Massey University
COLLEGE OF SCIENCES

159201

Week 6

Summer 2014

Te Kunenga
ki Pūrehuroa

Massey University
COLLEGE OF SCIENCES

L 21

Te Kunenga
ki Pūrehuroa

Sets and Bags

A set is a collection of unique items of the same type.

For example: Set S = { 6, 22, 8 }

Sets have no order, i.e. there is no “next” or “previous”.

However, we can define maximum (= 22) and minimum (= 6) if desired.

Operations on Sets

`insert` – check for uniqueness

`delete`

`membership` – is 6 a member of the set?

`IsEmpty`

Very important ones:

`union`

`intersection`

Operations on Sets

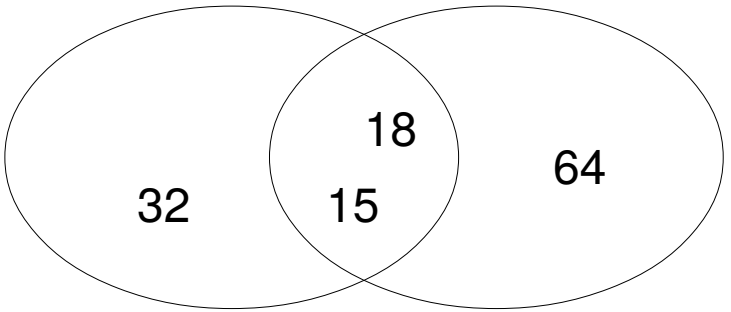
Example of union and intersection:

set A = { 15, 32, 18 }

set B = { 18, 15, 64 }

A union B = A ∪ B = { 15, 64, 18, 32 } note that uniqueness is retained

A intersection B = A ∩ B = { 15, 18 }



Operations on Sets

Many other operations are possible.

An interesting one is ProcessAll (function f) which applies f to all elements of the set.

For example:

```
set S = { 22, 3, 97 }
set X = S.ProcessAll (increment);
```

After this X would look like this: {23,4,98}

Bags

A bag is similar to a set but contains items that are not unique.

For example, bag B = { 14(3), 9(5), 38(2) }

This means that 14 occurs 3 times in the bag, 9 occurs 5 times, etc.

A set of New Zealand coins = { 50c, 10c, $2, 20c, $1 }

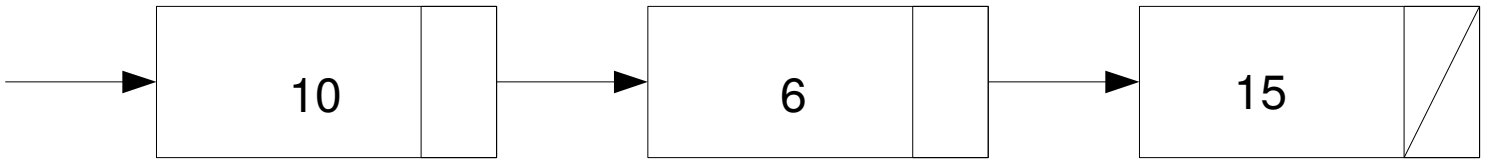
A bag of New Zealand coins = { 10c(3), $1(2), 50c(4), 20c(5), $2(400) }

Operations are similar to sets although uniqueness is not required.

Implementing a Set or a Bag

1) use a linked-list

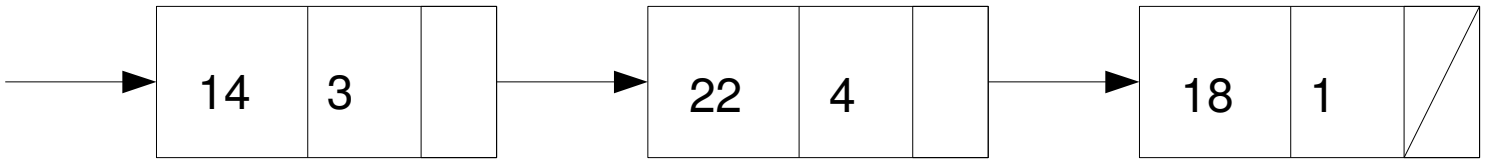
For example, the set S = { 10, 6, 15 } can be represented by:



Implementing a Set or a Bag

It may be useful to keep the items sorted to help when checking for uniqueness.

For example, the bag B = { 14(3), 22(4), 18(1) } can be represented by:



Implementing a Set or a Bag

2) use an array

You have the usual problems with fixed size and insert/delete in the middle

3) use a tree

It may be quite efficient, but overly complex

For instance, searching may be fast.

Implementing a Set: bit vector

4) use a bit vector

In this case, only sets (not with bags)

For example: set A = { 5, 1, 6, 3 } can be represented by the bit vector:

position from the right:	7	6	5	4	3	2	1	0
value (zero or one):	**0**	**1**	**1**	**0**	**1**	**0**	**1**	**0**

In a C++ program a bit vector can be:

Unsigned int or an unsigned char etc

Implementing a Set: bit vector

For example:

```
int bitvector; //declares a bit vector of 32(or 64?) bits
```

Use logical operations (&, |, ^) to manipulate the bit vector, e.g. 2 & 3 = 2

and(&) is used for intersection and or(|) is used for union.

Limitations on bit vectors – they can only be used to represent small sets.

Recall bitwise operators

AND &

X	Y	output
0	0	0
0	1	0
1	0	0
1	1	1

OR |

X	Y	output
0	0	0
0	1	1
1	0	1
1	1	1

Recall bitwise operators

XOR ^

X	Y	output
0	0	0
0	1	1
1	0	1
1	1	0

SHIFTS << multiply by 2

>> integer division by 2

bit vector Insert(num)

Given a binary number, how to insert a bit?

```
unsigned char bitvector;
```

0 1 1 0 1 0 1 0

Suppose we want to insert {2}:

```
unsigned char temp;
temp=temp << 2;
bitvector = bitvector | temp;
```


bit vector Remove(num)

Given a binary number, how to insert a bit?

```
unsigned char bitvector;
```

0 1 1 0 1 1 1 0

Suppose we want to remove {2}:

```
unsigned char temp;
temp=temp << 2;
bitvector = bitvector ^ temp;
```


bit vector Member(num)

Given a binary number, how to insert a bit?

```
unsigned char bitvector;

        0 1 1 0 1 1 1 0
```

Suppose we want to learn if {2} is an element:

```
unsigned char temp;
temp=temp << 2;
temp = temp & bitvector;
if(temp) return true;
else return false;
```


printbits(bitvector)

```
void printbits(unsigned char bitvector){
    for(int i=(sizeof(char)*8-1); i>=0; i--){
      if(i==3) printf(" ");
      unsigned char temp=1;
      temp=temp<<i;
      temp=bitvector&temp;
      if(temp) printf("1");
      else printf("0");
    }
    printf("\n");
}
```


Massey University
COLLEGE OF SCIENCES

L 22

Te Kunenga
ki Pūrehuroa

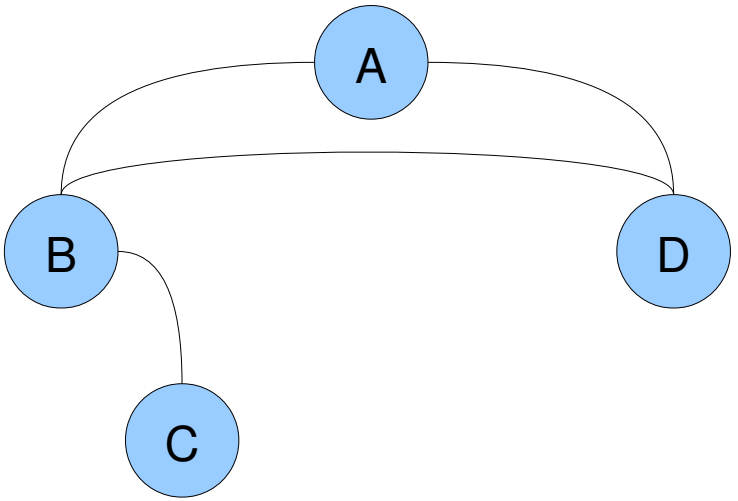
Graphs: introduction

Graph: an abstract representation of objects, where the objects can be connected. We have *vertices* and *edges*.

E.g. a graph G=(V,E)

V is a set of nodes(vertex)

E is a set of edges(arcs)



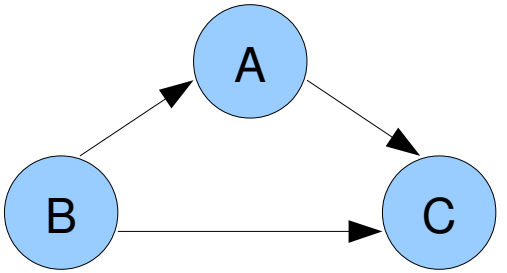
V = {A,B,C,D}

E = { (B,A),(C,B),(B,D),(A,D) }

Graphs: introduction

Definitions:

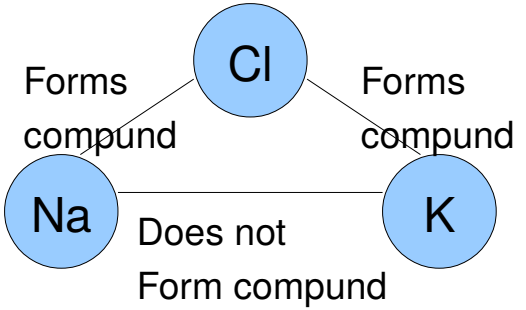
Directed graph : the edges are arrows



Note that the previous example was an undirected graph.

Labelled graph : words on the edges

A labelled graph can be directed or undirected.

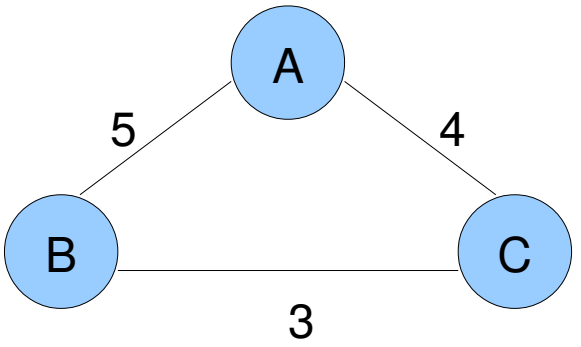


Graphs: introduction

Definitions:

Weighted graph : numbers on the edges

The numbers are known as weights



e.g., used in Computer Networks,
the edges might mean time to arrive,
cost etc.

Graphs: introduction

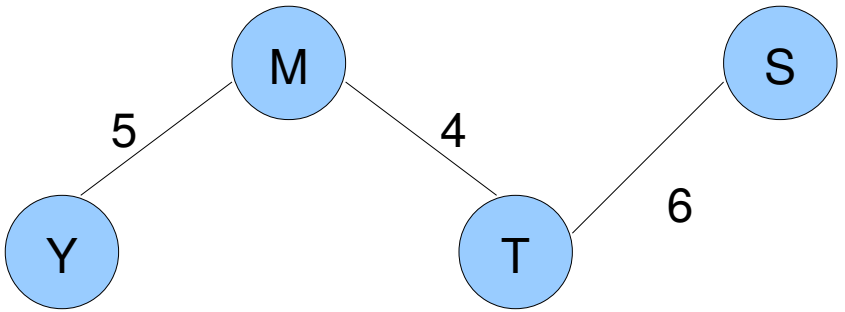
Definitions:

In this example, M and T are **adjacent** (there is an edge between them.)

M and S are **not adjacent**.

There is a **path** from Y to S as follows:

Y → M → T → S



Graphs: introduction

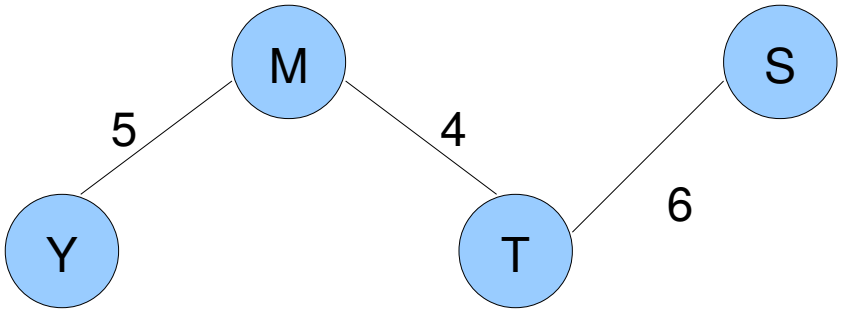
Definitions:

The **length of the path** is: 5 + 4 + 6 = 15

(For graphs that are not weighted, assume each edge has a length of 1.)

A **simple path** has distinct nodes (except perhaps first may be the same as last).

In the example, this path is not a simple path: M → T → M → Y



Graphs: introduction

Definitions:

A **connected graph** has a path from every node to every other node.

A **cycle** is a simple path (only distinct nodes) from node X back to node X (it must include other nodes).

An **acyclic** graph is a graph that contains no cycles.

Massey University
COLLEGE OF SCIENCES

Graphs: introduction

Why study graphs (or any data structure)?

If a new problem (such as searching the internet) can be represented as a graph then all existing theory, knowledge and algorithms about graphs is immediately available to be applied to the new problem.

Te Kunenga
ki Pūrehuroa

Graphs: operations

Insert a node

Delete a node – warning: this may affect existing edges

Insert an edge – must be based on two existing nodes

Delete an edge

Edit_Node – change the data in a node

Edit_Edge – change weight and/or label attached to the edge

Navigate – move around the graph (traversals)

Find_Path – find a path between two nodes

Find_Shortest_Path

Find_Cycle

IsEmpty, isAdjacent

Count_Nodes, Count_Edges

Process_All_Nodes, Process_Adjacent_Nodes

Massey University
COLLEGE OF SCIENCES

L 23

Te Kunenga
ki Pūrehuroa

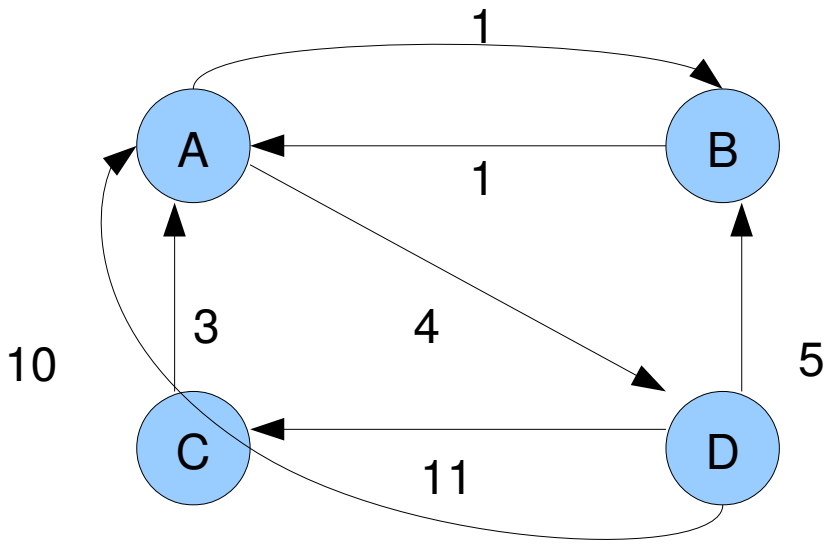
Graph Implem.: directed, weighted

1) Using Sets

G=(V,E) where

V={A,B,C,D} and

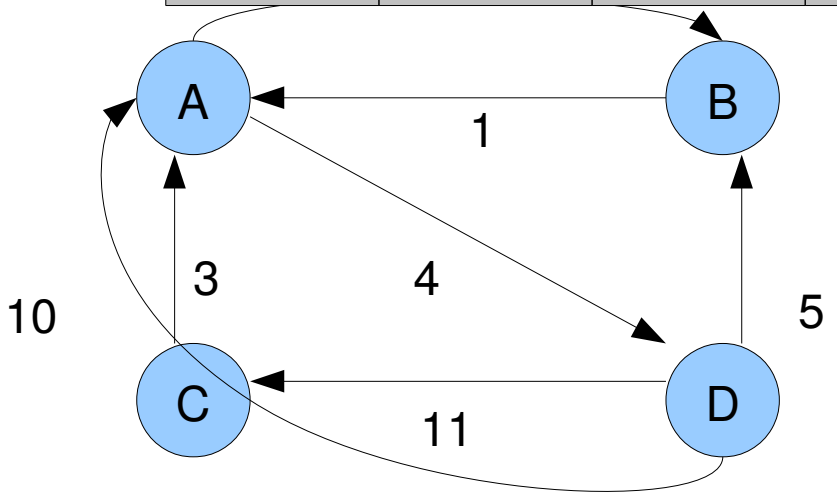
E={(A,B,1), (C,A,3), (D,C,11), (D,B,5),(B,A,1), (A,D,4),(D,A,10)}



Graph Implem.: directed, weighted

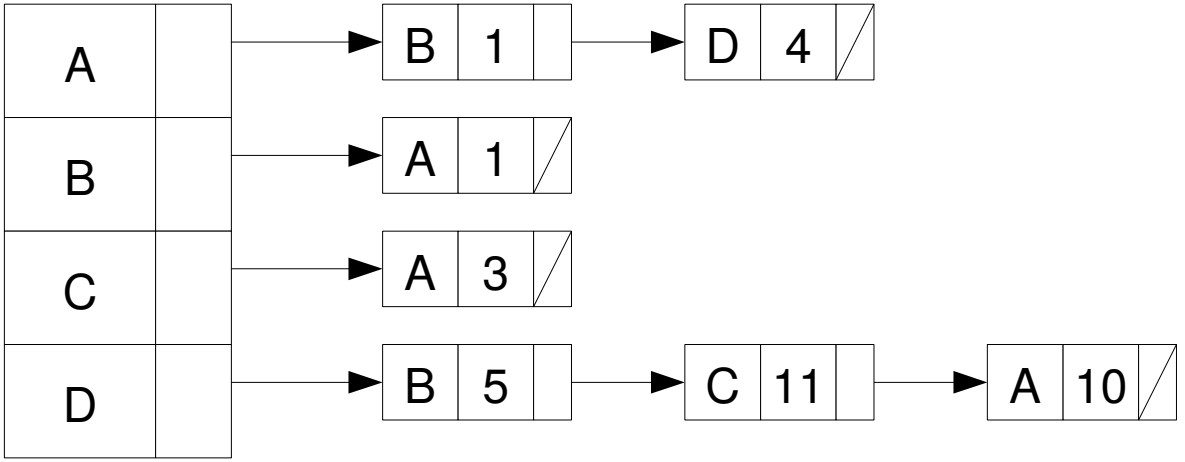
2) Adjacency Matrix

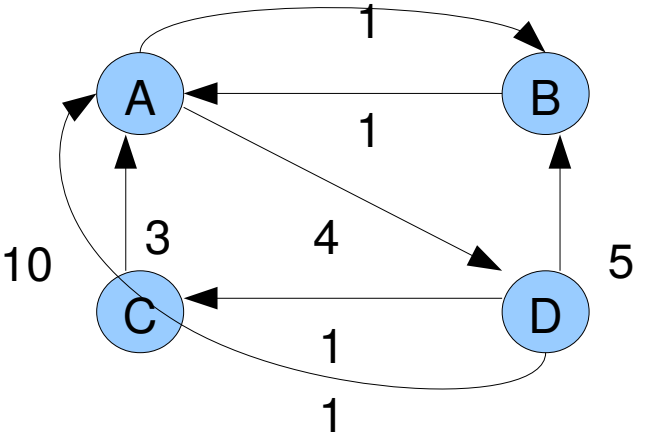
from \ to	A	B	C	D
A	0	1	0	4
B	1	0	0	0
C	3	0	0	0
D	10	5	11	0



Graph Implem.: directed, weighted

3) Adjacency list





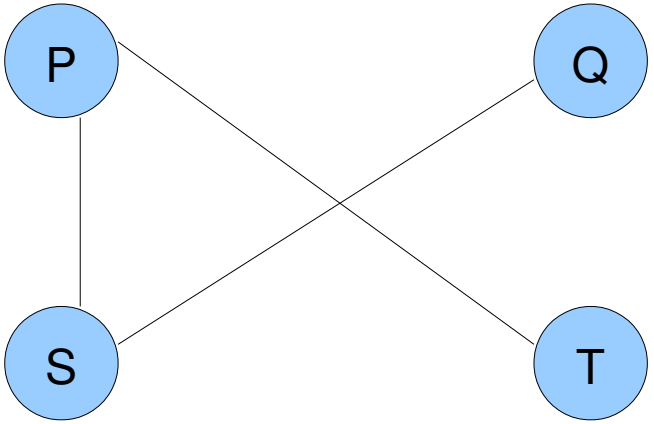
Graph Implementation: undirected

1) Using Sets

G=(V,E) where

V={T,P,S,Q} and

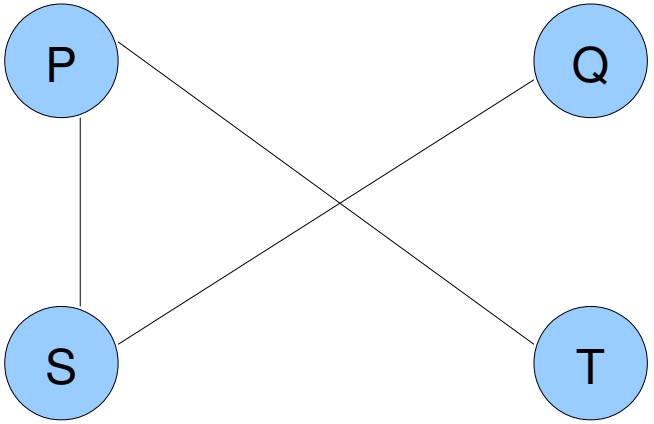
E={(P,T),(T,P),(S,Q),(Q,S),(P,S),(S,P)}



Graph Implementation: undirected

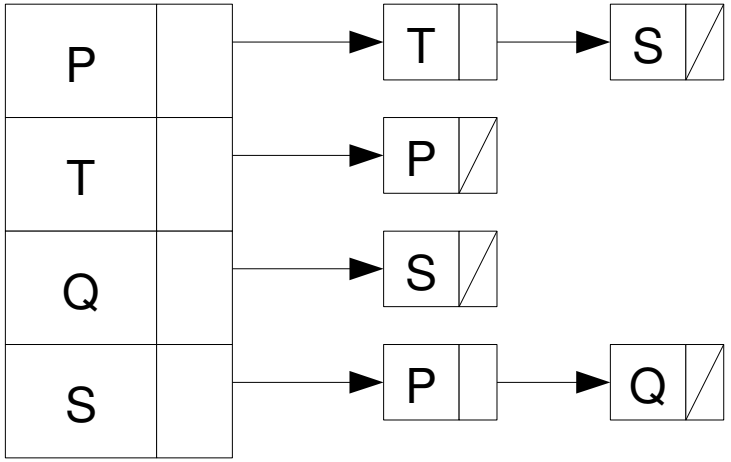
2) Adjacency Matrix

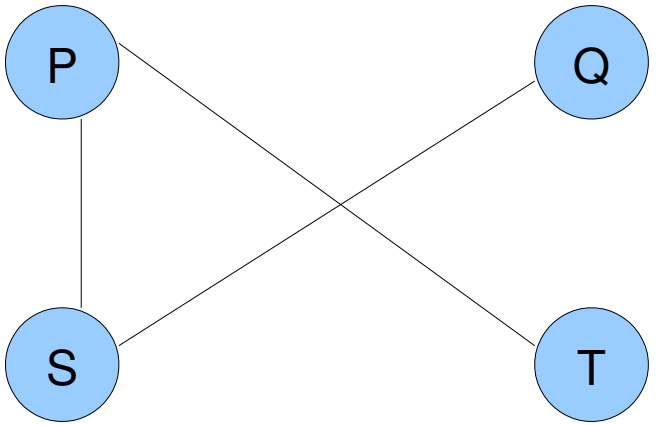
from \ to	P	T	Q	S
P	0	1	0	1
T	1	0	0	0
Q	0	0	0	1
S	1	0	1	0



Graph Implementation: undirected

3) Adjacency list





Implementation of operations

The choice of implementation will affect operations. For example, the operation “weight_of_edge” which returns the weight of a given edge, will be far more efficient using an adjacency matrix rather than an adjacency list.

Inserting is always a problem as more space is required.

Deleting is always a problem – can be solved by using adjacency list with a vector.

When deleting in an adjacency matrix – could “mark” a node as deleted and reuse space later.

Using a vector with an adjacency list

The “list of nodes” in an adjacency list could be an *array*, a *vector* or a *linked-list*.

A vector is useful because it combines the properties of an array (such as random access) with those of a list (such as dynamic memory allocation.

The sample program presented next uses a vector of floats. A vector used to represent a graph would actually be a vector of “header nodes” – which are different from “list nodes”.

Using a vector with an adjacency list

```
#include <vector>
using namespace std;
class Graph{
private:
  vector<GraphNode> adjlist;
public:
  Graph(){};
  ~Graph(){};
  void AddNewGraphNode(char newgraphnode);
  void AddNewEdgeBetweenGraphNodes(char A, char B,
     int distance);
  void PrintAllGraphNodesWithCosts();
};
```


GraphNode and linked-lists Node

```
struct GraphNode{//used by the vector
   char key;
   Node *listpointer;
};

struct Node {//used by the linked-lists
   char key;
   int distance;
   Node *next;
};
```


functions for the linked-lists

```
void AddNodetoFront(Node*& listpointer, char
newkey, int newdistance){
   Node *temp;
   temp = new Node;
   temp->key = newkey;
   temp->distance=newdistance;
   temp->next = listpointer;
   listpointer = temp;
}
```


functions for the linked-lists

```
void PrintLLnodes(Node*& listpointer){
  Node *temp;
  temp = listpointer;
  while(temp!=NULL){
    printf("to node %c dist: %d \n", temp->key,
temp->distance);
    temp=temp->next;
  }
}
```


Methods for Graph class

```
void Graph::AddNewGraphNode(char newgraphnode){
  GraphNode temp;
  temp.key=newgraphnode;
  temp.listpointer=NULL;//important
  adjlist.push_back(temp);
}
```


Methods for Graph class

```
void  Graph::AddNewEdgeBetweenGraphNodes(char A,
char B, int distance){
  //find which node A is
  int a;
  for (a=0;adjlist.size();a++){
    if (A==adjlist[a].key) break;
  }
  AddNodetoFront(adjlist[a].listpointer, B,
distance);
}
```


Methods for Graph class

```
void Graph::PrintAllGraphNodesWithCosts(){
   for (int a=0;a<adjlist.size();a++){
      printf("From Node %c: \n", adjlist[a].key);
      PrintLLnodes(adjlist[a].listpointer);
   }
}
```


Main

```
Main(){//creates the graph and traverses it
 Graph mygraph;
 mygraph.AddNewGraphNode('A');
 mygraph.AddNewGraphNode('B');
 mygraph.AddNewGraphNode('C');
 mygraph.AddNewGraphNode('D');
 mygraph.AddNewEdgeBetweenGraphNodes('A', 'B', 1);
 mygraph.AddNewEdgeBetweenGraphNodes('A', 'D', 4);
 mygraph.AddNewEdgeBetweenGraphNodes('B', 'A', 1);
 mygraph.AddNewEdgeBetweenGraphNodes('C', 'A', 3);
 mygraph.AddNewEdgeBetweenGraphNodes('D', 'B', 5);
 mygraph.AddNewEdgeBetweenGraphNodes('D', 'C', 11);
 mygraph.AddNewEdgeBetweenGraphNodes('D', 'A', 10);
 mygraph.PrintAllGraphNodesWithCosts();}
```

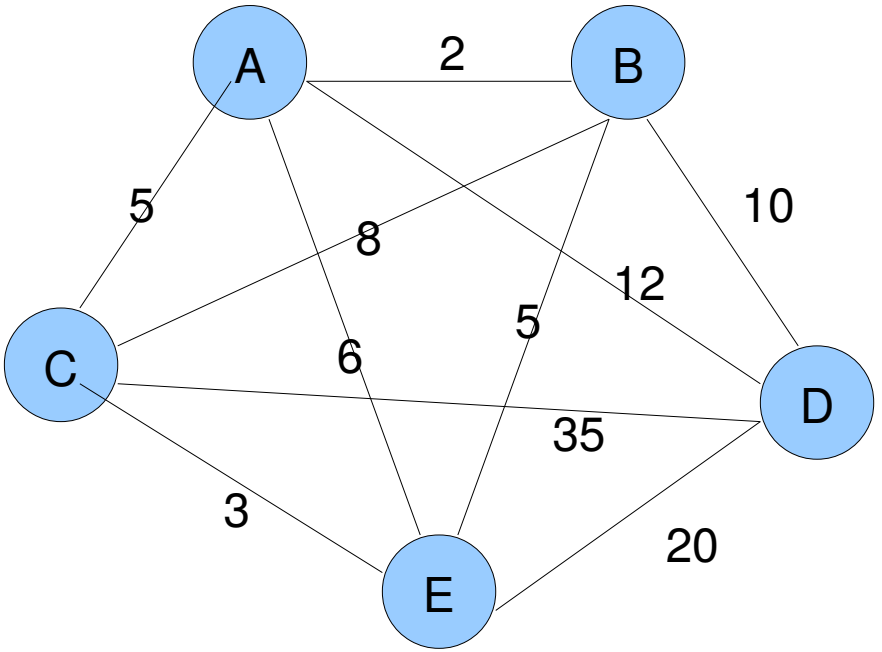

Massey University
COLLEGE OF SCIENCES

L 24

Te Kunenga
ki Pūrehuroa

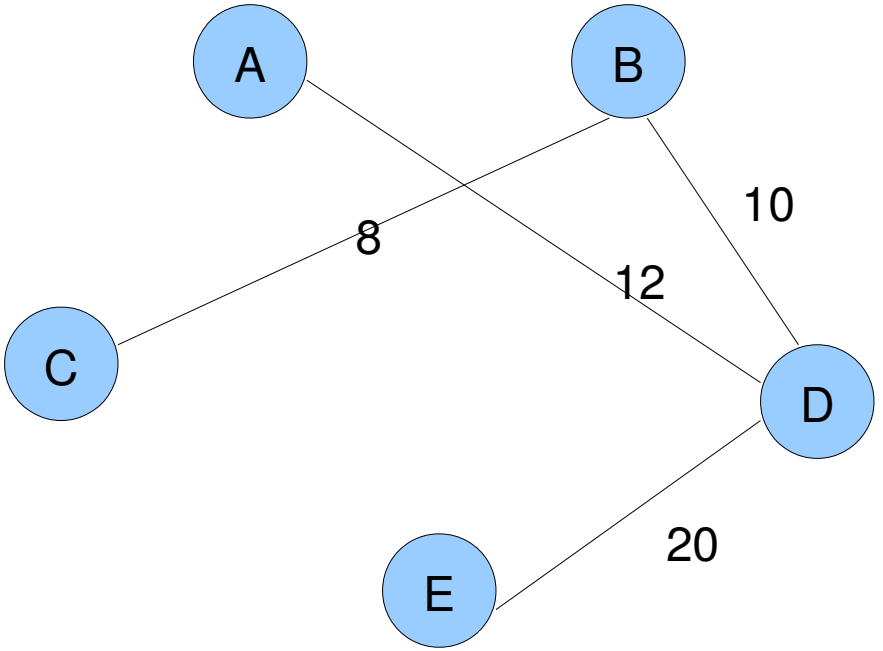
Graphs: Kruskal and Dijkstra

A spanning tree for a graph is the set of all the nodes plus a subset of the edges such that all nodes are connected and no cycles are present



Graphs: spanning tree 1

Many possibilities, e.g.:

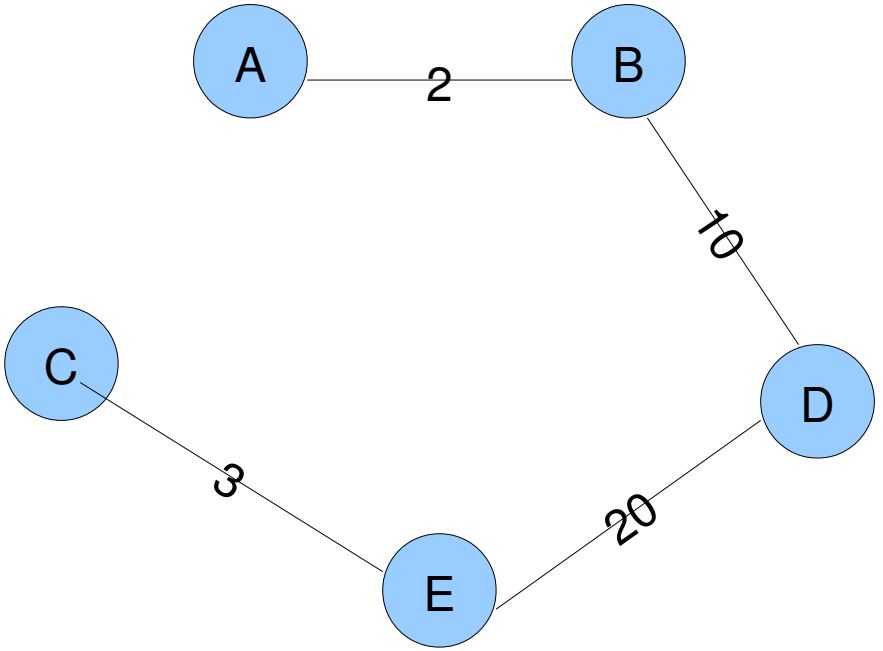


The cost for this spanning tree is:

10+20+12+8=50

Graphs: spanning tree 2

Another example:



And the cost is:

20+10+3+2=35

Kruskal's algorithm

There are a lot of spanning trees for any one graph. Spanning trees are used to find the minimum connections in a graph. They have many uses, e.g. telephone lines, computer network, deliveries.

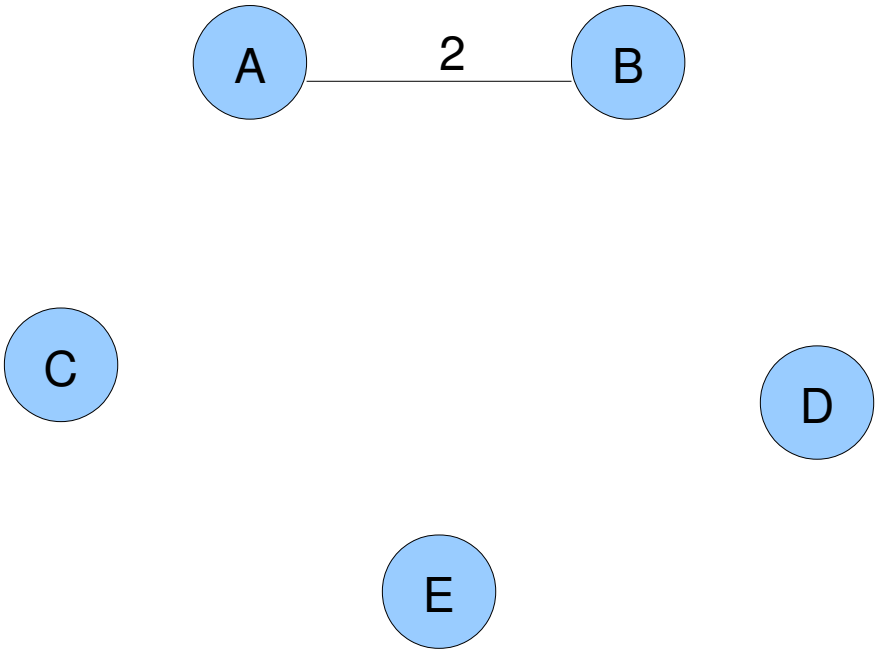
To find the Minimum Cost Spanning Tree (MCST) we use **Kruskal's Algorithm**

1. Start with all the nodes and no edges.
2. Add the next-smallest edge such that no cycles are formed.
3. Repeat step (2) until all nodes are connected.

Massey University
COLLEGE OF SCIENCES

Kruskal's algorithm

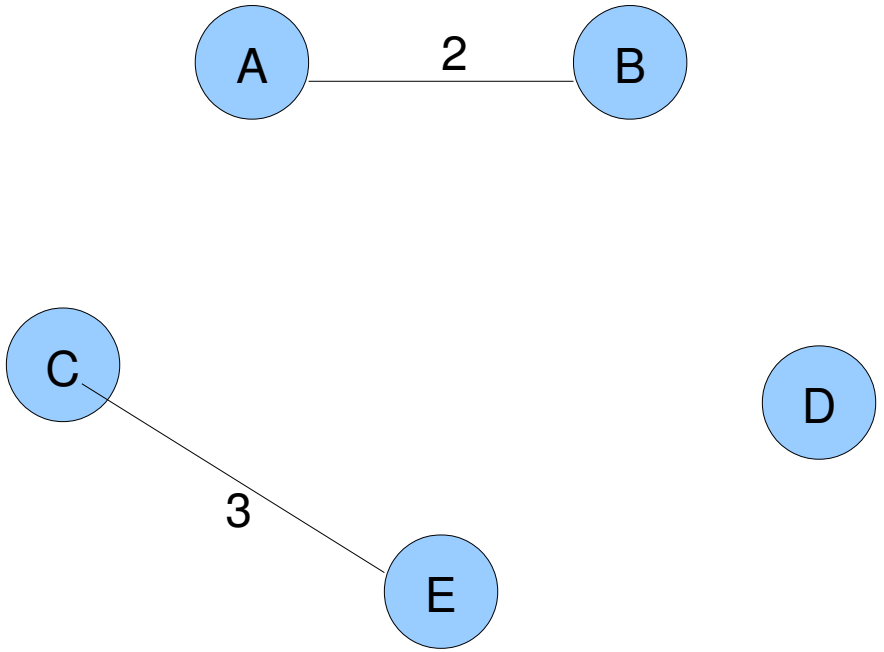
Step 1



Te Kunenga
ki Pūrehuroa

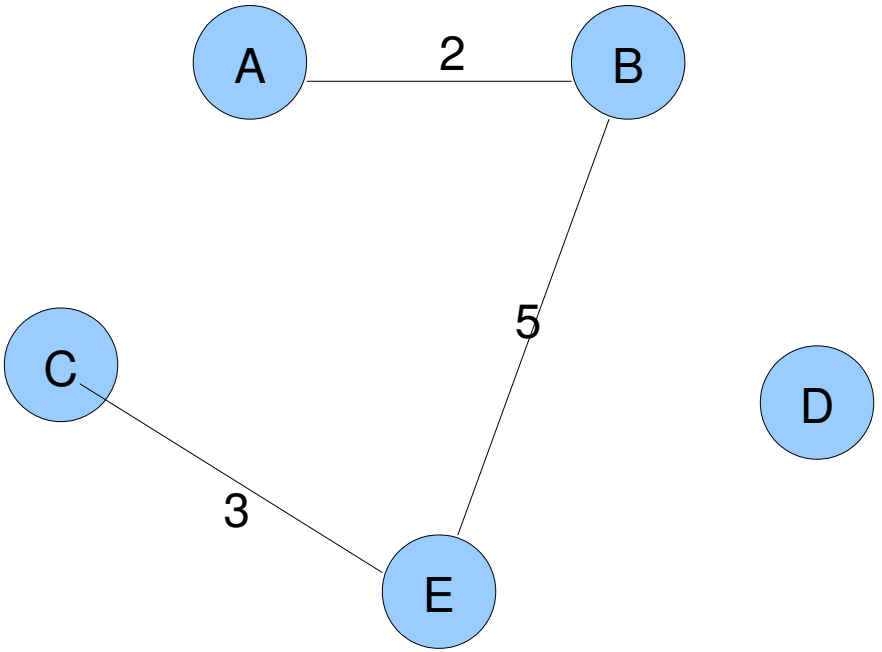
Kruskal's algorithm

Step 2



Kruskal's algorithm

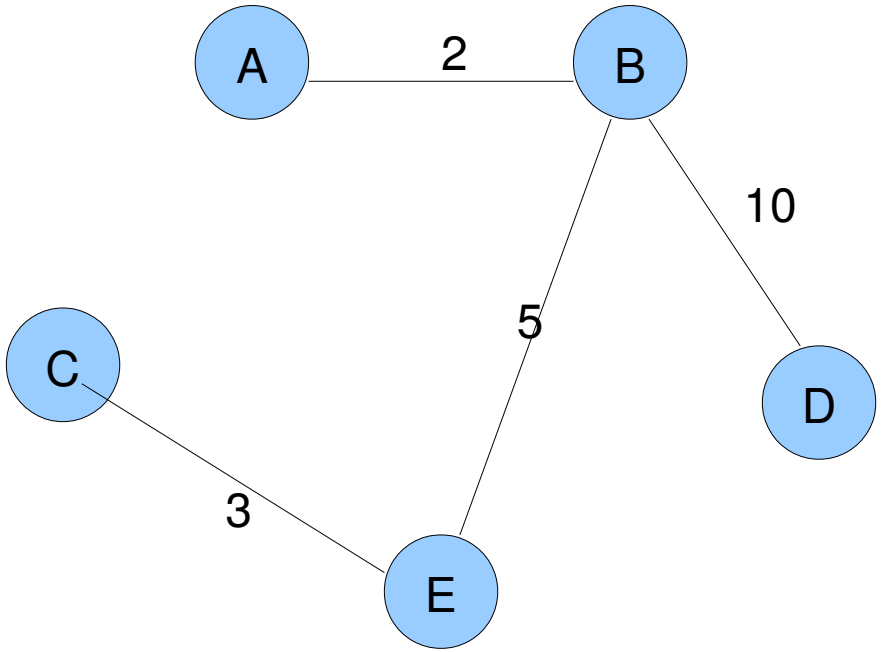
Step 3



Kruskal's algorithm

Step 4: all nodes linked, cost:

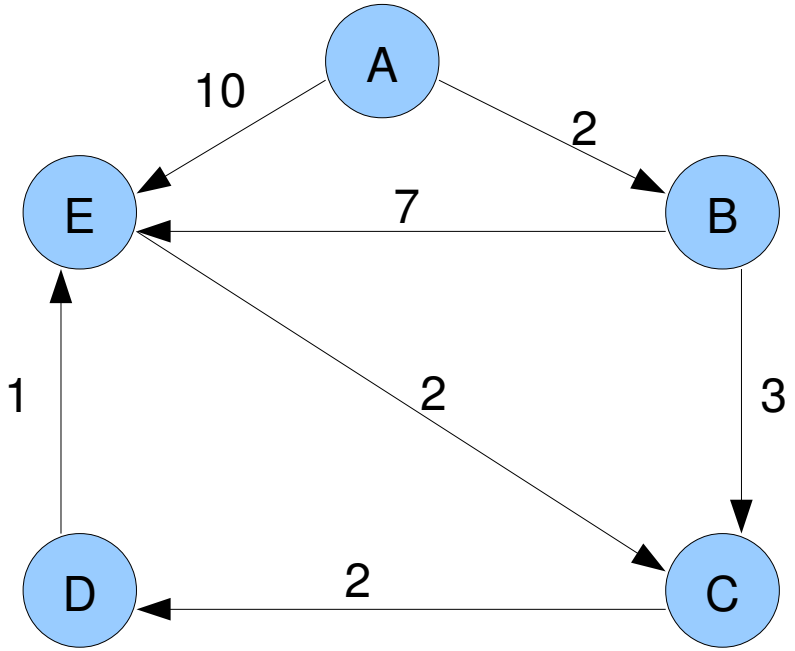
2+3+5+10=20 (this is a MCST, are there others with the same cost?)



Dijkstra's algorithm (1959)

Find the shortest paths (typical problem for networks!).

E.g., let's find the shortest path from node A to all other nodes.



Dijkstra's algorithm (1959)

Dijkstra only solves the problem is the costs are non-negative (there are other algorithms that can solve that too, e.g., Bellman-Ford)

Lets consider the following variables:

d is the distance value of a node

s is the state of a node (**d**, permanent or temp)

c is the cost of a link

current is the current node

Dijkstra's algorithm steps

Step 1:

- assign a node to zero and permanent

  e.g., $s_A=(0,p)$

-Assign all other node to (∞, t)

Step 2:

- find reachable nodes from current, updating d for these nodes: $d_j = \min(d_j , d_i+c_{ij})$

- find node j with the smallest d_j

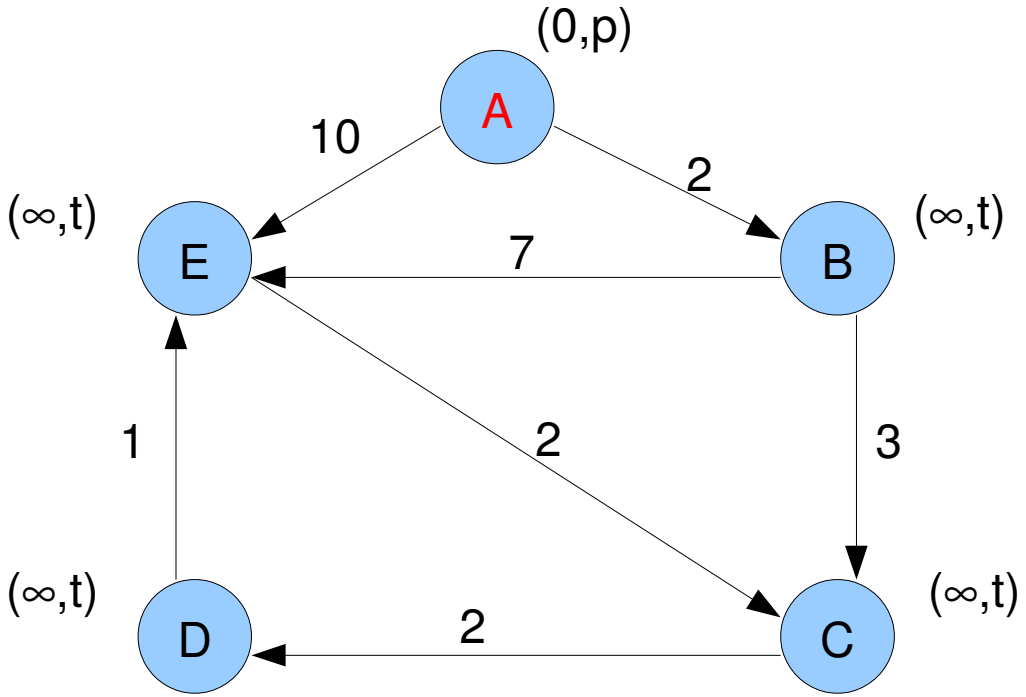
- change current and its state to (d_j, p)

Step 3:

- if all nodes are labelled p, end, otherwise step 2

Dijkstra's algorithm example

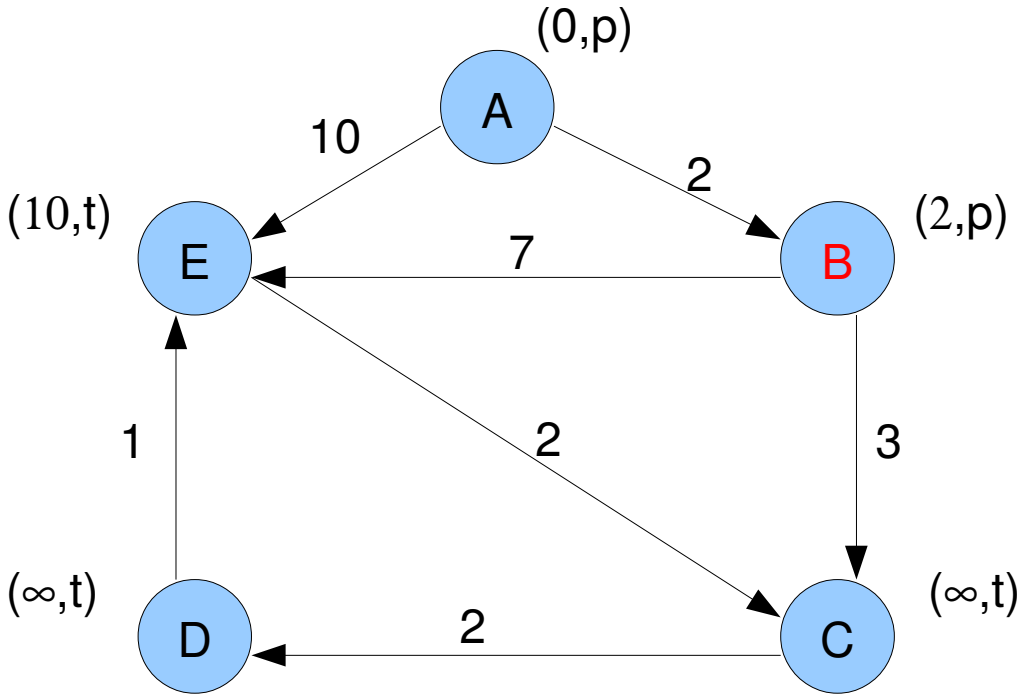
Starting with A:



Dijkstra's algorithm example

Nodes E and B are reachable: $d_E=\min(\infty , 0+10)=10$

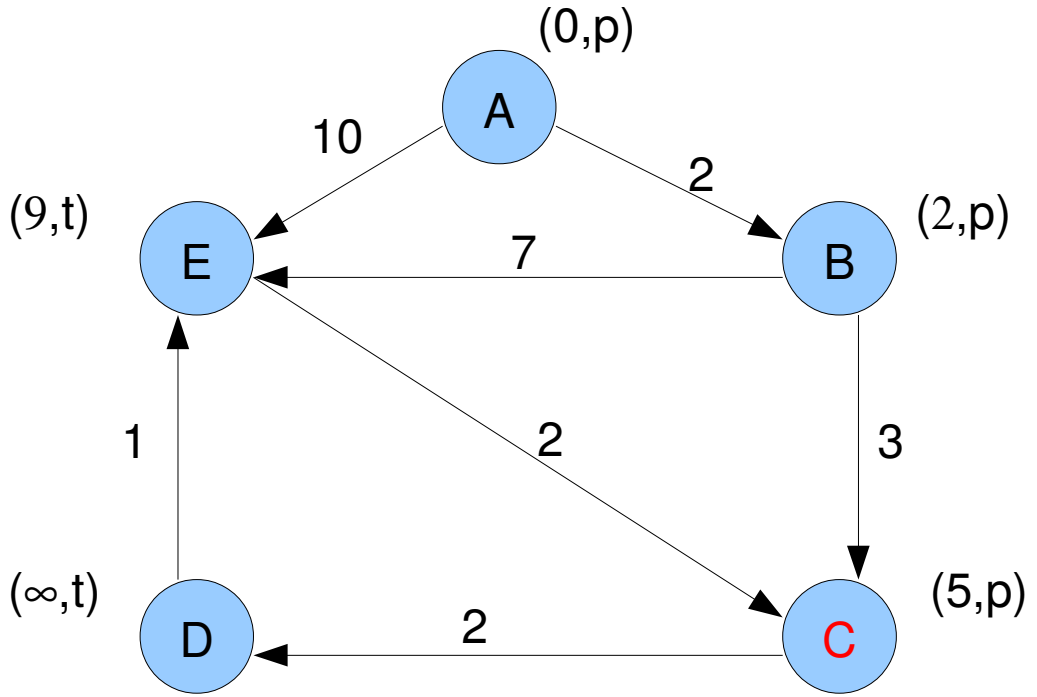
$d_B=\min(\infty , 0+2)=2$, B becomes p.



Dijkstra's algorithm example

Nodes E and C are reachable: $d_E=\min(10, 2+7)=9$

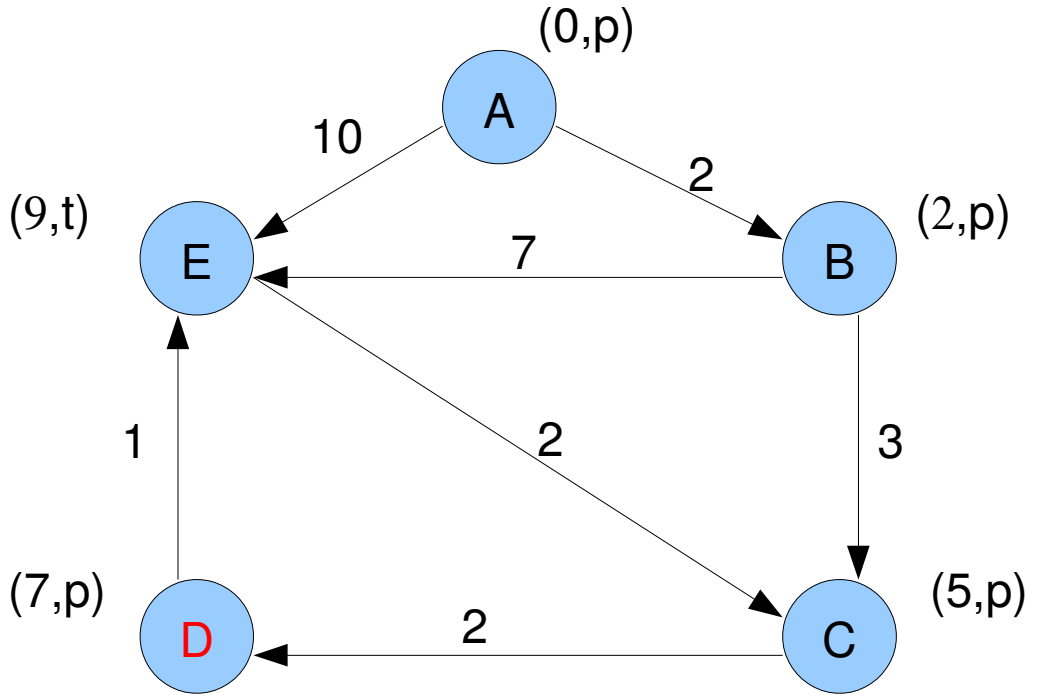
$d_C=\min(\infty , 2+3)=5$, C becomes p.



Dijkstra's algorithm example

Node D is reachable: $d_D = \min(\infty, 5+2) = 7$

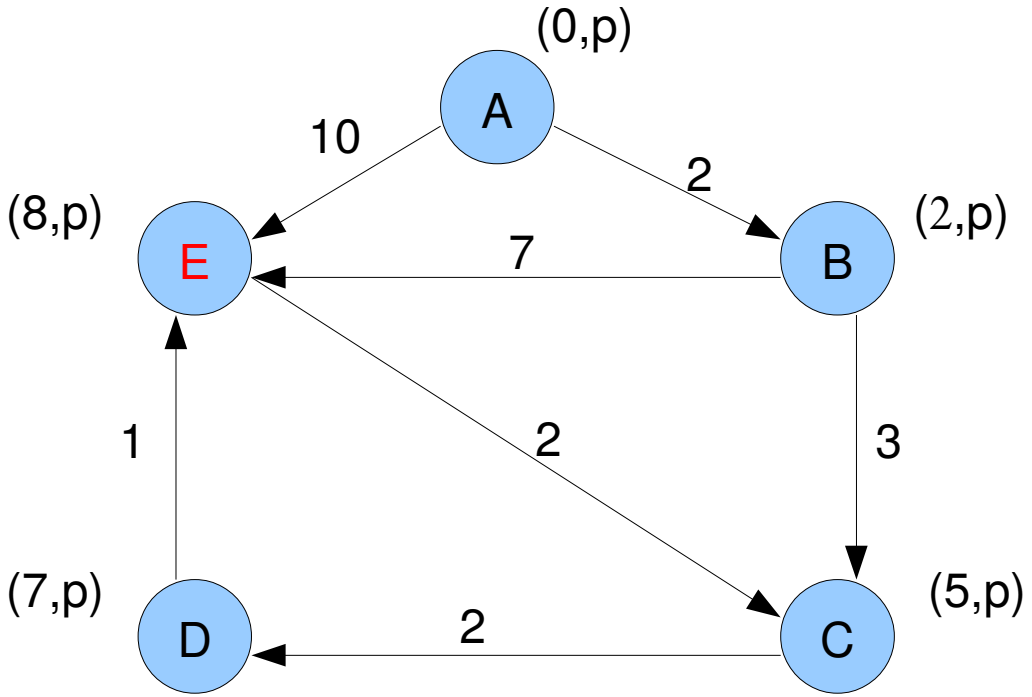
D becomes p.



Dijkstra's algorithm example

Nodes E is reachable: $d_E=\min(9 , 7+1)=8$

E becomes p. End the algorithm (all nodes=p)



Massey University
COLLEGE OF SCIENCES

Dijkstra's algorithm example

permanent nodes	frontier nodes	possible paths	final path
{ }	{ A }	A → A = 0	A → A = 0
{ A }	{ B, E }	A → B = 2 A → E = 10	A → B = 2
{ A, B }	{ E, C }	A → E = 10 A → B → E = 9 A → B → C = 5	A → B → C = 5
{ A, B, C }	{ E, D }	A → E = 10 A → B → E = 9 A → B → C → D = 7	A → B → C → D = 7
{ A, B, C, D }	{ E }	A → E = 10 A → B → E = 9 A → B → C → D → E = 8	A → B → C → D → E = 8
{ A, B, C, D, E }	{ }	stop	

Te Kunenga
ki Pūrehuroa

Dijkstra's algorithm (1959)

Notes:

The set of permanent nodes contains all nodes in the current "known area".

The set of frontier nodes contains all nodes immediately adjacent to permanent nodes.

Possible paths can include permanent nodes but not frontier nodes (except for the last node).

Choose the shortest possible path to transfer to the list of final paths.

At each step, the destination node in the final path is added to the set of permanent nodes.