**COMPSCI 330: Design and Analysis of Algorithms** 02/08/2017 - 02/20/2017

# Graph Representations and Traversal

*Lecturer: Debmalya Panigrahi* *Scribe: Tianqi Song, Fred Zhang, Tianyu Wang*

## 1 Overview

This lecture covers basic definitions about graph, graph representations, Depth First Search (DFS), Breadth First Search (BFS) and introduction to shortest paths.[1]

## 2 Definitions

A *graph* $G = (V, E)$, where $V$ is the set of vertices, and $E$ is the set of edges. An edge $e \in E$ is an unordered pair $(u, v)$ in undirected graphs, where $u, v \in V$. In directed graphs, an edge $e$ is an ordered pair. A *path* from a vertex $u$ to a vertex $v$ is a sequence of vertices $(w_0, w_1, ...w_k)$, where $u = w_0$, $v = w_k$ and $(w_{i-1}, w_i) \in E$ for all $1 \leq i \leq k$. The path is a *cycle* if $u = v$. The *length* of a path in an unweighted graph is the number of edges on the path. The *length* of a path in a weighted graph is the sum of the weights of all edges on the path. The *distance* between two vertices $u$ and $v$ is defined as the length of *the shortest path* connecting them.

## 3 Graph Representations

*Adjacency Matrix*: For a graph $G = (V, E)$, we number the vertices as $1, 2, ..., |V|$. Construct a $|V| \times |V|$ matrix $A$, where $A[i, j] = 1$ if $(i, j) \in E$, otherwise $A[i, j] = 0$. The space requirement of the adjacency matrix is $\Theta(|V|^2)$.
*Adjacency List*: For a graph $G = (V, E)$, we construct an array of $|V|$ lists and one for each vertex. A vertex $v$ is in the list of a vertex $u$ iff $(u, v) \in E$. The space requirement of the adjacency list is $\Theta(|V| + |E|)$.

## 4 Applications

There are many applications of graphs such as:

- *Map Coloring*: Map coloring is the problem of coloring the vertices of a graph such that no two neighboring vertices have the same color. The name stems from the old problem of coloring countries on a geographical map such that the countries' borders can be distinguished by using different colors for neighboring countries. In that problem, each vertex represents a country, and there is an edge between two countries whenever they share a border.
- *Facebook Graph*: A node represents a person and an edge represents the friendship between two people. A triangle mean the three people mutually know each other. A node $v$ whose neighboring nodes not in most cases not mutually connected can represent an advertising page.

[1] Most of the material in this note is from previous notes by Ang Li, Roger Zou and Wenshun Liu for this class in Fall 2014.

- *PageRank*: A node represents a webpage and an edge represents a hyperlink. A node that has a lot of incoming edges (meaning that a lot of webpages have links to this page) can represent an important or popular webpage. On the other hand, A node that has a lot of outgoing edges (meaning that this page directs to a lot of other pages) can represent an advertising webpage.

# 5 Depth-first Search (DFS) and DFS tree

## 5.1 Depth-first Search (DFS)

The depth-first search is the first graph search algorithm we see. The algorithm starts at some root vertex, which can be arbitrarily chosen, then explores as far as possible before backtracking. We will keep two global arrays $s$ and $f$ for starting time and finish time for each node. We also keep a global variable $t$ to record the current time.

**Algorithm 1** Depth first search (DFS)

```
1: function DFS(G)
2:     t = 0
3:     for each vertex v ∈ G.V do
4:         if v is unmarked then
5:             DFS-Trav(G, v)
```

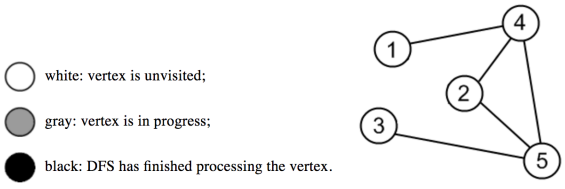
**Algorithm 2** Depth first search (DFS) from a source vertex

```
1: function DFS-TRAV(G, v)
2:     t = t + 1
3:     s[v] = t
4:     Mark v
5:     for each w such that (v, w) ∈ G.E do
6:         if w is unmarked then
7:             DFS-Trav(G, w)
8:     f[v] = t
```

The running time of DFS is $O(|V|+|E|)$ using adjacency list, where $G=(V,E)$.

### An example of DFS

Consider the problem speficified by the following figure.




(a) the natations we conform to (b) graph of interest

The process of the DFS is illustrated in Figure 2. DFS proceed from left to right on each row and from top to bottom row by row.

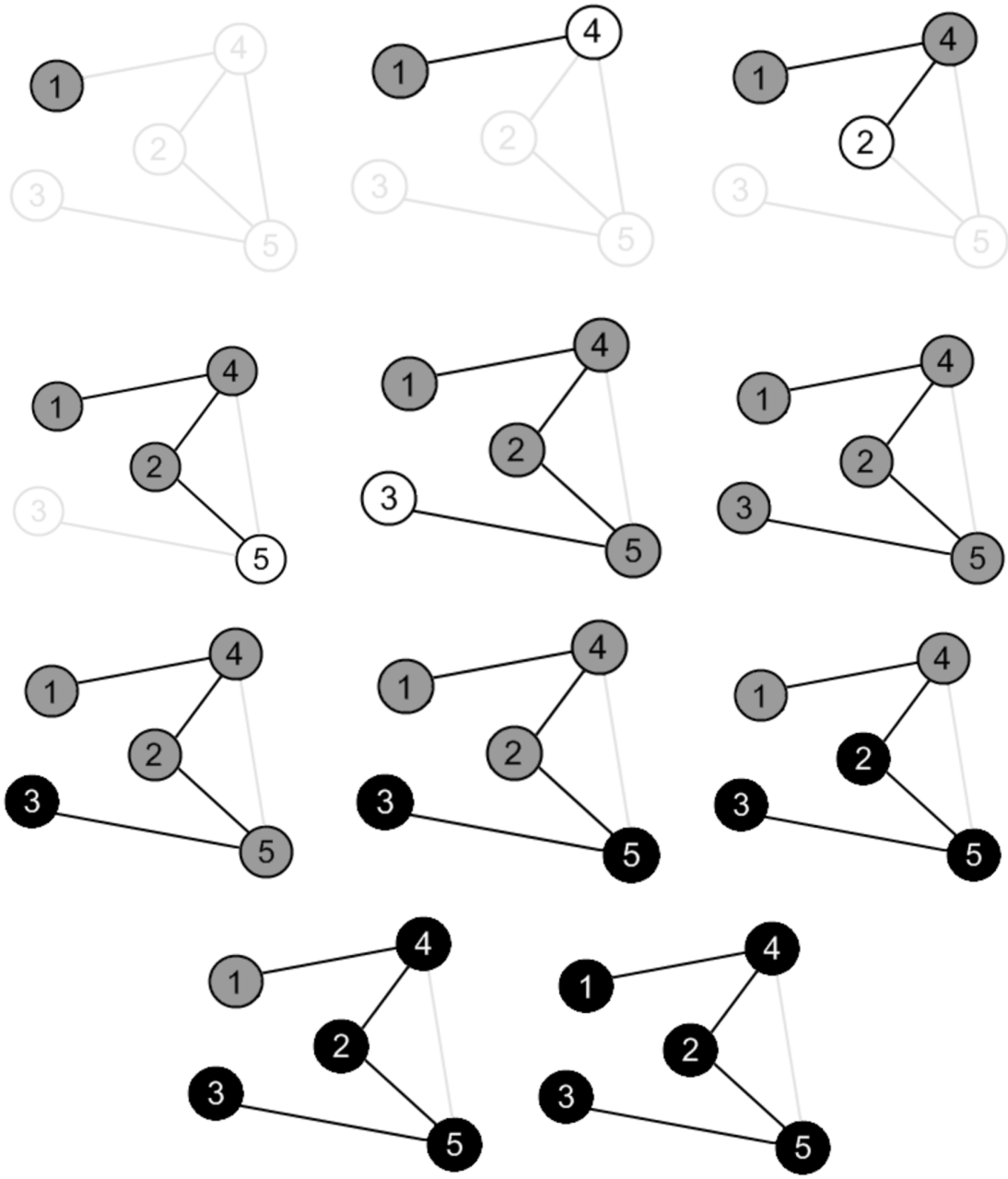



Figure 2: DFS example

## 5.2 Pre Order and Post Order

$s[v]$ records the *pre order* of vertex $v$ and $f[v]$ records the *post order* of vertex $v$. When we use a stack to keep track of the order of node visiting. The *pre order* of a vertex is the time when the vertex is pushed into

the stack during DFS. The *post order* of a vertex is the time when the vertex is popped out of the stack.

### 5.3 DFS tree and edge classification

A DFS tree can be constructed during DFS. An edge $(u,v)$ is in the tree if a vertex is found for the first time by exploring edge $(u,v)$. There are four types of edges clarified by DFS tree: tree edge, forward edge, cross edge, back edge as shown in Figure 4. A forward edge is an edge from a vertex to one of its descendants that are not tree edge. A cross edge is an edge from a vertex $u$ to a vertex $v$ such that the subtrees rooted at $u$ and $v$ have no overlap. A back edge is an edge from a vetex to one of its ancestors. Table **??** is a chart of edge types with their corresponding *pre*/*post* orders.

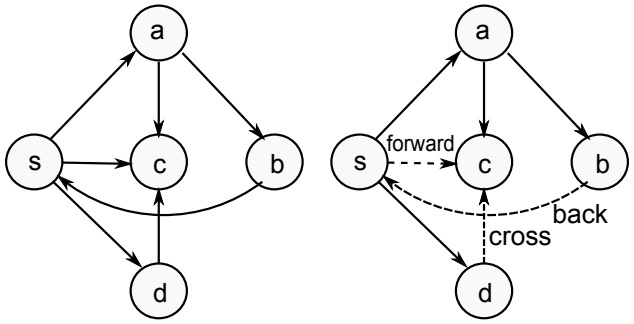



Figure 3: A graph is shown on the left. We run DFS on the graph using vertex $s$ as the source and the edges can be categorized into four types as shown on the right: the solid edges are tree edges.

| **Edge Type** $(u,v)$ | ***pre/post* order** |
|---|---|
| Tree/forward | $pre(u) < pre(v) < post(v) < post(u)$ |
| Back | $pre(v) < pre(u) < post(u) < post(v)$ |
| Cross | $pre(v) < post(v) < pre(u) < post(u)$ |

Table 1: Edge Type and *Pre/Post* Order

## 6 Cycle Detection in Directed Acyclic Graph (DAG)

Now, let we have a look at some applications of DFS. A simple application is cycle detection in DAG. Notice that this problem reduces to back edge detection!

## 6.1 Algorithm for Cycle Detection

**Algorithm 3** Cycle Detection

```
1: function CYCLE-DETECTION(G = (V, E))
2:     for v ∈ V do
3:         v.color ← white
4:         v.p ← null
5:     for v ∈ V do
6:         if v.color = white then
7:             DFS-OUPUT(v)
8:     Output: "No cycles"
```

**Algorithm 4** DFS-cycle-detection

```
1: function DFS-OUTPUT(v, G)
2:     v.color ← gray
3:     for u is a neighbour of v do
4:         if u.color = white then
5:             u.p ← v
6:             DFS-OUTPUT(u, G)
7:         else
8:             Output "One cycle found"
9:     u.color ← black
```

## 6.2 Correctness

The correctness relies on the following thoerem.

**Theorem 1.** *A directed graph G is acyclic if and only if DFS on this graph has no back edges.*

*Proof.* For one direction, observe that if there exists a back edge in the DFS tree, then there is a cycle in the graph. For the other direction, suppose that there exists a cylce and let vertex $v$ be the first vertex in the cylce that is visited by DFS. Then all other vertices in the cycle are descendents of $v$ in the DFS tree including a vertex $u$ such that edge $(u,v)$ exists in the graph, which results in a back edge. □

With Theorem 1 and Table 1, there are cycles $iff$ there are back edges and the algorithm correctly identifies back edges. Therefore the algorithm is correct.

## 6.3 Running Time

The algorithm is a modified version of DFS. Therefore it runs in time $O(m+n)$, same as the DFS.

# 7 Topological Sorting for Directed Acyclic Graph (DAG)

Next, we will see an application of DFS. The idea of *topological sorting* is based on this observation that all edges in a DFS tree of a DAG point to the same direction—there is no edge pointing backward. It is defined as follows.

**Definition 1.** *A topological sort or topological ordering of a directed graph is a ordering of its vertices such that for every directed edge $(u,v)$ from vertex u to vertex v, u comes before v in the ordering.*

## 7.1 Algorithm for Topological Sorting

To get a topological sorting of a DAG, one can run a DFS and list the vertices in decreasing order of their post order values. The pseudocode can be written as follows.

**Algorithm 5** Topological Sorting Algorithm

```
1: function TopologicalSort(G)
2:     Intiate an empty list L.
3:     Start a DFS on G.
4:     for each vertex do
5:         During the DFS, when it is finished and poped out of stack, insert it onto the front of L.
6:     return the list L.
```

To give an example of how the algorithm runs: In this example, the first backtrack occurs at vertex $h$, which

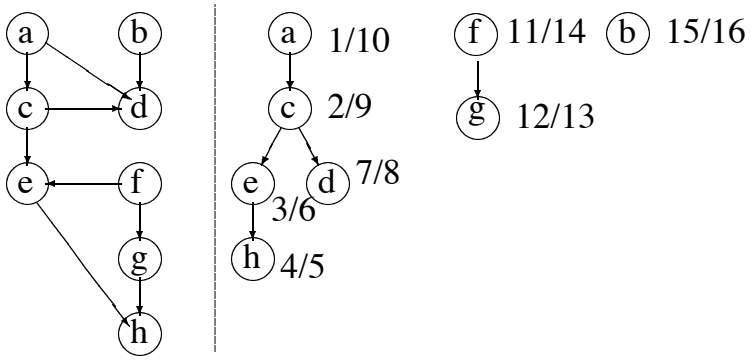



Figure 4: On the left side is the orignal graph and the right side is the DFS forest with vertices and their pre-order and post-order values. Here, we get a forest, since in a directed graph, one round of DFS starting from an arbitrary vertex may not be able to visit all vertices.

is finished and popped off stack first. The algorithm puts it to the front of the list and continues to backtrack until vertex $a$ is finished. Then the algorithm starts a DFS at $f$, only exploring $f$ and $g$ and prepending them onto the head of list. Finally, vertex $b$ is left unvisited, and it has the largest post-order time stamp. Note the ordering lists the vertices precisely by their post-order values.

## 7.2 Correctness

By Theorem 1 and Table 1, the for every edge $(u,v)$ in a DAG, the finishing time of $u$ is greater than that of $v$, as there are no back edges and the remaining three classes of edges have this property. Therefore the

algorithm outputs the correct ordering.

### 7.3 Running Time

In addition to running the DFS, the algorithm performs a constant-time operation on each vertex, that is, inserting them onto a list. This part, however, is dominated by the running time of DFS. Hence, the algorithm runs in time $O(m+n)$, same as the DFS.

## 8 Connectivity and Strong Connectivity

**Definition 2.** *In an undirected graph $G = (V,E)$, a vertex $u$ is connected to a vertex $v$ if there exists a path from $u$ to $v$.*

**Definition 3.** *In a directed graph $G = (V,E)$, a vertex $u$ is **weakly** connected to a vertex $v$ if there exists a path from $u$ to $v$.*

**Definition 4.** *In a directed graph $G = (V,E)$, a vertex $u$ is **strongly** connected to a vertex $v$ if there exists a path from $u$ to $v$, and a path from $v$ to $u$.*

Strong connectivity of vertices in a directed graph $G = (V,E)$ is an *equivalence relation* because it's

- reflexive, meaning that any vertex is in its own connected component.
- transitive, meaning that if $u$ is in the same connected component as $v$ and $v$ is in the same connected component as $w$, then $u$ is in the same connected component as $w$.
- symmetric, meaning that for any two vertices $u$ and $v$, if $u$ is in the same connected component as $v$, then $v$ is in the same connected component as $u$.

### 8.1 Strongly Connected Components (SCC)

**Definition 5.** *In a directed graph $G = (V,E)$, a Strongly Connected Component (SCC) is a maximal subset $C \subseteq V$ s.t. any two vertices $u,v \in C$ are strongly connected.*

Let us first point out some defintions and observations, and then design an algorithm to find SCCs. First, the vertex set of any directed graph can be partitioned into SCCs. If we contract each SCC into a new node, then we get a DAG of SCCs. A *source* SCC is a SCC that has no edges going into it. A *sink* SCC is a SCC that has no edges going out.

### 8.2 An Efficient Algorithm for SCCs

To come up with an efficient algorithm to find all the SCCs of a directed graph, observe that DFS from any vertex $v$ in a sink SCC $C_{\mathsf{sink}}$ only visits vertices in $C_{\mathsf{sink}}$. Thus, if we have some way of finding such a vertex $v$, we can do DFS from $v$, delete all vertices DFS visits, and repeat. Yet, the issue is that finding such a vertex $v$ seems non-trivial. The solution lies in the fact that if we run DFS on a directed graph $G$, the vertex with latest finishing time belongs to a source SCC. Then by reversing the direction of all edges in $G$ to construct a new graph $G^R$, a source SCC in $G^R$ corresponds to a sink SCC in $G$.

**Algorithm 6** Strongly Connected Components

1: **function** SCC($G$)
2: Reverse all edges in $G$ to construct graph $G^R$.
3: Run DFS on $G^R$, keep a *finishing time* table, and find the vertex $v$ with the latest *finishing time*.
4: Run DFS from $v$ on $G$.
5: Let $C_i$ be the set of all vertices visited by DFS starting from $v$. Set $V \leftarrow V \setminus C_i$ and remove related edges.
6: Check the *finishing time* table for the vertices remaining in $V$ and identify the vertex $v$ with the latest *finishing time*. Go to line 4. Repeat until $V = \emptyset$.

## 8.3 Correctness

**Lemma 2.** *For two strongly connected components $C$ and $D$ of a graph $G$ s.t. there is an edge $(i, j)$ with $i \in C$ and $j \in D$. Let $f(v)$ denote the finishing time of vertex $v$ in some execution of DFS on the reversed graph $G^R$. Then*

$$\max_{v \in C} f(v) < \max_{v \in D} f(v)$$

*Proof.* Recall that the equivalence relation defined by the SCCs is symmetric. Therefore $G$ and $G^R$ have the same SCCs. Consider two SCCs $C$ and $D$ such that there exists an edge $(j, i)$, with $j \in S$ and $i \in C$. Let $v$ be the first vertex of $C \cup D$ visited by DFS in $D^R$. There are now two cases.
Case 1, suppose that $v \in C$. Since there is no cycle of SCCs, there is no directed path from $v$ to $D$ in $G^R$. Since DFS discovers everything reachable and nothing more, it will finish exploring all vertices in $C$ without reaching any vertices in $D$. Thus, every finishing time in $C$ will be smaller that every finishing time in C2, and this is even stronger than the assertion of the lemma.
Case 2, suppose that $v \in D$. Without loss of generality, let the DFS execution starts at $v$. Since DFS discovers everything reachable and nothing more, the call to DFS at $v$ will finish exploring all of the vertices in $C \cup D$ before ending. Thus, the finishing time of $v$ is the latest amongst vertices in $C \cup D$, and in particular is larger than all finishing times in $C$. This completes the proof. □

**Theorem 3.** *In Algorithm 6, whenever DFS is called on a vertex v, the vertices explored - and assigned a common leader - by this call are precisely those in v's SCC in G.*

*Proof.* We prove this by induction. Let $S$ denote the vertices already explored by previous calls to DFS (initially empty). Inductively, the set $S$ is the union of zero or more SCCs of $G$. Suppose DFS is called on a vertex $v$ and let $C$ denote $v$'s SCC in G. Since the SCCs of a graph are disjoint, $S$ is the disjoint union of SCCs of G, and $v \notin S$, no vertices of $C$ lie in $S$ (by the transitivity of strong connectivity). Thus, this call to DFS will explore, at the least, all vertices of C. By Lemma 2 , every outgoing edge $(i, j)$ from $C$ leads to some SCC $C'$ that contains a vertex $w$ with a finishing time larger than f(v). Since vertices are processed in decreasing order of finishing time, $w$ has already been explored and belongs to $S$; since $S$ is the union of SCCs, it must contain all of $C'$. Summarizing, every outgoing edge from $C$ leads directly to a vertex that has already been explored. Thus this call to DFS explores the vertices of $C$ and nothing else. This completes the inductive step and the proof of correctness. □

Intuitively, when DFS is invoked on $G$, processing vertices in decreasing order of finishing times, the successive calls to DFS peel off the SCCs of the graph one at a time, like layers of an onion.

### 8.4 Running Time

Constructing $G^R$ from $G$ (line 2) can be performed in $O(m)$. Finding a vertex in the *sink* SCC can be done in $O(m)$. These two steps can be done before the main loop (line 4-6). Each iteration $i$ of line 4-6 takes $O(m_i)$, where $m_i$ is the size of $C_i$. Because once a vertex and associated edges are deleted they can never be visited again, this means that $O(\sum_i m_i) = O(m)$. Thus the worst case running time is $O(m)$.

## 9 Breadth First Search (BFS)

### 9.1 Description

Besides depth-first search, there is another important graph search algorithm, called *breath-first search* (BFS). In BFS, we will start with some vertex, visit all its neighbors, and then recurse on each one of them. The procedure captures the intuition of "breadth-first", since it exlores neighboring vertices before going any further. The algorithm can be implemented by queue, instead of stack as in DFS, as follows.

**Algorithm 7** Breadth First Search (BFS)

```
1: function BFS(G, s)
2:     mark s.
3:     enqueue s in Q.
4:     while Q ≠ ∅ do
5:         dequeue v from Q.
6:         for all neighbors u of v do
7:             if u is unmarked then
8:                 mark u.
9:                 enqueue u in Q.
```

Similar to DFS, BFS also enables the idea of a BFS tree. A BFS tree can be constructed during a BFS. An edge $(u,v)$ is in the tree if a vertex is found for the first time by exploring edge $(u,v)$.

**Lemma 4.** *A BFS tree does not contain any forward edges.*

*Proof.* We prove the lemma by contradiction. If such a *forward edge* $(u,v)$ existed, $v$ would have been put into the queue at an earlier time. The edge $(u,v)$ would have become a *tree edge*. □

### 9.2 Running Time

The algorithm visits every vertex once and checks every edge once. Hence, the running time of BFS is $O(n+m)$, where $n$ is the number of vertices and $m$ is the number of edges in the graph.

### 9.3 Application: Shortest Path

One natural application of BFS is to find shortest path on unweighted graph. We simply let the BFS tree mark the distance from the source vertex to any reachable vertex. We expand the above pseudocode such that distances can be recorded:

**Algorithm 8** BFS tree with distances recorded

```
1: function BFST(G = (V,E), s)
2:     mark(s)
3:     dist(s)= 0
4:     dist(v)= ∞ for v ∈ V and v ≠ s
5:     enqueue s in Q
6:     while Q ≠ ∅ do
7:         dequeue v from Q
8:         for all neighbors u of v do
9:             if u is unmarked then
10:                mark(u)
11:                put edge (v,u) in the BFS tree
12:                dist(u)= dist(v)+1
13:                enqueue u in Q
```

After we run Algorithm 8, dist($u$) record the shortest path from $u$ to $s$. In an unweighted graph, length of path is measured by the number of edges in that path. And the shortest path from $u$ to $v$ is the smallest length of all paths from $u$ to $v$ (the distance from $u$ to $v$ is defined to be $\infty$ if there is no path from $u$ to $v$).
In the next section, we will study briefly shortest path on weighted graph.

# 10 Shortest Paths

BFS computes shortest paths in unweighted graphs. However, for weighted graphs, BFS is not an efficient algorithm to find shortest paths. If we have to use BFS to find shortest paths on weighted graphs, we need to first reconstruct the graph (split some edges into multiple segments) so that each edge has unit weight. Moreover, this method will only work for positively weighted graphs. Dijkstra's algorithm is able to find the shortest paths from a source vertex to all vertices for any graph with non-negative edges.

**Algorithm 9** Dijkstra's Algorithm

```
1: function DIJKSTRA(G, s)
2:     dist[s] = 0;
3:     dist[v] = +∞ for all v ≠ s
4:     add all vertices to a heap H
5:     while H ≠ ∅ do
6:         v = EXTRACT_MIN(H)
7:         for all edges (v,u) do
8:             if dist[u] > dist[v] + l(u,v) then
9:                 dist[u] = dist[v] + l(u,v)
```

## 10.1 Running Time Analysis

The running time of Dijkstra's Algorithm is $O((m+n)\log n)$ using a heap, where $n$ is the number of vertices and $m$ is the number of edges in the graph. With a Fibonacci heap, the running time can be reduced to $O(m+n\log n)$.