What is a graph?

Definition A *graph* G consists of a set $V(G)$ called *vertices* together with a collection $E(G)$ of pairs of vertices. Each pair $\{x, y\} \in E(G)$ is called an *edge* of G.

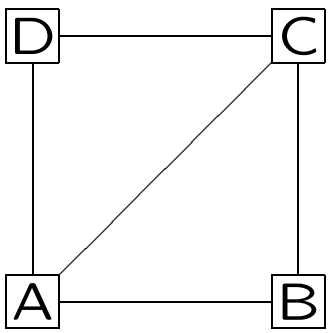
Example If

$$V(G) = \{A, B, C, D\}$$

and

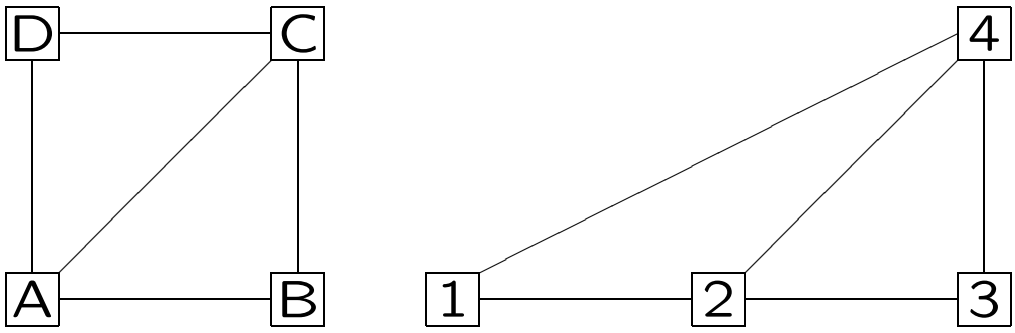
$$E(G) = \{\{A, B\}, \{C, D\}, \{A, D\}, \{B, C\}, \{A, C\}\}$$

then G is a graph with 4 vertices and 5 edges.



Isomorphisms

Consider the following two graphs:



Apart from the "names" of the vertices and the geometric positions it is clear that these two graphs are basically the same — in this situation we say that they are *isomorphic*.

Definition Two graphs G_1 and G_2 are *isomorphic* if there is a one-one mapping $\phi : V(G_1) \to V(G_2)$ such that $\{\phi(x), \phi(y)\} \in E(G_2)$ if and only if $\{x, y\} \in E(G_1)$.

In this case the isomorphism is given by the mapping

$$\phi(A) = 2 \quad \phi(B) = 3 \quad \phi(C) = 4 \quad \phi(D) = 1$$

Some graphs

Graphs are used to model a wide range of commonly occurring situations, enabling questions about a particular problem to be reduced to certain well-studied "standard" graph theory questions.

For examples consider the three graphs G_1, G_2 and G_3 defined as follows:

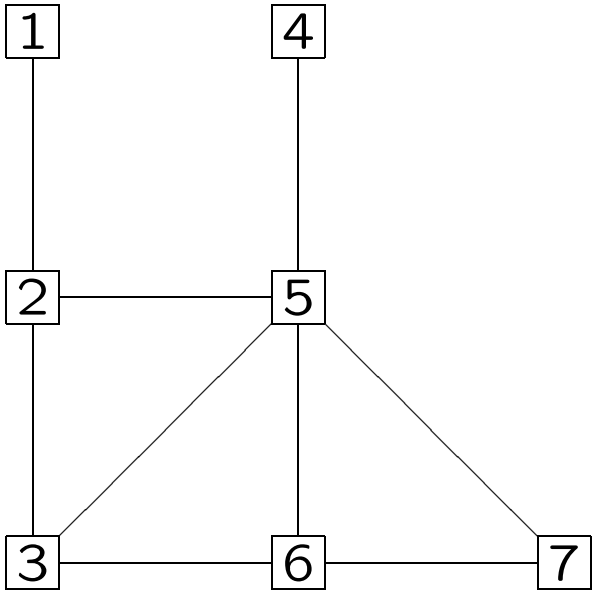
$V(G_1) =$ all the telephone exchanges in Australia, and $\{x, y\} \in E(G_1)$ if exchanges x and y are physically connected by fibre-optic cable.

$V(G_2) =$ all the airstrips in the world, and $\{x, y\} \in E(G_2)$ if there is a direct passenger flight from x to y.

$V(G_3) =$ all the people who have ever published a paper in a refereed journal in the world, and $\{x, y\} \in E(G_3)$ if x and y have been joint authors on a paper.

An example graph

Consider the following graph G_4.



The graph G_4 has 7 vertices and 9 edges.

Basic properties of graphs

Let us consider some of the basic terminology of graphs:

Adjacency If $\{x, y\} \in E(G)$, we say that x and y are *adjacent* to each other, and sometimes write $x \sim y$. The number of vertices adjacent to v is called the *degree* or *valency* of v.

Theorem The sum of the degrees of the vertices of a graph is even.

Paths A *path* of *length* n in a graph is a sequence of vertices $v_1 \sim v_2 \sim \cdots \sim v_{n+1}$ that contains no vertex twice. A *cycle* of length n is a sequence of vertices $v_1 \sim v_2 \sim \cdots v_n \sim v_1$ such that the vertices $\{v_1, v_2, \ldots, v_n\}$ are distinct.

Distance The *distance* between two vertices x and y in a graph is the length of the shortest path between them.

Subgraphs

If G is a graph, then a subgraph H is a graph such that

$$V(H) \subseteq V(G)$$

and

$$E(H) \subseteq E(G)$$

A *spanning* subgraph H has the property that $V(H) = V(G)$ — in other words H has been obtained from G only by removing edges.

An *induced* subgraph H must contain every edge of G whose endpoints lie in $V(H)$ — in other words H has been obtained from G by removing vertices and their adjoining edges.

Connectivity, forests and trees

Connected A graph G is *connected* if there is a path between any two vertices. If the graph is not connected then its connected components are the maximal induced subgraphs that are connected.

Forests A forest is a graph that has no cycles.

Trees A tree is a forest with only one connected component. It is easy to see that a tree with n vertices must have exactly $n-1$ edges.

The vertices of degree 1 in a tree are called the *leaves* of the tree.

Directed and weighted graphs

There are two important extensions to the basic definition of a graph.

Directed graphs In a directed graph, an edge is an ordered pair of vertices, and hence has a direction. In directed graphs, edges are often called *arcs*.

The graph G_2 above is really a directed graph.

Weighted graphs In a weighted graph, each of the edges is assigned a weight (usually a non-negative integer). More formally we say that a weighted graph is a graph G together with a weight function $w : E(G) \to \mathbf{R}$ (then $w(e)$ represents the weight of the edge e).

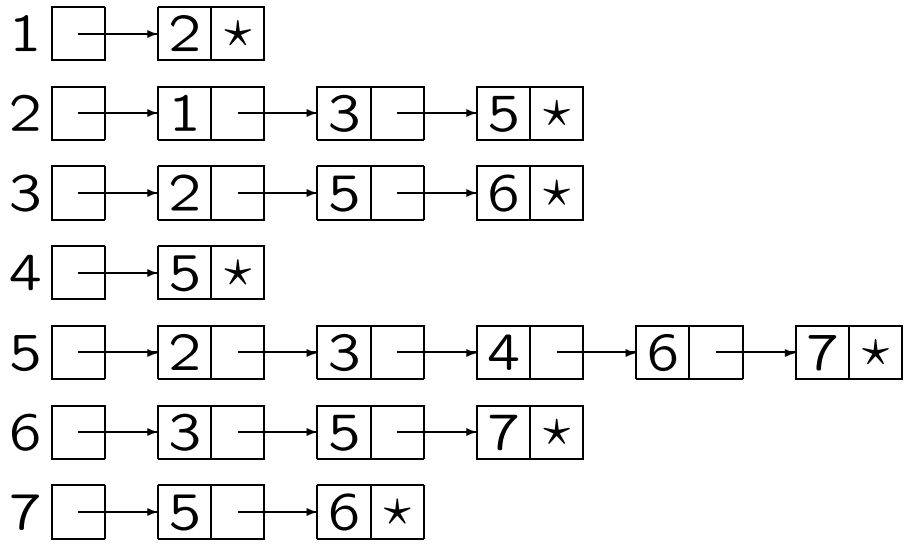
Weights on edges are often used to represent things like travel costs, transmission costs, road capacities and so on.

Representation of graphs

There are two main ways to represent a graph — adjacency lists or an adjacency matrix.

Adjacency lists The graph G is represented by an array of $|V(G)|$ linked lists, with each list containing the neighbours of a vertex.

Therefore we would represent G_4 as follows:



This representation requires two list elements for each edge and therefore the space required is $\Theta(|V(G)| + |E(G)|)$.

NOTE: In general to avoid writing $|V(G)|$ and $|E(G)|$ we shall simply put $V = |V(G)|$ and $E = |E(G)|$.

Adjacency matrix

The *adjacency matrix* of a graph G is a $V \times V$ matrix A where the rows and columns are indexed by the vertices and such that $A_{ij} = 1$ if and only if vertex i is adjacent to vertex j.

For graph G_4 we have the following

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

The adjacency matrix representation uses $\Theta(V^2)$ space.

For a *sparse* graph E is much less than V^2, and hence we would normally prefer the adjacency list representation.

For a *dense* graph E is close to V^2 and the adjacency matrix representation is preferred.

More on the two representations

For small graphs or those without weighted edges it is often better to use the adjacency matrix representation anyway.

It is also easy and intuitive to define adjacency matrix representations for directed and weighted graphs.

However your final choice of representation depends precisely what questions you will be asking. Consider how you would answer the following questions in both representations (in particular, how much time it would take).

Is vertex v adjacent to vertex w in an undirected graph?

What is the out-degree of a vertex v in a directed graph?

What is the in-degree of a vertex v in a directed graph?

Breadth-first search

Searching through a graph is one of the most fundamental of all algorithmic tasks, and therefore we shall examine several techniques for doing so.

Breadth-first search is a simple but extremely important technique for searching a graph. This search technique starts from a given vertex v and constructs a spanning tree for G, called the *breadth-first tree*. It uses a (first-in, first-out) *queue* as its main data structure.

Following CLR, as the search progresses, we will divide the vertices of the graph into three categories, *black* vertices which are the vertices that have been fully examined and incorporated into the tree, *grey* vertices which are the vertices that have been seen (because they are adjacent to a tree vertex) and placed on the queue, and *white* vertices, which have not yet been examined.

Breadth-first search initialization

The final breadth-first tree will be stored as an array called π where $\pi[x]$ is the immediate parent of x in the spanning tree. Of course, as v is the root of this tree, $\pi[v]$ will remain undefined.

To initialize the search we mark the colour of every vertex as *white* and the queue is empty. Then the first step is to mark the colour of v to be *grey*, put $\pi[v]$ to be undefined and add v to the queue.

Breadth-first search repetitive step

Then the following procedure is repeated until the queue is empty.

Take vertex w from the head of the queue
for each vertex x adjacent to w **do**
    if *colour[x]* is *white* **then**
        $\pi[x] \leftarrow w$
        colour[x] $\leftarrow$ *grey*
        Add x to the queue.
    end if
end for
colour[w] $\leftarrow$ *black*

At the end of the search, every vertex in the graph will have colour *black* and the parent or predecessor array π will contain the details of the breadth-first search tree.

Queues revisited

Recall that a *queue* is a data structure whereby the element taken off the data structure is the element that has been on the queue for the longest time.

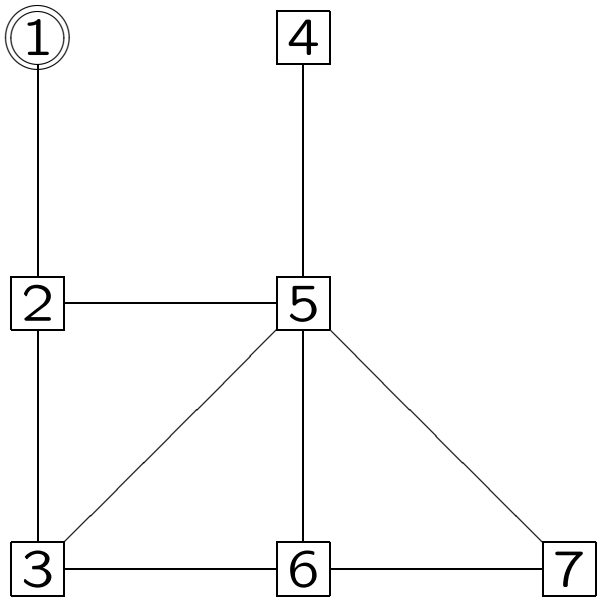
If the maximum length of the queue is known in advance (and is not too great) then a queue can be very efficiently implemented by simply using an array.

An array of n elements is initialized, and two pointers called *head* and *tail* are maintained — the head gives the location of the next element to be removed, while the tail gives the location of the first empty space in the array.

It is trivial to see that both enqueueing and dequeing operations take $\Theta(1)$ time.

See CLR, Section 11.1 for further details.

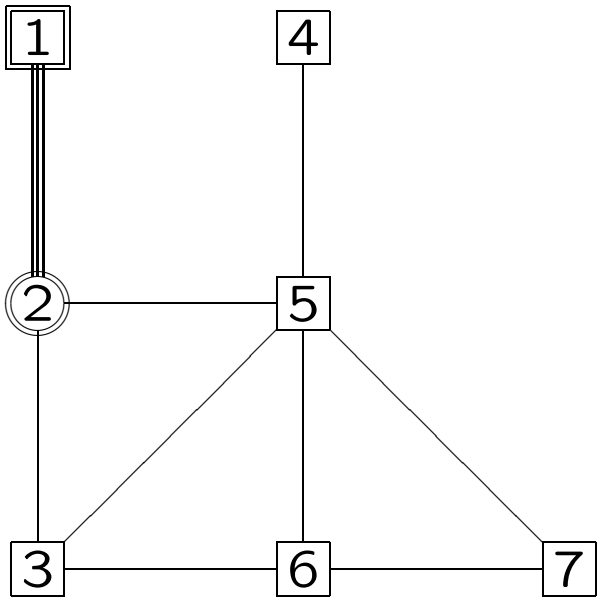
Example of breadth-first search



Head
↓
queue 1

x	*colour*[x]	$\pi[x]$
1	*grey*	**undef**
2	*white*	
3	*white*	
4	*white*	
5	*white*	
6	*white*	
7	*white*	

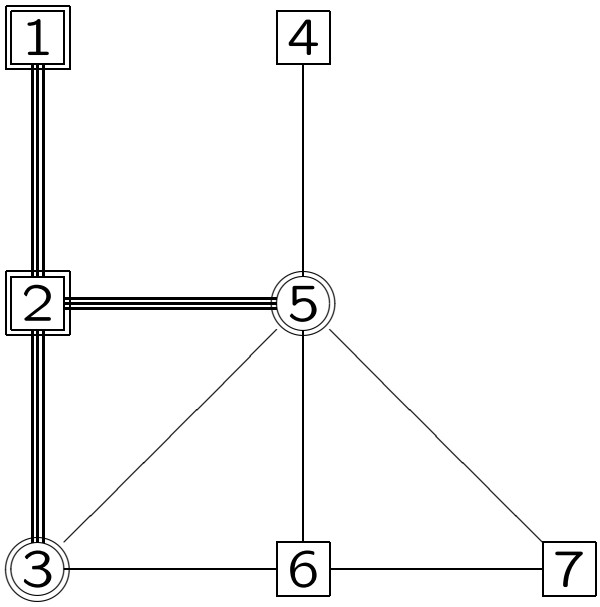
After visiting vertex 1



Head
↓
queue 1 2

x	*colour[x]*	$\pi[x]$
1	*black*	**undef**
2	*grey*	1
3	*white*	
4	*white*	
5	*white*	
6	*white*	
7	*white*	

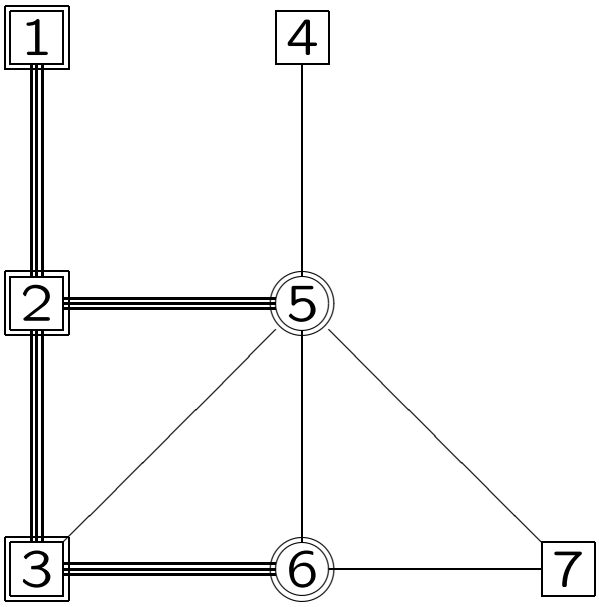
After visiting vertex 2



Head
↓
queue 1 2 3 5

x	*colour[x]*	$\pi[x]$
1	*black*	**undef**
2	*black*	1
3	*grey*	2
4	*white*	
5	*grey*	2
6	*white*	
7	*white*	

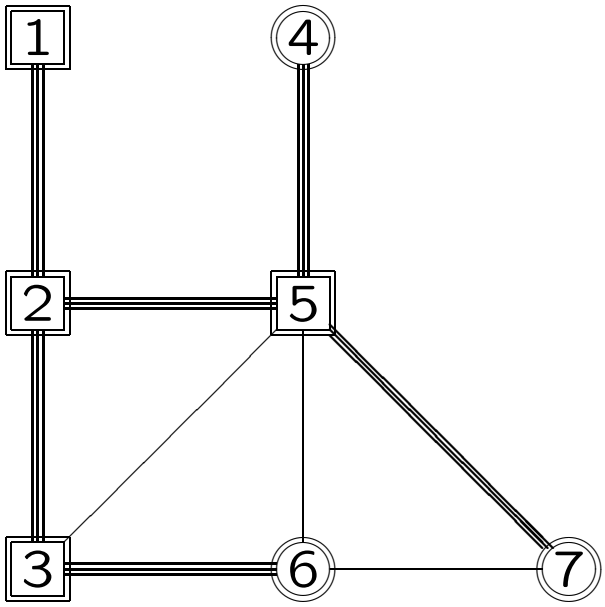
After visiting vertex 3



Head
↓
queue 1 2 3 5 6

x	*colour[x]*	$\pi[x]$
1	*black*	**undef**
2	*black*	1
3	*black*	2
4	*white*	
5	*grey*	2
6	*grey*	3
7	*white*	

After visiting vertex 5

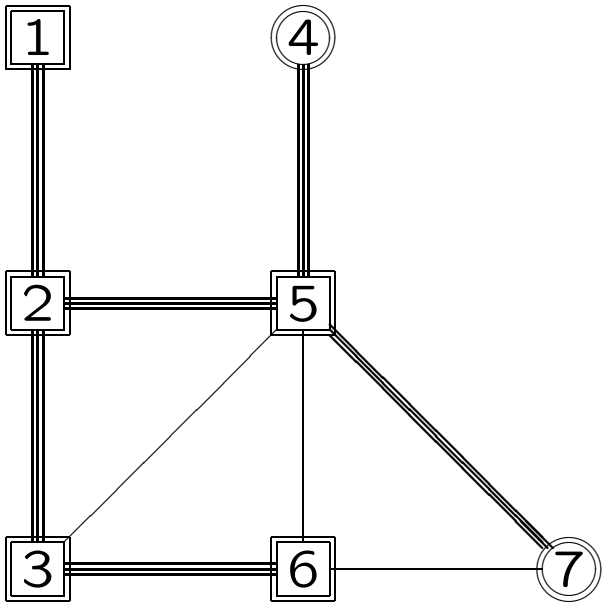


Head
↓

queue 1 2 3 5 6 4 7

x	*colour[x]*	$\pi[x]$
1	*black*	**undef**
2	*black*	1
3	*black*	2
4	*grey*	5
5	*black*	2
6	*grey*	3
7	*grey*	5

After visiting vertex 6

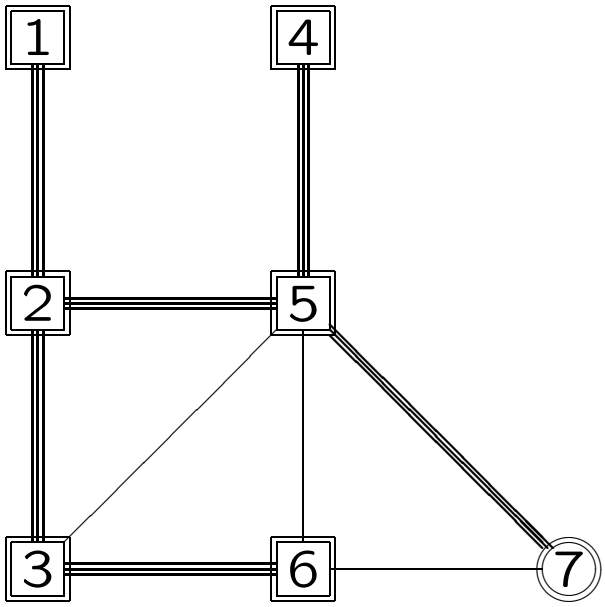


Head
↓

queue 1 2 3 5 6 4 7

x	*colour[x]*	$\pi[x]$
1	*black*	**undef**
2	*black*	1
3	*black*	2
4	*grey*	5
5	*black*	2
6	*black*	3
7	*grey*	5

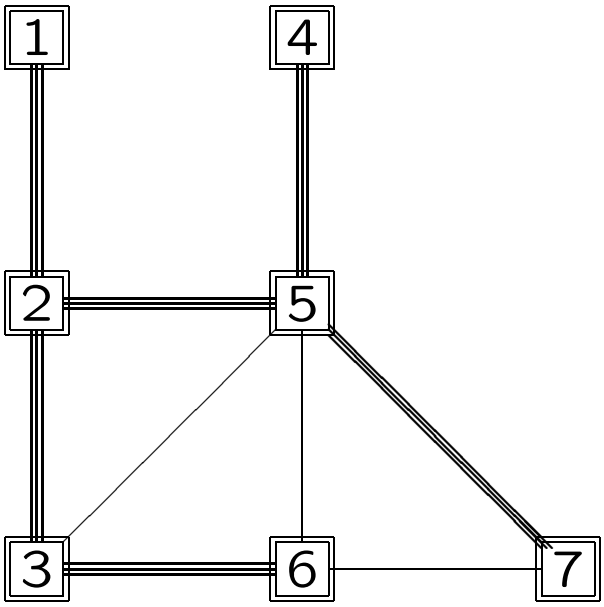
After visiting vertex 4



Head
↓
queue 1 2 3 5 6 4 7

x	*colour[x]*	$\pi[x]$
1	*black*	**undef**
2	*black*	1
3	*black*	2
4	*black*	5
5	*black*	2
6	*black*	3
7	*grey*	5

After visiting vertex 7



Head
↓

queue 1 2 3 5 6 4 7

x	*colour[x]*	$\pi[x]$
1	*black*	**undef**
2	*black*	1
3	*black*	2
4	*black*	5
5	*black*	2
6	*black*	3
7	*black*	5

At termination

At the termination of breadth-first search every vertex in the same connected component as v is a black vertex and the array π contains details of a spanning tree for that component — the breadth-first tree.

Time analysis

During the breadth-first search each vertex is enqueued once and dequeued once. As each enqueueing/dequeuing operation takes constant time, the queue manipulation takes $\Theta(V)$ time. At the time the vertex is dequeued, the adjacency list of that vertex is completely examined. Therefore we take $\Theta(E)$ time examining all the adjacency lists and the total time is $\Theta(V + E)$.

Uses of BFS

Breadth-first search is particularly useful for certain simple tasks such as determining whether a graph is connected, or finding the distance between two vertices.

The vertices of G are examined in order of increasing distance from v — first v, then its neighbours, then the vertices at distance 2 from v and so on. The spanning tree constructed provides a shortest path from any vertex back to v just by following the array π.

Therefore it is simple to modify the breadth-first search to provide an array of distances *dist* where *dist[*u*]* is the distance of the vertex u from the source vertex v.

Breadth-first search finding distances

To initialize the search we mark the colour of every vertex as *white* and the queue is empty. Then the first step is to mark the colour of v to be *grey*, put $\pi[v]$ to be undefined, put *dist[v]* to be 0, and add v to the queue.

Then until the queue is empty we repeat the following procedure.

Take vertex w from the head of the queue
for each vertex x adjacent to w **do**
    if *colour[x]* is *white* **then**
        dist[x] $\leftarrow$ *dist[w]+1*
        $\pi[x] \leftarrow w$
        colour[x] $\leftarrow$ *grey*
        Add x to the queue
    end if
end for
colour[w] $\leftarrow$ *black*

Depth-first search

Depth-first search is another important technique for searching a graph. Similarly to breadth-first search it also computes a spanning tree for the graph, but the tree is very different.

The structure of depth-first search is naturally *recursive* so we will give a recursive description of it. Nevertheless it is useful and important to consider the non-recursive implementation of the search.

The fundamental idea behind depth-first search is to visit the next unvisited vertex, thus extending the current path as far as possible. When the search gets stuck in a "corner" we back up along the path until a new avenue presents itself (this is called *backtracking*).

Basic recursive depth-first search

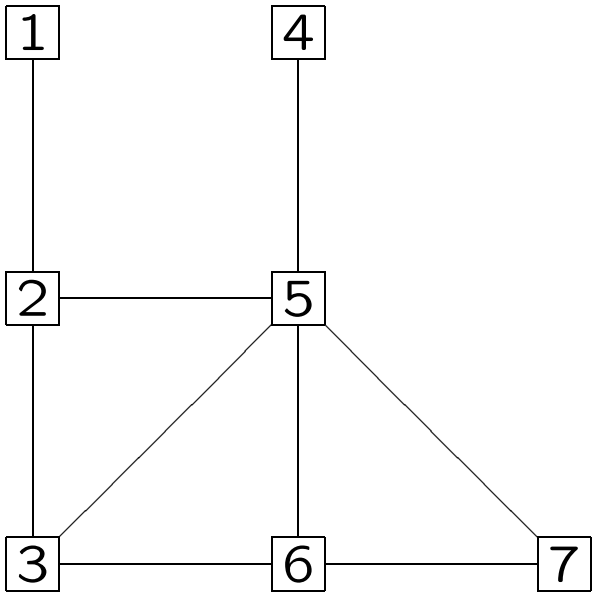
The following recursive program computes the depth-first search tree for a graph G starting from the source vertex v.

To initialize the search we mark the colour of every vertex as *white*. Then we call the recursive routine DFS(v) where v is the source vertex.

procedureDFS(w)
colour[w] $\leftarrow$ *grey*
for each vertex x adjacent to w **do**
    if *colour[x]* is *white* **then**
        $\pi[x] \leftarrow w$
        DFS(x)
    end if
end for
colour[w] $\leftarrow$ *black*

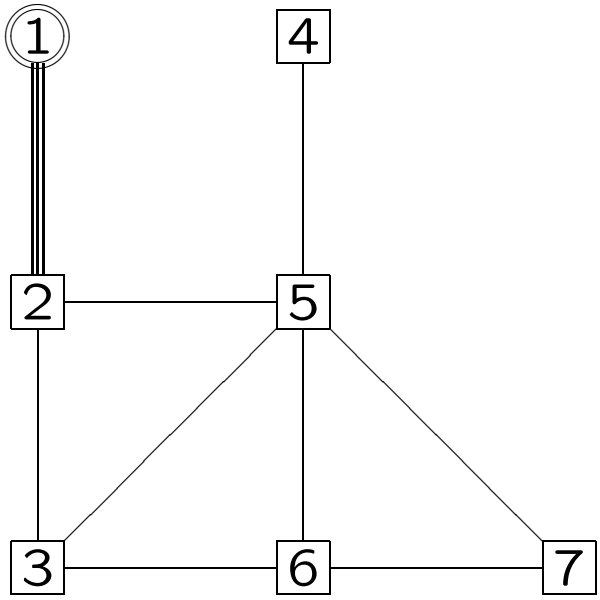
At the end of this depth-first search procedure we have produced a spanning tree containing every vertex in the connected component containing v.

Example of depth-first search



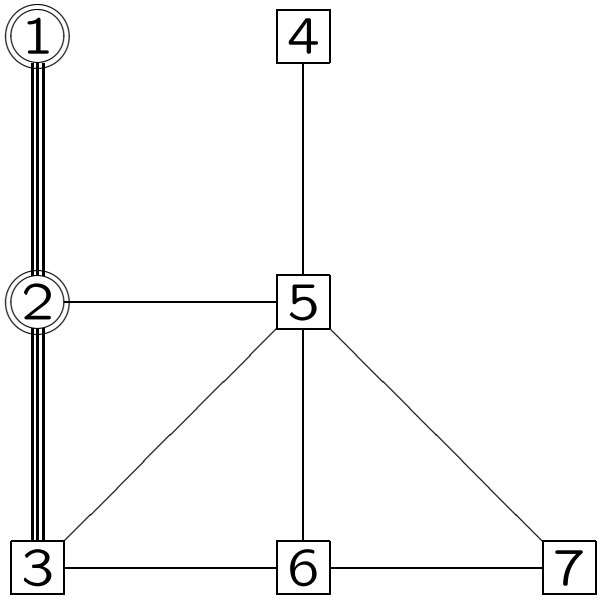
x	*colour*[x]	$\pi[x]$
1	*white*	**undef**
2	*white*	
3	*white*	
4	*white*	
5	*white*	
6	*white*	
7	*white*	

Immediately prior to calling DFS(2)



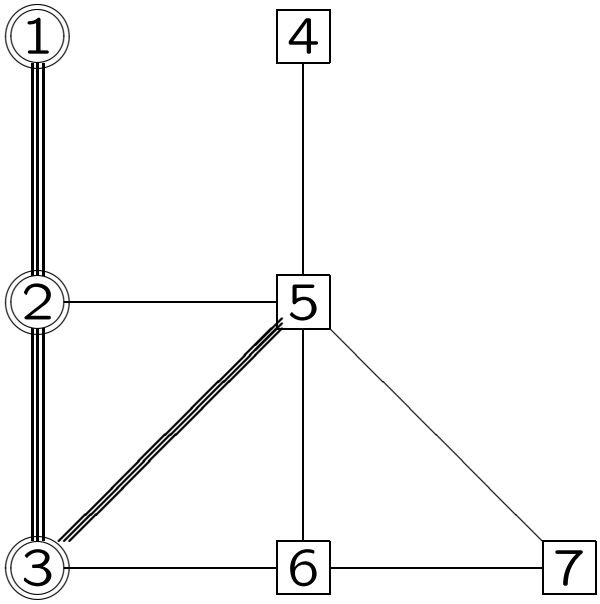
x	*colour[x]*	$\pi[x]$
1	*grey*	**undef**
2	*white*	1
3	*white*	
4	*white*	
5	*white*	
6	*white*	
7	*white*	

Immediately prior to calling DFS(3)



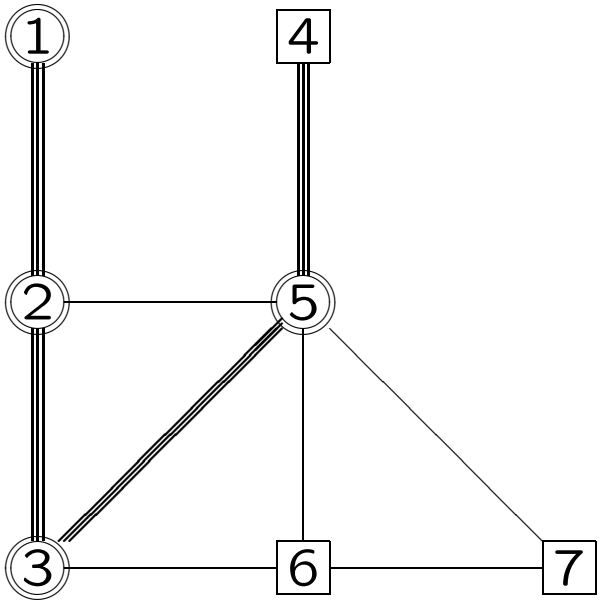
x	*colour[x]*	$\pi[x]$
1	*grey*	**undef**
2	*grey*	1
3	*white*	2
4	*white*	
5	*white*	
6	*white*	
7	*white*	

Immediately prior to calling DFS(5)



x	*colour[x]*	$\pi[x]$
1	*grey*	**undef**
2	*grey*	1
3	*grey*	2
4	*white*	
5	*white*	3
6	*white*	
7	*white*	

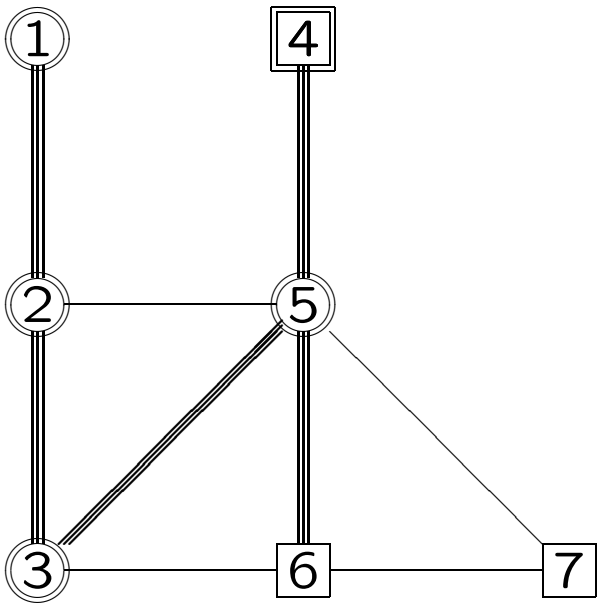
Immediately prior to calling DFS(4)



x	*colour[x]*	$\pi[x]$
1	*grey*	**undef**
2	*grey*	1
3	*grey*	2
4	*white*	5
5	*grey*	3
6	*white*	
7	*white*	

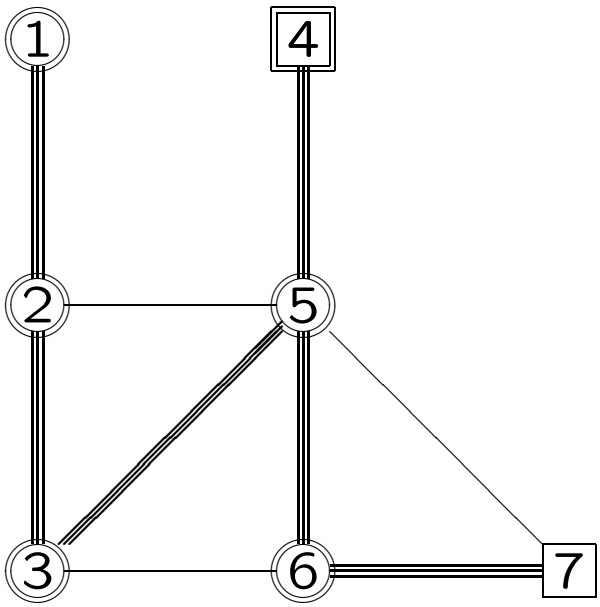
Immediately prior to calling DFS(6)

Now the call to DFS(4) actually finishes without making any more recursive calls so we return to examining the neighbours of vertex 5, the next of which is vertex 6.



x	*colour[x]*	$\pi[x]$
1	*grey*	**undef**
2	*grey*	1
3	*grey*	2
4	*black*	5
5	*grey*	3
6	*white*	5
7	*white*	

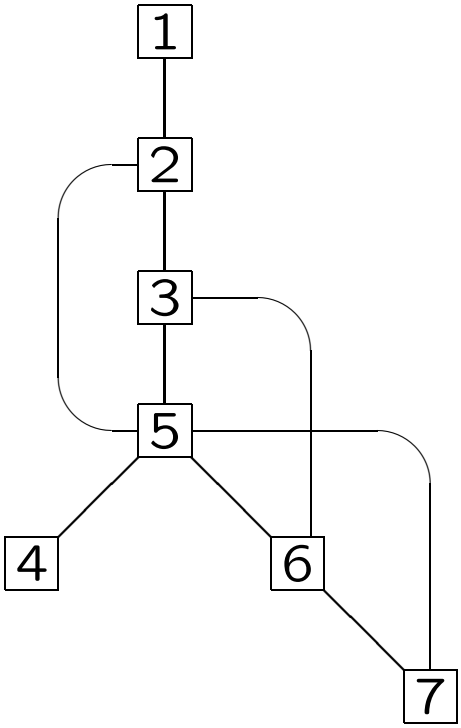
Immediately prior to calling DFS(7)



x	*colour[x]*	$\pi[x]$
1	*grey*	**undef**
2	*grey*	1
3	*grey*	2
4	*black*	5
5	*grey*	3
6	*grey*	5
7	*white*	6

The depth-first search tree

After completion of the search we can draw the depth-first search tree for this graph:



In this picture the slightly thicker straight edges are the *tree edges* and the remaining edges are the *back edges* — the back edges arise when we examine an edge (u, v) and discover that its endpoint v no longer has the colour *white*

Analysis of DFS

The running time of DFS is easy to analyse as follows.

First we observe that the routine DFS(w) is called exactly once for each vertex w; during the execution of this routine we perform only constant time array accesses, and run through the adjacency list of w once.

Running through the adjacency list of each vertex exactly once takes $\Theta(E)$ time overall, and hence the total time taken is $\Theta(V + E)$.

In fact, we can say more and observe that because every vertex and every edge are examined precisely once in both BFS and DFS, the time taken is $\Theta(V + E)$.

Discovery and finish times

The operation of depth-first search actually gives us more information than simply the depth-first search tree; we can assign two times to each vertex.

Consider the following modification of the search, where *time* is a global variable that starts at time 1.

procedure DFS(w)
colour[w] $\leftarrow$ *grey*
discovery[w] $\leftarrow$ *time*
time $\leftarrow$ *time+1*
for each vertex x adjacent to w **do**
    if *colour[x]* is *white* **then**
        $\pi[x] \leftarrow w$
        DFS(x)
    end if
end for
colour[w] $\leftarrow$ *black*
finish[w] $\leftarrow$ *time*
time $\leftarrow$ *time+1*

The parenthesis property

This assigns to each vertex a *discovery* time, which is the time at which it is first discovered, and a *finish* time, which is the time at which all its neighbours have been searched and it no longer plays any further role in the search.

The discovery and finish times satisfy a property called the *parenthesis property*.

Imagine writing down an expression consisting entirely of labelled parentheses — at the time of discovering vertex u we open a parenthesis $(_u$ and a the time of finishing with u we close the parenthesis $_u)$.

Then the resulting expression is a well-formed expression with correctly nested parentheses.

For our example depth-first search we get:

$$(_1\ (_2\ (_3\ (_5\ (_4\ 4)\ (_6\ (_7\ 7)\ 6)\ 5)\ 3)\ 2)\ 1)$$

Depth-first search for directed graphs

A depth-first search on an undirected graph produces a classification of the edges of the graph into *tree edges*, or *back edges*. For a directed graph, there are further possibilities. The same depth-first search algorithm can be used to classify the edges into four types:

tree edges If the procedure DFS(u) calls DFS(v) then (u, v) is a tree edge

back edges If the procedure DFS(u) explores the edge (u, v) but finds that v is an already visited ancestor of u, then (u, v) is a back edge

forward edges If the procedure DFS(u) explores the edge (u, v) but finds that v is an already visited descendant of u, then (u, v) is a forward edge

cross edges All other edges are cross-edges

Topological sort

We shall consider a classic simple application of depth-first search.

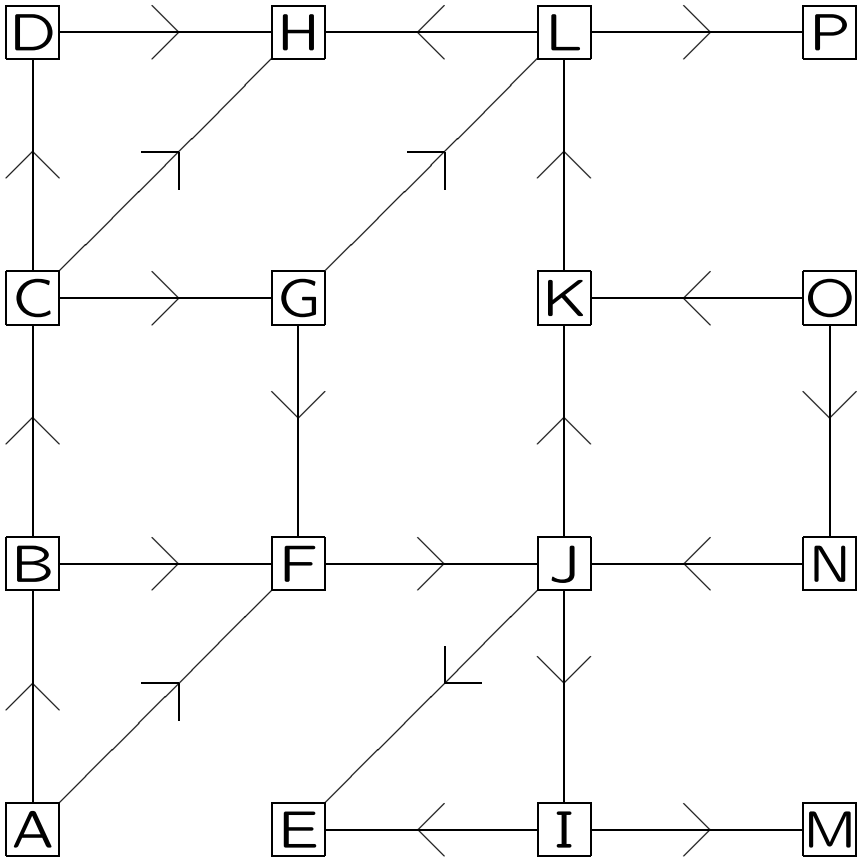
Definition A *directed acyclic graph (dag)* is a directed graph with no directed cycles.

Theorem In a depth-first search of a dag there are no back edges.

Consider now some complicated process in which various jobs must be completed before others are started. We can model this by a graph D where the vertices are the jobs to be completed and there is an edge from job u to job v if job u must be completed before job v is started. Our aim is to find some linear ordering of the jobs such that they can be completed without violating any of the constraints.

This is called finding a *topological sort* of the dag D.

Example of a dag to be topologically sorted



What is the appropriate linear order in which to do these jobs so that all the precedences are satisfied.

Algorithm for TOPOLOGICAL SORT

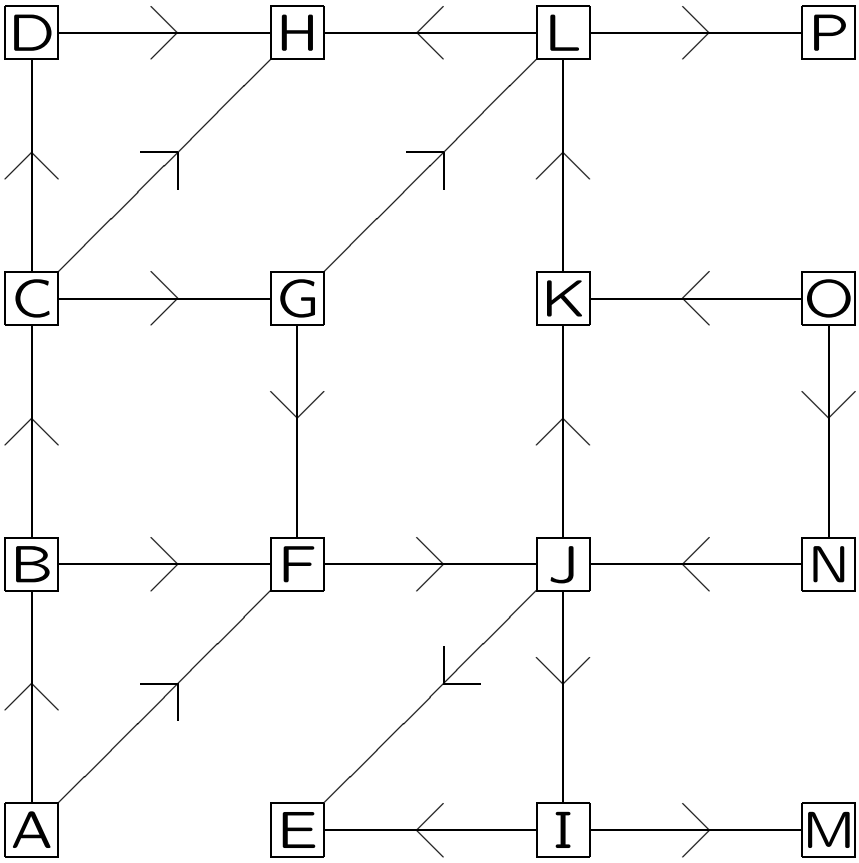
The algorithm for topological sort is an extremely simple application of depth-first search.

Algorithm

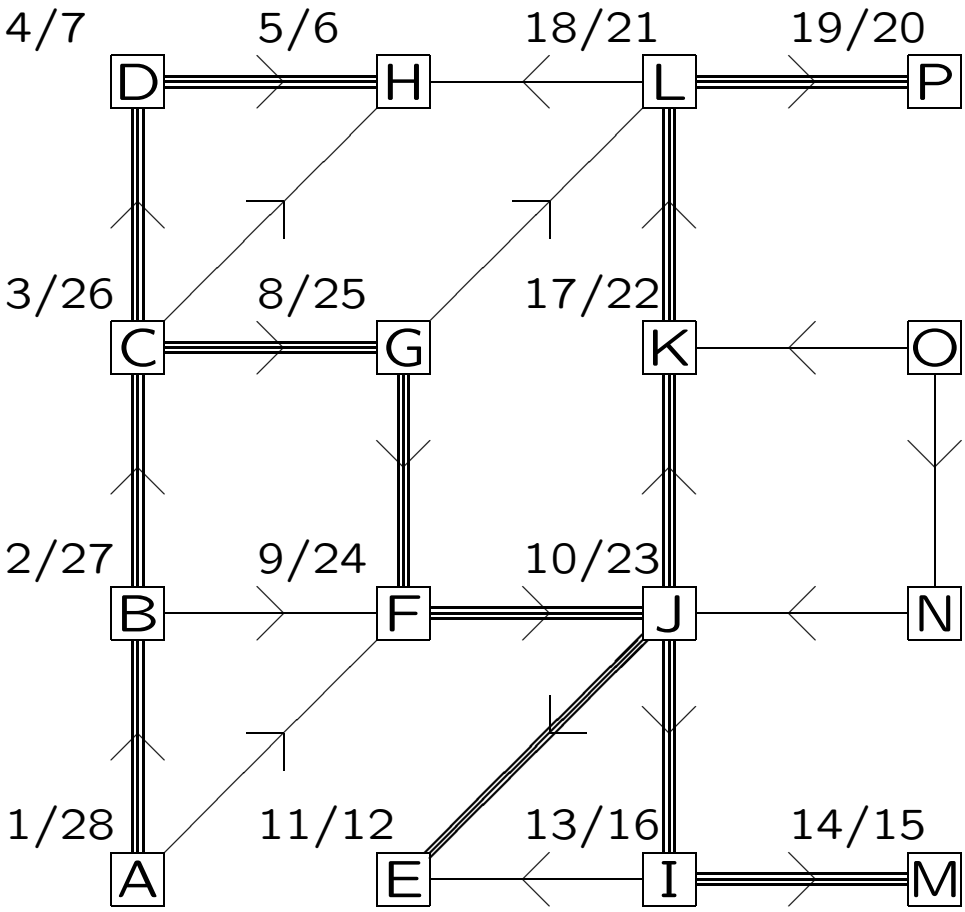
Apply the depth-first search procedure to find the finishing times of each vertex. As each vertex is finished, put it onto the *front* of a linked list.

At the end of the depth-first search the linked list will contain the vertices in topologically sorted order.

Doing the topological sort

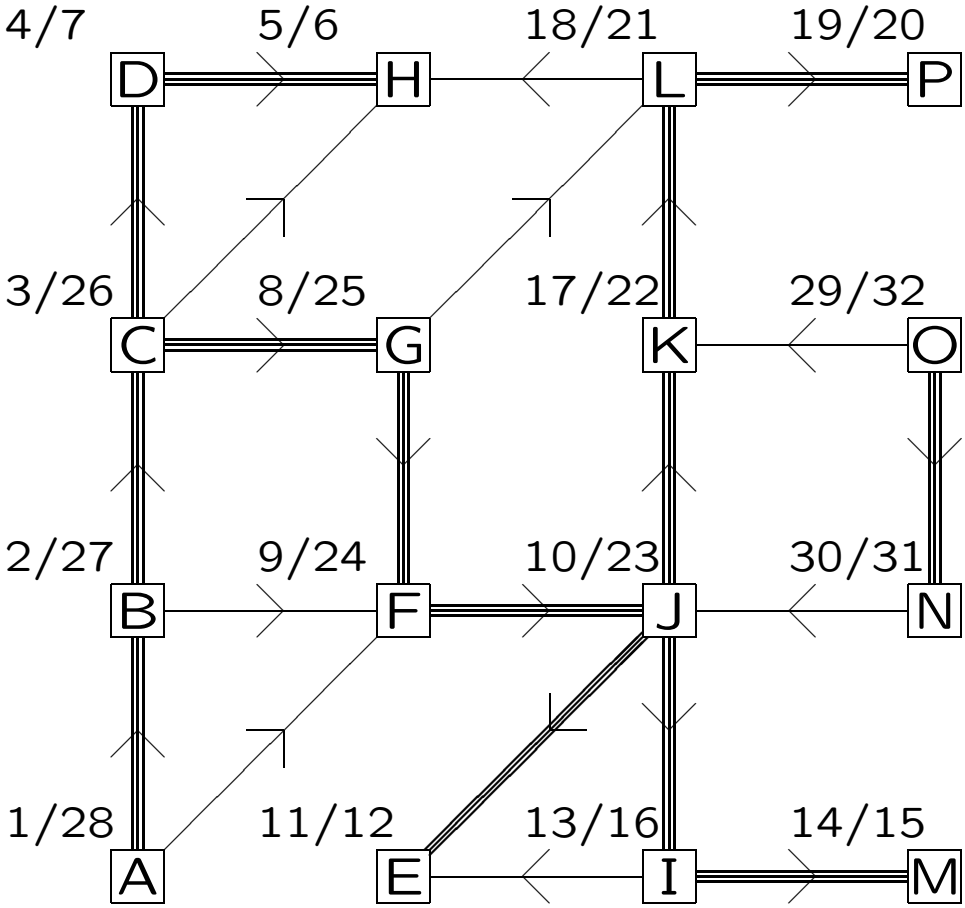


After the first depth-first search



Notice that there is a component that has not been reached by the depth-first search. To complete the search we just repeatedly perform depth-first searches until all vertices have been examined.

After the entire search



As the vertices were placed at the front of a linked list as they became finished the final topological sort is: $O - N - A - B - C - G - F - J - K - L - P - I - M - E - D - H$

A topologically sorted dag has the property that any edges drawn in the above diagram will got from left-to-right.

Analysis and correctness

Time analysis of the algorithm is very easy — to the $\Theta(V + E)$ time for the depth-first search we must add $\Theta(V)$ time for the manipulation of the linked list. Therefore the total time taken is again $\Theta(V + E)$.

Why does it work?

We shall try to show that for any edge (u, v) in the dag the finishing time $f(u) > f(v)$.
Consider the stage at which the edge (u, v) is encountered. If (u, v) is a tree edge, then the depth-first search proceeds from u to v and clearly finishes with v before finally returning to u. On the other hand if (u, v) is a forward or cross edge, then the vertex v has already been completely examined by this stage, and hence vertex u must have a later finishing time. The edge (u, v) cannot be a back edge because a dag has no cycles.

Other uses for DFS

DFS is the standard algorithmic method for solving the following two problems:

Strongly connected components In a directed graph D a strongly connected component is a maximal subset S of the vertices such that for any two vertices u, $v \in S$ there is a directed path from u to v and from v to u.

Depth-first search can be used on a digraph to find strongly connected components in time $\Theta(V + E)$.

Biconnected components In a connected graph G, an *articulation point* is a vertex whose removal disconnects the graph.

Depth-first search can be used on a graph to find all the articulation points in time $\Theta(V + E)$.

Minimum spanning tree (MST)

Consider a group of villages in a remote area that are to be connected by telephone lines. There is a certain cost associated with laying the lines between any pair of villages, depending on their distance apart, the terrain and some pairs just cannot be connected.

Our task is to find the minimum possible cost in laying lines that will connect all the villages.

This situation can be modelled by a weighted graph W, in which the weight on each edge is the cost of laying that line. A *minimum spanning tree* in a graph is a subgraph that is (1) a spanning subgraph (2) a tree and (3) has a lower weight than any other spanning tree.

It is clear that finding a MST for W is the solution to this problem.

The greedy method

Definition A *greedy algorithm* is an algorithm in which at each stage a locally optimal choice is made.

A greedy algorithm is therefore one in which no overall strategy is followed, but you simply do whatever looks best at the moment.

For example a mountain climber using the greedy strategy to climb Everest would at every step climb in the steepest direction. From this analogy we get the computational search technique known as *hill-climbing*.

In general greedy methods have limited use, but fortunately, the problem of finding a minimum spanning tree can be solved by a greedy method.

Kruskal's method

Kruskal invented the following very simple method for building a minimum spanning tree. It is based on building a forest of lowest possible weight and continuing to add edges until it becomes a spanning tree.

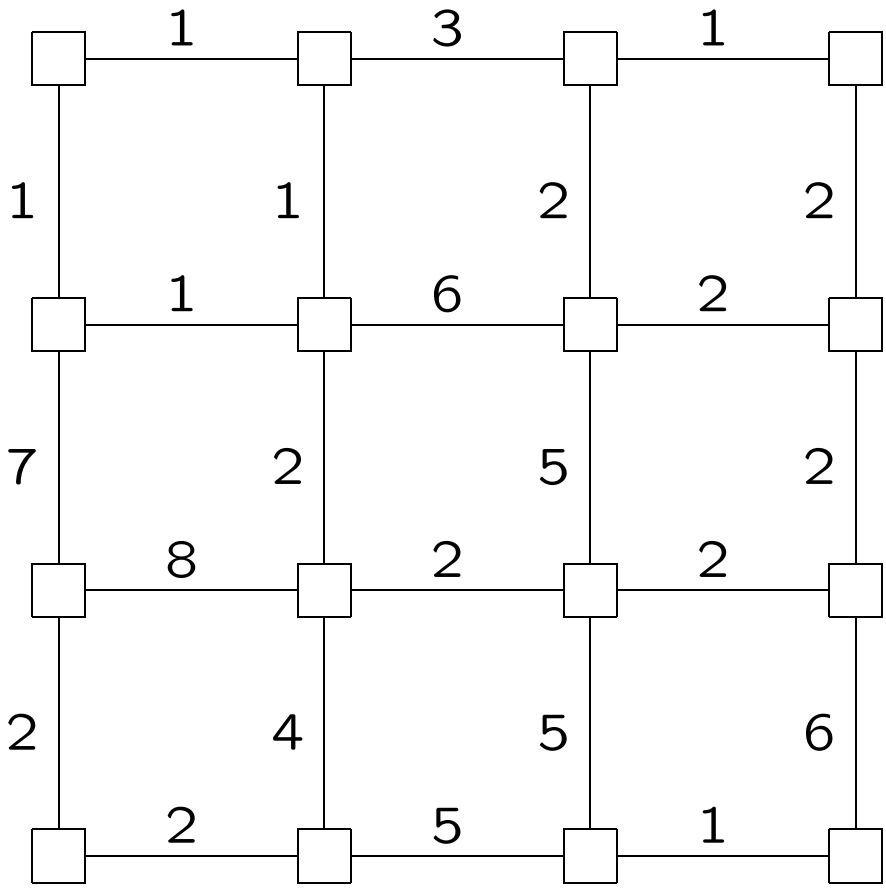
Kruskal's method

Initialize F to be the forest with all the vertices of G but none of the edges.

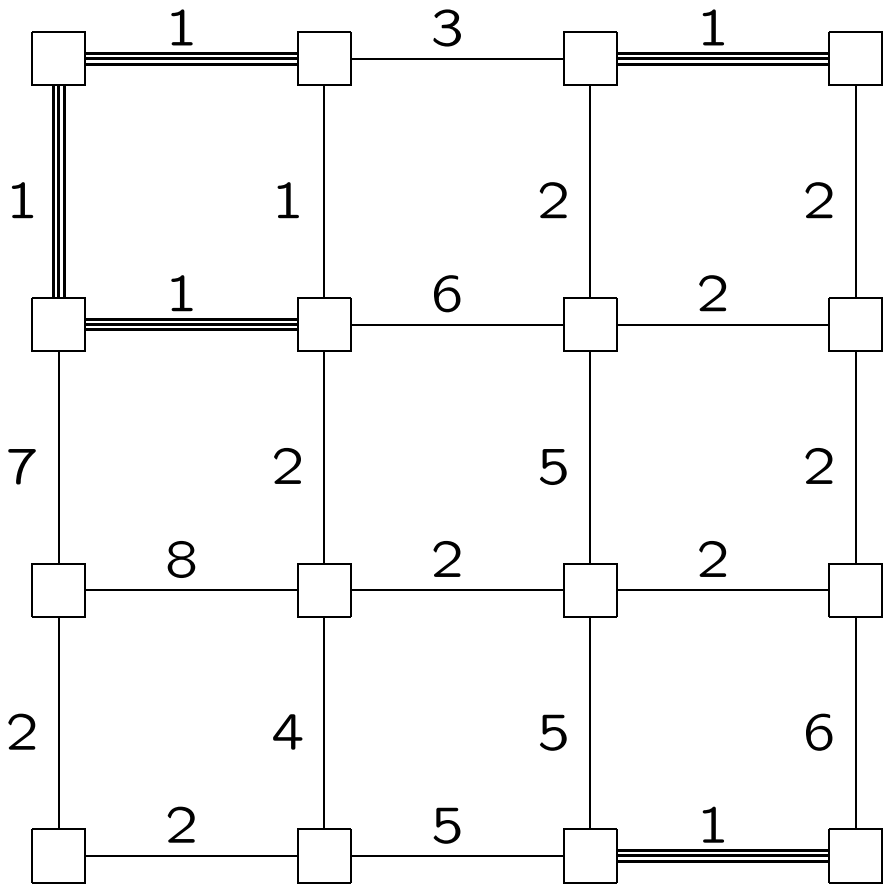
repeat
Pick an edge e of minimum possible weight
if $F \cup \{e\}$ is a forest **then**
$F \leftarrow F \cup \{e\}$
end if
until F contains $n - 1$ edges

Therefore we just keep on picking the smallest possible edge, and adding it to the forest, providing that we never create a cycle along the way.

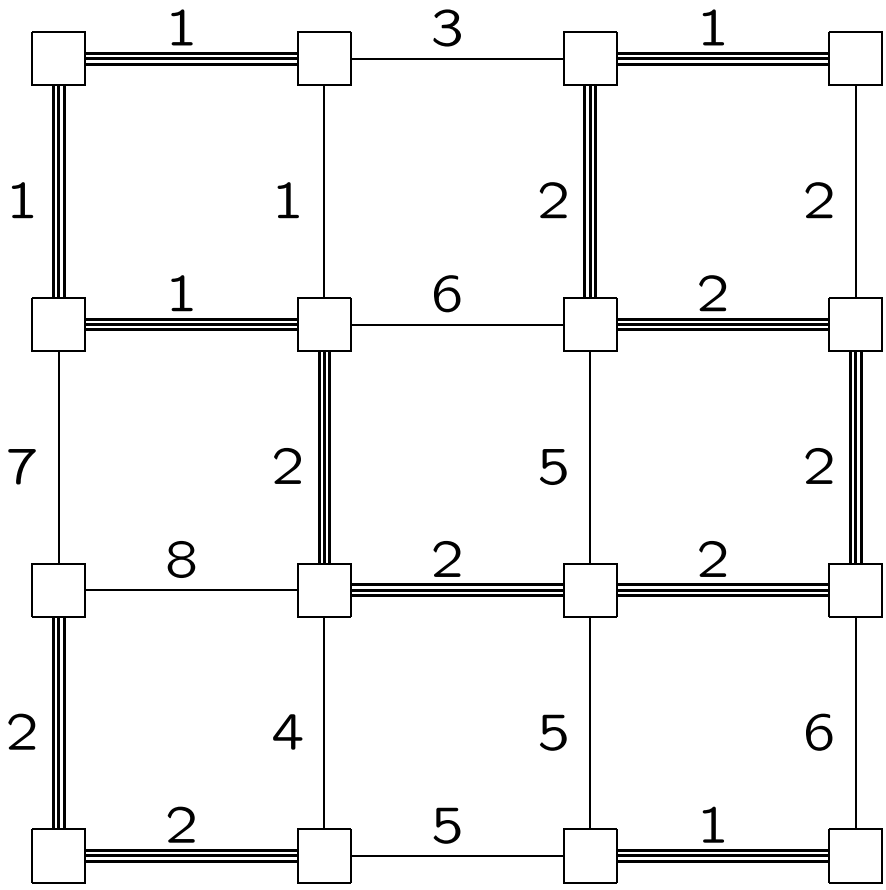
Example



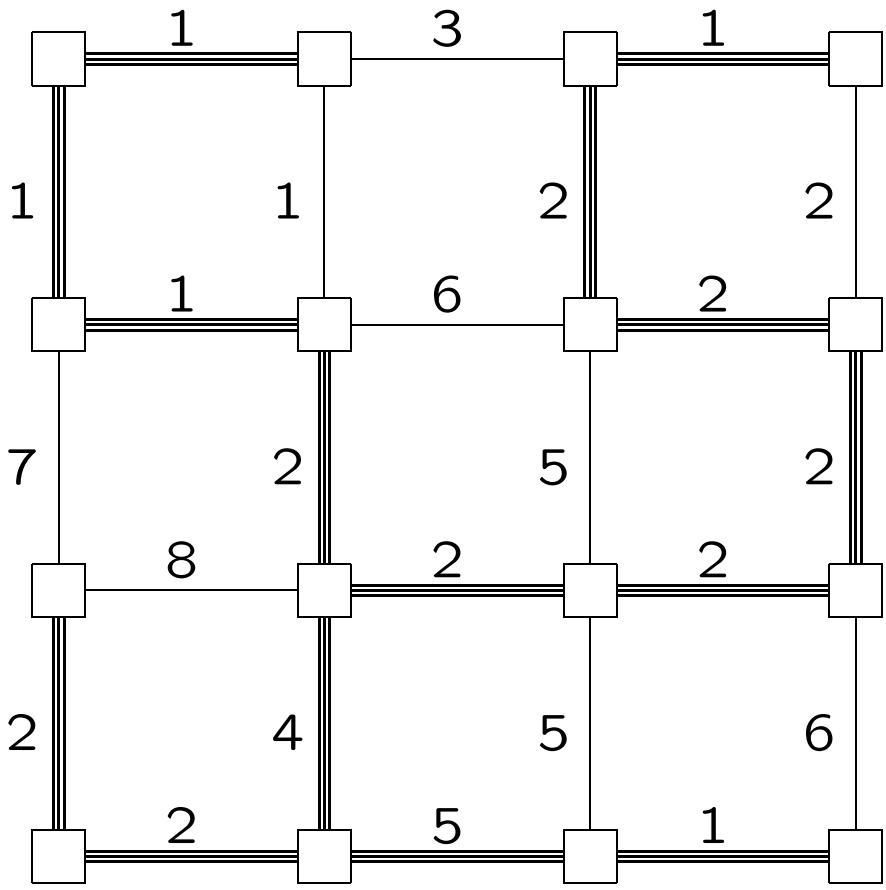
After using edges of weight 1



After using edges of weight 2



The final MST



Prim's algorithm

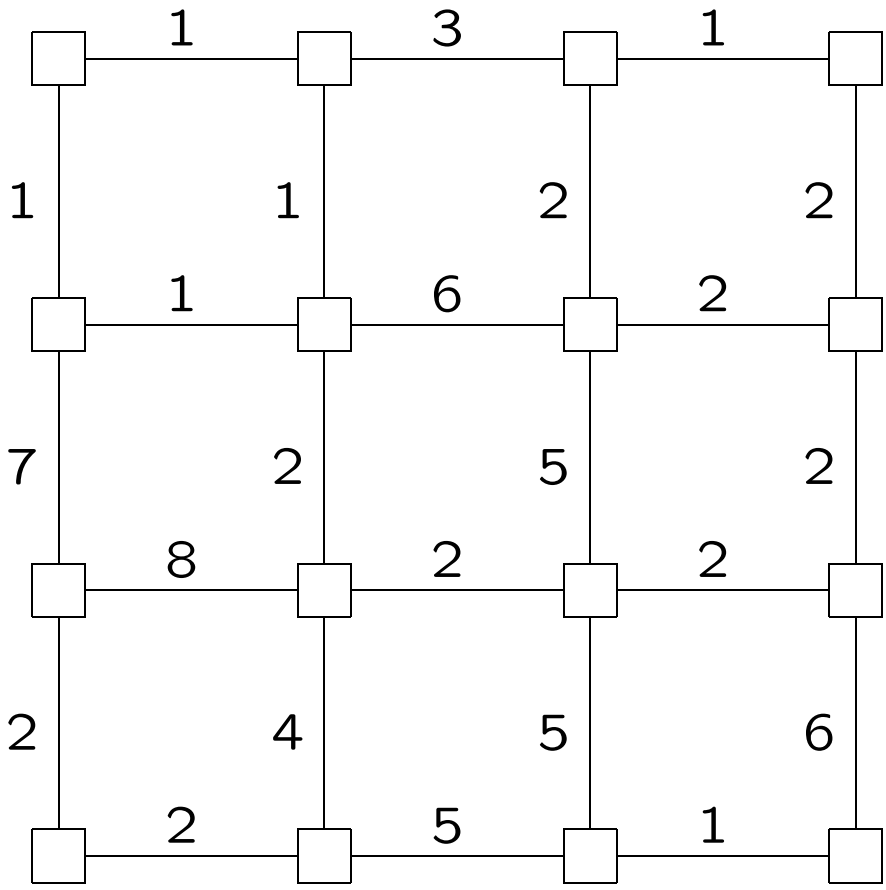
Prim's algorithm is another greedy algorithm for finding a minimum spanning tree.

The idea behind Prim's algorithm is to grow a minimum spanning tree edge-by-edge by always adding the shortest edge that touches a vertex in the current tree.

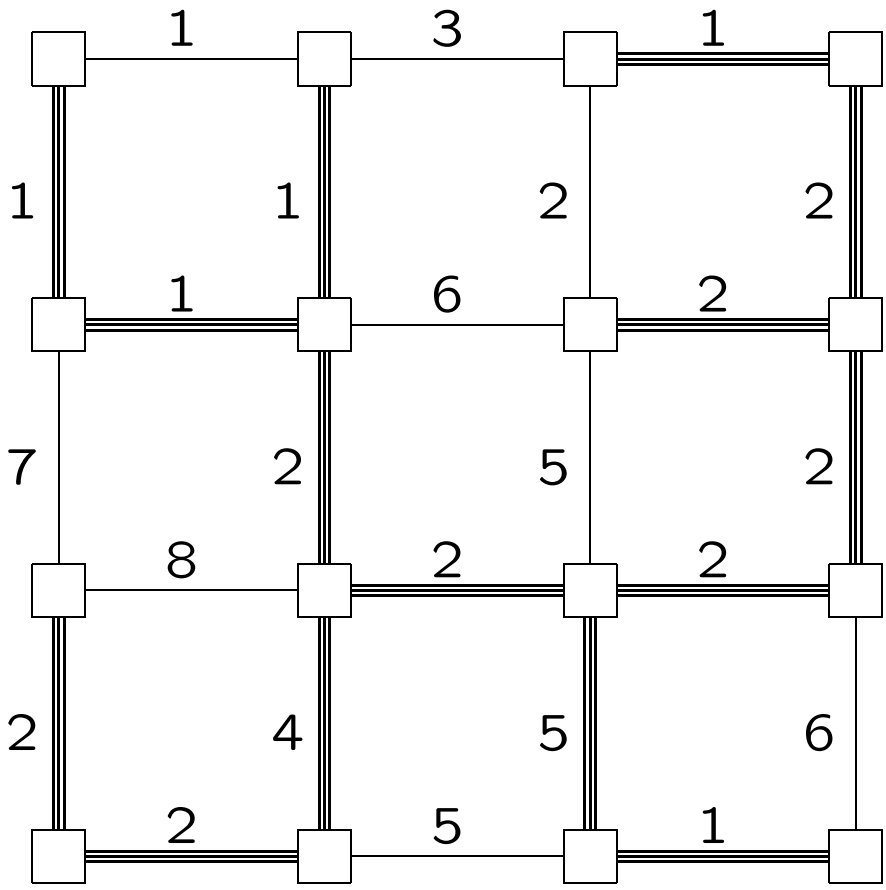
Notice the difference between the algorithms: Kruskal's algorithm always maintains a spanning subgraph which only becomes a tree at the final stage.

On the other hand, Prim's algorithm always maintains a tree which only becomes spanning at the final stage.

Prim's algorithm in action



One solution



Problem solved?

As far as a mathematician is concerned the problem of a minimum spanning tree is well-solved. We have two simple algorithms both of which are guaranteed to find the best solution. (After all, a greedy algorithm must be one of the simplest possible).

In fact, the reason why the greedy algorithm works in this case is well understood — the collection of all the subsets of the edges of a graph that do not contain a cycle forms what is called a (graphic) **matroid**.

Loosely speaking, a greedy algorithm always works on a matroid and never works otherwise.

Implementation issues

In fact the problem is far from solved because we have to decide how to *implement* the two greedy algorithms.

The details of the implementation of the two algorithms are interesting because they use (and illustrate) two important data structures — the *partition* and the *priority queue*.

Implementation of Kruskal

The main problem in the implementation of Kruskal is to decide whether the next edge to be added is allowable — that is, does it create a cycle or not.

Suppose that at some stage in the algorithm the next shortest edge is $\{x, y\}$. Then there are two possibilities:

x and y lie in different trees of F: In this case adding the edge does not create any new cycles, but merges together two of the trees of F

x and y lie in the same tree of F: In this case adding the edge creates a cycle and the edge should not be added to F

Therefore we need data structures that allow us to quickly find the tree to which an element belongs and quickly merge two trees.

Partitions or disjoint sets

The appropriate data structure for this problem is the *partition* (sometimes known under the name disjoint sets). Recall that a partition of a set is a collection of disjoint subsets (called cells) that cover the entire set.

At the beginning of Kruskal's algorithm we have a partition of the vertices into the discrete partition where each cell has size 1. As each edge is added to the minimum spanning tree the number of cells in the partition decreases by one.

The operations that we need for Kruskal's algorithm are

Union(cell,cell) Creates a new partition by merging two cells

Find(element) Finds the cell containing a given element

Naive partitions

One simple way to represent a partition is simply to choose one element of each cell to be the "leader" of that cell. Then we can simply keep an array π of length n where $\pi(x)$ is the leader of the cell containing x.

Example Consider the partition of 8 elements into 3 cells as follows:

$$\{0, 2 \mid 1, 3, 5 \mid 4, 6, 7\}$$

We could represent this as an array as follows

x	0	1	2	3	4	5	6	7
$\pi(x)$	0	1	0	1	4	1	4	4

Then certainly the operation **Find** is straightforward — we can decide whether x and y are in the same cell just by comparing $\pi(x)$ with $\pi(y)$.

Thus **Find** has complexity $\Theta(1)$.

Updating the partition

Suppose now that we wish to update the partition by merging the first two cells to obtain the partition

$$\{0, 1, 2, 3, 5 \mid 4, 6, 7\}$$

We could update the data structure by running through the entire array π and updating it as necessary.

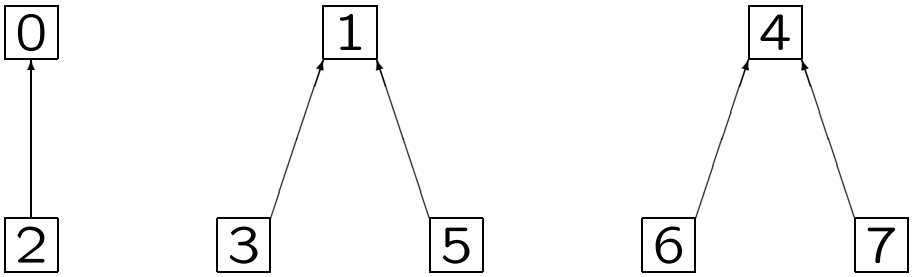
x	0	1	2	3	4	5	6	7
$\pi(x)$	0	0	0	0	4	0	4	4

This takes time $\Theta(n)$, and hence the merging operation is rather slow.

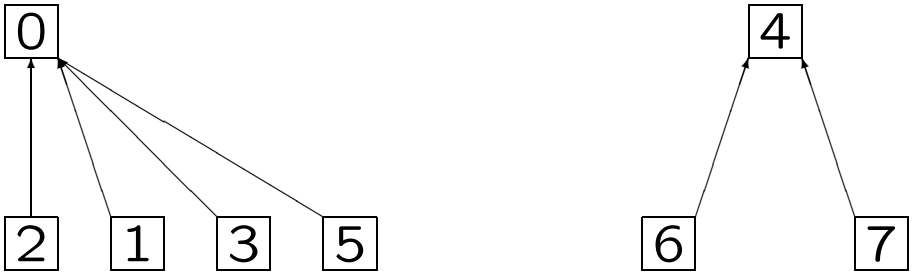
Can we improve the time of the merging operation?

The disjoint sets forest

Consider the following graphical representation of the data structure above, where each element points (upwards) to the "leader" of the cell.



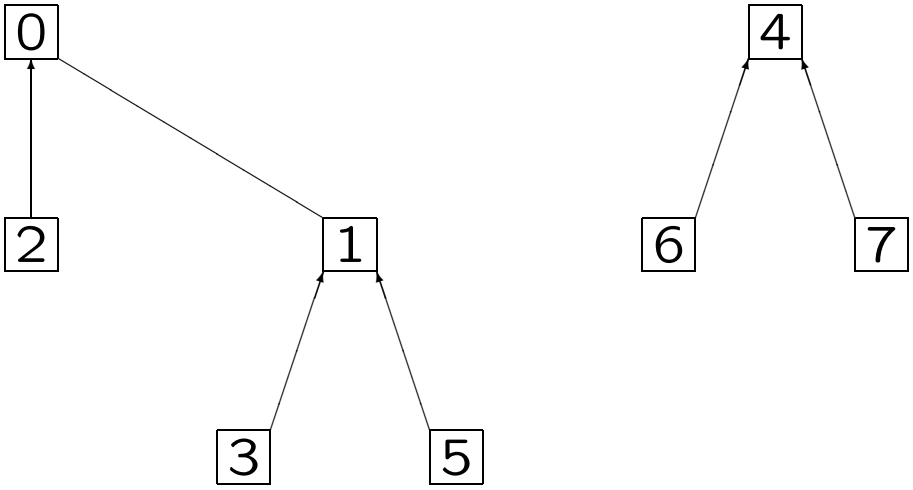
Merging two cells is accomplished by adjusting the pointers so they point to the new leader.



However we can achieve something similar by just adjusting one pointer — suppose we simply change the pointer for the element 1, by making it point to 0 instead of itself.

The new data structure

Just adjusting this one pointer results in the following data structure.



This new improved merging has complexity only $\Theta(1)$. However we have now lost the ability to do the **Find** properly. In order to correctly find the leader of the cell containing an element we have to run through a little loop:

Find(x)
while $x \neq \pi(x)$
$x = \pi(x)$

This new find operation may take time $O(n)$ so we seem to have gained nothing.

Union-by-rank heuristic

There are two heuristics that can be applied to the new data structure, that speed things up enormously at the cost of maintaining a little extra data.

Let the *rank* of a root node of a tree be the height of that tree (the maximum distance from a leaf to the root).

The *union-by-rank* heuristic tries to keep the trees *balanced* at all times. When a merging operation needs to be done, the root of the shorter tree is made to point to the root of the taller tree. The resulting tree therefore does not increase its height unless both trees are the same height in which case the height increases by one.

Path compression heuristic

The path compression heuristic is based on the idea that when we perform a **Find(x)** operation we have to follow a path from x to the root of the tree containing x.

After we have done this why do we not simply go back down through this path and make all these elements point *directly* to the root of the tree, rather than in a long chain through each other?

This is reminiscent of our naive algorithm, where we made *every* element point directly to the leader of its cell, but it is much cheaper because we only alter things that we needed to look at anyway.

Complexity of Kruskal

In the worst case, we will perform E operations on the partition data structure which has size V. By the complicated argument in CLR we see that the total time for these operations if we use both heuristics is $O(E \lg^* V)$.

However we must add to this the time that is needed to sort the edges — because we have to examine the edges in order of length. This time is $O(E \lg E)$ if we use a sorting technique such as quicksort, and hence the overall complexity of Kruskal's algorithm is $O(E \lg E)$.

Implementation of Prim

For Prim's algorithm we repeatedly have to select the next vertex that is *closest* to the tree that we have built so far. Therefore we need some sort of data structure that will enable us to associate a value with each vertex (being the distance to the tree under construction) and rapidly select the vertex with the lowest value.

From our study of Data Structures we know that the appropriate data structure is a *priority queue* and that a priority queue is implemented by using a *heap*.

The priority queue ADT

Recall that a *priority queue* is an abstract data type that stores objects with an associated key. The priority of the object depends on the value of the key.

The operations associated with this data type include

insert(entry,key) Places an entry with its associated key into the data structure

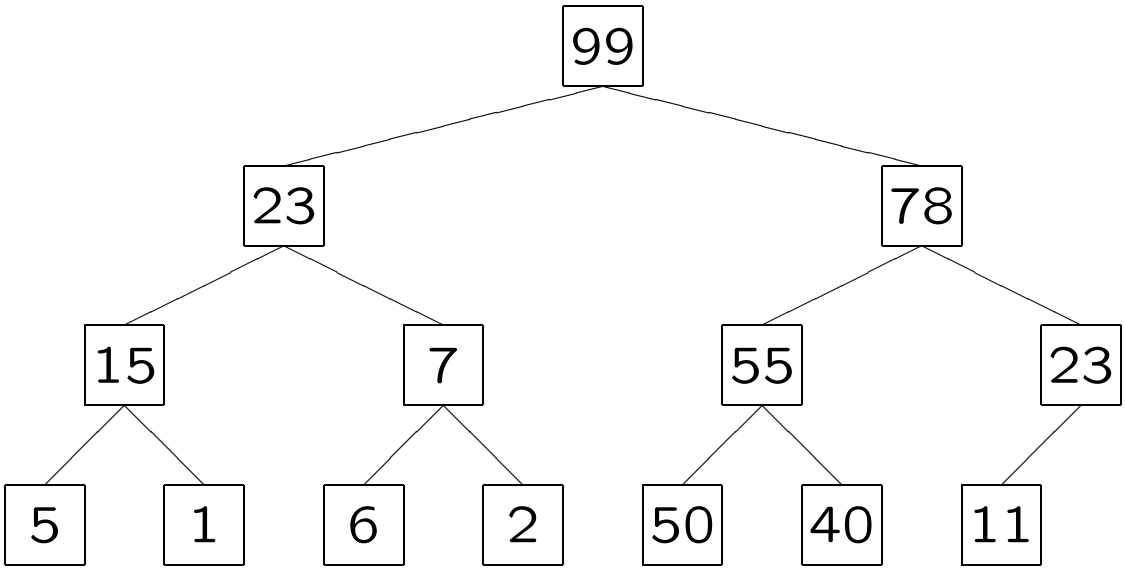
change(entry,newkey) Changes the value of the key associated with a given entry

max(queue) Returns the element with the highest priority

extractmax(queue) Returns the element with the highest priority from the queue and deletes it from the queue

Heaps

Recall that a *heap* is a binary tree such that the key associated with any node is larger than (or equal to) the key associated with either of its children. This means that the root of the binary tree has the largest key.



We would actually *store* the heap as a linear array:

99	23	78	15	7	55	23	5	1	6	2	50	40	11

Notice that if the bottom level of the binary tree is not complete then it is filled from the left.

Parents and Children

Suppose the array A is indexed from 1. Then $A[1]$ holds the root of the binary tree, which by the heap property is the element with the largest key value.

The two children of the root are stored in $A[2]$ and $A[3]$.

Following CLR, we see that the left hand child of node i is $2i$ and the right hand child of node i is $2i + 1$.

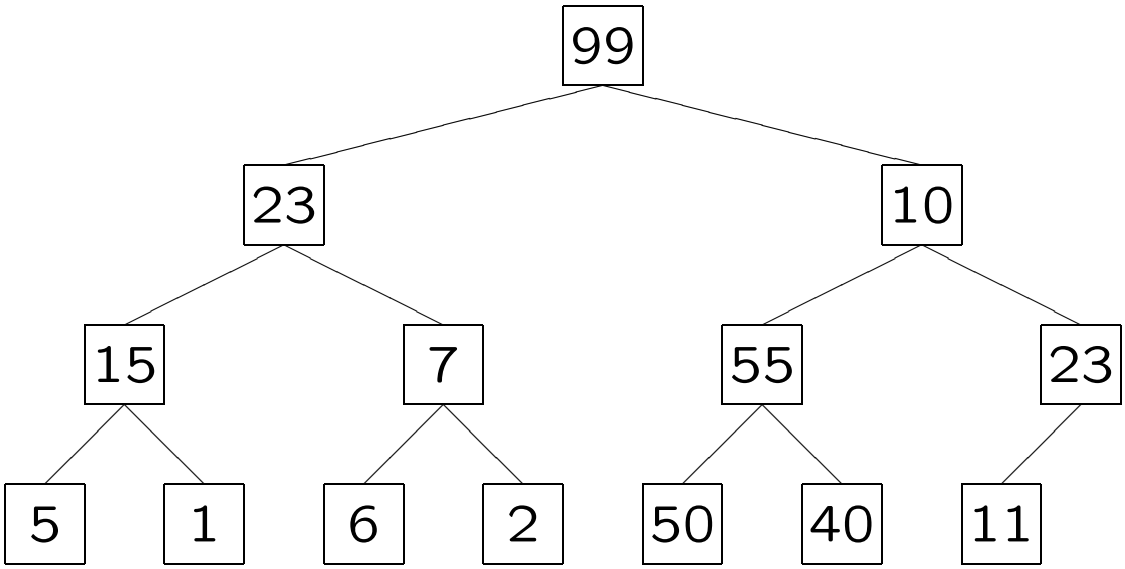
Conversely, the parent of node i is $\lfloor i/2 \rfloor$.

Therefore using the array representation it is easy to access the parents and the children of each node.

Heapify

Most of the operations we wish to perform on the heap will alter the data structure. Often these operations will destroy the heap property and it will have to be restored.

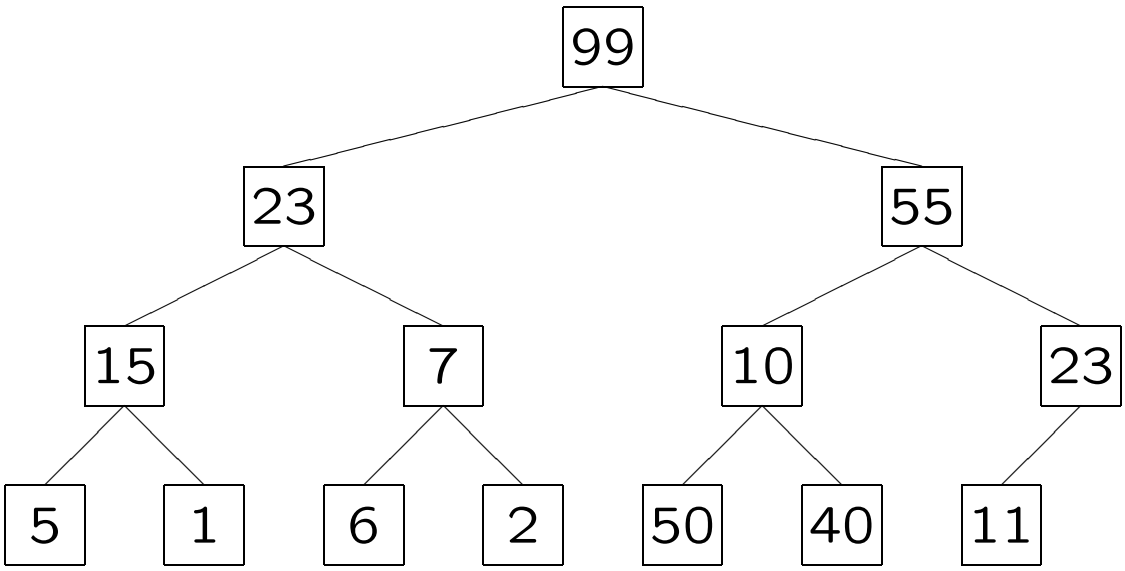
For example, suppose we decide to alter the value associated with one of the nodes — for example we wish to set $A[3] = 10$.



The change is easily accomplished, but the resulting structure is no longer a heap - the entry $A[3]$ is no longer larger than both of its children.

Heapify continued

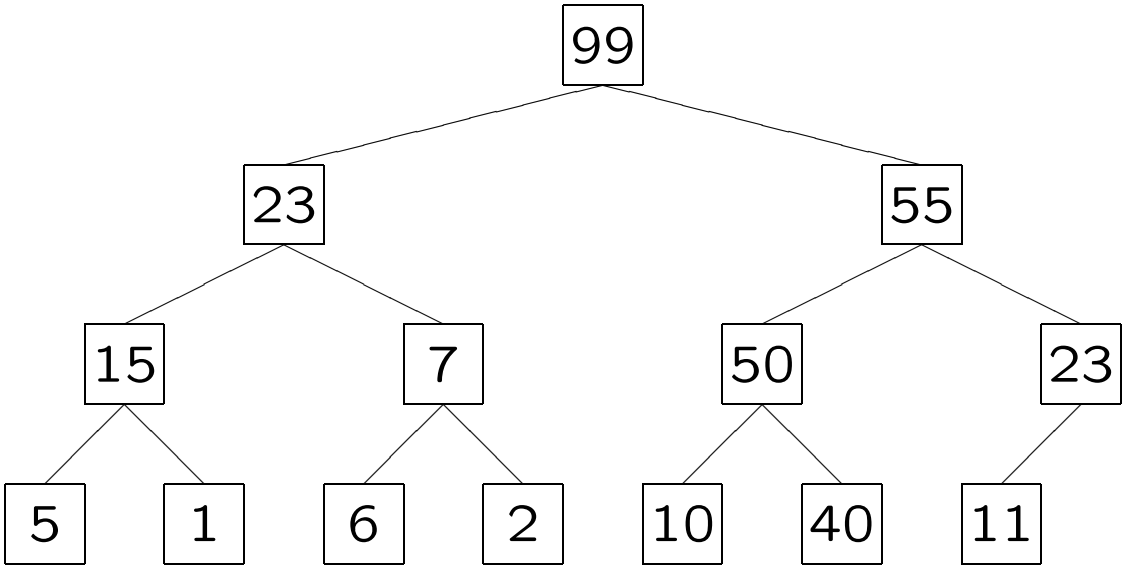
The problem is that $A[3]$ is smaller than its children — we can fix this by swapping $A[3]$ with the *larger* of its two children.



This means that the problem at $A[3]$ has been fixed — however we *may* have introduced a problem at $A[6]$.

Heapify continued

So we now examine $A[6]$ and if it is smaller than its children we perform another exchange.



The procedure outlined above, whereby a small element "percolates" down the tree is called *heapify*.

Heapify takes a position i in the tree as an argument, it assumes that the two children of i are proper heaps, but that maybe the key value $A[i]$ is not larger than the key values of its children.

It then "runs down" the tree swapping entries if necessary until the heap property is restored.

Complexity of heapify

A binary tree with n elements in it has a height of $\lg n$ and hence *heapify* performs at most $\lg n$ exchanges. Therefore it has a complexity of $O(\lg n)$.

Now consider how all the operations necessary for a priority queue can be accomplished by using a heap together with *heapify*.

insert(entry,key) The key is entered at the end of the array — that is, in the last position in the tree. The resulting structure may not be a heap, because the value of the new key may be greater than its parent. If this is the case, then exchange the two keys and proceed to examine the parent.

In the worst case $\lg n$ exchanges will have to be done, hence **insert** has complexity $O(\lg n)$.

Priority queue operations

change(entry,newkey) The appropriate key is changed. If its value is reduced we call *heapify* to restore the heap property, and if the value is increased then we simply compare it with its parent and exchange if the new value is greater than that of the parent (and repeat if necessary). Either operation has a complexity of $O(\lg n)$.

max(queue) The main feature of a heap is that the root of the binary tree is the entry with the largest key value. Hence **max** merely returns the root of the tree, taking constant time $\Theta(1)$.

extractmax(queue) In this case we must also delete the root from the tree, and then restore the heap property. This can be achieved by moving the final entry in the tree to the newly-vacated root position and then calling *heapify(1)* to restore the heap property. This involves only a few constant time operations, together with one call to *heapify* so has complexity $O(\lg n)$.

Prim's algorithm

It is now easy to see how to implement Prim's algorithm. We first initialize the priority queue to contain every vertex with equal priority ∞. Then a single vertex s is chosen and its priority is changed to 0.

Here we want low key values to represent high priorities, so we will rename our priority queue operation to **extractmin**.

At each stage of the algorithm we extract the vertex u with the highest priority (that is, the lowest key value!). We then examine the neighbours of u. For each neighbour v, there are two possibilities:

(1) If v is not in the queue, then it is already in the spanning tree being constructed and we do not consider it further.

(2) If v is currently on the priority queue, then we see whether this new edge (u, v) should cause an update in the priority of v. If the value $weight(u, v)$ is lower than the current key value, then we change the key value of v to $weight(u, v)$ and set $\pi(v) = u$.

Complexity of Prim

The complexity of Prim's algorithm is dominated by the heap operations.

Every vertex is extracted from the priority queue at some stage, hence the **extractmin** operations in the worst case take time $O(V \lg V)$.

Also, every edge is examined at some stage in the algorithm and each edge examination potentially causes a **change** operation. Hence in the worst case these operations take time $O(E \lg V)$.

Therefore the total time is

$$O(V \lg V + E \lg V) = O(E \lg V)$$

Priority-first search

Let us generalize the ideas behind this implementation of Prim's algorithm.

Consider the following very general graph-searching algorithm. We will later show that by choosing different specifications of the priority we can make this algorithm do very different things. This algorithm will produce a *priority-first search tree*.

The key-values or priorities associated with each vertex are stored in an array called key.

Initially we set $key[v]$ to ∞ for all the vertices $v \in V(G)$ and build a heap with these keys — this can be done in time $O(V)$.

Then we select the source vertex s for the search and perform **change(**s**,**0**)** to change the key of s to 0, thus placing s at the top of the priority queue.

The operation of PFS

After initialization the operation of PFS is as follows:

while $Q \neq \emptyset$
    $u \leftarrow$ **extractmin(**Q**)**
    for each v adjacent to u **do**
        if $v \in Q$ and *PRIORITY* $< key[v]$
            $\pi[v] \leftarrow u$
            change(v,*PRIORITY*)
        end if
    end for
end while

It is important to notice how the array π is managed — for every vertex $v \in Q$ with a finite key value, $\pi[v]$ is the vertex *not in* Q that was responsible for the key of v reaching the highest priority it has currently reached.

Complexity of PFS

The complexity of this search is easy to calculate — the main loop is executed V times, and each **extractmin** operation takes $O(\lg V)$ yielding a total time of $O(V \lg V)$ for the extraction operations.

During all V operations of the main loop we examine the adjacency list of each vertex exactly once — hence we make E calls, each of which may cause a **change** to be performed. Hence we do at most $O(E \lg V)$ work on these operations.

Therefore the total is

$$O(V \lg V + E \lg V) = O(E \lg V).$$

Prim's algorithms is PFS

Prim's algorithm can be expressed as a priority-first search by observing that the priority of a vertex is the weight of the shortest edge joining the vertex to the rest of the tree.

This is achieved in the code above by simply replacing the string *PRIORITY* by

$$weight(u, v)$$

At any stage of the algorithm:

• The vertices not in Q form the tree so far.

• For each vertex $v \in Q$, $key[v]$ gives the length of the shortest edge from v to a vertex in the tree, and $\pi[v]$ shows which tree vertex that is.

Shortest paths

Let G be a directed weighted graph. The *shortest path* between two vertices v and w is the path from v to w for which the sum of the weights on the path-edges is lowest. Notice that if we take an unweighted graph to be a special instance of a weighted graph, but with all edge weights equal to 1, then this coincides with the normal definition of shortest path.

The weight of the shortest path from v to w is denoted by $\delta(v, w)$.

Let $s \in V(G)$ be a specified vertex called the *source* vertex.

The *single-source shortest paths* problem is to find the shortest path from s to every other vertex in the graph (as opposed to the *all-pairs shortest paths problem*, where we must find the distance between every pair of vertices).

Dijkstra's algorithm

Dijkstra's algorithm is a famous single-source shortest paths algorithm suitable for the cases when the weights are all non-negative.

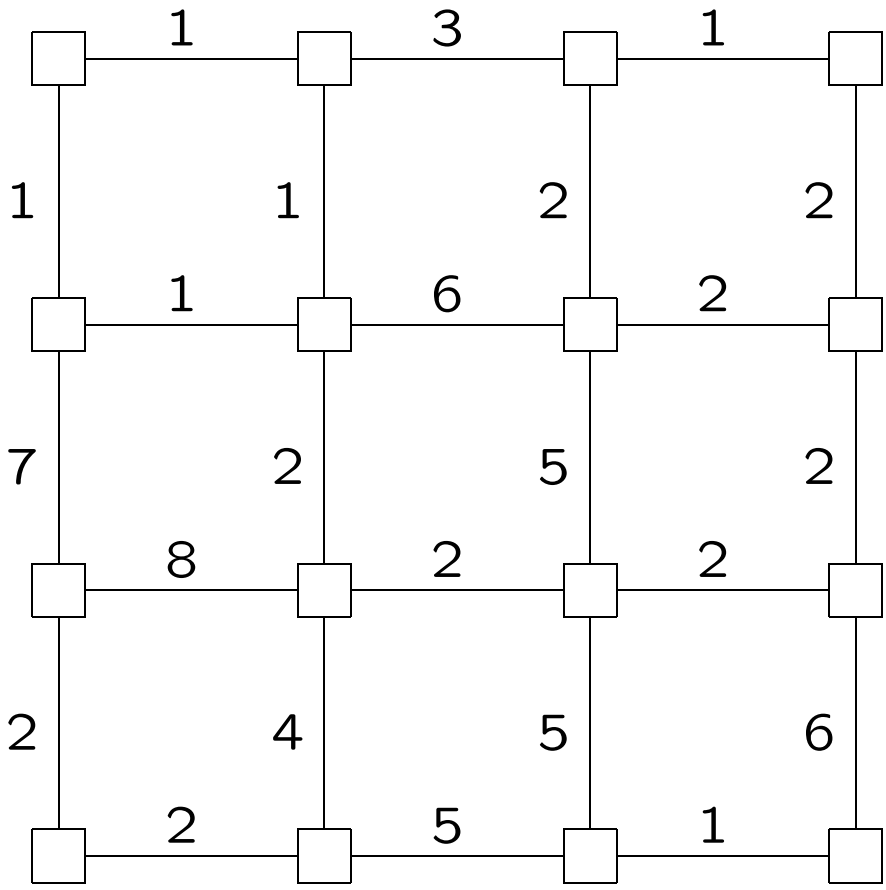
Dijkstra's algorithm can be implemented as a priority-first search by taking the priority of a vertex $v \in Q$ to be the shortest path from s to v that consists entirely of vertices in the priority-first search tree (except of course for v).

This can be implemented as a PFS by replacing PRIORITY with

$$key[u] + weight(u, v)$$

At the end of the search, the array key[] contains the lengths of the shortest paths from the source vertex s.

Dijkstra's algorithm in action

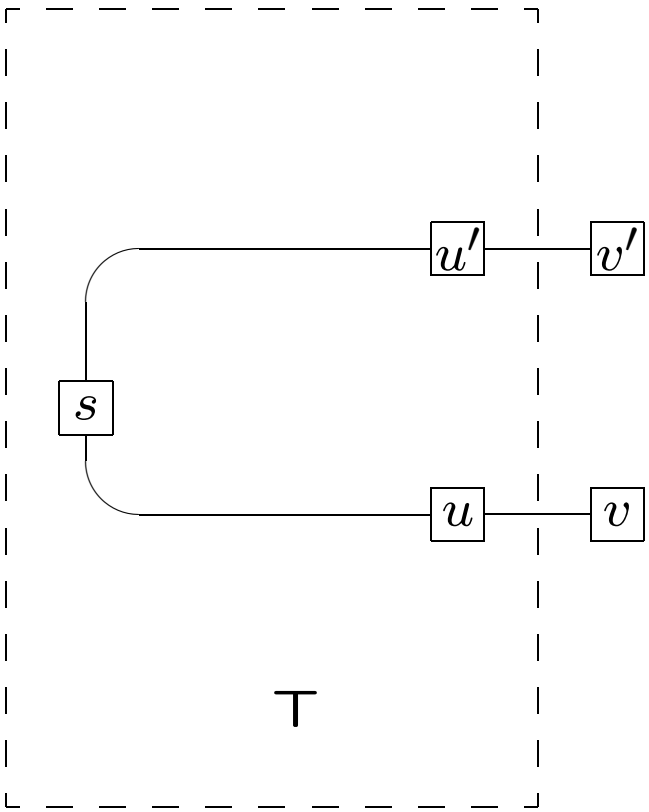


Proof of correctness

It is fairly easy to prove that Dijkstra's algorithm is correct by proving the following claim (where $T = V(G) - Q$).

- At the time that a vertex v is removed from Q and placed into T $key[v] = \delta(s, v)$.

To prove this claim we consider the moment at which v is removed from Q.



Proof

Suppose v is removed from Q and that the parent of v is u. Then

$$key[v] = \delta(s, u) + weight(u, v)$$

If this value is not the true value of $\delta(s, v)$ then there must be a shorter path from s to v. This shorter path must leave T at some stage, say the edge (u', v'). Then consider the value $key[v']$ — because the edge (u', v') was examined when u' was added to T, the value of this key is at most $\delta(s, u') + weight(u', v')$. This value however is less than the length of the shortest path from s to v and hence is less than the key for v, which is impossible.

Therefore this situation cannot occur and we conclude that there cannot be any shorter paths from s to v and the result holds.

Relaxation

Consider the following property of Dijkstra's algorithm.

• At any stage of Dijkstra's algorithm the following inequality holds:

$$\delta(s, v) \leq key[v]$$

This is saying that the *key[]* array always holds a collection of *upper bounds* on the actual values that we are seeking. We can view these values as being our "best estimate" to the value so far, and Dijkstra's algorithm as a procedure for systematically improving our estimates to the correct values.

The fundamental step in Dijkstra's algorithm, where the bounds are altered is when we examine the edge (u, v) and do the following operation

$$key[v] \leftarrow \mathbf{min}(key[v], key[u] + weight(u, v))$$

This is called *relaxing* the edge (u, v).

Relaxation schedules

Consider now an algorithm that is of the following general form:

- Initially an array $d[]$ is initialized to have $d[s] = 0$ for some source vertex s and $d[v] = \infty$ for all other vertices
- A sequence of edge relaxations is performed, possibly altering the values in the array $d[]$.

We observe that the value $d[v]$ is always an upper bound for the value $\delta(s, v)$ because relaxing the edge (u, v) will either leave the upper bound unchanged or replace it by a better estimate from an upper bound on a path from $s \rightarrow u \rightarrow v$.

Dijkstra's algorithm is a particular schedule for performing the edge relaxations that guarantees that the upper bounds converge to the exact values.

Negative edge weights

Dijkstra's algorithm cannot be used when the graph has some negative edge-weights (why not? find an example).

In general, no algorithm for shortest paths can work if the graph contains a cycle of negative total weight (because a path could be made arbitrarily short by going round and round the cycle). Therefore the question of finding shortest paths makes no sense if there is a negative cycle.

However, what if there are some negative edge weights but no negative cycles?

The Bellman-Ford algorithm is a relaxation schedule that can be run on graphs with negative edge weights. It will either *fail* in which case the graph has a negative cycle and the problem is ill-posed, or will finish with the single-source shortest paths in the array d[].

Bellman-Ford algorithm

The initialization step is as described above. Let us suppose that the weights on the edges are given by the function w.

Then consider the following relaxation schedule:

for $i = 1$ **to** $|V(G)| - 1$ **do**
    for each edge $(u, v) \in E(G)$ **do**
        RELAX(u,v)
    end for each
end for

Finally we make a single check to determine if we have a failure:

for each edge $(u, v) \in E(G)$ **do**
    if $d[v] > d[u] + w(u, v)$ **then** FAIL
    end if
end for each

Complexity of Bellman-Ford

The complexity is particularly easy to calculate in this case because we know exactly how many relaxations are done — namely $E(V-1)$, and adding that to the final failure check loop, and the initialization loop we see that Bellman-Ford is $O(EV)$

There remains just one question — how does it work?

To answer this, let us consider some of the properties of relaxation in a graph with no negative cycles.

Property 1 Consider an edge (u, v) that lies on the shortest path from s to v. If the sequence of relaxations includes relaxing (u, v) at a stage when $d[u] = \delta(s, u)$, then $d[v]$ is set to $\delta(s, v)$ and never changes after that.

Correctness of Bellman-Ford

Once convinced that Property 1 holds it is now simple to see that the algorithm is correct for graphs with no negative cycles.

Consider any vertex v and let us examine the shortest path from s to v, namely

$$s \sim v_1 \sim v_2 \cdots v_k \sim v$$

Now at the initialization stage $d[s] = 0$ and it always remains the same. After one pass through the main loop the edge (s, v_1) is relaxed and by Property 1, $d[v_1] = \delta(s, v_1)$ and it remains at that value. After the second pass the edge (v_1, v_2) is relaxed and after this relaxation we have $d[v_2] = \delta(s, v_2)$ and it remains at this value.

As the number of edges in the path is at most $|V(G)| - 1$, after all the loops have been performed $d[v] = \delta(s, v)$.

All-pairs shortest paths

Now we turn our attention to constructing a complete table of shortest distances, which must contain the shortest distance between any pair of vertices.

If the graph has no negative edge weights then we could simply make V runs of Dijkstra's algorithm, at a total cost of $O(VE \lg V)$, whereas if there are negative edge weights then we could make V runs of the Bellman-Ford algorithm at a total cost of $O(V^2E)$.

The two algorithms we shall examine both use the adjacency matrix representation of the graph, hence are most suitable for dense graphs. Recall that for a weighted graph the weighted adjacency matrix A has $weight(i, j)$ as its ij-entry, where $weight(i, j) = \infty$ if i and j are not adjacent.

A dynamic programming method

Dynamic programming is a general algorithmic technique for solving problems that can be characterised by two features:

- The problem is broken down into a collection of smaller subproblems
- The solution is built up from the stored values of the solutions to all of the subproblems

For the all-pairs shortest paths problem we define the simpler problem to be

"What is the length of the shortest path from vertex i to j that uses at most m edges?"

We shall solve this for $m = 1$, then use that solution to solve for $m = 2$, and so on ...

The initial step

We shall let $d_{ij}^{(m)}$ denote the distance from vertex i to vertex j along a path that uses at most m edges, and define $D^{(m)}$ to be the matrix whose ij-entry is the value $d_{ij}^{(m)}$.

As a shortest path between any two vertices can contain at most $V - 1$ edges, the matrix $D^{(V-1)}$ contains the table of all-pairs shortest paths.

Our overall plan therefore is to use $D^{(1)}$ to compute $D^{(2)}$, then use $D^{(2)}$ to compute $D^{(3)}$ and so on.

The case $m = 1$

Now the matrix $D^{(1)}$ is easy to compute — the length of a shortest path using at most one edge from i to j is simply the weight of the edge from i to j. Therefore $D^{(1)}$ is just the adjacency matrix A.

The inductive step

What is the smallest weight of the path from vertex i to vertex j that uses at most m edges? Now a path using at most m edges either be

(1) A path using less than m edges
(2) A path using exactly m edges, composed of a path using $m-1$ edges from i to an auxiliary vertex k and the edge (k, j).

We shall take the entry $d_{ij}^{(m)}$ to be the lowest weight path from the above choices.

Therefore we get

$$d_{ij}^{(m)} = \min\left(d_{ij}^{(m-1)}, \min_{1 \le k \le V}\{d_{ik}^{(m-1)} + w(k,j)\}\right)$$

$$= \min_{1 \le k \le V}\{d_{ik}^{(m-1)} + w(k,j)\}$$

Example

Consider the weighted graph with the following weighted adjacency matrix:

$$A = D^{(1)} = \begin{pmatrix} 0 & \infty & 11 & 2 & 6 \\ 1 & 0 & 4 & \infty & \infty \\ 10 & \infty & 0 & \infty & \infty \\ \infty & 2 & 6 & 0 & 3 \\ \infty & \infty & 6 & \infty & 0 \end{pmatrix}$$

Let us see how to compute an entry in $D^{(2)}$, suppose we are interested in the $(1,3)$ entry:

Then we see that

$1 \to 1 \to 3$ has cost $0 + 11 = 11$
$1 \to 2 \to 3$ has cost $\infty + 4 = \infty$
$1 \to 3 \to 3$ has cost $11 + 0 = 11$
$1 \to 4 \to 3$ has cost $2 + 6 = 8$
$1 \to 5 \to 3$ has cost $6 + 6 = 12$

The minimum of all of these is 8, hence the $(1,3)$ entry of $D^{(2)}$ is set to 8.

Computing $D^{(2)}$

$$\begin{pmatrix} 0 & \infty & 11 & 2 & 6 \\ 1 & 0 & 4 & \infty & \infty \\ 10 & \infty & 0 & \infty & \infty \\ \infty & 2 & 6 & 0 & 3 \\ \infty & \infty & 6 & \infty & 0 \end{pmatrix} \begin{pmatrix} 0 & \infty & 11 & 2 & 6 \\ 1 & 0 & 4 & \infty & \infty \\ 10 & \infty & 0 & \infty & \infty \\ \infty & 2 & 6 & 0 & 3 \\ \infty & \infty & 6 & \infty & 0 \end{pmatrix}$$

$$= \begin{pmatrix} 0 & 4 & 8 & 2 & 5 \\ 1 & 0 & 4 & 3 & 7 \\ 10 & \infty & 0 & 12 & 16 \\ 3 & 2 & 6 & 0 & 3 \\ 16 & \infty & 6 & \infty & 0 \end{pmatrix}$$

If we multiply two matrices $AB = C$, then we compute

$$c_{ij} = \sum_{k=1}^{k=V} a_{ik} b_{kj}$$

If we replace the multiplication $a_{ik} b_{kj}$ by addition $a_{ik} + b_{kj}$ and replace summation Σ by the minimum min then we get

$$c_{ij} = \min_{k=1}^{k=V} a_{ik} + b_{kj}$$

which is precisely the operation we are performing to calculate our matrices.

The remaining matrices

Proceeding to compute $D^{(3)}$ from $D^{(2)}$ and A, and then $D^{(4)}$ from $D^{(3)}$ and A we get:

$$D^{(3)} = \begin{pmatrix} 0 & 4 & 8 & 2 & 5 \\ 1 & 0 & 4 & 3 & \boxed{6} \\ 10 & \boxed{14} & 0 & 12 & \boxed{15} \\ 3 & 2 & 6 & 0 & 3 \\ 16 & \infty & 6 & \boxed{18} & 0 \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 4 & 8 & 2 & 5 \\ 1 & 0 & 4 & 3 & 6 \\ 10 & 14 & 0 & 12 & 15 \\ 3 & 2 & 6 & 0 & 3 \\ 16 & \boxed{20} & 6 & 18 & 0 \end{pmatrix}$$

A new matrix "product"

Recall the method for computing $d_{ij}^{(m)}$, the (i,j) entry of the matrix $D^{(m)}$ using the method similar to matrix multiplication.

$d_{ij}^{(m)} \leftarrow \infty$
for $k = 1$ **to** V **do**
    $d_{ij}^{(m)} = \min(d_{ij}^{(m)}, d_{ik}^{(m-1)} + w(k,j))$
end for

Let us give this new matrix product the name $\star$.

Then we have

$$D^{(m)} = D^{(m-1)} \star A$$

Hence it is an easy matter to see that we can compute as follows:

$$D^{(2)} = A \star A \qquad D^{(3)} = D^{(2)} \star A \ldots$$

Complexity of this method

The time taken for this method is easily seen to be $\Theta(V^4)$ as it performs V matrix "multiplications" each of which involves a triply nested **for** loop with each variable running from 1 to V.

However we can reduce the complexity of the algorithm by remembering that we do not need to compute *all* the intermediate products $D^{(1)}$, $D^{(2)}$ and so on, but we are only interested in $D^{(V-1)}$. Therefore we can simply compute:

$$D^{(2)} = A \star A$$

$$D^{(4)} = D^{(2)} \star D^{(2)}$$

$$D^{(8)} = D^{(4)} \star D^{(4)}$$

Therefore we only need to do this operation at most $\lg V$ times before we reach the matrix we want.

Floyd-Warshall

The Floyd-Warshall algorithm uses a different dynamic programming formalism.

For this algorithm we shall define $d_{ij}^{(k)}$ to be the length of the shortest path from i to j whose intermediate vertices all lie in the set $\{1, \ldots, k\}$.

As before, we shall define $D^{(k)}$ to be the matrix whose (i, j) entry is $d_{ij}^{(k)}$.

The initial case

What is the matrix $D^{(0)}$ — the entry $d_{ij}^{(0)}$ is the length of the shortest path from i to j with *no* intermediate vertices. Therefore $D^{(0)}$ is simply the adjacency matrix A.

The inductive step

For the inductive step we assume that we have constructed already the matrix $D^{(k-1)}$ and wish to use it to construct the matrix $D^{(k)}$.

Let us consider all the paths from i to j whose intermediate vertices lie in $\{1, 2, \ldots, k\}$. There are two possibilities for such paths

(1) The path does not use vertex k
(2) The path does use vertex k

The shortest possible length of all the paths in category (1) is given by $d_{ij}^{(k-1)}$ which we already know.

If the path does use vertex k then it must go from vertex i to k and then proceed on to j, and the length of the shortest path in this category is $d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$.

The overall algorithm

The overall algorithm is then simply a matter of running V times through a loop, with each entry being assigned as the minimum of two possibilities. Therefore the overall complexity of the algorithm is just $O(V^3)$.

$D^{(0)} \leftarrow A$
for $k = 1$ **to** V **do**
    for $i = 1$ **to** V **do**
        for $j = 1$ **to** V **do**
            $d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$
        end for j
    end for i
end for k

At the end of the procedure we have the matrix $D^{(V)}$ whose (i, j) entry contains the length of the shortest path from i to j, all of whose vertices lie in $\{1, 2, \ldots, V\}$ — in other words, the shortest path in total.

Example

Consider the weighted directed graph with the following adjacency matrix:

$$D^{(0)} = \begin{pmatrix} 0 & \infty & 11 & 2 & 6 \\ 1 & 0 & 4 & \infty & \infty \\ 10 & \infty & 0 & \infty & \infty \\ \infty & 2 & 6 & 0 & 3 \\ \infty & \infty & 6 & \infty & 0 \end{pmatrix}$$

Let us see how to compute $D^{(1)}$

$$D^{(1)} = \begin{pmatrix} 0 & \infty & 11 & 2 & 6 \\ 1 & 0 & 4 & & \\ 10 & \infty & 0 & & \\ \infty & 2 & 6 & 0 & 3 \\ \infty & \infty & 6 & \infty & 0 \end{pmatrix}$$

To find the $(2,4)$ entry of this matrix we have to consider the paths through the vertex 1 — is there a path from $2-1-4$ that has a better value than the current path? If so, then that entry is updated.

The entire sequence of matrices

$$D^{(2)} = \begin{pmatrix} 0 & \infty & 11 & 2 & 6 \\ 1 & 0 & 4 & \boxed{3} & \boxed{7} \\ 10 & \infty & 0 & \boxed{12} & \boxed{16} \\ \boxed{3} & 2 & 6 & 0 & 3 \\ \infty & \infty & 6 & \infty & 0 \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & \infty & 11 & 2 & 6 \\ 1 & 0 & 4 & 3 & 7 \\ 10 & \infty & 0 & 12 & 16 \\ 3 & 2 & 6 & 0 & 3 \\ \boxed{16} & \infty & 6 & \boxed{18} & 0 \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & \boxed{4} & \boxed{8} & 2 & \boxed{5} \\ 1 & 0 & 4 & 3 & \boxed{6} \\ 10 & \boxed{14} & 0 & 12 & \boxed{15} \\ 3 & 2 & 6 & 0 & 3 \\ 16 & \boxed{20} & 6 & 18 & 0 \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 4 & 8 & 2 & 5 \\ 1 & 0 & 4 & 3 & 6 \\ 10 & 14 & 0 & 12 & 15 \\ 3 & 2 & 6 & 0 & 3 \\ 16 & 20 & 6 & 18 & 0 \end{pmatrix}$$

Finding the actual shortest paths

In both of these algorithms we have not addressed the question of actually finding the paths themselves.

For the Floyd-Warshall algorithm this is achieved by constructing a further sequence of arrays $P^{(k)}$ whose (i, j) entry contains a predecessor of j on the path from i to j. As the entries are updated the predecessors will change — if the matrix entry is not changed then the predecessor does not change, but if the entry does change, because the path originally from i to j becomes re-routed through the vertex k, then the predecessor of j becomes the predecessor of j on the path from k to j.