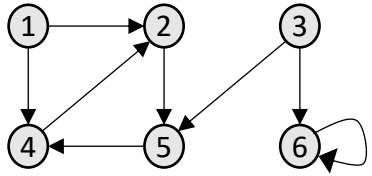


Graphs 3 – Weighted graphs
Lecture 21

COSC 242 – Algorithms and Data Structures

Today’s outline

1. Weighted graphs
2. Shortest paths
3. Dijkstra’s algorithm
4. Minimum spanning Tree
5. Prim’s algorithm

Today’s outline

1. Weighted graphs
2. Shortest paths
3. Dijkstra’s algorithm
4. Minimum spanning Tree
5. Prim’s algorithm

Weighted graphs

Graphs become more useful when we can give weights or costs to the edges. Weighted graphs can be used to model:

- maps with weights representing distances
- water networks with weights representing water capacities of pipes
- electrical circuits with weights representing resistance or maximum voltage or maximum current
- computer or phone networks with weights representing length of wires between nodes

Weighted graphs

One of the canonical applications for weighted graphs is finding the shortest path between two nodes. These algorithms are used in Google Maps for example.

We will focus on **single-source shortest paths**. These problems have a particular source vertex, s, and construct shortest paths to all other vertices in the graph (if they exist).

Dijkstra's algorithm and Prim's algorithm

Today we will be looking at two closely related algorithms for determining optimal paths through a graph.

Both approaches are examples of **greedy algorithms**. We will explore greedy algorithms in our next lecture, but briefly, a greedy algorithm is a simple heuristic that makes the locally optimal choice at each stage.

Dijkstra's algorithm resembles both breadth-first search (BFS) and Prim's algorithm. Both use a min-priority queue to find the "lightest" vertex outside a given set.

Today’s outline

1. Weighted graphs
2. Shortest path algorithms
3. Dijkstra’s algorithm
4. Minimum spanning Tree
5. Prim’s algorithm

Shortest paths problem

In a **shortest paths problem**, we have a weighted directed graph $\mathrm{G} = (\mathrm{V}, \mathrm{E})$, with a weight function $w : E \to \mathbb{R}$, that maps edges to real-valued weights.

The **weight** $\mathrm{w}(\mathrm{p})$ of path $p = \langle v_0, v_1, \ldots, v_k \rangle$ is the sum of the weights of its constituent edges: $\sum_{i=1}^{k} w(v_{i-1}, v_i)$.

Math glossary

You can read $w : E \to \mathbb{R}$ as "the function *w* that maps elements of *E* to elements of $\mathbb{R}$.
That is, if (*a*,*b*) ∈ *w*, then we write $\mathrm{b} = \mathrm{w}(\mathrm{a})$.
Function arrow notation and textbook Appendix B.3
Set notation

Shortest paths problem

We define the **Shortest path weight** $\delta(u, v)$ from *u* to *v* as:

$$\delta(u,v) = \begin{cases} min\{w(p): u \rightsquigarrow^p v\} & \text{If there is a path from } u \text{ to } v, \\ \infty & \text{otherwise.} \end{cases}$$

A **shortest path** from vertex *u* to vertex *v* is therefore any path *p* with weight $w(p) = \delta(u, v)$.

Notation glossary

$\rightsquigarrow^p$ – See Appendix B.4. We say that if there is a path *p* from *u* to *v*, then *v* is **reachable** from *u* via *p*, which can be written as $u \rightsquigarrow^p v$ if G is directed.

See also, Brackets

Shortest path algorithms and variants

Dijkstra's algorithm is a solution for the **single-source shortest-paths problem**: given a graph $G = (V, E)$, we want to find the shortest path from a given source vertex $s \in V$ to each vertex $v \in V$.

This algorithm for single-source can solve many other problems including: single-destination shortest-paths, single-pair shortest-path, All-pairs shortest-paths. The textbook delves into some of these.

A shortest path algorithm relies on the property that a shortest path between two vertices contains other shortest paths within it.

Today’s outline

1. Weighted graphs
2. Shortest paths
3. Dijkstra’s algorithm
4. Minimum spanning Tree
5. Prim’s algorithm

Relaxation

Single-source shortest path algorithms use the technique **relaxation**.

For each vertex $v \in V$, we maintain *v.d*, which is an upper bound on the weight of a shortest path from source *s* to *v*.

The value *v.d* is a **shortest path estimate**.

We begin by initialising the shortest path estimates:

```
procedure Initialize-Single-Source(G,s)
1:  for each vertex v ∈ G.V   // Initialise all vertices
2:      v.d = ∞               // Set all distances to inf
3:      v.π = NIL             // No predecessor
4:  s.d = 0                   // Set source distance to 0
```

Relaxation

Relaxing an edge (u, v) consists of testing whether we can improve the shortest path to *v* found so far by going through *u*. If we can, update *v.d* and *v.π*.

The process of relaxation *may* decrease the shortest-path estimate *v.d*, and update *v*'s predecessor *v.π*.

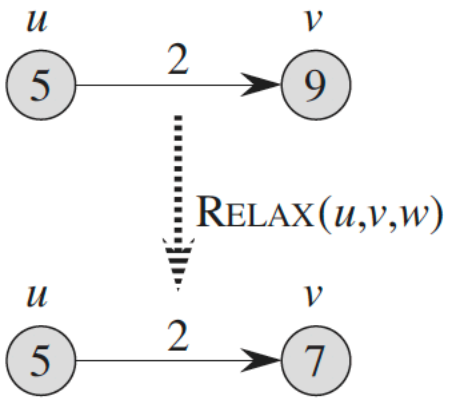
```
procedure Relax(u,v,w)
1:   if v.d > u.d + w(u,v)   // Is u's adj vertex v.d longer than this path?
2:        v.d = u.d + w(u,v) // Update v's distance to shorter value
3:        v.π = u            // u is v's new predecessor
```

Example of relaxation

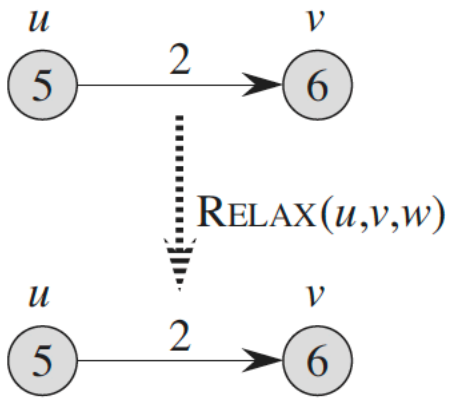
Here we relax an edge of weight $w(u,v) = 2$.

a) The value of $v.d$ decreases because $v.d > u.d + w(u,v)$

b) $v.d \not> u.d + w(u,v)$ before relaxation, so relaxation step leaves $v.d$ unchanged.



(a)



(b)

Dijkstra's algorithm

Dijkstra's algorithm solves the **single-source shortest-paths** problem on a weighted directed graph, for which all edge weights are non-negative.

The algorithm maintains a set *S* of vertices whose final shortest-path weights from source vertex *s* have been calculated.

The algorithm works by repeatedly selecting a vertex $u \in V - S$, with the minimum shortest-path estimate *d*. It then adds *u* to *S*, and relaxes all edges leaving *u*.

We use a min-priority queue of *Q* vertices, keyed by *d*.

Dijkstra's algorithm


```
procedure Dijkstra(G,w,s)
1:   Initialise-Single-Source(G,s)  // Initialise all vertices
2:   S = ∅                          // Set of vertices with shortest-paths
3:   Q = G.V                       // Set of unvisited vertices
4:   while Q ≠ ∅                   // While still vertices to explore
5:       u = Extract-Min(Q)        // Get vertex with min shortest path d
6:       S = S ∪ {u}               // Add to set with shortest paths
7:       for each vertex v ∈ G.Adj[u] // For each adjacent vertex
8:           Relax(u,v,w)          // Update v.d, v.π if new shortest path
end procedure
```

Dijkstra's algorithm - Single procedure version

```
procedure Dijkstra(G,w,s)
1:    for each vertex v ∈ G.V                  // Initialise all vertices
2:        v.d = ∞
3:        v.π = NIL
4:    s.d = 0                                  // Set source distance to 0
5:    S = ∅                                    // Set of vertices with shortest-paths
6:    Q = G.V                                  // Set of unvisited vertices
7:    while Q ≠ ∅                              // While still vertices to explore
8:        u = Extract-Min(Q)                   // Get vertex with min shortest path d
9:        S = S ∪ {u}                          // Add to set with shortest paths
10:       for each vertex v ∈ G.Adj[u]         // For each adjacent vertex
11:           if v.d > u.d + w(u,v)            // Is u's adj vertex v.d longer than this path?
12:               v.d = u.d + w(u,v)           // Update v's distance to shorter value
13:               v.π = u                      // u is v's new predecessor
end procedure
```

Example: Dijkstra's algorithm

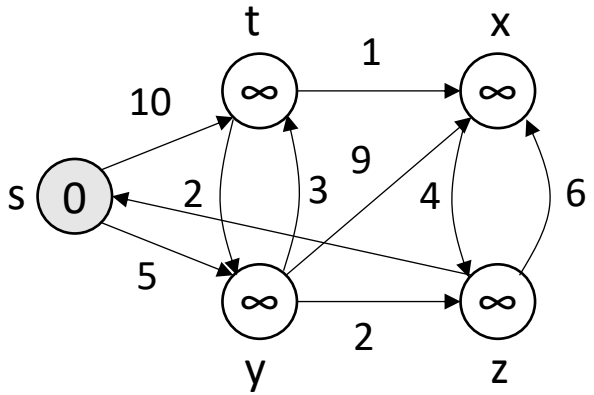
Line numbers from single procedure code

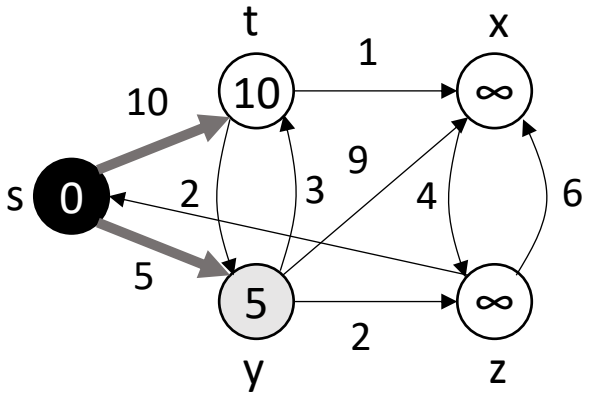
s is source vertex. Shortest-path estimates, *v.d*, appear within vertices. Shaded edges indicate predecessor values. Black vertices are in set *S*. White vertices are in min priority queue $Q = V - S$.

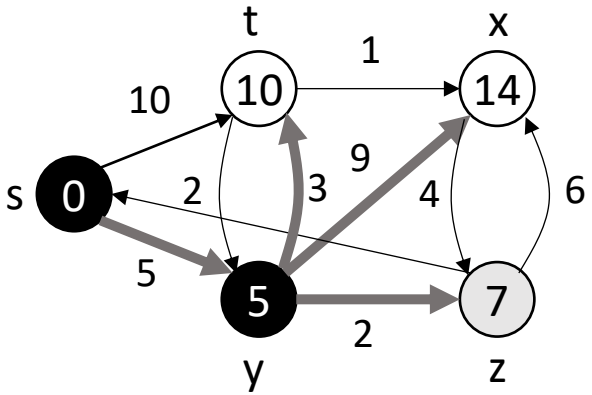
(a) Just before the while loop (Lines 4-8). Always begin at source *s*, as *s.d* = 0, Line 4.

(b) Add s to *S* (coloured black), and relax adjacent vertices *t* and *y*. *y* has the minimum *v.d* (shaded grey), and will be selected next.

(c) Add y to S, adjacent vertices *t*, *x*, and *z*. Note that *t* is updated again.







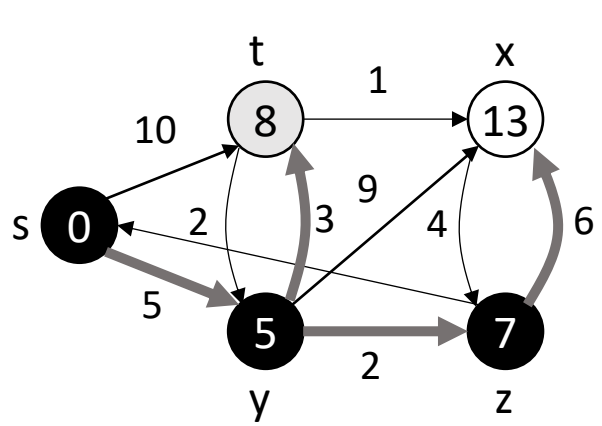
continued

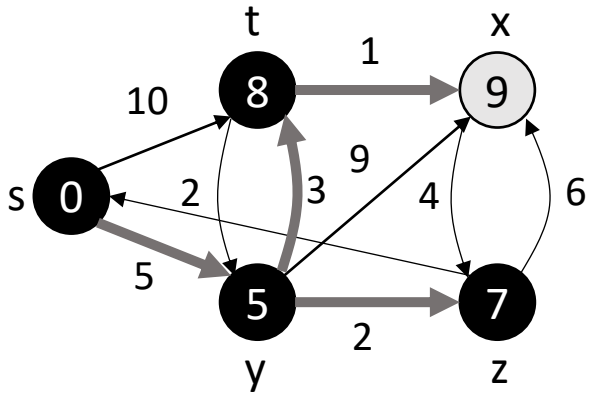
Example: Dijkstra's algorithm

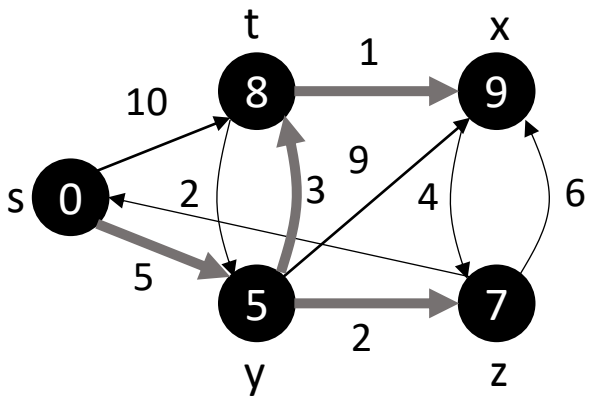
d) *z* is selected next. Update adjacent vertex *x*. *t* will be selected next.

e) Add *t* to S, update adjacent and final vertex *x*.

f) Add *x* to S. While loop (Line 7) will now terminate s all vertices have moved from *Q* to *S*.



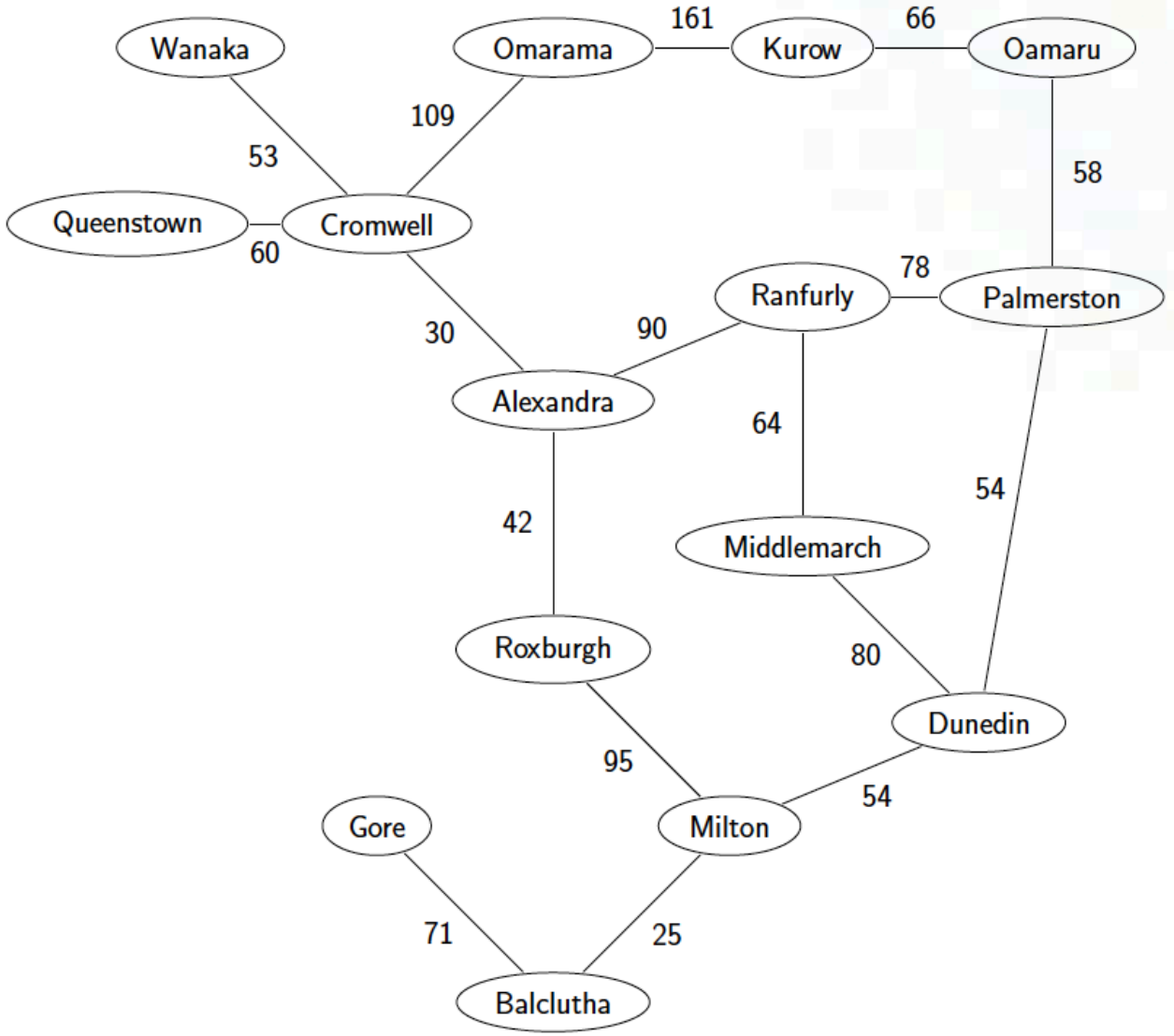




Class challenge

Activity Time.

Apply Dijkstra's algorithm, beginning at source vertex Gore.



Today’s outline

1. Weighted graphs
2. Shortest paths
3. Dijkstra’s algorithm
4. Minimum spanning Tree
5. Prim’s algorithm

Minimum spanning tree

Hypothetical scenario

You want to wire the campus' computer network. You need to make sure that every building is physically connected.

However, fibre cable is expensive. You want minimise the total length of cable used to connect all the buildings.

How do you know which buildings to connect to form a single connected network that minimises cable length/cost?

Minimum spanning tree

For this problem, the single-source shortest paths algorithm might not give us the overall minimum length of cable. Why?

Because Dijkstra's algorithm is concerned with minimising the path weight from a source vertex to all other vertices.

It does not minimise the total path weight in the graph. Our class challenge on SL34 will highlight this difference.

We need a different algorithm - a **minimum spanning tree algorithm**.

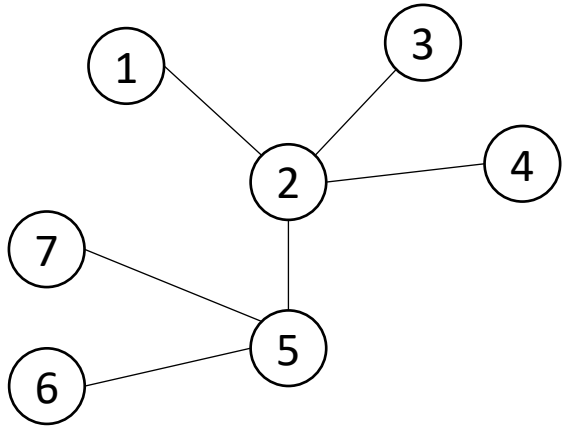
Trees

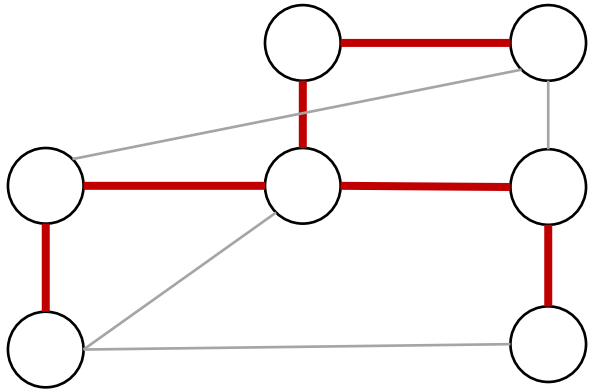
A **tree** is an undirected graph in which any two vertices are connected by *exactly one* path. All undirected acyclic graphs are trees.

A **spanning tree** of an undirected graph, *G*, is a connected acyclic subgraph of *G* that contains all the vertices in *G*, with a *minimum possible number of edges*.

A graph may have more than one spanning tree.

Tree





Spanning tree
Formed by red edges in Graph

MST definition

A **minimum spanning tree** of a weighted undirected graph, *G*, is a spanning tree of *G* with minimum total weight. That is, the sum of the edge weights is a minimum.

Put another way, an **MST** is a subset of the edges of a weighted undirected graph that connects all the vertices together, without any cycles, and with the minimum possible total edge weight.

Today’s outline

1. Weighted graphs
2. Shortest paths
3. Dijkstra’s algorithm
4. Minimum spanning Tree
5. Prim’s algorithm

MST Algorithm

Prim's algorithm is a classic MST algorithm that works in a similar way to Dijkstra's algorithm.

We start with an empty spanning tree. We use a min-priority queue to select the vertex with the lowest edge weight.

These vertices become ‘connected’, growing our spanning tree.

MST Algorithm

Unlike with Dijkstra's, our tree is not stored in an explicit data structure.

Rather, the tree is formed implicitly through our predecessor-child edges $v.\pi$, and the variable $v.key$, the minimum weight edge connecting v to a vertex in the MST.

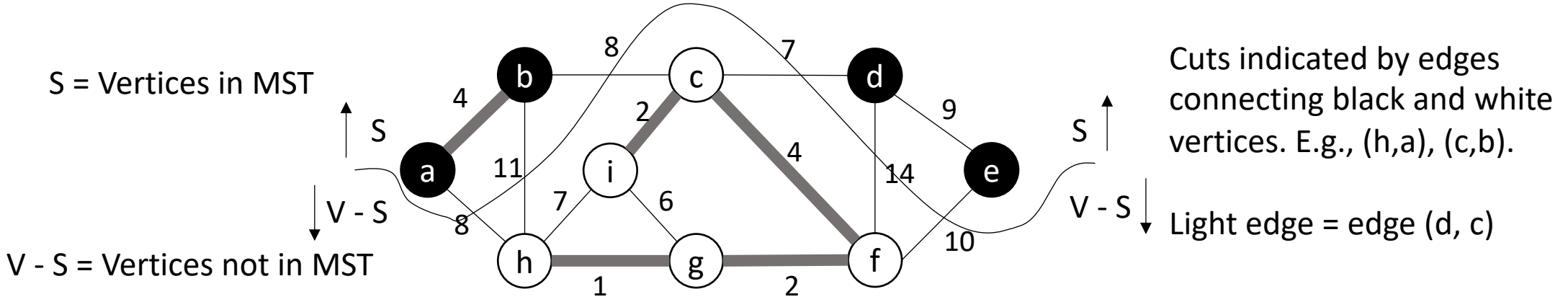
The set S represents vertices in our MST. We now have two sets of vertices:

- Set S = vertices in our MST
- Set $V - S$ = vertices not in our MST (those in the min-priority queue).

Cuts

In graph theory, a **cut** is a group of edges that connects two sets of vertices in a graph. An edge **crosses** the cut if one vertex is in *S*, the other is in *V* – *S*.

A **light edge** is an edge crossing a cut with the minimum weight of any edge crossing the cut. In the figure below, edge (d, c) is a light edge.



Cuts and Prim's algorithm

In Prim's algorithm, at each step we find the *light edge* – an edge that crosses the *cut* that connects set *S* (in our MST) with set *V* - *S* (not in MST). That edge has the minimum edge weight.

We dequeue this connecting vertex (remove from V - S), adding it to our MST by setting $v.\pi$ and setting edge weight in *v.key* (added to *S*).

Prim's algorithm

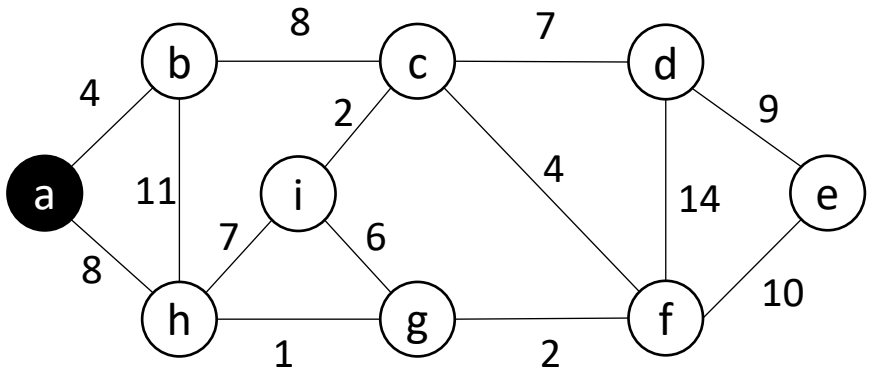

```
procedure MST-Prim(G,w,r)                    // r = root of minimum spanning tree
1:    for each vertex u ∈ G.V                // Initialise graph
2:        u.key = ∞
3:        u.π = NIL
4:    r.key = 0
5:    Q = G.V                                // Set V – S (not in MST)
6:    while Q ≠ ∅
7:        u = Extract-Min(Q)                 // Get vertex on light-edge that crosses cut (.key)
8:        for each vertex v ∈ G.Adj[u]       // Update u's adjacent vertices not in MST
9:            if v ∈ Q and w(u,v) < v.key    // Update non-MST vertices with lower weight edge?
10:               v.π = u
11:               v.key = w(u,v)
end procedure
```

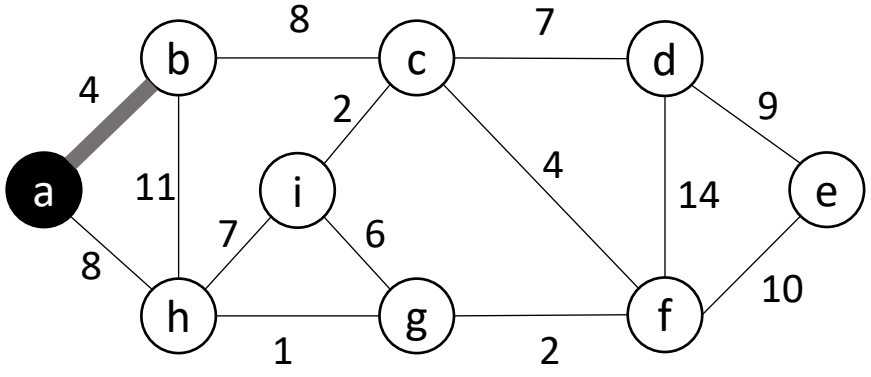
Example: Prim's algorithm

a) root (a) always selected first (key = 0), Line 7. Update keys for *b*, *h*.

b) *b* selected next (only b, h < ∞), and add edge (b, a) to tree. Update key for *c*. After this, the algorithm could add either (c, b) or (h, a) to the tree. Both are light edges.

Root = vertex a



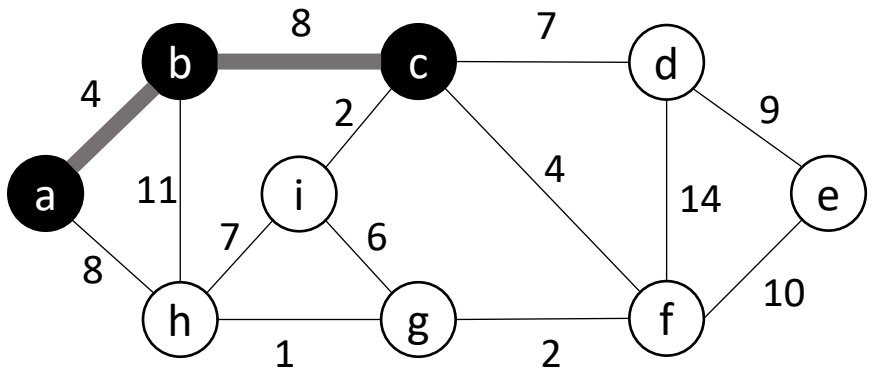


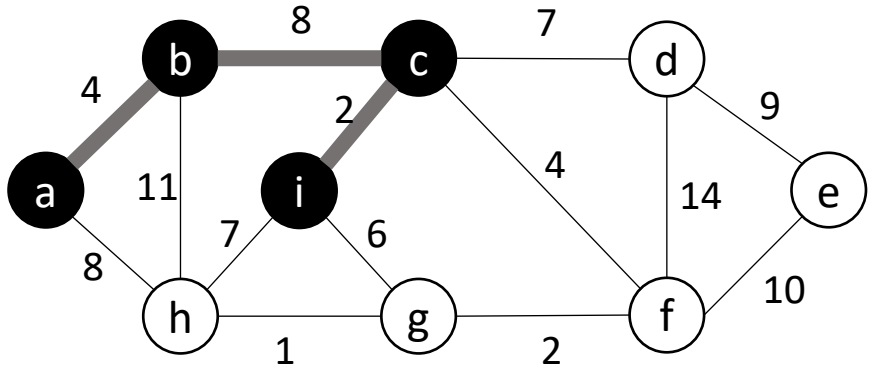
Example: Prim's algorithm

c) Algorithm chose (c, b). Update keys on *d, i*, and *f*.

d) *i* must be selected next. Update keys for *g* and *h*. Now, vertices d, g, h, and f have keys < ∞. Of these, *f* is the smallest (with 4) and will be selected next.

Root = vertex a



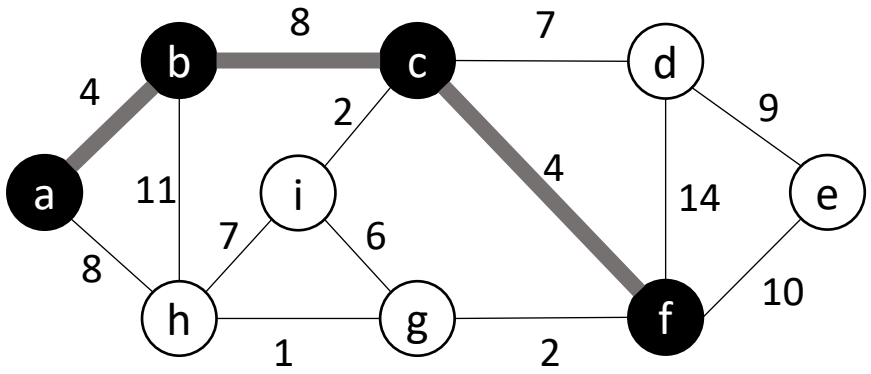


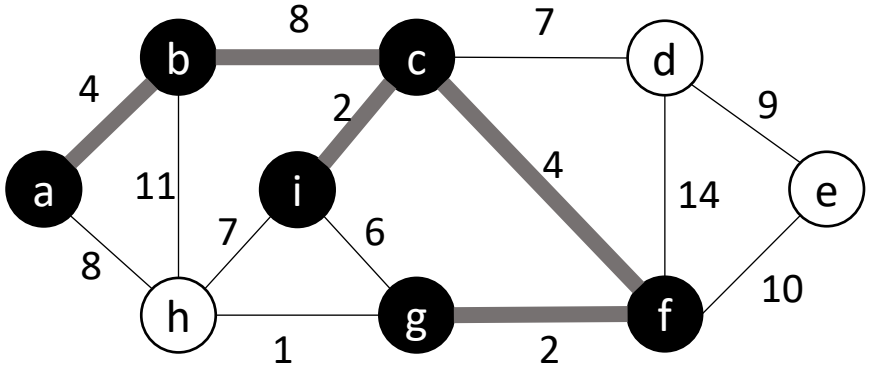
Example: Prim's algorithm

e) Select f, update keys of *e* and *g*. *g*'s new key (g, f) is lower than the key from (g, i).

f) All vertices now have keys < ∞. We next select *g*, as it has the lowest key (key = 2).

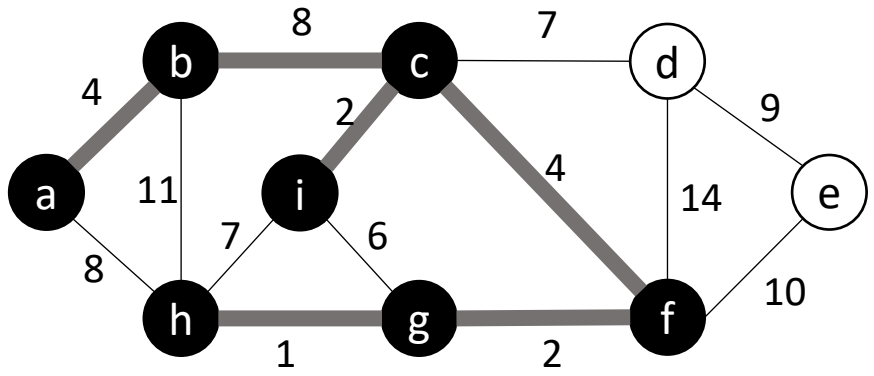
Root = vertex a

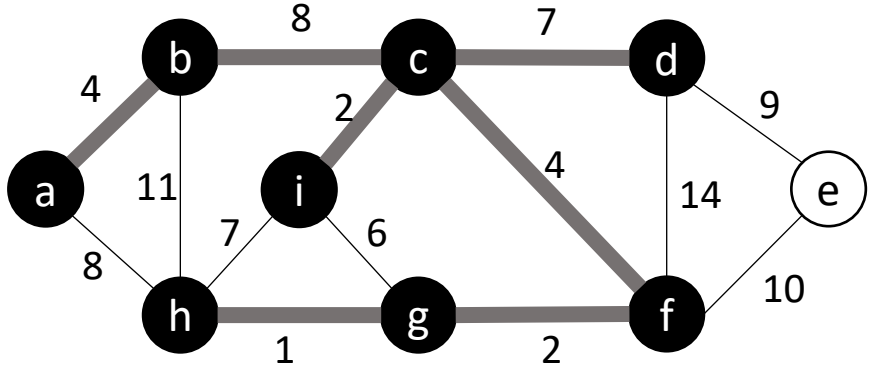


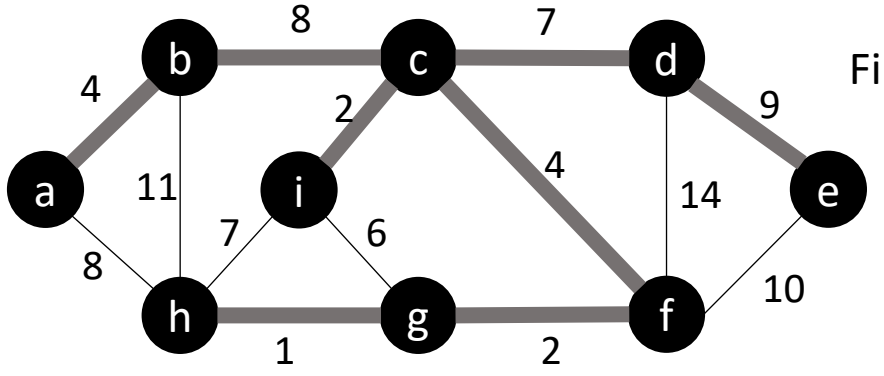


Example: Prim's algorithm

Root = vertex a



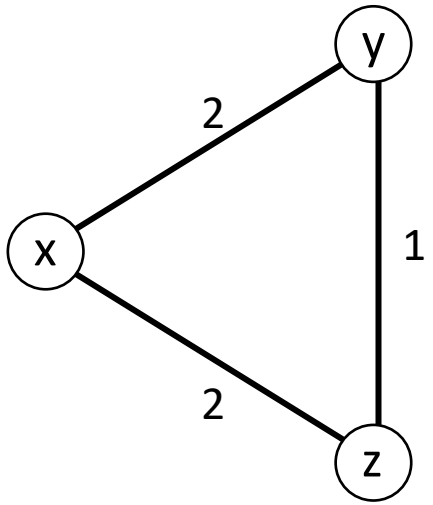




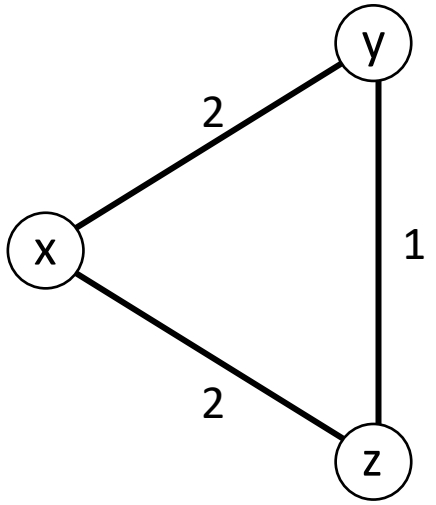
Finished

Class challenge

To illustrate the difference between Dijkstra's and Prim's algorithms, apply them respectively to the following graphs, starting at 'x'.



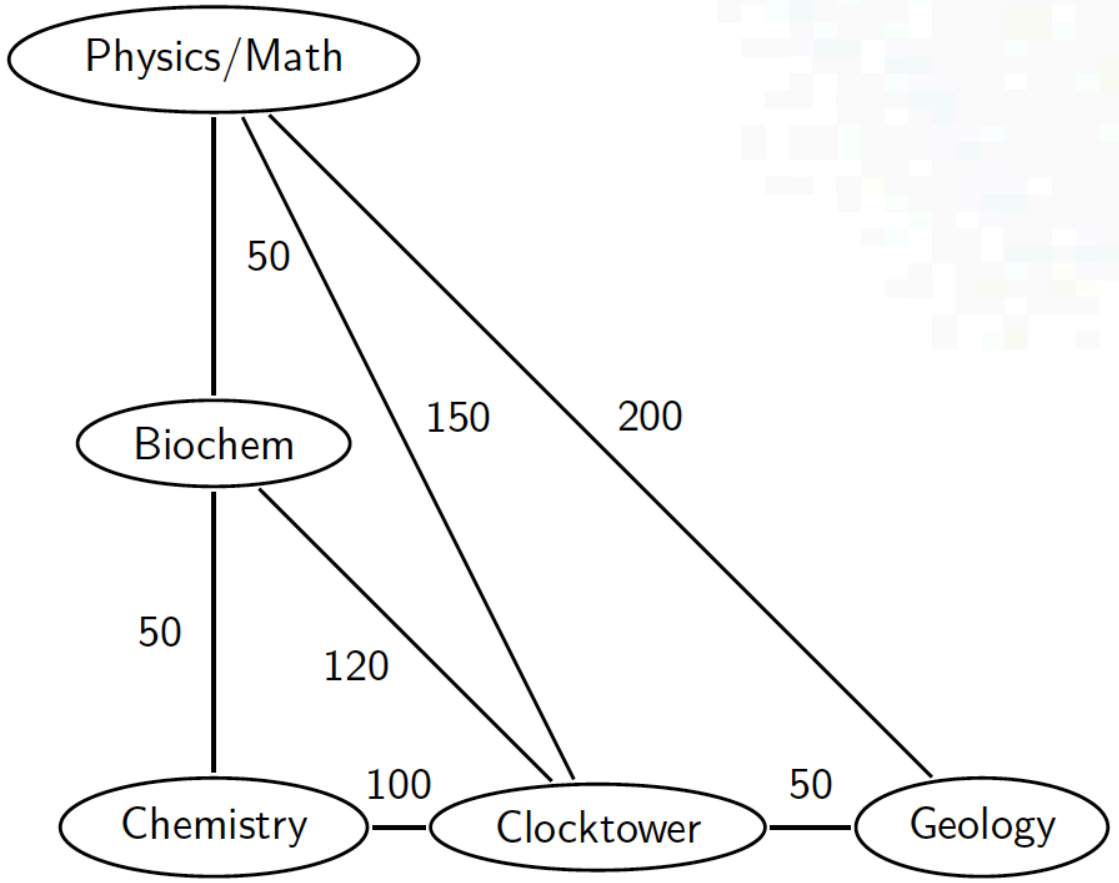
Dijkstra's



Prim's

Class challenge

Create an MST using Clocktower as root.



Suggested reading

Dijkstra's algorithm is discussed in Section 24.3 of the textbook. A general introduction to single-source shortest path algorithms, which forms the basis of Dijkstra's algorithm, is in Section 24 introduction.

Prim's algorithm is covered in Section 23.2.

Priority queues are covered in Section 6.5.

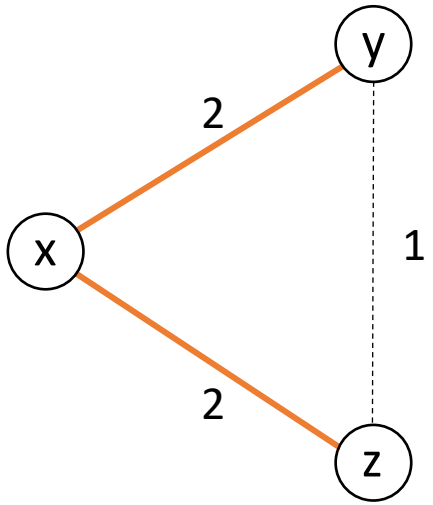
Paths are reviewed in Appendix B.4.

Arrow notation in Appendix B.3.

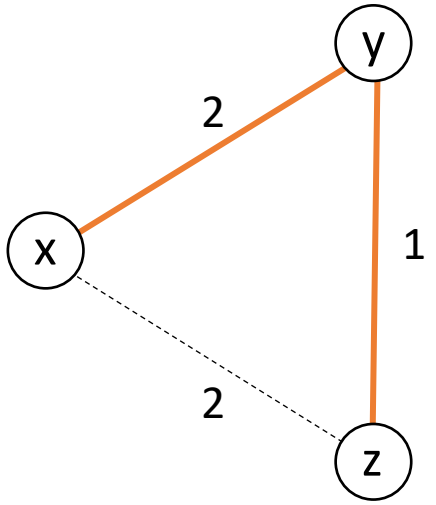
Solutions

Class challenge

To illustrate the difference between Dijkstra's and Prim's algorithms, apply them respectively to the following graphs.



Dijkstra's



Prim's