Algorithmics I

Section 3 – Graphs & graph algorithms

Dr. Gethin Norman

School of Computing Science
University of Glasgow

gethin.norman@glasgow.ac.uk

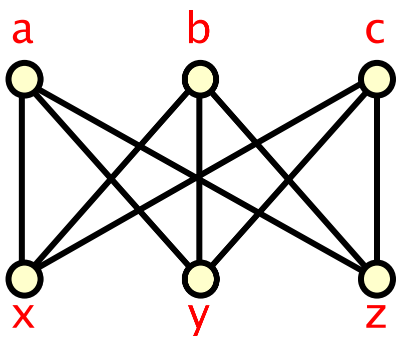
Graph basics

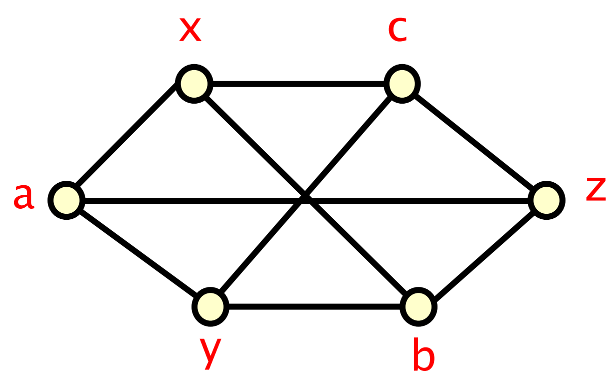
(undirected) graph **G = (V,E)**

- V is finite set of vertices (the vertex set)
- E is set of edges, each edge is a subset of V of size 2 (the edge set)

Pictorially:

- a vertex is represented by a point
- an edge by a line joining the relevant pair of points
- a graph can be drawn in different ways
- e.g. two representations of the same graph

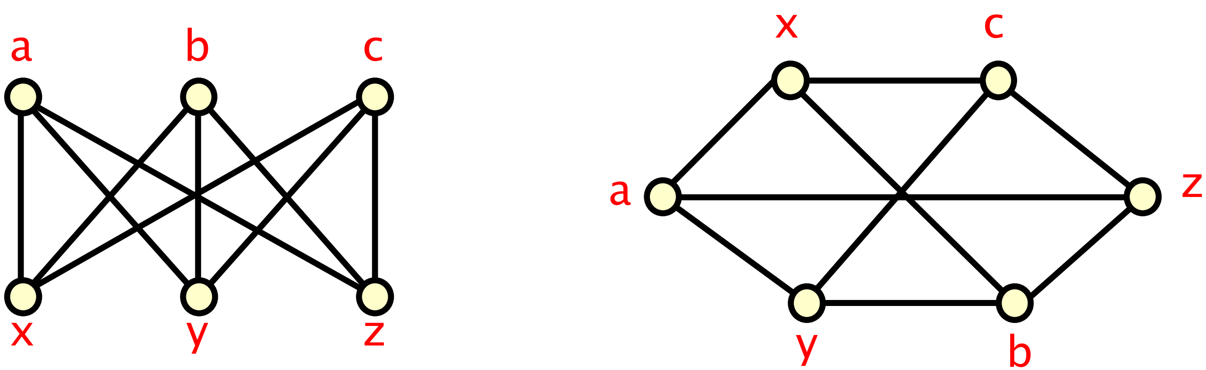




V= {a,b,c,x,y,z}

E= { {a,x},{a,y},{a,z},
{b,x},{b,y},{b,z},
{c,x},{c,y},{c,z} }

Graph basics

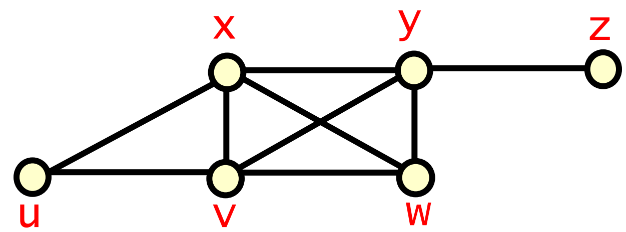


In this graph:

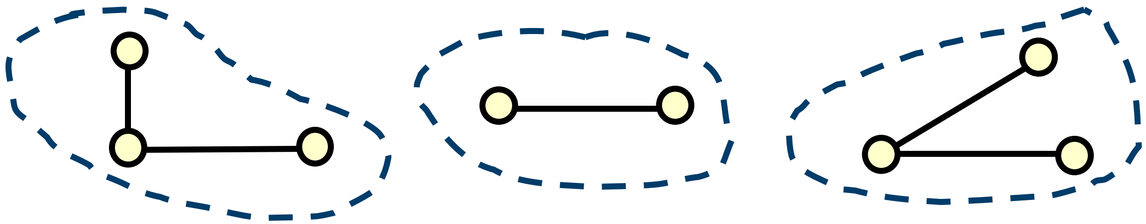
- vertices a & z are adjacent that is {a,z} is an element of the edge set E
- vertices a & b are non-adjacent that is {a,b} is not an element of E
- vertex a is incident to edge {a,x}
- a→x→b→y→c is a path of length 4 (number of edges)
- a→x→b→y→a is a cycle of length 4
- all vertices have degree 3
  - i.e. all vertices are incident to three edges

Graph basics – Definitions

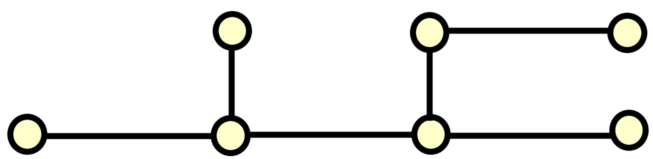
A graph is: connected, if every pair of vertices is joined by a path



A non-connected graph has two or more connected components



- A graph is a tree if it is connected and acyclic (no cycles)

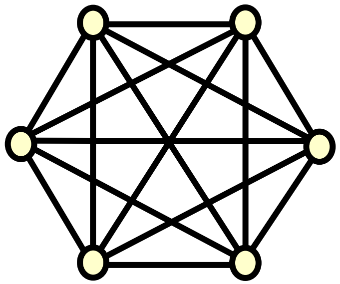


a tree with n vertices has n-1 edge
- at least n-1 edges to be connected
- at most n-1 edges to be acyclic

- A graph is a forest if it is acyclic and components are trees

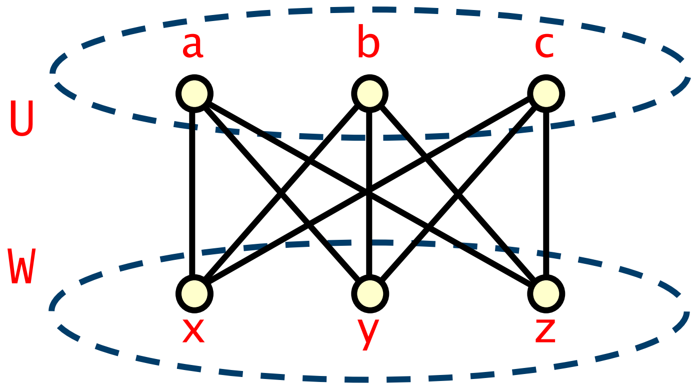
Graph basics – Definitions

A graph is **complete** (**a clique**) if every pair vertices is joined by an edge



K_6, the clique on 6 vertices

A graph is **bipartite** if the vertices are in two **disjoint** sets **U** & **W** and **every** edge joins a vertex in **U** to a vertex in **W**

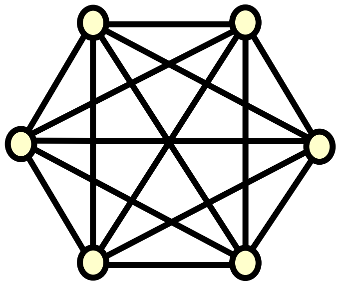


the complete bipartite graph $K_{3,3}$

it is complete since all edges between vertices in U and W are present

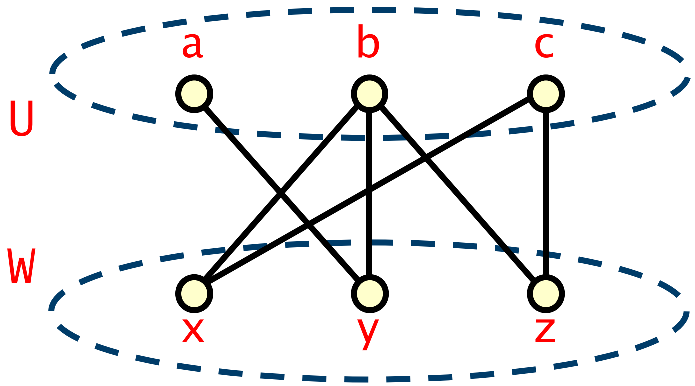
Graph basics – Definitions

A graph is complete (a clique) if every pair vertices is joined by an edge



K_6, the clique on 6 vertices

A graph is bipartite if the vertices are in two disjoint sets U & W and every edge joins a vertex in U to a vertex in W



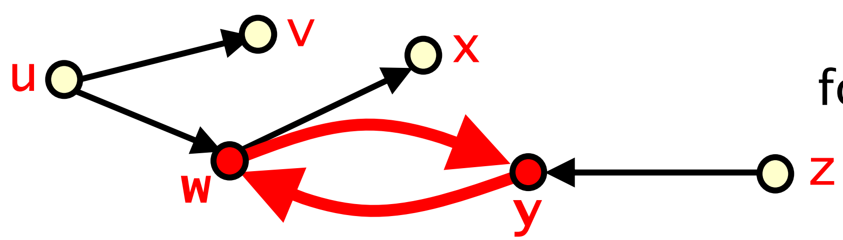
bipartite graphs do not need to be complete

Graph basics – Directed graphs

A **directed graph** (digraph) $D = (V,E)$

- V is the finite set of vertices and E is the finite set of edges
- here each edge is an ordered pair (x,y) of vertices

Pictorially: edges are drawn as directed lines/arrows



for example $(u,v),(w,y),(y,w) \in E$

- u is adjacent to v and v is adjacent from u
- y has in-degree 2 and out-degree 1

In a digraph, paths and cycles must follow edge directions

- e.g. $u \rightarrow w \rightarrow x$ is a path and $w \rightarrow y \rightarrow w$ is a cycle

Graph representations – Undirected graphs

Undirected graph: Adjacency matrix

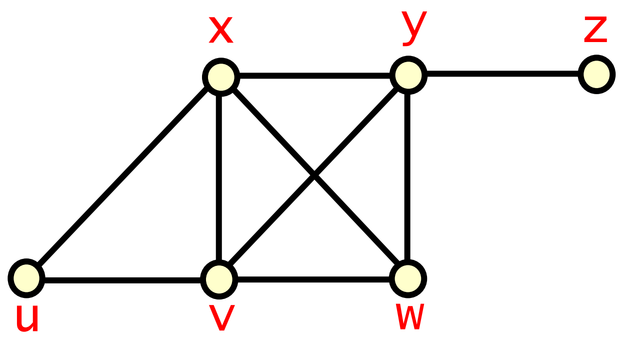
- one row and column for each vertex
- row `i`, column `j` contains a `1` if `i`th and `j`th vertices adjacent, `0` otherwise

Undirected graph: Adjacency lists

- one list for each vertex
- list `i` contains an entry for `j` if the vertices `i` and `j` are adjacent

Graph representations – Undirected graphs

Undirected graph G



Adjacency matrix for **G**

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

$|V| \times |V|$ array

Adjacency lists for **G**

u: v→x
v: u→w→x→y
w: v→x→y
x: u→v→w→y
y: v→w→x→z
z: y

$2 \times |E|$ entries in all

Graph representations – Directed graphs

Directed graph: Adjacency matrix

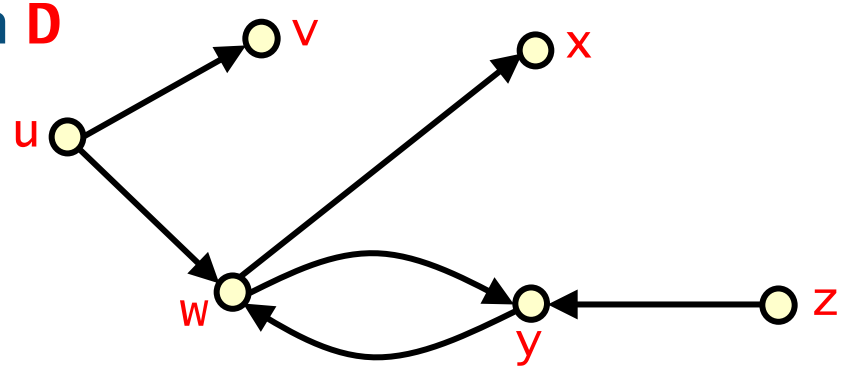
- one row and column for each vertex
- row `i`, column `j` contains a `1` if there is an edge **from** `i` **to** `j` and `0` otherwise

Directed graph: Adjacency lists

- one list for each vertex
- the list for vertex `i` contains vertex `j` if there is an edge **from** `i` **to** `j`

Graph representations – Directed graphs

Directed graph **D**



Adjacency matrix for **D**

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

$|V| \times |V|$ array

Adjacency lists for **D**

u: v→w
v:
w: x→y
x:
y: w
z: y

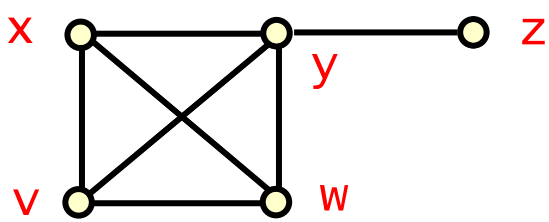
$|E|$ entries in all

Implementation – Adjacency lists

Recall adjacency list for an undirected graph

- one list for each vertex
- list `i` contains an element for `j` if the vertices `i` and `j` are adjacent

graph G



adjacency lists for G

```
v: w→x→y
w: v→x→y
x: v→w→y
y: v→w→x→z
z: y
```

Implementation: define classes for

- the entries of adjacency lists
- the vertices (includes a linked list representing its adjacency list)
- graphs (includes the size of the graph and an array of vertices)
  - array allows for efficient access using "index" of a vertex

Implementation – Adjacency lists

```
 /** class to represent an entry in the adjacency list of a vertex
in a graph */
 public class AdjListNode {

   private int vertexIndex; // the vertex index of the entry

   // possibly other fields, for example representing properties
   // of the edge such as weight, capacity, …

   /** creates a new entry for vertex indexed i */
   public AdjListNode(int i){
     vertexIndex = i;
   }
   public int getVertexIndex(){ // gets the vertex index of the entry
     return vertexIndex;
   }
   public void setVertexIndex(int i){ // sets vertex index to i
     vertexIndex = i;
   }
 }
```

Implementation – Adjacency lists

```
import java.util.LinkedList; // we require the linked list class

/** class to represent a vertex in a graph */
public class Vertex {

  private int index; // the index of this vertex
  private LinkedList<AdjListNode> adjList; // the adjacency list of vertex

  // possibly other fields, e.g. representing data stored at the node

  /** create a new instance of vertex with index i */
  public Vertex(int i) {
    index = i; // set index
    adjList = new LinkedList<AdjListNode>();// create empty adjacency list
  }

   /** return the index of the vertex */
  public int getIndex(){
    return index;
  }
```

Implementation – Adjacency lists

```
 // class Vertex continued

   /** set the index of the vertex */
   public void setIndex(int i){
    index = i;
   }
   /** return the  adjacency list of the vertex */
   public LinkedList<AdjListNode> getAdjList(){
    return adjList;
   }
   /** add vertex with index j to the adjacency list */
   public void addToAdjList(int j){
     adjList.addLast(new AdjListNode(j));
   }
   /** return the degree of the vertex */
   public int vertexDegree(){
     return adjList.size();
   }
}
```

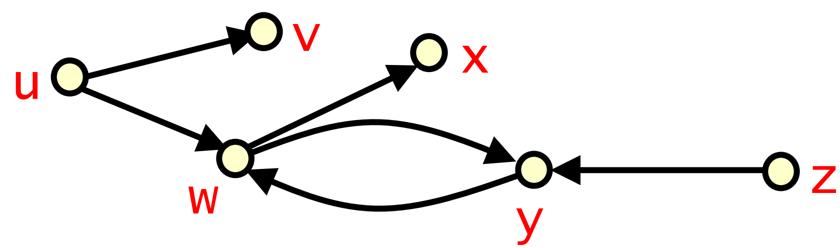
Implementation – Adjacency lists

```java
import java.util.LinkedList; // again require the linked list class
/** class to represent a graph */
public class Graph {

  private Vertex[] vertices; // the vertices
  private int numVertices = 0; // number of vertices

  // possibly other fields representing properties of the graph

  /** Create a Graph with n vertices indexed 0,...,n-1  */
  public Graph(int n) {
    numVertices = n;
    vertices = new Vertex[n];
    for (int i = 0; i < n; i++) vertices[i] = new Vertex(i);
  }
  /** returns number of vertices in the graph */
  public int size(){
    return numVertices;
  }
}
```

Graph search and traversal algorithms

Graph search and traversal algorithms

- a systematic way to explore a graph (when starting from some vertex)



Example: web crawler collects data from hypertext documents by traversing a directed graph **D** where

- vertices are hypertext documents
- (u,v) is an edge if document u contains a hyperlink to document v

A search/traversal visits all vertices by travelling along edges

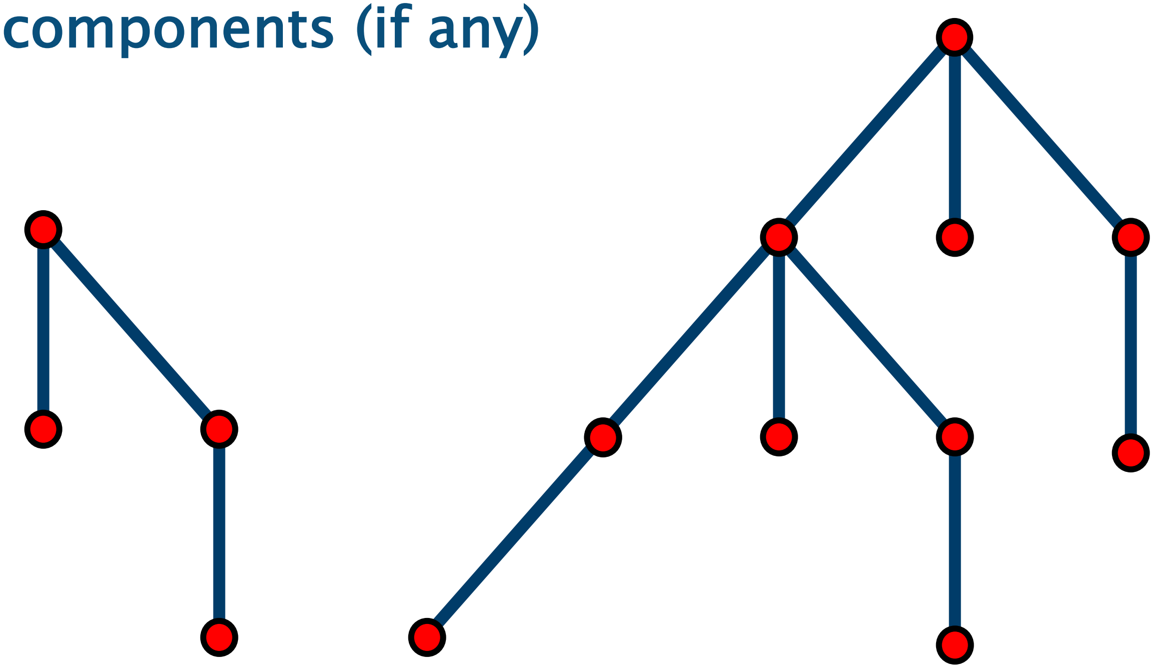
- traversal is efficient if it explores graph in $O(|V|+|E|)$ time

Depth first search/traversal (DFS)

From starting vertex

- follow a path of unvisited vertices until path can be extended no further
- then backtrack along the path until an unvisited vertex can be reached
- continue until we cannot find any unvisited vertices

Repeat for other components (if any)



Depth first search/traversal (DFS)

From starting vertex

- follow a path of unvisited vertices until path can be extended no further
- then backtrack along the path until an unvisited vertex can be reached
- continue until we cannot find any unvisited vertices

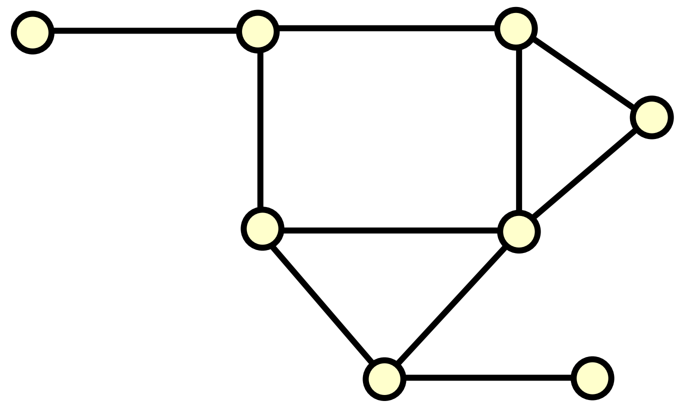
Repeat for other components (if any)

The edges traversed form a spanning tree (or forest)

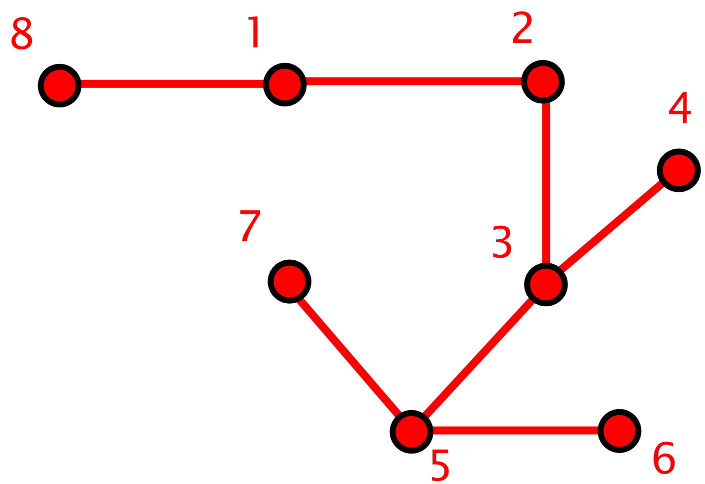
- a depth-first spanning tree (forest)
- spanning tree of a graph is tree composed of all the vertices and some (or perhaps all) of the edges of the graph

Depth first traversal – Example

Undirected graph **G**



Depth first spanning tree of **G**



Implementation – DFS – Add to vertex class

```
private boolean visited; // has vertex been visited in a traversal?

private int pred; // index of the predecessor vertex in a traversal

public boolean getVisited(){
  return visited;
}
public void setVisited(boolean b){
  visited = b;
}
public int getPred(){
  return pred;
}
public void setPred(int i){
  pred = i;
}
```

Implementation – DFS – Add to graph class

```
/** visit vertex v, with predecessor index p, during a dfs */
private void visit(Vertex v, int p){
  v.setVisited(true); // update as now visited
  v.setPred(p);  // set predecessor (indicates edge used to find vertex)
  LinkedList<AdjListNode> L = v.getAdjList(); // get adjacency list

  for (AdjListNode node : L){ // go through all adjacent vertices
    int i = node.getIndex(); // find index of current vertex in list
    if (!vertices[i].getVisited()) // if vertex has not been visited
      visit(vertices[i], v.getIndex()); // continue dfs search from it
      // setting the predecessor vertex index to the index of v
  }
}
/** carry out a depth first search/traversal of the graph */
public void dfs(){
  for (Vertex v : vertices) v.setVisited(false); // initialise
  for (Vertex v : vertices) if (!v.getVisited()) visit(v,-1);
  // if vertex is not yet visited, then start dfs on vertex
  // -1 is used to indicate v was not found through an edge of the graph
}
```

Analysis – Depth first search

Each vertex is visited, and each element in the adjacency lists is processed, so overall $O(n+m)$

- where n is the number of vertices and m the number of edges

Can be adapted to the adjacency matrix representation

- but now $O(n^2)$ since look at every entry of the adjacency matrix

Some applications

- to determine if a given graph is connected
- to identify the connected components of a graph
- to determine if a given graph contains a cycle (see tutorial questions)
- to determine if a given graph is bipartite (see tutorial questions)

Breadth first search/traversal (BFS)

Search fans out as widely as possible at each vertex

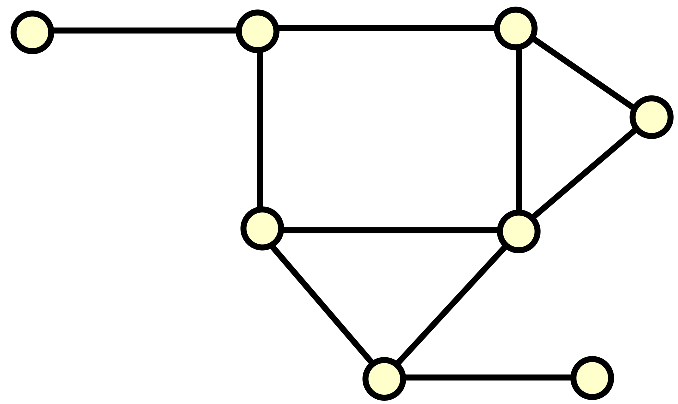
- from the current vertex, visit all the adjacent vertices
  this is referred to as processing the current vertex
- vertices are processed in the order in which they are visited
- continue until all vertices in current component have been processed
- then repeat for other components
  (if there are any)

Again the edges traversed form a spanning tree (or forest)

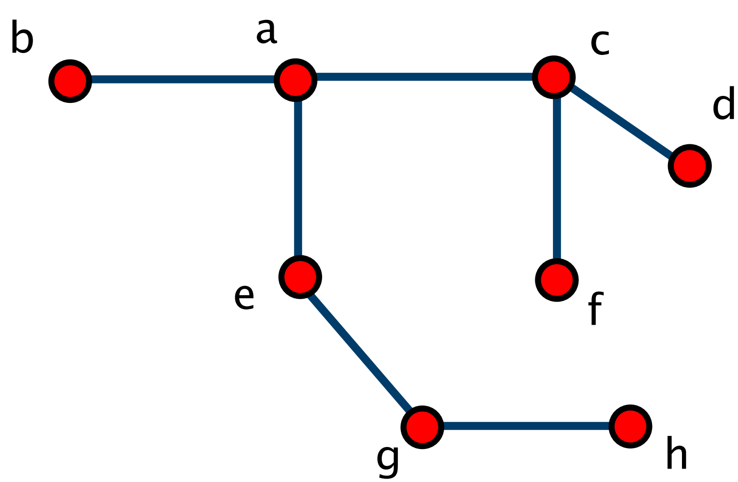
- a breadth-first spanning tree (forest)
- spanning tree of a graph is tree composed of all the vertices and some
  (or perhaps all) of the edges of the graph

Breadth first traversal – Example

Undirected graph **G**



Breadth first spanning tree of **G**



Analysis – Breadth first search

Complexity

- each vertex is visited and queued exactly once
- each adjacency list is traversed once
- so overall O(n+m) (n is the number of vertices and m number of edges)
- can adapt to adjacency matrix representation but O(n^2) (as for DFS)

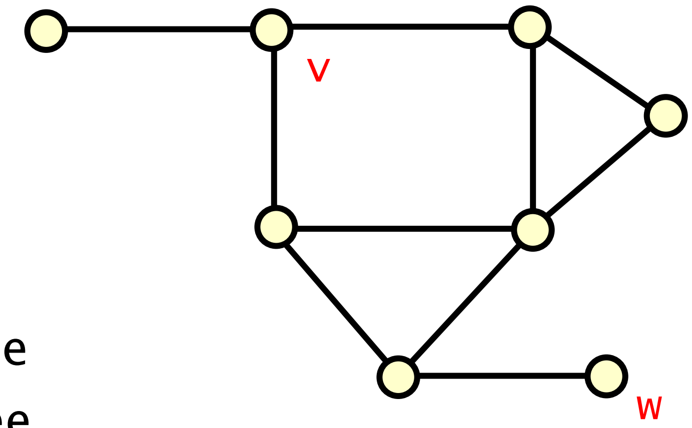
Example application

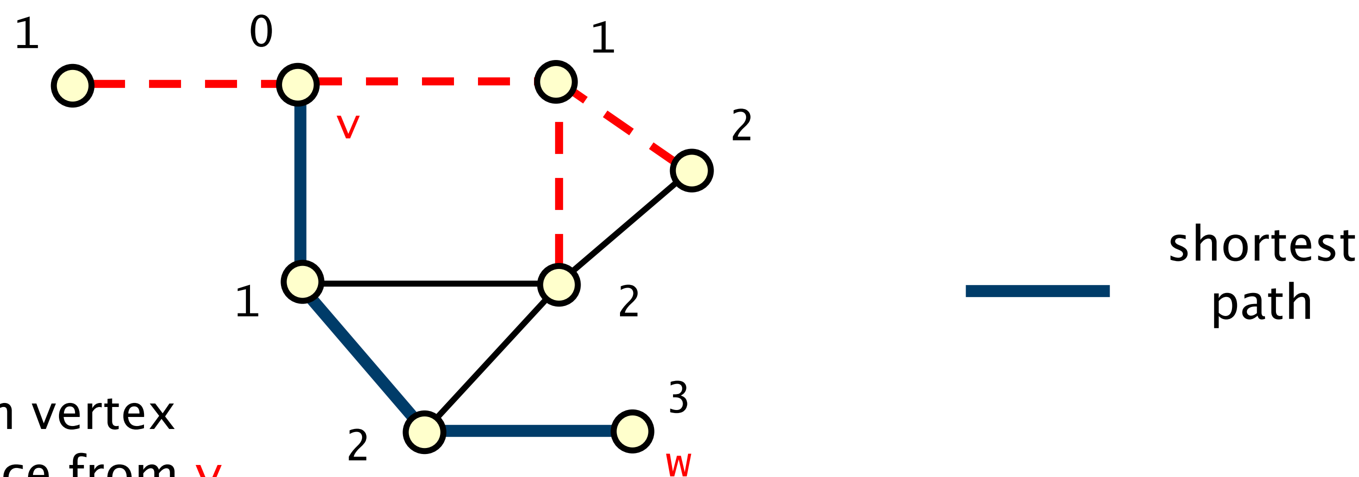
- finding the distance between two vertices, say v and w, in a graph
- the distance is the number of edges in the shortest path from v to w
- assign distance to v to be 0
- carry out a breadth-first search from v
- when visiting a new vertex for first time, assign its distance to be 1 + the distance to its predecessor in the BF spanning tree
- stop when w is reached

Distance between two vertices – Example

Distance between v and w

- assign distance to v to be 0
- carry out a breadth-first search from v
- when visiting a new vertex for first time
  assign its distance to be 1 + the distance
  to its predecessor in the BF spanning tree



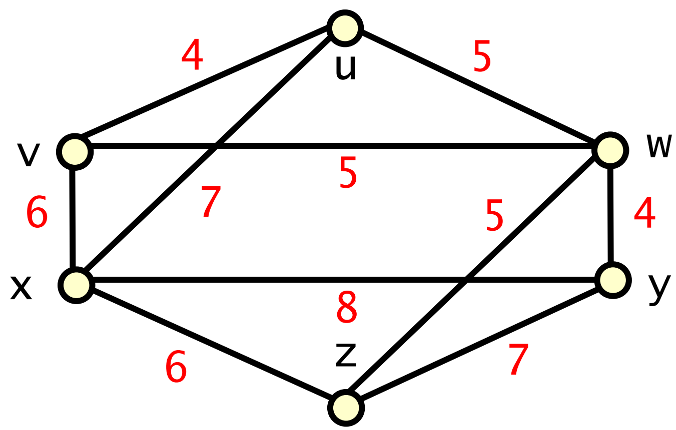


number beside each vertex
indicates the distance from v

Weighted graphs

Each edge **e** has an integer **weight** given by `wt(e)>0`

- graph may be undirected or directed
- weight may represent length, cost, capacity, etc
- if an edge is not part of the graph its weight is infinity



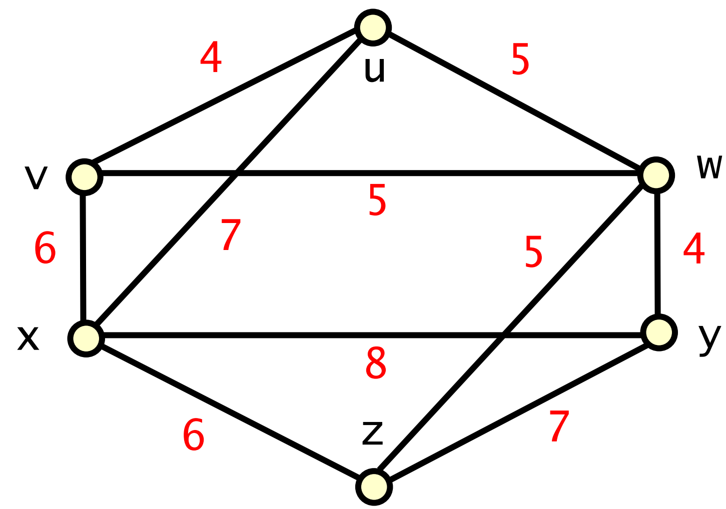
Example: cost of sending a message down a particular edge

- could be a monetary cost or some combination of time and distance
- can be used to formulate the shortest path problem for routing packets

Weighted graphs – Representation

Adjacency matrix becomes **weight matrix**

Adjacency lists include weight in node



adjacency matrix

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

adjacency list

u:v(4)→w(5)→x(7)
v:u(4)→w(5)→x(6)
w:u(5)→v(5)→y(4)→z(5)
x:u(7)→v(6)→y(8)→z(6)
y:w(4)→x(8)→z(7)
z:w(5)→x(6)→y(7)

Weighted graphs – Shortest Paths

Given a weighted (un)directed graph and two vertices **u** and **v** find a **shortest path** between **u** and **v** (for directed from **u** to **v**)

- where the length of a path is the sum of the weights of its edges

Example: weights are distances between airports

- shortest path between San Francisco and Miami

Applications include:

- flight reservations
- internet packet routing
- driving directions

Edsger Dijkstra, in an interview in 2010...

"… the algorithm for the shortest path, which I designed in about 20 minutes. One morning I was shopping in Amsterdam with my young fiancé, and tired, we sat down on the cafe terrace to drink a cup of coffee, and I was just thinking about whether I could do this, and I then designed the algorithm for the shortest path."

Dijkstra, E.W. A note on two problems in Connexion with graphs. Numerische Mathematik 1, 269–271 (1959)

Dijkstra describes the algorithm in English in 1956 (he was 26 years old)

- most people were programming in assembly language
- only one high-level language: Fortran by John Backus at IBM and not quite finished

No big O notation in 1959, in the paper, Dijkstra says: “my solution is preferred to another one … the amount of work to be done seems considerably less.”

Dijkstra's algorithm (as seen in NOSE2)

Algorithm finds shortest path between one vertex u and all others

- based on maintaining a set S containing all vertices for which shortest path with u is currently known
- S initially contains only u (obviously shortest path between u and u is 0)
- eventually S contains all the vertices (so all shortest paths are known)

Each vertex v has a label d(v) indicating the length of a shortest path between u and v passing only through vertices in S

- if no path exists then we set to d(v) infinity
- if v is in S, then d(v) is the length of the shortest path between u and v

Invariant of the algorithm: if v is in S and w is not, then the length of the shortest path between u and w is at least that between u and v

- this means the weight of the edge between u and w is at least d(v)

Dijkstra's algorithm (as seen in NOSE2)

Algorithm finds shortest path between one vertex **u** and all others

- based on maintaining a set S containing all vertices for which shortest path with u is currently known
- S initially contains only u (obviously shortest path between u and u is 0)
- eventually S contains all the vertices (so all shortest paths are known)

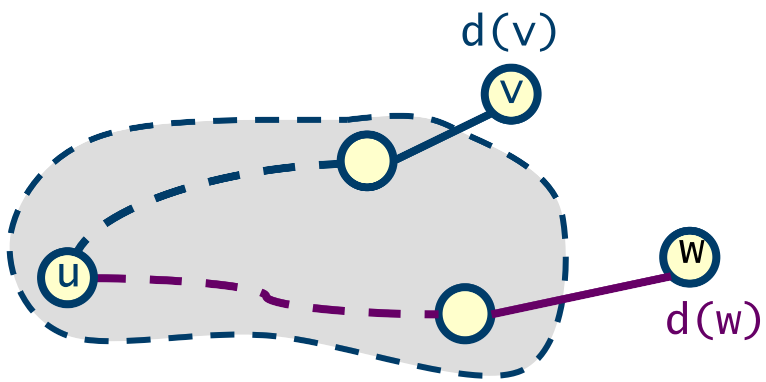
Each vertex **v** has a label **d(v)** indicating the length of a shortest path between **u** and **v** passing **only** through vertices in **S**

- at each step we add to S the vertex v not in S such that d(v) is minimum
- after having added a vertex v to S, carry out edge relaxation operations i.e. we update the length d(w) for all vertices w still not in S
  - d(w) is the length of a shortest path between u and v passing only through vertices in S
  - and S has changed since we have added vertex v to S

Dijkstra's algorithm – Edge relaxation

Each vertex v has a label d(v) indicating the length of a shortest path between u and v passing only through vertices in S

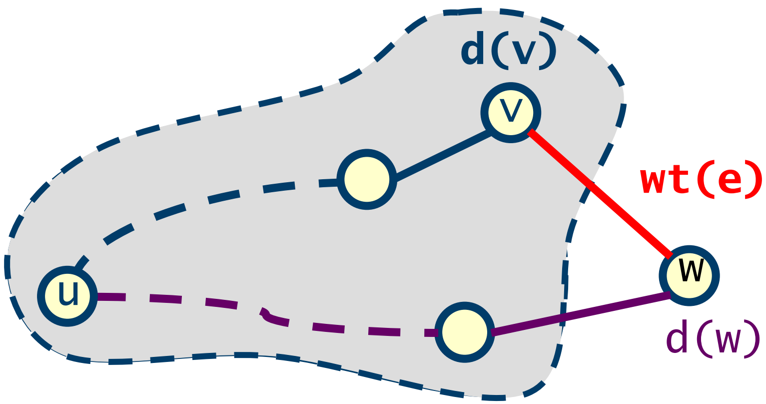
- suppose v and w are not in S then we know
  - the shortest path between u and v passing only through S equals d(v)
  - the shortest path between u and w passing only through S equals d(w)



Dijkstra's algorithm – Edge relaxation

Each vertex v has a label d(v) indicating the length of a shortest path between u and v passing only through vertices in S

- suppose v and w are not in S then we know
  - the shortest path between u and v passing only through S equals `d(v)`
  - the shortest path between u and w passing only through S equals `d(w)`
- now suppose v is added to S and the edge `e = {v,w}` has weight `wt(e)`
- calculate the shortest path between u and w passing only through `S∪{v}`



shortest path is either:

- original path through S of length `d(w)`
- path combining edge e and shortest path between v and u which has length `wt(e) + d(v)`

therefore length updated to:

`d(w) = min{ d(w), d(v) + wt(e) }`

Dijkstra's algorithm – Pseudo code

```
// S is set of vertices for which shortest path with u is known
// d(w) represents length of a shortest path between u and w
// passing only through vertices of S

S = {u}; // initialise S
for (each vertex w) d(w) = wt(u,w); // initialise lengths

while (S != V){ // still vertices to add to S
  find v not in S with d(v) minimum;
  add v to S;
  for (each w not in S and adjacent to v) // perform relaxation
    d(w) = min{ d(w) , d(v)+wt(v,w) };
}
```

Dijkstra's algorithm – Complexity

```
S = {u}; // initialise S
 for (each vertex w) d(w) = wt(u,w); // initialise lengths

 while (S != V){ // still vertices to add to S
   find v not in S with d(v) minimum;
   add v to S;
   for (each w not in S and adjacent to v) // perform relaxation
     d(w) = min{ d(w) , d(v)+wt(v,w) };
 }
```

Analysis (n vertices and m edges) using unordered array for lengths

- $O(n)$ to initialise lengths
- finding minimum is $O(n^2)$ overall
  - each time it takes $O(n)$ and there are $n-1$ to find
- relaxation is $O(m)$ overall
  - each edge is considered once and updating length takes $O(1)$
  - note: we are not considering each iteration of the while loop but overall ops

hence $O(n^2)$ overall (number of edges at most $n(n-1)$)

Dijkstra's algorithm – Pseudo code

```
S = {u}; // initialise S
 for (each vertex w) d(w) = wt(u,w); // initialise lengths

 while (S != V){ // still vertices to add to S
   find v not in S with d(v) minimum;
   add v to S;
   for (each w not in S and adjacent to v) // perform relaxation
     d(w) = min{ d(w) , d(v)+wt(v,w) };
 }
```

Analysis (**n** vertices and **m** edges) using a heap for lengths

- `O(n)` to initialise lengths and create heap
- finding minimum is `O(n log n)` overall
  - each time it takes `O(log n)` and there are `n-1` to find
- relaxation is `O(m log n)` overall
  - each edge is considered once and updating length takes `O(log n)`
  - note: this involves updating a specific value in the heap not the root so care must be taken (need to keep track of positions of vertices in the heap)

Dijkstra's algorithm – Pseudo code

```
S = {u}; // initialise S
 for (each vertex w) d(w) = wt(u,w); // initialise lengths

 while (S != V){ // still vertices to add to S
   find v not in S with d(v) minimum;
   add v to S;
   for (each w not in S and adjacent to v) // perform relaxation
     d(w) = min{ d(w) , d(v)+wt(v,w) };
 }
```

Analysis (**n** vertices and **m** edges) using a heap for lengths

- `O(n)` to initialise lengths and create heap
- finding minimum is `O(n log n)` overall
  - each time it takes `O(log n)` and there are `n-1` to find
- relaxation is `O(m log n)` overall
  - each edge is considered once and updating lengths takes `O(log n)`

hence **O(m log n)** overall (more edges than vertices)

- a graph with n vertices has $O(n^2)$ edges

Spanning trees

Spanning tree:

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

Problem: for a weighted connected undirected graph, find a minimum weight spanning tree

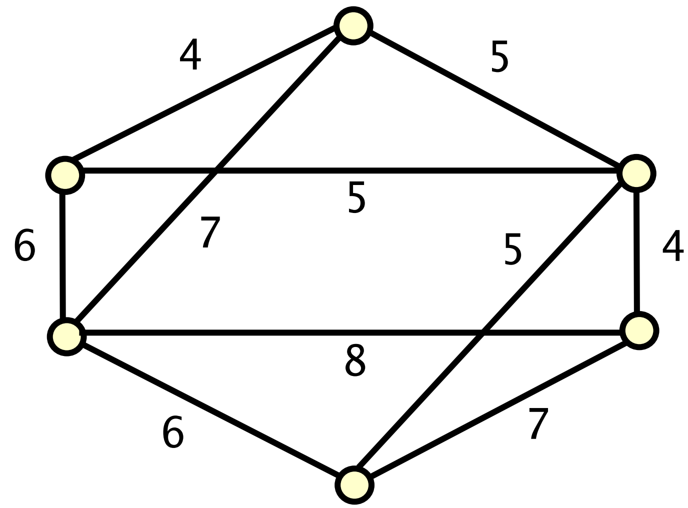
- this represents the 'cheapest' way of interconnecting the vertices

Applications include:

- design of networks for computer, telecommunications, transportation, gas, electricity, ...
- clustering, approximating the travelling salesman problem

Weighted graphs – Example – Spanning tree

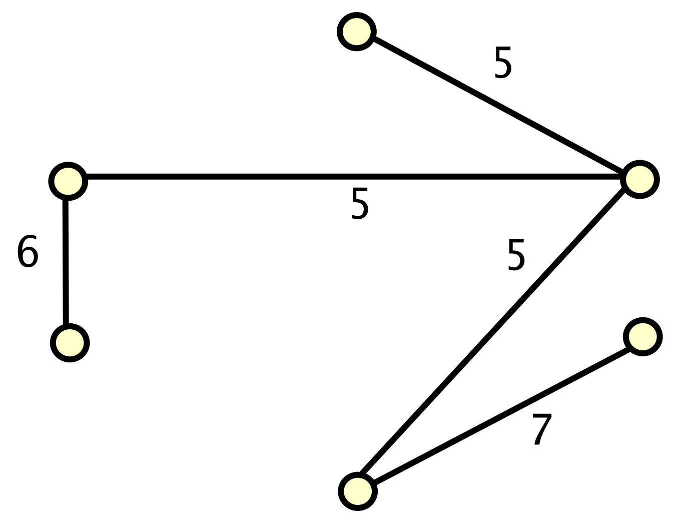
Weighted graph G



spanning tree:
subgraph which is
both a tree and
'spans' every vertex

Spanning tree for G

– weight 28

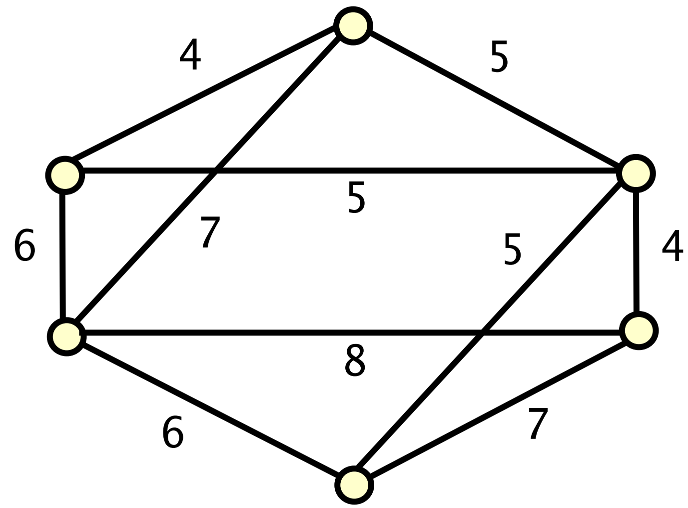


delete edges while still 'spanning' vertices

cannot delete any more edges and we have a tree

Weighted graphs – Example – Spanning tree

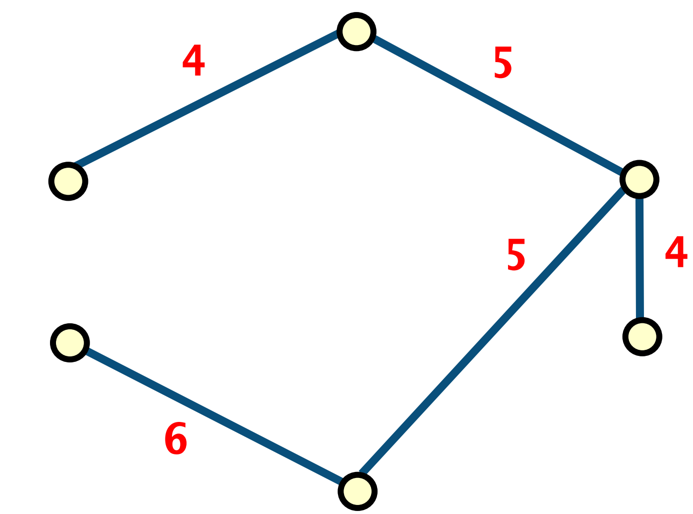
Weighted graph G



spanning tree:
subgraph which is
both a tree and
'spans' every vertex

Spanning tree for G

– weight 24



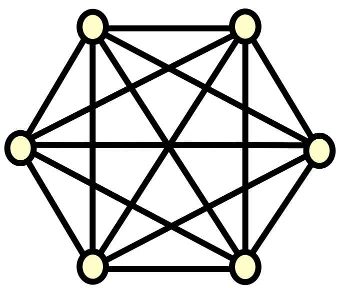
delete edges while still
'spanning' vertices

cannot delete any
more edges and
we have a tree

Minimum weight spanning tree problem

An example of a problem in combinatorial optimisation

- find ‘best’ way of doing something among a (large) number of candidates
- can always be solved, at least in theory, by exhaustive search
- however this may be infeasible in practice
- typically an exponential-time algorithm
- e.g. K_n (clique of size n) has n^{n-2} spanning trees (Cayley’s formula)
  - recall: a graph is a clique if every pair vertices is joined by an edge



- a much more efficient algorithm may be possible
  and is true in the case of minimum weight spanning trees

Minimum weight spanning tree problem

An example of a problem in combinatorial optimisation

- find 'best' way of doing something among a (large) number of candidates
- can always be solved, at least in theory, by exhaustive search
- however this may be infeasible
- typically an exponential-time algorithm

The Prim-Jarnik minimum spanning tree algorithm

- an example of a greedy algorithm
- it makes a sequence of decisions based on local optimality
- and ends up with the globally optimal solution

For many problems, greedy algorithms do not yield optimal solution

- see examples later in the course

The Prim-Jarnik algorithm

Min spanning tree is constructed by choosing a sequence of edge

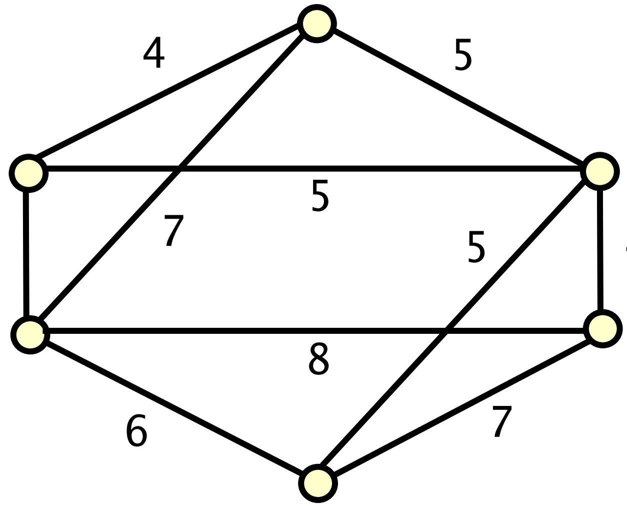
```
set an arbitrary vertex r to be a tree-vertex (tv);
set all other vertices to be non-tree-vertices (ntv);
while (number of ntv > 0){
    find edge e = {p,q} of graph such that
        p is a tv;
        q is an ntv;
        wt(e) is minimised over such edges;
    adjoin edge e to the (spanning) tree;
    make q a tv;
}
```

Analysis (n is the number of vertices)

- intitialisation O(n) (n operations to set vertices to be **tv** or **ntv**)
- the outer loop is executed n-1 times
- the inner loop checks all edges from a tree-vertex to a non-tree-vertex
- there can be $O(n^3)$ of these so overall the algorithm is $O(n^3)$

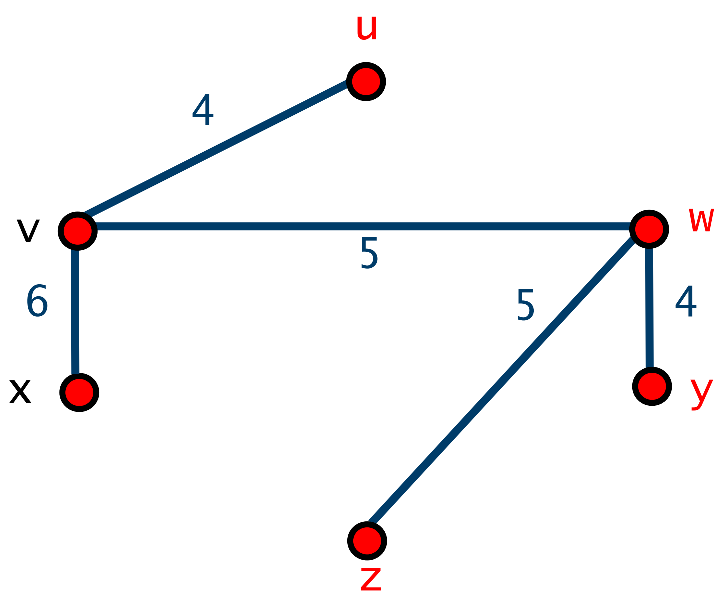
The Prim–Jarnik algorithm – Example

Weighted graph G



Minimum spanning tree for G

- weight 24



Dijkstra's refinement

Introduce a attribute bestTV for each non-tree vertex (**ntv**) q

– bestTV is set to the tree vertex (tv) p for which wt({p,q}) is minimised

```
set an arbitrary vertex r to be a tree-vertex (tv);
set all other vertices to be non-tree-vertices (ntv);
for (each ntv s) set s.bestTV = r; // r is the only tv

while (size of ntv > 0){
  find ntv q for which wt({q, q.bestTV}) is minimal;
  adjoin {q, q.bestTV} to the tree;
  make q a tv;

  for (each ntv s) update s.bestTV;
  // update bestTV as tree vertices have changed

}
```

Dijkstra's refinement – Analysis

```
set an arbitrary vertex r to be a tree-vertex (tv);
set all other vertices to be non-tree-vertices (ntv);
for (each ntv s) set s.bestTV = r; // r is the only tv

while (size of ntv > 0){
  find ntv q for which wt({q, q.bestTV}) is minimal;
  adjoin {q, q.bestTV} to the tree;
  make q a tv;

  for (each ntv s) update s.bestTV; // update as tvs have changed

}
```

– initialisation is O(n)

Dijkstra's refinement – Analysis

```
set an arbitrary vertex r to be a tree-vertex (tv);
set all other vertices to be non-tree-vertices (ntv);
for (each ntv s) set s.bestTV = r; // r is the only tv

while (size of ntv > 0){
  find ntv q for which wt({q, q.bestTV}) is minimal;
  adjoin {q, q.bestTV} to the tree;
  make q a tv;

  for (each ntv s) update s.bestTV; // update as tvs have changed

}
```

- initialisation is O(n)
- while loop is executed n-1 times

Dijkstra's refinement – Analysis

```
set an arbitrary vertex r to be a tree-vertex (tv);
set all other vertices to be non-tree-vertices (ntv);
for (each ntv s) set s.bestTV = r; // r is the only tv

while (size of ntv > 0){
  find ntv q for which wt({q, q.bestTV}) is minimal;
  adjoin {q, q.bestTV} to the tree;
  make q a tv;

  for (each ntv s) update s.bestTV; // update as tvs have changed

}
```

– initialisation is O(n)

– while loop is executed n-1 times

– first part takes O(n)

  - O(n) to find minimal **ntv** and O(1) to adjoin and update

Dijkstra's refinement – Analysis

```
set an arbitrary vertex r to be a tree-vertex (tv);
set all other vertices to be non-tree-vertices (ntv);
for (each ntv s) set s.bestTV = r; // r is the only tv

while (size of ntv > 0){
  find ntv q for which wt({q, q.bestTV}) is minimal;
  adjoin {q, q.bestTV} to the tree;
  make q a tv;

  for (each ntv s) update s.bestTV; // update as tvs have changed

}
```

- initialisation is O(n)
- while loop is executed n-1 times
- second part (inner loop) takes O(n)
  - for each **ntv** s only need to compare weights for **s.bestTV** and new **tv** vertex (i.e. q) to update the value of **s.bestTV**

Dijkstra's refinement – Analysis

```
set an arbitrary vertex r to be a tree-vertex (tv);
set all other vertices to be non-tree-vertices (ntv);
for (each ntv s) set s.bestTV = r; // r is the only tv

while (size of ntv > 0){
  find ntv q for which wt({q, q.bestTV}) is minimal;
  adjoin {q, q.bestTV} to the tree;
  make q a tv;

  for (each ntv s) update s.bestTV; // update as tvs have changed
}
```

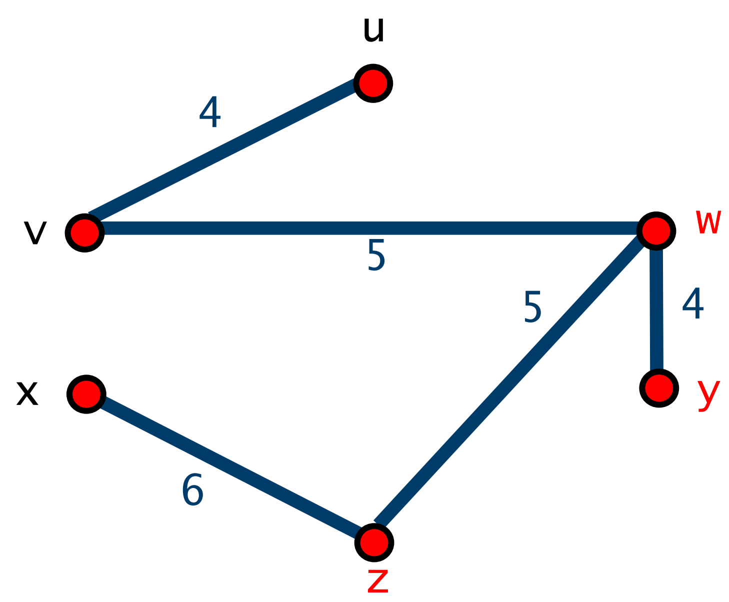
– initialisation is O(n)
– while loop is executed n-1 times
– first part and second part each take O(n)
– overall the algorithm is $O(n^2)$

Dijkstra's refinement – Example

Weighted graph G

Minimum spanning tree for G

- weight 24



q	q.bestTV	wt({q.bestTV,q})
u	–	–
v	–	–
w	–	–
x	–	–
y	–	–
z	–	–

The Prim–Jarnik algorithm – Correctness

Is the algorithm correct ?

- i.e. does it return a minimum weight spanning tree for any graph G

Proof will not be part of the exam

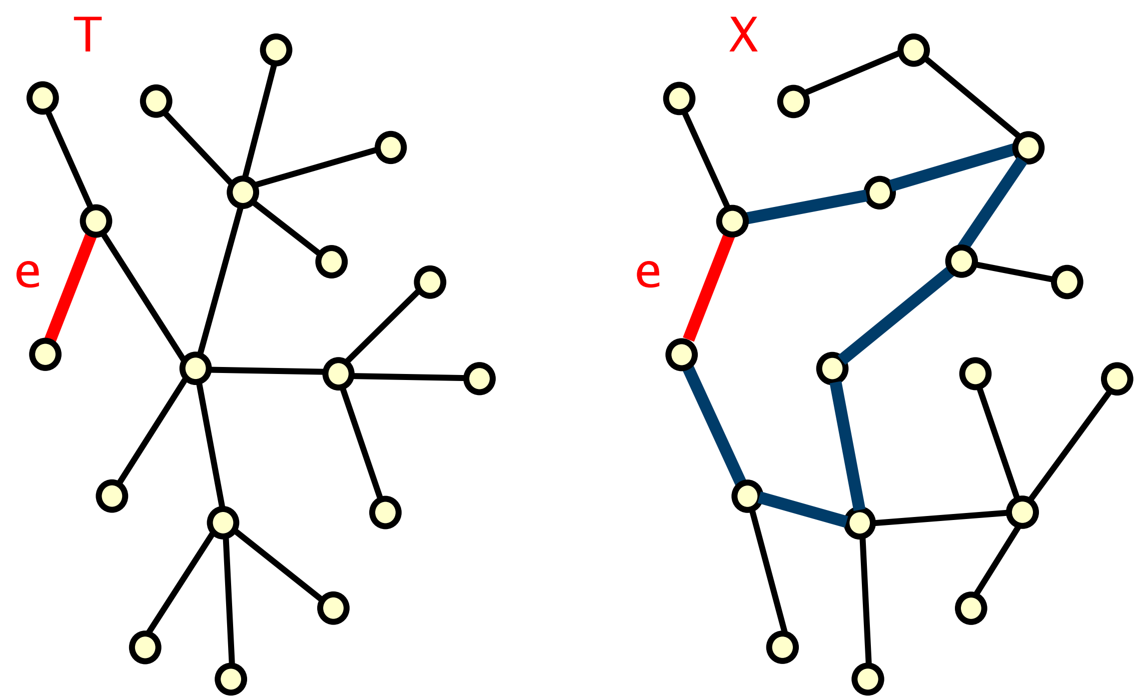
Proof:

- suppose for graph G the algorithm returns the tree T
- compare T with a minimum spanning tree X of G
- if they are the same we are happy (it is a minimum weight spanning tree)
- therefore remains to consider the case when they are different…

The Prim–Jarnik algorithm – Correctness

Suppose that T and X are different

- T tree returned by the algorithm and X a minimum spanning tree of G
- let e be the first edge chosen to be in T that is not in X



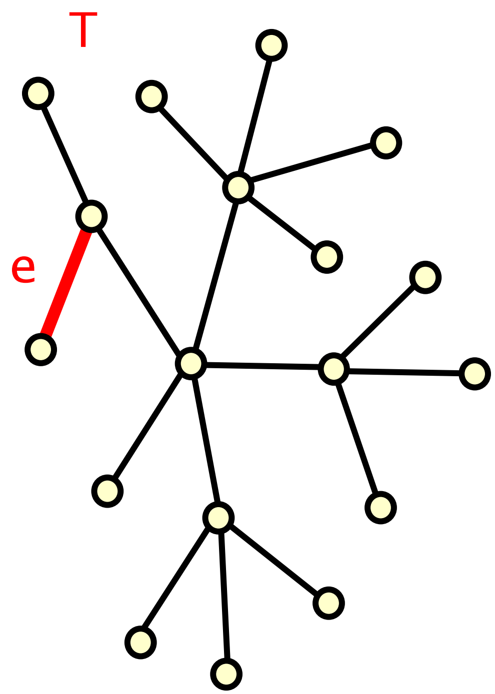
adding e to X we get a cycle C

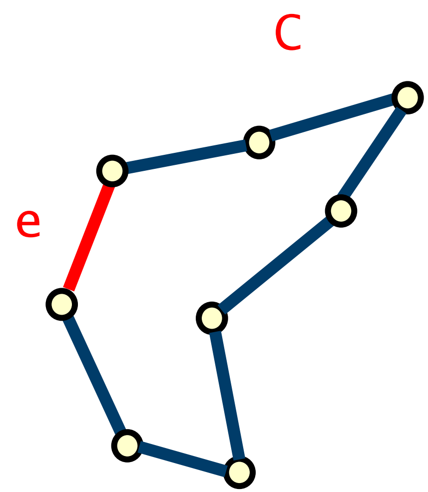
(since X is a spanning tree)

The Prim–Jarnik algorithm – Correctness

Suppose that T and X are different

- T tree returned by the algorithm and X a minimum spanning tree of G
- let e be the first edge chosen to be in T that is not in X





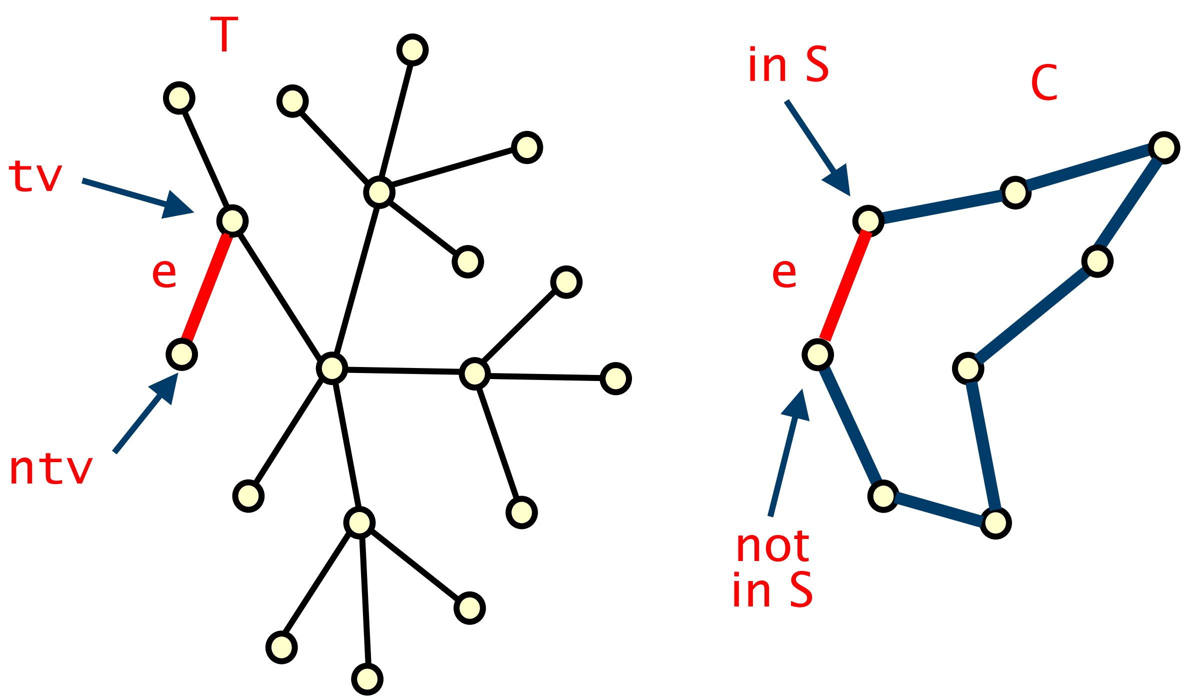
adding e to X we get a cycle C

(since X is a spanning tree)

The Prim–Jarnik algorithm – Correctness

Suppose that T and X are different

- T tree returned by the algorithm and X a minimum spanning tree of G
- let e be the first edge chosen to be in T that is not in X



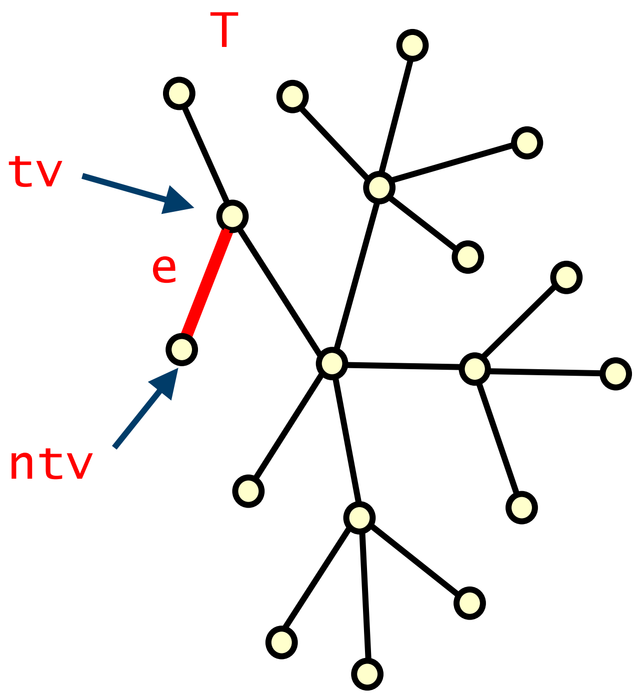
let S be the set of tree verticies (tvs) at the point when the algorithm selected e

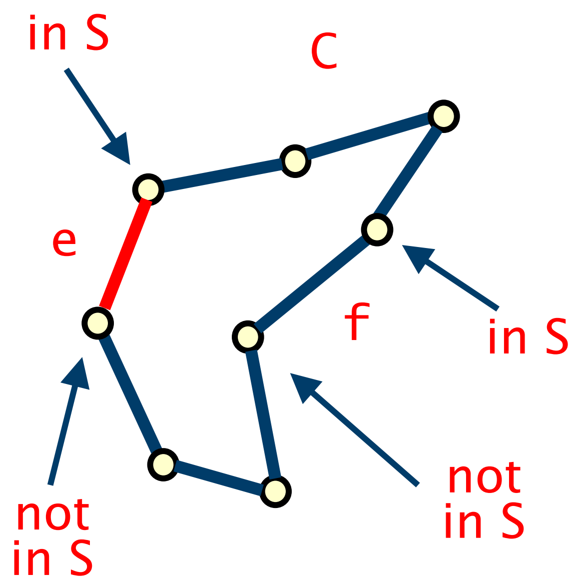
now by definition of the algorithm one end of the edge e is in S, and the other is not in S

The Prim–Jarnik algorithm – Correctness

Suppose that T and X are different

- T tree returned by the algorithm and X a minimum spanning tree of G
- let e be the first edge chosen to be in T that is not in X





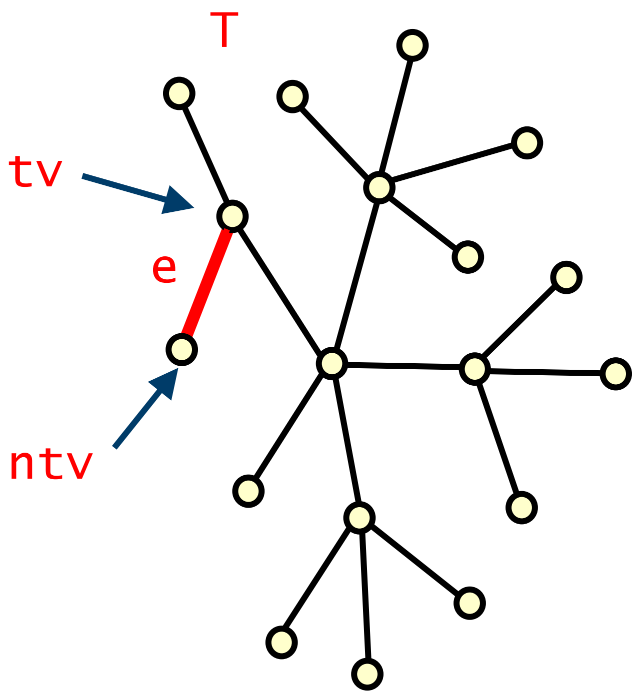
it follows that C must have another edge f that connects a vertex in S with one that is not

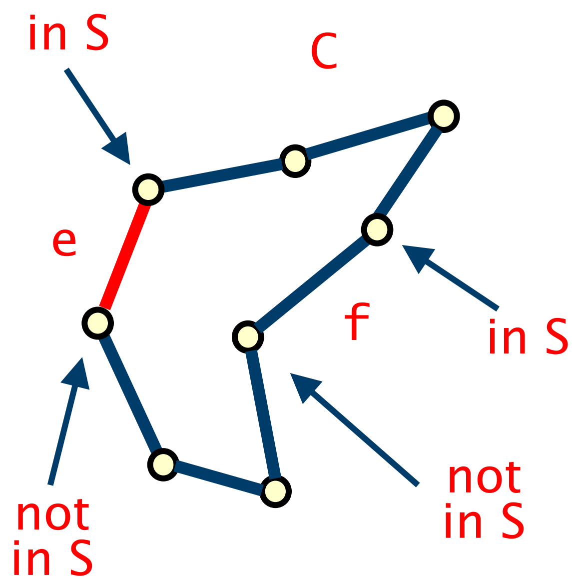
i.e. a tv with a ntv

The Prim–Jarnik algorithm – Correctness

Suppose that T and X are different

- T tree returned by the algorithm and X a minimum spanning tree of G
- let e be the first edge chosen to be in T that is not in X





we also have:

wt(f)≥wt(e)

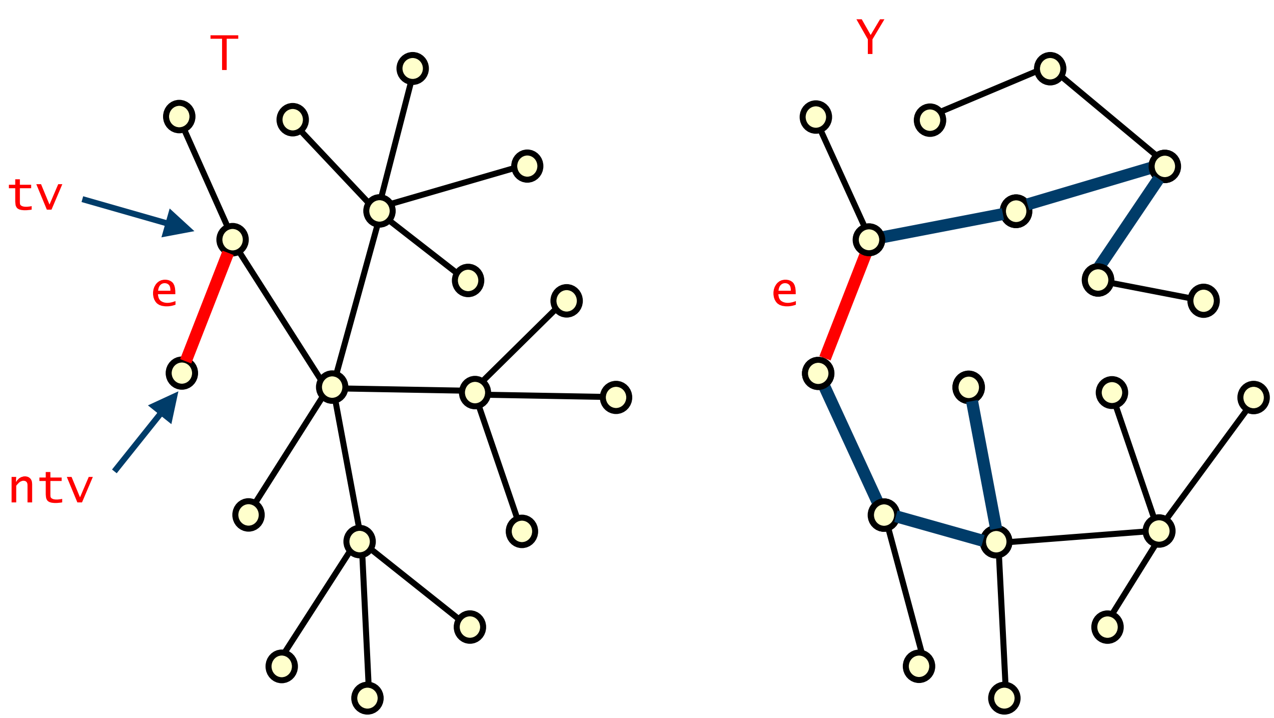
since the algorithm picks e and not f

we can replace f by e in X to get another spanning tree Y

The Prim–Jarnik algorithm – Correctness

Suppose that T and X are different

- T tree returned by the algorithm and X a minimum spanning tree of G
- let e be the first edge chosen to be in T that is not in X



we also have:

wt(f)≥wt(e)

since the algorithm picks e and not f

we can replace f by e in X to get another spanning tree Y

since **wt(f)≥wt(e)**, weight of Y cannot be greater than X, and since X is minimal, Y is minimal

The Prim–Jarnik algorithm – Correctness

Suppose that T and X are different

- T tree returned by the algorithm and X a minimum spanning tree of G
- let e be the first edge chosen to be in T that is not in X

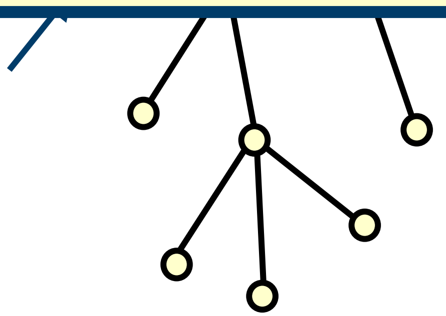
we also have:

continuing the process we can convert X to T maintaining minimality

which proves that T is indeed a minimal spanning tree

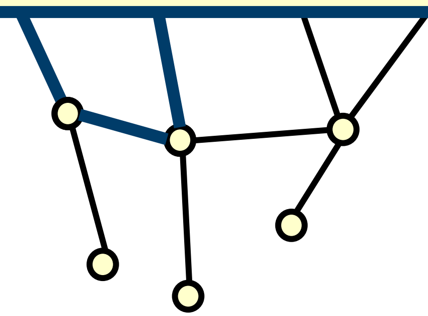
hence the algorithm is correct

T



ntv

Y



spanning tree Y

since **wt(f)≥wt(e)**, weight of Y cannot be greater than X, and since X is minimal, Y is minimal

Directed Acyclic Graphs –Topological ordering

A Directed Acyclic Graph (DAG) is a directed graph with no cycles

A topological order on a DAG is a labelling of the vertices `1,…,n` such that `(u,v)`∈`E` implies `label(u)<label(v)`

- many applications, e.g. scheduling, PERT networks, deadlock detection

A directed graph D has a topological order if and only if it is a DAG

- obviously impossible if D has a cycle (try to label the vertices in a cycle)

A source is a vertex of in-degree 0 and a sink has out-degree 0

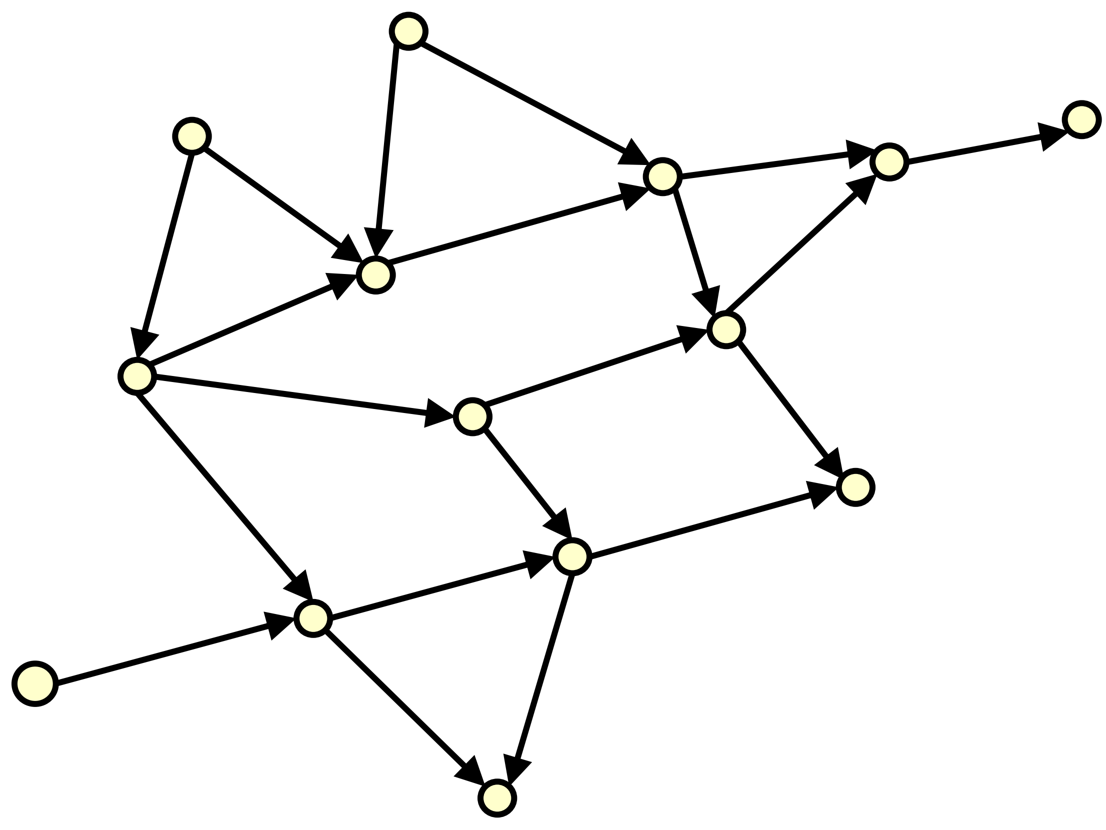
Basic fact: a DAG has at least one source and at least one sink

- forms the basis of a topological ordering algorithm

Directed Acyclic Graphs – Example

Directed acyclic graph D

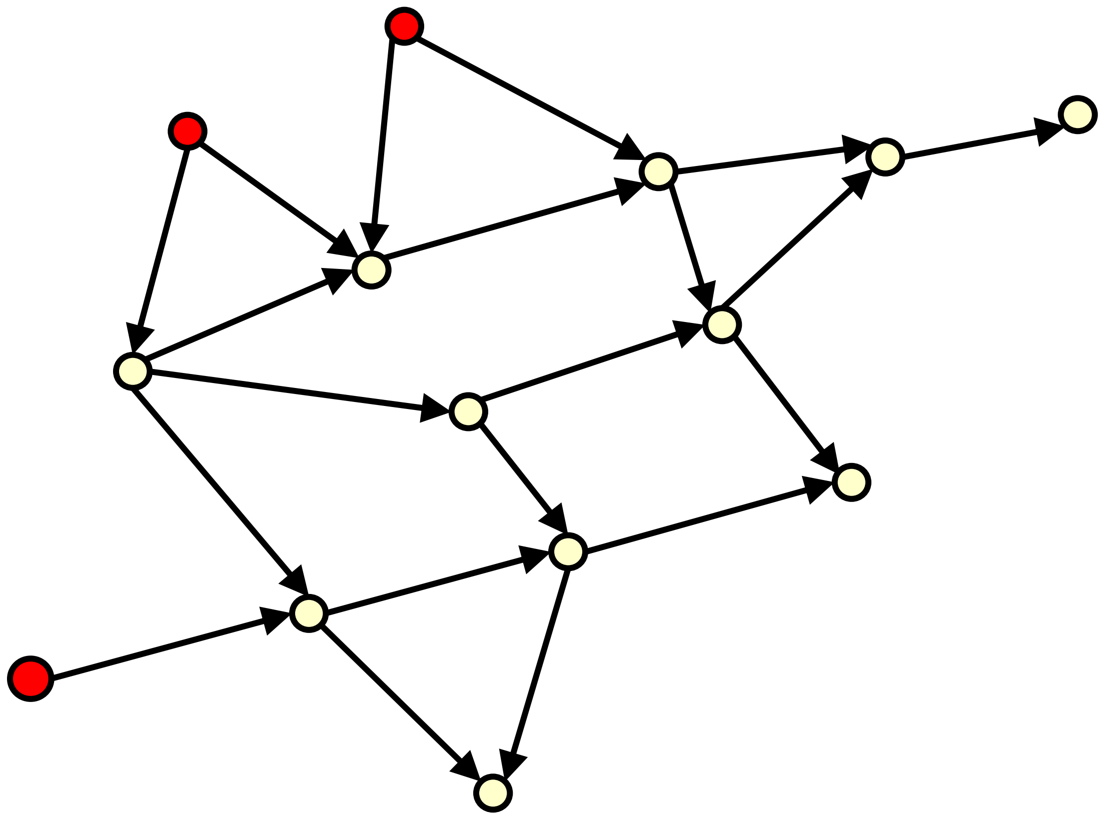
- with more than one source and more than one sink



Directed Acyclic Graphs – Example

Directed acyclic graph D

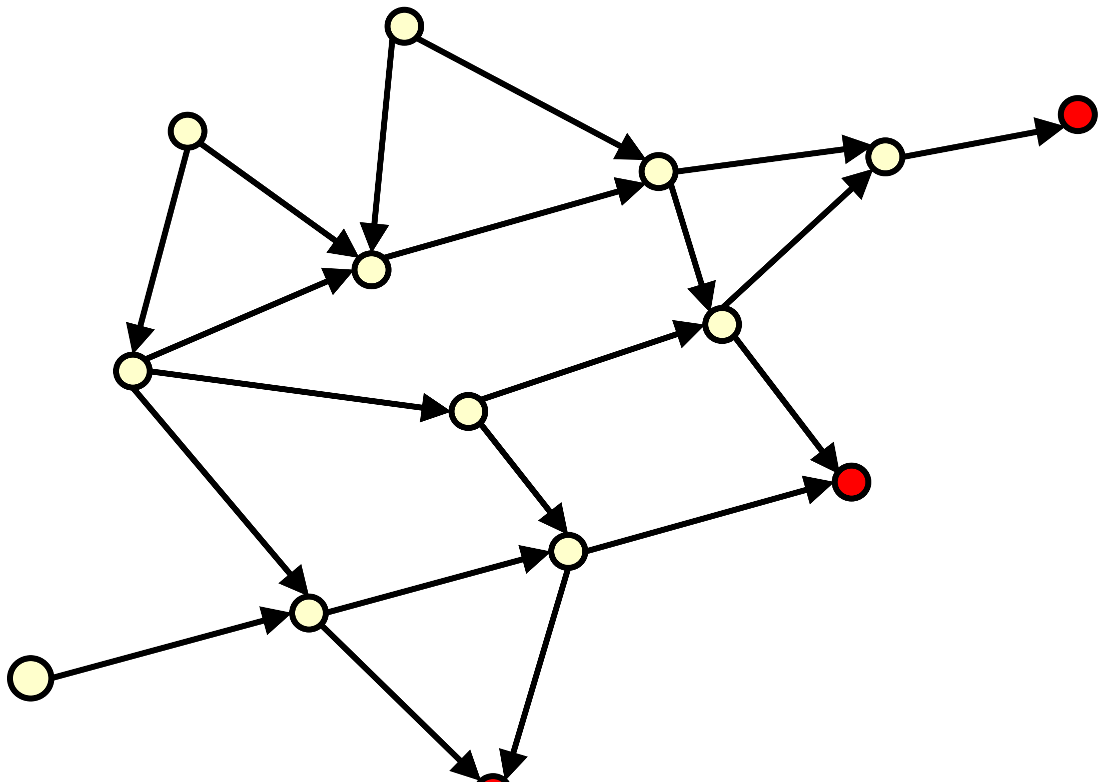
- with more than one source and more than one sink



Directed Acyclic Graphs – Example

Directed acyclic graph D

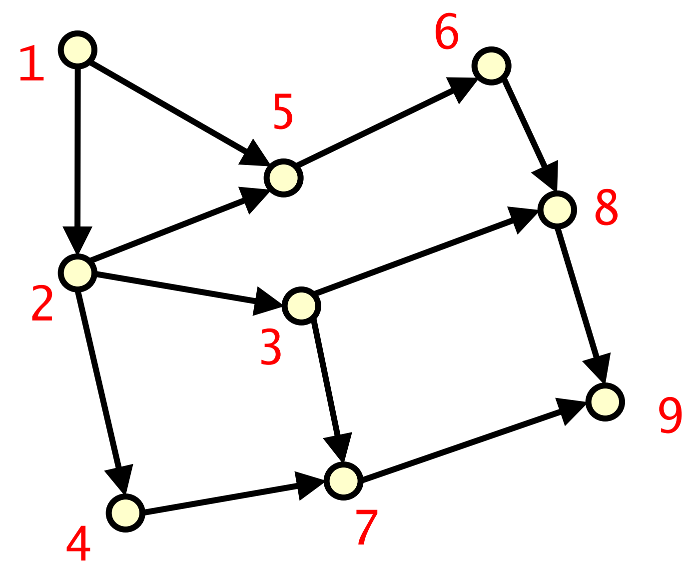
- with more than one source and more than one sink



Directed Acyclic Graphs – Example

Directed acyclic graph **D**

Topological ordering of **D**



Source vertex (in-degree equals 0)

Sink vertex (out-degree equals 0)

A topological order on a DAG is a labelling of the vertices **1,…,n** such that **(u,v)∈E** implies **label(u)<label(v)**

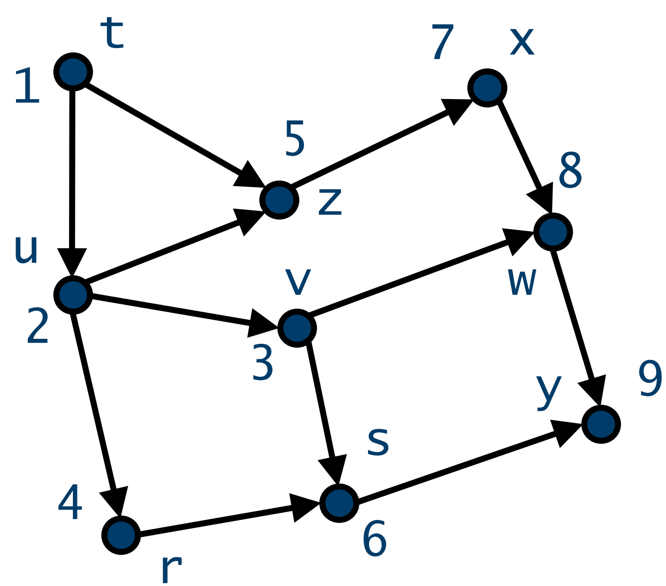
Topological ordering algorithm

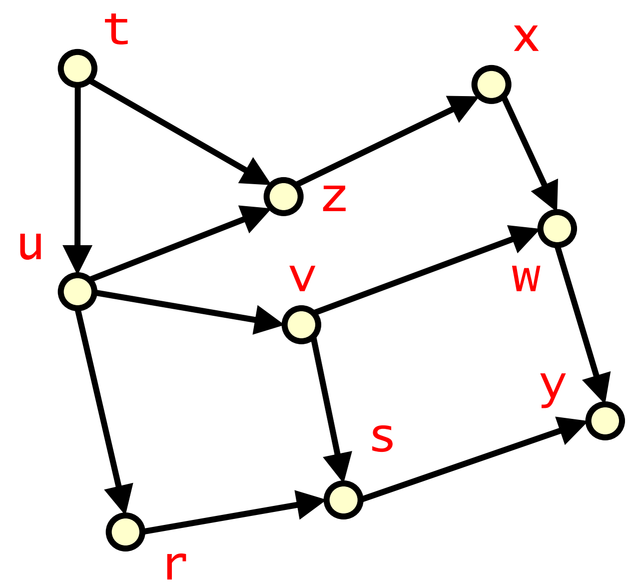
```
// assume each vertex has 2 integer attributes: label and count
// count is the number of incoming edges from unlabelled vertices
// label will give the topological ordering

for (each vertex v) v.setCount(v.getInDegree()); // initial count values

Set up an empty sourceQueue

for (each vertex v) // add vertices with no incoming edges to the queue
  if (v.getCount() == 0) add v to sourceQueue; // i.e. source vertices

int nextLabel = 1; // initialise labelling (gives topological ordering)
while (sourceQueue is non-empty){
  dequeue v from sourceQueue;
  v.setLabel(nextLabel++); // label vertex (and increment nextLabel)
  for (each w with (v,w) ∈ E){ // consider each vertex w adjacent from v
    w.setCount(w.getCount() - 1); // update attribute count
    // add vertex to source queue if there are no incomming vertices
    if (w.getCount() == 0) add w to sourceQueue;
  }
}
```

Topolgocal ordering – Example

Directed acyclic graph D

source queue: ⟨⟩





a topological ordering on D

Topological ordering algorithm – Correctness

A vertex is given a label only when the number of incoming edges from unlabelled vertices is zero

- all predecessor vertices must already be labelled with smaller numbers

Analysis (n vertices, m edges)

- **for adjacency matrix representation**
  - finding in-degree of each vertex is $O(n^2)$ (scan each column)
  - main loop is executed n times within it one row is scanned $O(n)$
  - so overall the algorithm is $O(n^2)$

Topological ordering algorithm – Correctness

A vertex is given a label only when the number of incoming edges from unlabelled vertices is zero

- all predecessor vertices must already be labelled with smaller numbers
- dependent on using a queue (first in first out for labelling)

Topological ordering algorithm – Analysis

Analysis (`n` vertices, `m` edges)

- **for adjacency lists representation**
  - finding in-degree of each vertex is `O(n+m)` (scan adjacency lists)
  - main loop is executed `n` times within it one list is scanned (and the same list is never scanned twice)
  - so every list is scanned again and overall algorithm is `O(n+m)`

Deadlock detection

Determining whether a digraph contains a cycle

Method 1 (an adaptation of the topological ordering algorithm)

- if the source queue becomes empty before all vertices are labelled, then there must be a cycle
- if all vertices can be labelled, then the digraph is acyclic

Method 2 (an adaptation of depth-first-search)

- when a vertex u is '`visited`' check whether there is an edge from u to a vertex v which is on the current path from the current starting vertex
- the existence of such a vertex indicates a cycle (adaptation of depth first search since need to 'remember' current path)
- see tutorials for more detail