Elementary Graph Theory

- WE will be looking at :
  - What is a Graph?
  - Adjacency Matrices
  - Adjacency Lists
  - Breadth First Search
  - Depth First Search
  - Minimum Spanning Trees; What are they?
  - Kruskal’s Algorithm for creating a MST
  - Prim’s Algofithm for creating a MST

A Graph

- A graph can be thought of a collection of vertices (V) and edges (E), so we write,

  $$G = (V, E)$$

- Graphs can be **directed**, or **undirected**, **weighted** or **unweighted**.

- A directed graph, or digraph, is a graph where the edge set is an ordered pair.

- That is, edge 1 being connected to edge 2 does not imply that edge 2 is connected to edge 1. (i.e. it has direction – trees are special kinds of directed graphs)

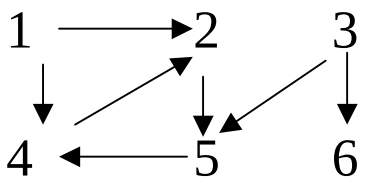
- An undirected graph is a graph where the edge set in an unordered pair.

- That is, edge 1 being connected to edge 2 does imply that edge 2 is connected to edge 1.

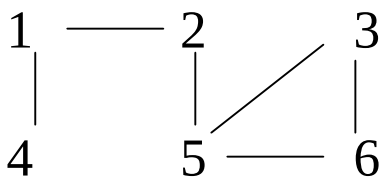
A Weighted Graph

- A weighted graph is graph which has a value associated with each edge. This can be a distance, or cost, or some other numeric value associated with the edge.

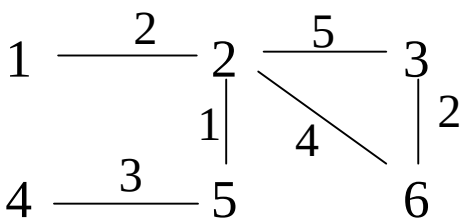
(a) A directed graph



(b) An undirected graph



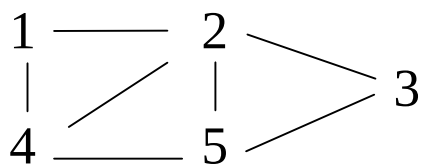
(c) A weighted graph



Representing a Graph

- There are two standard ways to represent a graph G = (V,E) in computer science
  - As a collection of adjacency lists
  - As an adjacency matrix

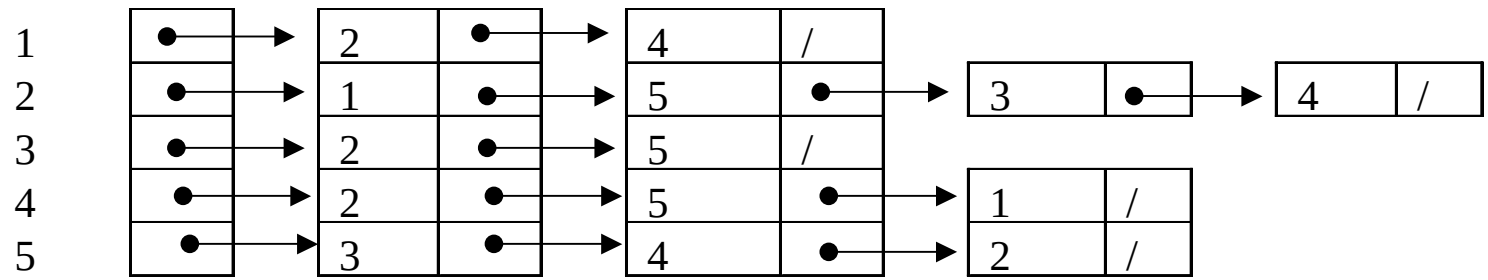
- Take the following graph



- We can represent this in two ways

Adjacency Lists & matrices

- As an adjacency list this is



- As an adjacency matrix this is

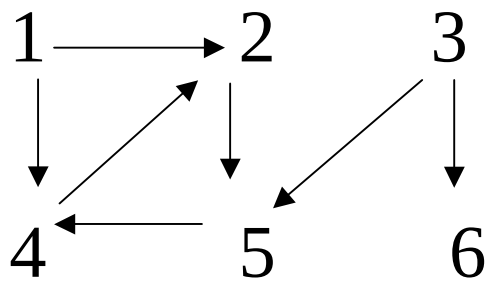
	1	2	3	4	5
1	0	1	0	1	0
2	1	0	1	1	1
3	0	1	0	0	1
4	1	1	0	0	1
5	0	1	1	1	0

Adjacency Matrix

- To construct an adjacency matrix we first number each vertices in our digraph 1 to n
- The adjacency matrix is then a nxn matrix, in which row i and column j is 1 (or true) if vertex j is adjacent to vertex i. Otherwise it is 0 (or false)
- Lets have a look at an example…

Adjacency Matrix : Example

- For a directed graph such as



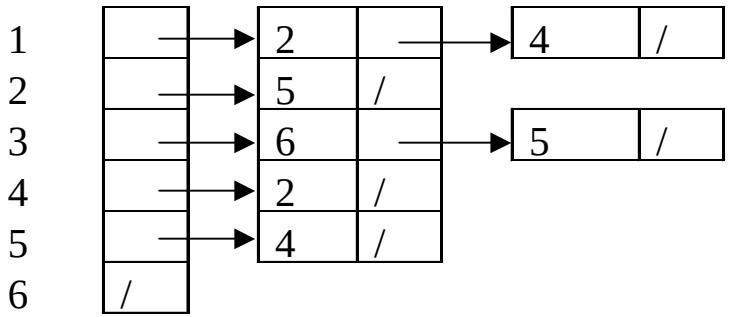
- The adjacency matrix is

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	0

Adjacency List

- An adjacency list is a representation of adjacent vertices in the digraph as an array of pointers to linked row-lists (from the adjacency matrix).

- The adjacency list for the last example is :



- Adjacency matrices are generally quite sparse so there is a lot of wasted space. It is for this reason that the adjacency list representation is usually preferred, as it provides a compact way to represent **sparse** graphs.

- A sparse graph is defined as one for which |E| is much less that |V| squared

- Sparse graphs are very common in real-world situations

Representing Graphs

- An adjacency matrix representation may be preferred when the graph is **dense**, or when we need to be able to tell quickly if there is an edge connecting two given vertices.

- If a graph is a weighted graph, then the weight can be stored with the vertex in the adjacency list.

- In Java we can do this by adding the extra weight information in our class Node (remember this from linked lists?)

- To store the weight in an adjacency matrix, simply store the weight instead of a 1 in the entry for vertex i and vertex j

Searching a Graph

- Breadth-First-Search (BFS) is a search algorithm which forms the basis for many important graph algorithms
- Given a graph G = (*V*, *E*) and a source vertex *s*, breadth-first search systematically explores the edges of G to discover every vertex that is reachable from *s*
- It also produces a “breadth-first-tree” with root *s* that contains all such reachable vertices
- For any vertex *v*, reachable from *s*, the path in the breadth-first tree from *s* to *v* corresponds to a “shortest path” from *s* to *v* in G. That is, a path containing the fewest number of edges
- The algorithm works on both directed and undirected graphs

Breadth First Search

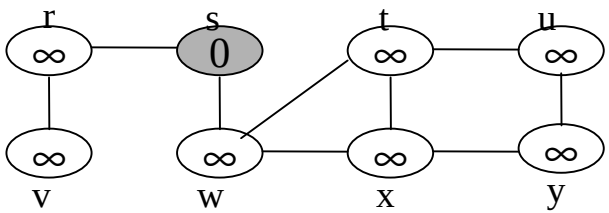
- The algorithm is called Breadth-First-Search because it discovers all vertices at distance *k* from *s* before discovering any vertices at distance *k*+1
- Using the BFS we speak of colouring the nodes as they are discovered.
- A white node has not yet been discovered. A grey node has been discovered but may have some adjacent white vertices. All adjacent vertices of a black node have been discovered
- The algorithm we are going to look at assumes the graph is represented as an adjacency list
- The algorithm also uses a queue to manage the set of grey vertices

BFS Algorithm

- The distance from the source *s* to vertex *i* is stored in the array d[*i*]

```
BFS(G, s)
   for each vertex i ∈ V(G) - {s}          // Initialise the adjacency list for paths from vertex s
         colour[i] = white;  d[i] = ∞;      // Set all colours to white and distance to some default
   endfor
   colour[s] = grey;  d[s] = 0; Q = {s};    // Set vertex s to grey , distance to ourselves is 0, join Queue
   while Q != empty                         // While Queue is not empty
         i = Q.head;                        // Vertex i is vertex at head of Queue
         for each j ∈ Adj[i]                // For each vertex j which is part of adj. list of vertex i
            if colour[j] == white then      // If vertex has not been discovered yet (i.e. colour is white)
               colour[j] = grey;            // Mark as discovered & possibly more undiscovered adj. vertices.
               d[j] = d[i] + 1;             // Mark distance from start vertex to this vertex
               Enqueue(Q, j);               // Add vertex j to the Queue
            endif
         endfor
         Dequeue(Q);  colour[i] = black     // Finished with vertex i so leave Queue, Mark as visited  & no
    endwhile                                // more adj. vertices to vertex i that have not been discovered
```

BFS Example (1)



Q

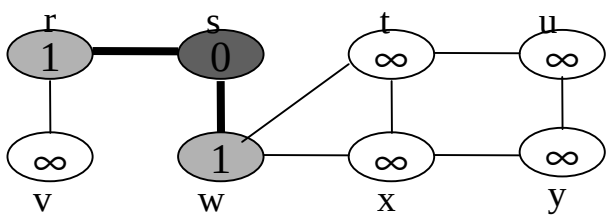
s (0)

1st time in while loop

i = s(0)

s(0) → w (∞) → r (∞)

j



s (0)

Q

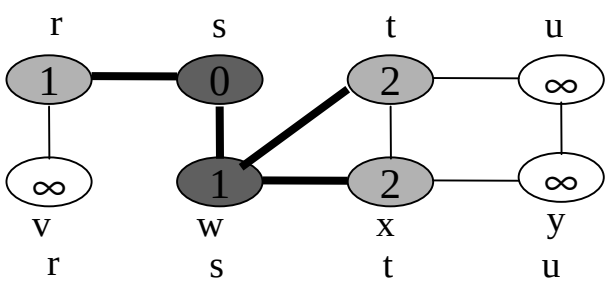
w(1)
r(1)

2nd time in while loop

i = w(1)

w(1) → s(0) → t (∞) → x (∞)

j



w (1)

Q

r(1)
t(2)
x(2)

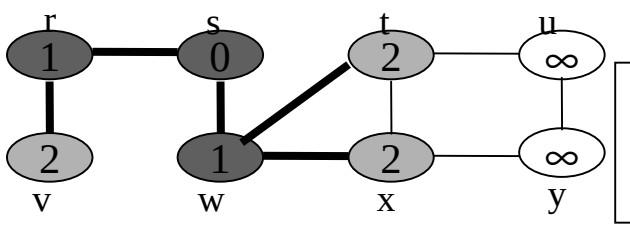
Start of 3rd time in while loop

i = r (1)

r (1) → s(0) → v (∞)

j

BFS Example cont...(2)



Q

t(2)
x(2)
v(2)

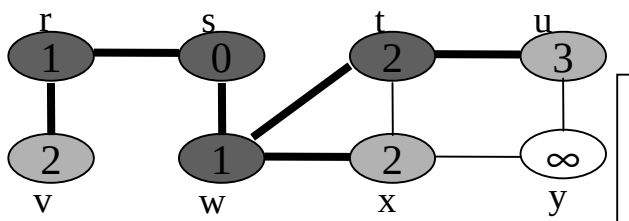
r (1)

4th time in while loop

i = t(2)

t(2) → w(1) → x (2) → u (∞)

j



Q

x(2)
v(2)
u(3)

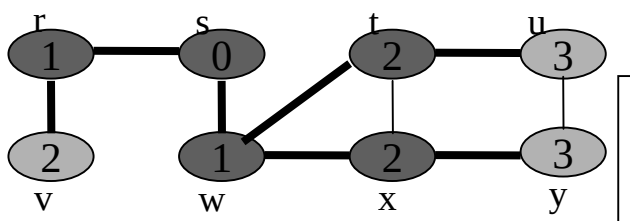
t (2)

5th time in while loop

i = x(2)

x(2) → w(1) → t (2) → y (∞)

j



Q

v(2)
u(3)
y(3)

x (2)

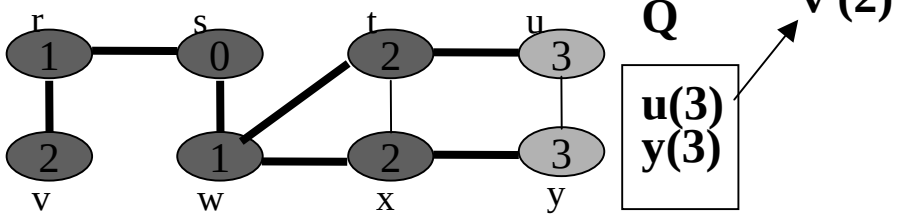
6th time in while loop

i = v(2)

v(2) → r(1)

j

BFS Example cont... (3)

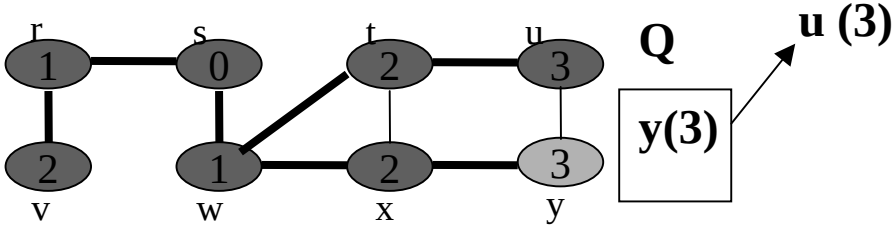


7th time in while loop

i = u(3)

u(3) → t(2) → y(3)

j

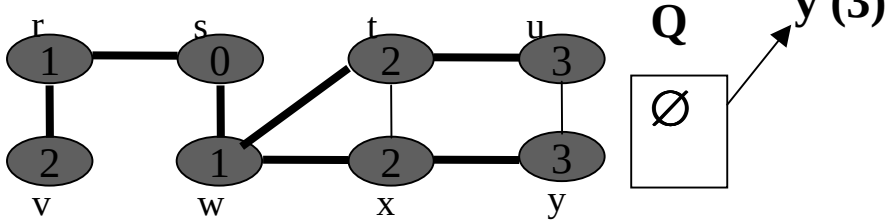


8th time in while loop

i = y(3)

y(3) → x(2) → u(3)

j



At End of 8th time in while loop

y(3) leaves the Queue – the queue is now empty and hence we exit the while loop and algorithm ends. All vertices are searched Breadth first.

Analysis of BFS on G = (V,E)

- After initialisation, no vertex is ever whitened, so each vertex is enqueued only once, and hence dequeued only once.
- The operations of enqueueing and dequeueing take O(1) time, so the total time devoted to queue operations is O(j)
- The adjacency list of each vertex is scanned only when the vertex is dequeued, the adjacency list of each vertex is scanned at most once.
- Since the sum of the lengths of all the adjacency lists is E, at most O(E) time is spent in scanning the adjacency lists.
- The total running time of BFS is O(j + E)
- Breadth First Search runs in time linear in the size of the adjacency list G

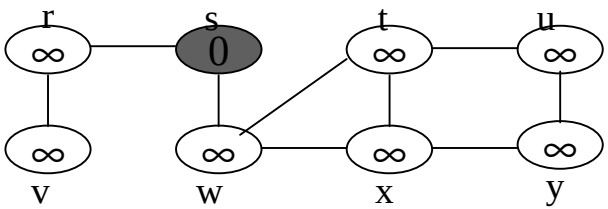
Depth First Search

- DFS is the opposite of BFS. Instead or search breadth first, as the name suggests we are now searching by going down the depth of the graph first.

- DFS also runs in O(j+E) time. Instead of using a queue, however, the DFS can use a stack or can be implemented recursively. Again d[j] represents distance from vertex j to start vertex s.

```
DFS(G, s)
    for each vertex i ∈ V[G] - {s}          // initialise all vertices on path of vertex s
        colour(i) = white,  d[i] = ∞        // set colours to white and distance to some default
    endfor
    Stack.push(s),  d[s] = 0                // push start vertex s onto stack and set distance to 0
    while Stack != empty                    // While stack is not empty
        i = Stack.Pop(); colour(i) = black  // Vertex j is whatever vertex at top of stack is, Mark as black
        for each j ∈ adj[i]                 // for all vertex i that is part of adj. list with vertex j at start
          if colour(j) == white then        //  if not yet discovered
            colour(j) = grey;               // Mark as discovered but possibly more undiscovered
            d[j] = d[i] + 1; Stack.push(j)  // vertices, Mark the distance of vertex i and push onto stack
          endif
        endfor
    endwhile
```

DFS Example (1)



Stack

s (0)

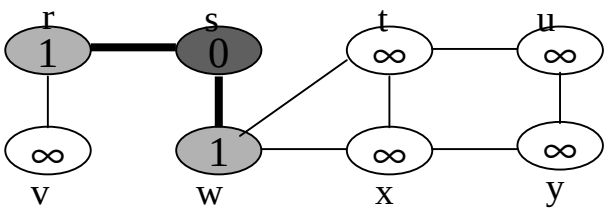
pop

1st time in while loop

i = s(0)

s(0) → r (∞) → w (∞)

j



Stack

r(1)
w(1)

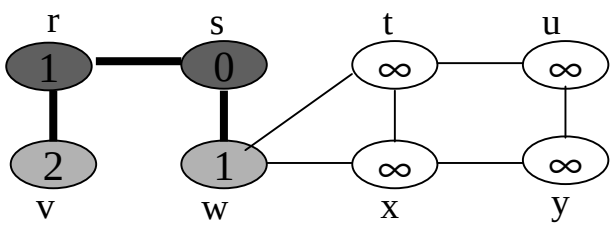
pop

2nd time in while loop

i = r(1)

r(1) → v (∞) → s (0)

j



Stack

v(2)
w(1)

pop

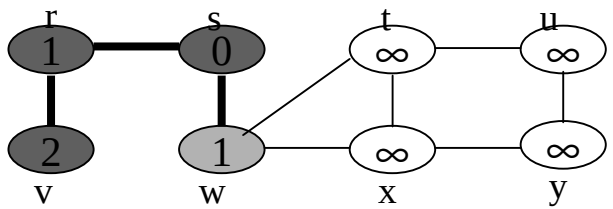
3rd time in while loop

i = v(2)

v(2) → r (1)

j

DFS Example cont…(2)



Stack

w(1)

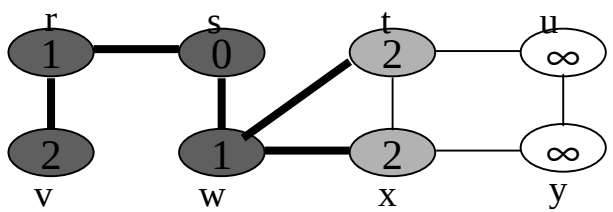
pop

4th time in while loop

i = w(1)

w(1) → s (0) → t (∞) → x (∞)

j



Stack

t(2)
x(2)

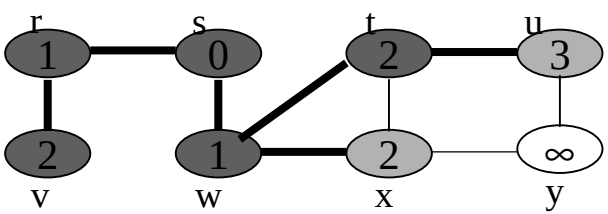
pop

5th time in while loop

i = t(2)

t(2) → w(1) → x (2) → u (∞)

j



Stack

u(3)
x(2)

pop

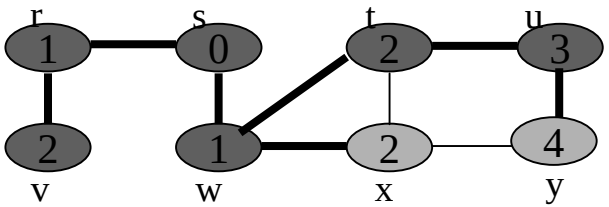
6th time in while loop

i = u(3)

u(3) → t(2) → y (∞)

j

DFS Example cont… (3)



Stack

y(4)
x(3)

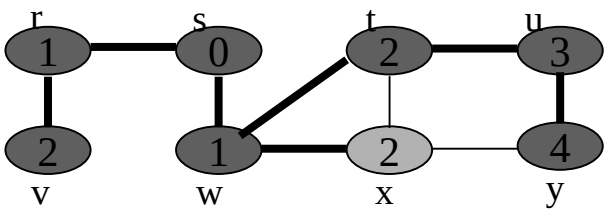
pop

7th time in while loop

i = y(4)

y(4) → u(3) → x (2)

j



Stack

x(3)

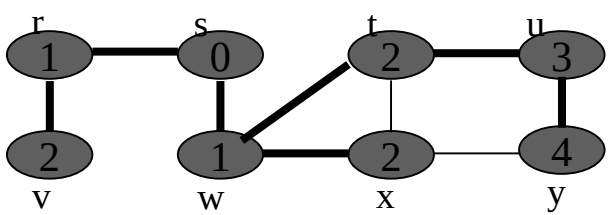
pop

8th time in while loop

i = x(3)

x(3) → w(1) → t (2) → y (4)

j



Stack

∅

At End of 8th time in while loop

- The last vertex gets popped off the stack. The stack is now empty and the algorithm ends. All vertices are searched from the Depth first.

Summing up BFS & DFS

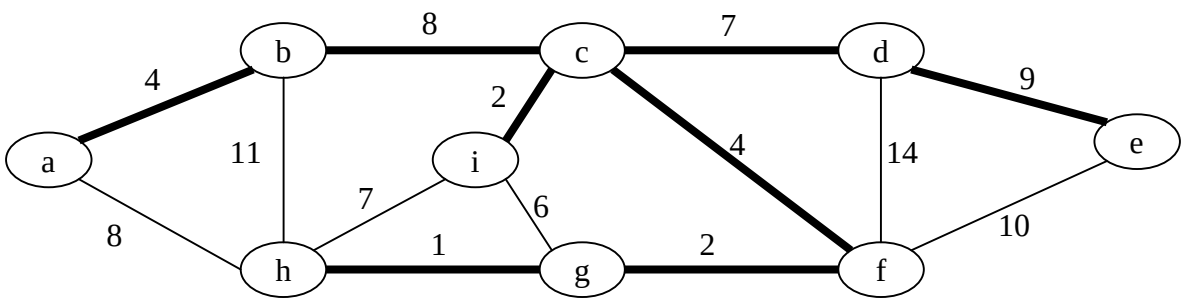
- We can look at BFS is like an army of searchers fanning out.
- We can look at DFS is like a single searcher probing as deeply as possible, retreating only when hitting dead ends.
- Many useful algorithms utilise BFS and DFS
  - Example: Many network / routing problems (e.g. shortest path , cheapest path problems etc.)
- In general we use BFS algorithms for finding shortest path problems as DFS doesn't really do this for us. However, DFS is used elsewhere where BFS is not appropriate to the problem (i.e. possible longest path problems).

Minimum Spanning Trees - Introduction

- In electronic circuitry, we often want to make the pins of several components electrically equivalent by connecting them.
- To interconnect a set of *n* pins, we can use an arrangement of *n*-1 wires, each connecting two pins.
- Of all such arrangements, the one that uses the least amount of wire is usually the most desirable
- This situation can be modeled as a connected, undirected graph G = (V, E), where V is the set of pins, E is the set of possible connections between pairs of pins.
- We want to find the tree which connects all vertices and minimises the sum of the weights on its edges

Minimum Spanning Tree

- Here we will look at two algorithms to find the minimum spanning tree of an undirected weighted graph.
- These algorithms use similar approaches, but work in very different ways
- The following is an example of a graph, with the bold lines outlining the Minimum Spanning Tree



Algorithms

- The two algorithms we will look at use a greedy strategy
- The algorithms differ in how they apply this approach
- The basic idea is that we start with an empty tree and add what we call “safe edges”.
- We continue this until all nodes are included in the tree
- By a “safe” edge we mean one that does not form a cycle and one which will be an edge in the Minimum Spanning Tree
- Extra Info :
  - If a vertex of a graph has a path back to itself it is said to have a “Hamiltonian Cycle”.
  - If a path exists between two vertices of a graph such that we visit every vertex of the graph on the way the once only, then we say that a “Hamiltonian Path” exists.
  - Hamiltonian Path problems exist in may common day problems (ie Chip board analysis, Network routing, DNA strand computation etc. – which by the way all boil down to the Traveling Salesman problem – an NP (NP-complete problem)).

Kruskal's Algorithm

- In Kruskal's algorithm the edges are sorted in increasing order and the edges with least weight are added one at a time

- As each edge is added the algorithm checks if adding it will create a cycle in tree. If the edge will not cause a cycle then add it to the set of safe edges

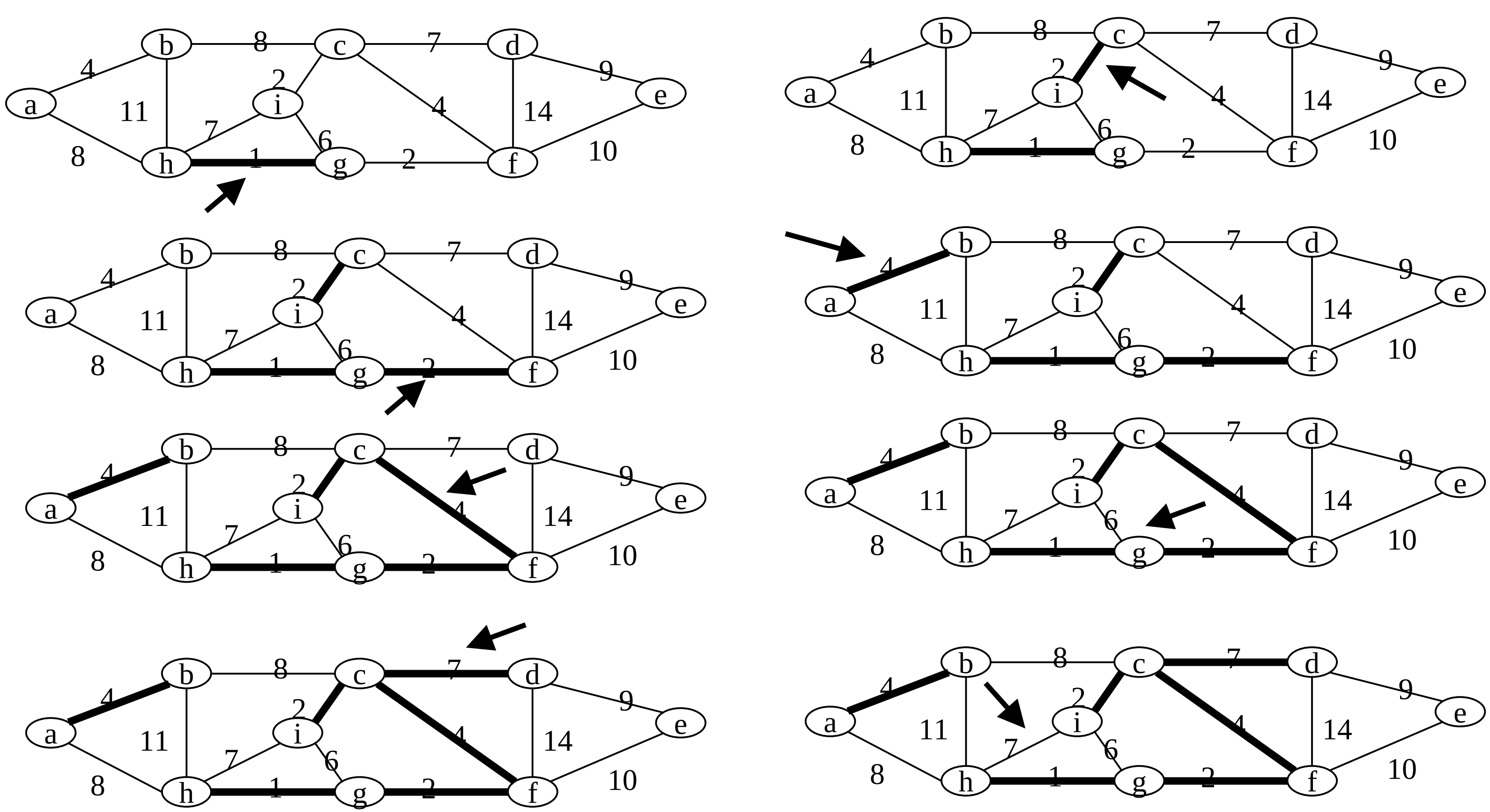
Kruskal's Algorithm

- In this algorithm, let A be the list of edges which are basically a set of non-cylic vertices that make up that edge

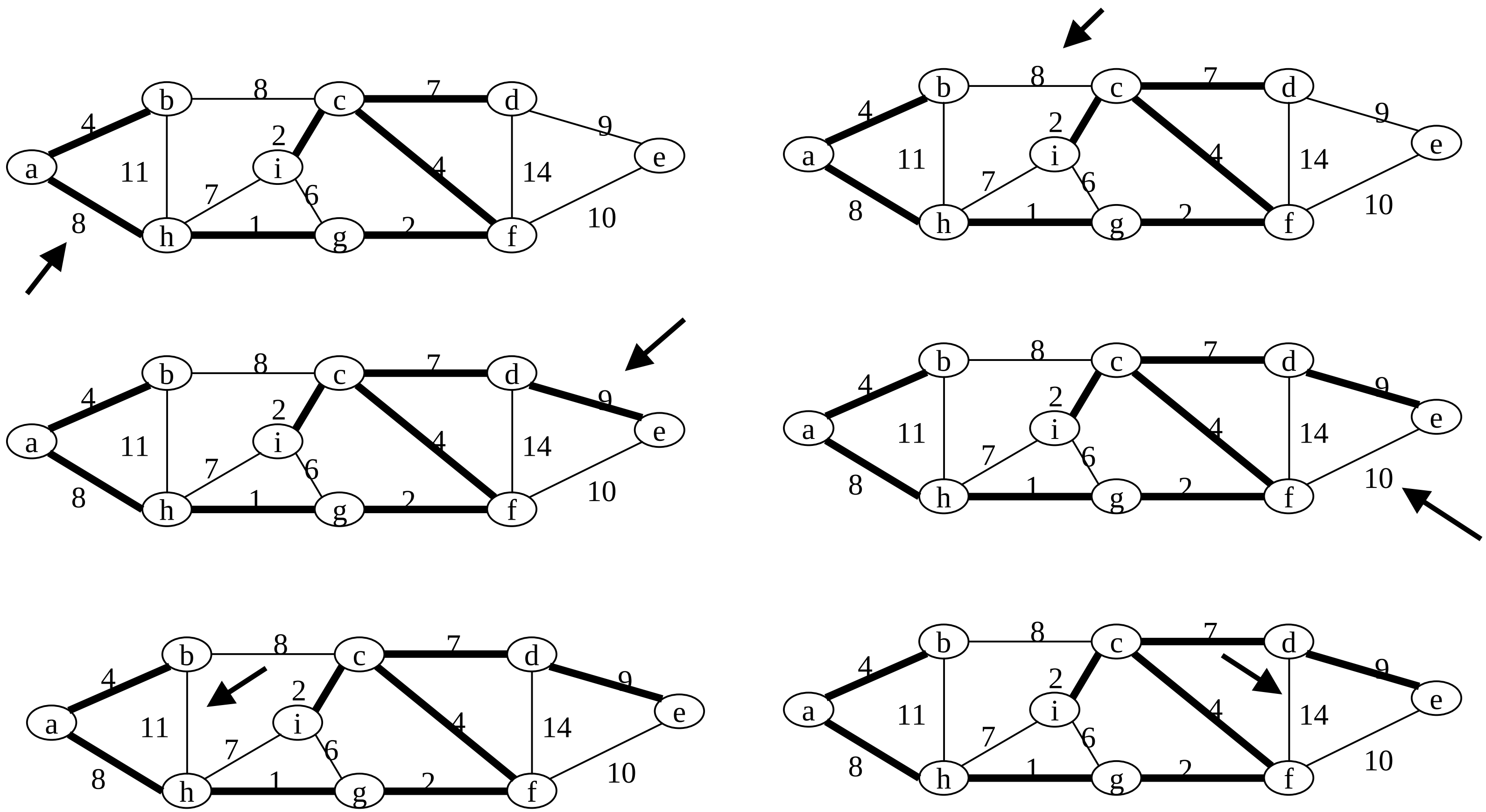
```
MST-Kruskal(G, w)
    A = {0}                          // Initialise edge list
    for each vertex v ∈ V[G]         // Make vertex sets out of all vertices
      Make-Set(v)                    // in the graph
    endfor

    sort the edges of E by nondecreasing weight w

    for each edge (u, v) ∈ E, by nondecreasing weight
      if Find-Set(u) != Find-Set(v) then     // if sets not the same (i.e. not a cycle)
                A = A ∪ {(u, v)}             // Put edge into edge list
                Union(u, v)                  // Join both sets up into the 1 set
      endif
    endfor
    return A                                 // return list of edges
endalg
```

Kruskal Example

- Start with edge of least weight, If not a cycle add it



Kruskal Example cont.



Kruskal Discussion

- This is an abstract view of the operations involved. How we implement the Make-Set, Find-Set and Union operations are issues to be addressed.
- We need to develop data structures to model this. The situation is that we want to have a way of representing a disjoint set.
- Each set can be represented by a linked list.
- However, here we are interested in the overall approach rather than the implementation details.
- An efficient implementation of this algorithm runs in O(E lg E) time.

Problems with Kruskal

- Kruskal's algorithm grows a series of trees which are disjoint
- Growing a set of disjoint trees may be a problem in some applications
- Kruskal also continues checking edges even when all nodes are inserted
- Checking the edges can be inefficient in that a graph can have up V^2 edges
- Many of these edges will be creating cycles
- Can we have an algorithm that is based on looking at nodes rather than edges?

Prim's Algorithm

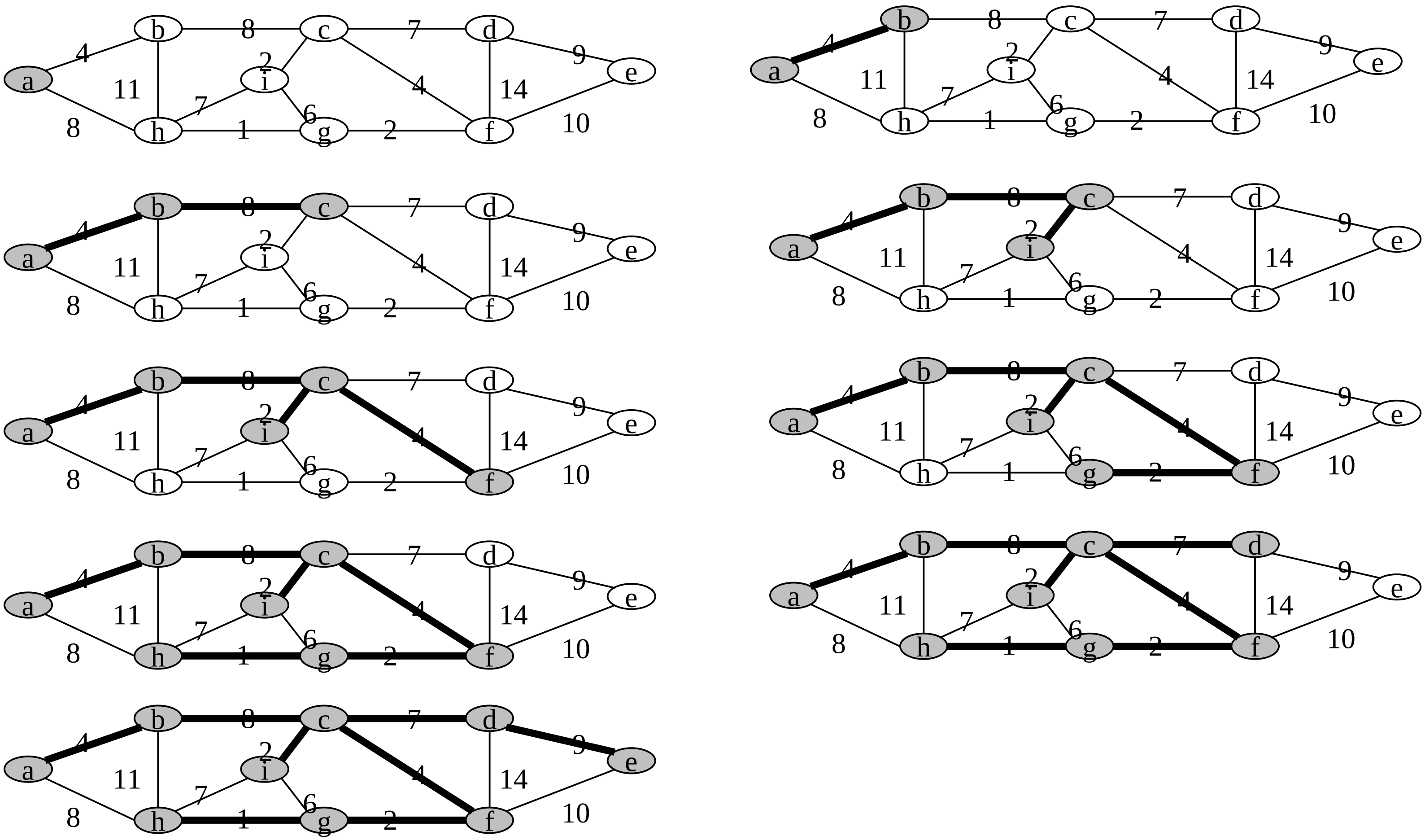
- Prim's algorithm has the property that the edges in the set A always form a single tree.
- Prim's algorithm finishes when all vertices are included in the Minimum Spanning Tree.
- Both of these properties address the issues raised as disadvantages with Kruskal
- Again the strategy is a greedy one.
- The algorithm starts with an arbitrary root vertex r and grows the tree until it spans all the vertices in V.
  - The initial step is to put all vertices onto a priority queue based on a key that is the minimum weight of any edge connecting v to a vertex in the tree.
  - The source vertex has 0 as its key, so it is chosen first and its adjacency list explored for the lightest edge which is added to the tree.
  - At all times the lightest edge between the included vertices and the vertices not included in the tree is added.
  - This continues until all vertices are removed from the priority queue.

Prim's Algorithm

- At the end of the following algorithm, "*parent*" holds the parent – child relationship between each vertex in the MST. (i..e it effectively can be used to describe the tree for us)

```
MST-Prim(G, w, r)
    Q = V[G]                                  // Put vertices in graph into priority queue
    for each u ∈ Q                            // For each vertex u in the priority queue
        key[u] = ∞                            // Init minimum weight key for vertex u to infinity
    endfor

    key[r] = 0                                // set source vertex minimum weight key to zero
    parent[r] = nil                           // source vertex has no parent – it is the root node

    while Q != nil                            // While priority queue is not empty
      u = Extract-Min(Q)                      // Extract next vertex from Pqueue to MST, let it be u

      for each v ∈ Adj[u]                     // For all vertex v part of adj. list of u
        if v ∈ Q and w(u, v) < key[v] then    // If v is still in queue (i.e. not part of MST yet) & weight of
                                              // edge u,v is less than current min key associated with v
          parent[v] = u                       // u is the parent of v
          key[v] = w(u,v)                     // Set min edge key for v to weight of edge u,v
        endif
      endfor
    endwhile                                  // Each time we loop back up, remember that the priority
Endalg                                        // is resorted in accordance to the new weighted keys!!
```

Prim Example



Discussion of Prim

- The algorithm stores the vertices in a priority queue based on minimum weight
- This weight represents the weight on an edge connecting that vertex to the MST being grown
- Initially these are set to infinity
- When a vertex is removed from the queue its adjacency list is checked and the correct weights inserted
- The queue always has weights on the edges adjacent to the existing MST
- The performance of Prim's algorithm depends on how we implement the priority queue.
- If we implement it as a heap, then the running time is O(E lg V).

MST Summary

- A Minimum Spanning Tree is a tree which connects all vertices in a graph and minimises the sum of the weights on its edges
- This has many useful applications
- The problem can be solved using a greedy strategy
- Kruskal's algorithm solves by sorting the edges
- Prim's algorithm solves by growing the tree from an arbitrary root by taking the shortest edge connecting the tree to the yet to be included vertices
- Prim is slightly more efficient than Kruskal