Università
Ca'Foscari
Venezia

Report of advanced algorithms and programming methods project

Luca Bizzotto

875814

Anno accademico 2021/2022

Indice

0.1 Introduction

This project needs to be contextualized in an algorithmic environment, the problem addressed is to implement a matching algorithm in particular an auction algorithm that works on any undirected weighted graph. One of the main goals was to interact with the Boost Graph Library, consequently in this paper we will focus on the following subjects:

- an introduction to the Boost Graph Library
- a little of graph theory
- and finally how we addressed the problem it self.

0.2 Boost Graph Library

The Boost Graph Library is a, really wide, generic library which aim is to give a generic interface that allows access to a graph's structure, but hides the details of the implementation. Graphs are mathematical abstractions that are useful for solving many types of problems in computer science; for this reason it is really important to have an abstraction that makes us able to work in ease with this mathematical tools. As we said above the Boost Graph Libraries stores a wide variety of methods, data structures and features however in this report we will only present the ones we used to address the problem of our interest. Notice also that we will be more schematic as possible in order to make the comprehension of the C++ code connected to this project easier. In the following list we will present three main fundamental templates and concepts of the BGL and then the methods we used the most.

- **adjacency_list:** This class implements a generalized adjacency list graph structure. The template parameters provide many configuration options so that we can pick a version of the class that best meets our needs. An adjacency-list is basically a two-dimensional structure, where each element of the first dimension represents a vertex, and each of the vertices contains a one-dimensional structure that is its edge list. The template parameters are the following:
  - **OutEdgeList:** selector for the container used to represent the edge-list for each of the vertices; it defaults to vecs
  - **VertexList:** selector for the container used to represent the vertex-list of the graph; it defaults to vecS
  - **Directed:** selector to choose whether the graph is directed, undirected, or directed with bidirectional edge access (access to both out-edges and in-edges). The options are directedS, undirectedS, and bidirectionalS; it defaults to directedS.

  - **VertexProperties:** for specifying internal property storage; it defaults to no_property.

- **Property Maps**The main link between the abstract mathematical nature of graphs and the concrete problems they are used to solve is the properties that are attached to the vertices and edges of a graph, things like distance, capacity, weight, color, etc. There are many ways to attach properties to graph in terms of data-structure implementation, but graph algorithms should not have to deal with the implementation details of the properties. The property map interface defined in Section Property Map Concepts provides a generic method for accessing properties from graphs. This is the interface used in the BGL algorithms to access properties

- **Graph Concepts:** the heart of the Boost Graph Library (BGL) is the interface, or concepts, that define how a graph can be examined and manipulated in a data-structure neutral fashion. In fact, the BGL interface does not even need to be implemented using a data-structure since for some problems it is easier or more efficient to define a graph implicitly based on some functions. The BGL interface does not appear as a single graph concept, instead it is factored into much smaller pieces. The reason for this is that the purpose of a concept is to summarize the requirements for particular algorithms. Usually, a particular algorithm does not need every kind of graph operation but only a small subset. Furthermore, there are many graph data-structures that can not provide efficient implementations of all the operations, but provide highly efficient implementations of the operations necessary for a particular algorithm. By factoring the graph interface into many smaller concepts the graph algorithm writer is provided with a good selection from which to choose the concept that is the closest match for its algorithm.

  - **boost::graph_traits:**These represent the graphs associated types, a

graph has quite a few associated types such as vertex_descriptor, edge_descriptor, edge_iterator, etc. Notice that for every graph concept it is not required to define all the associated types.

Methods:

- **edges(g):** returns an iterator-range (a pair of edge_iterator providing access to all the edges in the graph g.
- **out_edges()** function takes two arguments: the first argument is the vertex and the second is the graph object. The function returns a pair of iterators which provide access to all of the out-edges of a vertex.
- **num_edges(g), num_vertices(g)** they return respectively the number of edges and of vertices present in a graph g.
- **vertex_index_map :** A mapping from vertices of the input graph to indexes in the range [0 : : : num_vertices(g)). If this parameter is not provided, the vertex index map is assumed to be available as an interior property of the graph, accessible by calling get(vertex_index, g).
- **get(property, g)**Function to get a property map.

To download the library and use its method: download Boost graph libary

0.3 Graph Theory and Bipartite Graph

We revise briefly the basics of graph theory. Graphs are mathematical structures used to model pairwise relations between objects, a graph is usually said to be made up of vertices (also called nodes) which are connected by arcs (or edges). Usually we make a distinction between directed graph, where all the edges are directed from one vertex to another and undirected graph, where all the edges are considered to be bidirectional. Among the other distinctions we can do on graphs we are particularly interested in the concept of weighted graph: a weighted graph is one in which edges have weights i.e. each edge has a numerical label.

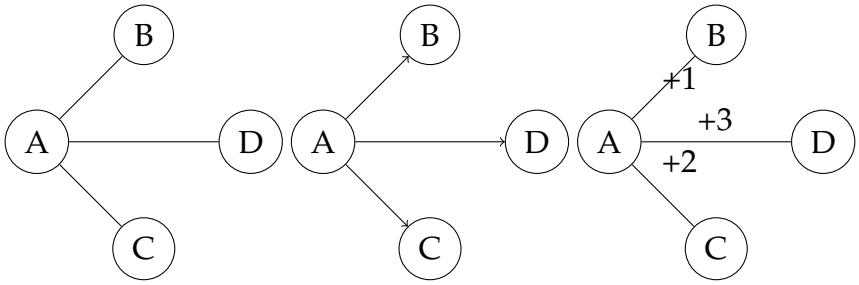

FIGURA 1: Undirected Graph, Directed Graph, Weighted Graph

A bipartite graph, also called a bigraph, is a set of graph vertices decomposed into two disjoint sets such that no two graph vertices within the same set are adjacent.

Let G an undirected bipartite graph G(V_1, V_2, E).

The disjoint vertex sets $V_1\{1....i...n_1\}$, $V_2\{1....j...n_2\}$

The edge set E $\subseteq$ V1 $\times V2$, the weight function $w : E \rightarrow \mathbb{R}$.

The weight w_{ij} indicates the weight of the edge between vertex i and vertex j.

A subset $M \subseteq E$ in the graph G is calling matching iff $|M| = 1$ or

$(v_1, w_1) \in M \wedge (v_2, w_2) \in M \implies (v_1 \neq v_2) \wedge (w_1 \neq w_2)$

where $v_1, v_2 \in V_1$, $w_1, w_2 \in V_2 \wedge (v_1, w_1) \neq (v_2, w_2)$

Edges and vertices in M are called matched edges and matched vertices, respectively. Edges and vertices not in M are called free or unmatched.

The total weight of the matching is computed by $W = \sum_{ij \in M} w_{ij}$.
Bipartite graphs may be characterized in several different ways:

- A graph is bipartite if and only if it does not contain an odd cycle.
- A graph is bipartite if and only if it is 2-colorable, (i.e. its chromatic number is less than or equal to 2).
- A graph is bipartite if and only if every edge belongs to an odd number of bonds, minimal subsets of edges whose removal increases the number of components of the graph.
- A graph is bipartite if and only if the spectrum of the graph is symmetric.

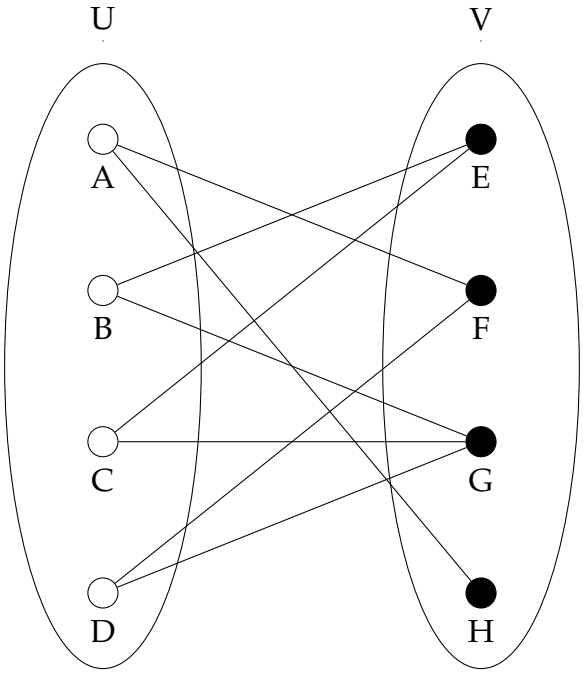

FIGURA 2: Example of Bipartite Graph with its subsets of vertex U, V

0.4 Description of auction algorithm

The problem:

The term "auction algorithm" applies to several variations of a combinatorial optimization algorithm which solves assignment problems, and network optimization problems. Given a weighted bipartite graph G with the disjoint vertex sets V1, V2 and with the edge set E $\subseteq$ V1 $\times V2$, a matching M is a set of edges where no pair of edges share a common vertex.

Maximizing the number of edges in the matching or the weight of the matching are typical goals in bioinformatics, computer vision, Gaussian elimination, and other areas in computational science.

An auction algorithm has been used in a business setting to determine the best prices on a set of products offered to multiple buyers. It is an iterative procedure, so the name "auction algorithm" is related to a sales auction, where multiple bids are compared to determine the best offer, with the final sales going to the highest bidders.

Real-world economic auctions serve as a metaphor for a major class of maximum weighted matching algorithms called auction-based algorithms. The idea of the auction principle can be used to find a feasible assignment of buyers to objects by bargaining for the most valuable objects. The process of finding a matching in a bipartite graph G(V_1, V_2, E) can be modeled as an auction process.

A vertex i $\in$V$_1$ is interpreted as a buyer, vertex j $\in$V$_2$ corresponds to an object, and a weighted edge between the buyers and the objects represents the benefit that a buyer gains by purchasing the object. A price, which is initially set to zero, is assigned to each object which must be paid by the buyer for being its owner. Each buyer gets a turn and proposes a bid for exactly one object which is then allocated to it. The price of the object is then updated to reflect the last bid. Then, the profit for the buyer can be computed by subtracting its current

price from the benefit of the object. The buyer, who is starting with the auction procedure, submits a bid for the object with the highest benefit as the prices are initialized to zero, and the object with the highest profit is equal to the one with the highest benefit. If every buyer prefers a different object, the auction process will end here, and it returns a maximum weighted matching. However, in a realistic scenario, buyers do not have such disjoint interests and often compete for the same valuable object. Then, the question arises, What is an appropriate bid for the buyer as the bid should be price-benefit optimal and the current price of the object increases by the bid? In matching market analysis, where economical interactions between disjoint sets of buyers and sellers are modeled as bipartite graphs, it has been shown that buyers should compute a new bid based on the profits obtained by the two most valuable objects. Thus, the price of the most valuable object is updated by the difference between the highest and second-highest profits. The reason for this special value of the price update is that the buyer is indifferent between the best and second-best object in the subsequent iterations: the profit of the best object, the benefit minus the updated price, is equal to the profit of the second-best object. Hence, if other buyers overbid the buyer, it is worthwhile for the buyer to bid for a different object than in the previous iteration. This process will terminate to an assignment at equilibrium where every buyer is satisfied with the obtained object. This type of auction is called an ascending-bid auction, since the prices for the objects are raised after every iteration.

0.5 Approach

Auction algorithms find a maximum weighted matching via the game-theoretic idea of an ascending-bid auction. The mapping of the bipartite graph to the auction process is performed as described earlier: vertices of V1 and V2 represent buyers and objects, respectively, and weighted edges symbolize the benefits. The auction-based algorithm consists of three phases: the initialization phase, the bidding phase, and the assignment phase. Each object j has an associated price p_j, which is initially set to zero. In the bidding phase in the method choose(I), an unassigned buyer *i* is chosen from the set of unassigned buyers I, here in a cyclic order. At the beginning, a buyer with the smallest index in the set is selected, followed by the second-smallest index in the next auction iteration, until the largest index entry has been reached. Then, the procedure is repeated with an unassigned buyer at the smallest index in the updated set. Each buyer bids for the object j_i, where object *j* has maximal profit for buyer *i*. If such an object does not exist,there is no profitable object available and the buyer will remain unmatched. Otherwise, the highest profit u_i for buyer *i* is computed by w_{ij} - p_j , while the second-highest profit v_i is calculated by ignoring the most valuable object j_i . The bid is computed by subtracting the second-highest profit v_i from the highest profit ui, i.e., u_i - v_i. After the unassigned buyer has submitted the bid, the designated object is assigned to the bidder. The new price is calculated by increasing the old price by the corresponding bid. The next iteration is started with an unassigned buyer. This process is repeated until every buyer has been matched to an object. During the algorithm the number of assigned buyers is never decreased, but an assigned buyer may get unassigned again. The pseudocode of the algorithm will be :

$INPUT \leftarrow BipartiteGraph G = (V1, V2, E, w)$

$OUTPUT \leftarrow MatchingM$

$M \leftarrow \emptyset$ Current matching

$I \leftarrow \{i : 1 \leqslant i \leqslant n1\}$ ▷ Set of unassigned buyer

$pj \leftarrow 0 for j = 1..., n2$ ▷ Initialize price for object

while $I \neq \emptyset$ **do** ▷ Auction iteration

  $i \leftarrow choose(I)$ ▷ Determine a free buyer

  $j(i) \leftarrow argmax(j)\{wij - pj\}$ ▷ Find the best object for buyer i

  $u_i \leftarrow wij(i) - pj$ ▷ Store object for most valuable object

  if $u_i > 0$ **then**

    $vi \leftarrow \max_{j \neq j(i)} \{wij - pj\}$ ▷ Store the second best profit

    $pij \leftarrow pij + u(i) - v(i)$ ▷ Update price with the bid

    $M \leftarrow M \cup \{i, j(i)\}$ ▷ Assign buyer to desire object

    $I \leftarrow I \setminus \{i\}$ ▷ Remove buyer i

    $M \leftarrow M \setminus \{k, j(i)\}$ ▷ Free previous owner k if available

    $I \leftarrow I \cup \{k\}$ ▷ Insert the buyer removed if available

  else

    $I \leftarrow I \setminus \{i\}$ ▷ No object found

  end if

end while

0.6 Implementation Choices

In this paragraph we just specify a couple of items used in the final solution, more implementation details are explained in the code comments. In our implementation we had to create a library Header.h file in which we used "type" of graph, represented with the Boost adjacency_list (undirected, with edge property the weight). The reason why we created a Support class, this was done to simplify certain types of operations and to programming using object-oriented approach. The overall idea was to repeatedly find a Bipartite Graph through the function is_bipartite(g).

This function tests a given graph for bipartiteness using a DFS-based coloring approach. The bipartition is recorded in the color map partition_map, which will contain a two-coloring of the graph, i.e. an assignment of black and white to the vertices such that no edge is monochromatic. The output of this function is PartitionMap that assigns to each vertex either a white or a black color, according to the partition.

Once we had run the is_bipartite() we used the PartitionMap to split the vertices V in two subset V_1, V_2 obtaining the two subsets of the bipartiteness.

Then we choose to used a matrix,to store the information regarding the graph where each row i indicate a buyer, each column j indicate an object and a_{ij} indicate the weight of the edge $(i, j) \in E$.

Since we had to keep the reference from the index position in the matrix and its corresponding object/buyer we used two maps (buyer_map, object_map) to stored it.

We created an inner class Info to stored information about the objects, such as:

- **index:** that maintain reference from the index position in the matrix and its corresponding object
- **price_object:** contain the price of the object initially is set to 0

- **check_owner:** it is a flag and its aim it is to know if the object has an owner or not
- **owner:** the owner of the object

We used a support library in order to make the implementation details invisible to the user, in this library we implemented what follows:

- **Support(const my_graph &g)** the constructor
- **initialize_matrix(std::vector<std::vector<float> > matrix)** initialize the matrix
- **print_matrix()** print the matrix
- **auction_algorithm()** checks if the graph is bipartite and then call is_bipartite and auction_algorithm_with_partition
- **auction_algorithm_with_partition(iterator_property_map map)** divide the nodes in buyer and objects using the partition that we found
- **create_matrix()** Create the matrix that store information about the graph, each $a_{ij} \in Matrix$ represent the weight of the edge $i, j \in E$
- **do_auction()** It does the auction iteration
- **find_best_object(graph_traits<my_graph>::vertex_descriptor buyer_i)** find the best and second best object for buyer_i
- **bid(graph_traits<my_graph>::vertex_descriptor buyer_i, std::vector <graph_traits<my_graph>::vertex_descriptor> best_objects)** find the right bid to do
- **populate_best_matching()** populate the vector of matching
- **get_matching()** return the matching output

- **summary_information()** print a summary of the information that we found

0.7 Example of execution

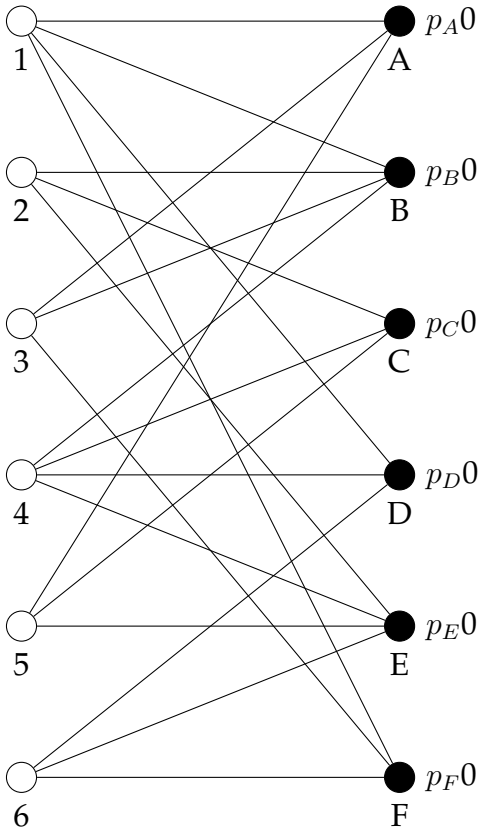

FIGURA 3: The input Graph

$$\begin{array}{cccccc} A & B & C & D & E & F & \\ \left[\begin{array}{cccccc} 9 & 6 & 0 & 3 & 0 & 2 \\ 0 & 2 & 7 & 0 & 1 & 0 \\ 5 & 4 & 0 & 0 & 0 & 3 \\ 0 & 6 & 8 & 3 & 4 & 0 \\ 8 & 0 & 4 & 0 & 1 & 0 \\ 0 & 0 & 0 & 7 & 6 & 5 \end{array}\right] & \begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \end{array}$$

FIGURA 4: Weighted adjacency matrix

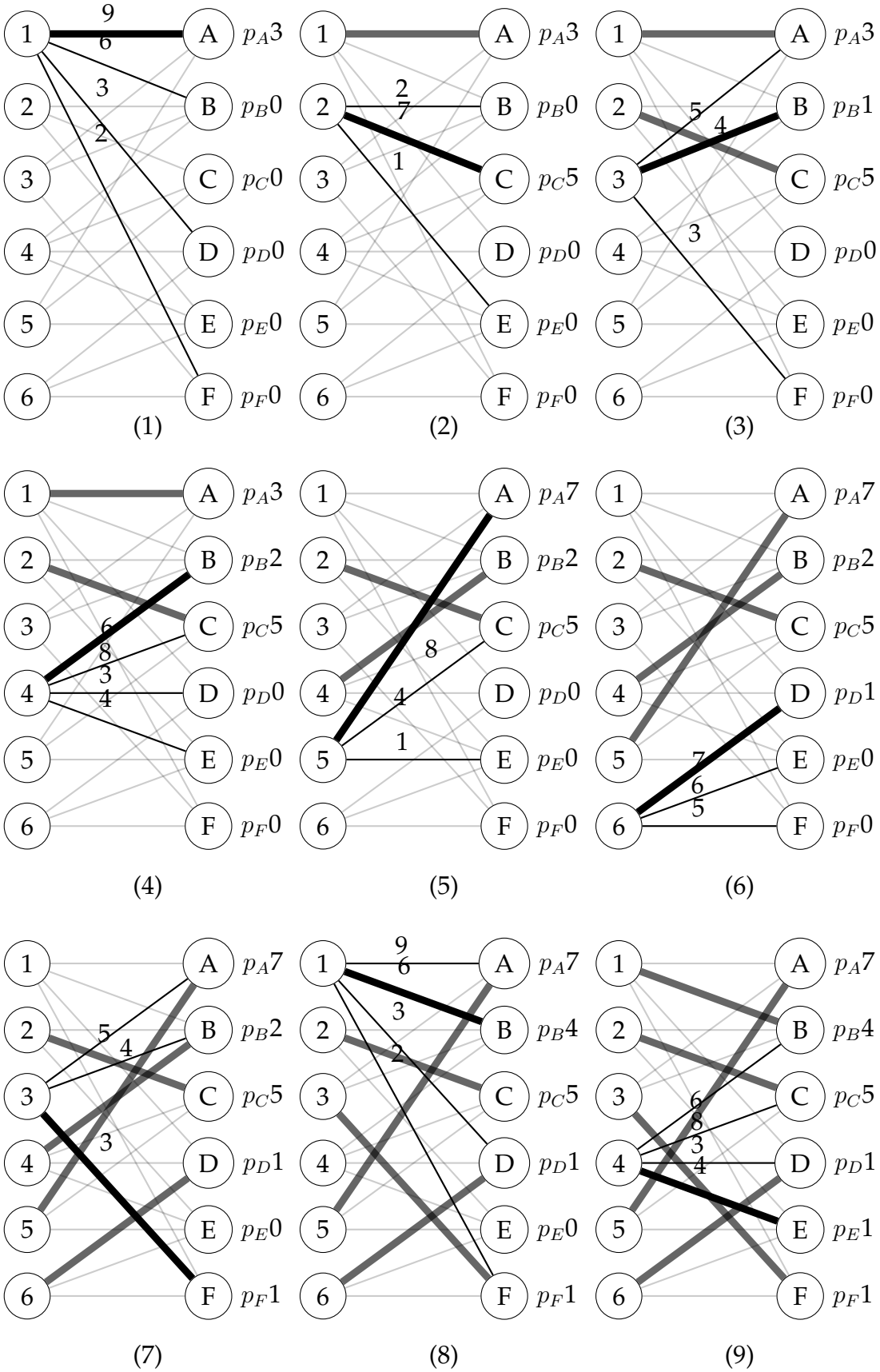

FIGURA 5: Iteration of the algorithm

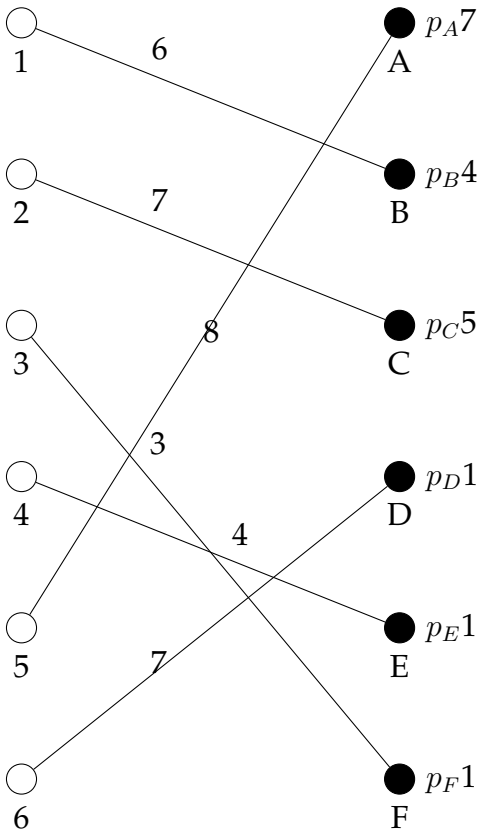

FIGURA 6: The matching output, with W = 35