Algorithmic Matching

Mathematical approaches to optimally allocate topics to students

Frederik Heck

Abstract To match several enteties, like students with topics, graph theory is commonly used. This theory and other helpful concepts are discussed here, specifically adjusted for the student-topic-assignment problem. An explicit JAVA-program is presented and shortly explained as coronation.

Informatik Seminar
Coached by Simon Kramer
BFH
06.01.2020

Contents

List of Figures

1. Introduction

1.1 The student-topic-assignment problem

Imagine there is a group of students participating in a course in which they have to chose one of several pre-defined topics. Because every of these topics may only be chosen by one single student, it is likely that some of the students topic wishes will overlap. Overlapping is the situation when student A and student B both want to be assigned to topic X. Knowing this it is supposable that no assignment from topics to students can be made where all wishes are fulfilled. To avoid random assignments the course leader decides that every student must give an ordered priority list of four wishes, where a higher priority signalizes that the student has a higher interest in getting the topic assigned. With this information it is possible for the course leader to find a student-to-topic-assignment-set where every single assignment is very likely to base on one of the four (or more) wishes of a student. Finding such an assignment-set can be seen as a problem. We will call it the student-topic-assignment problem or in short, the *STA-problem*.

Let us now consider, that every topic is led by an expert, where one expert leads several topics. Which expert leads which topic is fixely defined by the course-leader. Every expert has a minimum pensum to fulfill, meaning he must lead a minimum number of topics. We call a STA-problem which has to respect the just formulated additional condition an *extended STA-problem*. An assignment solving the STA-problem will only solve the extended STA-problem if the minimum pensum of all experts is complied.

How to find a solution for the STA-problem and the extended STA-problem will be discussed in this document. One knowing the mathematical principles of graph-theory will observe that the two problems may be seen as a so-called matching-problem. There are already established algorithms to solve such matching-problems. To understand them it is necessary to know some principles of graph-theory. Therefore, these principles will be discussed in a first part of the document, including an overview of the state-of-the-art algorithmic, that can solve problems out of the family of match-making-theory (section 2.1). A conceptual approach to solve the two STA-problems as well as an explicit algorithm solving the unextended STA-problem in the JAVA-programming-language will be shown and explained in a second part (section 2.2).

2. **Study**

2.1 Matching-Theory: An Overview

2.1.1 Graph-Theory

A **graph** $G = (V, E)$ is a mathematical structure consisting of two sets V and E, where V is a finite set of **vertex** and E is a set of **edge**. Each edge joins a different pair of two distinct vertices v_1 and v_2, written as $e = (v_1, v_2) = (v_2, v_1)$. These joined vertices v_1 and v_2 are called **adjacent**.

Sometimes edges are ordered pairs of vertices, noted as $e^{\rightarrow} = (v_1^{\rightarrow}, v_2)$. Note that $(v_1^{\rightarrow}, v_2) \neq (v_2^{\rightarrow}, v_1)$. We call $e^{\rightarrow}$ a **directed edge**, going from v_1 to v_2, where v_1 is called a **node-source** and v_2 a **node-sink**. If all edges of a graph are directed, we call it a **directed graph**. Likewise, an **undirected** graph has no directed edges at all.

A **path** is a finite sequence of edges in a graph, where each adjacent edge-pair has one vertex in common. In case of directed graphs, the common vertex is the node-sink of the first edge of the pair, while the common vertex is the node-source for the second edge of the pair.

A **bipartite graph** is a graph, consisting of two vertex sets X and Y, where every edge is in the form (v_x, v_y), while $v_x \in X$ and $v_y \in Y$.

Drawing graphs often helps understanding its structure. Vertices are represented by dots or circles, while undirected edges ar represented by lines. Directed edges are represented by an arrow-line pointing from node-sink to node-source.

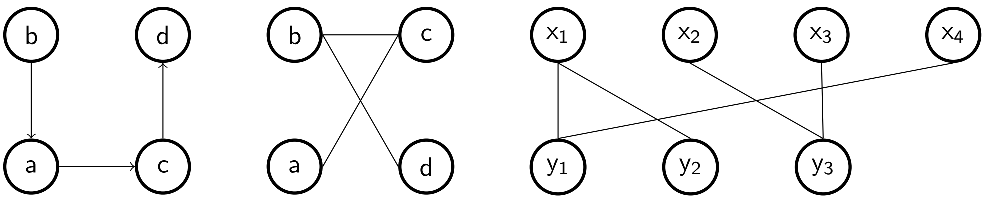

Figure 2.1: Three example graphs. *Left:* directed, *Mid:* undirected, *Right:* bipartite

Remark: Some literature may use the word *node* instead of *vertex*.

2.1.2 Network-Flows

Matching-problems can be reduced by the problem of finding a maximum flow in a network. A **network** N is a directed graph, where each edge $e^{\rightarrow}$ has a flow with a maximum flow-**capacity** $k(e^{\rightarrow})$. The **flow** $f(e^{\rightarrow})$ is a function returning an integer value, where $0 \leq f(e^{\rightarrow}) \leq k(e^{\rightarrow})$. A **flow-unit** is a flow with the value of 1. Increasing or decreasing the flow can be done by adding / removing flow-units.

A network may have a source v_{IN} and a sink v_{OUT}. All flow in the network is **generated** by the source and **swallowed** by the sink. We will call it **super-source** and **super-sink** in this document to distinguish between above defined node-source and node-sink. If the network has several vertices that generate or swallow flow we can simply add an imaginary super-source / super-sink vertex. The super-source will be connected with all vertices v_g that were generating flow-units. We set the capacity of the edges $(v_{IN}^{\rightarrow}, v_g)$ equal the maximum generation-amount f_g of the respective v_g. Now the super-source generates the sum of the flow-units of all f_g. To add an imaginary super-sink, we apply a very similar procedure. An example network is shown below, in figure 2.3.

A network **cut** is a partition of the graph into two disjoint vertex-subsets P and complement $\overline{P}$. We normally speak of a v_{IN} - v_{OUT} cut if $v_{IN} \in P$ and $v_{OUT} \in \overline{P}$. However, to keep it simple, we will just call it *cut* in this document

For better comprehension one may think of the canalisation system of a city as the network, where the pipes are the edges of the graph, houses and pipe-crossroads are the vertices, while the flow is the water-volume floating through a pipe. We intuitively know that a pipe has a minimum and a maximum water-volume that may pass through. The minimum, of course, is no water at all, while the maximum, which here represents our capacity $k(e^{\rightarrow})$, is given by the pipes size. A flow-unit could be one liter or more realistic one water-molecule. Assume the city has a giant water tank, where all water is collected, cleaned and sterilized. All water in the city comes from this tank and will return to this tank after usage. This tank would be our super-source as well as our super-sink.

2.1.3 A Matching Problem

We define a **matching** M as a set of independent edges in a bipartite graph $G = (S, T, E)$, where S and T are the two vertex sets and E is the set of edges, joining vertices of S and T. Every vertex of an **independent** edge is only joined with exactly one other vertex. We can also have a matching in non-bipartite graphs, the definition for matchings in such graphs is similar. An example bipartite matching is shown below, in figure 2.4 and 2.5.

Instead of calling a pair of vertices as *joined* it makes sense to call it *matched*, if it is part of the matching M. Adding a vertex pair to a matching is called *match the vertex pair*. Removing a matched vertex pair from a matching is called *unmatching*. Given this information it makes sense to name E as the set of possible matches, where M is a concrete pick from independent edges of E.

A **S-matching** is a matching where every vertex in S is matched. A matching is a **maximum matching** if no other matching for the graph can be found where the number of matched edges is higher than in the maximum matching.

A **matching problem** is a problem where one aims to find the best possible matching that can be made for the graph G. How *best possible matching* is defined, depends on the use case. Normally the best matching is a maximum matching or a S-Matching (respectively T-Matching).

The here discussed STA-problem can be perfectly treated as a matching problem. The two sets of our bipartite graph are represented by topics and students. Every student is a vertex/node of set S and every topic is a node of set T. To solve the problem we aim to find a matching, with every student assigned to a topic. How we have seen, this is called a S-Matching. In the next subsection 2.1.4 algorithms are shown, that are used to solve such problems in general. In section 2.2 an explicit algorithm for the STA-problem and conceptual approaches for the extended STA-problem will be shown.

2.1.4 Algorithmic

The established way to solve matching problems could be described as three phases:

1. Model the problem as a graph.
2. Transform the graph into a network, where each edge $e^{\rightarrow}$ has a capacity $k(e^{\rightarrow})$ of 1.
3. Find the maximum flow of the network. If an edge has a flow $f(e^{\rightarrow}) = 1$ this is a matched edge. If the edge has as a zero-flow, mathematically written as $k(e^{\rightarrow}) = 0$, it is an unmatched edge.

In the following these three phases are explained in detail. The explanation will be supported by a simplified STA-problem as an example matching-problem, but you could apply the technique to any other problem instead.

Phase 1: Model the graph Think of what you want to match. Normally you will have some sort of two groups. Men and women at pair dances, tourists and guides at city tours or students and topics in our example. In this two-groups-case you want to model a bipartite graph, where each set represent one group. If you have just one big group, where everything may be matched with everything else, you will need a simple graph with only one vertex set. Think of a pair dance where men may also be dancing with other men and likewise women with other women. In the following we assume, that we have a bipartite graph.

However your situation is, you will represent every element that may be matched by a node of a graph. In our example every student is a node of the S-set of a bipartite graph, while every topic is a node of the T-set of the same bipartite graph. Now we simply connect all vertices by edges, which may be matched.

Assume the following:
We have three students s_1 - s_3 and four topics t_1 - t_4.
s_1 wants to be matched with t_1 or t_3,
s_2 wants to be matched with t_2 or t_3,
s_3 wants to be matched with t_4 or t_2.

In this simplified example for a STA-problem we ignore priorities of topic wishes. The resulting graph is $G = (S, T, E)$, where $S = (s_1, s_2, s_3)$, $T = (t_1, t_2, t_3, t_4)$ and $E = ((s_1, t_1), (s_1, t_3), (s_2, t_2), (s_2, t_3), (s_3, t_4), (s_3, t_2))$. If we draw the graph it looks like that:

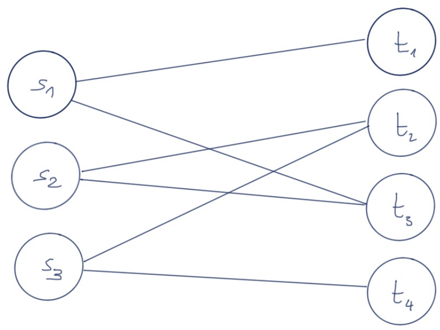

Figure 2.2: An example **graph** for the STA-problem

Phase 2: Transform the graph into a network As a next step we need to transform the graph into a network. That is easily done. Let us make the idea visual at first: In our example we want to make a flow from students to topics. If we match a student to a topic we increase the flow of our network. A good matching is one, in which the flow from students to topics is high.

Being aware of the idea, the practical procedure can be made:

1. Transform all edges of the graph into directed edges. Make sure that all node-sources are in the same vertex-set. Note that this results in having all node-sinks in the other vertex-set of the bipartite graph. In our example we direct all edges such that every student is a node-source and every topic is a node-sink. Every possible match then is a directed edge going form a student to a topic.

   We could also direct the edges from topics to students. It is just a matter of represention. Both works, but since we want to assign students to topics the direction from student to topic is more intuitive.

2. Add a new edge and let it be the super-source v_{IN}. Join it with all student-vertices, having each edge directed from the super-source to the student as the sink-vertex. The super-source generates all flow-units that later will float through the whole network. We say that it may generate an infinite number of flow-units.

3. Likewise add a new edge and let it be the super-sink v_{OUT}. Join it with all topic-vertices, having each edge directed from the topic as the node-source to the super-sink. The super-sink swallows all flow-units that where floating through the network. We define that it may swallow an infinite number of flow-units.

4. Assign every edge a flow-capacity $k(e^{\rightarrow})$ (or in the graphic below just k) of exactly 1.

The text form of the procedure may sound difficult, but if looking at the resulting network, represented visually, the easiness of this process becomes clear:

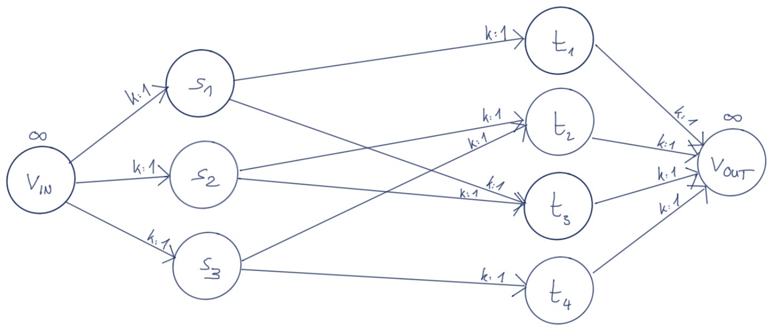

Figure 2.3: An example **network** based on figure 2.2

Note that in the graphic the ∞-symbols signalize that super-source and -sink may generate and swallow an endless flow-unit number. Now that we have our network we only need to define its behaviour: If we match a student s with a topic t, we increase the flow from super-source to s by 1. Then we increase the flow from s to t by 1 as well. Lastly, we increase the flow from t to super-sink by 1.

Since all edges have a capacity of 1, the model makes it impossible to match a student or a topic twice.

Understanding the procedure now, make sure you also understand that the total-flow f_{total} generated by the super-source equals the total-flow swallowed by the super-sink, and most important, f_{total} equals the number of matches from students to topics as well.

Phase 3: Find the maximum flow in the network This phase is the most complicated part. It makes use of an established algorithm to find the maximum-flow in a network. It will match and unmatch (student-topic) pairs and with that increase and decrease the total-flow of the network stepwise, such that the flow will be highest possible. The algorithm is called augmenting flow algorithm and makes use of finding network-chains. A **chain** in a directed graph is a path that ignores the direction of edges. The algorithm tries to find augmenting flow chains. An augmenting flow chain is a chain that contains forwardly directed edges, where the flow $f(e^{\rightarrow})$ is smaller than the respective capacity $k(e^{\rightarrow})$ and backwardly directed edges, that contain a flow $f(e^{\rightarrow})$ that is greater than 0. The idea of that is, that the flow from the backwardly directed edges may be removed and added to the forwardly directed edges. Because the augmenting flow chains the algorithm is looking for do start at vertex v_{IN} and end at v_{OUT}, the change of flow will always improve the matching. This is best shown by example. In the following the explicit algorithm is presented and afterwards its application on our example network. The augmenting flow algorithm, also called blossom algorithm, was first developed by Jack Edmonds [3] and works in polynomial computation time. Note however, that the following algorithm is only a reduction of the established blossom-algorithm and doesn't implement Edmonds blossom concept, since this is only needed for non-bipartite graphs.

Augmenting Flow Algorithm

The algorithm will label each vertex q with two labels: $(b^{\pm}, \triangle(q))$, where b is the previous vertex that is just before q in the flow-chain from v_{IN} to q. $\triangle(q)$ is the number of additional flow, that can be sent from v_{IN} to q.

On forwardly directed edges flow can be added, which is marked as b^+. The number that may be added is the difference between capacity and flow: $k(e^{\rightarrow})$ - $f(e^{\rightarrow})$.

On backwardly directed edges flow can be removed, which is marked as b^-. The number that may be removed is the flow $f(e^{\rightarrow})$.

The algorithm will endlessly try to find augmenting chains, till no more optimizations are possible. This is when a saturated cut is found.

1. Give the super-source v_{IN} the labels (-, ∞)

2. Now scan a vertex. Let us call the vertex being scanned p and its second label $\triangle(p)$. Initially, $p = v_{IN}$.

(a) Check each incoming edge $e = (q^{\rightarrow}, p)$. If $f(e^{\rightarrow}) > 0$ and q is unlabeled, then label q with $(p^{-}, \triangle(q))$, where $\triangle(q) = \min[\triangle(p), f(e^{\rightarrow})]$

(b) Check each outgoing edge $e = (p^{\rightarrow}, q)$. If $s(e^{\rightarrow}) = k(e^{\rightarrow}) - f(e^{\rightarrow}) > 0$ and q is unlabeled, then label q with $(p^{+}, \triangle(q))$, where $\triangle(q) = \min[\triangle(p), s(e^{\rightarrow})]$

3. If v_{OUT} has been labeled, go to step 4. Otherwise choose another labeled vertex to be scanned (which was not yet scanned) and go to Step 2. If there are no more labled vertices to scan, let P be the set of labled vertices, and now $(P, \overline{P})$ form a saturated cut. The flow from P to $\overline{P}$ will then be maximum. From this we can conclude that we have found the maximum flow of the network, since every cut must contain the whole flow of a network. Stop the algorithm.

4. Find a v_{IN} - v_{OUT} chain of vertices, where every $\triangle(q)$ of a vertex has a positive value. The minimal number of $\triangle(q)$ in the chain, tells us by how many flow-units the flow may be increased. We increase it by adding flow of this number on all forwardly-directed edges, and removing flow of this number on all backwardly-directed edges.

5. Start again at 1.

To make all these words and symbols more understandable, an example application:

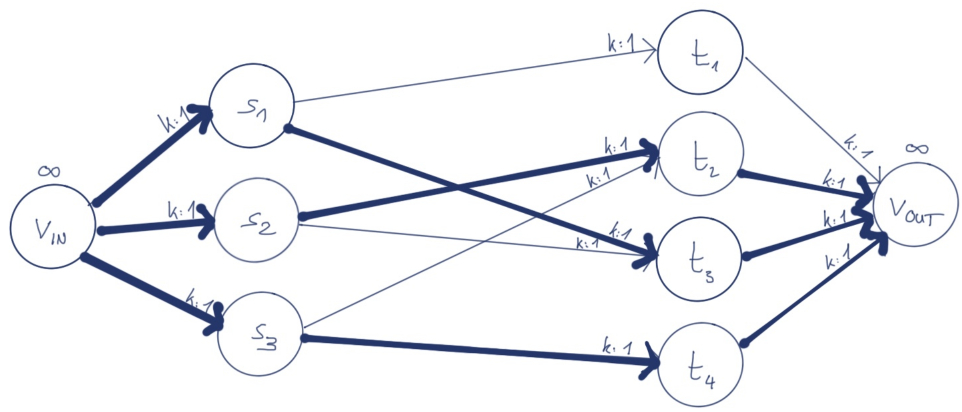

Figure 2.4: An example **maximum matching** based on figure 2.3. Thick edges represent a match.

If we try to apply the algorithm to this network, it will come to a short end. While scanning v_{IN} as first vertices, no other vertex may be labeled in *step 2(a)* since there is per definition no incoming edge in the super-source. In *step 2(b)* we don't label anything as well, since all outgoing edges are saturated, meaning $s(e^{\rightarrow}) = 0$. We then go to *step 3* and because no other vertex is labeled, the algorithm ends, telling us the matching is maximum. If we look at the drawing of the network 2.4 we can see why: Every student has a topic assigned already.

To see a more realistic application, we add a new student s_0, that wants to be matched with t_2:

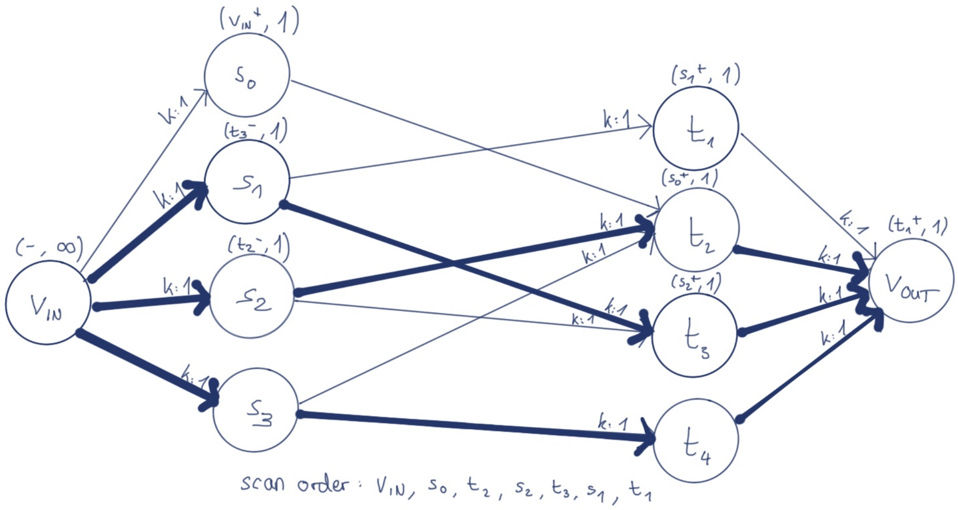

Figure 2.5: An example **non maximum matching** with labels of the augmenting flow algorithm.

Having v_{IN} scanned first again, we can now label an outgoing edge in *step 2(b)*, since the edge to the new student s_0 is unsaturated. $s(e^{\rightarrow}) = 1$, meaning the flow can be increased by one flow-unit. We indicate that with a label. Then, in *step 3*, we jump back to *step 2*, since we have another labled vertex s_0. Similarly, we label t_2, which will be scanned next. From there we have a saturated incoming edge s_2, so we can first label something in *step 2(a)*. We label s_2 and scan it next. The procedure goes on till v_{OUT} is labled. The full scan order is indicated in the figure above. When v_{OUT} is scanned the algorithm jumps to *step 4* and searches for a v_{IN} - v_{OUT} chain. There is only one namely v_{IN} - s_0 - t_2 - s_2 - t_3 - s_1 - t_1 - v_{OUT}. We now remove all flow on backwardly directed edges in this augmenting chain, and add flow to all forwardly directed edges. The flow- reduction / addition is for each edge exactly 1 flow-unit, like the second label $\triangle(q)$ indicates. $\triangle(q)$ is always one, because every capacity is limited to one. We draw the improved graph:

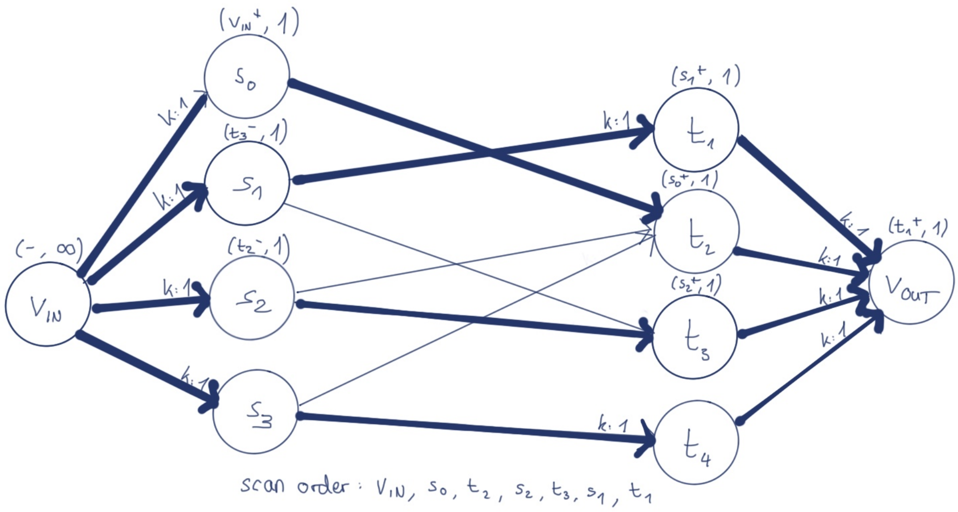

Figure 2.6: Application of augmenting flow chains: matching of figure 2.5 improved

Now, going to *step 5* of the algorithm, we must repeat the whole process. We jump back to *step 1*, but this time with the improved network as input. Again, the algorithm will not be able to lable anything but v_{IN}. The algorithm stops, proving us that we have a maximum matching. We can see that this is true, because every student is matched with a topic.

In more complicated cases however, the algorithm would go through more than 2 iterations, always looking for v_{IN} - v_{OUT} flow-chains.

Time complexity and other approaches An article[4], written two years before this document was made, gives a formalization of Edmunds algorithm, and is proving the time complexity to *O(VE²)*, where *V* means the length of the vertex set of the graph, *E* the length of the edge set.

However, besides the idea of Edmund, there are other approaches to solve matching-problems. Some of them obtain other, or even better time complexity. To cite the essence of the article, to be found in its introduction:

" Dinitz generalized the idea of augmenting paths to blocking flows, obtaining an $O(V^2E)$ algorithm. Karzanov was the first to propose an $O(V^3)$ algorithm based on the idea of preflows. Based on Karzanov's ideas, Goldberg and Tarjan proposed the generic push–relabel algorithm. Implementations of the push–relabel algorithm are among the most efficient maximum flow algorithms. While the generic algorithm has a time complexity of $O(V^2E)$, specific variants of the algorithm achieve even lower complexities down to $O(V^2\sqrt{E})$. "

Since we will use Edmunds approach to solve our problem, we won't go into the concepts of other approaches.

2.1.5 References

The information in this section 2.1 mostly bases on chapter four of Tucker's Applied Combinatorics [1]. The theory presented here is in a condensed form and aims to cover only those aspects, that are necessary to understand if one wants to solve the *STA-problems*. For further detail or proof for given definitions refer to the book. An explicit JAVA-algorithm that implements the procedure described in section 2.1.4 is given in chapter 7.1 in graph library by H.T. Lau [2]. Note that we will need a slightly different algorithm for the STA-problem, how explained in section 2.2.1.

During reserach some wikipedia articles came in very handy as well, since they brought fast and easy definitions and explenations. The most important are listed in the bibliography [6].

2.2 Solving the STA-problem

In this section we will see how to solve the introduced STA-problem. Therefore, we will describe a conceptual solution at first for tightening the understanding of the materia and giving some additional conceptual thoughts. In a second part we will look at an explicit solution written in Java, where the program reads the information from a CSV-file and prints out the optimal matching. Note that it doesn't solve the extended STA-problem, however with the following presented ideas, it should be possible to adapt it, such that it solves the extension.

2.2.1 Conceptual Approach

STA-problem

To solve our STA-problem, we can apply the presented algorithmic from section 2.1.4 in most parts. However, there is one big additional condition we must think of. We need to be aware that students give their topic wishes in a priority-ordered-list to the course leader. We want to respect students high-priority wishes more than their low-priority wishes. Therefore, we need to model this condition.

To model the weights, we create a weight-matrix W of size $|S| * |T|$, where S is the student vertex set of the bipartite graph, T the topic vertex set. For every student s_x choosing a topic t_y with priority p we store the weight $w_{xy} = p$ in the matrix W. Note that in more general, i.e. non bipartite cases, the matrix is of size $|S+T| * |S+T|$. Then we would have a quadratic, symmetrical matrix. In our algorithm below we will use such a quadratic matrix, however, assume in the following that it is in form $|S| * |T|$, making the explanation of the concept easier.

Given by our problem, a student choses four priorities, meaning we must store four weights in the matrix per student. Every other student-topic combination, that does not get a priority / weight assigned, gets the weight set to ∞. As a next step we would need to model an algorithm minimizing the edges weight-sum in the matching M. Having every edge-weight w_∞ going from a student s to a topic t, where s does not have t as part of his 4-length priority list, set to $w_\infty = \infty$, the algorithm will never match such an edge.

The principle of minimizing the sum of all edge-weights in the matching *M* is not trivial and deserves an own article describing it. This was done for one recurrent technique basing on Edmunds algorithm.[5] The by the article provided procedure is solvable in polynomial time as well, thus V^3.

Extended STA-problem

Till now we have not discussed how to include our extended condition in our algorithmic. How can we make sure that the minimum pensum of all experts is surely fulfilled? We have already seen how to maximize flow in networks. In the following two ideas to solve the extension are shown, that make use of network-graphs and maximization. However, at first we will need to create a model, on which both ideas rely on.

The extended Network-Model Remember the example network from section 2.1.4 Phase 2. We will now add an additional layer between topics and super-sink, which we call the expert-layer, since every node represents an expert.

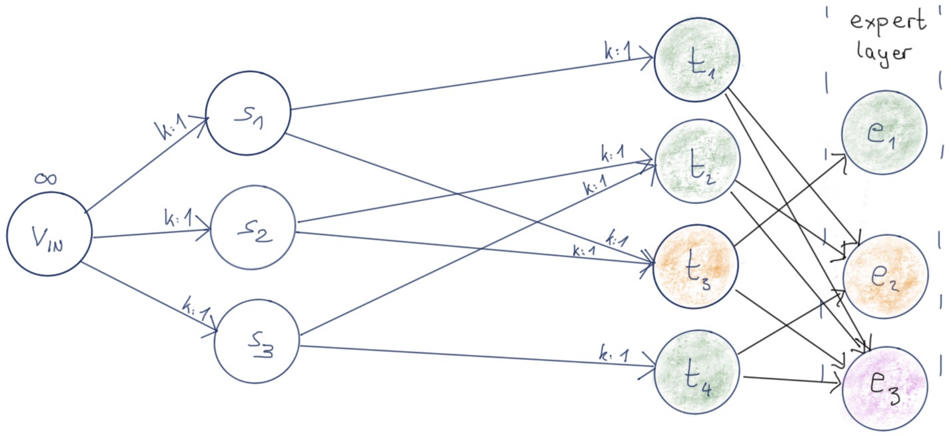

Figure 2.7: First sketch of a model, representing an extended STA-problem (incomplete)

The colour code shows which topic is led by which expert in this example. Expert e_1 leads three topics t_1, t_2, t_4. Expert e_2 leads t_3 and expert e_3 no topic at all.

Intuitively we would connect every topic-vertex with the corresponding expert-vertex, like we connected every student-vertex with all his topic-vertices he wanted to be assigned to. With such a model, we couldn't use a flow maximization algorithm, since the pensum of a professor has a lower-barrier. With network-graphs we can only model upper-barriers, using capacities. Because of that, for topic-expert connections we won't make a normal connection but do an inverse one: We connect every topic with every expert, but not with the expert, that is leading the topic. Every on this way created edge has a capacity $k(e^{\rightarrow})$ of exactly 1.

We now define a flow-behaviour, that is unusual in network-graphs: If a student matches a topic, the outgoing flow of every edge, that leaves the topic, will be increased by one flow-unit. With this, the flow going into an expert-node will increase everytime, when a student matches a topic, that is led by a different expert.

If we have *n* students and an expert *e* has the minimum-pensum number *p*, the model should make sure, that at most *n* - *p* students chose topics, not led by this expert *e*. With this our extended condition, respecting expert-pensums, would be fulfilled. Therefore we give every expert a flow-capacity of $k(e^{\rightarrow})$ = *n* - *p*. Since the expert *e* is not an edge but a vertex, we need to transform it into an edge, since only edges may have capacities. We simply add a helping layer with the same number of nodes like in the expert layer and connect each expert with one different node of the helping layer. Now every expert is represented by two distinct nodes and one edge connecting those two nodes. To this edge $e^{\rightarrow}$ we can assign the capacity $k(e^{\rightarrow})$.

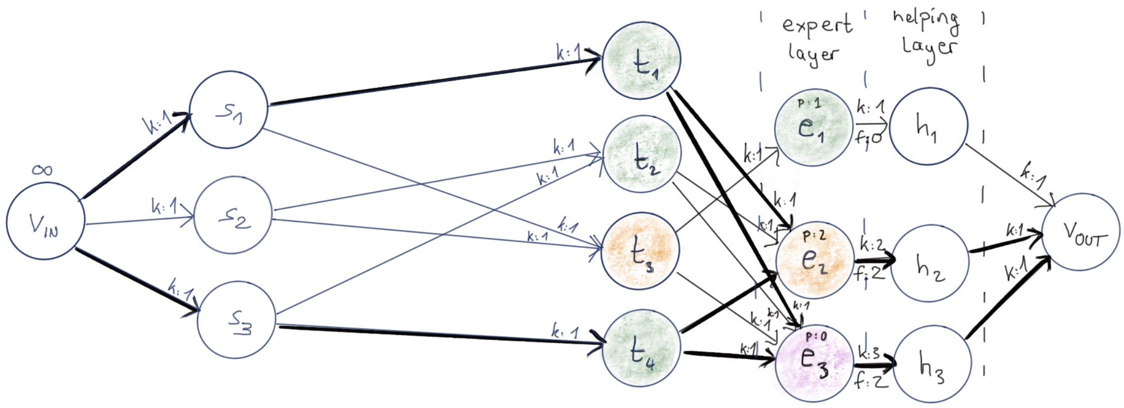

Figure 2.8: Example network model, representing an extended STA-problem (complete)

With having all this done, we set up a perfect model, that will allow only such assignments that respect all pensums of the experts. If you have difficulties to see that this model works, it is recommended to play through a simple example on paper. Note all capacities on the expert edges and watch how they behave depending on different student-topic assignments.

A little example is already drawn in the last figure. Thick edges represent a flow of one or more flow-units, thin edges represent a zero flow. In the expert nodes the pensums are noted as *p*. The number of students is three, so $n = 3$. With this we can calculate all capacities between expert and helping layer, also noted, as *k*. The actual flow, already floating between expert and helping layer is noted as well, as *f*.

Now we want to know which topic s_2 can be assigned to. t_2 and t_3 are still available. We know that we will increase the flow on all outgoing edges of a topic. If we would want to assign the student with t_2 we needed to increase the flow from t_2 to e_2 and from t_2 to e_3. However, e_2 cannot have any more flow-units, since edge $(e_2^{\rightarrow}, h_2)$ is already saturated. We can only assign s_2 to t_3.

As you can see at this example the capacities exactly block connections, that should be blocked, having as a consequence that only matchings are made that are not in conflict with our wished extension.

Idea 1: Dynamical Weight Hacking The now presented idea, is the easier one and makes dynamic changes to the already introduced weight matrix *W* used for storing student-topic priorities. Everytime the algorithm adds / removes a student-topic match to the matching, we make two tests:

1. Is one of the expert edges saturated ($k(e^{\rightarrow})$ - $f(e^{\rightarrow}) = 0$)?

   If yes: Every topic that is connected with the corresponding expert can't be chosen anymore, it is *blocked*. Therefore, we set every weight *w* in *W* to ∞, if *w* stores a weight unequal ∞ of a connection going into a blocked topic. If for example topic *"example"* is connected with a saturated expert, all priorities of students that wished to be assigned to topic *"example"* will be overwritten with ∞. However, make sure, that you can reset the overwritten priorities, how described in the second test.

2. Is one of the expert edges, that once was saturated, not saturated anymore? If yes: Every topic that is connected with the corresponding expert can again be chosen, it is *unblocked*. Reset every weight *w* in *W* to its initial value, if *w* stores a weight of a connection going into an unblocked topic.

If this feature is implemented, the algorithm will never match blocked topics. However, it is first to show, that the dynamical weight-hacking doesn't interference with the matching-algorithm. We don't know if the algorithm works and if yes, how well it performs. The advantage of this procedure is its simplicity, while its behaviour is yet unknown.

Idea 2: Systematic approach While for the first method no proof is given, that it will come to a good result, the now presented algorithm, will come to a perfect solution. However, it is more complicated and in terms of time-complexity and memory not efficient in cases, where the difference *n* - *t* is high. *n* is the number of students and *t* the number of topics. This is because the algorithm is not deterministic.

1. Store the number n of students. Ignore which student wants to be assigned to which topic, and find an arrangement A of topics that contains n topic nodes ($|A| = n$). Be sure that the arrangement respects the pensums of the experts, by using our extended network-model. For that, let topics being part of an arrangement behave like they were matched (even if no real matching to a student was made yet). Now start the following sub-sequence of the algorithm for A:
   (a) We delete every topic-node x, that is not part of A. Therefore, we copy our weight matrix W and call it $W_{censored}$. Now we edit $W_{censored}$. Namely, we delete all rows and columns representing a topic x.
   (b) Now we start the normal STA-algorithm, having $W_{censored}$ as input matrix. We store the sum of the weights of all edges, being part of the resulting matching. We call this number ranking r.
2. We calculate which of the arrangements has the lowest ranking r. This will be the perfect assignment. If two or more assignements all have the same r, and this r-value is the lowest found r-value, every of these assignments can be chosen.

Note that the algorithm is not deterministic and the first step will be started for every possible assignment A. When implementing this idea, the first step would be finding a combinatorial algorithm, listing all possible arrangements A.

How mentioned this algorithm will find a perfect solution, but is more complicated than the first presented. Also, it and can be very inefficient.

2.2.2 Explicit Algorithm in Java

The algorithm, that will be presented here, makes use of three classes, that all may be found and explained generally in some words in the next three subsections. It is a console-based program expecting a CSV-file as input, storing all data, which student wants to be assigned to which topic with which priority. The result of the algorithm is presented in the console in simple form.

Explicit explanations may be found in the header-lines of the source code, given as addition to this document as *.JAVA-files*. Besides that, a documentation-pdf is provided giving detailed information how to use the algorithm and understanding its output.

Main-STA

As the main class its assignment is to control the input of the program (Step 1), steer the calculation (Step 2) and present the output of it (Step 3). For the first step the CSV-Reader class will read in all required data and return it. As the second step a method of an adopted library will be used. The library is all together in one big class, here called Graph-Algo. The output as third and last step is all generated within this main-class.

```
public class mainSTA {

    /* The program expects 3 inputs:

    */
    static int studentCount;
    static int topicCount;
    static String csvFile;

    //static helper arrays for data processing
    static double[][] assignmentMatrix;
    static int[] solutionArray;

    public static void main(String[] assignmentData) {
        try{
            //STEP 1: Read in

            /*
            // use this for debugging
            studentCount = 37;
```

```
            topicCount = 42;
            csvFile = "doc\\example.csv";
            //*/

            //*
            // use this for console-based input
            studentCount = Integer.parseInt(assignmentData[0]);
            topicCount = Integer.parseInt(assignmentData[1]);
            csvFile = assignmentData[2];
            //*/

            makeAssignments(); //STEP 2: Main work
            printAssignments(); //STEP 3: Output

        }catch(Exception e){
            System.out.println("There was an error."
                                    + " Please check your input data.\n");
            System.out.println("The program expects this input:");
            System.out.println(">>>[studentCount] [topicCount] [csvFile]\n");
            System.out.println("For further information"
                                    + " consult the documentation.");
        }
    }

    public static void makeAssignments(){
        //matrix preparation
        int matrixSize = studentCount + topicCount;
        solutionArray = new int[matrixSize + 1];
        assignmentMatrix = CSVReader.readAssignmentMatrix(csvFile,
                                                    studentCount, topicCount);

        // do matching algorithm
        GraphAlgo.minSumMatching(matrixSize, assignmentMatrix,
                                                        solutionArray);
    }

    //present result of algorithm
    public static void printAssignments(){
        System.out.println("\nOptimal matching:");
        int topic, priority;
        for(int student = 1; student <= studentCount; ++student) {
            topic = solutionArray[student] - studentCount;
            priority = (int)assignmentMatrix[student][solutionArray[student]];
            if(priority < 1000)
                System.out.println(" s" + student
                        + " -- t" + topic + " [P" + priority + "]");
            else{
                System.out.println(" s" + student
                            + " --  no matching possible");
            }
        }

        System.out.println("\nTotal optimal matching cost = "
                                            + assignmentMatrix[0][0]);

    }
}
```

CSV-Reader

The CSV-Reader expects a special structure of CSV-file as input. Student-IDs are stored in a row-header, topic-IDs in a column-header. The content of the table is the corresponding priority a student has to a topic.

This classes key-method *readAssignmentMatrix()* will parse the content and fill it into a two-dimensional array. It will then extend the array to a special form needed for the graph-algorithm and return it.

```
public class CSVReader {

    // Pass the csv file-source and an empty array
    public static double[][] readAssignmentMatrix(String csvFile,
                                        int studentCount, int topicCount) {
        BufferedReader br = null;
        String line = "";
        String cvsSplitBy = ";";

        // +1 because first row and column are empty
        int matrixSize = studentCount + topicCount + 1;
        double[][] assignmentMatrix = new double[matrixSize][matrixSize];

        // Initialize all elements high weight, so they won't be matched.
        // Later some elements will be overwritten.
        double highWeight = 1000000.0;
        for (int i = 1; i < assignmentMatrix.length; i++) {
            for (int j = 1; j < assignmentMatrix[i].length; j++)
                assignmentMatrix[i][j] = highWeight;
        }

        //read the csv-file line by line
        try {

            br = new BufferedReader(new FileReader(csvFile));
            int lineNum = 0;
            while ((line = br.readLine()) != null) {

                if (lineNum == 0) { // skip header
                    //lineNum as index below will start with 1,
                    //thats as wished because first row is empty
                    lineNum++;
                    continue;
                }

                if (lineNum > 37) break;

                // store all values of the csv in an array,
                // use semicolon as separator
                String[] row = line.split(cvsSplitBy);

                // fill in a row in assignmentMatrix by taking the content
                // of the row-array. In the csv-file should never be more values
                // per line of interest then topics exist. Note that the first
                // value in the assignmentMatrix has to be empty (=> i = 1)
                for (int i = 1; i <= topicCount; i++) {
                    if(i < row.length && !row[i].equals("")){
                        // only do something if the array has
                        // still values that are not empty

                        // fill in the assignmentMatrix:
                        assignmentMatrix[lineNum][studentCount + i]
                                            = Double.parseDouble(row[i]+".0");
                    }
                }
                lineNum++;
            }
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (IOException e) {
```

```
            e.printStackTrace();
        } finally {
            if (br != null) {
                try {
                    br.close();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }

        return assignmentMatrix;
    }

}
```

Graph-Algo

The following method is part of *A Java Library of Graph Algorithms and Optimization* by H.T.Lau [2] and can be found in chapter 7.2 of the book. It was adopted in the program without making any changes to it. It is the heart of the STA-algorithm, since it *is the algorithm*: It implements the concept discussed above, in section 2.2.1.

```
public static void minSumMatching(int n, double weight[][], int sol[])
{
  int nn,i,j,head,min,max,sub,idxa,idxc;
  int kk1,kk3,kk6,mm1,mm2,mm3,mm4,mm5;
  int index=0,idxb=0,idxd=0,idxe=0,kk2=0,kk4=0,kk5=0;
  int aux1[] = new int[n+(n/2)+1];
  int aux2[] = new int[n+(n/2)+1];
  int aux3[] = new int[n+(n/2)+1];
  int aux4[] = new int[n+1];
  int aux5[] = new int[n+1];
  int aux6[] = new int[n+1];
  int aux7[] = new int[n+1];
  int aux8[] = new int[n+1];
  int aux9[] = new int[n+1];
  double big,eps,cswk,cwk2,cst,cstlow,xcst,xwork,xwk2,xwk3,value;
  double work1[] = new double[n+1];
  double work2[] = new double[n+1];
  double work3[] = new double[n+1];
  double work4[] = new double[n+1];
  double cost[] = new double[n*(n-1)/2 + 1];
  boolean fin,skip;

  // initialization
  eps = 1.0e-5;
  fin = false;
  nn = 0;
  for (j=2; j<=n; j++)
    for (i=1; i<j; i++) {
      nn++;
      cost[nn] = weight[i][j];
    }
  big = 1.;
  for (i=1; i<=n; i++)
    big += cost[i];
  aux1[2] = 0;
  for (i=3; i<=n; i++)
    aux1[i] = aux1[i-1] + i - 2;
  head = n + 2;
  for (i=1; i<=n; i++) {
    aux2[i] = i;
```

```
    aux3[i] = i;
    aux4[i] = 0;
    aux5[i] = i;
    aux6[i] = head;
    aux7[i] = head;
    aux8[i] = head;
    sol[i] = head;
    work1[i] = big;
    work2[i] = 0.;
    work3[i] = 0.;
    work4[i] = big;
  }
  // start procedure
  for (i=1; i<=n; i++)
    if (sol[i] == head) {
      nn = 0;
      cwk2 = big;
      for (j=1; j<=n; j++) {
        min = i;
        max = j;
        if (i != j) {
          if (i > j) {
            max = i;
            min = j;
          }
          sub = aux1[max] + min;
          xcst = cost[sub];
          cswk = cost[sub] - work2[j];
          if (cswk <= cwk2) {
            if (cswk == cwk2) {
              if (nn == 0)
                if (sol[j] == head) nn = j;
              continue;
            }
            cwk2 = cswk;
            nn = 0;
            if (sol[j] == head) nn = j;
          }
        }
      }
      if (nn != 0) {
        work2[i] = cwk2;
        sol[i] = nn;
        sol[nn] = i;
      }
    }
  // initial labeling
  nn = 0;
  for (i=1; i<=n; i++)
    if (sol[i] == head) {
      nn++;
      aux6[i] = 0;
      work4[i] = 0.;
      xwk2 = work2[i];
      for (j=1; j<=n; j++) {
        min = i;
        max = j;
        if (i != j) {
          if (i > j) {
            max = i;
            min = j;
          }
          sub = aux1[max] + min;
```

```
            xcst = cost[sub];
            cswk = cost[sub] - xwk2 - work2[j];
            if (cswk < work1[j]) {
              work1[j] = cswk;
              aux4[j] = i;
            }
          }
        }
      }
  if (nn <= 1) fin = true;
  // examine the labeling and prepare for the next step
  iterate:
  while (true) {
    if (fin) {
      // generate the original graph by expanding all shrunken blossoms
      skip = false;
      value = 0.;
      for (i=1; i<=n; i++)
        if (aux2[i] == i) {
          if (aux6[i] >= 0) {
            kk5 = sol[i];
            kk2 = aux2[kk5];
            kk4 = sol[kk2];
            aux6[i] = -1;
            aux6[kk2] = -1;
            min = kk4;
            max = kk5;
            if (kk4 != kk5) {
              if (kk4 > kk5) {
                max = kk4;
                min = kk5;
              }
              sub = aux1[max] + min;
              xcst = cost[sub];
              value += xcst;
            }
          }
        }
      for (i=1; i<=n; i++) {
        while (true) {
          idxb = aux2[i];
          if (idxb == i) break;
          mm2 = aux3[idxb];
          idxd = aux4[mm2];
          kk3 = mm2;
          xwork = work4[mm2];
          do {
            mm1 = mm2;
            idxe = aux5[mm1];
            xwk2 = work2[mm1];
            while (true) {
              aux2[mm2] = mm1;
              work3[mm2] -= xwk2;
              if (mm2 == idxe) break;
              mm2 = aux3[mm2];
            }
            mm2 = aux3[idxe];
            aux3[idxe] = mm1;
          } while (mm2 != idxd);
          work2[idxb] = xwork;
          aux3[idxb] = idxd;
          mm2 = idxd;
          while (true) {
```

```
        work3[mm2] -= xwork;
        if (mm2 == idxb) break;
        mm2 = aux3[mm2];
      }
      mm5 = sol[idxb];
      mm1 = aux2[mm5];
      mm1 = sol[mm1];
      kk1 = aux2[mm1];
      if (idxb != kk1) {
        sol[kk1] = mm5;
        kk3 = aux7[kk1];
        kk3 = aux2[kk3];
        do {
          mm3 = aux6[kk1];
          kk2 = aux2[mm3];
          mm1 = aux7[kk2];
          mm2 = aux8[kk2];
          kk1 = aux2[mm1];
          sol[kk1] = mm2;
          sol[kk2] = mm1;
          min = mm1;
          max = mm2;
          if (mm1 == mm2) {
            skip = true;
            break;
          }
          if (mm1 > mm2) {
            max = mm1;
            min = mm2;
          }
          sub = aux1[max] + min;
          xcst = cost[sub];
          value += xcst;
        } while (kk1 != idxb);
        if (kk3 == idxb) skip = true;
      }
      if (skip)
        skip = false;
      else {
        while (true) {
          kk5 = aux6[kk3];
          kk2 = aux2[kk5];
          kk6 = aux6[kk2];
          min = kk5;
          max = kk6;
          if (kk5 == kk6) break;
          if (kk5 > kk6) {
            max = kk5;
            min = kk6;
          }
          sub = aux1[max] + min;
          xcst = cost[sub];
          value += xcst;
          kk6 = aux7[kk2];
          kk3 = aux2[kk6];
          if (kk3 == idxb) break;
        }
      }
    }
  }
  weight[0][0] = value;
  return;
}
```

```
    cstlow = big;
    for (i=1; i<=n; i++)
      if (aux2[i] == i) {
        cst = work1[i];
        if (aux6[i] < head) {
          cst = 0.5 * (cst + work4[i]);
          if (cst <= cstlow) {
            index = i;
            cstlow = cst;
          }
        }
        else {
          if (aux7[i] < head) {
            if (aux3[i] != i) {
              cst += work2[i];
              if (cst < cstlow) {
                index = i;
                cstlow = cst;
              }
            }
          }
          else {
            if (cst < cstlow) {
              index = i;
              cstlow = cst;
            }
          }
        }
      }
    if (aux7[index] >= head) {
      skip = false;
      if (aux6[index] < head) {
        idxd = aux4[index];
        idxe = aux5[index];
        kk4 = index;
        kk1 = kk4;
        kk5 = aux2[idxd];
        kk2 = kk5;
        while (true) {
          aux7[kk1] = kk2;
          mm5 = aux6[kk1];
          if (mm5 == 0) break;
          kk2 = aux2[mm5];
          kk1 = aux7[kk2];
          kk1 = aux2[kk1];
        }
        idxb = kk1;
        kk1 = kk5;
        kk2 = kk4;
        while (true) {
          if (aux7[kk1] < head) break;
          aux7[kk1] = kk2;
          mm5 = aux6[kk1];
          if (mm5 == 0) {
            // augmentation of the matching
            // exchange the matching and non-matching edges
            //   along the augmenting path
            idxb = kk4;
            mm5 = idxd;
            while (true) {
              kk1 = idxb;
              while (true) {
                sol[kk1] = mm5;
```

```
    mm5 = aux6[kk1];
    aux7[kk1] = head;
    if (mm5 == 0) break;
    kk2 = aux2[mm5];
    mm1 = aux7[kk2];
    mm5 = aux8[kk2];
    kk1 = aux2[mm1];
    sol[kk2] = mm1;
  }
  if (idxb != kk4) break;
  idxb = kk5;
  mm5 = idxe;
}
// remove all labels on on-exposed base nodes
for (i=1; i<=n; i++)
  if (aux2[i] == i) {
    if (aux6[i] < head) {
      cst = cstlow - work4[i];
      work2[i] += cst;
      aux6[i] = head;
      if (sol[i] != head)
        work4[i] = big;
      else {
        aux6[i] = 0;
        work4[i] = 0.;
      }
    }
    else {
      if (aux7[i] < head) {
        cst = work1[i] - cstlow;
        work2[i] += cst;
        aux7[i] = head;
        aux8[i] = head;
      }
      work4[i] = big;
    }
    work1[i] = big;
  }
nn -= 2;
if (nn <= 1) {
  fin = true;
  continue iterate;
}
// determine the new work1 values
for (i=1; i<=n; i++) {
  kk1 = aux2[i];
  if (aux6[kk1] == 0) {
    xwk2 = work2[kk1];
    xwk3 = work3[i];
    for (j=1; j<=n; j++) {
      kk2 = aux2[j];
      if (kk1 != kk2) {
        min = i;
        max = j;
        if (i != j) {
          if (i > j) {
            max = i;
            min = j;
          }
          sub = aux1[max] + min;
          xcst = cost[sub];
          cswk = cost[sub] - xwk2 - xwk3;
          cswk -= (work2[kk2] + work3[j]);
```

```
                        if (cswk < work1[kk2]) {
                          aux4[kk2] = i;
                          aux5[kk2] = j;
                          work1[kk2] = cswk;
                        }
                      }
                    }
                  }
                }
              }
              continue iterate;
            }
            kk2 = aux2[mm5];
            kk1 = aux7[kk2];
            kk1 = aux2[kk1];
          }
          while (true) {
            if (kk1 == idxb) {
              skip = true;
              break;
            }
            mm5 = aux7[idxb];
            aux7[idxb] = head;
            idxa = sol[mm5];
            idxb = aux2[idxa];
          }
        }
        if (!skip) {
          // growing an alternating tree, add two edges
          aux7[index] = aux4[index];
          aux8[index] = aux5[index];
          idxa = sol[index];
          idxc = aux2[idxa];
          work4[idxc] = cstlow;
          aux6[idxc] = sol[idxc];
          msmSubprogramb(idxc,n,big,cost,aux1,aux2,aux3,aux4,
                         aux5,aux7,aux9,work1,work2,work3,work4);
          continue;
        }
        skip = false;
        // shrink a blossom
        xwork = work2[idxb] + cstlow - work4[idxb];
        work2[idxb] = 0.;
        mm1 = idxb;
        do {
          work3[mm1] += xwork;
          mm1 = aux3[mm1];
        } while (mm1 != idxb);
        mm5 = aux3[idxb];
        if (idxb == kk5) {
          kk5 = kk4;
          kk2 = aux7[idxb];
        }
        while (true) {
          aux3[mm1] = kk2;
          idxa = sol[kk2];
          aux6[kk2] = idxa;
          xwk2 = work2[kk2] + work1[kk2] - cstlow;
          mm1 = kk2;
          do {
            mm2 = mm1;
            work3[mm2] += xwk2;
            aux2[mm2] = idxb;
```

```
          mm1 = aux3[mm2];
        } while (mm1 != kk2);
        aux5[kk2] = mm2;
        work2[kk2] = xwk2;
        kk1 = aux2[idxa];
        aux3[mm2] = kk1;
        xwk2 = work2[kk1] + cstlow - work4[kk1];
        mm2 = kk1;
        do {
          mm1 = mm2;
          work3[mm1] += xwk2;
          aux2[mm1] = idxb;
          mm2 = aux3[mm1];
        } while (mm2 != kk1);
        aux5[kk1] = mm1;
        work2[kk1] = xwk2;
        if (kk5 != kk1) {
          kk2 = aux7[kk1];
          aux7[kk1] = aux8[kk2];
          aux8[kk1] = aux7[kk2];
          continue;
        }
        if (kk5 != index) {
          aux7[kk5] = idxe;
          aux8[kk5] = idxd;
          if (idxb != index) {
            kk5 = kk4;
            kk2 = aux7[idxb];
            continue;
          }
        }
        else {
          aux7[index] = idxd;
          aux8[index] = idxe;
        }
        break;
      }
      aux3[mm1] = mm5;
      kk4 = aux3[idxb];
      aux4[kk4] = mm5;
      work4[kk4] = xwork;
      aux7[idxb] = head;
      work4[idxb] = cstlow;
      msmSubprogramb(idxb, n, big, cost, aux1, aux2, aux3, aux4,
                     aux5, aux7, aux9, work1, work2, work3, work4);
      continue iterate;
    }
    // expand a t-labeled blossom
    kk4 = aux3[index];
    kk3 = kk4;
    idxd = aux4[kk4];
    mm2 = kk4;
    do {
      mm1 = mm2;
      idxe = aux5[mm1];
      xwk2 = work2[mm1];
      while (true) {
        aux2[mm2] = mm1;
        work3[mm2] -= xwk2;
        if (mm2 == idxe) break;
        mm2 = aux3[mm2];
      }
      mm2 = aux3[idxe];
```

```
      aux3[idxe] = mm1;
    } while (mm2 != idxd);
    xwk2 = work4[kk4];
    work2[index] = xwk2;
    aux3[index] = idxd;
    mm2 = idxd;
    while (true) {
      work3[mm2] -= xwk2;
      if (mm2 == index) break;
      mm2 = aux3[mm2];
    }
    mm1 = sol[index];
    kk1 = aux2[mm1];
    mm2 = aux6[kk1];
    idxb = aux2[mm2];
    if (idxb != index) {
      kk2 = idxb;
      while (true) {
        mm5 = aux7[kk2];
        kk1 = aux2[mm5];
        if (kk1 == index) break;
        kk2 = aux6[kk1];
        kk2 = aux2[kk2];
      }
      aux7[idxb] = aux7[index];
      aux7[index] = aux8[kk2];
      aux8[idxb] = aux8[index];
      aux8[index] = mm5;
      mm3 = aux6[idxb];
      kk3 = aux2[mm3];
      mm4 = aux6[kk3];
      aux6[idxb] = head;
      sol[idxb] = mm1;
      kk1 = kk3;
      while (true) {
        mm1 = aux7[kk1];
        mm2 = aux8[kk1];
        aux7[kk1] = mm4;
        aux8[kk1] = mm3;
        aux6[kk1] = mm1;
        sol[kk1] = mm1;
        kk2 = aux2[mm1];
        sol[kk2] = mm2;
        mm3 = aux6[kk2];
        aux6[kk2] = mm2;
        if (kk2 == index) break;
        kk1 = aux2[mm3];
        mm4 = aux6[kk1];
        aux7[kk2] = mm3;
        aux8[kk2] = mm4;
      }
    }
    mm2 = aux8[idxb];
    kk1 = aux2[mm2];
    work1[kk1] = cstlow;
    kk4 = 0;
    skip = false;
    if (kk1 != idxb) {
      mm1 = aux7[kk1];
      kk3 = aux2[mm1];
      aux7[kk1] = aux7[idxb];
      aux8[kk1] = mm2;
      do {
```

```
        mm5 = aux6[kk1];
        aux6[kk1] = head;
        kk2 = aux2[mm5];
        mm5 = aux7[kk2];
        aux7[kk2] = head;
        kk5 = aux8[kk2];
        aux8[kk2] = kk4;
        kk4 = kk2;
        work4[kk2] = cstlow;
        kk1 = aux2[mm5];
        work1[kk1] = cstlow;
      } while (kk1 != idxb);
      aux7[idxb] = kk5;
      aux8[idxb] = mm5;
      aux6[idxb] = head;
      if (kk3 == idxb) skip = true;
    }
    if (skip)
      skip = false;
    else {
      kk1 = 0;
      kk2 = kk3;
      do {
        mm5 = aux6[kk2];
        aux6[kk2] = head;
        aux7[kk2] = head;
        aux8[kk2] = kk1;
        kk1 = aux2[mm5];
        mm5 = aux7[kk1];
        aux6[kk1] = head;
        aux7[kk1] = head;
        aux8[kk1] = kk2;
        kk2 = aux2[mm5];
      } while (kk2 != idxb);
      msmSubprograma(kk1, n, big, cost, aux1, aux2, aux3, aux4, aux5,
                     aux6, aux8, work1, work2, work3, work4);
    }
    while (true) {
      if (kk4 == 0) continue iterate;
      idxb = kk4;
      msmSubprogramb(idxb, n, big, cost, aux1, aux2, aux3, aux4,
                     aux5, aux7, aux9, work1, work2, work3, work4);
      kk4 = aux8[idxb];
      aux8[idxb] = head;
    }
  }
}
```

3. Conclusion

We have seen that there are several polynomial solutions for maximum matching problems. In detail we have looked at a simplified approach by Jack Edmonds using network flows, a concept of graph theory. Focussing on a specific matching problem - the *STA-problem*, or student to topic assignment problem - we needed to make two adjustments on the basic augmenting flow concept by Edmonds:

The first adjustment was adding weights to the edges in the student-topic graph representing priorities. We combine Edmonds idea with a weight-minimization algorithm to have a proper solution for our *STA-problem*.

The second adjustment was for having a special condition fulfilled, here called as *extended STA-problem*. Therefore, two different ideas were presented in this document.

After the explanation of the different mathematical and logical concepts, needed to understand how to solve the *STA-problem*, we have seen an explicit algorithm in JAVA running in polynomial time. It comes as a console-based program, solving only the unextended *STA-problem*. However, with the here presented knowledge it should be possible to include the extension.

Glossary

bipartite graph a graph is bipartite if it consists of two vertex sets. Every vertex is only connected with vertices of the other set. Graphically an arrow goes into this node.. 5

capacity a network term. 5

chain a network term. 8

edge One of the two basic units in graphs connecting vertices the other basic unit.. 5

extended STA-problem a STA problem that respects pensums of experts. 4

flow a network term. 5

graph a mathematical structure consisting of a vertex-set and an edge-set connecting the vertices. 5

matching a special set of vertex-pairs in a graph. 6

matching problem finding a special matching (for example maximum matching). 6

maximum matching special form of matching often desired to find. 6

network a special directed graph having a flow. 5

node-source having a directed edge it is the node where the edge goes out. Graphically an arrow goes out of this node.. 5

path a sequence of edges in a graph. 5

STA-problem the problem of finding a student to topic assignment. 4

super-sink a unique vertex in a network swallowing all flow. 5

super-source a unique vertex in a network generating all flow. 5

vertex also called node. Besides edges one of the two basic units of which graphs are formed. 5

Bibliography

[1] Alan Tucker, *Applied Combinatorics*, 6th edition, 1995.

[2] Hang T. Lau, *A Java Library of Graph Algorithms and Optimization*, 2007.

[3] Jack Edmonds, *Path, Trees and Flowers*, 1965.

[4] P. Lammich, S. R. Sefidgar, *Formalizing Network Flow Algorithms*, 2017.
https://link.springer.com/article/10.1007

[5] O. B. Matsiy, A. V. Morozov, A. V. Panishev, *A Recurrent Algorithm to Solve the Weighted Matching Problem*, 2016.
https://link.springer.com/article/10.1007/s10559-016-9876-4

[6] Helpful Wikipedia-Articles, *Graph Theory, Matching, Flow Network and Blossom algorithm*:
https://en.wikipedia.org/wiki/Graph_theory
https://en.wikipedia.org/wiki/Matching_(graph_theory)
https://en.wikipedia.org/wiki/Flow_network
https://en.wikipedia.org/wiki/Blossom_algorithm

Declaration of Authorship

Hereby I declare, that the content of this document is result of my work and thinking. Everything that inspired me to the here discussed topic in a direct manner is referenced. Still I'm aware that human mind mostly relies on conditioning, so I wouldn't call that what I didn't referenced as "my work", but rather a partial mirror of all that what have shaped me in my life. Still on a formal, non-philospical perspective, that what I didn't referenced is hereby declared as my work.