Artificial Intelligence

Shortest Path Problem

G. Guérard

Department of Nouvelles Energies
Ecole Supérieur d'Ingénieurs Léonard de Vinci

Lecture 3

Outline

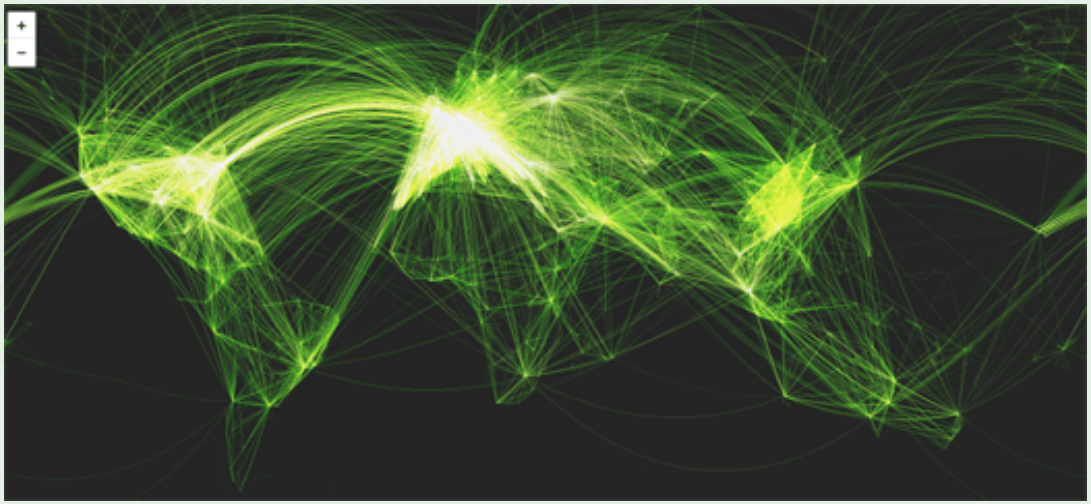
Overview

Weighted Graphs

In a weighted graph, each edge has an **associated numerical value**, called the **weight** of the edge. It may represent *distances, costs, etc.*

Example

In a flight route graph, the weight of an edge represents the distance in miles between the endpoint airports.



Applications

SHORTEST-PATHS is a broadly usefull problem-solving model:

- **Maps:** *traffic, traffic planning, traffic congestion pattern, robot navigation*
- **Telecommunication:** *routing, protocols, pipelining*
- **Software:** *texture mapping, typesetting, subroutine*
- **Finance:** *approximating piecewise linear functions, exploiting arbitrage opportunities, scheduling.*

Paradigm

Shortest path

Given a **weighted graph** and **two vertices**, we want to find *a path of minimum total weight*. Length of a path is the sum of the weights of its edges.

How can we solve this problem? Which paradigm is usefull?

1. BRUTE-FORCE: find all paths, pick the shortest.
2. NAIVE: build a spanning tree, find the unique path between the two vertices (no optimality).
3. RECURSIVE: use graph search algorithm and store the shortest path (no optimality).
4. DYNAMIC PROGRAMMING: *we have to verify the conditions...*

Linear Programming

Given a directed graph $G = (V, A)$ with source node s, target node t, and cost w_{ij} for each edge (i, j) in A, consider the program with variables x_{ij}

$$\text{minimize } \sum_{ij \in A} w_{ij} x_{ij} \text{ subject to } x \geq 0 \text{ and for all } i,$$

$$\sum_j x_{ij} - \sum_j x_{ji} = \begin{cases} 1, & \text{if } i = s; \\ -1, & \text{if } i = t; \\ 0, & \text{otherwise.} \end{cases}$$

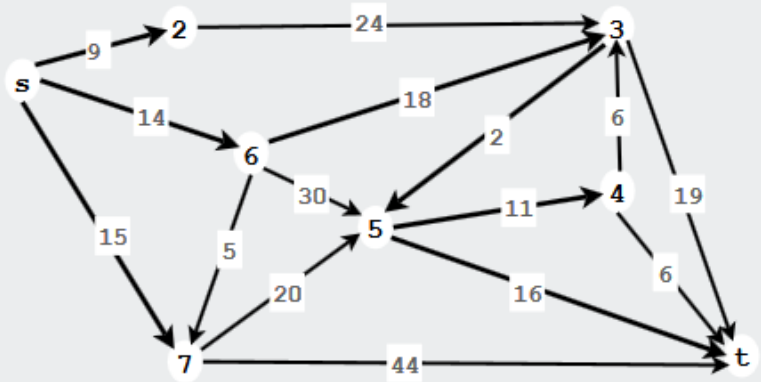
The intuition behind this is that x_{ij} is an indicator variable for whether edge (i, j) is part of the shortest path: 1 when it is, and 0 if it is not. We wish to select the set of edges with minimal weight, subject to the constraint that this set forms a path from s to t .
The dual for this linear program is (*see the example next slide*):
maximize $y_t - y_s$ subject to for all ij, $y_i - y_i \leq w_{ij}$

Linear Programming

One variable per vertex, one inequality per edge.

minimize	x_t
subject to the constraints	$x_s + 9 \le x_2$
	$x_s + 14 \le x_6$
	$x_s + 15 \le x_7$
	$x_2 + 24 \le x_3$
	$x_3 + 2 \le x_5$
	$x_3 + 19 \le x_t$
	$x_4 + 6 \le x_3$
	$x_4 + 6 \le x_t$
	$x_5 + 11 \le x_4$
	$x_5 + 16 \le x_t$
	$x_6 + 18 \le x_3$
	$x_6 + 30 \le x_5$
	$x_6 + 5 \le x_7$
	$x_7 + 20 \le x_5$
	$x_7 + 44 \le x_t$
	$x_s = 0$
	$x_2, \dots, x_t \ge 0$

interpretation:
x_i = length of shortest path from source to i



Linear Programming

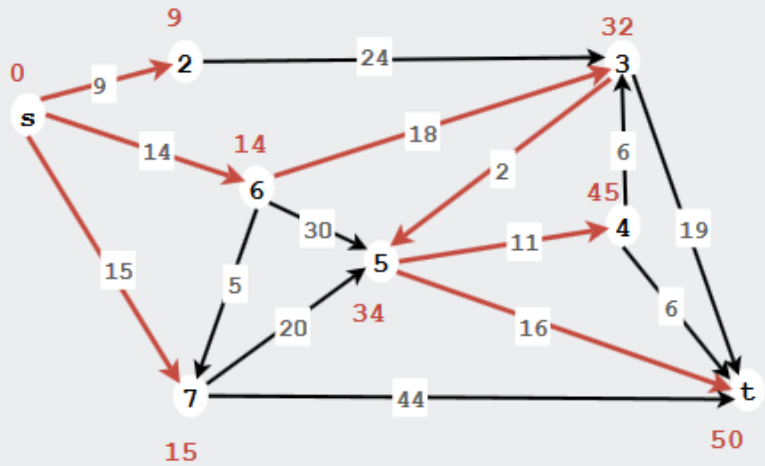
One variable per vertex, one inequality per edge.

minimize x_t

subject to the constraints

$x_s + 9 \le x_2$
$x_s + 14 \le x_6$
$x_s + 15 \le x_7$
$x_2 + 24 \le x_3$
$x_3 + 2 \le x_5$
$x_3 + 19 \le x_t$
$x_4 + 6 \le x_3$
$x_4 + 6 \le x_t$
$x_5 + 11 \le x_4$
$x_5 + 16 \le x_t$
$x_6 + 18 \le x_3$
$x_6 + 30 \le x_5$
$x_6 + 5 \le x_7$
$x_7 + 20 \le x_5$
$x_7 + 44 \le x_t$
$x_s = 0$
$x_2, \ldots, x_t \ge 0$

interpretation: x_i = length of shortest path from source to i



solution

$x_s = 0$
$x_2 = 9$
$x_3 = 32$
$x_4 = 45$
$x_5 = 34$
$x_6 = 14$
$x_7 = 15$
$x_t = 50$

Bellman's Principle of Optimality

Principle of Optimality

An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision.
A problem that can be broken apart like this is said to have optimal substructure.

Also known as a **dynamic programming equation**, it is a **necessary condition for optimality** associated with the mathematical optimization method known as dynamic programming. It writes the value of a decision problem at a certain point in time in terms of the payoff from some initial choices and the value of the remaining decision problem that results from those initial choices. This breaks a dynamic optimization problem into simpler subproblems.

Bellman's Principle of Optimality

Property 1

A SUBPATH OF A SHORTEST PATH IS ITSELF A SHORTEST PATH.

Let $\Omega^*(x_i, x_j)$ be shortest path between x_i,$x_j \in V^2$ and $x_k \in \Omega^*$ a point on the path.
Then, principle of optimality give that $\Omega(x_i, x_k)$ and $\Omega(x_k, x_j)$ are shortest sub-paths between x_i,x_k and x_k,x_j.
Suppose there exits a shorter path $\Omega^{'}(x_i, x_k)$. Thus,
$\Omega^{'}(x_i, x_j) = \Omega^{'}(x_i, x_k) + \Omega(x_k, x_j) < \Omega(x_i, x_k) + \Omega(x_k, x_j)$.
$\Omega^{'}(x_i, x_j) < \Omega^*(x_i, x_j)$ so there is a contradiction to t $\Omega^*(x_i, x_j)$ being shortest path.

Tree of shortest paths

Property 2

THERE IS A TREE OF SHORTEST PATHS FROM A START VERTEX TO ALL THE OTHER VERTICES.

First Idea

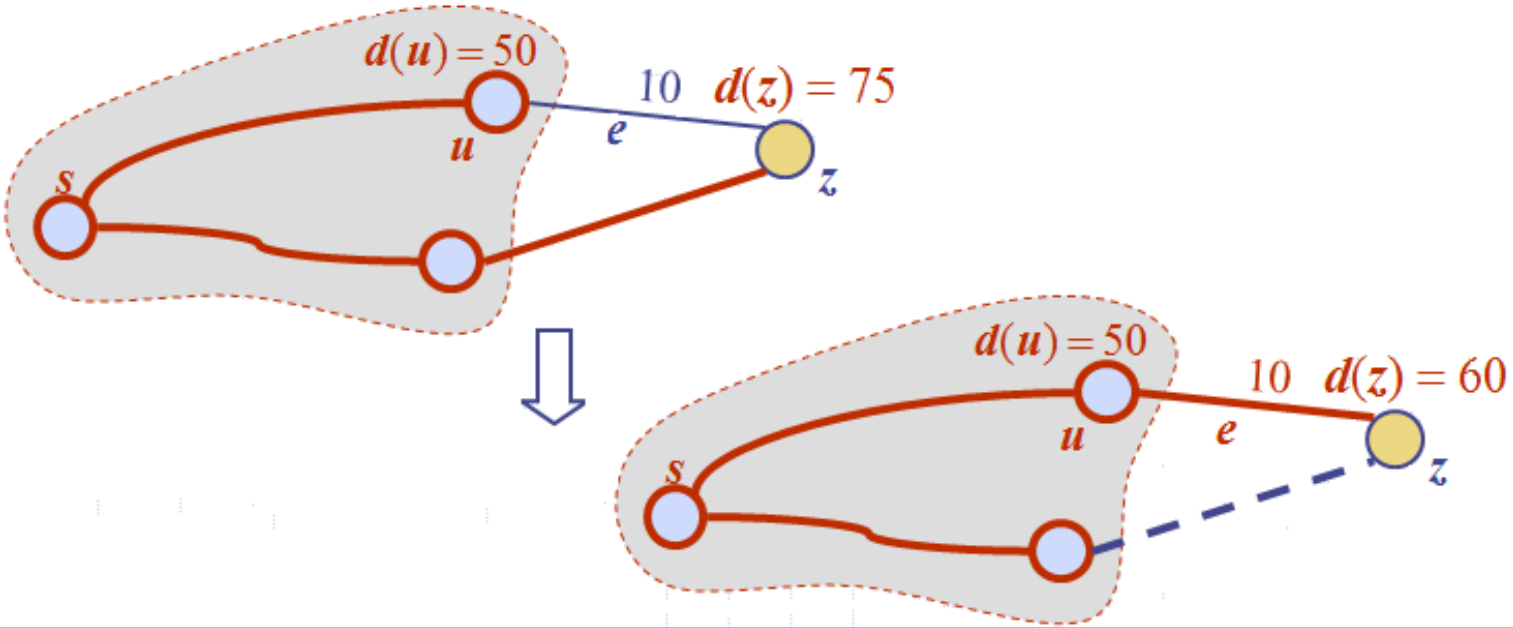
Assume the graph is connected, edges are undirected and weights are nonnegative. We grow a cloud of vertices, beginning with s and eventually covering all the vertice. We store with each vertex v a label $d(v)$ representing the distance of v from s in the subgraph consisting of the cloud and its adjacent vertices. At each step: we add to the cloud the vertex u outside the cloud with the smallest distance label; we update the labels of the vertices adjacent to u (*see spanning tree*).

Tree of shortest paths

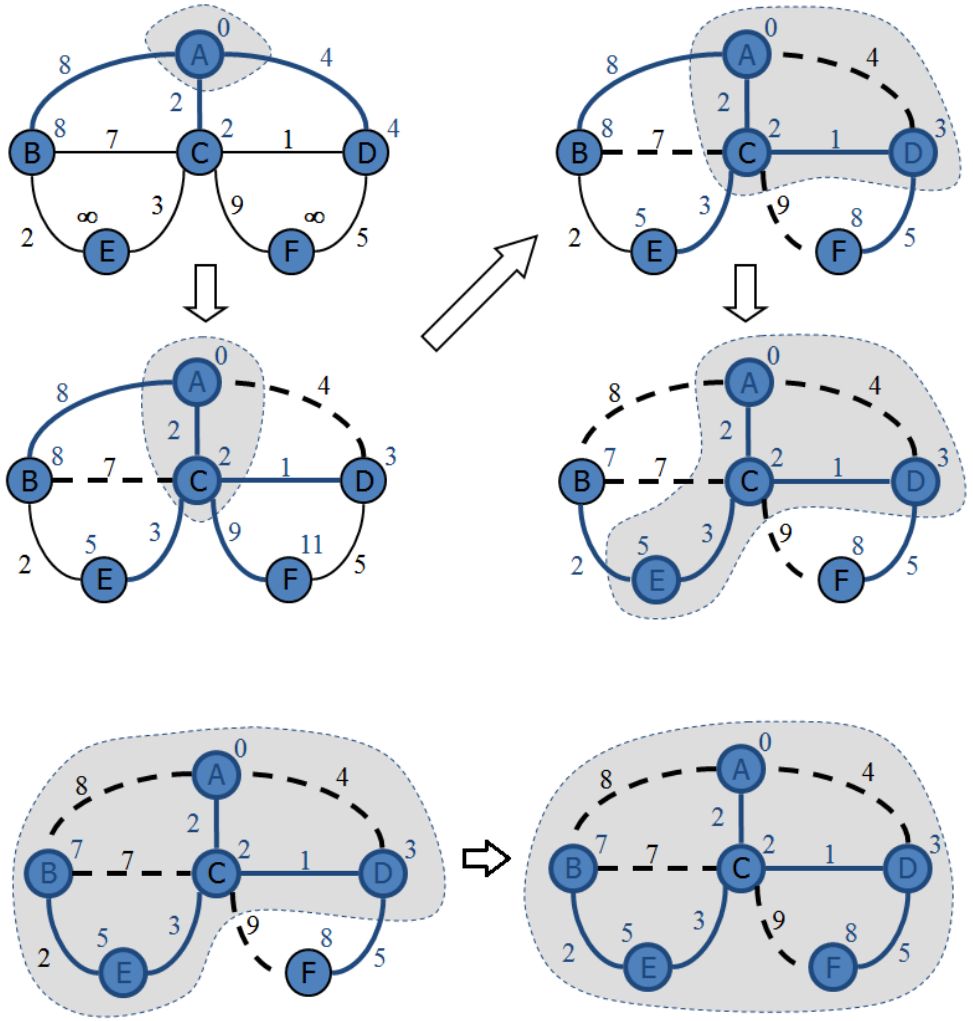
Consider the previous conditions.

Edge relaxation

Consider an edge $e = (u, z)$ such that u is the vertex most recently added to the cloud and z is not in the cloud. The relaxation of edge e updates distance $d(z)$ as follows
$d(z) \leftarrow min\{d(z), d(u) + weight(e)\}$



Example



Definition

Observation

Once we determine the shortest path to a vertex v, then the paths that continue from v to each of its adjacent vertices could be the shortest path to each of those neighbour vertices.

Visited vertex

A vertex for which we have determined the shortest path to it. Once we set a vertex as VISITED, that is final, and we won't visit that vertex again.

Marked vertex

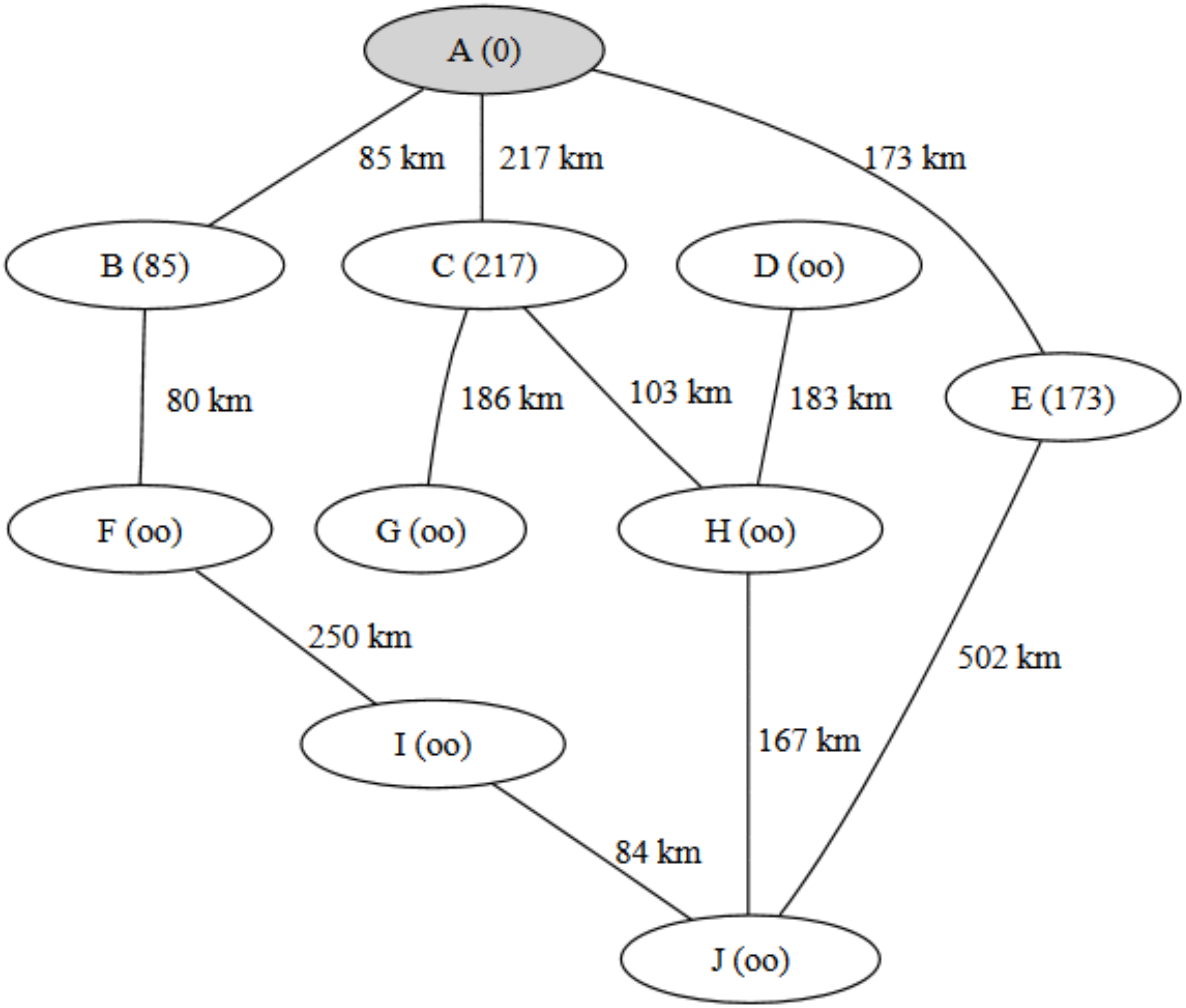
A vertex for which a path to it has been found. We mark that path as a CANDIDATE for shortest path to that vertex.

Algorithm

DIJKSTRA's algorithm follows these steps:

- *Initialize:* $d(start) = 0$, *otherwise* $d(i) = \infty$
- *At each iteration:*
  - *Select the unvisited vertex with smallest non-*∞ *distance, denoted* a. *Set it as visited.*
  - *Mark each of the vertices* b *adjacent to* a *(its neighbours)*
    - *If a neighbour was not marked, set its distance to* a*'s distance plus the weight of the edge going to that neighbour.*
    - *If it was marked, overwrite its distance if the result is smaller than its current distance.*
    - *i.e.* $d[b] = \min(d[b], d[a] + weight(a, b))$.
- *Ends when we visit the target vertex or no more non-*∞ *distances.*

Example

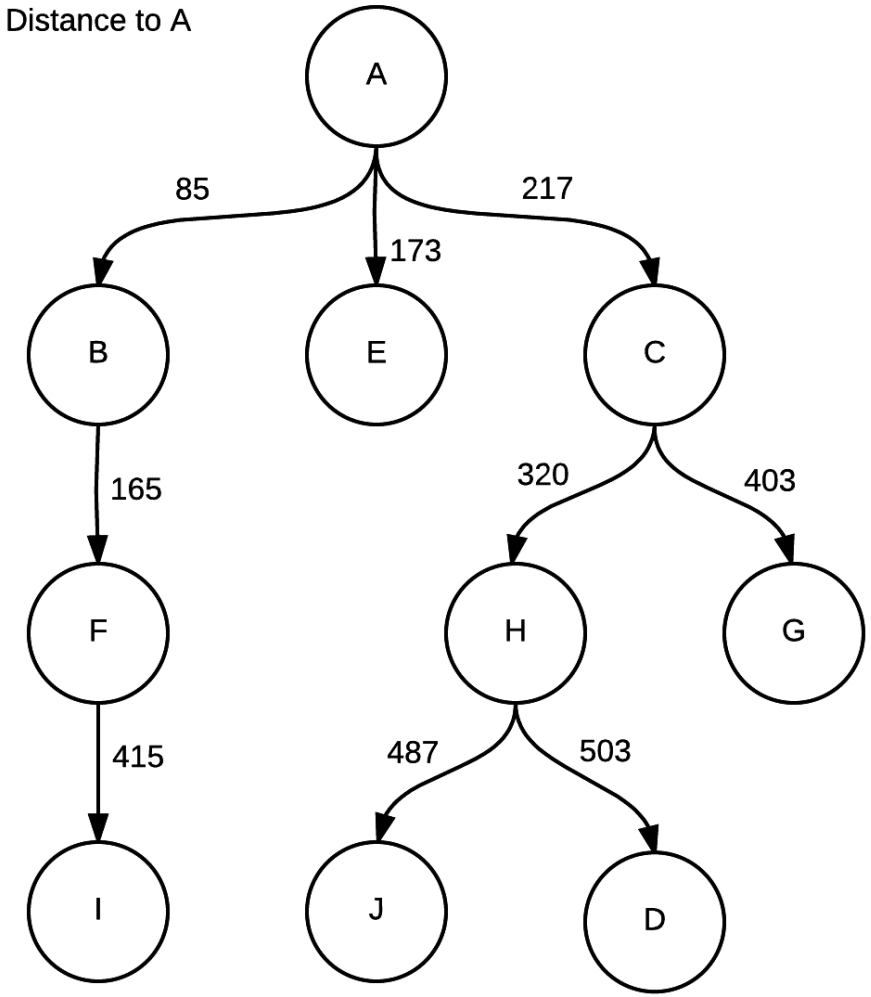

A (0)
85 km
217 km
173 km
B (85)
C (217)
D (oo)
E (173)
80 km
186 km
103 km
183 km
F (oo)
G (oo)
H (oo)
250 km
502 km
I (oo)
167 km
84 km
J (oo)

Example

distance	A	B	C	D	E	F	G	H	I	J
initiate	0	∞	∞	∞	∞	∞	∞	∞	∞	∞
$A(0)$		**85(A)**	217(A)	∞	173(A)	∞	∞	∞	∞	∞
$B(85_A)$			217(A)	∞	173(A)	**165(B)**	∞	∞	∞	∞
$F(165_B)$			217(A)	∞	**173(A)**		∞	∞	415(F)	∞
$E(173_A)$			**217(A)**	∞			∞	∞	415(F)	675(E)
$C(217_A)$				∞			403(C)	**320(C)**	415(F)	675(E)
$H(320_C)$				503(H)			**403(C)**		415(F)	487(H)
$G(403_C)$				503(H)					**415(F)**	487(H)
$I(415_F)$				503(H)						**487(H)**
$J(503_H)$				**503(H)**						

$B(85_A)$ means that we marked B, with $d(B) = 85$ and on the shortest paths' tree, A is the parent of B.

Example



Conditions

Dijkstra's algorithm can be used if the following conditions are true:

1. *No negative weight:*
    1. *we valid the shortest path at each step*
    2. *if there are only positive weight, after adding a new vertex, we add the adjacent edge to update the marked vertices. Thus, once a path is valided, we cannot found a shortest path than it.*
    3. *if there are negative weight, it is possible to find a path, through visited or non visited vertices, which have a smaller distance than the known paths. We cannot guaranted a shortest path at each iteration.*
2. *Oriented or non-oriented graph (if possible connected).*
3. *Finite number of vertices.*
4. *An unique source (see the initialization).*

Dijkstra's algorithm is a greedy dynamic programming algorithm, it visits all possible solutions.

Definition

As shown in the previous slide, a negative weight make Dijkstra's algorithm obsolete. BELLMAN'S algorithm, also called DAG (directed acyclic graph) algorithm can compute a shortest path in a directed acyclic graph.

Observation

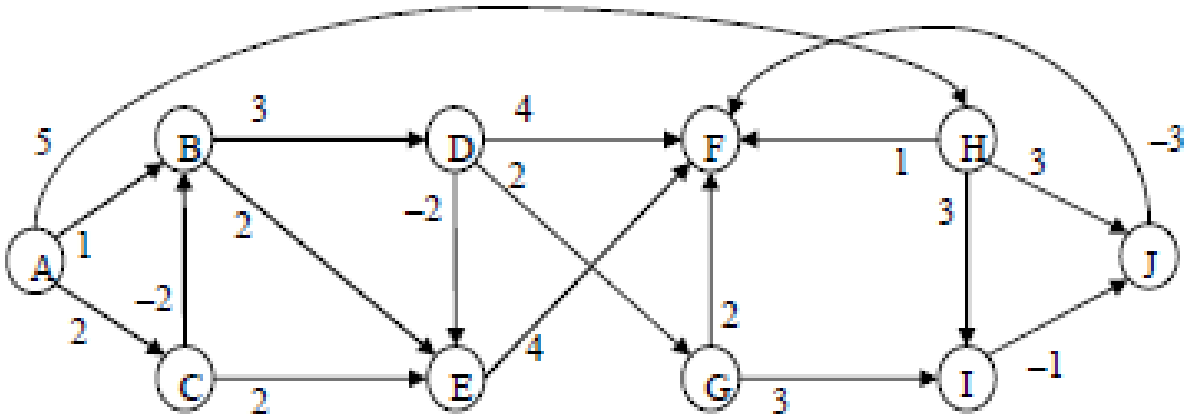
If we know all edge (u, v) with u in the cloud and v outside the cloud. We know all shortest paths to u, so we can compute $d(v) = min\{d(u) + weight(u, v)\}$.

Algorithm

BELLMAN's algorithm follows these steps:

- *Initialize:* $d(start) = 0$, *otherwise* $d(i) = \infty$
- *At each iteration:*
  - *Select the unvisited vertex which have all its predecessor visited.*
  - *Mark each of the vertices* b *adjacent to* a *(its neighbours)*
    - *If a neighbour was not marked, set its distance to* a*'s distance plus the weight of the edge going to that neighbour.*
    - *If it was marked, overwrite its distance if the result is smaller than its current distance.*
    - *i.e.* $d[b] = \min(d[b], d[a] + weight(a, b))$.
- *Ends when we visit the target vertex or no more valid vertices.*

Example



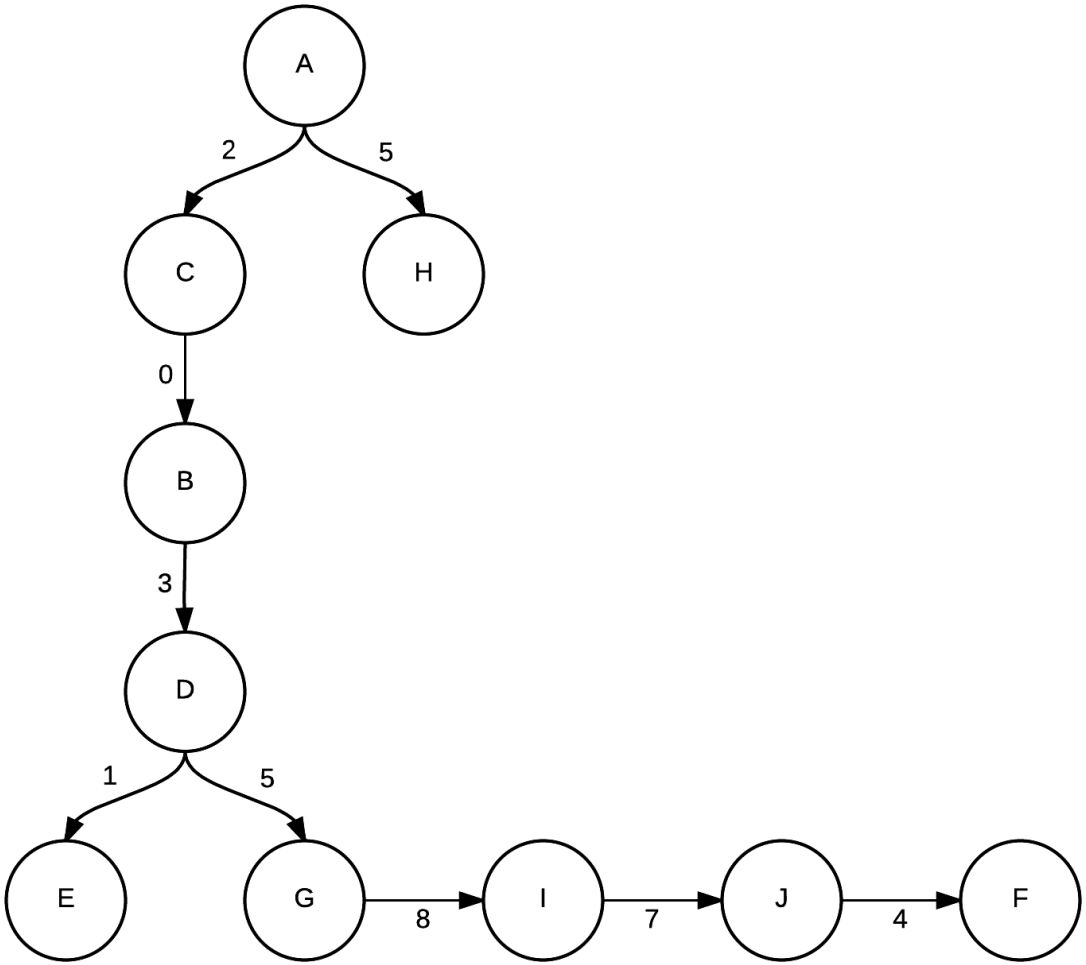
*A*is a start vertex (source), *F* is an end vertex (sink).

Example

iteration	A	B	C	D	E	F	G	H	I	J
0	0	∞	∞	∞	∞	∞	∞	∞	∞	∞
1		1(A)	**2(A)**	∞	∞	∞	∞	5(A)	∞	∞
2		**0(C)**		∞	4(C)	∞	∞	5(A)	∞	∞
3				**3(B)**	2(B)	∞	∞	5(A)	∞	∞
4					**1(D)**	7(D)	5(D)	5(A)	∞	∞
5						5(E)	**5(D)**	5(A)	∞	∞
6						5(E)		**5(A)**	8(G)	∞
7						5(E)			**8(G)**	8(H)
8						5(E)				**7(I)**
9						**4(J)**				

3(B) means that the distance from A is 3, and its predecessor is B

Example

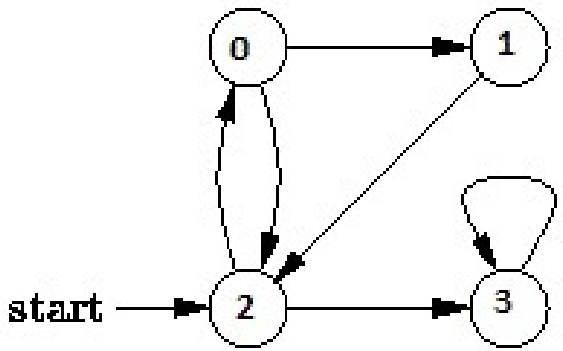


Conditions

Bellman's algorithm can be used if the following conditions are true:

1. *Positive or negative weight with no cycle:*
   1. *we valid a vertex when all its predecessor are visited*
   2. *if there is a cycle, there is a path $(v_1, \ldots, v_i, v_1)$. To visite v_1, we have to visite all predecessors of v_1, including v_i.*
   3. *by recurrence, we can visite v_i if we visite v_1. This is not possible.*
2. *Oriented or non-oriented graph (if possible connected).*
3. *Finite number of vertices.*
4. *An unique source (see the initialization).*

Bellman's algorithm is a greedy dynamic programming algorithm (similar to a bread first search), it visits all possible solutions.



Definition

BELLMAN-FORD's algorithm can compute a shortest path in any graph.

Observation

If a graph contains a negative-weight cycle, then some shortest paths may not exist

Algorithm

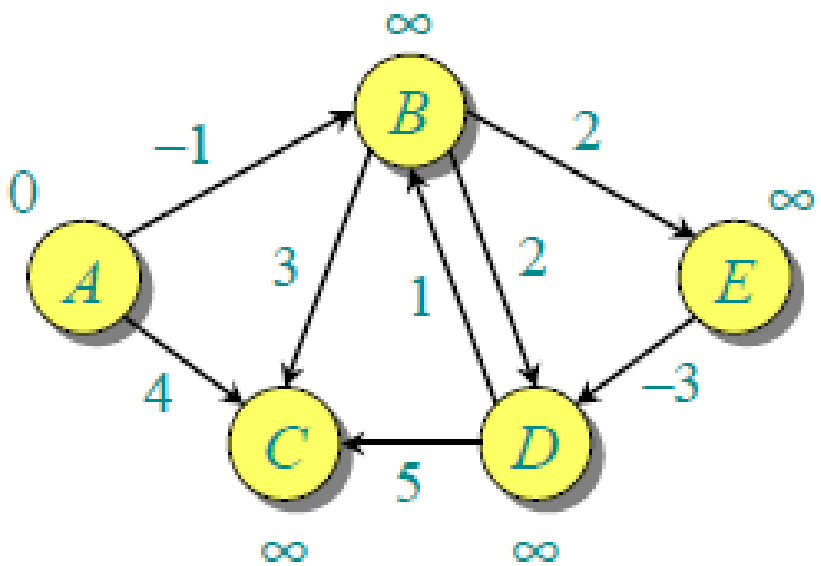
Find all shortest-path lengths from a source to all vertices or determines that a negative-weight cycle exists.

Algorithm

BELLMAN-FORD's algorithm follows these steps:

- *Initialize:* $d(start) = 0$, *otherwise* $d(i) = \infty$
- *For* i *from 1 to* $|V| - 1$ *(the diameter of a graph is at most* $|V| - 1$*)*
  - $d[b] = \min(d[b], d[a] + weight(a, b))$ *for each edge* $(u, v) \in A$
- *For each edge* $(u, v) \in A$*, if* $d[b] > d[a] + weight(a, b)$ *then report a negative-weight cycle.*

Example



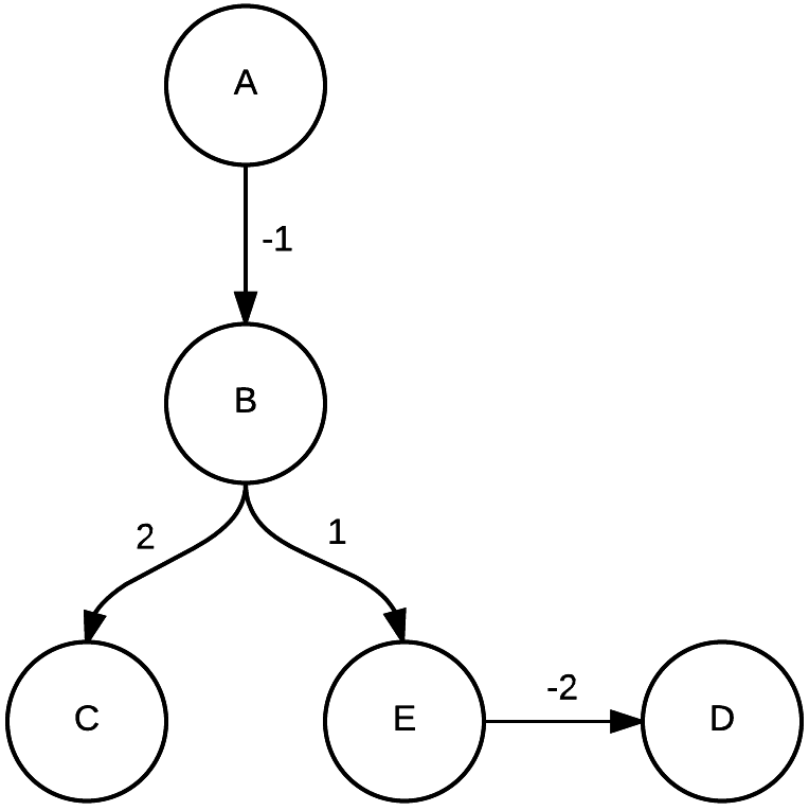
A	B	C	D	E
0	∞	∞	∞	∞

Example

Iteration	A	B	C	D	E
0	0	∞	∞	∞	∞
1	0	**-1(A)**	**4(A)**	∞	∞
2	0	-1(A)	**2(B)**	**1(B)**	**1(B)**
3	0	-1(A)	2(B)	**-2(E)**	1(B)
4	0	-1(A)	2(B)	-2(E)	1(B)

-1(A) means that the distance from A is -1, and its predecessor is A. Only vertices in bold have effect the next iteration (for each vertices, compute the minimum value of these shortest paths plus edge to this vertex). *If two iteration found the same result, we can stop the algorithm.*

Example



Conditions

Bellman-Ford's algorithm is a greedy dynamic programming algorithm that computes shortest paths of increasing size. It is suitable to any graph.

Transitive closure

WE WANT TO DETERMINE THE SHORTEST PATHS BETWEEN ALL PAIRS OF VERTICES.

We could use Dijkstra of Bellman-Ford, with each vertex in turn as the source. Can we do better ?

In the FLOYD-WARSHALL algorithm, we assume we are access to a graph with n vertices as a n^2 adjacency matrix W. The weights of the edges are represented as follows:

$$W_{ij} = \begin{cases} 0 & \text{if } i = k \\ w_{ij} & \text{if such edge exists.} \\ \infty & \text{otherwise} \end{cases}$$

Transitive closure

Optimal substructure

For a path $p = p_1, \ldots, p_k$, define the intermediate vertices of p to be the vertices $p_2, \ldots, p_{k-1}$. Let $d_{ij}^{(k)}$ be the weight of a shortest path from i to j such that the intermediate vertices are all in the set $\{1, \ldots, k\}$. If there is no shortest path from i to j of tis form, then $d_{ij}^{(k)} = \infty$; in the case $k = 0$, $d_{ij}^{(0)} = W_{ij}$; on the other hand, for $k = n$, we have to determine a dynamic-proramming recurrence.

Transitive closure

Let p be a shortest path from i to j with all intermediate vertices in the set $\{1, \ldots, k\}$.

Observation

If k is not an intermediate vertex of p, then p is also a shortest path with all intermediate vertices in the set $\{1, \ldots, k-1\}$. If k is an intermediate vertex of p, then we decompose p into a path p_1 between i and k, and a path p_2 between k and j; they are shortest paths.

Recurrence

We therefore have the following recurrence for

$$d_{ij}^{(k)} = \begin{cases} W_{ij} & \text{if } k = 0 \\ min\left\{ d_{ij}^{(k-1)},\ d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \right\} & \text{if } k \geq 1 \end{cases}.$$

Algorithm

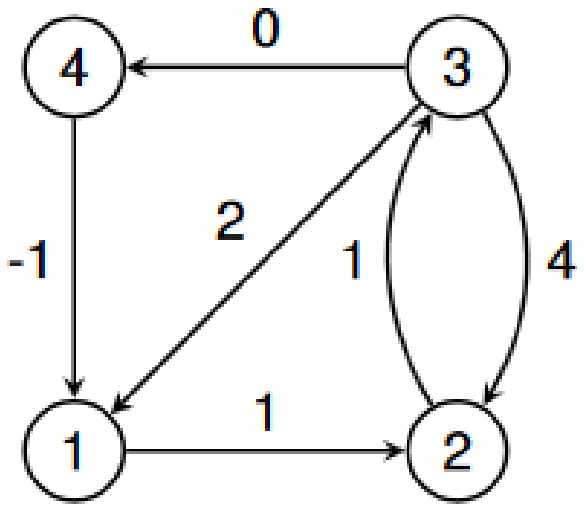
Based on the recurrence, we can give the following bottom-up algorithm for computing $d_{ij}^{(n)}$ for all pairs i, j.

- *Initialize:* $d(start) = w_{ij}$, *otherwise* $d(i) = \infty$
- *For k from 1 to n*
  - *for i from 1 to n*
    - *for j from 1 to n*

$$d_{ij}^{(k)} \leftarrow min\left\{d_{ij}^{(k-1)},\ d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right\}$$

- *Ends when transitive closure is done.*

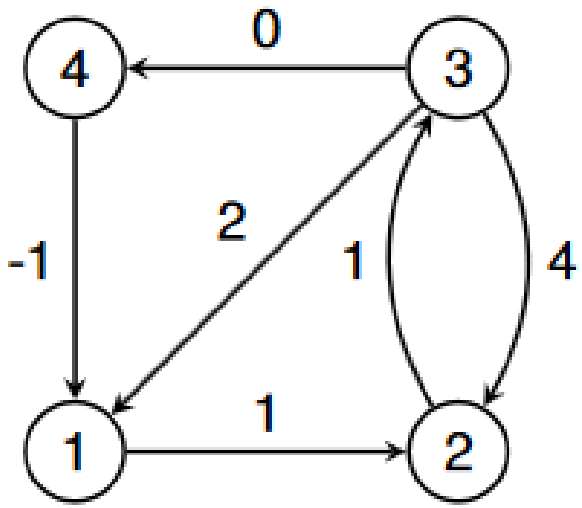
Example



$$\begin{pmatrix} 0 & 1 & \infty & \infty \\ \infty & 0 & 1 & \infty \\ 2 & 4 & 0 & 0 \\ -1 & \infty & \infty & 0 \end{pmatrix}$$

$$d^{(1)} = \begin{pmatrix} 0 & 1 & \infty & \infty \\ \infty & 0 & 1 & \infty \\ 2 & 3 & 0 & 0 \\ -1 & 0 & \infty & 0 \end{pmatrix}, \quad d^{(2)} = \begin{pmatrix} 0 & 1 & 2 & \infty \\ \infty & 0 & 1 & \infty \\ 2 & 3 & 0 & 0 \\ -1 & 0 & 1 & 0 \end{pmatrix}$$

Example



$$\begin{pmatrix} 0 & 1 & \infty & \infty \\ \infty & 0 & 1 & \infty \\ 2 & 4 & 0 & 0 \\ -1 & \infty & \infty & 0 \end{pmatrix}$$

$$d^{(3)} = \begin{pmatrix} 0 & 1 & 2 & 2 \\ 3 & 0 & 1 & 1 \\ 2 & 3 & 0 & 0 \\ -1 & 0 & 1 & 0 \end{pmatrix}, \quad d^{(4)} = \begin{pmatrix} 0 & 1 & 2 & 2 \\ 0 & 0 & 1 & 1 \\ -1 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 \end{pmatrix}.$$

Predecessor matrix

We know the values of shortest paths, how to determine the tree?

Predecessor matrix

We define a sequence of matrices $\Pi^{(0)}, \ldots, \Pi^{(n)}$ such that $\Pi_{ij}^{(k)}$ is the predecessor of j in a shortest path from i to j only using vertices in the set $\{1, \ldots, k\}$. Then, for $k = 0$,

$$\Pi_{ij}^{(0)} = \begin{cases} null & \text{if } i = j \text{ or } W_{ij} = \infty \\ i & \text{if } i \neq j \text{ and } W_{ij} \neq \infty \end{cases}.$$ For $k \geq 1$, we have essentially the same recurrence as for $d^{(k)}$. Formally,

$$\Pi_{ij}^{(k)} = \begin{cases} \Pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \Pi_{kj}^{(k-1)} & \text{otherwise} \end{cases}.$$

Algorithm

- *Initialize:* $d(start) = w_{ij}$, *otherwise* $d(i) = \infty$
- *For k from 1 to n*
  - *for i from 1 to n*
    - *for j from 1 to n*

      ◦ *if* $d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$
      $d_{ij}^{(k)} \leftarrow d_{ij}^{(k-1)}$ *and* $\Pi_{ij}^{(k)} \leftarrow \Pi_{ij}^{(k-1)}$
      ◦ *else*
      $d_{ij}^{(k)} \leftarrow d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$ *and* $\Pi_{ij}^{(k)} \leftarrow \Pi_{kj}^{(k-1)}$
- *Ends when transitive closure is done.*

Example

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

Example

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

Fundamental knowledge

YOU HAVE TO KNOW BEFORE THE TUTORIAL:

1. *Shortest path problem*
2. *Dynamic programming approach*
3. *Dijkstra's algorithm*
4. *Ford-Bellman's algorithm.*
5. ***Know all the recurrence functions!***