COMP225: Algorithms and Data Structures

Applications of Graphs (1)

Mark Dras

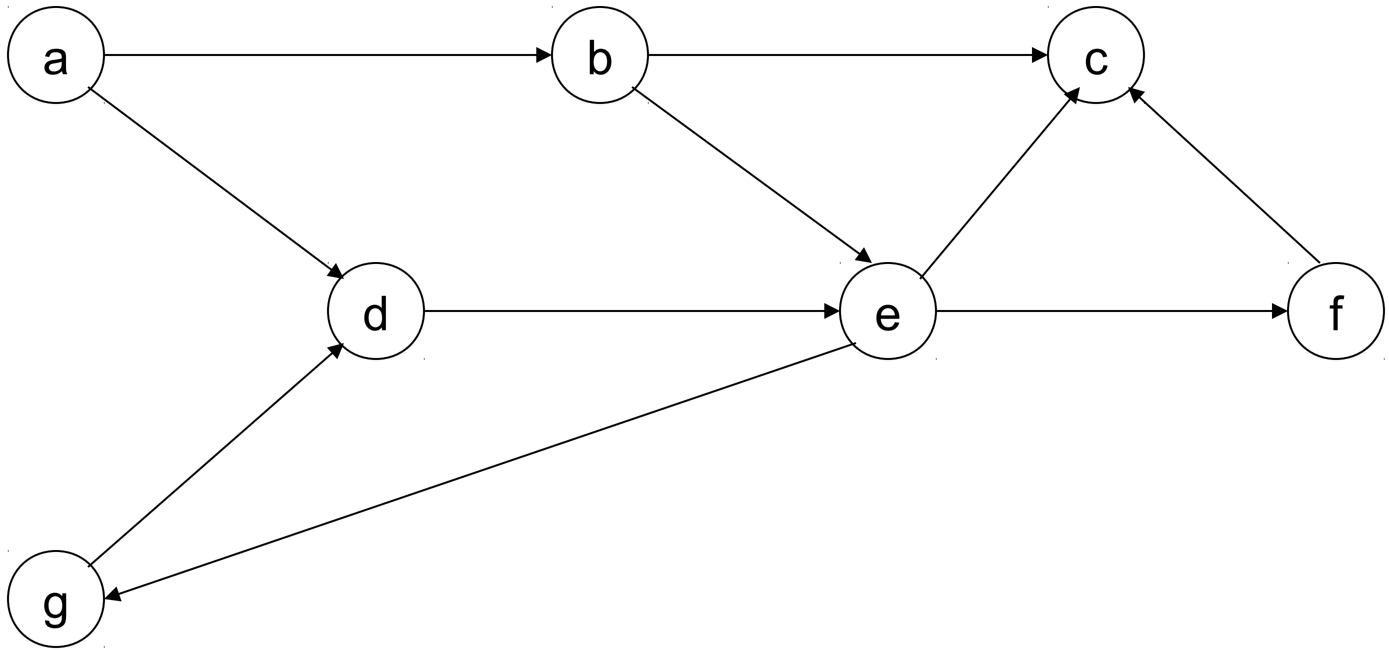
Mark.Dras@mq.edu.au

E6A380

Outline

- Directed graphs
- Graph operations
  - topological sort
  - spanning trees
  - minimal spanning trees

A Directed Graph



Directed Graph Definition

- Edges
  - for a directed graph, $(v_1,v_2) \neq (v_2,v_1)$
  - if there is an edge (v_1,v_2), then v_2 is adjacent to v_1, but not vice versa
    - alternatively, v_2 is a successor of v_1, and v_1 is a predecessor of v_2
- Degree of vertices
  - in-degree: how many predecessors
  - out-degree: how many successors

Application: Garbage Collection

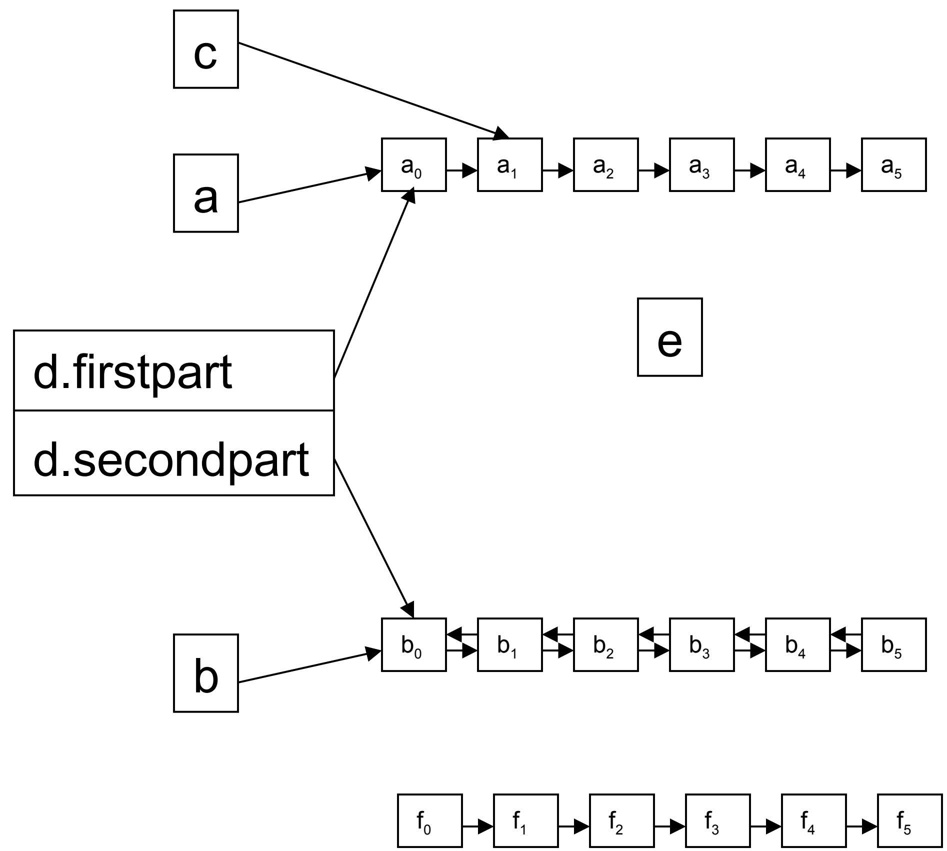
- Traversals can be defined on directed graphs in the same way as on undirected graphs
  - the only difference is in the definition of adjacency lists
- This kind of traversal is used in automatic garbage collection:
  - in C++, garbage collection is manual
    - memory is reclaimed using e.g. delete
  - in Java, garbage collection is automatic

Garbage Collection

- Memory heap contains a lot of objects, some still currently accessible (“live”), some not
  - want to free up memory that’s not accessible
  - how to determine this?
- Running programs store variables and objects in the runtime stack
  - referred to as root objects
  - anything accessible from these is live

Garbage Collection

```
void someFn (node* a, node* b)
{
    node* c;
    if (a)
        c = a->next;
        cout << c->data << endl;
    else
        c = 0;
    nodeStruct* d;
    d->firstpart = a;
    d->secondpart = b;
}
```



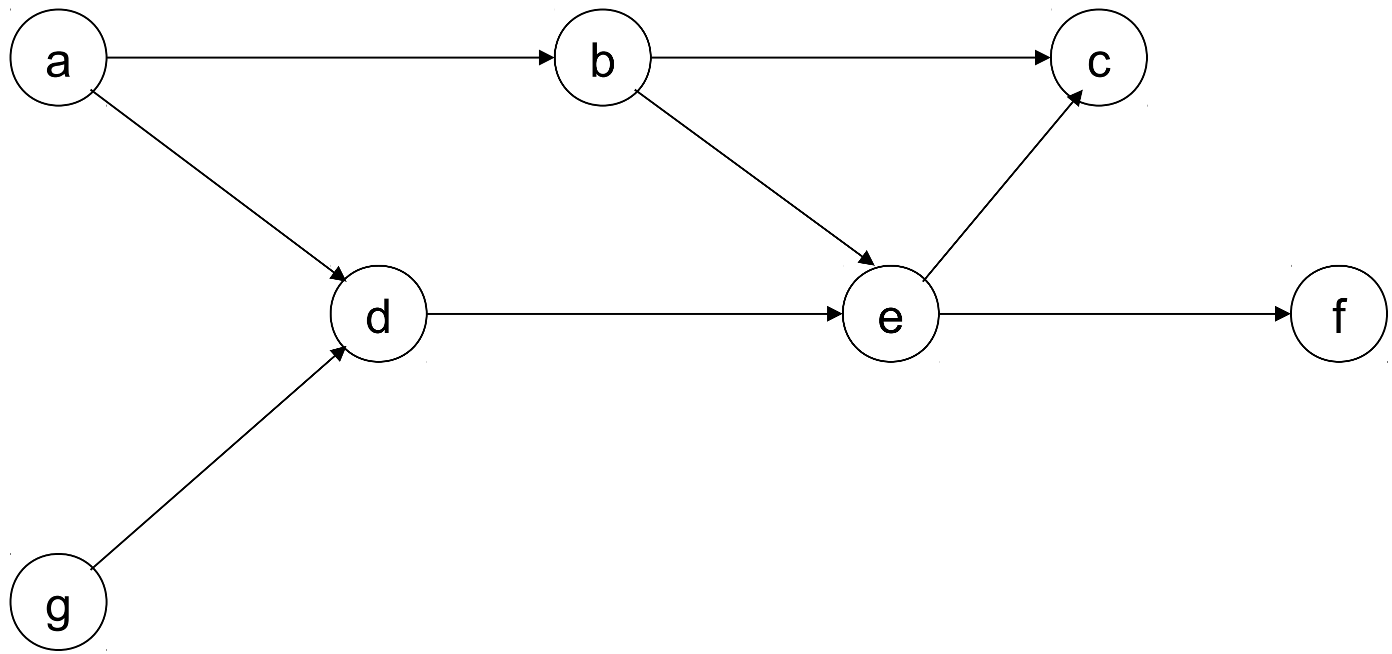
Garbage Collection

- Most common algorithm is called mark-sweep
- Do a graph traversal from root objects (in previous example, *a, b, c, d*)
  - each element is tagged with a mark bit, set when it's visited in a traversal
- Then visit each object in the heap and see if it's been marked
- In practice, there are also space considerations

Outline

- Directed graphs
- Graph operations
  - topological sort
  - spanning trees
  - minimal spanning trees

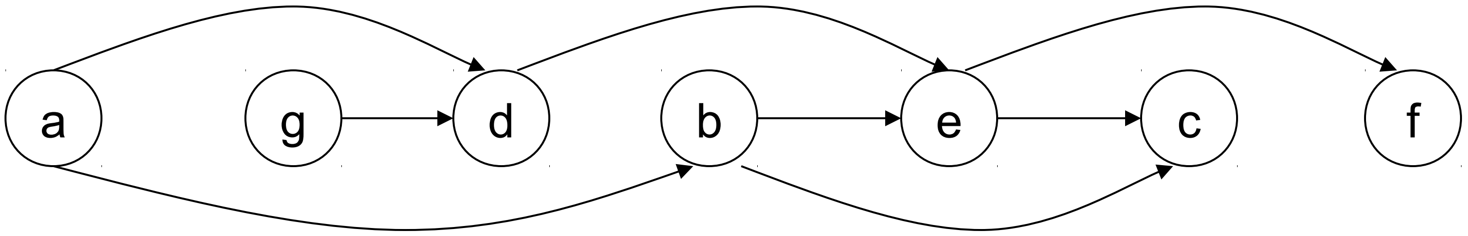
A Directed Acyclic Graph (DAG)

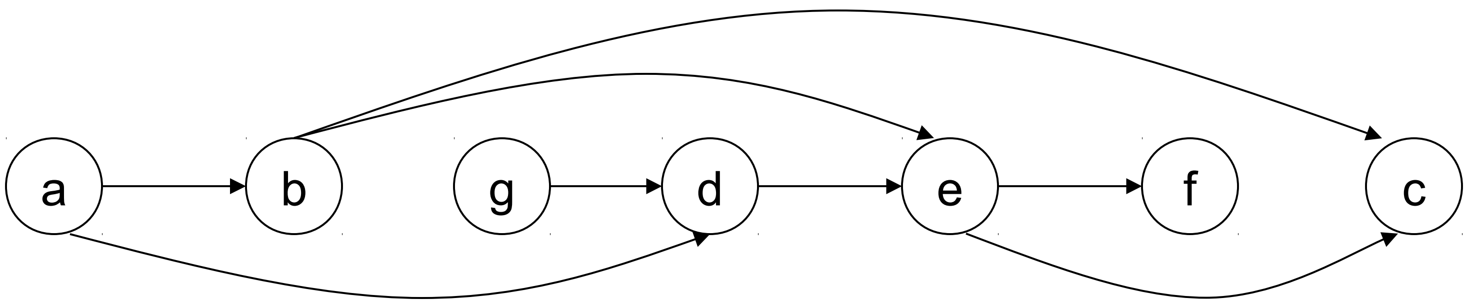


Topological order

- Only defined on DAGs
- The previous graph has at least two possible topological orders
  - *a, g, d, b, e, c, f*
  - *a, b, g, d, e, f, c*
- Note that before a node can be placed in the topological ordering, all of its predecessors must be in the ordering
  - you can see this by 'squashing' the graph
  - then, no back arrows

Linear Order





Orderings

- Q: Are there more than two orderings?

- General principle in ordering
  - nodes with the largest in-degree or out-degree are the most constrained ("bottlenecks")

TopSort Algorithms

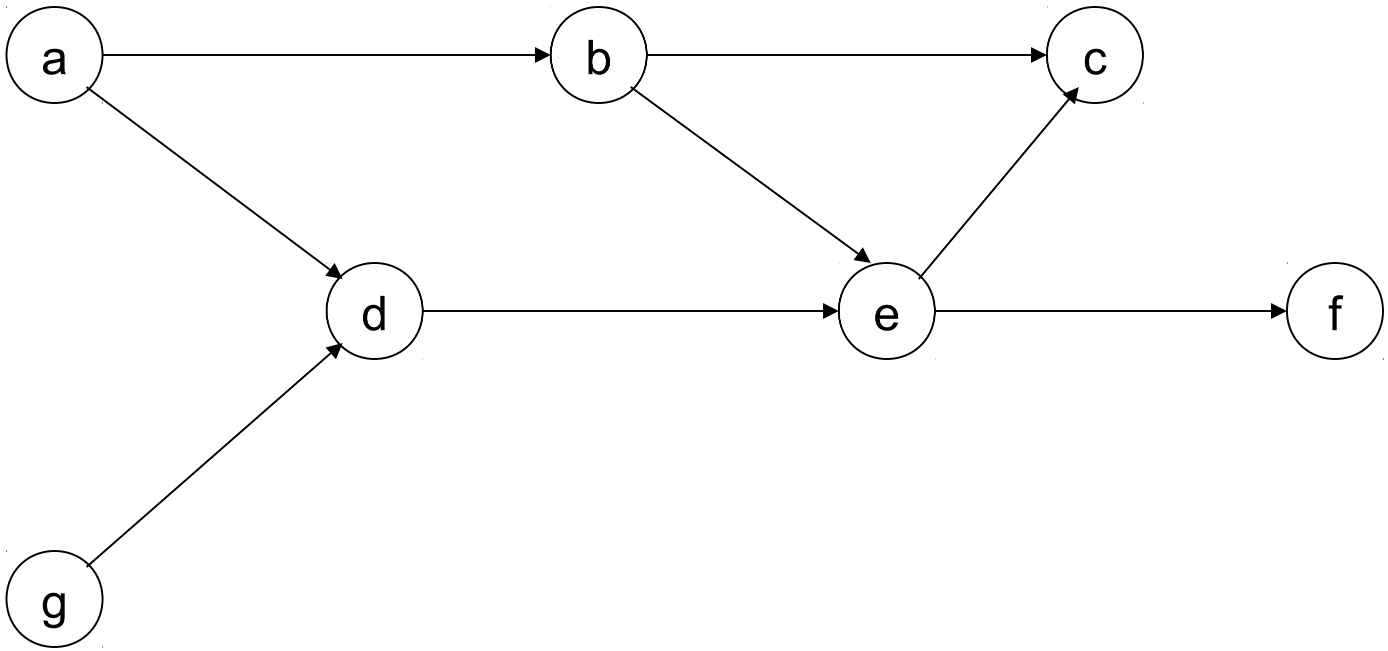
- Can start from the 'end':
  - find nodes with no successor, and successively remove them from the graph
- Can start from the 'start':
  - find nodes with no predecessor
  - start from these and do a DFS

TopSort Algorithm 1

```
topSort1(in theGraph: Graph, out aList:List)
// arranges the vertices in theGraph into a topological order and places
// them in aList

n = number of vertices in theGraph
for (step = 1 through n) {
    select a vertex v that has no successors
    aList.insert(1,v)
    delete from theGraph vertex v and its edges
}
```

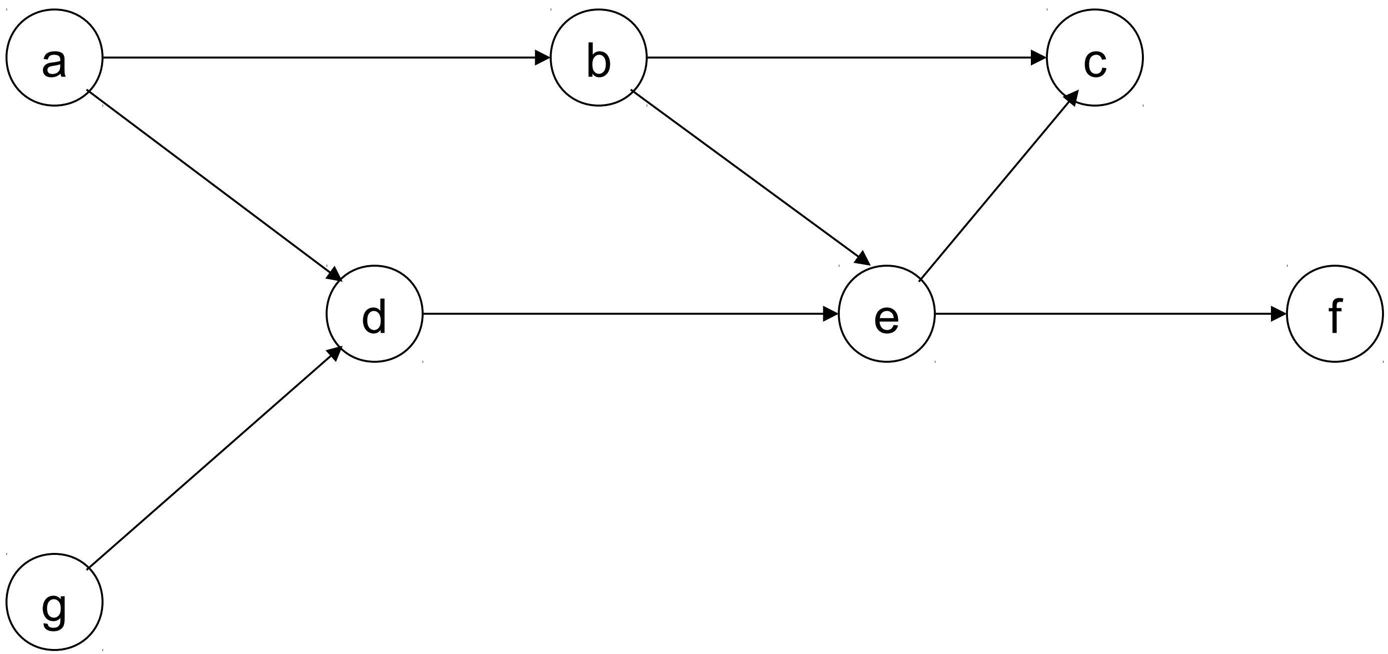
topSort1



TopSort Algorithm 2

```
topSort2(in theGraph: Graph, out aList:List)
// arranges the vertices in theGraph into a topological order and places
// them in aList
s.createStack()
for (all vertices v in the graph)
    if (v has no predecessors) {
            s.push(v)
            mark v as visited
    }
while (!s.isEmpty()) {
    if (all vertices adjacent to the vertex on the top of the stack have been visited) {
            s.pop(v)
            aList.insert(1,v)
    }
    else {
            select an unvisited vertex u adjacent to the vertex on top of the stack
            s.push(u)
            mark u as visited
    }
}
```

topSort2



TopSort: Exercise

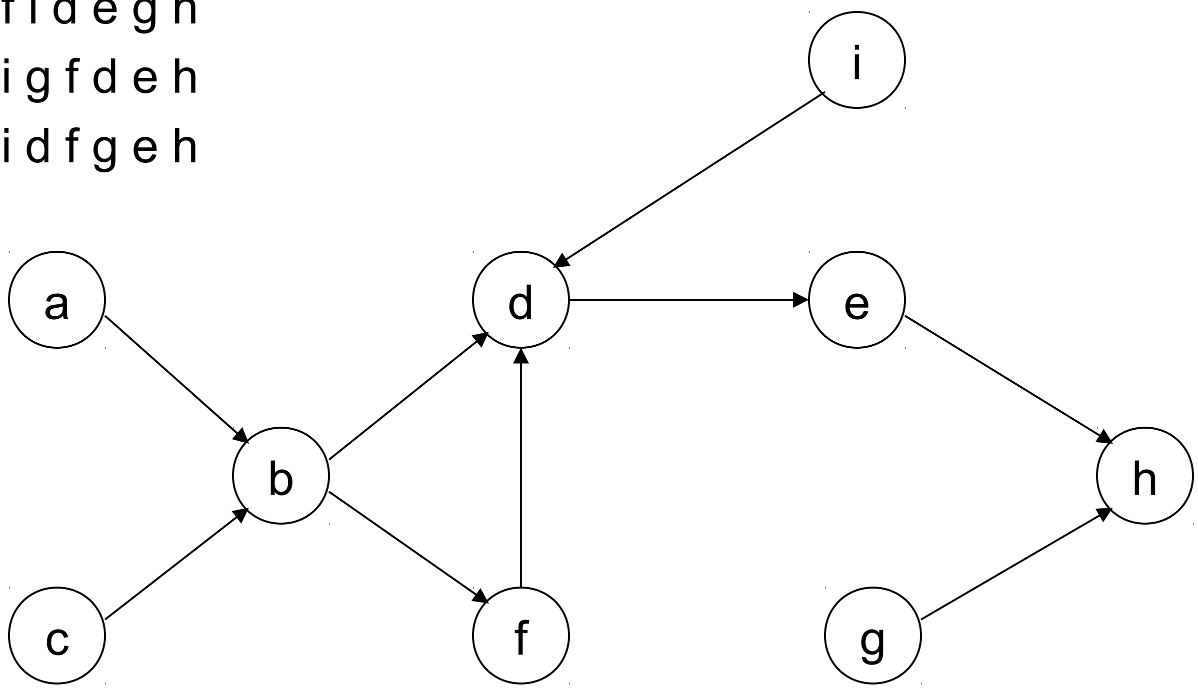
- Are these valid topological orderings?

  a b c d e f g h i

  a c b f i d e g h

  c a b i g f d e h

  a c b i d f g e h



Outline

- Directed graphs
- Graph operations
  - topological sort
  - spanning trees
  - minimal spanning trees

Spanning Trees

- Recall that a tree is a kind of graph
  - it has no cycles
  - so, if you successively remove edges from a connected graph such that the remaining graph is connected, at some point you will have a tree
- **Spanning tree** (of a connected undirected graph *G*): a subgraph of *G* that contains all *G*'s vertices and enough edges to form a tree

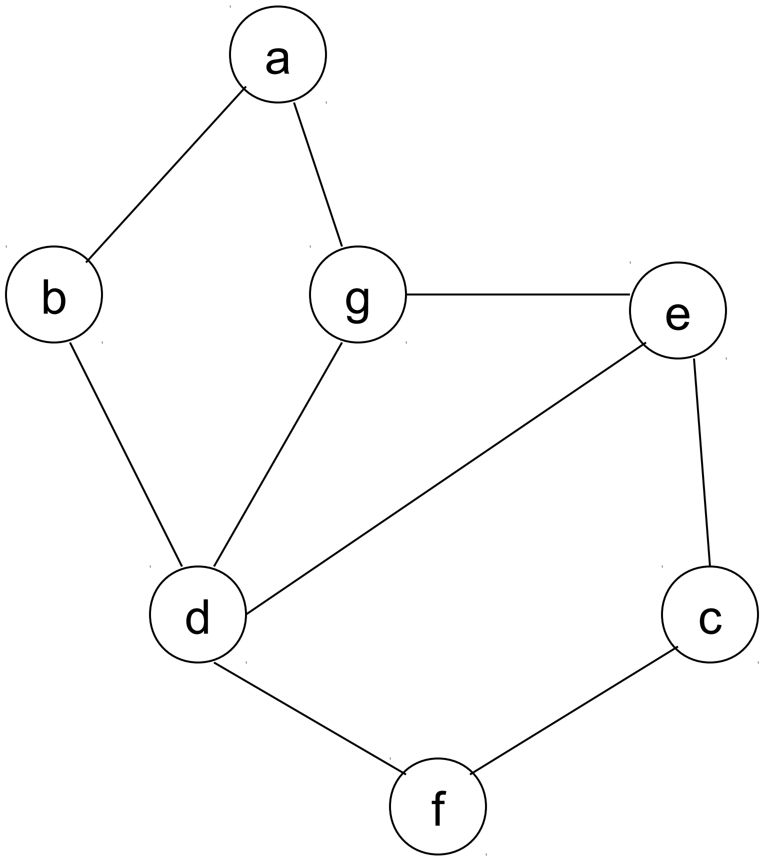
Graphs and Cycles

- A connected undirected graph that has n vertices must have at least $n-1$ edges
- A connected undirected graph that has n vertices and exactly $n-1$ edges cannot contain a cycle

Graphs and Cycles

- A connected undirected graph that has *n* vertices and more than *n-1* edges must contain at least one cycle
  - aside: what if the graph's not connected?
- Therefore, to get a spanning tree, remove edges until only *n-1* edges remain, at each step maintaining a connected graph
- Can use depth-first or breadth-first traversal

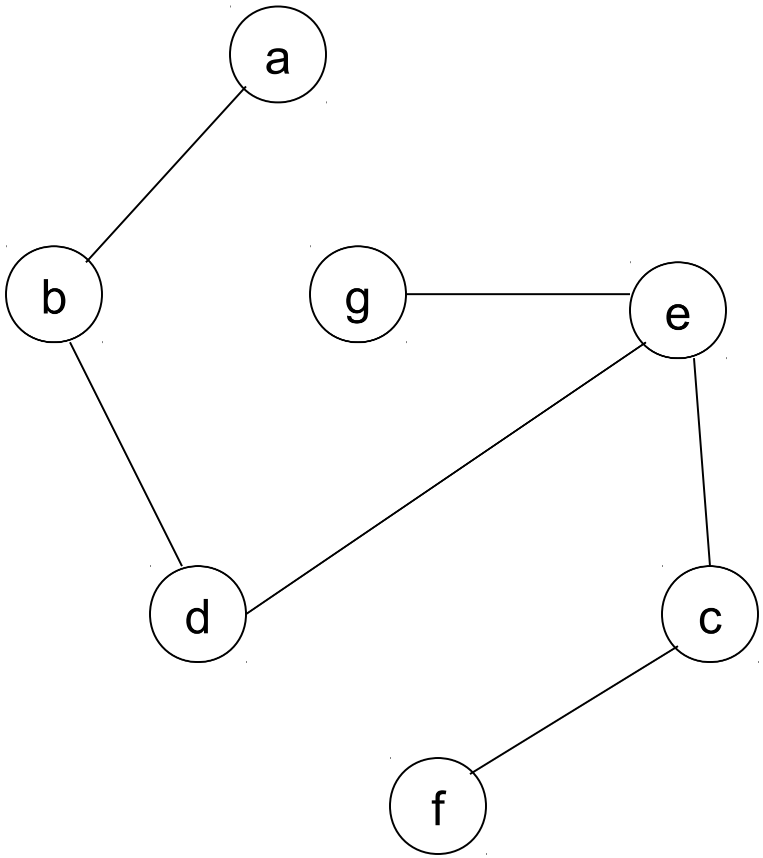
Example: Depth-First



depth-first traversal (starting from a):

a-b-d-e-c-f-g

Example: Depth-First



DF spanning tree starting from a

DF Algorithm

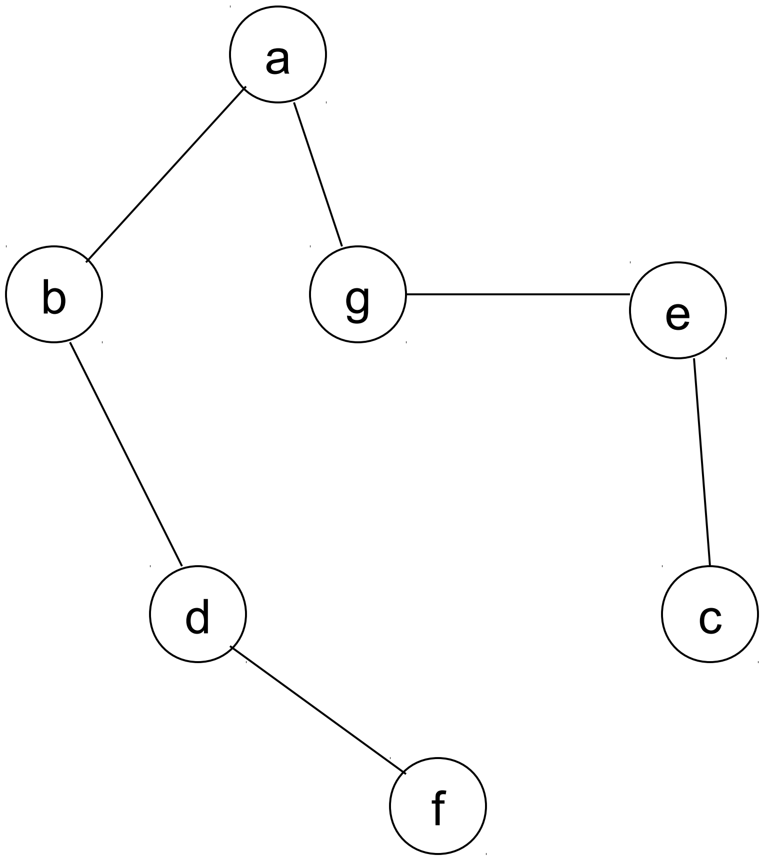
```
depthFirstSearch()
   for all vertices v
      num(v) = 0;
   edges = null;
   i = 1;
   while there is a vertex v such that num(v) == 0
      DFS(v);
   output edges;

dfs (v)
   num(v) = i++;   // mark v as visited
   for (all vertices u adjacent to v)
      if num(u) == 0    // u is unvisited
         attach edge(uv) to edges   // mark (u,v) as a discovery edge
         dfs(u)
```

DF Algorithm

- How do you mark edges?
  - typically, there's an extra data element in the objects that make up the adjacency list or the adjacency matrix
  - also possible to have a separate data member as part of the graph
- Q: Is the spanning tree unique?

Example: Breadth-First



BF spanning tree starting from a:

BF Algorithm

```
bfs ()
   for all vertices v
      num(v) = 0;
   edges = null;
   i = 1;
   q.createQueue();
   while there is a vertex v such that num(v) == 0
      num(v) = i++;      // mark v as visited
      q.push(v);
      while (!q.isEmpty()) {
         v = q.pop()
         // loop invariant: there is a path from former front of queue to every vertex in queue
         for (all unvisited vertices u adjacent to v) {
              num(u) = i++;    // mark u as visited
              attach edge(vu) to edges    // mark (v,u) as discovery edge
              q.push(u)
         }
      }
   }
   output edges
}
```

Outline

- Directed graphs
- Graph operations
  - topological sort
  - spanning trees
  - minimal spanning trees

Minimum Spanning Trees

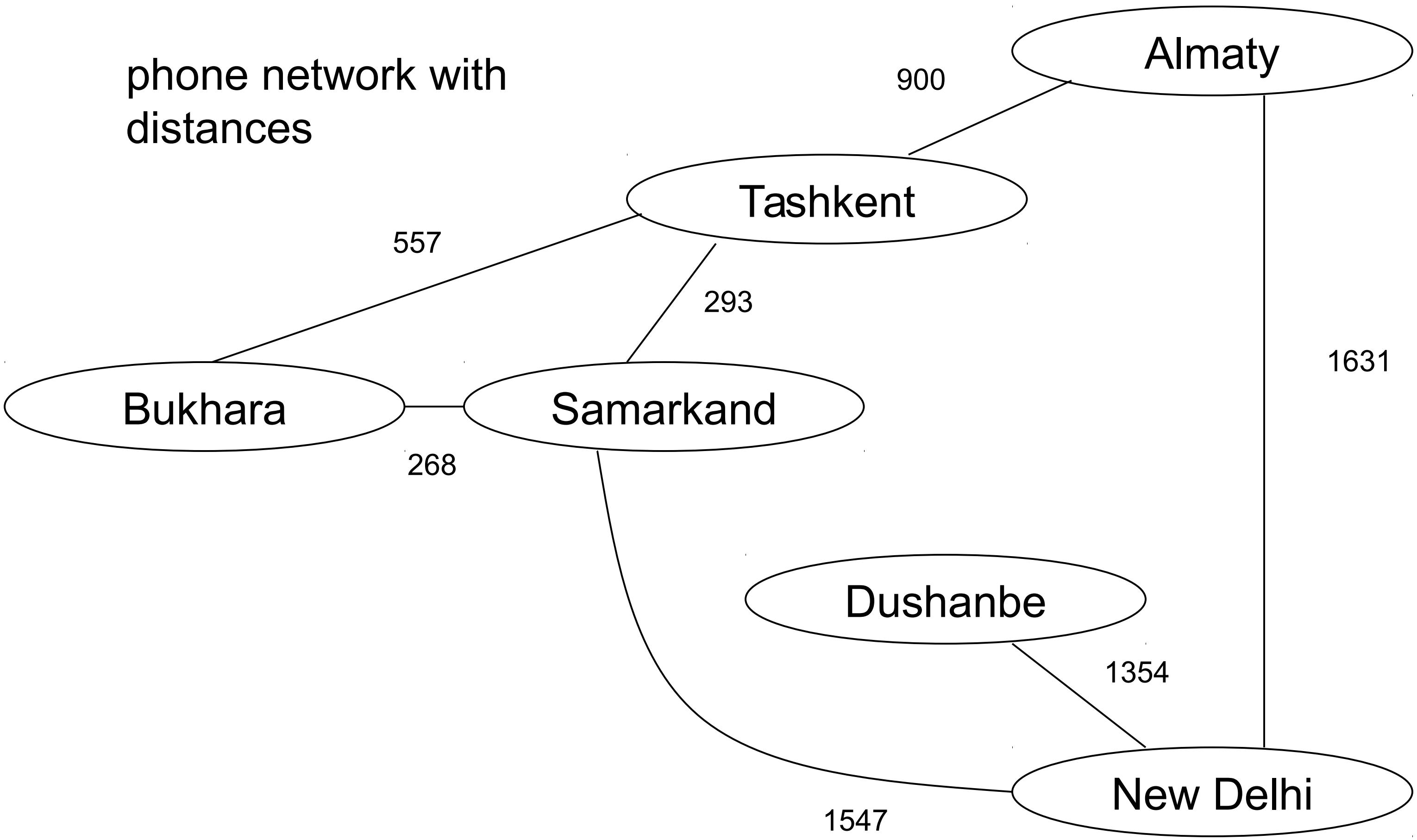
- In a weighted graph, costs are associated with edges
  - an obvious problem to solve then is to find the spanning tree with the least cost
  - cost of spanning tree is defined as the sum of costs of its component edges

Prim's Algorithm

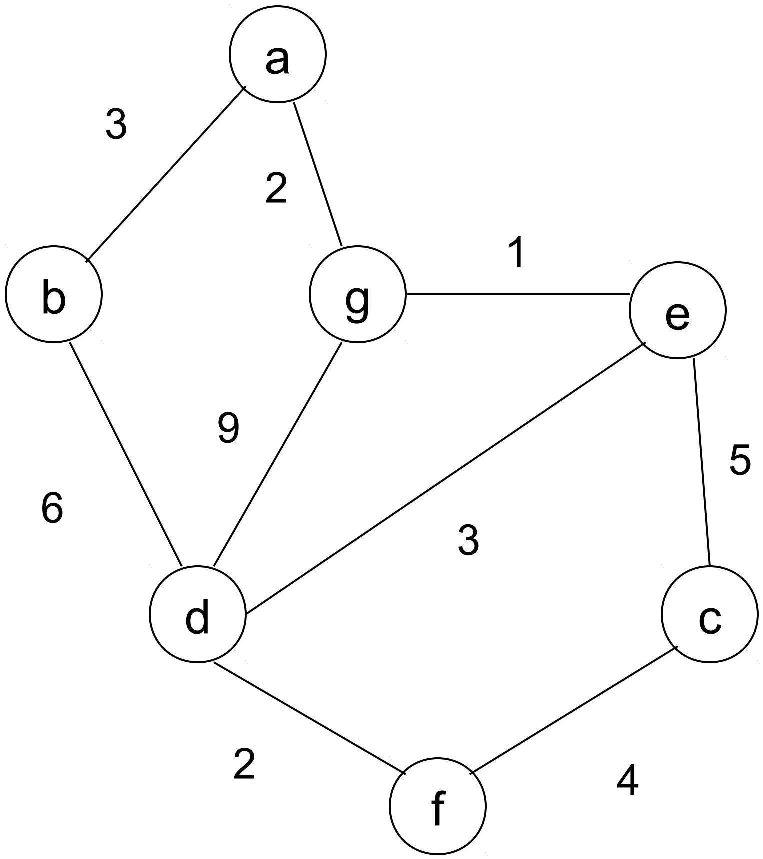
- To construct a minimum spanning tree
  - begin with a tree containing only one vertex
  - at each stage, add the least cost edge that begins from some vertex in the tree and ends with some vertex not in the tree

Example

phone network with distances



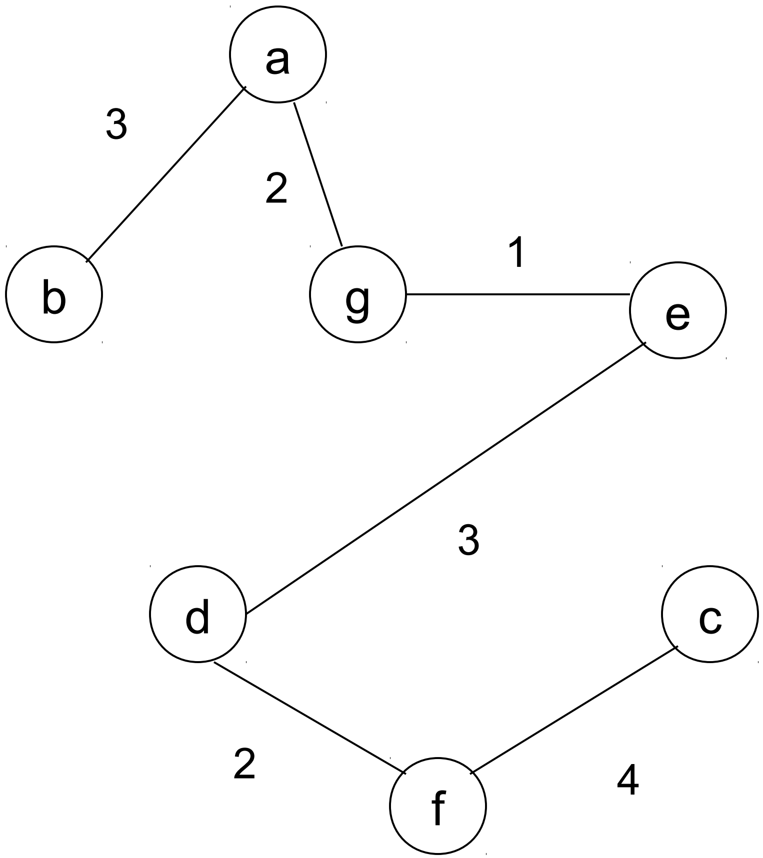
Example: Prim's Algorithm



node order visit (starting from a):

a-g-e-d-f-b-c

Example: Prim's Algorithm



node order visit (starting from a):

a-g-e-d-f-b-c

Prim's Algorithm

```
primsAlgorithm (in v:Vertex)
// determines a minimum spanning tree for a weighted, connected, undirected
// graph whose weights are non-negative, beginning with vertex v

mark v as visited and include it in the minimum spanning tree
while (there are unvisited vertices) {
    find the least-cost edge (v,u) from any visited vertex v to unvisited vertex u
    mark u as visited
    add the vertex u and the edge (v,u) to the minimum spanning tree
}
```

Prim's Algorithm

- Note that it's quite different from a traversal
  - in a traversal, at any point you only have one node as the base to move to the next step (the top of the stack for DF, the front of the queue for BF)
  - in Prim's, you can move from **any** node in the spanning tree to any other adjacent node

Prim's Algorithm

- Comparison
  - traversal (recall from previous lecture)
    - assume adjacency list representation, with list ordered by node ordering
    - at any point in the traversal, there's only one choice, and the next node is the first element in the adjacency list
    - there are $|V|$ such node visits (plus checking of edges): $O(V + E)$

Prim's Algorithm

- Comparison
  - Prim's
    - assume the adjacency list is ordered by weight
    - at any point in the search, the next choice can come from any of the nodes in spanning tree
      - on average there are $|V|/2$
    - then choose first element of adjacency list
    - there are *n* nodes visited in the search
    - therefore $O(V^2)$

Optimality of Prim's Algorithm

- Greedy algorithm
  - often aren't optimal, but Prim's Algorithm is
- Proof by contradiction
  - assume some graph G where PA does not return an MST
  - then there's some point at which we inserted edge (x,y) that took us away from the MST
  - but there's some path from x to y in the MST, using (u,v) shorter than (x,y)
  - PA would have then chosen (u,v) not (x,y)

An Improvement

- Keep track of the lowest cost edge at each node
- This can be in the form of a heap-based priority queue
- Following is a variant of Prim's

PQ-Prim

```
PQPrim (in v:Vertex)
// determines a minimum spanning tree for a weighted, connected, undirected
//     graph whose weights are non-negative, beginning with vertex v
// uses a global priority queue

for each vertex u in G {
    set u.key = ∞
    set u.parent = nil
}
v.key = 0
initialise a min priority queue Q with all vertices from G
while Q is not empty {
    u = EXTRACT-MIN(Q)
    for each vertex x adjacent to u {
        if x in is Q and the weight of (u,x) < x.key {
            x.parent = u
            x.key = weight of (u,x)
        }
    }
}
```

PQ-Prim

- Difference from previous algorithm is with global priority queue Q
  - don’t need to go through all nodes already in MST
  - insertion and deletion operations are O(log E)
- Aside: priority queues can also have a DECREASE-KEY operation
  - functionality is to decrease the value of an element in the PQ
  - complexity for PQ of size n is O(log n)
  - not proved here, but consider deletion folllowed by insertion

PR-Prim

- Now, while loop executes |V| times, and each EXTRACT-MIN operation takes O(log V) time
  - therefore O(V log V) for all calls to EXTRACT-MIN
- The for loop executes O(E) times
  - sum of all adjacency lists is 2|E|
  - assignment of key in for loop is an implicit DECREASE-KEY, which is O(log V)
  - total for these calls is O(E log V)
- Therefore complexity is O(V log V + E log V)

Directed Graphs

- Note that Prim's doesn't work for directed graphs
  - cycles cause complications
  - algorithms do exist for this, like Chu-Liu-Edmonds