UNIVERSITÀ DI PISA

Dipartimento di Informatica
Corso di Laurea Triennale in Informatica

Tesi di Laurea

THE MINIMAL REPRESENTATIVE SELECTION PROBLEM: APPLICATIONS, OPTIMAL ALGORITHMS AND EFFICIENT HEURISTICS

Relatore:	Candidato:
Prof. ANNA BERNASCONI	DAVIDE RUCCI

ANNO ACCADEMICO 2016 - 2017

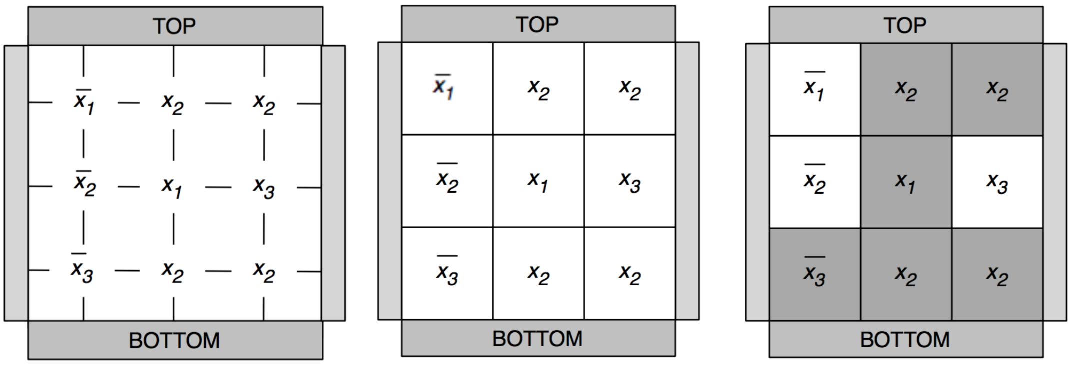
Table of Contents

Introduction

In recent years, the use of *nanoscale technologies* to create electronic devices has revived interest in the use of regular structures for defining complex logic functions. One such structure is the *switching lattice*, that was initially proposed by Akers in 1972 [1], which is a wire mesh that results from laying m parallel wires perpendicularly on top of n parallel wires, with some kind of switches inserted wherever two wires cross. The kind of switch we consider is called "four-terminal" switch. A four-terminal switch is, ideally, a simple device that is linked with four neighbors and that can be either *on* or *off*: in the first case, the current arriving in the switch can flow to all of its four neighbors while in the other case, its neighboring cells are all disconnected.

These lattices can be used to implement a Boolean function by, firstly, connecting all the wire ends at the top to a "top plate" T and all the wire ends at the bottom to a "bottom plate" B and, secondly, having each four-terminal switch controlled by an input. An input wire can be inverted, thus allowing to associate each four-terminal switch to a Boolean literal or its negation, namely x or $\neg x$. Some input combinations will create a (possibly meandering) path between top and bottom plates: we associate those combinations with output being 1, or *true*, while every combination that leaves T and B unconnected is associated with the output 0, or *false*. In a similar way, we can also equip a switching lattice with "left" and "right" plates, L and R respectively, and the left-to-right connectivity can be associated to a Boolean function as well. In this way, a single lattice can implement two different Boolean functions, although not simultaneously. Altun and Riedel, in 2012, exploited this possibility to develop a new synthesis algorithm [2] for this kind of switching lattices, that is, an algorithm that, given a target Boolean function, assigns a literal to each four-terminal switch in a way such that there is connectivity between T and B if and only if the target function evaluates to true.

If we picture the m wires laid down horizontally, we can visualize a matrix made by m rows and n columns of switches, as shown in Figure 1.



(a) A four-terminal switching network implementing the Boolean function $f = \neg x_1 \neg x_2 \neg x_3 + x_1 x_2 + x_2 x_3$.

(b) Its corresponding lattice form.

(c) The lattice evaluated on the assignment $\langle 1, 1, 0\rangle$. Note that the corresponding output is 1 since there is a connection from top to bottom (the grey path).

Figure 1: An example of a switching lattice.

We now describe the algorithm proposed in [2] for the synthesis of a $m \times n$ switching lattice. Let f_T be the function to be implemented by the lattice, and let f_T^D be the *dual function* of f_T, i.e., the function obtained from f_T by interchanging the *AND* and *OR* operations, as well as interchanging the 0 and 1

constants. All the functions are represented in ISOP form, which stands for *Irredundant Sum-Of-Products*. Informally, a Boolean function f is in ISOP form if it is expressed by an OR (sum) of ANDs (products) and if no product can be removed from it without changing f. For example, $f = (x_1x_2) + (\neg x_1x_3)$ is in ISOP form, and its dual function is $f^D = (x_1 + x_2)(\neg x_1 + x_3)$.
The algorithm is as follows (suppose that f_T and f_T^D have n and m products, respectively):

- Start with a $m \times n$ lattice;
- Assign each product of f_T to a column, and each product of f_T^D to a row;
- Compute the intersection[1] of the literals in the corresponding products.
- From the resulting sets associated with each cell, pick one literal arbitrarily and assign it to the corresponding site of the lattice.

This way, there will be a connected path from T to B if and only if f_T evaluates to 1. A proof of the correctness of this algorithm can be found in [2, Section 3.2], while Figure 2 provides a visualization of it.

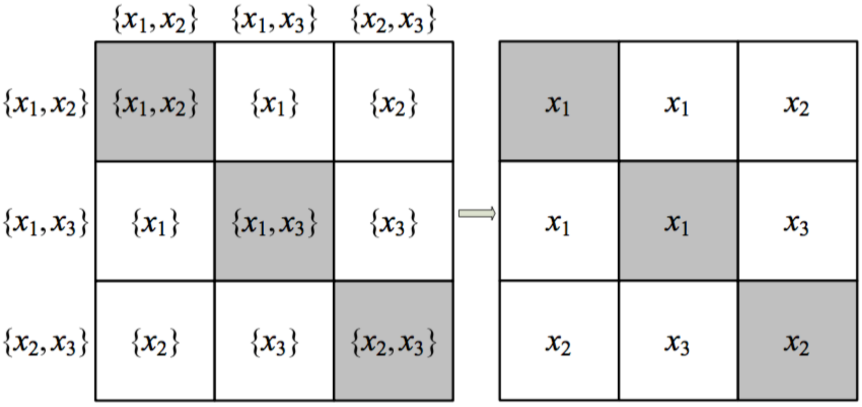

{x1,x2} {x1,x3} {x2,x3}
{x1,x2} {x1,x2} {x1} {x2}
{x1,x3} {x1} {x1,x3} {x3}
{x2,x3} {x2} {x3} {x2,x3}
x1 x1 x2
x1 x1 x3
x2 x3 x2

Figure 2: Left: the intersections computed by the algorithm for the function $f_T = x_1x_2 + x_1x_3 + x_2x_3$ on a 3×3 switching lattice. Note that $f_T^D = f_T$. Right: an arbitrarily assignment of the variables to their corresponding sites on the lattice.

Our work is motivated by the last step of the algorithm, specifically by the way in which the literals are chosen from the sets created after doing the rows-columns intersection. When synthesizing this kind of lattices it may be a good idea to not pick too many times the same literal, because plugging the same literal in many different cells has a higher cost, in terms of current supply needed, than plugging, say, many different literals in fewer cells. For this reason, our work focuses on *minimizing the maximum frequency* of all the literals, which we will call *variables*, across all the sets.

Although this problem has not been directly studied in the past, we have found some interesting connections to many different, and known, problems such as graphs orientation problems, networks flow problems and scheduling problems. In fact, the latter area gave us the most important results from a theoretical point of view, since it has allowed us to state that our problem can be solved in polynomial time by providing two algorithms that are able to solve always correctly our problem. However, we have put the majority of our efforts in developing heuristics for this problem, for basically two good reasons. The first focuses on execution times: heuristics, by their nature, tend to be way faster than "standard" algorithms at the cost of possibly introducing errors in the solutions they give; the heuristics that we propose, and in particular the first two, are very fast even with large input instances and they have shown, in our experiments, a success rate high enough to induce one to prefer them instead of an always-optimal algorithm. The second reason, which is in some sense the most important within this context, is that one of the two optimal algorithms that we are going to describe in Chapter 2 bases his work on an initialization with a heuristic. In other words, this algorithm takes an arbitrary, possibly non-optimal, solution and repeatedly improves it until it becomes optimal. It is clear that the better its initialization, the less time the algorithm will take to produce an optimal solution.

---

[1] The intersection will always contain at least one element, as shown in [2, Lemma 1].

Chapter 1

Preliminaries

Let $\mathcal{N} = \{x_1, x_2, \ldots, x_n\}$ be a set of n variables and let $\mathcal{F}$ be a family of m subsets of $\mathcal{N}$. We consider the problem of finding a representative for each subset in $\mathcal{F}$ such that the frequency of the most chosen one is minimal, i.e., the maximum frequency is minimal. In general, $\mathcal{F}$ can contain repeated sets. We call this problem *Minimal Representative Selection*, or **MRS**, which can be formally defined as follows:

Definition 1.1 (MRS)**.** Given $\mathcal{N}$ and $\mathcal{F}$, the **MRS** problem is:

> **INSTANCE**: a set $\mathcal{N}$ of n variables and a family $\mathcal{F}$ of m subsets of $\mathcal{N}$.
>
> **QUESTION**: find an assignment of representatives for every subset in $\mathcal{F}$ such that the frequency of the most selected one is minimized.

We will show that **MRS** is in $\mathcal{P}$ by describing two algorithms [4] that solve it in polynomial time. These two algorithms come from the area of scheduling problems. In fact, we will show how this problem can be reduced to, at least, three different problems, the first one is valid only for a restricted version of **MRS** [5], while the other two can solve every instance of it [4, 6].
For various reasons, we can put weights on the subsets in $\mathcal{F}$, but this will make **MRS** $\mathcal{NP}$-hard (see the the reduction from PARTITION in [5]).

1.0.1 Notation

The size of an input instance of **MRS** is defined as $k = \sum_{i=1}^{m} |\mathcal{F}[i]|$ where $\mathcal{F}[i]$ is the ith set in $\mathcal{F}$. We will refer to the *value*[1] of the optimal solution as t^*, while the value of a generic solution will be denoted by t. Let S be a subset of $\mathcal{N}$ and let $\mathcal{F}_{|S} = \{A \in \mathcal{F} \mid \forall\, a \in A \implies a \in S\}$. We call $\mathcal{F}_{|S}$ the *subfamily induced by S*. To lighten the notation, we denote the cardinality of the two sets $\mathcal{F}_{|S}$ and S as m_{S} and n_{S} respectively.

1.1 Related Work

The **MRS** problem as defined above has not been directly studied in the past. However, there are some interesting points in common with problems from different areas such as graph orientation problems [5], network flow problems [6] and a variant of bipartite maximum matching called *semi-matching*, used to solve a particular type of the *offline scheduling problem* [4]. These problems are all solvable in polynomial time, making our problem polynomially solvable too, thanks to three simple reductions.

We first explain the reduction to a graph orientation problem that is possible, however, only for a restricted version of **MRS**, which we call **2-MRS**. In this restricted version, we ask that all subsets in $\mathcal{F}$ have cardinality *exactly* 2 and are distinct. The instances of this problem can be then modeled with a graph as follows. Let $G = (V, E)$ be a simple, undirected, weighted graph with $V = \mathcal{N}$ and $E = \mathcal{F}$. We can say that choosing a variable from each subset, represented by the edges in G, is equivalent to

---

[1] The value of a solution of **MRS** is the frequency of the most chosen variable as representative.

choosing an orientation for every member of E. Since we want to minimize the maximum frequency of the representatives, we have to minimize the maximum indegree or outdegree of the vertices of G depending on how we choose to orient the edges. We focus on minimizing the *outdegree*, meaning that the orientation of the edges will be from the variable chosen as representative of each subset, to the other variable paired with it. The results we report here come from [5], while the corresponding results for the indegree version can be found in [7].

The *Minimum Maximum Outdegree Orientation* problem, or *MMO* is, formally, the problem of giving an orientation to every edge of a weighted graph which minimizes the maximum weighted outdegree of its vertices and it is in general $\mathcal{NP}$-hard, but in the special case of all equal weights, the problem becomes polynomial. Since we have not put weights on the subsets in $\mathcal{F}$, we can consider the weight function of G as $w(\{i,j\}) = 1$ for every $\{i,j\} \in E$ and we can solve **2-MRS** in $\mathcal{O}(m^2)$ time thanks to the algorithm Reverse described in [5]. The basic strategy of the aforementioned algorithm is to find first an arbitrary orientation of edges and then repeatedly improve it by reversing the orientation of edges along a path that satisfies two inequalities related to the outdegree of its first and last vertices. Figure 1.1 shows an example of the graph corresponding to the **2-MRS** instance defined by $\langle \mathcal{N} = \{1,2,3,4,5\}, \mathcal{F} = \{\, \{1,2\}, \{1,3\}, \{2,3\}, \{2,4\}, \{4,5\} \,\} \rangle$ and its optimal solution $(1,3,2,4,5)$.

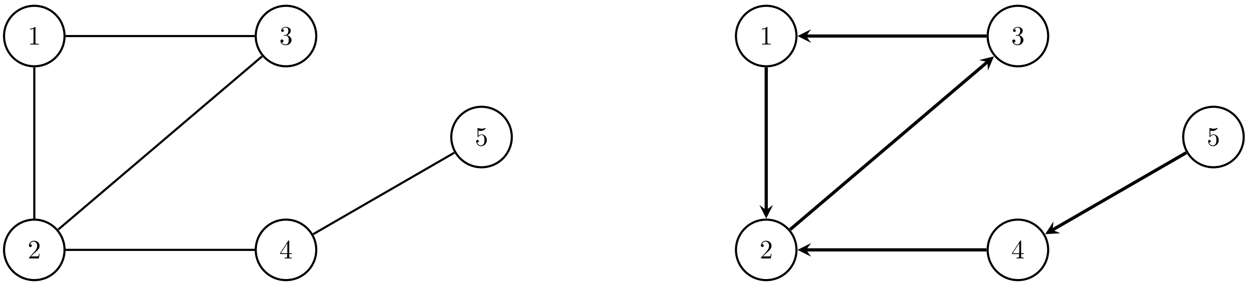


Figure 1.1: The graph of an instance of **MRS** (left) and the solution obtained by applying the Reverse algorithm to it (right).

The area of scheduling problems is, however, the one that provides us with the most useful reductions for solving every instance of **MRS**. The scheduling problem that is equivalent to **MRS** is a restricted version of the *offline Parallel Machine Scheduling* problem, or *PMS*, which has received intensive study in the literature (see, for example, [8,9]). The parallel machine scheduling problem, denoted by the so-called *three-field representation* $R \mid \circ \mid C_{\max}{}^2$ [10], is the problem of assigning m jobs to n identical, unrelated, machines. The goal is to minimize one or more objectives such as the *makespan*[3] of the schedule or the *flow time*[4] of the jobs. If all the jobs require one unit of time to complete and can be executed only on a specific subset of the available machines, the problem is denoted by $R|p_j = 1, \mathcal{M}_j|C_{\max}{}^5$. Minimizing the makespan is, in particular, the main objective of **MRS**, as it is equivalent to minimize the maximum frequency of all the representatives chosen.

We first show how **MRS** can be reduced to PMS with unit-length jobs. Every variable in $\mathcal{N}$ is a machine, and every subset in $\mathcal{F}$ can be thought as a job that can be executed by the machines denoted by the variables it contains. For example, the subset $\{1,2,3\}$ of $\mathcal{N} = \{1,2,3,4,5\}$ represents a unit-length job that can be executed either on machine 1, 2 or 3, but not on machines 4 or 5. More formally, we can define $J = \mathcal{F}$ as the set of jobs, $\mathcal{M} = \mathcal{N}$ as the set of machines and fix $p_j = 1$ for every $1 \leq j \leq m$. The solution of PMS, that is an assignment of jobs to machines, can be seen as an assignment of sets to a variable, thus creating the choice of representatives for each subset in $\mathcal{F}$.

The $R|p_j = 1, \mathcal{M}_j|C_{\max}$ problem, obtained by the reduction from **MRS**, can be solved by a polynomial

---

[2] For the sake of clarity, R stands for *unrelated* parallel machines, $\circ$ indicates that there are no particular restrictions on the jobs, and $C_{\max}$ is the objective we want to reach, that is the one of minimizing the makespan.

[3] The *makespan* of a schedule is the difference, in time, between its start and its finish. With unit-length jobs, it represents the maximum number of jobs assigned to a machine.

[4] The *flow time* of a job is defined as the time at which the job is completed minus the time at which the job was made available for processing to its machine.

[5] Note that the second field is no longer empty: it contains a restriction on the time that it takes for each job to complete ($p_j = 1$) and a restriction on the machines that are capable to execute the job j ($\mathcal{M}_j \subseteq \mathcal{M}$ = the set of all the machines available).

time algorithm for solving the *network maximum flow* problem (see, for example, algorithms in [11]), thus involving a further reduction to graphs. Let $G = (V, E, w)$ be a simple, directed, and capacitated graph with $V = \mathcal{F} \cup \mathcal{N} \cup \{s, t\}$. We denote by x_i, for $1 \leq i \leq n$, the vertices that represent a variable in $\mathcal{N}$ and by y_j, for $1 \leq j \leq m$, the vertices that represent a subset in $\mathcal{F}$. Since s and t will be the source and the sink of the maximum flow, respectively, E will be constructed as follows:

- an edge (s, x_i) for every variable-vertex x_i;
- an edge (x_i, y_i) if and only if the variable represented by vertex x_i is in the corresponding subset of $\mathcal{F}$ represented by the vertex y_j;
- an edge (y_j, t) for every subset-vertex y_j.

Then we define the *capacity function* $w : E \to \mathbb{N}$ as:

- $w(s, x_i) = c$ where c is a parameter;
- $w(x_i, y_j) = w(y_j, t) = 1$ for every i and j.

The following theorem correlates the maximum flow problem and PMS. The proof can be found in [6].

Theorem 1.1. *The problem* $R|p_j = 1, \mathcal{M}_j|C_{max}$ *has a feasible makespan of* $C_{max} \leq c$ *if and only if the value of the maximum flow in* G *is* m.

The algorithm proposed in [6] is a simple "binary search" of the minimum possible value of c. We report the pseudocode here:

Algorithm 1 (Binary Search)

```
function BS($G$)
    $l \leftarrow 1$
    $u \leftarrow m$
    repeat
        $c \leftarrow \left\lfloor \frac{l+u}{2} \right\rfloor$
        $f \leftarrow$ The maximum flow in $G$
        if $f = m$ then                                   ▷ $C^*_{max} \leq c$
            $u \leftarrow c$
        else                                              ▷ $C^*_{max} > c$
            $l \leftarrow c + 1$
    until ($l = u$)
    return The schedule of makespan $l$    ▷ The optimal $c$ value is reached ($C^*_{max} = l$)
```

Theorem 1.2. *The Binary Search algorithm correctly solves the* $P|p_j = 1, \mathcal{M}|C_{max}$ *problem in* $\mathcal{O}(m^3 \log m)$ *time. [6]*

Although this is a polynomial algorithm that correctly solves every instance of **MRS**, we are not completely satisfied by its time bound, cubic in m, which can be very large in practice and sometimes it can be even exponential in n. This is because the main strategy of this algorithm, i.e., finding the maximum flow in G, is not specialized on solving this kind of problems. For this reason, in the next chapter we describe two algorithms with lower running times: $\mathcal{O}(mk) = \mathcal{O}(m^2 n)$ and $\mathcal{O}(m^2 n^2)$ respectively. The second bound, however, is loose and we will show that in practice, the latter algorithm performs even better than the former.

1.2 Characterization of optimal solutions

We provide three theoretical results on **MRS**. The first two are bounds on the solutions, that can be used for developing good heuristics, and the third allows us to simplify the input instances, i.e., to reduce the size of the input. Recall that t and t^* denote the value of a generic and an optimal solution respectively.

Proposition 1.2.1 (Weak Lower Bound)**.** $t^* \geq \left\lceil \frac{m}{n} \right\rceil$

Proof. Let r be the number of representatives chosen in a solution of **MRS**. Clearly $r = m$. For the sake of contradiction, suppose that $t^* < \left\lceil \frac{m}{n} \right\rceil$. We now distinguish two cases: first assume that $n \mid m$. This implies that $t^* < \frac{m}{n}$. We can give an upper bound on r by saying that we choose each variable in $\mathcal{N}$ as representative at most t^* times.

$$r \leq n \cdot t^* < n \cdot \left\lceil \frac{m}{n} \right\rceil = n \cdot \frac{m}{n} = m$$

with the strict inequality coming from our hypothesis on t^*. This leads us to conclude that $r < m$, which is a contradiction.
Conversely, if $n \nmid m$, then $t^* < \left\lceil \frac{m}{n} \right\rceil \implies t^* \leq \left\lfloor \frac{m}{n} \right\rfloor$, so

$$r \leq n \cdot \left\lfloor \frac{m}{n} \right\rfloor < n \cdot \frac{m}{n} = m$$

which is the same contradiction as above. In other words we left at least one set without its representative, thus $t^* \geq \left\lceil \frac{m}{n} \right\rceil$. ☐

In a similar way, we can improve this lower bound by considering all the subfamilies induced by any subset $S \subseteq \mathcal{N}$ on $\mathcal{F}$.

Theorem 1.3 (Lower Bound)**.** $t^* \geq \max\limits_{S \subseteq \mathcal{N}} \left\lceil \frac{m_S}{n_S} \right\rceil$

Proof. We distinguish two cases as before. First, assume that $n_S \mid m_S$ and suppose, for the sake of contradiction, that there exists $S \subseteq \mathcal{N}$ such that $t^* < \left\lceil \frac{m_S}{n_S} \right\rceil$. This implies that $t^* < \frac{m_S}{n_S}$. Now, using the same argument as in Proposition 1.2.1, we can obtain the following upper bound on the number of representatives chosen in $\mathcal{F}_{|S}$:

$$m_S \leq n_S \cdot t^* < n_S \cdot \frac{m_S}{n_S} = m_S$$

which is a contradiction. Conversely, if $n_S \nmid m_S$, the hypothesis $t^* < \left\lceil \frac{m_S}{n_S} \right\rceil$ implies that $t^* \leq \left\lfloor \frac{m_S}{n_S} \right\rfloor$. The upper bound now is

$$m_S \leq n_S \cdot t^* \leq n_S \cdot \left\lfloor \frac{m_S}{n_S} \right\rfloor < n_S \cdot \frac{m_S}{n_S} = m_S$$

which is also a contradiction, with the strict inequality coming from the well-know properties of the ceiling and floor functions. This means that not reaching this lower bound on every possible $\mathcal{F}_{|S}$ implies leaving at least one set without its representative. ☐

This theorem follows the intuition that, when there are many singletons of the same variable, the lower bound is at least the number of occurrences of that singleton. In the other cases, it allows us to say that there exists a "difficult core" in the initial family that inevitably increases the global lower bound. It also allows us to simplify the instances of **MRS** by immediately choosing as representatives the variables that appear less then this lower bound without altering the value of the optimal solution, as explained with the following Corollary.

Corollary 1.3.1 (Input Simplification)**.** *If a variable in $\mathcal{N}$ appears less than t^* times in $\mathcal{F}$ then there exists an optimal solution in which that variable is chosen as representative in every subset containing it.*

Proof. Suppose that the variable x appears in $z < t^*$ sets of $\mathcal{F}$. We are going to prove that there exists an optimal solution with x being chosen exactly z times by giving an algorithm that constructs this solution from an optimal solution already obtained. Let W be an optimal solution for **MRS** with x chosen less than z times and apply the following algorithm to it:

1. Sort the variables according to the number of times they have been chosen as representative, in ascending order, ignoring the variable x;
2. Loop through the sorted variables:

(i) If the current variable appears in a set containing x, change the representative of this set to be x, otherwise consider the next variable;

(ii) If x has been chosen z times, stop.

This algorithm never changes the representative of a set which contains x and the *last* most chosen variable, otherwise W would not have been an optimal solution. It also cannot worsen the value of W because it only decreases frequencies of the other variables and increases the one of x which is bounded by $z < t^*$. □

It is important to note that this algorithm can be recursively applied to the instance it outputs, because the instance created is actually a new, *independent*, one with different values of m and n.
Before proceeding, we give some practical considerations. This kind of simplification is generally possible with gaussian-distributed variables across the sets in $\mathcal{F}$, if m is sufficiently large. This is likely to happen because, given the nature of **MRS**, m can be even exponential in n, thus increasing the value of $\lceil \frac{m}{n} \rceil$ which is, almost always, the only lower bound efficiently computable[6]. With a large $\lceil \frac{m}{n} \rceil$ and a gaussian distribution of the variables, there will be at least one "removable" variable in many instances. The same reasoning cannot be applied when the variables are uniformly distributed across the sets in $\mathcal{F}$ because, for large values of m, a variable will tend to appear the same number of times with respect to another one, making the simplification to be roughly impossible.

---

[6] A lower bound higher than the one given in Proposition 1.2.1 can still be obtained by counting the occurrences of the most-frequent singleton.

Chapter 2

Two optimal algorithms

In this chapter we describe two algorithms for computing an optimal solution of **MRS**. These algorithms have been designed to compute an optimal semi-matching in bipartite graphs [4]. We first define semi-matchings, then we describe some of their properties and the two optimal polynomial-time algorithms.

Let $G = (U \cup V, E)$ be an undirected, bipartite graph, in which $E \subseteq U \times V$. A *matching* M in G is a set of edges, $M \subseteq E$, such that each vertex in $U \cup V$ is an endpoint of at most one edge in M. In the following we will say that a vertex $u \in U$ is *matched* with the vertex $v \in V$ if there is an edge in M from u to v and vice-versa.

Definition 2.1 (Semi-Matching)**.** Let $G = (U \cup V, E)$ be a bipartite graph. A semi-matching in G is a subset of edges, $M \subseteq E$, such that each vertex $u \in U$ is matched with *exactly* one vertex $v \in V$.

Remark. A semi-matching cannot exist if G has at least an isolated U-vertex.

Note that finding a semi-matching in G is trivial, just match each vertex in U with one vertex of V. Recall that semi-matchings in [4] are used to solve an offline scheduling problem, in which the optimization objectives are many and different from each other. One is to minimize the *makespan* of the schedule, while another possibile objective can be to minimize the average completion time, of the tasks. The following algorithms were designed to reach these many different optimization objectives at the same time and we will show how this can be achieved in practice.

The scheduling problem defined in Section 1.1 can be translated in terms of semi-matchings as follows: each task is represented by a vertex $u \in U$ and each machine is represented by a vertex $v \in V$. There is an undirected edge $\{u, v\}$ if the task u can be executed on the machine v. The edges in the semi-matching found mean to assign job u to machine v.

2.1 Definitions and Properties

Recall the equivalence of U and V to our $\mathcal{F}$ and $\mathcal{N}$, respectively. For $v \in V$, let $\deg(v)$ denote the degree of vertex v or, in load balancing terms, the number of tasks that machine v is capable to process. Given a semi-matching M, we denote with $\deg_M(v)$ the degree of v in M, but we will frequently refer to it as the *load* on vertex v. Since each task requires one unit of time to complete, we can define the *cost* of a vertex $v \in V$ as its jobs' *flow time*.

Definition 2.2 (Vertex Cost)**.** $\mathrm{cost}_M(v) = \sum_{i=1}^{\deg_M(v)} i = \frac{(\deg_M(v)+1)\deg_M(v)}{2}$

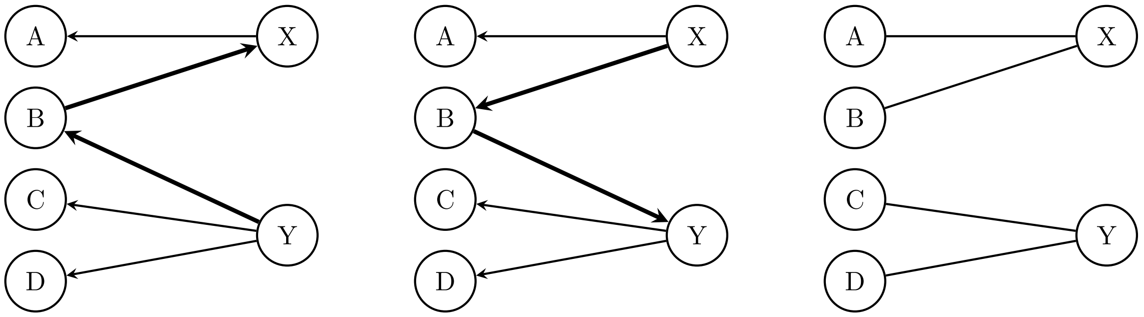
This is because we suppose that all jobs are available for processing to their assigned machine at time 0. In a similar way, we define the *total cost* of a semi-matching M as the sum of all V-vertices' cost.

Definition 2.3 (Total Cost)**.** $C(M) = \sum_{i=1}^{n} \mathrm{cost}_M(v_i)$

A semi-matching is optimal when its cost is minimized and an optimal semi-matching is also optimal with respect to other optimization objectives, as shown later in this section.
The following definitions create the basis for the theoretical results that will be used in the next algorithms.

Definition 2.4 (Alternating Path)**.** Let M be a semi-matching in G. An alternating path is a sequence of edges $P = (\{v_1, u_1\}, \{u_1, v_2\}, \{v_2, u_2\}, \ldots, \{u_{k-1}, v_k\})$ in which $v_i \in V$, $u_i \in U$ and $\{v_i, u_i\} \in M$ for each $1 \leq i \leq k$.

For convenience, in the following we often treat paths as a sequence of vertices instead of a sequence of edges, i.e., $P = (v_1, u_1, v_2, u_2, \ldots, u_{k-1}, v_k)$. So far we have considered an undirected graph, but in the following we assume instead that edges in E are directed. More precisely, the edges of G in a semi-matching M will be directed from a vertex $v \in V$ to a vertex $u \in U$ while the others will be directed from U to V. When switching matching and non-matching edges along path P we create a new semi-matching with the load on v_1 decreased by one, the load on v_k increased by one and the load of the other $v \in V$ unchanged. This derived semi-matching is denoted by $P \oplus M$, because it is formally equivalent to the symmetric difference of sets, i.e., $A \oplus B = (A \setminus B) \cup (B \setminus A)$.



(a) Edges in bold form P, an alternating path from Y to X, which are both right-hand vertices. Note that $\deg_M(Y) = 3$ while $\deg_M(X) = 1$.

(b) The result of switching matching and non-matching edges along the bold path is $P \oplus M$. Note how $\deg_M(Y)$ *decreases* to 2 and $\deg_M(X)$ *increases* to 2 as well, reducing the total cost of M.

(c) The resulting semi-matching is the one obtained from picking the edges that were oriented from right to left.

Figure 2.1: An example of alternating path.

Definition 2.5 (Cost-reducing Path)**.** An alternating path in which $\deg_M(v_1) > \deg_M(v_k) + 1$ is called a *cost-reducing path.*

Cost-reducing paths are so-called because the new semi-matching $P \oplus M$ will have cost $C(P \oplus M) = C(M) - (\deg_M(v_1) - \deg_M(v_k) - 1)$ which is less than the previous cost, $C(M)$. For example, the bold path in Figure 2.1(a) is a cost-reducing path. These type of paths represent a solid theoretical base of semi-matchings as we are going to show with the following theorem [4].

Theorem 2.1. *Let $G = (U \cup V, E)$ be a bipartite graph. A semi-matching M in G is optimal if and only if no cost-reducing path exists in it.*

Proof. It is clear that if M is optimal, no cost-reducing path can exist, otherwise M would not have been optimal because the load on its first vertex could be reduced by 1. Therefore we prove that a cost-reducing path must exist if M is not optimal.
Let O be an optimal semi-matching in G, chosen such that the symmetric difference $O \oplus M$ is minimized and suppose that M is not optimal. This implies that M has greater total cost than O, i.e., $C(O) < C(M)$. Let G_d be the subgraph induced by the edges of $O \oplus M$ and color with green the edges of $O \setminus M$ and with red the ones of $M \setminus O$. Direct the green edges from U to V and the red edges from V to U.
To complete the proof we first state and prove the following proposition.

Proposition 2.1.1. *The graph G_d is acyclic and for every directed path P in G_d from $v_1 \in V$ to $v_2 \in V$, $deg_O(v_2) \leq deg_O(v_1)$ holds.*

Proof. Let $P = (v_1, \ldots, v_2)$ be a directed path in G_d. By the choice of directions for the edges, P must be made of alternating red and green edges. If $\deg_O(v_2) > \deg_O(v_1) + 1$ then the reverse of P is a cost-reducing path for O, in contradiction to the optimality of O. Conversely, if $\deg_O(v_2) = \deg_O(v_1) + 1$, then we can get an optimal assignment with smaller symmetric difference from M by alternating the

assignment of the U-vertices along P, in contradiction to the minimality of $O \oplus M$. Therefore it must be that $\deg_O(v_2) \leq \deg_O(v_1)$.
A similar argument shows that G_d is also acyclic. ■

Recall that both O and M are semi-matchings, implying that $\sum\limits_{v \in V} \deg_O(v) = \sum\limits_{v \in V} \deg_M(v) = |U|$. Since $C(O) < C(M)$, there exists $v_1 \in V$ such that $\deg_M(v_1) > \deg_O(v_1)$. Starting from v_1, we can build an alternating red-green path P' as follows:

1. From an arbitrary vertex $v \in V$, if $\deg_{M \setminus O}(v) \geq 1$ and $\deg_M(v) \geq \deg_M(v_1) - 1$, we build P' by following an arbitrary red edge directed out from v.

2. From an arbitrary vertex $u \in U$, we build P' by following the single green edge directed out from u. Such a green edge must always exist, because O and M are semi-matchings.

3. Otherwise, stop.

By Proposition 2.1.1, G_d is acyclic and therefore P' is well defined and finite.
Now let $v_2 \in V$ be the final vertex on the path. We distinguish two cases: if $\deg_M(v_2) < \deg_M(v_1) - 1$ then P' is a cost-reducing path relative to M. If $\deg_{M \setminus O}(v_2) = 0$, we know that $\deg_M(v_2) < \deg_O(v_2)$ since P' arrived at v_2 by following a green edge. By Proposition 2.1.1 we also know that $\deg_O(v_2) \leq \deg_O(v_1)$. Recall that v_1 was chosen such that $\deg_O(v_1) < \deg_M(v_1)$. Combining these three inequalities yields $\deg_M(v_2) < \deg_O(v_2) \leq \deg_O(v_1) < \deg_M(v_1)$, implying that $\deg_M(v_2) < \deg_M(v_1) - 1$, hence P' is a cost-reducing path relative to M.
Since P' is a cost-reducing path relative to M in both cases, the proof is complete. □

So far we have considered the objective of minimizing only $C(M) = \sum\limits_{i=1}^{n} \text{cost}_M(v_i)$, where $\text{cost}_M(v) = \sum_{i=1}^{\deg_M(v)} i$ which is, in other words, the flow time of the tasks' schedule. We now prove that we can extend these results to a wider set of cost functions. First, we formally define a *cost function.*

Definition 2.6 (Cost Function). A cost function for a matching M is a function

$$\text{cost}_f(M) = \sum_{i=1}^{n} f(\deg_M(v_i))$$

where $f : \mathbb{R}^+ \to \mathbb{R}$ is a *strictly convex* function. If f is weakly convex, then $\text{cost}_f(M)$ is called a *weak cost function.*

For example, our $C(M)$ is the cost function induced by $f(x) = \sum\limits_{j=1}^{x} j = \frac{x(x+1)}{2}$.
We can now extend the concept of cost-reducing paths according to this definition.

Lemma 2.1. *Let P be a cost-reducing path. For any cost function $cost_f$, switching matching and non-matching edges along P yields a semi-matching $M' = P \oplus M$ such that $cost_f(M') < cost_f(M)$. If $cost_f$ is a weak cost function then $cost_f(M') \leq cost_f(M)$.*

Proof. First, we recall the following property of convex functions:

Proposition 2.1.2. *If f is strictly convex then $x > y \implies f(x+1) - f(x) > f(y+1) - f(y)$. If f is weakly convex then the inequality holds but is not strict.*

Proof. A property of strictly convex functions is that

$$\frac{f(b) - f(a)}{b - a} < \frac{f(c) - f(a)}{c - a} < \frac{f(c) - f(b')}{c - b'}$$

if $a < b < c$ and $a < b' < c$ [12]. Setting $a = y, b = y + 1, b' = x$ and $c = x + 1$ we obtain

$$f(y+1) - f(y) < \frac{f(x+1) - f(y)}{x + 1 - y} < f(x+1) - f(x)$$

If f is weakly convex then all the stated inequalities hold but are not strict. ■

Next, consider the matching M and the cost-reducing path $P = (v_1, \dots, v_k)$. Let $x = \deg_M(v_1) - 1$ and $y = \deg_M(v_k)$. The change in cost_f at v_1 due to switching edges along path P is $f(x) - f(x+1)$. Similarly, the change in cost_f at v_k is $f(y+1) - f(y)$, therefore

$$\mathrm{cost}_f(M') = \mathrm{cost}_f(M) - (f(x+1) - f(x)) + (f(y+1) - f(y))$$

applying Proposition 2.1.2, and the thesis follows. ☐

This result is very important because it provides an extension to Theorem 2.1, which can be reformulated as follows.

Theorem 2.2 (Optimal Semi-Matching). *Let M be a semi-matching in G. If cost_f is a strict cost function and if M is optimal with respect to cost_f, then no cost-reducing path relative to M exists. Conversely, if no cost-reducing path relative to M exists, then M is optimal with respect to cost_f.*

Proof. The proof is the same as the one for Theorem 2.1. ☐

This theorem implies a strong corollary that allows us to say that if M has no cost-reducing paths, then M is optimal with respect to *every* cost function.

Corollary 2.2.1. *A semi-matching M that is optimal with respect to any strict cost function is optimal with respect to every cost function, strict or weak.*

We are now ready to show that an optimal semi-matching minimizes, among many different objectives, the maximum load on any machine, thus satisfying the primary request of **MRS**.
Let $X = (x_1, \dots, x_n)$ denote the *load vector* of M, where $x_i = \deg_M(v_i)$. In the following theorem, we prove that an optimal semi-matching is also optimal with respect to every L_p-norm[1] of the vector X, for any finite p.

Theorem 2.3. *Let $1 < p < \infty$. A semi-matching is optimal if and only if it is optimal with respect to the L_p-norm of its load vector.*

Proof. Fix any $p > 1$ and define $f_p(x) = x^p$. Note that f_p is strictly convex, so cost_{f_p} is a strict cost function. Let X be the load vector for M. Since $\|X\|_p = \mathrm{cost}_{f_p}(M)^{1/p}$, M optimizes $\|X\|_p$ if and only if it optimizes cost_{f_p}. By Corollary 2.2.1, M is optimal with respect to cost_{f_p} if and only if M is an optimal semi-matching. ☐

Our primary objective for **MRS**, however, is to minimize the maximum load on every machine. This is achieved by minimizing the L_∞-norm[2] of the load vector X as we show below.

Proposition 2.3.1. *Let $f_\infty(x) = (n+1)^x$. A semi-matching that minimizes $cost_{f_\infty}$ minimizes the maximal load on a single V-vertex.*

Proof. Let M be a semi-matching that minimizes cost_{f_∞}. Assume, for the sake of contradiction, that M does not minimize the maximal load on a single V-vertex, and that this objective is achieved by a different semi-matching M'. Let Δ and Δ' be the maximal load on a single V-vertex in M and M' respectively. Thus, $\mathrm{cost}_{f_\infty}(M) \geq (n+1)^\Delta$ and $\mathrm{cost}_{f_\infty}(M') \leq n(n+1)^{\Delta'}$ (this is because, at worst, all the n V-vertices have load Δ'). However, since $\Delta' \leq \Delta - 1$, it follows that $\mathrm{cost}_{f_\infty}(M') < \mathrm{cost}_{f_\infty}(M)$, which is a contradiction. ☐

This claim leads us to conclude the following.

Theorem 2.4. *An optimal semi-matching M is also optimal with respect to the L_∞-norm.*

Proof. Since cost_{f_∞} is a cost function, Corollary 2.2.1 implies that M is optimal with respect to cost_{f_∞}. By Proposition 2.3.1, M minimizes the maximal load on a V-vertex and therefore M is optimal with respect to the L_∞-norm. ☐

---

[1] For any $p \in \mathbb{N}, p > 1$, the L_p-norm of vector $\mathbf{x} = (x_1, \dots, x_k)$ is defined as $\|\mathbf{x}\|_p = \left(\sum_{i=1}^{k} |x_i|^p\right)^{1/p}$.

[2] The L_∞-norm of vector $\mathbf{x}$ is defined as the maximum absolute value of its components.

Remark. The converse of Theorem 2.4 is not valid, i.e., minimizing the L_∞-norm does not imply minimization of other cost functions. This is because the converse of Proposition 2.3.1 does not hold: minimizing the makespan does not necessarily minimize cost_{f_∞}.

We now have all the theoretical background needed to describe the following two optimal algorithms.

2.2 $\mathcal{A}_{SM1}$: an $\mathcal{O}(mk)$ optimal algorithm

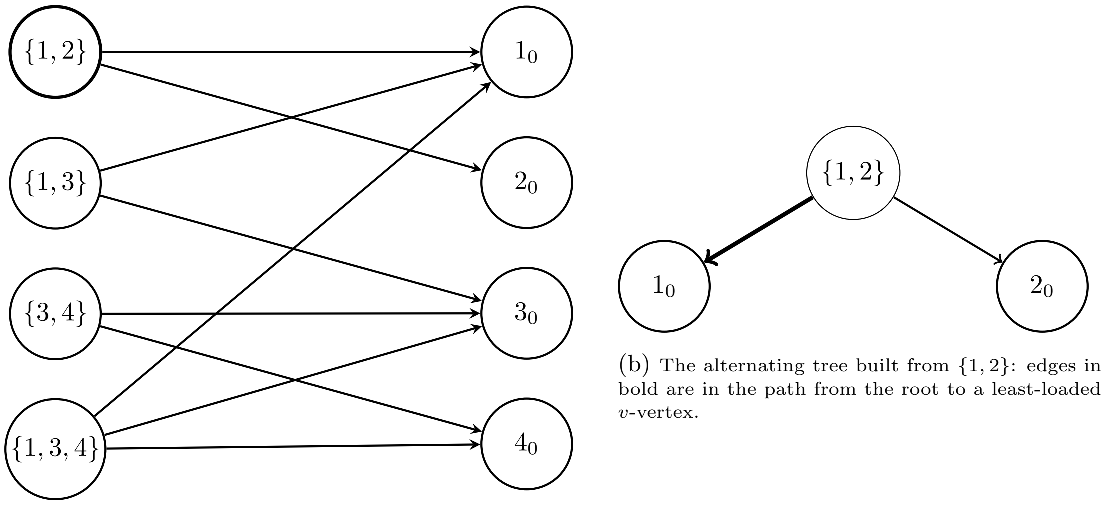
The first algorithm we describe is a modification of the Hungarian algorithm [13] for bipartite matching and it has a time complexity of $\mathcal{O}(|U|\,|E|)$ or, equally, $\mathcal{O}(mk)$ in terms of **MRS**.

Basically, the Hungarian algorithm for finding maximum bipartite matchings, considers each left-hand vertex u in turn and builds an *alternating search tree*, i.e. a tree of alternating paths, rooted at u. When an unmatched right-hand vertex v (that is a vertex $v \in V$ with $\deg_M(v) = 0$) is found, the matching and non-matching edges along the u-v path are switched so that u and v are no longer unmatched.

Similarly, this algorithm [4] maintains a partial semi-matching M starting with the empty set. In each iteration, it considers a vertex $u \in U$ and builds an alternating search tree from it using a directed breadth-first search. To perform this directed search we give an orientation to the edges in G according to M: the edges in M are directed from V to U and edges not in M are directed from U to V. We then select, in this tree, a path P from u to the v-vertex with the least load possible and we form $P \oplus M$ by switching the orientation of the edges in P. This will result in u being no longer unmatched and $\deg_M(v)$ increased by 1. We repeat this process for every $u \in U$ and since each iteration matches u with a single v-vertex and does not change $\deg_M(u)$ for any other $u \in U$, the resulting selection of edges is a semi-matching.

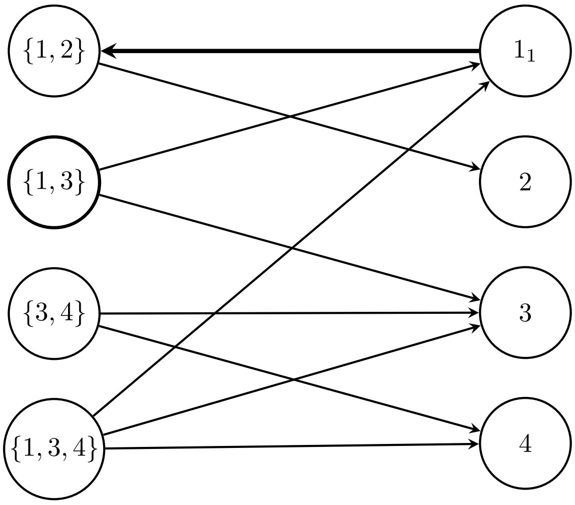
Before discussing the correctness and the time complexity of $\mathcal{A}_{\text{SM1}}$ we provide an example to visualize how the algorithm works. Recall that the input graph of this algorithm is directed.

Figure 2.2: Simulation of $\mathcal{A}_{\text{SM1}}$ on the instance $\langle \mathcal{N} = \{1,2,3,4\}, \mathcal{F} = \{\ \{1,2\}, \{1,3\}, \{3,4\}, \{1,3,4\}\ \}\rangle$. The numbers in subscript denote the *load* of the corresponding vertex.

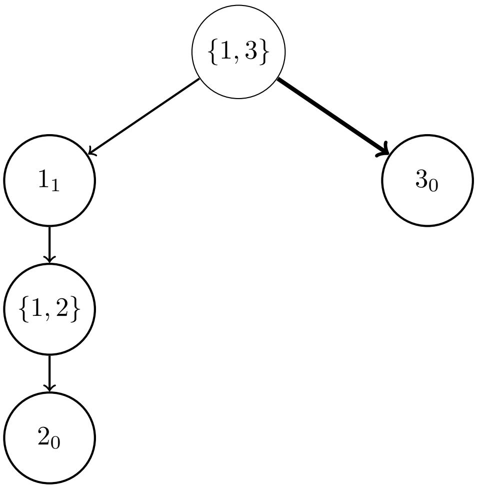


(a) The initial graph considered by $\mathcal{A}_{\text{SM1}}$.

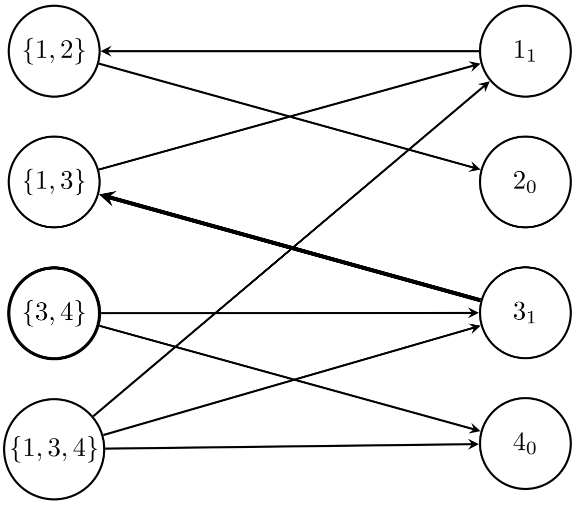
(b) The alternating tree built from $\{1,2\}$: edges in bold are in the path from the root to a least-loaded v-vertex.



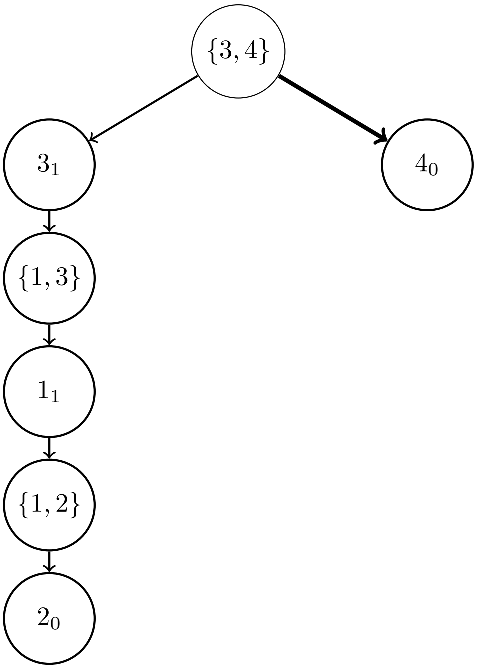
(c) The updated graph obtained by switching the orientation of the bold edges.



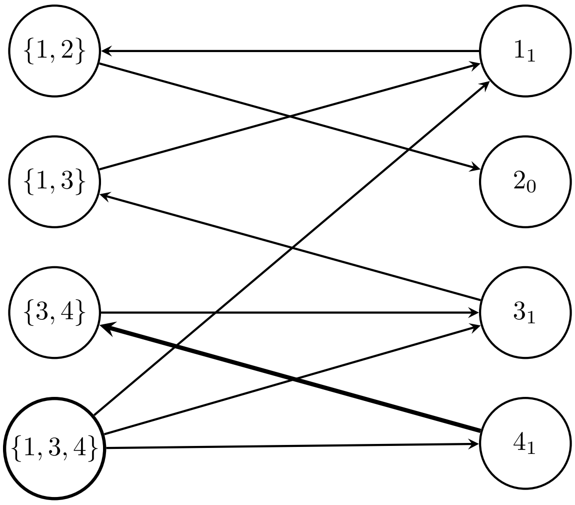
(d) The alternating tree built from $\{1,3\}$.



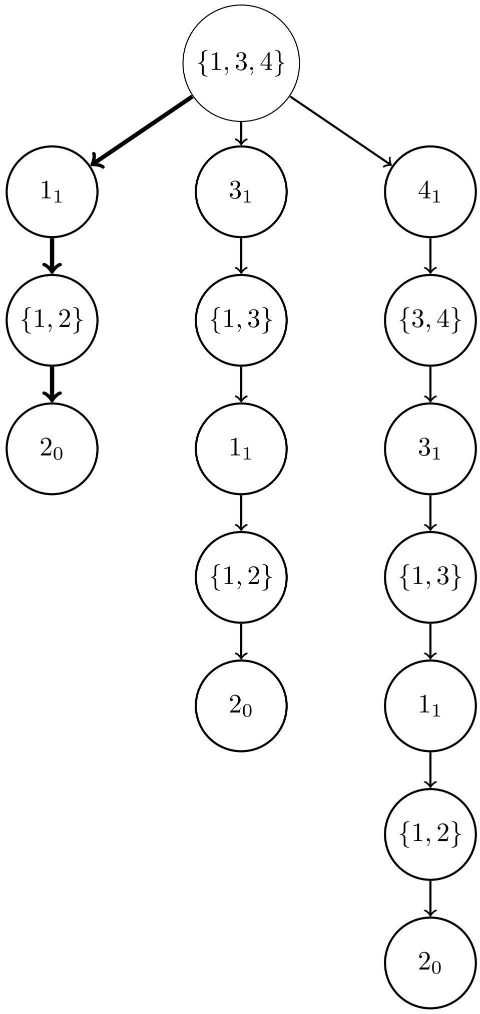
(e) The updated graph.



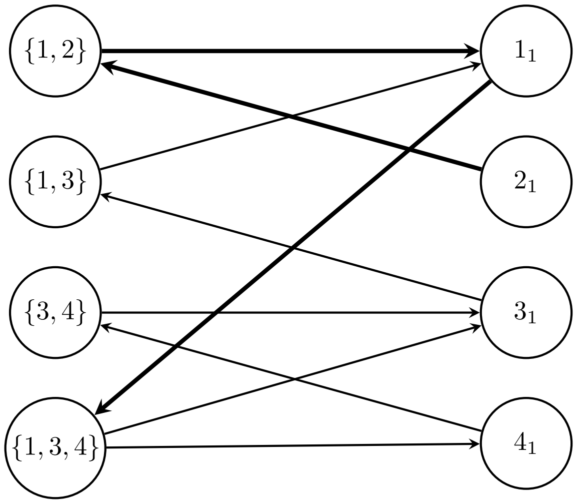
(f) The alternating tree built from $\{3,4\}$.



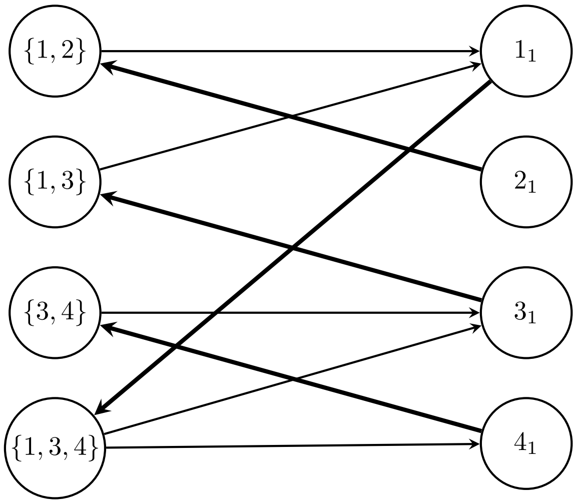
(g) The updated graph.



(h) The alternating tree built from $\{1, 3, 4\}$: note that now the least loaded v-vertex is 2.



(i) The updated graph.



(j) The result computed by $\mathcal{A}_{\mathrm{SM1}}$. Edges in M are the ones in bold, directed from right to left. The corresponding solution of **MRS** is then **2, 3, 4, 1**.

2.2.1 Correctness and complexity analysis

We now prove the correctness of $\mathcal{A}_{\text{SM1}}$ [4].

Theorem 2.5. *Algorithm $\mathcal{A}_{SM1}$ produces an optimal semi-matching.*

Proof. We show that no cost-reducing path is created during the execution of the algorithm. In particular, no cost-reducing path exists at the end of its execution. Thus, by Theorem 2.1, the resulting matching is optimal.

Assume the opposite, and let $P^* = (v_1, u_1, \ldots, v_{k-1}, u_{k-1}, v_k)$ be the *first* cost-reducing path created by the algorithm. Let M_1 be the matching after the iteration in which P^* is created. Thus, $\deg_{M_1}(v_1) > \deg_{M_1}(v_k) + 1$. Without loss of generality, by taking a sub-path of P^*, we can assume that there exists some x such that $\deg_{M_1}(v_1) \geq x+1$, $\deg_{M_1}(v_i) = x$, $\forall i \in \{2, \ldots, k-1\}$ and $\deg_{M_1}(v_k) \leq x-1$. Consider now the last iteration in which the load on v_1 is increased and let u' be the U-vertex added to the assignment at this iteration: by definition of the algorithm, v_1 is a least loaded V-vertex reachable from u'. Thus, the search tree built for u' includes only V-vertices with load at least x, in particular, v_k is not reachable from u'.

Given that the path P^* exists, at some iteration occurring after the one in which u' is added, all the edges (u_i, v_i) of P^* are in the matching. Let u^* be the U-vertex, added after u', whose addition to the assignment creates P^*. We show a contradiction to the way u^* is assigned. Specifically, we show that when adding u^*, the algorithm increases the load on some vertex with load at least x, while v_k, whose load at that time is at most $x-1$ is also reachable from u^*. To this aim, we first state and prove the following two propositions.

Proposition 2.5.1. *When u^* is added, the load on v_k is at most $x-1$ and v_k is in the alternating tree rooted at u^*.*

Proof. The load on any v-vertex can only increase during the algorithm. Since the load on v_k in M_1 is $x-1$ and u^* is added before P^* exists, $x-1$ is an upper bound on the load on v_k at the time u^* is added. To see that v_k is in T^* (the search tree built during the assignment of u^*) recall the assumption that the path P^* is created by the assignment of u^*. Let (u_i, v_i) be the last edge (i.e. the farthest from v_1) of path P^* that is added to the matching in this iteration. Thus, v_i is reachable from u^* and since the edge (u_j, v_j) is already in the matching $\forall j, i < j \leq k$, the suffix of P^* from v_i to v_k must also be in T^*. ■

Proposition 2.5.2. *When u^* is added, the load on some vertex with load at least x is increased.*

Proof. Suppose the opposite. We show that a cost-reducing path exists *before* u^* is added, contradicting our choice of P^* as the first cost-reducing path created by the algorithm. Once again, we use the fact that P^* is created when u^* is added. Let (u_i, v_i) be the first edge (i.e. the closest to v_1) of P^* which is added to the matching at this iteration. Thus, before adding u^*, the vertex v_i is reachable from v_1. Let $P' = (u^*, \ldots, v^*)$ be the path from u^* to the least loaded vertex in T^*: note that v_i must appear in P'. therefore the path from v_1 to v_i can be extended to reach v^* using the suffix of P' from v_i to v^*. All the edges of this suffix are available *before* u^* is added, since they were all available to T^*. The load on v^* must be at least x, otherwise the above path from v_1 to v^* is a cost-reducing path which exists before P^*. ■

Combining these two claims contradicts the execution of the algorithm and therefore P^* cannot exist. □

To bound the running time of $\mathcal{A}_{\text{SM1}}$ we first observe that there will be exactly $|U|$ iterations, or equally m iterations in terms of **MRS**. Each iteration requires at most $\mathcal{O}(|E|) = \mathcal{O}(k)$ time to build the alternating search tree and at most $\mathcal{O}(\min\{|U|, |V|\}) = \mathcal{O}(|E|) = \mathcal{O}(k)$ time to switch edges along the alternating path. Therefore the total running time of the algorithm is $\mathcal{O}(|U|\,|E|) = \mathcal{O}(mk) = \mathcal{O}(m^2 n)$.

2.3 $\mathcal{A}_{SM2}$: a faster-in-practice algorithm

The time bound of the previous algorithm is the same as the Hungarian's one, in fact, $\mathcal{A}_{\mathrm{SM1}}$ is just a variant of it. However, in practice, $\mathcal{A}_{\mathrm{SM1}}$ tends to be slower because of the big size of the alternating trees it builds in each iteration. For this reason, we describe another algorithm from [4] which has a theoretically *worst* time bound but that, in practice, performs much better.

Theorem 2.1 proved that a semi-matching is optimal if and only if the graph does not contain a cost-reducing path. $\mathcal{A}_{\mathrm{SM2}}$ uses that result to find an optimal semi-matching as follows:

1. Start with an initial semi-matching M;
2. While there exists a cost-reducing path P, reduce the cost of M by switching matching and non-matching edges along P.

Since each iteration reduces the total cost by an integral amount, the cost can only be reduced a finite number of times, therefore this algorithm must always terminate. Moreover, if the initial semi-matching is nearly optimal, this algorithm terminates after few iterations, thus, finding a good initial M is important and we start by describing this procedure. Note that $\mathcal{A}_{\mathrm{SM2}}$, however, does not rely on a good initial semi-matching and it can always find an optimal solution.

Finding an initial semi-matching. A simple approach would be to assign each left-hand vertex to a right-hand vertex arbitrarily, however, the following *greedy* algorithm have shown to perform well in practice.

First, the U-vertices are *sorted* by increasing degree. Each U-vertex is then considered in turn and assigned to a V-neighbor with least *load*. In case of a tie, a V-neighbor with least *degree* is chosen. The purpose of this strategy is to allow more constrained vertices (i.e. the ones with fewer neighbors) to "choose" their matching vertices first, leaving the vertices with more "freedom" (i.e. vertices with higher outdegree) available for the successive steps of the algorithm.

The total time required to find this initial matching is $\mathcal{O}(|E|) = \mathcal{O}(m)$, since every edge is examined exactly once, and the sorting can be done using a linear-time sorting algorithm such as counting sort or bucket sort.

Finding cost-reducing paths. The critical operation of the $\mathcal{A}_{\mathrm{SM2}}$ algorithm is the method to find cost-reducing paths. A simple approach to find an initial V-vertex of a cost-reducing path can be to grow a depth-first search tree of alternating paths rooted at v. To determine if G has any cost-reducing paths at all, it suffices to perform n depth-first searches, one for each $v \in V$. Unfortunately, this approach performs much redundant work. The following claim helps us to avoid this work.

Proposition 2.5.3. *Let $v, w \in V$ be two vertices such that $deg_M(v) \geq deg_M(w)$ and there is an alternating path from v to w. If there is no cost-reducing path starting from v, then there is no cost-reducing path starting from w.*

Proof. We prove the contrapositive. Since there is an alternating path from v to w, any cost-reducing path starting from w can be extended to a cost-reducing path starting from v. ☐

We can now proceed to find cost-reducing paths in G as follows. First we find a vertex $v \in V$ such that

(i) v has not been visited by any previous depth-first search;

(ii) its degree (in the matching M) is maximum among all such vertices.

To find such a vertex v quickly, we keep the non-visited V-vertices ordered by their load in an array of $|U| + 1$ *buckets.* Next, we build a depth-first search tree of alternating paths rooted at v in the usual manner. If this depth-first search tree doest not contain a cost-reducing path, then Proposition 2.5.3 allows us to conclude that there is no cost-reducing path starting from any of the vertices that were visited. Since this algorithm considers each edge in G at most once and there are only $|U| + 1$ buckets, it finds a cost-reducing path or lack thereof in $\mathcal{O}(|U| + |E|) = \mathcal{O}(|E|) = \mathcal{O}(k)$ time.

After finding a cost-reducing path, we can reduce the cost of the matching by switching matching and non-matching edges along the path. This step clearly takes $\mathcal{O}(|E|) = \mathcal{O}(k)$ time as well. Recall that this switching process only affects the load on the first and last vertices on the path.

2.3.1 Complexity Analysis

As argued earlier, the initial semi matching can be found in $\mathcal{O}(|E|) = \mathcal{O}(k)$ time. After this initial step, we iteratively find and remove cost-reducing paths. Finding one takes $\mathcal{O}(|E|) = \mathcal{O}(k)$ time, while switching matching and non-matching edges along it requires $\mathcal{O}(\min\{|U|, |V|\}) = \mathcal{O}(|E|) = \mathcal{O}(k)$ time. Thus, the runtime of $\mathcal{A}_{\text{SM2}}$ is $\mathcal{O}(I \cdot |E|) = \mathcal{O}(I \cdot k)$, where I is the number of iterations needed to achieve optimality, so it remains to determine I. Note that a simple bound of $I = |U|^2$ may be obtained by observing that the worst possible initial matching has total cost at most $\mathcal{O}(|U|^2)$ and that each iteration reduces the cost by at least 1.
The following theorem derives, instead, an improved bound of $\mathcal{O}(\min\{|U|^{3/2}, |U||V|\}) = \mathcal{O}(\min\{m^{3/2}, mn\})$. However, in real instances of **MRS** it is very difficult for this minimum to be $m^{3/2}$, since m is in general much greater than $n = |V|$ and can even be exponential in n. This is why we present a restricted version of the theorem, while the complete one can be found in [4].

Theorem 2.6. *$\mathcal{A}_{SM2}$ requires at most $\mathcal{O}(|U||V|)$ iterations to achieve optimality.*

Proof. For a given initial semi-matching M_0, assume that the V-vertices are sorted by load in non-increasing order (i.e. $\deg_{M_0}(v_1) \geq \cdots \geq \deg_{M_0}(v_m)$). In each iteration, the algorithm identifies a cost-reducing path $P = (v_a, \ldots, v_b)$. By switching matching and non-matching edges along this path we remove one unit of load from v_a and add one unit to v_b. This load balancing process can be described as follows: initially we have n towers of blocks, where the ith tower consists of $\deg_{M_0}(v_i)$ blocks. Reducing the cost using path P amounts to move one block from the tower corresponding to v_a to the tower corresponding to v_b. We can bound the number of iterations of $\mathcal{A}_{\text{SM2}}$ by bounding the total number of possible block moves. The proof is based on the following properties:

1. Since we always move load units to a less loaded vertex, operations may be ordered such that the V-vertices remain sorted in non-increasing order of their loads throughout the execution of the algorithm. In other words, the towers are always sorted from highest to lowest.
2. By the definition of cost-reducing paths, when moving a load unit from v_a to v_b, the load on v_a before the iteration is *strictly* greater than the load on v_b after the iteration.

For any value of n, the total number of moves is at most nm since by the above properties each block may be moved at most n times. ☐

Therefore we can state that the total running time of $\mathcal{A}_{\text{SM2}}$ is $\mathcal{O}(|U||V||E|) = \mathcal{O}(mnk) = \mathcal{O}(m^2n^2)$ which is worse than the previous algorithm. However, the greedy initialization in practice performs so well that often significantly fewer iterations are required to achieve optimality, making this algorithm the preferred choice. Note that it is possible to create "bad" input graphs, in which the number of iterations needed is $\Omega(m^{3/2})$, but most cost-reducing paths in these type of instances are very short, thus each iteration takes roughly constant time.

Chapter 3

Heuristics

In this chapter we discuss three heuristics for **MRS**. It may seem quite odd to develop heuristics for a problem that we can solve always *exactly*, however, these three heuristics, and in particular the first two, are much faster than the optimal algorithms described in Chapter 2. For this reason, it may be a good choice to trade an optimal result with a significant reduction of computational time since, as we argued earlier, the size of m in real instances of **MRS** can be very large. Another reason to develop good heuristics for **MRS**, lies in the $\mathcal{A}_{\mathrm{SM2}}$ initialization step, as it is clearly an initialization with a heuristic. The better this heuristic, the lower iterations of the algorithm needed.

The first two heuristics share many points in common and they are based on a greedy approach. The third heuristic uses instead a completely different, more numeric approach, in a way that will be discussed later in this chapter. From our experiments, we can state that all of the three heuristics have a very high percentage of success, compared with the optimal algorithms.

In the pseudocode of the heuristics we assume to have a field called *representative* in each $S \in \mathcal{F}$ as the way to associate the representative to its subset, and also a field *cardinality*, already set, that keeps the cardinality of each subset. In this chapter we distinguish between the *occurrences* and the *frequency* of the variables, which are the number of appearances throughout the input instance and the number of times a variable has been chosen as representative, respectively.

3.1 Pure Greedy Heuristic

The first and the most basic strategy we present is a simple "pure" greedy algorithm that always chooses the variable that, so far, has been chosen the least amount of times during its process, hence its name. The algorithm breaks ties by selecting the first variable of the tie, but as a general rule we can say that we choose an arbitrary variable. We keep the sets of $\mathcal{F}$ sorted by their cardinality in ascending order to have more freedom with the choice of variables as the algorithm proceeds.

This heuristic runs in $\mathcal{O}(k)$ time, therefore it is linear in the size of the input. We provide the pseudocode and the complexity analysis.

Algorithm 2 (Pure Greedy) Always selects the least-chosen variable.

```
function GREEDY($Sets$)
    SORT($Sets$)                          ▷ according to their cardinality in ascending order
    $freq \leftarrow$ new array of length $n$
    for $i \leftarrow 1$ to $n$ do
        $freq[i] \leftarrow 0$
    for all $S \in Sets$ do
        $min \leftarrow \infty$
        $repr \leftarrow -1$
        for all $v \in S$ do
            if $freq[v] < min$ then
                $repr \leftarrow v$
                $min \leftarrow freq[v]$
        $S.representative \leftarrow repr$
        $freq[repr]$++
```

Consider, for example, the same input instance as the one of Section 1.1,

$$\left\langle \mathcal{N} = \{1,2,3,4,5\}, \mathcal{F} = \left\{ \{1,2\}, \{1,3\}, \{2,3\}, \{2,4\}, \{4,5\} \right\} \right\rangle .$$

The Pure Greedy algorithm produces the following selection of representatives: $1, 3, 2, 4, 5$ (the order is the same of the subsets in the input), which is an optimal solution with maximum frequency of 1. However, with the following instance things get worse.

$$\left\langle \mathcal{N} = \{1,2,3,4\}, \mathcal{F} = \left\{ \{1,2\}, \{1,3\}, \{3,4\}, \{1,3,4\} \right\} \right\rangle$$

The algorithm produces the result $1, 3, 4, 1$, with a value of 2. This is clearly not an optimal solution, because the optimal value for this instance is 1 and is achieved, for example, by the choice of $2, 1, 3, 4$. Specifically, the heuristic here fails because of the tie-breaking mechanism that chose 1 in the first subset instead of 2, and it is not a problem of the implementation. Suppose that we break ties by choosing the *second* variable in the tie instead of the first: it is clear that this instance will be correctly solved. But if we swap the first and the second subset of the input, we get back with the same problem[1]. This happens because the algorithm *is not aware* of what the subsequent sets will be like, and blindly proceeds throughout them.

Theorem 3.1. *The running time of the pure greedy heuristic is $\mathcal{O}(k) = \mathcal{O}(mn)$.*

Proof. We can sort the sets in $\mathcal{O}(m + n)$ time using a linear time sorting algorithm. This is possible because we sort the sets according to their cardinality, thus the sorting merely becomes a sorting of integers bounded by the value of n, which is the highest cardinality possible for each subset in $\mathcal{F}$. The second `for` loop is just a complete scan of the input, therefore it takes $\mathcal{O}(k)$ time. The total running time is then $\mathcal{O}(m + n + k) = \mathcal{O}(k) = \mathcal{O}(mn)$. ☐

---

[1] Here we are supposing that the linear-time sorting algorithm is also stable. Such an algorithm is, for example, *Counting Sort*.

As a practical consideration, we can say that often $k << mn$, as the mn limit is reached only when *all* of the m subsets are of cardinality n, which is an extreme situation that almost never appears in real instances of **MRS**.

3.2 Forward Greedy Heuristic

We developed this heuristic with the problem of *blindness* that affects the pure greedy one in mind, thus we decided to make the algorithm follow the opposite direction or, in other words, to focus primarily on the future occurrences of the variables and not only on the past frequencies. For this reason, we say that this heuristic uses a "forward greedy" approach. The sets are kept sorted by their cardinality as in the previous heuristic for the same reason. More formally, the selection strategy of this algorithm is as follows:

1. Among all the variables in the current subset that have not been chosen more than $\lceil \frac{m}{n} \rceil$ times, choose the one with lowest number of occurrences across all the (remaining) subsets. If such a variable does not exists, choose a variable using the pure greedy strategy;
2. In case of ties, select the variable that has been chosen the lowest number of times, i.e. the variable with the least frequency (as in the pure greedy heuristic);
3. In case of successive ties, select a variable arbitrarily.

As before, we provide the pseudocode of this heuristic and we discuss its time complexity.

Algorithm 3 (Forward Greedy)

```
 1: function ForwardGreedy($Sets$)
 2:     Sort($Sets$)                              ▷ according to their cardinality in ascending order
 3:     $freq \leftarrow$ new array of length $n$
 4:     $occ \leftarrow$ new array of length $n$             ▷ contains the occurrences of each variable
 5:     for $i \leftarrow 1$ to $n$ do
 6:         $freq[i] \leftarrow 0$
 7:     for all $S \in Sets$ do
 8:         for all $v \in S$ do
 9:             $occ[v]$++                             ▷ count the occurrences of each variable
10:     for all $S \in Sets$ do
11:         $candidate_f \leftarrow$ the first variable in $S$ with the lowest $freq$ value
12:         $candidate_o \leftarrow$ the first variable in $S$ with the lowest value of $occ$
13:         if $freq[candidate_o] < \lceil \frac{m}{n} \rceil$ then   ▷ if we already chose $candidate_o$ less than $\lceil \frac{m}{n} \rceil$ times
14:             $S.representative \leftarrow candidate_o$
15:         else
16:             $S.representative \leftarrow candidate_f$
17:
18:         for all $v \in S$ do
19:             $occ[v] --$                    ▷ remove the occurrences of all the variables contained in $S$
20:
21:         $freq[S.representative]$++
```

The reason of the check at line 13 is based on Proposition 1.2.1: in an ideal solution, its value would be $\lceil \frac{m}{n} \rceil$ and so we try to not exceed this mean when choosing a variable. Note that we use this lower bound as a basic approach because the possibly-higher lower bound provided in Theorem 1.3 takes exponential time to be calculated.

As an example, consider the second input instance provided in the previous Section:

$$\langle \mathcal{N} = \{1, 2, 3, 4\}, \mathcal{F} = \{\ \{1, 2\}, \{1, 3\}, \{3, 4\}, \{1, 3, 4\}\ \} \rangle .$$

This time, the heuristic gives the solution $2, 1, 3, 4$ with value 1, which is optimal. Note that the result does not depend on the order of the subsets in the input. However, this is not true in general.

Theorem 3.2. *The running time of the forward greedy heuristic is* $\mathcal{O}(k) = \mathcal{O}(mn)$.

Proof. We can sort the sets using the same linear-time sorting algorithm as in the pure greedy heuristic, at the cost of $\mathcal{O}(m + n)$ time. The first `for all` loop is a scan of the input and it takes $\mathcal{O}(k)$ time. Finding the variable with lowest frequency and the variable with lowest occurrences can be done with a scan of the current S, which takes $\mathcal{O}(n)$ time, and the same argument applies to the last `for all` loop. Therefore, the `for all` loop starting at line `10` takes $\mathcal{O}(k)$ time. Therefore the total running time is $\mathcal{O}(m + n + k) = \mathcal{O}(k) = \mathcal{O}(mn)$. □

We finally note that the initialization step of $\mathcal{A}_{\mathrm{SM2}}$ is a hybrid between our pure greedy heuristic and this one, because it chooses according only to *frequencies* (referred as the *load* in $\mathcal{A}_{\mathrm{SM2}}$), but breaks ties by choosing the variable with the least *occurrences* (previously referred as the *degree*).

3.3 Pagerank inspired Heuristic

The last heuristic we describe uses a completely different strategy to build a solution, and it is a more numerical one compared with the two previous heuristics.

Unlike any other previous algorithm (including optimal ones), this algorithm creates a solution by iteratively *removing* variables from the subsets, i.e., at each iteration it excludes a variable from a subset. At the end of its execution there will be only one variable left in each subset and they will be chosen as representatives. For this reason, we chose the name “Pagerank” for the heuristic, according to the nature of the Markov Matrix used by Google, that is obtained by successive transformations of the initial hyperlinking matrix [14].

We build a $m \times n$ matrix W (called the *relative weight matrix* or *weight matrix*) with variables on the columns and subsets on the rows and we populate it with the uniform probability that we have, at any given time, to choose the variable j from the *i-th* subset in $\mathcal{F}$.

Definition 3.1 (Weight Matrix)**.** The *weight matrix* is a $m \times n$ matrix, W, with

$$W_{i,j} = \begin{cases} \frac{1}{|\mathcal{F}[i]|} & \text{if } j \in \mathcal{F}[i] \\ 0 & \text{otherwise} \end{cases} .$$

We define the *total weight* (sometimes only *weight* in the following) of a variable as the sum over its column, that is, $weight(j) = \sum_{i=1}^{m} W_{i,j}$. Our goal is to minimize the weight of each variable as much as possible, in particular the maximum one. The algorithm proceeds as follows:

1. Choose the variable with maximum weight, and look for a subset that contains it with its highest relative weight (without considering the “1”s). In case of ties on the maximum weight, select one arbitrarily, ignoring the variables that have only “1”s in their column. In case of ties among the subsets, select one arbitrarily;

2. Exclude the variable from the chosen subset by setting its relative weight to 0;

3. Update the relative weights of the variables in the chosen subset, having its cardinality *logically* decreased by one;

4. Repeat this procedure until there is only one variable left in each subset, i.e., in every row of the matrix there are $n - 1$ zeros and one “1”.

Tables 3.1 and 3.2 show an example of iteration of this algorithm.

Subset	**1**	**2**	**3**	**4**
$\{1,2\}$	0.5	0.5	—	—
$\{1,3\}$	0.5	—	0.5	—
$\{3,4\}$	—	—	0.5	0.5
$\{1,3,4\}$	0.33	—	0.33	0.33
Total Weights	1.33	0.5	1.33	0.83

Table 3.1: The initial matrix built by the algorithm for the input instance used in the previous sections, $\langle \mathcal{N} = \{1,2,3,4\}, \mathcal{F} = \{\{1,2\}, \{1,3\}, \{3,4\}, \{1,3,4\}\}\rangle$. The circled numbers denote the maximum relative and total weight (with ties broken arbitrarily).

Subset	**1**	**2**	**3**	**4**
$\{1,2\}$	—	1	—	—
$\{1,3\}$	0.5	—	0.5	—
$\{3,4\}$	—	—	0.5	0.5
$\{1,3,4\}$	0.33	—	0.33	0.33
Total Weights	0.83	1	1.33	0.83

Table 3.2: The result of zeroing the variable “1” from the first subset. The new maximum relative and total weights are circled.

Intuitively, prior to the iteration on Table 3.1, we had a probability of $\frac{1}{2}$ of picking either 1 or 2 from the first subset. After that iteration, the variable 1 *certainly* will not be chosen as representative, with 2 being picked instead (because it has a probability of 1).
We now provide the pseudocode for this heuristic and we discuss its time complexity.

Algorithm 4 (Pagerank)

```
function Pagerank($Sets$)
    $W \leftarrow$ new $m \times n$ matrix
    $total\_weights \leftarrow$ new array of length $n$
    for all $S \in Sets$ do                              ▷ Initialization of the matrix
        for all $v \in S$ do
            $W[S][v] \leftarrow \frac{1}{|S|}$
            $total\_weights[v] \leftarrow total\_weights[v] + W[S][v]$
    for $i \leftarrow 1$ to $k - m$ do
        $v \leftarrow$ the variable with maximum $total\_weight$
        $S \leftarrow$ a subset in which $v$ appears with highest relative weight
        $total\_weights[v] \leftarrow total\_weights[v] - W[S][v]$
        $W[S][v] \leftarrow 0$
        Decrease the (logical) cardinality of $S$ by 1
        for all $x \in S \setminus \{v\}$ do
            $W[S][x] \leftarrow W[S][x] + \frac{1}{|S|(|S|+1)}$          ▷ With $|S|$ being the logical cardinality of $S$
            $total\_weights[v] \leftarrow total\_weights[v] + \frac{1}{|S|(|S|+1)}$
    for all $S \in Sets$ do
        $S.representative \leftarrow$ the only variable of $S$ with $W[S][variable] = 1$
```

Theorem 3.3. *The running time of the pagerank heuristic is* $\mathcal{O}(mn^2 \log n)$.

Proof. We use n balanced binary search trees (we use AVL trees in our implementation) to keep track of which variable is in which subset with a particular relative weight. Specifically, for each variable in $\mathcal{N}$ we keep a pointer to the root of its AVL, with its nodes consisting of:

- A key, which is one of the possible $n-2$ relative weights (we do not keep track of the weights 0 and 1);
- A hash table, H, of length m;
- An array, A, of length m;
- The current size of the array A (and therefore the index of the first empty slot in A), namely $A.size$.

The satellite data keep track of which subsets contain the variable represented by the AVL. Note that a list, a hash table or an array alone do not allow us to make insertion, deletions, and search of elements in constant time, a result that can be instead achieved with a pair of a hash table and an array using the following techniques:

```
function INSERT($set$)
    $A[A.size] \leftarrow set$
    $H[set] \leftarrow A.size$
    $A.size{+}{+}$
```

```
function REMOVE($set$)
    $d \leftarrow A[A.size - 1]$
    $i \leftarrow H[set]$
    $A[i] \leftarrow d$
    $H[d] \leftarrow i$
    $H[set] \leftarrow$ NIL
    $A.size - -$
```

The number of iterations of the algorithm is fixed to be *exactly* $k-m$, since we zero a variable from a subset at each iteration, leaving one variable in every subset, thus zeroing exactly $k-m$ variables in total.

At each iteration, we can find the maximum total weight in constant time by keeping it updated every time we modify a relative weight. Finding a subset in which the variable to be zeroed appears with its highest relative weight can be done in $\mathcal{O}(n \log n)$ time in the following way:

(i) In the corresponding AVL, get the node with the maximum key (requires $\mathcal{O}(\log n)$ time);

(ii) If that node contains at least one element, i.e., if $A.size > 0$, grab an element (for example the first, or also a random one) from the node (requires $\mathcal{O}(1)$ time);

(iii) If that node does not contain any elements, repeat the process with the predecessor node (requires at most n searches of a predecessor, each of them takes $\mathcal{O}(\log n)$ time $\implies \mathcal{O}(n \log n)$ total time).

Zeroing the variable from the subset found above takes $\mathcal{O}(1)$ time, and updating the relative weight of the other variables in that subset (and their total weights) requires $\mathcal{O}(n)$ time. To update the corresponding AVLs we may simply remove and re-insert the subsets' index in them, requiring $\mathcal{O}(\log n)$ total time. Therefore, each iteration takes $\mathcal{O}(n \log n)$ time, and since there will be $k-m$, or equally $\mathcal{O}(k)$, iterations we can conclude that the algorithm runs in $\mathcal{O}(kn \log n) = \mathcal{O}(mn \cdot n \log n) = \mathcal{O}(mn^2 \log n)$ time. □

The time bound just obtained is higher than the ones for the previous heuristics. However, thanks to the nature of **MRS**, we know that almost always $n << m$, therefore this time bound will be practically much lower than the one of $\mathcal{A}_{\text{SM1}}$, which is $\mathcal{O}(m^2 n)$.

A refinement for this heuristic could be to not break ties arbitrarily when selecting the subset with the variable of highest relative weight, but rather to choose the subset that contains that variable *and* the variable of *lowest* total weight. This is because the weights of the variables will denote, at the end of the algorithm, the number of times they have been chosen. Zeroing a variable with high total weight reflects the attempt to minimize the maximum frequency, and doing so from a subset that contains also a variable much "lighter" will increase its total weight, which is the least dangerous at the moment. If such a subset does not exist, look for the *second* lowest total weight and so on. However, we were not able to stay under the $\mathcal{O}(m^2n)$ time limit with this refinement, and in this case the algorithms described in Chapter 2 are a superior choice.

Chapter 4

Experimental Results

In this chapter we provide some statistics about the practical execution time of all the algorithms presented in the previous chapters along with statistics on the quality of the three heuristics, obtained by comparing their result to the one of the optimal algorithms. To represent the instances of **MRS** as close to the reality as possible (i.e., to use numbers that can actually be involved during the process of synthesis on switching lattices of Boolean functions), we tested the algorithms under the assumption that $m >> n$. For example, for every value of n, we will generate instances with, $m = \Theta(n^2)$, $\Theta(n^3)$ and, when possible, $m = 2^{n-1}$.

The test cases can be divided into two large groups, based on the probability distribution of the variables across the subsets: we chose a Normal distribution and an Uniform distribution. Roughly speaking, one can expect that the instances generated with the uniform probability will be harder than the ones generated with the Gaussian one, because each variable will tend to appear the same amount of times across the subsets, making any kind of simplification (like the one described in Corollary 1.3.1) nearly impossible. The cardinality of the subsets always follows the uniform distribution. Table 4.1 and 4.2 are organized as follows:

1. The first column contains the pair $\langle n, m\rangle$. For each of these pairs, we generate 100 independent instances;
2. The second column contains the average k, which corresponds to the average size of the input across the 100 instances. More formally, $k_{avg} = \frac{\sum\limits_{i=1}^{100}\sum\limits_{j=1}^{m}|\mathcal{F}_i[j]|}{100}$;
3. The remaining columns represent the average time spent by each algorithm to compute the solution. Like before, it is obtained by averaging the time spent for each of the 100 input instances;
4. The second table shows the same kind of data for input instances generated with the uniform distribution.

$\langle \mathbf{n}, \mathbf{m}\rangle$	$\mathbf{k_{avg}}$	$\mathcal{A}_{\mathbf{SM1}}$	$\mathcal{A}_{\mathbf{SM2}}$	**Greedy**	**GreedyFWD**	**Pagerank**
$\langle 10, 100\rangle$	515	$0.005371s$	$0.000065s$	$0.000017s$	$0.000024s$	$0.000309s$
$\langle 10, 512\rangle$	2756	$0.142710s$	$0.000292s$	$0.000070s$	$0.000113s$	$0.001240s$
$\langle 20, 400\rangle$	4198	$0.174935s$	$0.000367s$	$0.000088s$	$0.000164s$	$0.004596s$
$\langle 10, 1000\rangle$	5464	$0.578575s$	$0.000484s$	$0.000124s$	$0.000194s$	$0.002187s$
$\langle 30, 900\rangle$	13958	$1.947167s$	$0.000936s$	$0.000184s$	$0.000400s$	$0.031149s$
$\langle 50, 2500\rangle$	63809	$24.948925s$	$0.003540s$	$0.000683s$	$0.001695s$	$0.272875s$
$\langle 20, 8000\rangle$	83976	$103.82581s$	$0.005931s$	$0.001307s$	$0.002416s$	$0.107823s$
$\langle 30, 27000\rangle$	418418	$1725.06s$	$0.024287s$	$0.004805s$	$0.009786s$	$0.830832s$
$\langle 50, 125000\rangle$	3188839	†	$0.162482s$	$0.033316s$	$0.076941s$	$15.668205s$
$\langle 20, 2^{19}\rangle$	5504172	†	$0.371158s$	$0.088621s$	$0.168090s$	$8.491898s$

Table 4.1: Average execution times over 100 randomly-generated instances with the Gaussian distribution. Results are marked with a '†' when an algorithm is stopped after 18 hours of computation.

$\langle \mathbf{n}, \mathbf{m} \rangle$	$\mathbf{k_{avg}}$	$\mathcal{A}_{\mathbf{SM1}}$	$\mathcal{A}_{\mathbf{SM2}}$	**Greedy**	**GreedyFWD**	**Pagerank**
$\langle 10, 100 \rangle$	552	0.005762s	0.000072s	0.000019s	0.000028s	0.000350s
$\langle 10, 512 \rangle$	2819	0.161303s	0.000289s	0.000057s	0.000109s	0.001537s
$\langle 20, 400 \rangle$	4198	0.178702s	0.000313s	0.000065s	0.000148s	0.004927s
$\langle 10, 1000 \rangle$	5491	0.684278s	0.000540s	0.000134s	0.000220s	0.002989s
$\langle 30, 900 \rangle$	13949	2.005043s	0.000825s	0.000173s	0.000431s	0.029007s
$\langle 50, 2500 \rangle$	63739	23.337943s	0.002764s	0.000581s	0.001639s	0.253003
$\langle 20, 8000 \rangle$	83962	87.022895s	0.004335s	0.001061s	0.002184s	0.095918s
$\langle 30, 27000 \rangle$	418429	†	0.020709s	0.004799s	0.011405s	0.908683s
$\langle 50, 125000 \rangle$	3187990	†	0.134957s	0.029164s	0.079760s	14.569035s
$\langle 20, 2^{19} \rangle$	5504567	†	0.375886s	0.096140s	0.194809s	9.670038s

Table 4.2: Average execution times over 100 randomly-generated instances with the Uniform distribution. Results are marked with a '†' when an algorithm is not executed.

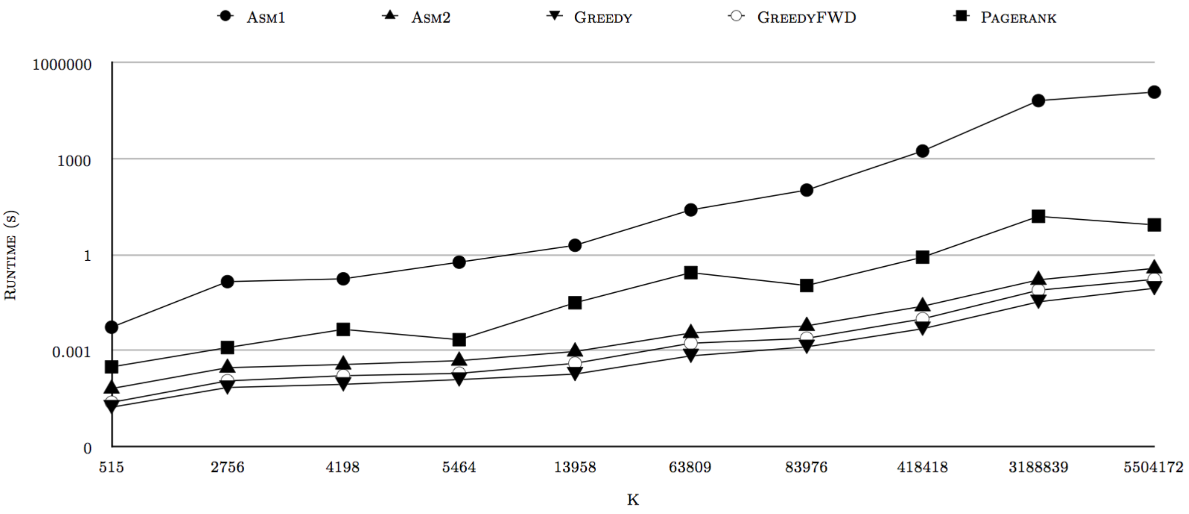


Figure 4.1: Visual comparison of the running times obtained with the Gaussian distribution. The graph obtained with the Uniform distribution is analogous.

These results show how $\mathcal{A}_{\mathrm{SM1}}$ is practically unfeasible for real instances of **MRS**, since its running time increases drastically even for relatively small increases of the input size, while the other algorithms have no problems in handling even a huge amount of sets in practice, such as 2^{19}. We can note that the average k, i.e., the average size of each input instance generated, is roughly equal to $mn/2$. This is important for the linear-time heuristics, since their running time is practically halved with respect to the $\mathcal{O}(mn)$ time bound. Another thing to note is that for a major part of Table 4.2 the running times are higher than the ones of Table 4.1, practically confirming our hypothesis that the instances generated with a uniform distribution are generally harder.

We also tested the success rate of our heuristics, comparing their result with the one obtained by $\mathcal{A}_{\mathrm{SM2}}$. In the table below we provide the percentage of *failures* of the heuristics, along with their average deviation from the optimal value. We wanted also to test the quality of the $\mathcal{A}_{\mathrm{SM2}}$'s initialization heuristic, to compare it against ours. The test cases have the same m and n of the tables before, but we also added a few more, smaller, ones.

$\langle \mathbf{n}, \mathbf{m} \rangle$	$\mathbf{k_{avg}}$	**Greedy**	**GreedyFWD**	**Pagerank**	$\mathcal{A}_{\text{SM2}}$ **heuristic**
$\langle 15, 20 \rangle$	157	—	—	—	—
$\langle 10, 30 \rangle$	162	—	—	—	—
$\langle 10, 50 \rangle$	266	1 (+1)	—	—	—
$\langle 20, 50 \rangle$	505	—	—	—	—
$\langle 10, 100 \rangle$	545	—	—	—	—
$\langle 50, 100 \rangle$	2543	10 (+1)	—	4 (+1)	—
$\langle 10, 512 \rangle$	2752	—	—	—	—
$\langle 20, 400 \rangle$	4203	—	—	—	—
$\langle 100, 100 \rangle$	5060	19 (+1)	—	4(+1)	—
$\langle 10, 1000 \rangle$	5459	—	—	—	—
$\langle 30, 900 \rangle$	13927	—	—	—	—
$\langle 50, 2500 \rangle$	63876	—	—	—	—
$\langle 20, 8000 \rangle$	83930	—	—	—	—
$\langle 30, 27000 \rangle$	418305	—	—	—	—
$\langle 50, 125000 \rangle$	3188873	—	—	—	—
$\langle 20, 2^{19} \rangle$	5504229	—	—	—	—

Table 4.3: Average errors and average deviation from the optimal value of the heuristics over 100 randomly-generated instances with the Gaussian distribution.

$\langle \mathbf{n}, \mathbf{m} \rangle$	$\mathbf{k_{avg}}$	**Greedy**	**GreedyFWD**	**Pagerank**	$\mathcal{A}_{\text{SM2}}$ **heuristic**
$\langle 15, 20 \rangle$	158	1 (+1)	—	—	—
$\langle 10, 30 \rangle$	158	—	—	—	—
$\langle 10, 50 \rangle$	262	1 (+1)	—	1 (+1)	—
$\langle 20, 50 \rangle$	515	—	—	—	—
$\langle 10, 100 \rangle$	516	—	—	—	—
$\langle 50, 100 \rangle$	2525	7 (+1)	—	5 (+1)	—
$\langle 10, 512 \rangle$	2785	—	—	—	—
$\langle 20, 400 \rangle$	4213	—	—	—	—
$\langle 100, 100 \rangle$	4975	3 (+1)	—	—	—
$\langle 10, 1000 \rangle$	5467	—	—	—	—
$\langle 30, 900 \rangle$	13994	—	—	—	—
$\langle 50, 2500 \rangle$	63894	—	—	—	—
$\langle 20, 8000 \rangle$	83993	—	—	—	—
$\langle 30, 27000 \rangle$	418457	—	—	—	—
$\langle 50, 125000 \rangle$	3188750	—	—	—	—
$\langle 20, 2^{19} \rangle$	5504227	—	—	—	—

Table 4.4: Average errors and average deviation from the optimal value of the heuristics over 100 randomly-generated instances with the Uniform distribution.

As we can see from the tables above, every heuristic performs very well in general with only few errors for relatively small input instances (i.e. for relatively small values of k). As the size of the input grows, errors will practically tend to zero. In particular, our Greedy Forward heuristic and the heuristic used for the initialization of $\mathcal{A}_{\text{SM2}}$ show a 100% success rate which is also extremely useful for the running time of the latter algorithm. In fact, the running time of $\mathcal{A}_{\text{SM2}}$ is, in practice, twice as the running time of the Greedy Forward heuristic which is in turn very similar to the $\mathcal{A}_{\text{SM2}}$'s. This is because checking if the solution obtained by the heuristic does not contain a cost-reducing path is done in $\mathcal{O}(|E|) = \mathcal{O}(k)$ time (which corresponds to an iteration of the algorithm) and the two heuristics run in $\mathcal{O}(k)$ time as well. However, we were able to find some input instances that make the latter heuristic failing to catch the optimal solution. On of these counterexamples is the following:

$$\langle \mathcal{N} = \{1,2,3,4\}, \mathcal{F} = \{\, \{1,2\}, \{1,3\}, \{1,4\}, \{3,4\} \,\} \rangle$$

The $\mathcal{A}_{\mathrm{SM2}}$'s initialization heuristic provides a solution of $2,3,4,3$ (having a value of 2). This happens because the heuristic does not update the degree of the v-vertex it creates during its execution, i.e., does not decrease the number of *future occurrences* of the variables once a subset has ben processed. This is in contrast to our Greedy Forward heuristic which, thanks to this updates of the future occurrences, correctly gives the solution $2,1,4,3$ (with value 1). However, the tables above show that the $\mathcal{A}_{\mathrm{SM2}}$'s initialization heuristic runs in this kind of errors only for very small values of n and m, in practice.

Chapter 5

Conclusions and Future Work

In the last years, switching lattices have seen renewed interest especially thanks to the increased use of nanoscale technologies in building new electronic *chips* and our work addresses one of the problems that one encounters when developing one such device.

We have seen how our optimization problem can be solved by reducing it to, at least, three different types of problems, each of them known to be polynomially solvable. We chose to focus primarily on the scheduling area, but the other two areas can be explored as well. In fact, it is also possible to generalize the graph orientation problem to hypergraphs[1], which represent an immediate translation of the instances of **MRS**, just by putting $V = \mathcal{N}$ and $E = \mathcal{F}$. **MRS** can then be solved by directing the hyperedges of the hypergraph (see, for example, [15] to have an idea on how a hypergraph can be directed). The time bound for solving this problem using the network flow reduction, which we recall it to be $\mathcal{O}(m^3 \log m)$, is loose. Specifically, the m^3 term comes from the complexity of the algorithm used for solving the maximum flow problem and therefore, the total running time can be reduced by using a more efficient algorithm for solving that problem.

The first of the two optimal algorithm that we described works in the "good old fashion" of constructing a solution starting from an empty one, while the second one improves a starting, possibly already optimal, solution. This shows how "informed" algorithm can beat a theoretically-faster algorithm in practice, and also gives a reason to develop good heuristics. We believe that the Pagerank heuristic, in particular, can serve as a basis for creating a new optimal algorithm that works using a totally different and more numerical approach. For example, one can provide the algorithm with some form of autocorrection, like the one implicitly performed by $\mathcal{A}_{\mathrm{SM1}}$ at the moment of following a path on the alternating search tree.

All of the heuristics we presented can also find application in areas where the approach *has to be* heuristical, thanks to the reductions we have shown in Chapter 1. For example, having a weight on our subsets is equivalent to have a different execution time for each job in a scheduling environment. Moreover, weights on the subsets become weights on the edges of a graph that has to be directed, like in the MMO problem, which we recall it to be the problem of finding an orientation for all edges of an undirected graph such that the maximum outdegree (or equivalently, indegree) is minimum. These problems, when weighted with weights greater than 1, become $\mathcal{NP}$-hard, and require heuristic or approximation algorithms.

Again in the context of switching lattices, it can also be a good idea to assign the same variable to a group of neighboring cells to simplify the resulting architecture (and perhaps meeting some practical needs such as cost reductions), and this can be reflected in **MRS**, for example, by putting weights on the variables instead of on the sets, and one can investigate on how all the algorithms proposed here react to weighted instances.

---

[1] An hypergraph $G = (V, E)$ is a graph in which the edges are formed by subsets of V of any cardinality, not just pairs of vertices.

Bibliography

[1] S. B. Akers, "A rectangular logic array", *IEEE Transactions on Computers*, vol. 21, no. 8, pp. 848–857, 1972.

[2] M. Altun and M. D. Riedel, "Logic synthesis for switching lattices", *IEEE Transactions on Computers*, vol. 61, no. 11, pp. 1588–1600, 2012.

[3] G. Gange, H. Søndergaard, and P. J. Stuckey, "Synthesizing optimal switching lattices", *ACM Transactions on Desing Automation of Electronic Systems*, vol. 20, pp. 1–14, nov 2014.

[4] N. J. Harvey, R. E. Ladner, L. Lovász, and T. Tamir, "Semi-matchings for bipartite graphs and load balancing", *Journal of Algorithms*, pp. 53–78, 2006.

[5] Y. Asahiro, E. Miyano, H. Ono, and K. Zenmyo, "Graph orientation algorithms to minimize the maximum outdegree", *International Journal of Foundations of Computer Science*, pp. 197–215, 2010.

[6] Y. Lin and W. Li, "Parallel machine scheduling of machine-dependent jobs with unit-length", *European Journal of Operational Research*, pp. 261–266, 2004.

[7] V. Venkateswaran, "Minimizing maximum indegree", *Discrete Applied Mathematics*, vol. 143, pp. 374–378, 2004.

[8] E. Mokotoff, "Parallel machine scheduling problems: A survey", *Asia-Pacific Journal of Operational Research*, no. 18, pp. 193–242, 2001.

[9] T. C. E. Cheng and C. C. S. Sin, "A state-of-the-art review of parallel machine scheduling research", *European Journal of Operational Research*, no. 47, pp. 271–292, 1990.

[10] B. Chen, C. Potts, and G. Woeginger, "A review of machine scheduling: Complexity, algorithms and approximability", in *Handbook of Combinatorial Optimization*, pp. 21–169, Kluwer Academic Publishers, 1998.

[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms.* The MIT Press, 3rd ed., 2009.

[12] W. Rudin, *Principles of Mathematical Analysis.* McGraw-Hill, 3rd ed., 1976.

[13] H. W. Kuhn, "The hungarian method for the assignment problem", *Naval Research Logistics Quarterly*, pp. 83–97, 1955.

[14] A. N. Langville and C. D. Meyer, "Deeper inside pagerank", *Internet Mathematics*, vol. 1, no. 3, pp. 335–380, 2003.

[15] G. Gallo, G. Longo, S. Pallottino, and S. Nguyen, "Directed hypergraphs and applications", *Discrete Applied Mathematics*, vol. 42, pp. 177–201, 1993.

Appendices

Appendix A

Code Listing

We first provide the code for the header files, and then the code for the source ones.

Listing A.1: types.h

```c
#include <stdio.h>

#ifndef types_h
#define types_h

typedef struct {
  int* elems;    // Elements
  int card;      // Cardinality
  int repr;      // Representative
  int id;        // ID (for sorting)
} set_t;

typedef enum {
  NOEDGE = 0, // No edge between to vertices
  LTR = 1,    // Edge from a left-hand vertex to a right-hand one (u->v)
  RTL = -1    // Edge from a right-hand vertex to a left-hand one (v->u)
} adj_t;

/* Utility procedure to print an array of set_t */
static inline void print_sets(set_t* sets, int dim) {
  for(int i=0;i<dim;i++) {
    printf("{ ");
    for(int j=0;j<sets[i].card;j++)
      printf("%i ", sets[i].elems[j]+1);
    printf("} [id=%i,card=%i]\n",sets[i].id,sets[i].card);
  }
  printf("\n");
}

/* Utility procedure to print one set_t */
static inline void print_set(set_t set) {
  printf("{ ");
  for(int i=0;i<set.card;i++)
    printf("%i ", set.elems[i]+1);
  printf("}");
}

/* Utility functions used for counting_sort */
static inline int cardinality(set_t set) {
  return set.card;
}

static inline int id(set_t set) {
  return set.id-1; // -1 is necessary for the counting sort to work
}

static inline int representative(set_t set) {
  return (set.repr != -1) ? set.repr : 0;
}

#endif /* types_h */
```

Listing A.2: rng.h

```
#ifndef rng_h
#define rng_h

#define RNG_UNIF 0
#define RNG_GAUSS 1
#define RNG_EXPO 2

#define STD_DEV 3.5 // Standard Deviation for the Normal Distribution

/* lambda for the exponential distribution */
int my_rand(int type, int min, int max, double lambda);

#endif /* rng_h */
```

Listing A.3: queue.h

```
#ifndef queue_h
#define queue_h

#include <stdlib.h>

/* Integer Queue */

typedef struct queue_elem {
  int elem;
  struct queue_elem *next;
} queue_elem_t;

typedef struct {
  queue_elem_t *head;
  queue_elem_t *tail;
  int count;
} queue_t;

static inline queue_t* new_queue() {
  queue_t *q = malloc(sizeof(*q));
  if(q == NULL) return NULL;

  q->head = NULL;
  q->tail = NULL;
  q->count = 0;

  return q;
}

static inline void enqueue(queue_t* q, int elem) {
  //printf("enqueue %i\n", elem);
  if(q == NULL) return;

  queue_elem_t *new = malloc(sizeof(queue_elem_t));
  if(new == NULL) return;

  new->elem = elem;
  new->next = NULL;

  if(q->head == NULL) { // Empty queue
    q->head = new;
    q->tail = new;
  }
  else {
    q->tail->next = new;
    q->tail = new;
  }
  (q->count)++;
}

static inline int dequeue(queue_t* q) {
  //printf("Dequeue ");
  if(q == NULL) return -1;

  if(q->head == NULL) return -1; // Empty queue

  queue_elem_t* old_head = q->head;
  int to_return = q->head->elem;

  q->head = q->head->next;
  free(old_head); // Free memory of the old node
  (q->count)--;
  //printf("%i\n", to_return);

  return to_return;
}

static inline void destroy_queue(queue_t* q) {
  while(q->head != NULL) {
    queue_elem_t* old = q->head;
    q->head = q->head->next;
    free(old);
  }
  free(q);
}
#endif /* queue_h */
```

Listing A.4: intlist.h

```
#ifndef intlist_h
#define intlist_h

#include <stdio.h>
#include <stdlib.h>

typedef struct elem {
  int val;
  struct elem * next;
} elem_t;

typedef struct {
  elem_t * head;
  int length;
} intlist_t;

static inline intlist_t* new_list() {
  intlist_t* new = malloc(sizeof(intlist_t)); // Placeholder
  if(new == NULL) return NULL;
  new->head = NULL;
  new->length = 0;
  return new;
}

static inline int extract_head(intlist_t *list) {
  if(list == NULL || list->head == NULL) return -1;
  elem_t *tmp = list->head;
  int to_return = tmp->val;

  list->head = list->head->next;
  free(tmp);

  (list->length)--;

  return to_return;
}

static inline void insert(intlist_t *list, int elem) {
  if(list == NULL) return;

  elem_t *new = malloc(sizeof *new);
  if(new == NULL) return;
  new->val = elem;
  new->next = NULL;

  if(list->head == NULL) {
    list->head = new;
  }
  else {
    elem_t *curr = list->head;
    while(curr->next != NULL) {
      curr = curr->next;
    }
    curr->next = new;
  }

  (list->length)++;
}

static inline int is_empty(intlist_t *list) {
  return (list->head == NULL) ? 1 : 0;
}

```

```
static inline void remove_elem(intlist_t *list, int elem) {
  if(list == NULL || list->head == NULL) return;
  if(list->head->next == NULL && list->head->val == elem) {
    free(list->head);
    list->head = NULL;
    return;
  }
  elem_t *prev = NULL, *curr = list->head;
  while(curr != NULL && curr->val != elem) {
    prev = curr;
    curr = curr->next;
  }
  if(curr != NULL) {
    if(prev == NULL) {
      elem_t *tmp = list->head;
      list->head = list->head->next;
      free(tmp);
    }
    else {
      prev->next = curr->next;
      free(curr);
    }
  }

  (list->length)--;
}

static inline int in_list(intlist_t *list, int elem) {
  if(list == NULL || list->head == NULL) return 0;
  elem_t* curr = list->head;
  while(curr != NULL) {
    if(curr->val == elem)
      return 1;
    curr = curr->next;
  }
  return 0;
}

static inline void destroy_list(intlist_t *list) {
  while(list->head != NULL) {
    elem_t* tmp = list->head;
    list->head = list->head->next;
    free(tmp);
  }
  free(list);
}

#endif /* intlist_h */
```

Listing A.5: countingsort.h

```
#ifndef countingsort_h
#define countingsort_h

#include <stdio.h>

#include "types.h"

set_t* counting_sort(set_t* sets, int dim, int k, int (*key)(set_t));

int* integer_counting_sort(int* a, int dim, int max);

#endif /* countingsort_h */
```

Listing A.6: sm.h

```
/**
 * Algorithms for SEMI-MATCHINGS
 *
 */

#ifndef sm_h
#define sm_h

#include "types.h"

typedef enum {
  WHITE,
  GREY,
  BLACK
} color_t;

void asm1(set_t* sets, int m, int n);

set_t* asm2(set_t** sets, int m, int n);

#endif /* sm_h */
```

Listing A.7: pagerank.h

```
#ifndef pagerank_h
#define pagerank_h

#include "types.h"

set_t* pagerank_heuristic(set_t* sets, int m, int n, int k); // K = sum of the cardinality

#endif /* pagerank_h */
```

Listing A.8: avlnode.h

```
#ifndef avlnode_h
#define avlnode_h

#include "rng.h"

/* Nodes of an AVL */
typedef struct avl_node_s {
  struct avl_node_s *left;
  struct avl_node_s *right;
  struct avl_node_s *parent;
  double key;
  struct node_data_s {
    int size; // Size and last_position of a
    int *h; // Hash
    int *a; // Array
  } data;
} avl_node_t;

/* Gets a random element from the array a of the field "data" */
static inline int avl_node_get_random_element(avl_node_t *node) {
  return node->data.a[my_rand(RNG_UNIF, 0, node->data.size-1, 1)];
}

/* Adds an element to a node */
static inline void avl_node_insert_element(avl_node_t *node, int elem) {
  node->data.a[node->data.size] = elem;
  node->data.h[elem] = node->data.size;
  (node->data.size)++;
}

/* Tells if elem is in node */
static inline int avl_node_contains(avl_node_t* node, int elem) {
  return (node->data.h[elem] != -1) ? 1 : 0;
}

/* Removes an element from a node */
static inline void avl_node_remove_element(avl_node_t *node, int elem) {
  int d = node->data.a[(node->data.size)-1]; // Size-1 = last non-empty position
  int i = node->data.h[elem]; // Index in the array of the element to be removed
  node->data.a[i] = d; // The last element of the array
  node->data.h[d] = i; // Update the index of element d
  node->data.h[elem] = -1; // Remove the element form the hash table
  (node->data.size)--;
}


#endif /* avlnode_h */
```

Listing A.9: avl.h

```
#ifndef avl_h
#define avl_h

#include "types.h"
```

```
#include "avlnode.h"

/* AVL */
typedef struct avl_tree_s {
  avl_node_t *root;
} avl_tree_t;


/* Creates a new AVL tree. */
avl_tree_t *avl_create(void);

/* Insert a new node. */
void avl_insert(avl_tree_t *tree, double key, int element, int arraysize);

/* Find the node containing the given key */
avl_node_t *avl_find(avl_tree_t *tree, double key);

/* Finds the first node (in decreasing order) with data.size > 0 */
avl_node_t *avl_find_nonempty_maximum(avl_tree_t *tree);

/* Destroys an AVL */
void avl_destroy(avl_tree_t *tree);

#endif /* avl_h */
```

Listing A.10: greedy.h

```
#ifndef greedy_h
#define greedy_h

#include "types.h"

set_t* pure_greedy_heuristic(set_t** sets, int m, int n);

set_t* forward_greedy_heuristic(set_t** sets, int m, int n);

set_t* asm2_heuristic(set_t** sets, int m, int n);

#endif /* greedy_h */
```

Listing A.11: rng.c

```
#include <math.h>
#include <float.h>
#include <stdlib.h>

#include "rng.h"

/* Generatore di numeri casuali normal-distributed */
static double generateGaussianNoise(double mu, double sigma) {
  const double epsilon = DBL_MIN;
  const double two_pi = 2.0*3.14159265358979323846;

  static double z0, z1;
  static int generate = 0;
  generate = !generate;

  if (!generate)
    return z1 * sigma + mu;

  double u1, u2;
  do
  {
    u1 = rand() * (1.0 / RAND_MAX);
    u2 = rand() * (1.0 / RAND_MAX);
  }
  while ( u1 <= epsilon );

  z0 = sqrt(-2.0 * log(u1)) * cos(two_pi * u2);
  z1 = sqrt(-2.0 * log(u1)) * sin(two_pi * u2);
  return z0 * sigma + mu;
}

int my_rand(int type, int min, int max, double lambda) {
  switch(type) {
    case RNG_UNIF: {
      double gen = rand() / (RAND_MAX + 1.0);
      gen *= (max - min + 1);
      gen += min;
      return (int) gen;
    }
    case RNG_GAUSS: {
      int gen;
            while((gen = (int) generateGaussianNoise(min + (max - min) / 2.0, (max - min) / 2.0 / STD_DEV)) > max || gen < min);
      return gen;
    }
    case RNG_EXPO: {
      double gen = rand() / (RAND_MAX + 1.0);
      double num = (-log(gen) / lambda);
      while(num < min || num > max) {
        gen = rand() / (RAND_MAX + 1.0);
        num = (-log(gen) / lambda);
      }
      return (int) floor(num);
    }
    default:
      return 0; // Not yet implemented
  }
}
```

Listing A.12: countingsort.c

```
#include <stdlib.h>

#include "countingsort.h"

set_t* counting_sort(set_t* sets, int dim, int k, int (*key)(set_t)) {
  int i;
  int *C = calloc(k+1, sizeof(int));
  if(C == NULL) return NULL;
  set_t *B = malloc(dim * sizeof *B);
  if(B == NULL) return NULL;

  for(i=0;i<dim;i++) {
    C[key(sets[i])]++;
  }

  for(i=1;i<=k;i++) {
    C[i] = C[i-1] + C[i];
  }

  for(i=dim-1;i>=0;i--) {
    B[C[key(sets[i])]-1] = sets[i]; // -1 because B starts from 0
    C[key(sets[i])]--;
  }

  free(C);

  return B;
}

int* integer_counting_sort(int* a, int dim, int max) {
  int i;
  int *C = calloc(max+1, sizeof *C);
  if(C == NULL) return NULL;
  int *B = malloc(dim * sizeof *B);
  if(B == NULL) return NULL;

  for(i=0;i<dim;i++) {
    C[a[i]]++;
  }

  for(i=1;i<=max;i++) {
    C[i] = C[i-1] + C[i];
  }

  for(i=dim-1;i>=0;i--) {
    B[C[a[i]]-1] = a[i];
    C[a[i]]--;
  }

  free(C);

  return B;

}
```

Listing A.13: asm1.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

#include "sm.h"
#include "queue.h"
#include "intlist.h"
#include "countingsort.h"

/* BFS on the graph adj, from root, with BF-Tree built in parents */
static int bfs(adj_t** adj, int root, int* parents, color_t* colors, int m, int n) {
  int i;
  intlist_t* neighbors;
  queue_t *q = new_queue();

  for(i=0;i<m+n;i++)
    colors[i] = WHITE;
  enqueue(q, root);
  colors[root] = GREY;

  int bestV = -1;
  int degM_bestV = INT_MAX;

  while(q->count != 0) {
    int w = (dequeue(q));

    neighbors = new_list();

    if(w < m) { // It's a U vertex (left-hand)
      for(i=0;i<n;i++) {
        if(adj[w][i] == LTR) { // i is an unmatched neighbor
          insert(neighbors, i+m); // V-vertices are denoted by their matrix-index + m
        }
      }
    }
    else { // It's a V vertex (right-hand)
      int degM_w = 0;
      for(i=0;i<m;i++) {
        if(adj[i][w-m] == RTL) { // i is a matched neighbor
          insert(neighbors, i); // U-vertices are denoted just by their matrix-index
          degM_w++;
        }
      }
      if(bestV == -1 ||degM_w < degM_bestV) {
        bestV = w-m;
        // Update the degM_bestV
        degM_bestV = 0;
        for(i=0;i<m;i++) {
          if(adj[i][bestV] == RTL) {
            degM_bestV++;
          }
        }
      }
    }
    int next = extract_head(neighbors);
    while(next != -1) { // Scan the entire list
      if(colors[next] == WHITE) {
        parents[next] = w;
        enqueue(q, next);
        colors[next] = GREY;
      }
      next = extract_head(neighbors);
    }

    destroy_list(neighbors);
    colors[w] = BLACK;
  }

  destroy_queue(q);
  return bestV + m; // Return the best V-vertex found from this root
}

/* Asm1 as described in Semi-Matchings paper */
void asm1(set_t* sets, int m, int n) {
    if(m >= 25000) return;
  int i, j;
  /* First step: build the graph via its incidence matrix */
  adj_t** adj = malloc(m * sizeof(adj_t*)); // Allocate the matrix

  if(adj == NULL) return;
  for(i=0;i<m;i++) { // Theta(k)
    adj[i] = calloc(n, sizeof(adj_t)); // Allocate the rows of the matrix - zeroed to not waste time
    for(j=0;j<sets[i].card;j++) { // Add the edges to the graph based on the sets
      adj[i][sets[i].elems[j]] = LTR; // Edge from the set to the variable
    }
  }

  int* parents = malloc((m+n) * sizeof(int));
  if(parents == NULL) return;
  color_t* vert_colors = malloc((m+n) * sizeof *vert_colors);
  if(vert_colors == NULL) return;

  for(i=0;i<m;i++) { // m BFS
    int v = bfs(adj, i, parents, vert_colors, m, n); // Will be > m for sure
    int u = parents[v]; // Will be < m for sure
    // Add {u, v} to the matching M - done by setting the direction of edge {u, v} to RTL.
    adj[u][v-m] = RTL;
    // Switch edges along path from u to v
    while(u != i) { // u != root
      v = parents[u]; // > m
      adj[u][v-m] = LTR; // Remove {u, v} from matching M
      u = parents[v];
      adj[u][v-m] = RTL; // Add {u, v} to matching M
    }
  }

  for(i=0;i<m;i++) { // Set the final representative for each set
    for(j=0;j<n;j++) {
      if(adj[i][j] == RTL) // Should be just one, and it will be the representative
        sets[i].repr = j; // The VARIABLE-1, not its position
    }
  }

  /* Release all the memory */
  free(parents);
  free(vert_colors);
  for(i=0;i<m;i++)
    free(adj[i]);
  free(adj);
}
```

Listing A.14: asm2.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

#include "sm.h"
#include "queue.h"
#include "intlist.h"
#include "countingsort.h"

int DFS(adj_t** adj, int vert, int original_root, intlist_t* S, int* load, int m, int n, int* parents, int* start) {
  int i;

  for(i=0;i<m;i++) {
    if(adj[i][vert] == RTL) { // For each u in MatchedNeighbors(vert) -- u is our i
      parents[i] = vert+m; // vert is in v

      for(int w=0;w<n;w++) {
        if(adj[i][w] == LTR) { // For each w in UnmatchedNeighbors(i)
          if(!in_list(S, w) || load[w] > load[vert]) { //degM(adj, w, m, n) > degM(adj, vert, m, n)) { // w already visited or load on w worse than load on v
            continue; // skip to next w
          }
          remove_elem(S, w);
          parents[w+m] = i; // Because w is in V
          if(load[w] <= load[original_root] - 2) { // From the original root!
            *start = w; // Start of the path
            return 1;
          }
          DFS(adj, w, original_root, S, load, m, n, parents, start);
        }
      }
    }
  }
  return 0;
}

int find_cost_reducing_path(adj_t** adj, int* load, int m, int n, int* parents, int* root_to_return) {
  int i;
  intlist_t *not_visited = new_list();
  /* At the beginning, not_visited = V */
  for(i=0;i<n;i++)
    insert(not_visited, i);

  while(!is_empty(not_visited)) {
    /* Look for a V-vertex with max degM(v) */
    int max_degM = -1;
    int root = -1;
    elem_t* curr = not_visited->head;
    while(curr != NULL) {
      // Calcolo il deg_M del vertice curr
      int degM_v = load[curr->val]; //degM(adj, curr->val, m, n);

      if(degM_v > max_degM) {
        max_degM = degM_v;
        root = curr->val;
      }
      curr = curr->next;
    }
    /* The V-vertex chosen is in root */

    remove_elem(not_visited, root); // Remove root from not_visited
    parents[root+m] = -1; // Because parents are all together, v indices > m
    int start_path;

    if(DFS(adj, root, root, not_visited, load, m, n, parents, &start_path) == 1) {
      *root_to_return = root; // Path from start_path to root
      destroy_list(not_visited);
      return start_path;
    }
  }
  destroy_list(not_visited);

  return -1; // No cost-reducing path exists
}

/* Asm2 as described in Semi-Matchings paper */
set_t* asm2(set_t** sets, int m, int n) {
  int i, j;
  /* First step: sort the U-vertices (the sets) by increasing degree (cardinality) */
  set_t* sorted_sets = counting_sort(*sets, m, n, cardinality); // Sort the U vertices by increasing degree

  /* Build the graph via its incidence matrix */
  adj_t** adj = malloc(m * sizeof(adj_t*)); // Allocate the matrix
  int* deg = calloc(n, sizeof *deg); // Allocate the deg array that will contain the degree of every V-vertex
  int* degM = calloc(n, sizeof *deg); // Allocate the degM array that will contain the degree of every V-vertex in the matching M
  if(adj == NULL) return NULL;
  if(deg == NULL) return NULL;
  if(degM == NULL) return NULL;

  for(i=0;i<m;i++) { // Theta(k)
    adj[i] = calloc(n, sizeof(adj_t)); // Allocate the rows of the matrix - zeroed to not waste time
    for(j=0;j<sorted_sets[i].card;j++) { // Add the edges to the graph based on the sets
      adj[i][sorted_sets[i].elems[j]] = LTR; // Edge from the set to the variable
      deg[sorted_sets[i].elems[j]]++; // Count the degree (occurrences) of this variable
    }
  }

  /* GREEDY INITIALIZATION */
  int *load = calloc(n, sizeof *load); // The load is equal to the degree of a V-vertex in the matching M
  if(load == NULL) return NULL;
  /* Assign each U-vertex with its least-loaded V-neighbor (in case of ties, choose the V-neighbor with minimum degree) */
  for(i=0;i<m;i++) {
    int load_min = INT_MAX;
    int deg_min = INT_MAX;
    int v_min = 0;
    for(j=0;j<sorted_sets[i].card;j++) { // Search the least-loaded V-neighbor
      if(load[sorted_sets[i].elems[j]] < load_min) {
        load_min = load[sorted_sets[i].elems[j]];
        v_min = sorted_sets[i].elems[j];
      }
    }

    for(j=0;j<sorted_sets[i].card;j++) { // Break ties
      if(load[sorted_sets[i].elems[j]] == load_min && deg[sorted_sets[i].elems[j]] < deg_min) {
        deg_min = deg[sorted_sets[i].elems[j]];
        v_min = sorted_sets[i].elems[j];
      }
    }

    /* v_min is the least-loaded/least-degree V-neighbor of i */
    adj[i][v_min] = RTL; // Add {u, v} to matching M
    load[v_min]++; // And thus increase the load of v_min
  }
  /* END: GREEDY INITIALIZATION */

  /* Next: Find and remove cost-reducing paths */
  int* parents = malloc((m+n) * sizeof(int));
  if(parents == NULL) return NULL;
  int root, start;

  while((start = find_cost_reducing_path(adj, load, m, n, parents, &root)) != -1) { // While there exists a cost-reducing path
    /* Switch matching and non-matching edges along P */
    // The cost-reducing path is from start to root

    // Switch edges along cost-reducing path from start to root
    int v = start; // < m
    int u = parents[v+m];
    while(v-m != root) { // u != root (root < m)
      adj[u][v] = RTL; // Add {v, u} to matching M
      load[v]++; // Increase the load of this v

      v = parents[u]; // > m

      adj[u][v-m] = LTR; // Remove {u, v} from matching M
      load[v-m]--; // Decrease the load of this v

      u = parents[v];
    }
    for(i=0;i<m+n;i++) // Reset the parents
      parents[i] = -1;
  }


  for(i=0;i<m;i++) { // Set the final representative for each set
    for(j=0;j<n;j++) {
      if(adj[i][j] == RTL) // Should be just one, and it will be the representative
        sorted_sets[i].repr = j; // The VARIABLE-1, not its position
    }
  }

  set_t* sets_to_return = counting_sort(sorted_sets, m, m+1, id);

  /* Release all the memory */
  for(i=0;i<m;i++) {
    free(adj[i]);
  }
  free(adj);
  free(deg);
  free(degM);
  free(load);
  free(parents);
  free(*sets);
  free(sorted_sets);

  return sets_to_return;
}
```

Listing A.15: avl.c

```c
#define _XOPEN_SOURCE 500 /* Enable certain library functions (strdup) on linux. See feature_test_macros(7) */

#include <time.h>
#include <stdlib.h>
#include <stdio.h>
```

```
#include <limits.h>
#include <string.h>
#include <assert.h>

#include "avl.h"

/* Creates a new AVL tree. */
avl_tree_t *avl_create() {
  avl_tree_t *tree = NULL;

  if((tree = malloc(sizeof( avl_tree_t ))) == NULL)
    return NULL;

  tree->root = NULL;

  return tree;
}

/* Initialize a new node. */
static avl_node_t *avl_create_node(double key, int arraysize) {
  avl_node_t *node = NULL;

  if( ( node = malloc( sizeof( avl_node_t ) ) ) == NULL ) {
    return NULL;
  }

  node->left = NULL;
  node->right = NULL;
  node->parent = NULL;
  node->key = key;
  node->data.size = 0;
  node->data.a = malloc(arraysize * sizeof(int));
  if(node->data.a == NULL)
    return NULL;
  node->data.h = malloc(arraysize * sizeof(int));
  if(node->data.h == NULL)
    return NULL;

  return node;
}

/* Find the height of an AVL node recursively */
static int avl_node_height( avl_node_t *node ) {
  int height_left = 0;
  int height_right = 0;

  if(node->left) height_left = avl_node_height(node->left);
  if(node->right) height_right = avl_node_height(node->right);

  return height_right > height_left ? ++height_right : ++height_left;
}

/* Find the balance of an AVL node */
static int avl_balance_factor( avl_node_t *node ) {
  int bf = 0;

  if(node->left ) bf += avl_node_height(node->left);
  if(node->right) bf -= avl_node_height(node->right);

  return bf;
}

/* Left Left Rotate */
static avl_node_t *avl_rotate_leftleft( avl_node_t *node ) {
  avl_node_t *a = node;
  avl_node_t *b = a->left;

  a->left = b->right;
  if(b->right != NULL) b->right->parent = a; //////
  b->right = a;
  if(b != NULL) b->parent = a->parent; ///////
  a->parent = b; ///////


  return b;
}

/* Left Right Rotate */
static avl_node_t *avl_rotate_leftright( avl_node_t *node ) {
  avl_node_t *a = node;
  avl_node_t *b = a->left;
  avl_node_t *c = b->right;

  a->left = c->right;
  if(c->right != NULL) c->right->parent = a; ///////
  b->right = c->left;
  if(c->left != NULL) c->left->parent = b; ///////
  c->left = b;
  b->parent = c; ///////
  c->right = a;
  c->parent = a->parent; ///////
  a->parent = c; //////
  return c;
}

/* Right Left Rotate */
static avl_node_t *avl_rotate_rightleft( avl_node_t *node ) {
  avl_node_t *a = node;
  avl_node_t *b = a->right;
  avl_node_t *c = b->left;

  //print_albero(a);
  a->right = c->left;
  if(c->left != NULL) c->left->parent = a; ///////
  b->left = c->right;
  if(c->right != NULL) c->right->parent = b; ///////
  c->right = b;
  b->parent = c; ///////
  c->left = a;
  c->parent = a->parent; /////////.
  a->parent = c; ///////

  return c;
}

/* Right Right Rotate */
avl_node_t *avl_rotate_rightright( avl_node_t *node ) {
  avl_node_t *a = node;
  avl_node_t *b = a->right;

  a->right = b->left;
  if(b->left != NULL) b->left->parent = a; ///////
  b->left = a;
  if(b != NULL) b->parent = a->parent; ////////
  if(a != NULL) a->parent = b; ////////

  return( b );
}

/* Balance a given node */
static avl_node_t *avl_balance_node( avl_node_t *node ) {
  return node;
  avl_node_t *newroot = NULL;

  /* Balance our children, if they exist. */
  if(node->left)
    node->left  = avl_balance_node(node->left);
  if(node->right)
    node->right = avl_balance_node(node->right);

  int bf = avl_balance_factor(node);

  if(bf >= 2) {
    /* Left Heavy */

    if(avl_balance_factor(node->left) <= -1)
      newroot = avl_rotate_leftright(node);
    else
      newroot = avl_rotate_leftleft(node);

  } else if(bf <= -2) {
    /* Right Heavy */

    if(avl_balance_factor(node->right) >= 1)
      newroot = avl_rotate_rightleft(node);
    else
      newroot = avl_rotate_rightright(node);

  } else {
    /* This node is balanced -- no change. */
    newroot = node;
  }

  return newroot;
}

/* Balance a given tree */
static void avl_balance( avl_tree_t *tree ) {
  return;
  avl_node_t *newroot = NULL;
  newroot = avl_balance_node(tree->root);

  if(newroot != tree->root) {
    tree->root = newroot;
    tree->root->parent = NULL;
  }
}

/* Insert a new node. */
void avl_insert( avl_tree_t *tree, double key, int element, int arraysize) {
  avl_node_t *node = NULL;
  avl_node_t *next = NULL;
  avl_node_t *last = NULL;

  /* Well, there must be a first case */
  if( tree->root == NULL ) {
    node = avl_create_node(key, arraysize);
    avl_node_insert_element(node, element);

    tree->root = node;

    /* Okay.  We have a root already.  Where do we put this? */
  } else {
    next = tree->root;

    while( next != NULL ) {
      last = next;

      if( key < next->key ) {
        next = next->left;

      } else if( key > next->key ) {
        next = next->right;

        /* Have we already inserted this node? */
      } else if( key == next->key ) {
        avl_node_insert_element(next, element);
        return; // No need to balance, just append the element in the list!
      }
    }

    node = avl_create_node(key, arraysize);
    avl_node_insert_element(node, element);

    if( key < last->key ) last->left = node;
    if( key > last->key ) last->right = node;

    node->parent = last;

  }

  avl_balance(tree);
}

/* Find the node containing a given key */
avl_node_t *avl_find( avl_tree_t *tree, double key ) {
  avl_node_t *current = tree->root;

  while( current && current->key != key ) {
    if( key > current->key )
      current = current->right;
    else
      current = current->left;
  }

  return current;
}
```

```

static avl_node_t *avl_maximum(avl_node_t *node) {
  if(node == NULL)
    return NULL;

  while(node->right != NULL)
    node = node->right;

  return node;
}

static avl_node_t *avl_predecessor(avl_node_t *node) {
  if(node == NULL)
    return NULL;

  if(node->left != NULL)
    return avl_maximum(node->left);

  avl_node_t *parent = node->parent;
  while(parent != NULL && node == parent->left) {
    node = parent;
    parent = parent->parent;
  }

  return parent;
}

/* Finds the first node (in decreasing order) with data.size > 0 */
avl_node_t *avl_find_nonempty_maximum(avl_tree_t *tree) {
  avl_node_t *maximum = avl_maximum(tree->root);
  if(maximum->data.size == 0) {
    maximum = avl_predecessor(maximum);
    while(maximum != NULL && maximum->data.size == 0) {
      maximum = avl_predecessor(maximum);
    }
  }

  return maximum;
}

/* Frees the memory held by a node */
static void destroy_node(avl_node_t *node) {
  if(node == NULL) return;

  if(node->left)
    destroy_node(node->left);
  if(node->right)
    destroy_node(node->right);

  if(node->data.a != NULL) free(node->data.a);
  if(node->data.h != NULL) free(node->data.h);
  free(node);
}

/* Destroys the entire tree */
void avl_destroy(avl_tree_t *tree) {
  destroy_node(tree->root);
  free(tree);
}
```

Listing A.16: greedy.c

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <math.h>

#include "greedy.h"
#include "countingsort.h"

set_t* pure_greedy_heuristic(set_t** sets, int m, int n) { // Runs in O(k) time
  int i, j;
  int candidate;
  int min_freq;
  //print_sets(sets, m);
  set_t* sorted_sets = counting_sort(*sets, m, n, cardinality); // O(m+n)
  int* frequencies = calloc(n, sizeof(int));
  if(frequencies == NULL) return NULL;

```

```c
  for(i=0;i<m;i++) { // O(k)
    min_freq = INT_MAX;
    candidate = -1;
    for(j=0;j<sorted_sets[i].card;j++) {
      if(frequencies[sorted_sets[i].elems[j]] < min_freq) {
        min_freq = frequencies[sorted_sets[i].elems[j]]; // Select the least-chosen variable from each set, using direct access
        candidate = sorted_sets[i].elems[j];
      }
    }
    sorted_sets[i].repr = candidate;
    frequencies[candidate]++;
  }

  set_t* sets_to_return = counting_sort(sorted_sets, m, m, id); // O(m + n) ---- IDs go from 1 to m+1

  free(frequencies);
  free(sorted_sets);
  free(*sets);
  return sets_to_return;
}

/* Select the least-appearing var, if it does not exceed the mean.
 * If it does, choose according the pure greedy strategy (i.e. the least-chosen)
 */
set_t* forward_greedy_heuristic(set_t** sets, int m, int n) {
  int i, j;
  //print_sets(sets, m);
  set_t* sorted_sets = counting_sort(*sets, m, n, cardinality);
  int* occurrences = calloc(n, sizeof(int));
  if(occurrences == NULL) return NULL;
  int* frequencies = calloc(n, sizeof(int));
  if(frequencies == NULL) return NULL;
  int mean = (int) ceil((double) m/n);
  int candidate_freq, candidate_occ, min_freq, min_occ;

  for(i=0;i<m;i++)
    for(j=0;j<sorted_sets[i].card;j++)
      occurrences[sorted_sets[i].elems[j]]++; // Count the occurrences of all the variables

  for(i=0;i<m;i++) {
    candidate_freq = -1;
    candidate_occ = -1;
    min_freq = INT_MAX;
    min_occ = INT_MAX;
    for(j=0;j<sorted_sets[i].card;j++) {
      if(occurrences[sorted_sets[i].elems[j]] < min_occ && frequencies[sorted_sets[i].elems[j]] < mean) {
        candidate_occ = sorted_sets[i].elems[j];
        min_occ = occurrences[sorted_sets[i].elems[j]];
      }
      if(frequencies[sorted_sets[i].elems[j]] < min_freq) {
        candidate_freq = sorted_sets[i].elems[j];
        min_freq = frequencies[sorted_sets[i].elems[j]];
      }
    }
    if(candidate_occ == -1) // Did not find a future variable chosen less than the mean time
      sorted_sets[i].repr = candidate_freq; // Pure greedy
    else
      sorted_sets[i].repr = candidate_occ; // candidate_occ is the least-appearing var that does not exceed the mean (>=)

    /* Keep the occurrences updated */

    frequencies[sorted_sets[i].repr]++; // Increase the frequency of the chosen variable
    for(j=0;j<sorted_sets[i].card;j++)
      occurrences[sorted_sets[i].elems[j]]--; // Decrease the occurrences of the variables in this set
  }

  set_t* sets_to_return = counting_sort(sorted_sets, m, m, id); // O(m + n) ---- IDs go from 1 to m+1

  free(frequencies);
  free(occurrences);
  free(sorted_sets);
  free(*sets);
  return sets_to_return;
}

set_t* asm2_heuristic(set_t** sets, int m, int n) {
    int i, j;
    int candidate;
    int min_freq, min_occ;
    set_t* sorted_sets = counting_sort(*sets, m, n, cardinality); // O(m+n)
    int* frequencies = calloc(n, sizeof(int));
    if(frequencies == NULL) return NULL;
    int* occurrences = calloc(n, sizeof(int));
    if(occurrences == NULL) return NULL;

    for(i=0;i<m;i++)
        for(j=0;j<sorted_sets[i].card;j++)
            occurrences[sorted_sets[i].elems[j]]++; // Count the occurrences of all the variables (the degrees)

    for(i=0;i<m;i++) { // O(k)
        min_freq = INT_MAX;
        min_occ = INT_MAX;
        candidate = -1;
        for(j=0;j<sorted_sets[i].card;j++) {
            if(frequencies[sorted_sets[i].elems[j]] < min_freq) {
                min_freq = frequencies[sorted_sets[i].elems[j]]; // Select the least-chosen variable from each set, using direct access
                candidate = sorted_sets[i].elems[j];
            }
        }
        for(j=0;j<sorted_sets[i].card;j++) {
            /* Tie-breaking mechanism */
            if(frequencies[sorted_sets[i].elems[j]] == min_freq && occurrences[sorted_sets[i].elems[j]] < min_occ) {
                min_occ = occurrences[sorted_sets[i].elems[j]];
                candidate = sorted_sets[i].elems[j];
            }
            /* This heuristic does not decrease the occurrences of the variables in the current set */
        }
        sorted_sets[i].repr = candidate;
        frequencies[candidate]++;

    }

    set_t* sets_to_return = counting_sort(sorted_sets, m, m, id); // O(m + n) ---- IDs go from 1 to m+1

    free(frequencies);
    free(occurrences);
    free(sorted_sets);
    free(*sets);
    return sets_to_return;
}
```

Listing A.17: pagerank.c

```c
#include <stdlib.h>
#include <float.h>
#include <math.h>

#include "pagerank.h"
#include "avlnode.h"
#include "avl.h"
#include "intlist.h"
#include "rng.h"

/* Inside the sets there are the variables -1! */

void print_matrix(double **W, int m, int n) {
  for(int i=0;i<m;i++) {
    for(int j=0;j<n;j++) {
      printf("%lf\t",W[i][j]);
    }
    printf("\n");
  }
}

set_t* pagerank_heuristic(set_t* sets, int m, int n, int k) { // K = sum of the cardinality
  int i, j;
  /* Weights */
  double **W = malloc(m * sizeof *W);
  double *total_weights = calloc(n, sizeof *total_weights);
  int *ones = calloc(n, sizeof *ones);
  int *not_zero = calloc(n, sizeof *not_zero);
  double max_total_weight = 0.;
  int var_max_total_weight = -1;
  /* AVLs */
  avl_tree_t **avls = malloc(n * sizeof *avls);

  if(W == NULL) return NULL;
  if(total_weights == NULL) return NULL;
  if(avls == NULL) return NULL;
  if(ones == NULL) return NULL;
  if(not_zero == NULL) return NULL;

  /* Initialize the AVLs */
  for(i=0;i<n;i++)
    avls[i] = avl_create(); /* They are indexed by the variables */

  /* Initialize the Weights matrix and the variables' AVLs */
  for(i=0;i<m;i++) { // Theta(k)
    W[i] = calloc(n + 1, sizeof *(W[i])); // n+1 to store the current cardinality in the last column
    for(j=0;j<sets[i].card;j++) {
      int var = sets[i].elems[j];
      W[i][var] = (double) 1./sets[i].card; // Populate the matrix...
      total_weights[var] += (double) 1./sets[i].card; // ... and update the total weights of the variables
      if(total_weights[var] > max_total_weight) { // Update the max total weight
        max_total_weight = total_weights[var];
        var_max_total_weight = var;
      }
      not_zero[var]++;
      /* The variable var appears in the i-th set, with relative weight of W[i][var] */
      if(W[i][var] != 1.) {
        //printf("Inserisco nell'avl %lf della variabile %i\n", W[i][var], var);
        avl_insert(avls[var], W[i][var], i, m); // Insert in the avls[var]: the key W[i][var] which is the relative weight of var in the i-th set
      }
      else { // Singleton
        ones[var]++;
        sets[i].repr = var;
      }
    }
    W[i][n] = sets[i].card;
  }

  /* Core of the algorithm */
  for(i=0;i<k-m;i++) {
    /* Pick the maximum total weight -> max_total_weight */
    /* Where is the var of maximum total weight with its heighest relative weight */

    avl_node_t *maximum_node = avl_find_nonempty_maximum(avls[var_max_total_weight]); // Mi da il primo nodo (massimo) in cui A.size != 0

    int chosen_set = avl_node_get_random_element(maximum_node); // Dammi un insieme con quella variabile e pi  alto relative weight

    avl_node_remove_element(maximum_node, chosen_set); // Rimuovi dal nodo dell'avl quell'insieme (chosen_set)

    /* Zero the variable and decrease the total maximum */
    total_weights[var_max_total_weight] -= W[chosen_set][var_max_total_weight];
    W[chosen_set][var_max_total_weight] = 0.;
    not_zero[var_max_total_weight]--; // One more zero added to this set
    W[chosen_set][n]--; // Decrease the logical cardinality of chosen_set

    max_total_weight = 0;
    for(j=0;j<n;j++) { // Update all the variables in this set
      if(W[chosen_set][j] != 0.) {
        avl_node_remove_element(avl_find(avls[j], W[chosen_set][j]), chosen_set); // This variable does not appear in this subset with this set anymore

        total_weights[j] -= W[chosen_set][j]; // Update total weights
        W[chosen_set][j] = (double) 1. / W[chosen_set][n]; // 1 / current cardinality

        if(W[chosen_set][j] == 1) {// Do we have a new representative?
          sets[chosen_set].repr = j;
          ones[j]++;
        }

        total_weights[j] += W[chosen_set][j];
        if(W[chosen_set][j] != 1.) {// Only re-add this variable if it is not the only left in the set
          avl_insert(avls[j], W[chosen_set][j], chosen_set, m); // Add back the new relative weight
        }
      }

      if(total_weights[j] >= max_total_weight && not_zero[j] != ones[j]) { // Update the max total weight
        max_total_weight = total_weights[j];
        var_max_total_weight = j;
      }
    }
  }

  /* Release the memory */
  free(total_weights);
  for(i=0;i<n;i++)
    avl_destroy(avls[i]);
  free(avls);
  for(i=0;i<m;i++)
    free(W[i]);
  free(W);
  free(ones);

  return sets;
}
```

Listing A.18: main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <dirent.h>
#include <unistd.h>
#include <time.h>
#include <float.h>
#include <sys/types.h>
#include <sys/stat.h>

#include "types.h"
#include "sm.h"
#include "greedy.h"
#include "pagerank.h"
#include "countingsort.h"
#include "rng.h"

#define timing(f) start = clock(); \
(f); \
end = clock(); \
time_spent = (double) (end - start) / CLOCKS_PER_SEC;

typedef struct {
  double* time_asm1;
  double* time_asm2;
  double* time_greedy;
  double* time_greedyfwd;
    double* time_asm2heur;
  double* time_pagerank;
  int errors_greedy;
  int errors_greedyfwd;
  int errors_pagerank;
    int errors_asm2heur;
  int badness_greedy;
  int badness_greedyfwd;
    int badness_asm2heur;
  int badness_pagerank;
} alg_stats_t;

static int is_regular_file(const char *path)
{
  struct stat path_stat;
  stat(path, &path_stat);
  return S_ISREG(path_stat.st_mode);
}

static set_t* build_sets_from_userinput(int *m, int* n, int **occ) {
  set_t* sets;

  printf("How many sets? ");
  scanf("%i", m);
  printf("How many vars? ");
  scanf("%i", n);

  *occ = calloc(*n, sizeof *occ);
  if(*occ == NULL) return NULL;
  sets = malloc(*m * sizeof(*sets));
  if(sets == NULL) return NULL;

  for(int i=0;i<*m;i++) {
    printf("How many elements in set %i? ", i+1);
    scanf("%i", &sets[i].card);
    sets[i].elems = malloc(sets[i].card * sizeof(int));
    if(sets[i].elems == NULL) return NULL;
    sets[i].id = i+1; // For the counting sort to work
    sets[i].repr = -1;

    for(int j=0;j<sets[i].card;j++) {
      scanf("%i", &sets[i].elems[j]);
      sets[i].elems[j]--;
      ((*occ)[sets[i].elems[j]])++;
    }
  }

  return sets;
}

static set_t* build_sets_from_file(const char *filename, int *m, int *n, int **occ) {
  set_t* sets;

  if(is_regular_file(filename)) { // File
    FILE* fileinput = fopen(filename, "r");
    if(fileinput == NULL) return NULL;

    fscanf(fileinput, "%i", m);
    fscanf(fileinput, "%i", n);

    *occ = calloc(*n, sizeof *occ);
    if(*occ == NULL) return NULL;
    sets = malloc(*m * sizeof *sets);
    if(sets == NULL) return NULL;

    for(int i=0;i<*m;i++) {
      /* Format: CARDINALITY {x1 x2 x3 ... xn}\n */
      fscanf(fileinput, "%i {", &sets[i].card);
      sets[i].id = i+1; // For the counting sort to work
      sets[i].repr = -1;
      sets[i].elems = malloc(sets[i].card * sizeof(*(sets[i].elems)));

      for(int j=0;j<sets[i].card;j++) {
        fscanf(fileinput, "%i", &sets[i].elems[j]);
        sets[i].elems[j]--;
        ((*occ)[sets[i].elems[j]])++;
      }

      fscanf(fileinput, "}\n");
    }

    fclose(fileinput);
    return sets;
  }
  else { // Directory
    /* Placeholder */
  }
  return NULL;
}

/* Build M sets with N vars, rand_type_vars distributed. Cardinality distribution type is
     rand_type_card */
static set_t* build_sets_randomly(int m, int n, int **occ, int rand_type_card, int rand_type_vars) {
  set_t* sets = malloc(m * sizeof *sets);
  if(sets == NULL) return NULL;
  *occ = calloc(n, sizeof *occ);
  if(*occ == NULL) return NULL;
  int k, ok;

  for(int i=0;i<m;i++) {
    sets[i].card = my_rand(rand_type_card, 1, n, (double) 1./(floor((double) n/2.)));
    sets[i].elems = malloc(sets[i].card * sizeof(int));
    if(sets[i].elems == NULL) return NULL;
    sets[i].id = i+1; // For the counting sort to work
    sets[i].repr = -1;

    for(int j=0;j<sets[i].card;j++) {
      ok = 1;
      sets[i].elems[j] = my_rand(rand_type_vars, 1, n, (double) 1./n);
      for(k=0;k<j;k++) {
        if(sets[i].elems[k] == sets[i].elems[j]-1) { // Already generated this var, try another one
          j--;
          ok = 0;
          break;
        }
      }
      if(ok) { // Successfully added a new variable
        sets[i].elems[j]--;
        ((*occ)[sets[i].elems[j]])++;
      }
    }

    int* tmp = integer_counting_sort(sets[i].elems, sets[i].card, n); // Sort the variables in the set
    free(sets[i].elems);
    sets[i].elems = tmp;
  }

  return sets;
}

static int check_max_freq(set_t* sets, int m, int n, int* frequencies, int* var_of_max_freq) {
  int i;
  for(i=0;i<n;i++)
    frequencies[i] = 0;

  for(i=0;i<m;i++) { // Print the solution and count the frequencies
    frequencies[sets[i].repr]++;
    //printf("%i from ",sets[i].repr+1);
    //print_set(sets[i]);
    //printf("\n");
  }
  /* Search the maximum frequency */
  int maximum_frequency = 0;
  *var_of_max_freq = -1;
  for(i=0;i<n;i++) {
    if(frequencies[i] > maximum_frequency) {
      maximum_frequency = frequencies[i];
      *var_of_max_freq = i;
    }
  }

  return maximum_frequency;
}

static set_t* simplify(set_t* sets, int* m, int* n, int* occ) {
  int i, j;
  int mean = (int) ceil((double) (*m)/(*n));
  int discarded = 0;

  for(i=0;i<*m;i++) {
    for(j=0;j<sets[i].card;j++) {
      if(occ[sets[i].elems[j]] < mean) {
        sets[i].repr = sets[i].elems[j];
        discarded++;
        printf("[SIMPLIFY] Took %i from ", sets[i].repr+1);
        print_set(sets[i]);
        printf("\n");
        break;
      }
    }
  }

  if(discarded > 0) {
    set_t* sorted_sets = counting_sort(sets, *m, *n, representative);

    for(i=(*m)-discarded;i<*m;i++)
      free(sorted_sets[i].elems);

    sorted_sets = realloc(sorted_sets, ((*m)-discarded) * sizeof *sorted_sets);

    *m = (*m) - discarded;

    sorted_sets = counting_sort(sorted_sets, *m, (*m) + discarded, id);

    for(i=0;i<*m;i++)
      sorted_sets[i].id = i+1;

    return sorted_sets;
  }
  return sets;
}


void call_algorithms(set_t** sets, int m, int n, int *input_size, alg_stats_t *stats) {
  int* frequencies;
  int maximum_frequency;
  int var_of_max_freq;
  int i;
  static int last_time_position = 0;
  static int sum_k = 0;
  int optimum;

  /* Variables used for timing */
  clock_t start, end;
  double time_spent;

  /* Prepare the frequencies array */
  frequencies = calloc(n, sizeof *frequencies);
  if(frequencies == NULL) return;

  /* Call Asm1 */
  printf("#### Asm1 ####\n");
  timing(asm1(*sets, m, n)); // Call Asm1
  if(m < 25000) {
    maximum_frequency = check_max_freq(*sets, m, n, frequencies, &var_of_max_freq); // Count the maximum frequency
    printf("The maximum frequency is %i (variable %i). Time: %lf sec.\n", maximum_frequency, var_of_max_freq+1, time_spent);
    optimum = maximum_frequency;
  }
  else {
    optimum = -1;
  }
  if(stats != NULL && stats->time_asm1 != NULL)
    stats->time_asm1[last_time_position] = time_spent; // Store the time it took

  /* Call Asm2 */
  printf("\n#### Asm2 ####\n");
  for(i=0;i<m;i++)
    (*sets)[i].repr = -1; // Remove the previous representatives
  timing(*sets = asm2(sets, m, n));

  maximum_frequency = check_max_freq(*sets, m, n, frequencies, &var_of_max_freq); // Count the maximum frequency
  printf("The maximum frequency is %i (variable %i). Time: %lf sec.\n", maximum_frequency, var_of_max_freq+1, time_spent);
  if(stats != NULL && stats->time_asm2 != NULL)
    stats->time_asm2[last_time_position] = time_spent; // Store the time it took

    if(optimum == -1) optimum = maximum_frequency; // Remember the result of Asm2 as the optimal value

  /* Call the pure greedy heuristic */
  printf("\n#### PURE GREEDY HEURISTIC ####\n");
  for(i=0;i<m;i++)
    (*sets)[i].repr = -1; // Remove the previous representatives
  timing(*sets = pure_greedy_heuristic(sets, m, n));

  maximum_frequency = check_max_freq(*sets, m, n, frequencies, &var_of_max_freq); // Count the maximum frequency
    printf("The maximum frequency is %i (variable %i). Time: %lf sec.\n", maximum_frequency, var_of_max_freq+1, time_spent);
  if(stats != NULL && stats->time_greedy != NULL)
    stats->time_greedy[last_time_position] = time_spent; // Store the time it took
  if(maximum_frequency != optimum) {
    stats->badness_greedy += maximum_frequency-optimum;
    (stats->errors_greedy)++; // Check if the result is the same as the one of Asm1
  }

  /* Call the forward greedy heuristic */
  printf("\n#### FORWARD GREEDY HEURISTIC ####\n");
  for(i=0;i<m;i++)
    (*sets)[i].repr = -1; // Remove the previous representatives
  timing(*sets = forward_greedy_heuristic(sets, m, n));

  maximum_frequency = check_max_freq(*sets, m, n, frequencies, &var_of_max_freq); // Count the maximum frequency
  printf("The maximum frequency is %i (variable %i). Time: %lf sec.\n", maximum_frequency, var_of_max_freq+1, time_spent);
  if(stats->time_greedyfwd != NULL)
    stats->time_greedyfwd[last_time_position] = time_spent; // Store the time it took
  if(maximum_frequency != optimum) {
    stats->badness_greedyfwd += maximum_frequency-optimum;
    (stats->errors_greedyfwd)++; // Check if the result is the same as the one of Asm1
  }

    /* Call the Asm2 heuristic */
    printf("\n#### ASM2 INIT HEURISTIC ####\n");
    for(i=0;i<m;i++)
        (*sets)[i].repr = -1; // Remove the previous representatives
    timing(*sets = asm2_heuristic(sets, m, n));

    maximum_frequency = check_max_freq(*sets, m, n, frequencies, &var_of_max_freq); // Count the maximum frequency
    printf("The maximum frequency is %i (variable %i). Time: %lf sec.\n", maximum_frequency, var_of_max_freq+1, time_spent);
    if(stats->time_asm2heur != NULL)
        stats->time_asm2heur[last_time_position] = time_spent; // Store the time it took
    if(maximum_frequency != optimum) {
        stats->badness_asm2heur += maximum_frequency-optimum;
        (stats->errors_asm2heur)++; // Check if the result is the same as the one of Asm1
    }

  /* Call the Pagerank heuristic */
  printf("\n#### PAGERANK HEURISTIC ####\n");
  int k = 0; // Count the cardinalities of all the sets
  for(i=0;i<m;i++) {
    (*sets)[i].repr = -1; // Remove the previous representatives
    k += (*sets)[i].card;
  }
  sum_k += k;
  timing(*sets = pagerank_heuristic(*sets, m, n, k));

  maximum_frequency = check_max_freq(*sets, m, n, frequencies, &var_of_max_freq); // Count the maximum frequency
  printf("The maximum frequency is %i (variable %i). Time: %lf sec.\n", maximum_frequency, var_of_max_freq+1, time_spent);
  if(stats->time_pagerank != NULL)
    stats->time_pagerank[last_time_position] = time_spent; // Store the time it took
  if(maximum_frequency != optimum) {
    stats->badness_pagerank += maximum_frequency-optimum;
    (stats->errors_pagerank)++; // Check if the result is the same as the one of Asm1
  }

  if(stats->time_asm1 != NULL || stats->time_asm2 != NULL || stats->time_greedy != NULL || stats->time_greedyfwd != NULL || stats->time_pagerank != NULL)
    last_time_position++; // Update the index of the time arrays

  if(input_size != NULL)
    *input_size = sum_k;

  free(frequencies);
}

int main(int argc, const char * argv[]) {
  int m, n;
  set_t* sets;
  int i;
  int *occurrences;

  /* Set the random seed */
  unsigned int seed = (argc == 5) ? (unsigned int) atoi(argv[4]) : (unsigned int) time(NULL); // Save the seed for debug purposes;
  printf("RNG Seed: %u\n\n", seed);
  srand(seed);

  /* Build the input instance */
  switch(argc) {
    case 5:
    case 4: { // m sets, n vars, x different instances
      int instances = atoi(argv[3]);
      n = atoi(argv[2]);
      m = atoi(argv[1]);
  FILE* stats_file = fopen("stats_nuove_unif.csv", "a");
  if(stats_file == NULL) return -3;
  /* Create the arrays for storing execution times of the algorithms */
  alg_stats_t stats = {
    malloc(instances * sizeof(double)),
    malloc(instances * sizeof(double)),
    malloc(instances * sizeof(double)),
            malloc(instances * sizeof(double)),
    malloc(instances * sizeof(double)),
    malloc(instances * sizeof(double)),
    0,
    0,
    0,
    0,
    0,
    0,
            0,
            0
  };
  double *t_asm1 = stats.time_asm1;
  double *t_asm2 = stats.time_asm2;
  double *t_greedy = stats.time_greedy;
  double *t_greedyfwd = stats.time_greedyfwd;
  double *t_pagerank = stats.time_pagerank;
        double *t_asm2heur = stats.time_asm2heur;
  double min_time_asm1 = DBL_MAX, min_time_asm2 = DBL_MAX, min_time_greedy = DBL_MAX, min_time_greedyfwd = DBL_MAX, min_time_pagerank = DBL_MAX, min_time_asm2heur = DBL_MAX;
  double max_time_asm1 = -1., max_time_asm2 = -1., max_time_greedy = -1., max_time_greedyfwd = -1., max_time_pagerank = -1., max_time_asm2heur = -1.;
  double sum_asm1 = 0., sum_asm2 = 0., sum_greedy = 0., sum_greedyfwd = 0., sum_pagerank = 0., sum_asm2heur = 0.;
  int total_input_size = 0;
  int simplify_sets = 0;
  int removed_vars = 0;

  clock_t start = clock();

  for(i=0;i<instances;i++) {
    /* Build and print the input */
            sets = build_sets_randomly(m, n, &occurrences, RNG_UNIF, RNG_GAUSS);
    printf("[#%i] m = %i, n = %i\n", i, m, n);
    simplify_sets += m; // For statistical purposes
    //print_sets(sets, m);
    sets = simplify(sets, &m, &n, occurrences);
    printf("\nAfter Simplifying: m = %i, n = %i\n", m, n);
    simplify_sets -= m;

    /* Check which variables have been simplified */
    for(int j=0;j<n;j++)
      occurrences[j] = 0;

    for(int x=0;x<m;x++) {
      for(int j=0;j<sets[x].card;j++) {
        occurrences[sets[x].elems[j]]++;
      }
    }

    for(int j=0;j<n;j++) {
      if(occurrences[j] == 0)
        removed_vars++;
    }

    //print_sets(sets, m);

    /* Call the algorithms on this instance */
    call_algorithms(&sets, m, n, &total_input_size, &stats);

    /* Update the time stats */
    if(t_asm1[i] < min_time_asm1)
      min_time_asm1 = t_asm1[i]; // Min
    if(t_asm1[i] > max_time_asm1)
      max_time_asm1 = t_asm1[i]; // Max
    sum_asm1 += t_asm1[i]; // Mean
    /* Time stats for ASM2 */
    if(t_asm2[i] < min_time_asm2)
      min_time_asm2 = t_asm2[i]; // Min
    if(t_asm2[i] > max_time_asm2)
      max_time_asm2 = t_asm2[i]; // Max
    sum_asm2 += t_asm2[i]; // Mean
    /* Time stats for Greedy */
    if(t_greedy[i] < min_time_greedy)
      min_time_greedy = t_greedy[i]; // Min
    if(t_greedy[i] > max_time_greedy)
      max_time_greedy = t_greedy[i]; // Max
    sum_greedy += t_greedy[i]; // Mean
    /* Time stats for Greedy FWD */
    if(t_greedyfwd[i] < min_time_greedyfwd)
      min_time_greedyfwd = t_greedyfwd[i]; // Min
    if(t_greedyfwd[i] > max_time_greedyfwd)
      max_time_greedyfwd = t_greedyfwd[i]; // Max
    sum_greedyfwd += t_greedyfwd[i]; // Mean
    /* Time stats for Pagerank */
    if(t_pagerank[i] < min_time_pagerank)
      min_time_pagerank = t_pagerank[i]; // Min
    if(t_pagerank[i] > max_time_pagerank)
      max_time_pagerank = t_pagerank[i]; // Max
    sum_pagerank += t_pagerank[i]; // Mean
            /* Time stats for ASM2 Heuristic */
            if(t_asm2heur[i] < min_time_asm2heur)
                min_time_asm2heur = t_asm2heur[i]; // Min
            if(t_asm2heur[i] > max_time_asm2heur)
                max_time_asm2heur = t_asm2heur[i]; // Max
            sum_asm2heur += t_asm2heur[i]; // Mean

    /* Free the sets */
    for(int j=0;j<m;j++)
      free(sets[j].elems);
    free(sets);
    free(occurrences);

    /* Reset the old the values of n and m for the next instance to generate */
    n = atoi(argv[2]);
    m = atoi(argv[1]);
  }

  clock_t end = clock();

  printf("\n#### TIME STATS ####\n");
  printf("[ASM1] Min: %.6lf\tMax: %.6lf\tMean: %.6lf.\n", min_time_asm1, max_time_asm1, sum_asm1
/ (double) instances);
  printf("[ASM2] Min: %.6lf\tMax: %.6lf\tMean: %.6lf.\n", min_time_asm2, max_time_asm2, sum_asm2
/ (double) instances);
  printf("[GREEDY] Min: %.6lf\tMax: %.6lf\tMean: %.6lf.\n", min_time_greedy, max_time_greedy,
sum_greedy / (double) instances);
  printf("[GREEDY FWD] Min: %.6lf\tMax: %.6lf\tMean: %.6lf.\n", min_time_greedyfwd,
max_time_greedyfwd, sum_greedyfwd / (double) instances);
  printf("[PAGERANK] Min: %.6lf\tMax: %.6lf\tMean: %.6lf.\n", min_time_pagerank,
max_time_pagerank, sum_pagerank / (double) instances);
        printf("[ASM2 HEURISTIC] Min: %.6lf\tMax: %.6lf\tMean: %.6lf.\n", min_time_asm2heur,
max_time_asm2heur, sum_asm2heur / (double) instances);
  printf("[TOTAL TIME] %.6lf\n\n", (double) (end - start) / CLOCKS_PER_SEC);
  printf("#### GENERAL STATS ####\n");
  printf("[MEAN INPUT SIZE] %i\n", (int) total_input_size/instances);
  printf("[AVG SETS REMOVED BY SIMPLIFY IN EACH INSTANCE] %i\n", (int) ceil((double)
simplify_sets / instances));
  printf("[AVG VARS REMOVED BY SIMPLIFY IN EACH INSTANCE] %i\n", (int) ceil((double) removed_vars
/ instances));
  printf("[GREEDY] Errors: %i\n", stats.errors_greedy);
  printf("[GREEDY FWD] Errors: %i\n", stats.errors_greedyfwd);
  printf("[PAGERANK] Errors: %i\n", stats.errors_pagerank);
        printf("[ASM2 HEURISTIC] Errors: %i (deviation: %.2lf) \n", stats.errors_asm2heur,
(stats.errors_asm2heur > 0) ? (double) stats.badness_asm2heur / stats.errors_asm2heur : 0.);
  printf("RNG Seed: %u\n", seed);

  fprintf(stats_file,
"%u,%i,%i,%i,%i,%i,%i,%i,%i,%.2lf,%.2lf,%.2lf,%.2lf,%i,%i,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,%lf,
      seed,
      m,
      n,
      instances,
      (int) total_input_size/instances,
      stats.errors_greedy,
      stats.errors_greedyfwd,
      stats.errors_pagerank,
                stats.errors_asm2heur,
      (stats.errors_greedy > 0) ? (double) stats.badness_greedy / stats.errors_greedy : 0.,
      (stats.errors_greedyfwd > 0) ? (double) stats.badness_greedyfwd / stats.errors_greedyfwd :
0.,
          (stats.errors_pagerank > 0) ? (double) stats.badness_pagerank / stats.errors_pagerank : 0.,
                    (stats.errors_asm2heur > 0) ? (double) stats.badness_asm2heur / stats.errors_asm2heur : 0.,
          (int) ceil((double) simplify_sets / instances),
          (int) ceil((double) removed_vars / instances),
          sum_asm1 / (double) instances,
          sum_asm2 / (double) instances,
          sum_greedy / (double) instances,
          sum_greedyfwd / (double) instances,
          sum_pagerank / (double) instances,
          min_time_asm1,
          min_time_asm2,
          min_time_greedy,
          min_time_greedyfwd,
          min_time_pagerank,
          max_time_asm1,
          max_time_asm2,
          max_time_greedy,
          max_time_greedyfwd,
          max_time_pagerank
      );

      fclose(stats_file);
      free(t_asm1);
      free(t_asm2);
      free(t_greedy);
      free(t_greedyfwd);
      free(t_pagerank);
            free(t_asm2heur);
      return 0; // quit the entire program
    }
    case 2:
      sets = build_sets_from_file(argv[1], &m, &n, &occurrences);
      break;
    default:
      sets = build_sets_from_userinput(&m, &n, &occurrences);
      break;
  }

  if(sets == NULL) {
    fprintf(stderr, "There was an error during the creation of the instance. [QUIT]\n");
    return -1;
  }

  printf("Input: m = %i, n = %i\n", m, n);
  print_sets(sets, m);
  sets = simplify(sets, &m, &n, occurrences);
  printf("\nAfter Simplifying: m = %i, n = %i\n", m, n);
  print_sets(sets, m);

  alg_stats_t stats = {
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    NULL,
    0,
    0,
    0,
    0,
    0,
    0,
    0,
    0
  };
  call_algorithms(&sets, m, n, NULL, &stats);
  /* Free the sets */
  for(i=0;i<m;i++)
    free(sets[i].elems);
  free(sets);
  free(occurrences);
  return 0;
}
```