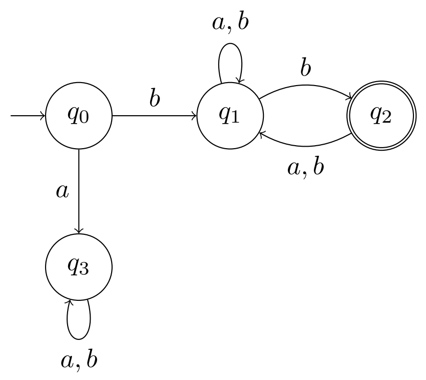


Figure 5.14: An NFA that recognises $b(a|b)^*b$.

The corresponding DFA for $b(a|b)^*b$ is given below.

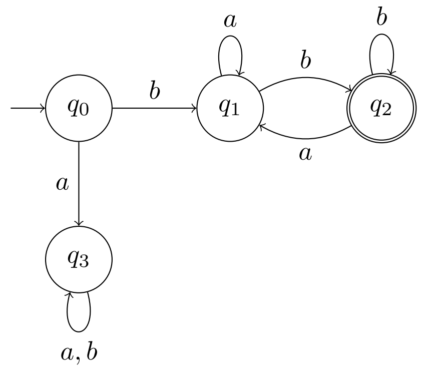


Figure 5.15: A DFA that recognises $b(a|b)^*b$.

Finally, the following DFA recognises strings that contain an odd number of a's- the regular expression is $b^*ab^*(ab^*ab^*)^*$.

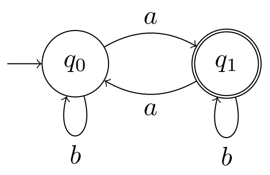


Figure 5.16: A DFA that recognises $b^*ab^*(ab^*ab^*)^*$.

5.3 Pushdown Automata

We saw that the language

$$L = \{a^n b^n \mid n > 0\}$$

cannot be recognised by a DFA. So, there are some functions/languages that we would regard as computable that cannot be computed by a finite-state automaton. Therefore, DFAs are not an adequate model of a general-purpose computer.

So, we consider another model of computation that recognises L. It cannot be done without some form of memory, such as a stack. For instance, as we read a's, we can push them onto a stack, and we can pop them as we read the b's. The stack works as a counter and ensures that the number of a's and b's are equal.

Pushdown automata extend extend finite-state automata with a stack. It consists of a finite input alphabet Σ, a finite set of stack symbols G, a finite set of states Q, including a start state and a set of accepting states, and a control or transition relation $T \subseteq (Q \times \Sigma \cup \{\epsilon\} \times G \cup \{\epsilon\} \times (Q \times G \cup \{\epsilon\}))$, where ϵ is the empty string. A tuple $(q_a, w, \alpha, (q_b, \beta))$ represents a transition from the state q_a to the state q_b in the case where w is the next letter on the input tape and α is on the top of the stack- this then gets popped and we push β. If we have the empty string ϵ, then we do not pop or push. A PDA accepts an input if and only if after the input has been read, the stack is empty and the control is in an accepting state.

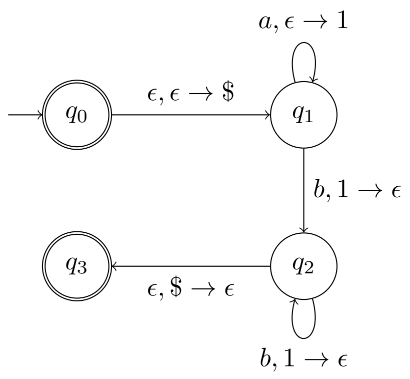
There is no explicit test to check that the stack is empty. However, this can be achieved by adding a special symbol (\$) to the stack at the start of the computation. We never add the symbol at any other point during the computation. Therefore, we can check that the stack is empty by checking whether the symbol \$ is on the top of the stack.

The PDAs defined here are non-deterministic. In this case, deterministic PDAs are less powerful. This differs from finite state automata where non-determinism does not add any power. So, there are languages that can be recognised by a non-deterministic PDA but not by a deterministic PDA. An example of this is the language of palindromes. A palindrome is a string that reads the same both forwards and backwards.

Using a PDA, we can recognise a palindrome by pushing the first half of the sequence onto the stack. Then, as we read each new character, we check if it is the same as the top element on the stack and we pop this element. We then enter an accepting state if all the checks succeed. Also, if the string is of odd length, then the middle character should not be added to the stack.

We require non-determinism here to know when to stop pushing onto the stack. We also need it since we do not know whether the length of the string is even or odd. We cannot work this out first and then check the string since we would need to read the string twice- this is not possible with a stack. We would need an unbounded number of states as the string could be of any finite length.

Now, we consider an example.



Here, we start at q_0. An empty string is accepted since q_0 is also an accepting state. We first add \$ to the stack to mark the bottom of the stack. Next, if we encounter an a, we add 1 to the stack. Instead, if we encounter a b, we remove 1 from the stack. If we get to the end of the string with an empty stack (i.e. all the a's have been removed), then we remove \$ and get to an accepting state. This PDA therefore recognises strings of the form $a^n b^n$ for $n \geq 0$. If there are more a's than b's, then the stack doesn't get emptied by the end of the string, and we are stuck at q_2. Instead, if there are more b's than a's, then we get stuck at q_3.

Clearly, PDAs are more powerful than FSMs. For example, the language

$$L = \{a^n b^n \mid n \geq 0\}$$

cannot be recognised by a FSM, but is recognised by a PDA. The languges that can be recognised by a non-deterministic PDA are called the context-free languages. Although PDAs are more powerful than FSM, it is not an adequate model of a general purpose computer. For example, it cannot recognise the language

$$L = \{a^n b^n c^n \mid n \geq 0\},$$

although we can write a program that recognises it.

5.4 Turing Machines

A Turing machine is used to recognise a particular language, and consists of:

- a finite alphabet Σ, including a blank symbol (denoted by $\#$);
- an unbound tape of square, where each square can hold a single symbol of Σ, and the tape is unbounded in both directions;
- a tape head that scans a single square, which can read the square and write to it, and then move one square left or right;
- a set of states, including a starting state s_0 and two halting states- accepting s_Y and rejecting s_N;
- a transition function, which tells us how to go from one state to other, what character to read from the tape, what to write to the tape and which way to move the tape head. It is of the form

  $$f : ((S \setminus \{s_Y, s_N\}) \times \Sigma) \to (S \times \Sigma \times \{\texttt{left}, \texttt{right}\}.$$

  If $f(s, \sigma) = (s', \sigma', d)$, then we read the symbol σ from the tape in state s. We then move it to state s', overwrite the symbol σ on the tape with the symbol σ', and move the tape head one square in the direction $d \in \{\texttt{left}, \texttt{right}\}$.

The (finite) input string is placed on the tape. We assume initially that all the other squares of the tape contain blanks. The tape head is placed on the first symbol of the input. The Turing machine starts in state s_0. If the machine halts in s_Y, the answer is `yes`- the input gets accepted. Instead, if the machine halts in s_N, the answer is `no`- the input gets rejected.

We now consider the palindrome problem. We can write a simple program in a high-level language to do this, as shown below.

```
bool isPalindrome(String string):
    if string.length < 2:
        return true
    else:
        if string[0] != string[-1]:
            return false
        else:
            return isPalindrome(string[1:-1])
```

We will design a Turing machine that solves this problem. For simplicity, we assume that the string is composed of a's and b's. Formally defining a Turing machine for even simple problems is hard, so we will start with a pseudocode version. Turing machines are quite simple, making them easy to do proofs in. However, it also means that it is not easy to program using them.

The pseudocode that solves the palindrome problem for Turing machines is given below.

```
read symbol
erase symbol
enter state that remembers symbol
move tapehead to end
if only blank characters:
    enter accepting state
else if last character == erased character:
    erase symbol
else:
    enter rejecting state
if no input:
    enter accepting state
else:
    move tapehead to start
    repeat
```

We need the following states, with the alphabet $\Sigma = \{\#, a, b\}$:

- s_0 to read and erase the leftmost symbol;
- s_1, s_2 to move right to look for the end, remembering the erased symbol, i.e. s_1 when read (and erased) a and s_2 when read (and erased) b;
- s_3, s_4 to test for the appropriate rightmost symbol, i.e. s_3 testing against a and s_4 testing against b; and
- s_5 to move back to the leftmost symbol.

Moreover, we need the following transitions:

- From s_0, we enter s_Y if a blank is read, or move to s_1 or s_2 depending on whether an a or a b is read, erasing it in either case.
- We stay in s_1 or s_2, moving right until a blank is read, at which point, we enter s_3 or s_4, and move left.
- From s_3 or s_4, we enter s_Y if a blank is read, s_N if the 'wrong' symbol is read. Otherwise, we erase it, and enter s_5, and move left.
- In s_5, we move left until a blank is read, then move right and enter s_0.

A Turing machine can be described by its state transition diagram, which is a directed graph where each state is represented by a vertex, and $f(s, \sigma) = (s', \sigma', d)$ is represented by an edge from vertex s to vertex s', labelled $\sigma \to \sigma', d$.

The Turing machine for the palindrome problem is therefore the following.

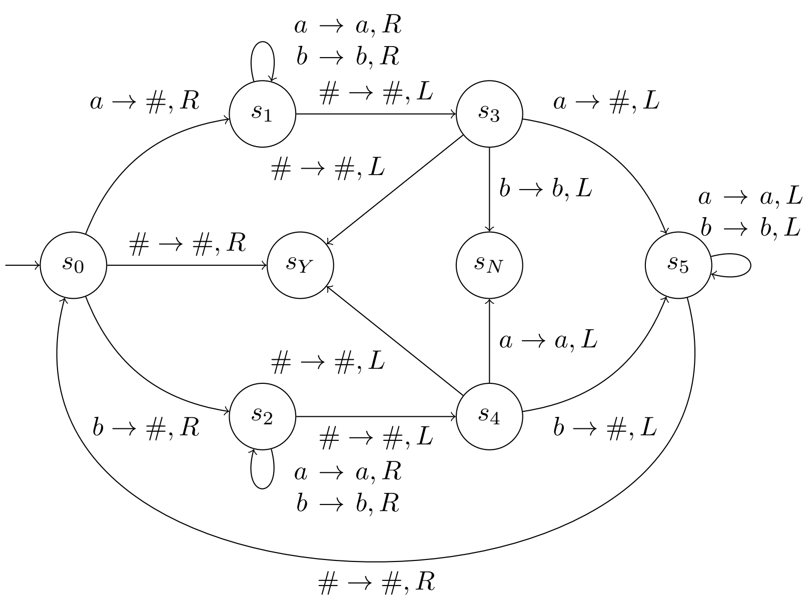


Figure 5.17: A Turing machine that recognises palindromes over the alphabet $\Sigma = \{a, b\}$.

We shall now show that the language consisting of strings of the form $a^n b^m c^{n+m}$, for $n, m \geq 0$ is decidable. One solution is to first delete the a's from the start each time and delete a corresponding c from the end of the string. This will not introduce blanks as we are merely deleting from the start and the end. So, there is no ambiguity as to where the start and the end of the string is. After we have delete all the a's, we are left with $b^m c^m$, so we delete one b from the front and one c from the end. To do this, we need two 'restart' states- one where we are removing an a, and one where we are removing a b.

The pseudocode for the algorithm is given below.

```
restart1 => if blank:
    accept
else if c: // must start with an a or b
    reject
else if a: // starts with a, so find c
    delete
    move to end
    if blank, a or b: // need c at the end
        reject
    else:
        delete
        move to start
        goto restart1 // now at start
else if b: // starts with b (no more a's), so find c
    delete
    move to the end
    if blank, a or b: // need c at the end
        reject
    else:
        delete
        move to start
        goto restart2 // now at start
restart2 => if blank:
    accept
else if a or c: // must start with b
    reject
else if b:
    delete
    move to end
    if blank, a or b: // need c at the end
        reject
    else:
        delete
        move to start
        goto restart2 // now at start
```

The TM corresponding that decides this language is given below.

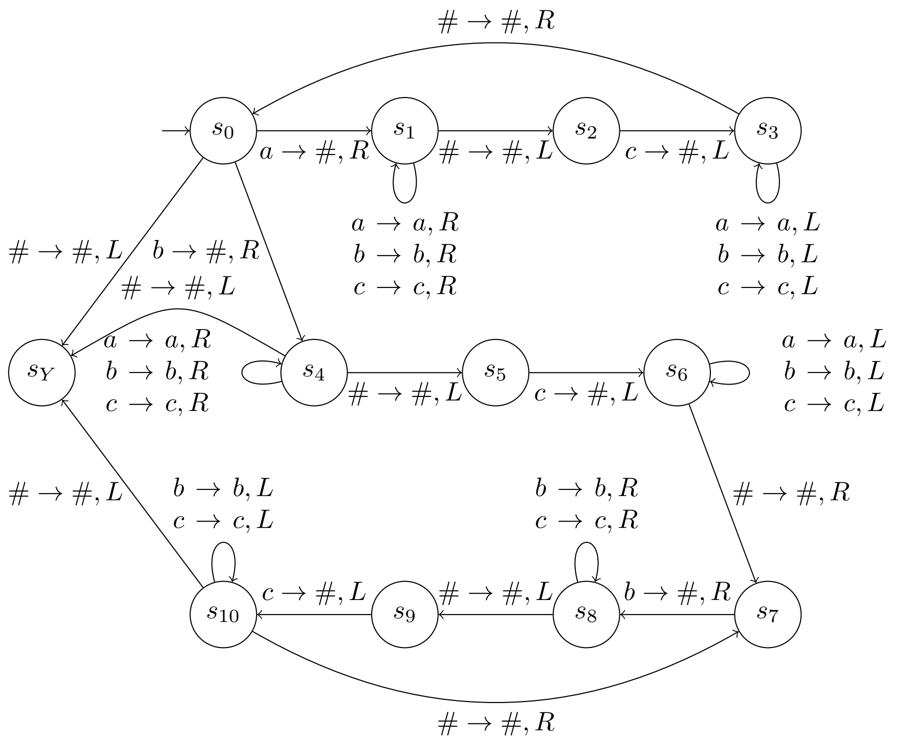


Figure 5.18: A Turing machine that recognises strings of the form $a^n b^m c^{n+m}$ for $m, n \geq 0$

Any path not shown is connected to s_N.

Turing Machines as Functions

A Turing machine can be thought of as a function. For instance, say that a Turing machine accepts the language L. In that case, it can be thought of as computing a function f, where $f(x) = 1$ if $x \in L$ and 0 otherwise.

We can further extend this to use Turing machines to compute a function. It has a set H of halting states, and the function it computes is defined by $f(x) = y$, where x is the initial string on the tape, and y is the string on the tape when the machine halts. For example, the palindrome TM could be redefined so that it deletes the tape contents, and instead of entering s_Y, it writes 1 on the tape and enters a halt state, or instead of entering s_N, it writes 0 on the tape and enters a halt state.

We will now construct a Turing machine that computes the function $f(k) = k + 1$, where the input is in binary. If $k = 100010$, we get $k + 1 = 100011$- we just change the 0 into a 1. Now, if $K = 100111$, then $k + 1 = 101000$- we change all the 1s at the end to a 0, and change the first 0 we encounter into a 1. We have another case- if $k = 11111$, then $k + 1 = 100000$. So, if we cannot find a rightmost 0, then all the entries are 0. So, we replace the first blank before the input with 1, then move right. If it is a 1, we replace it with a 0. If we end up at a blank, we halt.

So, the pseudocode for this function is:

```
move tapehead to end
move left until 0 or blank
do:
    change value to 1
    move right
until tapehead at end
```

We will now consider the states we need:

- We need the initial state s_0, which moves to the right, seeking the end of the input (the first blank);
- We need the state s_1, looking for the rightmost 0 or blank;
- We need the state s_2 to find the first 0 or blank. We change it to 1 and move right, changing 1s into 0s.
- We need a halting state s_3.

Moreover, we have the following transitions:

- From s_0, we enter s_1 at the first blank;
- From s_1, we enter s_2 if we find a 0 or a blank;
- From s_2, we enter s_3 at the first blank.

The corresponding Turing machine is therefore the following:

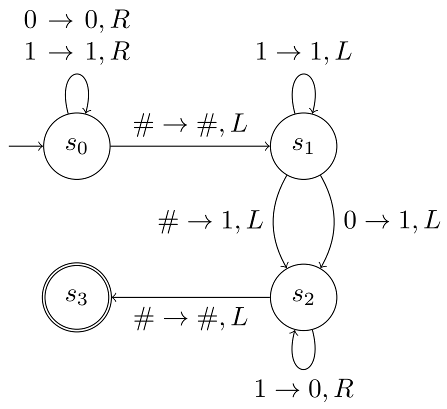


Figure 5.19: A Turing machine that adds 1 to a binary number.

We can also construct a Turing machine that returns 1 if it is divisible by 4, and 1 otherwise. The alphabet is $\Sigma = \{0, 1, \#\}$. We will assume that there is at least one non-blank. A binary number is divisible by 4 if the final two digits are 0. The only edge case is 0, which has a single zero. So, a number is divisible by 4 if the last digit is a 0, and the one before it is 0 or blank. In that case, we erase everything and write 1. Otherwise, we erase everything and write 0.

So, the Turing machine that checks divisibility by 4 is the following.

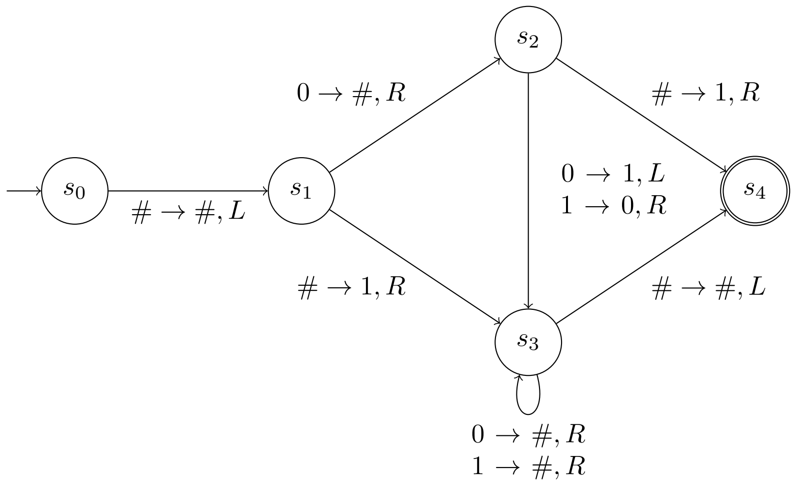


Figure 5.20: A Turing machine that determines whether a binary string is divisible by 4.

A language L is Turing-recognisable if there exists some Turing machine that recognises it. That is, given an input string x, if $x \in L$, then the TM halts in state s_Y. But, if $x \notin L$, then TM halts in state s_N, or it doesn't halt. A language L is Turing-decidable if there exists some Turing machine that decides it. That is, given an input string x, if $x \in L$, then the TM halts in state s_Y, and if $x \notin L$, then the TM halts in state s_N. Clearly, every Turing-decidable language is Turing-recognisable, but not every Turing-recognisable language is Turing-decidable. For example, the language corresponding to the Halting problem is Turing-recognisable, but not decidable- if a program terminates, we will enter s_Y, but not s_N if it does not.

A function $f : \Sigma^* \to \Sigma^*$ is Turing-computable if there is a Turing machine M such that for any input x, the machine M halts with output $f(x)$.

A Turing machine may be enhanced in various ways. For example, we can have multiple tapes, or a 2-dimensional tape. Also, we can make the Turing machine non-deterministic. It turns out that none of these enhancements change the computing power. That is, every language/function that is recognisable/decidable/computable with an enhanced Turing machine is recognisable/decidable/computable with a basic Turing machine. This can be proved by showing that a basic Turing machine can simulate any of these enhanced Turing machines.

5.5 Counter Programs and the Church-Turing Thesis

Counter programs are a different model of computation. All general-purpose programming languages have essentially the same computational power. A program written in one language could be translated (or compiled) into a functionally equivalent program in any other. Counter programs are the simplest form of a programming language but with the same computational power.

Counter programs have variables of type `int`, labelled statements are of the form `L: unlabelled_statement`, and unlabelled statements are of the form `x = 0`, `x = y+1`, `x = y-1` or `if x==0 goto L`, for some labelled statement `L`, and `halt`. This has the same computational power as a normal programming language, but is not as easy to code in. For example, we can write a counter program to evaluate the product $x \cdot y$:

```
u = 0 // dummy variable
z = 0 // product

A: if x == 0 goto C // end of outer loop
    x = x-1 // perform this loop x times
    v = y+1 // each time around the loop, set v to equal y
    v = v-1 // in a slightly contrived way

B: if v == 0 goto A // end of inner loop
    v = v-1 // perform this loop v times (i.e. y)
    z = z+1 // each time, incrementing z

    if u == 0 goto B // always goes to B

C: halt
```

We have seen that DFA and PDA are not appropriate models of computation. However, it turns out that Turing machines are an appropriate model of computation- this is known as Church-Turing thesis. It is based on the fact that a whole range of different computational models turn out to be equivalent in terms of what they can compute. So, it is reasonable to infer that any one of these models encapsulates what is effectively computable.

That is, it states that everything 'effectively computable' is computable by a Turing machine. It is a thesis and not a theorem since it uses the inform term 'effectively computable'. This means that there is an effective procedure for computing the value of the function, including all computers/programming languages that we know about at present, and even those we do not.

There are many equivalent computational models, such as lambda calculus, Turing machines, recursive functions, production systems, counter programs and all general-purpose programming languages. Each of these can simulate the others.