## Content

<table>
<thead>
<tr>
<th></th>
<th>Algorithms</th>
<th>Assignment and Knapsack problem</th>
<th>Bin Packing</th>
<th>Scheduling</th>
<th>Graph algorithms</th>
<th>Linear Programming</th>
<th>Heuristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5.1 Definitions</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5.2 Depth-First-Search (DFS) and Breadth-First-Search (BFS)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5.3 Topological Sorting (TopSort)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5.4 Single-Source-Shortest-Paths in Networks (SSSP)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Definitions
Graphs: Applications (1)

Graphs are important for a lot of different applications. Graphs capture relationships between objects in a visual way.
Graphs: Applications (2)

Graphs are important for a lot of different applications. Graphs capture relationships between objects in a visual way.

Quelle: Simchi-Levi et al., Designing & Managing the Supply Chain, 2007
Historical Background

1736: Leonhard Euler
„7 Bridges of Königsberg“
Do you think we can find a path that crosses each bridge once?
Historical Background

1736: Leonhard Euler
„7 Bridges of Königsberg“
Do you think we can
find a path that crosses
each bridge once?

Euler’s idea: It doesn't matter
where on the north side you are.
You must come and go via a bridge.
Collapse the entire area to a point.
Graphs: Definitions (1)

Directed Graph \( G = (V, E) \)
- \( V \) = finite set of vertices, \( n=|V| \), \( E \) = finite set of edges, \( m=|E| \)
  Ex. \( V=\{A, B, C, D\} \), \( E=\{e_1, e_2, e_3, e_4, e_5\} \),
  \( e_1=(D,B) \), \( e_2=(B,C) \), \( e_3=(C,D) \), \( e_4=(D,A) \), \( e_5=(B,D) \)

Edge \( e = (u,v) \)
- \( u \) is called adjacent from \( v \) and \( v \) is called adjacent from \( u \)
- \( u \) is called the initial vertex of \( (u,v) \), and \( v \) is called the terminal vertex of \( (u,v) \)
- \( e \) connects \( u \) and \( v \) and is called incident with the vertices \( u \) and \( v \)
- if \( u=v \), then \( e \) is a loop
- \( \text{indeg}(v) = \text{deg}^-(v) = \text{in-degree of } v = \) number of edges with \( v \) as their terminal vertex (a loop contributes 1)
  if \( \text{indeg}(u) = 0 \), then \( u \) is a source
  if \( \text{outdeg}(v) = 0 \), then \( v \) is a sink
- \( \text{outdeg}(v) = \text{deg}^+(v) = \text{out-degree of } v = \) number of edges with \( v \) as their initial vertex (a loop contributes 1)
- A path \( x_0, x_1, \ldots, x_n \) from \( u=x_0 \) to \( v=x_n \) of length \( n \geq 1 \) in a directed graph
  is a sequence of edges \( (x_0, x_1), (x_1, x_2), \ldots, (x_{n-1}, x_n) \).
  Ex. B, C, D, A is a path of length 3 with the edges (B,C), (C,D), (D,A)
- The path is a circuit/cycle if it begins and ends at the same vertex and has length greater or equal to 1.
Graphs: Definitions (2)

Undirected Graph \( G = (V, E) \)

If there is an edge from \( u \) to \( v \), then there is also an edge from \( v \) to \( u \). In this case, we don't need arrows on the edges (and the edges are assumed to go both ways):

- \( V = \) finite set of vertices, \( n = |V| \), \( E = \) finite set of edges, \( m = |E| \)
  
Ex. \( V = \{A, B, C, D\} \), \( E = \{ e_1, e_2, e_3, e_4 \} \), \( e_1 = \{B,D\} \), \( e_2 = \{B,C\} \), \( e_3 = \{C,D\} \), \( e_4 = \{A,D\} \)

Edge \( e = \{u,v\} \)

- \( u \) and \( v \) are called adjacent or neighbors in \( G \)
- \( u \) and \( v \) are the endpoints of the edge \( e = \{u,v\} \), \( e \) connects \( u \) and \( v \) and is called incident with the vertices \( u \) and \( v \)
- (Rosen) there are loops allowed in undirected graphs
- \( \text{deg}(v) = \) degree of \( v = \) number of edges incident with \( v \) (a loop contributes twice)
- A path \( x_0, x_1, \ldots, x_n \) from \( u = x_0 \) to \( v = x_n \) of length \( n \geq 1 \) in an undirected graph is a sequence of edges \( \{x_0, x_1\}, \{x_1, x_2\}, \ldots, \{x_{n-1}, x_n\} \).
  
A path is simple if it does not contain the edge more than once.
  
Ex. B, D, C is a simple path of length 2 with the edges \( \{B,D\}, \{D,C\} \)
- The path is a circuit/cycle if it begins and ends at the same vertex and has length greater or equal to 1.
An undirected graph $G$ is connected if there is a path between every pair of distinct vertices of the graph. There is a simple path between every pair of distinct vertices of a connected undirected graph. Disconnected graphs can be broken up into pieces where each is connected. These pieces are called Connected Components.

A directed graph is weakly connected if there is a path between any two vertices in the underlying undirected graph. A directed graph is strongly connected if there is a path from $a$ to $b$ and from $b$ to $a$ whenever $a$ and $b$ are vertices in the graph.
A tree is a connected undirected graph with no simple cycles.
- Any tree is a simple graph.
- Any tree has a unique simple path between any two of its vertices.
- A tree with $n$ vertices has $n-1$ edges.

A rooted tree is a tree in which one vertex has been designated as the root and each edge is directed away from the root.

If $v$ is a vertex other than the root, the parent of $v$ is a unique vertex $u$ such that there is a directed edge from $u$ to $v$. When $u$ is the parent of $v$, $v$ is called a child of $u$.

A vertex of a rooted tree is called a leaf if it has no children. Vertices that have children are called internal vertices.

Which of them are Trees?

(a)  
(b)  
(c)  
(d)
Trees: Definitions (2)

The **level** of a vertex $v$ in a rooted tree is the length of the unique path from the root to its vertex. The level of the root is defined to be zero.

The **height** of a rooted tree is the maximum of the levels of vertices.

A **binary tree** is a rooted tree where every vertex has no more than two children. There are at most $2^h$ leaves in a binary tree of height $h$.

A binary tree of height $h$ is **balanced** if all leaves are at levels $h$ or $h-1$.

In a **full binary tree**, every internal vertex has exactly two children. A full binary tree with $i$ internal vertices contains $n=2i+1$ vertices.
Max. Number of Edges in (Un-)Directed Graphs

If $G$ is a simple undirected graph with $n$ vertices, what is the maximum number $m$ of edges that $G$ can have?

A) $n^2$
B) $n^2/2$
C) $n(n-1)/2$
D) $n(n+1)/2$
E) $n$

If $G$ is a directed graph with $n$ vertices, what is the maximum number of ordered pairs of vertices $(v,w)$ that could be connected by edges in $G$?

A) $n$
B) $2n$
C) $n^2$
D) $n(n-1)/2$
E) $2^n$

Proofs by Induction.
Representing Graphs

Adjacency Matrix:

\[
G = \begin{pmatrix}
1 & 2 & 3 & 4 & 5 & 6 \\
1 & 0 & 1 & 1 & 0 & 0 \\
2 & 0 & 0 & 0 & 0 & 0 \\
3 & 1 & 0 & 0 & 1 & 0 \\
4 & 1 & 0 & 0 & 0 & 0 \\
5 & 0 & 0 & 1 & 0 & 1 \\
6 & 0 & 1 & 0 & 1 & 1 \\
\end{pmatrix}
\]

\[
O(n^2) \text{ space}
\]
Representing Graphs

Adjacency List:

\[ O(n+m) \text{ space} \]
DFS and BFS
Distance

Definition: In a graph $G$ with at least two vertices $u$ and $v$, the distance from $u$ to $v$ is the length of the shortest path from $u$ to $v$.

If there is no path from $u$ to $v$ then we say that the distance is infinite.

The distance from a vertex to itself is 0.
BFS: algorithm

**procedure BFS(G, s)**

Input: Graph $G = (V,E)$, (directed or undirected) and a vertex $s$ in $V$.
Output: For all vertices $u$ reachable from $s$, $dist(u)$ is the distance from $s$ to $u$.
and for all vertices $u$ not reachable from $s$, $dist(u) = \infty$.

for each vertex $u$ in $V$:
    $dist(u) = \infty$

$dist(s) = 0$

$Q = [s]$ (queue just containing $s$)

while $Q$ is not empty
    $u = eject(Q)$
    for all edges $(u,v)$ in $E$
        if $dist(v) = \infty$ then
            $inject(Q,v)$
            $dist(v) = dist(u) + 1$

$eject = \text{de-queue}$

$inject = \text{en-queue}$

---

C E A S D B
Correctness of BFS

Loop Invariant: After the $d$-th iteration through the while-loop,
(1) all nodes at distance $\leq d$ from $s$ have their distances correctly set
(2) all other nodes have their distances set to infinity
(3) the queue contains exactly the nodes at distance $d$.

Proof (by induction):
Induction Base: $d=0$, dist($s$)=0.
Inductive Hypothesis: same as Loop Invariant for $d$-1.
Inductive Step:
During the $d$-th run through the while-loop, vertex $u$ with distance $d$-1 from $s$ is ejected.
Among all adjacent vertices $v$ of $u$, only those with dist($v$)=infinity are injected and the distance of these vertices is set to dist($v$) = dist($u$)+1 = $d$-1 + 1 = $d$. 
BFS: Runtime

Procedure BFS(G, s)
Input: Graph G = (V, E), (directed or undirected) and a vertex s in V.
Output: For all vertices u reachable from s, dist(u) is the distance from s to u.
and for all vertices u not reachable from s, dist(u) = ∞

for each vertex u in V:
    dist(u) = 0
Q = [s] (queue just containing s)
while Q is not empty
    u = eject(Q)  eject = de-queue
    for all edges (u, v) in E
        if dist(v) = ∞ then
            inject(Q, v) inject = en-queue
            dist(v) = dist(u) + 1

n = |V|, m = |E|
n initialization
2n injections and ejections
2m look at all edges twice

→ O(n + m)
BFS: Shortest Path Tree

procedure BFS(G, s)
Input: Graph G = (V,E), (directed or undirected) and a vertex s in V.
Output: For all vertices u reachable from s, dist(u) is the distance from s to u. and for all vertices u not reachable from s, dist(u) = ∞
for each vertex u in V:
    dist(u) = ∞
    dist(s) = 0
    Q = [s] (queue just containing s)
    while Q is not empty:
        u = eject(Q)
        for all edges (u,v) in E
            if dist(v) = ∞ then
                inject(Q,v)
                dist(v) = dist(u) + 1

Question: Is there a way to output the dist-values and the shortest path tree?
BFS: Shortest Path Tree

procedure BFS(G, s)
Input: Graph G = (V,E), (directed or undirected) and a vertex s in V.
Output: For all vertices u reachable from s, dist(u) is the distance from s to u.
and for all vertices u not reachable from s, dist(u) = ∞

for each vertex u in V:
    dist(u) = ∞
    dist(s) = 0 \textbf{prev}(u)=\textbf{nil}
    Q = [s] (queue just containing s)
while Q is not empty
    u = eject(Q)
    for all edges (u,v) in E
        if dist(v) = ∞ then
            inject(Q,v)
            dist(v) = dist(u) + 1
            \textbf{prev}(v)=u
    eject = de-queue
    inject = en-queue

Question: Is there a way to output the dist-values and the shortest path tree?
Depth First Search

Depth First Search (DFS) is a method to explore a graph in a systematic way. DFS has been popularized by R. Tarjan (Depth-first search and linear graph algorithms, SIAM Journal on Computing 1 (1972), 146-160).

NOTE: (HERE) DIRECTED GRAPHS!

```
procedure DFS(G)
    t=0;
    for all u∈V do
        if color(u)=white then DFS-Visit;
    procedure DFS-Visit(u)
        color(u)=gray; t++; d[u]=t;
        for all v∈V with (u,v)∈E do
            if color(v)=white then DFS-Visit(v);
        color(u)=black;
```

Homework:

Correctness.
Find the loop invariant and prove the correctness of DFS(G).

Time Analysis.
The running time of DFS(G) is \(O(n+m)\).
**Depth First Search**

\[
\text{DFS}(u)
\]

1. Mark the "neighbor vertex" \( v \) of vertex \( u \) "with the smallest number" (if it hasn't been marked already).

2. Repeat Step 1 until all vertices have been marked.
Distance and Stack vs. Queue

- The main difference between DFS and BFS is that DFS uses a stack and BFS uses a queue.
- The queue gives us some extra information.
- The queue starts with just the node s, the only one that has distance 0. Then for each subsequent distance, there is a point when the queue contains all the nodes at distance d and nothing else. As these nodes are ejected, their undiscovered neighbors are the next nodes injected into the end of the queue.
DFS vs. BFS

- DFS: determine if a graph has a circuit  
  (EXERCISE)
- DFS: find strongly connected components of a directed graph  
  (EXERCISE)
- DFS: find what vertices can be reached by a given vertex
- DFS: find a spanning tree of a graph
- DFS: find a path from a vertex s to a vertex v
- DFS: carry out a topological sort of a graph
- DFS: find sinks and sources in directed acyclic graphs
- DFS gives info about the whole graph
- DFS uses a stack
- DFS is not good for finding shortest distances between vertices
- BFS gives info related to a given vertex within the graph
- BFS uses a queue
- BFS does not restart at other connected components since all vertices not connected to your starting vertex are distance infinity away
Topological Sorting
Topological Sorting

Applications:
- Project Management
- Shortest Paths in Networks

Precondition:
- directed graphs that have no cycles are called directed acyclic graphs, or DAGs
- every DAG has a source
- G is a DAG if and only if G–v is a DAG

Topological Ordering:
- In a topological ordering, it must be true that for every edge v → w, v comes before w in the ordering.

A must be finished before B starts and so on.
Can this project be accomplished?

Whenever there's a cycle, we can't find a prerequisite ordering.
Topological Sorting: Motivation

Topological Sort =
construction of a total ordering
compatible with a partial ordering

Partial Ordering:
a relation that is
reflexive \((a,a) \in R\)
antisymmetric \((a,b) \in R \text{ and } (b,a) \in R \Rightarrow a=b\)
transitive \((a,b) \in R \text{ and } (b,c) \in R \Rightarrow (a,c) \in R\)

Total Ordering:
in addition
either \((a,b) \in R \text{ or } (b,a) \in R\)

Examples:
,,is older than“ is no partial ordering (not reflexive)
,,Divides“ is no total ordering (5 does not div 7 and 7 does not div 5) but partial \((r, a, t)\)
Example for a total ordering:,,\leq"
Topological Sorting: Motivation

A before B (B can only be completed after A has been finished)
B before C
C before D
D before B

A must be finished before B starts and so on.
Can this project be accomplished?

Whenever there's a cycle, we can't find a prerequisite ordering.
Topological Sorting


Let S be a partial ordered set.
TopSort: Example

Activities:
A: Building Walls
B: Roof Timbering
C: Roof Tiling
D: Rendering inside
E: Rendering outside

Dependencies:
A before B, C, D, E
B before C, D, E
C before D, E
TopSort: Example

Activities:
A: Building Walls
B: Roof Timbering
C: Roof Tiling
D: Rendering inside
E: Rendering outside

Dependencies:
A before B, C, D, E
B before C, D, E
C before D, E
TopSort: Example

Activities:
A: Building Walls
B: Roof Timbering
C: Roof Tiling
D: Rendering inside
E: Rendering outside

Implementation:
• Maintain an integer array, InDegree[], where for each i between 1 and n, InDegree[i] is the number of incoming edges to vertex i.
• Maintain a list of sources, S, either a stack or a queue.

Choose the first x in S;
for (each y adjacent to x)
    InDegree[y]--;
if (InDegree[y] == 0) add y to S;

Homework:
• Correctness
• Time Analysis
TopSort: Example

Activities:
A: Building Walls
B: Roof Timbering
C: Roof Tiling
D: Rendering inside
E: Rendering outside

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
</tr>
</tbody>
</table>
TopSort: Example

Activities:
A: Building Walls  
B: Roof Timbering  
C: Roof Tiling  
D: Rendering inside  
E: Rendering outside

A \rightarrow  B \rightarrow  C \rightarrow \{D, E\}

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>-1 = 0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td>-1 = 1</td>
<td>-1 = 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>3</td>
<td>-1 = 2</td>
<td>-1 = 1</td>
<td>-1 = 0</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>3</td>
<td>-1 = 2</td>
<td>-1 = 1</td>
<td>-1 = 0</td>
<td></td>
</tr>
</tbody>
</table>
TopSort: Example

Solution:
{A, G} -> {B, C, E} -> {D, H} -> {F, I}
SSSP
Single Source Shortest Paths

Compute the shortest paths from a single source $s$ to all other vertices in the network.

$\delta(B) =$ length of the shortest path from the source $s$ to $B$: distance from $s$ to $B$
Single Source Shortest Paths

Compute the shortest paths from a single source $s$ to all other vertices in the network.

$$\delta(B)=\text{length of the shortest path from the source } s \text{ to } B:$$

$$\text{distance from } s \text{ to } B$$

Question: Is this always possible? Find two possibilities where you actually cannot find a shortest path between $s$ and another vertex.
Single Source Shortest Paths

Compute the shortest paths from a single source $s$ to all other vertices in the network.

$\delta(B) =$ length of the shortest path from the source $s$ to $B$:

distance from $s$ to $B$

Question: Is this always possible? Find two possibilities where you actually cannot find a shortest path between $s$ and another vertex.

1. There is no path from $s$ to $v$ \[ \delta(v) = \infty \]
2. The path from $s$ to $v$ contains a negative cycle: \[ \delta(v) = -\infty \]
Single Source Shortest Paths

Compute the shortest paths from a single source s to all other vertices in the network.

\[ \delta(B) = \text{length of the shortest path from the source } s \text{ to } B: \]
\[ \text{distance from } s \text{ to } B \]

Lemma 1: For each edge \((u, v) \in E\) we have \(\delta(v) \leq \delta(u) + c(u, v)\).
Lemma 2: Subpaths of shortest paths are shortest paths.
Lemma 3: Let \((u, v)\) be the last edge on a shortest path \(P\) from \(s\) to \(v\).
Then: \(\delta(v) = \delta(u) + c(u, v)\).
Bellman / Dijkstra / Ford and their results

Richard Ernest Bellman (1920-1984, 63)
- New York City - Los Angeles
- Dynamic Programming
- Bellman, Richard, On a routing problem, Quarterly of Applied Mathematics 16, 87–90, 1958

Edsger Wybe Dijkstra (1930-2002, 72)
- Rotterdam - Nuenen
- 1972 Turing award
- Dijkstra, Edsger Wybe, A note on two problems in connexion with graphs, Numerische Mathematik 1, 269–271, 1959

Lester Randolph Ford, Jr. (1927)
- Houston
- Maximum Flow in Networks
- Ford Jr., Lester Randolph, Network Flow Theory, RAND Corporation: Paper P-923, Santa Monica, California, 1956
The algorithm was first proposed by Shimbel in 1955, but is instead named after Richard Bellman and Lester Ford, Jr., who published it in 1958 and 1956, respectively. Edward F. Moore also published the same algorithm in 1957, and for this reason it is also sometimes called the Bellman–Ford–Moore algorithm.
Bellman/Ford

Initialize\((G, s)\)
\[
d[s] = 0;
\]
\[\text{for all } (v \in V \setminus \{s\}) \text{ do } d[v] = \infty\]

Relax\((u, v)\)
\[
\text{if } (d[v] > d[u] + c(u, v)) \text{ then } d[v] = d[u] + c(u, v);
\]

BellmanFord\((G, s)\)
Initialize\((G, s)\);
for \((k = 1, \ldots, n - 1)\)
\[
\text{for all } (u, v) \in E \text{ Relax}(u, v);
\]

\(d[v]\) is the length of a path from \(s\) to \(v\),
\(d[s]=0\)
Bellman/Ford

Initialize\((G, s)\)
\(d[s] = 0;\)
for all \((v \in V \setminus \{s\})\) do \(d[v] = \infty\)

Relax\((u, v)\)
if \((d[v] > d[u] + c(u, v))\) then \(d[v] = d[u] + c(u, v);\)

BellmanFord\((G, s)\)
Initialize\((G, s)\);
for \((k = 1, \ldots, n - 1)\)
    for all \((u, v) \in E\) Relax\((u, v)\);

Running Time: \(O(nm)\) \(n - 1\) phases and each phase needs \(O(m)\) time

Correctness: When the algorithm terminates, we have \(d[v] = \delta(v)\) for all \(v \in V\) with \(\delta(v) > -\infty\).
Bellman for acyclic networks
Bellman for acyclic networks

Initialize($G, s$)
$d[s] = 0$;
for all $(v \in V \setminus \{s\})$ do $d[v] = \infty$

Relax($u, v$)
if $(d[v] > d[u] + c(u, v))$ then $d[v] = d[u] + c(u, v)$;

Bellman($G, s$)
Initialize($G, s$); Topsort($G$);
$V^* = V$;
while($V^* \neq \emptyset$)
    choose $u \in V^*$ that is next in topological order and delete $u$ from $V^*$;
    for all $(u, v) \in E$ Relax($u, v$);

Running Time: $O(n + m)$

Correctness: When the algorithm terminates, we have $d[v] = \delta(v)$ for all $v \in V$ with $\delta(v) > -\infty$.

$d[v]$ is the length of a path from $s$ to $v$, $d[s]=0$
Bellman: Example 1

1. TopSort
Bellman: Example 1

1. TopSort

\[ E \rightarrow B \rightarrow D \rightarrow F \rightarrow G \rightarrow C \rightarrow A \]
Bellman: Example 1

1. E → B → D → F → G → C → A

---

E © B © D © F © G © C © A
1. TopSort
Bellman: Example 2

1. TopSort

A → {B, E} → {C, F, I} → {D, G} → H
Bellman: Example 2

A → {B, E} → {C, F, I} → {D, G} → H
Question:
Why is the rule „continue at that vertex v that comes next in the topological order“ always optimal?
Bellman for acyclic networks

**Question:**
Why is the rule „continue at that vertex v that comes next in the topological order“ always optimal?

**Answer:**
1. Among all possible paths from s to v you always keep only the (so far) shortest path from v to s.
2. When you select a vertex v that comes next in the topological order you never can return to that vertex.

**Homework:**
- Correctness
- Time Analysis
Dijkstra for non-negative networks

Dijkstra
Dijkstra for non-negative networks

\[
\text{Initialize}(G, s) \\
\quad d[s] = 0; \\
\quad \text{for all } (v \in V \setminus \{s\}) \text{ do } d[v] = \infty
\]

\[
\text{Relax}(u, v) \\
\quad \text{if } (d[v] > d[u] + c(u, v)) \text{ then } d[v] = d[u] + c(u, v);
\]

\[
\text{Dijkstra}(G, s) \\
\quad \text{Initialize}(G, s); \\
\quad V^* = V; \\
\quad \text{while}(V^* \neq \emptyset) \\
\quad \quad \text{choose } u \in V^* \text{ with } d[u] \text{ minimal and delete } u \text{ from } V^*; \\
\quad \quad \text{for all } (u, v) \in E \quad \text{Relax}(u, v);
\]

Running Time: depends on how \( V^* \) is implemented.

Correctness: When the algorithm terminates, we have
\[
d[v] = \delta(v) \text{ for all } v \in V \text{ with } \delta(v) > -\infty.
\]

d[v] is the length of a path from s to v, 
d[s]=0
Dijkstra: Example 1

A  4  2  4  5  1  1  E
    √    √  √
B   1  3  2
    √    √  √
C
    √    √  √
D
    √    √  √
E  4  5  2
    √    √  √
F
    √    √  √
Dijkstra: Example 1

1. A
2. B
3. C
4. D
5. E
6. F

1. A/0
2. C/2
3. B/3
4. E/4
5. D/5
6. F/5

Graph showing weights and paths from A to each node.
Dijkstra: Example 2

The graph shown represents a network with weighted edges. The numbers on the edges indicate the weights of the connections between nodes. Here's a breakdown of the graph:

- **Nodes:** A, B, C, D, E, F, G, H, I
- **Edges and Weights:**
  - A to B: 2
  - A to F: 9
  - B to G: 6
  - G to H: 15
  - H to I: 4
  - H to F: 11
  - B to C: 4
  - C to I: 15
  - I to E: 1
  - E to D: 1
  - E to G: 3

This graph can be used to demonstrate the Dijkstra's algorithm for finding the shortest path in a weighted graph.
Dijkstra: Example 2

1. A/0
2. B/2
3. C/6
4. D/8
5. G/8
6. E/9
7. F/9
8. H/12
9. I/16

Dijkstra:
- A/0
- B/2
- C/6
- D/8
- E/9
- F/15
- G/15
- H/12
- I/16
- I/9
- I/21
- H/23
- I/10
- H/20
- G/8
- F/15
- H/12
- I/16

Example 2

A/0
B/2
C/6
D/8
E/9
F/15
G/15
H/12
I/16
Dijkstra for non-negative networks

**Question:**
Why is the rule „continue at that vertex \( v \) that has minimal distance from \( s \)“ always optimal?
Dijkstra for non-negative networks

**Question:**
Why is the rule „continue at that vertex \( v \) that has minimal distance from \( s \)“ always optimal?

**Answer:**
1. Among all possible paths from \( s \) to \( v \) you always keep only the (so far) shortest path from \( v \) to \( s \).
2. When you select a vertex \( v \) that has minimal distance from \( s \) then there might still be another path to that node - but because of the non-negativity never with a smaller distance.
Homework: Correctness and Time analysis

Literature: Dasgupta et al., p.110

procedure dijkstra(G, l, s)
for all u in V
    dist(u) := infinity
    prev(u) := nil
dist(s) := 0
H := makequeue(V)  \(1\) time
while H is not empty
    u := delete_min(H)  \(n\) times
    for all edges \((u, v)\) in E  \(m\) times
        if dist(v) > dist(u) + l(u, v) then
            dist(v) := dist(u) + l(u, v)
            prev(v) := u
            decreasekey(H, v)

Homework:
- Correctness
- Time Analysis
### How to implement \( V^* / H \) (Priority Queue)?

<table>
<thead>
<tr>
<th>Operation</th>
<th>Array</th>
<th>Binary Heap</th>
<th>Fibonacci Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialize to the values of ( V )</td>
<td>( O(n) )</td>
<td>( O(n) )</td>
<td>( n O(1) )</td>
</tr>
<tr>
<td>ExtractMin ( d )-value</td>
<td>( n \Theta(n) )</td>
<td>( n O(\log n) )</td>
<td>( n O(\log n) )</td>
</tr>
<tr>
<td>Change ( d )-value</td>
<td>( m O(1) )</td>
<td>( m O(\log n) )</td>
<td>( m O(1) )</td>
</tr>
<tr>
<td>Total Time</td>
<td>( O(n^2 + m) )</td>
<td>( O((n + m) \log n) )</td>
<td>( O(n \log n + m) )</td>
</tr>
</tbody>
</table>

- **Array** [Dijkstra 1962]:
  - \( O(n) \) initialize
  - \( n \Theta(n) \) extract
  - \( m O(1) \) change
  - Total time: \( O(n^2 + m) \) or \( O(n^2) \)

- **Binary Heap** [Williams 1964]:
  - \( O(n) \) initialize
  - \( n O(\log n) \) extract
  - \( m O(\log n) \) change
  - Total time: \( O((n + m) \log n) \) or \( O(n^2 \log n) \)

- **Fibonacci Heap** [Fredman/Tarjan 1987]:
  - \( n O(1) \) initialize
  - \( n O(\log n) \) extract
  - \( m O(1) \) change
  - Total time: \( O(n \log n + m) \) or \( O(n^2) \)