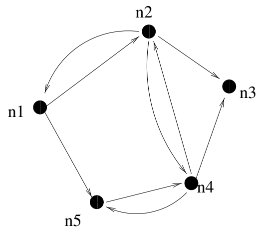
3: Graphs

Graphs

A *directed graph* 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.

Directed graphs also known as *digraphs*



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.

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.

Main disadvantage is that this requires $O(n^2)$ storage even if graphs has much fewer than n^2 edges.

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 `SML` 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.

Diversion: Greedy algorithms

Used to solve optimisation problems

Given a problem, find a solution, amongst many possible, 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 *objective function*

Example: Coin choosing

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

$$£87.68 =$$

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 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

But . . .

Back to the coins example:-

What happens if we introduce a 12p coin?

$$35p = ?$$

Back to Dijkstra

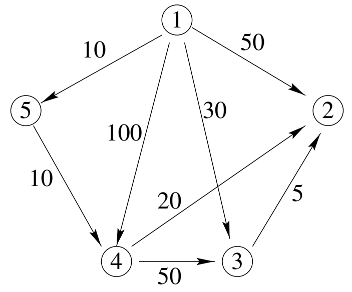
Want to find the shortest distance from a single node to each other node in the graph.

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.

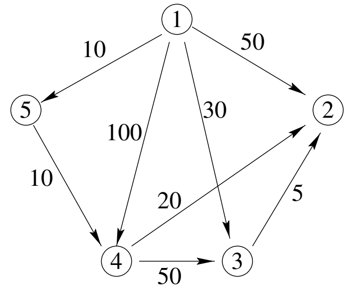
At each stage, add to S the node not currently in S which has the shortest special path.



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

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*, e is close to n^2 and so the straightforward implementation is better

If graph is *sparse* (e much smaller than n^2), the heap implementation is better

All pairs shortest paths

Problem
Given a directed graph

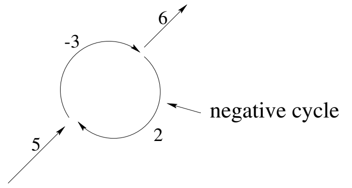
$$G = \langle N, E \rangle$$

where each edge has an associated 'length' (or 'cost'), find the shortest (cheapest) distance between *each pair* of nodes in the graph.

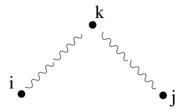
Recall that Dijkstra's algorithm gave shortest distance from a given node to all other nodes. Could use that n times (n number of nodes)

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

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

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 $1\dots k$ as intermediates)

Have two possibilities

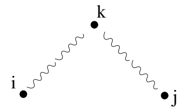
- The path doesn't actually use the node k
- It does use the node k

In first case, use only intermediate nodes $1\dots(k-1)$ 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

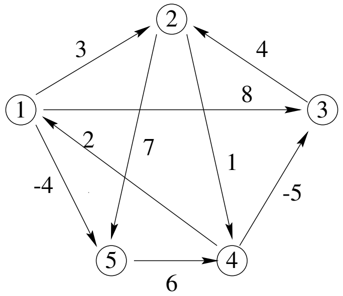
$$d(k,i,j) = \min(d(k-1,i,j), d(k-1,i,k) + d(k-1,k,j))$$

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

Floyd-Warshall – an example



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

$$D_0 = \begin{array}{ccccc} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{array} \qquad P_0 = \begin{array}{ccccc} * & 1 & 1 & * & 1 \\ * & * & * & 2 & 2 \\ * & 3 & * & * & * \\ 4 & * & 4 & * & * \\ * & * & * & 5 & * \end{array}$$

$$D_1 = \begin{array}{ccccc} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \underline{5} & -5 & 0 & \underline{-2} \\ \infty & \infty & \infty & 6 & 0 \end{array} \qquad P_1 = \begin{array}{ccccc} * & 1 & 1 & * & 1 \\ * & * & * & 2 & 2 \\ * & 3 & * & * & * \\ 4 & \underline{1} & 4 & * & \underline{1} \\ * & * & * & 5 & * \end{array}$$

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

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

$$D_2 = \begin{array}{ccccc} 0 & 3 & 8 & \underline{4} & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \underline{5} & \underline{11} \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{array} \qquad P_2 = \begin{array}{ccccc} * & 1 & 1 & \underline{2} & 1 \\ * & * & * & 2 & 2 \\ * & 3 & * & \underline{2} & \underline{2} \\ 4 & 1 & 4 & * & 1 \\ * & * & * & 5 & * \end{array}$$

$$D_3 = \begin{array}{ccccc} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & \underline{-1} & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{array} \qquad P_3 = \begin{array}{ccccc} * & 1 & 1 & 2 & 1 \\ * & * & * & 2 & 2 \\ * & 3 & * & 2 & 2 \\ 4 & \underline{3} & 4 & * & 1 \\ * & * & * & 5 & * \end{array}$$

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

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

$$D_4 = \begin{array}{ccccc} 0 & 3 & \underline{-1} & 4 & -4 \\ \underline{3} & 0 & \underline{-4} & 1 & \underline{-1} \\ \underline{7} & 4 & 0 & 5 & \underline{3} \\ 2 & -1 & -5 & 0 & -2 \\ \underline{8} & \underline{5} & \underline{1} & 6 & 0 \end{array} \qquad P_4 = \begin{array}{ccccc} * & 1 & \underline{4} & 2 & 1 \\ \underline{4} & * & \underline{4} & 2 & \underline{1} \\ \underline{4} & 3 & * & 2 & \underline{1} \\ 4 & 3 & 4 & * & 1 \\ \underline{4} & \underline{3} & \underline{4} & 5 & * \end{array}$$

$$D_5 = \begin{array}{ccccc} 0 & \underline{1} & \underline{-3} & \underline{2} & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{array} \qquad P_5 = \begin{array}{ccccc} * & \underline{3} & \underline{4} & \underline{5} & 1 \\ 4 & * & 4 & 2 & 2 \\ 4 & 3 & * & 2 & 2 \\ 4 & 3 & 4 & * & 1 \\ 4 & 3 & 4 & 5 & * \end{array}$$

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

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)

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

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/cs2011.html`

The Floyd-Warshall algorithm

```
void floyd(two_d_array edge_length, two_d_array D,
           two_d_array pred, int nodes)
{
  int i, j, k;

/* Initialize D and pred */
/* pred[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++ )
      {
        D[i][j] = edge_length[i][j];
        if (i != j && edge_length[i][j] != INF)
          pred[i][j] = i;
        else
```



```
          pred[i][j] = NOT_A_VERTEX;
      }
  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( D[i][k] + D[k][j] < D[i][j] )
          { /*update min and predecessor*/
            D[i][j] = D[i][k] + D[k][j];
            pred[i][j] = pred[k][j];
          }
}
```


Finding the paths

```
int print_shortest_path(two_d_array pred,
                        two_d_array edge_length,
                        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;

  if (i ==j)
    {
      printf("%2d",i);
      len =0;
    }
  else
```



```
    if (pred[i][j] == NOT_A_VERTEX)
      printf("No path from %d to %d\n",i,j);
    else
      {
        rest=print_shortest_path(pred,edge_length,
                                 i,pred[i][j]);
        printf("%2d",j);
        len=rest+edge_length[pred[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*

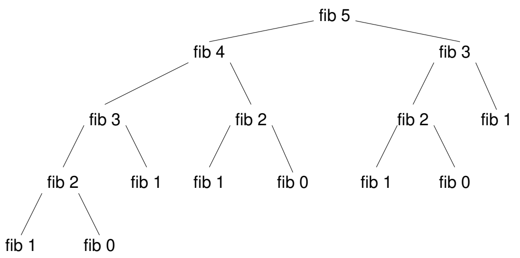
Recall the Fibonacci numbers (again)

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

Obvious function (SML) to calculate F_k is

```
fun fib 0 = 0
  | fib 1 = 1
  | fib n = 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*?

```
int fib (int n)
{
  int i, Fib[MAXSIZE];

  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

```
int fib_tab[MAXN];

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

```
void init_fib ()
{
  int i;
  fib_tab[0] = 0;
  fib_tab[1] = 1;
  for (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

```
void dfs (int v) {
  int w;
  mark[v] = visited;
  printf("Visiting %d\n",v);
  for (w = 0; w < graph.Size; w++) {
    if (adj(v,w) && (mark[w] == unvisited)) {
      dfs(w);
    }
  }
}
```

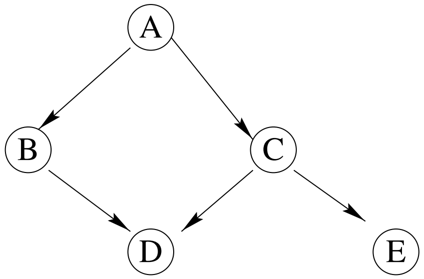


```
void dfsearch (void) {
  int i,w;
  for (i=0; i < graph.Size; i++)
    mark[i] = unvisited;
  for (w = 0; w < graph.Size; w++) {
    if (mark[w] == unvisited) {
      dfs(w);
    }
  }
}
```


Directed Acyclic Graphs

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

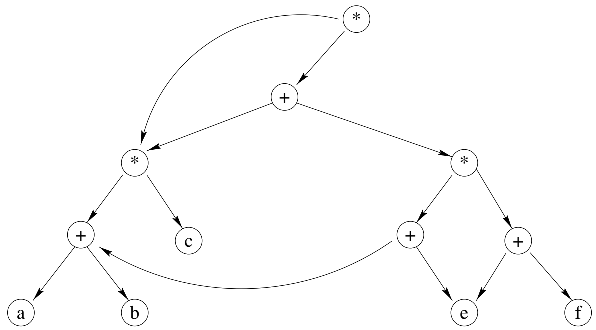
Every tree is a dag, but not every dag is a tree



Useful for representing tree-like structures *with sharing*

For example, algebraic expressions

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

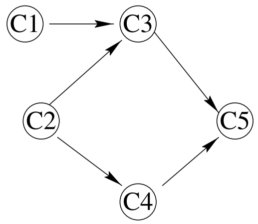


Also used for expressing *dependencies*

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 *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

Can do topological sort using depth first search

```
void dfs (int v) {
  int w;
  mark[v] = visited;
  for (w = 0; w < graph.Size; w++) {
    if (adj(v,w) && (mark[w] == unvisited)) {
      dfs(w);
    }
  }
  printf("%d\n",v);
}
```



```
void dfsearch (void) {
  int i,w;
  for (i=0; i < graph.Size; i++)
    mark[i] = unvisited;
  for (w = 0; w < graph.Size; w++) {
    if (mark[w] == unvisited) {
      dfs(w);
    }
  }
}
```

Then nodes in reverse order will be topologically sorted

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

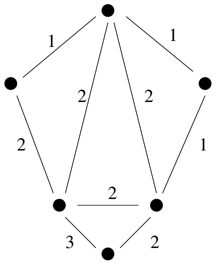
Spanning Trees

Given a connected, undirected graph

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

where each edge has an associated 'length' (or 'weight')

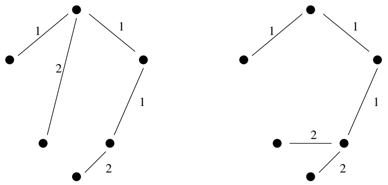
Want a subset T of edges E, such that the graph remains connected if only the edges in T are used, and sum of the lengths of edges in T is as small as possible.



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

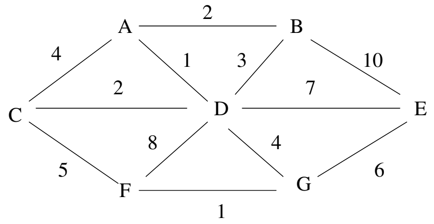
Kruskal's Algorithm

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

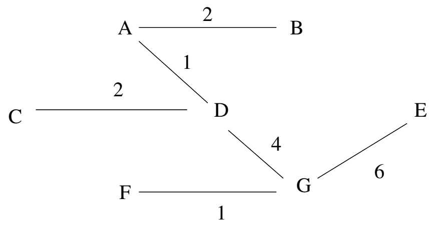


Ordered edge list

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

	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*

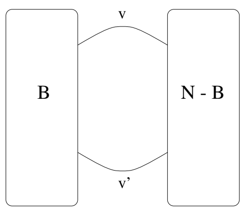
Proof of Kruskal's algorithm

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 of property



Suppose the two sets are B and $N - B$ and the minimum spanning tree has edges U.

Suppose v is the shortest edge joining a node in B to one in $N - B$.

If v is not in U, then add it.

This will create a cycle (why?), so some edge other than v, say v' must connect B and $N - B$.

Deleting v' from U still leaves a spanning tree, and if the length of v is not

equal to that of v' 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 on slide 64.