Trees

Mathematically speaking trees are a *special class* of a graph.

The relationship of a trees to a graph is very important in solving many problems in Maths and Computer Science

However, in computer science terms it is sometimes convenient to think of certain trees (especially *rooted trees* — **more soon**) as separate data structures.

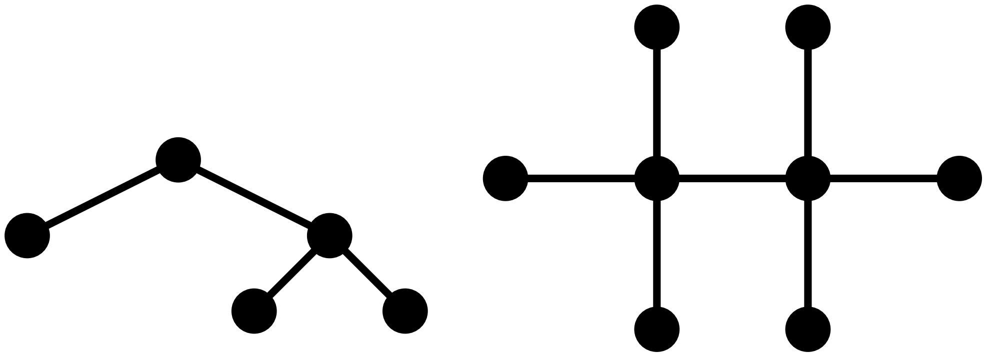
- They have they own variations of data structure
- They have many specialised algorithms to traverse, search *etc.*
- Very common data structure in almost all areas of computer science.

We will study both of the above aspects,
but will focus on applications

Definition 2.27 (Tree).

A tree T is a connected graph that has no cycles.

Example 2.16 (Simple Trees).



Theorem 2.28 (Equivalent definitions of a tree).

Let T be a graph with n vertices.

Then the following statetments are equivalent.

- *T is connected and has no cycles.*
- *T has $n - 1$ edges and has no cycles.*
- *T is connected and has $n - 1$ eges.*
- *T is connected and the removal of any edge disconnects T.*
- *Any two vertices of T are connected by exactly one path.*
- *T contains no cycles, but the additon of any new edge creates a cycle.*

Problem 2.17 (Trees v Graphs).

Why are trees a very common data structure in computer science algorithms and applications?

Which are more commonly used: Trees or Graphs?

Research you answer by finding some key applications of trees and graphs.

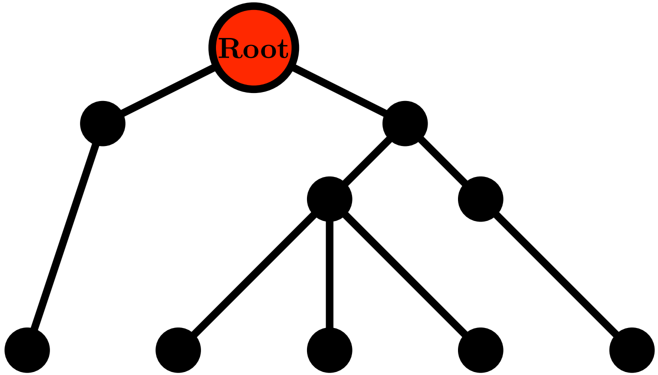
Justify any conclusion reached.

Rooted trees

Many applications in Computer Science make use of so-called **rooted trees**, especially **binary trees**.

Definition 2.29 (Rooted tree).

If one vertex of a tree is singled out as a starting point and all the branches fan out from this vertex, we call such a tree a **rooted tree**.



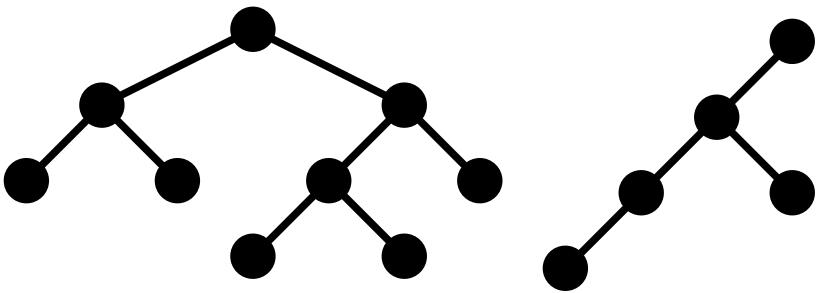
Binary Trees

Rooted trees can have many different forms.

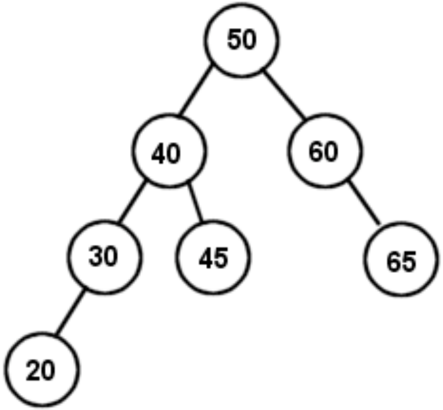
A very simple form is also a very important one:

Definition 2.30 (Binary Tree)**.**

A rooted tree in which there are at most two descending branches at any vertex is called a **binary tree***.*



Example 2.17 (Binary Tree Example: Sorting).



Create tree via:

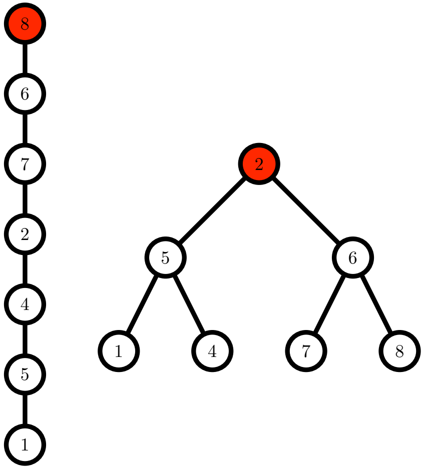
- *First number is the root.*
- *Put number in tree by traversing to an end vertex*
  - *If number less than or equal vertex number go left branch*
  - *If number greater than vertex number go right branch*

Tree above for the sequence: 50 40 60 30 20 65 45

Example 2.18 (Root/Binary Tree Example: Stacks/Binary Tree).

Rooted trees can be used to store data in a computer's memory in many different ways.

Consider a list of seven numbers $1, 5, 4, 2, 7, 6, 8$. *The following trees show two ways of storing this data, as a binary tree and as a stack.*



Both trees are rooted trees and both representations have their advantages. However it is important in both cases to know the starting point of the data, i.e. **the root**.

Example 2.19 (Huffman Coding: Data compression).

The standard ASCII code uses 8 *bits to represent a character* [1].
So any character sequence, of length n**, is** $n \times 8$ **bits long**

E.g: `EIEIO`

$$\overbrace{01000101}^{\texttt{E(69)}}\overbrace{01001001}^{\texttt{I(73)}}\overbrace{01000101}^{\texttt{E(69)}}\overbrace{01001001}^{\texttt{I(73)}}\overbrace{01001111}^{\texttt{O(79)}} = 5 \times 8 = 40 \; bits$$

The Main aim of Data Compression is find a way to use less bits per character, *E.g.:*

$$\overbrace{xx}^{\texttt{E(2bits)}}\;\overbrace{yy}^{\texttt{I(2bits)}}\;\overbrace{xx}^{\texttt{E(2bits)}}\;\overbrace{yy}^{\texttt{I(2bits)}}\;\overbrace{zzz}^{\texttt{O(3bits)}} = \overbrace{(2 \times 2)}^{\texttt{2×E}} + \overbrace{(2 \times 2)}^{\texttt{2×I}} + \overbrace{3}^{O} = 11$$
bits

[1]**Note:** We consider character sequences here for simplicity. Other token streams can be used — e.g. Vectorised Image Blocks, Binary Streams.

Huffman Coding: Counting Up

Using binary trees one can find a way of representing characters that requires less bits :

- We construct minimal length encodings for messages when the frequency of letters in the message is known.
- Build a binary tree based on the frequency — essentially a special sorting procedure
- Traverse the tree to assemble to minimal length encodings

A special kind of binary tree, called a Huffman coding tree is used to accomplish this.

Basic Huffman Coding Compression Algorithm

For a given sequence of letters:

1. Count the frequency of occurrance for the letters in the sequence.
2. Sort the frequencies into increasing order
3. **Build the Huffman coding tree**:
   - Choose the two smallest values, make a (sub) binary with these values.
   - Accumulate the sum of these values
   - Replace the sum in place of original two smallest values and repeat from 1.
   - *Construction of tree is a bottom up insertion of sub-trees at each iteration.*
4. **Making the codes**: Traverse tree in top down fashion
   - Assign a 0 to left branch and a 1 to the right branch
   - Accumulate 0s and 1s for each character from root to end vertex.
   - This is the **Huffman code** for that character.

Basic Huffman Coding Example (1)

Consider a sequence of characters:

```
EIEIOEIEIOEEIOPPEEEEPPSSTT .......   EEEPPPPTTSS
```

and suppose we know that the frequency of occurrance for six letters in a sequence are as given below:

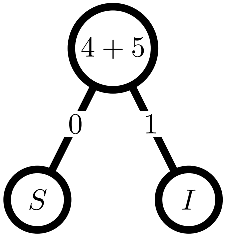
$$\begin{array}{cc} E & 29 \\ I & 5 \\ O & 7 \\ P & 12 \\ S & 4 \\ T & 8 \end{array}$$

Basic Huffman Coding Example (2)

To build the Huffman tree, we sort the frequencies into increasing order (4, 5, 7, 8, 12, 29).

$$\begin{array}{cc} S & 4 \\ I & 5 \\ O & 7 \\ T & 8 \\ P & 12 \\ E & 29 \end{array}$$

Then we choose the two smallest values S and I (4 and 5), and construct a binary tree with labeled edges:

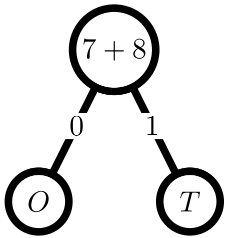


Basic Huffman Coding Example (3)

Next, we replace the two smallest values S (4) and I (5) with their sum, getting a new sequence, (7, 8, 9, 12, 29).

$$\begin{array}{cc} O & 7 \\ T & 8 \\ SI & 9 \\ P & 12 \\ E & 29 \end{array}$$

We again take the two smallest values, O and T, and construct a labeled binary tree:

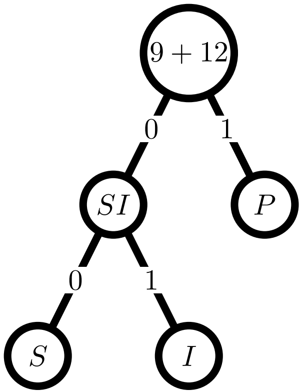


Basic Huffman Coding Example (4)

We now have the frequencies (15, 9, 12, 29) which must be sorted into (9, 12, 15, 29)

$$\begin{array}{cc} SI & 9 \\ P & 12 \\ OT & 15 \\ E & 29 \end{array}$$

and the two lowest, which are IS (9) and P (12), are selected once again:

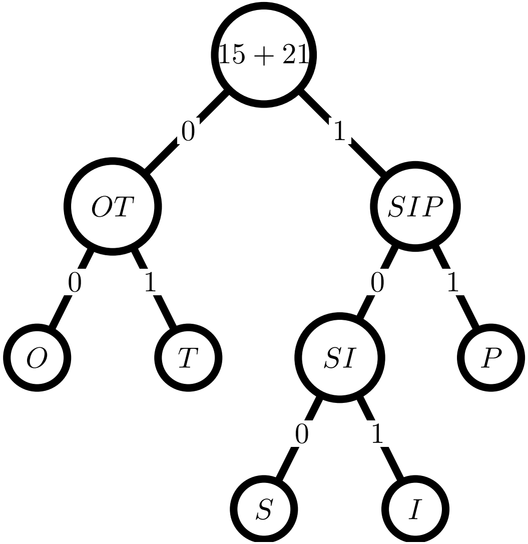


Basic Huffman Coding Example (5)

We now have the frequencies (21,15, 29) which must be sorted into (15, 21, 29)

OT	15
SIP	21
E	29

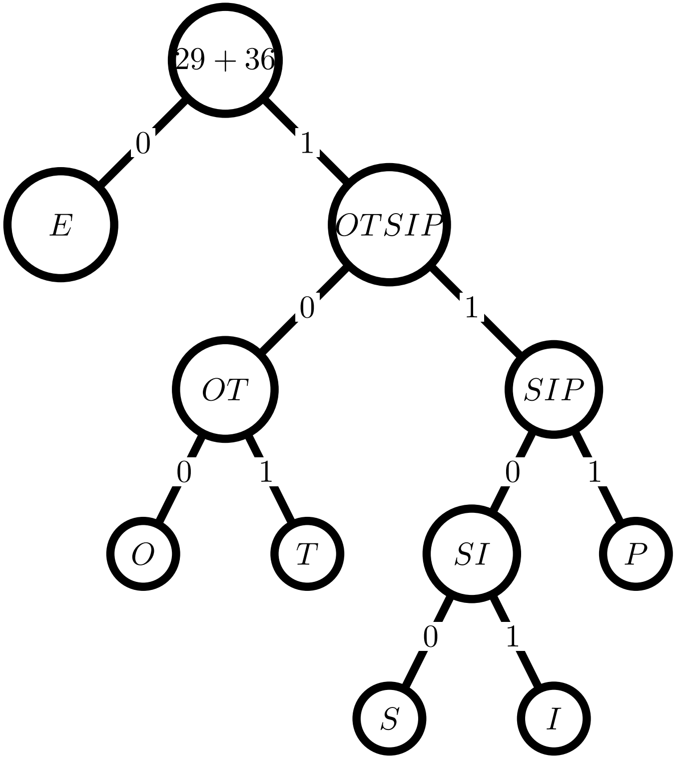
Now, we combine the two lowest which are OT (15) and ISP (21):



Basic Huffman Coding Example (6): Final Tree

The two remaining frequencies, 36 and 29, are now combined into the final tree.

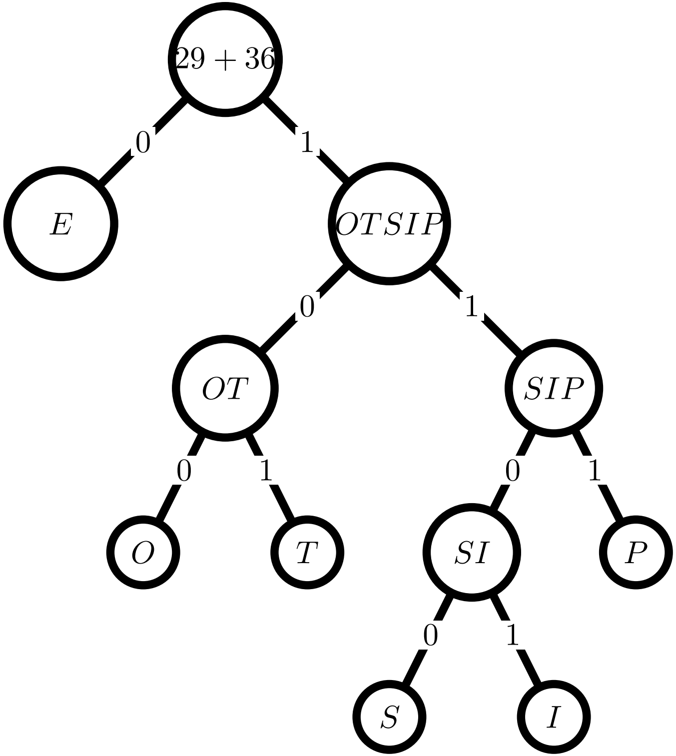
E	29
$OTSIP$	36



Basic Huffman Coding Example (7)

From this final tree, we find the encoding for this alphabet:

E	0
I	1101
P	111
O	100
S	1100
T	101



Basic Huffman Coding Example (9): Getting the Message

So looking at the frequency of the letters and their new compact codings:

$$\begin{array}{ccc} S & 4 & 1100 \\ I & 5 & 1101 \\ O & 7 & 100 \\ T & 8 & 101 \\ P & 12 & 111 \\ E & 29 & \mathbf{0} \end{array}$$

We see that the highest occurring has less bits

Basic Huffman Coding Example (10): Getting the Message

Using this code, a message like `EIEIO` would be coded as:

$$\overbrace{0}^{\texttt{E}}\ \overbrace{1101}^{\texttt{I}}\ \overbrace{0}^{\texttt{E}}\ \overbrace{1101}^{\texttt{I}}\ \overbrace{100}^{\texttt{O}}$$

This code is called a **prefix encoding**:

As soon as a 0 is read, you know it is an E. 1101 is an I — you do not need to see any more bits.

When a 11 is seen, it is either I or P or S, etc.

Note: Clearly for every message the code book needs to be known (= transmitted) for decoding

Basic Huffman Coding Example (11): Getting the Message

If the message had been coded in the ”normal” ASCII way, each letter would have required 8 bits.

The entire message is 65 characters long so 520 bits would be needed to code the message (8*65).

Using the Huffman code, the message requires:

$$\overbrace{1 \times 29}^{E} + \overbrace{3 \times 12}^{P} + \overbrace{3 \times 8}^{T} + \overbrace{3 \times 7}^{O} + \overbrace{4 \times 5}^{I} + \overbrace{4 \times 3}^{S} = 142 \text{ bits.}$$

A simpler **static** coding table can be applied to the *English Language* by using average frequency counts for the letters.

Problem 2.18.

Work out the Huffman coding for the following sequence of characters:

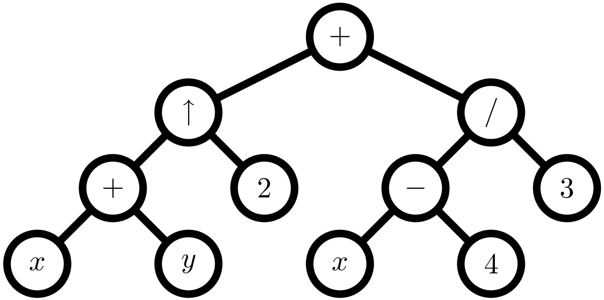
```
BAABAAALLBLACKBAA
```

Example 2.20 (Prefix Expression Notation).

In compiler theory and other applications this notation is important.

Consider the expression: $((x + y)^2 + ((x - 4)/3)$

*We can represent this as a binary tree (*an expression tree*):*



If we traverse the tree in preorder*, that is top-down, we can obtain the prefix notation of the expression:*

$$+ \uparrow + \, x \, y \, 2 \, / \, - \, x \, 4 \, 3$$

Why is this so important?

Example 2.21 (Two Player Game Playing:The Min-Max Algorithm).
A classic Artificial Intelligence Paradigm

The Min-Max algorithm is applied in two player games, such as tic-tac-toe, checkers, chess, go, and so on.

All these games have at least one thing in common, they are logic games.

This means that they can be described by a set of rules and premisses:

- It is possible to know from a *given point* in the game, all the *next available moves* to the end of the game.
- They are 'full information games':

  Each player knows everything about the possible moves of the adversary.

Two Player Game Playing:The Min-Max Algorithm (Cont.)

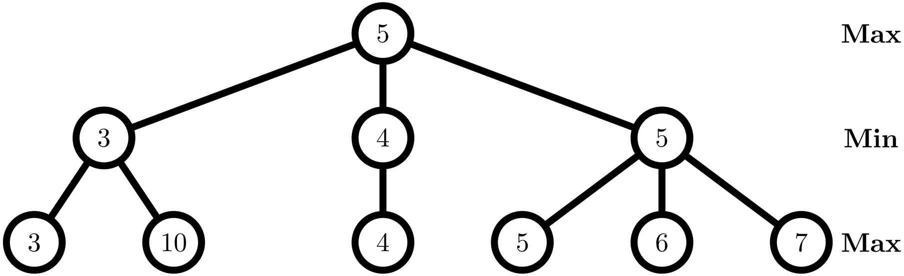
A tree representation of the game is used:

- Vertices represent points of the decision in the search — the state of play at a given position.
- Valid plays are connected with edges.
- We **search the tree** in special way to find winning moves

Two Player Game Playing:The Min-Max Algorithm (Cont.)

Two players involved, MAX and MIN and we want MAX to win.

- A search tree is generated, depth-first, starting with the current game position upto the end game position.
- Then, the final game position is evaluated from MAX point of view:



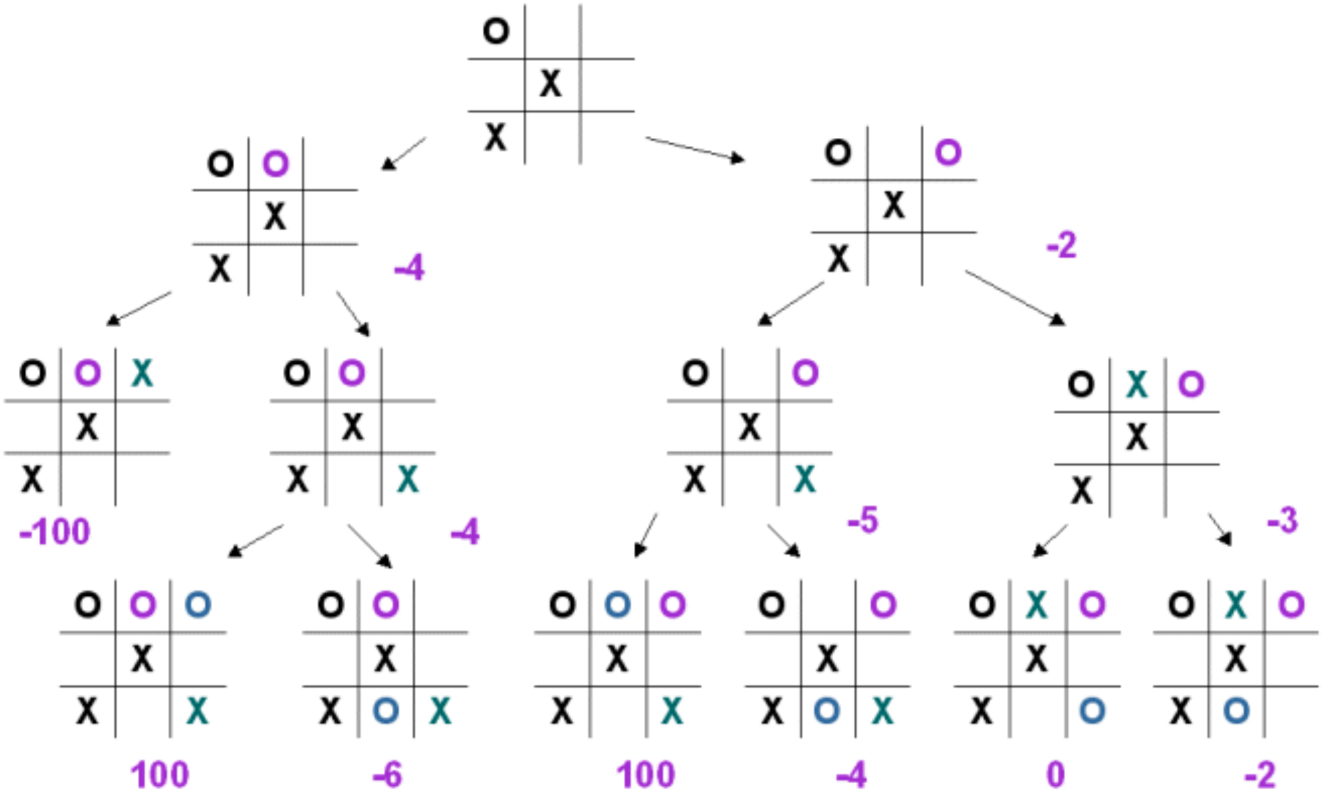
- Search works by selecting

  all vertices that belong to MAX receiving the maximun value of it's children,

  all vertices for MIN with the minimun value of it's children.
- **Winning path** is path with *highest sum*

Min-Max: Noughts and Crosses Example

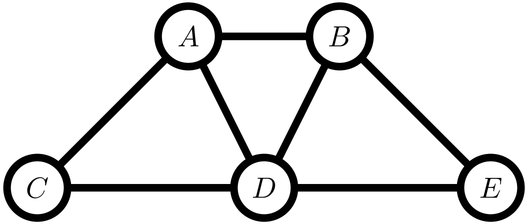
A real game example for a Noughts and Crosses game:

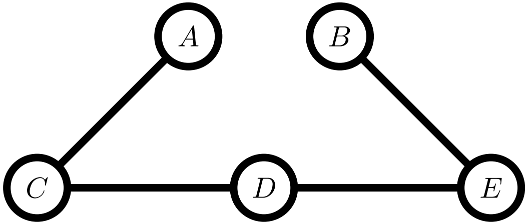


Spanning Tree

Definition 2.31 (Spanning Tree).

Let G be a connected graph. Then a spanning tree in G is a subgraph of G that includes every vertex and is also a tree.





Problem 2.19 (Spanning Tree).
Draw another spanning tree of the above graph?

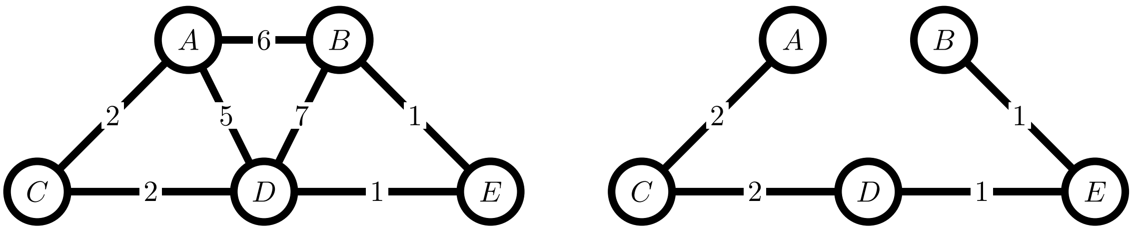
Minimum Connector Problem (1)

In this section we investigate the problem of constructing a spanning tree of minimum weight. This is one of the important problems in graph theory as it has many applications in real life, such as networking/protocols and TSP.

We start with the formal definition of a minimum spanning tree.

Definition 2.32. *Let T be a spanning tree of minimum total weight in a connected graph G. Then T is a minimum spanning tree or a minimum connector in G.*

Example 2.22.

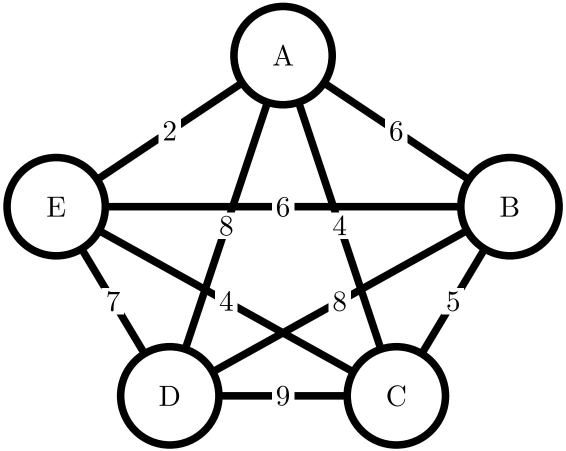


Minimum Connector Problem (2)

The minimum connector problem can now be stated as follows:

> *Given a weighted graph G, find a minimum spanning tree.*

Problem 2.20 (Minimum Connector Problem)**.**
Find the minimum spanning tree of the following graph:



Minimum Connector Problem: Solution

There exist various algorithms to solve this problem. The most famous one is Prim's algorithm .

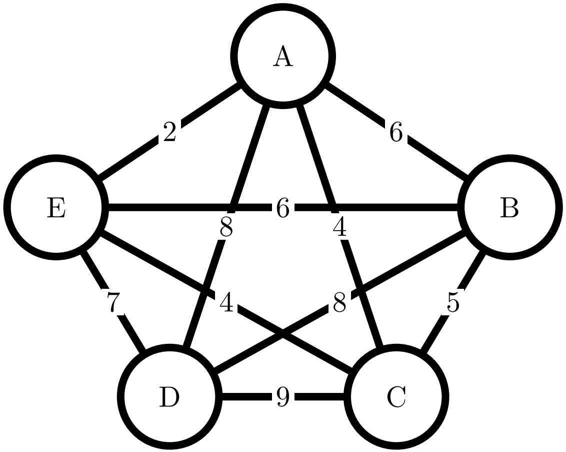
Algorithm 2.33 (Prim's algorithm)**.**
Alogirthm to find a minimum spanning tree in a connected graph.

- *START with all the vertices of a weighted graph.*
- *Step 1: Choose and draw any vertex.*
- *Step 2: Find the edge of least weight joining a drawn vertex to a vertex not currently drawn. Draw this weighted edge and the corresponding new vertex .*
- *REPEAT Step 2 until all the vertices are connected, then STOP.*

Note: When there are two or more edges with the same weight, choose any of them. We obtain a connected graph at each stage of the algorithm.

Example 2.23 (Prim's algorithm).

Use Prim's Algorithm to find the minimum spanning tree in the connected graph below:



Prim's Algorithm Summary

Prim's algorithm can be summarised as follows:

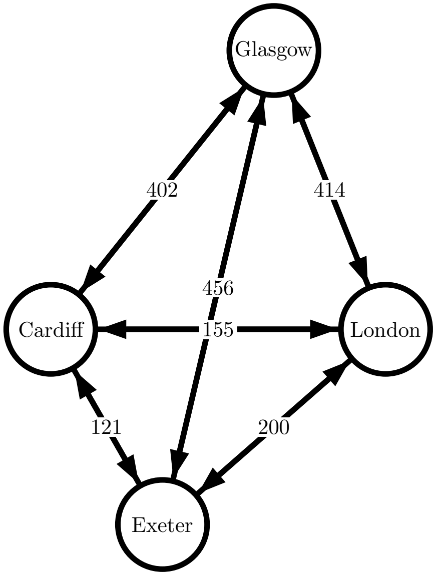
- Put an arbitrary vertex of G into T
- Successively add edges of minimum weight joining a vertex already in T to a vertex not in T until a spanning tree is obtained.

The Travelling Salesperson Problem

The travelling salesperson problem is one the most important problems in graph theory. It is simply stated as:

A travelling salesperson wishes to vistit a number of places and return to his starting point, selling his wares as he goes. He wants to select the route with the least total length. There are two questions:

- *Which route is the shortest one?*
- *What is the total length of this route?*



The Travelling Salesperson Problem: Maths Description

The mathematical description of this problem uses **Hamiltonian cycles**.

We also need the concept of a complete graph.

Definition 2.34 (Complete Graph)**.**

A **complete graph** *G is a graph in which each vertex is joined to each of the other vertices by exactly one edge.*

The travelling salesperson problem can then be more mathematically formally stated as:

Given a weighted, complete graph, find a minimum-weight Hamiltonian cycle.

The Travelling Salesperson Problem: Complexity

The travelling salesperson problem is quite complex due to the fact that the numbers of cycles to be considered *grows rapidly* with each **extra city** (vertex).

If for example the salesperson wanto to visit 100 cities, then

$$\frac{1}{2}(99!) = 4.65 * 10^{155}$$

cycles need to be considered.

This leads to the following problems:

- There is **no known efficient algorithm** for the travelling salesperson problem.
- Thus we just look for a good (!) **lower and/or upper bounds** for the length of a *minimum-weight Hamiltonian cycle.*

Upper bound of the travelling salesperson problem

To get an upper bound we use the following algorithm:

Algorithm 2.35 (The heuristic algorithm)**.** *The idea for the heuristic algorithm is similar to the idea of Prim's algorithm, except that we build up a cycle rather than a tree.*

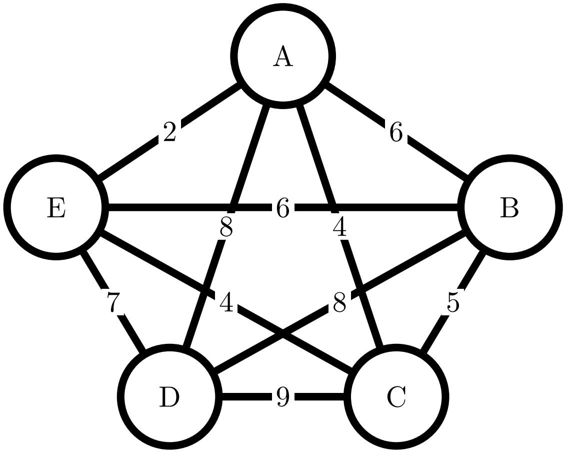
- *START with all the vertices of a complete weighted graph.*
- *Step 1: Choose any vertex and find a vertex joined to it by an edge of minimum weight. Draw these two vertices and join them with two edges to form a cycle. Give the cycle a clockwise rotation.*
- *Step 2: Find a vertex not currently drawn, joined by an edge of least weight to a vertex already drawn. Insert this new vertex into the cycle in front of the 'nearest' already connected vertex.*
- *REPEAT Step 2 until all the vertices are joined by a cycle, then STOP.*

The total weight of the resulting Hamiltonian cycle is then an upper bound for the solution to the travelling salesperson problem.

Be aware that the upper bound depends upon the city we start with.

Example 2.24 (Travelling salesperson problem: heuristic algorithm).

Use the heuristic algorithm to find an upper bound for the Travelling salesperson problem for the following graph:



Travelling Salesperson Problem: Heuristic Algorithm Summary

The heuristic algorithm can be summarized as follows:

To construct a cycle C that gives an upper bound to the travelling salesperson problem for a connected weighted graph G, build up the cycle C step by step as follows.

- Choose an arbitrary vertex of G and its 'nearest neighbour' and put them into C.
- Successively insert vertices joined by edges of minimum weight to a vertex already in C to a vertex not in C, until a Hamiltonian cycle is obtained.

Lower bound for the travelling salesperson problem

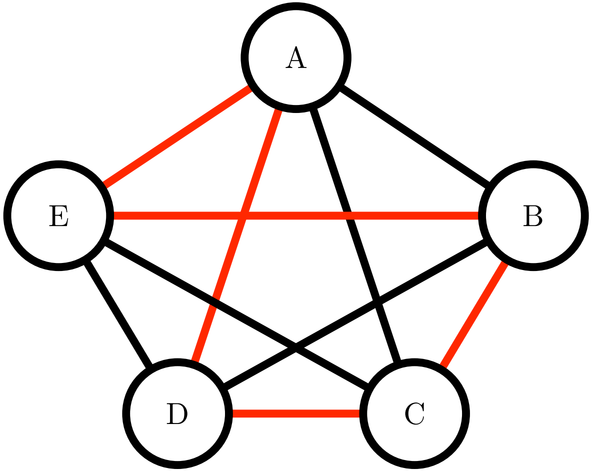
To get a better approximation for the actual solution of the travelling salesperson:

it is useful to get *not only* an **upper bound** for the solution **but also a lower bound**.

The following example outlines a simple method of how to obtain such a lower bound.

Example 2.25 (Lower bound for the travelling salesperson problem).

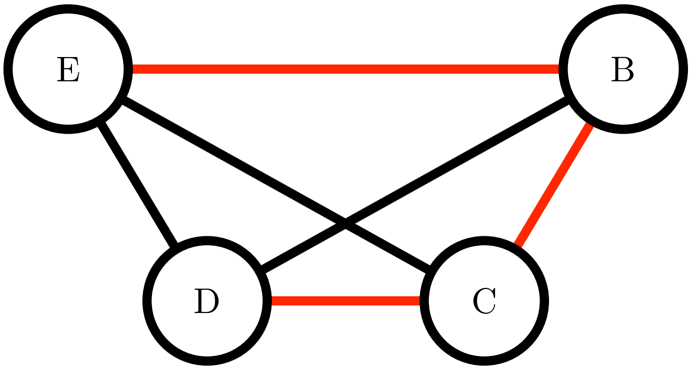
Consider the graph below:



Assume that $ADCBEA$ is a minimum weight Hamiltonian cycle.

Lower bound for the travelling salesperson problem example cont.

If we remove the vertex A from this graph and its incident edges: **we get a path passing through the remaining vertices.**

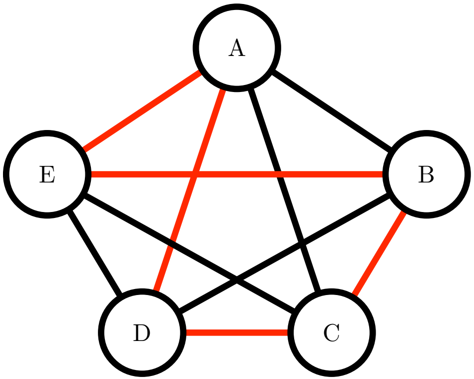


Such a path is certainly a **spanning tree** of the complete graph formed by these **remaining** vertices.

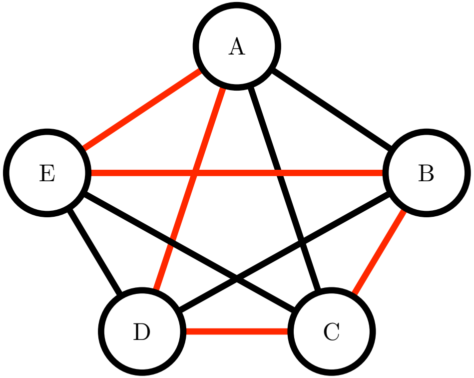
Lower bound for the travelling salesperson problem example cont.

Therefore the weight of the Hamiltonian cycle $ADCBEA$ is given by:

$$\texttt{total weight of Hamiltonian cycle} =$$
$$\texttt{total weight of spanning tree connecting } B, C, D, E$$
$$+ \texttt{weights of two edges incident with } A$$



Lower bound for the travelling salesperson problem example cont.



and thus:

$$\begin{aligned}&\texttt{total weight of Hamiltonian cycle} \geq \\ &\texttt{total weight of spanning tree connecting } B, C, D, E \\ &+ \texttt{weights of the } \textbf{two smallest} \texttt{ edges incident with } A\end{aligned}$$

The right hand side is therefore a lower bound for the solution of the travelling salesperson problem in this case.

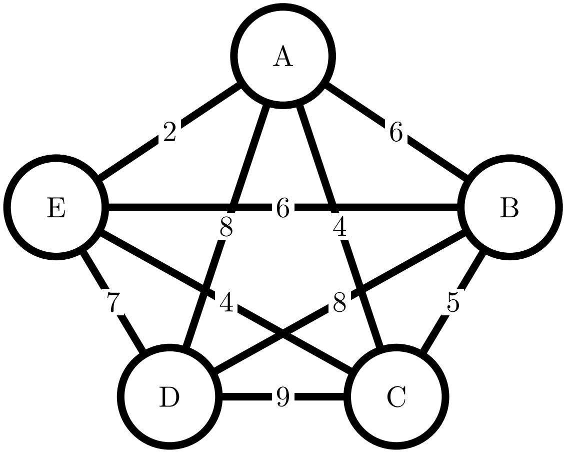
Algorithm 2.36 (Lower bound for the travelling salesperson problem).

- *Step 1: Choose a vertex V and remove it from the graph.*
- *Step 2: Find a minimum spanning tree connecting the remaining vertices, and calculate its total weight w.*
- *Step 3: Find the two smallest weights, w_1 and w_2, of edges incident with V.*
- *Step 4: Calculate the lower bound $w + w_1 + w_2$.*

Note: Different choices of the initial vertex V give different lower bounds.

Example 2.26 (Travelling salesperson problem: Lower Bound).

Use the Lower bound algorithm to find a lower bound for the Travelling salesperson problem for the following graph:



The Shortest Path Problem

This problem sounds very simple, again it is one of the most important problems in graph theory.

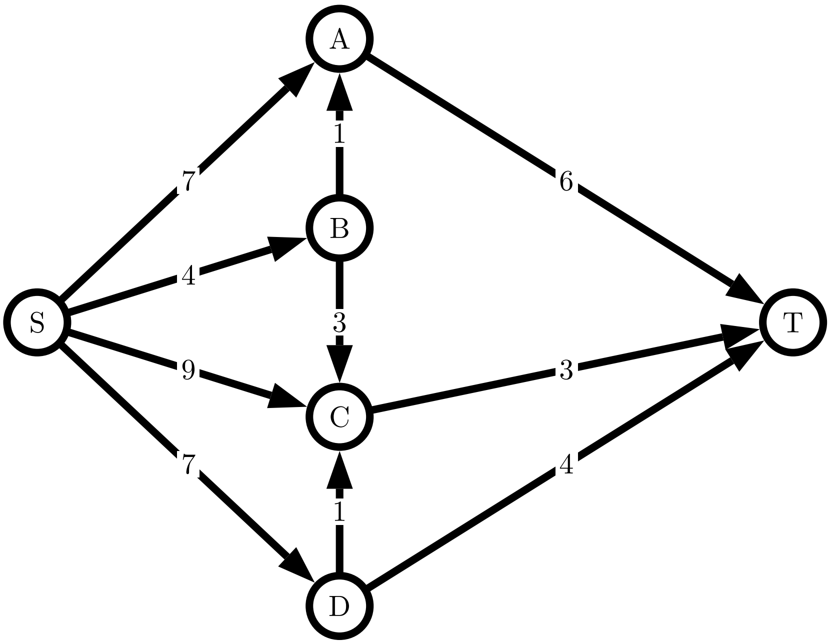
Problem 2.21 (Shortest path problem)**.**
Given a weighted graph G or a weighted digraph D, find the shortest path between two vertices of G or D.

We will only consider digraphs in this course.

To solve it we will use Dijktstra's algorithm.

Example 2.27 (Idea of Dijkstra's algorithm).

Consider the following digraph:



We want to find the shortest path from S to T.

Dijkstra's algorithm Example

The idea behind Dijkstra's algorithm is simple:

It is an iterative algorithm.

We start from S and calculate the shortest distance to each intermediate vertex as we go.

At each stage we assign to each vertex reached so far, a label reprsenting the distance from S to that vertex so far.

So at the start we assign S the potential 0.

Eventually each vertex acquires a permanent label, called potential, that represents the shortest path from S to that vertex.

We start each iteration from the vertex (or vertices) that we just assigned assigned a potential.

Once T has been assigned a potential, we find the shortest path(s) from S to T by tracing back through the labels.

Algorithm 2.37 (Dijkstra's algorithm).

Aim: Finding the **shortest path** *from a vertex S to a vertex T in a weighted digraph.*

- *START: Assign potential 0 to S.*
- **General Step**:
  1. *Consider the vertex (or vertices) just assigned a potential.*
  2. *For each such vertex V, consider each vertex W that can be reached from V along an arc VW and assign W the label*

  $$\texttt{potential of } V + \texttt{distance } VW$$

  unless W already has a smaller or equal label assigned from an earlier iteration.
  3. *When all such vertices W have been labelled, choose the smallest vertex label that is not already a potential and make it a potential at each vertex where it occurs.*
- *REPEAT the general step with the new potentials.*
- *STOP when T has been assigned a potential.*

The shortest distance from S to T is the potential of T.

To find the shortest path, trace backwards from T and include an arc VW whenever we have

$$\texttt{potential of } W - \texttt{potential of } V = \texttt{distance } VW$$

until S is reached.

Example 2.28 (Autoroute/Internet Routing). **Two real world examples we have already mentioned a few times are:**

Autoroute — apply Dijkstra's algorithm to work out how to go from place A to place B

Internet Routing — apply Dijkstra's algorithm to work out how to go from node A to node B

Two possible variations:

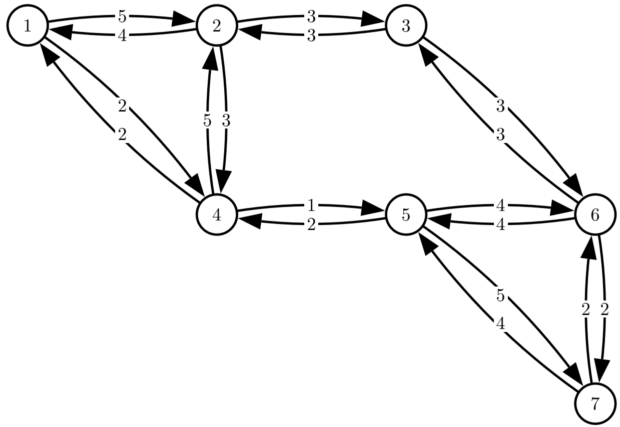
Static Routing — apply Dijkstra's algorithm to find optimal paths through a Digraph representation of the network.

Problem: Vulnerable if links or nodes modelled in network fail

Dynamic Routing — network continually monitors and updatdes link capacities. Each vertex maintains its own set of routing tables and so routing calculations can be *distributed* throughout the network.

Problem 2.22 (Internet Routing).

For the network graph below:



1. *Apply Dijkstra's algorithm to work out the best paths for vertex 1 to all other vertices. Represent this as a* shortest path tree.
2. *Construct a* *routing table* *to represent this information*
3. *Do the same for vertex 2* etc.
4. *Suppose the delay weight for vertex 2 to vertex 4 decreases from 3 to 1. How does this change the shortest path tree for vertex 2?*
5. *If the links between vertex 5 and 6 go down what happens to the shortest path trees and routing tables for vertices 1 and 2?*

Larger network graph for problem on previous page: