Introduction to Graph Algorithms for Shortest-Paths Problems

Zbyněk Křivka[1]

[1] *Faculty of Information Technology, Brno University of Technology, Brno, 612 66, Czech Republic*
krivka@fit.vutbr.cz

ABSTRACT

The paper introduces basic graph notions and definitions to describe few shortest-paths problems. Then, two standard graph representations are described, and two classical algorithms solving shortest-paths problems, namely Bellman-Ford and Dijkstra algorithms, are explained. In the end, new software tool supporting the explanation of graph algorithms is presented.

1. INTRODUCTION AND MOTIVATION

From the theoretical point of view, a graph is a mathematical structure representing a relation. In this introductory paper, we focus on finite binary relations where both the given set (so-called set of vertices) and the number of members (so-called edges) are finite.

The finite variant has many practical advantages, such as the possibility of intuitive graphical representation, which is usually very beneficial, and we use it in this paper as well. In the graphical representation, a vertex is represented as a small circle or node with a label, and an edge is drawn as a (labeled) connection or an arrow between two vertices.

Since a graph is an abstract mathematical notion, we can represent various problems with it, and consequently, we can use general properties and graph algorithms to solve specific questions or problems. Of course, we must deeply understand both the problem with its graph representation and the graph algorithm(s) to apply in order to get the correct results.

In this paper, we focus on several variants of one problem—finding the shortest path(s) in a given graph. Let us give two brief motivation examples in transportation and optimization. First, assume that we represent a map of cities and roads between them with distances as a graph, and we want to find the quickest/cheapest way from city A to city B. Second, more abstractly, assume we have a puzzle that can be solved by the finite state-space exploration, and we want to find an optimal solution, such as the minimal number of steps (see (Demel, 2002)). In optimization problems, we typically search for an optimal solution in a way that minimizes the cost based on the static state space exploration (e.g. knapsack problem, ...).

The rest of the paper is organized as follows. Section 2 gives all the basic terminology and definitions. In Section 3, two the most common graph representations are discussed. Then, Section 4 establishes the problems and introduces several algorithms to solve them. Finally, Section 5 describes demonstrational software tool called Graph Simulator. The presented algorithms can be studied in more details in Chapters 22 through 25 in (Cormen et al., 2009).

2. PRELIMINARIES AND BASIC GRAPH DEFINITIONS

We assume that the reader is familiar with basic notions in algebra, such as sets, sequences, and relations (see Sections 2.1 through 2.3 in (Meduna et al., 2013)).

2.1. Directed and Undirected Graphs

The section defines two kinds of graphs. In directed graphs, the direction of every edge matters. On contrary, in undirected graphs, we are not interested in the direction of an edge, so it can be used to represent a symmetric binary relation.

Definition 1. *A* directed graph *or* digraph*, denoted as G, is a pair $G = (V, E)$ where V is a finite set of* vertices *(nodes), and $E \subseteq V^2$ is a set of* edges *(arrows, arcs). From mathematical point of view, E is a binary relation on V. An edge (u, u) is called a* self-loop*. If (u, v) is an edge, we say that (u, v) is* incident from u *and* incident to v*, that is v is* adjacent *to u.*

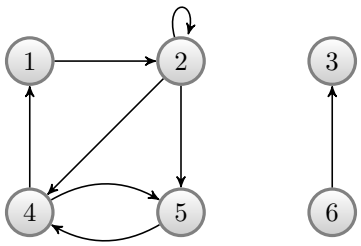

Figure 1. Directed graph G_1

To illustrate the set representation of a digraph, the graph in Figure 1 can be written as $G_1 = (\{1, 2, \ldots, 6\}, \{(1, 2), (2, 2), (2, 4), (2, 5), (4, 1), (4, 5), (5, 4), (6, 3)\})$.

Definition 2. *An* undirected graph *G is a pair $G = (V, E)$ where V and E has the same meaning as in a directed*

graph, but $E \subseteq \binom{V}{2}$*; that is,* E *consists of unordered pairs of vertices. Formally, an edge is a set* $\{u, v\}$*, where* $u, v \in V$ *and* $u \neq v$*. As a convention,* $\{u, v\}$*,* (u, v)*, and* (v, u) *denote the same edge. That is, an undirected graph is a digraph where the direction of edges does not matter and self-loops are forbidden.*

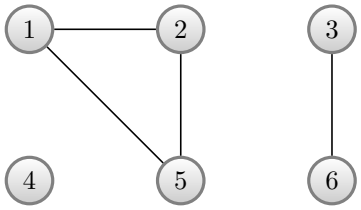

Figure 2. Undirected graph G_2

The graph in Figure 2 can be formally described as $G_2 = (\{1, 2, \ldots, 6\}, \{\{1, 2\}, \{1, 5\}, \{2, 5\}, \{3, 6\}\})$.

Definition 3. *A graph* $G = (V, E)$ *together with a* weight *function* $w \colon E \to \mathbb{R}$ *is called a* weighted graph.

2.2. Paths and Reachability

To describe some kind of tour through the nodes of a graph, we use the notions of (simple) path, subpath, and cycle.

Definition 4. *A* path $p = \langle v_0, v_1, v_2, \ldots, v_k \rangle$ *is a connected sequence of vertices where* $(v_{i-1}, v_i) \in E$ *for all* $i = 1, 2, \ldots, k$*,* $k \geq 0$*. The length of* p *equals to the number of edges in* p*. Note that the length of a self-loop is 1. If there is path* p *from* u *to* u'*, we say that* u' *is* reachable *from* u *by* p*, denoted as* $u \overset{p}{\rightsquigarrow} u'$*. A path is* simple *if all vertices in the path are distinct. In other words, there is no repetition of vertices in the sequence.*

For example, in Figure 1, we can see $\langle 1, 2, 2, 4, 5, 4 \rangle$ and $\langle 4, 1, 2 \rangle$ as a path and a simple path, respectively. Notice that we can give an example of a sequence of vertices such as $\langle 2, 1, 4, 5 \rangle$ that is not a path at all.

Definition 5. *A* subpath s *of* $p = \langle v_0, v_1, v_2, \ldots, v_k \rangle$ *is a contiguous subsequence,* $s = \langle v_i, v_{i+1}, v_{i+2}, \ldots, v_j \rangle$*, for* $0 \leq i \leq j \leq k$*. A path* $c = \langle v_0, v_1, v_2, \ldots, v_k \rangle$ *is a* cycle*, if* $k \geq 1$ *and* $v_0 = v_k$*. A closed simple path is called a* simple cycle*. A graph containing no cycles is called* acyclic*.*

For example, in Figure 1, $\langle 1, 2, 4, 5, 4, 1 \rangle$, $\langle 1, 2, 4, 1 \rangle$, and $\langle 2, 2 \rangle$ are a cycle, simple cycle, and self-loop, respectively. In undirected graphs, we usually assume that a cycle have to contain at least three distinct vertices. For instance in G_2, $\langle 3, 6, 3 \rangle$ is not considered to be a cycle, but $\langle 5, 1, 2 \rangle$ is a simple cycle.

Definition 6. *A graph* $G' = (V', E')$ *is a* subgraph *of* G*, if* $V' \subseteq V$ *and* $E' \subseteq E$*. An undirected graph is* connected *if every pair of vertices is connected by a path. A connected, acyclic, undirected graph is a* tree*.*

Observe that, for a single tree, $|E| = |V| - 1$.

2.3. Time and Space Complexity

In order to roughly analyze time and space complexity of the algorithms and graph representations, respectively, we assume that a *step* is a sequence of primitive actions in the computational environment that takes always almost the same amount of time, and, in analogy, a *cell* is able to store some primitive value in the computational environment. For instance, in a computer, an 32-bit integer can be stored in one cell and assigned to some variable in one step. Since the notion of a step is really rough, a constant number of assignments can be understood as one step as well.

3. Graph Representations

In this section, we show two ways of a graph representation: (1) Adjacency-list and (2) Adjacency-matrix representation. Hereafter, let $G = (V, E)$ be a graph, $n = |V|$, and $m = |E|$. We call a graph to be *sparse* if m is significantly lower than n^2 (written as $m \ll n^2$); otherwise, it is *dense*. For simplification, we assume that there is a linear order on the set of vertices, so the vertices can be straightforwardly denoted by a consecutive list of integers, such as $V = \{1, 2, \ldots, n\}$.

3.1. Adjacency-list representation

A graph $G = (V, E)$ is represented as an array $Adj[1 \ldots n]$ with n lists, where for each vertex $u \in V$, the corresponding list is unsorted and contains all vertices adjacent to u, so $Adj[u]$ stores all vertices v such that $(u, v) \in E$.

This representation is effective for sparse graphs, and its space complexity depends linearly on the number of vertices and edges. Every edge is represented by a vertex in the list referenced from some $Adj[w \in V]$; that is, the sum of lengths of all lists of neighbors from Adj is m. In addition, the array Adj has size linearly dependent on the number of vertices; even for an isolated graph (no edges), we need n storage cells in the array. In total, the space complexity is about $m + n$ cells.

In order to represent a weighted graph, we extend the elements stored in every adjacency list. For every $(u, v) \in E$, we store in $Adj[u]$-list a structure containing two items: (1) vertex v and (2) value of $w(u, v)$.

As a conclusion, notice that adjacency-list is beneficial for sparse graphs in terms of space, but it takes linear time to find whether some given edge (u, v) belongs to the given graph as the answer requires to search of the whole unsorted list $Adj[u]$ with length up to n elements. For instance, in G_3 and its adjacency-list representation in Figure 4, we can see that $Adj[3]$-list contains adjacent vertices 6 and 5, so the check whether $(3, 5) \in E$ takes about 3 steps (1 step to access index 3 of Adj, and 2 steps to go through the linked-list until we reach vertex 5).

3.2. Adjacency-matrix representation

Let $G = (V, E)$ be a graph and assume $V = \{1, 2, \ldots, n\}$. *Adjacency matrix* $A = (a_{ij})$ is a matrix of size $n \times n$ such that

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E, \\ 0 & \text{otherwise.} \end{cases}$$

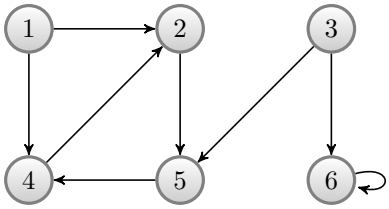

Figure 3. Directed graph G_3

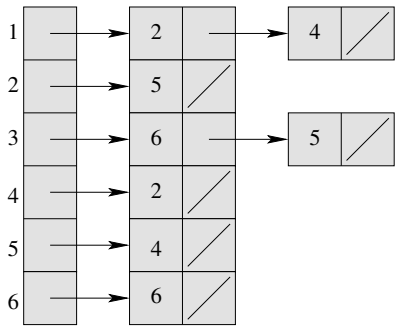

Figure 4. Adjacency-list representation of G_3

The main advantage of this representation is that we can get a quick answer whether two given vertices are connected by an edge in just one step. For instance, given the adjacency matrix for G_3 in Figure 5, we can quickly see that $A[4, 2]$ is 1, so $(4, 2) \in E$.

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Figure 5. Adjacency-matrix representation of G_3

Since we always store the whole matrix in this representation, we need n^2 cells to represent a graph with n vertices. Note that it does not matter how many edges the graph has because we store yes/no information for every possibility of an edge in comparison to the adjacency-list representation where only existing edges are stored. Therefore, the adjacency-matrix representation is more effective for dense graphs where the number of edges m is close to n^2.

The *transpose* matrix of $A = (a_{ij})$ is a matrix $A^T = (a^T_{ij})$, where $a^T_{ij} = a_{ji}$. Observe that if adjacency matrix A represents an undirected graph, then $A = A^T$. That is, an adjacency matrix is symmetric along its main diagonal, as you can see for example in Figure 7 with adjacency matrix for graph G_4 (see Figure 6). Hence, it is enough to store just one half of A. Note that if self-loops are forbidden, the main diagonal of adjacency matrix is made by zeros only. Let $G = (V, E)$ be a weighted graph, then we can combine the adjacency-matrix representation and the

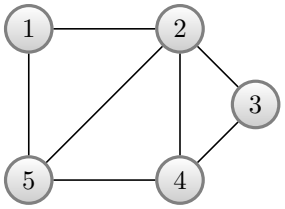

Figure 6. Undirected graph G_4

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Figure 7. Adjacency-matrix representation of G_4

information about the weight of each edge.

$$a_{ij} = \begin{cases} w(i,j) & \text{if } (i,j) \in E, \\ \text{NIL} & \text{otherwise,} \end{cases}$$

where NIL is a special value, mostly 0 or ∞, that is needed to represent that this edge is not in the represented graph.

Example 7. *Let $deg_-(u)$ and $deg_+(u)$ be the number of edges incident from u and incident to u, respectively. Given a vertex x and a digraph represented by the adjacency-list or adjacency-matrix representation, respectively, how long does it take to compute* degrees $deg_-(x)$ *and $deg_+(x)$?*

Solution. See the algorithms to compute deg_+ for a given vertex in Figures 8 and 9. Obviously, the time complexity of the first algorithm depends linearly on the number of edges in the whole graph. On the other hand, the time complexity of the second solution is only dependent linearly on the number of vertices which is usually lower (especially for dense graphs).

```
IN-DEGREE-LISTS(Adj, x)
1  deg ← 0
2  for each vertex u ∈ V
3      do for each vertex v ∈ Adj[u]
4             do if v = x
5                    then deg ← deg + 1
6  return deg
```

Figure 8. A procedure to find $deg_+(x)$ in G represented by an adjacency list, Adj.

Example 8. *The* transpose *of a directed graph $G = (V, E)$ is the graph $G^T = (V, E^T)$, where $E^T = \{(v, u) \in V \times V : (u, v) \in E\}$. Thus, G^T is G with all its edges reversed. Describe an efficient algorithm for computing G^T from G for the adjacency-list representation of G. Analyze the time complexity of your algorithm.*

Hint. *Inspire yourself in* IN-DEGREE-LISTS *procedure where you can see how to traverse all edges of the graph in the adjacency-list representation (see Figure 8).*

```
IN-DEGREE-MATRIX(A, x)
1  deg ← 0
2  for i ← 1 to n
3       do deg ← deg + A[i, x]
4  return deg
```

Figure 9. A procedure to find $deg_+(x)$ using adjacency-matrix representation, A.

4. SHORTEST PATHS

In this section, we first define notion of the shortest path between two given vertices and its weight (or cost). Then, several variants of the problem are discussed with deeper focus on the single-source shortest-paths problem. Before we explain algorithms solving the given problem, we describe how the shortest paths from a single source into all other vertices can be effectively represented.

4.1. Definition

Given weighted directed graph $G = (V, E)$ and weight function $w : E \to \mathbb{R}$. The *weight* (or *cost*) of path $p = \langle v_0, v_1, \ldots, v_k \rangle$ is

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

The *shortest-path weight* from u to v is

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \overset{p}{\rightsquigarrow} v\} & \text{if some } u \rightsquigarrow v \text{ exists} \\ \infty & \text{otherwise} \end{cases}$$

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

Observe that there can be several shortest paths between u and v or even none if v is not reachable from u.

4.2. Variants

To be complete, there are two kinds of the shortest paths problems. First, when some starting vertex s is given, and only paths outgoing from s are examined. Second, we are looking for a shortest path, but between vertices that are not known yet, so we compute the shortest paths for all pairs of vertices.

The most common problem is to find a shortest path from vertex s into some vertex t (so-called *single-pair shortest-path problem*). Since, during this kind of search, we also find shortest paths into all other reachable vertices, we preferably discuss a generalized variant called the *single-source shortest-paths problem*. If you are solving a problem how close all other vertices are to the given target vertex (so-called *single-destination shortest-paths problem*), try to reverse the direction of each edge and then apply an algorithm for the single-source shortest-paths problem. Of course, do not forget to revert the direction of edges in the result as well. Finally, in the *all-pairs shortest-paths problem*, we can apply a single-source shortest-paths problem algorithm for every vertex and join the results, but the challenge is to do it faster.

4.3. Single-Source Shortest-Paths Problem

The very basic idea of the problem decomposition is based on Lemma 9 stating that if we find shortest subpath, we can use it as a part of a shortest path.

4.3.1. Subpaths of Shortest Paths

Lemma 9. *Let $G = (V, E)$ be directed graph with weight function $w : E \to \mathbb{R}$. Let $p = \langle v_1, v_2, \ldots, v_k \rangle$ be a shortest path from v_1 to v_k. For any $1 \leq i \leq j \leq k$, let $p_{ij} = \langle v_i, v_{i+1}, \ldots, v_j \rangle$ be the subpath of p from v_i to v_j. Then, p_{ij} is a shortest path from v_i to v_j.*

Proof. Decompose p as $v_1 \overset{p_{1i}}{\rightsquigarrow} v_i \overset{p_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$, where $w(p) = w(p_{1i}) + w(p_{ij}) + w(p_{jk})$. Assume that there is p'_{ij} from v_i to v_j with $w(p'_{ij}) < w(p_{ij})$. Then,

$$v_1 \overset{p_{1i}}{\rightsquigarrow} v_i \overset{p'_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k$$

where $w(p_{1i}) + w(p'_{ij}) + w(p_{jk}) < w(p)$. That is a contradiction. □

Negative-weight cycles. If G contains no negative-weight cycles reachable from source s, $s \in V$, then for all $v \in V$, $\delta(s, v)$ remains well defined (even if negative). If G contains a negative-weight cycle reachable from s, δ is not well defined since the following algorithms endlessly repeat traverse of the negative-weight cycle in order to decrease the total weight of the path. If there is a negative-weight cycle on some path from s to v, we define $\delta(s, v) = -\infty$.

Note that even if we consider a graph with negative-weight cycle, there is always some shortest simple path that does not visit any vertex repeatedly. The problem of most algorithms for searching a shortest path is that they do really just search a path without being simple, so the revisiting of some vertices is possible and it is not checked to preserve effectiveness.

4.3.2. Representing Shortest Paths

Let $G = (V, E)$ be a graph and π is an array of size of n cells. For each $v \in V$, $\pi[v]$ is set to a *predecessor* of v; that is, a vertex or NIL. If $\pi[v] = u \neq \text{NIL}$, then $(u, v) \in E$. We define *predecessor subgraph* $G_\pi = (V_\pi, E_\pi)$ induced by π as follows.

$$V_\pi = \{v \in V : \pi[v] \neq \text{NIL}\} \cup \{s\}$$

$$E_\pi = \{(\pi[v], v) \in E : v \in (V_\pi - \{s\})\}$$

After a searching algorithm (see Sections 4.3.4 and 4.3.5) is finished, G_π is a *shortest-paths tree* rooted at s containing shortest paths from s to all other reachable vertices. Note that $\pi[v] = \text{NIL}$ means there is no predecessor of v such as v is not reachable from s or v is a root of the predecessor subgraph.

```
PRINT-PATH(G, s, v)
1  if v = s
2     then print s
3     else if π[v] = NIL
4             then print "No path from " s " to " v "!"
5             else PRINT-PATH(G, s, π[v])
6                  print v
```

Figure 10. Recursive procedure to print the shortest path stored in an array π.

To print the path from s to v stored in π, use recursive procedure PRINT-PATH(G, s, v) (see Figure 10).

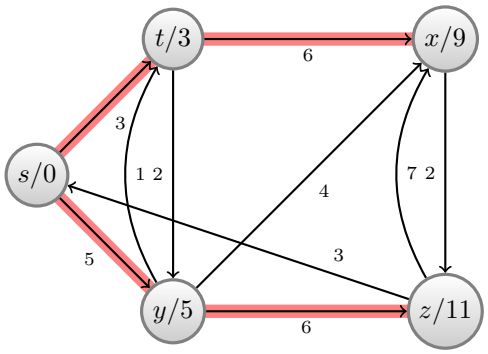

Figure 11. Shortest paths of G_5 from source vertex s. The label of each node is in the form $v/\delta(s, v)$.

Let us note that to save space in the arguments of the procedures in this paper, some data are passed using global variables; namely, d and π arrays are global array available in any procedure.

Example 10. *In Figure 11, we can see one possibility of G_π according to $\pi[s, t, x, y, z] = [\text{NIL}, s, s, t, y]$ using the red highlighting of edges.*

Let us illustrate how to print a shortest path from s to x in G_5 from π by the execution of PRINT-PATH(G_5, s, x). *Since $\pi[x] = t$, we execute* PRINT-PATH(G_5, s, t) *and, in analogy, for $\pi[t] = s$, we execute* PRINT-PATH(G_5, s, s) *which prints vertex s and finishes the recursive execution of* PRINT-PATH(G_5, s, s). *Then, t is printed in* PRINT-PATH(G_5, s, t), *and, finally,* PRINT-PATH(G_5, s, x) *prints x, so we get stx describing $\langle s, t, x\rangle$ as a shortest path from s to x.*

Try to find another π that could represent another shortest paths in G_5 and step through the execution of PRINT-PATH(G_5, s, x) *to see another shortest path from s to x.*

4.3.3. Basic Algorithm Components

In both algorithms discussed in this paper, we need the same initialization of π and d array (see Figure 12) and the improvement of d-value for a vertex is done by the relaxation procedure (see Figure 13).

As already describe in Section 4.3.2, π-array represents searched and found shortest paths from source vertex s. In addition, d-array stores the estimations of the distance of each vertex from s. In the beginning, we do not know

```
INITIALIZE-SINGLE-SOURCE(G, s)
1  for each vertex v ∈ V
2       do d[v] ← ∞
3          π[v] ← NIL
4  d[s] ← 0
```

Figure 12. Common initialization of π and d arrays.

```
RELAX(u, v, w)
1  if d[v] > d[u] + w(u, v)
2     then d[v] ← d[u] + w(u, v)
3          π[v] ← u
```

Figure 13. Relaxing the given edge according to w.

which vertices are reachable from s, so we set $d[v]$ to ∞ and the predecessor $\pi[v]$ to NIL. As s is trivially reachable from itself, set $d[s]$ to 0. Considering the time complexity of the initialization, we just do some constant number of steps for each vertex, so it is n steps in total.

The basic action repeatedly performed in both introduced algorithms is the relaxation (see Figure 13). This procedure checks the given edge (u, v) and the current distance estimation in d-array and if the estimation for v can be improved (decreased), both $d[v]$ and $\pi[v]$ are updated so that the new path estimation uses subpath $s \rightsquigarrow u$ followed by edge (u, v). All actions in the procedure are primitive, so they can be made in constant time.

4.3.4. Bellman-Ford Algorithm

The first algorithm is general enough to handle any weighted graph. For the given graph $G = (V, E)$, weight function w, and source vertex $s \in V$, the algorithm fills d and π arrays. If G contains negative-weight cycles reachable from s, it returns FALSE; otherwise, we get TRUE and π contains the shortest paths from s (see Figure 14).

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i ← 1 to n − 1
3       do for each edge (u, v) ∈ E
4               do RELAX(u, v, w)
5  for each edge (u, v) ∈ E
6       do if d[v] > d[u] + w(u, v)
7             then return FALSE
8  return TRUE
```

Figure 14. Bellman-Ford Algorithm

To describe the basic principle, we need to realize that a shortest path $p = \langle v_0, v_1, v_2, \ldots, v_k\rangle$ can be found by picking the right edge one by one. That is, if we can guess the right order of edges in p, we can construct it just in k steps by taking (v_0, v_1) to RELAX procedure to get $\langle v_0, v_1\rangle$ as a shortest subpath of p. Then, taking (v_1, v_2) to RELAX procedure to get $\langle v_0, v_1, v_2\rangle$ as a one-edge longer shortest subpath of p, and so on. Unfortunately, we are not able to do such magic guessing in an arbitrary graph[1], so we ap-

[1] We are able to do that in an acyclic graph where we establish topological

ply a brute-force approach in Bellman-Ford algorithm (see Lines 2-4 in Figure 14).

In more detail, we take any sorting of edges from E such as $e = \langle (u_1, u'_1), (u_2, u'_2), \ldots, (u_m, u'_m) \rangle$. For instance, we can take a sorting given by a traversal of all edges, such as in Lines 2 and 3 in Figure 8. If we are relaxing all edges of G in sorting e, we do $(n-1) \cdot m$ calls of RELAX procedure on edges in the following order (the list has $(n-1)$ rows of sorting e)

$$\begin{array}{llll} (u_1, u'_1), & (u_2, u'_2), & \ldots, (v_0, v_1), \ldots, & (u_m, u'_m), \\ (u_1, u'_1), & (u_2, u'_2), & \ldots, (v_1, v_2), \ldots, & (u_m, u'_m), \\ \vdots & & \ddots & \vdots \\ (u_1, u'_1), & (u_2, u'_2), & \ldots & (u_m, u'_m) \end{array}$$

Now, you can see that apart from many potentially useless relaxations we guarantee that edges of p are relaxed in the right order. In the first row we relax (v_0, v_1), in the second row (v_1, v_2), and so on. Note that (v_1, v_2) can be in front of (v_0, v_1) in e. Then, since every simple path in a graph has at most $n-1$ edges (as a simple path is a special variant of a tree; see Definition 6), we can assume that every shortest path in a graph without negative-weight cycle has at most $n-1$ edges, therefore $n-1$ iterations over all edges is enough. Observe that also shortest paths to other vertices (not just v_k) are found since the correct order of relaxations occurs for paths $s \rightsquigarrow v'$ as well in the above $(n-1) \cdot m$ relaxations.

To be more precise, if there occurs no change of d and π in one entire iteration, then we can skip the rest of iterations. On the other hand, if a relaxation of some edge is possible at Lines 5-7, it means we are able to improve some shortest path which implies that some negative-weight cycle is reachable from s in G. In that case, the algorithm just returns FALSE without any further problem analysis; otherwise, it returns TRUE.

Time Complexity. Obviously, the initialization on Line 1 takes n steps. Lines 2-4 are performed $(n-1)$-times and each iteration takes m steps. Lines 5-7 take also m steps. In total, the algorithm does roughly $m \cdot n$ steps.

Example 11. *Given G_6 in Figure 15, let us demonstrate in the following table how Bellman-Ford algorithm computes the shortest paths from source vertex s. For every vertex $v \in V$, there is a column with $d[v]/\pi[v]$-values in its cells. The first row describes the situation after the initialization, $(i+1)$-th row describes the situation when ith iteration over all edges is done. Each iteration relaxes the edges in the order (t,x), (t,y), (t,z), (x,t), (y,x), (y,z), (z,x), (z,s), (s,t), (s,y).*

	s	t	x	y	z
init	$0/\text{NIL}$	∞/NIL	∞/NIL	∞/NIL	∞/NIL
1	$0/\text{NIL}$	$6/s$	∞/NIL	$7/s$	∞/NIL
2	$0/\text{NIL}$	$6/s$	$4/y$	$7/s$	$2/t$
3	$0/\text{NIL}$	$2/x$	$4/y$	$7/s$	$2/t$
4	$0/\text{NIL}$	$2/x$	$4/y$	$7/s$	$-2/t$

sorting in $(n+m)$ steps. Then, we do the relaxation of all nodes in the topological order, which is, in fact, the right order to get the shortest paths. See Section 24.2 in (Cormen et. al, 2009).

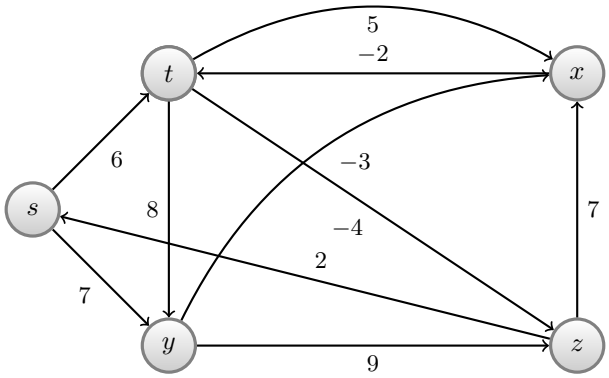

Figure 15. Weighted directed graph G_6 to illustrate processing by Bellman-Ford Algorithm.

It it easy to see that in the additional iteration no changes occur, so Bellman-Ford algorithm returns TRUE.

4.3.5. Dijkstra Algorithm

Dijkstra algorithm requires weighted graph without negative edges on its input. For the given graph $G = (V, E)$ represented by adjacency list Adj, weight function w such that for each $(u, v) \in E$, $w(u, v) \geq 0$, and source vertex $s \in V$, the algorithm computes and fills d and π arrays (see Figure 16). The presented variant uses a special abstract data type called *min-priority queue* to store vertices waiting for processing. Min-priority queue works like a classical queue but the vertices with the lowest d-value are dequeued (pull out) with the priority from the queue using EXTRACT-MIN operation.

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  ENQUEUE-ALL(Q, V, s)
3  while Q ≠ ∅
4        do u ←EXTRACT-MIN(Q)
5           for each vertex v ∈ Adj[u]
6                 do RELAX(u, v, w)
```

Figure 16. Dijkstra Algorithm

Using auxiliary procedure ENQUEUE-ALL, we initialize min-priority queue Q and we enqueue (put) all vertices into Q one by one starting with s. Then, the algorithm iterates while we have some vertices in Q. We always take a vertex with the lowest d-value from Q into variable u and on Lines 5-6 we try to relax all edges incident from u.

Notice several properties of the algorithm: (1) since in the beginning only s has non-infinite priority, it will be always the first vertex to be dequeue in the following **while**-cycle; (2) after we dequeue vertex u, it is considered to be finished so its distance estimation $d[u]$ is, in fact, equal to $\delta(s, u)$ (the shortest distance of u from s) hereafter and u is never enqueued again.

Considering G_5, after application of Dijkstra algorithm, we get the shortest paths from s as highlighted in Figure 11. If you study the development of d-array, you will see

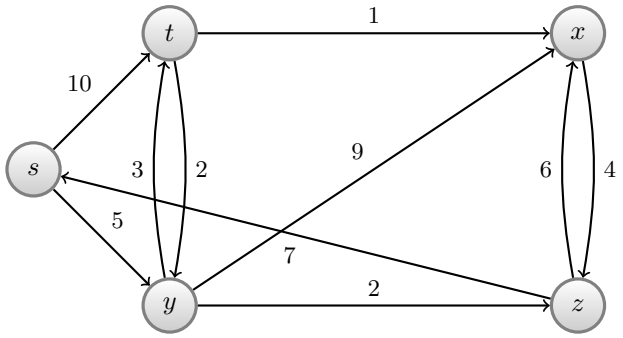

Figure 17. Weighted directed graph G_7 to illustrate processing by Dijkstra Algorithm.

that, for each $v \in V$, after the initialization of $d[v]$ to ∞, there occurs at most one change of $d[v]$ such that $d[v] = \delta(s, v)$. Therefore, to illustrate the algorithm more didactically, consider the following example.

Example 12. *Consider G_7 in Figure 17 and examine the computation of Dijkstra algorithm with G_7 as its input to find the shortest paths from source vertex s. To describe the processing, we write down d and π arrays in the following table. The first row describes the state after initialization. The following rows of the table describes the state after an iteration with some u taken from min-priority queue as the vertex with the smallest d-value. The gray cells cannot be changed anymore, and they denote that the vertex labelling the corresponding column is already removed from the queue.*

	s	t	x	y	z
init	0/NIL	∞/NIL	∞/NIL	∞/NIL	∞/NIL
$u \leftarrow s$	0/NIL	$10/s$	∞/NIL	$5/s$	∞/NIL
$u \leftarrow y$	0/NIL	$8/y$	$14/y$	$5/s$	$7/y$
$u \leftarrow z$	0/NIL	$8/y$	$13/z$	$5/s$	$7/y$
$u \leftarrow t$	0/NIL	$8/y$	$9/t$	$5/s$	$7/y$
$u \leftarrow x$	0/NIL	$8/y$	$9/t$	$5/s$	$7/y$

To understand the algorithm in more depth, try to modify the weights of edges of G_7 such that there will be some negative-weight edges (but not cycle) such that Dijkstra algorithm produces incorrect results.

Time Complexity. The analysis depends on the way of implementation of min-priority queue used by Dijkstra algorithm. First, consider simple and intuitive implementation of min-priority queue using unsorted array where the decrease of the priority of the given vertex is very simple and possible in constant number of steps. On the other hand, finding the min-priority vertex in EXTRACT-MIN operation takes up to n steps since we need to go through the whole array and check it vertex by vertex. Then, in Dijkstra algorithm, Line 1 takes n steps and Line 2 as well. We iterate n-times and in each iteration, we need to do EXTRACT-MIN and some number of relaxations. Since we do the relaxation for every edge exactly once, we can aggregate all the steps in the whole loop as $n^2 + m$. For a simple graph, $m \leq n^2$, we can disregard m in $n^2 + m$ and we can also disregard linear time complexity of Lines 1-2 to get quadratic time complexity with respect to the number of vertices; that is n^2 steps.

Second, if we implement min-priority queue using heaps, we can get better time complexity but the code complexity increases. For sparse graphs, we get the rough time complexity $m \cdot \log n$ using binary heap or even better $n \cdot \log n + m$ for Fibonacci heap. Let us point out that in heap implementation, we need to change also Line 2 in RELAX procedure, because the decrease of a priority stored in $d[v]$ is non-primitive step since we need to change the storage of vertices in the heap that models the min-priority queue. The discussion of this implementation is beyond the focus of this introductory paper.

As you can see, in comparison to Bellman-Ford algorithm, we can perform Dijkstra algorithm significantly faster, but, in general, we cannot work with negative-weight edges.

Example 13 (Measuring-cup Problem)**.** *Assume that we have a 1-litre cup and a 3-litre cup. We can tap/fill a cup from a wine barrel, we can pour from one cup to another as much as possible without spilling, and we can empty a cup completely. How to measure 2 litres?*

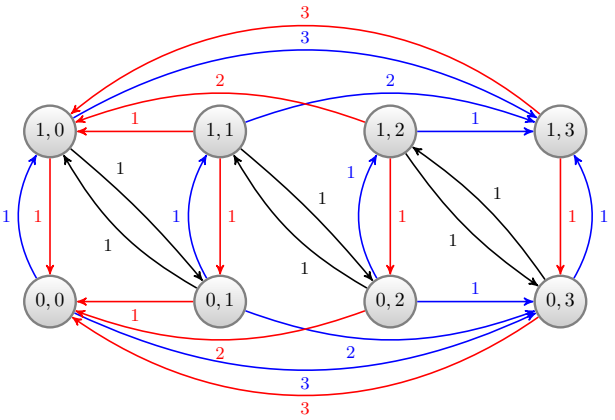

Figure 18. Graph G_j representing Measuring-cup problem for 1-litre and 3-litre cup. The edges that model wine tapping/filling, emptying, and pouring are blue, red, and black, respectively.

We can represent the problem by a directed graph $G_j = (V_j, E_j)$ (see Figure 18). Each vertex represents the amount of wine in each cup and edges represent the changes of this amount according to the actions described above. If we assume that all actions take the same amount of effort, we can consider unit weights, that is, for all $u, v \in V_j$, $w(u, v) = 1$, then, we can use Dijkstra algorithm to find the shortest paths from $(0, 0)$ in order to see how to reach $(0, 2)$ as the target state. Then, we find the shortest path

$$p\colon\ (0,0) \xrightarrow{1} (0,3) \xrightarrow{1} (1,2) \xrightarrow{1} (0,2)$$

People who love wine are unhappy with the result since we spilled one litre of wine in the last edge of p, so we can change weights of red edges (ones that empty a cup) to $|E_j|$. In fact, it is enough to define the weight as in Figure 18, where each edge denotes a time to pour or spill the wine that depends on its amount. The next search of the shortest

path finds more suitable solution

$$p' : \ (0,0) \xrightarrow{1} (1,0) \xrightarrow{1} (0,1) \xrightarrow{1} (1,1) \xrightarrow{1} (0,2)$$

To give more demanding challenge, assume we have 3-litre cup and 5-litre cup. Is it possible to measure 4 litres?

Of course, do Dijkstra or even Bellman-Ford algorithm for bigger graphs such as G_j would be very lengthy and error-prone. Therefore, it is very useful to have some software tool to experiment with the algorithms.

5. Demonstrational Tool

There is a new graphical software tool called *Graph Simulator* that allows to play with the discussed algorithms. The application with graphical user interface (see Figure 19) was implemented by Jakub Varadinek and Otto Michalička in Java programming language. The user interface is in English or Czech. The application works in two modes: (1) the graph editing and (2) the algorithm simulation.

During the graph editing, the middle-mouse-button click can add new vertex and double click on the label of the vertex allows us to change it. By clicking on a vertex and then another vertex, we can create a directed edge. By the double click on the label of an edge, we can edit the weight of the edge. In the side window, we can see the adjacency-list representation of the graph. If some algorithms require to pick up some vertex (for instance, source vertex), we can select a vertex by clicking on it and use menu *Edit* and item *Set Selected*. Of course, it is possible to save the graph or load another one from an external file.

In order to start a simulation, select the algorithm in menu *Simulation* to set the application into the simulation mode when new window will pop up. Here, you will see the graph (uneditable), the algorithm, the simulation buttons, and the content of used variables. Using simulation buttons, you can go one step back in the simulation, stop the simulation, start/pause the simulation, and step forward in the simulation. You can also place breakpoints in the algorithms by clicking the line number and then, the simulation pauses when it reaches this line. In addition, you can choose the speed of the simulation using the slider.

The application supports several graph algorithms working with the adjacency-list representation:

- Breadth-First Search (BFS)
- Depth-First Search (DFS)
- Topological Sorting
- Finding of strongly-connected components (SCC)
- Bellman-Ford Algorithm
- Dijkstra Algorithm

Download at http://www.fit.vutbr.cz/~krivka/graphsim

6. Conclusion

Obviously, this paper only introduces two classical algorithms for finding a shortest path in the given graph. If you are interested, continue your study in Part IV of (Cormen et al., 2009) where you can find more detailed discussion of these algorithms and additional algorithms such as Floyd-Waschall algorithm for all-pairs shortest-paths problem or linear time algorithm for single-source shortest-paths problem in acyclic graphs. If you are more interested in single-pair shortest-path problem, focus on A^* algorithm.

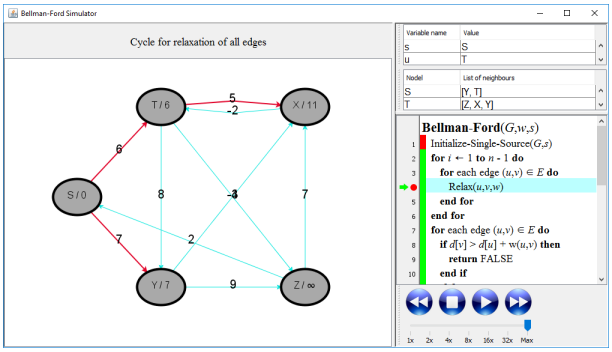

Figure 19. Graph Simulator - the demonstration tool for basic graph algorithms

Acknowledgements

My thanks go to José Ignation Farrán Martín for his help with typesetting of this paper and to Radim Kocman for his proof-reading and many valuable comments. This work was supported by SOCRATES/ERASMUS teacher exchange programme and the BUT grant FIT-S-17-3964.

References

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms* (Third ed.). The MIT Press.

Demel, J. (2002). *Grafy a jejich aplikace [in czech]*. Academia.

Meduna, A., Vrábel, L., & Zemek, P. (2013). *Mathematical Foundations of Formal Language Theory*. Retrieved from `http://www.fit.vutbr.cz/~izemek/grants.php.en?file=%2Fproj%2F589%2FText.pdf&id=589`

Biography

Zbyněk Křivka is both a theoretically and pragmatically oriented computer scientist. He defended his Ph.D. thesis in 2008 at Brno University of Technology and published several journal articles in the theory of formal languages. Nowadays, he is an assistant professor at Faculty of Information Technology, Brno University of Technology, teaching Formal languages and Compilers, Compiler Construction, Principles of Programming Languages, and Graph Algorithms courses. For the details, see http://www.fit.vutbr.cz/~krivka/.en.