UNIVERSITY OF WROCŁAW
INSTITUTE OF COMPUTER SCIENCE

# CNF Encodings of Cardinality Constraints Based on Comparator Networks

*Author:*
Michał KARPIŃSKI

*Supervisor:*
dr hab. Marek PIOTRÓW

October 31, 2019

# Acknowledgments

This thesis – the culmination of my career as a graduate student (or as an apprentice, in a broader sense) – is a consequence of many encounters I have had throughout my life, not only as an adult, but also as a youth. It is because I was lucky to meet these particular people on my path, that I am now able to raise a toast to my success. A single page is insufficient to mention all those important individuals (which I deeply regret), but still I would like to express my gratitude to a few selected ones.

First and foremost, I would like to thank my advisor – Marek Piotrów – whose great effort, dedication and professionalism guided me to the end of this journey. I especially would like to thank him for all the patience he had while repeatedly correcting the same lazy mistakes I made throughout my work as a PhD student. If I had at least 10% of his determination, I would have finished this thesis sooner. For what he has done, I offer my everlasting respect. I would also like to thank Prof. Krzysztof Loryś for advising me for many years and not loosing faith in me.

I thank a few selected colleagues, classmates and old friends, those who have expanded my horizons, and those who on many occasions offered me a helping hand (in many ways). They have been good friends to me for many years, and although I have lost contact with most them I still feel obliged to mention them here: Aleksander Balicki, Marcin Barczyszyn, Paweł Bogdan, Michał Bruliński, Piotr Gajowiak, Marcin Kolarczyk, Michał Krasowski, Kacper Król, Michał Lepka, Mateusz Lewandowski, Marcin Oczeretko, Jacek Olchawa, Krzysztof Piecuch, Damian and Dominik Rusak, Grzegorz Sobiegraj, Kuba Stępniewicz, Damian Straszak, Michał Szczepanowski, Jakub Tarnawski, Łukasz Zapart.

I would like to separately mention my colleagues from the Institute of Computer Science, for the great time I had during the years of my PhD program: residents of the room 340 – Paweł Garncarek, Piotr Polesiuk; residents of the room 202 – Jan Chorowski, Adam Kaczmarek, Adrian Łańcucki, Michał Różański; and all other wonderful people of the Institue of Computer Science. Andrés Aristizábal, even though your time in Poland was short, I think of you as my brother.

One particular person needs to be mentioned separately – Maciej Pacut – my friend, my roommate, and the fellow PhD student, who gave me the final push to continue my graduate studies after obtaining master's degree. Whether his suggestion was an epiphany, or a curse, I will ponder for years to come.

Finally, I would like to thank my family, especially my mother, for providing for me and for allowing me to peacefully pursue my dreams.

# Abstract

Boolean Satisfiability Problem (SAT) is one of the core problems in computer science. As one of the fundamental NP-complete problems, it can be used – by known reductions – to represent instances of variety of hard decision problems. Additionally, those representations can be passed to a program for finding satisfying assignments to Boolean formulas, for example, to a program called MINISAT. Those programs (called SAT-solvers) have been intensively developed for many years and – despite their worst-case exponential time complexity – are able to solve a multitude of hard practical instances. A drawback of this approach is that clauses are neither expressive, nor compact, and using them to describe decision problems can pose a big challenge on its own.

We can improve this by using high-level constraints as a bridge between a problem at hand and SAT. Such constraints are then automatically translated to eqisatisfiable Boolean formulas. The main theme of this thesis revolves around one type of such constraints, namely Boolean Cardinality Constraints (or simply *cardinality constraints*). Cardinality constraints state that at most (at least, or exactly) $k$ out of $n$ propositional literals can be true. Such cardinality constraints appear naturally in formulations of different real-world problems including cumulative scheduling, timetabling or formal hardware verification.

The goal of this thesis is to propose and analyze new and efficient methods to *encode* (translate) cardinality constraints into equisatisfiable proposition formulas in CNF, such that the resulting SAT instances are small and that the SAT-solver runtime is as short as possible. The ultimate contribution of this thesis is the presentation and analysis of several new translation algorithms, that improve the state-of-the-art in the field of encoding cardinality constraints. Algorithms presented here are based on *comparator networks*, several of which have been recently proposed for encoding cardinality constraints and experiments have proved their efficiency. With our constructions we achieve better encodings than the current state-of-the-art, in both theoretical and experimental senses. In particular, they make use of so called *generalized comparators*, that can be efficiently translated to CNFs. We also prove that any encoding based on generalized comparator networks preserves *generalized arc-consistency* (GAC) – a theoretical property that guarantees better propagation of values in the SAT-solver computation.

Finally, we explore the possibility of using our algorithms to encode a more general type of constraints - *the Pseudo-Boolean Constraints*. To this end we implemented a PB-solver based on the well-known MINISAT+ and the experimental evaluation shows that on many instances of popular benchmarks our technique outperforms other state-of-the-art PB-solvers.

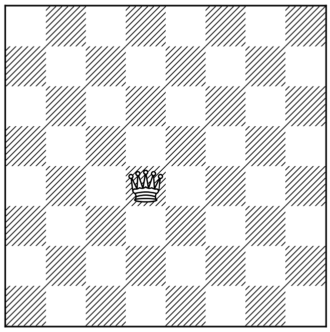
# Contents

# Part I

# SAT-solving and Constraint Programming

# Chapter 1

# Introduction



Several hard decision problems can be efficiently reduced to the Boolean satisfiability (SAT) problem and tried to be solved by recently-developed SAT-solvers. Some of them are formulated with the help of different high-level constraints, which should be either encoded into CNF formulas [34, 58, 67] or solved inside a SAT-solver by a specialized extension [33]. In this thesis we study how a SAT-solver can be used to solve Boolean Constraint Problems by translation to clauses.

The major part of this thesis is dedicated to encoding Boolean Cardinality Constraints that take the form $x_1 + x_2 + \cdots + x_n \;\#\; k$, where $x_1, x_2, \ldots, x_n$ are Boolean literals (that is, variables or their negations), # is a relation from the set $\{<, \leq, =, \geq, >\}$ and $k, n \in \mathbb{N}$. Such cardinality constraints appear naturally in formulations of various real-world problems including cumulative scheduling [72], timetabling [4] or formal hardware verification [20].

The goal of this thesis is to study the technique of translating Boolean Cardinality Constraints into SAT, based on comparator networks approach. We propose several new classes of networks and we prove their utility both theoretically and experimentally. We show that on many instances of popular benchmarks our algorithms outperform other state-of-the-art solvers. The detailed description of the results is given in Section 1.3.

This introductory chapter familiarizes the reader with the concepts of SAT-solving and Constraint Programming (CP) – the central topics providing motivation for this thesis. First, we take a look at the SAT problem and its continuous interest in computer science. We show some applications of SAT and give a short summary of the history of SAT-solving. Then, we turn to the notion of *Constraint Satisfaction Problem* (CSP) and define the main object that is studied in this thesis, namely, *the clausal encoding of cardinality constraints*. We end this chapter with a section explaining how the rest of this thesis is organized and how it contributes to the field of constraint programming.

## 1.1 Brief History of SAT-solving

SAT, or in other words, *Boolean Satisfiability Problem* or *satisfiability problem of propositional logic*, is a decision problem in which we determine whether a given Boolean formula (often called *propositional formula*) has a satisfying assignment or not. We use propositional formulas in conjunctive normal form (CNF). A CNF formula is a conjunction (binary operator $\wedge$) of clauses. Each clause is a disjunction (binary operator $\vee$) of literals, where literal is an atomic proposition $x_i$ or its negation $\neg x_i$. In general, CNF formula on $n$ variables and $m$ clauses can be expressed as:

$$\psi = \bigwedge_{i=1}^{m} \left( \bigvee_{j \in P_i} x_j \vee \bigvee_{j \in N_i} \neg x_j \right),$$

where $P_i, N_i \subseteq \{1, \dots, n\}$, $P_i \cap N_i = \emptyset$, $n, m \in \mathbb{N}$. To make certain ideas more clear, when translating something to CNF, we also call implication (binary operator $\Rightarrow$) a clause, keeping in mind the following equivalence:

$$x_1 \wedge x_2 \wedge \cdots \wedge x_n \Rightarrow y_1 \vee y_2 \vee \cdots \vee y_m \iff \neg x_1 \vee \neg x_2 \vee \cdots \vee \neg x_n \vee y_1 \vee y_2 \vee \cdots \vee y_m.$$

We also present formulas simply as sets of clauses, and single clauses as a sets of literals, for succinctness. Note that any propositional formula can be transformed into an equivalent formula in CNF, in linear time [21].

A *truth assignment* (also called *instantiation* or *assignment*) is a partial function $I$ that maps variables $x \in V$ to the elements of set $\{true, false\}$. Therefore, a single variable can be either true, false or free. The truth values of propositional logic *true* and *false* will be represented by 1 and 0, respectively. A variable $x$ is said to be assigned to (or fixed to) 0 by instantiation $I$ if $I(x) = 0$, assigned to 1 if $I(x) = 1$, and free if $I(x)$ is undefined. In non-ambiguous context, $x = 1$ denotes $I(x) = 1$ (similarly for $x = 0$). We also generalize this concept for sets of variables, so that if we write $V = 1$, we mean that for each $x \in V$, $I(x) = 1$ (same for $V = 0$). An instantiation $I$ of $V$ is said to be *complete* if it fixes all the variables in $V$. The instantiations that are not complete are said to be *partial*. We further extend our notation, such that if $\phi$ is a Boolean formula, then a value of $\phi$ under assignment $I$ is denoted by $I(\phi)$, which can be either 0, 1 or undefined. Furthermore, if $I$ is a complete instantiation and $\bar{x} = \langle x_1, \dots, x_n \rangle$ is a sequence of Boolean literals, then $I(\bar{x}) = \langle I(x_1), \dots, I(x_n) \rangle$.

Although the language of SAT is very limited, it is very powerful, allowing us to model many mathematical and real-world problems. Unfortunately we do not expect to see a fast algorithm that could solve all SAT instances, as the problem is NP-complete by the famous theorem of Cook [30]. The situation did not improve much after almost 50 years. The best known deterministic algorithm solving SAT runs in worst-case time $O(1.439^n)$ [52], where $n$ is the number of variables.

On the positive side, specialized programs called *SAT-solvers* have emerged, and even though they process CNFs – in worst case – in exponential time with respect to the size of the formula, for many practical instances they can quickly determine the satisfiability of the formula. This section presents the most important milestones in the field of SAT-solving. Before we look at historical results, let us see why solving SAT instances is a task of great significance.

### 1.1.1 Applications

**Mathematical puzzles.** We begin our journey with some academic examples. First, let us take a look at a 170-year old puzzle called *8-Queens Puzzle* (a recurring theme of this thesis). The 8-Queens Puzzle is the problem of placing eight chess queens on an $8 \times 8$ chessboard so that no two queens threaten (attack) each other. Chessboard consists of 64 squares with eight rows called *ranks* labeled 1–8 and eight columns called *files* labeled a–h. The *queen* chess piece from its position observes all other squares on the rank, file and both diagonals that she currently occupies (see marked squares in Figure 1.1a). Thus, a solution requires that no two queens share the same rank, file, or diagonal. In Figure 1.1b we see a chessboard with four queens on it: queen on a1 attacks queen on f6, queen on f2 also attacks queen on f6, and since the relation of "attacks" is symmetrical, queen on f6 attacks both queens on a1 and f2. On the other hand, queen on c4 does not threaten any other queen on the board.

The 8-Queens Puzzle can be generalized to $n$-Queens Puzzle – a problem of placing $n$ non-attacking queens on an $n \times n$ chessboard – for which solutions exist for all natural numbers $n > 3$. For simplicity, let us focus on the case $n = 4$ and try to construct a clause set that is satisfiable, if and only if 4-Queens Puzzle has a solution.

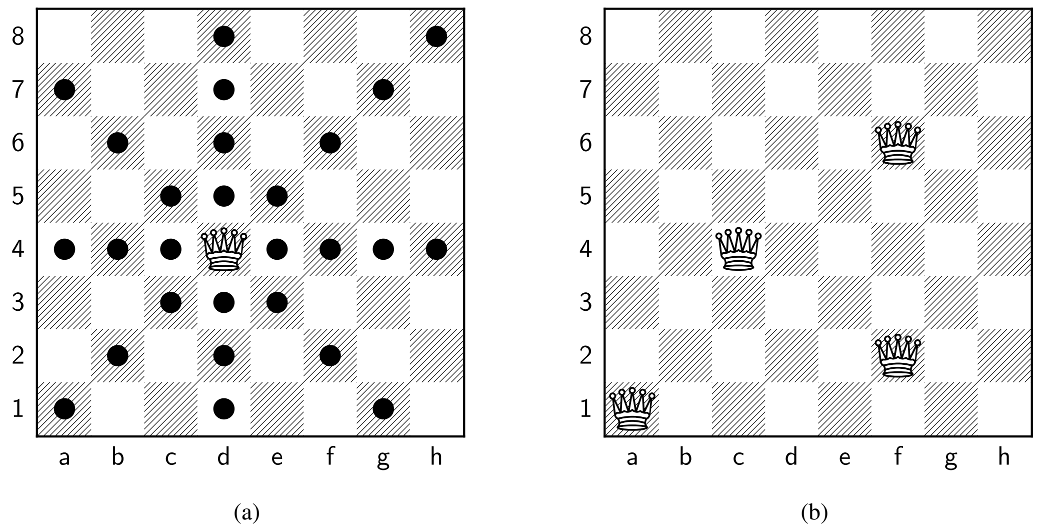



Figure 1.1: Rules of the n-Queens puzzle: (a) all squares that a single queen can attack; (b) sample placement of queens on board

We would like to model our SAT instance in a way that allows easy extraction of a solution from each satisfying assignment. A natural way to achieve it is to introduce a variable $x_{i,j}$ for each square on the board, where $i \in \{a,b,c,d\}$ and $j \in \{1,2,3,4\}$. This way, we can relate a placement of a queen on a square $\{i,j\}$ with the assignment of the variable $x_{i,j}$. Now, we need to create a set of clauses that restricts the placement of queens according to the rules. We do this by adding a set of clauses $\psi$ that are satisfiable, if and only if exactly one variable is set to true for every set of variables that represents some rank and file, and at most one variable is true for each diagonal. For example, for the 1st rank we add:

$$(\neg x_{a,1} \vee \neg x_{b,1}), (\neg x_{a,1} \vee \neg x_{c,1}), (\neg x_{a,1} \vee \neg x_{d,1}), (\neg x_{b,1} \vee \neg x_{c,1}), (\neg x_{b,1} \vee \neg x_{d,1}), (\neg x_{c,1} \vee \neg x_{d,1}),$$
$$(x_{a,1} \vee x_{b,1} \vee x_{c,1} \vee x_{d,1}),$$

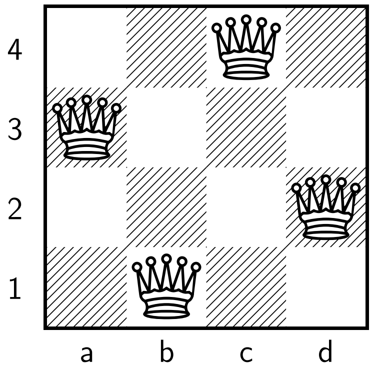
and for the a2-c4 diagonal, we add:

$$(\neg x_{a,2} \vee \neg x_{b,3}), (\neg x_{a,2} \vee \neg x_{c,4}), (\neg x_{b,3} \vee \neg x_{c,4}).$$

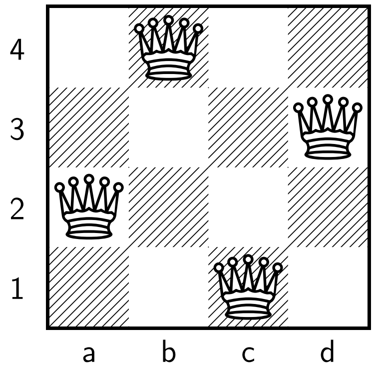
We proceed similarly for all other ranks, files and diagonals on the board. We do not show the full encoding for readability reasons. This would require printing exactly 84 clauses in total for $\psi$. This example illustrates that even for small instances of the considered problem, we can get large sets of clauses.

For $n = 4$, the n-Queens Puzzle has exactly two solutions, as presented in Figure 1.2. As we can see, for both $S_1 = \{x_{a,3}, x_{b,1}, x_{c,4}, x_{d,2}\}$ and $S_2 = \{x_{a,2}, x_{b,4}, x_{c,1}, x_{d,3}\}$, setting either $S_1 = 1$ or $S_2 = 1$ (and the rest of the variables to 0) gives a satisfying assignment of $\psi$. Figuring out whether there is no other solution simply by examining $\psi$ would be cumbersome, but even the simplest SAT-solver would find the answer instantly.

There are many more logic puzzles that could be modeled as SAT instances, for example: Sudoku, Magic squares, Nonograms etc. However, SAT-solving would not become so popular if it was used only for recreational purposes. Thus, we now turn to more practical applications.




(a)




(b)

Figure 1.2: All solutions to 4-Queens puzzle

**Timetabling.** In many real-world situations it is useful to have a chart showing some events scheduled to take place at particular times. Examples of that would be: a chart showing the departure and arrival times of trains, buses, or planes; a class schedule for students and teachers. Another example can be a shift schedule at a workplace. Let us consider the following hypothetical situation.

Assume we own a restaurant and among all our employees we hire 5 waiters: Adam, Brian, Carl, Dan and Eddy. We want to create a shift schedule for them for the next week (starts on Monday, ends on Sunday) given the following constraints:

1. Carl cannot work with Dan, because they do not like each other.
2. Eddy is still learning, therefore he has to work together with either Adam or Brian, who are more experienced.
3. No waiter can work for three consecutive days.
4. Brian cannot work on weekends, because of his studies.
5. Each day, at least two waiters need to work, except Friday, when we expect a lot of customers. Then at least three waiters have to be present.

We can easily model this problem as a SAT instance. First, we create variables indicating that a waiter is assigned to a day of the week. To this end we introduce $x_{v,i}$ for each waiter $v \in W = \{A,B,C,D,E\}$ and for each day $i \in \{1,2,3,4,5,6,7\}$ (1 - Monday, 2 - Tuesday, etc.). Now, if $x_{v,i}$ is true, then a waiter $v$ has to come to work on day $i$. We add the following clauses to encode the given constraints:

1. We simply add $(\neg x_{C,i} \vee \neg x_{D,i})$ for each day $i$.
2. If Eddy works in a given day $i$, then either Adam or Brian has to work: $(x_{E,i} \Rightarrow x_{A,i} \vee x_{B,i})$.
3. For each waiter $v \in W$ and day $i < 6$ we add: $(\neg x_{v,i} \vee \neg x_{v,i+1} \vee \neg x_{v,i+2})$.
4. Brian's absence on Saturday and Sunday can be handled by two singleton clauses: $(\neg x_{B,6}) \wedge (\neg x_{B,7})$, virtually setting $x_{B,6}$ and $x_{B,7}$ to 0.

5. We add a clause $(x_{A,i} \vee x_{B,i} \vee x_{C,i} \vee x_{D,i} \vee x_{E,i})$ for each day $i$ to force at least one waiter to appear for work each day. To make at least two waiters go to work each day $i$, we add a clause for each waiter $v \in W$: $(x_{v,i} \Rightarrow \bigvee_{v' \in W-\{v\}} x_{v',i})$. To make three waiters come on Friday, we add the set of clauses: $((x_{v,5} \wedge x_{v',5}) \Rightarrow \bigvee_{v'' \in W-\{v,v'\}} x_{v'',5})$, for each pair of $v, v' \in W$, where $v \neq v'$.

The union of all above clauses forms a CNF that is satisfiable, if and only if it is possible to create a shift schedule that satisfied all the constraints. The solution can be extracted by simply looking at the truth assignment of the variables. If no solution exists, then as managers we need to either relax some of the constraints or hire more staff.

The problem of timetabling goes beyond our hypothetical considerations. For example, Asín and Nieuwenhuis [4] present SAT encodings for Curriculum-based Course Timetabling and shows that experiments performed on real-world instances improves on what was then considered the state-of-the-art.

**Stable Marriages.** In examples above encodings were fairly straightforward, with a 1-to-1 correspondence between variable definitions and values' truth assignments. A slightly more complex example is shown here. It is based on a study by Gent et al. [38].

In the *stable marriage problem with preference lists* we have $n$ men and $n$ women. Each man $i$ ranks the women in order of preference, and women similarly rank men, creating preference lists $L_i^m$ for man and $L_i^w$ for women. The closer a person is on the list the more desirable that person is as a mate. The problem is to marry men and women in a *stable* way, meaning that there is no incentive for any two individuals to elope and marry each other. A person is willing to leave his/her current partner for a new partner only if he/she is either unmatched or considers the new partner better than the current one. A pair who mutually prefer each other than their partners is a blocking pair, and a matching without blocking pairs is stable.

We define a variable $x_{i,p}$ to be true, if and only if the man $i$ is either unmatched or matched to the women in position $p$ or later in his preference list, where $1 \leq p \leq |L_i^m|$. We also define $x_{i,|L_i^m|+1}$ which is true, if and only if man $i$ is unmatched. Likewise, we define variables $y_{j,q}$ for each women $j$. Note that each variable valuation corresponds to a set of possibilities and that marriages are modeled indirectly via the preference lists. We define the clause set as follows:

- Let $1 \leq i \leq n$. Each man or woman is either matched with someone in their preference list or is unmatched: $x_{i,1} \wedge y_{i,1}$.

- Let $2 \leq p \leq |L_i^m|$ and $2 \leq q \leq |L_j^w|$. If a man $i$ (women $j$) gets his (hers) $(p-1)$-th or better choice, then he (she) certainly gets his (hers) $(p)$-th or better choice: $(\neg x_{i,p} \Rightarrow \neg x_{i,p+1}) \wedge (\neg y_{j,p} \Rightarrow \neg y_{j,p+1})$.

- Now we express the monogamy constraints. Let $p$ be the rank of women $j$ in the preference list of man $i$, and $q$ be the rank of man $i$ in the preference list of woman $j$. If man $i$ has partner no better than women $j$ or is unmatched, and women $j$ has a partner she prefers to man $i$, then man $i$ cannot be matched to women $j$: $(x_{i,p} \wedge \neg y_{j,q} \Rightarrow x_{i,p+1})$. Similarly: $(y_{j,q} \wedge \neg x_{i,p} \Rightarrow y_{j,q+1})$.

- Finally, we enforce stability by stating that if man $i$ is matched to a woman he ranks no better than woman $j$, then woman $j$ must be matched to a man she ranks no lower than man $i$, and vice-versa. Again, let $p$ be the rank of women $j$ in the preference list of man $i$, and $q$ is the rank of man $i$ in the preference list of woman $j$. Then, we add: $(x_{i,p} \Rightarrow \neg y_{j,q+1}) \wedge (y_{j,q} \Rightarrow \neg x_{i,p+1})$.

In the book by Biere et al. [19] authors report on multitude of different problems originating from computer science, that can be handled by translation to SAT. The list of applications consists of, among others: software verification, bounded model checking, combinatorial design, and even statistical physics. Finally, before reviewing the history of SAT-solving, we take a look at the optimization version of SAT, called MaxSAT, by which not only decision problems but also optimization ones can be modeled with clauses.

**MaxSAT.** SAT-solvers usually either report on the found solution or inform the user, that the given instance is unsatisfiable. In practice, we would like to gain more insight about the unsatisfiable instance, for example, which set of clauses causes the unsatisfiability or what is the maximum number of clauses that can be satisfied by some truth assignment. In the example with creating a shift schedule for the restaurant, if we knew that the instance has no solutions, and clauses representing the first constraint makes the CNF unsatisfiable, then we could just remove them, tell Carl and Dan to act professionally regardless of their personal animosity, and move on with the schedule. We could do that if we modeled our problem in MaxSAT.

*Maximum Satisfiability Problem*, or MaxSAT in short, consists of finding an assignment that maximizes the number of satisfied clauses in the given CNF. Even though MaxSAT is NP-hard, some success was made in finding approximate solutions. The first MaxSAT polynomial-time approximation algorithm, created in 1974, with a performance guarantee of $1/2$, is a greedy algorithm by Johnson [41], and the best theoretical result is of Karloff and Zwick [43], who gave a $7/8$ algorithm for $Max3SAT$ (variant, where at most three literals are allowed per clause). We do not expect to get any better than that, unless $P = NP$ [40].

Nevertheless, in many applications an exact solution is required. Similarly to SAT, we would also like to solve MaxSAT instances quickly in practical applications. In recent years, there has been considerable interest in developing efficient algorithms and several families of algorithms (MaxSAT-solvers) have been proposed. For recent survey, see [60].

In the context of this section, one particular method is of interest to us: the SAT-based approach. It was first developed by Le Berre and Parrain [53]. Given a MaxSAT instance $\psi = \{C_1, \ldots, C_m\}$, a new blocking variable $v_i$, $1 \leq i \leq m$, is added to each clause $C_i$, and solving the MaxSAT problem for $\psi$ is reduced to minimize the number of satisfied blocking variables in $\psi' = \{C_1 \vee v_1, \ldots, C_m \vee v_m\}$. Then, a SAT-solver that supports cardinality constraints (possibly by algorithms given in this thesis in Chapters 4–6) solves $\psi'$, and each time a satisfying assignment $A$ is found, a better satisfying assignment is searched by adding the cardinality constraint $v_1 + \cdots + v_m < B(A)$, where $B(A)$ is the number of blocking variables satisfied by $A$. Once $\psi'$ with a newly created cardinality constraint is unsatisfiable, the latest satisfying assignment is declared to be an optimal solution.

### 1.1.2 Progress in SAT-solving

The success of SAT-solving and its expanding interest comes from the fact that SAT stands at the crossroads of many fields, such as logic, graph theory, computer engineering and operations research. Thus, many problems with origin in one of these fields usually have multiple translations to SAT. Additionally, with plethora of ever-improving SAT-solvers available for both individual, academic and commercial use, one has almost limitless possibilities and modeling tools for solving a variety of scientific and practical problems.

The first program to be considered a SAT-solver was devised in 1960 by Davis and Putnam [32]. Their algorithm based on resolution is now called simply DP (Davis-Putnam). Soon after, an improvement to DP was proposed by Davis, Logemann and Loveland [31]. The new algorithm called DPLL (Davis-Putnam-Logemann-Loveland) guarantees linear worst-case space complexity.

The basic implementation of a SAT-solver consists of a backtracking algorithm that for a given formula $\psi$ chooses a literal, assigns a value to it (let us say true), and simplifies all the clauses containing that literal resulting in a new formula $\psi'$. Then, a recursive procedure checks the satisfiability of $\psi'$. If $\psi'$ is satisfied, then $\psi$ is also satisfied. Otherwise, the same recursive check is done assuming the opposite truth value (false) of the chosen literal. DPLL enhances this simple procedure by eagerly using *unit propagation* and *pure literal elimination* at every step of the algorithm. Unit propagation eliminates *unit clauses*, i.e., clauses that contain only a single unassigned literal. Such clause can be trivially satisfied by assigning the necessary value to make the literal true. Pure literals are variables that occur in the formula in only one polarity. Pure literals can always be assigned in a way that makes all clauses containing them true. Such clauses can be deleted from the formula without changing its satisfiability.

Most modern methods for solving SAT are refinements of the basic DPLL framework, and include improvements to variable choice heuristics or early pruning of the search space. In the DPLL algorithm it is unspecified how one should choose the next variable to process. Thus, many heuristics have emerged over the years. For example, the unit propagation rule was generalized to the *shortest clause rule*: choose a variable from a clause containing the fewest free literals [25]. Another example is the *majority rule* [24]: choose a variable with the maximum difference between the number of its positive and negative literals. Later, it was observed that the activity of variable assignment was an important factor in search space size. This led to the VSIDS (Variable State Independent Decaying Sum) heuristic of Chaff [61] that assigns to each variable a score proportional to the number of clauses that variable is in. As the SAT algorithm progresses, periodically, all scores are divided by a constant. VSIDS selects the next variable with the highest score to determine where the algorithm should branch.

Another refinement is a method of early detection of a sequence of variable decisions which results in a lot of propagation, and therefore reduction of the formula. The idea is to run the backtracking procedure for some constant number of steps and then rollback the calculations remembering the sequence of variable decisions that reduced the input formula the most. This way, the most promising parts of the search tree can be explored first, so the chance of finding a solution early increases. This framework has been established as *look-ahead* algorithms and the main representatives of this trend are the breath-first technique by Stålmarck [73] and depth-first technique (now called *restarts*) implemented in Chaff [61].

Arguably, the biggest step forward in the field of SAT-solving was construction of the CDCL algorithm (*conflict-driven clause-learning*). This turned many problems that were considered intractable into trivially solvable problems. The CDCL algorithm can greatly reduce the search space by discovering early that certain branches of the search tree do not need to be explored. In short, the goal of CDCL is to deduce new clauses when discovering a conflict (unsatisfiable clause) during the exploration of the search tree. Those clauses are constructed, so that the same conflict cannot be repeated. First solver to successfully apply this technique was Chaff [61]. Then in MiniSat [33] several improvements were implemented. Now, many more top solvers are built upon CDCL as the main heuristic, for example, Glucose [11].

After CDCL, no major improvement has been made. The current trend shifts toward parallelization of SAT-solvers. The related work on this topic can be found, for example, in the PhD thesis of Norbert Manthey [57]. See [19] for further reading on the subject of satisfiability.

## 1.2 Introduction to Constraint Programming

Problem formulations in Section 1.1.1 use expressions like: "**exactly one** queen" or "**at least two** waiters". These are examples of *constraints*, and more specifically – cardinality constraints. In the example with MaxSAT such constraints were defined explicitly. In this section we introduce the field of *Constraint Programming* and provide motivation behind its success. We base this section on parts of the book by Apt [8].

Informally, a *constraint* on a sequence of variables is a relation on their domains. It can be viewed as a requirement that states which combinations of values from the variable domains are allowed. To solve a given problem by means of constraint programming we express it as a *constraints satisfaction problem* (CSP), which consists of a finite set of constraints. To achieve it, we introduce a set of variables ranging over specific domains and constraints over these variables. Constraints are expressed in a specific language, for instance, in SAT variables range over the Boolean domain and the only constraints that the language allows are clauses.

**Example 1.1.** In the n-Queens puzzle, exactly one queen must be placed on each row and column, and at most one on each diagonal of the chessboard. This can be modeled as a CSP by introducing the following constraints:

- $Eq_1(x_1,\ldots,x_n)$ with semantics that exactly one out of $n$ propositional literals $\{x_1,\ldots,x_n\}$ can be true,
- $Lt_1(x_1,\ldots,x_n)$ stating that at most one out of $n$ propositional literals $\{x_1,\ldots,x_n\}$ can be true.

Returning to the example from Section 1.1.1, if $n = 4$ then we can express the problem using the following set of constraints:

$$
\begin{aligned}
&Eq_1(x_{a,1},x_{b,1},x_{c,1},x_{d,1}),Eq_1(x_{a,2},x_{b,2},x_{c,2},x_{d,2}),Eq_1(x_{a,3},x_{b,3},x_{c,3},x_{d,3}),Eq_1(x_{a,4},x_{b,4},x_{c,4},x_{d,4}),\\
&Eq_1(x_{a,1},x_{a,2},x_{a,3},x_{a,4}),Eq_1(x_{b,1},x_{b,2},x_{b,3},x_{b,4}),Eq_1(x_{c,1},x_{c,2},x_{c,3},x_{c,4}),Eq_1(x_{d,1},x_{d,2},x_{d,3},x_{d,4}),\\
&Lt_1(x_{a,2},x_{b,1}),Lt_1(x_{a,3},x_{b,2},x_{c,1}),Lt_1(x_{a,4},x_{b,3},x_{c,2},x_{d,1}),Lt_1(x_{b,4},x_{c,3},x_{d,2}),Lt_1(x_{c,4},x_{d,3}),\\
&Lt_1(x_{a,3},x_{b,4}),Lt_1(x_{a,2},x_{b,3},x_{c,4}),Lt_1(x_{a,1},x_{b,2},x_{c,3},x_{d,4}),Lt_1(x_{b,1},x_{c,2},x_{d,3}),Lt_1(x_{c,1},x_{d,2}).
\end{aligned}
$$

The CNF from Section 1.1.1 required 84 clauses. The above CSP is definitely more succinct.

The next step is to solve the CSP by a dedicated *constraint solver* which can incorporate various domain specific methods and general methods, depending on the type of constraints we are dealing with. The domain specific methods are usually provided in the form of implementations of special purpose algorithms, for example, a program that solves systems of linear equations, or a package for linear programming. On the other hand, the general methods are concerned with the ways of reducing the search space. These algorithms maintain equivalence while simplifying the considered problem. One of the aims of constraint programming is to search for efficient domain specific methods that can be used instead of the general methods and to apply them into a general framework. While solving CSPs we are usually interested in: determining whether the instance has a solution, finding all solutions, and finding all (or some) optimal solutions w.r.t. some goal function.

One of the advantages of modeling problems with constraints over the development of classic algorithms is the following. The classic computational problem is usually formulated in a very basic form and rarely can be applied to real-world situation as it is. In fact, the shorter the description, the more "fundamental" the problem feels, and the bigger interest in the community. For example: finding a matching in graph with some properties, the shortest path between nodes in graph, the minimum spanning tree, the minimum cut, the number of sub-words in compressed text, etc. In the real-world industrial applications, there are plenty of constraints to handle at once. The hope that someone would construct a classic algorithm to solve some complicated optimization problem we throw at them is bleak. We will now take a peek at couple of such problems.

### 1.2.1 Applications

Apt [8] provides a long list of problems where CSPs were successfully applied. The list consists of: interactive graphic systems, scheduling problems, molecular biology, business applications, electrical engineering, numerical computation, natural language processing, computer algebra. It has been 15 years since the book was published and more applications for constraint programming have emerged since then. One can find more recent (and more exotic) examples by studying the *Application Track* of the *Principles and Practice of Constraint Programming* conference. We reference some of them from the recent years here.

**Facade-Layout Synthesis.** This problem occurs when buildings are renovated to improve their thermal insulation and reduce the impact of heating on the environment. It involves covering a facade with a set of disjoint and configurable insulating panels. This can be viewed as a constrained rectangle packing problem for which the number of rectangles to be used and their size are not known *a priori*. Barco et al. [16] devise a CSP for the facade-layout problem. They point out that buildings energetic consumption represents more than one third of total energy consumption in developed countries, which provides great motivation for studying this problem.

**Differential Harvesting.** In grape harvesting, the machines (harvesters) are supplied with two hoppers – tanks that are able to differentiate between two types of grape quality. Optimizing harvest consists on minimizing the working time of a grape harvester. Given estimated qualities and quantities on the different areas of the vineyard, the problem is to optimize the routing of the harvester under several constraints. Briot et al. [22] model the differential harvest problem as a constraint optimization problem and present experimental results on real data.

**Transit Crew Rescheduling.** Scheduling urban and trans-urban transportation is an important issue for industrial societies. The urban transit crew scheduling problem is one of the most important optimization problems related to this issue. Lorca et al. [55] point out that this problem has been intensively studied from a tactical point of view, but the operational aspect has been neglected while the problem becomes more and more complex and prone to disruptions. In their paper, they present how the constraint programming technologies are able to recover the tactical plans at the operational level in order to efficiently help in answering regulation needs after disruptions.

**Reserve Design.** An interesting problem originates from the field of ecology – the delineation of areas of high ecological or biodiversity value. The selection of optimal areas to be preserved necessarily results from a compromise between the complexity of ecological processes and managers' constraints. A paper by Justeau-Allaire et al. [42] shows that constraint programming can be the basis of a unified, flexible and extensible framework for planning the reserve. They use their model on a real use-case addressing the problem of rainforest fragmentation in New Caledonia.

### 1.2.2 Types of Constraints

Over the years there have been many constraint types used in CSPs. This thesis studies *cardinality constraints* over Boolean domain. For $k \in \mathbb{N}$, $n$ propositional literals $\{x_1, \dots, x_n\}$ and a relation $\# \in \{<, \leq, =, \geq, >\}$, a cardinality constraints takes the form:

$$x_1 + x_2 + \cdots + x_n \# k.$$

Informally, it means that at least (at most, or exactly) $k$ out of $n$ propositional literals can be true. The main contribution of this thesis is the presentation of new methods to translate such constraints into CNF formulas. For the historical review on this subject we dedicate entire Chapter 3.

We are also interested in a closely related generalization of cardinality constraints called *Pseudo-Boolean constraints*, or PB-constraints, in short. We define them in a similar way, but with additional integer coefficients $\{a_1, \ldots, a_n\}$:

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n \# k.$$

From other types of constraints heavily studied by the community, we choose to mention the following:

- $all\_different(x_1, \ldots, x_n)$ – a constraint stating that each variable – defined with its own domain – need to be assigned a unique value.
- Linear inequalities over reals, i.e., the language of Linear Programming [71].

### 1.2.3 Clausal Encoding

There are many native CSP solvers available for different types of constraints. This thesis focuses on another approach in which we *encode* or *translate* a CSP (in our case a set of cardinality constraints) into a CNF formula. The generality and success of SAT-solvers in recent years has led to many CSPs being encoded and solved via SAT [7, 14, 15, 63, 69, 75]. The idea of encoding cardinality constraints into SAT is captured by the following definition.

**Definition 1.1** (clausal encoding). Let $k, n \in \mathbb{N}$. A clause set $E$ over variables $V = \{x_1, \ldots, x_n, s_1, \ldots, s_m\}$ is a *clausal encoding* of $x_1 + x_2 + \cdots + x_n \leq k$ if for all assignments $\alpha : \{x_1, \ldots, x_n\} \to \{0, 1\}$ there is an extension of $\alpha$ to $\alpha^* : V \to \{0, 1\}$ that satisfies $E$ if and only if $\alpha$ satisfies the original constraint $x_1 + x_2 + \cdots + x_n \leq k$, i.e., if and only if at most $k$ out of the variables $x_i$ are set to 1 by $\alpha$.

The similar notions can be defined for relations other than $\leq$ and other types of constraints, for example, Pseudo-Boolean constraints, but we omit that to avoid repetition.

The main idea in developing algorithms for solving constraint satisfaction problems is to reduce a given CSP to another one that is equivalent but easier to solve. The process is called *constraint propagation* and the algorithms that achieve this reduction are called *constraint propagation algorithms*. In case of SAT-solving we use *unit propagation*.

Informally, Unit Propagation (UP) is a process, where for a given CNF formula and a partial assignment (initially – empty), clauses are sought in which all literals but one are false (say $l$) and $l$ is undefined (initially only clauses of size one satisfy this condition). This literal $l$ is set to true and the process is iterated until a fix point is reached. A formal definition is presented in Chapter 2.

We try to construct encodings that guarantee better propagation of values using unit propagation. We use is the notion of *general arc-consistency* (GAC), often shortened to just *arc-consistency* (which usually has different meaning in CP theory and deals with binary constraints). We use this notion in the context of cardinality constraints and unit propagation in SAT. Informally, an encoding of $x_1 + x_2 + \cdots + x_n \leq k$ is arc-consistent if as soon as $k$ input variables are fixed to 1, unit propagation will fix all other input variables to 0. A formal definition is presented in Chapter 2. If the encoding is arc-consistent, then this has a positive impact on the practical efficiency of SAT-solvers.

## 1.3 Thesis Contribution and Organization

At the highest level, the thesis improves the state-of-the-art of encoding cardinality constraints by introducing several algorithms based on selection networks. The structure of the thesis and the contribution is the following. The first part consists of three chapters, one of them is this introduction, and the other two are:

- In Chapter 2 we introduce the necessary definitions, notation and conventions used throughout the thesis. We define comparator networks and introduce basic constructions used in encoding of cardinality constraints. We give the definition of a *standard encoding* of cardinality constraints. We show how a single comparator can be encoded using a set of clauses and how to generalize a comparator to directly select $m$ elements from $n$ inputs. This is the main building block of our fastest networks presented here – the generalized selection networks. We also present a proof of arc-consistency for all encodings presented in the later parts. In fact, we give the first rigorous proof of a more stronger statement, that any standard encoding based on generalized selection networks preserves arc-consistency. There are several results where researchers use properties of their constructions to prove arc-consistency, which is usually long and technical (see, for example, [3, 10, 27]). In [46] we relieve some of this burden by proving that the standard encoding of any selection network preserves arc-consistency. Here we generalize our previous proof to the extended model of selection networks.
- In Chapter 3 we present a historic review of methods for translating cardinality constraints and Pseudo-Boolean constraints into SAT.

The second part consists of two chapters dedicated to the pairwise selection networks:

- We begin Chapter 4 with the presentation of the *Pairwise Selection Network* (PSN) by Codish and Zazon-Ivry [27], which is based on the *Pairwise Sorting Network* by Parberry [65]. The goal of this chapter is to improve the PSN by introducing two new classes of selection networks called *Bitonic Selection Networks* and *Pairwise (Half-)Bitonic Selection Networks*. We prove that we have produced a smaller selection network in terms of the number of comparators. We estimate also the size of our networks and compute the difference in sizes between our selection networks and the PSN. The difference can be as big as $n \log n/2$ for $k = n/2$.
- In Chapter 5 we show construction of the *m-Wise Selection Network*. This is the first attempt at creating an encoding that is based on the generalized selection networks introduced in Chapter 2. The algorithm uses the same *pairwise* idea as in Chapter 4, but the main difference is that the basic component of the new network is an $m$-selector (for $m \geq 2$). The new algorithm works as follows. The inputs are organized into $m$ columns, in which elements are recursively selected and, after that, columns are merged using a dedicated merging network. The construction can be directly applied to any values of $k$ and $n$ (in contrast to the previous algorithms from Chapter 4). We show the high-level algorithm for any $m$, but we only present a complete construction (which includes a merging network) for $m = 4$ for theoretical evaluation, where we prove that using 4-column merging networks produces smaller encodings than their 2-column counterpart.

The third part focuses on the generalized version of the odd-even selection networks. Here we show our best construction for encoding cardinality constraints, as well as the description of a PB-solver based on the same algorithm:

- In Chapter 6 we show more efficient construction using generalized comparator networks model. We call our new network the *m-Odd-Even Selection Network*. It generalizes the standard odd-even algorithm similarly to how *m-Wise Selection Network* generalizes PSN in Chapter 5. The inputs are organized into $m$ columns, and after recursive calls, resulting elements are merged using a dedicated merging network. The calculations show that encodings based on our merging networks use less number of variables and clauses, when compared to the classic 2-column construction. In addition, we investigate the influence of our encodings on the execution times of SAT-solvers to be sure that the new algorithms can be used in practice. We show that generalized comparator networks are superior to standard selection networks previously proposed in the literature, in the context of translating cardinality constraints into propositional formulas. We also conclude that although encodings based on pairwise approach use less number of variables than odd-even encodings, in practice, it is the latter that achieve better reduction in SAT-solving runtime. It is a helpful observation, because from the practical point of view, implementing odd-even networks is a little bit easier.

- In Chapter 7 we explore the possibility of using our algorithms in encoding Pseudo-Boolean constraints, which are more expressive than simple cardinality constraints. We describe the system for solving PB-problems based on the popular MINISAT+ solver by Eén and Sörensson [34]. Recent research have favored the approach that uses Binary Decision Diagrams (BDDs), which is evidenced by several new constructions and optimizations [2, 70]. We show that encodings based on comparator networks can still be very competitive. We have extended MINISAT+ by adding a construction of selection network called 4-Way Merge Selection Network (an improved version of 4-Odd-Even Selection Network from Chapter 6), with a few optimizations based on other solvers. In Chapter 6 we show a top-down, divide-and-conquer algorithm for constructing 4-Odd-Even Selection Network. The difference in our new implementation is that we build our network in a bottom-up manner, which results in the easier and cleaner implementation. Experiments show that on many instances of popular benchmarks our technique outperforms other state-of-the-art PB-solvers.

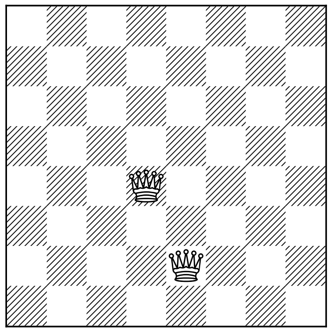
- Finally, in the last chapter we summarize the results presented in this thesis and show the possibilities for future work.

The contributions mentioned above are based on several scientific papers which are mostly the joint work of the author and the supervisor ([44–48]). Note that we do not explore the topic of SAT computation itself. Although the use of constantly improving SAT-solvers is an inseparable part of our work, we focus solely on translating constraints into CNFs. The advantage of such approach is that our techniques are not bound by the workings of a specific solver. As new, faster SAT-solvers are being produced, we can just swap the one we use in order to get better results.

We would like to point out that our constructions are not optimal in the sense of computational complexity. Our algorithms are based on classical sorting networks which use $O(n\log^2 n)$ comparators. From the point of view of the $O$ notation, we do not expect to breach this barrier. We are aware of an optimal sorting network which uses only $O(n\log n)$ comparators by the celebrated result of Ajtai, Komlós and Szemerédi [5], but the constants hidden in $O(n\log n)$ are so large, that CNF encodings based on such networks would never be used in practice – and this is a very important point for the Constraint Programming community. There has been many improvements to the original $O(n\log n)$ sorting network [39, 66, 68], but at the time of writing of this thesis none is yet applicable to encoding of constraints.

# Chapter 2

# Preliminaries



In this chapter we introduce definitions and notations used in the rest of the thesis. We take a special care in introducing comparator networks, as they are the central mechanism in all our encodings. To this end we present several different approaches to define comparator networks and explain the choices of comparator models we make for presenting our main results. As an example, we show different ways to construct a classic odd-even sorting network by Batcher [17]. Next, we present how to encode a comparator network into a set of clauses and how to make encodings of sorting (selection) networks enforce cardinality constraints. Then, we extend the model of comparator network so that the atomic operation does not handle only two inputs, but any fixed number of inputs. This way we construct networks which we call *Generalized Selection Networks* (GSNs). Finally, we show that any encoding of cardinality constraints based on GSNs preserves arc-consistency.

## 2.1 The Basics

Let $X$ denote a totally ordered set, for example the set of natural numbers $\mathbb{N}$ or the set of binary values $\{0,1\}$. We introduce the auxiliary "smallest" element $\perp \notin X$, such that for all $x \in X$ we have $\perp < x$. Thus, $X \cup \{\perp\}$ is totally ordered. The element $\perp$ is used in the later chapters to simplify presentation of algorithms.

**Definition 2.1** (sequences)**.** A sequence of length $n$, say $\bar{x} = \langle x_1, \dots, x_n \rangle$, is an element of $X^n$. In particular, an element of $\{0,1\}^n$ is called a *binary* sequence. Length of a sequence is denoted by $|\bar{x}|$. We say that a sequence $\bar{x} \in X^n$ is *sorted* if $x_i \geq x_{i+1}$, $1 \leq i < n$. Given two sequences $\bar{x} = \langle x_1, \dots, x_n \rangle$ and $\bar{y} = \langle y_1, \dots, y_m \rangle$ we define several operations and notations:

- *concatenation* as $\bar{x} :: \bar{y} = \langle x_1, \dots, x_n, y_1, \dots, y_m \rangle$,
- *domination* relation: $\bar{x} \succeq \bar{y} \iff \forall_{i \in \{1,\dots,n\}} \forall_{j \in \{1,\dots,m\}}\, x_i \geq y_j$,
- *weak domination* relation (if $n = m$): $\bar{x} \succeq_w \bar{y} \iff \forall_{i \in \{1,\dots,n\}} x_i \geq y_i$,
- $\bar{x}_{odd} = \langle x_1, x_3, \dots \rangle$, $\bar{x}_{even} = \langle x_2, x_4, \dots \rangle$,
- $\bar{x}_{a,\dots,b} = \langle x_a, \dots, x_b \rangle$, $1 \leq a \leq b \leq n$,
- $\bar{x}_{left} = \bar{x}_{1,\dots,\lfloor n/2 \rfloor}$, $\bar{x}_{right} = \bar{x}_{\lfloor n/2 \rfloor + 1,\dots,n}$,
- *prefix/suffix* operators: $\mathrm{pref}(i, \bar{x}) = \bar{x}_{1,\dots,i}$ and $\mathrm{suff}(i, \bar{x}) = \bar{x}_{i,\dots,n}$, $1 \leq i \leq n$,

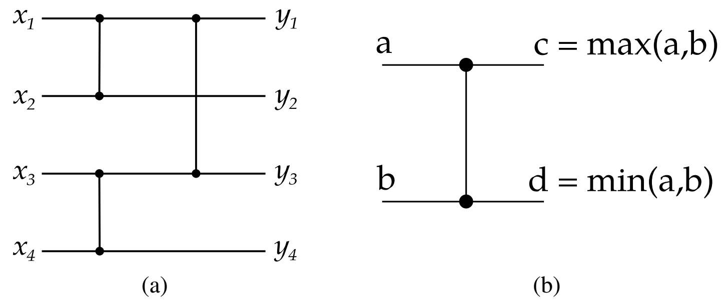



Figure 2.1: a) An example of comparator network; b) a single comparator

- the number of occurrences of a given value $b$ in $\bar{x}$ is denoted by $|\bar{x}|_b$,
- and the result of removing all occurrences of $b$ in $\bar{x}$ is written as $\text{drop}(b, \bar{x})$.

**Definition 2.2** (top k sorted sequence)**.** A sequence $\bar{x} \in X^n$ is top $k$ sorted, with $k \leq n$, if $\langle x_1, \ldots, x_k \rangle$ is sorted and $\langle x_1, \ldots, x_k \rangle \succeq \langle x_{k+1}, \ldots, x_n \rangle$.

**Definition 2.3** (bitonic sequence)**.** A sequence $\bar{x} \in X^n$ is a bitonic sequence if $x_1 \leq \ldots \leq x_i \geq \ldots \geq x_n$ for some $i$, where $1 \leq i \leq n$, or a circular shift of such sequence. We distinguish a special case of a bitonic sequence:

- *v-shaped*, if $x_1 \geq \ldots \geq x_i \leq \ldots \leq x_n$

and among v-shaped sequences there are two special cases:

- *non-decreasing*, if $x_1 \leq \ldots \leq x_n$,
- *non-increasing*, if $x_1 \geq \ldots \geq x_n$.

**Definition 2.4** (zip operator)**.** For $m \geq 1$ given sequences (column vectors) $\bar{x}^i = \langle x^i_1, \ldots, x^i_{n_i} \rangle$, $1 \leq i \leq m$ and $n_1 \geq n_2 \geq \cdots \geq n_m$, let us define the zip operation that outputs the elements of the vectors in row-major order:

$$\text{zip}(\bar{x}^1, \ldots, \bar{x}^m) = \begin{cases} \bar{x}^1 & \text{if } m = 1 \\ \text{zip}(\bar{x}^1, \ldots, \bar{x}^{m-1}) & \text{if } |\bar{x}^m| = 0 \\ \langle x^1_1, x^2_1, \ldots, x^m_1 \rangle :: \text{zip}(\bar{x}^1_{2,\ldots,n_1}, \ldots, \bar{x}^m_{2,\ldots,n_m}) & \text{otherwise} \end{cases}$$

## 2.2 Comparator Networks

We construct and use comparator networks in this thesis. Traditionally comparator networks are presented as circuits that receive $n$ inputs and permute them using comparators (2-sorters) connected by "wires". Each comparator has two inputs and two outputs. The "upper" output is the maximum of inputs, and the "lower" one is the minimum. The standard definitions and properties of them can be found, for example, in [50]. The only difference is that we assume that the output of any sorting operation or comparator is in a non-increasing order. We begin with presenting different ways to model comparator networks and explain strengths and weaknesses of such models. We use the network from Figure 2.1a as a running example.

### 2.2.1 Functional Representation

In our first representation we model comparators as functions and comparator networks as a composition of comparators. This way networks can be represented in a clean, strict way.

**Definition 2.5** (comparator as a function)**.** Let $\bar{x} \in X^n$ and let $i, j \in \mathbb{N}$, where $1 \leq i < j \leq n$. A comparator is a function $c^n_{i,j}$ defined as:

$$c^n_{i,j}(\bar{x}) = \bar{y} \iff y_i = \max\{x_i, x_j\} \wedge y_j = \min\{x_i, x_j\} \wedge \forall_{k \neq i,j}\, x_k = y_k$$

> **Example 2.1.** Notice that a comparator is defined as a function of the type $X^n \to X^n$, that is, it takes a sequence of length $n$ as an input and outputs the same sequence with at most one pair of elements swapped. For example, let $X = \{0,1\}$ and $\bar{x} = \langle 1,1,0,0,1 \rangle$. Then $c^5_{3,5}(\bar{x}) = \langle 1,1,1,0,0 \rangle$.

**Definition 2.6** (comparator network as a composition of functions)**.** We say that $f^n : X^n \to X^n$ is a comparator network of order $n$, if it can be represented as the composition of finite number of comparators, namely, $f^n = c^n_{i_1,j_1} \circ \cdots \circ c^n_{i_k,j_k}$. The number of comparators in a network is denoted by $|f^n|$. Comparator network of size 0 is denoted by $id^n$.

> **Example 2.2.** Figure 2.1a is an example of a simple comparator network consisting of 3 comparators. It outputs the maximum from 4 inputs on the top horizontal line, namely, $y_1 = \max\{x_1, x_2, x_3, x_4\}$. Using our notation, we can say that this comparator network is constructed by composing three comparators: $\max^4 = c^4_{1,3} \circ c^4_{3,4} \circ c^4_{1,2}$.

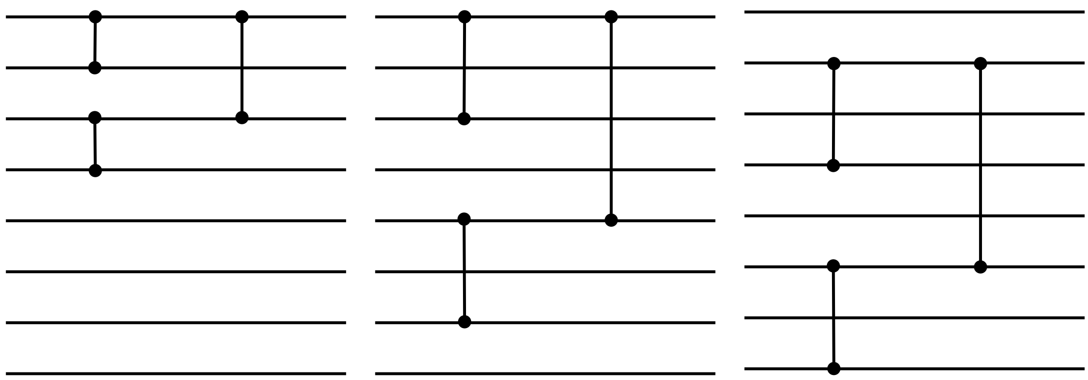
Most comparator networks which are build for sorting (selection) purposes are presented using divide-and-conquer paradigm. In particular, the classic sorting networks are variations of the merge-sort algorithm. In such setup, to present comparator networks as composition of functions, we require that all smaller networks created using recursive calls be of the same order as the resulting one. In other words, each of them has to have same number of inputs. In order to achieve that goal, we define a *rewire* operation. Let $[n] = \{1, \ldots, n\}$, for any $n \in \mathbb{N}$. Rewire is a map $\rho : [m] \to [n]$ ($m < n$) that is a monotone injection. We use $\rho^*$ as a map from order $m$ comparator network to order $n$ comparator network, applying $\rho$ to each comparator in the network, that is, $\rho^*(c^m_{i_1,j_1} \circ \cdots \circ c^m_{i_k,j_k}) = c^n_{\rho(i_1),\rho(j_1)} \circ \cdots \circ c^n_{\rho(i_k),\rho(j_k)}$. Some useful rewirings are presented below (rewirings of type $[n] \to [2n]$). Examples of how they work are shown in Figure 2.2.

$$left(f^n) = \rho^*_1(f^n) \quad (\text{where } \rho_1(i) = i), \qquad right(f^n) = \rho^*_2(f^n) \quad (\text{where } \rho_2(i) = n + i)$$

$$odd(f^n) = \rho^*_3(f^n) \quad (\text{where } \rho_3(i) = 2i - 1), \qquad even(f^n) = \rho^*_4(f^n) \quad (\text{where } \rho_4(i) = 2i)$$

**Definition 2.7** (k-selection network)**.** A comparator network $sel^n_k$ is a *k-selection network* (of order $n$), if for each $\bar{x} \in X^n$, $sel^n_k(\bar{x})$ is top $k$ sorted.

Notice that by the definition of a selection network, $sort^n = sel^n_n$ is a *sorting network* (of order $n$), that is, for each $\bar{x} \in X^n$, $sort^n(\bar{x})$ is sorted.



(a) $left(max^4)$ (b) $odd(max^4)$ (c) $even(max^4)$

Figure 2.2: Rewirings of comparator network $max^4$ to order 8 comparator networks

**Example 2.3.** Let us try to recreate the odd-even sorting network by Batcher [17] using the functional representation. To simplify the presentation we assume that $n$ is a power of 2 and $X = \{0,1\}$.
Construction of the *odd-even* sorting network uses the idea of merge-sort algorithm: in order to sort a list of $n$ elements, first partition the list into two lists (each of size $n/2$), recursively sort those lists, then merge the two sorted lists. Using functional representation it looks like this:

$$oe\_sort^1 = id^1$$
$$oe\_sort^n = merge^n \circ left(oe\_sort^{n/2}) \circ right(oe\_sort^{n/2})$$

Network $merge^n$ merges two sorted sequences into one sorted sequence, that is: if $\bar{x} \in \{0,1\}^{n/2}$ and $\bar{y} \in \{0,1\}^{n/2}$ are both sorted, then $merge^n(\bar{x} :: \bar{y})$ is also sorted. Notice that when unfolding the recursion, the sorting network can be viewed as a composition of mergers, so we only need to specify how a single merger is constructed. In the *odd-even* approach, merger uses the idea of *balanced sequences*. Sequence $\bar{x} \in \{0,1\}^n$ is called balanced, if $\bar{x}_{odd}$ and $\bar{x}_{even}$ are sorted and $0 \leq |\bar{x}_{odd}|_1 - |\bar{x}_{even}|_1 \leq 2$. We define a balanced merger $bal\_merge^n$ that sorts a given balanced sequence. The balanced merger can be constructed in a straightforward way:

$$bal\_merge^2 = id^2$$
$$bal\_merge^n = c^n_{2,3} \circ c^n_{4,5} \circ \cdots \circ c^n_{n-2,n-1}$$

Finally, we get the merger for the odd-even sorting network:

$$oe\_merge^2 = c^2_{1,2}$$
$$oe\_merge^{2n} = bal\_merge^{2n} \circ odd(oe\_merge^n) \circ even(oe\_merge^n)$$

The number of comparators used in the odd-even sorting network is:

$$|oe\_sort^n| = \frac{1}{4}n(\log n)(\log n - 1) + n - 1.$$

Functional representation allows for comparator networks to be presented in a formal, rigorous way. As seen in the example above, algorithms written in such form are very clean. One can also see the work of Codish and Zazon-Ivry [78] to confirm that proofs of properties of such networks are – in certain sense – elegant. Unfortunately this is only true if the structure of the algorithm is simple, like in the odd-even sorting network. Networks presented in the later chapters are more complex and therefore need a more practical representation.

### 2.2.2 Declarative Representation

Another approach is to view comparators as relations on their inputs and outputs. Figure 2.1b depicts a single comparator with inputs $a$ and $b$, and outputs $c$ and $d$.

**Definition 2.8** (comparator as a relation)**.** Let $a,b,c,d \in X$. A comparator is a relation defined as:

$$comp(\langle a,b\rangle,\langle c,d\rangle) \Longleftrightarrow c = \max(a,b) \wedge d = \min(a,b)$$

**Definition 2.9** (comparator network as a relation)**.** Let $\bar{x} \in X^n$, $\bar{y} \in X^m$ and $\bar{z} \in X^p$. We say that a relation $net(\bar{x},\bar{y},\bar{z})$ is a comparator network, if it is the conjunction of a finite number of comparators, namely, $net(\bar{x},\bar{y},\bar{z}) = comp(\bar{a}_1,\bar{b}_1) \wedge \cdots \wedge comp(\bar{a}_k,\bar{b}_k)$, where elements from $\bar{x}$ appear exactly once in $\bar{a}_1 :: \cdots :: \bar{a}_k :: \bar{y}$, elements from $\bar{y}$ appear exactly once in $\bar{b}_1 :: \cdots :: \bar{b}_k :: \bar{x}$, and elements from $\bar{z}$ appear exactly once in $\bar{b}_1 :: \cdots :: \bar{b}_k$ and at most once in $\bar{a}_1 :: \cdots :: \bar{a}_k$.

In the definition above, for readability we split the sequence of parameters into three groups: $\bar{x}$ are inputs of the network, $\bar{y}$ are outputs, and $\bar{z}$ are auxiliary elements, which can appear once as an input in some comparator and once as an output in some other comparator.

**Example 2.4.** In the declarative representation, a comparator network from Figure 2.1a can be written as:

$$\begin{aligned} &max(\langle x_1,x_2,x_3,x_4\rangle,\langle y_1,y_2,y_3,y_4\rangle,\langle z_1,z_2\rangle) = \\ &\quad = comp(\langle x_1,x_2\rangle,\langle z_1,y_2\rangle) \wedge comp(\langle x_3,x_4\rangle,\langle z_2,y_4\rangle) \wedge comp(\langle z_1,z_2\rangle,\langle y_1,y_3\rangle) \end{aligned}$$

Definition 2.9 implies that comparator networks can be written in terms of propositional logic, where the only non-trivial component is a comparator. We can see the advantage over functional approach – we do not need rewiring operations to specify networks.

**Example 2.5.** Let us again see the construction of the odd-even sorting network, this time using declarative representation. The entire algorithm can be written as a relation *oe_sort* described below. To simplify the presentation we assume that $n$ is a power of 2.

$$\begin{aligned} &oe\_sort(\langle x\rangle,\langle x\rangle,\langle\rangle), \\ &oe\_sort(\langle x_1,\ldots,x_{2n}\rangle,\langle y_1,\ldots,y_{2n}\rangle,\langle z_1,\ldots,z_{2n}\rangle :: S_1 :: S_2 :: M) \;= \\ &\qquad oe\_sort(\langle x_1,\ldots,x_n\rangle,\langle z_1,\ldots,z_n\rangle,S_1) \wedge \\ &\qquad oe\_sort(\langle x_{n+1},\ldots,x_{2n}\rangle,\langle z_{n+1},\ldots,z_{2n}\rangle,S_2) \wedge \\ &\qquad oe\_merge(\langle z_1,\ldots,z_n,z_{n+1},\ldots,z_{2n}\rangle,\langle y_1,\ldots,y_{2n}\rangle,M). \end{aligned}$$

$$
\begin{aligned}
&oe\_merge(\langle x_1,x_2\rangle,\langle y_1,y_2\rangle,\langle\rangle) = comp(\langle x_1,x_2\rangle,\langle y_1,y_2\rangle),\\
&oe\_merge(\langle x_1,\dots,x_{2n}\rangle,\langle y_1,\dots,y_{2n}\rangle,\langle z_1,z'_1,\dots,z_n,z'_n\rangle :: M_1 :: M_2) \;=\\
&\qquad oe\_merge(\langle x_1,x_3,\dots,x_{2n-1}\rangle,\langle z_1,\dots,z_n\rangle,M_1) \wedge\\
&\qquad oe\_merge(\langle x_2,x_4,\dots,x_{2n}\rangle,\langle z'_1,\dots,z'_n\rangle,M_2) \wedge\\
&\qquad bal\_merge(\langle z_1,z'_1,\dots,z_n,z'_n\rangle,\langle y_1,\dots,y_{2n}\rangle).
\end{aligned}
$$

$$
\begin{aligned}
&bal\_merge(\langle x_1,x_2\rangle,\langle x_1,x_2\rangle),\\
&bal\_merge(\langle x_1,\dots,x_n\rangle,\langle x_1\rangle :: \langle y_2,\dots,y_{n-1}\rangle :: \langle x_n\rangle) \;=\\
&\qquad comp(\langle x_2,x_3\rangle,\langle y_2,y_3\rangle) \wedge\cdots\wedge comp(\langle x_{n-2},x_{n-1}\rangle,\langle y_{n-2},y_{n-1}\rangle).
\end{aligned}
$$

The semantics of $sort(\bar{x},\bar{y},\bar{z})$ is that $\bar{y}$ is sorted and is a permutation of $\bar{x}$, while $\bar{z}$ is a sequence of auxiliary elements.

Networks given so far have been presented in a formal, rigorous way, but as we will see in further chapters, it is easier to reason about and prove properties of comparator networks when presented as pseudo-code or a list of procedures. We remark that it is possible to restate all the algorithms presented in this thesis in the form of composition of functions or as relations, but this would vastly complicate most of the proofs, which are already sufficiently formal.

### 2.2.3 Procedural Representation

Although functional and declarative approaches has been used in the context of encoding cardinality constraints (functional in [78], declarative in [9]), they are not very practical, as modern solvers are written in general-purpose, imperative, object-oriented programming languages like Java or C++. In order to make the implementation of the algorithms presented in this thesis easier (and to simplify the proofs), we propose the procedural representation of comparator networks [45, 47, 48], in which a network is an algorithm written in pseudo-code with a restricted set of operations. What we want, for a given network, is a procedure that generates its declarative representation, i.e., the procedure should define a set of comparators. At the same time we want to treat such procedures as oblivious sorting algorithms, so that we can easily prove their correctness. We explain our approach with the running example.

**Algorithm 2.1** $max^4$

**Input:** $\langle x_1,x_2,x_3,x_4\rangle \in X^4$
**Ensure:** The output is top 1 sorted and is a permutation of the inputs
$\bar{z} \leftarrow sort^2(x_1,x_2)$
$\bar{z'} \leftarrow sort^2(x_3,x_4)$
**return** $sort^2(z_1,z'_1) :: \langle z_2,z'_2\rangle$

**Example 2.6.** A comparator network from Figure 2.1a can be expressed as pseudo-code like in Algorithm 2.1. The algorithm showcases several key properties of our representation:

- The only allowed operation that compares elements is $sort^2$, which puts two given elements in non-increasing order.
- New sequences can be defined and assigned values, for example, as a result of $sort^2$ or a sub-procedure.
- The algorithm returns a sequence that is a permutation of the input sequence in order to highlight the fact that it represents a comparator network. For example, in Algorithm 2.1 we concatenate $\langle z_2, z'_2 \rangle$ to the returned sequence.

**Algorithm 2.2** $oe\_sort^n$

**Input:** $\bar{x} \in \{0,1\}^n$; $n$ is a power of 2
**Ensure:** The output is sorted and is a permutation of the inputs

**if** $n = 1$ **then return** $\bar{x}$
$\bar{y} \leftarrow oe\_sort^{n/2}(\bar{x}_{left})$
$\bar{y}' \leftarrow oe\_sort^{n/2}(\bar{x}_{right})$
**return** $oe\_merge^n(\bar{y}, \bar{y}')$

**Algorithm 2.3** $oe\_merge^n$

**Input:** $\bar{x}^1, \bar{x}^2 \in \{0,1\}^{n/2}$; $\bar{x}^1$ and $\bar{x}^2$ are sorted; $n$ is a power of 2
**Ensure:** The output is sorted and is a permutation of the inputs

**if** $n = 2$ **then return** $sort^2(x^1_1, x^2_1)$
$\bar{y} \leftarrow oe\_merge^{n/2}(\bar{x}^1_{odd}, \bar{x}^2_{odd})$
$\bar{y}' \leftarrow oe\_merge^{n/2}(\bar{x}^1_{even}, \bar{x}^2_{even})$
$z_1 \leftarrow y_1$; $z_n \leftarrow y'_n$
**for all** $i \in \{1, \dots, n/2 - 1\}$ **do** $\langle z_{2i}, z_{2i+1} \rangle \leftarrow sort^2(y'_i, y_{i+1})$
**return** $\bar{z}$

**Example 2.7.** The odd-even sorting network is presented in Algorithm 2.2. It uses Algorithm 2.3 – the merger – as a sub-procedure. From this example we can also see that the code convention allows for loops, conditional statements and recursive calls, but they cannot depend on the results of comparisons between elements in sequences.

## 2.3 Encoding Cardinality Constraints

What we want to achieve using comparator networks, is to produce a clausal encoding for a given cardinality constraint. Notice that a cardinality constraint in Definition 1.1 is defined in terms of "$\leq$" relation. We now show that it is in fact the only type of relation we need to be concerned about.

**Observation 2.1.** *Let $k, n \in \mathbb{N}$ where $k \leq n$, and let $\langle x_1, \dots, x_n \rangle$ be a sequence of Boolean literals. Then:*

1. $x_1 + x_2 + \cdots + x_n \geq k$ *is eqivalent to* $\neg x_1 + \neg x_2 + \cdots + \neg x_n \leq n - k$*,*

2. $x_1 + x_2 + \cdots + x_n > k$ *is eqivalent to* $\neg x_1 + \neg x_2 + \cdots + \neg x_n \leq n - k - 1$*,*

3. $x_1 + x_2 + \cdots + x_n < k$ *is equvalent to* $x_1 + x_2 + \cdots + x_n \leq k - 1$*,*

4. $x_1 + x_2 + \cdots + x_n = k$ *is equvalent to* $x_1 + x_2 + \cdots + x_n \leq k$ *and* $\neg x_1 + \neg x_2 + \cdots + \neg x_n \leq n - k$*.*

As we can see, in every case we can reduce the cardinality constraint to the equivalent one which uses the "$\leq$" relation. In case of equality relation this produces two cardinality constraints, but then we handle them (encode them) separately. Therefore from now on, when we mention a cardinality constraint, we mean the one in the form:

$$x_1 \ + \ x_2 \ + \ \ldots \ + \ x_n \ \leq \ k.$$

We now describe how to translate a cardinality constraint into equisatisfiable set of clauses. We begin with an encoding of a single comparator. If we use the notation as in Figure 2.1b, then the maximum on the upper output is translated into a disjunction of the inputs $(a \vee b)$ and the minimum on the lower output is translated into a conjunction of the inputs $(a \wedge b)$. Therefore a single comparator with inputs $\langle a, b \rangle$ and outputs $\langle c, d \rangle$ can be encoded using the formula:

$$(c \Leftrightarrow a \vee b) \wedge (d \Leftrightarrow a \wedge b),$$

which is equivalent to the following six clauses:

$$\begin{array}{lll} a \Rightarrow c & \qquad a \wedge b \Rightarrow d & \qquad d \Rightarrow a \\ b \Rightarrow c & \qquad c \Rightarrow a \vee b & \qquad d \Rightarrow b \end{array}$$

Encoding consists of translating every comparator into a set of clauses. Thus different representations of a comparator network in the previous section can be viewed as different procedures that outputs the same set of comparators. Since the cardinality constraints are over Boolean domain, we are using comparator networks in the context of Boolean formulas, therefore we limit the domain of the inputs to 0-1 values.

The clausal encoding of cardinality constraints is defined as follows. First, we build a sorting network with inputs $\langle x_1, \ldots, x_n \rangle$ and outputs $\langle y_1, \ldots, y_n \rangle$. Since it is a sorting network, if exactly $k$ inputs are set to 1, we will have $\langle y_1, \ldots, y_k \rangle$ set to 1 and $\langle y_{k+1}, \ldots, y_n \rangle$ set to 0. Therefore, in order to enforce the constraint $x_1 + x_2 + \cdots + x_n \leq k$, we need to do two things: translate each comparator into a set of clauses, and add a unit clause $\neg y_{k+1}$, which virtually sets $y_{k+1}$ to 0. This way the resulting CNF is satisfiable if and only if at most $k$ input variables are set to 1.

Two improvements to this basic encoding can be made. Notice that we do not need to sort the entire input sequence, but only the first $k+1$ largest elements. Therefore rather than using sorting networks, we can use *selection networks*.

**Example 2.8.** The first selection network used in the context of encoding cardinality constraints is called *Pairwise Selection Network* by Codish and Zazon-Ivry [78]. They construct selection networks recursively, just like sorting networks: in order to get top $k$ sorted sequence, first we split input in half and get top $k$ sorted sequences for both parts, then we merge the results. In reality, the only thing we are actually doing is cutting out "unnecessary" comparators from sorting networks. In Figure 2.3 we present odd-even and pairwise selection networks for $n = 8$ and $k = 3$. Removed comparators from sorting networks are marked with dashed lines. Notice that with pairwise approach we save 5 comparators, where with odd-even it is only 4.

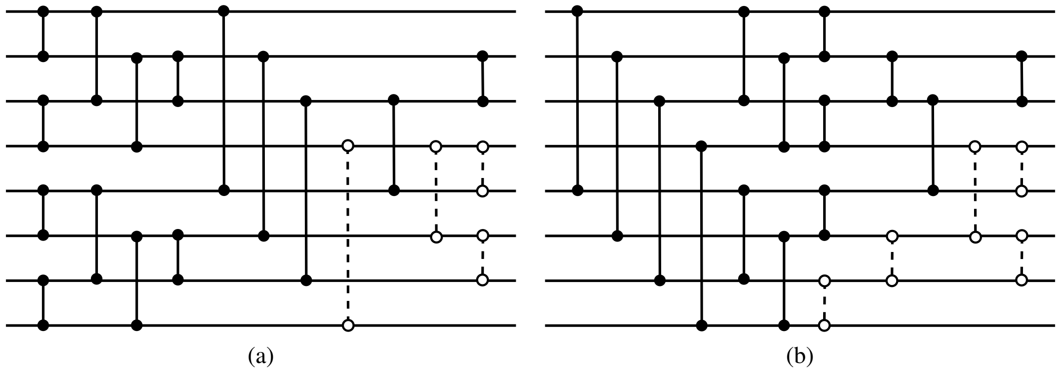



Figure 2.3: a) Odd-Even Selection Network; b) Pairwise Selection Network; $n = 8$, $k = 3$

Another improvement lies in the encoding of a single comparator. In the encoding of $x_1 + x_2 + \cdots + x_n \leq k$ one can use 3 out of 6 clauses to encode a single comparator:

$$a \Rightarrow c, \quad b \Rightarrow c, \quad a \wedge b \Rightarrow d. \tag{2.1}$$

Notice that this encoding only guarantees that if at least $i \in \{0,1,2\}$ inputs in a single comparator are set to 1, then at least $i$ top outputs must be set to 1. This generalizes to the entire network, that is, if at least $k$ inputs of a selection network are set to 1, then at least $k$ top outputs must be set to 1. Although this set of clauses are not equivalent to the comparator of Definition 2.5, it is enough to be used in the encoding of $x_1 + x_2 + \cdots + x_n \leq k$ while still maintaining the arc-consistency (Definition 2.12). Thanks to that, we can use half as many clauses to encode a network, which significantly reduces the size of an instance passed to a SAT-solver.

### 2.3.1 Generalized Selection Networks

The encoding of cardinality constraints using comparator networks has been known for some time now. In [3, 10, 27, 34] authors are using sorting (selection) networks to encode cardinality constraints in the same way as we: inputs and outputs of a comparator are Boolean variables and comparators are encoded as a CNF formula. In addition, the $(k+1)$-th greatest output variable $y_{k+1}$ of the network is forced to be 0 by adding $\neg y_{k+1}$ as a clause to the formula that encodes $x_1 + \cdots + x_n \leq k$. The novelty in most of our constructions is that rather than using simple comparators (2-sorters), we also use comparators of higher order as building blocks ($m$-sorters, for $m \geq 2$). Since a sorter is just a special case of a selector, so we only need one definition. For a selector we want to output top $m$ sorted elements from the inputs. The following definition captures this notion.

**Definition 2.10** (m-selector of order n)**.** Let $n, m \in \mathbb{N}$, where $m \leq n$. Let $\bar{x} = \langle x_1, \ldots, x_n \rangle$ and $\bar{y} = \langle y_1, \ldots, y_m \rangle$ be sequences of Boolean variables. The Boolean formula $s^n_m(\bar{x}, \bar{y})$ which consists of the set of clauses $\{x_{i_1} \wedge \cdots \wedge x_{i_p} \Rightarrow y_p : 1 \leq p \leq m, 1 \leq i_1 < \cdots < i_p \leq n\}$ is an *m-selector* of order $n$.

Notice that we are identifying a selector with its clausal encoding. We do this often in this thesis, in non-ambiguous context. Observe that a 2-selector of order 2 gives the same set of clauses as in Eq. 2.1. In fact, Definition 2.10 is a natural generalization of the encoding of a single comparator.

**Example 2.9.** We would like to encode the network that selects maximum out of four inputs into a set of clauses. We can use the network from Figure 2.1a to do it. If we name the input variables of the longer comparator as $\{z_1, z_2\}$, then the entire network can be encoded by encoding each 2-sorter separately. This produces the clause set $\{x_1 \Rightarrow z_1, x_2 \Rightarrow z_1, x_1 \wedge x_2 \Rightarrow y_2\} \cup \{x_3 \Rightarrow z_2, x_4 \Rightarrow z_2, x_3 \wedge x_4 \Rightarrow y_4\} \cup \{z_1 \Rightarrow y_1, z_2 \Rightarrow y_1, z_1 \wedge z_2 \Rightarrow y_3\}$. This approach uses 6 auxiliary variables (not counting $x_i$'s) and 9 clauses. Another way to encode the same network is to simply use a single 1-selector of order 4. This gives the clause set $\{x_1 \Rightarrow y_1, x_2 \Rightarrow y_1, x_3 \Rightarrow y_1, x_4 \Rightarrow y_1\}$, where we only need 1 additional variable and 4 clauses. Notice that to achieve $y_1 = \max\{x_1, x_2, x_3, x_4\}$ we are only interested in the value of the top output variable, therefore we do not need to assert other output variables.

Informally, *Generalized Selection Networks* (GSNs) are selection networks that use selectors as building blocks. For example, in declarative representation (Definition 2.9) one would substitute the relation $comp(\langle x_1, x_2\rangle, \langle z_1, y_2\rangle)$ for a more general $sel(\langle x_1, \dots, x_n\rangle, \langle y_1, \dots, y_m\rangle)$ which is true if and only if $\langle y_1, \dots, y_m\rangle$ is sorted and contains $m$ largest elements from $\langle x_1, \dots, x_n\rangle$. In procedural representation we generalize $sort^2$ operation to $sel^n_m$ which outputs $m$ sorted, largest elements from $n$ inputs. In this context, $sort^n$ is the same operation as $sel^n_n$. In the end, those modifications are the means to obtain a set of selectors (for a given network), which then we encode as in Definition 2.10.

Encodings of cardinality constraints using (generalized) selection networks where each selector is encoded as described in Definition 2.10 and additional clause $\neg y_{k+1}$ is added are said to be encoded in a *standard* way.

### 2.3.2 Arc-Consistency of The Standard Encoding

We formally define the notion of arc-consistency. A partial assignment $\sigma$ is consistent with a CNF formula $\phi = \{C_1, \dots, C_m\}$, if for each $1 \le i \le m$, $\sigma(C_i)$ is either true or undefined.

**Definition 2.11** (unit propagation)**.** Unit propagation (UP) is a process that extends a partial, consistent assignment $\sigma$ of some CNF formula $\phi$ into a partial assignment $\sigma'$ using the following rule repeatedly until reaching a fix point: if there exists a clause $(l \vee l_1 \vee \cdots \vee l_k)$ in $\phi$ where $l$ is undefined and either $k = 0$ or $l_1, \dots, l_k$ are fixed to 0, then extend $\sigma$ by fixing $\sigma(l) = 1$.

**Definition 2.12** (arc-consistency)**.** Let $n, k \in \mathbb{N}$ and let $\bar{x} = \{x_1, \dots, x_n\}$ be a set of propositional literals. Encoding $\phi$ of the cardinality constraint $x_1 + x_2 + \cdots + x_n \le k$ is *arc-consistent* if the two following conditions hold:

- in any partial assignment consistent with $\phi$, at most $k$ propositional variables from $\bar{x}$ are assigned to 1, and
- in any partial assignment consistent with $\phi$, if exactly $k$ variables from $\bar{x}$ are assigned to 1, then all the other variables occurring in $\bar{x}$ must be assigned to 0 by unit propagation.

We already mentioned in the previous chapter that arc-consistency property guarantees better propagation of values using unit propagation and that this has a positive impact on the practical efficiency of SAT-solvers. Here we show our first result of this thesis, that is, we prove that any encoding based on the standard encoding of GSNs preserves arc-consistency. Our proof is the generalization the proof of arc-consistency for selection networks [46]. For the sake of the proof we give a precise, clausal definition of a GSN, which will help us formally prove the main theorem. We define networks as a sequence of *layers* and a layer as a sequence of selectors. But first, we prove two technical lemmas regarding selectors. We introduce the convention, that $\langle x_1, \dots, x_n\rangle$

will denote the input and $\langle y_1,\ldots,y_m\rangle$ will denote the output of some order $n$ comparator network (or GSN). We would also like to view them as sequences of Boolean variables, that can be set to either true (1), false (0) or undefined.

The following lemma shows how 0-1 values propagates through a single selector.

**Lemma 2.1.** *Let $m \leq n$. A single m-selector of order n, say $s^n_m$, with inputs $\langle x_1,\ldots,x_n\rangle$ and outputs $\langle y_1,\ldots,y_m\rangle$ has the following* propagation properties, *for any partial assignment $\sigma$ consistent with $s^n_m$:*

1. *If k input variables ($1 \leq k \leq n$) are set to 1 in $\sigma$, then UP sets all $y_1,\ldots,y_{min(k,m)}$ to 1.*
2. *If $y_k = 0$ (for $1 \leq k \leq m$) and exactly $k-1$ input variables are set to 1 in $\sigma$, then UP sets all the rest input variables to 0.*

*Proof.* Let $\pi : \{1,\ldots,k\} \to \{1,\ldots,n\}$ be a 1-1 function such that input variables $x_{\pi(1)},\ldots,x_{\pi(k)}$ are set to 1. The following clauses exist: $(x_{\pi(1)} \Rightarrow y_1)$, $(x_{\pi(1)} \wedge x_{\pi(2)} \Rightarrow y_2)$, $\ldots$, $(x_{\pi(1)} \wedge \cdots \wedge x_{\pi(\min(k,m))} \Rightarrow y_{\min(k,m)})$. Therefore UP will set all $y_1,\ldots,y_{min(k,m)}$ to 1.

Now assume that $y_k = 0$ and let $\pi : \{1,\ldots,k-1\} \to \{1,\ldots,n\}$ be a 1-1 function such that input variables $I = \{x_{\pi(1)},\ldots,x_{\pi(k-1)}\}$ are set to 1. For each input variable $u \notin I$ there exist a clause $u \wedge x_{\pi(1)} \wedge \cdots \wedge x_{\pi(k-1)} \Rightarrow y_k$. Therefore $u$ will be set to 0 by UP. $\square$

We say that a formula $\phi$ can be reduced to $\phi'$ by a partial assignment $\sigma$, if we do the following procedure repeatedly, until reaching the fix point: take clause $C$ (after applying assignment $\sigma$) of $\phi$. If it is satisfied, then remove it. If it is of the form $C = (0 \vee \psi)$, then exchange it to $C' = \psi$. Otherwise, do nothing. Notice that $\phi$ is satisfied iff $\phi'$ is satisfied.

**Lemma 2.2.** *Let $n,m \in \mathbb{N}$ and let $s^n_m$ be an m-selector of order n with inputs $\langle x_1,\ldots,x_n\rangle$ and outputs $\langle y_1,\ldots,y_m\rangle$. Let $1 \leq i \leq n$. If we set $x_i = 1$, then $s^n_m$ can be reduced to $s^{n-1}_{m-1}$ by the partial assignment used in unit propagation, where $s^{n-1}_{m-1}$ is an $(m-1)$-selector of order $n-1$.*

*Proof.* By Lemma 2.1, UP sets $y_1 = 1$. Then each clause which contains $x_i$ can be reduced: $(x_i \Rightarrow y_1)$ is satisfied and can be removed; $(x_i \wedge x_q \Rightarrow y_2)$ can be reduced to $(x_q \Rightarrow y_2)$, for each $q \neq i$; and in general, let $Q$ be any subset of $\{1,\ldots,n\} \setminus \{i\}$, then clause $(x_i \wedge (\bigwedge_{q\in Q} x_q) \Rightarrow y_{|Q|+1})$ can be reduced to $(\bigwedge_{q\in Q} x_q \Rightarrow y_{|Q|+1})$. The remaining clauses form an $(m-1)$-selector of order $n-1$. $\square$

**Definition 2.13** (layer)**.** Let $r \in \mathbb{N}$, $n_1,m_1,\ldots,n_r,m_r \in \mathbb{N}$, $S = \langle s^{n_1}_{m_1}(\bar{x}^1,\bar{y}^1),\ldots,s^{n_r}_{m_r}(\bar{x}^r,\bar{y}^r)\rangle$ and let $\bar{x} = \langle x_1,\ldots,x_n\rangle$, $\bar{y} = \langle y_1,\ldots,y_m\rangle$, $\bar{x}^i = \langle x^i_1,\ldots,x^i_{n_i}\rangle$, $\bar{y}^i = \langle y^i_1,\ldots,y^i_{m_i}\rangle$ be sequences of Boolean variables, for $1 \leq i \leq r$. A Boolean formula $L^{(n,m)}(\bar{x},\bar{y},S) = \bigwedge s^{n_i}_{m_i}(\bar{x}^i,\bar{y}^i)$ is a *layer* of order $(n,m)$, for some $n,m \in \mathbb{N}$, if:

1. $n = \sum n_i$, $m = \sum m_i$,
2. for each $1 \leq i \leq r$, $s^{n_i}_{m_i}(\bar{x}^i,\bar{y}^i)$ is an $m_i$-selector of order $n_i$,
3. for each $1 \leq i < j \leq r$, $\bar{x}^i$ and $\bar{x}^j$ are disjoint; $\bar{y}^i$ and $\bar{y}^j$ are disjoint,
4. for each $1 \leq i \leq r$, $\bar{x}^i$ is a subsequence of $\bar{x}$ and $\bar{y}^i$ is a subsequence of $\bar{y}$,

**Definition 2.14** (generalized network)**.** Let $n,m \in \mathbb{N}$, where $m \leq n$, and let $n_1,m_1,\ldots,n_r,m_r \in \mathbb{N}$. Let $\bar{x} = \langle x_1,\ldots,x_n\rangle$, $\bar{y} = \langle y_1,\ldots,y_m\rangle$, $\bar{x}^i = \langle x^i_1,\ldots,x^i_{n_i}\rangle$, $\bar{y}^i = \langle y^i_1,\ldots,y^i_{m_i}\rangle$ be sequences of Boolean variables, and let $S_i$ be a sequence of selectors, for $1 \leq i \leq r$. Let $L = \langle L^{(n_1,m_1)}_1(\bar{x}^1,\bar{y}^1,S_1),\ldots, L^{(n_r,m_r)}_r(\bar{x}^r,\bar{y}^r,S_r)\rangle$. A Boolean formula $f^n_m(\bar{x},\bar{y},L) = \bigwedge L^{(n_i,m_i)}_i$ is a *generalized network* of order $(n,m)$, if:

1. $\bar{x} = \bar{x}^1$, $\bar{y} = \bar{y}^r$, and $\bar{y}^i = \bar{x}^{i+1}$, for $1 \leq i < r$,

2. for each $1 \leq i \leq r$, $L^{(n_i,m_i)}(\bar{x}^i, \bar{y}^i, S_i)$ is a layer of order $(n_i, m_i)$,

**Definition 2.15** (generalized selection network)**.** Let $n,m \in \mathbb{N}$, where $m \leq n$. Let $\bar{x} = \langle x_1, \dots, x_n \rangle$ and $\bar{y} = \langle y_1, \dots, y_m \rangle$ be sequences of Boolean variables and let $L$ be a sequence of layers. A generalized network $f^n_m(\bar{x}, \bar{y}, L)$ is a *generalized selection network* of order $(n,m)$, if for each 0-1 assignment $\sigma$ that satisfies $f^n_m$, $\sigma(\bar{y})$ is sorted and $|\sigma(\bar{y})|_1 \geq \min(|\sigma(\bar{x})|_1, m)$.

To encode a cardinality constraint $x_1 + \dots + x_n \leq k$ we build a GSN $f^n_{k+1}(\bar{x}, \bar{y}, L)$ and we add to it a singleton clause $\neg y_{k+1}$ which practically sets the variable $y_{k+1}$ to false. Such encoding we will call a *standard encoding*.

**Definition 2.16** (standard encoding)**.** Let $k,n \in \mathbb{N}$, where $k \leq n$ and let $\bar{x} = \langle x_1, \dots, x_n \rangle$ and $\bar{y} = \langle y_1, \dots, y_{k+1} \rangle$ be sequences of Boolean variables. A Boolean formula $\phi^n_k(\bar{x}, \bar{y}, L) = f^n_{k+1}(\bar{x}, \bar{y}, L) \wedge \neg y_{k+1}$ is a *standard encoding* of the constraint $x_1 + \dots + x_n \leq k$, if $f^n_{k+1}(\bar{x}, \bar{y}, L)$ is a generalized selection network of order $(n, k+1)$ (for some sequence $L$ of layers).

We use the following convention regarding equivalences of Boolean variables: the symbol "$=$" is used as an assignment operator or value equivalence. We use the symbol "$\equiv$" as equivalence of variables, that is, if $a \equiv b$ then $a$ is an alias for $b$ (and vice-versa). We define $V[\phi]$ as the set of Boolean variables in formula $\phi$.

Let $n,k,r \in \mathbb{N}$, $n_1, k_1, \dots, n_r, k_r \in \mathbb{N}$. Let $\bar{x} = \langle x_1, \dots, x_n \rangle$, $\bar{y} = \langle y_1, \dots, y_{k+1} \rangle$, $\bar{x}^i = \langle x^i_1, \dots, x^i_{n_i} \rangle$, $\bar{y}^i = \langle y^i_1, \dots, y^i_{k_i} \rangle$ be sequences of Boolean variables, for $1 \leq i \leq r$. For the rest of the chapter assume that $f^n_{k+1}(\bar{x}, \bar{y}, L)$ is a GSN of order $(n, k+1)$, where $L = \langle L^{(n_1,k_1)}_1(\bar{x}^1, \bar{y}^1, S_1), \dots, L^{(n_r,k_r)}_r(\bar{x}^r, \bar{y}^r, S_r) \rangle$ is a sequence of layers, $S_i$ is a sequence of selectors (for $1 \leq i \leq r$).

We will now define a notion that captures a structure of propagation through the network for a single variable.

**Definition 2.17** (path)**.** A *path* is a sequence of Boolean variables $\langle z_1, \dots, z_p \rangle$ such that $\forall_{1 \leq i \leq p}\, z_i \in V[f^n_{k+1}]$ and for all $1 \leq i < p$ there exists an $m$-selector of order $n'$ in $f^n_{k+1}$ with inputs $\langle a_1, \dots, a_{n'} \rangle$ and outputs $\langle b_1, \dots, b_m \rangle$ such that $z_i \in \langle a_1, \dots, a_{n'} \rangle$ and $z_{i+1} \in \langle b_1, \dots, b_m \rangle$.

**Definition 2.18** (propagation path)**.** Let $x$ be an undefined variable. A path $\bar{z}_x = \langle z_1, \dots, z_p \rangle$ is a *propagation path*, if $z_1 \equiv x$ and $p$ is the largest integer such that $\langle z_2, \dots, z_p \rangle$ is a sequence of variables that would be set to 1 by UP, if we set $z_1 = 1$.

**Lemma 2.3.** *Let $1 \leq i \leq n$ and let $z_{x_i} = \langle z_1, \dots, z_p \rangle$ be the propagation path, for some $p \in \mathbb{N}$. Then (i) each $z_j$ is an input to a layer $L^{(n_j,k_j)}$, for $1 \leq j < r$, and (ii) $z_p \equiv z_r \equiv y_1$.*

*Proof.* We prove (i) by induction on $j$. If $j = 1$ then $z_1 \equiv x_i$ is the input to the first layer, by the definition of generalized network. Take any $j \geq 1$ and assume that $z_j$ is an input to a layer $L^{(n_j,k_j)}$. From the definition of a layer, inputs of selectors of $L^{(n_j,k_j)}$ are disjoint, therefore $z_j$ is an input to a unique selector $s^{n'}_{m'} \in S_j$, for some $n' \geq m'$. By Lemma 2.1, if $z_j = 1$ then $z_{j+1}$ – the output of $s^{n'}_{m'}$ – is set to 1. Since $j < r$, $z_{j+1}$ is an input to layer $L^{(n_{j+1},k_{j+1})}$, by the definition of generalized network. This ends the inductive step, therefore (i) is true.

Using (i) we know that $z_{r-1}$ is an input to layer $L^{(n_{r-1},k_{r-1})}$. Using similar argument as in the inductive step in previous paragraph, we conclude that $z_r \equiv y_t$, for some $1 \leq t \leq k+1$. Let $\sigma$ be a partial assignment which fixed variables $\langle z_1, \dots, z_p \rangle$ by unit propagation. We can extend $\sigma$ so that every other variable in $f^n_{k+1}$ is set to 0, then by Definition 2.15 the output $\sigma(\bar{y})$ is sorted and $|\sigma(\bar{y})|_1 \geq 1$, therefore we conclude that $t = 1$, so (ii) is true. $\square$

**Lemma 2.4.** *Let $1 \leq i \leq n$. If we set $x_i = 1$ in a partial assignment $\sigma$ (where the rest of the variables are undefined), then unit propagation will set $y_1 = 1$. Furthermore, $f^n_{k+1}$ can be reduced to $f^{n-1}_k$ by $\sigma$, where $f^{n-1}_k$ is a generalized selection network of order $(n-1,k)$.*

*Proof.* First part is a simple consequence of Lemma 2.3. Take partial assignment $\sigma$ (after unit propagation set $y_1 = 1$). The assigned variables are exactly the ones from the propagation path $z_{x_i} = \langle z_1,\dots,z_r\rangle$. Therefore, by Lemma 2.3, we need to consider clauses from a single selector of each layer. Take each $1 \leq j \leq r$ and $s^{n'}_{m'} \in S_j$ such that $z_j$ is an input of $s^{n'}_{m'}$. By Lemma 2.2, $s^{n'}_{m'}$ can be reduced to a smaller selector $s^{n'-1}_{m'-1}$. Thus $f^n_{k+1}$ can be reduced to $f^{n-1}_k$ with inputs $\{x_1,\dots,x_n\} \setminus \{x_i\}$ and outputs $\{y_2,\dots,y_{k+1}\}$, which is a generalized selection network of order $(n-1,k)$. □

**Lemma 2.5.** *Let $1 \leq i \leq k+1$ and let $\pi : \{1,\dots,i\} \to \{1,\dots,n\}$ be a 1-1 function. Assume that inputs $x_{\pi(1)},\dots,x_{\pi(i)}$ are set to 1, and the rest of the variables are undefined. Unit propagation will set variables $y_1,\dots,y_i$ to 1.*

*Proof.* By setting 1 to variables $x_{\pi(1)},\dots,x_{\pi(i)}$ in an (arbitrarily) ordered way and repeated application of Lemma 2.4. □

The process of propagating 1's given in Lemma 2.5 we call a *forward propagation*. We are ready to prove the main result.

**Theorem 2.1.** *Consider the standard encoding $\phi^n_k(\bar{x},\bar{y},L) = f^n_{k+1}(\bar{x},\bar{y},L) \wedge \neg y_{k+1}$. Assume that k inputs are set to 1 and forward propagation has been performed which set all $y_1,\dots,y_k$ to 1. Then the unit propagation will set all undefined input variables to 0.*

*Proof.* First, reduce $f^n_{k+1}$ to $f^{n'}_1$ ($n' = n-k$), by repeated application of Lemma 2.4. Let new input variables be $\langle x'_1,\dots,x'_{n'}\rangle$ and let the single output be $y'_1 \equiv y_{k+1}$. Since the standard encoding contains a singleton clause $\neg y_{k+1}$, then $y'_1 = y_{k+1} = 0$. Let $\langle L'^{(n'_1,k'_1)}_1,\dots,L^{(n'_r,k'_r)}_r\rangle$ be the sequence of layers in $f^{n'}_1$, where $n'_1,k'_1,\dots,n'_r,k'_r \in \mathbb{N}$. Let $0 \leq i < r$, we prove the following statement by induction on $i$:

$$S(i) = \text{unit propagation sets all input variables of layer } L^{(n'_{r-i},k'_{r-i})}_{r-i} \text{ to } 0.$$

When $i = 0$, then we consider the last layer $L^{(n'_r,k'_r)}_r$. Since it has only one output, namely $y'_1$, it consists of a single selector $s^{n'_r}_1$. We know that $y'_1 = 0$, then by Lemma 2.1 all input variables are set to 0 by UP, therefore $S(0)$ holds. Take any $i \geq 0$ and assume that $S(i)$ is true. That means that all input variables of $L^{(n'_{r-i},k'_{r-i})}_{r-i}$ are set to 0. From the definition of generalized network, those inputs are the outputs of the layer $L^{(n'_{r-i-1},k'_{r-i-1})}_{r-i-1}$. Therefore each selector of layer $L^{(n'_{r-i-1},k'_{r-i-1})}_{r-i-1}$ has all outputs set to 0. By Lemma 2.1 all input variables of this selectors are set to 0 by UP, and therefore all inputs of $L^{(n'_{r-i-1},k'_{r-i-1})}_{r-i-1}$ are set to 0. Thus we conclude that $S(i+1)$ holds. This completes the induction step.

We know that $S(r-1)$ is true, which means that all input variables of layer $L^{(n'_1,k'_1)}_1$ are set to 0, those are exactly the input variables $\langle x'_1,\dots,x'_{n'}\rangle$ of $f^{n'}_1$. Those variables are previously undefined input variables of $f^n_{k+1}$, which completes the proof. □

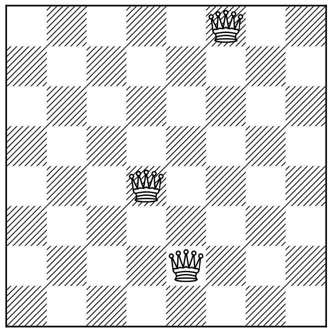
## 2.4 Summary

In this chapter we have given definitions and conventions used in the rest of the thesis. We have presented several comparator network models and we conclude that for the purpose of this thesis the procedural representation is sufficient to present all our encodings. We deconstructed a comparator into a set of clauses and showed how 0-1 values are being propagated therein.

We have also defined a model based on layers of selectors and we showed the first rigorous proof that any standard encoding based on generalized selection networks preserves arc-consistency. We believe that this result will relieve future researchers of this topic from the burden of proving that their encodings are arc-consistent, which was usually a long and technical endeavor.

# Chapter 3

# Encodings of Boolean Cardinality Constraints



The practical importance of encoding cardinality constraints into SAT resulted in a large number of research papers published in the last 20 years. In this chapter we review some of the most significant methods found in the literature for three types of constraints: *at-most-k*, *at-most-one* and *Pseudo-Boolean*. Although Pseudo-Boolean constraints are more general than cardinality constraints, many methods for efficiently encoding cardinality constraints were first developed for Pseudo-Boolean constraints.

## 3.1 At-Most-k Constraints

Boolean cardinality constraints – which are the main focus of this thesis – are also called at-most-k constraints in the literature [35], which is understandable, since we require that **at most** $k$ out of $n$ propositional literals can be true, and constraints with other relations (at least, or exactly) can be easily reduced to the "at most" case (Observation 2.1). Here we present the most influential ideas in the topic of translating such constraints into propositional formulas. The overview is presented in the chronological order.

**Binary adders.** Warners [77] considered encoding based on using adders where numbers are represented in binary. The original method was devised for PB-constraints and involved using binary addition of terms of the LHS (left-hand side) of the constraint $a_1x_1 + a_2x_2 + \cdots + a_nx_n \leq k$ using *adders* and *multiplicators*, then comparing the resulting sum to the binary representation of $k$. On the highest level, the algorithm uses a divide-and-conquer strategy, partitioning the terms of the LHS into two sub-sums, recursively encoding them into binary numbers, then encoding the summation of those two numbers using an adder. An adder is simply a formula that models the classical addition in columns:

$$\begin{array}{ccccccc} & p^A_{M-1} & p^A_{M-2} & \cdots & p^A_2 & p^A_1 & p^A_0 \\ + & p^B_{M-1} & p^B_{M-2} & \cdots & p^B_2 & p^B_1 & p^B_0 \\ \hline p^C_M & p^C_{M-1} & p^C_{M-2} & \cdots & p^C_2 & p^C_1 & p^C_0 \end{array}$$

In the above we compute $A + B = C$, where $p^X_i$ is a propositional variable representing $i$-th bit of number $X$, for $1 \leq i \leq M$. We assume that $M$ is a constant bounding the number of bits in $C$.

The addition is done column-by-column, from right to left, taking into consideration the potential carries (propositional variables $c_{i,j}$'s):

$$\left(p_0^C \Leftrightarrow (p_0^A \Leftrightarrow \neg p_0^B)\right) \wedge \left(c_{0,1} \Leftrightarrow (p_0^A \wedge p_0^B)\right) \wedge \bigwedge_{j=1}^{M} \left(p_j^C \Leftrightarrow (p_j^A \Leftrightarrow p_j^B \Leftrightarrow c_{j-1,j})\right) \wedge$$

$$\bigwedge_{j=1}^{M-1} \left(c_{j,j+1} \Leftrightarrow (p_j^A \wedge p_j^B) \vee (p_j^A \wedge c_{j-1,j}) \vee (p_j^B \wedge c_{j-1,j})\right) \wedge \left(c_{M-1,M} \Leftrightarrow p_M^C\right).$$

Notice that $(p_j^A \Leftrightarrow p_j^B \Leftrightarrow c_{j-1,j})$ is true if and only if 1 or 3 variables from the set $\{p_j^A, p_j^B, c_{j-1,j}\}$ are true.

For the base case, we have to compute the multiplication of $a_i$ and $x_i$. This is also done in a straightforward way. In the following, let $B_{a_i}$ be the set containing indices of all 1's in the binary representation of $a_i$. We get:

$$\bigwedge_{k \in B_{a_i}} \left(p_k^{a_i} \Leftrightarrow x_i\right) \wedge \bigwedge_{k \notin B_{a_i}} \neg p_k^{a_i}$$

Finally, we have to enforce the constraint $(\leq k)$, which is done like so:

$$\bigwedge_{i \notin B_k} \left( p_i^{LHS} \Rightarrow \neg \bigwedge_{j \in B_k:\, j>i} p_j^{LHS} \right),$$

where $p_0^{LHS}, p_1^{LHS}, \ldots$ are propositional variables representing the binary number of the sum of the LHS.

**Example 3.1.** Let $k = 26$, as in the example from [77]. Then $B_k = \{1,3,4\}$. In order to enforce $LHS \leq k$, we add the following clauses:

$$\left(p_0^{LHS} \Rightarrow \neg(p_1^{LHS} \wedge p_3^{LHS} \wedge p_4^{LHS})\right) \wedge \left(p_2^{LHS} \Rightarrow \neg(p_3^{LHS} \wedge p_4^{LHS})\right).$$

We can check that this indeed enforces the constraint for some sample values of the LHS. For example, if $LHS = 25$, then $\langle p_0^{LHS}, p_1^{LHS}, p_2^{LHS}, p_3^{LHS}, p_4^{LHS} \rangle = \langle 1,0,0,1,1 \rangle$, and we can see that the formula above is satisfied. However, for $LHS = 30$, we get $\langle p_0^{LHS}, p_1^{LHS}, p_2^{LHS}, p_3^{LHS}, p_4^{LHS} \rangle = \langle 0,1,1,1,1 \rangle$, and the second implication evaluates to false.

In Lemma 2 of [77] it is shown that number of variables and clauses of the proposed encoding is bounded by $2n(1+\log(a_{max}))$ and $8n(1+2\log(a_{max}))$, respectively ($a_{max} = \max\{a_i\}$). Therefore, in case of cardinality constraints, the encoding uses $O(n)$ variables and clauses. This encoding is small, but does not preserve arc-consistency.

**Totalizers.** Bailleux and Boufkhad [12] presented an encoding based on the idea of a totalizer. The totalizer is a binary tree, where the leaves are the constraint literals $x_i$'s. With each inner node the number $s$ is associated which represents the sum of the leaves in the corresponding sub-tree. The number $s$ is represented in unary, by $s$ auxiliary variables. The encoding is arc-consistent and uses $O(n \log n)$ variables and $O(n^2)$ clauses. Here we briefly explain the idea of a totalizer as described in [12].

We begin with defining unary representation of a number $v$ such that $0 \leq v \leq n$. An integer $v$ can be modeled by a set $V = \{v_1, v_2, \ldots, v_n\}$ of $n$ propositional variables. Each possible value of

$v$ is encoded as a complete instantiation of $V$, such that if $v = x$, then $x$ 1's follow $(n-x)$ 0's, i.e., $v_1 = 1, v_2 = 1, \ldots, v_x = 1, v_{x+1} = 0, \ldots, v_n = 0$. A partial instantiation of $V$ is said to be *pre-unary* if for each $v_i = 1$, $v_j = 1$ for any $j < i$ and for each $v_i = 0$, $v_j = 0$ for any $j, i \leq j \leq n$. In other words, $V$ has its prefix set with 1's and its suffix with 0's (prefix and suffix cannot overlap in this context).

The advantage of using the unary representation is that the integer can be specified as belonging to an interval. The inequality $x \leq v \leq y$ is specified by the partial pre-unary instantiation of $V$ that fixes to 1 any $v_i$ such that $i \leq x$ and fixes to 0 any $v_j$ such that $j \geq y+1$.

**Example 3.2.** Following an example from [12], consider $n = 6$ and a partial instantiation such that $v_1 = v_2 = 1$, $v_5 = v_6 = 0$ and $v_3, v_4$ are free. Then the corresponding integer $v$ is such that $2 \leq v \leq 4$.

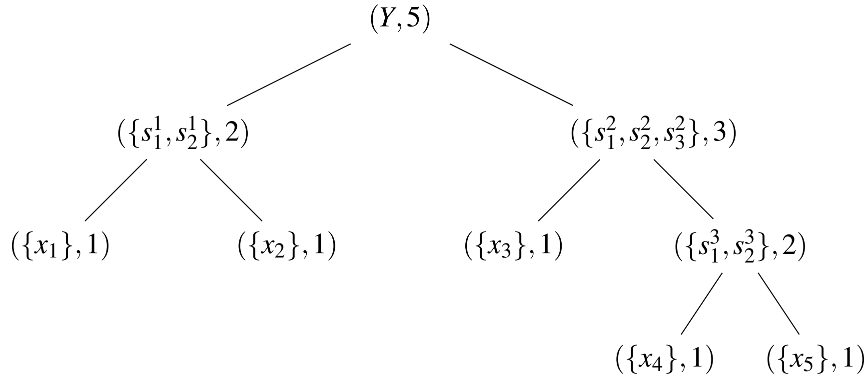
The totalizer is a CNF formula defined on 3 sets of variables: $X = x_1, \ldots, x_n$ – inputs, $Y = y_1, \ldots, y_n$ – outputs, and a set S of *linking variables*. These sets can be described by a binary tree built as follows. We start from an isolated root node labeled $n$ and we proceed iteratively: to each node labeled by $m > 1$, we connect two children labeled by $\lfloor m/2 \rfloor$ and $\lceil m/2 \rceil$, respectively. This produces a binary tree with $n$ leaves labeled 1. Next, each variable in $X$ is allocated to a leaf in a bijective way. Set $Y$ is allocated to the root node. To each internal node labeled by an integer $m$, a set of $m$ new variables is allocated that is used to represent an unary value between 1 and $m$. All those internal variables produce the set $S$.

We now define an encoding that ensures that $m = m_1 + m_2$ in any complete instantiation of the variables belonging to some sub-tree of a totalizer labeled $m$ with two children labeled $m_1$ and $m_2$. Let $S = \{s_1, \ldots, s_m\}$, $S_1 = \{s_1^1, \ldots, s_{m_1}^1\}$ and $S_2 = \{s_1^2, \ldots, s_{m_2}^2\}$ be the sets of variables related to $m$, $m_1$ and $m_2$, respectively. We add the following set of clauses:

$$\bigwedge_{\substack{0 \leq a \leq m_1 \\ 0 \leq b \leq m_2 \\ 0 \leq c \leq m \\ a+b=c}} \left( (s_a^1 \wedge s_b^2 \Rightarrow s_c) \wedge (s_{c+1} \Rightarrow s_{a+1}^1 \vee s_{b+1}^2) \right),$$

where $s_0^1 = s_0^2 = s_0 = 1, s_{m_1+1}^1 = s_{m_2+1}^2 = s_{m+1} = 0$.

**Example 3.3.** Borrowing an example from [12], for $n = 5$, $X = \{x_1, x_2, x_3, x_4, x_5\}$ and $Y = \{y_1, y_2, y_3, y_4, y_5\}$ the following tree is obtained by the totalizer procedure:



Here, the set of linking variables is $S = \{s_1^1, s_2^1, s_1^2, s_2^2, s_3^2, s_1^3, s_2^3\}$. Let us encode the unary addition on a node labeled 3 in the totalizer above (clauses with constants 0's and 1's are already simplified):

$$(s_1^3 \Rightarrow s_1^2) \wedge (s_2^3 \Rightarrow s_2^2) \wedge (x_3 \Rightarrow s_1^2) \wedge (x_3 \wedge s_1^3 \Rightarrow s_2^2) \wedge (x_3 \wedge s_2^3 \Rightarrow s_3^2) \wedge$$
$$(s_1^2 \Rightarrow x_3 \vee s_1^3) \wedge (s_2^2 \Rightarrow x_3 \vee s_2^3) \wedge (s_3^2 \Rightarrow x_3) \wedge (s_2^2 \Rightarrow s_1^3) \wedge (s_3^2 \Rightarrow s_2^3).$$

Notice that the first row of clauses represents the relation $c \geq a+b$ and the second row of clauses represents the relation $c \leq a+b$, where $c$, $a$ and $b$ are possible values of the unary representations of nodes labeled 3 and its children, respectively.

Büttner and Rintanen [23] made an improvement to the encoding of Bailleux and Boufkhad [12] by noticing that counting up to $k+1$ suffices to enforce the constraint. Therefore, they reduced the number of variables and clauses used in each node of the totalizer. Their encoding improves the previous result for small values of $k$ as it requires $O(nk)$ variables and $O(nk^2)$ clauses. They also proposed a novel encoding based on encoding the injective mapping between the true $x_i$ variables and $k$ elements. This idea requires $O(nk)$ variables and clauses, but is not arc-consistent.

**Counters.** Sinz [74] proposed two encodings based on counters. The first uses a sequential counter where numbers are represented in unary and the second uses a parallel counter with numbers represented in binary. The first encoding uses $O(nk)$ variables and clauses and the second encoding uses $O(n)$ variables and clauses. Only the encoding based on a sequential counter is arc-consistent. We present here the construction of a sequential counter.

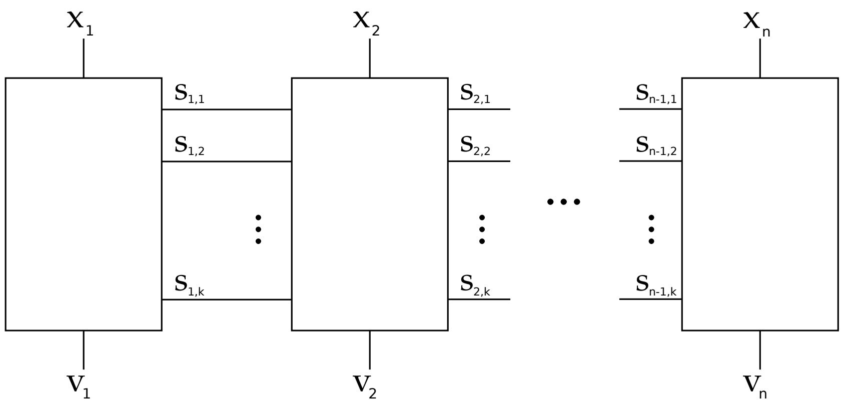



Figure 3.1: Schema of a sequential counter

The general idea is to build a count-and-compare hardware circuit and then translate this circuit to a CNF. Such counter is presented in Figure 3.1 and consists of $n$ sub-circuits, each computing a partial sum $s_i = \sum_{j=1}^{i} x_j$. The values of all $s_i$'s are represented as unary numbers, i.e., we have a sequence of variables $\langle s_{i,1}, \ldots, s_{i,k} \rangle$ for each $s_i$. Additional variable $v_i$ (the overflow bit) is related to each sub-circuit and is set to true if the partial sum $s_i$ is greater than $k$. To convert the circuit to a CNF, we first build a series of implications for the partial sum bits $s_{i,j}$'s ($j$-th bit of $s_i$) and the overflow bits $v_i$. We can then simplify the formula noting that all overflow bits have to be zero, in order to enforce the constraint $\leq k$. The resulting set of clauses is as follows:

$$(x_1 \Rightarrow s_{1,1}) \wedge (x_n \Rightarrow \neg s_{n-1,k}) \wedge \bigwedge_{1<j\leq k} (\neg s_{1,j}) \wedge$$
$$\bigwedge_{1<i<n} \Bigg( (x_i \Rightarrow s_{i,1}) \wedge (s_{i-1,1} \Rightarrow s_{i,1}) \wedge (x_i \Rightarrow \neg s_{i-1,k}) \wedge$$
$$\bigwedge_{1<j\leq k} \Big( (x_i \wedge s_{i-1,j-1} \Rightarrow s_{i,j}) \wedge (s_{i-1,j} \Rightarrow s_{i,j}) \Big) \Bigg).$$

**Sorting Networks.** One of the most influential ideas was introduced by Eén and Sörensson [34]. They proposed an encoding of Pseudo-Boolean constraints based on the odd-even sorting networks. A PB-constraint is decomposed into a number of interconnected sorting networks, where each sorter represents an adder of digits in a mixed radix base. Detailed explanation of this technique is done in Section 3.3. In case of cardinality constraints a single sorting network is required and we have already explained the details of this technique in the previous chapter. This encoding (using sorting networks) requires $O(n \log^2 n)$ variables and clauses and is arc-consistent.

It is worth noting that although we reduce cardinality constraints to the form $x_1 + x_2 + \cdots + x_n \leq k$, and we use 3-clause representation for each comparator ($\{a \Rightarrow c, b \Rightarrow c, a \wedge b \Rightarrow d\}$) and assert the output $\neg y_{k+1}$, we do this only to simplify the presentation. In practice, when dealing with other types of cardinality constraints, one should do the following:

- For $x_1 + x_2 + \cdots + x_n \geq k$ encode each comparator with the set of clauses $\{d \Rightarrow a, d \Rightarrow b, c \Rightarrow a \vee b\}$ and add a clause $y_k$.
- For $x_1 + x_2 + \cdots + x_n = k$ encode each comparator with 6-clause representation $\{a \Rightarrow c, b \Rightarrow c, a \wedge b \Rightarrow d, d \Rightarrow a, d \Rightarrow b, c \Rightarrow a \vee b\}$ and assert both $y_k$ and $\neg y_{k+1}$.

The situation where reduction from one inequality to the other is beneficial is when $k > \lfloor n/2 \rfloor$, for example, given $x_1 + x_2 + \cdots + x_n \geq k$ we reduce it to $\neg x_1 + \neg x_2 + \cdots + \neg x_n \leq n - k$ (following Observation 2.1). In the resulting cardinality constraint $n - k < \lceil n/2 \rceil$. This does not make the encoding smaller in case of sorting networks, but in case of selection networks this might vastly reduce the size of the resulting CNF.

**Selection Networks.** Further improvements in encoding cardinality constraints are based on the aforementioned idea of Eén and Sörensson [34]. Basically, in order to make more efficient encodings, more efficient sorting networks are required. It was observed that we do not need to sort the entire input sequence, but only the first $k+1$ largest elements. Hence, the use of *selection networks* allowed to achieve the complexity $O(n \log^2 k)$ in terms of the number of variables and clauses. In the last years several selection networks were proposed for encoding cardinality constraints and experiments proved their efficiency. They were based mainly on the odd-even or pairwise comparator networks. Codish and Zazon-Ivry [27] introduced Pairwise Selection Networks that used the concept of Parberry's Pairwise Sorting Network [65]. Their construction was later improved (we show this result in Chapter 4). Abío, Asín, Nieuwenhuis, Oliveras and Rodríguez-Carbonell [3, 10] defined encodings that implemented selection networks based on the odd-even sorting networks. In [3] the authors proposed a mixed parametric approach to the encodings, where so called *Direct Cardinality Networks* are chosen for small sub-problems and the splitting point is optimized when large problems are divided into two smaller ones. They proposed to minimize the function $\lambda \cdot num\_vars + num\_clauses$ in the encodings. The constructed encodings are small and efficient.

It's also worth noting that using encodings based on selection networks give an extra edge in solving optimization problems for which we need to solve a sequence of problems that differ only in the decreasing bound of a cardinality constraint. In this setting we only need to add one more clause $\neg y_k$ for a new value of $k$, and the search can be resumed keeping all previous clauses as it is. This works because if a comparator network is a $k$-selection network, then it is also a $k'$-selection network for any $k' < k$. This property is called *incremental strengthening* and most state-of-the-art SAT-solvers provide a user interface for it.

## 3.2 At-Most-One Constraints

Much research has been done on cardinality constraints for small values of $k$. The special case of at-most-k constraint is when $k = 1$, which results in at-most-one constraint (AMO, in short), which could be viewed as the simplest type of constraint and at the same time, the most useful one. It is due to the fact that AMO constraints are most widely used constraints during the process of translating a practical problem into a propositional satisfiability instance. We reference some of those encodings here.

For convenience, we denote $AMO(X)$ and $ALO(X)$ to be at-most-one and at-least-one clauses for the set of propositional variables $X = \{x_1, \ldots, x_n\}$, respectively, and we define $EO(X) = AMO(X) \wedge ALO(X)$. We use a running example $AMO(x_1, \ldots, x_8)$ to illustrate the encodings.

**Binomial Encoding.** The simplest encoding is the *binomial* encoding, sometimes also called the *naive* encoding. It is referenced in many papers, for example in [35]. The idea of this encoding is to express that all possible combinations of two variables are not simultaneously assigned to true. This requires $\binom{n}{2}$ clauses:

$$\bigwedge_{i=1}^{n-1} \bigwedge_{j=i+1}^{n} (\neg x_i \vee \neg x_j).$$

The encoding does not require any additional variables, but the quadratic number of clauses makes it impractical for large values of $n$. Nevertheless, due to its simplicity, it is widely used in practice. Notice that we used this encoding to translate the 4-Queens Puzzle to SAT in Section 1.1.1.

**Example 3.4.** In the running example, the binomial encoding produces the following set of clauses:

$$\begin{array}{l}
(\neg x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_4) \wedge (\neg x_1 \vee \neg x_5) \wedge (\neg x_1 \vee \neg x_6) \wedge (\neg x_1 \vee \neg x_7) \wedge (\neg x_1 \vee \neg x_8) \\
(\neg x_2 \vee \neg x_3) \wedge (\neg x_2 \vee \neg x_4) \wedge (\neg x_2 \vee \neg x_5) \wedge (\neg x_2 \vee \neg x_6) \wedge (\neg x_2 \vee \neg x_7) \wedge (\neg x_2 \vee \neg x_8) \\
(\neg x_3 \vee \neg x_4) \wedge (\neg x_3 \vee \neg x_5) \wedge (\neg x_3 \vee \neg x_6) \wedge (\neg x_3 \vee \neg x_7) \wedge (\neg x_3 \vee \neg x_8) \\
(\neg x_4 \vee \neg x_5) \wedge (\neg x_4 \vee \neg x_6) \wedge (\neg x_4 \vee \neg x_7) \wedge (\neg x_4 \vee \neg x_8) \\
(\neg x_5 \vee \neg x_6) \wedge (\neg x_5 \vee \neg x_7) \wedge (\neg x_5 \vee \neg x_8) \\
(\neg x_6 \vee \neg x_7) \wedge (\neg x_6 \vee \neg x_8) \\
(\neg x_7 \vee \neg x_8)
\end{array}$$

**Binary Encoding.** The *binary* encoding [36] uses $\lceil \log n \rceil$ auxiliary variables $\{b_1, \ldots, b_{\lceil \log n \rceil}\}$ to reduce the number of clauses to $n \log n$. The idea is to create a mapping between each label of the variables $\{x_1, \ldots, x_n\}$ to its binary representation using the auxiliary variables ($b_j$ represent j-th bit of the number, for $1 \le j \le \lceil \log n \rceil$) so that the truth assignment of one input variable $x_i$ implies that the rest of the variables evaluate to false:

$$\bigwedge_{i=1}^{n} \bigwedge_{j=1}^{\lceil \log n \rceil} (x_i \Rightarrow B(i,j)),$$

where $B(i,j) \equiv b_j$, if $j$-th bit of $i-1$ (represented in binary) is 1, otherwise $B(i,j) \equiv \neg b_j$.

**Example 3.5.** In the running example, the binary encoding produces the following set of clauses:

$$\begin{aligned}
&(x_1 \Rightarrow \neg b_1) \wedge (x_1 \Rightarrow \neg b_2) \wedge (x_1 \Rightarrow \neg b_3) \wedge \\
&(x_2 \Rightarrow b_1) \wedge (x_2 \Rightarrow \neg b_2) \wedge (x_2 \Rightarrow \neg b_3) \wedge \\
&(x_3 \Rightarrow \neg b_1) \wedge (x_3 \Rightarrow b_2) \wedge (x_3 \Rightarrow \neg b_3) \wedge \\
&(x_4 \Rightarrow b_1) \wedge (x_4 \Rightarrow b_2) \wedge (x_4 \Rightarrow \neg b_3) \wedge \\
&(x_5 \Rightarrow \neg b_1) \wedge (x_5 \Rightarrow \neg b_2) \wedge (x_5 \Rightarrow b_3) \wedge \\
&(x_6 \Rightarrow b_1) \wedge (x_6 \Rightarrow \neg b_2) \wedge (x_6 \Rightarrow b_3) \wedge \\
&(x_7 \Rightarrow \neg b_1) \wedge (x_7 \Rightarrow b_2) \wedge (x_7 \Rightarrow b_3) \wedge \\
&(x_8 \Rightarrow b_1) \wedge (x_8 \Rightarrow b_2) \wedge (x_8 \Rightarrow b_3)
\end{aligned}$$

**Commander Encoding.** In the *commander* encoding [49] one splits the input variables into $m$ disjoint sets $\{G_1, \dots, G_m\}$ and introduce $m$ auxiliary commander variables $\{c_1, \dots, c_m\}$, one for each set. The constraint is enforced by adding clauses so that exactly one variable from $G_i \cup \neg c_i$ is true and at most one of the commander variables are true:

$$\bigwedge_{i=1}^{m} EO(\{\neg c_i\} \cup G_i) \wedge AMO(c_1, \dots, c_m)$$

where $EO(\{\neg c_i\} \cup G_i) = AMO(\{\neg c_i\} \cup G_i) \wedge ALO(\{\neg c_i\} \cup G_i)$ by definition. ALO part can be easily translated into a single clause and AMO parts can be encoded either recursively or by another AMO encoding (like the binomial encoding).

**Example 3.6.** In the running example, if we set $m = 4$ and we divide the input set $X = \{x_1, \dots, x_8\}$ into subsets $G_1 = \{x_1, x_2\}$, $G_2 = \{x_3, x_4\}$, $G_3 = \{x_5, x_6\}$ and $G_4 = \{x_7, x_8\}$, and we use the binomial encoding for the AMO parts, then the commander encoding produces the following set of clauses:

$$\begin{aligned}
&(c_1 \vee \neg x_1) \wedge (c_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_2) \wedge (\neg c_1 \vee x_1 \vee x_2) \wedge \\
&(c_2 \vee \neg x_3) \wedge (c_2 \vee \neg x_4) \wedge (\neg x_3 \vee \neg x_4) \wedge (\neg c_2 \vee x_3 \vee x_4) \wedge \\
&(c_3 \vee \neg x_5) \wedge (c_3 \vee \neg x_6) \wedge (\neg x_5 \vee \neg x_6) \wedge (\neg c_3 \vee x_5 \vee x_6) \wedge \\
&(c_4 \vee \neg x_7) \wedge (c_4 \vee \neg x_8) \wedge (\neg x_7 \vee \neg x_8) \wedge (\neg c_4 \vee x_7 \vee x_8) \\
&\wedge \\
&(\neg c_1 \vee \neg c_2) \wedge (\neg c_1 \vee \neg c_3) \wedge (\neg c_1 \vee \neg c_4) \wedge (\neg c_2 \vee \neg c_3) \wedge (\neg c_2 \vee \neg c_4) \wedge (\neg c_3 \vee \neg c_4).
\end{aligned}$$

**Product Encoding.** Chen [26] proposed the *product* encoding, where the idea is to arrange the input variables into a 2-dimensional array and to enforce that in at most one column and at most one row the variable can be set to true. Let $p,q \in \mathbb{N}$ such that $p \times q \geq n$. We introduce the row variables $R = \{r_1,\dots,r_p\}$ and column variables $C = \{c_1,\dots,c_q\}$. We map the inputs into a 2-dimensional array and enforce the constraint in the following way:

$$AMO(R) \wedge AMO(C) \wedge \bigwedge_{1\leq i\leq p,\, 1\leq j\leq q}^{1\leq k\leq n,\, k=(i-1)q+j} (x_k \Rightarrow r_i \wedge c_j),$$

where $AMO(R)$ and $AMO(C)$ can be computed recursively or by another encoding.

**Example 3.7.** In the running example, if we set $p = q = 3$, then the arrangement of the variables can be illustrated as follows:

| | $c_1$ | $c_2$ | $c_3$ |
|---|---|---|---|
| $r_1$ | $x_1$ | $x_2$ | $x_3$ |
| $r_2$ | $x_4$ | $x_5$ | $x_6$ |
| $r_3$ | $x_7$ | $x_8$ | |

If we use the binomial encoding for $AMO(R)$ and $AMO(C)$, then the product encoding produces the following set of clauses:

$$\begin{aligned}
&(\neg r_1 \vee \neg r_2) \wedge (\neg r_1 \vee \neg r_3) \wedge (\neg r_2 \vee \neg r_3) \wedge \\
&(\neg c_1 \vee \neg c_2) \wedge (\neg c_1 \vee \neg c_3) \wedge (\neg c_2 \vee \neg c_3) \\
&\wedge \\
&(x_1 \Rightarrow r_1 \wedge c_1) \wedge (x_2 \Rightarrow r_1 \wedge c_2) \wedge (x_3 \Rightarrow r_1 \wedge c_3) \wedge \\
&(x_4 \Rightarrow r_2 \wedge c_1) \wedge (x_5 \Rightarrow r_2 \wedge c_2) \wedge (x_6 \Rightarrow r_2 \wedge c_3) \wedge \\
&(x_7 \Rightarrow r_3 \wedge c_1) \wedge (x_8 \Rightarrow r_3 \wedge c_2).
\end{aligned}$$

**Bimander Encoding.** Recently, hybrid approaches have emerged, for example the *bimander encoding* [62] borrows ideas from both *binary* and *commander* encodings, and the experiments show that the new encoding is very competitive compared to other state-of-art encodings. The encoding is obtained as follows: we partition a set of input variables $X = \{x_1,\dots,x_n\}$ into $m$ disjoint subsets $\{G_1,\dots,G_m\}$ such that each subset consists of $g = \lceil n/m \rceil$ variables. This step is similar to the commander encoding, but instead of using $m$ commander variables, we introduce auxiliary variables $\{b_1,\dots,b_{\lceil \log m \rceil}\}$, just like in the binary encoding. The new variables take over the role of commander variables in the new encoding. The bimander encoding produces the following set of clauses:

$$\bigwedge_{i=1}^{m} AMO(G_i) \wedge \bigwedge_{i=1}^{m} \bigwedge_{h=1}^{g} \bigwedge_{j=1}^{\lceil \log m \rceil} (x_{i,h} \Rightarrow B(i,j)),$$

where $x_{i,h}$ is the $h$-th element in $G_i$ and $B(i,j)$ is defined the same as in binary encoding.

**Example 3.8.** In the running example, if we set $m = 3$, thus obtaining $G_1 = \{x_1, x_2, x_3\}$, $G_2 = \{x_4, x_5, x_6\}$ and $G_3 = \{x_7, x_8\}$, and we use the binomial encoding for the AMO part, then the bimander encoding produces the following set of clauses:

$$
\begin{aligned}
&(\neg x_1 \vee \neg x_2) \wedge (\neg x_1 \vee \neg x_3) \wedge (\neg x_2 \vee \neg x_3) \wedge \\
&(\neg x_4 \vee \neg x_5) \wedge (\neg x_4 \vee \neg x_6) \wedge (\neg x_5 \vee \neg x_6) \wedge \\
&(\neg x_7 \vee \neg x_8) \\
&\wedge \\
&(x_1 \Rightarrow \neg b_1) \wedge (x_1 \Rightarrow \neg b_2) \wedge (x_2 \Rightarrow \neg b_1) \wedge (x_2 \Rightarrow \neg b_2) \wedge \\
&(x_3 \Rightarrow \neg b_1) \wedge (x_3 \Rightarrow \neg b_2) \wedge (x_4 \Rightarrow b_1) \wedge (x_4 \Rightarrow \neg b_2) \wedge \\
&(x_5 \Rightarrow b_1) \wedge (x_5 \Rightarrow \neg b_2) \wedge (x_6 \Rightarrow b_1) \wedge (x_6 \Rightarrow \neg b_2) \wedge \\
&(x_7 \Rightarrow \neg b_1) \wedge (x_7 \Rightarrow b_2) \wedge (x_8 \Rightarrow \neg b_1) \wedge (x_8 \Rightarrow b_2).
\end{aligned}
$$

## 3.3 Pseudo-Boolean Constraints

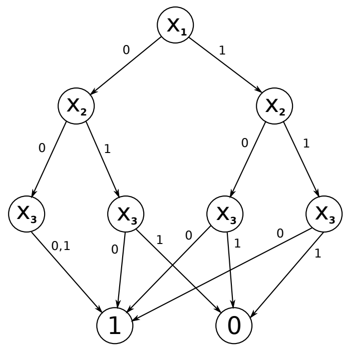
The current trend in encoding cardinality constraints involve comparator networks. The experiments show vast superiority over other approaches. Nevertheless, some methods for encoding PB-constraints are worth mentioning here (several were already referenced in Section 3.1), as PB-constraints are a superset of cardinality constraints. For example, Eén and Sörensson [34] developed a PB-solver called MiniSat+, where the solver chooses between three techniques to generate SAT encodings for Pseudo-Boolean constraints. These convert the constraint to: a BDD structure, a network of binary adders, a network of sorters. The network of adders is the most concise encoding, but it can have poor propagation properties and often leads to longer computations than the BDD based encoding. We introduce two techniques that are the basis for what is considered to be the current state-of-the-art in PB-solving.

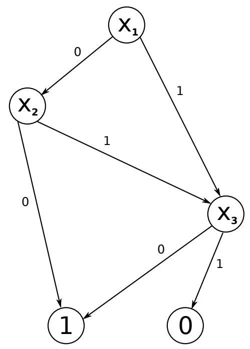
**Reduced Ordered BDDs.** Recent development in PB-solvers show superiority of encodings based on *Binary Decision Diagrams* (BDDs). The main advantage of BDD-based encodings is that the resulting size of the formula is not dependent on the size of the coefficients of a PB-constraint. The first to apply BDDs in the context of encoding PB-constraints were Bailleux, Boufkhad and Roussel [13]. In the worst case, the size of the resulting CNF formula of their BDD encoding is exponential with respect to the size of the encoded PB-constraint, but when applied to cardinality constraints, the encoding is arc-consistent and uses $O(n^2)$ variables and clauses.

Abío et al. [2] show a construction of *Reduced Ordered BDDs* (ROBDDs), which produce arc-consistent, efficient encoding for PB-constraints. Here we briefly describe their method. A Reduced Ordered BDD for a PB-constraint $a_1x_1 + a_2x_2 + \cdots + a_nx_n \leq k$ is obtained as follows. An ordering of the variables is established, suppose that it is $\langle x_1, x_2, \ldots, x_n \rangle$, for convenience. We build a directed graph with a root node $x_1$. A node has two children: false child and true child. False child represent the PB-constraint assuming $x_1 = 0$ (i.e., $a_2x_2 + a_3x_3 + \ldots a_nx_n \leq k$), and its true child represents $a_2x_2 + a_3x_3 \leq k - a_1$. The process is repeated until we reach the last variable. Then, a constraint of the form $0 \leq K$ is the true node (1) if $K \geq 0$, and the false node (0) if $K < 0$. This results in what is called an Ordered BDD. For obtaining a Reduced Ordered BDD, two reductions are applied (until fix-point): removing nodes with identical children and merging isomorphic subtrees. This reduces the size of the initial BDD. We encode BDDs into CNFs by introducing an auxiliary variable $a$ for every node. If the select variable of the node is $x$ and the auxiliary variables for the false and true child are $f$ and $t$, respectively, then add the *if-then-else* clauses:

$$\neg x \wedge \neg f \Rightarrow \neg a \qquad x \wedge \neg t \Rightarrow \neg a \qquad \neg f \wedge \neg t \Rightarrow \neg a$$
$$\neg x \wedge f \Rightarrow a \qquad x \wedge t \Rightarrow a \qquad f \wedge t \Rightarrow a$$

**Example 3.9.** This example is taken from Section 2 of [2]. Consider a PB-constraint $2x_1 + 3x_2 + 5x_3 \leq 6$ and the ordering $\langle x_1, x_2, x_3 \rangle$. The Ordered BDD for this constraint looks like in the left figure:




The root node has $x_1$ as selector variable. Its false child represent the PB-constraint assuming $x_1 = 0$ (i.e., $3x_2 + 5x_3 \leq 6$), and its true child represents $2 + 3x_2 + 5x_3 \leq 6$, that is, $3x_2 + 5x_3 \leq 4$. The two children have the next variable ($x_2$) as selector, and the process is repeated until we reach the last variable. Then, a constraint of the form $0 \leq K$ is the true node (1 on the graph) if $K \geq 0$, and the false node (0 on the graph) if $K < 0$. The Reduced Ordered BDD for this constraint is presented in the right figure above.

In [2] authors show how to produce polynomial-sized ROBDDs and how to encode them into SAT with only 2 clauses per node, and present experimental results that confirm that their approach is competitive with other encodings and state-of-the-art Pseudo-Boolean solvers. They present a proof that there are PB-constraints that admit no polynomial-size ROBDD, regardless of the variable order, but they also show how to overcome the possible exponential blowup of BDDs by carefully decomposing the coefficients of a given PB-constraint.

For further improvements, one can look at the work of Sakai and Nabeshima [70], where they extend the ROBDD construction to support constraints in the band form: $l \leq \langle \text{Linear term} \rangle \leq h$. They also propose an incremental SAT-solving strategy of binary/alternative search for minimizing values of a given goal function and their experiments show significant speed-up in SAT-solver runtime.

**Sorting Networks.** We revisit the concept of using sorting networks, which have been successfully applied to encode cardinality constraints. To demonstrate how sorters can be used to translate PB-constraints, consider the following example from [34]:

$$x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + 2z_1 + 3z_2 \geq 4$$

The sum of coefficients is 11. We build a sorting network of size 11, feeding $z_1$ into two of the inputs, $z_2$ into three of the inputs, and all the signals $x_i$ into one input each. To assert the constraint, one just asserts the fourth output bit of the sorter, like in Figure 3.2.

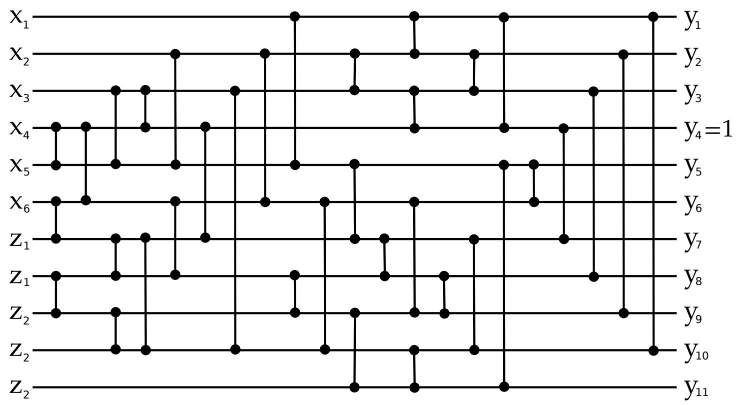



Figure 3.2: Sorting network with 11 inputs. Fourth output is set to 1 in order to assert the $\geq 4$ constraints. The network uses 35 comparators, which is almost optimal (the current best known lower bound is 33 [29]).

The shortcoming of this approach is that the resulting size of a CNF after transformation of the sorting network can get exponential if the coefficients get bigger. Consider an example from [2]: both constraints $3x_1 + 2x_2 + 4x_3 \leq 5$ and $30001x_1 + 19999x_2 + 39998x_3 \leq 50007$ are equivalent. The Boolean function they represent can be expressed, for example, by the clauses $\bar{x}_1 \vee \bar{x}_3$ and $\bar{x}_2 \vee \bar{x}_3$. But clearly, a sorting network for the left constraint would be smaller.

To remedy this situation the authors of MINISAT+ propose a method to decompose the constraint into a number of interconnected sorting networks, where sorters play the role of adders on unary numbers in a *mixed radix representation*.

In the classic base $r$ radix system, positive integers are represented as finite sequences of digits $\mathbf{d} = \langle d_0, \ldots, d_{m-1} \rangle$ where for each digit $0 \leq d_i < r$, and for the most significant digit, $d_{m-1} > 0$. The integer value associated with $\mathbf{d}$ is $v = d_0 + d_1 r + d_2 r^2 + \cdots + d_{m-1} r^{m-1}$. A mixed radix system is a generalization where a base $\mathbf{B}$ is a sequence of positive integers $\langle r_0, \ldots, r_{m-1} \rangle$. The integer value associated with $\mathbf{d}$ is $v = d_0 w_0 + d_1 w_1 + d_2 w_2 + \cdots + d_{m-1} w_{m-1}$ where $w_0 = 1$ and for $i \geq 0$, $w_{i+1} = w_i r_i$. For example, the number $\langle 2, 4, 10 \rangle_{\mathbf{B}}$ in base $\mathbf{B} = \langle 3, 5 \rangle$ is interpreted as $2 \times \mathbf{1} + 4 \times \mathbf{3} + 10 \times \mathbf{15} = 164$ (values of $w_i$'s in boldface).

The decomposition of a PB-constraint into sorting networks is roughly as follows: first, find a "suitable" finite base $\mathbf{B}$ for the set of coefficients, for example, in MINISAT+ base is chosen so that the sum of all the digits of the coefficients written in that base, is as small as possible. Then for each element $r_i$ of $\mathbf{B}$ construct a sorting network where the inputs to $i$-th sorter are those digits $\mathbf{d}$ (from the coefficients) where $d_i$ is non-zero, plus the potential carry bits from the $(i-1)$-th sorter.

**Example 3.10.** We show a construction of a sorting network system using an example from [28], where authors show a step-by-step process of translating a PB-constraint $\psi = 2x_1 + 2x_2 + 2x_3 + 2x_4 + 5x_5 + 18x_6 \geq 23$. Let $\mathbf{B} = \langle 2, 3, 3 \rangle$ be the considered mixed radix base. The representation of the coefficients of $\psi$ in base $\mathbf{B}$ is illustrated as a $6 \times 4$ matrix:

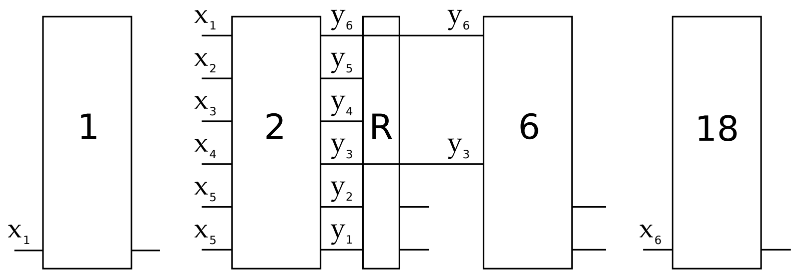
$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 2 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The rows of the matrix correspond to the representation of the coefficients in base **B**. Weights of the digit positions of base **B** are $\bar{w} = \langle 1, 2, 6, 18 \rangle$. Thus, the decomposition of the LHS of $\psi$ is:

$$\mathbf{1} \cdot (x_5) + \mathbf{2} \cdot (x_1 + x_2 + x_3 + x_4 + 2x_5) + \mathbf{6} \cdot (0) + \mathbf{18} \cdot (x_6)$$

Now we construct a series of four sorting networks in order to encode the sums at each digit position of $\bar{w}$. Given values for the variables, the sorted outputs from these networks represent unary numbers $d_1$,$d_2$,$d_3$,$d_4$ such that the LHS of $\psi$ takes the value $\mathbf{1} \cdot d_1 + \mathbf{2} \cdot d_2 + \mathbf{6} \cdot d_3 + \mathbf{18} \cdot d_4$.

The final step is to encode the carry operation from each digit position to the next. The first three outputs must represent valid digits (in unary) for **B**. In our example the single potential violation to this is $d_2$, which is represented in 6 bits. To this end we add two components to the encoding: (1) each third output of the second network is fed into the third network as carry input; and (2) a *normalizer* $R$ is added to encode that the output of the second network is to be considered modulo 3. The full construction is illustrated below:




In the end, to enforce the constraint, we have to add clauses representing the relation $\geq 23$ (in base **B**). It is done by lexicographical comparison of bits representing LHS to bits representing $23_{(\mathbf{B})}$. See [34] for a detailed description of the algorithm.

On a final note, some research has been done on finding optimal mixed radix base for the aforementioned construction. For example, Codish et al. [28] present an algorithm which scales to find an optimal base consisting of elements with values up to $1,000,000$ and they consider several measures of optimality for finding the base. They show experimentally that in many cases finding a better base leads also to better SAT-solving time.

## 3.4 Summary

The list of encodings presented here is not exhaustive, as many more encodings have been proposed in the past for different types of constraints. For at-most-one constraints one can look into the log encoding [76], ladder encoding [37], and generalizations of the bimander encoding [15]. For at-most-k constraints there exists, for example, the partial sum encoding [6] and perfect hashing encoding [18].

Here we present the comparison of the encodings introduced in the previous sections. All at-most-k constraints can be reduced to at-most-one constraints by setting $k = 1$. On the other hand, some encodings of at-most-one constraints have generalized constructions for the at-most-k constraints, for example, Firsch and Giannaros [35] give generalizations for binary, commander and product encodings. For the binomial encoding the number of clauses grows significantly when considering the at-most-k constraint. In the worst case of $k = \lceil n/2 \rceil - 1$ it requires $O(2^n/\sqrt{n/2})$ clauses. We summarize the encodings in Table 3.1.

| Method | Type | Origin | New vars. | Clauses | AC |
|---|---|---|---|---|---|
| binomial | $\leq 1$ | folklore | 0 | $\binom{n}{2}$ | yes |
| | $\leq k$ | | 0 | $\binom{n}{k+1}$ | yes |
| binary | $\leq 1$ | Frisch et al. [36] | $O(\log n)$ | $O(n \log n)$ | yes |
| | $\leq k$ | Firsch & Giannaros [35] | $O(kn)$ | $O(kn \log n)$ | no |
| commander | $\leq 1$ | Kwon & Klieber [49] | $n/2$ | $3.5n$ | yes |
| | $\leq k$ | Firsch & Giannaros [35] | $kn/2$ | $\left(\binom{2k+2}{k+1} + \binom{2k+2}{k-1} \cdot n/2\right)$ | yes |
| product | $\leq 1$ | Chen [26] | $2\sqrt{n} + O(\sqrt[4]{n})$ | $2n + 4\sqrt{n} + O(\sqrt[4]{n})$ | yes |
| | $\leq k$ | Firsch & Giannaros [35] | $(k+1)O(\sqrt[k]{n})$ | $(k+1)(n + O(k\sqrt[k]{n}))$ | yes |
| bimander | $\leq 1$ | Mai & Nguyen [62] | $\lceil \log m \rceil$ | $n^2/2m + n\lceil \log m \rceil - n/2$ | yes |
| adders | $\leq k$ | Warners [77] | $2n$ | $8n$ | no |
| | PB | | $2n(1 + \log(a_{\max}))$ | $8n(1 + 2\log(a_{\max}))$ | no |
| totalizers | $\leq k$ | Büttner & Rintanen [23] | $O(kn)$ | $O(k^2 n)$ | yes |
| seq. counter | $\leq 1$ | Sinz [74] | $n-1$ | $3n-4$ | yes |
| | $\leq k$ | | $k(n-1)$ | $2nk + n - 3k - 1$ | yes |
| par. counter | $\leq k$ | | $2n-2$ | $7n - 3\lfloor \log n \rfloor - 6$ | no |
| BDDs | $\leq k$ | Bailleux et al. [13] | $O(n^2)$ | $O(n^2)$ | yes |
| | PB | Abío et al. [2] | $O(n^3 \log(a_{\max}))$ | $O(n^3 \log(a_{\max}))$ | yes |
| sort. net. | $\leq k$ | Eén & Sörensson [34] | $O(n \log^2 n)$ | $O(n \log^2 n)$ | yes |
| | PB | | $O((\sum a_i) \log^2(\sum a_i))$ | $O((\sum a_i) \log^2(\sum a_i))$ | yes |
| sel. net. | $\leq k$ | [3, 10, 27, 46] | $O(n \log^2 k)$ | $O(n \log^2 k)$ | yes |

Table 3.1: Comparison of different encodings for at-most-one, at-most-k and Pseudo-Boolean constraints. We report on the number of new variables that needs to be introduced, the number of generated clauses and whether an encoding achieves some form of arc-consistency.

One can also compare different encodings based on other measures. For example, Chen [26] reports that his product AMO encoding is better than sequential AMO encoding and binary AMO encoding in terms of total number of literals appearing in the clauses. Chen's product encoding requires $4n + 8\sqrt{n} + O(\sqrt[4]{n})$ literals, while sequential encoding and binary encoding requires $6n - 8$ and $2n \log n$ literals, respectively.

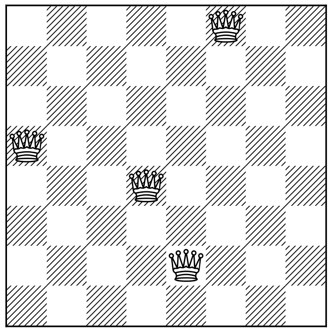
For at-most-one constraints one can also take a look at a very interesting, recent, theoretical result by Kučera et al. [51]. In their paper authors show a lower bound for the number of clauses the encoding for AMO constraints needs to have in order to preserve the *complete propagation* property – a generalization of arc-consistency in which we not only require consistency enforced on the input variables (as in Definition 2.12) but for the auxiliary variables as well. The lower bound is $2n + \sqrt{n} - O(1)$ and the product encoding is the closest to that barrier.

# Part II

# Pairwise Selection Networks

# Chapter 4

# Pairwise Bitonic Selection Networks



It has already been observed that using selection networks instead of sorting networks is more efficient for the encoding of cardinality constraints. Codish and Zazon-Ivry [27] introduced Pairwise Cardinality Networks, which are networks derived from pairwise sorting networks that express cardinality constraints. Two years later, same authors [78] reformulated the definition of Pairwise Selection Networks and proved that their sizes are never worse than the sizes of corresponding Odd-Even Selection Networks. To show the difference they plotted it for selected values of $n$ and $k$.

In this chapter we give a new construction of smaller selection networks that are based on the pairwise selection ones and we prove that the construction is correct. We estimate also the size of our networks and compute the difference in sizes between our selection networks and the corresponding pairwise ones. The difference can be as big as $n \log n/2$ for $k = n/2$.

To simplify the presentation we assume that $n$ and $k$ are powers of 2. The networks in this chapter are presented such that the inputs can be over any totally ordered set $X$. In the context of encoding Boolean constraints we would like to set $X = \{0,1\}$, but the proofs in this chapter are general enough to work with any $X$.

## 4.1 Pairwise Selection Network

Here we present the basis for our constructions in this chapter (and the next chapter). It is called the *Pairwise Selection Network* and it was created by Codish and Zazon-Ivry [78]. This class of networks uses a component called a *splitter*.

**Definition 4.1** (splitter)**.** A comparator network $f^n$ is a *splitter* if for any sequence $\bar{x} \in X^n$, if $\bar{y} = f^n(\bar{x})$, then $\bar{y}_{left}$ weakly dominates $\bar{y}_{right}$.

**Observation 4.1.** *The splitter (on n inputs) – denoted as* $split^n$ *from now on – can be constructed by comparing inputs i and* $i+n/2$*, for* $i = 1..n/2$ *(see Figure 4.1a).*

The construction is presented in Algorithm 4.1. The sub-procedures used are: $max^n$ – select maximum element out of $n$ inputs, and $pw\_merge^n_k$ – a *Pairwise Merging Network* (Algorithm 4.2). If $k = n$ we need to sort the input sequence, therefore we use the odd-even sorting network (Algorithm 2.2) in this case. The last step of Algorithm 4.1 produces a top $k$ sorted sequence given the outputs of the recursive calls.

**Algorithm 4.1** $pw\_sel^n_k$

**Input:** $\bar{x} \in X^n$; $n$ and $k$ are powers of 2; $1 \le k \le n$
**Ensure:** The output is top $k$ sorted and is a permutation of the inputs
1: **if** $k = 1$ **then return** $max^n(\bar{x})$
2: **if** $k = n$ **then return** $oe\_sort^n(\bar{x})$
3: $\bar{y} \leftarrow split^n(\bar{x})$
4: $\bar{l} \leftarrow pw\_sel^{n/2}_{\min(n/2,k)}(\bar{y}_{left})$
5: $\bar{r} \leftarrow pw\_sel^{n/2}_{\min(n/2,k/2)}(\bar{y}_{right})$
6: **return** $pw\_merge^n_k(\bar{l} :: \bar{r})$

**Algorithm 4.2** $pw\_merge^n_k$

**Input:** $\bar{l} :: \bar{r} \in X^n$; $|l| = |r|$; $\bar{l}$ is top $k$ sorted, $\bar{r}$ is top $k/2$ sorted and $\mathrm{pref}(k/2, \bar{l}) \succeq_w \mathrm{pref}(k/2, \bar{r})$; $n$ and $k$ are powers of 2; $1 \le k < n$
**Ensure:** The output is top $k$ sorted and is a permutation of the inputs
1: **if** $n \le 2$ **or** $k = 1$ **then return** $\mathrm{zip}(\bar{l}, \bar{r})$
2: $\bar{y} \leftarrow pw\_merge^{n/2}_{k/2}(\bar{l}_{odd} :: \bar{r}_{odd})$
3: $\bar{y}' \leftarrow pw\_merge^{n/2}_{k/2}(\bar{l}_{even} :: \bar{r}_{even})$
4: $\bar{z} \leftarrow \mathrm{zip}(\bar{y}, \bar{y}')$, $z'_1 = z_1$, $\bar{z}'_{2k..n} = \bar{z}_{2k..n}$
5: **for all** $i \in \{1, \dots, k-1\}$ **do** $\langle z'_{2i}, z'_{2i+1} \rangle \leftarrow sort^2(z_{2i}, z_{2i+1})$
6: **return** $\bar{z}'$

Notice that since we introduced a splitter (Step 3), in the recursive calls we need to select $k$ top elements from the first half of $\bar{y}$, but only $k/2$ elements from the second half. The reason: $r_{k/2+1}$ cannot be one of the first $k$ largest elements of $\bar{l} :: \bar{r}$. First, $r_{k/2+1}$ is smaller than any one of $\langle r_1, \dots, r_{k/2} \rangle$ (by the definition of top $k$ sorted sequence), and second, $\langle l_1, \dots, l_{k/2} \rangle$ weakly dominates $\langle r_1, \dots, r_{k/2} \rangle$, so $r_{k/2+1}$ is smaller than any one of $\langle l_1, \dots, l_{k/2} \rangle$. From this argument we make the following observation:

**Observation 4.2.** *If $\bar{l} \in X^{n/2}$ is top $k$ sorted, $\bar{r} \in X^{n/2}$ is top $k/2$ sorted and $\langle l_1, \dots, l_{k/2} \rangle$ weakly dominates $\langle r_1, \dots, r_{k/2} \rangle$, then $k$ largest elements of $\bar{l} :: \bar{r}$ are in $\langle l_1, \dots, l_k \rangle :: \langle r_1, \dots, r_{k/2} \rangle$.*

We would like to note, that the number of comparators used in the merger is: $|pw\_merge^n_k| = k \log k - k + 1$. The detailed proof of correctness of network $pw\_sel^n_k$ can be found in Section 6 of [78]. The networks in [78] are given in the functional representation.

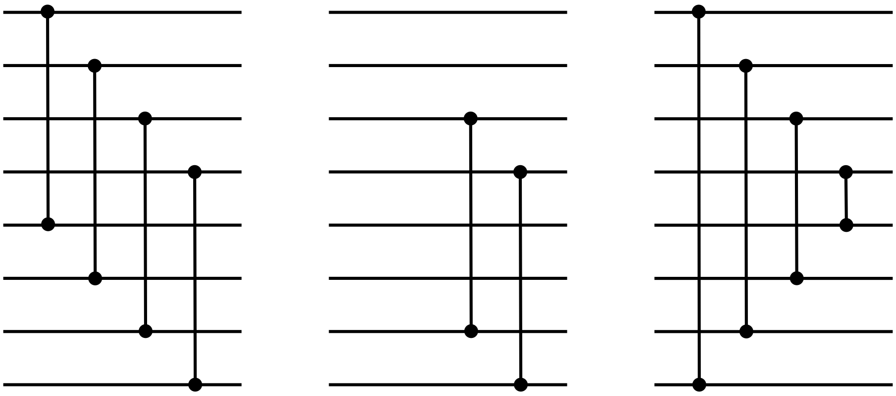
**Theorem 4.1.** *Let $n, k \in \mathbb{N}$, where $1 \le k \le n$ and let $n$ and $k$ be powers of 2. Then $|pw\_sel^n_k| \le |oe\_sel^n_k|$, where $oe\_sel^n_k$ is the Odd-Even Selection Network, for which $|oe\_sel^n_k| = (n/4)(\log^2 k + 3 \log k + 4) - k \log k - 1$.*

*Proof.* See Theorems 11 and 14 of [78]. $\square$

## 4.2 Bitonic Selection Network

We now present the construction of the *Bitonic Selection Network*. We use it to estimate the sizes of our improved pairwise selection network from the next section. We begin with a useful property of splitters and bitonic sequences proved by Batcher:

**Lemma 4.1.** *If $\bar{b} \in X^n$ is bitonic and $\bar{y} = split^n(\bar{b})$, then $\bar{y}_{left}$ and $\bar{y}_{right}$ are bitonic and $\bar{y}_{left} \succeq \bar{y}_{right}$.*



(a) (b) (c)

Figure 4.1: a) splitter; b) half-splitter; c) bitonic splitter

*Proof.* See Appendix B of [17]. □

**Definition 4.2** (bitonic splitter)**.** A comparator network $f^n$ is a bitonic splitter if for any two sorted sequences $\bar{x}, \bar{y} \in X^{n/2}$, if $\bar{z} = bit\_split^n(\bar{x} :: \bar{y})$, then (1) $\bar{z}_{left} \succeq \bar{z}_{right}$ and (2) $\bar{z}_{left}$ and $\bar{z}_{right}$ are bitonic.

**Observation 4.3.** *We can construct a bitonic splitter $bit\_split^n$ by joining inputs $\langle i, n-i+1 \rangle$, for $i = 1..n/2$, with a comparator (see Figure 4.1c). Notice that this is a consequence of Lemma 4.1, because given two sorted sequences $\langle x_1, \dots, x_n \rangle$ and $\langle y_1, \dots, y_n \rangle$, a sequence $\langle x_1, \dots, x_n \rangle :: \langle y_n, \dots, y_1 \rangle$ is bitonic. Size of a bitonic splitter is $|bit\_split^n| = n/2$.*

We now present the procedure for construction of the Bitonic Selection Network. We use the odd-even sorting network *oe_sort* and the network *bit_merge* (also by Batcher [17]) for sorting bitonic sequences, as black-boxes. As a reminder: $bit\_merge^n$ consists of two steps, first we use $\bar{y} = split^n(\bar{x})$, then recursively compute $bit\_merge^{n/2}$ for $\bar{y}_{left}$ and $\bar{y}_{right}$ (base case, $n = 2$, consists of a single comparator). In Figure 4.5a we present $bit\_merge^{16}$. Size of this network is: $|bit\_merge^n| = n \log n/2$. Bitonic Selection Network $bit\_sel^n_k$ is constructed by the procedure given in Algorithm 4.3.

**Algorithm 4.3** $bit\_sel^n_k$

**Input:** $\bar{x} \in X^n$; $n$ and $k$ are powers of 2; $1 \le k \le n$
**Ensure:** The output is top $k$ sorted and is a permutation of the inputs
**let** $l = n/k$ **and** $\bar{r} = \langle\rangle$
**for all** $i \in \{0, \dots, l-1\}$ **do** $B_{i+1} \leftarrow oe\_sort^k(\langle x_{ik+1}, \dots, x_{(i+1)k} \rangle)$
**while** $l > 1$ **do**
    **for all** $i \in \{1, 3, \dots, l-1\}$ **do**
        $\bar{y}^i \leftarrow bit\_split^{2k}(B_i :: B_{i+1})$
        $B'_{\lceil i/2 \rceil} \leftarrow bit\_merge^k(\bar{y}^i_{left})$
        $\bar{r} \leftarrow \bar{r} :: \bar{y}^i_{right}$    # residue elements
    **let** $l = l/2$ **and** relabel $B'_i$ to $B_i$, for $1 \le i \le l$.
**return** $B_1 :: \bar{r}$

First, we partition input $\bar{x}$ into $l$ consecutive blocks, each of size $k$, then we sort each block with $oe\_sort^k$, obtaining $B_1, \dots, B_l$. Then, we collect blocks into pairs $\langle B_1, B_2 \rangle, \dots, \langle B_{l-1}, B_l \rangle$ and perform a bitonic splitter on each of them. By Lemma 4.1 $k$ largest elements in $\bar{y}^i$ are in $\bar{y}^i_{left}$, and

$\bar{y}^i_{left}$ is bitonic, therefore we can use bitonic merger (Step 6) to sort it. The algorithm continues until one block remains.

**Theorem 4.2.** *A comparator network $bit\_sel^n_k$ is a selection network.*

*Proof.* Let $\bar{x} \in X^n$ be the input to $bit\_sel^n_k$. After Step 2 we get sorted sequences $B_1, \ldots, B_l$, where $l = n/k$. Let $l_m$ be the value of $l$ after $m$ iterations of the loop in Step 3. Let $B^m_1, \ldots, B^m_{l_m}$ be the blocks after $m$ iterations. We prove by induction that:

*$P(m)$: if $B_1, \ldots, B_l$ are sorted and are containing k largest elements of $\bar{x}$, then after m-th iteration of the loop in Step 3: $l_m = l/2^m$, $B^m_1, \ldots, B^m_{l_m}$ are sorted and are containing k largest elements of $\bar{x}$.*

If $m = 0$, then $l_0 = l$, so $P(m)$ holds. We show that $\forall_{m \geq 0}\ (P(m) \Rightarrow P(m+1))$. Consider $(m+1)$-th iteration of the while loop. By the induction hypothesis $l_m = l/2^m$, $B^m_1, \ldots, B^m_{l_m}$ are sorted and are containing $k$ largest elements of $\bar{x}$. We show that $(m+1)$-th iteration does not remove any element from $k$ largest elements of $\bar{x}$. To see this, notice that if $\bar{y}^i = bit\_split^{2k}(B^m_i :: B^m_{i+1})$ (for $i \in \{1, 3, \ldots, l_m - 1\}$), then $\bar{y}^i_{left} \succeq \bar{y}^i_{right}$ and that $\bar{y}^i_{left}$ is bitonic (by Definition 4.2). Because of those two facts, $\bar{y}^i_{right}$ is discarded and $\bar{y}^i_{left}$ is sorted using $bit\_merge^k$. After this, $l_{m+1} = l_m/2 = l/2^{m+1}$ and blocks $B^{m+1}_1, \ldots, B^{m+1}_{l_{m+1}}$ are sorted. Thus $P(m+1)$ is true.

Since $l = n/k$, then by $P(m)$ we see that the while loop terminates after $m = \log \frac{n}{k}$ iterations and that $B_1$ is sorted and contains $k$ largest elements of $\bar{x}$. □

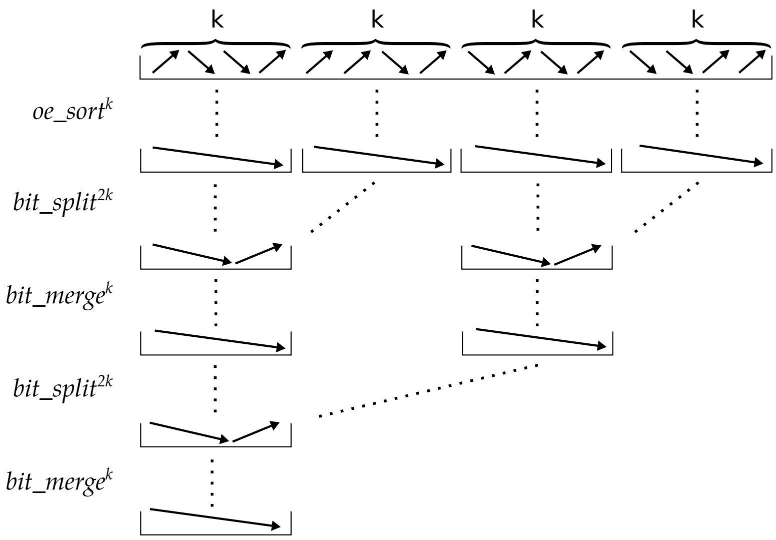



Figure 4.2: A Bitonic Selection Network – a construction diagram

A construction diagram of the Bitonic Selection Network is shown in Figure 4.2. The size of the Bitonic Selection Network is:

$$
\begin{aligned}
|bit\_sel^n_k| &= \frac{n}{k}|oe\_sort^k| + \left(\frac{n}{k} - 1\right)\left(|bit\_split^{2k}| + |bit\_merge^k|\right) \\
&= \frac{1}{4}n\log^2 k + \frac{1}{4}n\log k + 2n - \frac{1}{2}k\log k - k - \frac{n}{k} \qquad (4.1)
\end{aligned}
$$

The $|oe\_sort^k|$ was already shown in Example 2.3. The rest is a straightforward calculation.

In Figure 4.3 we present a Bitonic Selection Network for $n = 8$ and $k = 2$.

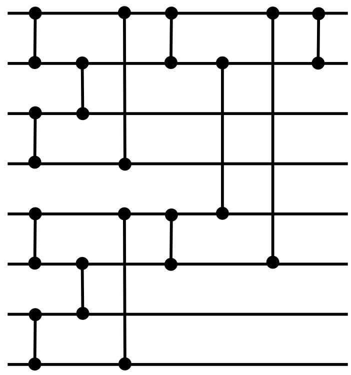


Figure 4.3: A Bitonic Selection Network for $n = 8$ and $k = 2$

**Algorithm 4.4** $pw\_bit\_merge^n_k$

**Input:** $\bar{l} :: \bar{r} \in X^n$; $\bar{l}$ is top $k$ sorted, $\bar{r}$ is top $k/2$ sorted; $\mathrm{pref}(k/2, \bar{l}) \succeq_w \mathrm{pref}(k/2, \bar{r})$; $k$ is a power of 2

**Ensure:** The output is top $k$ sorted and is a permutation of the inputs

1: $\bar{y} \leftarrow bit\_split^k(l_{k/2+1}, \ldots, l_k, r_1, \ldots, r_{k/2})$
2: $\bar{b} \leftarrow \langle l_1, \ldots, l_{k/2} \rangle :: \langle y_1, \ldots, y_{k/2} \rangle$
3: $\bar{p} \leftarrow \mathrm{suff}(k/2, \bar{y}) :: \mathrm{suff}(n/2 - k, \bar{l}) :: \mathrm{suff}(n/2 - k/2, \bar{r})$ # residue elements
4: **return** $bit\_merge^k(\bar{b}) :: \bar{p}$

## 4.3 Pairwise Bitonic Selection Network

As mentioned in Section 4.1, only the first $k/2$ elements from the second half of the input are relevant when we get to the merging step in $pw\_sel^n_k$. We exploit this fact to create a new, smaller merger. We use the concept of bitonic sequences, therefore we call the new merger $pw\_bit\_merge^n_k$ and the new selection network $pw\_bit\_sel^n_k$ (the *Pairwise Bitonic Selection Network*). The network $pw\_bit\_sel^n_k$ is generated by substituting the last step of $pw\_sel^n_k$ with $pw\_bit\_merge^n_k$. The new merger is presented as Algorithm 4.4.

**Theorem 4.3.** *The output of Algorithm 4.4 consists of sorted k largest elements from input $\bar{l} :: \bar{r}$, assuming that $\bar{l} \in X^{n/2}$ is top k sorted and $\bar{r} \in X^{n/2}$ is top $k/2$ sorted and $\langle l_1, \ldots, l_{k/2} \rangle$ weakly dominates $\langle r_1, \ldots, r_{k/2} \rangle$.*

*Proof.* We have to prove two things: (1) $\bar{b}$ is bitonic and (2) $\bar{b}$ consists of $k$ largest elements from $\bar{l} :: \bar{r}$.

(1) Let $j$ be the last index in the sequence $\langle k/2+1, \ldots, k \rangle$, for which $l_j > r_{k-j+1}$. If such $j$ does not exist, then $\langle y_1, \ldots, y_{k/2} \rangle$ is non-decreasing, hence $\bar{b}$ is bitonic (non-decreasing). Assume that $j$ exists, then $\langle y_{j-k/2+1}, \ldots, y_{k/2} \rangle$ is non-decreasing and $\langle y_1, \ldots, y_{k-j} \rangle$ is non-increasing. Adding the fact that $l_{k/2} \geq l_{k/2+1} = y_1$ proves, that $\bar{b}$ is bitonic (v-shaped).

(2) By Observation 4.2, it is sufficient to prove that $\bar{b} \succeq \langle y_{k/2+1}, \ldots, y_k \rangle$. Since $\forall_{k/2<j\leq k}\, l_{k/2} \geq l_j \geq \min\{l_j, r_{k-j+1}\} = y_{3k/2-j+1}$, then $\langle l_1, \ldots, l_{k/2} \rangle \succeq \langle y_{k/2+1}, \ldots, y_k \rangle$ and by Definition 4.2: $\langle y_1, \ldots, y_{k/2} \rangle \succeq \langle y_{k/2+1}, \ldots, y_k \rangle$. Therefore $\bar{b}$ consists of $k$ largest elements from $\bar{l} :: \bar{r}$.

The bitonic merger in Step 4 receives a bitonic sequence, so it outputs a sorted sequence, which completes the proof. $\square$

The first step of improved pairwise merger is illustrated in Figure 4.4. We use $k/2$ comparators in the first step and $k \log k/2$ comparators in the last step. We get a merger of size $k \log k/2 + k/2$, which is better than the previous approach ($k \log k - k + 1$). In the following we show that we can do even better and eliminate the $k/2$ term.

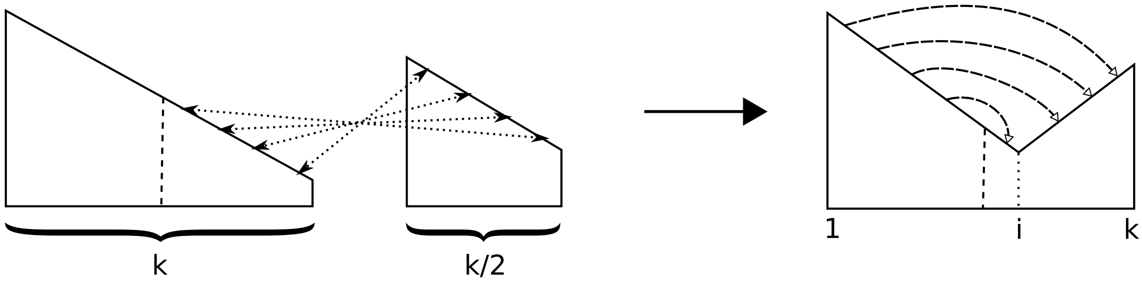



Figure 4.4: Constructing a bitonic sequence. Arrows on the right picture show directions of inequalities. Sequence on the right is v-shape s-dominating at point $i$.

The main observation is that the result of the first step of *pw_bit_merge* operation $\langle b_1, \ldots, b_k \rangle$ is not only bitonic, but what we call *v-shape s-dominating*.

**Definition 4.3** (s-domination)**.** A sequence $\bar{b} = \langle b_1, \ldots, b_k \rangle$ is s-dominating if $\forall_{1 \le j \le k/2}\ b_j \ge b_{k-j+1}$.

**Lemma 4.2.** *If $\bar{b} = \langle b_1, \ldots, b_k \rangle$ is v-shaped and s-dominating, then (1) $\bar{b}$ is non-increasing or (2) $\exists_{k/2<i<k}\ b_i < b_{i+1}$.*

*Proof.* Assume that $\bar{b}$ is not non-increasing. Then $\exists_{1 \le j < k}\ b_j < b_{j+1}$. Assume that $j \le k/2$. Since $\bar{b}$ is v-shaped, $b_{j+1}$ must be in non-decreasing part of $\bar{b}$. If follows that $b_j < b_{j+1} \le \ldots \le b_{k/2} \le \ldots \le b_{k-j+1}$. That means that $b_j < b_{k-j+1}$. On the other hand, $\bar{b}$ is s-dominating, thus $b_j \ge b_{k-j+1}$ – a contradiction. □

We say that a sequence $\bar{b}$ is *v-shape s-dominating at point i* if $i$ is the smallest index greater than $k/2$ such that $b_i < b_{i+1}$ or $i = k$ for a non-increasing sequence.

**Lemma 4.3.** *Let $\bar{b} = \langle b_1, \ldots, b_k \rangle$ be v-shape s-dominating at point i, then $\langle b_1, \ldots, b_{k/4} \rangle \succeq \langle b_{k/2+1}, \ldots, b_{3k/4} \rangle$.*

*Proof.* If $\bar{b}$ is non-increasing, then the lemma holds. From Lemma 4.2: $k/2 < i < k$. If $i > 3k/4$, then by Definition 2.3: $b_1 \ge \ldots \ge b_{3k/4} \ge \ldots \ge b_i$, so lemma holds. If $k/2 < i \le 3k/4$, then by Definition 2.3: $b_1 \ge \ldots \ge b_i$, so $\langle b_1, \ldots, b_{k/4} \rangle \succeq \langle b_{k/2+1}, \ldots, b_i \rangle$. Since $b_i < b_{i+1} \le \ldots \le b_{3k/4}$, it suffices to prove that $b_{k/4} \ge b_{3k/4}$. By Definition 4.3 and 2.3: $b_{k/4} \ge b_{3k/4+1} \ge b_{3k/4}$. □

**Definition 4.4** (half-splitter)**.** A *half-splitter* is a comparator network constructed by comparing inputs $\langle k/4+1, 3k/4+1 \rangle, \ldots, \langle k/2, k \rangle$ (normal splitter with first $k/4$ comparators removed; see Figure 4.1b). We call it $half\_split^k$.

**Lemma 4.4.** *If $\bar{b}$ is v-shape s-dominating, then $half\_split^k(\bar{b}) = split^k(\bar{b})$.*

*Proof.* Directly from Lemma 4.3. □

**Lemma 4.5.** *Let $\bar{b} = \langle b_1, \ldots, b_k \rangle$ be v-shape s-dominating. Let $\bar{w} = half\_split^k(\bar{b})$. The following statements are true: (1) $\bar{w}_{left}$ is v-shape s-dominating; (2) $\bar{w}_{right}$ is bitonic; (3) $\bar{w}_{left} \succeq \bar{w}_{right}$.*

**Algorithm 4.5** $pw\_hbit\_merge^n_k$

**Input:** $\bar{l} :: \bar{r} \in X^n$; $\bar{l}$ is top $k$ sorted, $\bar{r}$ is top $k/2$ sorted; $\mathrm{pref}(k/2,\bar{l}) \succeq_w \mathrm{pref}(k/2,\bar{r})$; $k$ is a power of 2

**Ensure:** The output is top $k$ sorted and is a permutation of the inputs

1: $\bar{y} \leftarrow bit\_split^k(l_{k/2+1},\dots,l_k,r_1,\dots,r_{k/2})$
2: $\bar{b} \leftarrow \langle l_1,\dots,l_{k/2}\rangle :: \langle y_1,\dots,y_{k/2}\rangle$
3: $\bar{p} \leftarrow \mathrm{suff}(k/2,\bar{y}) :: \mathrm{suff}(n/2-k,\bar{l}) :: \mathrm{suff}(n/2-k/2,\bar{r})$ # residue elements
4: **return** $half\_bit\_merge^k(\bar{b}) :: \bar{p}$

**Algorithm 4.6** $half\_bit\_merge^k$

**Input:** $\bar{b} \in X^k$; $\bar{b}$ is v-shaped s-dominating, $k$ is a power of 2

**Ensure:** The output is sorted and is a permutation of the inputs

1: **if** $k = 2$ **then return** $\bar{b}$
2: $\bar{b'} \leftarrow half\_split(b_1,\dots,b_k)$
3: $\bar{l'} \leftarrow half\_bit\_merge^{k/2}(\bar{b'}_{left})$
4: $\bar{r'} \leftarrow bit\_merge^{k/2}(\bar{b'}_{right})$
5: **return** $\bar{l'} :: \bar{r'}$

*Proof.* (1) Let $\bar{y} = \bar{w}_{left}$. First, we show that $\bar{y}$ is v-shaped. If $\bar{y}$ is non-increasing, then it is v-shaped. Otherwise, let $j$ be the first index from the range $\{1,\dots,k/2\}$, where $y_{j-1} < y_j$. Since $y_j = \max\{b_j,b_{j+k/2}\}$ and $y_{j-1} \geq b_{j-1} \geq b_j$, thus $b_j < b_{j+k/2}$. Since $\bar{b}$ is v-shaped, element $b_{j+k/2}$ must be in non-decreasing part of $\bar{b}$. It follows that $b_j \geq \dots \geq b_{k/2}$ and $b_{j+k/2} \leq \dots \leq b_k$. From this we can see that $\forall_{j \leq j' \leq k/2}\ y_{j'} = \max\{b_{j'},b_{j'+k/2}\} = b_{j'+k/2}$, so $y_j \leq \dots \leq y_{k/2}$. Therefore $\bar{y}$ is v-shaped.

Next, we show that $\bar{y}$ is s-dominating. Consider any $j$, where $1 \leq j \leq k/4$. By Definition 2.3 and 4.3: $b_j \geq b_{k/2-j+1}$ and $b_j \geq b_{k-j+1}$, therefore $y_j = b_j \geq \max\{b_{k/2-j+1},b_{k-j+1}\} = y_{k/2-j+1}$, thus proving that $\bar{y}$ is s-dominating. Concluding: $\bar{y}$ is v-shape s-dominating.

(2) Let $\bar{z} = \bar{w}_{right}$. By Lemma 4.4: $\bar{z} = split^k(\bar{b})_{right}$. We know that $\bar{b}$ is a special case of bitonic sequence, therefore using Lemma 4.1 we get that $\bar{z}$ is bitonic.

(3) By Lemma 4.4: $\bar{w} = split^k(\bar{b})$. We know that $\bar{b}$ is a special case of bitonic sequence, therefore using Lemma 4.1 we get $\bar{w}_{left} \succeq \bar{w}_{right}$. □

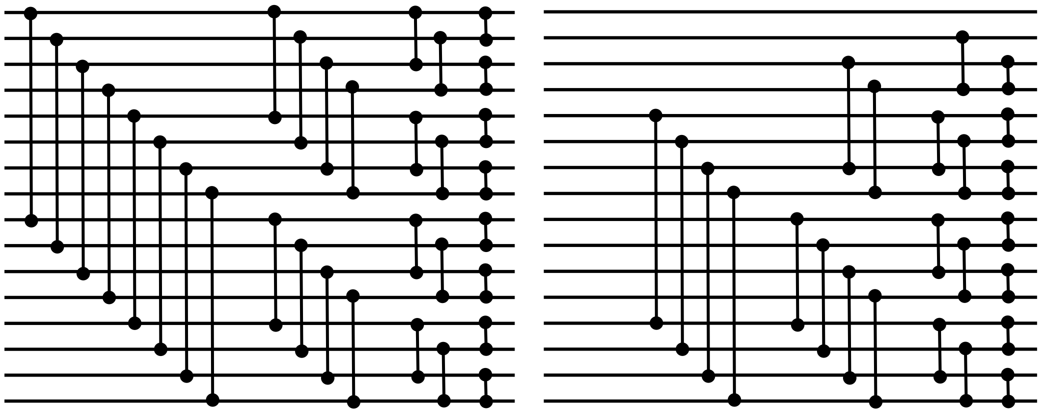
Using $half\_split$ and Batcher's $bit\_merge$ and successively applying Lemma 4.5 to the resulting v-shape s-dominating half of the output, we have all the tools needed to construct the improved pairwise merger using half-splitters, which we present as Algorithms 4.5 and 4.6.

In Figure 4.5 the difference between bitonic and half-bitonic merger is shown for $n = 16$. The following theorem states that the construction of $pw\_hbit\_merge^n_k$ is correct.

**Theorem 4.4.** *The output of Algorithm 4.5 consists of sorted k largest elements from input $\bar{l} :: \bar{r}$, assuming that $\bar{l} \in X^{n/2}$ is top k sorted and $\bar{r} \in X^{n/2}$ is top $k/2$ sorted and $\langle l_1,\dots,l_{k/2}\rangle$ weakly dominates $\langle r_1,\dots,r_{k/2}\rangle$. Also, $|pw\_hbit\_merge^n_k| = k\log k/2$.*

*Proof.* Since Step 1 in Algorithm 4.5 is the same as in Algorithm 4.4, we can reuse the proof of Theorem 4.3 to deduce, that $\bar{b}$ is v-shaped and is containing $k$ largest elements from $\bar{l} :: \bar{r}$. Also, since $\forall_{1 \leq j \leq k/2}\ l_j \geq l_{k-j+1}$ and $l_j \geq r_j$, then $b_j = l_j \geq \max\{l_{k-j+1},r_j\} = b_{k-j+1}$, so $\bar{b}$ is s-dominating.

We prove by the induction on $k$, that if $\bar{b}$ is v-shape s-dominating, then $half\_bit\_merge^k(\bar{b})$ is sorted. For the base case, consider $k = 2$ and a v-shape s-dominating sequence $\langle b_1,b_2\rangle$. By Definition 4.3 this sequence is already sorted and we are done. For the induction step, consider $\bar{b'} = half\_split^k(\bar{b})$. By Lemma 4.5 we get that $\bar{b'}_{left}$ is v-shape s-dominating and $\bar{b'}_{right}$ is bitonic.



(a) (b)

Figure 4.5: a) bitonic merging network; b) half-bitonic merging network; $n = 16$

Using the induction hypothesis we sort $\bar{b}'_{left}$ and using bitonic merger we sort $\bar{b}'_{right}$. By Lemma 4.5: $\bar{b}'_{left} \succeq \bar{b}'_{right}$, which completes the proof of correctness.

As mentioned in Definition 4.4: $half\_split^k$ is just $split^k$ with the first $k/4$ comparators removed. So $half\_bit\_merge^k$ is just $bit\_merge^k$ with some of the comparators removed. Let us count them: in each level of recursion step we take half of comparators from $split^k$ and additional one comparator from the base case ($k = 2$). We sum them together to get:

$$1 + \sum_{i=0}^{\log k - 2} \frac{k}{2^{i+2}} = 1 + \frac{k}{4}\left(\sum_{i=0}^{\log k - 1}\left(\frac{1}{2}\right)^i - \frac{2}{k}\right) = 1 + \frac{k}{4}\left(2 - \frac{2}{k} - \frac{2}{k}\right) = \frac{k}{2}$$

Therefore, we have:

$$|pw\_hbit\_merge^n_k| = k/2 + k\log k/2 - k/2 = k\log k/2$$

□

The only difference between $pw\_sel$ and our $pw\_hbit\_sel$ is the use of improved merger $pw\_hbit\_merge$ rather than $pw\_merge$. By Theorem 4.4, we can conclude that $|pw\_merge^n_k| \geq |pw\_hbit\_merge^n_k|$, so it follows that:

**Corollary 4.1.** *For* $1 \leq k \leq n$, $|pw\_hbit\_sel^n_k| \leq |pw\_sel^n_k|$.

## 4.4 Sizes of New Selection Networks

In this section we estimate the size of $pw\_hbit\_sel^n_k$. To this end we show that the size of $pw\_hbit\_sel^n_k$ is upper-bounded by the size of $bit\_sel^n_k$ and use this fact in our estimation. We also compute the exact difference between sizes of $pw\_sel^n_k$ and $pw\_hbit\_sel^n_k$ and show that it can be as big as $n\log n/2$.

We have the recursive formula for the number of comparators of $pw\_hbit\_sel^n_k$:

$$|pw\_hbit\_sel^n_k| = \begin{cases} |pw\_hbit\_sel^{n/2}_k| + |pw\_hbit\_sel^{n/2}_{k/2}| + & \\ +|split^n| + |pw\_hbit\_merge^k| & \text{if } k < n \\ |oe\_sort^k| & \text{if } k = n \\ |max^n| & \text{if } k = 1 \end{cases} \tag{4.2}$$

**Lemma 4.6.** *For* $1 \leq k < n$ *(both powers of 2),* $|pw\_hbit\_sel^n_k| \leq |bit\_sel^n_k|$.

*Proof.* Let $aux\_sel^n_k$ be the comparator network that is generated by substituting recursive calls in $pw\_hbit\_sel^n_k$ by calls to $bit\_sel^n_k$. Size of this network (for $1 < k < n$) is:

$$|aux\_sel^n_k| = |bit\_sel^{n/2}_k| + |bit\_sel^{n/2}_{k/2}| + |split^n| + |pw\_hbit\_merge^k| \tag{4.3}$$

Lemma 4.6 follows from Lemma 4.7 and Lemma 4.8 below, where we show that:

$$|pw\_hbit\_sel^n_k| \leq |aux\_sel^n_k| \leq |bit\_sel^n_k|$$

□

**Lemma 4.7.** *For $1 < k < n$ (both powers of 2), $|aux\_sel^n_k| \leq |bit\_sel^n_k|$.*

*Proof.* We compute both values from Eq. 4.1 and Eq. 4.3:

$$\begin{aligned} |aux\_sel^n_k| &= \frac{1}{4}n\log^2 k + \frac{5}{2}n - \frac{1}{4}k\log k - \frac{5}{4}k - \frac{3n}{2k} \\ |bit\_sel^n_k| &= \frac{1}{4}n\log^2 k + \frac{1}{4}n\log k + 2n - \frac{1}{2}k\log k - k - \frac{n}{k} \end{aligned}$$

We simplify both sides to get the following inequality:

$$n - \frac{1}{2}k - \frac{n}{k} \leq \frac{1}{2}(n-k)\log k$$

which can be easily proved by induction. □

**Lemma 4.8.** *For $1 \leq k < n$ (both powers of 2), $|pw\_hbit\_sel^n_k| \leq |aux\_sel^n_k|$.*

*Proof.* By induction. For the base case, consider $1 = k < n$. If follows by definitions that $|pw\_hbit\_sel^n_k| = |aux\_sel^n_k| = n - 1$. For the induction step assume that for each $(n', k') \prec (n, k)$ (in lexicographical order) the lemma holds, we get:

$$\begin{aligned} &|pw\_hbit\_sel^n_k| \\ &= |pw\_hbit\_sel^{n/2}_{k/2}| + |pw\_hbit\_sel^{n/2}_k| + |split^n| + |pw\_hbit\_merge^k| \\ &\qquad \textbf{(by the definition of } pw\_hbit\_sel\textbf{)} \\ &\leq |aux\_sel^{n/2}_{k/2}| + |aux\_sel^{n/2}_k| + |split^n| + |pw\_hbit\_merge^k| \\ &\qquad \textbf{(by the induction hypothesis)} \\ &\leq |bit\_sel^{n/2}_{k/2}| + |bit\_sel^{n/2}_k| + |split^n| + |pw\_hbit\_merge^k| \\ &\qquad \textbf{(by Lemma 4.7)} \\ &= |aux\_sel^n_k| \\ &\qquad \textbf{(by the definition of } aux\_sel\textbf{)} \end{aligned}$$

□

Let $N = 2^n$ and $K = 2^k$. We compute the upper bound for $P(n,k) = |pw\_hbit\_sel^N_K|$ using $B(n,k) = |bit\_sel^N_K|$. First, we prove a technical lemma below. The value $P(n,k,m)$ denotes the number of comparators used in the network $pw\_hbit\_sel^N_K$ after $m$ levels of recursion (of Eq. 4.2). Notice that if $0 < k < n$, then:

$$P(n,k) = P(n-1,k) + P(n-1,k-1) + k2^{k-1} + 2^{n-1} \tag{4.4}$$

Term $k2^{k-1}$ corresponds to $|pw\_hbit\_merge^K|$ and $2^{n-1}$ to $|split^N|$.

**Lemma 4.9.** *Let:*

$$P(n,k,m)=\sum_{i=0}^{m-1}\sum_{j=0}^{i}\binom{i}{j}\left((k-j)2^{k-j-1}+2^{n-i-1}\right)+\sum_{i=0}^{m}\binom{m}{i}P(n-m,k-i).$$

*Then* $\forall_{0\leq m\leq \min(k,n-k)}\ P(n,k,m)=P(n,k)$.

*Proof.* By induction on $m$. If $m=0$, then $P(n,k,0)=P(n,k)$. Choose any $m$ such that $0\leq m<\min(k,n-k)$ and assume that $P(n,k,m)=P(n,k)$. We show that $P(n,k,m+1)=P(n,k)$. We have:

$$\begin{aligned}
P(n,k,m+1)&=\sum_{i=0}^{(m-1)+1}\sum_{j=0}^{i}\binom{i}{j}\left((k-j)2^{k-j-1}+2^{n-i-1}\right)+\underbrace{\sum_{i=0}^{m+1}\binom{m+1}{i}P(n-m-1,k-i)}_{(4.5)}\\
&=\sum_{i=0}^{m-1}\sum_{j=0}^{i}\binom{i}{j}\left((k-j)2^{k-j-1}+2^{n-i-1}\right)+\sum_{i=0}^{m}\binom{m}{i}\left((k-i)2^{k-i-1}+2^{n-m-1}\right)\\
&\quad+\sum_{i=0}^{m}\binom{m}{i}\left(P(n-m-1,k-i)+P(n-m-1,k-i-1)\right)\\
&\overset{(4.4)}{=}\sum_{i=0}^{m-1}\sum_{j=0}^{i}\binom{i}{j}\left((k-j)2^{k-j-1}+2^{n-i-1}\right)+\sum_{i=0}^{m}\binom{m}{i}P(n-m,k-i)\\
&=P(n,k,m)\overset{IH}{=}P(n,k)
\end{aligned}$$

$$\begin{aligned}
&\sum_{i=0}^{m+1}\binom{m+1}{i}P(n-m-1,k-i) \qquad (4.5)\\
&\quad=P(n-m-1,k)+P(n-m-1,k-m-1)+\sum_{i=1}^{m}\left(\binom{m}{i}+\binom{m}{i-1}\right)P(n-m-1,k-i)\\
&\quad=\left(P(n-m-1,k)+\sum_{i=1}^{m}\binom{m}{i}P(n-m-1,k-i)\right)\\
&\qquad+\left(P(n-m-1,k-m-1)+\sum_{i=0}^{m-1}\binom{m}{i}P(n-m-1,k-i-1)\right)\\
&\quad=\sum_{i=0}^{m}\binom{m}{i}\left(P(n-m-1,k-i)+P(n-m-1,k-i-1)\right)
\end{aligned}$$

□

**Lemma 4.10.** $P(n,k,m)\leq 2^{n-2}\left(\left(k-\frac{m}{2}\right)^2+k+\frac{7m}{4}+8\right)+2^k\left(\frac{3}{2}\right)^m\left(\frac{k}{2}-\frac{m}{6}\right)-2^k(k+1)-2^{n-k}\left(\frac{3}{2}\right)^m$.

*Proof.* The first inequality below is a consequence of Lemma 4.9 and 4.6. We also use the following known equations: $\sum_{k=0}^{n}\binom{n}{k}x^{k-1}k=n(1+x)^{n-1}$, $\sum_{k=0}^{n}\binom{n}{k}k^2=n(n+1)2^{n-2}$, $\sum_{k=0}^{n-1}x^{k-1}k=\frac{(1-x)(-nx^{n-1})+(1-x^n)}{(1-x)^2}$.

$$
\begin{aligned}
P(n,k,m) &\le \underbrace{\sum_{i=0}^{m-1}\sum_{j=0}^{i}\binom{i}{j}\left((k-j)2^{k-j-1}+2^{n-i-1}\right)}_{(4.6)}+\underbrace{\sum_{i=0}^{m}\binom{m}{i}B(n-m,k-i)}_{(4.9)}\\
&=\left(2^k\left(\frac{3}{2}\right)^m\left(k+1-\frac{m}{3}\right)-2^k(k+1)+m2^{n-1}\right)\\
&\quad+2^{n-2}\left(k^2-km+\frac{m(m-1)}{4}+k+8\right)\\
&\quad+2^k\left(\frac{3}{2}\right)^m\left(-\frac{k}{2}+\frac{m}{6}-1\right)-2^{n-k}\left(\frac{3}{2}\right)^m\\
&=2^{n-2}\left(\left(k-\frac{m}{2}\right)^2+k+\frac{7m}{4}+8\right)+2^k\left(\frac{3}{2}\right)^m\left(\frac{k}{2}-\frac{m}{6}\right)\\
&\quad-2^k(k+1)-2^{n-k}\left(\frac{3}{2}\right)^m
\end{aligned}
$$

$$
\begin{aligned}
&\sum_{i=0}^{m-1}\sum_{j=0}^{i}\binom{i}{j}\left((k-j)2^{k-j-1}+2^{n-i-1}\right) \\
&\qquad=\underbrace{\sum_{i=0}^{m-1}\sum_{j=0}^{i}\binom{i}{j}(k-j)2^{k-j-1}}_{(4.7)}+\underbrace{\sum_{i=0}^{m-1}\sum_{j=0}^{i}\binom{i}{j}2^{n-i-1}}_{(4.8)}\\
&\qquad=\left(2^k\left(\frac{3}{2}\right)^m(k+1-\frac{m}{3})-2^k(k+1)\right)+(m2^{n-1})
\end{aligned}
\tag{4.6}
$$

$$
\begin{aligned}
\sum_{i=0}^{m-1}\sum_{j=0}^{i}\binom{i}{j}(k-j)2^{k-j-1} &= k2^{k-1}\sum_{i=0}^{m-1}\sum_{j=0}^{i}\binom{i}{j}2^{-j}-2^{k-1}\sum_{i=0}^{m-1}\sum_{j=0}^{i}\binom{i}{j}2^{-j}j\\
&=k2^{k-1}\sum_{i=0}^{m-1}\left(\frac{3}{2}\right)^i-2^{k-1}\frac{1}{2}\sum_{i=0}^{m-1}\left(\frac{3}{2}\right)^{i-1}i\\
&=k2^{k-1}2\left(\left(\frac{3}{2}\right)^m-1\right)-2^{k-1}\left(2-\left(\frac{3}{2}\right)^{m-1}(3-m)\right)\\
&=2^k\left(\frac{3}{2}\right)^m\left(k+1-\frac{m}{3}\right)-2^k(k+1)
\end{aligned}
\tag{4.7}
$$

$$
\sum_{i=0}^{m-1}\sum_{j=0}^{i}\binom{i}{j}2^{n-i-1}=\sum_{i=0}^{m-1}\left(2^{n-i-1}\sum_{j=0}^{i}\binom{i}{j}\right)=\sum_{i=0}^{m-1}2^{n-i-1}2^i=m2^{n-1}
\tag{4.8}
$$

$$
\begin{aligned}
&\sum_{i=0}^{m}\binom{m}{i}(2^{n-m-2}(k-i)^2+2^{n-m-2}(k-i)-2^{k-i-1}(k-i)\\
&\qquad -2^{n-m-k+i}+2^{n-m+1}-2^{k-i}) \qquad (4.9)\\
&=\sum_{i=0}^{m}\binom{m}{i}2^{n-m-2}(k-i)^2+\sum_{i=0}^{m}\binom{m}{i}2^{n-m-2}(k-i)-\sum_{i=0}^{m}\binom{m}{i}2^{k-i-1}(k-i)\\
&\quad -\sum_{i=0}^{m}\binom{m}{i}2^{n-m-k+i}+\sum_{i=0}^{m}\binom{m}{i}2^{n-m+1}-\sum_{i=0}^{m}\binom{m}{i}2^{k-i}\\
&=2^{n-m-2}(k^2 2^m-km2^m+m(m+1)2^{m-2})+2^{n-m-2}(k2^m-m2^{m-1})\\
&\quad -2^{k-1}\left(k\left(\frac{3}{2}\right)^m-2^{-m}3^{m-1}m\right)-2^{n-m-k}3^m+2^{n+1}-2^k\left(\frac{3}{2}\right)^m\\
&=2^{n-2}\left(k^2-km+\frac{m(m-1)}{4}+k+8\right)\\
&\quad +2^k\left(\frac{3}{2}\right)^m\left(-\frac{k}{2}+\frac{m}{6}-1\right)-2^{n-k}\left(\frac{3}{2}\right)^m
\end{aligned}
$$

□

**Theorem 4.5.** *For* $m=\min(k,n-k)$, $P(n,k)\le 2^{n-2}\left(\left(k-\frac{m}{2}-\frac{7}{4}\right)^2+\frac{9k}{2}+\frac{79}{16}\right)+2^k\left(\frac{3}{2}\right)^m\left(\frac{k}{2}-\frac{m}{6}\right)-2^k(k+1)-2^{n-k}\left(\frac{3}{2}\right)^m$.

*Proof.* Directly from Lemmas 4.9 and 4.10. □

We now present the *size difference* $SD(n,k)$ between Pairwise Selection Network and our network. Merging step in $pw\_sel_K^N$ costs $2^k k-2^k+1$ and in $pw\_hbit\_sel_K^N$: $2^{k-1}k$, so the difference is given by the following equation:

$$
SD(n,k)=\begin{cases}0 & \text{if } n=k\\ 0 & \text{if } k=0\\ 2^{k-1}k-2^k+1+ & \\ +SD(n-1,k)+SD(n-1,k-1) & \text{if } 0<k<n\end{cases} \tag{4.10}
$$

**Theorem 4.6.** *Let* $S_{n,k}=\sum_{j=0}^{k}\binom{n-k+j}{j}2^{k-j}$. *Then:*

$$
SD(n,k)=\binom{n}{k}\frac{n+1}{2}-S_{n,k}\frac{n-2k+1}{2}-2^k(k-1)-1
$$

*Proof.* By straightforward calculation one can verify that $S_{n,0}=1$, $S_{n,n}=2^{n+1}-1, S_{n-1,k-1}=\frac{1}{2}(S_{n,k}-\binom{n}{k})$ and $S_{n-1,k-1}+S_{n-1,k}=S_{n,k}$. It follows that the theorem is true for $k=0$ and $k=n$. We prove the theorem by induction on pairs $(k,n)$. Take any $(k,n)$, $0<k<n$, and assume that theorem holds for every $(k',n')\prec(k,n)$ (in lexicographical order). Then we have:

$$
\begin{aligned}
SD(n,k) &= 2^{k-1}k - 2^k + 1 + SD(n-1,k) + SD(n-1,k-1) \\
&= 2^{k-1}k - 2^k + 1 + \binom{n-1}{k}\frac{n}{2} + \binom{n-1}{k-1}\frac{n}{2} - 2^k(k-1) - 1 \\
&\quad - 2^{k-1}(k-2) - 1 - (S_{n-1,k}\frac{n-2k}{2} + S_{n-1,k-1}\frac{n-2k+2}{2}) \\
&= \binom{n}{k}\frac{n}{2} - S_{n,k}\frac{n-2k}{2} - S_{n-1,k-1} - 2^k(k-1) - 1 \\
&= \binom{n}{k}\frac{n+1}{2} - S_{n,k}\frac{n-2k+1}{2} - 2^k(k-1) - 1
\end{aligned}
$$

□

**Corollary 4.2.** $|pw\_sel^N_{N/2}| - |pw\_hbit\_sel^N_{N/2}| = N\frac{\log N - 4}{2} + \log N + 2$*, for* $N = 2^n$*, where* $n \in \mathbb{N}$.

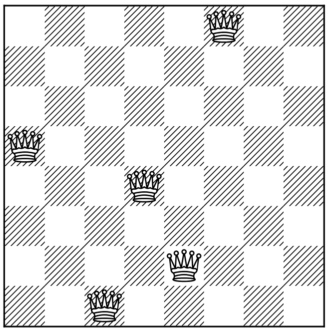
## 4.5 Summary

We have constructed a new family of selection networks, which are based on the pairwise selection ones, but require less comparators to merge subsequences. The difference in sizes grows with $k$ and is equal to $n\frac{\log n - 4}{2} + \log n + 2$ for $k = n/2$. Less comparators means less variables and clauses generated when translating cardinality constraints into CNFs, which is the goal for proposing new (smaller) networks.

The shortcoming of our new networks is that the size of an input sequence to the merging procedure is required to be a power of 2, since the construction uses modified version of bitonic merging networks [17] which require $k$ to be the power of 2. One could replace $k$ with the nearest power of 2 (by padding the input sequences), but it would have a negative impact on the encoding efficiency.

# Chapter 5

# Generalized Pairwise Selection Networks



Here we show a construction of the *m*-**Wise Selection Network** – the first generalized selection network of this thesis. As in the previous chapter, we use the pairwise approach. The idea is to split inputs into $m$ columns, perform a pre-processing on them (generalized form of a splitter), recursively select elements in the columns, and then return a top $k$ sorted sequence from the selected elements using a dedicated merging network. This network works for any values of $n$ and $k$, in contrast to the networks from the previous chapter. We show a complete construction for $m = 4$ and perform a theoretical comparison with the Pairwise Selection Network.

## 5.1 m-Wise Sequences

In the Pairwise Sorting Network [65], the first step is to sort pairs and split the input into two equally sized sequences $\bar{x}$ and $\bar{y}$, such that for any $i$, $x_i \geq y_i$ (weak domination). Then $\bar{x}$ and $\bar{y}$ are sorted independently and finally merged. Even after sorting, the property $x_i \geq y_i$ is maintained (see [65]). For example, the pair of sequences $\langle 110, 100 \rangle$ are pairwise sorted. We would like to extend this notion to cover larger number of sequences, suppose $m \geq 2$. For example, the tuple $\langle 111, 110, 100, 000 \rangle$ is 4-wise (sorted). Furthermore, the sequences we define may not be of equal size. It is because in the Pairwise Selection Network we merge two sorted sequences of size $\min(n/2, k)$ and $\min(n/2, k/2)$, and in our construction, the i-th sequence (to be merged) is of size at most $\lfloor k/i \rfloor$, where $1 \leq i \leq m$. But it can happen that for some $i$ we have $\lfloor n/m \rfloor \leq \lfloor k/i \rfloor$, i.e., there are at most as many elements in a sequence as the number of largest elements to be returned. Therefore we should consider the minimum of the two values. The variable $c$ in the following definition serves this purpose.

**Definition 5.1** (m-wise sequences)**.** Let $c, k, m \in \mathbb{N}$, $1 \leq k$ and $k/m \leq c$. Moreover, let $k_i = \min(c, \lfloor k/i \rfloor)$ and $\bar{x}^i \in X^{k_i}$, $1 \leq i \leq m$. The tuple $\langle \bar{x}^1, \dots, \bar{x}^m \rangle$ is $m$-wise of order $(c, k)$ if:

1. $\forall_{1 \leq i \leq m}\, \bar{x}^i$ is sorted,
2. $\forall_{1 \leq i \leq m-1}\, \forall_{1 \leq j \leq k_{i+1}}\, x^i_j \geq x^{i+1}_j$.

**Observation 5.1.** *Let a tuple $\langle \bar{x}^1, \dots, \bar{x}^m \rangle$ be m-wise of order $(c, k)$. Then:*

1. $|\bar{x}^1| \geq |\bar{x}^2| \geq \cdots \geq |\bar{x}^m| = \lfloor k/m \rfloor$,
2. $\sum_{i=1}^{m} |\bar{x}^i| \geq k$,

3. *if* $|\bar{x}^{i-1}| > |\bar{x}^i| + 1$ *then* $|\bar{x}^i| = \lfloor k/i \rfloor$.

*Proof.* The first statement is obvious. To prove the second one let $k_i = |\bar{x}^i|$, $1 \le i \le m$. Then, if $k_i = c$ for each $1 \le i \le m$, we have $\sum_{i=1}^{m} k_i = mc \ge mk/m = k$ and we are done. Let $1 \le i \le m$ be the first index such that $k_i \ne c$, therefore $k_i = \lfloor k/i \rfloor < c$. Thus, for each $1 \le j < i$: $k_j = c > k_i$, therefore $k_j \ge k_i + 1$. From this we get that $\sum_{j=1}^{m} k_j \ge \sum_{j=1}^{i} k_j \ge i\lfloor k/i \rfloor + i - 1 \ge k$.

The third one can be easily proved by contradiction. Assume that $k_i = c$, then from the first property $k_{i-1} \ge k_i = c$. By Definition 5.1, $k_{i-1} = \min(c, \lfloor k/(i-1) \rfloor) \le c$, so $k_{i-1} = c$, a contradiction. □

**Definition 5.2** (m-wise merger)**.** A comparator network $f^s_{(c,k)}$ is an *m-wise merger of order* $(c,k)$, if for each $m$-wise tuple $T = \langle \bar{x}^1, \dots, \bar{x}^m \rangle$ of order $(c,k)$, such that $s = \sum_{i=1}^{m} |\bar{x}^i|$, $f^s_{(c,k)}(T)$ is top $k$ sorted.

## 5.2 m-Wise Selection Network

Now we present the algorithm for constructing the $m$-Wise Selection Network (Algorithm 5.1). In Algorithm 5.1 we use $mw\_merge^s_{(c,k)}$, that is, an $m$-Wise Merger of order $(c,k)$, as a black box. We give detailed constructions of $m$-Wise Merger for $m = 4$ in the next section.

The idea is as follows: first, we split the input sequence into $m$ columns of non-increasing sizes (lines 2–5) and we sort rows using sorters (lines 6–8). Then we recursively run the selection algorithm on each column (lines 9–11), where at most $\lfloor k/i \rfloor$ items are selected from the $i$-th column. In obtained outputs, selected items are sorted and form prefixes of the columns. The prefixes are padded with zeroes (with $\perp$'s) in order to get the input sizes required by the $m$-wise property (Definition 5.1) and, finally, they are passed to the merging procedure (line 12–13).

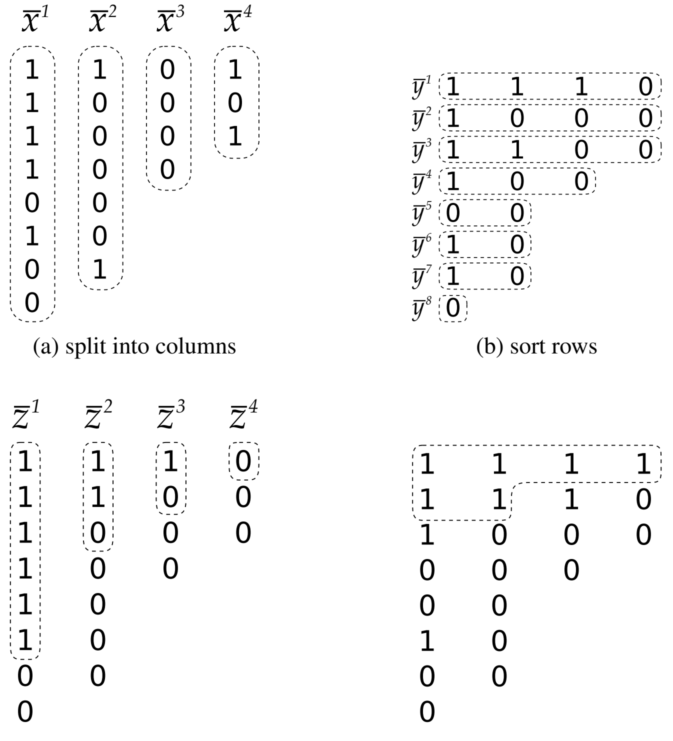
**Example 5.1.** In Figure 5.1 we present a sample run of Algorithm 5.1. The input is a sequence 1111010010000010000101, with parameters $\langle n_1, n_2, n_3, n_4, k \rangle = \langle 8, 7, 4, 3, 6 \rangle$. The first step (Figure 5.1a) is to arrange the input in columns (lines 2–5). In this example we get $\bar{x}^1 = \langle 11110100 \rangle$, $\bar{x}^2 = \langle 1000001 \rangle$, $\bar{x}^3 = \langle 0000 \rangle$ and $\bar{x}^4 = \langle 101 \rangle$. Next we sort rows using $1, 2, 3$ or 4-sorters (lines 6–8), the result is visible in Figure 5.1b. We make recursive calls in lines 9–11 of the algorithm. Items selected recursively in this step are marked in Figure 5.1c. Notice that in $i$-th column we only need to select $\lfloor k/i \rfloor$ largest elements. This is because of the initial sorting of the rows. Next comes the merging step, which is selecting $k$ largest elements from the results of the previous step. How exactly those elements are obtained and output depends on the implementation. We choose the convention that the resulting $k$ elements must be placed in the row-major order in our column representation of the input (see Figure 5.1d).

**Lemma 5.1.** *Let* $\bar{t}^i = \langle \bar{y}^1_i, \dots, \bar{y}^{n_i}_i \rangle$*, where* $\bar{y}^j$*,* $j = 1, \dots, n_1$*, is the result of Step 8 in Algorithm 5.1. For each* $1 \le i < m$*:* $|\bar{t}^i|_1 \ge |\bar{t}^{i+1}|_1$.

*Proof.* Take any $1 \le i < m$. Consider element $y^j_i$ (for some $1 \le j \le n_i$). Since $\bar{y}^j$ is sorted, we have $y^j_i \ge y^j_{i+1}$, therefore if $y^j_{i+1} = 1$ then $y^j_i = 1$. Thus $|\bar{t}^i|_1 \ge |\bar{t}^{i+1}|_1$. □

**Corollary 5.1.** *For each* $1 \le i \le m$*, let* $\bar{z}^i$ *be the result of Step 11 in Algorithm 5.1. Then for each* $1 \le i < m$*:* $|\bar{z}^i|_1 \ge |\bar{z}^{i+1}|_1$.

*Proof.* For $1 \le i \le m$, let $\bar{t}^i$ be the same as in Lemma 5.1. Take any $1 \le i < m$. Comparator network only permutes its input, therefore $|\bar{z}^i|_1 = |mw\_sel^{n_i}_{s_i}(\bar{t}^i)|_1 = |\bar{t}^i|_1 \ge |\bar{t}^{i+1}|_1 = |mw\_sel^{n_{i+1}}_{s_{i+1}}(\bar{t}^{i+1})|_1 = |\bar{z}^{i+1}|_1$. The inequality comes from Lemma 5.1. □



(a) split into columns (b) sort rows

(c) select recursively (d) merge columns

Figure 5.1: Sample run of the 4-Wise Selection Network

**Theorem 5.1.** *Let $n,k \in \mathbb{N}$, such that $k \leq n$. Then $mw\_sel^n_k$ is a k-selection network.*

*Proof.* We prove by induction that for each $n,k \in \mathbb{N}$ such that $1 \leq k \leq n$ and each $\bar{x} \in \{0,1\}^n$: $mw\_sel^n_k(\bar{x})$ is top $k$ sorted. If $1 = k \leq n$ then $mw\_sel^n_k = max^n$, so the theorem is true. For the induction step assume that $n \geq k \geq 2$, $m \geq 2$ and for each $(n^*,k^*) \prec (n,k)$ (in lexicographical order) the theorem holds. We have to prove the following two properties:

1. The tuple $\langle \mathrm{pref}(l_1,\bar{z}^1) :: \perp^{k_1-l_1}, \dots, \mathrm{pref}(l_m,\bar{z}^m) :: \perp^{k_m-l_m}\rangle$ is $m$-wise of order $(k_1,k)$.
2. The sequence $\bar{w} = \mathrm{pref}(l_1,\bar{z}^1) :: \cdots :: \mathrm{pref}(l_m,\bar{z}^m)$ contains $k$ largest elements from $\bar{x}$.

Ad. 1): Observe that for any $i$, $1 \leq i \leq m$, we have $n_i \leq n_1 < n$ and $l_i \leq l_1 \leq k$. Thus, $(n_i,l_i) \prec (n,k)$ and $\bar{z}_i$ is top $l_i$ sorted due to the induction hypothesis. Therefore, $\mathrm{pref}(l_i,\bar{z}^i)$ is sorted and so is $\mathrm{pref}(l_i,\bar{z}^i) :: \perp^{k_i-l_i}$. In this way, we prove that the first property of Definition 5.1 is satisfied. To prove the second one, fix $1 \leq j \leq l_{i+1}$ and assume that $z^{i+1}_j = 1$. We are going to show that $z^i_j = 1$. Since $\bar{z}^i$ and $\bar{z}^{i+1}$ are both top $l_{i+1}$ sorted (by the induction hypothesis) and since $|\bar{z}^i|_1 \geq |\bar{z}^{i+1}|_1$ (from Corollary 5.1), we have $|\mathrm{pref}(l_{i+1},\bar{z}^i)|_1 \geq |\mathrm{pref}(l_{i+1},\bar{z}^{i+1})|_1$, so $z^i_j = 1$.

Ad. 2): It is easy to observe that $\bar{z} = \bar{z}^1 :: \cdots :: \bar{z}^m$ is a permutation of the input sequence $\bar{x}$. If all 1's in $\bar{z}$ are in $\bar{w}$, we are done. So assume that there exists $z^i_j = 1$ for some $1 \leq i \leq m$, $l_i < j \leq n_i$. We show that $|\bar{w}|_1 \geq k$. From the induction hypothesis we get $\mathrm{pref}(l_i,z^i) \succeq \langle z^i_j \rangle$, which implies that $|\mathrm{pref}(l_i,z^i)|_1 = l_i$. From (1) and the second property of Definition 5.1 it is clear that all $z^s_t$ are 1's, where $1 \leq s \leq i$ and $1 \leq t \leq l_i$, therefore $|\mathrm{pref}(l_i,\bar{z}^1) :: \cdots :: \mathrm{pref}(l_i,\bar{z}^i)|_1 = i \cdot l_i$. Moreover, since $j > l_i$, we have $l_i = \lfloor k/i \rfloor$; otherwise we would have $j > n_i$. If $i = 1$, then $|\bar{w}|_1 \geq 1 \cdot l_1 = k$ and (2) holds.

Otherwise, $l_1 > l_i$, so from the definition of $l_i$, $l_1 \geq \cdots \geq l_i$, hence there exists $r \geq 1$ such that $\forall_{r<i' \leq i}\ l_r > l_{i'} = l_i$. Notice that since $|\mathrm{pref}(l_i,z^i)|_1 = l_i$ and $z^i_j = 1$ where $j > l_i$ we get $|z^i|_1 \geq l_i + 1$.

**Algorithm 5.1** $mw\_sel^n_k$

**Input:** $\bar{x} \in X^n$; $n_1, \ldots, n_m \in \mathbb{N}$ where $n > n_1 \geq \cdots \geq n_m$ and $\sum n_i = n$; $1 \leq k \leq n$
**Ensure:** The output is top $k$ sorted and is a permutation of the inputs

```
if k = 1 then return max^n(x̄)
offset = 1
for all i ∈ {1,...,m} do                                   # Splitting the input into columns.
    x̄^i ← ⟨x_offset, ..., x_{offset+n_i−1}⟩
    offset+ = n_i
for all i ∈ {1,...,n_1} do                                 # Sorting rows.
    m' = max{j : n_j ≥ i}
    ȳ^i ← sort^{m'}(⟨x_i^1, ..., x_i^{m'}⟩)
for all i ∈ {1,...,m} do                                   # Recursively selecting items in columns.
    k_i = min(n_1, ⌊k/i⌋);  l_i = min(n_i, ⌊k/i⌋)           # k_i ≥ l_i
    z̄^i ← mw_sel^{n_i}_{l_i}(ȳ_i^1, ..., ȳ_i^{n_i})
s = ∑_{i=1}^m k_i;  c = k_1;  out = suff(l_1+1, z̄^1) :: ··· :: suff(l_m+1, z̄^m)
res ← mw_merge^s_{(c,k)}(⟨pref(l_1, z̄^1) :: ⊥^{k_1−l_1}, ..., pref(l_m, z̄^m)⟩ :: ⊥^{k_m−l_m})
return drop(⊥, res) :: out
```

From Corollary 5.1 we have that for $1 \leq r' \leq r$: $|\bar{z}^{r'}|_1 \geq l_i + 1$. Using $l_{r'} \geq l_r > l_i$ and the induction hypothesis we get that each $\bar{z}^{r'}$ is top $l_i + 1$ sorted, therefore $|\mathrm{pref}(l_i + 1, \bar{z}^{r'})|_1 = l_i + 1$. We finally have that $|\bar{w}|_1 \geq |\mathrm{pref}(l_i + 1, \bar{z}^1) :: \cdots :: \mathrm{pref}(l_i + 1, \bar{z}^r) :: \mathrm{pref}(l_i, \bar{z}^{r+1}) :: \cdots :: \mathrm{pref}(l_i, \bar{z}^i)|_1 = r(l_i + 1) + (i - r)l_i = r(l_{r+1} + 1) + (i - r)l_{r+1} = i \cdot l_{r+1} + r \geq (r+1)\lfloor k/(r+1) \rfloor + r \geq k$. In the second to last inequality, we use the facts: $l_{r+1} = l_i = \lfloor k/i \rfloor < n_i \leq n_{r+1}$ from which $l_{r+1} = \lfloor k/(r+1) \rfloor$ follows.

From the statements (1) and (2) we can conclude that $mw\_merge^s_{(n,k)}$ returns the $k$ largest elements from $\bar{x}$, which completes the proof. □

## 5.3 4-Wise Merging Network

In this section the merging algorithm for four columns is presented. The input to the merging procedure is $R = \langle \mathrm{pref}(n_1, \bar{y}^1), \ldots, \mathrm{pref}(n_m, \bar{y}^m) \rangle$, where each $\bar{y}^i$ is the output of the recursive call in Algorithm 5.1. The main observation is the following: since $R$ is $m$-wise, if you take each sequence $\mathrm{pref}(n_i, \bar{y}^i)$ and place them side by side, in columns, from left to right, then the sequences are sorted in rows and columns. The goal of the networks is to put the $k$ largest elements in top rows. It is done by sorting slope lines with decreasing slope rate, in lines 1–14 (similar idea can be found in [79]). The algorithm is presented in Algorithm 5.2. The pseudo-code looks non-trivial, but it is because we need a separate sub-case every time we need to use either $sort^2$, $sort^3$ or $sort^4$ operation, and this depends on the sizes of columns and the current slope.

After slope-sorting phase some elements might not be in the desired row-major order, therefore the correction phase is needed, which is the goal of the sorting operations in lines 15–18. Figure 5.2 shows the order relations of elements after $i = \lceil \log n_1 \rceil$ iterations of the **while** loop (disregarding the upper index $i$, for clarity). Observe that the order should be possibly corrected between $z_{j-1}$ and $w_{j+1}$ and then the 4-tuples $\langle x_j, w_j, z_{j-1}, y_{j-1} \rangle$ and $\langle x_{j+1}, w_{j+1}, z_j, y_j \rangle$ should be sorted to get the row-major order. Lines 17–18 addresses certain corner cases of the correction phase.

An input to Algorithm 5.2 must be 4-wise of order $(c, k)$ and the output should be top $k$ sorted. Using the 0-1 principle, we can assume that sequences (in particular, inputs) are binary. Thus, the algorithm gets as input four sorted 0-1 columns with the additional property that the numbers of 1's in successive columns do not increase. Nevertheless, the differences between them can be quite big. The goal of each iteration of the main loop in Algorithm 5.2 is to decrease the

$$\begin{tikzcd}
 & & y_{j-1} \arrow[r] \arrow[dl] & z_{j-1} \arrow[dl] \\
w_j \arrow[r] & x_j \arrow[r] \arrow[dl] & y_j \arrow[r] \arrow[dl] & z_j \\
w_{j+1} \arrow[r] & x_{j+1} & &
\end{tikzcd}$$

Figure 5.2: The order relation among elements of neighboring rows. Arrows shows non-increasing order. Relations that follow from transitivity are not shown.

**Algorithm 5.2** $4w\_merge^s_{(c,k)}$

**Input:** 4-wise tuple $\langle \bar{w},\bar{x},\bar{y},\bar{z}\rangle$ of order $(c,k)$, where $s = |\bar{w}|+|\bar{x}|+|\bar{y}|+|\bar{z}|$ and $c = |\bar{w}|$.
**Ensure:** The output is top $k$ sorted and is a permutation of the inputs

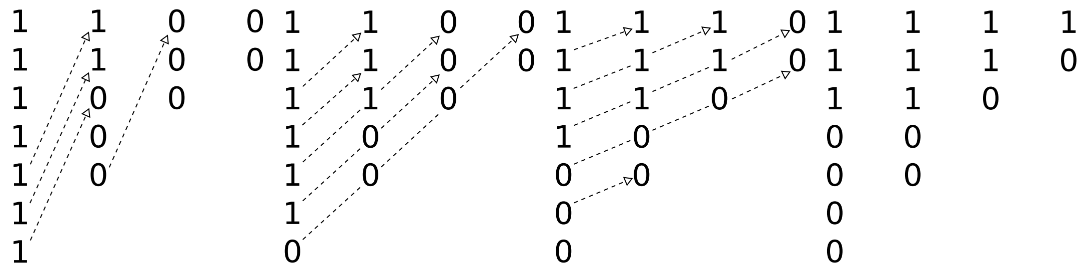
$k_1 = |\bar{w}|$; $k_2 = |\bar{x}|$; $k_3 = |\bar{y}|$; $k_4 = |\bar{z}|$ # $k_4 = \lfloor k/4 \rfloor$, see def. of $m$-wise tuple.
$d \leftarrow \min\{l \in \mathbb{N} \,|\, 2^l \geq k_1\}$
$h \leftarrow 2^d$; $i \leftarrow 0$; $\langle \bar{w}^0,\bar{x}^0,\bar{y}^0,\bar{z}^0\rangle = \langle \bar{w},\bar{x},\bar{y},\bar{z}\rangle$; $h_0 \leftarrow h$
**while** $h_i > 1$ **do** # Define the $(i+1)$-th stage of $4w\_merge^s_{(n,k)}$.
  $i \leftarrow i+1$; $h_i \leftarrow h_{i-1}/2$
  $\langle \bar{w}^i,\bar{x}^i,\bar{y}^i,\bar{z}^i\rangle = \langle \bar{w}^{i-1},\bar{x}^{i-1},\bar{y}^{i-1},\bar{z}^{i-1}\rangle$
  **for all** $j \in \{1,\dots,\min(k_3-h_i,k_4)\}$ **do**
    **if** $j+3h_i \leq k_1$ **and** $j+2h_i \leq k_2$ **then** $sort^4(z^i_j, y^i_{j+h_i}, x^i_{j+2h_i}, w^i_{j+3h_i})$
    **else if** $j+2h_i \leq k_2$ **then** $sort^3(z^i_j, y^i_{j+h_i}, x^i_{j+2h_i})$ # $w^i_{j+3h_i}$ is not defined.
    **else** $sort^2(z^i_j, y^i_{j+h_i})$ # Both $x^i_{j+2h_i}$ and $w^i_{j+3h_i}$ are not defined.
  **for all** $j \in \{1,\dots,\min(k_2-h_i,k_3,h_i)\}$ **do**
    **if** $j+2h_i \leq k_1$ **then** $sort^3(y^i_j, x^i_{j+h_i}, w^i_{j+2h_i})$
    **else** $sort^2(y^i_j, x^i_{j+h_i})$ # $w^i_{j+2h_i}$ is not defined.
  **for all** $j \in \{1,\dots,\min(k_1-h_i,k_2,h_i)\}$ **do** $sort^2(x^i_j, w^i_{j+h_i})$
# Define two more stages to correct local disorders.
**for all** $j \in \{1,\dots,\min(k_1-2,k_4)\}$ **do** $sort^2(z^i_j, w^i_{j+2})$
**for all** $j \in \{1,\dots,\min(k_2-1,k_4)\}$ **do** $sort^4(y^i_j, z^i_j, w^i_{j+1}, x^i_{j+1})$
**if** $k_1 > k_4$ **and** $k_2 = k_4$ **then** $sort^3(y^i_{k_4}, z^i_{k_4}, w^i_{k_4+1})$ # $x^i_{k_4+1}$ is not defined.
**if** $k \bmod 4 = 3$ **and** $k_1 > k_3$ **then** $sort^2(y^i_{k_4+1}, w^i_{k_4+2})$ # $y^i_{k_4+1}$ must be corrected.
**return** $\mathrm{zip}(\bar{w}^i,\bar{x}^i,\bar{y}^i,\bar{z}^i)$ # Returns the columns in row-major order.

maximal possible difference by the factor of two. Therefore, after the main loop, the differences are bounded by one.

**Example 5.2.** A sample run of slope sorting phase of Algorithm 5.2 is presented in Figure 5.3. The arrows represent the sorting order. Notice how the 1's are being *pushed* towards upper-right side. In the end, the differences between the number of 1's in consecutive columns are bounded by one.

**Observation 5.2.** *Let $k_1,\dots,k_4$ be as defined in Algorithm 5.2. For each i, $0 \leq i \leq \lceil \log(k_1) \rceil$, let $c_i = \min(k_3, \lfloor k/4 \rfloor + h_i)$, $b_i = \min(k_2, c_i + h_i)$ and $a_i = \min(k_1, b_i + h_i)$, where $h_i$ is as defined in Algorithm 5.2. Then we have: $k_1 \geq a_i \geq b_i \geq c_i \geq \lfloor k/4 \rfloor$ and the inequalities: $a_i - b_i \leq h_i$, $b_i - c_i \leq h_i$ and $c_i - \lfloor k/4 \rfloor \leq h_i$ are true.*

*Proof.* From Observation 5.1.(1) we have $k_1 \geq k_2 \geq k_3 \geq \lfloor k/4 \rfloor$. It follows that $c_i \geq \lfloor k/4 \rfloor$ and $b_i \geq \min(k_3, \lfloor k/4 \rfloor + h_i) = c_i$ and $k_1 \geq a_i \geq \min(k_2, c_i + h_i) = b_i$. Moreover, one can see that $c_i - \lfloor k/4 \rfloor = \min(k_3 - \lfloor k/4 \rfloor, h_i) \leq h_i$, $b_i - c_i = \min(k_2 - c_i, h_i) \leq h_i$ and $a_i - b_i = \min(k_1 - b_i, h_i) \leq h_i$, so we are done. $\square$



(a) (b) (c) (d)

Figure 5.3: Sample run of the slope-sorting phase of the 4-Wise Merging Network

**Lemma 5.2.** *Let $a_i, b_i, c_i$, $0 \le i \le \lceil \log(k_1) \rceil$, be as defined in Observation 5.2. Let us define $\breve{w}^i = pref(a_i, \bar{w}^i)$, $\breve{x}^i = pref(b_i, \bar{z}^i)$, $\breve{y}^i = pref(c_i, \bar{y}^i)$ and $\breve{z}^i = \bar{z}^i$. Then only the items in $\breve{w}^i$, $\breve{x}^i$, $\breve{y}^i$ and $\breve{z}^i$ take part in the i-th iteration of sorting operations in lines 8-14 of Algorithm 5.2.*

*Proof.* An element of the vector $\bar{z}^i$, that is, $z^i_j$ is sorted in one of the lines 8–10 of Algorithm 5.2, thus for all such $j$ that $1 \le j \le k_4 = \lfloor k/4 \rfloor$. An element $y^i_j$ of $\bar{y}^i$ is sorted in lines 8–10 of the first inner loop and in lines 12–13 of the second inner loop. In the first three of those lines we have the bounds on $j$: $1 + h_i \le j \le \min(k_3, k_4 + h_i)$ and in the last three lines - the bounds: $1 \le j \le \min(k_3, h_i)$. The sum of these two intervals is $1 \le j \le min(k_3, \lfloor k/4 \rfloor + h_i) = c_i$. Similarly, we can analyze the operations on $x^i_j$ in lines 8–10, 12–13 and 14. The range of $j$ is the sum of the following disjoint three intervals: $[1+2h_i, \min(k_2, k_3+h_i, k_4+2h_i)]$, $[1+h_i, \min(k_3+h_i, 2h_i)]$ and $[1, \min(k_2, h_i)]$ that give us the interval $[1, \min(k_2, k_3+h_1, k_4+2h_i)] = [1, b_i]$ as their sum. The analysis of the range of $j$ used in the operations on $w^i_j$ in Algorithm 5.2 can be done in the same way. □

**Lemma 5.3.** *Let $\breve{w}^i$, $\breve{x}^i$, $\breve{y}^i$ and $\breve{z}^i$ be as defined in Lemma 5.2. Then for all i, $0 \le i \le \lceil \log_2 k_1 \rceil$, after the i-th iteration of the while loop in Algorithm 5.2 the sequences $\breve{w}^i, \breve{x}^i, \breve{y}^i, \breve{z}^i$ are of the form $1^{p_i}0^*$, $1^{q_i}0^*$, $1^{r_i}0^*$ and $1^{s_i}0^*$, respectively, and:*

$$p_i \ge q_i \ge r_i \ge s_i, \tag{5.1}$$

$$p_i - q_i \le h_i \text{ and } q_i - r_i \le h_i \text{ and } r_i - s_i \le h_i, \tag{5.2}$$

$$p_i + q_i + r_i + s_i \ge \min(k, p_0 + q_0 + r_0 + s_0). \tag{5.3}$$

*Proof.* By induction. At the beginning we have $h_0 \ge k_1 \ge k_2 \ge k3 \ge k4 = \lfloor k/4 \rfloor$ and therefore $\breve{w}^0 = \bar{w}$, $\breve{x}^0 = \bar{x}$, $\breve{y}^0 = \bar{y}$ and $\breve{z}^0 = \bar{z}$). All four sequences $\bar{w}, \bar{x}, \bar{y}, \bar{z}$ are sorted (by Definition 5.1), thus, they are of the form $1^{p_0}0^*$, $1^{q_0}0^*$, $1^{r_0}0^*$, and $1^{s_0}0^*$ respectively. By Definition 5.1.(2), for each pair of sequences: $(\bar{w}_{1..k_2}, \bar{x})$, $(\bar{x}_{1..k_3}, \bar{y})$ and $(\bar{y}_{1..k_4}, \bar{z})$, if there is a 1 on the $j$ position in the right sequence, it must be a corresponding 1 on the same position in the left one. Therefore, we have: $p_0 \ge q_0 \ge r_0 \ge s_0 \ge 0$. Moreover, $p_0 - q_0 \le p_0 \le k_1 \le h_0$, $q_0 - r_0 \le q_0 \le k_2 \le h_0$ and $r_0 - s_0 \le r_0 \le k_3 \le h_0$. Finally, $p_0 + q_0 + r_0 + s_0 \ge \min(k, p_0 + q_0 + r_0 + s_0)$, thus the lemma holds for $i = 0$.

In the inductive step $i > 0$ observe that the elements of $\breve{w}^i$, $\breve{x}^i$, $\breve{y}^i$ and $\breve{z}^i$ are defined by the sort operations over the elements with the same indices from vectors $\breve{w}^{i-1}$, $\breve{x}^{i-1}$, $\breve{y}^{i-1}$ and $\breve{z}^{i-1}$. This means that the values of $w^{i-1}_{a_i+1,\dots,a_{i+1}}$, $x^{i-1}_{b_i+1,\dots,b_{i+1}}$ and $y^{i-1}_{c_i+1,\dots,c_{i+1}}$ are not used in the $i$-th iteration. Therefore, the numbers of 1's in columns that are sorted in the $i$-th iteration are defined by values: $p'_{i-1} = \min(a_i, p_{i-1})$, $q'_{i-1} = \min(b_i, q_{i-1})$, $q'_{i-1} = \min(c_i, r_{i-1})$ and $s'_{i-1} = s_{i-1}$. In the following we prove that the numbers with primes have the same properties as those without them.

$$p'_{i-1} \geq q'_{i-1} \geq r'_{i-1} \geq s'_{i-1}, \tag{5.4}$$

$$p'_{i-1} - q'_{i-1} \leq h_{i-1} \text{ and } q'_{i-1} - r'_{i-1} \leq h_{i-1} \text{ and } r'_{i-1} - s'_{i-1} \leq h_{i-1}, \tag{5.5}$$

$$p'_{i-1} + q'_{i-1} + r'_{i-1} + s'_{i-1} \geq \min(k, p_{i-1} + q_{i-1} + r_{i-1} + s_{i-1}). \tag{5.6}$$

The proofs of inequalities in 5.4 are quite direct and follow from the monotonicity of min. For example, we can observe that $p'_{i-1} = \min(a_i, p_{i-1}) \geq \min(b_i, q_{i-1}) = q'_{i-1}$, since we have $a_i \geq b_i$ and $p_{i-1} \geq q_{i-1}$, by Observation 5.2 and the induction hypothesis. The others can be shown in the same way.

Let us now prove one of the inequalities of Eq. (5.5), say, the second one. We have $q'_{i-1} - r'_{i-1} = \min(b_i, q_{i-1}) - \min(c_i, r_{i-1}) \leq \min(c_i + h_i, r_{i-1} + h_{i-1}) - \min(c_i, r_{i-1}) \leq h_{i-1}$, by Observation 5.2, the fact that $h_{i-1} = 2h_i$ and the induction hypothesis. The proofs of the others are similar.

The proof of Eq. (5.6) is only needed if at least one of the following inequalities are true: $p'_{i-1} < p_{i-1}$, $q'_{i-1} < q_{i-1}$ or $r'_{i-1} < r_{i-1}$. Obviously, the inequalities are equivalent to $a_i < p_{i-1}$, $b_i < q_{i-1}$ and $c_i < r_{i-1}$, respectively. Therefore, to prove 5.6 we consider now three separate cases: (1) $a_i \geq p_{i-1}$ and $b_i \geq q_{i-1}$ and $c_i < r_{i-1}$, (2) $a_i \geq p_{i-1}$ and $b_i < q_{i-1}$ and (3) $a_i < p_{i-1}$.

In the case (1) we have $p'_{i-1} = p_{i-1}$, $q'_{i-1} = q_{i-1}$ and $r'_{i-1} = c_i = \min(k_3, h_i + \lfloor k/4 \rfloor) < r_{i-1} \leq q_{i-1} \leq p_{i-1}$. It follows that $p'_{i-1} + q'_{i-1} + r'_{i-1} = p_{i-1} + q_{i-1} + c_i \geq c_i + 1 + c_i + 1 + c_i = 3c_i + 2$. Since $k_3 \geq r_{i-1} \geq c_i + 1 = \min(k_3, \lfloor k/4 \rfloor + h_i) + 1$, we can observe that $c_i$ must be equal to $\lfloor k/4 \rfloor + h_i$. In addition, by the induction hypothesis we have $s_{i-1} \geq r_{i-1} - h_{i-1} \geq c_i + 1 - 2h_i$. Merging those facts we can conclude that $p'_{i-1} + q'_{i-1} + r'_{i-1} + s'_{i-1} \geq 4c_i + 3 - 2h_i = 4\lfloor k/4 \rfloor + 4h_i + 3 - 2h_i \geq k$, so we are done in this case.

In the case (2) we have $p'_{i-1} = p_{i-1}$ and $q'_{i-1} = b_i = \min(k_2, c_i + h_i) < q_{i-1} \leq p_{i-1}$. It follows that $p'_{i-1} + q'_{i-1} \geq 2b_i + 1$. Since $k_2 \geq q_{i-1} \geq b_i + 1 = \min(k_2, c_i + h_i) + 1$, we can observe that $b_i$ must be equal to $c_i + h_i$. In addition, by the induction hypothesis and Observation 5.2, we can bound $r'_{i-1}$ as $r'_{i-1} = \min(c_i, r_{i-1}) \geq \min(b_i - h_i, q_{i-1} - h_{i-1}) \geq b_i + 1 - 2h_i$. Therefore, $p'_{i-1} + q'_{i-1} + r'_{i-1} \geq 3b_i + 2 - 2h_i = 3c_i + 2 + h_i$. Since $c_i$ is defined as $\min(k_3, \lfloor k/4 \rfloor + h_i)$, we have to consider two sub-cases of the possible value of $c_i$. If $c_i = k_3 \leq \lfloor k/4 \rfloor + h_i$, then we have $k_2 \geq q_{i-1} \geq b_i + 1 = c_i + h_i + 1 \geq k_3 + 2$. By Observation 5.1.(3), $k_3$ must be equal to $\lfloor k/3 \rfloor$, thus $3c_i + 2 = 3k_3 + 2 \geq k$, and we are done. Otherwise, we have $c_i = \lfloor k/4 \rfloor + h_i$ and since $s'_{i-1} = s_{i-1} \geq r_{i-1} - h_{i-1} \geq b_i + 1 - 4h_i = c_i + 1 - 3h_i$, we can conclude that $p'_{i-1} + q'_{i-1} + r'_{i-1} + s'_{i-1} \geq 4c_i + 3 - 2h_i = 4\lfloor k/4 \rfloor + 3 + 2h_i \geq k$.

The last case $a_i < p_{i-1}$ can be proved be the similar arguments. Having (5.4.5.5,5.6), we can start proving the inequalities from the lemma. Observe that in Algorithm 5.2 the values of vectors $\bar{w}^i$, $\bar{x}^i$, $\bar{y}^i$ and $\bar{z}^i$ are defined with the help of three types of sorters: $sort^4$, $sort^3$ and $sort^2$. The smaller sorters are used, when the corresponding index is out of the range and an input item is not available. In the following analysis we would like to deal only with $sort^4$ and in the case of smaller sorters we extend artificially their inputs and outputs with 1's at the left end and 0's at the right end. For example, in line 18 we have $\langle y^i_j, x^i_{j+h_i} \rangle \leftarrow sort^2(y^{i-1}_j, x^{i-1}_{j+h_i})$ so we can analyze this operation as $\langle 1, y^i_j, x^i_{j+h_i}, 0 \rangle \leftarrow sort^4(1, y^{i-1}_j, x^{i-1}_{j+h_i}, 0)$. The 0 input corresponds to the element of $\bar{w}^{i-1}$ with index $j + 2h_i$, where $j + 2h_i > k_1 \geq a_i$, and the 1 input corresponds to the element of $\bar{z}^{i-1}$ with index $j - h_i$, where $j - h_i < 0$. A similar situation is in lines 10, 12, 16 and 22, where $sort^2$ and $sort^3$ are used. Therefore, in the following we assume that elements of input sequences $\bar{w}^{i-1}$, $\bar{x}^{i-1}$, $\bar{y}^{i-1}$ and $\bar{z}^{i-1}$ with negative indices are equal to 1 and elements of the inputs with indices above $a_i$, $b_i$, $c_i$ and $\lfloor k/4 \rfloor$, respectively, are equal to 0. This assumption does not break the monotonicity of the sequences and we also have the property that $w^{i-1}_j = 1$ if and only if $j \leq p'_{j-1}$ (and similar ones for $\bar{x}^{i-1}$ and $q'_{i-1}$, and so on).

It should be clear now that, under the assumption above, we have:

$$
\begin{aligned}
w^i_j &= \min(w^{i-1}_j, x^{i-1}_{j-h_i}, y^{i-1}_{j-2h_i}, z^{i-1}_{j-3h_i}) && \text{for } 1 \le j \le a_i, && (5.7a)\\
x^i_j &= \mathrm{2nd}(w^{i-1}_{j+h_i}, x^{i-1}_j, y^{i-1}_{j-h_i}, z^{i-1}_{j-2h_i}) && \text{for } 1 \le j \le b_i, && (5.7b)\\
y^i_j &= \mathrm{3rd}(w^{i-1}_{j+2h_i}, x^{i-1}_{j+h_i}, y^{i-1}_j, z^{i-1}_{j-h_i}) && \text{for } 1 \le j \le c_i, && (5.7c)\\
z^i_j &= \max(w^{i-1}_{j+3h_i}, x^{i-1}_{j+2h_i}, y^{i-1}_{j+h_i}, z^{i-1}_j) && \text{for } 1 \le j \le \lfloor k/4 \rfloor, && (5.7d)
\end{aligned}
$$

where 2nd and 3rd denote the second and the third smallest element of its input, respectively. Since the functions min, 2nd, 3rd and max are monotone and the input sequences are monotone. we can conclude that $w^i_j \ge w^i_{j+1}$, $x^i_j \ge x^i_{j+1}$, $y^i_j \ge y^i_{j+1}$ and $z^i_j \ge z^i_{j+1}$. Let $p_i$, $q_i$, $r_i$ and $s_i$ denote the numbers of 1's in them. Clearly, we have $p_i + q_i + r_i + s_i = p'_{i-1} + q'_{i-1} + r'_{i-1} + s'_{i-1}$, thus $p_i + q_i + r_i + s_i \ge \min(k, p_{i-1} + q_{i-1} + r_{i-1} + s_{i-1}) \ge \min(k, p_0 + q_0 + r_0 + s_0)$, by the induction hypothesis. Thus. we have proved monotonicity of $\bar{w}^i$, $\bar{x}^i$, $\bar{y}^i$ and $\bar{z}^i$ and that Eq. (5.3) holds.

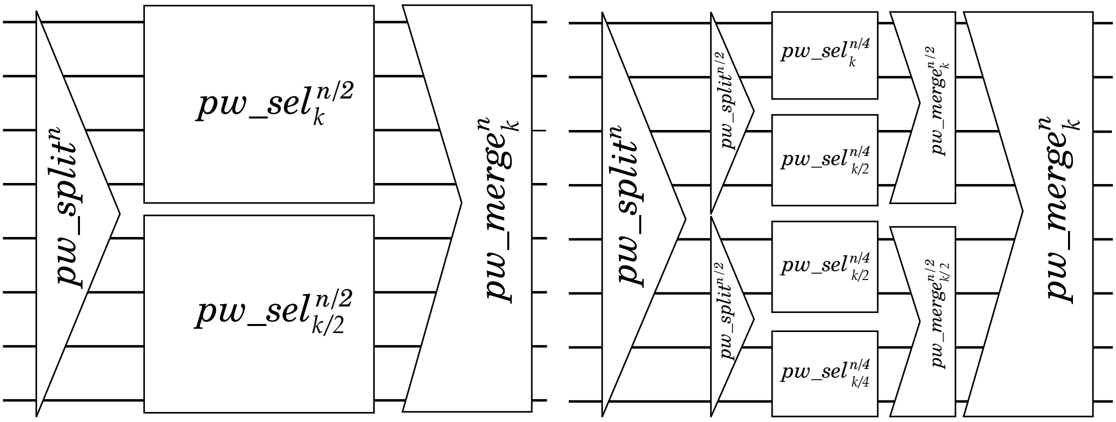
To prove Eq. (5.2) we show that $p_i - h_i \le q_i$, $q_i - h_i \le r_i$, $r_i - h_i \le s_i$, that is, that the following equalities are true: $x^i_{p_i-h_i} = 1$, $y^i_{q_i-h_i} = 1$ and $z^i_{r_i-h_i} = 1$. The first equality follows from the fact that $w^i_{p_i} = 1$ and $w^i_{p_i} \le x^i_{p_i-h_i}$, because they are output in this order by a single sort operation. The other two equalities can be shown by the similar arguments.

The last equation we have to prove is Eq. (5.1). By the induction hypothesis and our assumption we have $w^{i-1}_j \ge x^{i-1}_j \ge y^{i-1}_j \ge z^{i-1}_j$, for any $j$. We know also that the vectors are non-increasing. We use these facts to show that $w^i_{q_i} = 1$, which is equivalent to $p_i \ge q_i$. Since $w^i_{q_i} = \min(w^{i-1}_{q_i}, x^{i-1}_{q_i-h_i}, y^{i-1}_{q_i-2h_i}, z^{i-1}_{q_i-3h_i})$, we have to prove that all the arguments of the min function are 1's. From the definition of $q_i$ we have $1 = x^i_{q_i} = \mathrm{2nd}(w^{i-1}_{q_i+h_i}, x^{i-1}_{q_i}, y^{i-1}_{q_i-h_i}, z^{i-1}_{q_i-2h_i})$, thus the maximum of any pair of arguments of 2nd must be 1. Now we can see that $w^{i-1}_{q_i} \ge \max(w^{i-1}_{q_i+h_i}, x^{i-1}_{q_i}) \ge 1$, $x^{i-1}_{q_i-h_i} \ge \max(x^{i-1}_{q_i}, y^{i-1}_{q_i-h_i}) \ge 1$, $y^{i-1}_{q_i-2h_i} \ge \max(y^{i-1}_{q_i-h_i}, z^{i-1}_{q_i-2h_i}) \ge 1$ and $z^{i-1}_{q_i-3h_i} \ge \max(y^{i-1}_{q_i-h_i}, z^{i-1}_{q_i-2h_i}) \ge 1$. In the last inequality we use the fact that for any $j$ it is true that $z^{i-1}_{j-2h_i} \ge y^{i-1}_j$ (because $r'_{i-1} - 2h_i \le s'_{i-1}$, by Eq. (5.5)). Thus, $w^i_{q_i}$ must be 1 and we are done. The other two inequalities can be proved in the similar way with the help of two additional relations: $x^{i-1}_{j-2h_i} \ge w^{i-1}_j$ and $y^{i-1}_{j-2h_i} \ge x^{i-1}_j$, which follows from the induction hypothesis. □

After $i = \lceil \log k_1 \rceil$ iterations of the main loop in Algorithm 5.2 we have $h_i = 1$ and, by Lemma 5.3, elements in vectors (columns) $\check{w}^i$, $\check{x}^i$, $\check{y}^i$ and $\check{z}^i$ are in non-increasing order. In the following part of this subsection the value of $i$ is fixed and we do not write it as the upper index. By Eq. (5.2) and Eq. (5.1) of the lemma, we have also the same order in diagonal lines: $w_j \le x_{j-1} \le y_{j-2} \le z_{j-3}$ and in rows: $w_j \ge x_j \ge y_j \ge z_j$. From Eq. (5.3) it follows that the vectors contains the $k$ largest elements of the input sequences: $\check{w}^0$, $\check{x}^0$, $\check{y}^0$ and $\check{z}^0$. The goal of the lines 26–30 in Algorithm 5.2 is to correct the order in the vectors in such a way that the $k$ largest elements appear at the beginning in the row-major order. Figure 5.2 shows the mentioned-above order relations. Observe that the order should be possible corrected between $z_{j-1}$ and $w_{j+1}$ and then the 4-tuples $\langle x_j, w_j, z_{j-1}, y_{j-1} \rangle$ and $\langle x_{j+1}, w_{j+1}, z_j, y_j \rangle$ should be sorted to get the row-major order.

**Theorem 5.2.** *The output of Algorithm 5.2 is top k sorted.*

*Proof.* The zip operation outputs its input vectors in the row-major order. By Eq. (5.3) of Lemma 5.3, we know that elements in the *out* sequence are dominated by the $k$ largest elements in the output vectors of the main loop. From the order diagram given in Fig. 5.2 it follows that $\langle y_{i-1}, \max(z_{i-1}, w_{j+1}), w_j, x_j \rangle$ dominates $\langle y_i, z_i, \min(z_{i-1}, w_{j+1}), x_{i+1} \rangle$, thus, after the sorting operations in lines 15–16, the values appear in the row-major order. The two special cases, where the whole sequence of four elements is not available for the $sort^4$ operation, are covered by lines




(a) one step (b) two steps

Figure 5.4: The Pairwise Selection Network

17 and 18. If the vector $\check{x}^i$ does not have the element with index $k_4+1$, then just 3 elements are sorted in line 17. If $k$ mod $4 = 3$ then the element $y_{k_4+1}$ is the last one in desired order, but the vector $\check{z}^i$ does not have the element with index $k_4+1$, so the network must sort just 2 elements in line 18. Note that in the first $\lfloor k/4 \rfloor$ rows we have $4\lfloor k/4 \rfloor$ elements of the top $k$ ones, so the values of $k - 4\lfloor k/4 \rfloor$ leftmost elements in the row $k_4+1$ should be corrected. □

## 5.4 Comparison of Pairwise Selection Networks

The number of comparators in the Pairwise Selection Network (Algorithm 4.1) can be defined using this recursive formula:

$$|pw\_sel_k^n| = \begin{cases} |pw\_sel_k^{n/2}| + |pw\_sel_{k/2}^{n/2}| + \\ +|pw\_split^n| + |pw\_merge_k^n| & \text{if } k < n \\ |oe\_sort^k| & \text{if } k = n \\ |max^n| & \text{if } k = 1 \end{cases} \tag{5.8}$$

We denote the splitting step as a network $pw\_split^n$. One can check that it requires $|pw\_split^n|$ $= n/2$ comparators and the merging step requires $|pw\_merge_k^n| = k\log k - k + 1$ comparators [78]. In the formula above we assume $n$ and $k$ to be powers of 2. This way it is always true that $\min(n/2, k) = n/2$, if $k < n$, thus simplifying the calculations.

The schema of this network is presented in Figure 5.4. We want to count the number of variables and clauses used when merging 4 outputs of the recursive steps, therefore we expand the recursion by one level (see Figure 5.4b).

**Lemma 5.4.** *Let $k,n \in \mathbb{N}$ be powers of 2, and $k < n$. Then $V(pw\_merge_k^{n/2}) + V(pw\_merge_{k/2}^{n/2}) + V(pw\_merge_k^n) = 5k\log k - 6k + 6$ and $C(pw\_merge_k^{n/2}) + C(pw\_merge_{k/2}^{n/2}) + C(pw\_merge_k^n) = \frac{15}{2}k\log k - 9k + 9$.*

*Proof.* The number of 2-comparator used is:

$$|pw\_merge_k^{n/2}| + |pw\_merge_{k/2}^{n/2}| + |pw\_merge_k^n| = \frac{5}{2}k\log k - 3k + 3$$

Elementary calculation gives the desired result. □

| $i$ | $h_i$ | #2-comparators | #3-comparators | #4-comparators |
|---|---|---|---|---|
| 1 | $\frac{k}{2}$ | $\frac{k}{2}$ | 0 | 0 |
| 2 | $\frac{k}{4}$ | $\frac{k}{4}+\frac{k}{12}$ | $\frac{k}{4}$ | 0 |
| 3 | $\frac{k}{8}$ | $\frac{k}{8}$ | $\frac{k}{8}$ | $\frac{5k}{24}$ |
| $\geq 4$ | $\frac{k}{2^i}$ | $\frac{k}{2^i}$ | $\frac{k}{2^i}$ | $\frac{k}{4}$ |
| | Sum | $\frac{13}{12}k-1$ | $\frac{k}{2}-1$ | $\frac{1}{4}k\log k-\frac{13k}{24}$ |

Table 5.1: Number of comparators used in different iterations of Algorithm 5.2

We now count the number of variables and clauses for the 4-Wise Selection Network, again, disregarding the recursive steps.

**Lemma 5.5.** *Let $k \in \mathbb{N}$. Then:*

$$V(4w\_merge_k^{4k}) = k\log k + \frac{7}{6}k - 5,$$
$$C(4w\_merge_k^{4k}) = \frac{15}{4}k\log k - \frac{33}{24}k - 10.$$

*Proof.* We separately count the number of 2, 3 and 4-comparators used in the merger (Algorithm 5.2). By the assumption that $k \leq n/4$ we get $|\bar{w}| = k$, $|\bar{x}| = k/2$, $|\bar{y}| = k/3$, $|\bar{z}| = k/4$ and $h_1 = k/2$. We consider iterations 1, 2 and 3 separately and then provide the formulas for the number of comparators for iterations 4 and beyond. Results are summarized in Table 5.1.

In the first iteration $h_1 = k/2$, which means that sets of $j$-values in the first two inner loops (lines 7–10 and 11–13) are empty. On the other hand, $1 \leq j \leq \min(k_1 - h_1, k_2, h_1) = k/2$ (line 14), therefore $k/2$ 2-comparators are used. In fact it is true for every iteration $i$ that $\min(k_1 - h_i, k_2, h_i) = h_i = k/2^i$. We note this fact in column 3 of Table 5.1 (the first term of each expression). For the next iterations we only need to consider the first and second inner loops of the algorithm.

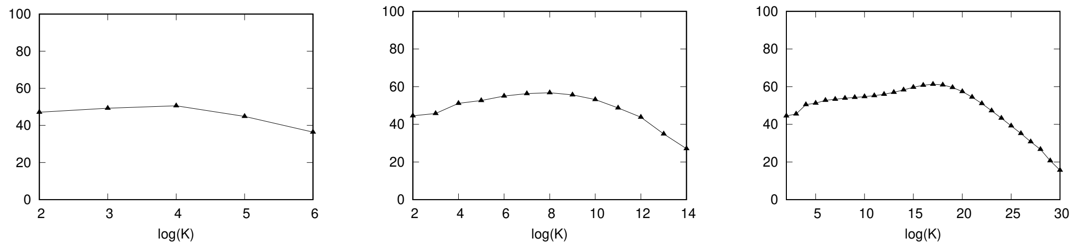
In the second iteration $h_2 = k/4$, therefore in the first inner loop only the condition in line 10 holds, and only when $j \leq k/3 - k/4 = k/12$, hence $k/12$ 2-comparators are used. In the second inner loop the $j$-values satisfy condition $1 \leq j \leq \min(k_2 - h_2, k_3, h_2) = k/4$. Therefore the condition in line 12 holds for each $j \leq k/4$, hence $k/4$ 3-comparators are used. In fact it is true for every iteration $i \geq 2$ that $\min(k_2 - h_i, k_3, h_i) = k/2^i$, so the condition in line 12 is true for $1 \leq j \leq k/2^i$. We note this fact in column 4 of Table 5.1. For the next iterations we only need to consider the first inner loop of the algorithm.

In the third iteration $h_3 = k/8$, therefore $j$-values in the first inner loop satisfy the condition $1 \leq j \leq \min(k_3 - h_3, k_4) = 5k/24$ and the condition in line 8 holds for each $j \leq 5k/24$, hence $5k/24$ 4-comparators are used.

From the forth iteration $h_i \leq k/16$, therefore for every $1 \leq j \leq \min(k_3 - h_i, k_4) = k/4$ the condition in line 8 holds. Therefore $k/4$ 4-comparators are used.

What's left is to sum 2,3 and 4-comparators throughout $\log k$ iterations of the algorithm. The results are are presented in Table 5.1. Elementary calculation gives the desired result. □

Let:




(a) $N = 2^7$ (b) $N = 2^{15}$ (c) $N = 2^{31}$

Figure 5.5: Percentage of variables saved in 4-Wise Selection Networks compared to Pairwise Selection Networks, for selected values of $N$ and $K$. Graphs are plotted from the formula $100 \cdot (V(pw\_sel^N_K) - V(4w\_sel^N_K))/V(pw\_sel^N_K)$.

$$\begin{aligned}
V_{PSN} &= V(pw_merge^{n/2}_k) + V(pw_merge^{n/2}_{k/2}) + V(pw_merge^n_k),\\
C_{PSN} &= C(pw_merge^{n/2}_k) + C(pw_merge^{n/2}_{k/2}) + C(pw_merge^n_k),\\
V_{4W} &= V(4w_merge^{4k}_k),\\
C_{4W} &= C(4w_merge^{4k}_k).
\end{aligned}$$

The following corollary shows that 4-column pairwise merging networks produces smaller encodings than their 2-column counterpart.

**Corollary 5.2.** *Let $k \in \mathbb{N}$ such that $k \geq 4$. Then $V_{4W} < V_{PCN}$ and $C_{4W} < C_{PCN}$.*

For the following theorem, note that $V(4w\_split^n) = V(pw\_split^n) = n$.

**Theorem 5.3.** *Let $n,k \in \mathbb{N}$ such that $1 \leq k \leq n/4$ and $n$ and $k$ are both powers of 4. Then $V(4w\_sel^n_k) \leq V(pw\_sel^n_k)$.*

*Proof.* By induction. For the base case, consider $1 = k < n$. It follows that $V(4w\_sel^n_k) = V(pw\_sel^n_k) = V(max^n)$. For the induction step assume that for each $(n',k') \prec (n,k)$ (in lexicographical order), where $k \geq 4$, the inequality holds, we get:

$$\begin{aligned}
&V(4w_sel^n_k) = V(4w_split^n) + \textstyle\sum_{1\leq i\leq 4} V(4w_sel^{n/4}_{k/i}) + V(4w_merge^{4k}_k)\\
&\qquad\qquad\textbf{(by the construction of } 4w_sel\textbf{)}\\
&\leq V(4w_split^n) + \textstyle\sum_{1\leq i\leq 4} V(pw_sel^{n/4}_{k/i}) + V(4w_merge^{4k}_k)\\
&\qquad\qquad\textbf{(by the induction hypothesis)}\\
&\leq V(pw_split^n) + 2V(pw_split^{n/2}) + \textstyle\sum_{1\leq i\leq 4} V(pw_sel^{n/4}_{k/i})\\
&\quad + V(pw_merge^{n/2}_k) + V(pw_merge^{n/2}_{k/2}) + V(pw_merge^n_k)\\
&\quad \textbf{(by Corollary 5.2 and because } V(4w_split^n) < V(pw_split^n) + 2V(pw_split^{n/2})\textbf{)}\\
&\leq V(pw_split^n) + 2V(pw_split^{n/2}) + V(pw_sel^{n/4}_k) + 2V(pw_sel^{n/4}_{k/2})\\
&\quad + V(pw_sel^{n/4}_{k/4}) + V(pw_merge^{n/2}_k) + V(pw_merge^{n/2}_{k/2}) + V(pw_merge^n_k)\\
&\qquad\qquad\textbf{(because } V(pw_sel^{n/4}_{k/3}) \leq V(pw_sel^{n/4}_{k/2})\textbf{)}\\
&= V(pw_sel^n_k)\\
&\qquad\qquad\textbf{(by the construction of } pw_sel\textbf{)}
\end{aligned}$$

$\square$

In Figure 5.5 we show what percentage of variables is saved while using our 4-Wise Selection Networks instead of Pairwise Selection Networks. We see that the number of variables saved can be up to 60%.

## 5.5 Summary

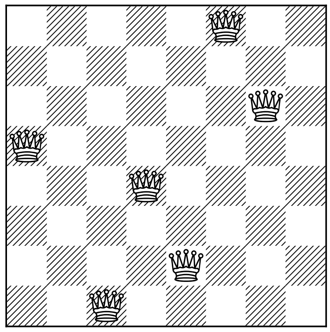
In this chapter we presented a family of multi-column selection networks based on the pairwise approach, that can be used to encode cardinality constraints. We showed a detailed construction where the number of columns is equal to 4 and we showed that the encoding is smaller than its 2-column counterpart.

# Part III

# Odd-Even Selection Networks

# Chapter 6

# Generalized Odd-Even Selection Networks



Even though it has been shown that the pairwise networks use less comparators than the odd-even networks [78] (for selected values of $n$ and $k$), it is the latter that achieve better practical results in the context of encoding cardinality constraints [3]. In this chapter we show a construction of a generalized selection network based on the odd-even approach called the 4-*Odd-Even Selection Network*. We show that our network is not only more efficient, but it is also easier to implement (and to prove its correctness) than the GSN based on the pairwise approach from the previous chapter.

The construction is the generalization of the multi-way merge sorting network by Batcher and Lee [54]. The main idea is to split the problem into 4 sub-problems, recursively select $k$ elements in them and then merge the selected subsequences using an idea of multi-way merging. In such a construction, we can encode more efficiently comparators in the combine phase of the merger: instead of encoding each comparator separately by 3 clauses and 2 additional variables, we propose an encoding scheme that requires 5 clauses and 2 variables on average for each pair of comparators.

We give a detailed construction for the 4-Odd-Even Merging Network. We compare the numbers of variables and clauses of the encoding and its counterpart: the 2-Odd-Even Merging Network [27]. The calculations show that encodings based on our network use fewer variables and clauses, when $k < n$.

The construction is parametrized by any values of $k$ and $n$ (just like $m$-Wise Selection Network from the previous chapter), so it can be further optimized by mixing them with other constructions. For example, in our experiments we mixed them with the direct encoding for small values of parameters. We show experimentally that multi-column selection networks are superior to standard selection networks previously proposed in the literature, in context of translating cardinality constraints into propositional formulas.

We also empirically compare our encodings with other state-of-the-art encodings, not only based on comparator networks, but also on binary adders and binary decision diagrams. Those are mainly used in encodings of Pseudo-Boolean constraints, but it is informative to see how well they perform when encoding cardinality constraints.

At the end of this chapter we show how we can generalize the 4-Odd-Even Selection Network to the *m-Odd-Even Selection Network*, for any $m \geq 2$, just like we showed the *m-Wise Selection Network* in Chapter 5.

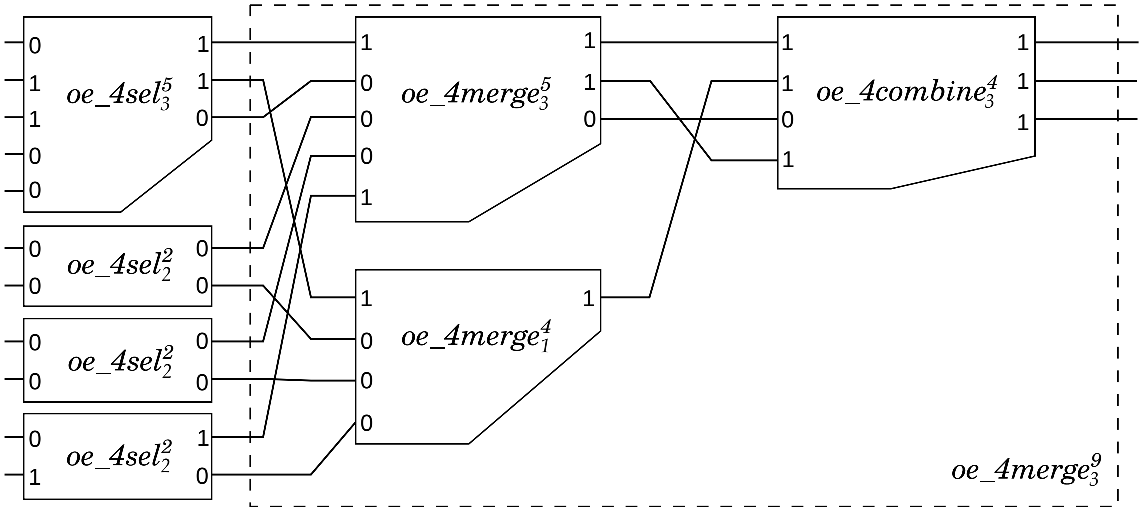



Figure 6.1: An example of 4-Odd-Even Selection Network, with $n = 11$, $k = 3$, $n_1 = 5$, $n_2 = n_3 = n_4 = 2$

## 6.1 4-Odd-Even Selection Network

We begin with the top-level algorithm for constructing the 4-Odd-Even Selection Network (Algorithm 6.1) where we use $oe\_4merge^s_k$ as a black box. It is a 4-merger of order $k$, that is, it outputs top $k$ sorted sequence from the inputs consisting of 4 sorted sequences. We give detailed construction of a 4-merger called *4-Odd-Even Merger* in the next sub-section.

The idea we use is the generalization of the one used in 2-Odd-Even Selection Network from [27], which is based on the Odd-Even Sorting Network by Batcher [17], but we replace the last network with Multiway Merge Sorting Network by Batcher and Lee [54]. We arrange the input sequence into 4 columns of non-increasing sizes (lines 3–6) and then recursively run the selection algorithm on each column (lines 9–11), where at most top $k$ items are selected from each column. Notice that each column is represented by ranges derived from the increasing value of variable *offset*. Notice further, that sizes of the columns are selected in such a way that in most cases all but first columns are of equal length and the length is a power of two (lines 3–5) that is close to the value of $k/4$ (observe that $[k/6, k/3)$ is the smallest symmetric interval around $k/4$ that contains a power of 2). Such a choice produces much longer propagation paths for small values of $k$ with respect to $n$. In the recursive calls selected items are sorted and form prefixes of the columns, which are then the input to the merging procedure (line 13). The base case, when $k = 1$ (line 2), is handled by the selector $sel^n_1$.

**Example 6.1.** In Figure 6.1 we present a schema of 4-Odd-Even Selection Network, which selects 3 largest elements from the input 01100000001. In this example, $n = 11$, $k = 3$, $n_1 = 5$, $n_2 = n_3 = n_4 = 2$. First, the input is passed to the recursive calls, then the procedure $oe\_4merge^9_3$ is applied (Algorithm 6.2).

**Theorem 6.1.** *Let $n, k \in \mathbb{N}$, such that $k \leq n$. Then $oe\_4sel^n_k$ is a k-selection network.*

*Proof.* Observe that $\bar{y} = \bar{y}^1 :: \cdots :: \bar{y}^4$ is a permutation of the input sequence $\bar{x}$. We prove by induction that for each $n, k \in \mathbb{N}$ such that $1 \leq k \leq n$ and each $\bar{x} \in \{0,1\}^n$: $oe\_4sel^n_k(\bar{x})$ is top $k$ sorted. If $1 = k \leq n$ then $oe\_4sel^n_k = sel^n_1$, so the theorem is true. For the induction step assume

**Algorithm 6.1** $oe\_4sel^n_k$

**Input:** $\bar{x} \in \{0,1\}^n$; $0 \leq k \leq n$
**Ensure:** The output is top $k$ sorted and is a permutation of the inputs

**if** $k = 0$ **or** $n \leq 1$ **then return** $\bar{x}$
**else if** $k = 1$ **then return** $sel^n_1(\bar{x})$
**if** $n < 8$ **or** $k = n$ **then** $n_2 = \lfloor (n+2)/4 \rfloor$; $n_3 = \lfloor (n+1)/4 \rfloor$; $n_4 = \lfloor n/4 \rfloor$; # divide evenly
**else if** $2^{\lceil \log(k/6) \rceil} \leq \lfloor n/4 \rfloor$ **then** $n_2 = n_3 = n_4 = 2^{\lceil \log(k/6) \rceil}$ # divide into powers of 2
**else** $n_2 = n_3 = n_4 = \lfloor k/4 \rfloor$ # otherwise, if the power of 2 is too far from k/4
$n_1 = n - n_2 - n_3 - n_4$ # $n = n_1 + \cdots + n_4$ and $n_1 \geq n_2 \geq n_3 \geq n_4$
$offset = 1$
**for all** $i \in \{1, \dots, 4\}$ **do**
  $k_i = \min(k, n_i)$
  $\bar{y}^i \leftarrow oe\_4sel^{n_i}_{k_i}(\langle x_{offset}, \dots, x_{offset+n_i-1} \rangle)$ # recursive calls
  $offset += n_i$
$s = \sum_{i=1}^{4} k_i$; $\overline{out} = \mathrm{suff}(k_1+1, \bar{y}^1) :: \cdots :: \mathrm{suff}(k_4+1, \bar{y}^4)$
**return** $oe\_4merge^s_k(\langle \mathrm{pref}(k_1, \bar{y}^1), \dots, \mathrm{pref}(k_4, \bar{y}^4) \rangle) :: \overline{out}$

**Algorithm 6.2** $oe\_4merge^s_k$

**Input:** A tuple of sorted sequences $\langle \bar{w}, \bar{x}, \bar{y}, \bar{z} \rangle$, where $1 \leq k \leq s = |\bar{w}| + |\bar{x}| + |\bar{y}| + |\bar{z}|$ and $k \geq |\bar{w}| \geq |\bar{x}| \geq |\bar{y}| \geq |\bar{z}|$.
**Ensure:** The output is top $k$ sorted and is a permutation of the inputs

**if** $|\bar{x}| = 0$ **then return** $\bar{w}$
**if** $|\bar{w}| = 1$ **then return** $sel^s_k(\bar{w} :: \bar{x} :: \bar{y} :: \bar{z})$ # Note that $s \leq 4$ in this case
$s_a = \lceil |\bar{w}|/2 \rceil + \lceil |\bar{x}|/2 \rceil + \lceil |\bar{y}|/2 \rceil + \lceil |\bar{z}|/2 \rceil$; $k_a = \min(s_a, \lfloor k/2 \rfloor + 2)$;
$s_b = \lfloor |\bar{w}|/2 \rfloor + \lfloor |\bar{x}|/2 \rfloor + \lfloor |\bar{y}|/2 \rfloor + \lfloor |\bar{z}|/2 \rfloor$; $k_b = \min(s_b, \lfloor k/2 \rfloor)$
$\bar{a} \leftarrow oe\_4merge^{s_a}_{k_a}(\bar{w}_{odd}, \bar{x}_{odd}, \bar{y}_{odd}, \bar{z}_{odd})$ # Recursive calls.
$\bar{b} \leftarrow oe\_4merge^{s_b}_{k_b}(\bar{w}_{even}, \bar{x}_{even}, \bar{y}_{even}, \bar{z}_{even})$
**return** $oe\_4combine^{k_a+k_b}_k(\mathrm{pref}(k_a, \bar{a}), pref(k_b, \bar{b})) :: \mathrm{suff}(k_a+1, \bar{a}) :: \mathrm{suff}(k_b+1, \bar{b})$

that $n \geq k \geq 2$ and for each $(n^*, k^*) \prec (n, k)$ (in lexicographical order) the theorem holds. We have to prove that the sequence $\bar{w} = \mathrm{pref}(k_1, \bar{y}^1) :: \cdots :: \mathrm{pref}(k_4, \bar{y}^4)$ contains $k$ largest elements from $\bar{x}$. If all 1's from $\bar{y}$ are in $\bar{w}$, we are done. So assume that there exists $y^i_j = 1$ for some $1 \leq i \leq 4$, $k_i < j \leq n_i$. We show that $|\bar{w}|_1 \geq k$. Notice that $k_i = k$, otherwise $j > k_i = n_i$ – a contradiction. Since $|\bar{y}^i| = n_i \leq n_1 < n$, from the induction hypothesis we get that $\bar{y}^i$ is top $k_i$ sorted. In consequence, each element of $\mathrm{pref}(k_i, \bar{y}^i)$ is greater or equal to $y^i_j$, which implies that $|\mathrm{pref}(k_i, y^i)|_1 = k_i = k$. We conclude that $|\bar{w}|_1 \geq |\mathrm{pref}(k_i, y^i)|_1 = k$. Note also that in the case $n = k$ we have all $k_i = \min(n_i, k) < k$, so the case is correctly reduced.

Finally, using $oe\_4merge^s_k$ the algorithm returns $k$ largest elements from $\bar{x}$, which completes the proof. □

## 6.2 4-Odd-Even Merging Network

In this section we give the detailed construction of the network $oe\_4merge$ – the 4-Odd-Even Merger – that merges four sequences (columns) obtained from the recursive calls in Algorithm 6.1. We can assume that input columns are sorted and of length at most $k$.

The network is presented in Algorithm 6.2. The input to the procedure is $\langle \mathrm{pref}(k_1, \bar{y}^1)$, ..., $\mathrm{pref}(k_4, \bar{y}^4) \rangle$, where each $\bar{y}^i$ is the output of the recursive call in Algorithm 6.1. The goal is to return the $k$ largest (and sorted) elements. It is done by splitting each input sequence into two parts, one containing elements of odd index, the other containing elements of even index. Odd

**Algorithm 6.3** *oe_4combine*$^s_k$

**Input:** A pair of sorted sequences $\langle \bar{x}, \bar{y} \rangle$, where $k \le s = |\bar{x}| + |\bar{y}|$, $|\bar{y}| \le \lfloor k/2 \rfloor$, $|\bar{x}| \le \lfloor k/2 \rfloor + 2$ and $|\bar{y}|_1 \le |\bar{x}|_1 \le |\bar{y}|_1 + 4$.
**Ensure:** The output is sorted and is a permutation of the inputs

Let $x(i)$ denote 0 if $i > |\bar{x}|$ or else $x_i$. Let $y(i)$ denote 1 if $i < 1$ or 0 if $i > |\bar{y}|$ or $y_i$, otherwise.
**for all** $j \in \{1, \ldots, |\bar{x}| + |\bar{y}|\}$ **do**
  $i = \lceil j/2 \rceil$
  **if** $j$ is even **then** $a_j \leftarrow \max(\max(x(i+2), y(i)), \min(x(i+1), y(i-1)))$
  **else** $a_j \leftarrow \min(\max(x(i+1), y(i-1)), \min(x(i), y(i-2)))$
**return** $\bar{a}$

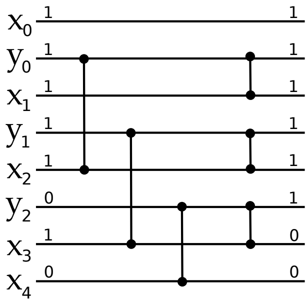



Figure 6.2: Comparators of *oe_4combine*$^8_6$ and the result of ordering the sequence 11111010

sequences and even sequences are then recursively merged (lines 5–6) into two sequences $\bar{a}$ and $\bar{b}$ that are top $k$ sorted. The sorted prefixes are then combined by *oe_4combine* into a sorted sequence to which the suffixes of $\bar{a}$ and $\bar{b}$ are appended. The result is top $k$ sorted. For base cases, since we assume that $|\bar{w}| \ge |\bar{x}| \ge |\bar{y}| \ge |\bar{z}|$, we need only to check – in line 1 – if $\bar{x}$ is empty (then only $\bar{w}$ is non-empty) or – in line 2 – if $\bar{w}$ contains only a single element – then the rest of the sequences contains at most one element and we can simply order them with a selector. In other cases we have $|\bar{w}| \ge 2$ and $|\bar{x}| \ge 1$, thus $|\bar{w}_{odd}| < |\bar{w}|$ and $|\bar{w}_{even}| < |\bar{w}|$, so the sizes of sub-problems solved by recursive calls decrease.

Our network is the generalization of the classic Multiway Merge Sorting Network by Batcher and Lee [54], where we use 4-way mergers and each merger consists of two sub-mergers and a combine sub-network. The goal of our network is to select and sort the $k$ largest items of four sorted input sequences. The combine networks are described and analyzed in [54].

In the case of 4-way merger, the combine operation by Batcher and Lee [54] uses two layers of comparators to fix the order of elements of two sorted sequences $\bar{x}$ and $\bar{y}$, as presented in Figure 6.2. The combine operation takes sequences $\langle x_0, x_1, \ldots \rangle$ and $\langle y_0, y_1, \ldots \rangle$ and performs a zip operation: $\langle x_0, y_0, x_1, y_1, x_2, y_2, \ldots \rangle$. Then, two layers of comparators are applied: $[y_i : x_{i+2}]$, for $i = 0, 1, \ldots$, resulting in $\langle x'_0, y'_0, x'_1, y'_1, \ldots \rangle$, and then $[y'_i : x'_{i+1}]$, for $i = 0, 1, \ldots$, to get $\langle x''_0, y''_0, x''_1, y''_1, \ldots \rangle$.

If we were to directly encode each comparator separately in a combine operation we would need to use 3 clauses and 2 additional variables on each comparator. The novelty of our construction is that the encoding of a combine phase requires 5 clauses and 2 variables on average for each pair of comparators, using the following observations:

$$y'_i = \max(y_i, x_{i+2}) \equiv y_i \vee x_{i+2} \quad i = 0, 1, \ldots$$
$$x'_i = \min(y_{i-2}, x_i) \equiv y_{i-2} \wedge x_i \quad i = 2, 3, \ldots$$

and

$$\begin{aligned} y_i'' &= \max(y_i', x_{i+1}') = y_i' \vee x_{i+1}' = y_i \vee x_{i+2} \vee (y_{i-1} \wedge x_{i+1}), \\ x_i'' &= \min(y_{i-1}', x_i') = y_{i-1}' \wedge x_i' = (y_{i-1} \vee x_{i+1}) \wedge y_{i-2} \wedge x_i \\ &= (y_{i-1} \wedge y_{i-2} \wedge x_i) \vee (y_{i-2} \wedge x_i \wedge x_{i+1}) = (y_{i-1} \wedge x_i) \vee (y_{i-2} \wedge x_{i+1}) \end{aligned}$$

In the above calculations we use the fact that the input sequences are sorted, therefore $y_{i-1} \wedge y_{i-2} = y_{i-1}$ and $x_i \wedge x_{i+1} = x_{i+1}$. By the above observations, the two calculated values can be encoded using the following set of 5 clauses:

$$y_i \Rightarrow y_i'', x_{i+2} \Rightarrow y_i'', y_{i-1} \wedge x_{i+1} \Rightarrow y_i'', y_{i-1} \wedge x_i \Rightarrow x_i'', y_{i-2} \wedge x_{i+1} \Rightarrow x_i''$$

if 1's should be propagated from inputs to outputs, otherwise:

$$y_i'' \Rightarrow y_{i-1} \vee x_{i+2}, y_i'' \Rightarrow y_i \vee x_{i+1}, x_i'' \Rightarrow x_i, x_i'' \Rightarrow y_{i-2}, x_i'' \Rightarrow y_{i-1} \vee x_{i+1}.$$

This saves one clause and two variables for each pair of comparators in the original combine operation, which scales to $\frac{1}{2}k$ clauses and $k$ variables saved for each two layers of comparators associated with the use of a 4-way merger. The pseudo code for our combine procedure is presented in Algorithm 6.3.

**Example 6.2.** In Figure 6.1, in dashed lines, a schema of 4-Odd-Even merger is presented with $s = 9$, $k = 3$, $k_1 = 3$ and $k_2 = k_3 = k_4 = 2$. First, the input columns are split into two by odd and even indexes, and the recursive calls are made. After that, a combine operation fixes the order of elements, to output the 3 largest ones. For more detailed example of Algorithm 6.2, assume that $k = 6$ and $\bar{w} = 100000$, $\bar{x} = 111000$, $\bar{y} = 100000$, $\bar{z} = 100000$. Then $\bar{a} = oe\_4merge_5^{12}(100, 110, 100, 100) = 111110000000$ and $\bar{b} = oe\_4merge_3^{12}(000, 100, 000, 000) = 100000000000$. The combine operation gets $\bar{x} = \text{pref}(5, \bar{a}) = 11111$ and $\bar{y} = \text{pref}(3, \bar{b}) = 100$. Notice that $|\bar{x}|_1 - |\bar{y}|_1 = 4$ and after zipping we get 11101011. Thus, two comparators from the first layer are needed to fix the order.

**Theorem 6.2.** *The output of Algorithm 6.2 is top k sorted.*

We start with proving a lemma stating that the result of applying network *oe_4combine* to any two sequences that satisfy the requirements of the network is sorted and is a permutation of inputs. Then we prove the theorem.

**Lemma 6.1.** *Let $k \geq 1$ and $\bar{x}, \bar{y} \in \{0,1\}^*$ be a pair of sorted sequences such that $k \leq s = |\bar{x}| + |\bar{y}|$, $|\bar{y}| \leq \lfloor k/2 \rfloor$, $|\bar{x}| \leq \lfloor k/2 \rfloor + 2$ and $|\bar{y}|_1 \leq |\bar{x}|_1 \leq |\bar{y}|_1 + 4$. Let $\bar{a}$ be the output sequence of $oe\_4combine_k^s(\bar{x}, \bar{y})$. Then for any $j$, $1 \leq j < s$ we have $a_j \geq a_{j+1}$. Moreover, $\bar{a}$ is a permutation of $\bar{x} :: \bar{y}$.*

*Proof.* Note first that the notations $x(i)$ and $y(i)$ (introduced in Algorithm 6.3) defines monotone sequences that extend the given input sequences $\bar{x}$ and $\bar{y}$ (which are sorted). Observe that the inequality is obvious for an even $j = 2i$, because $a_{2i} = \max(\max(x(i+2), y(i)), \min(x(i+1), y(i-1))) \geq \min(\max(x(i+2), y(i)), \min(x(i+1), y(i-1))) = a_{2j+1}$. Consider now an odd $j = 2i-1$ for which $a_{2i-1} = \min(\max(x(i+1), y(i-1)), \min(x(i), y(i-2)))$. We show that all three values: (1) $\max(x(i+1), y(i-1))$, (2) $x(i)$ and (3) $y(i-2)$ are upper bounds on $a_{2j}$. Then the minimum of them is also an upper bound on $a_{2j}$.

We have the following inequalities as the consequence of the assumptions: $x(l) \geq y(l) \geq x(l+4)$ for any integer $l$. Using them and the monotonicity of $x(i)$, $y(i)$ and the min / max functions, we have:

(1) $\max(x(i+1), y(i-1)) \geq \max(\max(x(i+2), y(i)), \min(x(i+1), y(i-1))) = a_{2j}$,

(2) $x(i) \geq \max(\max(x(i+2), y(i)), \min(x(i+1), y(i-1))) = a_{2j}$ and

(3) $y(i-2) \geq \max(\max(x(i+2), y(i)), \min(x(i+1), y(i-1))) = a_{2j}$.

In (2) we use $x(i) \geq x(i+1) \geq \min(x(i+1), y(i-1))$. In (3) - the similar ones.

To prove the second part of the lemma let us introduce an intermediate sequence $b_j$, $1 \leq j \leq s+1$ such that $b_{2i} = \max(x(i+2), y(i))$ and $b_{2i-1} = \min(x(i), y(i-2)$ and observe that it is a permutation of $\bar{x} :: \bar{y} :: 0$, since a pair $b_{2i}$ and $b_{2i+3} = \min(x(i+2), y(i))$ is a permutation of the pair $x(i+2)$ and $y(i)$. Now we can write $a_{2i}$ as $\max(b_{2j}, b_{2j+1})$ and $a_{2i+1}$ as $\min(b_{2j}, b_{2j+1})$, thus the sequence $\bar{a} :: 0$ is a permutation of $\bar{b}$ and we are done. □

*Proof of Theorem 6.2.* Let $k \geq 1$ and $\bar{w}$, $\bar{x}$, $\bar{y}$ and $\bar{z}$ be sorted binary sequences such that $k \leq s = |\bar{w}| + |\bar{x}| + |\bar{y}| + |\bar{z}|$ and $k \geq |\bar{w}| \geq |\bar{x}| \geq |\bar{y}| \geq |\bar{z}|$. Assume that they are the inputs to the network $oe\_4merge^s_k$, so we can use in the following the variables and sequences defined in it. The two base cases are: (1) all but first sequences are empty, and (2) all sequences contain at most one item. In both of them the network trivially select the top $k$ items. In the other cases the construction of $oe\_4merge^s_k$ is recursive, so we proceed by induction on $s$. Observe then that $s_a, s_b < s$, since $|\bar{w}| \geq 2$ and $|\bar{x}| \geq 1$. By induction hypothesis, $\bar{a}$ is top $k_a$ sorted and $\bar{b}$ is top $k_b$ and $\bar{a} :: \bar{b}$ is an permutation of the inputs. Let $\bar{c} = oe\_4combine^{k_a+k_b}_k(\mathrm{pref}(k_a, \bar{a}), \mathrm{pref}(k_b, \bar{b})$. By previous lemma $\bar{c}$ is sorted and is a permutation of $\mathrm{pref}(k_a, \bar{a}) :: \mathrm{pref}(k_b, \bar{b})$. Thus the output sequence $\bar{c} :: \mathrm{suff}(k_a + 1, \bar{a}) :: \mathrm{suff}(k_b + 1, \bar{b})$ is a permutation of $\bar{w} :: \bar{x} :: \bar{y} :: \bar{z}$ and it remains only to prove that the output is top $k$ sorted.

If $\mathrm{suff}(k_a + 1, \bar{a}) :: \mathrm{suff}(k_b + 1, \bar{b})$ contains just zeroes, there is nothing to prove. Assume then that it contains at least one 1. In this case we prove that $\mathrm{pref}(k_a, \bar{a}) :: \mathrm{pref}(k_b, \bar{b})$ contains at least $k$ 1's, thus $c_k$ is 1, and the output is top $k$ sorted. Observe that $k_a + k_b \geq k$, because $s_b \leq s_a \leq s_b + 4$ and $s_a + s_b = s \geq k$ so $s_a \geq \lceil s/2 \rceil$ and $s_b \geq \lceil s/2 \rceil - 2$.

It is clear that $|\bar{b}|_1 \leq |\bar{a}|_1 \leq |\bar{b}|_1 + 4$, because in each sorted input at odd positions there is the same number of 1's or one more as at even positions. Assume first that $|\mathrm{suff}(k_a + 1, \bar{a})|_1 > 0$. Then the suffix in non-empty, so $k_a = \lfloor k/2 \rfloor + 2$ and $\mathrm{pref}(k_a, \bar{a})$ must contain only 1's, thus $|\bar{a}|_1 \geq k_a + 1 \geq \lfloor k/2 \rfloor + 3$. It follows that $|\bar{b}|_1 \geq \lfloor k/2 \rfloor - 1$. If $\mathrm{pref}(k_b, \bar{b})$ contains only 1's then $\bar{c}$ also consists only of 1's. Otherwise the prefix must contain $\lfloor k/2 \rfloor - 1$ 1's and thus the total number of 1's in $\bar{c}$ is at least $\lfloor k/2 \rfloor + 2 + \lfloor k/2 \rfloor - 1 \geq k$.

Assume next that $|\mathrm{suff}(k_b + 1, \bar{b})|_1 > 0$. Then $\mathrm{pref}(k_b, \bar{b})$ must contain only 1's and $|\bar{b}|_1 \geq k_b + 1 \geq \lfloor k/2 \rfloor + 1$. Since $k_a \geq \lceil k/2 \rceil$ and $|\bar{a}|_1 \geq |\bar{b}|_1 \geq \lfloor k/2 \rfloor + 1$, we get $|\mathrm{pref}(k_a, \bar{a})|_1 \geq \lceil k/2 \rceil$ and finally $|\bar{c}|_1 \geq \lfloor k/2 \rfloor + \lceil k/2 \rceil \geq k$. □

## 6.3 Comparison of Odd-Even Selection Networks

In this section we estimate and compare the number of variables and clauses in encodings based on our algorithm to some other encoding based on the odd-even selection. Such encoding – which we call the 2-Odd-Even Selection Network – was already analyzed by Codish and Zazon-Ivry [27]. We start by counting how many variables and clauses are needed in order to merge 4 sorted sequences returned by recursive calls of the 4-Odd-Even Selection Network and the 2-Odd-Even Selection Network. Then, based on those values we prove that the total number of variables and clauses is almost always smaller when using the 4-column encoding rather than the 2-column encoding. In the next section we show that the new encoding is not just smaller, but also have better solving times for many benchmark instances.

To simplify the presentation we assume that $k \leq n/4$ and both $k$ and $n$ are the powers of 4. We also omit the ceiling and floor function in the calculations, when it is convenient for us.

**Definition 6.1.** Let $n,k \in \mathbb{N}$. For given (selection) network $f_k^n$ let $V(f_k^n)$ and $C(f_k^n)$ denote the number of variables and clauses used in the standard CNF encoding of $f_k^n$.

We remind the reader that a single 2-sorter uses 2 auxiliary variables and 3 clauses. In case of a 4-sorter the numbers are 4 and 15 (by Definition 2.10).

We count how many variables and clauses are needed in order to merge 4 sorted sequences returned by recursive calls of the 2-Odd-Even Selection Network and the 4-Odd-Even Selection Network, respectively. Two-column selection network using the odd-even approach is presented in [27]. We briefly introduce this network with the following three-step recursive procedure (omitting the base case):

1. Split the input $\bar{x} \in \{0,1\}^n$ into two sequences $\bar{x}^1 = \bar{x}_{odd}$ and $\bar{x}^2 = \bar{x}_{even}$.
2. Recursively select top $k$ sorted elements from $\bar{x}^1$ and top $k$ sorted elements from $\bar{x}^2$ (into $\bar{y}^1$ and $\bar{y}^2$, respectively).
3. Merge the outputs of the previous step using an 2-Odd-Even Merging Network and output the top $k$ from $2k$ elements (top $k$ elements from $\bar{y}^1$ and top $k$ elements from $\bar{y}^2$).

If we treat the merging step as a network $oe\_2merge_k^{2k}$, then the number of 2-sorters used in the 2-Odd-Even Selection Network can be written as:

$$|oe\_2sel_k^n| = \begin{cases} 2|oe\_2sel_k^{n/2}| + |oe\_2merge_k^{2k}| & \text{if } k < n \\ |oe\_sort^k| & \text{if } k = n \\ |max^n| & \text{if } k = 1 \end{cases} \tag{6.1}$$

One can check that Step 3 requires $|oe\_2merge_k^{2k}| = k\log k + 1$ 2-sorters (see [27]), which leads to the simple lemma.

**Lemma 6.2.** $V(oe\_2merge_k^{2k}) = 2k\log k + 2$, $C(oe\_2merge_k^{2k}) = 3k\log k + 3$.

The schema of this network is presented in Figure 6.3. In order to count the number of comparators used when merging 4 sorted sequences we need to expand the recursive step by one level (see Figure 6.3b).

Now we do the counting for our 4-way merging network based on Algorithm 6.2.

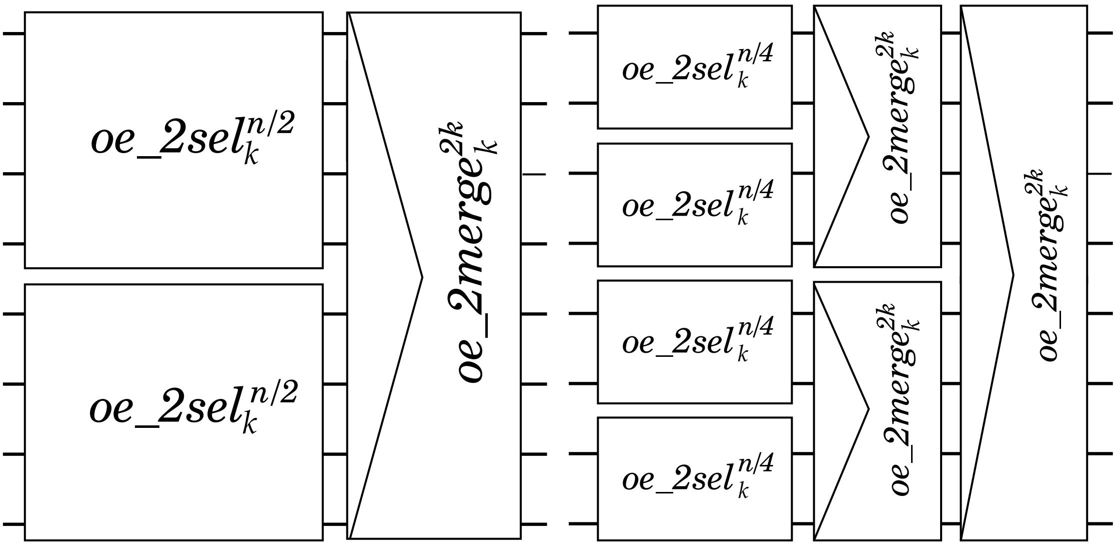
**Lemma 6.3.** *Let* $k \in \mathbb{N}$*, then:* $V(oe\_4merge_k^{4k}) \le (k-2)\log k + 5k - 1$*;* $C(oe\_4merge_k^{4k}) \le (\frac{5}{2}k - 5)\log k + 21k - 6$.

*Proof.* We separately count the number of variables and clauses used.

In the base case (line 2) we can assume – for the sake of the upper bound – that we always use 4-sorters. Notice, that the number of 4-sorters is only dependent on the variable $s$. The solution to the following recurrence gives the sought number: $\{A(4) = 1; A(s) = 2A(s/2), \text{for } s > 4\}$, which is equal to $s/4$. Therefore we use $s$ auxiliary variables and $(15/4)s$ clauses. We treat the recursive case separately below.

The number of variables used in the combine network is at most $k-1$, because a new variable is not needed for $a_i$, where $i > k$, because such $a_i$ can be replaced by a zero in clauses containing it, and not for $a_1 = x_1$. Therefore, the total number of variables is bounded by solution to the following recurrence:

$$B(s,k) = \begin{cases} 0 & \text{if } s \le 4 \\ B(s_a,k_a) + B(s_b,k_b) + k - 1 & \text{otherwise} \end{cases}$$




(a) one step (b) two steps

Figure 6.3: The 2-Odd-Even Selection Network

where $k \leq s = s_a + s_b \leq 4k$, $s_b \leq s_a \leq s_b + 4$ and $k_a = \min(s_a, \lfloor k/2 \rfloor + 2)$ and $k_b = \min(s_b, \lfloor k/2 \rfloor)$. Therefore $s/2 \leq s_a \leq s/2 + 2$, $s/2 - 2 \leq s_b \leq s/2$, $k_a \leq k/2 + 2$ and $k_b \leq k/2$. We claim that $B(s,k) \leq (k-2)(\log s - 2) + \frac{1}{4}s - 1$. This can be easily verified by induction.

The upper bound of the number of clauses can now be easily computed noticing that in the combine we require either 2 or 3 clauses for each new variable (depending on the parity of the index), therefore the number of clauses in the combiner is bounded by $2.5 \times \#vars + 3.5$. Constant factor 3 is added because additional clauses can be added for values $a_{k+1}$ and $a_{k+2}$ (see equations in Section 6.2). The overall number of clauses in the merger (omitting base cases) is then at most $2.5 \cdot B(s,k) + 3.5(k-1)$, where factor $(k-1)$ is the upper bound on the number of combines used in the recursive tree of the merger. Elementary calculations give the desired result. $\square$

Combining Lemmas 6.2 and 6.3 gives the following corollary.

**Corollary 6.1.** *Let $k \in \mathbb{N}$. Then $3V(oe\_2merge_k^{2k}) - V(oe\_4merge_k^{4k}) \geq (5k+2)\log(\frac{k}{2}) + 9 \geq 0$, and for $k \geq 8$, $3C(oe\_2merge_k^{2k}) - C(oe\_4merge_k^{4k}) \geq (\frac{13}{2}k + 5)\log(\frac{k}{8}) - \frac{3}{2}k + 30 \geq 0$.*

This shows that using our merging procedure gives a smaller encoding than its 2-column counterpart and the differences in the number variables and clauses used is significant.

The main result of this section is as follows.

**Theorem 6.3.** *Let $n, k \in \mathbb{N}$ such that $1 \leq k \leq n/4$ and $n$ and $k$ are both powers of 4. Then:*

$$dsV_k(n) \stackrel{df}{=} V(oe\_2sel_k^n) - V(oe\_4sel_k^n) \geq \frac{(n-k)(5k+2)}{3k} \log\left(\frac{k}{2}\right) + 3\left(\frac{n}{k} - 1\right).$$

*Proof.* Let $dV_k = 3V(oe\_2merge_k^{2k}) - V(oe\_4merge_k^{4k})$ (from Corollary 6.1), then:

$$\begin{aligned}
dsV_k(n) &= V(oe_2sel_k^n) - V(oe_4sel_k^n) \\
&= 2V(oe_2sel_k^{n/2}) + V(oe_2merge_k^{2k}) - 4V(oe_4sel_k^{n/4}) - V(oe_4merge_k^{4k}) \\
&= 4V(oe_2sel_k^{n/4}) + 3V(oe_2merge_k^{2k}) - 4V(oe_4sel_k^{n/4}) - V(oe_4merge_k^{4k}) \\
&= 4dsV_k(n/4) + dV_k
\end{aligned}$$

The solution to the above recurrence is $dsV_k(n) \geq \frac{1}{3}(\frac{n}{k}-1)dV_k$. Therefore:

$$\begin{aligned} dsV_k(n) &\geq \frac{1}{3}\left(\frac{n}{k}-1\right)\left((5k+2)\log\left(\frac{k}{2}\right)+9\right) \\ &= \frac{(n-k)(5k+2)}{3k}\log\left(\frac{k}{2}\right)+3\left(\frac{n}{k}-1\right). \end{aligned}$$

□

Similar theorem can be proved for the number of clauses (when $k \geq 8$).

## 6.4 Experimental Evaluation

As it was observed in [3], having a smaller encoding in terms of number of variables or clauses is not always beneficial in practice, as it should also be accompanied with a reduction of SAT-solver runtime. In this section we assess how our encoding based on the new family of selection networks affect the performance of a SAT-solver.

### 6.4.1 Methodology

Our algorithms that encode CNF instances with cardinality constraints into CNFs were implemented as an extension of MINICARD ver. 1.1, created by Mark Liffiton and Jordyn Maglalang[1]. MINICARD uses three types of solvers:

- *minicard* - the core MINICARD solver with native AtMost constraints,
- *minicard_encodings* - a cardinality solver using CNF encodings for AtMost constraints,
- *minicard_simp_encodings* - the above solver with simplification / pre-processing.

The main program in *minicard_encodings* has an option to generate a CNF formula, given a CNFP instance (CNF with the set of cardinality constraints) and to select a type of encoding applied to cardinality constraints. Program run with this option outputs a CNF instance that consists of collection of the original clauses with the conjunction of CNFs generated by given method for each cardinality constraint. No additional pre-processing and/or simplifications are made. Authors of *minicard_encodings* have implemented six methods to encode cardinality constraints and arranged them in one library called *Encodings.h*. Our modification of MINICARD is that we added implementation of the encoding presented in this chapter and put it in the library *Encodings_MW.h*. Then, for each CNFP instance and each encoding method, we used MINICARD to generate CNF instances. After preparing, the benchmarks were run on a different SAT-solver. Our extension of MINICARD, which we call KP-MINICARD, is available online[2].

In our evaluation we use the state-of-the-art SAT-solver COMINISATPS by Chanseok Oh[3] [64], which have collectively won six medals in SAT Competition 2014 and Configurable SAT Solver Challenge 2014. Moreover, the modification of this solver called MAPLECOMSPS won the Main Track category of SAT Competition 2016[4]. All experiments were carried out on the machines with Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz.

[1] See https://github.com/liffiton/minicard
[2] See https://github.com/karpiu/kp-minicard
[3] See http://cs.nyu.edu/%7echanseok/cominisatps/
[4] See http://baldur.iti.kit.edu/sat-competition-2016/

Detailed results are available online[5]. We publish spreadsheets showing running time for each instance, speed-up/slow-down tables for our encodings, number of time-outs met and total running time.

### 6.4.2 Encodings

We use our multi-column selection network for evaluation – the 4-Odd-Even Selection Network (**4OE**) based on Algorithms 6.1, 6.3 and 6.2. We compare our encoding to some others found in the literature. We consider the Pairwise Cardinality Networks [27]. We also consider a solver called MINISAT+[6] which implements techniques to encode Pseudo-Boolean constraints to propositional formulas [34]. Since cardinality constraints are a subclass of Pseudo-Boolean constraints, we can measure how well the encodings used in MINISAT+ perform, compared with our methods. The solver chooses between three techniques to generate SAT encodings for Pseudo-Boolean constraints. These convert the constraint to: a BDD structure, a network of binary adders, a network of sorters. The network of adders is the most concise encoding, but it can have poor propagation properties and often leads to longer computations than the BDD based encoding. The network of sorters is the implementation of classic odd-even (2-column) sorting network by Batcher [17]. Calling the solver we can choose the encoding with one of the parameters: *-ca*, *-cb*, *-cs*. By default, MINISAT+ uses the so called **Mixed** strategy, where program chooses which method (adders, BDDs or sorters) to use in the encodings. We do not include the **Mixed** strategy in the results, as the evaluation showed that it performs almost the same as *-cb* option. The generated CNFs were written to files with the option *-cnf=<file>*. Solver MINISAT+ have been slightly modified, namely, we fixed a pair of bugs such as the one reported in the experiments section of [1].

To sum up, here are the competitors' encodings used in this evaluation:

- **PCN** - the Pairwise Cardinality Networks (our implementation),
- **CA** - encodings based on Binary Adders (from MINISAT+),
- **CB** - encodings based on Binary Decision Diagrams (from MINISAT+),
- **CS** - the 2-Odd-Even Sorting Networks (from MINISAT+).

Encodings **4OE** and **PCN** were extended, following the idea presented in [3], where authors use Direct Cardinality Networks in their encodings for sufficiently small values of $n$ and $k$. Values of $n$ and $k$ for which we substitute the recursive calls with Direct Cardinality Network were selected based on the optimization idea in [3]. We minimize the function $\lambda \cdot V + C$, where $V$ is the number of variables and $C$ the number of clauses to determine when to switch to direct networks, and following authors' experimental findings, we set $\lambda = 5$.

Additionally, we compare our encodings with two state-of-the-art general purpose constraint solvers. First is the PBLIB ver. 1.2.1, by Tobias Philipp and Peter Steinke [67]. This solver implements a plethora of encodings for three types of constraints: at-most-one, at-most-k (cardinality constraints) and Pseudo-Boolean constraints. The PBLIB automatically normalizes the input constraints and decides which encoder provides the most effective translation. One of the implemented encodings for at-most-k constraints is based on the sorting network from the paper by Abío et al. [3]. One part of the PBLIB library is the program called *PBEncoder* which takes an input file and translate it into CNF using the PBLIB. We have generated CNF formulas from

[5]See http://www.ii.uni.wroc.pl/%7ekarp/sat/2018.html
[6]See https://github.com/niklasso/minisatp

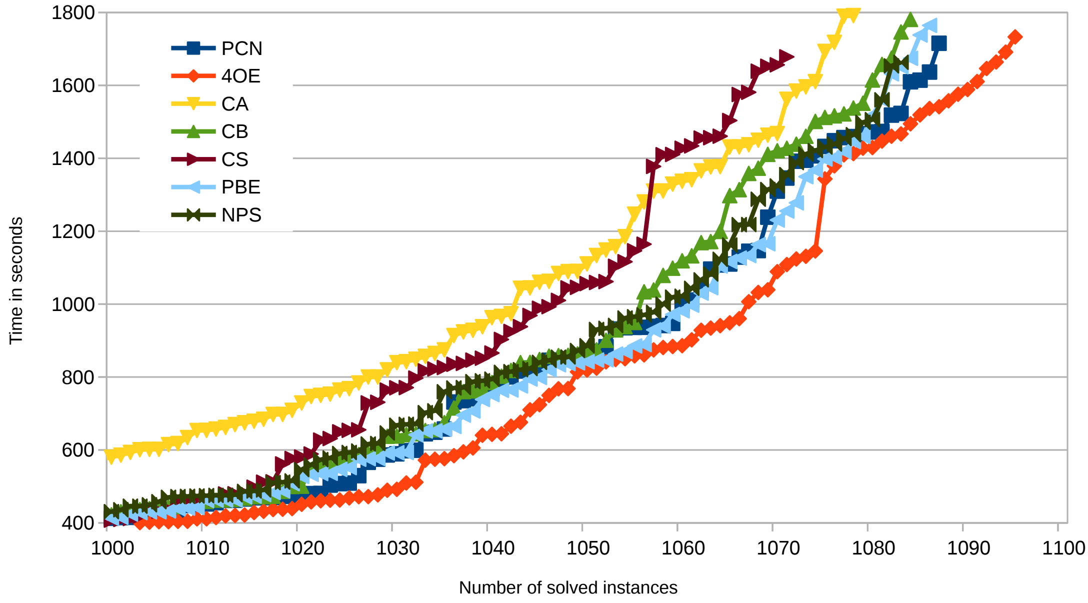



Figure 6.4: The number of solved instances of PB15 suite in a given time

all benchmark instances using this program, then we have run COMINISATPS on those CNFs. Results for this method are labeled **PBE** in our evaluation.

The second solver is the NPSOLVER by Norbert Manthey and Peter Steinke[7], which is a Pseudo-Boolean solver that translates Pseudo-Boolean constraints to SAT similar to MINISAT+, but which incorporates novel techniques. We have exchanged the SAT-solver used by default in NPSOLVER to COMINISATPS because the results were better with this one. Results for this method are labeled **NPS** in our evaluation.

### 6.4.3 Benchmarks

The set of benchmarks we used is **PB15 suite**, which is a set of instances from the Pseudo-Boolean Evaluation 2015[8]. One of the categories of the competition was *DEC-LIN-32-CARD*, which contains 2289 instances – we use these in our evaluation. Every instance is a collection of cardinality constraints.

### 6.4.4 Results

The time-out limit in the SAT-solver was set to 1800 seconds. When comparing two encodings we only considered instances for which at least one achieved the SAT-solver runtime of at least 10% of the time-out limit. All other instances were considered trivial, and therefore were not included in the speed-up/slow-down results. We also filtered out instances for which relative percentage deviation of the running time of encoding **A** w.r.t. the running time of encoding **B** was less than 10% (and vice-versa).

In Figure 6.4 we present a cactus plot, where x-axis gives the number of solved instances of PB15 suite and the y-axis the time needed to solve them (in seconds) using given encoding. From the plot we can see that the **4OE** encoding outperforms all other encodings.

[7] See http://tools.computational-logic.org/content/npSolver.php

[8] See http://pbeva.computational-logic.org/

| | 4OE speed-up | | | | | | 4OE slow-down | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | TO | 4.0 | 2.0 | 1.5 | 1.1 | Total | TO | 4.0 | 2.0 | 1.5 | 1.1 | Total | Time dif. | #s dif. |
| PCN | 9 | 11 | 5 | 5 | 5 | 35 | 1 | 2 | 1 | 3 | 3 | 10 | +02:55 | -8 |
| CA | 22 | 15 | 28 | 21 | 21 | 107 | 5 | 3 | 5 | 4 | 11 | 28 | +10:54 | -17 |
| CB | 18 | 11 | 15 | 8 | 24 | 76 | 7 | 2 | 6 | 3 | 28 | 46 | +04:54 | -11 |
| CS | 27 | 13 | 14 | 13 | 18 | 85 | 3 | 0 | 18 | 14 | 13 | 48 | +06:55 | -24 |
| PBE | 15 | 13 | 10 | 10 | 20 | 68 | 6 | 16 | 5 | 6 | 27 | 60 | +02:48 | -9 |
| NPS | 17 | 15 | 7 | 11 | 29 | 67 | 5 | 16 | 6 | 5 | 26 | 58 | +03:51 | -12 |

Table 6.1: Comparison of encodings in terms of SAT-solver runtime on PB15 suite. We count number of benchmarks for which **4OE** showed speed-up or slow-down factor with respect to different encodings, the difference in total running time of each encoding w.r.t. **4OE** and the difference in the number of solved instances of each encoding w.r.t. **4OE**.

Table 6.1 presents speed-up and slow-down factors for encoding **4OE** w.r.t. all other encodings. From the evaluation we can conclude that the best performing encoding is **4OE**. From the data presented in Table 6.1 our encoding achieve better speed-up factor w.r.t. all other encodings. Total running time for **4OE** is 629.78 hours on all 2289 instances. All other encodings required more time to finish the computation. Also, **4OE** solved the most number of instances – 1095. The second to last column of Table 6.1 shows the difference in total running time of all encodings w.r.t. **4OE** (in HH:MM format – hours and minutes). The last column indicates the difference in the number of solved instances of all encodings w.r.t. **4OE** (here all instances are counted, even the trivial ones). We can see, for example, that for **4OE** computations finished about 7 hours sooner for **4OE** than **CS**. This shows that using 4-column selection networks is more desirable than using 2-column selection/sorting networks for encoding cardinality constraints. Encodings **CA** and **CS** had the worst performance on PB15 suite. We can also see that even the state-of-the-art constraint solvers have larger running times and solved less instances on this set of benchmarks, as **PBE** and **NPS** finished computations more than about 3–4 hours later than **4OE**.

### 6.4.5 4-Wise vs 4-Odd-Even

Notice that in the evaluation we have omitted the 4-Wise Selection Network from Chapter 5. It is because preliminary experiments showed that the 4-Odd-Even Selection Network is superior, and since running the **PB15 suite** is very time-consuming we decided to showcase only our best encoding. To remedy this situation we have performed similar experiment on a different set of instances.

**MSU4 suite** is a set consisting of about 14000 benchmarks, each of which contains a mix of CNF formula and multiple cardinality constraints. This suite was created from a set of MaxSAT instances reduced from real-life problems, and then it was converted by the implementation of *msu4* algorithm [59]. This algorithm reduces a MaxSAT problem to a series of SAT problems with cardinality constraints. The MaxSAT instances were taken from the Partial Max-SAT division of the Third Max-SAT evaluation[9]. The time-out limit was set to 600 seconds.

The results are summarized in Figure 6.5. We show a cactus plot for **4OE**, **PCN** and **4WISE** – the implementation of the 4-Wise Selection Network based on Algorithms 5.1 and 5.2 (available in KP-MiniCard). Similar as before, we extended **4WISE** by using Direct Cardinality Networks for sufficiently small values of $n$ and $k$. The graph shows a clear difference in performance between all three encodings.

[9]See http://www.maxsat.udl.cat/08/index.php?disp=submitted-benchmarks

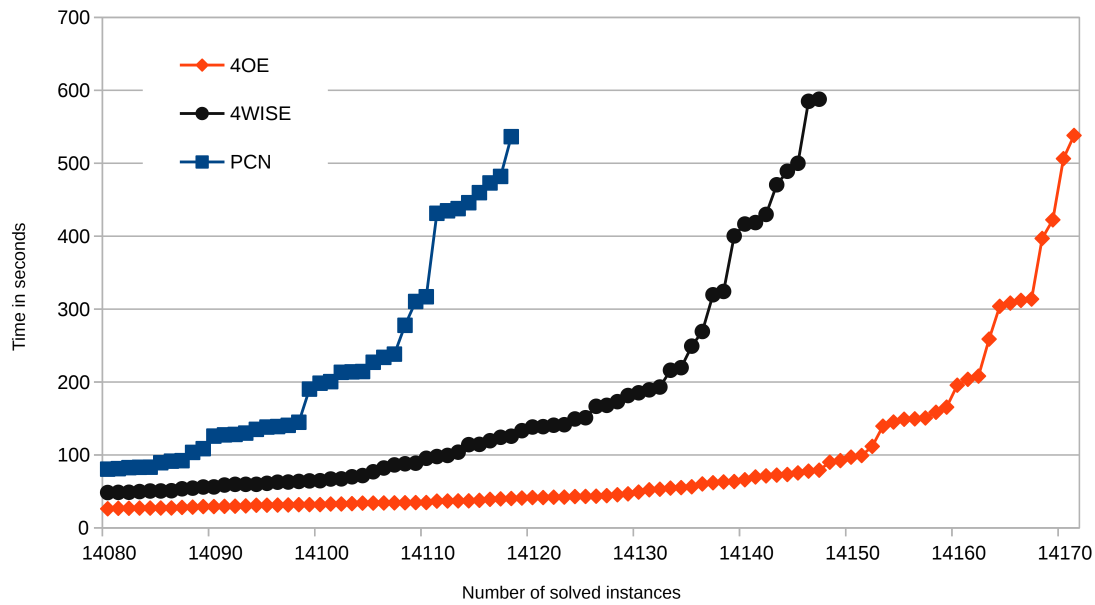



Figure 6.5: The number of solved instances of MSU4 suite in a given time

---

**Algorithm 6.4** $oe\_sel^n_k$

---

**Input:** $\bar{x} \in \{0,1\}^n$; $n_1, \dots, n_m \in \mathbb{N}$ where $n > n_1 \geq \dots \geq n_m$ and $\sum n_i = n$; $1 \leq k \leq n$

**Ensure:** The output is top $k$ sorted and is a permutation of the inputs

1: **if** $k = 1$ **then return** $sel^n_1(\bar{x})$

2: $offset = 1$

3: **for all** $i \in \{1, \dots, m\}$ **do**

4: $\quad k_i = \min(k, n_i)$

5: $\quad \bar{y}^i \leftarrow oe\_sel^{n_i}_{k_i}(\bar{x}_{offset,\dots,offset+n_i-1})$

6: $\quad offset += n_i$

7: $s = \sum_{i=1}^m k_i$; $\overline{out} = \mathrm{suff}(k_1+1, \bar{y}^1) :: \cdots :: \mathrm{suff}(k_m+1, \bar{y}^m)$

8: **return** $oe\_merge^s_k(\langle \mathrm{pref}(k_1, \bar{y}^1), \dots, \mathrm{pref}(k_m, \bar{y}^m)\rangle) :: \overline{out}$

---

## 6.5 m-Odd-Even Selection Network

We show that we can generalize our algorithm further, so that it can be parametrized by any value of $m \geq 2$. The construction of the $m$-Odd-Even Selection Network is presented in Algorithm 6.4.

We arrange the input sequence into $m$ columns of non-increasing sizes and we recursively run the selection algorithm on each column (lines 3–6), where at most $k$ items are selected from each column. Selected items are sorted and form prefixes of the columns and they are the input to the merging procedure (line 7–8). The base case, when $k = 1$, is handled by the selector $sel^n_1$.

**Theorem 6.4.** *Let $n, k \in \mathbb{N}$, such that $k \leq n$. Then $oe\_sel^n_k$ is a $k$-selection network.*

*Proof.* Observe that $\bar{y} = \bar{y}^1 :: \cdots :: \bar{y}^m$ is a permutation of the input sequence $\bar{x}$. We prove by induction that for each $n, k \in \mathbb{N}$ such that $1 \leq k \leq n$ and each $\bar{x} \in \{0,1\}^n$: $oe\_sel^n_k(\bar{x})$ is top $k$ sorted. If $1 = k \leq n$ then $oe\_sel^n_k = sel^n_1$, so the theorem is true. For the induction step assume that $n \geq k \geq 2$, $m \geq 2$ and for each $(n^*, k^*) \prec (n, k)$ (in lexicographical order) the theorem holds. We have to prove that the sequence $\bar{w} = \mathrm{pref}(k_1, \bar{y}^1) :: \cdots :: \mathrm{pref}(k_m, \bar{y}^m)$ contains $k$ largest elements from $\bar{x}$. If all 1's in $\bar{y}$ are in $\bar{w}$, we are done. So assume that there exists $y^i_j = 1$ for some $1 \le i \le m$, $k_i < j \le n_i$. We show that $|\bar{w}|_1 \ge k$. Notice that $k_i = k$, otherwise $j > k_i = n_i$ – a contradiction. Since $|\bar{y}^i| = n_i \le n_1 < n$, from the induction hypothesis we get that $\bar{y}^i$ is top $k_i$ sorted. In consequence, $\mathrm{pref}(k_i, \bar{y}^i) \succeq \langle y^i_j \rangle$, which implies that $|\mathrm{pref}(k_i, y^i)|_1 = k_i = k$. We conclude that $|\bar{w}|_1 \ge |\mathrm{pref}(k_i, y^i)|_1 = k$. Note also that in the case $n = k$ we have all $k_i = \min(n_i, k) < k$, so the case is correctly reduced.

Finally, using $oe\_merge^s_k$ the algorithm returns $k$ largest elements from $\bar{x}$, which completes the proof. □

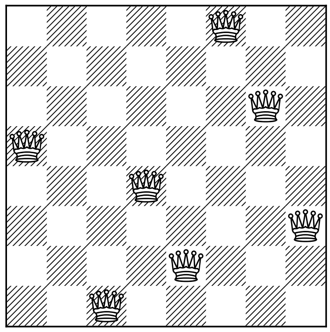
## 6.6 Summary

In this chapter we presented a multi-column selection networks based on the odd-even approach, that can be used to encode cardinality constraints. We showed that the CNF encoding of the 4-Odd-Even Selection Network is smaller than the 2-column version. We extended the encoding by applying Direct Cardinality Networks [3] for sufficiently small input. The new encoding was compared with the selected state-of-the-art encodings based on comparator networks, adders and binary decision diagrams as well as with two popular general constraints solvers. The experimental evaluation shows that the new encoding yields better speed-up and overall runtime in the SAT-solver performance.

We have also showed (here, and in the previous chapter) how to generalize the multi-column networks for any number of columns. The conclusion is that the odd-even algorithm is much easier to implement than the pairwise algorithm.

# Chapter 7

# Encoding Pseudo-Boolean Constraints



To replicate the success of our algorithm from the previous chapter in the field of PB-solving, we implemented the 4-Odd-Even Selection Network in MINISAT+ and removed the 2-Odd-Even Sorting Network from the original implementation [34]. In Chapter 6 we have showed a top-down, divide-and-conquer algorithm for constructing the 4-Odd-Even Selection Network. The difference in our new implementation is that we build our network in a bottom-up manner, which results in the easier and cleaner implementation.

We apply a number of optimization techniques in our solver, some based on the work of other researchers. In particular, we use optimal base searching algorithm based on the work of Codish et al. [28] and ROBDD structure [2] instead of BDDs for one of the encodings in MINISAT+. We also substitute sequential search of minimal value of the goal function in optimization problems with binary search similarly to Sakai and Nabeshima [70]. We use COMINISATPS [64] by Chanseok Oh as the underlying SAT-solver, as it has been observed to perform better than the original MINISAT [33] for many instances.

We experimentally compare our solver with other state-of-the-art general constraints solvers like PBLIB [67] and NAPS [70] to prove that our techniques are good in practice. There have been organized a series of Pseudo-Boolean Evaluations [56] which aim to assess the state-of-the-art in the field of PB-solvers. We use the competition problems from the PB 2016 Competition as a benchmark for the solver proposed in this chapter.

## 7.1 System Description

### 7.1.1 4-Way Merge Selection Network

It has already been observed that using selection networks instead of sorting networks is more efficient for the encoding of constraints [27]. This fact has been successfully used in encoding cardinality constraints, as evidenced, for example, by the results of this thesis. We now apply this technique to PB-constraints. Here we describe the algorithm for constructing a bottom-up version of the 4-Odd-Even Selection Network.

The procedure can be described as follows. Assume $k \leq n$ and that we have the sequence of Boolean literals $\bar{x}$ of length $n$ and we want to select $k$ largest, sorted elements, then:

- If $k = 0$, there is nothing to do.
- If $k = 1$, simply select the largest element from $n$ inputs using a 1-selector of order $n$.

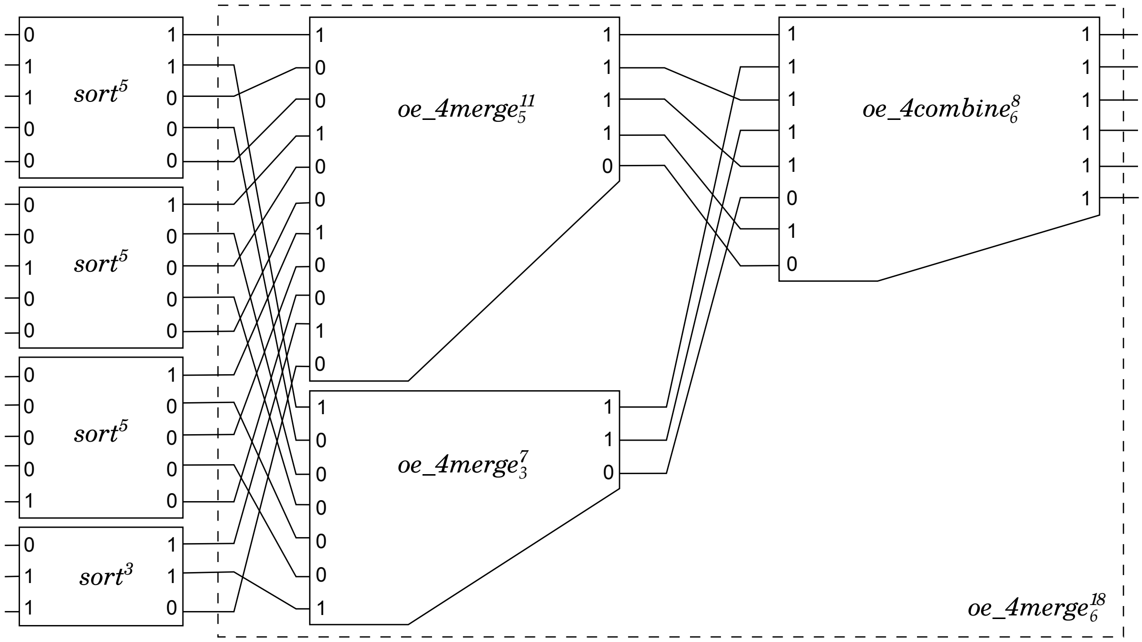



Figure 7.1: An example of 4-Odd-Even Selection Network, with $n = 18$ and $k = 6$

- If $k > 1$, then split the input into subsequences of either the same literals (of length at least 2) or sorted 5 singletons (using a $\min(5, k)$-selector of order 5). Next, sort subsequences by length, in a non-increasing order. In loop: merge each 4 (or less) consecutive subsequences into one (using 4-Odd-Even Merging Network as a sub-procedure) and select at most $k$ largest items until one subsequence remains.

**Example 7.1.** See Figure 7.1 for a schema of our selection network (where $n = 18$ and $k = 6$) which selects 6 largest elements from the input 011000010000001011. In the same figure, in dashed lines, a schema of 4-Odd-Even Merger is presented with $s = 18$, $k = 6$, $|\bar{w}| = |\bar{x}| = |\bar{y}| = 5$ and $|\bar{z}| = 3$. First, the inputs are split into two by odd and even indices, and the recursive calls are made. After that, a combine operation fixes the order of elements, to output the 6 largest ones.

In the following subsections we explain how we have extended the MiniSat+ solver to make the above process (and the entire PB-solving computation) more efficient.

### 7.1.2 Simplifying Inequality Assertions

We use the following optimization found in NaPS [70] for simplifying inequality assertions in a constraint. We introduce this concept with an example. In order to assert the constraint $a_1 l_1 + \cdots + a_n l_n \geq k$ the encoding compares the digits of the sum of the terms on the left side of the constraint with those from $k$ (in some base $\mathbf{B}$) from the right side. Consider the following example:

$$5x_1 + 7x_2 \geq 9$$

Assume that the base is $\mathbf{B} = \langle 2, 2 \rangle$. Then $9 = \langle 1, 0, 2 \rangle_{\mathbf{B}}$, but if we add 7 to both sides of the inequality:

$$7 + 5x_1 + 7x_2 \geq 16$$

then those constraints are obviously equivalent and $16 = \langle 0,0,4 \rangle_{\mathbf{B}}$. Now in order to assert the inequality we only need to assert a single output variable of the encoding of the sum of LHS coefficients (using a singleton clause). The constant 7 on the LHS has a very small impact on the size of LHS encoding. This simplification allows for the reduction of the number of clauses in the resulting CNF encoding, as well as allows better propagation.

### 7.1.3 Optimal Base Problem

We have mentioned that MINISAT+ searches for a mixed radix base such that the sum of all the digits of the coefficients written in that base, is as small as possible. In their paper [34] authors mention in the footnote that:

> *The best candidate is found by a brute-force search trying all prime numbers* $< 20$*. This is an ad-hoc solution that should be improved in the future. Finding the optimal base is a challenging optimization problem in its own right.*

Codish et al. [28] present an algorithm which scales to find an optimal base consisting of elements with values up to $1,000,000$ and they consider several measures of optimality for finding the base. They show experimentally that in many cases finding a better base leads also to better SAT solving time. We use their algorithm in our solver, but we restrict the domain of the base to prime numbers less than 50, as preliminary experiments show that numbers in the base are usually small.

### 7.1.4 Minimization Strategy

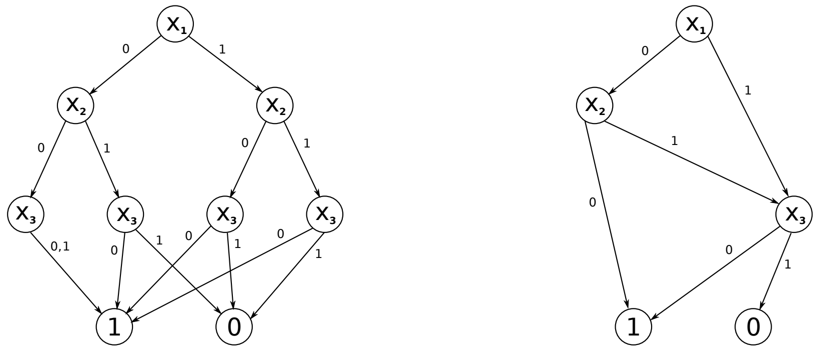
The key to efficiently solve Pseudo Boolean optimization problems is the repeated use of a SAT-solver. Assume we have a minimization problem with an objective function $f(x)$. First, without considering $f$, we run the solver on a set of constraints to get an initial solution $f(x_0) = k$. Then we add the constraint $f(x) < k$ and run the solver again. If the problem is unsatisfiable, $k$ is the optimal solution. If not, the process is repeated with the new (smaller) candidate solution $k'$. The minimization strategy is about the choice of $k'$. If we choose $k'$ as reported by the SAT-solver, then we are using the so called *sequential* strategy – this is implemented in MINISAT+.

Sakai and Nabeshima [70] propose the *binary* strategy for the choice of new $k'$. Let $k$ be the best known goal value and $l$ be the greatest known lower bound, which is initially the sum of negative coefficients of $f$. After each iteration, new constraint $p \Rightarrow f(x) < \lfloor (k(q-1)+l)/q \rfloor$ is added, where $p$ is a fresh variable (assumption) and $q$ is a constant (we set $q = 3$ as default value). Depending on the new SAT-solver answer, $\lfloor (k(q-1)+l)/q \rfloor$ becomes the new upper or lower bound (in this case $p$ is set to 0), and the process begin anew.

In our implementation we use binary strategy until the difference between the upper and lower bounds of the goal value is less than 96, then we switch to the sequential strategy. We do this in order to avoid a situation when a lot of computation is needed when searching for UNSAT answers, which could arise if only binary strategy was used. This was also observed in [70] and the authors have used it as a default strategy. Moreover, they propose to alternate between binary and sequential strategy depending on the SAT-solver answer in a given iteration.

### 7.1.5 ROBDDs Instead of BDDs

One of the encodings of MINISAT+ is based on *Binary Decision Diagrams* (BDDs). We have improved the implementation of this encoding by using the more recent *Reduced Ordered BDD*




(a) BDD (b) ROBDD

Figure 7.2: Construction of a BDD (left) and a ROBDD (right) for $2x_1 + 3x_2 + 5x_3 \leq 6$

(ROBDD) construction [2]. Now ROBDD is used to create a DAG representation of a constraint. One of the advantages of ROBBDs is that we can reuse nodes in the ROBDD structure, which results in a smaller encoding. This concept is illustrated in Figure 7.2, which shows an example from [2] of BDD and ROBDD for the PB-constraint $2x_1 + 3x_2 + 5x_3 \leq 6$. Two reductions are applied (until fix-point) for obtaining ROBDD: removing nodes with identical children and merging isomorphic subtrees. This was already explained in Section 3.3. See [2] for a more detailed example.

### 7.1.6 Merging Carry Bits

In the construction of interconnected sorters in MINISAT+ carry bits from one sorter are being fed to the next sorter. In the footnote in [34] it is mentioned that:

> *Implementation note: The sorter can be improved by using the fact that the carries are already sorted.*

We go with this suggestion and use our merging network to merge the carry bits with a sorted digits representation instead of simply forwarding the carry bits to the inputs of the next selection network.

### 7.1.7 SAT-solver

The underlying SAT-solver of MINISAT+ is MINISAT [33] created by Niklas Eén and Niklas Sörensson. It has served as an extension to many new solvers, but it is now quite outdated. We have integrated a solver called COMINISATPS by Chanseok Oh [64], which have collectively won six medals in SAT Competition 2014 and Configurable SAT Solver Challenge 2014. Moreover, the modification of this solver called MAPLECOMSPS won the Main Track category of SAT Competition 2016[1].

## 7.2 Experimental Evaluation

Our extension of MINISAT+ based on the features explained in this chapter, which we call KP-MINISAT+, is available online[2]. It should be linked with a slightly modified COMINISATPS, also

[1] See http://baldur.iti.kit.edu/sat-competition-2016/
[2] See https://github.com/karpiu/kp-minisatp

available online[3]. Detailed results of the experimental evaluation are also available online[4].

The set of instances we use is from the Pseudo-Boolean Competition 2016[5]. We use instances with linear, Pseudo-Boolean constraints that encode either decision or optimization problems. To this end, three categories from the competition have been selected:

- **DEC-SMALLINT-LIN** - 1783 instances of decision problems with small coefficients in the constraints (no constraint with sum of coefficients greater than $2^{20}$). No objective function to optimize. The solver must simply find a solution.
- **OPT-BIGINT-LIN** - 1109 instances of optimization problems with big coefficients in the constraints (at least one constraint with a sum of coefficients greater than $2^{20}$). An objective function is present. The solver must find a solution with the best possible value of the objective function.
- **OPT-SMALLINT-LIN** - 1600 instances of optimization problems. Like OPT-BIGINT-LIN but with small coefficients (as in DEC-SMALLINT-LIN) in the constraints.

We compare our solver (abbreviated to **KP-MS+**) with two state-of-the-art general purpose constraint solvers. First is the PBSOLVER from PBLIB ver. 1.2.1, by Tobias Philipp and Peter Steinke [67] (abbreviated to **PBLib** in the results). This solver implements a plethora of encodings for three types of constraints: at-most-one, at-most-k (cardinality constraints) and Pseudo-Boolean constraints. The PBLIB automatically normalizes the input constraints and decides which encoder provides the most effective translation. We have launched the program `./BasicPBSolver/pbsolver` of PBLIB on each instance with the default parameters.

The second solver is NAPS ver. 1.02b by Masahiko Sakai and Hidetomo Nabeshima [70] which implements improved ROBDD structure for encoding constraints in band form, as well as other optimizations. This solver is also built on the top of MINISAT+. NAPS won two of the optimization categories in the Pseudo-Boolean Competition 2016: OPT-BIGINT-LIN and OPT-SMALLINT-LIN. We have launched the main program of NAPS on each instance, with parameters `-a -s -nm`.

We also compare our solver with the original MINISAT+ in two different versions, one using the original MINISAT SAT-solver and the other using the COMINISATPS (the same as used by us). We label these **MS+** and **MS+COM** in the results. We prepared results for **MS+COM** in order to show that the advantage of using our solver does not come simply from changing the underlying SAT-solver.

We have launched our solver on each instance, with parameters `-a -s -cs -nm`, where `-cs` means that in experiments the solver used just one encoding technique, the 4-Odd-Even Selection Networks combined with a direct encoding of small sub-networks.

All experiments were carried out on the machines with Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz. Timeout limit is set to 1800 seconds and memory limit is 15 GB, which are enforced with the following commands: `ulimit -Sv 15000000` and `timeout -k 20 1809 <solver> <parameters> <instance>`.

We would like to note that we also wanted to include in this evaluation the winner of DEC-SMALLINT-LIN category, which is the solver based on the *cutting planes* technique, but we refrained from that for the following reason. We have not found the source code of this solver and the only working version found in the author's website[6] is a binary file without any documentation.

[3] See https://github.com/marekpiotrow/cominisatps
[4] See http://www.ii.uni.wroc.pl/%7ekarp/pos/2018.html
[5] See http://www.cril.univ-artois.fr/PB16/
[6] See http://www.csc.kth.se/%7eelffers/

| solver | DEC-SMALLINT-LIN | | | OPT-BIGINT-LIN | | | OPT-SMALLINT-LIN | | |
|---|---|---|---|---|---|---|---|---|---|
| | Sat | UnSat | cpu | Opt | UnSat | cpu | Opt | UnSat | cpu |
| KP-MS+ | 432 | 1049 | 647041 | 359 | 72 | 1135925 | 808 | 86 | 1289042 |
| NaPS | 348 | 1035 | 816725 | 314 | 69 | 1314536 | 799 | 81 | 1330536 |
| PBLib | 349 | 922 | 1104508 | – | – | – | 691 | 56 | 1611247 |
| MS+ | 395 | 951 | 895774 | 149 | 71 | 1647958 | 715 | 73 | 1515166 |
| MS+COM | 428 | 1027 | 703269 | 174 | 71 | 1609433 | 734 | 71 | 1491269 |

Table 7.1: Number of solved instances of PB-competition benchmarks.

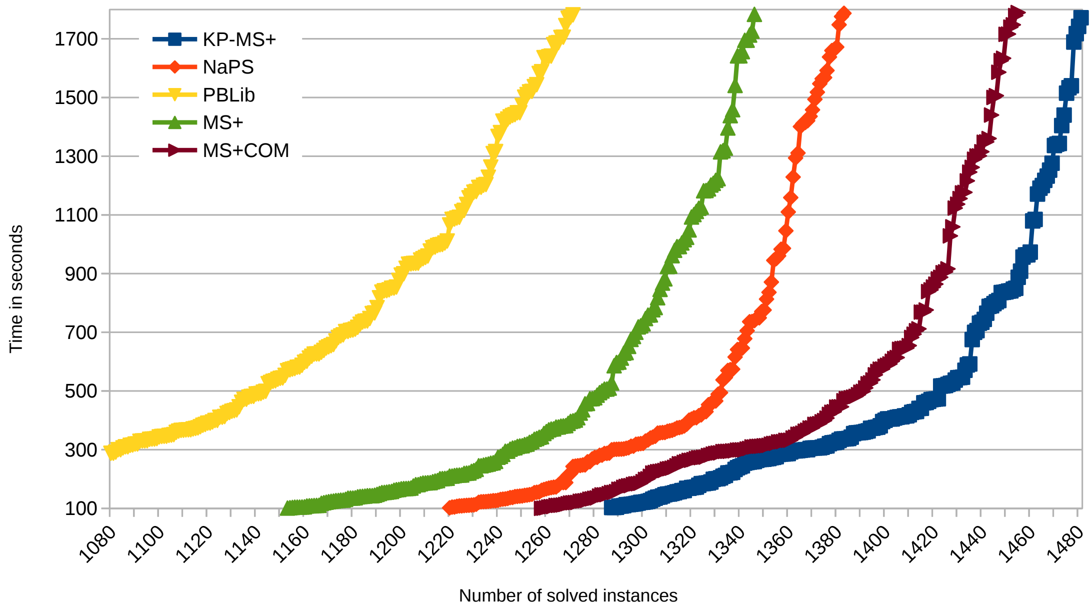



Figure 7.3: Cactus plot for DEC-SMALLINT-LIN division

Because of this, we were unable to get any meaningful results of running aforementioned program on optimization instances.

In Table 7.1 we present the number of solved instanced for each competition category. **Sat**, **UnSat** and **Opt** have the usual meaning, while **cpu** is the total solving time of the solver over all instances of a given category. Results clearly favor our solver. We observe significant improvement in the number of solved instances in comparison to NAPS in categories DEC-SMALLINT-LIN and OPT-BIGINT-LIN. The difference in the number of solved instances in the OPT-SMALLINT-LIN category is not so significant. Solver PBLIB had the worst performance in this evaluation. Notice that the results of PBLIB for OPT-BIGINT-LIN division is not available. This is because PBLIB is using 64-bit integers in calculations, thus could not be launched with all OPT-BIGINT-LIN instances.

Figures 7.3, 7.4 and 7.5 show cactus plots of the results, which indicate the number of solved instances within the time. We see clear advantage of our solver over the competition in the DEC-SMALLINT-LIN and OPT-BIGINT-LIN categories.

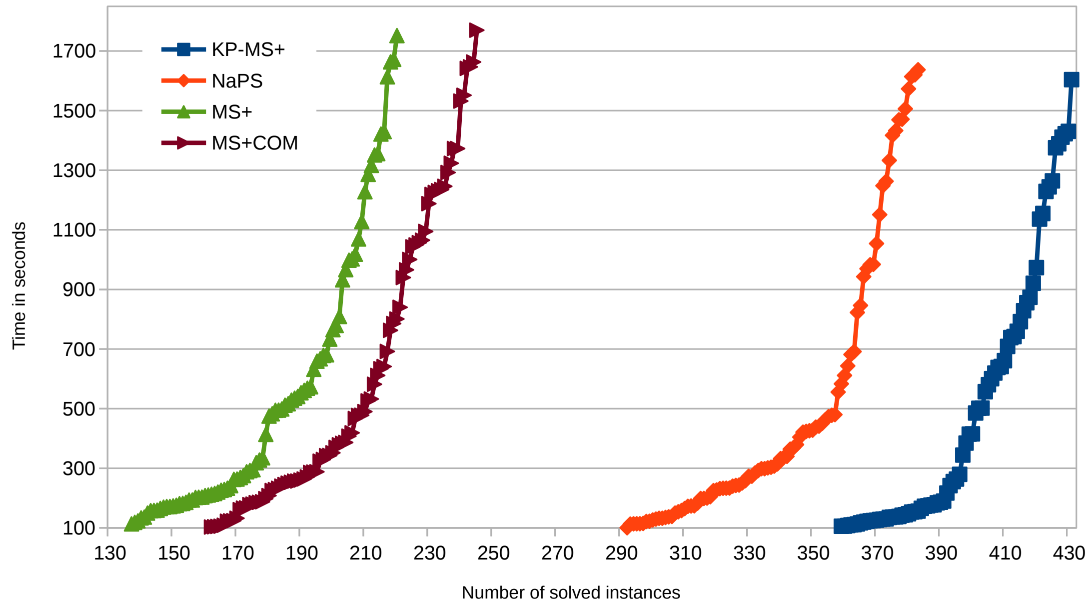



Figure 7.4: Cactus plot for OPT-BIGINT-LIN division

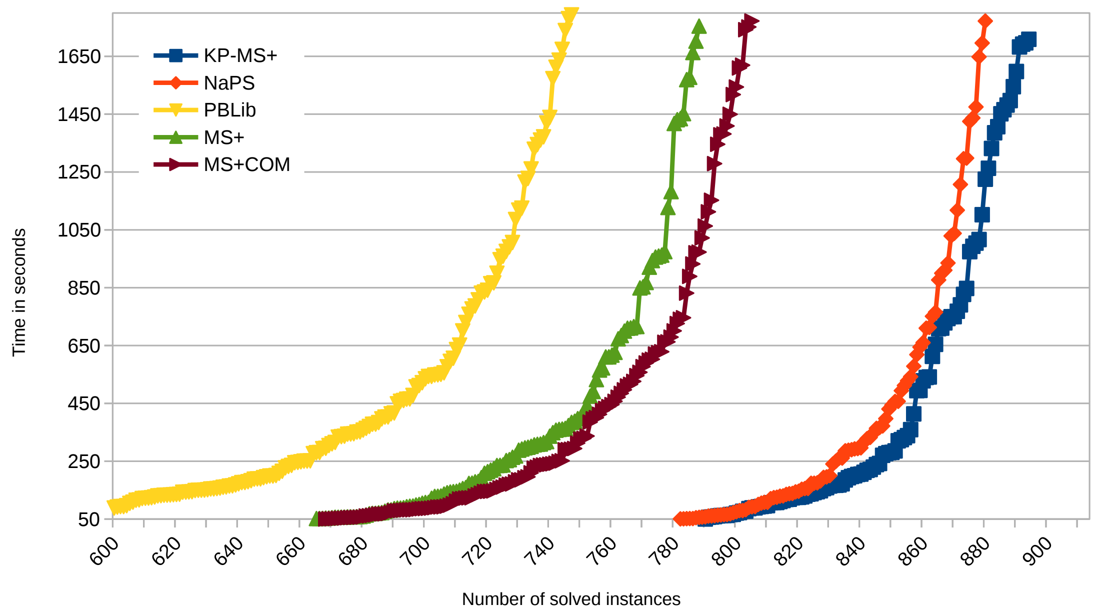


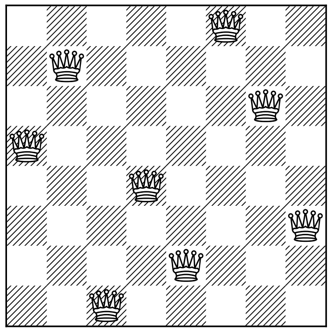

Figure 7.5: Cactus plot for OPT-SMALLINT-LIN division

## 7.3 Summary

In this chapter we proposed a new method of encoding PB-constraints into SAT based on sorters. We have extended the MiniSat+ with the 4-way merge selection algorithm and showed that this method is competitive to other state-of-the-art solutions. Our algorithm is short and easy to implement. Our implementation is modular, therefore it can be easily extracted and applied in other solvers.

# Chapter 8

# Final Remarks



This chapter concludes the thesis, as well as shows the solution to the 8-Queens Puzzle which we have been building throughout the dissertation (it is the only solution – modulo symmetry – with the property that no three queens are in a straight line!). Here we would like to give some final remarks and possibilities for future work.

We have shown new classes of networks that have efficient translations to CNFs. Encodings – of both cardinality and Pseudo-Boolean constraints – based on our 4-way Merge Selection Network are very competitive, as evidenced by the experimental evaluation presented here. Our encodings easily compete with other state-of-the-art encodings, even with the winners of recent competitions. We also prove the arc-consistency property for all encodings based on the standard encoding of generalized selection networks. This captures all the new encodings presented in this thesis, as well as all past (and future) encodings (based on sorting/selection networks).

Possibly the biggest mystery we can leave the reader with is: why encoding cardinality constraints using comparator networks is so efficient? It is in fact a very fundamental question. We can only see the empirical evidence as shown in this thesis, as well as other papers. We can compare two networks using various measures like the number of comparators, depth, the number of variables and clauses the encoding generates, or variables to clauses ratio. But all of those measures do not seem to give a conclusive answer to our question, especially if we want to collate our encodings with the ones not based on comparator networks. Arc-consistency property is important, but even with it we still cannot decisively distinguish between the top used encodings. In fact, being arc-consistent is a must if an encoding is to be competitive. It looks like another, more complex propagation property is needed in order to theoretically prove superiority of encodings based on comparator networks. Such property may have not been discovered yet.

We see, by the new constructions and the old ones referenced in the first part of this thesis, that comparator networks considered in the field are based exclusively on the odd-even or pairwise approach. Nevertheless, there are many other sorting networks proposed in the literature that could be used. It would be informative to see an empirical study on much wider collection of encodings based on networks not yet considered in practice. So a survey on this topic is a niche ready to be filled.

# Bibliography


[1] A. Aavani, D. G. Mitchell, and E. Ternovska. “New Encoding for Translating Pseudo-Boolean Constraints into SAT”. In: *SARA*. 2013.

[2] I. Abío, R. Nieuwenhuis, A. Oliveras, E. Rodríguez-Carbonell, and V. Mayer-Eichberger. “A New Look at BDDs for Pseudo-Boolean Constraints”. In: *Journal of Artificial Intelligence Research* 45 (2012), pp. 443–480.

[3] I. Abío, R. Nieuwenhuis, A. Oliveras, and E. Rodríguez-Carbonell. “A Parametric Approach for Smaller and Better Encodings of Cardinality Constraints”. In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2013, pp. 80–96.

[4] R. A. Achá and R. Nieuwenhuis. “Curriculum-based Course Timetabling with SAT and MaxSAT”. In: *Annals of Operations Research* 218.1 (2014), pp. 71–91.

[5] M. Ajtai, J. Komlós, and E. Szemerédi. “An 0 (n log n) sorting network”. In: *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*. ACM. 1983, pp. 1–9.

[6] F. A. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallah. “Generic ILP versus specialized 0-1 ILP: an update”. In: *Proceedings of the 2002 IEEE/ACM International Conference on Computer-aided Design*. ACM. 2002, pp. 450–457.

[7] C. Ansótegui and F. Manya. “Mapping Problems with Finite-Domain Variables into Problems with Boolean Variables”. In: *International Conference on Theory and Applications of Satisfiability Testing*. Springer. 2004, pp. 1–15.

[8] K. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.

[9] R. Asín, R. Nieuwenhuis, A. Oliveras, and E. Rodríguez-Carbonell. “Cardinality Networks and Their Applications”. In: *International Conference on Theory and Applications of Satisfiability Testing*. Springer. 2009, pp. 167–180.

[10] R. Asín, R. Nieuwenhuis, A. Oliveras, and E. Rodríguez-Carbonell. “Cardinality Networks: a Theoretical and Empirical Study”. In: *Constraints* 16.2 (2011), pp. 195–221.

[11] G. Audemard and L. Simon. “Predicting Learnt Clauses Quality in Modern SAT Solvers”. In: *IJCAI*. Vol. 9. 2009, pp. 399–404.

[12] O. Bailleux and Y. Boufkhad. “Efficient CNF encoding of Boolean cardinality constraints”. In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2003, pp. 108–122.

[13] O. Bailleux, Y. Boufkhad, and O. Roussel. “A Translation of Pseudo-Boolean Constraints to SAT”. In: *Journal on Satisfiability, Boolean Modeling and Computation* 2 (2006), pp. 191–200.

[14] P. Barahona, S. Hölldobler, and V.-H. Nguyen. “Efficient SAT-Encoding of Linear CSP Constraints”. In: *ISAIM*. 2014.

[15] P. Barahona, S. Hölldobler, and V.-H. Nguyen. “Representative Encodings to Translate Finite CSPs into SAT”. In: *International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems*. Springer. 2014, pp. 251–267.

[16] A. F. Barco, J.-G. Fages, E. Vareilles, M. Aldanondo, and P. Gaborit. “Open Packing for Facade-Layout Synthesis Under a General Purpose Solver”. In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2015, pp. 508–523.

[17] K. E. Batcher. “Sorting networks and their applications”. In: *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*. ACM. 1968, pp. 307–314.

[18] Y. Ben-Haim, A. Ivrii, O. Margalit, and A. Matsliah. “Perfect Hashing and CNF Encodings of Cardinality Constraints”. In: *International Conference on Theory and Applications of Satisfiability Testing*. Springer. 2012, pp. 397–409.

[19] A. Biere, M. Heule, and H. van Maaren. *Handbook of Satisfiability*. Vol. 185. IOS Press, 2009.

[20] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. “Symbolic Model Checking without BDDs”. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 1999, pp. 193–207.

[21] C. E. Blair, R. G. Jeroslow, and J. K. Lowe. “Some results and experiments in programming techniques for propositional logic”. In: *Computers & Operations Research* 13.5 (1986), pp. 633–645.

[22] N. Briot, C. Bessiere, and P. Vismara. “A Constraint-Based Approach to the Differential Harvest Problem”. In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2015, pp. 541–556.

[23] M. Büttner and J. Rintanen. “Satisfiability Planning with Constraints on the Number of Actions”. In: *ICAPS*. 2005, pp. 292–299.

[24] M.-T. Chao and J. Franco. “Probabilistic analysis of two heuristics for the 3-satisfiability problem”. In: *SIAM Journal on Computing* 15.4 (1986), pp. 1106–1118.

[25] M.-T. Chao and J. Franco. “Probabilistic analysis of a generalization of the unit-clause literal selection heuristics for the k satisfiability problem”. In: *Information Sciences: an International Journal* 51.3 (1990), pp. 289–314.

[26] J. Chen. “A New SAT Encoding of the At-Most-One Constraint”. In: *Proc. Constraint Modelling and Reformulation* (2010).

[27] M. Codish and M. Zazon-Ivry. “Pairwise Cardinality Networks”. In: *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer. 2010, pp. 154–172.

[28] M. Codish, Y. Fekete, C. Fuhs, and P. Schneider-Kamp. “Optimal Base Encodings for Pseudo-Boolean Constraints”. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2011, pp. 189–204.

[29] M. Codish, L. Cruz-Filipe, M. Frank, and P. Schneider-Kamp. “Twenty-Five Comparators is Optimal when Sorting Nine Inputs (and Twenty-Nine for Ten)”. In: *Tools with Artificial Intelligence (ICTAI), 2014 IEEE 26th International Conference on*. IEEE. 2014, pp. 186–193.

[30] S. A. Cook. “The Complexity of Theorem-Proving Procedures”. In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. ACM. 1971, pp. 151–158.

[31] M. Davis, G. Logemann, and D. Loveland. “A Machine Program for Theorem-Proving”. In: *Communications of the ACM* 5.7 (1962), pp. 394–397.

[32] M. Davis and H. Putnam. “A Computing Procedure for Quantification Theory”. In: *Journal of the ACM (JACM)* 7.3 (1960), pp. 201–215.

[33] N. Eén and N. Sörensson. “An Extensible SAT-solver”. In: *International Conference on Theory and Applications of Satisfiability Testing*. Springer. 2003, pp. 502–518.

[34] N. Eén and N. Sörensson. “Translating Pseudo-Boolean Constraints into SAT”. In: *Journal on Satisfiability, Boolean Modeling and Computation* 2 (2006), pp. 1–26.

[35] A. M. Frisch and P. A. Giannaros. “SAT Encodings of the At-Most-k Constraint. Some Old, Some New, Some Fast, Some Slow”. In: *Proc. of the Tenth Int. Workshop of Constraint Modelling and Reformulation*. 2010.

[36] A. M. Frisch, T. J. Peugniez, A. J. Doggett, and P. W. Nightingale. “Solving Non-Boolean Satisfiability Problems with Stochastic Local Search: A Comparison of Encodings”. In: *SAT 2005*. Springer, 2006, pp. 143–179.

[37] I. P. Gent and P. Nightingale. “A new encoding of AllDifferent into SAT”. In: *International Workshop on Modelling and Reformulating Constraint Satisfaction*. 2004, pp. 95–110.

[38] I. P. Gent, R. W. Irving, D. F. Manlove, P. Prosser, and B. M. Smith. “A Constraint Programming Approach to the Stable Marriage Problem”. In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2001, pp. 225–239.

[39] M. T. Goodrich. “Zig-zag sort: A Simple Deterministic Data-Oblivious Sorting Algorithm Running in O(n log n) Time”. In: *Proceedings of the Forty-sixth Annual ACM Symposium on Theory of Computing*. ACM. 2014, pp. 684–693.

[40] J. Håstad. “Some optimal inapproximability results”. In: *Journal of the ACM (JACM)* 48.4 (2001), pp. 798–859.

[41] D. S. Johnson. “Approximation Algorithms for Combinatorial Problems”. In: *Journal of Computer and System Sciences* 9.3 (1974), pp. 256–278.

[42] D. Justeau-Allaire, P. Birnbaum, and X. Lorca. “Unifying Reserve Design Strategies with Graph Theory and Constraint Programming”. In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2018, pp. 507–523.

[43] H. Karloff and U. Zwick. “A 7/8-Approximation Algorithm for MAX 3SAT?” In: *FOCS*. IEEE. 1997, p. 406.

[44] M. Karpiński. “Encoding cardinality constraints using standard encoding of generalized selection networks preserves arc-consistency”. In: *Theoretical Computer Science* 707 (2018), pp. 77–81.

[45] M. Karpiński and M. Piotrów. “Competitive Competitive Sorter-based Encoding of PB-Constraints into SAT”. In: *Proceedings of Pragmatics of SAT 2018*.

[46] M. Karpiński and M. Piotrów. “Smaller Selection Networks for Cardinality Constraints Encoding”. In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2015, pp. 210–225.

[47] M. Karpiński and M. Piotrów. “Encoding Cardinality Constraints using Generalized Selection Networks”. In: *arXiv preprint arXiv:1704.04389* (2017).

[48] M. Karpiński and M. Piotrów. “Encoding Cardinality Constraints using Multiway Merge Selection Networks”. In: *Accepted to Constraints* (2019).

[49] W. Klieber and G. Kwon. “Efficient CNF Encoding for Selecting 1 from N Objects”. In: *Proc. International Workshop on Constraints in Formal Verification*. 2007.

[50] D. E. Knuth. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1998.

[51] P. Kučera, P. Savický, and V. Vorel. “A Lower Bound on CNF Encodings of the At-Most-One Constraint”. In: *Theoretical Computer Science* (2018).

[52] K. Kutzkov and D. Scheder. “Using CSP To Improve Deterministic 3-SAT”. In: *arXiv preprint arXiv:1007.1166* (2010).

[53] D. Le Berre and A. Parrain. “The SAT4J library, Release 2.2, System Description”. In: *Journal on Satisfiability, Boolean Modeling and Computation* 7 (2010), pp. 59–64.

[54] D.-L. Lee and K. E. Batcher. “A Multiway Merge Sorting Network”. In: *IEEE Transactions on Parallel and Distributed Systems* 6.2 (1995), pp. 211–215.

[55] X. Lorca, C. Prud’Homme, A. Questel, and B. Rottembourg. “Using Constraint Programming for the Urban Transit Crew Rescheduling Problem”. In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2016, pp. 636–649.

[56] V. M. Manquinho and O. Roussel. “The First Evaluation of Pseudo-Boolean Solvers (PB’05)”. In: *Journal on Satisfiability, Boolean Modeling and Computation* 2 (2006), pp. 103–143.

[57] N. Manthey. “Towards Next Generation Sequential and Parallel SAT Solvers”. In: *KI-Künstliche Intelligenz* 30.3-4 (2016), pp. 339–342.

[58] N. Manthey and P. Steinke. “npSolver–A SAT Based Solver for Optimization Problems”. In: *Pragmatics of SAT* (2012).

[59] J. Marques-Sila and J. Planes. “Algorithms for Maximum Satisfiability using Unsatisfiable Cores”. In: *Adv. Techniques in Logic Synthesis, Optimizations and Applications*. Springer, 2011, pp. 171–182.

[60] A. Morgado, F. Heras, M. Liffiton, J. Planes, and J. Marques-Silva. “Iterative and core-guided MaxSAT solving: A survey and assessment”. In: *Constraints* 18.4 (2013), pp. 478–534.

[61] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. “Chaff: Engineering an Efficient SAT Solver”. In: *Proceedings of the 38th Annual Design Automation Conference*. ACM. 2001, pp. 530–535.

[62] V.-H. Nguyen and S. T. Mai. “A New Method to Encode the At-Most-One Constraint into SAT”. In: *Proceedings of the Sixth International Symposium on Information and Communication Technology*. ACM. 2015, pp. 46–53.

[63] V.-H. Nguyen, M. N. Velev, and P. Barahona. “Application of Hierarchical Hybrid Encodings to Efficient Translation of CSPs to SAT”. In: *Tools with Artificial Intelligence (ICTAI), 2013 IEEE 25th International Conference on*. IEEE. 2013, pp. 1028–1035.

[64] C. Oh. “Improving SAT Solvers by Exploiting Empirical Characteristics of CDCL”. PhD thesis. New York University, 2016.

[65] I. Parberry. “The Pairwise Sorting Network”. In: *Par. Processing Letters* 2.02n03 (1992), pp. 205–211.

[66] M. S. Paterson. “Improved Sorting Networks with O(log N) Depth”. In: *Algorithmica* 5.1-4 (1990), pp. 75–92.

[67] T. Philipp and P. Steinke. “PBLib–A Library for Encoding Pseudo-Boolean Constraints into CNF”. In: *International Conference on Theory and Applications of Satisfiability Testing*. Springer. 2015, pp. 9–16.

[68] N. Pippenger. “Selection Networks”. In: *SIAM Journal on Computing* 20.5 (1991), pp. 878–887.

[69] S. D. Prestwich. “CNF Encodings”. In: *Handbook of Satisfiability* 185 (2009), pp. 75–97.

[70] M. Sakai and H. Nabeshima. “Construction of an ROBDD for a PB-Constraint in Band Form and Related Techniques for PB-Solvers”. In: *IEICE TRANSACTIONS on Information and Systems* 98.6 (2015), pp. 1121–1127.

[71] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, 1998.

[72] A. Schutt, T. Feydy, P. J. Stuckey, and M. G. Wallace. “Why Cumulative Decomposition Is Not as Bad as It Sounds”. In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2009, pp. 746–761.

[73] M. Sheeran and G. Stålmarck. “A Tutorial on Stålmarck’s Proof Procedure for Propositional Logic”. In: *Formal Methods in System Design* 16.1 (2000), pp. 23–58.

[74] C. Sinz. “Towards an Optimal CNF Encoding of Boolean Cardinality Constraints”. In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2005, pp. 827–831.

[75] N. Tamura, A. Taga, S. Kitagawa, and M. Banbara. “Compiling finite linear CSP into SAT”. In: *Constraints* 14.2 (2009), pp. 254–272.

[76] T. Walsh. “SAT v CSP”. In: *International Conference on Principles and Practice of Constraint Programming*. Springer. 2000, pp. 441–456.

[77] J. P. Warners. “A linear-time transformation of linear inequalities into conjunctive normal form”. In: *Information Processing Letters* 68.2 (1998), pp. 63–69.

[78] M. Zazon-Ivry and M. Codish. “Pairwise Networks are Superior for Selection”. In: (2012).

[79] L. Zhao, Z. Liu, and Q. Gao. “An efficient multiway merging algorithm”. In: *Science in China Series E: Technological Sciences* 41.5 (1998), p. 543.