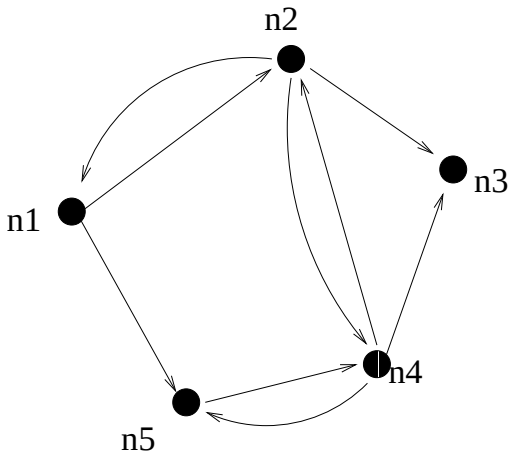
COMP20012

4: Graphs

Graphs

- A *directed graph* (digraph) is a pair (N, E) consisting of a set of *nodes* (or *vertices*) N, together with a relation E on N. Each element (n_1, n_2) of E is called an *edge*
- There is no restriction on the relation E.



- Recall equivalence between digraphs and relations from CS1021

- A *path* in a graph is a sequence of nodes $n_1, n_2, n_3 \ldots n_k$, where $(n_i, n_{i+1}) \in E$ for $1 \leq i < k$
- The length of a path is the number of edges in it ($k-1$ in the definition above)
- A *cycle* in a graph is a path of length at least 1, such that $n_1 = n_k$. Sometimes edges of graphs are *labelled*. The label may describe either the nature of the edge (e.g. name of train operator providing service on a rail network graph), or other attribute such as the distance or cost associated with the edge.

Representations of Directed Graphs

- If the number of nodes is fairly small and known before the graph's construction can use an *adjacency matrix*.
- Suppose $N = \{1, 2, \ldots n\}$
- The adjacency matrix for the graph $G = (N, E)$ is an $n \times n$ matrix A of booleans, where $A[i, j]$ is true if and only if there is an edge from i to j.

Representations of Directed Graphs

- Denoting true by 1 and false by 0 we have, for the graph above,

$$\begin{array}{ccccc} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{array}$$

- Given edge can be detected in constant time
- Easy to extend to labelled graph, just replace booleans by labels.
- Main disadvantage is that this requires $O(n^2)$ storage even if graphs has much fewer than n^2 edges.

Representations of Directed Graphs

- Alternative representation is *adjacency list*
- For each node n, we give a *list* of the nodes n', such that there is an edge (n, n').
- Easy to see how to implement this in `Java` or `C`.

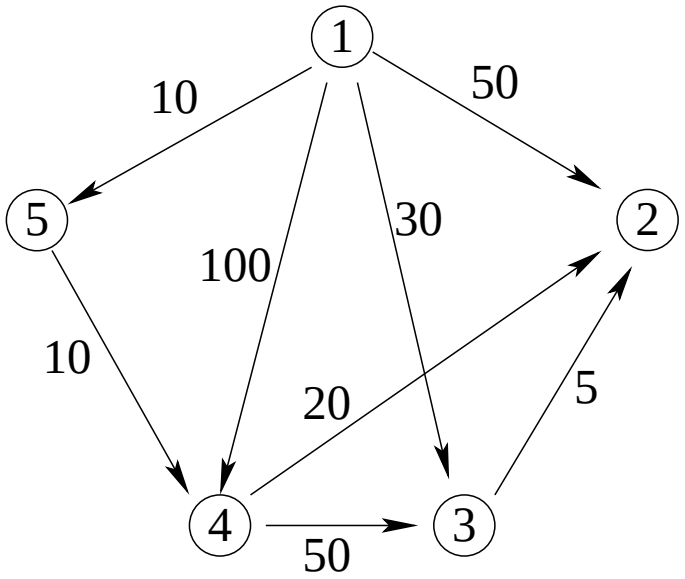
Single source shortest paths problem

Problem

Given a directed graph

$$G = \langle N, E \rangle$$

where each edge has an associated non-negative 'length' (or 'cost'), find the shortest (cheapest) distance from a single node (called the *source*) to each other node in the graph.



Use a *greedy* algorithm called *Dijkstra's* algorithm.

Edsger Dijkstra: 1930-2002

- 1972 recipient of the ACM Turing Award
- Responsible for the idea of building operating systems as explicitly synchronized sequential processes, for the formal development of computer programs, and for the intellectual foundations for the disciplined control of nondeterminacy.
- Designed and coded the first Algol 60 compiler.
- Leader of the movement to abolish the GOTO statement from programming.

Diversion: Greedy algorithms

Used to solve optimisation problems

- Have a problem, which has many possible solutions
- Want to find a solution which is in some sense ‘best’
- Want solution which minimises (or maximises) value of some function that measures *cost* or *value* of solution.
- Such function called the *objective function*

One example is the scheduling problem

- Given set of jobs

  $$j_1, j_2, \ldots j_n$$

  with running times

  $$t_1, t_2, \ldots t_n$$

- What is best way to schedule jobs in order to minimise average completion time?

Example: Coin choosing

Another example is change giving

- Want to give an amount of money using fewest number of coins/notes possible.
- Objective function is number of coins in selection
- What strategy do we usually adopt?
  – Choose largest coin available

For example

£86.68	=	£50 (leaving £36.68)
	+	£20 (leaving £16.68)
	+	£10 (leaving £6.68)
	+	£5 (leaving £1.68)
	+	£1 (leaving 68p)
	+	50 p (leaving 18p)
	+	10 p (leaving 8p)
	+	5 p (leaving 3p)
	+	2 p (leaving 1p)
	+	1 p

This is an example of a *greedy algorithm*

- Greedy algorithms can be used when a set (or list) of candidates is to be chosen to build up a solution (e.g. a set of coins, ordering of a set of jobs to be scheduled, set of edges of a graph).
- The greedy approach always makes what appears at the time to be the ‘best’ choice at each stage, without worrying about its effect on future choices.
- Once a candidate is included in the solution it is never removed, and once a candidate is excluded from the solution it is never reconsidered.
- No backtracking over choices.
- This makes greedy algorithms relatively simple and easy to implement.

But . . .

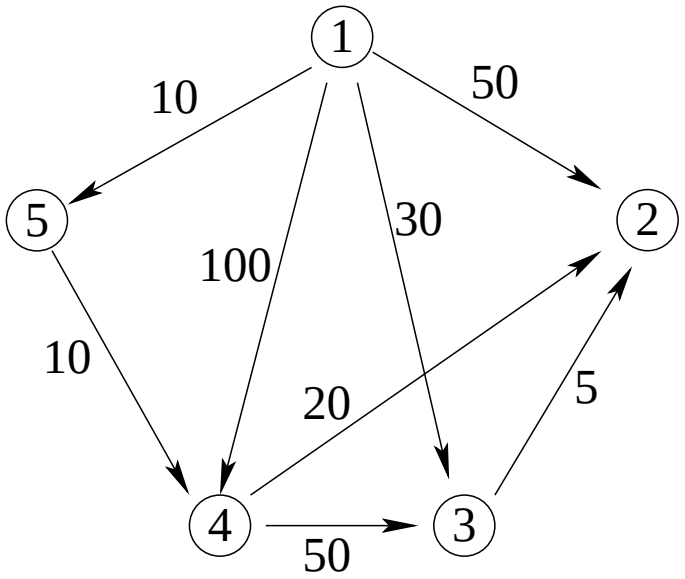
- Back to the coins example:-
- What happens if we introduce a 12p coin?

$$35p = ?$$

Back to Dijkstra

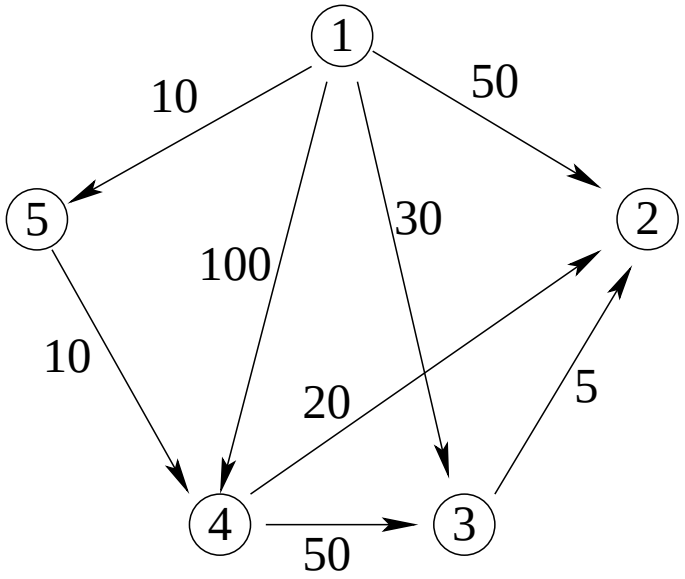
Want to find the shortest distance from a single node to each other node in the graph.

- Maintain a set S of *known* nodes
- A path from the source node to any other node is called *special* if uses only *known* nodes as intermediates
- At each step add a new vertex to S , and maintain a list D of lengths of shortest special paths from the source to every other node in the graph.
- At each step, add to S the node not currently in S which has the shortest special path.
- Adjust the values of D to take account of the new element of S



Step	Node to add	S	D
Init	-	{1}	[50,30,100,10]
1	5	{1,5}	[50,30,20,10]
2	4	{1,4,5}	[40,30,20,10]
3	3	{1,3,4,5}	[35,30,20,10]

- Note that this algorithm finds the shortest distance from the source to each node, but doesn't give us the routes.
- Can do this by adding a second array P which records the node that *precedes* the final node in each special path.



Step	Node to add	S	D	P
Init	-	{1}	[50,30,100,10]	[1,1,1,1]
1	5	{1,5}	[50,30,20,10]	[1,1,5,1]
2	4	{1,4,5}	[40,30,20,10]	[4,1,5,1]
3	3	{1,3,4,5}	[35,30,20,10]	[3,1,5,1]

Can build paths by working backwards from each node.

Description of algorithm

A pseudocode description of the algorithm is as follows (we maintain $C = N - S$ rather than S itself)

```
C = {2, 3, ... n}
for i = 2 to n do
    D[i] = L[1,i]

repeat n - 2 times
  v = some element of C with minimal D[v]
  C = C - {v}
  for each w in C do
    D[w] = min(D[w],D[v] + L[v,w])
```

Why does Dijkstra's algorithm work?

We will outline the proof that the algorithm does actually produce the right result

The proof is by induction. We prove that

a) if a node $i \neq 1$ is in S, then $D[i]$ is the length of the shortest path from the source to i

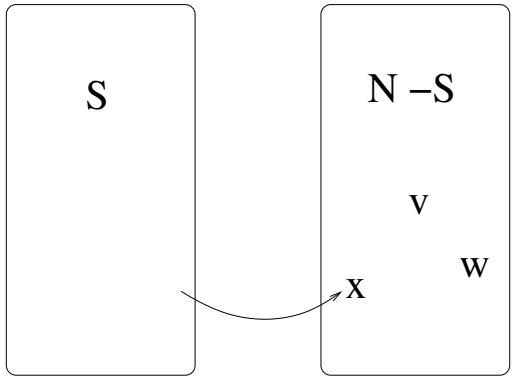
b) if a node $i \neq 1$ is *not* in S, then $D[i]$ is the length of the shortest *special* path from the source to i

Since S contains all nodes when the algorithm is complete this will prove the result.

- Base case: Initially only the source node is in S, so a) is obviously true. For all other nodes, the only special path is the direct path, and D contains the lengths of those paths, so b) is also true.

- Inductive step. Suppose a) and b) are both true at some stage, we show that adding a new node maintains both properties
  a) For every node in S before addition of v nothing changes
  Need to show that the $D[v]$ is the shortest path to v and not just the shortest special path
  Suppose that the shortest path to v is *not* a special path, ie it passes through some node not in S.

Let x be the first such node on the path from 1 to v.



The path from 1 to x is a special path, so its length is $D[x]$ (by inductive assumption).
If we go to v via x the distance must be at least $D[x]$, so $D[v]$ must be greater than $D[x]$
But v was chosen because it had smallest value of D, so we have a contradiction and so property a) is maintained

b) Now consider some w, different to v, not in S

- Need to show that $D[w]$ is the length of the *shortest* special path to w
- When v is added to S, we have two possibilities for shortest path to w, either it changes or it doesn't
- Suppose it changes, let x be the last node in S before w
- Length of this path is $D[x] + L[x, w]$
- For every x except v, we have already compared the old value of $D[w]$ with $D[x] + L[x, w]$ when we added x
- So only need to compare it with $D[v] + L[v, w]$
- This shows that condition b) is also maintained

Complexity of Dijkstra's algorithm

- Suppose graph has n nodes and e edges, using an adjacency matrix representation
- Initialisation take $O(n)$
- For each iteration, need to look at $n-1, n-2, n-3, \ldots$ values of $D[v]$
- This gives a total time of $O(n^2)$

Can improve on this by storing the values of D in a heap so that smallest value is always at the root.

- Initialisation take $O(n)$
- Removing smallest value from heap takes $O(\lg n)$
- Updating the heap also takes $O(\lg n)$. This happens at most once for each edge of the graph.
- This gives a time of $O((e+n)\lg n)$
- If the graph is *dense*, (i.e. e is close to n^2) the straightforward implementation is better
- If graph is *sparse* (e much smaller than n^2), the heap implementation is better

Huffman Codes

This section presents another greedy algorithm (not a graph algorithm) which makes an interesting use of trees

Problem

Information is contained in a file containing n characters, how can we store this information in as efficient a way as possible?

Need to know something about the contents.

- Suppose file contained only characters *a, b, c, d, e*, together with *blank* and *newline*
- Can encode these 7 characters in 3 bits

a	b	c	d	e	blank	newline
000	001	010	011	100	101	110

- Could store the information in original file in $3n$ bits (plus overhead for above table), rather than the usual $8n$ bits

Can do better if we use *variable length code* using the frequency of the characters – shorter codes for more frequent characters
Suppose frequencies and codes are

a	b	c	d	e	newline	blank
10	15	12	3	4	1	13
001	01	10	00000	0001	00001	11

Then need

$$10 \times 3 + 15 \times 2 + 12 \times 2 + 3 \times 5 + 4 \times 4 + 1 \times 5 + 13 \times 2 = 146 \text{ bits}$$

Standard representation needs $8 \times 58 = 464$ bits
Fixed length encoding requires $3 \times 58 = 174$ bits

In above variable length code, no character's code is a *prefix* of another's.
Such codes called *prefix codes*

a	b	c	d	e	newline	blank
10	15	12	3	4	1	13
001	01	10	00000	0001	00001	11

Sequence

01001111000101100011100010000 1

splits into

```
01|001|11|10|001|01|10|001|11|0001|00001|
```

which represents character sequence

```
"ba cabca e
"
```

Problem

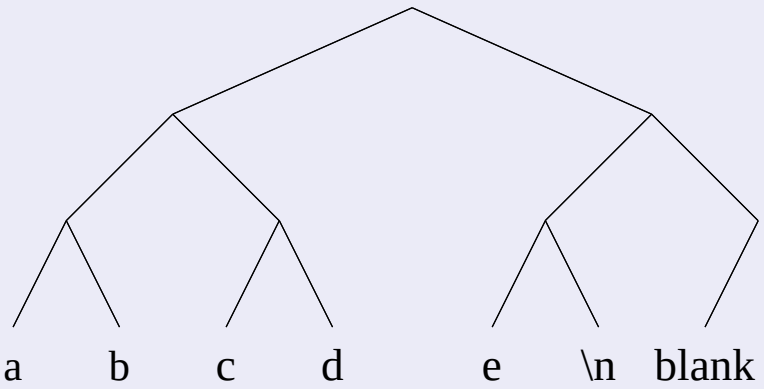
How do we generate such codes?

Can represent a binary code as a tree with data only at leaves (*trie*)

0 indicates left choice

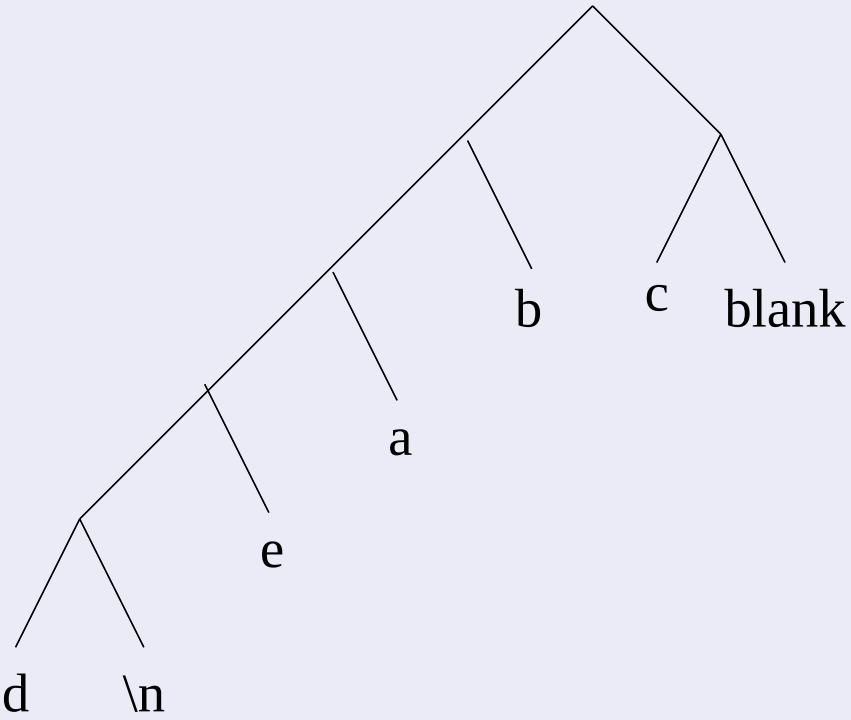
1 indicates right choice

Fixed length code



a	b	c	d	e	blank	newline
000	001	010	011	100	101	110

Prefix code



a	b	c	d	e	newline	blank
10	15	12	3	4	1	13
001	01	10	00000	0001	00001	11

Huffman's algorithm

Given a set of characters and frequencies, generate a prefix code

- Maintain a forest of trees, each with *weight* given by sum of frequencies at leaves
- Repeatedly merge two trees with *smallest weight* until have only one tree
- Start with forest of singleton trees

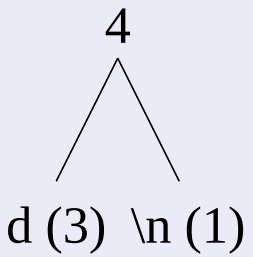
Huffman's algorithm

1

a (10) b (15) c (12) d (3) e (4) \n (1) bl (13)

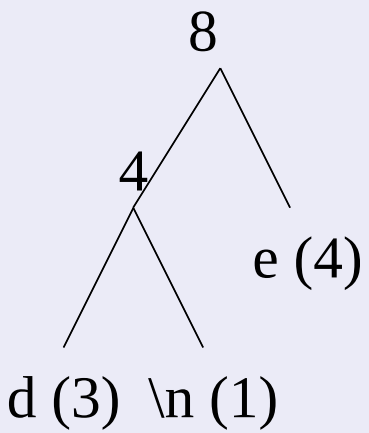
2

a (10) b (15) c (12) e (4) bl (13)



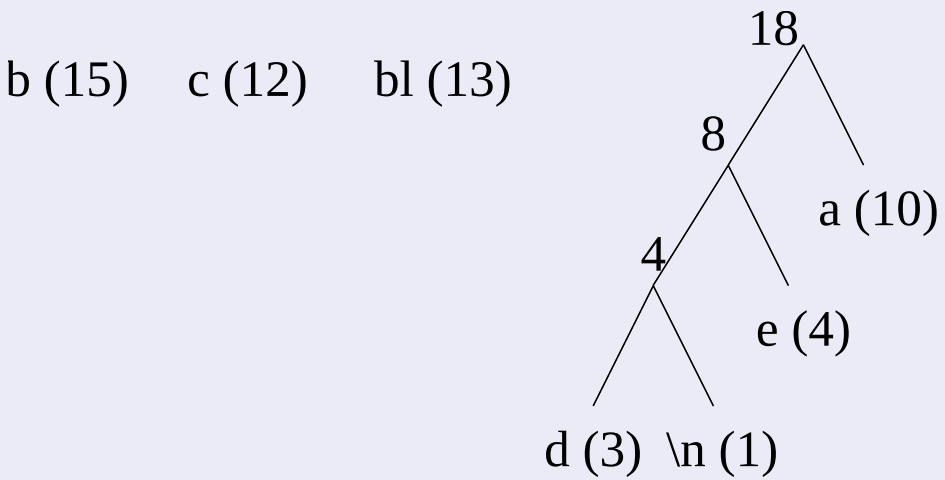
3

a (10) b (15) c (12) bl (13)

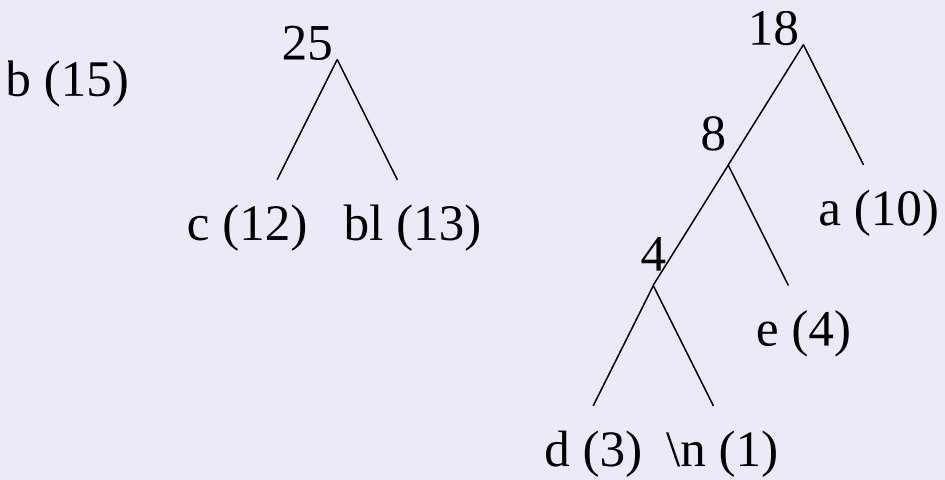


Huffman's algorithm

4

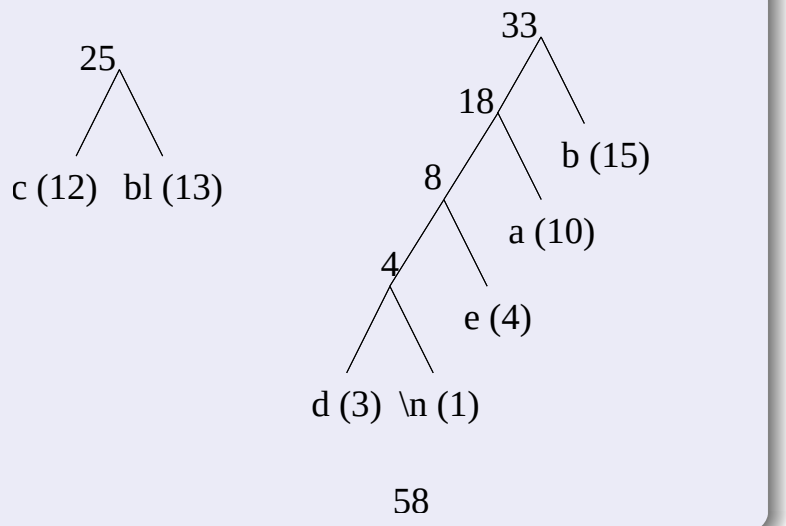


5

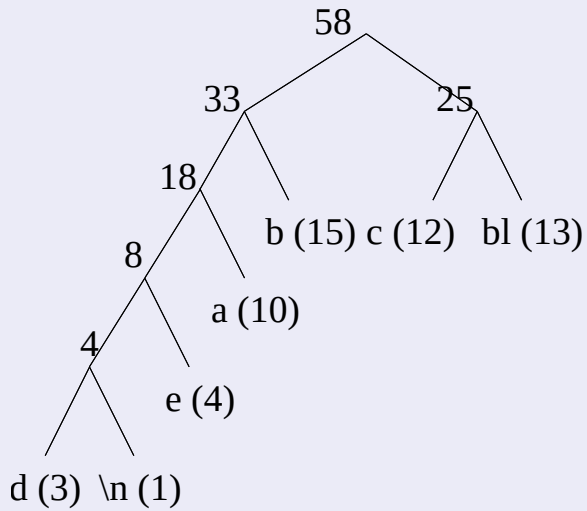


Huffman's algorithm

6



7



The code produced by Huffman's algorithm is, in one sense, the best code possible.

- The length of the encoded string is given by

$$\sum_{\textit{leaves}} (\text{frequency of char at leaf}) \times (\text{distance to root})$$

- This is the *weighted external path length* of the tree
- Claim that

No tree with the same frequencies at the leaves has lower weighted external path length than the Huffman tree

- The Huffman tree has the property that the two least frequently occurring characters α and β, must be at the deepest level of the tree
- Argue that if one of α and β not deepest then some other character γ must be and swapping γ with α (or β) would decrease weighted external path length
- Rest of the proof proceeds by induction

Compression techniques in action

- (This material is not in the syllabus, included merely for interest)
- Huffman encoding – used by unix utility `pack`.
- Adaptive Huffman encoding – used by (obsolete) unix utility `compact`.
- Lempel-Ziv encoding – used by unix utilities `compress` and `gzip`.

Gzip

- `gzip` finds duplicated strings in the input data. The second occurrence of a string is replaced by a pointer to the previous string, in the form of a pair (distance, length).
- A ‘sliding window’ of size 32K bytes is used, this means is that any given point in the data, there is a record of the previous 32K characters.
- The sliding window is split into 2 parts
- a *dictionary* – which is before the cursor
- a *lookahead buffer* – which starts at the cursor and goes forward.

When the next sequence of characters in the lookahead buffer to be compressed is identical to one starting in the dictionary, the sequence of characters is replaced by two numbers:

- a distance: representing how far back into the window the sequence starts,
- a length: representing the number of characters for which the sequence is identical.

For example

```
Blah blah blah blah blah!
```

The first 5 characters contain no repetitions, so appear verbatim
But looking at the next five characters:

```
 *****
Blah blah blah blah blah!
      *****
```

There is an exact match for those five characters within the sliding window, it starts five characters behind the point where we are now.

The data so far:

```
Blah blah b
```

The compressed form of the data so far:

```
Blah b[D=5,L=5]
```

The compression can be increased even further

- Now look at the character that follows the strings we've identified as identical.
- In both cases, it's 'l' – so we can make the length 6, and not just 5.
- Continuing in this way we eventually find that the 18 characters that start at the second character are identical to the 18 characters that start at the seventh character.
- We don't worry that these two strings overlap

It turns out our data can be compressed down to just this:

```
Blah b[D=5, L=18]
```

Think about how this would be decompressed

- Literals or match lengths are compressed with one Huffman tree, and match distances are compressed with another tree.
- The trees are stored in a compact form at the start of each block.
- Duplicated strings are found using a hash table.

The following example uses a dictionary of size 6 and lookahead buffer of size 4.

The cursor position is boxed, the dictionary is **bold faced** and the lookahead buffer is underlined.

Step	Input String															Output
1	x	x	z	x	x	z	x	y	z	x	y	x	y	x	z	x
2	**x**	x	z	x	x	z	x	y	z	x	y	x	y	x	z	(1,1,z)
3	**x**	**x**	**z**	x	x	z	x	y	z	x	y	x	y	x	z	(3,4,y)
4	x	x	**z**	**x**	**x**	**z**	**x**	**y**	z	x	y	x	y	x	z	(3,3,x)
5	x	x	z	x	x	z	**x**	**y**	**z**	**x**	**y**	**x**	y	x	z	(2,2,z)

Much more information on compression related matters can be found on the Info-Zip web site, `http://www.mirror.ac.uk/sites/ftp.cdrom.com/pub/infozip/`

All pairs shortest paths

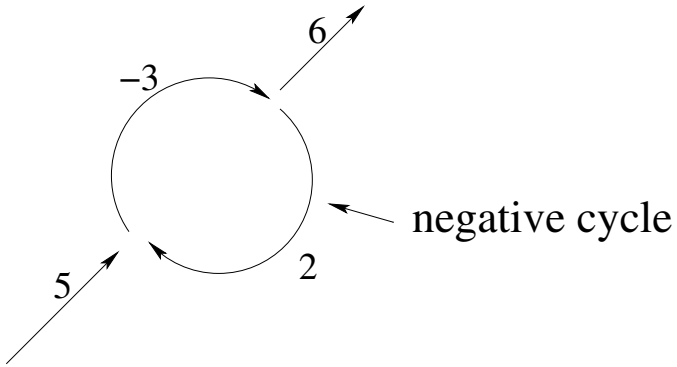
Back to graphs.

- **Problem**
  Given a directed graph

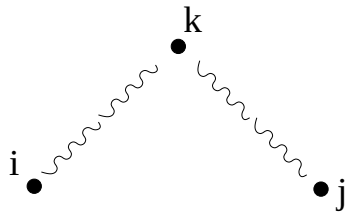
$$G = \langle N, E \rangle$$

  where each edge has an associated 'length' (or 'cost'), find the shortest (cheapest) distance between *each pair* of nodes in the graph.
- Recall that Dijkstra's algorithm gave shortest distance from a given node to all other nodes. Could use that n times (n number of nodes)
- The algorithm we now give is *much* simpler and can be used when graph has negative edges (Dijkstra's algorithm can't)

Although negative edges are allowed, we do not allow negative cycles (Why?)



- Suppose shortest path from *i* to *j* passes through *k*



- Then the path taken from *i* to *k* must be the shortest available
- So must path taken from *k* to *j* (Why?)

- Define function $d(k,i,j)$ as

$$d(k,i,j) \quad = \quad \text{shortest distance from } i \text{ to } j \text{ using only nodes } 1 \ldots k \text{ as intermediate points}$$

- If given lengths of edges are $L(i,j)$, then

$$d(0,i,j) = L(i,j)$$

  i.e. use direct paths only, no intermediate nodes

- How do we calculate $d(k,i,j)$ for $k > 0$?

Want to find the shortest path from i to j (using only $1 \ldots k$ as intermediates)

Have two possibilities

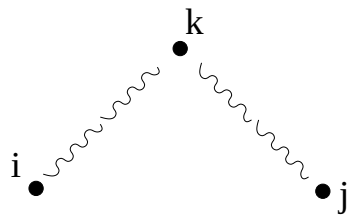
- The path doesn't actually use the node k
- It does use the node k

In first case, use only intermediate nodes $1 \ldots (k-1)$ and length of path is just

$$d(k-1,i,j)$$

In second it is

$$d(k-1,i,k) + d(k-1,k,j)$$



This gives

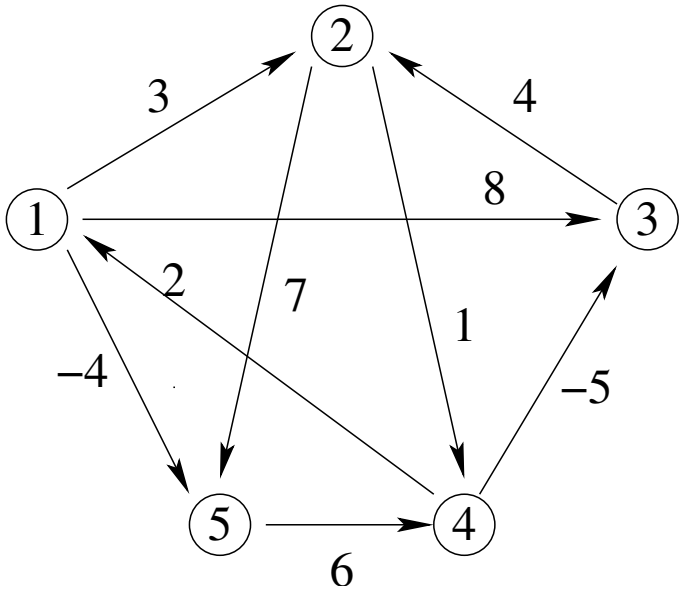
$$d(k,i,j) = \min(d(k-1,i,j), d(k-1,i,k) + d(k-1,k,j))$$

- Can do this recursively, but most efficient way to calculate this is bottom up
- For each k, $d(k)$ depends on the values of $d(k-1)$, so calculate $d(0)$, $d(1)$ and store the values for use in the next step
- This is the Floyd-Warshall algorithm

Robert W Floyd

- Won the Turing award in 1978.
- He was one of the inventors of the deterministic linear time selection algorithm.
- Also made early improvements in quicksort and quickselect.

Floyd-Warshall – an example



Store $d(k, i, j)$ in a matrix D_k

D_0

0	3	8	∞	-4
∞	0	∞	1	7
∞	4	0	∞	∞
2	∞	-5	0	∞
∞	∞	∞	6	0

P_0

*	1	1	*	1
*	*	*	2	2
*	3	*	*	*
4	*	4	*	*
*	*	*	5	*

D_1

0	3	8	∞	-4
∞	0	∞	1	7
∞	4	0	∞	∞
2	$\underline{5}$	-5	0	$\underline{-2}$
∞	∞	∞	6	0

P_1

*	1	1	*	1
*	*	*	2	2
*	3	*	*	*
4	$\underline{1}$	4	*	$\underline{1}$
*	*	*	5	*

$$d(1,i,j) = \min(d(0,i,j), d(0,i,1) + d(0,1,j))$$

$$d(2,i,j) = \min(d(1,i,j), d(1,i,2) + d(1,2,j))$$

D_2

0	3	8	$\underline{4}$	-4
∞	0	∞	1	7
∞	4	0	$\underline{5}$	$\underline{11}$
2	5	-5	0	-2
∞	∞	∞	6	0

P_2

*	1	1	$\underline{2}$	1
*	*	*	2	2
*	3	*	$\underline{2}$	$\underline{2}$
4	1	4	*	1
*	*	*	5	*

D_3

0	3	8	4	-4
∞	0	∞	1	7
∞	4	0	5	11
2	$\underline{-1}$	-5	0	-2
∞	∞	∞	6	0

P_3

*	1	1	2	1
*	*	*	2	2
*	3	*	2	2
4	$\underline{3}$	4	*	1
*	*	*	5	*

$$d(3,i,j) = \min(d(2,i,j), d(2,i,3) + d(2,3,j))$$

$$d(4,i,j) = \min(d(3,i,j), d(3,i,4) + d(3,4,j))$$

D_4

0	3	$\underline{-1}$	4	-4
$\underline{3}$	0	$\underline{-4}$	1	$\underline{-1}$
$\underline{7}$	4	0	5	$\underline{3}$
2	-1	-5	0	-2
$\underline{8}$	$\underline{5}$	$\underline{1}$	6	0

P_4

*	1	$\underline{4}$	2	1
$\underline{4}$	*	$\underline{4}$	2	$\underline{1}$
$\underline{4}$	3	*	2	$\underline{1}$
4	3	4	*	1
$\underline{4}$	$\underline{3}$	$\underline{4}$	5	*

D_5

0	$\underline{1}$	$\underline{-3}$	$\underline{2}$	-4
3	0	-4	1	-1
7	4	0	5	3
2	-1	-5	0	-2
8	5	1	6	0

P_5

*	$\underline{3}$	$\underline{4}$	$\underline{5}$	1
4	*	4	2	2
4	3	*	2	2
4	3	4	*	1
4	3	4	5	*

$$d(5,i,j) = \min(d(4,i,j), d(4,i,5) + d(4,5,j))$$

- The matrix D_5 contains the lengths of all the shortest paths, but doesn't record the paths themselves
- Can find the paths by recording the choices that change value of d

Define $p(k,i,j)$ to be the last node visited (before j) on the path from i to j (using only vertices $1 \ldots k$ as intermediates)

$$\begin{aligned} p(k,i,j) &= p(k-1,i,j) && \text{if } d(k,i,j) \text{ is same as } d(k-1,i,j) \\ &= p(k-1,k,j) && \text{otherwise} \end{aligned}$$

See the matrices $P_0 \ldots P_5$ in the above example
A Java applet showing an animated version of this algorithm can be found at `http://www.cs.man.ac.uk/~graham/COMP20012`

The Floyd-Warshall algorithm

```
public class Floyd
{
  int nodes;
  int [][] edgeLength;
  int [][] predecessor;
  int [][] distance;

  final static int inf = 1000000;
  final static int nonVertex = -1;
```

```
Floyd (int [][] edges)
  {
    edgeLength = edges;
    nodes = edges.length;
    distance = new int[nodes][nodes];
    predecessor = new int[nodes][nodes];
  }
```

```
void floyd()
{
  int i, j, k;

  /* Initialize distance and predecessor */
  /* predecessor[i][j] contains the last but one */
  /* vertex on the path from i to j */
  for( i=0; i < nodes; i++ )
    for( j=0; j < nodes; j++ )
      {
        distance[i][j] = edgeLength[i][j];
        if (i != j && edgeLength[i][j] != inf)
          predecessor[i][j] = i;
        else
          predecessor[i][j] = nonVertex;
      }
```

```
    for( k=0; k < nodes; k++ )
      {
        /* Consider each vertex as an intermediate */
        for( i=0; i < nodes; i++ )
          for( j=0; j < nodes; j ++ )
            if( distance[i][k] + distance[k][j] < distance[i][j] )
              { /*update min and predecessor*/
                distance[i][j] = distance[i][k] + distance[k][j];
                predecessor[i][j] = predecessor[k][j];
              }
%   printArray(distance);
      }
  }
```

```
int printShortestPath(int i,int j)
{
  /* Prints out the shortest path from i to j  */
  /* Returns the length of the shortest path  */
  /* (purely for checking purposes)  */
  int len,rest;
  len = 0;
  if (i == j)
    {
      System.out.print(i + " ");
      len =0;
    }
  else
```

```
    if (predecessor[i][j] == nonVertex)
      System.out.println("No path from " + i + " to " + j);
    else
      {
        rest=printShortestPath(i,predecessor[i][j]);
        System.out.print(j + " ");
        len=rest+edgeLength[predecessor[i][j]][j];
      }
  return(len);
}
```

Floyd-Warshall algorithm – example

```
Edge lengths          |Shortest path length   | Predecessor matrix
  0   3   8 Inf  -4 |  0   1  -3   2  -4    |  -1   2   3   4   0
Inf   0 Inf   1   7 |  3   0  -4   1  -1    |   3  -1   3   1   0
Inf   4   0 Inf Inf |  7   4   0   5   3    |   3   2  -1   1   0
  2 Inf  -5   0 Inf |  2  -1  -5   0  -2    |   3   2   3  -1   0
Inf Inf Inf   6   0 |  8   5   1   6   0    |   3   2   3   4  -1

Shortest paths
0 1: 0 4 3 2 1 |  2 0: 2 1 3 0    | 4 0: 4 3 0
0 2: 0 4 3 2   |  2 1: 2 1        | 4 1: 4 3 2 1
0 3: 0 4 3     |  2 3: 2 1 3      | 4 2: 4 3 2
0 4: 0 4       |  2 4: 2 1 3 0 4  | 4 3: 4 3
1 0: 1 3 0     |  3 0: 3 0        |
1 2: 1 3 2     |  3 1: 3 2 1      |
1 3: 1 3       |  3 2: 3 2        |
1 4: 1 3 0 4   |  3 4: 3 0 4      |
```

Dynamic programming

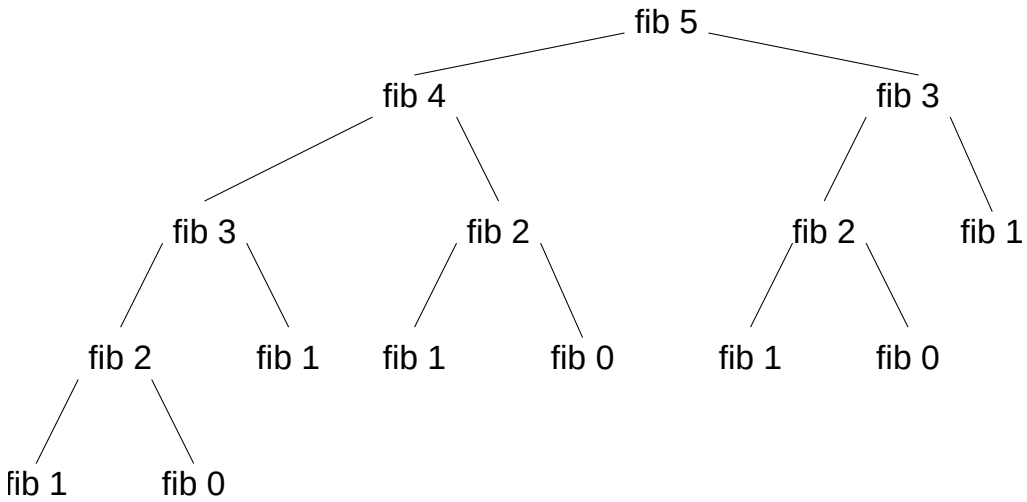
- The bottom-up approach adopted in this algorithm is typical of a family of algorithms which adopt a method called *dynamic programming*
- Basic idea behind dynamic programming is organisation of work in order to *avoid repetition of work already done*
- Recall the Fibonacci numbers (again)

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_{k+1} &= F_k + F_{k-1} \end{aligned}$$

Obvious recursive function to calculate F_k is

```
static int fib(int n) {
  if (n == 0) return 0;
  if (n == 1) return 1;
  return(fib(n-1) + fib(n-2));
}
```

- Consider calculation of $fib(5)$, have following recursive calls



- $fib(3)$ calculated twice
- $fib(2)$ calculated three times
- Know that, in order to calculate $fib(k)$, need to calculate

$$fib(0), fib(1), fib(2) \ldots fib(k-1)$$

- Why not calculate all of them *just once*?

```
static int fib (int n)
{
  int []  Fib = new int[n];

  Fib[0] = 0;
  Fib[1] = 1;

  for (i=2; i<=n ; i++)
    Fib[i] = Fib[i-1] + Fib[i-2];

  return(Fib[n]);
}
```

In fact, don't need to store them all, since only need previous 2 values, to calculate $fib(i)$. (Exercise: Code this)

- Original, recursive approach is *top down*
  This is *bottom up*
- Only works because
  - Relatively small number of subproblems
  - Know in advance which subproblems need solving
  - Many shared subproblems
- Dynamic programming is an approach to solving problems that uses this bottom-up, table-based style.

Memoisation

- Main benefit of dynamic programming approach is that don't have to repeat calculations already done – just decide in advance which calculations need to be done and do each of them once, recording the answer
- Disadvantage is that lose simplicity of top-down divide and conquer approach – have to organise bottom-up calculations.
- Can obtain the benefits of both by using an auxiliary data structure to remember values already calculated

For example, the Fibonacci function (yet again)

- Use an array to record values already calculated
- If value already calculated, just look it up
- Otherwise, calculate using the recursive definition and record the result

This process called *memoisation*

- A further *advantage* of memoised code is that *only* the values of the function that are needed are ever calculated.
- Some dynamic programming algorithms sometimes calculate unneeded values in order to keep the bottom-up organisation simple (the binary knapsack is a good example of this).

Memoised Fibonacci function

```
static final int MAXN = ??
static int fib_tab[MAXN];

static int fib (int n)
{
  if (fib_tab[n] == DUMMY)
    /* Calculate new value   */
    fib_tab[n] = fib(n-1) + fib(n-2);

  return(fib_tab[n]);
}
```

The table needs to be initialised before use

```
static void init_fib ()
{
  fib_tab[0] = 0;
  fib_tab[1] = 1;
  for (int i=2; i<MAXN ; i++)
    fib_tab[i] = DUMMY;
}
```

Disadvantages

Memoised divide and conquer implementations usually less efficient than hand-coded dynamic programming version
This is due to

- Recursion overheads
- Space taken is often more than is strictly necessary – e.g. in `fibonacci`, the memoised version uses $O(n)$ space, but the bottom-up approach uses only $O(2)$.

Traversing a graph

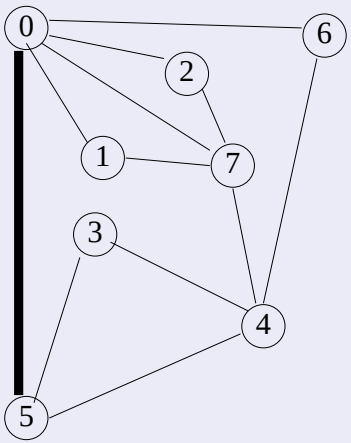
- Often need to visit all nodes in graph, need a systematic way of doing this
- One technique is *depth first search*
- Initially mark all nodes as *unvisited*
- Select a node *n* as start node and mark it as *visited*
- Each node adjacent to *n* is used as start node for a depth first search
- Once all nodes accessible from *n* have been visited, the search from *n* is complete
- If not all nodes have been visited need to pick a new, unvisited start node

- The following implementation assumes a representation of graphs using adjacency lists
- The nodes are indexed by integers $0 \ldots n$ and for each node we have a linked list of adjacent nodes
- The algorithm has a natural recursive implementation

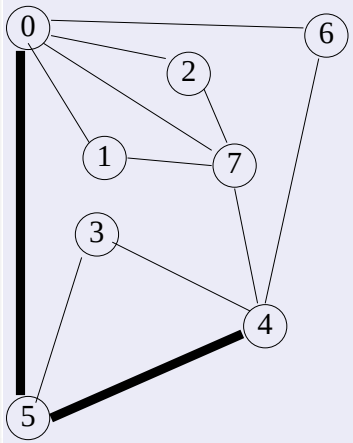
```
void dfs (int v) {
  visited[v] = true;
  System.out.println("Visiting " + v);

  for (ListNode n = graph.adjList[v]; n != null; n = n.next)
    {
      int v1 = n.vertexNum;
      if (!visited[v1])
        dfs(v1);
    }
  System.out.println("Finished " + v);
}
```

Depth First Search

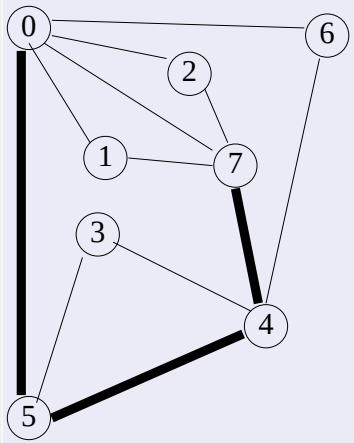
1



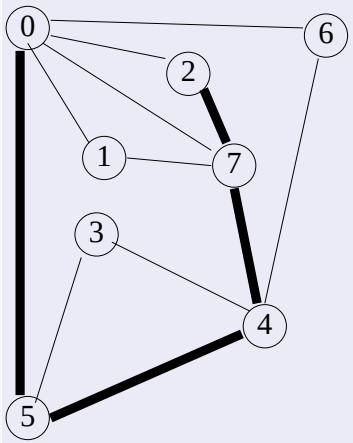
2



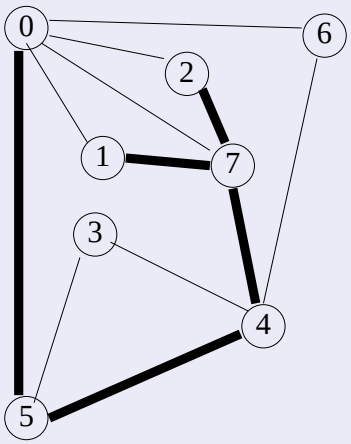
3



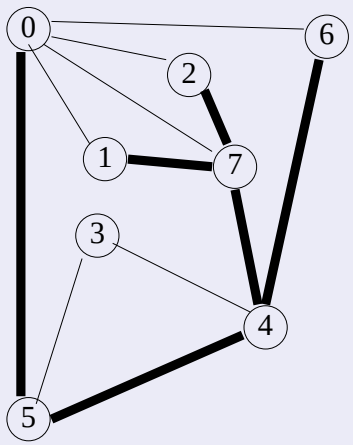
4



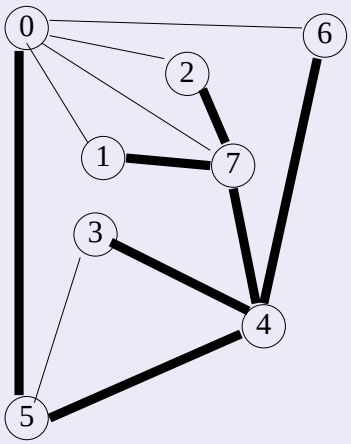
5



6



7

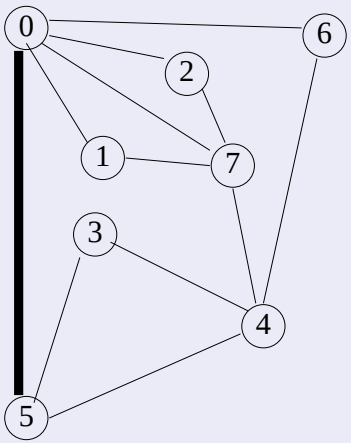


- Alternatively, use *breadth first search*
- Can't use the recursion stack to do all the work here, need a queue in which to store nodes awaiting a visit

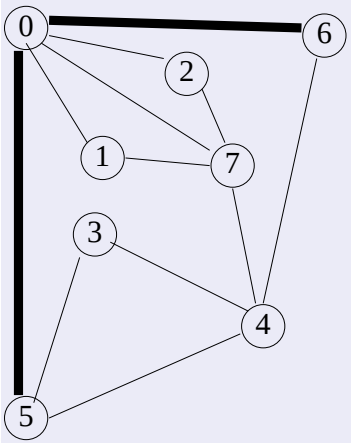
```
void bfs (int v) {
  IntQueue q = new IntQueue();
  q.enqueue(v);

  while (!q.isEmpty()) {
    int k = q.dequeue();
    if (!visited[k]) {
      System.out.println("Visiting " + k);
      visited[k] = true;
      for (ListNode n = graph.adjList[k]; n != null; n = n.next) {
        int neighbour = n.vertexNum;
        if (!visited[neighbour])
          q.enqueue(neighbour);
      }
    }
  }
}
```

Breadth First Search

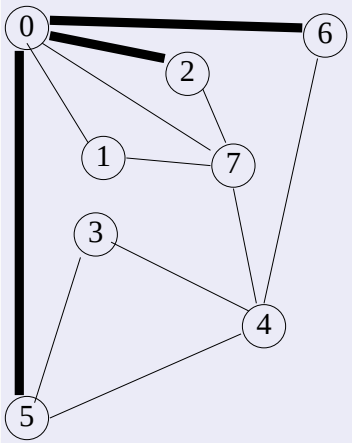
1



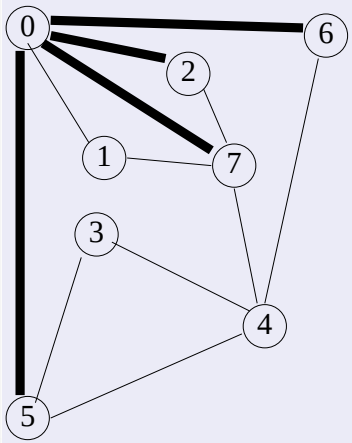
2



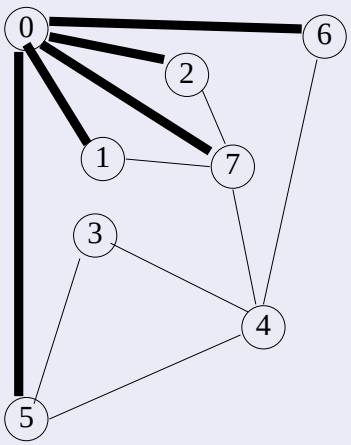
3



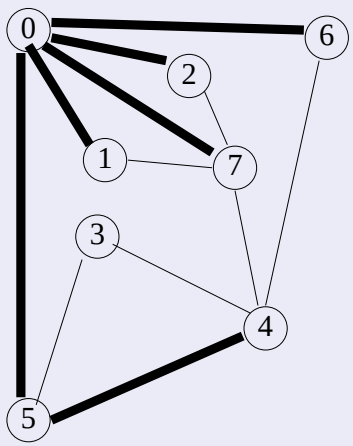
4



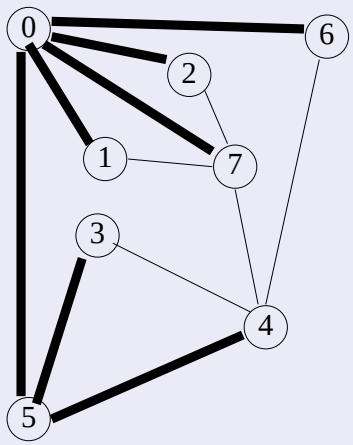
5



6



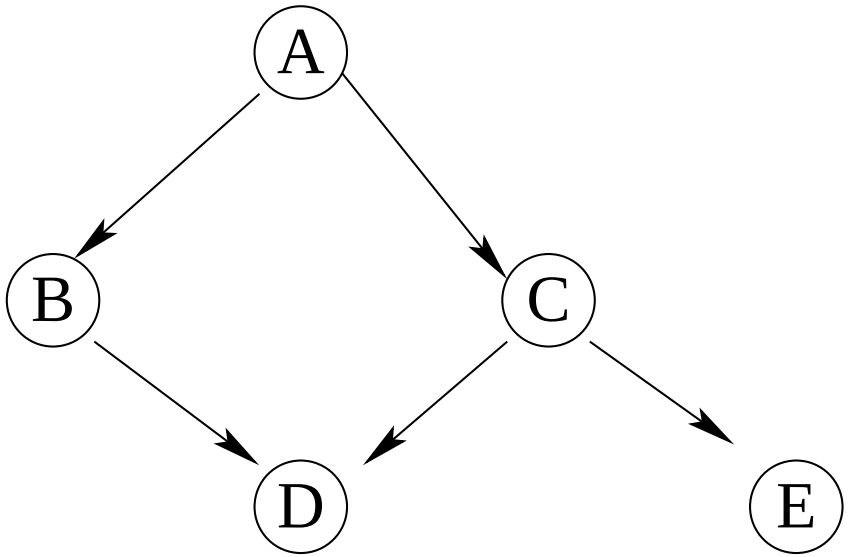
7



Directed Acyclic Graphs

A *Directed Acyclic Graph*, or *dag*, is a directed graph which contains no cycles.

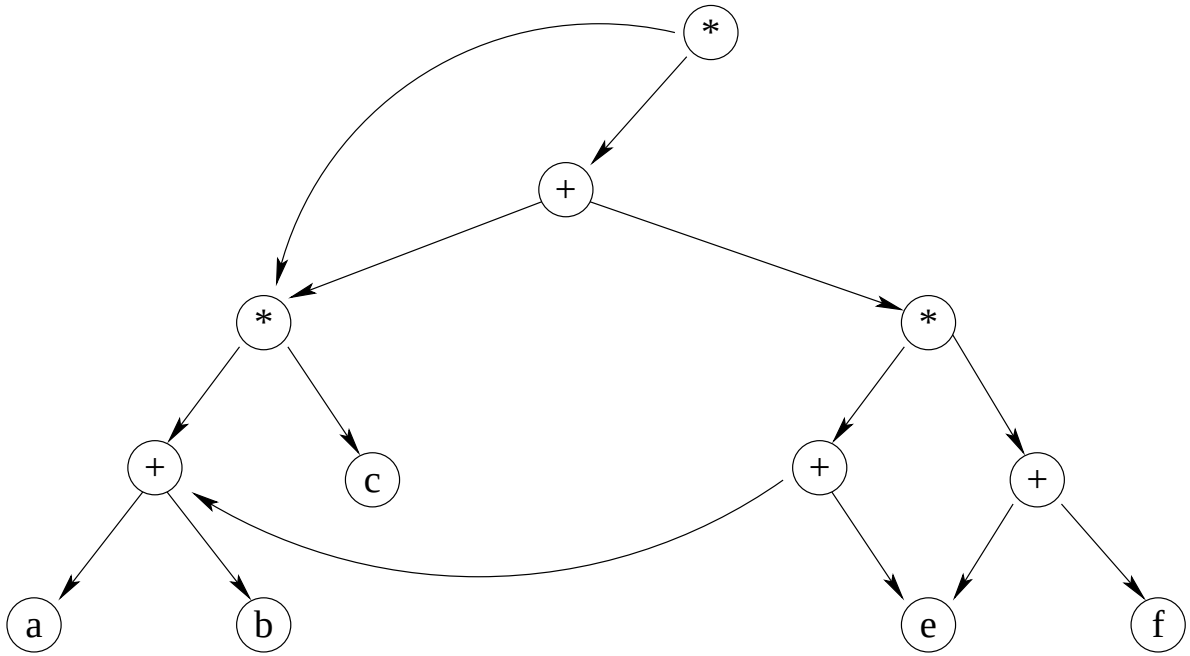
Every tree is a dag, but not every dag is a tree



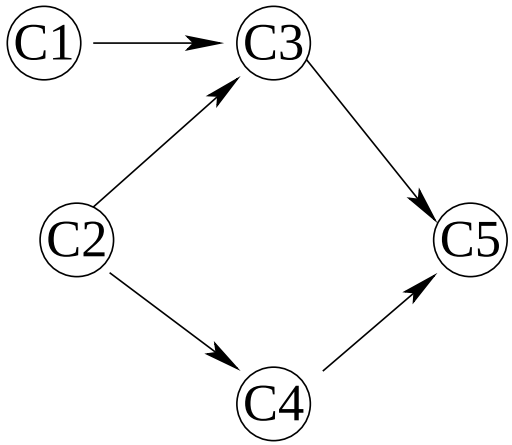
Useful for representing tree-like structures *with sharing*

For example, algebraic expressions

$$((a+b)*c+((a+b)+e)*(e+f))*((a+b)*c)$$



- Also used for expressing *dependencies*
- Large project can be split into number of smaller tasks, where one task can depend on the completion of another.
- Such dependencies are represented by a dag (why acyclic?)
- For example course prerequisites

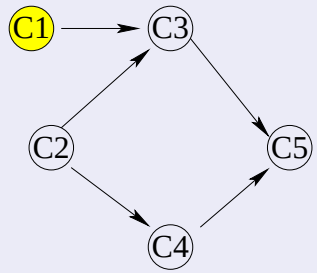


Topological Sort

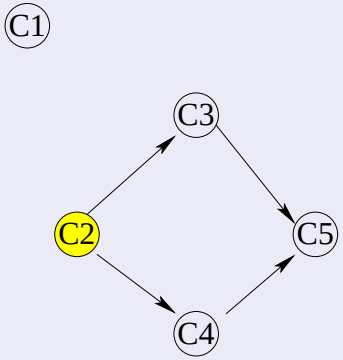
- A topological sort is process of assigning a *linear* ordering to vertices of a dag so that if there is an edge from i to j, then i appears before j in the order
- For example C1, C2, C3, C4, C5 is topological sort of course dependency dag
- Can take courses in that order without breaking dependencies Other orders possible

- Define the *indegree* of a vertex to be the number of incoming edges it has
- One way to implement topological sort is to find a vertex of indegree 0, add this vertex to our list, and then delete that vertex and all edges from it.
- Keep repeating this process until we run out of vertices.

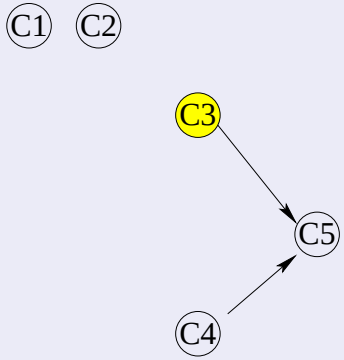
1


C1
C3
C2
C5
C4

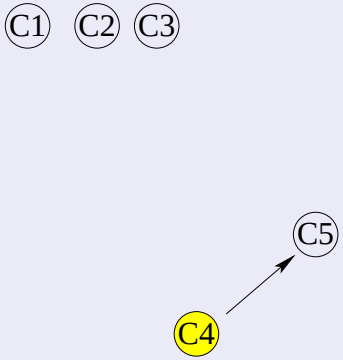
2


C1
C3
C2
C5
C4

3


C1
C2
C3
C5
C4

4


C1
C2
C3
C5
C4

5


C1
C2
C3
C4
C5

6


C1
C2
C3
C4
C5

In the implementation, we don't actually modify the graph by deleting edges, we just simulate this by modifying the indegree of relevant vertices.

```
public class TopSort {
  int [] inDegree;
  int [] topOrder;
  Graph graph;

  TopSort(Graph g) {
    graph = g;
    inDegree = new int[g.size];
    topOrder = new int[g.size];

    for (int v = 0; v < g.size; v++)
      for (ListNode n = graph.adjList[v]; n != null; n = n.next) {
        int neighbour = n.vertexNum;
        inDegree[neighbour]++;
      }
  }
```

```
private int findNewVertexOfDegreeZero ()
  {
    for (int i = 0; i < graph.size; i++)
      if (inDegree[i] == 0)
        {
          inDegree[i]--;
          return(i);
        }
    return(-1);
  }
```

```
void topSort () {
    int v;
    int count = 0;

    while ((v = findNewVertexOfDegreeZero()) >= 0)
      {
        topOrder[count++] = v;

        for (ListNode n = graph.adjList[v]; n != null; n = n.next)
          {
            int neighbour = n.vertexNum;
            inDegree[neighbour]--;
          }
      }
  }
```

We can improve the efficiency of this algorithm by storing the vertices of indegree 0 on a queue.

```
public class TopSort2 {
  int [] inDegree;
  int [] topOrder;
  Graph graph;
  IntQueue zeroQ;
```

```
TopSort2(Graph g) {
    graph = g;
    inDegree = new int[g.size];
    topOrder = new int[g.size];
    zeroQ = new IntQueue(g.size);
    for (int v = 0; v < g.size; v++)
      for (ListNode n = graph.adjList[v]; n != null; n = n.next) {
        int neighbour = n.vertexNum;
        inDegree[neighbour]++;
        }
    for (int v = 0; v < g.size; v++)
      if (inDegree[v] == 0)
        zeroQ.enqueue(v);
  }
```

```
private int findNewVertexOfDegreeZero ()
  {
    return(zeroQ.dequeue());
  }
```

```
void topSort () {
  int v;
  int count = 0;

  while (! zeroQ.isEmpty()) {
    v = findNewVertexOfDegreeZero();
    topOrder[count++] = v;

    for (ListNode n = graph.adjList[v]; n != null; n = n.next) {
      int neighbour = n.vertexNum;
      inDegree[neighbour]--;
      if (inDegree[neighbour] == 0)
        zeroQ.enqueue(neighbour);
    }
  }
}
```

This implementation will give the ordering
C1, C2, C4, C3, C5
for the course dependency graph, since the order in which vertices appear is the order in which they are put on the queue.

Undirected graphs

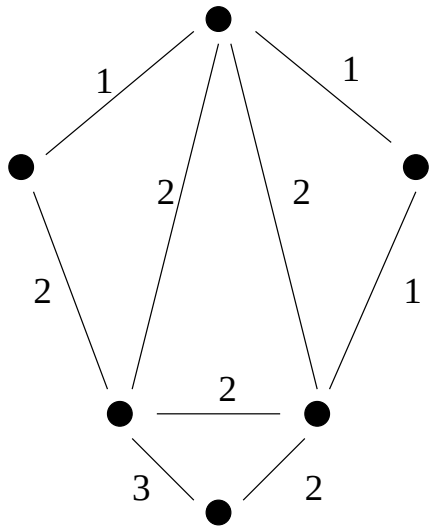
An undirected graph $G = (N, E)$ is a set of nodes and a set of edges
Each edge is an *unordered* pair of vertices.
Undirected graphs represent *symmetric* relations
Can obviously represent as directed graph with edges in both directions

Spanning Trees

Given a connected, undirected graph

$$G = \langle N, E \rangle$$

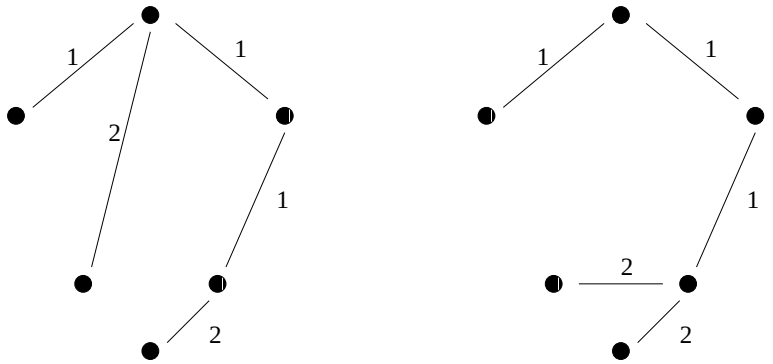
where each edge has an associated 'length' (or 'weight')
Want a subset T of edges E, such that the graph remains connected if only the edges in T are used, and sum of the lengths of edges in T is as small as possible.



Such a subgraph must be a *tree* (Why?)

Called *Minimum Spanning Tree*

For above graph, the following are both minimum spanning trees (cost 7)



What happens if we add an extra edge to a minimum spanning tree?

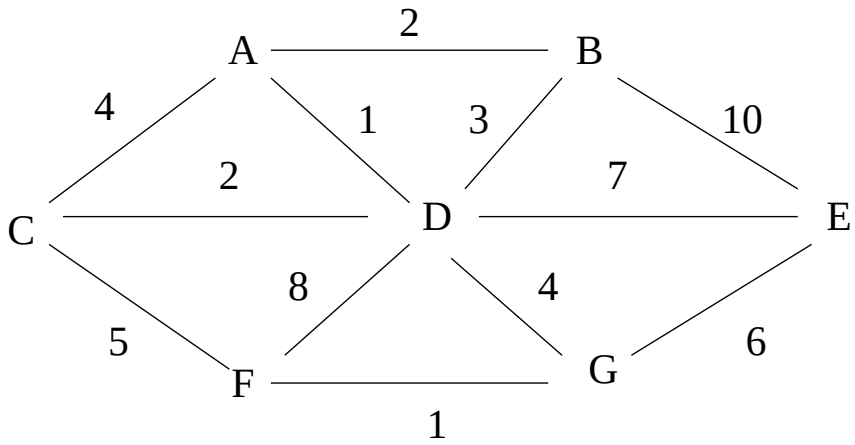
Problem Devise an algorithm to find a minimum spanning tree

Kruskal's Algorithm

Greedy algorithm to find minimum spanning tree.
Want to find set of edges T

- Start with $T = \emptyset$
- Keep track of connected components of graph with edges T
- Initially (when $T = \emptyset$), components are single nodes
- At each stage, add the cheapest edge that connects two nodes not already connected.

Example



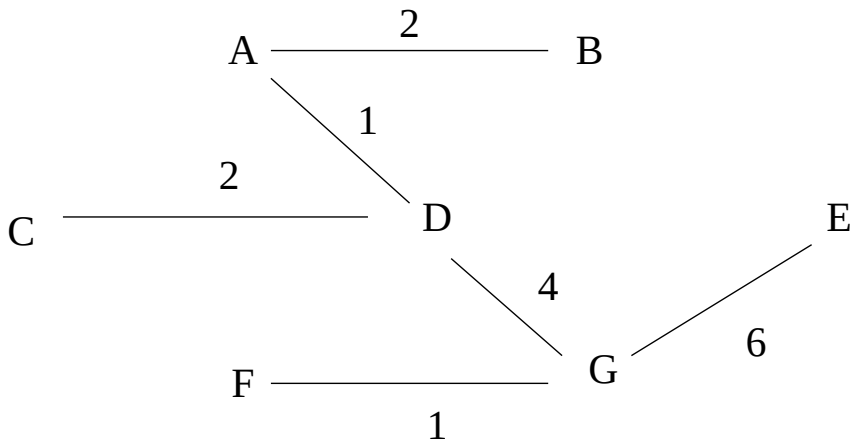
Ordered edge list

AD	FG	CD	AB	BD	DG	AC	CF	EG	DE	DF	BE
1	1	2	2	3	4	4	5	6	7	8	10

AD	FG	CD	AB	BD	DG	AC	CF	EG	DE	DF	BE
1	1	2	2	3	4	4	5	6	7	8	10

	Connected components
Initialise $T = \emptyset$	$\{A\},\{B\},\{C\},\{D\},\{E\},\{F\},\{G\}$
Add AD	$\{A,D\},\{B\},\{C\},\{E\},\{F\},\{G\}$
Add FG	$\{A,D\},\{B\},\{C\},\{E\},\{F,G\}$
Add CD	$\{A,D,C\},\{B\},\{E\},\{F,G\}$
Add AB	$\{A,B,D,C\},\{E\},\{F,G\}$
Reject BD	
Add DG	$\{A,B,D,C,F,G\},\{E\}$
Reject AC	
Reject CF	
Add EG	$\{A,B,D,C,F,G,E\}$

This gives minimum spanning tree



Implementation

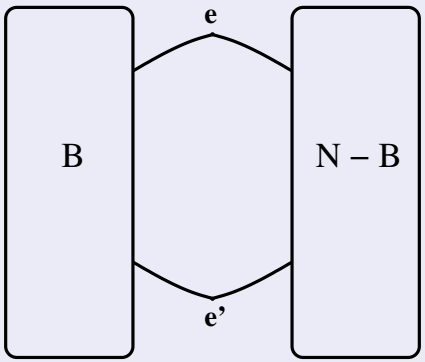
- The set of connected components used in Kruskal's algorithm is example of a *partition* (in this case of the set of nodes)
- Can use the simple *array representation of partition* described in the *Trees* section of the notes
- Since we need to access the shortest edge that is left, can use a heap-based implementation of *Priority Queue*

Proof of Kruskal's algorithm

We need to establish that the graph constructed by Kruskal's algorithm is indeed a minimum spanning tree.
Minimum spanning trees have the following property:

Given a graph $G = \langle N, E \rangle$, and a partition of the nodes into two sets, any minimum spanning tree contains the shortest (or one the shortest in the case of a tie) of the edges connecting a vertex in one of the sets to a vertex in the other

Proof.



- Suppose the two sets are B and $N - B$ and the minimum spanning tree has edges U.
- Suppose e is the shortest edge joining a node in B to one in $N - B$. If e is not in U, then add it.
- This will create a cycle (why?), so some edge other than e, say e' must connect B and $N - B$.
- Deleting e' from U still leaves a spanning tree, and if the length of e is not equal to that of e' it must be a smaller tree.
- This contradicts the fact that U is a minimum spanning tree, and the property is proved.

- The correctness of Kruskal's algorithm follows from this property.
- If the graph has n nodes and e edges, the complexity of Kruskal's algorithm is $O(e \lg n)$.

Prim's algorithm

An alternative to Kruskal's algorithm

- Choose an arbitrary starting node
- Maintain a set B of connected nodes
- At each stage, choose cheapest edge that connects an edge in B with an edge in $N - B$

Also a greedy algorithm

Example Same graph as before

AD	FG	CD	AB	BD	DG	AC	CF	EG	DE	DF	BE
1	1	2	2	3	4	4	5	6	7	8	10

	Connected nodes
	$\{D\}$ (arbitrary choice)
Add AD	$\{A, D\}$
Add CD	$\{A, C, D\}$
Add AB	$\{A, B, C, D\}$
Add DG	$\{A, B, C, D, G\}$
Add FG	$\{A, B, C, D, F, G\}$
Add EG	$\{A, B, C, D, E, F, G\}$

The correctness of Prim's algorithm also follows from the property described earlier.