Algorithms and Data Structures II.
Lecture Notes:
Algorithms on weighted graphs

Ásványi Tibor – asvanyi@inf.elte.hu

November 6, 2022

Contents

References

[1] Ásványi, T, Algorithms and Data Structures I. Lecture Notes
http://aszt.inf.elte.hu/∼asvanyi/ds/AlgDs1/AlgDs1LectureNotes.pdf

[2] Ásványi, T, Algorithms and Data Structures II. Lecture Notes: Elementary graph algorithms
http://aszt.inf.elte.hu/∼asvanyi/ds/AlgDs1/AlgDS2graphs1.pdf

[3] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.,
Introduction to Algorithms (Third Edititon), *The MIT Press*, 2009.

[4] Cormen, Thomas H., Algorithms Unlocked, *The MIT Press*, 2013.

[5] Narashima Karumanchi,
Data Structures and Algorithms Made Easy, *CareerMonk Publication*, 2016.

[6] Neapolitan, Richard E., Foundations of algorithms (Fifth edition), *Jones & Bartlett Learning*, 2015. ISBN 978-1-284-04919-0 (pbk.)

[7] Shaffer, Clifford A.,
A Practical Introduction to Data Structures and Algorithm Analysis, Edition 3.1 (C++ Version), 2011
(See http://aszt.inf.elte.hu/∼asvanyi/ds/C++3e20110103.pdf)

[8] Tarjan, Robert Endre, Data Structures and Network Algorithms, *CBMS-NSF Regional Conference Series in Applied Mathematics*, 1987.

[9] Weiss, Mark Allen, Data Structures and Algorithm Analysis in C++ (Fourth Edition),
Pearson, 2014.

[10] Ásványi Tibor, Detecting negative cycles with Tarjan's breadth-first scanning algorithm (2016)
http://ceur-ws.org/Vol-2046/asvanyi.pdf

1 Weighted graphs and their representations ([3] 22)

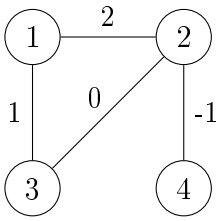
Definition 1.1 *A* weighted graph *is a graph* $G = (V, E)$ *with the weight function* $w : E \to \mathbb{R}$ *where* V *is the finite set of vertices, and* $E \subseteq V \times V \setminus \{(u, u) : u \in V\}$ *is the set of edges.*

Provided that $(u, v) \in E$, $w(u, v)$ *is its* weight *or* length *or* cost *(*weight, length *and* cost *are synonyms).*

Definition 1.2 *Given a weighted graph, the* weight *or* length *or* cost *of a path is the sum of the weights of the edges along the path.*

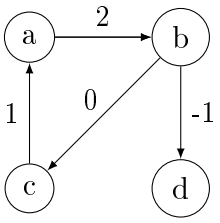
1.1 Graphical representation

The edges are labeled with their weights.



1 – 2, 2 ; 3, 1.
2 – 3, 0 ; 4, -1.

Figure 1: A weighted, undirected graph in graphical representation (on the left) and in textual representation.



a → b, 2.
b → c, 0 ; d, -1.
c → a, 1.

Figure 2: A weighted digraph in graphical representation (on the left) and in textual representation (on the right).

1.2 Textual representation

In case of undirected graphs "$u - v_{u_1}, w_{u_1}; \cdots ; v_{u_k}, w_{u_k}$." means that $(u, v_{u_1}), \cdots , (u, v_{u_k})$ are edges of the graph, in order with weights $w(u, v_{u_1}) = w_{u_1}, \cdots , w(u, v_{u_k}) = w_{u_k}$. (See Figure 1.)

In case of digraphs "$u \to v_{u_1}, w_{u_1}; \cdots ; v_{u_k}, w_{u_k}$." means that from vertex u come out directed edges $(u, v_{u_1}), \cdots , (u, v_{u_k})$, in order with weights $w(u, v_{u_1}) = w_{u_1}, \cdots , w(u, v_{u_k}) = w_{u_k}$. (See Figure 2.)

1.3 Adjacency matrix representation

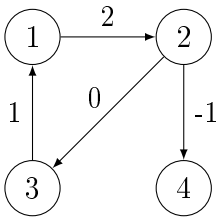
In the *adjacency matrix representation*, graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$ ($V = \{v_1, \cdots , v_n\}$) is represented with matrix $A/1 : \mathbb{R}_\infty[n, n]$ where $n = |V|$ is the number of vertices, $1..n$ are their sequence numbers, and for sequence numbers $i, j \in 1..n$,

$$A[i, j] = w(v_i, v_j) \iff (v_i, v_j) \in E$$

$$A[i, i] = 0$$

$$A[i, j] = \infty \iff (v_i, v_j) \notin E \wedge i \neq j$$

For example, on Figure 3, the digraph is represented with the adjacency matrix next to it.

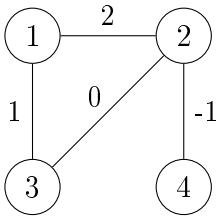


A	1	2	3	4
1	0	2	∞	∞
2	∞	0	0	-1
3	1	∞	0	∞
4	∞	∞	∞	0

Figure 3: A weighted digraph in graphical representation (on the left) and in adjacency matrix representation (on the right).

There are always zeros in the main diagonal, because we study only simple graphs (with no looping edge), and a vertex is available from itself on a path of zero length. See Figure 4.

Notice that the adjacency matrix is always symmetrical, if the graph is undirected, because in case $(v_i, v_j) \in E$, $(v_j, v_i) = (v_i, v_j) \in E$.



A	1	2	3	4
1	0	2	1	∞
2	2	0	0	-1
3	1	0	0	∞
4	∞	-1	∞	0

Figure 4: A weighted undirected graph in graphical representation (on the left) and in adjacency matrix representation (on the right).

1.4 Adjacency list representation

The adjacency list representation is similar to the textual one. Graph $G = (V, E)$ with weight function $w : E \to \mathbb{R}$ ($V = \{v_1, \cdots, v_n\}$) is represented with pointer array $A : Edge*[n]$ where type $Edge$ is the following one.

$Edge$
$+v : \mathbb{N}$
$+w : \mathbb{R}$
$+next : Edge*$

The roles of attributes v and $next$ are the same as in case unweighted graphs, but w is the weight (i.e. length) of the appropriate edge. In case of undirected graphs, each edge is represented twice, but in case of digraphs, each edge is represented only once. (See Figure 5.)

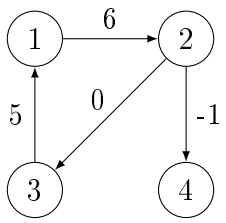
1.5 Space complexity of representing graphs

1.5.1 Adjacency matrices

Their space requirements can be calculated similarly as in the unweighted case. Provided that a float is represented in one word, the storage requirement of adjacency matrix representation is n^2 words in the general case. In case of undirected graphs, storing just the lower triangle matrix, we need $n*(n-1)/2$ words. $n*(n-1)/2 \in \Theta(n^2)$, consequently the space complexity of this representation is $\Theta(n^2)$ in both cases.

1.5.2 Adjacency lists

Each "Edge" contains an extra data member w compared to the unweighted case. Clearly, this fact does not influence the asymptotic storage requirements.



$A[1] \to v = 2$; $A[1] \to w = 6$; $A[1] \to next = \oslash$
$A[2] \to v = 3$; $A[2] \to w = 0$; $A[2] \to next \to v = 4$
; $A[2] \to next \to w = -1$; $A[2] \to next \to next = \oslash$
$A[3] \to v = 1$; $A[3] \to w = 5$; $A[3] \to next = \oslash$
$A[4] = \oslash$

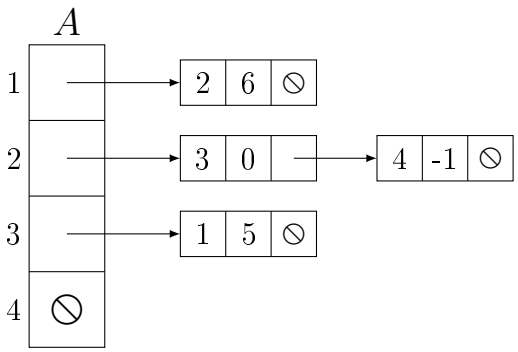
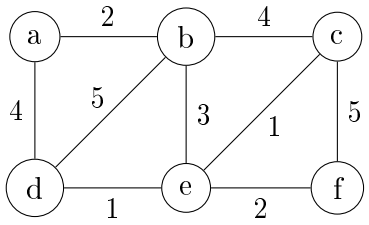


Figure 5: A weighted graph (on the left) in adjacency list representation (on the rignt).

1.6 The abstract class of weighted graphs

$\mathcal{G}_w$
$+\ V : \mathcal{V}\{\}$
$+\ E : \mathcal{E}\{\}\ //\ E \subseteq V \times V \setminus \{(u,u) : u \in V\}$
$+\ w : E \to \mathbb{R}\ //$ weights of edges

2 Minimum spanning trees (MSTs) ([3] 23)



a – b, 2 ; d, 4.
b – c, 4 ; d, 5 ; e, 3.
c – e, 1 ; f, 5.
d – e, 1.
e – f, 2.

Figure 6: Connected weighted undirected graph. The vertices can be cities, the edges are possible routes with the costs of building them. We want to ensure that we can go form each city to each other city while minimizing the costs of building the network.

In order to solve problems similar the one illustrated on Figure 6, we define the notion of *MST = Minimum Spanning Tree*. In this chapter we discuss algorithms computing the MSTs of connected weighted undirected graphs. (The weights of the edges may be negative.)

Definition 2.1 *Given undirected graph $G = (V, E)$, its* spanning forest *is graph $T = (V, F)$, if $F \subseteq E$, and T is (undirected) forest (i.e. T is an undirected graph, consisting of undirected tree components where the trees are pairwise disjunctive, and they cover V together).*

Definition 2.2 *Given undirected connected graph $G = (V, E)$, graph $T = (V, F)$ is its* spanning tree*, if $F \subseteq E$, and T is (undirected) tree.*

Definition 2.3 *Provided that $G = (V, E)$ is a weighted graph (tree, forest, etc.) with weight function $w : E \to \mathbb{R}$, then the weight of G is the sum of the weights of its edges:*

$$w(G) = \sum_{e \in E} w(e)$$

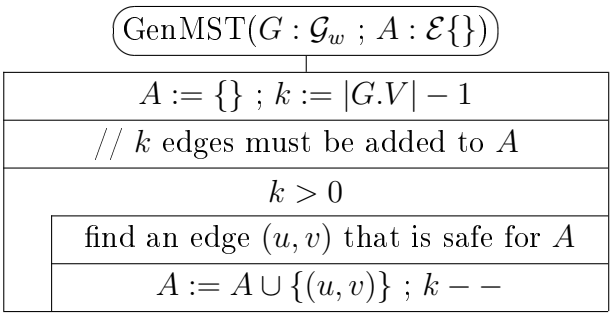
Definition 2.4 *Given undirected connected weighted graph G, T is the* minimum spanning tree*, i.e. MST of G, if*

- *T is spanning tree of G, and*
- *for each other spanning tree T' of G, $w(T) \leq w(T')$.*

Definition 2.5
We say that an A set of edges is a subset of the graph $G = (V, E)$, **iff** $A \subseteq E$.

2.1 A general method

The general algorithm below starts with the empty set $A = \{\}$. In each iteration, it adds a new edge to this set of edges. It has an invariant property: *Set A is a subset of some MST of G.* Consequently, A becomes the set of edges of an MST of G, when the size of A becomes $|G.V|-1$ (because the number of edges of a tree is always its number of vertices minus one).



Definition 2.6 *Let us suppose that $G = (V, E)$ is an undirected connected weighted graph, and the edge set A is a subset of some MST of G.*

Edge $(u, v) \in E$ is safe for A, **iff** $(u, v) \notin A$, *and $A \cup \{(u, v)\}$ is still a subset of some MST of G. (The two MSTs may be different.)*

Consequence 2.7 *Let us suppose that $G = (V, E)$ is an undirected connected weighted graph. Provided that we have the initially empty set A of edges, and in each step we add an edge to A which is safe for A, then after $|V|-1$ steps we receive an MST of G.*

Now we have the following question. *How to find an edge which is safe for A?* In order to answer it, we present the following notions and theorems.

Definition 2.8 *$(S, V \setminus S)$ is a* cut *of the graph $G = (V, E)$,* **iff** $\{\} \subsetneq S \subsetneq V$.

Definition 2.9 *An edge $(u, v) \in E$ of graph $G = (V, E)$* crosses *cut $(S, V \setminus S)$,* **iff** *one endpoint of edge (u, v) is in S and the other is in $V \setminus S$.*

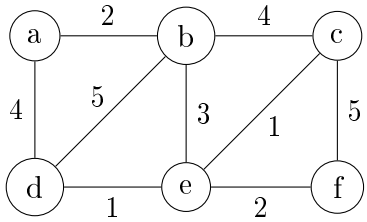
Definition 2.10 *An edge is a* light edge crossing a cut, **iff** *it crosses the cut and its weight is the minimum of the weights of the edges crossing the cut. (The expressions "*light edge crossing the cut*" and "*light edge in the cut*" are synonyms in this topic.)*

Definition 2.11 *A cut* respects *a set A of edges,* **iff** *no edge of A crosses the cut. (The verbs "*respects*" and "*avoids*" are synonyms in this topic.)*

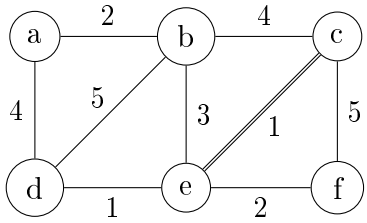
Theorem 2.12
Provided that $G = (V, E)$ is an undirected connected weighted graph, and
(1) the A set of edges is a subset *of some MST of G, and*
(2) the cut $(S, V \setminus S)$ respects *(i.e. avoids) the A set of edges, and*
(3) the edge $(u, v) \in E$ is a light edge crossing the cut $(S, V \setminus S)$
$\Longrightarrow$ *the edge (u, v) is* safe for *the A set of edges.*

Proof. $(u, v) \notin A$, because (u, v) crosses the cut, which respects A.
Let $T = (V, T_E)$ be an MST with the property $A \subseteq T_E$. There are two cases. (1) If $(u, v) \in T_E$, we are ready. (2) For $(u, v) \notin T_E$, we create a T' MST for which $A \cup \{(u, v)\}$ is a subset of T'. We note that T is a spanning tree, so in T there is exactly one path form u to v. Thus on this path there is an edge crossing the cut $(S, V \setminus S)$. Let (p, q) be such an edge. In this case, $w(p, q) \geq w(u, v) \wedge (p, q) \notin A$. Let us delete edge (p, q) from T. Then tree T becomes a spanning forest T'' of two trees. One of these trees contains vertex u, the other contains vertex v. Let us add edge (u, v) to T''. We receive a spanning tree again. Let us call it T'. As a result, $w(T') = w(T) - w(p, q) + w(u, v) \leq w(T)$. Remember that T was an MST of G. Thus $w(T') \geq w(T)$. Therefore $w(T') = w(T)$, and T' is also an MST of G. $\square$

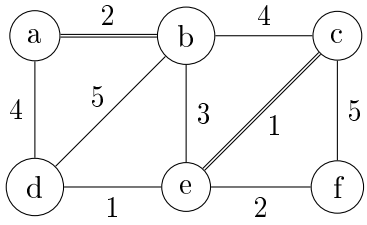
Based on the previous theorem, we have a method for building an MST. Let us denote the edges of the A set of edges with double lines. In the following example, in each step, we select a cut respecting A, we find a light edge crossing the cut, and add this edge safely to A. The example graph has 6 vertices. Thus we find an MST in 5 steps.



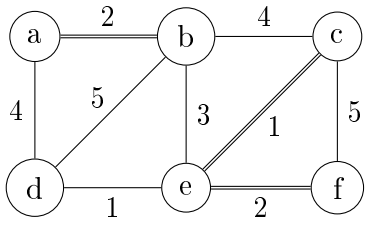
Initially $A = \{\}$, thus each cut is appropriate. Select cut $(\{a, b, c\}, \{d, e, f\})$. (c, e) is a light edge crossing this cut, consequentially it is safe for A. Add it to A.



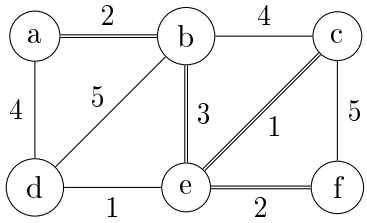
Now $A = \{(c, e)\}$. For example, cut $(\{a, f\}, \{b, c, d, e\})$ respects A. Edges (a, b) and (e, f) are the light edges crossing this cut. Both of them are safe for A. Select (a, b). Add it to A.



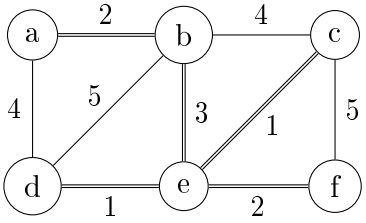
Now $A = \{(a, b), (c, e)\}$. For example, cut $(\{a, b, c, d, e\}, \{f\})$ respects A. Edge (e, f) is the light edge crossing this cut. Thus it is safe for A. Add it to A.



Now $A = \{(a, b), (c, e), (e, f)\}$. For example, cut $(\{a, b\}, \{c, d, e, f\})$ respects A. Edge (b, e) is the light edge crossing this cut. Thus it is safe for A. Add it to A.

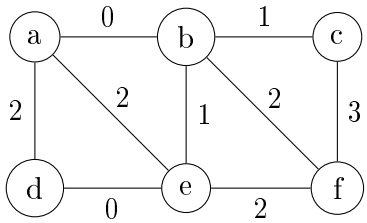


$A = \{(a, b), (b, e), (c, e), (e, f)\}$. Cut $(\{a, b, c, e, f\}, \{d\})$ respects A. Edge (d, e) is the light edge crossing this cut. Thus it is safe for A. Add it to A.



$A = \{(a, b), (b, e), (c, e), (d, e), (e, f)\}$. $|A| = 5 = |G.V| - 1$, thus A is the set of the edges of a *minimum spanning tree* (MST).

2.2 Algorithm of Kruskal



a – b, 0 ; d, 2 ; e, 2.
b – c, 1 ; e, 1 ; f, 2.
c – f, 3.
d – e, 0.
e – f, 2.

Figure 7: Connected weighted undirected graph

Components	edge	its weight	is it safe?
a, b, c, d, e, f	(a,b)	0	+
ab, c, d, e, f	(d,e)	0	+
ab, c, de, f	(b,c)	1	+
abc, de, f	(b,e)	1	+
abcde, f	(a,d)	2	-
abcde, f	(a,e)	2	-
abcde, f	(b,f)	2	+
abcdef	-	-	-

;

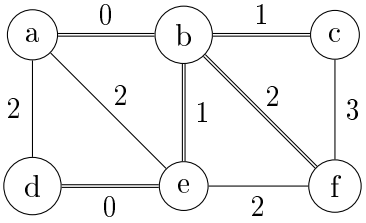
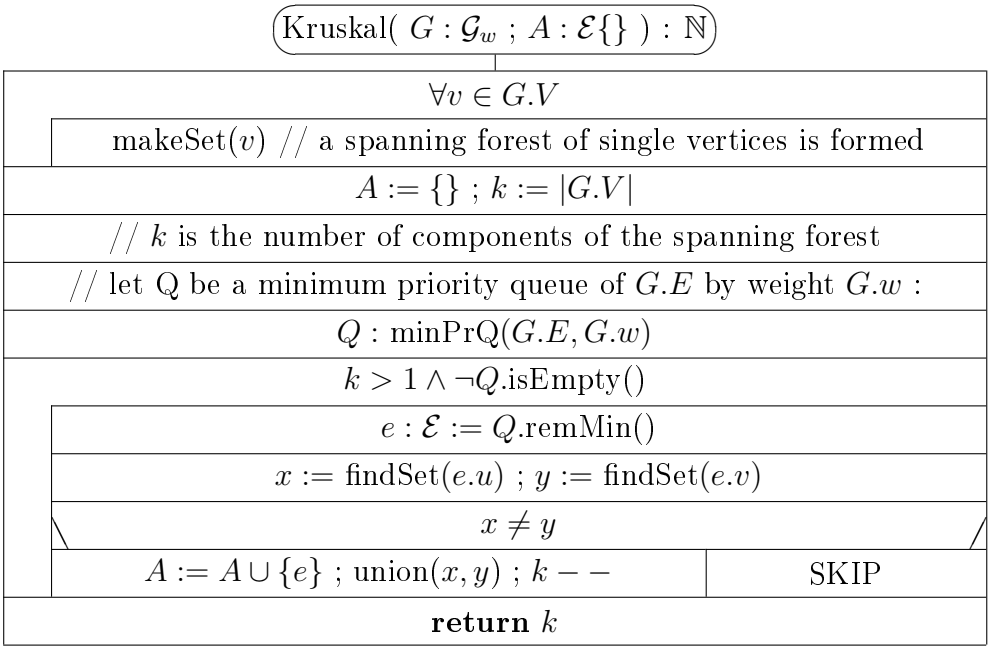


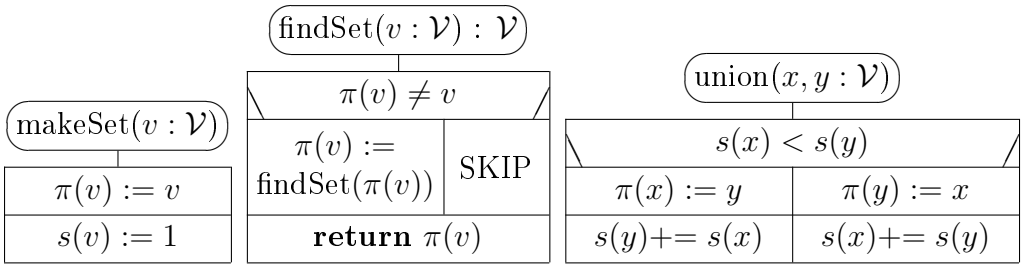
Figure 8: 7. Double lines show the edges of the MST found.



This algorithm checks whether G is connected. If so, it returns $k = 1$. Otherwise $k > 1$.

$MT(n, m) \in O(m * \lg n)$

2.2.1 The set operations of the Kruskal algorithm

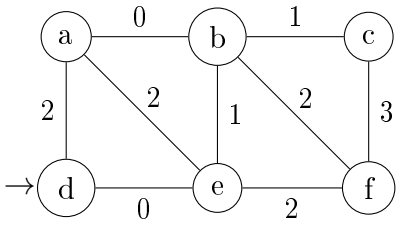


2.3 Algorithm of Prim

Prim($G : \mathcal{G}_w$; $r : \mathcal{V}$)

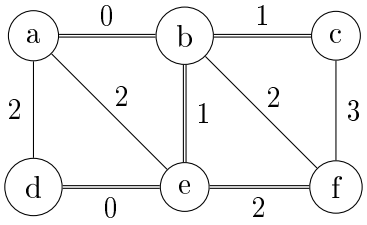
- $\forall v \in G.V$
  - $c(v) := \infty$; $p(v) := \oslash$ // costs and parents still undefined
  - // edge $(p(v), v)$ will be in the MST where $c(v) = G.w(p(v), v)$
- $c(r) := 0$ // r is the root of the MST where $p(r)$ remains $\oslash$
- // let Q be a minimum priority queue of $G.V \setminus \{r\}$ by label values $c(v)$:
- Q : minPrQ($G.V \setminus \{r\}, c$) // $c(v)$ = cost of light edge to (partial) MST
- $u := r$ // vertex $u = r$ has become the first node of the (partial) MST
- $\neg Q$.isEmpty()
  - // neighbors of u may have come closer to the partial MST
  - $\forall v : (u, v) \in G.E \wedge v \in Q \wedge c(v) > G.w(u, v)$
    - $p(v) := u$; $c(v) := G.w(u, v)$; Q.adjust(v)
  - $u := Q$.remMin() // $(p(u), u)$ is a new edge of the MST

$$MT_{\mathrm{P}rim}(n, m) \in O(m * \log n).$$



a – b, 0 ; d, 2 ; e, 2.
b – c, 1 ; e, 1 ; f, 2.
c – f, 3.
d – e, 0.
e – f, 2.

c values of nodes in Q						vertex	changes of labels p					
a	b	c	d	e	f	into	a	b	c	d	e	f
∞	∞	∞	0	∞	∞	MST	$\oslash$	$\oslash$	$\oslash$	$\oslash$	$\oslash$	$\oslash$
2	∞	∞		0	∞	d	d				d	
2	1	∞			2	e		e				e
0		1			2	b	b		b			
		1			2	a						
					2	c						
						f						
0	1	1	0	0	2	result	b	e	b	$\oslash$	d	e



Compared to the algorithm of Kruskal, using the algorithm of Prim, we have received another MST of the same graph.

3 Single-Source Shortest Paths ([3] 24)

Problem 3.1 *Given graph $G : \mathcal{G}_w$ where $s \in G.V$ is the source vertex. We search for a shortest path to each vertex available from s in G. This is called the* single-source shortest paths *problem.*

Note 3.2 *In the following algorithms the undirected graphs will be modeled with digraphs where for each edge (u, v) of the graph, (v, u) is also edge of the graph, and $w(u, v) = w(v, u)$.*

This model simplifies the forthcoming discussions. In the rest of these lecture notes, graph means digraph by default.

Definition 3.3 *A loop of a digraph is* negative, **iff** *the sum of the weights of its edges is negative.*

Consequence 3.4 *Problem 3.1 can be solved,* **iff** *there is no negative cycle available from s.*

Note 3.5 *Remember that undirected graphs are modeled with digraphs here. This simplification has a consequence: if an undirected graph contains a negative edge, it also contains a negative cycle in this model, according to definition 3.3. And in case of undirected graphs, we can solve the* single-source shortest paths *problem (i.e. 3.1),* **iff** *there is no negative edge available from s.*

Provided that problem 3.1 can be solved, we consider the result of path-finding now. We have two possibilities for each vertex $v \in G.V \setminus \{s\}$:
- If there is some path form s to v, $d(v)$ is the length of the shortest (i.e. optimal) path, and $\pi(v)$ is the parent of vertex v on such a path.
- If there is no path from s to v, $d(v) = \infty$ and $\pi(v) = \oslash$.

Considering vertex s, $d(s) = 0$ and $\pi(s) = \oslash$ are the appropriate results, because the optimal path consists of only vertex s.

The shortest path finding algorithms approximate the optimal paths step-by-step.

Let us consider a given moment of the running algorithm. Let the shortest path found form s to v be denoted by $s \rightsquigarrow v$. Let the length of this path be denoted by $w(s \rightsquigarrow v)$. Then we have a common invariant of the forthcoming algorithms solving problem 3.1, and this invariant becomes true when we initialize the algorithm:
- Provided that we already have found some path from s to v, $d(v) = w(s \rightsquigarrow v)$ and $\pi(v)$ is the parent of $v \neq s$ on $s \rightsquigarrow v$.

- If still there is no $s \rightsquigarrow v$, i.e. still we have not found any path from s to v, then $d(v) = \infty$ and $\pi(v) = \oslash$.

In order to approximate the optimal paths step-by-step, edges (u, v), i.e. $u \rightarrow v$ of the graph are considered systematically. Provided that path $s \rightsquigarrow u \rightarrow v$ turns out shorter than $s \rightsquigarrow v$, $s \rightsquigarrow u \rightarrow v$ becomes the new $s \rightsquigarrow v$. At code level this means that we perform the following conditional, which is called *relaxation*. (Because of the special features of the different algorithms, this relaxation may contain some additional statements.)

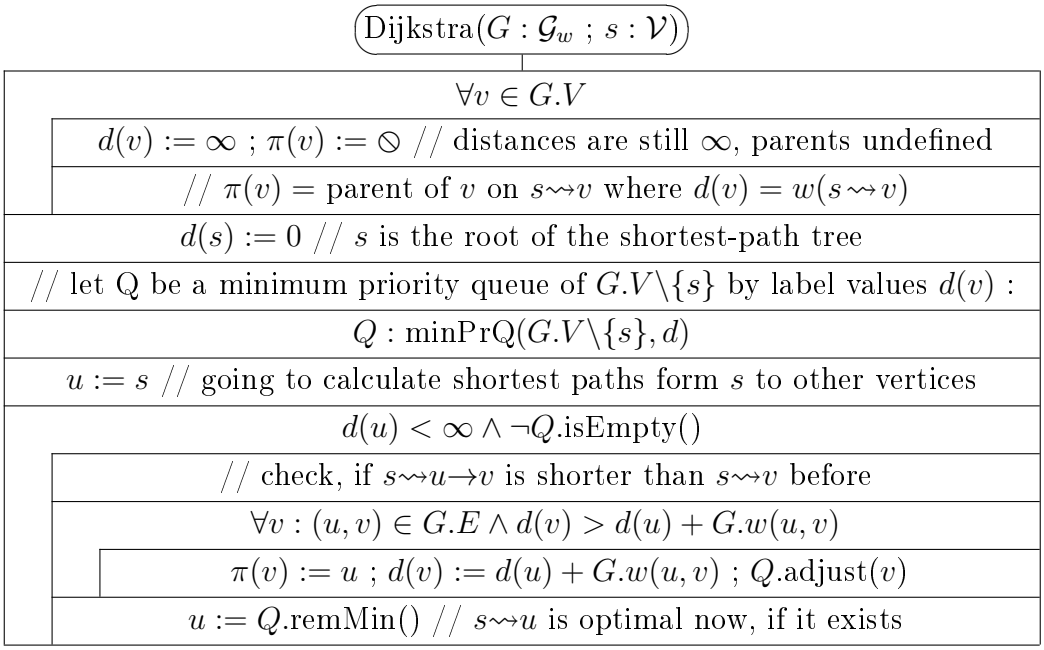
$d(v) > d(u) + G.w(u, v)$	
$\pi(v) := u \ ; \ d(v) := d(u) + G.w(u, v)$	SKIP

Many of the *single-source shortest paths* algorithms have an explicit set of the vertices *to be processed*. They repeatedly select and remove a vertex form this set, then perform relaxation on each edge going out from this vertex. All these relaxations together are called the *expansion* of this vertex. Processing a vertex means its selection + removal + expansion.

3.1 Dijkstra's algorithm

Precondition: In graph $G : \mathcal{G}_w$, $\forall (u, v) \in G.E$: $G.w(u, v) \geq 0$, i.e. each edge of the graph has non-negative weight.

In this case there is no negative cycle. Thus problem 3.1 (the single-source shortest paths problem) can be solved.



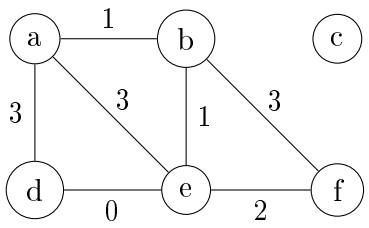
$MT_{\mathrm{Dijkstra}}(n, m) \in O((n + m) * \lg n)$. This result can be received similarly to that of algorithm Prim, because of the striking similarities between the two algorithms, although they solve completely different problems, and the interpretations of their results are also totally different. An important technical difference: here we sum up the weights of the edges along a path, but we have nothing to do with paths there.

$mT_{\mathrm{Dijkstra}}(n, m) \in \Theta(n)$, which is realized when s, i.e. the source vertex has no successor.

Exercise 3.6 *Implement Dijkstra's algorithm in case of adjacency matrix representation of the graph. Represent the priority queue with an unsorted array. What can we say about asymptotic run time efficiency (i.e. operational complexity)? Can we develop it with a more sophisticated representation of the priority queue?*

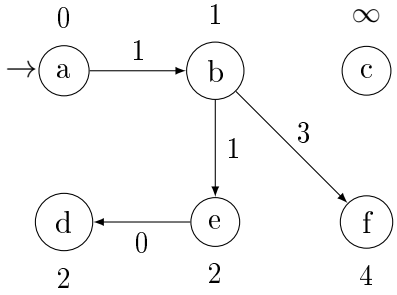
Exercise 3.7 *Implement Dijkstra's algorithm in case of adjacency lists representation of the graph. Be careful about keeping the theoretical* $MT_{\mathrm{Dijkstra}}(n, m) \in O((n + m) * \lg n)$ *and* $mT_{\mathrm{Dijkstra}}(n, m) \in \Theta(n)$. *(The appropriate representation and implementation of the priority queue is the key.)*

We illustrate Dijkstra's algorithm on the next graph where vertex **a** is the source node.



a – b, 1 ; d, 3 ; e, 3.
b – e, 1 ; f, 3.
c.
d – e, 0.
e – f, 2.

d values of nodes in Q						expanded	changes of π values					
a	b	c	d	e	f	vertex	a	b	c	d	e	f
0	∞	∞	∞	∞	∞	: d	$\oslash$	$\oslash$	$\oslash$	$\oslash$	$\oslash$	$\oslash$
	1	∞	3	3	∞	a : 0		a		a	a	
		∞	3	2	4	b : 1					b	b
		∞	2		4	e : 2				e		
		∞			4	d : 2						
		∞				f : 4						
0	1	∞	2	2	4	result	$\oslash$	a	$\oslash$	e	b	b



The shortest paths tree in case of $s = \mathrm{a}$. We display also the unavailable node/nodes with ∞ d value.

Theorem 3.8 *When we select vertex u for expansion (first with statement $u := s$, later with $u := Q.remMin()$), then we have already found optimal path to u, provided that $d(u) < \infty$.*

Theorem 3.9 *If vertex u is selected for expansion and $d(u) = \infty$, then no element of $Q \cup \{u\}$ is available form s.*

Consequence 3.10 *While Dijkstra's algorithm selects a vertex with finite d value for expansion, we already have the optimal path to this vertex.*

If $d(u) < \infty$ for each vertex selected, then the second loop of the algorithm makes Q empty (with $n-1$ iterations), and stops when it has found optimal path to each vertex of the graph.

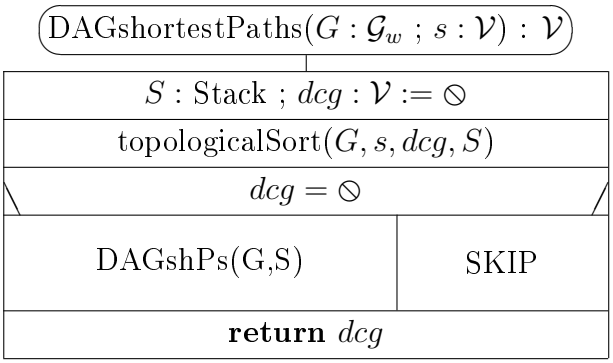
If some time it selects vertex u with $d(u) = \infty$, then no element of $Q \cup \{u\}$ is available from s. We stop, because $d(u) = \infty$. Each element of $Q \cup \{u\}$ has ∞ d value and $\oslash$ π value, while the vertices selected earlier are those available from s. And we have found optimal path to them.

3.2 Single-source shortest paths in DAGs

Precondition: Graph $G : \mathcal{G}_w$ is directed, and there is no loop available from s.

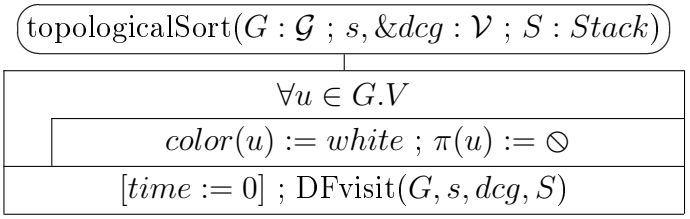
In this case there is no negative cycle available from s, and the *single-source shortest paths* problem has solution.

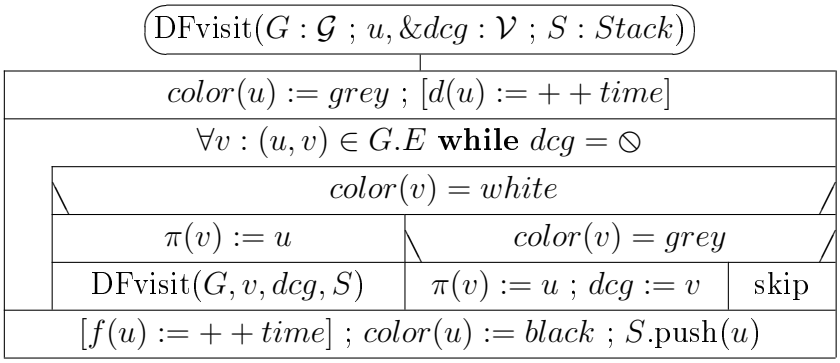
This algorithm checks its precondition. If it is satisfied, function DAGshortestPaths() returns $\oslash$. Otherwise it finds a directed loop, and it returns with some node of this loop. Starting from this vertex, and going through the π labels of the nodes. the loop can be traversed in reversed direction.

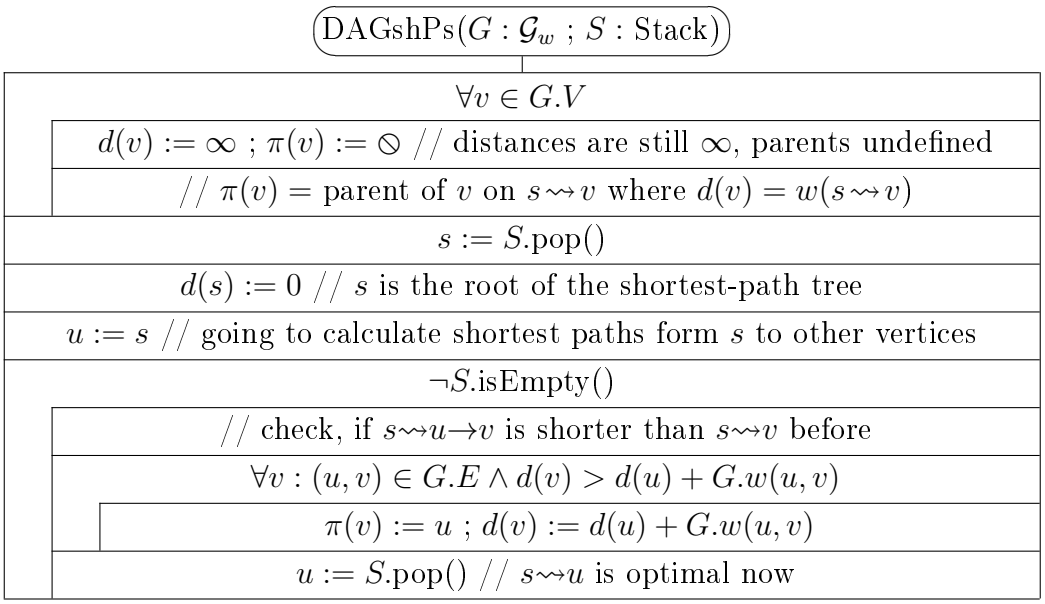


Procedure topologicalSort() tries to make a topological order of the vertices available from s.

Variable *time* (which is supposed to be global here) is needed only for the presentation of the run of the algorithm. Each statement corresponding to *time* can be omitted from an implementation. Thus these statements are put into square brackets in the structograms.





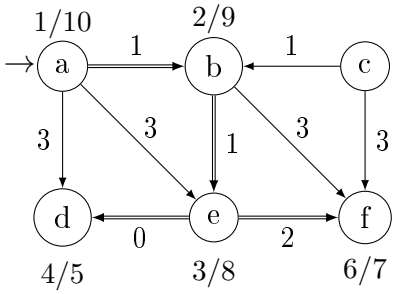


$$MT(n, m) \in \Theta(n + m) \quad \wedge \quad mT(n, m) \in \Theta(n)$$

Exercise 3.11 *When the operational complexity of this algorithm is the smallest? When it is the greatest? Why?*

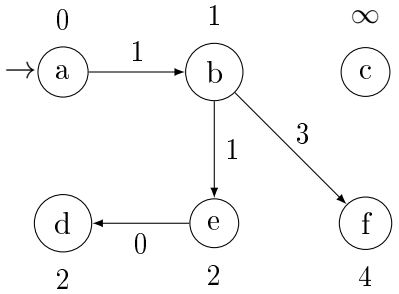
Theorem 3.12 *Provided that there is no loop available from s,*
- algorithm DAGshortestPaths() finds optimal path to the vertices available form s, and
- for each vertex x unavailable from s, it terminates with $d(x) = \infty$ and $\pi(x) = \oslash$.

We illustrate the algorithm on the DAG below where s = a. First we sort topologically the subgraph available from s. Then – after the usual initialization –, the vertices are expanded according to their topological order.



a → b, 1 ; d, 3 ; e, 3.
b → e, 1 ; f, 3.
c → b, 1 ; f, 3.
e → d, 0 ; f, 2.
$S = \langle$ a, b, e, f, d $\rangle$

changes of d values						expanded	changes of π labels					
a	b	c	d	e	f	vertex	a	b	c	d	e	f
0	∞	∞	∞	∞	∞	: d	⊘	⊘	⊘	⊘	⊘	⊘
	1		3	3		a : 0		a		a	a	
				2	4	b : 1					b	b
			2			e : 2				e		
						f : 4						
						d : 2						
0	1	∞	2	2	4	result	⊘	a	⊘	e	b	b



The shortest paths tree in case of $s =$ a. We display also the unavailable node/nodes with ∞ d value.

Exercise 3.13 *Implement the* DAG single source shortest paths *algorithm in case of* adjacency matrix *representation of the graph. What can you say about the asymptotic run time efficiency (i.e. operational complexity) of this implementation?*

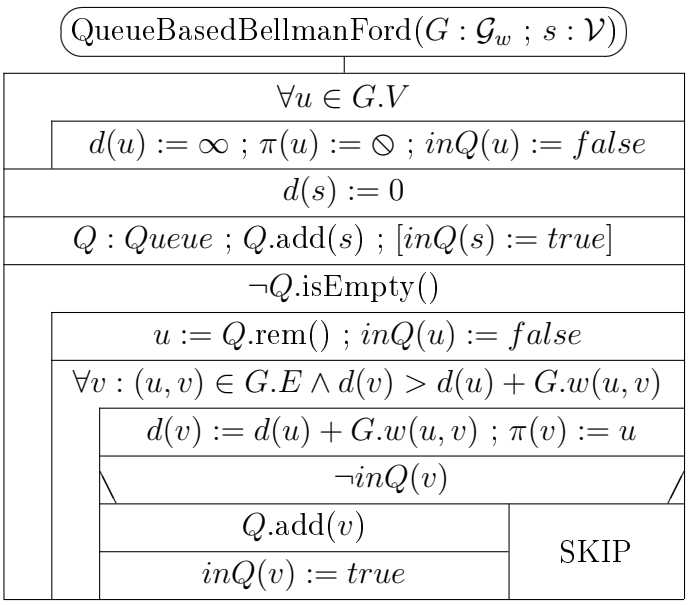
Exercise 3.14 *Implement the* DAG single source shortest paths *algorithm in case of* adjacency lists *representation of the graph. What can you say about the asymptotic run time efficiency (i.e. operational complexity) of this implementation?*

3.3 Queue-based Bellman-Ford algorithm

Precondition: There is no negative cycle available from s. (In case of directed graph there can be negative edges, i.e. edges with negative weights.)

The *Queue-based Bellman-Ford algorithm (QBF)* has unknown author (with neither theoretical analysis nor checking negative cycles). According to our best knowledge it was analyzed and published by professor Robert Endre Tarjan. He calls it *Breadth-first Scanning* in [8].[1]

QBF is similar to *breadth-first search* but the length of a path is calculated as the sum of the weights of the edges along the path. First it puts only s into the queue. Similarly to BFS, after the usual initialization, in the main loop, it repeatedly removes the first element of the queue, and expands it. Unlike BFS, QBF makes relaxation for each neighbor of this *actual node*. Provided that through the actual node, QBF finds a shorter path to a neighbor of it, QBF modifies the d and π labels of this neighbor accordingly. If this neighbor is not in the queue, QBF adds it to the end of the queue.



Because there is no operation $v \in Q$ for type Queue, we could introduce new type *Transparent_ queue*. We prefer introducing logic label $inQ(v)$ which is true $\iff$ $v \in Q$. (If we index the vertices from 1 to n, at implementation

[1] Tarjan received the Turing Award in 1986. The citation for the award states that it was: "For fundamental achievements in the design and analysis of algorithms and data structures." He has Hungarian roots.

level labels $inQ(v)$ can be represented by array $inQ/1 : \mathbb{B}[n]$. Undoubtedly this array could be part of the implementation of type *Transparent_ queue*.)

Provided that there is no negative cycle available from s, this simple version of QBF computes a path $s \overset{opt}{\rightsquigarrow} v$ for each vertex v available from s, and it stops with empty queue. If there is some negative cycle available from s, the expansions *go around* along such a negative cycle, and the algorithm gets into infinite loop.

3.3.1 Handling negative cycles

In order to handle this later case, Tarjan defined *passes* of QBF (see 3.3.5), and he proved that the algorithm stops in n passes, **iff** there is no negative cycle available from s. He also elaborated other criteria [8]. Here we use a relatively simple, easy-to-check criterion of the author of these lecture notes [10].

We introduce label $e(v)$ for each vertex v of the graph where $e(v)$ is the number of edges along $s \rightsquigarrow v$. We can prove that there is no negative cycle available from $s \iff$ during the run of QBF, for each vertex v accessed by its main loop, $e(v) < n$. (See [10] for the details.)

The statement above is clearly equivalent to the following one. There is some negative cycle available from $s \iff$ during the run of QBF, for some vertex v accessed by its main loop, $e(v) \geq n$ becomes true. In this case vertex v is part of some negative cycle which can be identified by the π labels of the vertices of this cycle, or starting from vertex v, the π labels of the vertices lead us into such a negative cycle. Obviously, the whole process of identifying this negative cycle needs maximum n steps where n is the number of the vertices of the graph.

Hereinafter, for each vertex v reached from s, we record label $e(v)$ besides labels $d(v)$ and $\pi(v)$.

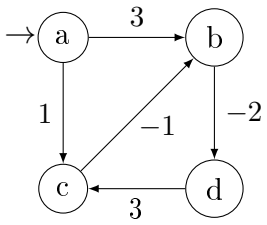
When we make a relaxation through edge (u, v), and we find that $e(v) \geq n$, then we identify some vertex of some negative cycle according to the previous method, and our version of algorithm QBF returns with this vertex.

Provided that during the run of QBF, after each relaxation $e(v) < n$, we stop with empty queue and return with $\oslash$.

It can be proved that our version of QBF stops in $O(n * m)$ time in both cases (where n is the number of vertices and m is the number of edges of the graph).

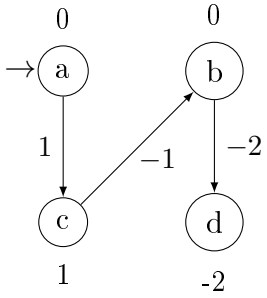
3.3.2 Illustration of finding optimal paths

Here we find a path $s \overset{opt}{\rightsquigarrow} v$ to each vertex v available from s.



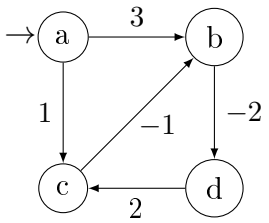
a → b, 3 ; c, 1.
b → d, −2.
c → b, −1.
d → c, 3.

changes of $d; e$				expanded	Q :	changes of π			
a	b	c	d	vertex	Queue	a	b	c	d
$0; 0$	∞	∞	∞	: $d; e$	$\langle$ a $\rangle$	⊘	⊘	⊘	⊘
	$3; 1$	$1; 1$		a: $0; 0$	$\langle$ b, c $\rangle$		a	a	
			$1; 2$	b: $3; 1$	$\langle$ c, d $\rangle$				b
	$0; 2$			c: $1; 1$	$\langle$ d, b $\rangle$		c		
				d: $1; 2$	$\langle$ b $\rangle$				
			$-2; 3$	b: $0; 2$	$\langle$ d $\rangle$				b
				d: $-2; 3$	$\langle\ \rangle$				
0	0	1	-2	final d and π values		⊘	c	a	b



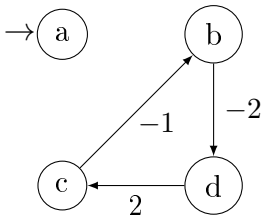
The shortest paths tree where $s = \text{a}$. We indicated only the d values of the nodes.

3.3.3 Illustration of handling negative cycles



a → b, 3 ; c, 1.
b → d, −2.
c → b, −1.
d → c, 2.

changes of $d; e$				expanded	Q :	changes of π			
a	b	c	d	vertex	Queue	a	b	c	d
$0; 0$	∞	∞	∞	: $d; e$	⟨ a ⟩	⊘	⊘	⊘	⊘
	$3; 1$	$1; 1$		a: $0; 0$	⟨ b, c ⟩		a	a	
			$1; 2$	b: $3; 1$	⟨ c, d ⟩				b
	$0; 2$			c: $1; 1$	⟨ d, b ⟩		c		
				d: $1; 2$	⟨ b ⟩				
			$-2; 3$	b: $0; 2$	⟨ d ⟩				b
		$0; 4$		d: $-2; 3$	⟨ ⟩			d	

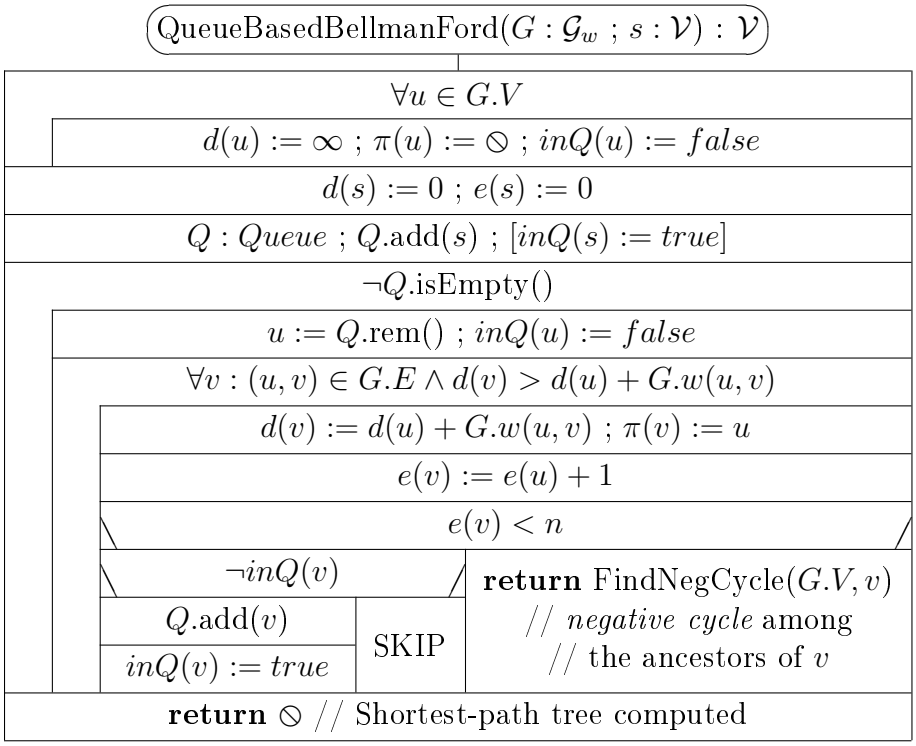


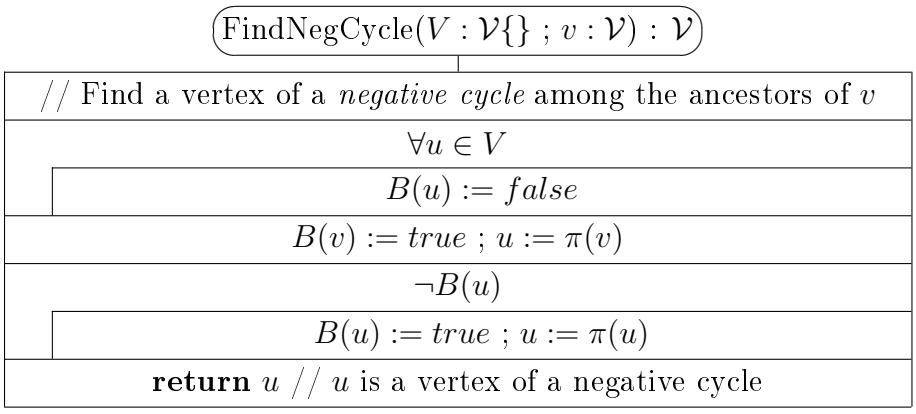
We stop here because $e(\mathrm{c}) = 4 = n$, which means that there is a negative cycle among the ancestors of vertex "c". Going back according to the π values we find the negative cycle.

3.3.4 Our version of Queue-based Bellman-Ford algorithm (QBF)

Here we give the structograms of the our version of QBF. Do not forget that when we have modified labels $d(v)$, $\pi(v)$ and $e(v)$ of vertex v, we add v to the end of the queue $\iff$ $e(v) < n$, and v is not currently in the queue.

If we know in advance that there is no negative cycle available from s, then those parts of the subsequent functions can be omitted which correspond to labels $e(v)$ or negative cycles (see the previous version of QBF).





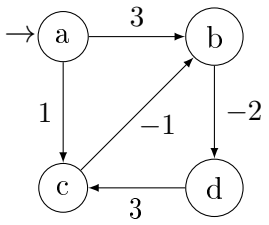
3.3.5 Analyzing QBF

We analyze only those cases when there is no negative cycle available from s.

In order to prove the correctness of QBF and to analyze its efficiency it is fundamental to divide its run into passes.

Definition 3.15 *Recursive definition of passes:*
– Pass 0: processing the source vertex (s).
– Pass (i+1): processing the vertices in the queue at the end of pass i.

For example, let us consider the illustration of our algorithm together with pass counting. **(Illustration of finding paths $s \overset{opt}{\rightsquigarrow} v$.)**



a → b, 3 ; c, 1.
b → d, −2.
c → b, −1.
d → c, 3.

changes of $d; e$				expanded	Q :	changes of π				
a	b	c	d	vertex	Queue	a	b	c	d	Pass
$0; 0$	∞	∞	∞	: $d; e$	$\langle$ a $\rangle$	⊘	⊘	⊘	⊘	
	$3; 1$	$1; 1$		a: $0; 0$	$\langle$ b, c $\rangle$		a	a		0.
			$1; 2$	b: $3; 1$	$\langle$ c, d $\rangle$				b	1.
	$0; 2$			c: $1; 1$	$\langle$ d, b $\rangle$		c			1.
				d: $1; 2$	$\langle$ b $\rangle$					2.
			$-2; 3$	b: $0; 2$	$\langle$ d $\rangle$				b	2.
				d: $-2; 3$	$\langle\ \rangle$					3.
0	0	1	-2	final d and π values		⊘	c	a	b	-

Property 3.16 *For each vertex u of the graph, if there is no negative cycle available from s, and there is some path $s \overset{opt}{\rightsquigarrow} u$ consisting of k edges, then by the beginning of pass k, $d(u) = w(s \overset{opt}{\rightsquigarrow} u)$, and $\langle s, \ldots, \pi^2(u), \pi(u), u\rangle$ is an optimal path.*

Proof. Use mathematical induction according to k. □

Lemma 3.17 *If there is no negative cycle available from s, then for each vertex u available from s, there is some path $s \overset{opt}{\rightsquigarrow} u$ consisting of maximum $n-1$ edges.*

Proof. In this case the paths containing no cycle are not longer then those containing cycle. A path containing no cycle has at most $n-1$ edges. Consequently there are finite number of paths containing no cycle. As a result, for each vertex u, there are finite number of paths from s to u which contain no cycle. Therefore there is optimal one among them. □

Theorem 3.18 *(Consequence of the Property and Lemma above)*
If there is no negative cycle available from $s \Rightarrow$ for each vertex u available

from s, there is some path $s \overset{opt}{\rightsquigarrow} u$ computed by the beginning of pass $n-1 \Rightarrow$ by the end of pass $n-1$ the queue becomes empty and the algortihm stops in $O(n*m)$ running time, i.e.

$$MT(n,m) \in O(n*m).$$

This theoretical time complexity does not guarantee efficiency. Fortunately practical tests support that in case of large, randomly generated sparse graphs ($m \in O(n)$) with positive edge-weights where the vertices of the graph are available from s, the average running time of QBF is statistically $\Theta(n)$, provided that we represent the graph with adjacency lists. And time complexity $\Theta(n)$ is the theoretical minimum. (Most of the networks can be modeled with large, sparse graphs. It is not accidental that this simple but general algorithm is often used for finding optimal paths in networks.)

4 All-Pairs Shortest Paths ([3] 25)

In this chapter, we consider how to compute the *transitive closure* of a finite binary relation. Next we introduce algorithm *Floyd-Warshall* that searches for shortest paths between each pairs of vertices of a weighted graph. Both algorithms

- are based on the adjacency representation of graphs,
- have computation complexity $\Theta(n^3)$,
- are classical examples of *dynamic programming* [3].

Floyd's *Floyd-Warshall algorithm* solves more general problem than the *Transitive Closure algorithm* of Roy and Warshall, and it is later than the other one.

Before going to the details of these algorithms, we shortly introduce dynamic programming.

4.1 Dynamic programming

Dynamic programming is similar to the *divide-and-conquer* method. It also has some trivial base case or cases which can be solved directly. It also divides more complex or larger problems into smaller subproblems, solves these subproblems and combines their solutions in order to solve the original problem. The main difference is that the *divide-and-conquer* method solves the subproblems independently from each other, like in case of *merge sort*. Thus it is effective when the subpromlems and subsubproblems etc. are typically independent from each other. It becomes inefficient when a larger problem has common subproblems and subsubproblems recursively. In *dynamic programming*, we solve each subproblem only once, and remember its solution whenever we meet that subproblem again.

For example, consider the Fibonacci function.
$F : \mathbb{N} \to \mathbb{N}$
$F_n = F_{n-1} + F_{n-2} \quad$ if $n > 1$
$F_1 = 1 \quad$ and $\quad F_0 = 0.$

Provided that $n > 1$, can divide computing F_n into computing F_{n-1} and F_{n-2}, and conquer with adding the results of the two recursive calls. As a result, for example
$F_9 = F_8 + F_7 = (F_7 + F_6) + F_7 = ([F_6 + F_5] + F_6) + (F_6 + F_5) =$
$([\{F_5 + F_4\} + F_5] + \{F_5 + F_4\}) + (\{F_5 + F_4\} + F_5) = \ldots$

Thus F_9 and F_8 are computed only once, F_7 is computed twice, F_6 is computed 3 times, F_5 is computed 5 times, and so on. In general, F_{n-k} is computed F_{k+1} times, and it follows that computing F_n needs exponential time of n.

On the contrary F_n can be computed in linear time, if we start with computing with F_2, F_3, F_4 in this order, and so on, always remembering the last two partial results. This is a trivial case of dynamic programming. In the subsequent two algorithms we use it in a much more complex way, and the partial results will be stored in matrices.

4.2 Transitive closure of a directed graph (TC)

In this subsection, for each pair of vertices (u, v) of a network/graph, we want to determine whether there is any path form u to v or not. We are interested neither in the path nor in its length. For this purpose we introduce the following notion.

Definition 4.1 *Given graph $G = (V, E)$, its transitive closure is relation $T \subseteq V \times V$ where*

$(u, v) \in T \iff$ *there is some path from vertex u to vertex v in graph G.*

Notation 4.2 $\mathbb{B} = \{0; 1\}$

Provided that the vertices of a graph can be identified by indices $1..n$, we can represent the graph with adjacency matrix $A/1 : \mathbb{B}[n, n]$. And we can represent its transitive closure with matrix $T/1 : \mathbb{B}[n, n]$ where

$T[i, j] \iff$ there is some path from vertex i to vertex j in the graph represented with matrix A.

In order to compute matrix T, let us define matrix sequence $\langle T^{(0)}, T^{(1)}, T^{(n)} \rangle$ where $T^{(n)} = T$.

Notation 4.3 *$i \overset{k}{\rightsquigarrow} j$ is a path from vertex i to vertex j where the indices of the vertices between i and j are $\leq k$ ($i > k$ and $j > k$ is possible, where $i, j \in 1..n$ and $k \in 0..n$).*

Definition 4.4 $T_{ij}^{(k)} \iff \exists\, i \overset{k}{\rightsquigarrow} j \quad (k \in 0..n \ \wedge \ i, j \in 1..n)$

Property 4.5 *(Recursive relation among the T matrices.)*

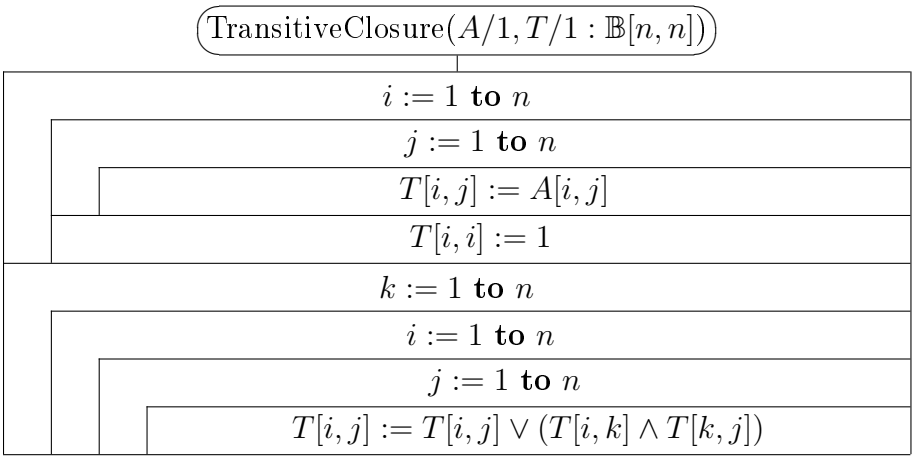
$T_{ij}^{(0)} = A[i, j] \vee (i = j) \qquad (i, j \in 1..n)$

$T_{ij}^{(k)} = T_{ij}^{(k-1)} \vee T_{ik}^{(k-1)} \wedge T_{kj}^{(k-1)} \quad (k \in 1..n \ \wedge \ i, j \in 1..n)$

Consequence 4.6
$T_{ik}^{(k)} = T_{ik}^{(k-1)} \quad \wedge \quad T_{kj}^{(k)} = T_{kj}^{(k-1)} \quad (k \in 1..n \ \wedge \ i, j \in 1..n)$

This means that column k and row k of matrix $T^{(k)}$ are the same as the appropriate column and row of matrix $T^{(k-1)}$. $(k \in 1..n)$

Let us notice that a single matrix T is enough for the whole computation, because $T_{ij}^{(k)}$ depends only on $T_{ij}^{(k-1)}$, $T_{ik}^{(k-1)}$ and $T_{kj}^{(k-1)}$, where $T_{ik}^{(k)} = T_{ik}^{(k-1)} \wedge T_{kj}^{(k)} = T_{kj}^{(k-1)}$



Clearly $T(n) \in \Theta(n^3)$ for procedure TransitiveClosure.

4.2.1 Computing transitive closure with breadth-first search

When BFS has been completed with $s = i$, the black vertices are those available from vertex i, and the white vertices are those unavailable from vertex i.

As a result, we can simplify BFS. We do not count d and π values just a boolean label $nonwhite(j)$ for each vertex j. Then $T[i, j] \iff nonwhite(j)$ where $j \in 1..n$. And this is row i of matrix T where $i \in 1..n$. Thus $MT(n, m) \in \Theta(n + m)$ for a single row of T.

Thus the whole matrix is computed with $\Theta(n*(n+m))$ maximal running time. This is $\Theta(n^2)$ on sparse graphs which is asymptotically better than algorithm TC.

But on dense graphs, this is asymptotically $\Theta(n^3)$ which means practically longer running time than that of the extremely simple TC algorithm.

4.3 The Floyd-Warshall algorithm (FW)

Notation 4.7 $\mathbb{R}_\infty = \mathbb{R} \cup \{\infty\}$

Given a weighted graph with adjacency matrix $A/1 : \mathbb{R}_\infty[n,n]$. FW computes matrix $D/1 : \mathbb{R}_\infty[n,n]$ where $D[i,j]$ is the length of an optimal path from vertex i to vertex j; or $D[i,j] = \infty$, if there is no path from i to j. FW also computes matrix $\pi/1 : \mathbb{N}[n,n]$ where $\pi[i,j]$ is the parent node of vertex j on an optimal path from i to j, if $i \neq j$ and there is some path from i to j. Otherwise $\pi[i,j] = 0$.

Precondition: There is no negative cycle in the graph. (This condition is checked by the algorithm.)
Task: FW constructs the following sequence of matrix pairs:
$\langle (D^{(0)}, \pi^{(0)}), (D^{(1)}, \pi^{(1)}), \ldots, (D^{(n)}, \pi^{(n)}) \rangle$ where $D^{(0)} = A$, $\pi^{(0)}$ and matrix pairs $(D^{(k)}, \pi^{(k)})$ $[k \in 1..n]$ can be constructed according to their properties which we are going to present, $D = D^{(n)} \wedge \pi = \pi^{(n)}$.

Notation 4.8 $i \underset{opt}{\overset{k}{\rightsquigarrow}} j$ *($k \in 1..n$) is a shortest path from vertex i to vertex j with two constraints:*

- *On this path, the indices of the vertices between vertex i and vertex j are $\leq k$.*
- *This path contains no cycle.*

Note 4.9

- *If $i = j$ then $i \underset{opt}{\overset{k}{\rightsquigarrow}} j = \langle i \rangle$.*
- *If $i \neq j$ then $\exists\, i \underset{opt}{\overset{0}{\rightsquigarrow}} j = \langle i, j \rangle \iff (i,j)$ is an edge of the graph.*
- *If $i \neq j \wedge k \in 1..n$, there are two possibilities about path $i \underset{opt}{\overset{k}{\rightsquigarrow}} j$:*
  $i \underset{opt}{\overset{k}{\rightsquigarrow}} j = i \underset{opt}{\overset{k-1}{\rightsquigarrow}} j \quad \vee \quad i \underset{opt}{\overset{k}{\rightsquigarrow}} j = i \underset{opt}{\overset{k-1}{\rightsquigarrow}} k \underset{opt}{\overset{k-1}{\rightsquigarrow}} j.$

Definition 4.10 $D_{ij}^{(k)} = \begin{cases} w(i \underset{opt}{\overset{k}{\rightsquigarrow}} j) & \text{if } i \overset{k}{\rightsquigarrow} j \text{ exists} \\ \infty & \text{if } i \overset{k}{\rightsquigarrow} j \text{ does not exist} \end{cases}$

Definition 4.11

$$\pi_{ij}^{(k)} = \begin{cases} \textit{the parent of vertex } j \textit{ on a path } \underset{opt}{i \overset{k}{\rightsquigarrow} j}, & \textit{if } i \neq j \wedge i \overset{k}{\rightsquigarrow} j \textit{ exists} \\ 0 \qquad \textit{if } i = j \vee i \overset{k}{\not\rightsquigarrow} j \textit{ does not exist} \end{cases}$$

Property 4.12 *Matrix $D^{(0)}$ is equal to adjacency matrix A of the graph, and matrix $D^{(n)}$ is equal to matrix D to be computed.*

Property 4.13

$$\pi_{ij}^{(0)} = \begin{cases} i & \textit{if } i \neq j \wedge (i,j) \textit{ is an edge of the graph} \\ 0 & \textit{if } i = j \vee (i,j) \textit{ is not edge of the graph} \end{cases}$$

And matrix $\pi^{(n)}$ is equal to matrix π to be computed.

Property 4.14 *based on note 4.9, provided that $k \in 1..n$:*
If $D_{ij}^{(k-1)} > D_{ik}^{(k-1)} + D_{kj}^{(k-1)}$
then $D_{ij}^{(k)} = D_{ik}^{(k-1)} + D_{kj}^{(k-1)} \wedge \pi_{ij}^{(k)} = \pi_{kj}^{(k-1)}$
else $D_{ij}^{(k)} = D_{ij}^{(k-1)} \wedge \pi_{ij}^{(k)} = \pi_{ij}^{(k-1)}$

Consequence 4.15 *Provided that $k \in 1..n$:*
$D_{ik}^{(k)} = D_{ik}^{(k-1)} \wedge \pi_{ik}^{(k)} = \pi_{ik}^{(k-1)} \quad \wedge \quad D_{kj}^{(k)} = D_{kj}^{(k-1)} \wedge \pi_{kj}^{(k)} = \pi_{kj}^{(k-1)}$

Note 4.16 *about the correctness of function* FloydWarshall(A, D, π) *on Figure 9. (We suppose here that the precondition is satisfied, i.e. there is no negative loop in the graph.)* We prove that $D = D^{(n)} \wedge \pi = \pi^{(n)}$ at the end of the function, where D and π are the matrices of the function, while $D^{(n)}$ and $\pi^{(n)}$ are the final theoretical matrices defined in 4.10 and 4.11.

Let us consider matrices D, π of the function and theoretical matrices $D^{(k)}$, $\pi^{(k)}$ where $k \in 1..n$.

The first double for-loop above initializes the matrices of the function as $D = D^{(0)} \wedge \pi = \pi^{(0)}$.

*A single iteration of the main loop ($k := 1$ **to** n) computes the matrix pair $(D^{(k)}, \pi^{(k)})$ from $(D^{(k-1)}, \pi^{(k-1)})$. And this computation is done in the matrix pair (D, π) of the function in the following way.*

Let us suppose that we are at the beginning of iteration k of the main loop where $D = D^{(k-1)} \wedge \pi = \pi^{(k-1)}$. (We have seen that it is true for $k = 1$.)

Now let us suppose that we arrive at condition $D[i,j] > D[i,k] + D[k,j]$ where $i, j \in 1..n$, and for the earlier iterations $i = i', j = j'$ of the inner, double loop $D[i',j'] = D_{i'j'}^{(k)} \wedge \pi[i',j'] = \pi_{i'j'}^{(k)}$ has been ensured.

Clearly $D[i,j] = D_{ij}^{(k-1)} \wedge \pi[i,j] = \pi_{ij}^{(k-1)}$, because $D[i,j]$ and *$\pi[i,j]$ has not been updated yet. And $D[i,k] = D_{ik}^{(k-1)} = D_{ik}^{(k)} \wedge D[k,j] = D_{kj}^{(k-1)} = D_{kj}^{(k)}$*

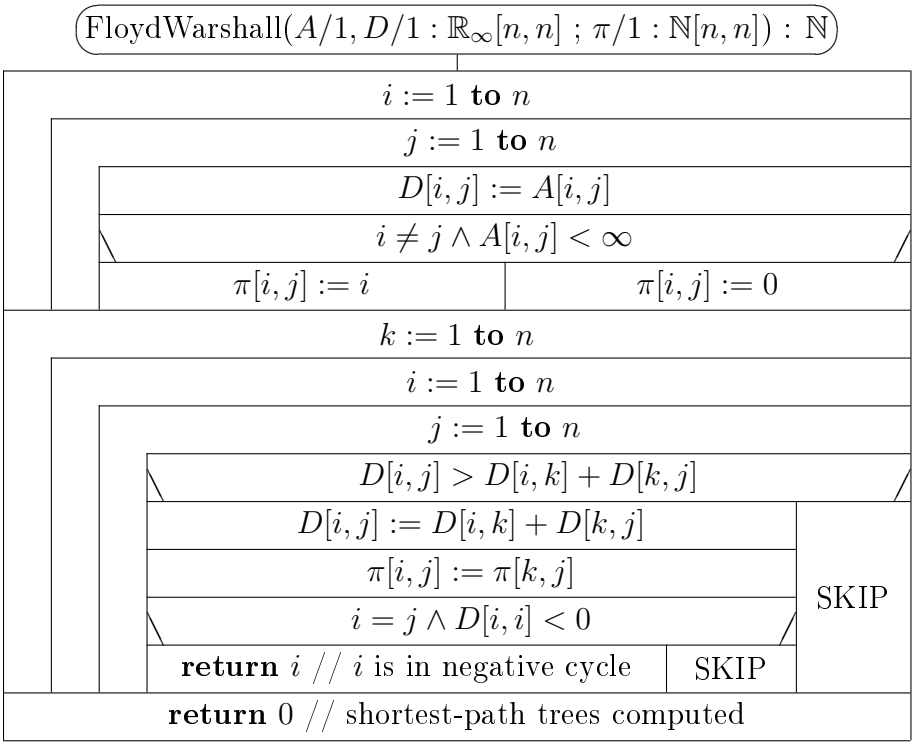


Figure 9: Algorithm FW – $MT(n) \in \Theta(n^3) \ \wedge \ mT(n) \in \Theta(n^2)$

$\wedge\ \pi[k,j] = \pi_{kj}^{(k-1)} = \pi_{kj}^{(k)}$ *according to Consequence 4.15. Performing this conditional statement* $D[i,j] = D_{ij}^{(k)} \wedge \pi[i,j] = \pi_{ij}^{(k)}$ *becomes true according to Property 4.14. (Clearly,* $D[i,i] < 0$ *becomes never true if there is no negative cycle in the graph.)*

Thus performing all the iterations of the inner double loop $D[i,j] = D_{ij}^{(k)} \wedge \pi[i,j] = \pi_{ij}^{(k)}$ *becomes true for all* $i,j \in 1..n$*. This means that* $D = D^{(k)} \wedge \pi = \pi^{(k)}$ *at the end of the actual iteration of the main loop. Therefore* $D = D^{(k-1)} \wedge \pi = \pi^{(k-1)}$ *will be true at the beginning of the next iteration for* k*.*

Consequently (by mathematical induction), $D = D^{(n)} \wedge \pi = \pi^{(n)}$ *will be true after the last iteration of the main loop, where* $k = n$*.*

Note 4.17 *If a negative value appears in the main diagonal of matrix* D *of the following program, then we have found a negative cycle. (Let the reader argue about this statement.)*

Note 4.18 *As a result of algorithm Transitive Closure,* $T[i,j] \iff D[i,j] < \infty$ *when algorithm FW has been finished* $(i,j \in 1..n)$*.*

Practically speaking, algorithm Transitive Closure solves its own task more efficiently than algorithm FW its own one, because of the simpler bodies of the loops.

4.3.1 Solving the All-Pairs Shortest Paths problem with the Single-Source Shortest Paths algorithms

Let us notice that the solution of the Floyd-Warshall (FW) algorithm, namely the ith rows of matrices D and π computed with it supply a solution of the Single-Source Shortest Paths problem for $s = i$, provided that there is no negative loop in the input graph.

Similarly, solving the Single-Source Shortest Paths problem for each $s \in 1..n$, we also receive a solution of the All-Pairs Shortest Paths problem.

Below we consider some cases.

Provided that our graph is a DAG, we can call the DAG Shortest Paths algorithm for each $s \in 1..n$, and a solution of the All-Pairs Shortest Paths problem can be computed with worst-case time complexity $MT(n, m) \in \Theta(n * (n + m))$.

For sparse graphs, where $m \in O(n)$, $\Theta(n * (n + m)) = \Theta(n^2)$, thus $MT(n, m) \in \Theta(n^2)$. This means that on sparse graphs, performing n times the DAG Shortest Paths algorithm is faster by an order of magnitude than performing the FW algorithm which runs in $MT(n), mT(n) \in \Theta(n^3)$ on DAGs.

For dense graphs, where $m \in \Theta(n^2)$, $\Theta(n * (n + m)) = \Theta(n^3)$, thus $MT(n, m) \in \Theta(n^3)$ which may cause slower run than that of FW, because of the higher constants hidden in the Θ-notation.

We receive similar results, if we apply Breadth-First Search versus FW to unweighted graphs.

Provided that our graph has no negative edge, we can call the Dijkstra algorithm for each $s \in 1..n$, and a solution of the All-Pairs Shortest Paths problem can be computed with worst-case time complexity $MT(n, m) \in O(n * (n + m) * \log n)$.

For sparse graphs, where $m \in O(n)$, $O(n*(n+m)*\log n) = O(n^2 * \log n)$, thus $MT(n, m) \in O(n^2 * \log n)$. This means that on sparse graphs, performing n times the Dijkstra algorithm is also faster by an order of magnitude than performing the FW algorithm which runs in $MT(n), mT(n) \in \Theta(n^3)$ on graphs with non-negative edges.

For dense graphs, where $m \in \Theta(n^2)$, $O(n*(n+m)*\log n) = O(n^3 * \log n)$, thus $MT(n, m) \in O(n^3 * \log n)$ which may cause slower run than that of FW, because $n^3 * \log n$ is asymptotically greater than n^3.

If we know only that the input graph contains no negative cycle, performing QBF for each $s \in 1..n$, $MT(n,m) \in O(n * n * m) = O(n^2 * m)$. Provided that our graph is **not** extremely sparse, i.e. we can suppose that $m \in \Omega(n)$, the upper estimate of the asymptotic running time of $n*$QBF is never better than the asymptotic running time of the FW algorithm which runs in $MT(n), mT(n) \in \Theta(n^3)$ on graphs with no negative cycle.