Lecture Notes on Data Structures and Algorithms: Graphs

Conrado Martínez
U. Politècnica Catalunya

April 19, 2016

Part V

Graphs

1. Graphs
2. Depth-First Search (DFS)
3. Breadth-First Search (BFS)
4. Shortest Paths: Dijkstra's Algorithm
5. Minimum Spanning Trees: Prim's Algorithm

Basic Graph Theory

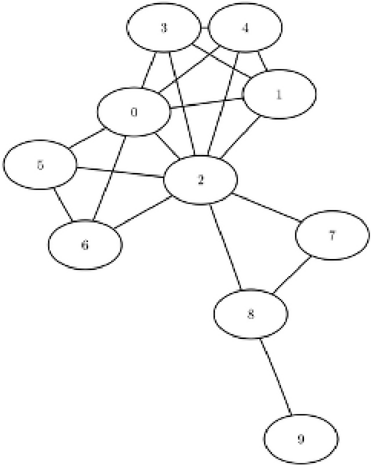
Definition

An (undirected) graph is a pair $G = \langle V, E \rangle$ where V is a finite set of vertices (a.k.a. nodes) and E is a set of edges; each edge $e \in E$ is an unordered pair $\{u, v\}$ with $u \neq v$ and $u, v \in V$.

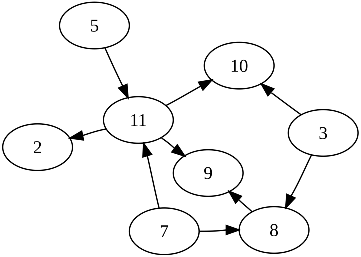
Definition

A directed graph or digraph is a pair $G = \langle V, E \rangle$ where V is the finite set of vertices and E is a set of arcs (but sometimes we will call them edges); each arc $e \in E$ is a pair (u, v) with $u \neq v$ and $u, v \in V$.

Basic Graph Theory

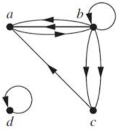


Undirected graph



Directed graph

Basic Graph Theory



When we have a multiset of arcs or edges instead of a set, and arcs or edges with both extremes identical (loops) are allowed, we talk about multigraphs (directed or undirected).
For an arc $e = (u, v)$, vertex u is called the source and a vertex v the target. We say that v is a successor of u; conversely, u is a predecessor of v. For an edge $e = \{u, v\}$, the vertices are called its extremes and we say u and v are adjacent. We also say that the edge e is incident to u and v.

Basic Graph Theory

The number of edges incident to a vertex u (equivalently, the number of vertices v adjacent to u) is called the degree of u (deg(u)).

For digraphs, we define the in-degree and out-degree of a vertex u, the number of predecessors and of successors, respectively, of the vertex u, and we note these as in-deg(u) and out-deg(u).

Lemma (Handshaking Lemma)

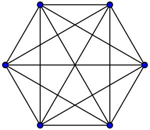
For any digraph $G = \langle V, E \rangle$

$$\sum_{v \in V} \textit{in-deg}(v) = \sum_{v \in V} \textit{out-deg}(v) = |E|$$

For any graph $G = \langle V, E \rangle$

$$\sum_{v \in V} \textit{deg}(v) = 2 \cdot |E|$$

Basic Graph Theory



For any graph G the number of edges $|E| = m$ is between 0 and

$$\binom{n}{2} = \frac{n(n-1)}{2}$$

where $n = |V|$. A graph with $m = n(n-1)/2$ is called complete and often denoted K_n. For digraphs, the number of arcs m must be between 0 and $n(n-1)$.

Given a family of graphs, the family is dense if for any member $m = \Theta(n^2)$; otherwise it is called sparse. For instance, the family of complete graphs is dense, and the family of cyclic graphs and the family of d-regular graphs (with d a constant) is sparse.

Basic Graph Theory

Definition

A path $P = v_0, v_1, v_2, \dots, v_n$ of length n in a graph $G = \langle V, E \rangle$ is a sequence of $n + 1$ vertices from G such that for all i, $0 \leq i < n$

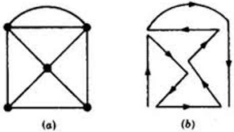
$$\{v_i, v_{i+1}\} \in E$$

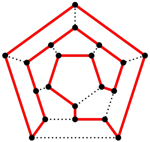
Vertex v_0 is called the origin of P and v_n the target.

(Directed) paths in digraphs are defined in the same way: for all i, $0 \leq i < n$, (v_i, v_{i+1}) is an arc in the digraph.

A path is called simple if no vertex appears more than once in the sequence, except possibly v_0 and v_n. If $v_0 = v_n$ the path is called a cycle.

Basic Graph Theory





A simple path that "visits" all vertices of the graph is a Hamiltonian path (or cycle, if closed). A graph is Hamiltonian if and only if contains at least one Hamiltonian path.

A path that contains all edges/arcs of the graph is an Eulerian path. A graph is Eulerian if and only if contains an Eulerian path.

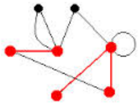
Basic Graph Theory

Definition

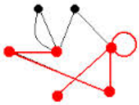
A graph $G = \langle V, E \rangle$ is connected if and only if there exists a path in G from u to v for all pairs of vertices $u, v \in V$.

The definition for digraphs is the same, but we will say that the digraph is strongly connected.

Basic Graph Theory



Subgraph (in red)



Induced Subgraph

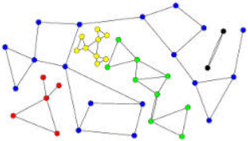
Definition

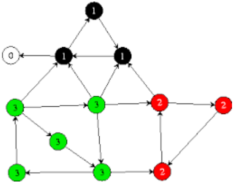
Given a graph $G = \langle V, E \rangle$, the graph $H = \langle V', E' \rangle$ is a subgraph of G if and only if $V' \subseteq V$, $E' \subseteq E$, and for all edges $e' = \{u', v'\} \in E'$ both u' and v' belong to V'.

If E' contains all edges in E which are incident to two vertices in V', the subgraph H is called the subgraph induced by V'. If $V' = V$ then H is called a spanning subgraph of G.

Subgraphs of digraphs are defined in a completely analogous way.

Basic Graph Theory



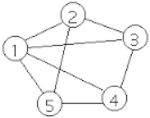


Definition

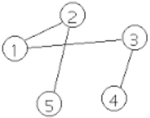
A connected component C of a graph G is a maximal connected induced subgraph of G. By *maximal* we mean that adding any vertex v to $V(C)$ the resulting induced subgraph is not connected.

For digraphs, the analogous concept is that of strongly connected components.

Basic Graph Theory



Graph



Spanning tree

Definition

A connected acyclic (that is, with no cycles) graph is a (free) tree. If G is a connected graph, a spanning subgraph $T = \langle V, E' \rangle$ which is a tree is a spannig tree. An acyclic graph is a forest and each of its connected components is a tree. A spanning forest is an acyclic spanning subgraph consisting of a set of trees, each a spanning tree for the corresponding connected components of G.

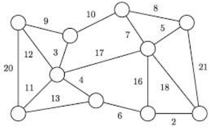
Basic Graph Theory

Trees (in the graph-theoretic sense) have no root and there is no order among the adjancent vertices to a given vertex in the tree.

Lemma

If $G = \langle V, E\rangle$ is a tree then $|E| = |V| - 1$. If G is a forest with c connected components (trees) then $|E| = |V| - c$.

Basic Graph Theory



We will often consider graphs or digraphs in which each edge (arc) bears a label. A labelling in a graph G is a function $\omega : E \to \mathcal{L}$ from the set of edges E to the (potentially infinite) set of labels $\mathcal{L}$.

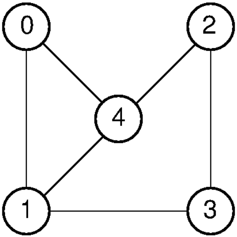
The most common case is that in which labels are numeric (integers, rational, real numbers); labelled graphs are then called weighted graphs (and the label $\omega(e)$ of and edge e is called its weight).

Implementation of Graphs

The two most frequent ways to implement a graph are:

1. Adjancency matrices: entry $A[i][j]$ of the adjancency matrix A is a Boolean indicating whether (i, j) is an edge/arc in $E(G)$ or not. For weighted (di)graph $A[i][j]$ stores the label assigned to the edge/arc (i, j).
2. Adjancency lists: we use an array or vector T such that for a vertex u, $T[u]$ points to a list of the edges incident to u or a list of the vertices $v \in V$ which are adjacent to u. For digraphs, we will have the list of the successors of a vertex u, for all vertices u, and, possibly, the list of predecessors of a vertex u, for all vertices u.

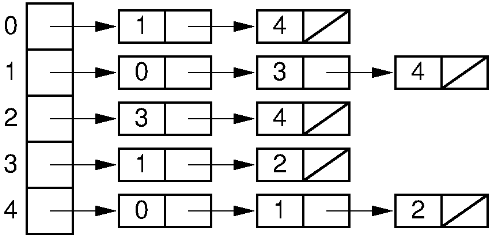
Implementation of Graphs



(a)

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

(b)



(c)

Implementation of Graphs

Adjacency matrices are usually too costly in space; if $|V(G)| = n$ it needs space $\Theta(n^2)$ to represent the graph, independent of the number of edges/arcs in the graph. Adjancency matrices are fine to represent dense graphs or when we need to answer very efficiently whether an edge $(u, v) \in E$ or not.

As a general rule, the preferred implementation will be adjancecy lists. They require space $\Theta(n + m)$ where $n = |V|$ and $m = |E|$; the space is thus linear in the size of the graph.

Implementation of Graphs

```
typedef int vertex;
typedef pair<vertex, vertex> edge;

class Graph {
// undirected graphs with V = {0, ..., n-1}
public:
// create en empty graph (no vertices and no edges)
      Graph();
// create en empty graph with n vertices (but no edges)
      Graph(int n);
// adds a new vertex to the graph; the new vertex will have identifier
// 'n' where n is the number of vertices in the graph, before adding the
// the new vertex
void add_vertex();
// adds edge (u,v) to the graph, either giving an edge e=(u,v) or its
// extremes u and v
void add_edge(vertex u, vertex v);
void add_edge(edge e);
// return the number of vertices and edges, respectively
int nr_vertices() const;
int nr_edges() const;
// return the list of edges incident to vertex u
list<edge> adjacent(vertex u) const;
...
private:
    int n, m;
    vector<list<edge>> T;
    // alternatives: vector<list<vertex>> T;
    //               vector<vector<vertex>> T;
    ...
}
```

Graph Traversals

Definition

Given a connected graph $G = \langle V, E \rangle$, a traversal of G is a sequence of all the vertices in $V(G)$ where each vertex v appears exactly once, and it holds that either v is the first vertex in the sequence or there exists an edge in $E(G)$ joining v with a vertex w appearing before v in the sequence.

A traversal of a graph is a sequence of traversals of the corresponding connected components. No vertex appears in the sequence if the traversal of the connected components of preceding vertices in the sequence hasn't been completed or the preceding vertex is in the same connected component.

Graph Traversals

For digraphs, a traversal starting in a vertex v visits every accesible vertex w from v exactly once; visiting a vertex $w \neq v$ implies that there exists some other vertex u preceding w in the traversal such that (u, w) is an arc in E.

A traversal of a (di)graph visits all its vertices, following the topology of the graph since except the initial vertices, all the others are visited by following an edge or arc in the graph.

Graph Traversals

Traversals (or a combination of them) allow us to efficiently solve many problems on graphs. For instance:

- Decide whether a graph is connected or not.
- Find the connected components of an undirected graph.
- Find the strongly connected components of a digraph.
- Find whether a graph contains cycles or not.
- Decide if a graph is bipartite or not (equivalently, if a graph is 2-coloreable or not)
- Decide whether a graph is biconnected or not (a graph is biconnected if and only if the removal of any vertex and its incident edges does not disconnect the graph).
- Find the shortest path (with least number of edges/arcs) between any two given vertices.
- Etc.

Part V

Graphs

1. Graphs
2. Depth-First Search (DFS)
3. Breadth-First Search (BFS)
4. Shortest Paths: Dijkstra's Algorithm
5. Minimum Spanning Trees: Prim's Algorithm

Depth-First Search

In a Depth-First Search (DFS) (cat: *recorregut en profunditat*, esp: *recorrido en profundidad*) of a graph G we visit a vertex v and from there we traverse, recursively, each non-visited vertex w which is adjacent/a successor of v.

When we visit a vertex u we say the vertex is open; it remains open until the recursive traversal of all its adjacent/successors has been finished, then u gets closed.

▸ DFS animation

Depth-First Search

The direct or DFS number of a vertex is the ordinal number in which the vertex is open by the DFS; if the DFS starts at vertex v, then the DFS number of v is 1.

The inverse number of a vertex is the ordinal number in which the vertex is closed by the DFS. The first vertex in the DFS for which all neighbors/successors have been visited has inverse number 1.

Depth-First Search

```
▷ $visited$, $ndfs$, $ninv$, $num\_dfs$, $num\_inv$ are global variables, for simplicity
procedure DFS($G$)
    for $v \in V(G)$ do
        $visited[v] :=$ false
        $ndfs[v] := 0; ninv[v] := 0$
    end for
    $num\_dfs := 0; num\_inv := 0$
    for $v \in V(G)$ do
        if $\neg visited[v]$ then
            DFS-REC$(G, v, v)$
        end if
    end for
end procedure
```

Depth-First Search

```
procedure DFS-REC($G$, $v$, $father$)
    PRE-VISIT$(v)$
    $visited[v]$ := true
    $num\_dfs := num\_dfs + 1; ndfs[v] := num\_dfs$
    for $w \in G$.ADJACENT$(v)$ do
        if $\neg visited[w]$ then
            PRE-VISIT-EDGE$(v, w)$
            DFS-REC$(G, w, v)$
            POST-VISIT-EDGE$(v, w)$
        else
▷ if $w \neq father$ there is a cycle (that conatins edge $(v, w)$)
        end if
    end for
    POST-VISIT$(v)$
    $num\_inv := num\_inv + 1; ninv[v] := num\_inv$
end procedure
```

Depth-First Search

```
typedef list<edge>::iterator edge_iter;
...

// Some useful macros
#define forall(v,G) for(vertex (v) = 0; (v) < (G).nr_vertices(); ++(v))

#define forall_adj(e, u, G) \
  for(edge_iter (e) = (G).adjacent((u)).begin(); \
      (e) != (G).adjacent((u))].end() ;  ++(e))
// this can be written as
// for (edge e : G.adjacent(u)) in C++11

#define target(eit) ((eit) -> second)
#define source(eit) ((eit) -> first)
```

Depth-First Search

```
void DFS(const Graph& G) {
    vector<bool> visited(G.nr_vertices(), false);
    vector<int> ndfs(G.nr_vertices(), 0);
    vector<int> ninv(G.nr_vertices(), 0);
    int num_dfs = 0;
    int num_inv = 0;

    forall(v, G)
       if (not visited[v])
          DFS(G, v, v, visited, num_dfs, num_inv, ndfs, ninv);
}
```

Depth-First Search

```
void DFS(const Graph& G, vertex v, vertex padre,
         vector<bool>& visited,
         int& num_dfs, int& num_inv,
         vector<int>& ndfs,
         vector<int>& ninv) {

     PRE-VISIT(v);
     visited[v] = true;
     ++num_dfs; ndfs[v] = num_dfs;
     forall_adj(e, v, G) {
        vertex w = target(e);
        if (not visited[w]) {
           PRE-VISIT-EDGE(v,w);
           DFS(G, w, v, visited, num_dfs, num_inv, ndfs, ninv);
           POST-VISIT-EDGE(v,w);
        } else {
          // we’ve found a cycle!
        }
    }
    POST-VISIT(v);
    ++num_inv; ninv[v] = num_inv;
}
```

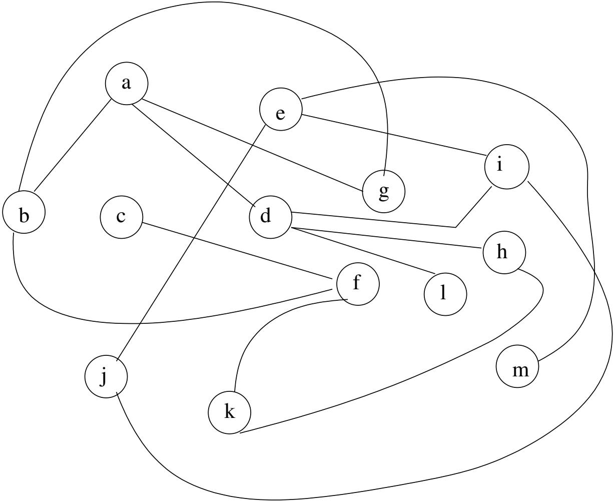
Depth-First Search

A call to DFS$(v, \ldots)$ visits the connected component of v and induces a spannig tree of that component; we call such spanning tree the DFS tree T_{DFS}.
All edges in the connected component can be thus classified in two types:

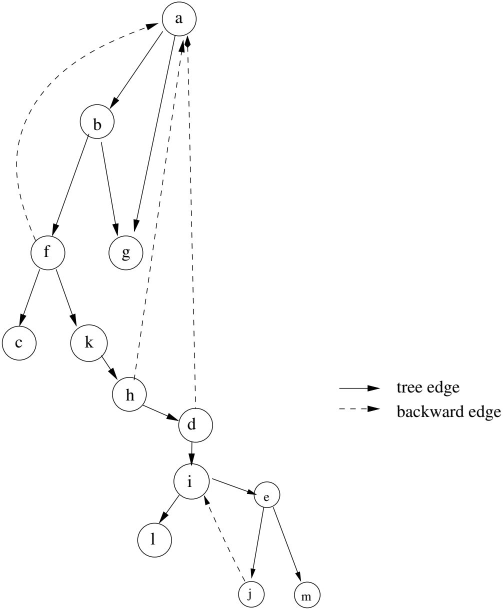
- Tree edges
- Backward edges. Backward edges join the currently visited vertex u with a previously visited vertex which is not the one from where the recursive traversal of u was launched. Backward edges close a cycle.

A full DFS of a graph induces a spanning forest.

Depth-First Search



Depth-First Search



Depth-First Search

Consider the following basic problem:
For each vertex v, we want $CC[v]$ to be the number of its connected component (the number is unimportant; what matters is that $CC[u] = CC[v]$ if and only if u and v are in the same CC). We also want $TreeEdges[i]$ to be the set of tree edges in CC number i and $BackEdges[i]$ to be the set of backward edges. The total number of CCs will be ncc.

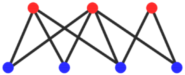
Depth-First Search

```
procedure DFS($G$)
    for $v \in V(G)$ do
        $visited[v] :=$ false
        $CC[v] := 0$
    end for
    $ncc := 0$
    for $v \in V(G)$ do
        if $\neg visited[v]$ then
            $ncc := ncc + 1$
            $TreeEdges[ncc] := \emptyset$
            $BackEdges[ncc] := \emptyset$
            DFS-REC($G, v, v$)
        end if
    end for
end procedure
```

Depth-First Search

```
procedure DFS-Rec($G$, $v$, $father$)
    $visited[v] :=$ true
    $CC[v] := ncc$
    for $w \in G.$Adjacent$(v)$ do
        if $\neg visited[w]$ then
            $TreeEdges[ncc] := TreeEdges[ncc] \cup \{(v, w)\}$
            DFS-Rec$(G, w, v)$
        else if $w \neq father$ then
            $BackEdges[ncc] := BackEdges[ncc] \cup \{(v, w)\}$
        end if
    end for
end procedure
```

Applications of DFS



Detecting cycles or find the CCs (as before) are trivial examples of applications of the DFS.

Another DFS-based simple algorithm allows us to decide if a given graph is bipartite or not.

A graph is bipartite if there exists a partition $\langle A, B\rangle$ of the set of vertices V (i.e., $A \cup B = V; A \cap B = \emptyset$) such that every edge joins a vertex in A with a vertex in B. Equivalently,

- a graph is bipartite if and only if it is 2-coloreable
- a graph is bipartite if and only if it contains no cycle of odd length

Why?

Applications of DFS: Bipartite Graphs

```
procedure IsBipartite($G$)
    for $v \in V(G)$ do
        $color[v] := -1$
    end for
    $c := 0$
    $is\_bip :=$ true
    for $v \in V(G)$ while $is\_bip$ do
        if $color[v] = -1$ then
            $is\_bip :=$ IsBipartite-Rec$(G, v, c)$
        end if
    end for
    return $is\_bip$
end procedure
```

Applications of DFS: Bipartite Graphs

```
procedure IsBipartite-Rec($G$, $v$, $c$)
    $is\_bip$ := true
    $color[v] := c$
    for $w \in G$.Adjacent$(v)$ while $is\_bip$ do
        if $color[w] = -1$ then
            $is\_bip$ := IsBipartite-Rec$(G, w, 1 - c)$
        else
            $is\_bip := (c \neq color[w])$
        end if
    end for
    return $is\_bip$
end procedure
```

The Cost of DFS

Let us assume that the cost of visiting a vertex (PRE- and POST-VISIT) or visiting an edge (either a tree or a backward edge) is $\Theta(1)$.
Then the cost of a DFS is

$$\sum_{v \in V} \Big(\Theta(1) + \Theta(\mathsf{deg}(v)) \Big) = \Theta \left(\sum_{v \in V} \Big(1 + \mathsf{deg}(v) \Big) \right) =$$

$$\Theta \left(\sum_{v \in V} 1 + \sum_{v \in V} \mathsf{deg}(v) \right) = \Theta(n + m)$$

DFS in Digraphs

DFS in digraphs are a bit more complicated to understand than DFS in undirected graphs. When we launch a DFS in a digraph from a vertex v we do not visit just its strongly connected component (SCC), but all accessible (not visited) vertices from v; that includes all vertices in v's SCC but some others.

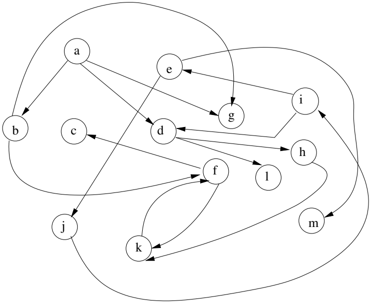
Each call to DFS$(v, \ldots)$ induces a directed tree, with v as a root. DFS numbers and inverse numbers are defined as in DFS for undirected graphs.

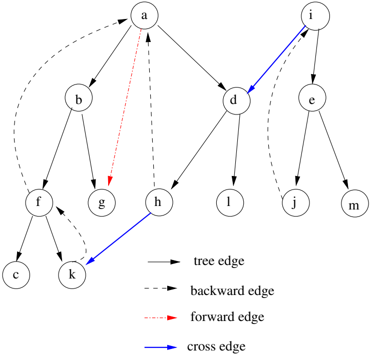
DFS in Digraphs

DFS in a digraph also induces a classification of the arcs, but we have now four different types:

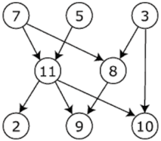
1. Tree edges: from currently visited vertex v to a non-visited vertex w
2. Backward edges: from currently visited vertex v to an ascendant w in the DFS tree T_{DFS}; $ndfs[w] < ndfs[v]$ and $ndfs[w]$ is open
3. Forward edges: from currently visited vertex v to a previously visited descendant w in T_{DFS}; $ndfs[v] < ndfs[w]$
4. Cross edges: from currently visited vertex v to a previously visited vertex w in the same DFS tree or a different traversal tree; $ndfs[w] < ndfs[v]$, but w is already closed ($ninv[w] \neq 0$)

DFS in Digraphs





Directed Acyclic Graphs



A directed acyclic graph (DAG) is a digraph that contains no cycles (as its name indicates).

DAGs have many applications since they modellize well requirements and precedence. For instance, in a large complex software system, each node is a subsystem and each arc (A, B) indicates that subsystem A uses subsystem B.

Vertices with no predecessors are called roots; vertices with no successors are called leaves of the DAG.

Topological Sort

A special type of traversal that makes sense in a DAG is a topological sort. A topological sort of a DAG G is a sequence of all its vertices such that for any vertex w, if v precedes w in the sequence then $(w, v) \notin E$.
In other words, no vertex is visited until all its predecessors have been visited.

Topological Sort

Suppose that we have $pred[v]$, the number of predecessors of v in G that have not yet been visited. Then if any vertex v has $pred[v] = 0$ we can visit it and decrement by one $pred[w]$, for all its successors w. The algorithm has thus two steps:

1. Compute the number of predecessors $pred[v]$ for all vertices $v \in G$
2. Repeatedly, until all vertices have been visited, visit a vertex v and update $pred[w]$ for all its successors.

Topological Sort

The two steps are basic variants of the DFS, which take advantadge of the fact that there are no cycles in the graph (by definition).

```
procedure TopologicalSort($G$)
    for $v \in G$ do $pred[v] := 0$
    end for
    for $v \in G$ do
        for $w \in G$.Successors($v$) do $pred[w] := pred[w] + 1$
        end for
    end for
    $Q := \emptyset$
    for $v \in G$ do
        if $pred[v] = 0$ then
            $Q$.PUSH_BACK($v$)
        end if
    end for
    ...
end procedure
```

Topological Sort (cont’d)

```
procedure TopologicalSort($G$)
    ...
    while $\neg Q$.EMPTY$()$ do
        $v :=$ $Q$.POP_FRONT$()$
        VISIT$(v)$
        for $w \in G$.Successors$(v)$ do
            $pred[w] := pred[w] - 1$
            if $pred[w] = 0$ then
                $Q$.PUSH_BACK$(w)$
            end if
        end for
    end while
end procedure
```

Part V

Graphs

1. Graphs
2. Depth-First Search (DFS)
3. Breadth-First Search (BFS)
4. Shortest Paths: Dijkstra’s Algorithm
5. Minimum Spanning Trees: Prim’s Algorithm

Breadth-First Search

In a Breadth-First Search (BFS) (cat: *recorregut en amplada*, esp: *recorrido en anchura*) of a graph G starting from some vertex s we visit all vertices in the connected component of s in increasing distance from s.

When a vertex is visited, all its adjacent non-visited vertices are put into a list of vertices yet to be visited.

Breadth-First Search

In BFS we keep a queue of vertices that have been not yet visited. All the vertices in the queue are at distance d or $d+1$ from the starting vertex s, where d is the distance to s of the last visited vertex. Moreover, all vertices at distance d in the queue precede the vertices at distance $d+1$, and all vertices of G at distance d from s have already been visited or they are in the queue.

The BFS thus visits the connected component of s by traversing the levels or layers defined by the distance to s.

Breadth-First Search

```
procedure BFS($G$, $s$, $D$)
▷ Assumes $G$ is connected; all vertices will be visited
▷ After execution, $D[v] =$ distance from $s$ to $v$
    for $v \in G$ do
        $D[v] := -1$ ▷ $D[v] = -1$ indicates that $v$ hasn’t been visited
    end for
    $Q := \emptyset$; $Q$.PUSH($s$); $D[s] := 0$
    while $\neg Q$.EMPTY() do
        $v := Q$.POP()
        VISIT($v$)
        for $w \in G$.ADJACENT($v$) do
            if $D[w] = -1$ then
                $D[w] := D[v] + 1$
                $Q$.PUSH($w$)
            end if
        end for
    end while
end procedure
```

The Cost of BFS

Assume that the cost of each visit of a vertex is $\Theta(1)$, and that the graph is implemented with adjancecy lists. Each vertex v will be inserted into Q once and removed and visited once; for each vertex we go through its adjacent vertices and push them into the queue, if not visited. Therefore the cost of BFS is clearly

$$\sum_{v \in V} \Theta(1) + \Theta(\mathsf{deg}(v)) = \Theta(|V| + \sum_{v \in V} \mathsf{deg}(v)) = \Theta(|V| + |E|)$$

that is, linear in the size of the graph.

Applications of BFS: Diameter and center

Given a connected graph G its diameter is the maximal distance between a pair of vertices. The center of the graph is the vertex such that the maximal distance to any other vertex is minimal.

A surprising finding in the study of many graphs/networks arising in many areas, specially in social sciences, is that despite the graphs are quite large their diameter is not. That result has been often been known as the six degrees of separation phenomenon.

▸ Erdös number ▸ The Oracle of Bacon

Applications of BFS: Diameter and center

To compute the diameter we need to perform a BFS starting from each vertex v in G. That will give us cost $\Theta(|V| \cdot (|V| + |E|)) = \Theta(n(n+m)) = \Theta(n \cdot m)$, since the graph is connected (then $m \geq n-1$).

A call BFS(G, s, D) allows us to compute $maxD[s] = \max\{D[v] \,|\, v \in G\}$; this can actually be done within the internal loop when we update $D[w] := D[v] + 1$. The diameter D is

$$D = \max\{maxD[s] \,|\, s \in G\}$$

while the center c is the vertex such that $maxD[c]$ is minimum,

$$c = \text{arg min}\{maxD[s] \,|\, s \in G\}$$

Part V

Graphs

1. Graphs
2. Depth-First Search (DFS)
3. Breadth-First Search (BFS)
4. Shortest Paths: Dijkstra’s Algorithm
5. Minimum Spanning Trees: Prim’s Algorithm

Dijkstra's Algorithm

Given a weighted digraph $G = \langle V, E \rangle$, Dijkstra's algorithm (1959) finds all shortest paths from a given vertex (the source) to all other vertices in the digraph.

If weights can be negative then the digraph can contain negative weight cycles, and then shortest paths are not well defined; hence, Dijkstra's algorithm can fail if there are arcs in the digraph with negative weight. We will assume from now on that all weights $\omega : E \to \mathbb{R}^+$ are positive.

Let $\mathcal{P}(u, v)$ denote the set of all (simple) paths between two vertices u and v of G. Given a path $\pi = [u, \ldots, v] \in \mathcal{P}(u, v)$ its weight is the sum of the weights of the arcs in π:

$$\omega(\pi) = \omega(u, v_1) + \omega(v_1, v_2) + \cdots + \omega(v_{n-1}, v).$$

Dijkstra's Algorithm

Let $\Delta(u,v) = \min\{\omega(\pi) \mid \pi \in \mathcal{P}(u,v)\}$ and $\pi^*(u,v)$ a path in $\mathcal{P}(u,v)$ with minimal weight. If $\mathcal{P}(u,v) = \emptyset$ then we define $\Delta(u,v) = +\infty$, by convention.

Our first version of Dijkstra's algorithm computes $\Delta(u,v) = \omega(\pi^*(u,v))$ for all $v \in V(G)$, for some given vertex u. Later we will see how to find the shortest paths $\pi^*(u,v)$, not just their weights.

Dijkstra's Algorithm

```
▷ $G = \langle V, E \rangle$ is a weighted digraph, all weights are positive
▷ $s \in V$
DIJKSTRA$(G, s, D)$
▷ For all $v \in V$, $D[v] = \Delta(s, v)$
```

```
▷ $G = \langle V, E \rangle$ is a weighted digraph, all weights are positive
▷ $s \in V$
DIJKSTRA$(G, s, D, path)$
▷ For all $v \in V$, $D[v] = \Delta(s, v)$
▷ For all $v \in V$, $D[v] < +\infty \implies path[v] = \pi^*(s, v)$
```

Dijkstra's Algorithm

Dijkstra's algorithm works iteratively. On each iteration the set of vertices is partitioned in two parts: the vertices that we have visited and those that haven't been visited yet, also called candidates. The invariant of the loop establishes that

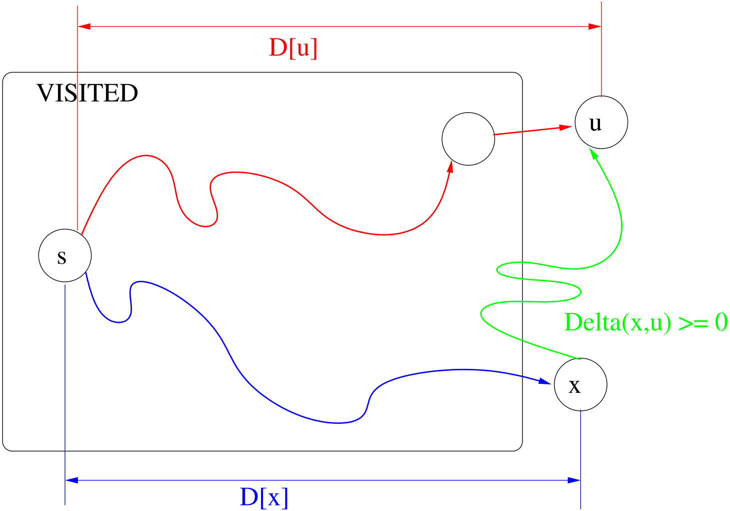
1. For all visited vertices u, $D[u] = \Delta(s, u)$
2. For all candidate vertices u, $D[u]$ is the weight of the shortest path from s to u which passes exclusively through visited vertices, i.e., only the last vertex in the path, u, is a candidate.

Each iteration of the main loop selects a candidate vertex, updates D and marks u as visited. When all vertices have been visited, we have the desired answer.

Dijkstra’s Algorithm

Which candidate vertex do we need to select on each iteration of Dijkstra’s algorithm? Intuitively, the candidate to choose is the one with minimal D. Let u be that vertex. According to the invariant, $D[u]$ is minimal among all paths passing only through visited vertices. But it must also be the minimal weight for all paths.
If there were a shortest path going through a candidate vertex x, the weight of that path would be $D[x] + \Delta(x, u) < D[u]$. Since $\Delta(x, u) \geq 0$ we would have $D[x] < D[u]$, and that’s a contradiction since by assumption $D[u]$ is minimal among all candidate vertices.

Dijkstra’s Algorithm

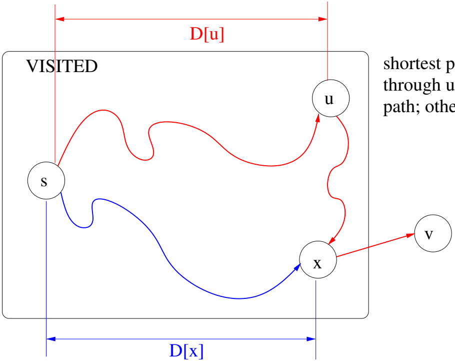


Dijkstra’s Algorithm

As we have just seen $D[v] = \Delta(s, v)$ for all visited vertices v, and for the vertex u to be visited. We need only to consider how to maintain the second part of the invariant.

Consider a candidate vertex v. Since u becomes a visited vertex in the current iteration, we may have a new shortest path from s to v going only through visited vertices that now include u.
A simple reasoning shows that if such shortest path includes u then u must be the immediate predecessor of v in the path (assuming the contrary leads to a contradiction).

Dijkstra’s Algorithm



shortest path from s to v passes
through u => u is the last visited vertex in the
path; otherwiseD[x] > D[u] => contradiction!

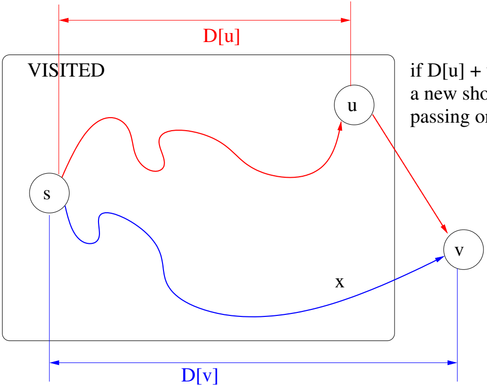
Dijkstra’s Algorithm

It follows that $D[v]$ can change if and only if v is a successor of u; in particular, if

$$D[v] > D[u] + \omega(u, v).$$

then $D[v]$ must be updated $D[v] := D[u] + \omega(u, v)$. The condition above is never true if v has already been visited.

Dijkstra’s Algorithm

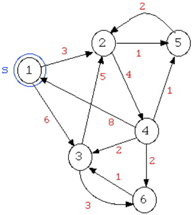


if D[u] + weight(u,v) < D[v] =>
a new shortest path from s to v, via u,
passing only through VISITED vertices

Dijkstra's Algorithm

```
procedure Dijkstra($G$, $s$, $D$)
    ▷ At the end $D[u] = \Delta(s, u)$ for all $u \in V(G)$
    ▷ $cand$ is the subset of candidates in $V(G)$
    for $v \in V(G)$ do
        $D[v] := +\infty$
    end for
    $D[s] := 0$
    $cand := V(G)$
    while $cand \neq \emptyset$ do
        $u :=$ the vertex in $cand$ with minimum $D$
        $cand := cand - \{u\}$
        for $v \in G.$Successors$(u)$ do
            $d := D[u] + G.$weight$(u, v)$
            if $d < D[v]$ then
                $D[v] := d$
            end if
        end for
    end while
end procedure
```

Dijkstra’s Algorithm



D						CANDIDATES
1	2	3	4	5	6	
0	∞	∞	∞	∞	∞	$\{1,2,3,4,5,6\}$
0	3	6	∞	∞	∞	$\{2,3,4,5,6\}$
0	3	6	7	4	∞	$\{3,4,5,6\}$
0	3	6	7	4	∞	$\{3,4,6\}$
0	3	6	7	4	8	$\{4,6\}$
0	3	6	7	4	8	$\{6\}$
						$\emptyset$

Dijkstra’s Algorithm

To compute the shortest paths (not just their weights) we need the following key observation: if the shortest path from s to u goes through x then the subsequence going from s to x must be a shortest path from s to x.
We shall hence compute a (implicit) shortest paths tree: for all vertices

$$path[v] = \begin{cases} s & \text{if } v = s, \\ u & \text{if } (u,v) \text{ is the last arc in } \pi^*(s,v), \\ \perp & \text{if } D[v] = \Delta(s,v) = +\infty, \end{cases}$$

($path[v] = \perp$ indicates that no path exists from s to v).

Dijkstra's Algorithm

The only changes to our code:

1. Initialize $path[s] := s$; Initialize $path[v] = \perp$ for all other vertices v
2. Update $path$ every time D is updated

```
...
for $v \in G.\textsc{Sucessors}(u)$ do
    $d := D[u] + G.\textsc{Weight}(u, v)$
    if $d < D[v]$ then
        $D[v] := d$
        $path[v] := u$
    end if
end for
...
```

To recover the full path from s to u, if $path[u] \neq \perp$ then it is enough to roll back:

$$\pi^*(s, u) = [path[u], path[path[u]], \ldots, path[\cdots[path[u]]\cdots], s]^{\text{reverse}}.$$

The Cost of Dijkstra's Algorithm

Let $n = |V(G)|$ and $m = |E(G)|$, as usual. Assume that the digraph is implemented with an adjacency matrix. Then the cost of the algorithm is $\Omega(n^2)$ since there are n iterations, and we incur a cost $\Omega(n)$ to go through the successors v of the chosen candidate (because we are scanning the corresponding row in the matrix). Finding the minimum D at each iteration is $\mathcal{O}(n)$ and therefore the cost of the algorithm is actually $\Theta(n^2)$. Let's now assume that the graph is implemented with adjancecy lists. Then cost of the internal loop that goes through the successors of u is

$$\Theta(\text{deg}(u) \cdot \text{cost of updating } D).$$

The Cost of Dijkstra's Algorithm

If the set $cand$ and the table D are implemented as simple arrays ($cand[i] = \mathtt{true}$ if and only if i is a candidate) then the cost of the algorithm is

$$\Theta(n^2 + m) = \Theta(n^2)$$

since we have cost $\Theta(i + \deg(u))$ during the iteration in which we have still i candidates and u is selected to be visited.

Improving the Cost of Dijkstra’s Algorithm

For arbitrary sets of vertices $V(G)$ (not just $V = \{0, \ldots, n-1\}$, we can achieve the same cost implementing $cand$ (and D) with hash tables. However the problem is that selecting a candidate to visit is still inefficient.

To improve the efficiency we need to convert $cand/D$ to a priority queue; the elements are the candidate vertices and the priority of candidate v is its $D[v]$.

Improving the Cost of Dijkstra's Algorithm

```
▷ $cand$: a min-priority queue
▷ elements are vertices in $G$,
▷ priority of $u$ is $D[u]$
...
▷ $D[v] = +\infty$ for all $v \neq s$; $D[s] = 0$
for $v \in V(G)$ do
    $cand.$INSERT$(v, D[v])$
end for
while $\neg cand.$EMPTY$()$ do
    $u := cand.$MIN$()$; $cand.$REMOVE_MIN$()$
    for $v \in G.$SUCESSORS$(u)$ do
        ...
    end for
end while
```

Improving the Cost of Dijkstra's Algorithm

But since we can update priorities, our data structure must support another method to decrease the priority of a given element.

```
...
for $v \in G.\text{SUCESSORS}(u)$ do
    $d := D[u] + G.\text{WEIGHT}(u, v)$
    if $d < D[v]$ then
        $D[v] := d$
        $cand.\text{DECREASE\_PRIO}(v, d)$
    end if
end for
...
```

Improving the Cost of Dijkstra's Algorithm

If all operations on the priority queu have cost $\mathcal{O}(\log n)$ then the cost of Dijkstra's algorithm is

$$\begin{aligned} D(n) &= \sum_{v \in V(g)} \mathcal{O}(\log n) \cdot (1 + \text{out-deg}(v)) \\ &= \Theta(n \log n) + \mathcal{O}(\log n) \sum_{v \in V(g)} \text{out-deg}(v) \\ &= \Theta(n \log n) + \mathcal{O}(m \log n) = \mathcal{O}((n+m) \log n) \end{aligned}$$

The cost of the initializations is $\mathcal{O}(n \log n)$.
Then the total cost is

$$\mathcal{O}((n+m) \log n)$$

The worst-case is actually $\Theta((m+n) \log n)$, but the cost in many situations is often significantly smaller since we need not to decrease the priority of all successors of the selected candidate, or the cost of decreasing the priority is not $\Theta(\log n)$.

Improving the Cost of Dijkstra’s Algorithm

What we need is a data structure that combines the features of priority queue and of dictionary (preferably a hash, if $V \neq \{0, \ldots, n-1\}$), since to find the priority of an element or to decrease its priority we must (quickly) locate the element within the heap.

```
template <class Elem, class Prio>
class DijkstraPrioQueue<Elem,Prio> {
public:
   DijkstraPrioQueue(int n = 0);
   void insert(const Elem& e, const Prio& prio);
   Elem min() const;
   void remove_min();
   Prio prio(const Elem& e) const; // returns the priority of element v
   bool contains(const Elem& e) const;
   void decrease_prio(const Elem& e, const Prio& newprio);
   bool empty() const;
   ...
};
```

Improving the Cost of Dijkstra’s Algorithm

```
typedef vector<list<pair<int,double>>> graph;

...
void Dijkstra(const graph& G, int s, ... ) {
  DijkstraPrioQueue<int,double> cand(G.size());

  for (int v = 0; v < G.size(); ++v)
    cand.insert(v, INFINITY);

  cand.decrease_prio(s, 0);

  while(not cand.empty()) {
    int u = cand.min(); double du = cand.prio(u);
    cand.remove_min();
    for (auto e : G[u]) {
          // e is a pair <v, weight(u,v)>
          int v = e.first;
          double d = du + e.second;
          if (d < cand.prio(v))
               cand.decrease_prio(v, d);
          }
    }
  }
}
```

Part V

Graphs

1. Graphs
2. Depth-First Search (DFS)
3. Breadth-First Search (BFS)
4. Shortest Paths: Dijkstra’s Algorithm
5. Minimum Spanning Trees: Prim’s Algorithm

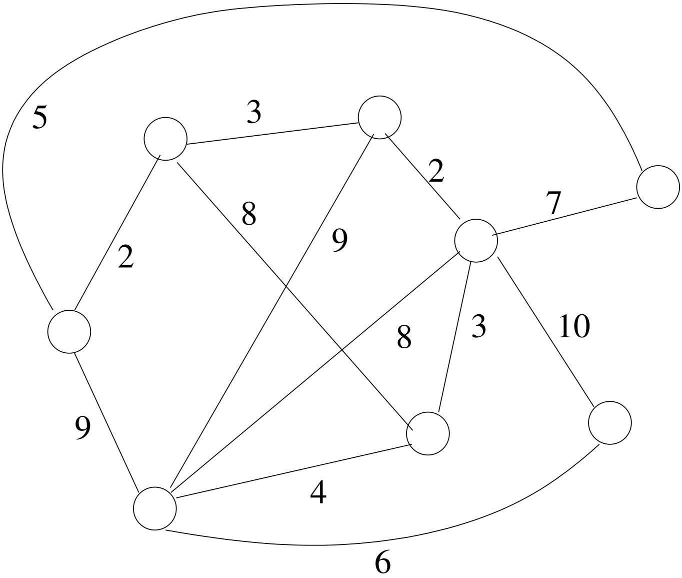
Minimum Spanning Trees (MSTs)

Given a weighted undirected connected graph $G = \langle V, E \rangle$, with weights $\omega : E \to \mathbb{R}$, a minimum spanning tree (MST) of G is a spanning subgraph $T = \langle V, A \rangle$ of G ($V(T) = V(G)$), that is a tree (connected and acyclic) and its total weight

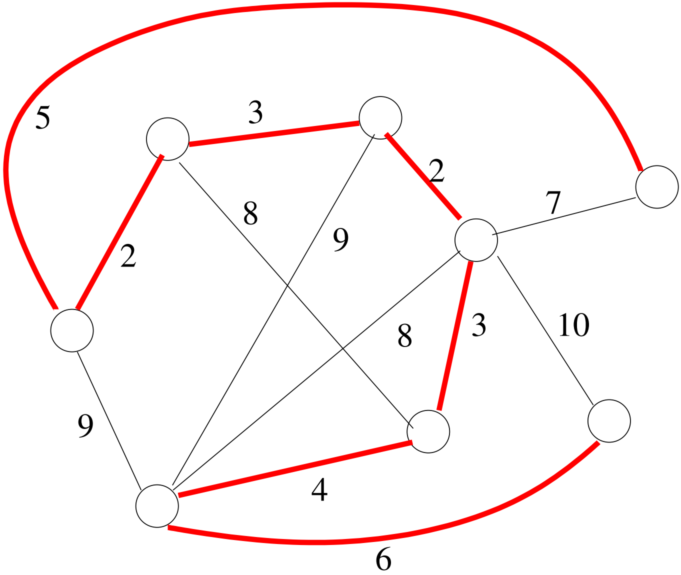
$$\omega(T) = \sum_{e \in A} \omega(e)$$

is minimum among all possible spanning trees of G.

Minimum Spanning Trees (MSTs)



Minimum Spanning Trees (MSTs)



Minimum Spanning Trees (MSTs)

There are many different algorithms to compute MSTs. All them follow the greedy scheme.

```
$A := \emptyset$; *Candidates* $:= E$
**while** $|A| \neq |V(G)| - 1$ **do**
    Select an edge $e \in$ *Candidates* that
      does not close a cycle in $T = \langle V, A \rangle$
    $A := A \cup \{e\}$
    *Candidates* $:=$ *Candidates* $- \{e\}$
**end while**
```

Minimum Spanning Trees (MSTs)

Definition

A subset of edges $A \subset E(G)$ is promising if and only if

1. A contains no cycle
2. A is a subset of the edges of some MST of G

Minimum Spanning Trees (MSTs)

Definition

A cut in a graph G is a partition of the set of vertices V in two subsets C and C', with

$$C \cup C' = V(G) \quad \text{and} \quad C \cap C' = \emptyset$$

Definition

An edge e respects a cut $\langle C, C' \rangle$ if both ends of e belong to the same part (C or C'); otherwise, we say that e crosses the cut.

Minimum Spanning Trees (MSTs)

Theorem

Let A be a promising set of edges that respects some cut $\langle C, C' \rangle$ of G. Let e be an edge of minimum weight among those crossing the cut $\langle C, C' \rangle$. Then

$$A \cup \{e\}$$

is promising.

Minimum Spanning Trees (MSTs)

Previous theorem gives us a “recipe” to design MST algorithms: start with an empty set of edges A; for each iteration, define what is the cut of G, then select the edge with minimum weight among those crossing the cut and add the edge to A.

Since A respects the cut and e crosses it, the addition of e to A cannot create a cycle.

Any algorithm following the scheme above is automatically guaranteed correct.

Minimum Spanning Trees (MSTs)

Proof of Theorem 3:
Let A' be the set of edges of some MST T' such that $A \subset A'$ (this is well defined, as A is assumed to be *promising*). As A respects the cut there must exist at least one edge crossing the cut belonging to A', otherwise T' would not be connected. Let e' be one such edge. If e' is of minimum weight then $A \cup \{e'\}$ is promising and the theorem is proved in this case. Assume now that e' is not of minimum weight.

Minimum Spanning Trees (MSTs)

Proof of Theorem 3 (cont'd):
The total weight (or cost) of T' includes:

1. the sum of the weights of the edges in A
2. the weight $\omega(e')$
3. the sum of the weight of some other edges

In the theorem we add the edge e of minimum weight crossing the cut to A (and we are assuming now that $e \notin A'$). Then it will create a cycle, since T' is a tree, and there will be some other $e' \in A'$ crossing the cut, hence part of the cycle too.

Minimum Spanning Trees (MSTs)

Proof of Theorem 3 (cont'd):
If we remove e'

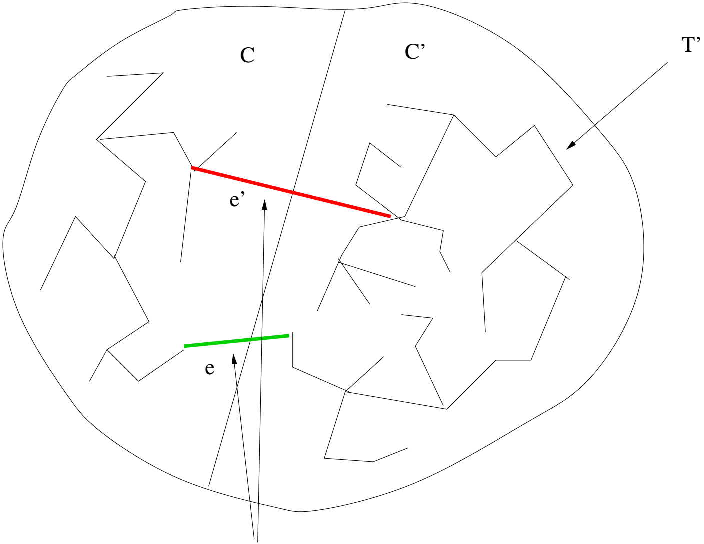
$$T = T' \cup \{e\} - \{e'\}$$

we get a new spanning tree T and its total weight is

$$\omega(T) = \omega(T') + \omega(e) - \omega(e') \leq \omega(T')$$

which is a contradiction unless $\omega(e) = \omega(e')$ and $\omega(T) = \omega(T')$.
That completes the proof, since $A \cup \{e\}$ is a subset of T which a MST of G.

Minimum Spanning Trees (MSTs)



podemos sustituir e' por e para obtener un nuevo MST

Prim’s Algorithm

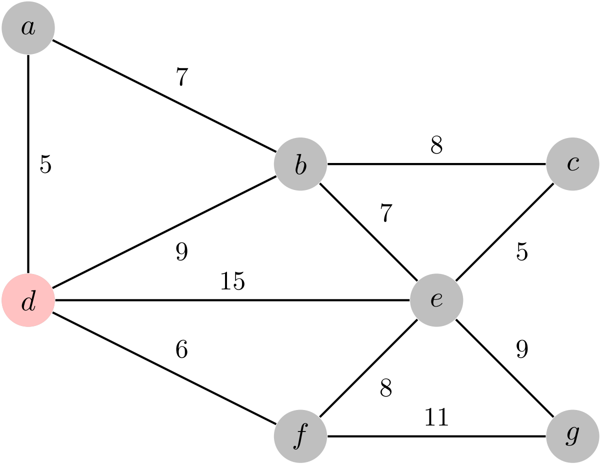
In Prim’s algorithm (a.k.a. Prim-Jarník algorithm), we maintain a subset *Visited* $\subseteq V(G)$ of vertices; each iteration of the algorithm selects the edge of minimum weight that joins a *Visited* vertex u with a non-*Visited* vertex v.

Such edge cannot create a cycle, and it is added to the current set of edges A; furthermore vertex v is marked as *Visited*. The set A is a spanning tree of the subset *Visited* of vertices.

Since the set A respects the cut $\langle \textit{Visited}, V(G) - \textit{Visited} \rangle$ and we select the edge of minimum weight crossing the cut to add it to A at each step, correctness immediately follows from Theorem 3.

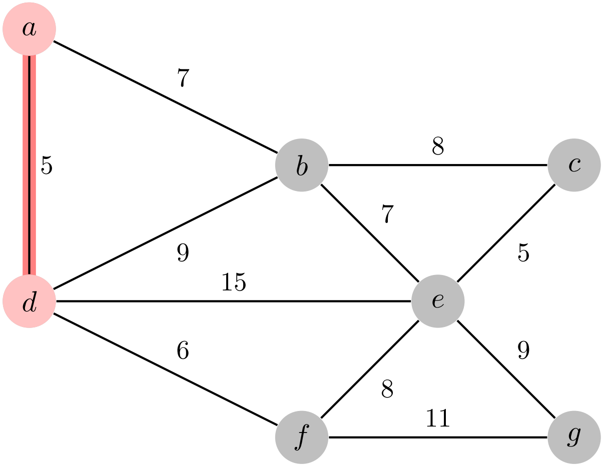
```
procedure Prim($G$)
    $A := \emptyset$; Visited $:= \{s\}$
    Candidates $:= \emptyset$
    for $v \in G$.Adjacent$(s)$ do
        Candidates $:=$ Candidates $\cup \{(s, v)\}$
    end for
    while $|A| \neq |V(G)| - 1$ do
        Select $e = (u, v)$
          from Candidates with minimum weight
        $A := A \cup \{e\}$
        Let $u$ be the Visited vertex:
        Visited $:=$ Visited $\cup \{v\}$
        for $w \in G$.Adjacent$(v)$ do
            if $w \notin$ Visited then
                Candidates $:=$ Candidates $\cup \{(v, w)\}$
            else
                Candidates $:=$ Candidates $- \{(v, w)\}$
            end if
        end for
    end while
    return $A$
end procedure
```

Prim's Algorithm



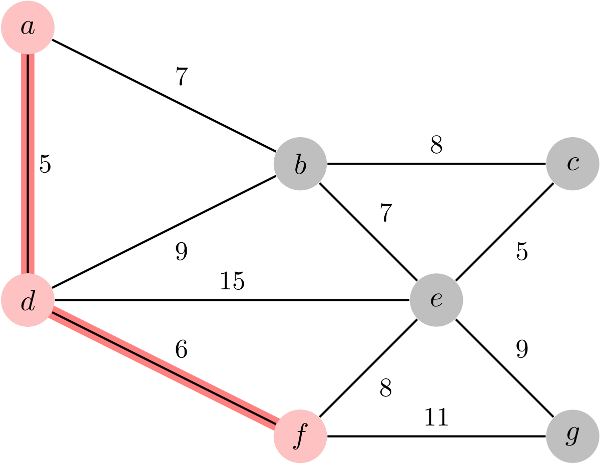
Source: Kjell Magne Fauske (TEXample.net)

Prim's Algorithm



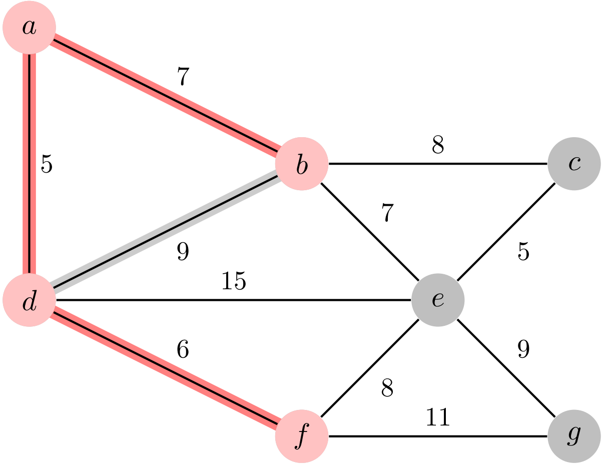
Source: Kjell Magne Fauske (TEXample.net)

Prim's Algorithm



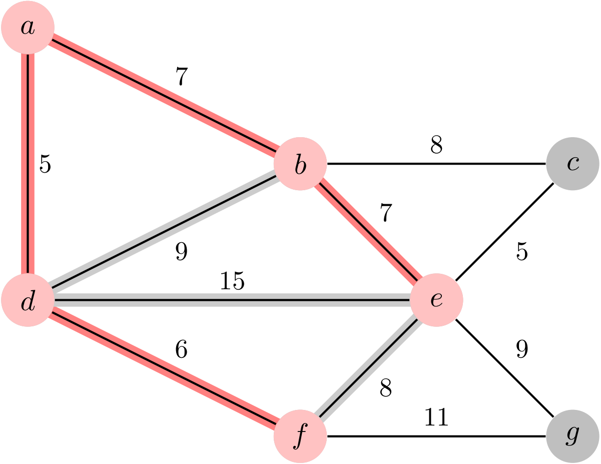
Source: Kjell Magne Fauske (TEXample.net)

Prim’s Algorithm



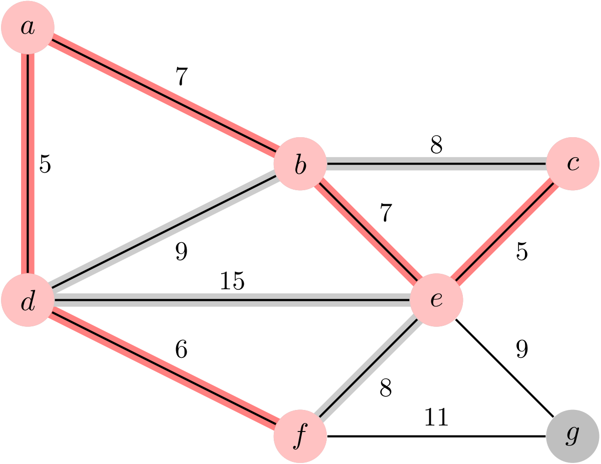
Source: Kjell Magne Fauske (TEXample.net)

Prim's Algorithm



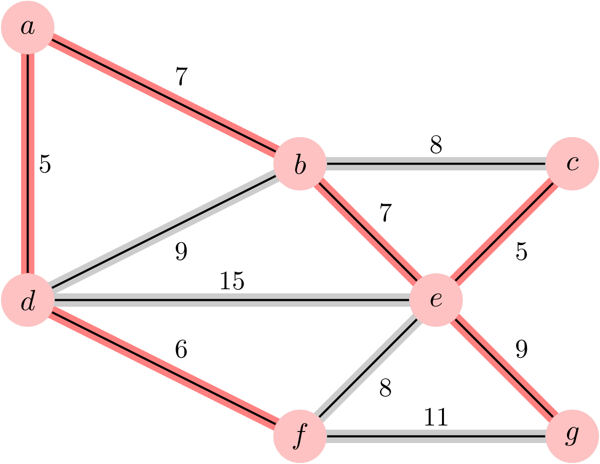
Source: Kjell Magne Fauske (TEXample.net)

Prim's Algorithm



Source: Kjell Magne Fauske (TEXample.net)

Prim’s Algorithm



Source: Kjell Magne Fauske (TEXample.net)

The Cost of Prim's Algorithm

The cost of Prim's algorithm depends how do we implement the selection of the minimum weight edge. If we use a priority queue, the selection of the edge to add at each step can be done in time $\Theta(\log m) = \Theta(\log n)$. We incur an additional $\Theta(1)$ for the visit of a vertex on each iteration. The internal loop through the adjacent vertices, adding new candidate edges and removing edges that connect the just visited vertex with previously visited vertices has cost

$$\Theta(\mathsf{deg}(v) \cdot \log m) = \Theta(\mathsf{deg}(v) \cdot \log n)$$

when the currently visited vertex is v.

The Cost of Prim's Algorithm

If we add the contributions of the costs for all vertices v (the main loop visits one vertex on each iteration) we get

$$\begin{aligned}\sum_{v \in V(G)} \Theta\left(\text{deg}(v)(1+\log n)\right) + \Theta(\log n) + \Theta(1) &= \Theta\left(\sum_{v \in V(G)} \text{deg}(v)(1 -\right. \\ &= \Theta(m \log n) + \Theta(m) +\end{aligned}$$

If we don't use a heap for candidate edges, the cost of the algorithm is then $\mathcal{O}(n \cdot m)$, since searching the candidate edge of minimum weight takes time $\mathcal{O}(m)$, and doing it n times will dominate the total cost of the algorithm giving the stated bound $\mathcal{O}(n \cdot m)$.