OS202 – TD2: Mandelbrot, matrix–matrix product, and parallelism

Santiago Florido Gomez

***Abstract*—This document summarizes the TD1 work on weighted graphs: computing shortest paths by minimum distances (Dijkstra, Bellman–Ford, Roy–Warshall) and building minimum spanning trees with Kruskal. The algorithms are implemented and applied to the provided graphs.**

I. Introduction

This report focuses on the TD1 exercises "Spanning trees and shortest paths". We model the problems as weighted graphs and implement algorithms to identify shortest paths by computing minimum distances and reconstructing paths between nodes. The study covers Dijkstra (single-source with non-negative weights), Bellman–Ford (graphs that may include negative weights), Roy–Warshall (all-pairs shortest paths), and Kruskal (minimum spanning trees for cost-efficient connectivity). The implementations are validated on the provided graphs (bank branch network and flight connections), and we briefly discuss the assumptions and complexity of each algorithm.

II. Trees

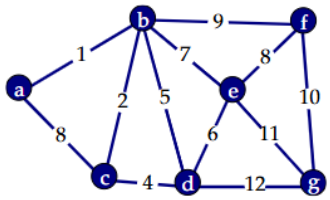

Fig. 1. Bank network with 7 branches and construction costs for each link (Exercise 1).

Kruskal is used for finding the minimum spanning tree of the structure presented in Figure 1.

Edge	ab	bc	cd	bd	de	ed	ef	ca	bf	fg	ge	gd
Weight	1	2	4	5	6	7	8	8	9	10	11	12
k	1	2	3	X	4	X	5	X	X	6	X	X

The main problem with the proposed structure is that, when finding the minimum spanning tree using Kruskal's algorithm, redundancy in the connections between different branches is eliminated, due to the absence of cycles that characterizes the tree resulting from Kruskal's application. For this reason, if a branch or an edge between branches fails, the bank's system would stop functioning (or become partially disconnected) from that point onward in the proposed spanning tree.

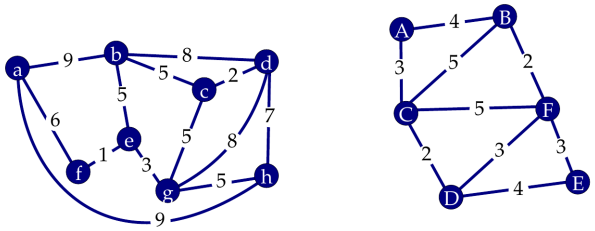

Fig. 2. Graphs for finding minimal spanning trees with Kruskal.

The following edge sets correspond to the results obtained by applying Kruskal's algorithm to build the *minimum spanning tree* (MST) of the graphs shown in Figure 2. In each case, Kruskal's method sorts all edges by non-decreasing weight and then iteratively adds the cheapest edge that connects two different connected components, avoiding the creation of cycles. The resulting MSTs are:

- **Image 1 (MST with Kruskal):**

$$\{(a, f, 6.0),\ (b, e, 5.0),\ (c, d, 2.0),\ (c, g, 5.0),\ (e, f, 1.0),\ (e, g, 3.0),\ (g, h, 5.0)\}$$

- **Image 2 (MST with Kruskal):**

$$\{(A, C, 3.0),\ (B, F, 2.0),\ (C, D, 2.0),\ (D, F, 3.0),\ (E, F, 3.0)\}$$

The following edge sets correspond to the results obtained by applying Kruskal's algorithm to compute the *maximum spanning tree* (MAXT) of the graphs shown in Images 1 and 2. In this maximization variant, all edges are sorted by *non-increasing* (decreasing) weight and the algorithm iteratively adds the heaviest edge that connects two different connected components, while still avoiding the creation of cycles. The resulting maximum spanning trees are:

- **Image 1 (MAXT with Kruskal):**

$$\{(a, b, 9.0),\ (a, f, 6.0),\ (a, h, 9.0),\ (b, d, 8.0),\ (b, e, 5.0),\ (c, g, 5.0),\ (d, g, 8.0)\}$$

- **Image 2 (MAXT with Kruskal):**

$$\{(A, B, 4.0),\ (B, C, 5.0),\ (C, F, 5.0),\ (D, E, 4.0),\ (E, F, 3.0)\}$$

III. Paths

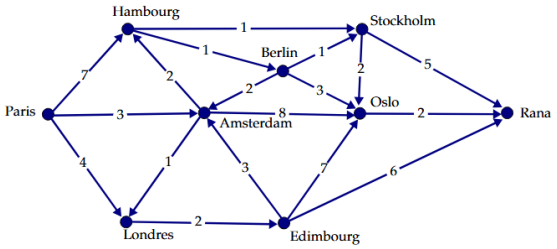

Fig. 3. Travel time (in hours) between European cities.

Using Dijkstra for obtaining the shortest path from Paris to Rana in the graph of Figure 3 we obtain the following.

i	pivot	H	A	L	B	O	E	S	R
–	–	∞	∞	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
1	P	7	3	4					
2	A	5		4		11			
3	L						6		
4	H				6			6	
5	E								12
6	D					9			
7	S					8			11
8	O								10

Pred	H	A	L	B	O	E	S	R
	A	P	A	H	S	L	H	O

$$P \to A \to H \to S \to O \to R$$

The following results show the predecessor relationships obtained by applying Dijkstra's algorithm from the source node r to all nodes in Figure 4.

TABLE I
Left graph: predecessor tree from node r.

Edges in the shortest-path tree
r – a (5.0)
r – b (4.0)
b – c (3.0)
b – g (9.0)
c – d (2.0)
c – f (6.0)
d – e (2.0)

TABLE II
Right graph: predecessor tree from node r.

Edges in the shortest-path tree
r – A (2.0)
r – G (3.0)
A – B (3.0)
A – F (1.0)
B – C (2.0)
F – D (4.0)
G – E (2.0)

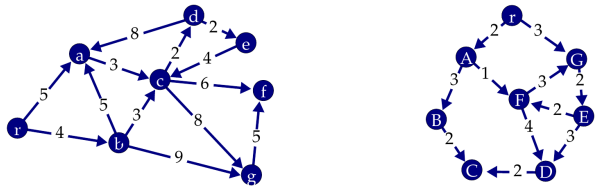

Fig. 4. Directed graphs used for Dijkstra shortest-path computation.

A. Weighted precedence graph using the Activity-on-Arc transformation

We represent the scheduling instance as a directed weighted graph obtained from the precedence graph by the *vertex-splitting* (Activity-on-Arc) construction. Let the original tasks be $D, a, b, c, d, e, f, g, h, i, F$, where D and F have zero duration. For each task vertex $x \in \{a, \ldots, i\}$ we create two vertices x_1 and x_2, and we replace the task by the directed arc (x_1, x_2) weighted by the duration of task x. All precedence arcs from the original graph are preserved and assigned weight 0: if $x \to y$ in the original precedence graph, then we add $x_2 \to y_1$ with weight 0. The start node D is connected to the start vertices of tasks with no predecessors, and the end node F is reached from the completion vertex of the last task.

a) Task arcs (durations).:

$$a_1 \to a_2(4), \quad b_1 \to b_2(6), \quad c_1 \to c_2(4), \quad d_1 \to d_2(12),$$
$$e_1 \to e_2(24), \quad f_1 \to f_2(10), \quad g_1 \to g_2(7), \quad h_1 \to h_2(10), \quad i_1 \to i_2(3$$

b) Precedence arcs (weight 0).:

$$D \to a_1, \quad D \to c_1, \quad D \to d_1,$$
$$c_2 \to b_1, \quad c_2 \to g_1,$$
$$a_2 \to e_1, \quad b_2 \to e_1, \quad b_2 \to f_1,$$
$$d_2 \to h_1, \quad f_2 \to h_1, \quad g_2 \to h_1,$$
$$e_2 \to i_1, \quad h_2 \to i_1,$$
$$i_2 \to F.$$

All these arcs have weight 0.

c) Why the graph is acyclic.: Every precedence arc encodes a strict *finish-before-start* constraint. A directed cycle would force some task to be completed before it can start, which is impossible. Therefore, the resulting directed graph is acyclic.

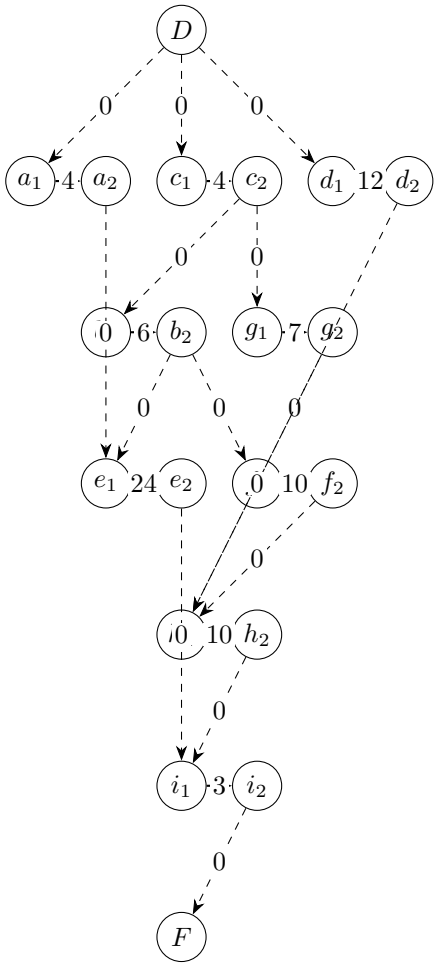

Fig. 5. Activity-on-Arc weighted graph: solid arcs represent tasks (duration as weight), dashed arcs represent precedence constraints (weight 0).

Since the constructed graph does not represent an optional route in which the change is located, but instead necessarily implies going through all stages, the minimum-cost calculation to go from **D** to **F** does not yield the expected result. However, because a task cannot start until its predecessors have finished, for tasks preceded by multiple nodes the earliest possible start time is necessarily equal to the completion time of the predecessor with the largest waiting time; that is, the **maximum** over its predecessors. Therefore, to determine the **minimum** duration of the entire project, it is convenient to identify the **maximum-duration path** between **D** and **F**, since this maximum is equivalent to that minimum required time.

d) Warshall computation (max-plus version).: To obtain this longest path, we apply the Warshall/Floyd–Warshall algorithm by replacing the min operator with max (and initializing missing arcs to $-\infty$):

$$\text{dist}[i][j] \leftarrow \max\big(\text{dist}[i][j],\ \text{dist}[i][k] + \text{dist}[k][j]\big).$$

The resulting `Distances` matrix directly gives the minimum project duration:

$$\text{dist}(D, F) = 37.$$

Thus, the minimum project duration is **37** (in the weight units of the graph).

e) Critical path and critical tasks.: The `Predecessors` matrix lets us reconstruct the path achieving this value:

$$D \to c_1 \to c_2 \to b_1 \to b_2 \to e_1 \to e_2 \to i_1 \to i_2 \to F.$$

Along this path, the *task* arcs (those of the form $x_1 \to x_2$) have the following durations:

$$c_1 \to c_2(4), \qquad b_1 \to b_2(6),$$
$$e_1 \to e_2(24), \qquad i_1 \to i_2(3),$$

so

$$4 + 6 + 24 + 3 = 37.$$

The **critical tasks** are therefore $\{c, b, e, i\}$, that is, the task arcs $c_1 \to c_2$, $b_1 \to b_2$, $e_1 \to e_2$, and $i_1 \to i_2$ (any increase in their duration delays F).

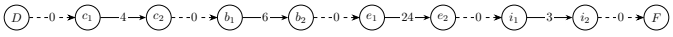

Fig. 6. Critical path (task arcs in bold) and precedence arcs (dashed, weight 0). Total duration: 37.

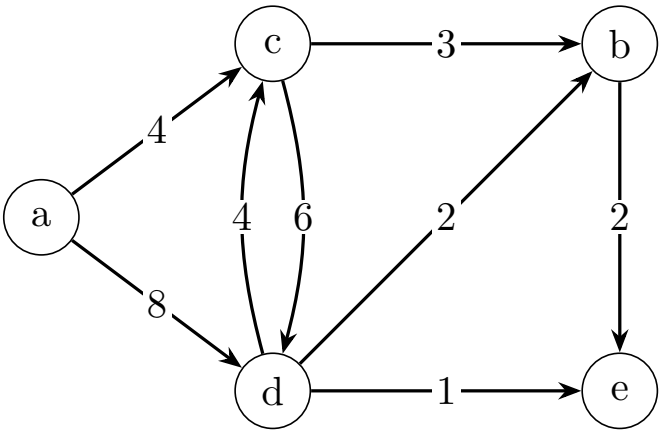

Fig. 7. Directed graph (Exercise 6).

Since there is a cyclic section in the graph, the use of the Roy–Warshall (Floyd–Warshall) algorithm is proposed. The results are presented in the following table.

Distances

	a	c	d	b	e
a	0	4	8	7	9
c	∞	0	6	3	5
d	∞	4	0	2	1
b	∞	∞	∞	0	2
e	∞	∞	∞	∞	0

Predecessors

	a	c	d	b	e
a	$-$	a	a	c	d
c	$-$	$-$	c	c	b
d	$-$	d	$-$	d	d
b	$-$	$-$	$-$	$-$	b
e	$-$	$-$	$-$	$-$	$-$

How to obtain the shortest paths from these tables

- The **Distances** matrix stores, at entry (i, j), the minimum distance (cost) from i to j.
- The **Predecessors** matrix stores, at entry (i, j), the *immediate predecessor* of j on a shortest path from i to j.
- To reconstruct the shortest path from i to j:
  1) If Distances$[i, j] = \infty$, then there is **no path** from i to j.
  2) If $i = j$, the path is simply (i).
  3) Otherwise, set $v \leftarrow j$ and build the path backwards by repeatedly applying
  $$v \leftarrow \text{Predecessors}[i, v]$$
  until reaching i.
  4) The shortest path is the resulting sequence, but read in **reverse order** (because it was built from j back to i).

Examples using your tables

- Shortest path from a to e: Predecessors$[a, e] = d$, then Predecessors$[a, d] = a$. Hence the path is $a \to d \to e$ and its cost is Distances$[a, e] = 9$.
- Shortest path from a to b: Predecessors$[a, b] = c$, and Predecessors$[a, c] = a$. Path: $a \to c \to b$, cost Distances$[a, b] = 7$.
- Shortest path from c to e: Predecessors$[c, e] = b$, and Predecessors$[c, b] = c$. Path: $c \to b \to e$, cost Distances$[c, e] = 5$.