Algorithms for Linearly Ordered Boolean Formulas

Christian Egeland
Department of Informatics
University of Bergen

A thesis submitted for the degree of
Master of Science

June 2016

Acknowledgements

First and foremost, I must thank my supervisor Jan Arne Telle at the Department of Informatics at the University of Bergen, for all his help and guidance throughout the process of writing this thesis.

I would also like to thank everyone in the algorithms group for their support and motivation in my work. To Torstein Jarl Strømme, Vigdis Sveinsdottir and Lars Kristian Hæhre, thank you for your help in reading and providing valuable comments. And last but certainly not least, I am thankful for my friends and family, for their support through my years of studying and through the whole process of researching and writing my thesis.

This would not have been possible without you. Thank you.

Contents

1 Introduction **1**

1.1 Background 1

1.2 New algorithms 2

1.3 Experiments 3

1.4 Car sequencing 5

1.5 Preliminary Terminology 5

1.5.1 Graph Theory 5

1.5.2 Graph Classes 5

1.5.3 Boolean Satisfiability Problem 6

1.6 Thesis structure 7

2 Interval Ordering **9**

2.1 Interval Ordering 9

2.1.1 Exact algorithm 11

2.1.2 Heuristic algorithm 11

2.1.3 Given two orders: merging algorithm 12

2.1.4 Given one order only: algorithm for q-CNF formulas 15

2.2 K-Interval Ordering 19

2.2.1 Given two orders: the Mk merging algorithm 21

2.2.2 Heuristics algorithm 21

2.3 PS-width of interval ordered boolean formulas 23

3 Experimental results **25**

3.1 Improving Mk . . . 25
3.1.1 Shifting heuristic (SH) . . . 27
3.2 Algorithms . . . 28
3.3 SAT competition . . . 29
3.3.1 Results for SAT RACE 15 . . . 30
3.3.2 Results for SAT COMP 11 . . . 31
3.3.3 Significant practical instances . . . 31
3.4 Testing linear ordering heuristics . . . 33
3.4.1 Generating instances . . . 33
3.4.2 Setup . . . 34
3.4.3 Results . . . 34

4 SAT encodings for the Car Sequencing Problem 39
4.1 Car Sequncing . . . 39
4.2 CNF Encoding . . . 41
4.2.1 Sequential Counter Encoding . . . 41
4.2.2 Capacity . . . 42
4.2.3 Link Cars and Options . . . 43
4.2.4 Complete Model . . . 43
4.2.5 Example encoding . . . 44

5 Benchmarking - Car Sequencing 45
5.1 CNF-SAT encoder . . . 45
5.2 Complete instances . . . 46
5.3 Capacity and Cardinality constraints . . . 48
5.4 Conclusion . . . 50

6 Conclusions 51
6.1 Summary . . . 51
6.2 Open Problems . . . 52

A Code repository 55

Bibliography 57

Chapter 1

Introduction

In this thesis we consider a class of propositional boolean formulas on which various problems related to satisfiability are efficiently solvable by a dynamic programming algorithm. The following chapters mainly consists of two larger parts: in the first part we describe the class of boolean formulas we are interested in and how to find them, and in the second part we investigate whether this class of formulas have any practical implications.

In the remainder of this chapter we provide a complete thesis overview starting with a brief background description of the chosen topic. In order to facilitate readability, full definitions are not provided right away, but will follow in their respective chapters.

1.1 Background

The problem of deciding whether a propositional boolean formula can be evaluated to True is called the Boolean Satisfiability Problem (SAT). SAT was the first problem shown to be NP-complete (Cook, 1971), and has since contributed to several theoretical results in the field of computer science. Besides being of great theoretical importance, it has many practical applications (e. g. planning [1], Bounded Model Checking [2]). The problem of deciding how many different ways a boolean formula can be satisfied is called the propositional model counting problem (#SAT), and if a boolean formula is encoded into conjunctive normal form, then the problem of finding the maximum number of clauses that can be satisfied in the formula is called Maximum Satisfiability Problem (MaxSAT). These two extensions provide a greater modeling power than SAT, and their associated decision problems are as well NP-complete. Given the computational complexity of these decision problems in general, we would like to address a smaller class of instances that allow us to find faster class specific algorithms. A new theoretical result given in [3] provides a polynomial time algorithm for formulas whose incidence graphs are interval bigraphs, that may be of practical use. This algorithm works in two stages: 1) Decomposing the boolean formula into an interval ordering and 2) Using the decomposition to solve the instance by dynamic programming (DP).

1.2 New algorithms

As for this thesis, we started early in 2015 to explore if stage 1) Decomposing the boolean formula into an interval ordering was of practical use, in particular 1a) whether there exists any practical algorithm deciding if the boolean formula has an interval ordering?, and 1b) whether this could be applied to some real-world problems? The known exact algorithms for stage 1 are polynomial but too slow or too complicated, and as a consequence a Greedy Heuristic (GH) for this problem was given [3] and tested on artificially generated instances. Deciding if a formula has an incidence graph, which is an interval bigraph, amounts to finding a linear ordering of its clauses and variables, a so-called interval order. Another approach to make 1a more practical is to reduce the problem complexity by assuming we are given two separate linear orders for the clauses and variables of a boolean formula and ask if there is a way to merge them together into an interval ordering. The first contribution of this thesis, is giving a linear time algorithm (MIO, see section 2.1.3) solving this problem. This result is then extended by an algorithm (CMIO, see section 2.1.4) that takes a linear ordering of the variables only, and finds a clause ordering such that they can be merged into an interval ordering, provided there exists an interval ordering respecting this variable ordering.

In late 2015 a new paper [4] extended the results of [3] to k-interval orders and gave a fast algorithm merging to minimum k-interval order (Mk) given two separate linear orders. When merging into a linear order where $k = 0$, we have an interval order thus encompassing the previous merging algorithm. Considering that the value of k will be strongly affected by the two linear orders being merged, we give a new heuristic algorithm, the Barycenter heuristic algorithm (BH), that produces the input to Mk: i. e. two separate linear orders. Presumably, when merged by Mk, the resulting ordering will have a *k-value* not too far from its optimum. We have also created a new algorithm called the Shifting heuristic (SH), which optimizes the output of Mk. See section 1.3 for a short description of SH. These new algorithms is our attempt to make part 1a feasible and giving us the tools we need to answer part 1b. After the introduction of these algorithms we continue with an empirical study. A simple overview of these algorithms is provided in the table below, describing their input, output values and where they originated from. The symbols σ and ρ indicates a linear ordering on the variables and clauses in a CNF formula.

Algorithm	Input	Output	Origin
MIO	$G = (var, cla, E)$, $\sigma(var)$ and $\rho(cla)$	$\sigma(var \cup cla)$	New in this thesis
CMIO	$G = (var, cla, E)$, $\sigma(var)$	$\sigma(var \cup cla)$	New in this thesis
GH	$G = (var, cla, E)$	$\sigma(var \cup cla)$	First given in paper [3]
Mk	$G = (var, cla, E)$, $\sigma(var)$ and $\rho(cla)$	$\sigma(var \cup cla)$ and k-value	First given in paper [4]
BH	$G = (var, cla, E)$, $\sigma(var)$ and $\rho(cla)$	$\sigma(var)$ and $\rho(cla)$	First given in [5], but was independently discovered in this thesis.
SH	$G = (var, cla, E)$ and $\sigma(var \cup cla)$	$\sigma(var \cup cla)$ and k-value	New in this thesis

Table 1.1: Overview of the algorithms presented in this thesis. Various combinations of some of these algorithms are used in the experimental part for computing a total linear ordering and its k-value of a CNF formula.

1.3 Experiments

In the experimental part of this thesis, we compare the various algorithms presented in this thesis by applying them to a broad variation of boolean formulas, where the primary goal is to minimize k, leading us towards classes of problems that have a nice linear structure. For the first experiment we benchmark problem instances taken from different SAT competitions. These competitions have many practical problem instances, allowing us to test a large amount of instances for linearity (i. e. finding a linear ordering for which k-value is low). We then continue with comparing GH+Mk with BH+Mk on the same artificially constructed instances used in [3], making it an ideal test to see how well they both minimize k knowing the optimal value is zero. The BH algorithm is an iterative procedure that (seems to) improve the *k-value* in a linear ordered CNF formula progressively by the number of iterations and approach a local optimum. Given this observation, we run BH multiple times varying the number of iterations for each tested instance. GH is an algorithm designed specifically for creating a linear ordering on CNF formulas having an interval ordering, and as a result, we do not expect BH+Mk to outperform GH on the artificially constructed instances from [3], but certainly perform reasonably well on these instances. Based on early results from running experiments with Mk, we discovered a negative consequence with a certain bias in the merging policy. This problem lead us to a simple shifting heuristic (SH), that given an Mk-merged ordering, SH shifts clauses based on a symmetry property and thereby improves the linear decomposition. We have included additional tests using the combinations BH+Mk+SH and GH+Mk+SH, as they are able to lower the *k-value* even further. We have provided a flowchart in figure 1.1 explaining how the experiments are conducted.

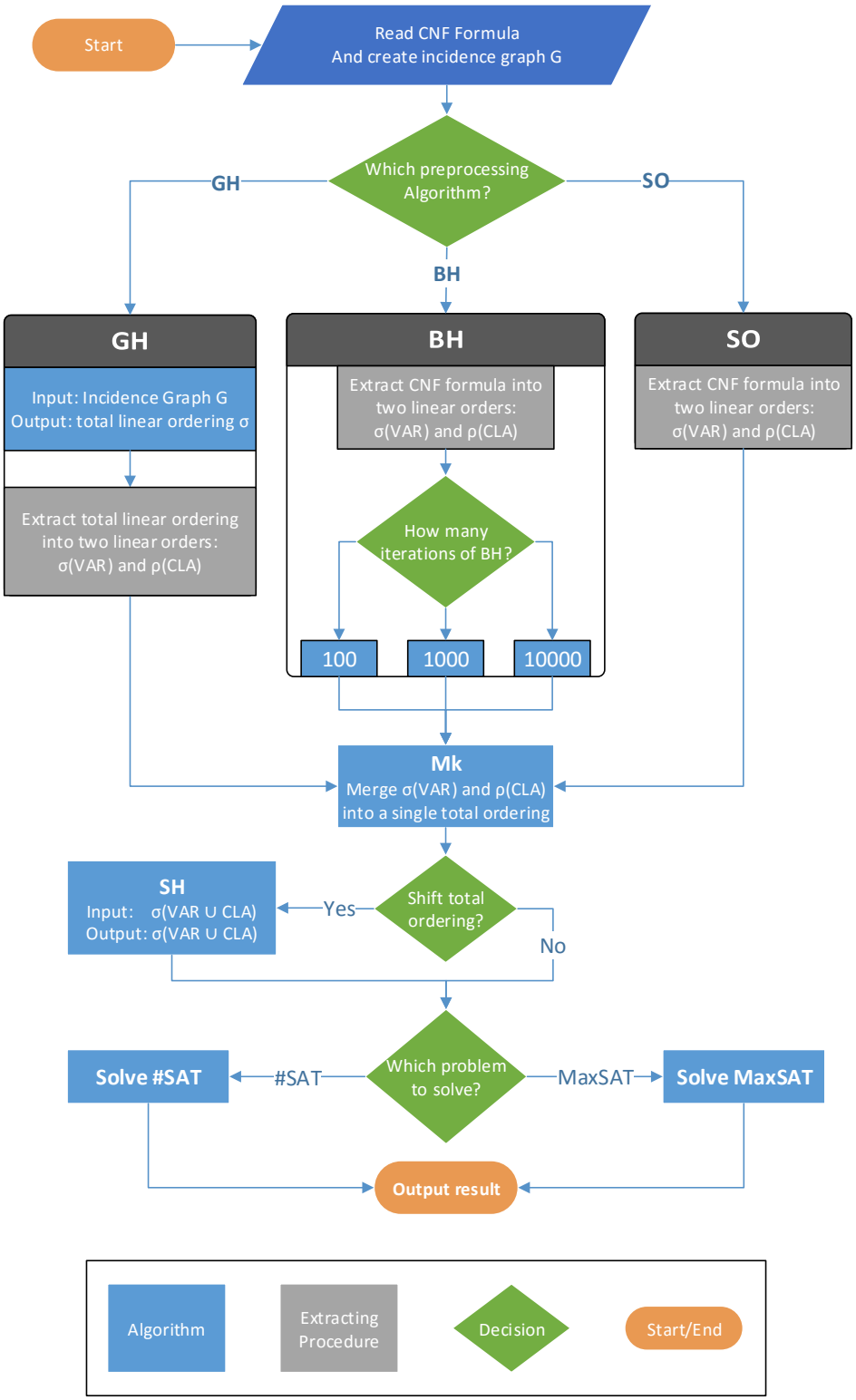

Figure 1.1: Flowchart displaying algorithms used for benchmarking throughout this thesis.

1.4 Car sequencing

After benchmarking instances from the SAT competitions and the artificially constructed CNF formulas that have an interval ordering, we go into a deeper analysis of a particular problem called Car Sequencing. We follow the paper [6] and describe how they encode a CNF formula from an instance of the car sequencing problem. After providing this description we create CNF formulas for car sequencing instances and run the same benchmarking algorithms as for the SAT competitions and artificially constructed CNF formulas. After this experimental part, we conclude the thesis with the contributed results, as well as listing questions open for further investigation.

1.5 Preliminary Terminology

For the convenience of the reader we define the general terminology used throughout this thesis for easy reference. More specific terminology is introduced in the appropriate chapters. Most of the notation and definitions in this thesis follows a standard mathematical description.

1.5.1 Graph Theory

Graph A graph G is a pair (V, E) where V is the set of *vertices* and E is the set of edges in G. An edge e is a binary relationship between two *vertices* u and v in G. Two *vertices* u and v are adjacent if there $\exists e \in E$ such that u and v are the endpoints of e.

Simple directed graph A graph is simple if it there are at most one edge between any pair of vertices, and it is directed if each edge has an direction, i. e. for a vertex x and a vertex y, the edge xy is an directed edge from x to y.

Subgraph A subgraph G' is the graph obtained by taking a subset of vertices and edges from G, denoted $G' \subseteq G$.

Induced subgraph An induced subgraph H, of a graph G, is the graph obtained by taking any $S \subseteq V(G)$, and for every pair of vertices $x, y \in S$, if $xy \in E(G)$, then $xy \in E(H)$. This induced subgraph H is denoted by $G[S]$.

1.5.2 Graph Classes

As it is very unpractical to work with all graphs together, we separate them into classes. We would then like to find more efficient algorithms for the specific classes.

Bipartite Graph Bipartite graphs (bigraphs) are a class of graphs where there exists a partition of the *vertices* into two disjoint sets A and B such that every edge in G is adjacent to some vertex $u \in A$ and some vertex $v \in B$. Equivalently, a bipartite graph is a graph that is two colored, that is for any edge each endpoint has opposite colors. For any graph G we can define

a bipartite subgraph by taking a subset of vertices $A \subseteq V(G)$ and keeping only edges which have an endpoint to a vertex in A and another in $V(G) \backslash A = \bar{A}$. We denote this bipartization of the graph G by $G[A, \bar{A}[$.

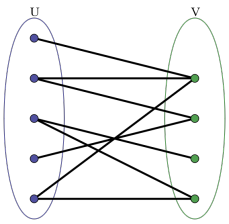

Figure 1.2: Bipartite graph

Interval Graph We say that a graph is an interval graph if we can assign a closed interval I_v on the real line for each vertex $v \in V(G)$ such that for any two vertex intervals I_u and I_v they intersect if and only if $uv \in E(G)$. An interval graph with its interval representation is illustrated in the figure below. See chapter 2 and figure 2.3 for an example of an interval bigraph.

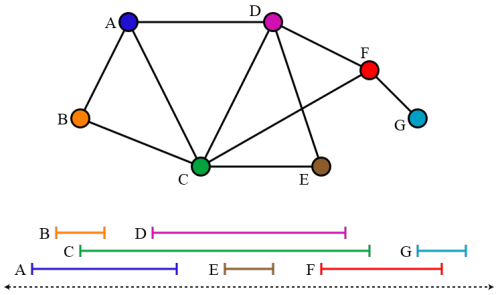

Figure 1.3: An interval graph with its interval representation

1.5.3 Boolean Satisfiability Problem

Propositional Boolean Formula A propositional boolean formula is a well formed mathematical expression consisting of variables, parentheses and the operators AND, OR and NOT. The variables can only take on boolean values, true or false.

SAT The boolean satisfiability problem, asks whether we can assign boolean values to variables in a propositional boolean formula such that the formula is evaluated to true. In this thesis we will only consider SAT encoded into CNF.

CNF Formula A propositional boolean formula in conjunctive normal form (CNF) consist of conjunction of clauses, where each clause is a disjunction of literals contained in parenthesis. A literal is a variable x or a negated variable $\neg x$. Example of a simple formula in CNF: $(x_1 \vee \neg x_2) \wedge (x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_3)$. We let cla denote the clauses and var denote the variables

in the CNF formula. In this thesis we will only consider boolean CNF formulas encoded in the DIMACS CNF format.

DIMACS CNF format The DIMACS CNF format is a widely used format for encoding CNF formulas in text files.

The input file usually starts with comments, starting with the letter c. A line starting with p defines the number of variables and clauses. Each of the next lines describes a clause: a positive integer corresponds to a positive literal, while a negative integer corresponds to a negative literal. Each of the clause lines ends with zero.

```
c A Sample .cnf file with 6 variables and 5 clauses.
p cnf 6 5
1 -2 3 0
-3 4 6 0
2 6 -1 0
-5 3 -4 0
5 -2 -6 0
```

This file encodes the following CNF formula:

$(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_3 \vee x_4 \vee x_6) \wedge (x_2 \vee x_6 \vee \neg x_1) \wedge (\neg x_5 \vee x_3 \vee \neg x_4) \wedge (x_5 \vee \neg x_2 \vee \neg x_6)$

Incidence graph of a CNF formula An incidence graph of a CNF formula F is the bipartite graph $I(F) = (V, E)$ where $V = cla(F) \cup var(F)$ and for a variable $x \in var(F)$ and a clause $c \in cla(F)$, then if $x \in c$ we have $xc \in E$.

q-CNF formula A q-CNF formula is a CNF formula where each clause has exactly q literals.

MaxSAT The MaxSAT problem asks what the maximum number of clauses is in a CNF formula that can be satisfied by some assignment to the variables.

#SAT The #SAT problem asks how many distinctive assignments satisfies a CNF formula.

1.6 Thesis structure

The reader should now be familiar with some of the terminology used throughout this thesis. In chapter 2, we first introduce interval orderings for CNF formulas and explain why such orderings are interesting. Secondly, we relax the properties of interval orderings to cover more CNF formulas, and lastly, we present algorithms finding such orderings. Next, in chapter 3, we apply the algorithms presented in chapter 2 to a wide range of SAT instances with the objective of trying to find real world problems having close to linear structure. Based on the test results we provide an improvement to these algorithms in chapter 3. In chapter 4, we go into depth of a particular problem called Car Sequencing. The whole chapter describes how to encode a car-sequencing problem instance into an equivalent boolean CNF formula, and in chapter 5 we measure how linear these formulas are with

respect to interval orderings. Finally, in chapter 6, a summary of the results and open problems is provided.

Chapter 2

Interval Ordering

In this chapter we define *(k-)interval ordering* for CNF formulas and give algorithms finding such an ordering if they exist, both with and without certain given linear orders on the variables and/or on the clauses. The major advantage of finding such structural properties in a CNF formula is that we can use it to apply efficient algorithms solving these instances. In section 2.1 we provide a formal definition of an interval ordering and how such orderings may be found under various restrictions. In section 2.2 we extend the definition of an interval ordering to a k-interval ordering, where the k-value defines a class border (i. e. the higher the k-value is, the larger the class is. CNF formulas with an interval ordering have k-value = 0). In section 2.3 we provide a full proof for an implicit result in [3] for CNF formulas having an interval ordering.

2.1 Interval Ordering

Definition 2.1. A CNF formula has an interval ordering if it is possible to linearly order (ordering of the elements is indicated by $<$) the variables and clauses such that for any variable x appearing in clause C:

1. for any variable x' where $x < x' < C$ then x' also appears in C,
2. for any clause C' where $C < C' < x$ then x also appears C'.

Definition 2.2. A bigraph $G = (A \cup B, E)$ is an interval bigraph if we can map every vertex from $A \cup B$ into an interval on the real line such that for every $u \in A$ and $v \in B$ their interval intersects if and only if $uv \in E$. See figure 2.3.

For completeness we give a full proof of the equivalence of these two notions.

Lemma 2.3. *A CNF formula F has an interval ordering if and only if the incidence graph of F is an interval bigraph.*

Proof. $\Rightarrow$ Assume we have a set $Z = var(F) \cup cla(F)$ and an interval ordering π on Z. We can construct an interval bigraph I from π by first mapping the elements as they appear in π into the

intervals $[i, i]$. See figure 2.1 for an example. We continue the construction of I by extending the intervals of each element in the following way: If z_i represents a variable x, we extend the left endpoint of the interval until it intersects all the clauses x occurs in that appear before z_i in π. This ensures that if a clause C containing x appears before x then their interval intersects. If z_i represents a clause C, then we extend the left interval until it intersects all the variables occurring in c, appearing before z_i in π. This ensures that if x appears before C in the ordering then their interval intersects. By this procedure we add all the necessary edges in $I(F)$ to I only by extending the intervals in one direction. See figure 2.3 for an example. It is easy to see that the graph is an interval graph, so

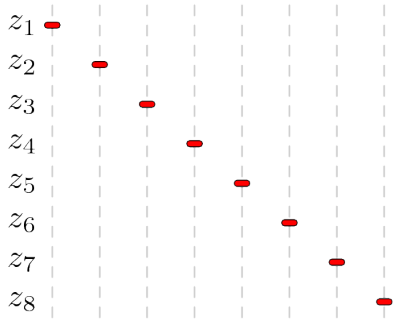

Figure 2.1: Interval graph

we need only to show that it is consistent with $I(F)$. By the construction we know that at least all edges in $I(F)$ are present, and since π is an interval ordering there cannot be any other edges than those in $I(F)$. By extending the interval to the left for a variable x, we know that all the clauses we intersects must contain x (property 2 of an interval ordering). And by extending the interval for a clause C, every variable we intersect must be a member of C (property 1 of an interval ordering). Since the graph we are interested in is a bipartization between clauses and variables, we see that I is indeed an interval bigraph.

$\Leftarrow$ To construct an interval ordering π from an interval bigraph I of $I(F)$, we order the vertices by their rightmost endpoint on its intervals in increasing order. See figure 2.2 for an example. If

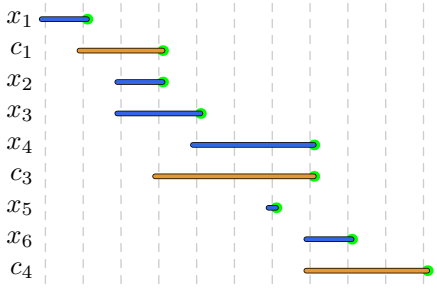

Figure 2.2: Interval ordering from graph using right endpoints (highlighter in green): $(x1, c1, x2, x3, x5, x4, c3, x6, c4)$. Note that $x_5 < x_4$.

two intervals share the same rightmost point we choose one arbitrarily. To see that this procedure

is correct, consider whenever we add a variable x to the linear ordering, then for any clause C intersecting x which is added later to the ordering, C must either end simultaneously or at a later point than x. If the algorithm then positions any variable x' between x and C, x' must intersect C, since it cannot end later than C ends or before x ends, preserving condition 1 of an interval ordering. And if we have a clause C added to the ordering, then any variable x occurring in C added after C in the linear ordering must end simultaneous or after C. Then for any other clause C' the algorithm positions in between C and x in the linear ordering, C' cannot end before C ends or after x ends, x will then intersect C' as well, preserving condition 2 of an interval ordering. This completes the proof. □

Example 2.4. $F := (x_1 \vee \bar{x_2}) \wedge (x_2 \vee \bar{x_3} \vee x_5) \wedge (x_3 \vee \bar{x_4} \vee x_5)$. Labeling the clauses by c_i where i indicates the i-th clause in F. We can easily see that the following ordering: $x_1\, c_1\, x_2\, x_3\, c_2\, c_3\, x_4\, x_5$ on F is an interval ordering by looking at the bipartized interval representation, see figure 2.3.

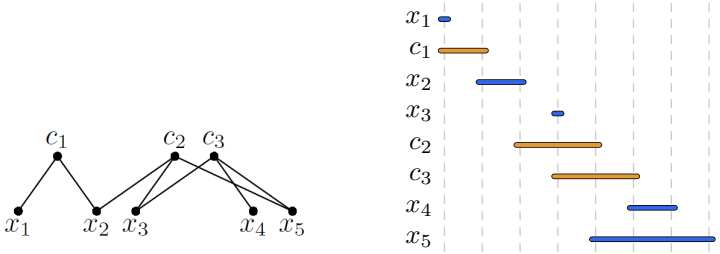

Figure 2.3: Incidence graph of F to the left and its interval intersection model to the right, showing that it is an interval bigraph.

2.1.1 Exact algorithm

Deciding whether a boolean formula can be linearly ordered into an interval ordering can be solved in time $\mathcal{O}(|V||E|^6(|V|+|E|)log|V|)$ by an algorithm published by Haiko Muller in 1997 [7]. A faster algorithm claiming a runtime $\mathcal{O}(|V|(|V|+|E|))$ was posted on arxiv in 2012 [8]. We will not go into details of these algorithms as they are polynomial but either too slow in practice or too complicated for doing an implementation as part of this master's thesis.

2.1.2 Heuristic algorithm

A greedy heuristic algorithm for finding a linear ordering on boolean formulas that hopefully is close to an interval ordering was given in [3]. The algorithm is given below as we will compare this method to other approaches later on.

Algorithm 2.1 GreedyOrder (GH)

input: $G = (V = (cla, var), E)$, a bigraph
output: σ, a linear ordering on the vertices

$L = \emptyset$, $R = V$, $i = 1$
for all $v \in V$ set $Ldegree(v) = 0$
while R is not empty **do**
  choose v: from vertices in R with max $Ldegree$ take one of smallest degree
  $\sigma(i) = v$, increment i, add v to L and remove v from R
  for all $w \in R$ with $vw \in E$ increment $Ldegree(w)$
end while

The GreedyOrder heuristic (GH) is a greedy algorithm which takes as input a bipartite graph $G = (cla, var, E)$, and outputs a linear ordering σ over all its vertices. For increasing values of i, GH assigns $\sigma(i)$ to be a vertex which has the highest number of already assigned neighbors, and among these choosing one which has the fewest non-assigned neighbors.

2.1.3 Given two orders: merging algorithm

In this section we give an algorithm which solve a more restricted problem than the exact algorithms in section 2.1.1, normally the problem where given two separate linear orderings for the clauses and variables of a boolean formula, can these be merged together into an interval ordering while keeping their respective orders? We consider this problem to be worthwhile researching as there might be boolean formulas where a natural linear ordering of its variables, and a separate linear ordering of its clauses are known. Another reason this can be valuable is that we can solve the general problem in steps (i. e. we decide orders for the variables and clauses first, and then, if possible, merge them together into an interval ordering), rather than in one complex algorithm. In this section we give an exact algorithm solving this merging problem in time $\mathcal{O}(|V| + |E|)$.

The merging algorithm takes as input: a bipartite graph $G = (cla, var, E)$ and two linear orderings, $\sigma(var) = \{x_1, x_2, ..., x_n\}$ and $\rho(cla) = \{c_1, c_2, ..., c_m\}$, and then outputs: a total linear ordering over var and clacorresponding to an interval ordering iff there exist an interval ordering respecting $\sigma(var)$ and $\rho(cla)$. Before giving the actual merging algorithm we must define a few important graph concepts and the graph structure prohibiting construction of an interval ordered boolean formula.

Definition 2.5. A forbidden structure of a graph class C is an induced subgraph H which breaks certain properties of the graph structure. We call such structures an obstruction of the graph class C. We have two such obstructions for an interval ordering arising from merging 2 orders, particularly H_1 and H_2 illustrated in figure 2.4.

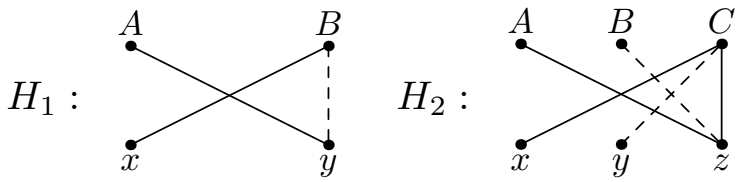

Figure 2.4: Obstructions when merging a boolean formula into an interval ordering. H_1 to the left and H_2 to the right. The variables are x, y and z, while the clauses are A, B and C. Variables and clauses are ordered such that $x < y < z$ and $A < B < C$. A solid edge represents an edge from the incidence graph, a dotted edge is non-edge in the incidence graph, while any other edge combination can occur in the incidence graph.

Lemma 2.6 (Interval Ordering Obstructions)**.** *For a bigraph $G = (A \cup B, E)$ with linear orders $\sigma(A)$ and $\rho(B)$, if either $G[S] = H_1$ or $G[S] = H_2$, for some $S \subseteq V(G)$, then $G[S]$ obstructs the properties of an interval ordering arising from merging of $\sigma(A)$ and $\rho(B)$*

Proof. First, consider H_1. By definition 2.1, we cannot order y after B since $y \in A$ but $y \notin B$. Furthermore we have that we cannot order y before B, since $x \in B$ but $y \notin B$. As a result there are no valid position for y to be ordered into. Next, consider H_2, by definition 2.1, we must order z before B since $z \in A$ but $z \notin B$. This gives us the following ordering $x < y < C$. However, given $x \in C$ and $y \notin C$, y cannot be ordered before C, giving no valid position for y to be ordered into. We note that the obstructions in Figure 2.6 and Lemma 2.7 appear in the paper [4] which was published while work on this thesis was ongoing. However, the following algorithm for merging interval orderings appear only in this thesis and was discovered independently by the author. □

Lemma 2.7. *For a bigraph $G = (cla, var, E)$, with linear orders $\sigma(var) = \{x_1, x_2, ..., x_n\}$ and $\rho(cla) = \{c_1, c_2, ..., c_n\}$, there exist an Interval Ordering of the vertices arising from merging $\sigma(var)$ and $\rho(cla)$ if and only if G does not contain any of the obstructions from lemma 2.6.*

Proof. $\Rightarrow$ By lemma 2.6 there cannot exist any such obstruction.
$\Leftarrow$ If the bipartite graph G does not induce any of the obstruction from lemma 2.6, we can merge $\sigma(var)$ and $\rho(cla)$ together to form an interval ordering by the merging algorithm below. The algorithm orders each variable from $\sigma(var)$ into $\rho(cla)$, while respecting the order $\sigma(var)$.

Algorithm 2.2 MergeIntervalOrder

input: $G = (cla, var, E)$, $\sigma(A) = \{x_1, x_2, ..., x_n\}$ and $\rho(B) = \{c_1, c_2, ..., c_m\}$
output: Interval ordering on $var \cup cla$ respecting $\sigma(var)$ and $\rho(cla)$ if one exist.

Merge variables into $\rho(B)$ preserving condition 1:
for $j := 1$ to n **do**
Insert x_j into the lowest valid position in $\rho(B)$ by the following rules:

- x_j must be ordered after x_{j-1}
- x_j must be ordered after every clause C containing any variable x_i where $i < j$ for which $x_j \notin C$ (Condition 1 of interval ordering)

end for

Check for violation of condition 2:
for all $x_j \in \rho(B)$ **do**
Let A be the leftmost clause containing x_j
if $\exists$ a clause B where $A < B < x_j$ and $x_j \notin B$ **then**
return fail
end if
end for

return $\rho(B)$

Correctness: If the algorithm succeeds we clearly have an interval ordering, since both conditions are preserved. (i. e. the first loop preserves the first condition, while the second loop verifies that the second condition holds as well). However, if the algorithm fails we have ordered at least one variable x_j too far to the right in the ordering, violating condition 2. We now prove that we cannot move x_j further left to fix this violation. Without loss of generality, let z be the leftmost variable in the ordering for which the algorithm fails. We then know that for some clause A and B, where $A < B < z$, z appears in A, but not in B, violating condition 2. We use this property to show that this gives us at least one of the obstructions H_1 or H_2:

1. If there exist some variable x where $x < z$, and x appears in B, we get the obstruction H_1, and we are unable to move z before B.

2. Assume there is no such variable as in 1, i. e. no variable x such that $x < z$ and $x \in B$. Then there must be some other clause, or clause and variable prohibiting z from being ordered before B. This requires a four step case analysis.

   (a) If we only have clauses between B and z then there must be some clause C where $B < C < z$ which has at least one variable y where $y < z$ and $y \in C$ but $z \notin C$. This gives us obstruction H_1.

   (b) If there are only variables between B and z then let y be the first variable after B such that $B < y < z$. But given we assume B has no variables x where $x < z$, y would be positioned before B, contradicting the order $B < y$.

(c) If there are both variables and clauses between B and z then these variables are never positioned right after B by the argument in 2b. Let y be the first variable after B where $B < y < z$ and let C be the clause appearing before y. Since y was not ordered before C, C must contain a variable x' where $x' < C$ and $x' \in C$. If not y would be positioned before C. If $z \notin C$ we have the first obstruction H_1. Since $z \in A$, $z \notin C$ and $x' \in C$ where the orders are $A < C$ and $x' < z$. Assume therefore that $z \in C$, but this will give us the obstruction H_2. Since $z \in A$, $z \notin B$, $z \in C$, $y \notin C$ and $x' \in C$ where the orders are $A < B < C$ and $x' < y < z$. This completes the proof.

Runtime: To achieve $\mathcal{O}(|V| + |E|)$ we describe the implementation we made for this algorithm.

First, consider the merging loop. The outer loop runs in time $\mathcal{O}(n)$. When positioning the variables into $\rho(cla)$ we will only need a single pass through the possible positions (this is because we cannot change the two orderings). Giving us runtime $\mathcal{O}(n + m) = \mathcal{O}(|V|)$. When validating a position for some variable x_j we can use a counter to represent the number of different clauses x_j must appear in (based on condition 1), and a boolean array over all clauses, where an clause element is set to true if x_j must appear in it (i. e. we try to position x_j before clauses containing a variable x_i where $i < j$). This takes time $\mathcal{O}(deg(x_j))$ to validate a single position. We check if the clauses x_j are adjacent to contains the required ones, compare the "hit" clauses with the counter. If it was an invalid position, then we must increment the position until we change at least one clause from true to false in the boolean array. For each next position check if the passed clause required x_j, time $\mathcal{O}(1)$ using the boolean array. Continue until you match the required clauses. When a valid position is found, update the boolean array based on condition 1. Positioning x_j takes a total time of $\mathcal{O}(deg(x_j) + \#checkedpositions)$. Given that we proceed with the next position if we found it to be invalid for some variable the total time to validate all positions for all variables is at most $\mathcal{O}(|cla| + |var| + |E|) = \mathcal{O}(|V| + |E|)$.

Second, consider the next loop for condition 2. Find clause A for the variable x_j, then check if x_j appears in all clauses between them. There cannot be more than $deg(x_j)$ such clauses. For all variables, we can check this condition in a total time $\mathcal{O}(|V| + |E|)$. And the proof is complete. □

2.1.4 Given one order only: algorithm for q-CNF formulas

We consider q-CNF formulas where every clause has q literals. In this section we present an algorithm for finding an interval ordering for a q-CNF boolean formula ϕ when given a specific ordering for the variables only, provided there exists an interval ordering of ϕ respecting the given ordering for the variables. The algorithm works by finding a permutation of the clauses in ϕ, such that it can be merged by the interval merging algorithm. The need for the restriction to q-CNF formulas becomes apparent in the proof of Lemma 2.8, for Rule 3.

We have already proved that we can merge two orders for the clauses and variables given they do not induce any of the obstructions in Lemma 2.6. Therefore our objective will be to resolve any potential obstructions induced by the clauses over the given ordering on the variables. We define three rules for handling the possible obstructions in Figure 2.6 that can arise between any pairs of clauses A,B, or triples A,B,C.

- R1: If obstruction H_1 can be formed then place B before A.
- R2: If obstruction H_2 can be formed and B has additional edge to a variable before z, then place B before A and C. (note that A and C are symmetric with respect to B, i. e. it does not matter if $A < C$ or $C < A$).
- R3: If obstruction H_2 can be formed and B has no additional edges to variables before z, then place B after A and C. (note that A and C are symmetric with respect to B, i. e. it does not matter if $A < C$ or $C < A$).

For rule R_1, it is easy to see that for obstruction H_1, B must appear before A in any clause ordering, see figure 2.5 for an illustration. For obstruction H_2 this seems non-trivial, but given we are not allowed to change the ordering of the variables we can still decide how these clauses should be ordered by a few observations.

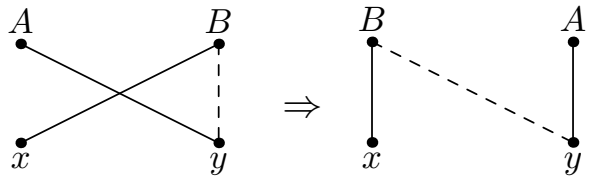

Figure 2.5: H_1 is resolved by moving clause B before clause A. A solid edge represents an edge from the incidence graph, a dotted edge is non-edge in the incidence graph, while any other edge combination can occur in the incidence graph.

Considering H_2, we must find every clause B and order them either before or after A and C, since B cannot be inserted in between. The following lemmas provides rules R_2 and R_3 for ordering A, B and C.

Lemma 2.8. *Given an incidence graph of a boolean q-CNF formula $G = (cla, var, E)$ and a linear ordering $\sigma(var) = \{x_1, x_2, ..., x_n\}$, if the obstruction H_2 can be formed, and B has an edge to a variable appearing before z, B must be positioned before A and C.*

Proof. Assume B has an edge to a variable x' appearing before z where B is positioned after A and C. We then immediately observe that the induced subgraph given by Az and Bx' gives the obstruction H_1. See figure 2.6. As a result B must be ordered before A, and then again before C, giving us R_2.

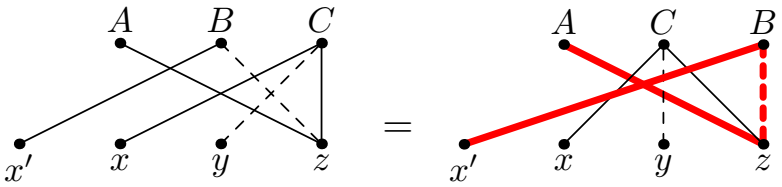

Figure 2.6: Obstruction H_1 is highlighted in red. A solid edge represents an edge from the incidence graph, a dotted edge is non-edge in the incidence graph, while any other edge combination can occur in the incidence graph.

□

Lemma 2.9. *Given an incidence graph of a boolean formula $G = (cla, var, E)$ and a linear ordering $\sigma(var) = \{x_1, x_2, ..., x_n\}$, if the obstruction H_2 can be formed, and B only has edges to variables appearing after z, B must be positioned after A and C.*

Proof. Given the obstruction H_2 we know that the clause C must be positioned before the variable y in any interval ordering (see rules for interval ordering). If we then position B before C, we know C must contain every variable in B (see rules for interval ordering). If not we then observe that for some variable x', where $x' \notin C$ that the induced subgraph given by Cx and Bx' gives us the obstruction H_1. However, given every clause has the same size q, C cannot contain everything B does, as well as the variable z. As a result B must be ordered after C, and then again after A, giving us R_3. See figure 2.7.

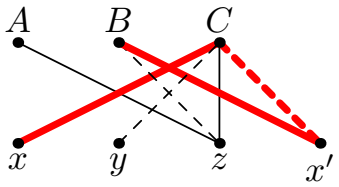

Figure 2.7: If B is positioned before C we get the obstruction H_1 which is highlighted in red. A solid edge represents an edge from the incidence graph, a dotted edge is non-edge in the incidence graph, while any other edge combination can occur in the incidence graph..

□

We have now obtained three rules of ordering, R_1, R_2 and R_3. To represent these rules over the clauses, we can build a simple directed graph R where each clause is represented by a single vertex. We add a directed edge from vertex A to vertex B in R iff one of the rules R_1, R_2 or R_3 tells us that A must be ordered before B in a final linear ordering of the clauses and variables.

Definition 2.10. (Rule graph) A simple directed graph R which represents the rules R_1, R_2 and R_3 over the clauses of a boolean formula is called a rule graph.

Lemma 2.11. *A rule graph R must be acyclic to prevent any obstruction.*

Proof. To make an interval ordering we need to position every clause and variables into a linear ordering without admitting any of the obstructions H_1 and H_2. For any cycle C in R with vertices v_1, v_2 $\dots$ v_t, where v_1 points to v_2 and so on, until v_t, this means they must be ordered in this exact order, and we cannot have an edge from v_t pointing to v_1 as it creates a contradiction, v_t cannot be both before and after v_1. □

Lemma 2.12. *Given an acyclic rule graph we can order the clauses such that the clause ordering does not induce any of the obstructions H_1 or H_2.*

Proof. By topological sorting the acyclic rule graph we get a linear ordering of the clauses where every edge is directed from left to right. This ordering then respects all the rules in the rule graph, giving us no obstructions in the bipartite graph induced by this new clause ordering and the given variable ordering σ. To see why this is true, let R be a rule graph with a topological sorting, then consider the following:

1. For any clause $A < B$ in the topological sorting, assume A and B induces the obstruction H_1 over the variables ordering. Then by rule R_1 B would have an edge from B to A, contradicting $A < B$.

2. For any clause $A < B < C$ in the topological sorting, assume A, B and C induces the obstruction H_2 over the variables ordering. Then by the rules R_2 or R_3 B would either have an edge from B to A and C, or from A and C to B, contradicting $A < B < C$.

See [9] for a proof of a topological sorting algorithm and its runtime complexity. □

The lemmas presented in this section give rise to the next theorem 2.13 for solving the problem of recognizing an interval ordering in a boolean q-CNF formula, given a strict ordering for the variables.

Theorem 2.13. *Given a q-CNF boolean formula ϕ with n variables and m clauses and a linear ordering σ for the variables, we can in time $\mathcal{O}(q \cdot m^3)$ find a permutation of the clauses, such that we can merge them into an interval ordering, provided there exists an interval ordering of ϕ respecting σ.*

Proof. We provide algorithm 2.3 for solving this problem, and prove it solves the problem in the given time.

Correctness: The algorithm applies rules R_1, R_2 and R_3 to construct the rule graph R. Applying Lemmas 2.7, and 2.12 correctness follows.

Runtime: To prove this algorithm runs in time $\mathcal{O}(q \cdot m^3)$, we need to prove that we can build the rule graph R within this time bound. The rule graph is represented by an adjacency matrix such that an edge insertion takes constant time. For each variable in $\sigma(var)$, we store its positions in the linear ordering into an array pos[] (i. e. if variable x_4 is positioned as the fifth variable

from the left, we let pos$[x_4] = 5$), takes time $\mathcal{O}(n)$. For each clause we store its lowest variable position in in the variable ordering (i. e. if variable x is the leftmost variable appearing in a clause C, store x in the array $leftmostVariableInClause[C]$), takes time $\mathcal{O}(m \cdot q)$. We will need the following observation to prove the correctness of R_1:

Observation 2.14. If two clauses $C < C'$ induces an obstruction over a variable $x \in C'$ and a variable $z \in C$, such that $x < z$, then H_1 will also be induced over any other variable $x' \in C'$ where $x' < x$.

For R_1 For each pair of clauses we check if they induce the obstruction H_1 from the variable ordering. We must consider each of the two orderings $C < C'$ and $C' < C$. Start with $C < C'$ and do the following: let x be the leftmost variable in C' (using observation 2.14). Then if there exist a variable $z \in C$ such that $pos[x] < pos[z]$ and $z \notin C'$, add the appropriate edge. Checking this takes time $\mathcal{O}(q)$, since we only need to consider one variable from C' and every variable in C. Repeat this for $C' < C$. In total we get the run time $\mathcal{O}(qm^2)$.

For R_2 and R_3 Consider the obstruction H_2. It takes time $\mathcal{O}(mq)$ to find every clause with the property of C (i. e. for each clause find its most left variable x and its most right variable z and see if the following is true: $pos[z] - pos[x] > q - 1$, then there is at least one variable y inside this sequence such that $C < y$). For all variables in C that have the property of z in H_2 we find its corresponding clauses A and B. In total we have $m - 1$ such clauses. Then for each clause B, do the following: check if its leftmost variable is positioned before z (using array $leftmostVariableInClause$), if all agree on the same direction, then we can add the appropriate edges, if they disagree then all A's must be both before and after B which is not possible. This takes $\mathcal{O}(m^2)$ time. Resulting in a total runtime $\mathcal{O}(q \cdot m^3)$ which completes the proof.

□

We believe that the runtime in theorem 2.3 can be reduced to $\mathcal{O}(q \cdot m^2)$ by using more sophisticated data structures. This is because in a simple graph there cannot exist more that m^2 edges, but the algorithm we have created add the same edge multiple times. This improvement is left for future work.

2.2 K-Interval Ordering

In this section we follow the paper [4] and generalize the interval ordering for CNF formulas to k-interval orderings. Note that an interval ordering is a 0-interval ordering. This will cover more classes of CNF formulas, in fact any CNF formula has a $k-$interval ordering for some value of k, but

Algorithm 2.3 q-CNF MergeIntervalOrder

input: $G = (cla, var, E)$ and a linear ordering $\sigma(var) = \{x_1, x_2, ..., x_n\}$
output: an interval ordering on $cla \cup var$ iff an interval ordering exists respecting σ

$R = cla$ ▷ Graph with the clause nodes from G

For rule R_1:
for each clause $c \in cla$ **do** ▷ Takes time: $\mathcal{O}(m)$
  for each clause $c' \in cla$ where $c' \neq c$ **do** ▷ Takes time: $\mathcal{O}(m)$
    if c and c' induce H_1 **then** ▷ Takes time: $\mathcal{O}(q)$
      add edge $c \to c'$ in R
    end if
  end for
end for

For rule R_2 and R_3:
for each clause $c \in cla$ **do** ▷ Takes time: $\mathcal{O}(m)$
  if c has the properties as C has in H_2 **then**
    for each variables in c having the property as z as in H_2 **do** ▷ Takes time: $\mathcal{O}(q)$
      find all A and B ▷ Takes time: $\mathcal{O}(m)$, (done once)
      if $A \neq \emptyset$ and $B \neq \emptyset$ **then**
        if any B is adjecent to vertex before z **then** ▷ Takes time: $\mathcal{O}(m)$, (done once)
          add edges $B \to C$ and $B \to A$ in R ▷ Takes time: $\mathcal{O}(m^2)$
        else
          add edges $C \to B$ and $A \to B$ in R ▷ Takes time: $\mathcal{O}(m^2)$
        end if
      end if
    end for
  end if
end for

if R can be topological sorted **then** ▷ Takes time: $\mathcal{O}(m + |E(R)|)$
  $\rho =$ the topological sorted clause ordering
  return interval ordering from merging σ and ρ with graph G
else
  return "impossible"
end if

the value k could be as large as the number of variables. We are interested in formulas having low k-value as the underlying DP algorithms for solving MaxSAT and #SAT run exponentially in this k-value. We do not know of any polynomial time exact algorithms for this problem in the general case and will instead focus on limited structure. In [4] an algorithm is given for merging two linear orders, one for the clauses and one for the variables, into a k-interval ordering such that the value of k is minimized. This algorithm is similar to the merging algorithm presented in section 2.2, and for $k = 0$ we have an interval ordering. Before describing this algorithm we start by a formal definition.

Definition 2.15. For any bipartite graph $G = (cla, var, E)$, if we can add at most k edges to each vertex in cla making it into an interval bigraph, then we define G as a k-interval bigraph. For a CNF formula F, if its incidence graph is a k-interval bigraph, then F is a k-interval CNF formula. We

extend the interval ordering for such graphs to a k-interval ordering.

Lemma 2.16. *A CNF formula is a $k-$interval CNF formula if and only if the clauses and variables can be totally ordered such that: for any clause C there are at most k variables x not appearing in C where either, $C < x$ with some clause $C' < C$ and x appearing in C', or $x < C$ with some variable $x' < x$ and x' appearing in C.*

Proof. Follows from definition 2.1 and 2.15. □

Observation 2.17. Note that in lemma 2.16 an upper boundary on the *k-value* is given by number of variables in the CNF formula. We will use this observation through the entire experimental part of the thesis.

2.2.1 Given two orders: the Mk merging algorithm

Given an incidence graph $G = (cla, var, E)$ and two linear orders, one for the clauses and one for the variables, we can merge them into a total linear ordering in time $\mathcal{O}(k\,|E|)$ where k is minimized using the Min k-interval merging (Mk) greedy algorithm. To find the minimized value of k we incrementally test whether it is possible to merge the two orders using some value k, starting from 0. The algorithm is given below. A proof of correctness and its runtime can be found in [4].

Algorithm 2.4 Min k-interval merging (Mk)

input: $G = (cla, var, E)$, $\sigma(A) = \{x_1, x_2, ..., x_n\}$ and $\rho(B) = \{c_1, c_2, ..., c_m\}$
output: minimum k such that the given orderings can be merged to a k-interval ordering

$k = -1$
L $k = k + 1$
start with the ordering $x_1, x_2, ..., x_n$
for $i := m$ **down to** 1 **do**
    insert c_i at the highest position below c_{i+1} where $EdgesAdded(c_i) \leq k$
    if no such position exist then break out of loop and **goto L**
end for

2.2.2 Heuristics algorithm

To obtain a total linear ordering over a CNF formula with a low k-value by the Mk algorithm, we need as input two linear orderings for the clauses and variables that, when merged by Mk, achieves a low k-value. To find these two input orderings we introduce a heuristic procedure, the Barycenter heuristic (BH), as computing the best linear orderings is probably not efficient. BH takes as input a CNF formula, extracts two linear orderings over the clauses and variables simply by their ordering in the CNF formula, and then reorders them to provide a good input to the Mk algorithm. BH reorders the two linear orderings based on the following assumption:

- In a total linear ordering for a CNF formula having the lowest possible k-value, a clause will not be too far away from the variables appearing in it.

This since the larger the separation is by non-member variables, the k-value generally increases. BH works by alternating between reordering clauses and reordering variables based on their neighbors average position (e. g. for a clause C, if Cs variables are in position 3,9,15, its average position is $\frac{3+9+15}{3} = 9$). The new ordering for C is then based on this average position (e. g. a clause with an average variable position 8 is ordered before a clause with an average variable position 9). By repeating this procedure for all clauses and then variables multiple times, we narrow the gap between a clause and its variables. The number of iterations depends on the size of the formula, but as the experimental chapter 3 reveals, we do not necessarily need to run BH until convergence (i. e. k-value is not improved by applying more iterations) as it does not improve the k-value substantial beyond a constant number of iterations. The BH algorithm algorithm runs in

Algorithm 2.5 Barycenter Heuristic (BH)

input: $G = (cla, var, E)$, $\sigma(var) = \{x_1, x_2, ..., x_n\}, \rho(cla) = \{c_1, c_2, ..., c_m\}$ and $i = \#iterations$
output: ζ, a linear ordering on $cla \cup var$

$orderVars = true$

while $i \geq 0$ **do** ▷ Takes time: $\mathcal{O}(\#iterations)$
  if $orderVars = true$ **then**
    for each variable x in $\sigma(var)$ **do** ▷ Takes time: $\mathcal{O}(|n|)$
      Calculate the average position of all clauses x occurs in. ▷ Time: $\mathcal{O}(deg(x))$
      Place x accordingly to its average position. ▷ Sorting: $\mathcal{O}(log(n)\, n)$
    end for
    $orderVars = false$
  else
    for each clause C in $\rho(cla)$ **do** ▷ Takes time: $\mathcal{O}(|m|)$
      Calculate the average position of all variables C contains. ▷ Time: $\mathcal{O}(deg(C))$
      Place C accordingly to its average position. ▷ Sorting: $\mathcal{O}(log(m)\, m)$
    end for
    $orderVars = true$
  end if
  $i = i - 1$
end while

Run Mk on $\sigma(var)$ and $\rho(var)$, and return the merged total linear ordering:
Return Mk(G, $\sigma'(var)$ and $\rho'(cla)$).

time $\mathcal{O}(\#iterations \cdot (max(n,m)^2) \cdot log(max(n,m)))$ which is sufficient for any of the tested boolean formulas. When discovering the BH heuristic algorithm we found that it is also used as a heuristic for a well known problem called Two-Sided Crossing Minimization(TSCM). This is also where the Barycenter name came from. The TSCM problem asks to minimize the number of edge crossings in the drawing of a bipartite graph where we fix each of the partition of vertices into two parallel layers. See figure 2.8 for an illustration. The BH heuristic is an $O(\sqrt{n})$-approximation algorithm for TSCM [5].

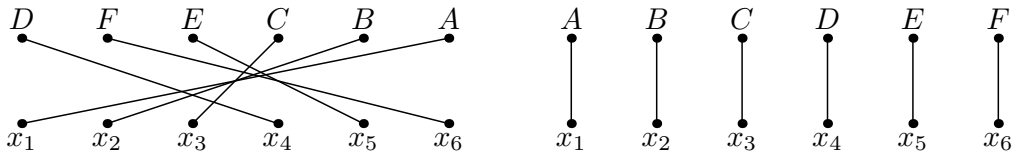

Figure 2.8: Crossing minimization in a two-layered bigraph. In the left figure we have a bipartite graph with multiple crossings. In the right we have the same bipartite graph, but with the minimum number of crossings. In this case the minimum number of crossings is 0.

We thought it would be interesting to check whether the problem of finding a minimum k-interval ordering and TSCM are equivalent in some way, e. g. if the crossings are decreased, does the k-value decrease as well? However, we can show that this does not appear to be the case. In the figures below we assume we have some subgraph G' in the bipartite graph G where the number of crossings is 4. If we reorder the top vertices in the graph the number of crossings is reduced by one, while the *k-value* increases by one. This gives us the impression that we do not necessarily improve the *k-value*, when the crossings are reduced.

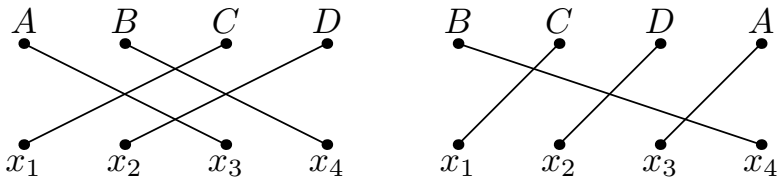

Figure 2.9: The left figure has 4 crossings and k-value 2. In the right figure we have the same bigraph with a different reordering of the top vertices giving us fewer crossings and k-value 3.

2.3 PS-width of interval ordered boolean formulas

We follow the presentation in paper [3].

Definition 2.18. For a CNF formula F with variables *var* and clauses *cla*, we define the ps-value of F as done in [4]. Given an assignment τ on var, we let $\text{sat}(F, \tau)$ denote the inclusion maximal set $C \subseteq cla$ such that every clause in C is satisfied by τ. The set C is then called projection satisfiable. The ps-value of a formula F is then the number of distinct projection satisfiable subsets given by different assignments τ, i. e. $|\text{PS}(F)|$, where

$$\text{PS}(F) = \{\text{sat}(F, \tau) : \tau \text{ is an assignment of var}\} \subseteq 2^{cla}.$$

We are interested in solving subproblems and combining them into a complete solution. This can be done by making "cuts" in a linear ordering on the clauses and variables of a CNF formula F. Assume a linear ordering $e_1, e_2, \ldots, e_{n+m}$ on the clauses and variables in F. For any $1 \leq i \leq n+m$, we get two disjoint sub-formulas $F_1(i)$ and $F_2(i)$ crossing between $e_1, e_2, \ldots, e_i$ and $e_{i+1}, \ldots, e_{n+m}$. $F_1(i)$ is the subformula we get by removing from F any clause not in $e_1, \ldots, e_i$ and any literal of

any variable not in $e_{i+1}, \ldots, e_{n+m}$. $F_2(i)$ is just the opposite, that is, remove any clause from F not in $e_{i+1}, \ldots, e_{n+m}$ and any literal of any variable not in $e_1, \ldots, e_i$. Let us demonstrate these sub-formulas through the following example.

Example 2.19. Assume we have a CNF formula F with three clauses: $c_1 = (x_1 \vee \bar{x_2} \vee x_4)$, $c_2 = (x_2 \vee \bar{x_3} \vee x_5)$ and $c_3 = (x_3 \vee \bar{x_4} \vee x_5)$ with the following linear ordering: $x_1\, c_1\, x_2\, x_3\, c_2\, c_3\, x_4\, x_5$. Let us make a "cut" in the ordering between x_3 and c_2, i. e. $i = 4$. Then $F_1(4) = (x_4)$ and $F_2(4) = (x_2 \vee \bar{x_3}) \wedge (x_3)$

We define the ps-width of this linear ordering to be the maximum ps-value of all the possible subformulas $F_1(1), F_2(1), F_1(2), F_2(2), \ldots, F_1(n+m), F_2(n+m)$ crossing a cut in the ordering. Since we want to minimize this width parameter we define the linear ps-width to be the minimized ps-width over all possible orderings on var $\cup$ cla.

The following result is implicit in [3]. We give a full proof.

Theorem 2.20. *Assume CNF formula F on m clauses has an interval ordering. Then the ps-width of F is at most $m + 1$*

Proof. Consider an interval ordering on F. Take any cut. The clauses crossing the cut can then be ordered $C_1 \ldots, C_q$ such that any variable in C_i is also in C_{i+1}, this follows from the definition of an interval ordering. We prove the theorem by induction on q. We let F_k denote the subformula having exactly k clauses, $C_1, C_2, \ldots, C_k$ crossing the cut. We need to show that for any k, $|\mathrm{PS}(F_k)| \leq k + 1$. If $k = 1$ then $|\mathrm{PS}(F_1)| = 2$, since either C_1 is satisfied or not. By induction we assume this is true for $q = k$, that is $|PS(F_k)| \leq k + 1$. For $q = k + 1$, we get some new clause C_{k+1} which contains all variables that any of the prior clauses contains. For this new clause there is only a single assignment to the variables that falsifies this new clause and every other assignment will satisfy it. We then get that $|\mathrm{PS}(F_{k+1})| \leq |\mathrm{PS}(F_k)| + 1$, adding one assignment that will falsify C_{k+1} regardless of what prior clauses are satisfied by this assignment. This means that by the induction assumption $|\mathrm{PS}(F_{k+1})| \leq k + 2$. This completes the proof. □

Chapter 3

Experimental results

As previously mentioned in the introduction, the experimentation we conduct in this thesis consists of comparing different algorithms by how well they minimize the k-value on various types of boolean formulas. We start this chapter by introducing an additional algorithm which is used in combination with those algorithms already presented in chapter 2. This new algorithm, called the shifting heuristic (SH), takes as input a total linear ordering on the clauses and variables of a boolean formula, and outputs a new optimized linear ordering. SH was a result of complications discovered when attempting to solve MaxSAT and #SAT on the total linear ordering produced by the Mk-algorithm. Not only did SH fix the problem in the Mk linear ordering, but often it would also reduce the k-value if we allowed it to reorder the clause ordering.

After introducing SH, we first list all various algorithm combination we use for experimentation throughout this thesis. We then continue with applying these on SAT instances taken from SAT competitions, hoping to find solvable real-world problems. After plotting these results we list a few instances we found which happen to have relatively low k-value compared to the number of variables. We do not go into the details of these instances as they were found very late in writing this thesis.

The last experimentation we conduct is on crafted boolean formulas having an interval ordering. These are purely artificial, but it is a good measure on how well these algorithm combinations perform when we know they have k-value zero. Because of time and memory limitation, we mostly test instances less than 1MB. See open problems section in the end of the thesis for this reason.

3.1 Improving Mk

When we first started experimenting with the different algorithms we noticed that we were often unable to solve MaxSAT or #SAT within reasonable time limits using the linear ordering produced by Mk. Even for small instances, where k was low, this appeared to be a problem. When investigating these linear orders we discovered that the problem is twofold. In each iteration of Mk, a clause is merged in the rightmost position as soon as it hits the threshold k while respecting the clause order (See algorithm 2.4). We call this a right-positioning bias. The implications are then:

1. a lot of clauses will likely need to include many variables, resulting in poor running time.

2. these variables will usually appear before the clause itself in the ordering, resulting in a highly asymmetrical linear ordering which is more complex to solve.

To be more precise about this increased complexity emerging from 2., we must first give a definition followed by an example.

Definition 3.1. Given a k-interval ordering σ, for every clause $C \in \sigma$, we define k_L to be the number of variables x we must add to C where $x < C$, and we define k_R to be the number of variables x we must add to C where $C < x$.

Example 3.2. To illustrate the right-ordering bias, we ran Mk on a small crafted boolean formula having an interval ordering. We gave Mk a random permutation on the variables and clauses, which resulted in k = 29. The results are displayed in the table below.

23,0	20,0	17,0	16,0	13,0	10,2	14,2	5,2	6,2	10,2
5,3	3,3	4,3	12,0	9,0	12,0	20,1	5,3	9,0	21,3
3,5	12,0	4,0	5,2	12,0	4,0	14,2	4,0	5,2	20,1
13,0	9,0	12,0	24,0	4,3	13,0	13,0	9,0	13,0	13,0
10,2	10,2	9,0	9,0	14,2	23,6	16,0	13,0	14,2	14,2

Table 3.1: Each cell contains k_L, k_R for a unique clause in the linear ordering produced by Mk. Note that k_L is generally much higher than k_R.

Solving #SAT on this particular instance gave us the following results:

Solving #SAT
Largest PS-set: 528000
Accumulated Left PS-set: 745
Accumulated Right PS-set: 3526439
Running time: 6.634 seconds.

In the example above we observed that k_L is generally much higher than k_R, and without going into details of the DP algorithm for solving MaxSAT and #SAT, the problem caused by this asymmetry is that this algorithm runs exponentially over these values, i. e. for some constant $a > 1$ and values $k_L \geq 0$ and $k_R \geq 0$, it takes time $\mathcal{O}(poly \cdot a^{k_L} + poly \cdot a^{k_R})$ to compute sub-formulas for cuts in the linear ordering. In other words, the overall running time would be better if we made k_L and k_R more symmetric, so that the running time would be more like $\mathcal{O}(poly \cdot a^{\frac{k_L+K_R}{2}} + poly \cdot a^{\frac{k_L+K_R}{2}})$. We do not prove this argument formally, but give a proof of concept through an example.

We designed a simple shifting heuristic (SH) for handling this problem.

3.1.1 Shifting heuristic (SH)

The Shifting heuristic (SH) takes as input an incidence graph of a CNF formula and a total linear ordering of its clauses and variables. Recall that the whole point of SH is to optimize the linear ordering before running the DP algorithm for solving MaxSAT and #SAT. We implemented two variants, one where we allow it to shift a clause passed other clauses, and thereby reorder the clause ordering, and one which did not. When we allowed the algorithm to reorder the clauses we generally observed that the k-value decreased. For that reason we will only use the SH variant that allows reordering of the clauses when conducting experiments.

Algorithm 3.1 Shifting Heuristic (SH)

input: $G = (cla, var, E)$, total ordering $\sigma = \{cla \cup var\}$
output: σ, a linear ordering on cla $\cup$ var

repeat
  for each clause C in σ **do**
    Calculate k_L and k_R for C
    if $k_L > k_R$ **then**
      Move C leftwards in σ, past $\frac{k_L - k_R}{2}$ variables.
    end if
  end for
until no clause changed position

Example 3.3. Processing the total linear ordering from the previous example with SH gives us the following improved k-values.

1,0	1,0	1,0	1,0	1,0	3,2	7,2	3,2	4,2	3,2
4,3	3,3	4,3	6,5	5,4	6,5	8,1	4,3	5,4	21,3
3,5	6,5	2,2	3,2	6,5	2,2	8,2	2,2	3,2	8,1
1,0	5,4	6,5	12,0	4,3	1,0	1,0	5,4	1,0	1,0
3,2	3,2	5,4	5,4	7,2	23,6	1,0	1,0	7,2	12,2

Table 3.2: Each cell contains k_L, k_R for a unique clause in the linear ordering after running SH. Note that in this particular instance, max $\{k_L + k_R\}$ is the same as before, but many k_L values are decreased.

Overall we observe a significant reduction in the number of variables added to the clauses. We now solve #SAT on this new optimized ordering, giving us the following result:

Solving #SAT
Largest PS-set: 480
Accumulated Left PS-set: 931
Accumulated Right PS-set: 7354
Running time: 0.075 seconds.

Comparing with the previous ordering note that the runtime is now reduced by a factor of 88.

3.2 Algorithms

The following combination of algorithms that all rely heavily on the Mk algorithm from section 2.1.3 are tested on each SAT instance, and each of the artificially constructed CNF formula have an interval ordering. All of these algorithms have the same output. Each algorithm finds a total linear ordering over the clauses and variables in a CNF formula using the Mk algorithm in various ways. For clarification, the reader should follow the flowchart provided in the introduction while reading this section.

BH100+Mk Given a CNF formula as input, the algorithm extracts the clauses and variables into two separate linear orders simply by their ordering in the CNF encoding. This means that the first clause from the encoding is the first clause in the clause ordering, and so forth, and the variable (a variable and its negation is treated the same way) from the first clause is the first variable in the variable ordering, followed by the second, and so forth, without repeating any of the variables. These two orders are then processed through the BH algorithm over 100 iterations which is then merged together into a total linear ordering by the Mk algorithm.

BH1000+Mk Identical to BH100+Mk, except for running 1000 iterations of BH.

BH10000+Mk Identical to BH100+Mk, except for running 10000 iterations of BH.

SO+Mk (Simple Order with Mk) Given a CNF formula as input, we extract the clauses and variables into two separate linear orders as above. These two orders are then merged together into a total linear ordering by the Mk algorithm.

GH+Mk Given a CNF formula as input, its incidence graph is processed through the greedy heuristic algorithm, creating a total linear ordering for the clauses and variables. This ordering is then extracted into two linear orderings preserving their respective orderings. These two linear orderings are then merged together with the Mk algorithm.

BH100+Mk+SH The total linear ordering from **BH100+Mk** above is processed through the **SH** algorithm where we allow reordering of the clauses.

BH1000+Mk+SH The total linear ordering from **BH1000+Mk** above is processed through the **SH** algorithm where we allow reordering of the clauses.

BH10000+Mk+SH The total linear ordering from **BH10000+Mk** above is processed through the **SH** algorithm where we allow reordering of the clauses.

GH+Mk+SH The total linear ordering from **GH+Mk** above is processed through the **SH** algorithm where we allow reordering of the clauses.

For easy reference, we have listed each of the procedures used above in the table below, along with their input and output values.

Algorithm	**Input**	**Output**
Extract variable and clause orderings	CNF formula F	$\sigma(var)$ and $\rho(cla)$
Extract variable ordering	CNF formula F	$\sigma(var)$
BH	$G = (var, cla, E)$, $\sigma(var)$ and $\rho(cla)$	$\sigma(var)$ and $\rho(cla)$
Mk	$G = (var, cla, E)$, $\sigma(var)$ and $\rho(cla)$	$\sigma(var \cup cla)$ and k-value
GH	$G = (var, cla, E)$	$\sigma(var \cup cla)$
SH	$G = (var, cla, E)$ and $\sigma(var \cup cla)$	$\sigma(var \cup cla)$ and k-value

Table 3.3: Overview of the algorithms used in various combinations for computing a total linear ordering and its *k-value* from a CNF formula.

3.3 SAT competition

The main purpose of the SAT competitions is to identify challenging benchmarks and promote new solvers for the SAT-problem as well as comparing them against the best known solvers (see satcompetition.org). We found that the SAT competitions were a particularly good source for many practical real-world problems, and because all instances are encoded in DIMACS, we could simply download any instance we wanted without any conversion. However, since the underlying problem is to solve MaxSAT and #SAT we cannot compare our results directly, as these problems are harder to solve. There are competitions entirely designated for MaxSAT and #SAT as well, but most of the instances we found there were either too large or encoded in another format. We considered converting the smaller instances into DIMACS, but avoided it since we had enough instances for benchmarking. A SAT competition usually consists of three different tracks containing various types of instances: random, industrial and crafted instances. We did only experiment with the industrial and crafted instances. While an industrial instance is a problem taken from the industry for solving some practical application, crafted instances often encode hard combinatorial problems which are challenging to typical SAT solvers (this includes, e. g. instances arising from difficult puzzle games (see satcompetition.org)). When running our algorithms on these instances we want to identify practical problems that can be solved efficiently using the DP algorithm for solving MaxSAT and #SAT, in which case, this new technique could possibly be preferred over previous existing methods. The first benchmarking we do is on instances taken from the SAT RACE competition held in 2015. We continue with benchmarking instances taken from SAT COMP 11.

The benchmarking results are plotted using a line chart. The vertical axis in the chart represents the *k-value* found by the various algorithms of a total linear ordering using a logarithmic scale, while the horizontal axis represents the different instances tested in increasing size from left to right. We

have also included a line representing the number of variables for each instance, allowing us to easily compare the k-value to number of variables. It is worth noting that the lines in the charts do not represent instances between them, it is simply a practical way to visualize the results.

3.3.1 Results for SAT RACE 15

The following figure displays the results after running the algorithms presented in section 3.2 over selected instances from SAT RACE 15.

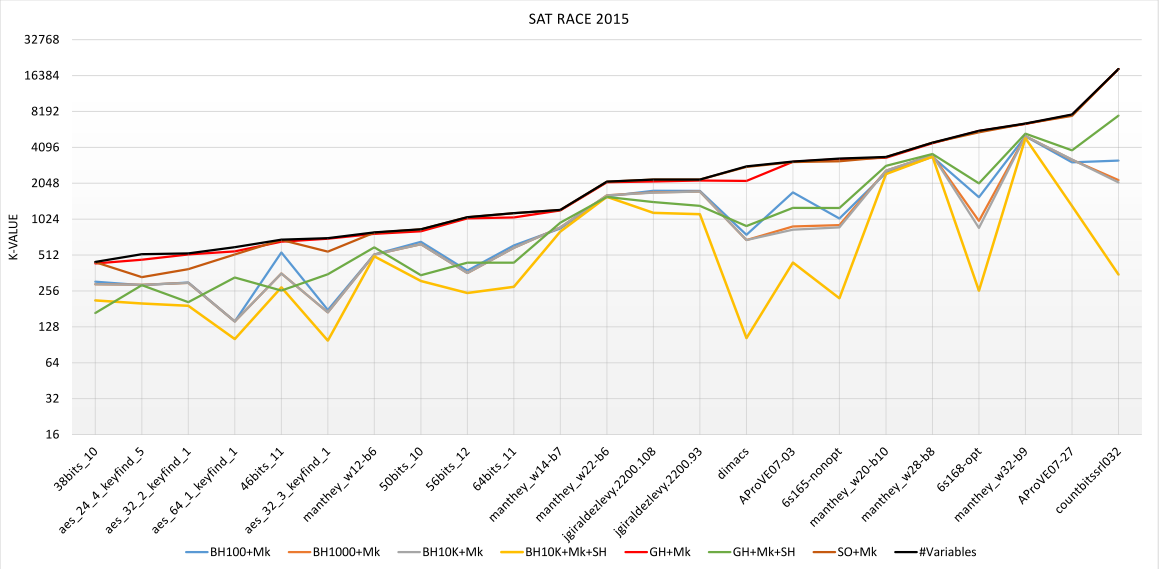

Figure 3.1: Results for 23 selected instances from SAT Race 2015. The horizontal axis represents each instance tested sorted by their number of variables, and the vertical axis represents k-value found by various algorithms. The black line in the chart represents an upper boundary for the k-value, in terms of the number of variables in each instance.

In figure 3.1 we observe that BH10000+Mk+SH is generally the best tested algorithm for finding a linear ordered of a CNF formula with the respect to minimum k-value. By increasing the number of iterations in BH, the k-value is improved in some instances, but increasing the number of iterations beyond 10000 did not improve k-value at all. The SO+Mk and the greedy heurstic GH+Mk overall performs poorly, as the k-value is usually as high as the number of variables in the instance, rendering them ineffective when decomposing these SAT instances. SH does a good job in reducing the k-value on instances already processed by GH+Mk and BH10000+Mk. Some of these instances had a low k-value compared to its number of variables, but still too high for any practical use. The single instance "dimacs" models a problem called symbolic simulation (see SAT RACE 2015), but since we only had a single instance we are not able to say anything about it in general. We leave this for future work.

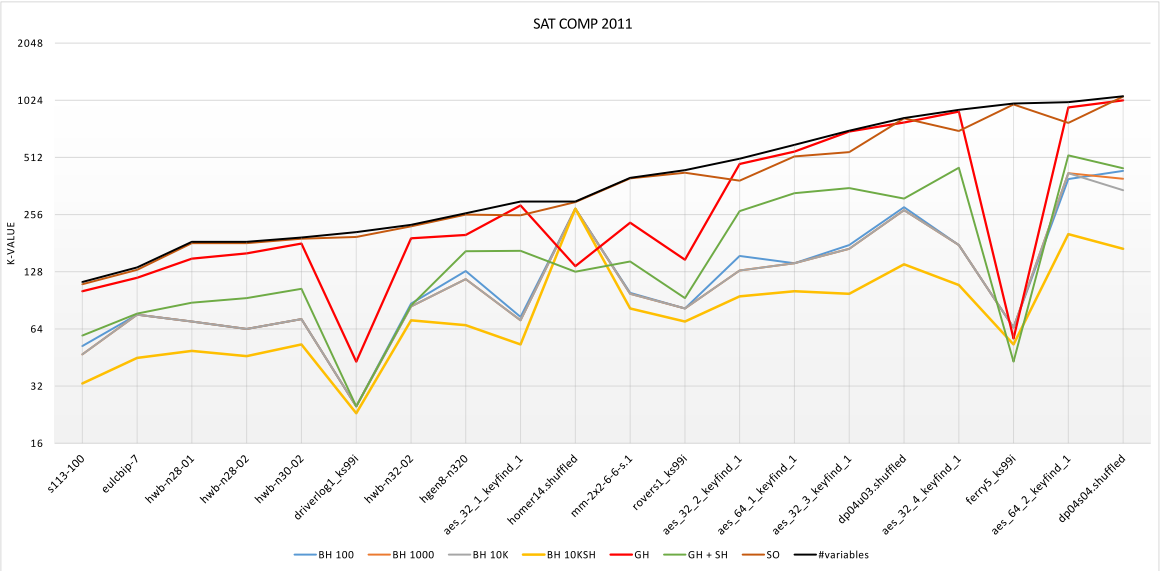

Figure 3.2: Results for 20 selected instances from SAT COMP 11. The horizontal axis represents each instance tested sorted by their number of variables, and the vertical axis represents k-value found by various algorithms. The black line in the chart represents an upper boundary for the k-value, in terms of the number of variables in each instance.

3.3.2 Results for SAT COMP 11

Each of the figures 3.2, 3.3 and 3.4 contains results over multiple SAT instances taken from SAT COMP 11. They are ordered by increasing size, i. e. the first figure contains smaller instances than the next, and so forth. We observe practically the same algorithmic results for instances tested in SAT COMP 11 as for instances tested in SAT RACE 15. The BH10000+SH is generally the best algorithm for minimizing k-value, while the worst is SO+Mk. GH+Mk performs almost as poorly as SO+Mk. Some of these instances may be solved efficiently by the DP algorithm for solving MaxSAT and #SAT. We describe them in the next section.

3.3.3 Significant practical instances

As the benchmarking results reveal, there are a few instances which have an exceptionally low k-value compared to the number of variables in the boolean formula. Some of these values are still considered to be high regarding the worst-case running time of the DP algorithm for MaxSAT and #SAT, but can hopefully be further decreased by other techniques. As the experimental part of this thesis is about recognizing boolean formulas for practical applications where this Dynamic Programming procedure is more favorable than others, we think this is a valuable result. Specifically it reveals that there exists practical problems which when encoded into a CNF formula have low k-value. In the figure below we have selected instances from the benchmarking results found in SAT Comp 11 where the k-value is very low compared to the number of variables. As the number of variables increases for each of these instances, the k-value still remains low.

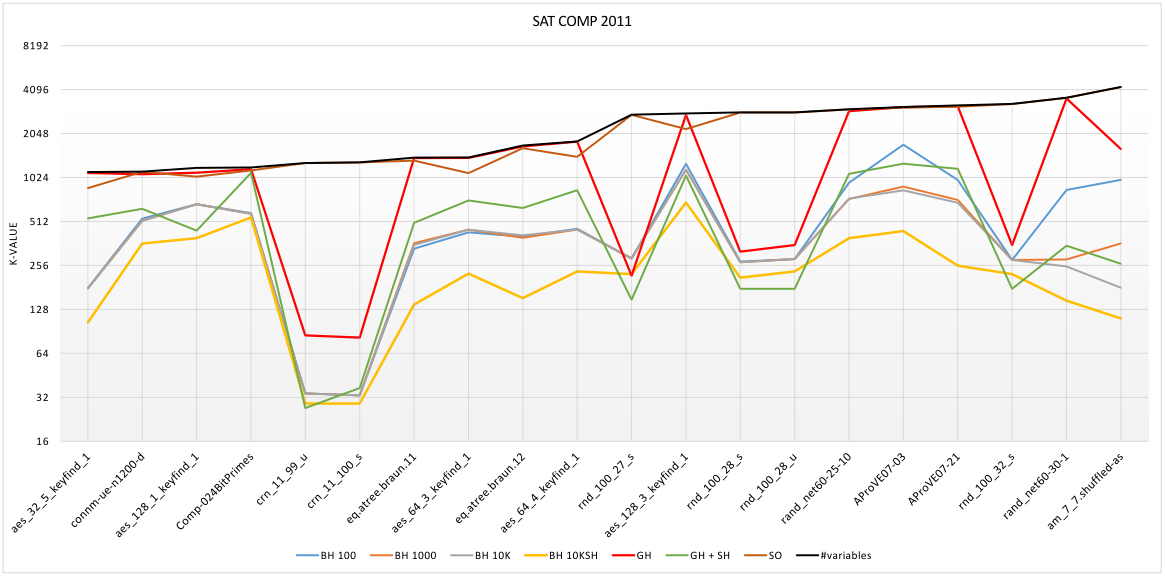

Figure 3.3: Results for 20 selected instances are taken from SAT COMP 11. The horizontal axis represents each instance tested sorted by their number of variables, and the vertical axis represents k-value found by various algorithms. The black line in the chart represents an upper boundary for the k-value, in terms of the number of variables in each instance.

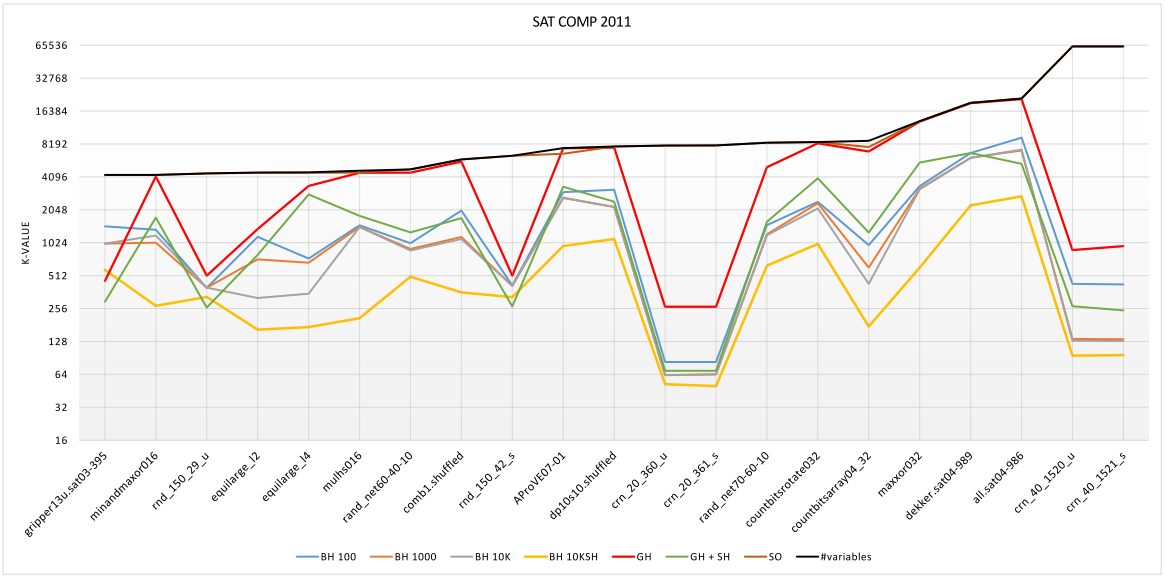

Figure 3.4: Results for 21 selected instances from SAT COMP 11. The horizontal axis represents each instance tested sorted by their number of variables, and the vertical axis represents k-value found by various algorithms. The black line in the chart represents an upper boundary for the k-value, in terms of the number of variables in each instance.

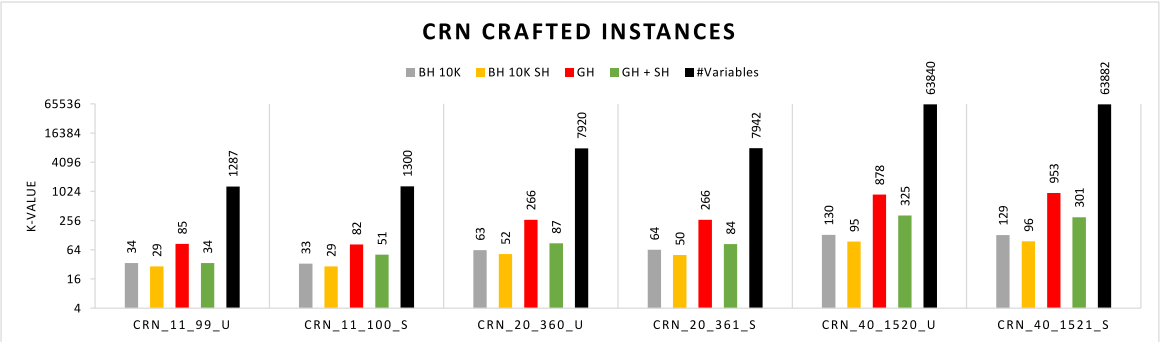

Figure 3.5: Results for a series of CNF formulas which encodes a particular problem called the *Automata synchronization*. Instances are selected from SAT COMP 11. The horizontal axis represents the various instances tested, and the vertical axis represents the *k-value* found by various algorithms. The black bar in the chart represents an upper boundary for the k-value, in terms of the number of variables in each instance.

These particular instances encode a problem called Automata synchronization. As mentioned earlier we will not conduct any in-depth analysis of the instances found in the SAT competitions, but we will describe what the Automata synchronization problem is, since it was the best result we found. The following description of automata synchronization is taken from [10, chapter 1, p.93] (we assume the reader is familiar with finite automata):

> Given a Deterministic Finite Automata (DFA) $M = (Q, \Sigma, \delta, q_0, F)$ and let h be some final state. A synchronizing sequence for M is a string $s \in \Sigma^*$ where $\delta(q, s) = h$ for every $q \in Q$. ($\delta(q, s)$ is extended to strings, so that $\delta(q, s)$ equals the state M ends up when M starts at state q and reads unput s.) We say that M is **synchronizable** if it has a synchronizing sequence for some state h.

3.4 Testing linear ordering heuristics

The CNF formulas that are tested in this section are generated by the same procedure as in [3]. To hide the natural interval ordering from its construction, all instances are randomly permuted, giving no advantage to either algorithm. The following subsections explain how such instances are generated, what instance sizes are tested, and the results given by the different combinations of algorithms.

3.4.1 Generating instances

For testing purposes, we generate the following type of instances for which each clause has t literals. A CNF formula on n variables and m clauses that has $n + m$ intervals on a real line. Each interval is generated by choosing left and right endpoints from 1 to $2(n + m)$.

- At step i, check which of the 4 cases below are legal (e. g. 3 is legal if there exists a live variable, i. e. with left endpoint $<$ i and no right endpoint) and randomly make one of the legal choices,

of the exception that case 4 is enforced for a live clause that at step i has accumulated exactly t overlapping variable intervals.

1. Start interval of new variable with left endpoint i
2. Start interval of new clause with left endpoint i
3. End interval of randomly chosen live variable by right endpoint i
4. End interval of randomly chosen live clause by right endpoint i

By the end of the process, boundary conditions are enforced to reach exactly m clauses, with n expected to be slightly smaller than m. For each clause interval we randomly choose each variable with overlapping interval as being either positive or negative in this clause. The resulting CNF formula will have an interval ordering given by the rightmost endpoint of these intervals.

3.4.2 Setup

To get a better understanding of how well these algorithms work for interval ordered formulas, we generate a few variations, varying the clause size and the number of variables. We construct multiple instances of the same size, since we do not expect that a single instance will represent all generated instance of the same size. The average result is then plotted in a diagram, resulting in a simple statistical measure.

3.4.3 Results

The benchmarking results are plotted using a line chart. The vertical axis in the chart represents the *k-value* found by the various algorithms of a total linear ordering using a logarithmic scale, while the horizontal axis represents the different instances tested in increasing size from left to right. It is worth noting that the lines in the charts do not represent instances between them, it is simply a practical way to visualize the results.

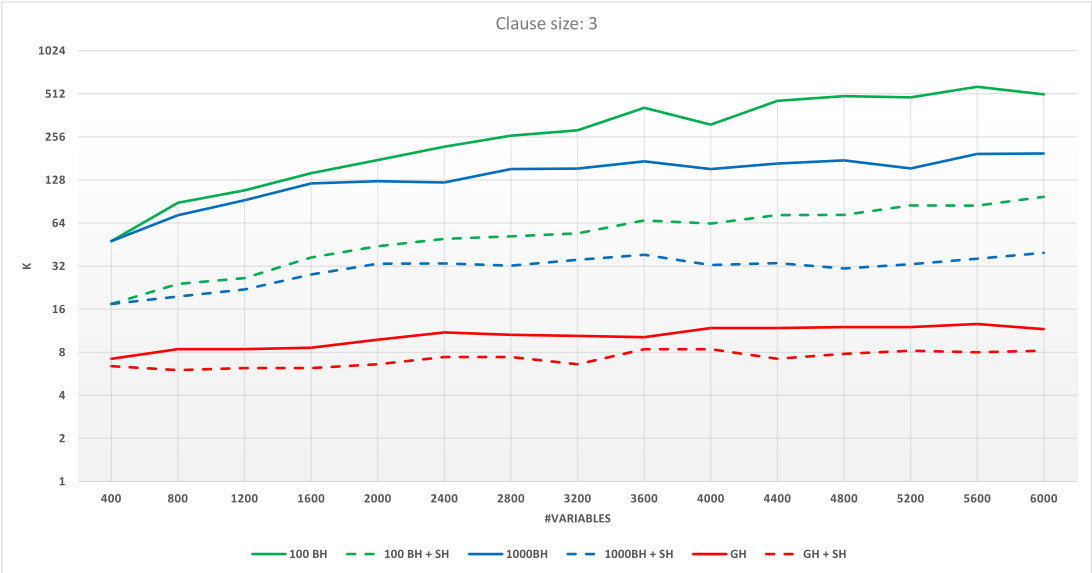

Figure 3.6: Benchmarking permuted type 2 interval ordering instances with clauses containing exactly 3 variables. Horizontal axis represents each instance tested sorted by their number of variables. Vertical axis represents the *k-value* found by the various algorithms.

As seen in figure 3.6 BH+Mk is performing poorly on these instances compared to GH. By shifting 100BH, 1000BH and GH we generally improve k-value with BH1000+SH not too far above GH+Mk+SH.

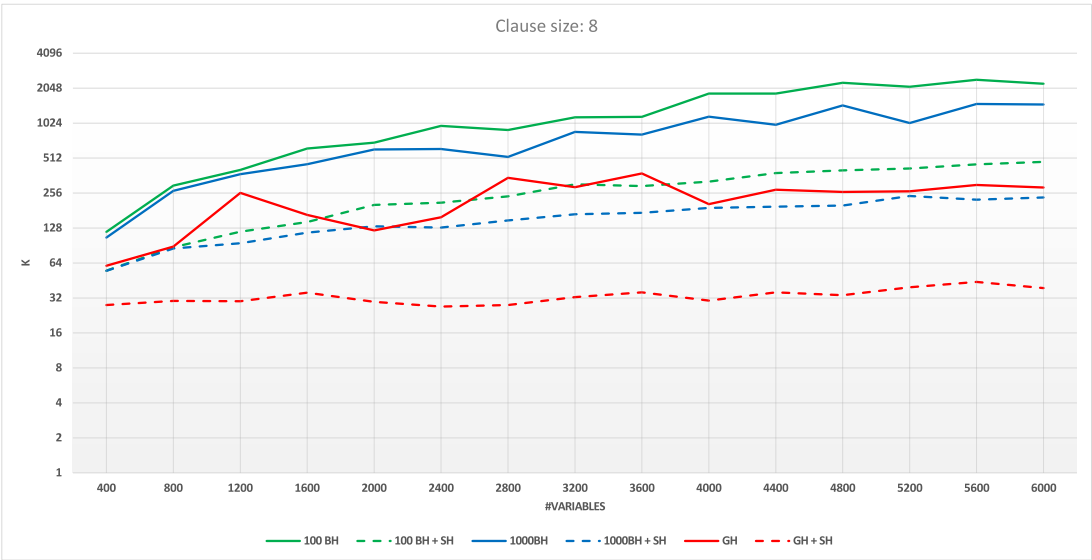

Figure 3.7: Benchmarking permuted type 2 interval ordering instances with clauses containing exactly 8 variables.Horizontal axis represents each instance tested sorted by their number of variables. Vertical axis represents the *k-value* found by the various algorithms.

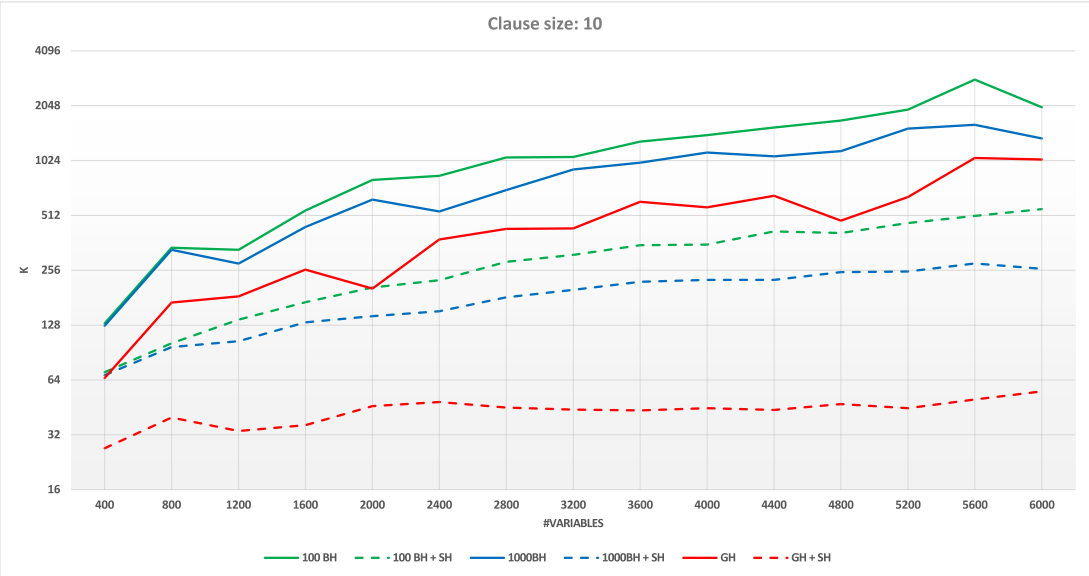

Figure 3.8: Benchmarking permuted type 2 interval ordering instances with clauses containing exactly 10 variables. Horizontal axis represents each instance tested sorted by their number of variables. Vertical axis represents the *k-value* found by the various algorithms.

From figures 3.7 and 3.8 we observe that when increasing the clause size, the k-value increases a great deal. Note that any combination of GH is still the best algorithm, while any combination with BH is the worst.

Figure 3.9: Benchmarking permuted type 2 interval ordering instances with clauses containing exactly 15 variables. Horizontal axis represents each instance tested sorted by their number of variables. Vertical axis represents the *k-value* found by the various algorithms.

From figure 3.9 we observe that when increasing the clause size further, GH+Mk is often worse than BH1000+Mk. However, combining the GH+Mk with SH is still the very best algorithm for

these instances.

In general, any combination using GH performs the best over these types of formulas, especially in combination with our SH algorithm. We think the reason for this is that the GH algorithm was more or less adapted to these specific formulas, since GH performs poorly on most of the other types of instances where $k > 0$, see sections 3.3.1 and 3.3.2.

Chapter 4

SAT encodings for the Car Sequencing Problem

We ended the previous chapter with benchmarking results over many different SAT and artificially constructed instances without going into details of any specific problem. In this chapter we will shed some light on this matter by digging into a problem called *car sequencing*. We follow the paper [6] which introduces some encodings of the car sequencing problem into a CNF formula based on sequential counters. We try to explain the bare minimum for understanding how the car sequencing problem is encoded, as most of the details are nontrivial and irrelevant for our purpose. In the next chapter we continue with benchmarking of some sample problem instances to see if we can achieve a linear ordering for its clauses and variables such that the k-value is low (i. e. it can be solved efficiently by the DP algorithm solving MaxSAT and #SAT).

This chapter is organized as follows. Section 4.1 describes the car sequencing problem formally, and introduces a pseudo boolean model for describing a problem instance. Section 4.2 then explains how this model can be translated into a CNF formula.

4.1 Car Sequncing

The car sequencing problem asks whether if it is possible to schedule a sequence of cars on an assembly line, for which different options are to be installed (e.g. radio, leather-seats, sun-roof, etc.) under certain capacity limitations for each workstation. Cars are divided into classes, such that one class may require radio and sun-roof to be installed, while another class require a different set of options. The problem was first introduced by Parello et al. in 1986 [11] and has been shown to be NP-complete [6]. Each workstation is responsible for installing a single option and its capacity is limited by the number of consecutive cars, it can process. We say more formally that out of q consecutive cars only u can have option o installed.

Before continuing with a more formal definition, we illustrate the problem by an example:

Example 4.1. Given a set of car classes $C = \{1, 2, 3\}$ and options $O = \{\text{radio, sun-roof}\}$. Class requirements are given by $1 = \{\emptyset\}$, $2 = \{\text{radio}\}$ and $3 = \{\text{radio, sun-roof}\}$. The number of cars to be produced are three of class 1, two of class 2 and two of class 3. Workstation $\frac{u_a}{q_a} = \frac{1}{2}$ and $\frac{u_b}{q_b} = \frac{1}{5}$. A valid solution is given by the following sequence: [3,1,2,1,2,1,3]. In fact, this is the only valid solution, since cars of class 2 and 3 cannot be sequenced one after another, and each car of class 3 must be separated by at least 4 cars of other classes.

We let C be the set of different car classes we want to assemble, and O be the set of available options. For each class $k \in C$ we denote the associated options required by O_k and the quantity of cars to be produced by d_k. For an option $l \in O_k$, l can only be installed on u out of q consecutive cars, we will denote this restriction by u_l/q_l. We model this problem by a pseudo Boolean formula as done in [6]. This model serves as a basis for the problem translation into CNF.

• We let the variable c_i^k be set to true if a car of class k is at position i, and a variable o_i^l denotes an option l in position i. A solution to this problem is a total sequence of all cars not breaking any of the subsequence restrictions.

The following equations describe a legal instance:

- Demand constraints: $\forall k \in C$

$$\sum_{i=1}^{n} c_i^k = d_k$$

  For every car class $k \in C$ we must have positioned the exact required amount of cars in class k into the total sequence.

- Capacity constraints: $\forall l \in O$ with ratio u_l/q_l

$$\bigwedge_{i=0}^{n-q_l} (\sum_{j=1}^{q_l} o_{i+j}^l \leq u_l)$$

  In any subsequence of length q_l we cannot install option l more than u_l times.

In every positions $i \in \{1 \dots n\}$ of the sequence it must hold:

- Link between classes and options: $\forall k \in C$

$$\forall l \in O_k : \; c_i^k - o_i^l \leq 0$$
$$\forall l \in O \wedge l \notin O_k : \; c_i^k + o_i^l \leq 1$$

  These two equations force the sequence to install correct options at each part of the sequence. The first equation lets us only install an option l in a class k if there is a car requiring this option there. The second prohibits the sequence of installing options not required by class k.

- We have exactly one car is in position i:

$$\sum_{k \in C} c_i^k = 1$$

4.2 CNF Encoding

Given the pseudo boolean formula of an instance above, we must translate it into a boolean CNF formula. The primary building blocks used for this are cardinality constraints of the form $\sum_{i \in \{1 \dots n\}} x_i = d$ and $\sum_{i \in \{1 \dots n\}} x_i \leq d$.

We start by describing a way to translate the cardinality constraints as a variant of the sequential encoding proposed by [12]. Then we show how to integrate the capacity constraint into these sequential counters. Lastly we show how the demand and capacity constraints are used to encode both classes and the available options to build a complete encoding of the problem.

4.2.1 Sequential Counter Encoding

In this section we provide a way to encode the following cardinality constraint used to represent cumulative sums: $\sum_{i \in \{1 \dots n\}} x_i = d$ where $x_i \in \{0, 1\}$ and $d \in \mathbb{N}$ is some fixed demand.

For each position i in the sequence, we define two types of variables:

1. x_i is set to true if and only if an object x (i. e. class or option) is in position i.
2. $s_{i,j}$ is set to true if and only if for any sequence $[0, 1, \dots i]$, x_i exists at least j times (for technical reasons $0 \leq j \leq d + 1$).

The following set of clauses (1) to (5) define the sequential counter encoding
$\forall i \in \{1 \dots n\}\ \forall j \in \{0 \dots d + 1\}$:

$$\neg s_{i-1,j} \vee s_{i,j} \tag{1}$$

$$x_i \vee \neg s_{i,j} \vee s_{i-1,j} \tag{2}$$

$\forall i \in \{1 \dots n\}\ \forall j \in \{1 \dots d + 1\}$:

$$\neg s_{i,j} \vee s_{i-1,j-1} \tag{3}$$

$$\neg x_i \vee \neg s_{i-1,j-1} \vee s_{i,j} \tag{4}$$

$$s_{0,0} \wedge s_{0,1} \wedge s_{n,d} \wedge \neg s_{n,d+1} \tag{5}$$

The variables $s_{i,j}$ bound the cumulative sums for a sequence $x_1 \dots x_i$. The encoding of each $s_{i,j}$ can be visualized by viewing them as a two dimensional grid where the horizontal line represents

3			U	U	U	U	U	U	U	U	U
2		U	?	?	?	?	?	?	?	?	L
1	U	?	?	?	?	?	?	?	?	L	
0	L	L	L	L	L	L	L	L	L		
$s_{i,j}$	0	1	2	3	4	5	6	7	8	9	10

3			U	U	U	U	U	U	U	U	U
2		U	0	0	0	0	0	1	1	1	L
1	U	0	1	1	1	1	1	1	1	L	
0	L	L	L	L	L	L	L	L	L		
$s_{i,j}$	0	1	2	3	4	5	6	7	8	9	10
x_i	0	0	1	0	0	0	0	1	0	0	0

Table 4.1: Before and after assigning values to the variables x_i

positions in the sequence and the vertical line represents the cumulative sums. The two clauses (1) and (3) enforce the counter to be monotonically increasing(i. e. non-decreasing), while clauses (2) and (4) set the interaction between variables x_i. When x_i is true the counter increases at position i, and when x_i is false we prevent the counter to increase at position i. The last clause (5) sets some initial values for the counter to start counting from 0 and the total sum at position n is equal to d.

Example 4.2. Table 4.1 shows the auxiliary variables $s_{i,j}$ before and after assigning x_2 and x_7 to true. We let $d = 2$ over a sequence of 10 variables. Cells marked with U/L (i. e. upper and lower boundaries), including cells above and below U/L cells, are set to false prior assignment to any x_i. This is simply because they cannot be set to true. Cells marked with question marks are unassigned variables.

4.2.2 Capacity

For each workstation we need to translate the capacity constraints into CNF. For each subsequence of length q, there can be at most u true assignments. We express these capacity constraints by the following sequence of cardinality expressions.

$$\bigwedge_{i=0}^{n-q} \left(\sum_{l=1}^{q} x_{i+l} \leq u \right)$$

These expressions can be translated into CNF by creating individual sequential counters for each subsequence. This will create independent auxiliary variables for each of the subsequences. However, it is possible to encode a more global view into the demand constraint by integrating capacity of each subsequence into the counter. The following expression gives a more global view.

$$\left(\sum_{l=1}^{n} x_{i+l} \leq u \right) \wedge \bigwedge_{i=0}^{n-q} \left(\sum_{l=1}^{q} x_{i+l} \leq u \right)$$

It is possible to reuse the auxiliary $s_{i,j}$ variables used in the sequential counter and create the following set of clauses: $\forall i \in \{q \dots n\}, \forall j \in \{u \dots d+1\}$:

$$\neg s_{i,j} \vee s_{i-q,j-u} \tag{6}$$

These clauses will restrict the internal counting not to exceed the capacity constraints. If the variable $\neg s_{i,j}$ is set to false, then we have not exceeded the capacity restriction. If it is set to true, then we must make sure that the subsequence $[i-q, \dots, i]$ can have an additional u objects.

4.2.3 Link Cars and Options

To complete the CNF translation we need to link the options with the respective car classes. We make the following clauses to retain this relation: $\forall i \in \{1 \dots n\}$,

$$\bigwedge_{\substack{k \in C \\ l \in O_k}} \neg c_i^k \vee o_i^l \tag{7}$$

$$\bigwedge_{\substack{k \in C \\ l \notin O_k}} \neg c_i^k \vee \neg o_i^l \tag{8}$$

The two clauses above follow directly from the pseudo boolean formula in the previous section. In the first clause, if there is a car of class k in position i then the literal $\neg c_i^k$ becomes false and the literal o_i^l must be true to satisfy this clause. This means we only install required options for this class. The other clause makes sure that we do not install any option not required by the class.

$$\bigwedge_{l \in O} \left(\neg c_i^k \vee \bigvee_{k \in C_l} \neg o_i^l \right) \tag{9}$$

The clauses in equation 9 are used to ensure some propagation of variables. We do not explain why this is, but since it is used in all encodings we have included it. Lastly, for each position in the sequence an additional sequential counter is used to make sure that the number of cars is exactly one.

4.2.4 Complete Model

We translate the demand constraints for each class through cardinality constraints. For each option $l \in O$ it is possible to identify the implicit demand by adding up the demand of all classes C_l that require this option.

$$d_l = \sum_{i=1}^{n} o_i^l = \sum_{k \in C_l} d_k$$

By this observation it is possible to use one capacity constraint for every class option since it will also restrict the demand d_k.

The paper [6] provides three CNF encodings E1, E2 and E3 used in their experimental section. We describe them here as they have done, since we will encode problems into all three encodings.

- E1 translates each capacity constraint separately by the clauses (1) to (5) with a fresh set of auxiliary variables.
- E2 translates the capacity constraints by clauses (6) and thus reuses the variables of the sequential counter on the demand constraint.
- E3 combines E1 and E2.

4.2.5 Example encoding

In this section we encode a very simple car sequencing problem for the purpose of exemplifying the CNF encoding.

Example 4.3. Given a set of classes $C = \{1, 2\}$ and a single option $O = \{\text{sun-roof}\}$. Class requirements are given by: class 1 = $\{\emptyset\}$ and class 2 = {sun-roof}. The workstation installing air-condition can handle every second car arriving. For a demand of one class 1 car and two class 2 cars, the only valid sequence would be to separate class 2 cars with a class 1 car, given the following sequence [2,1,2].

If we encode example 4.3 into CNF using the E2 encoding we get the following CNF formula consisting of 35 variables and 90 clauses:

$$(x_1 \vee \bar{x}_2)(\bar{x}_3 \vee x_4)(x_5 \vee x_3 \vee \bar{x}_4)(\bar{x}_2 \vee x_6)(x_5 \vee x_2 \vee \bar{x}_6)(\bar{x}_7 \vee x_8)(x_5 \vee x_7 \vee \bar{x}_8)(\bar{x}_4 \vee x_9)(x_{10} \vee x_4 \vee \bar{x}_9)(\bar{x}_6 \vee x_{11})(x_{10} \vee x_6 \vee \bar{x}_{11})(\bar{x}_8 \vee x_{12})(x_{10} \vee x_8 \vee \bar{x}_{12})(\bar{x}_1 \vee x_2)(x_3 \vee \bar{x}_6)(\bar{x}_5 \vee \bar{x}_3 \vee x_6)(x_2 \bar{x}_8)(\bar{x}_5 \vee \bar{x}_2 \vee x_8)(x_4 \bar{x}_{11})(\bar{x}_{10} \vee \bar{x}_4 \vee x_{11})(x_6 \vee \bar{x}_{12})(\bar{x}_{10} \vee \bar{x}_6 \vee x_{12})(x_3)(\bar{x}_7)(x_4)(\bar{x}_8)(x_{11})(\bar{x}_{12})(x_3 \vee \bar{x}_6)(x_4 \vee \bar{x}_{11})(x_{13} \vee \bar{x}_{14})(\bar{x}_{14} \vee x_{15})(x_{16} \vee x_{14} \vee \bar{x}_{15})(\bar{x}_{17} \vee x_{18})(x_{16} \vee x_{17} \vee \bar{x}_{18})(\bar{x}_{15} \vee x_{19})(x_{20} \vee x_{15} \vee \bar{x}_{19})(\bar{x}_{18} \vee x_{21})(x_{20} \vee x_{18} \vee \bar{x}_{21})(\bar{x}_{13} \vee x_{14})(x_{14} \vee \bar{x}_{18})(\bar{x}_{16} \vee \bar{x}_{14} \vee x_{18})(x_{15} \vee \bar{x}_{21})(\bar{x}_{20} \vee \bar{x}_{15} \vee x_{21})(x_{14})(\bar{x}_{17})(x_{15})(\bar{x}_{18})(x_{21})(\bar{x}_{22})(x_{14} \vee \bar{x}_{21})(x_{23} \vee \bar{x}_{24})(\bar{x}_{24} \vee x_{25})(x_{26} \vee x_{24} \vee \bar{x}_{25})(\bar{x}_{27} \vee x_{28})(x_{26} \vee x_{27} \vee \bar{x}_{28})(\bar{x}_{25} \vee x_{29})(x_{30} \vee x_{25} \vee \bar{x}_{29})(\bar{x}_{28} \vee x_{31})(x_{30} \vee x_{28} \vee \bar{x}_{31})(\bar{x}_{23} \vee x_{24})(x_{24} \vee \bar{x}_{28})(\bar{x}_{26} \vee \bar{x}_{24} \vee x_{28})(x_{25} \vee \bar{x}_{31})(\bar{x}_{30} \vee \bar{x}_{25} \vee x_{31})(x_{24})(\bar{x}_{27})(x_{25})(\bar{x}_{28})(x_{31})(\bar{x}_{32})(x_{24} \vee \bar{x}_{31})(\bar{x}_1 \vee \bar{x}_{23})(\bar{x}_5 \vee \bar{x}_{26})(\bar{x}_{10} \vee \bar{x}_{30})(\bar{x}_{13} \vee x_{23})(\bar{x}_{16} \vee x_{26})(\bar{x}_{20} \vee x_{30})(\bar{x}_{23} \vee x_{13})(\bar{x}_{26} \vee x_{16})(\bar{x}_{30} \vee x_{20})(\bar{x}_1 \vee x_{33})(\bar{x}_{13} \vee \bar{x}_{33})(x_1 \vee x_{13})(\bar{x}_5 \vee x_{34})(\bar{x}_{16} \vee \bar{x}_{34})(x_5 \vee x_{16})(\bar{x}_{10} \vee x_{35})(\bar{x}_{20} \vee \bar{x}_{35})(x_{10} \vee x_{20})$$

As we see the CNF encodings given in [6] for the car sequencing problem generates very large formulas. It would be interesting to see if some other encoding could be given, maybe one geared more directly to maintaining the linear constraints. We leave this for future work.

Chapter 5

Benchmarking - Car Sequencing

In this chapter we inspect the CNF encoding of the car sequencing problem presented in chapter 4 and analyse its complexity with respect to k-value. Our hope is that the linearity requirement (i. e. cars are aligned onto a linear assembly line) of the car sequence problem gives us a boolean formula with low k-value. To encode a problem instance into CNF we use the CNF-SAT encoder provided by [6]. Section 5.1 describes the car sequencing problem instances as provided by CSPLib. In section 5.2 we encode some instances into CNF and measure their *k-value* using the same algorithms as in section 3.2. We observe that the *k-value* of these problems increases as the instance size increases e. g. more cars, options or classes. Given this result we continue in section 5.3 with testing sub-formulas of these encodings, and observe that some of them are quite linear, i. e. k-value is low for capacity and cardinality sub-formulas. In section 5.4 we summarize the results.

5.1 CNF-SAT encoder

The CNF-SAT encoder lets us encode instances in all of the three encodings mentioned in chapter 4. It also lets us separate the cardinality and capacity constraints into disjoint boolean formulas allowing us to perform benchmarking separately.

The CNF-SAT encoder takes as input a data file in the following format:

- First line: number of cars; number of options; number of classes.
- Second line: capacity constraint for each option l in increasing order. i. e. the number of cars u that can have option l installed in a sequence of consecutive cars q.
- Third line: for each of the capacity constraints l, the maximum block size q in increasing order.
- A line for each different class containing: class index number x; number of cars in class x; and for each different option: 1 or 0 depending on whether class x requires the following option.

The following table illustrates the data file format:

10	5	6				
1	2	1	2	1		
2	3	3	5	5		
0	1	1	0	1	1	0
1	1	0	0	0	1	0
2	2	0	1	0	0	1
3	2	0	1	0	1	0
4	2	1	0	1	0	0
5	2	1	1	0	0	0

Table 5.1: Example of a car sequencing problem with 10 cars, 5 options and 6 classes. The capacity constraints are as follows from the first to the last option: $\frac{1}{2}, \frac{2}{3}, \frac{1}{3}, \frac{2}{5}$ and $\frac{1}{5}$. Each of the next lines describes the number of cars and options for a specific class. Example was given in [13].

5.2 Complete instances

The CNF-SAT encoder includes several problem instances where the number of cars varies from 100 to 400. Each of these instances are configured with roughly the same number of options and classes, 5 and 25, respectively. Instances with more than 100 cars quickly became too large for us to benchmark when translated into CNF. As a result, we decided to alter some of these larger instances by reducing the number of cars. We start by translating two problems into the three different encodings (E1, E2 and E3) to provide some insight in what type of encoding tends to have the lowest k-value. We continue by increasing the number of options and/or classes to see if this increases the k-value. The last instances we benchmark in this section are various unmodified problems instances with 100 cars.

The following figures display the results after running the various algorithms on CNF encoded car sequencing problems.

Figure 5.1: Results over two car sequencing problem instances of 25 cars encoded in all three encodings (E1, E2 and E3). The Horizontal axis represents the instances tested, while the vertical axis represents the *k-value* found by various algorithms. The black bar in the chart represents an upper boundary for the k-value, in terms of the number of variables in each instance.

In figure 5.1 we have encoded modified P09 and P18 such that they only encode 25 cars. We see

that the best algorithm for minimizing k-value is BH10000+Mk+SH. The worst is SO+Mk. GH+Mk and GH+Mk+SH is performing almost as poorly as SO+Mk. The k-values for these problems are very high.

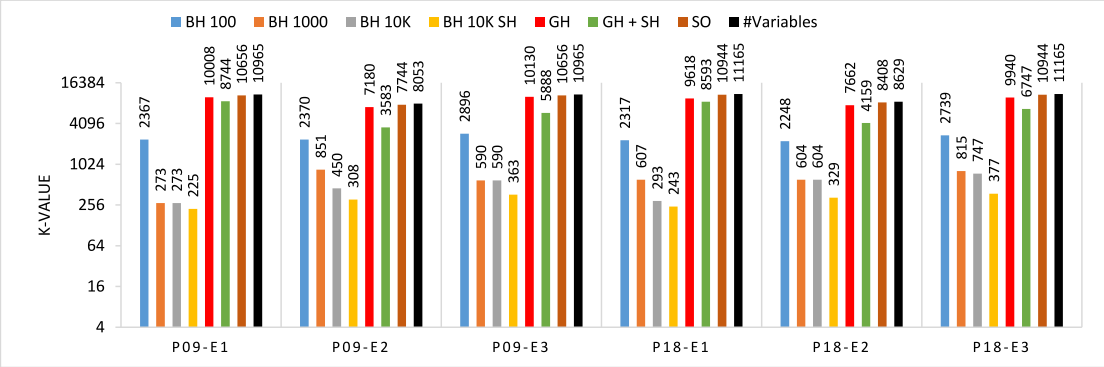

Figure 5.2: Results over two car sequencing problem instances of 50 cars encoded in all three encodings (E1, E2 and E3). The Horizontal axis represents the instances tested, while the vertical axis represents the *k-value* found by various algorithms. The black bar in the chart represents an upper boundary for the k-value, in terms of the number of variables in each instance.

By increasing the number of cars in the same instances as in figure 5.1 we achieve even worse results. The algorithms performance is similar to the previous ones.

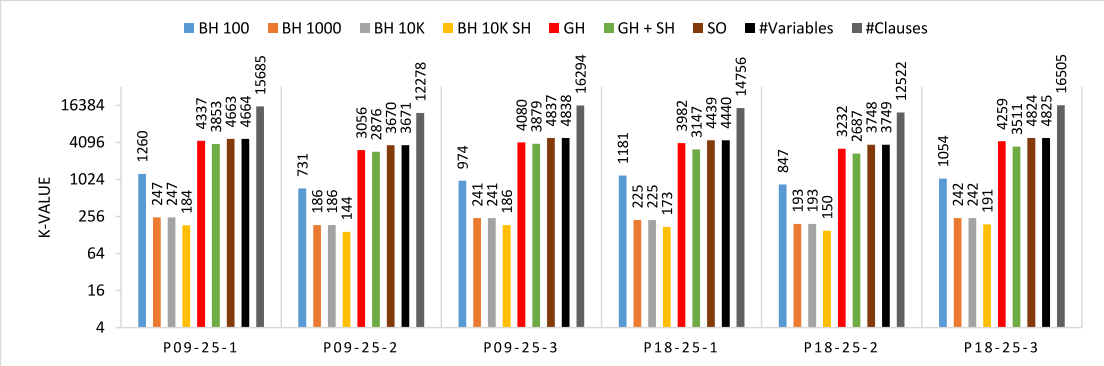

Figure 5.3: Results over two car sequencing problem instances of 25 cars encoded in all three encodings (E1, E2 and E3). The Horizontal axis represents the instances tested, while the vertical axis represents the *k-value* found by various algorithms. The black bar in the chart represents an upper boundary for the k-value, in terms of the number of variables in each instance. The instances are marked with 1, 2 and 3. 1: increased options from 5 to 8. 2: increased classes from 10 to 13. 3: 1 and 2 combined.

In figure 5.3 we have increased the number of options and classes in the same instances as in figure 5.1. We generally observe that the k-value is increased.

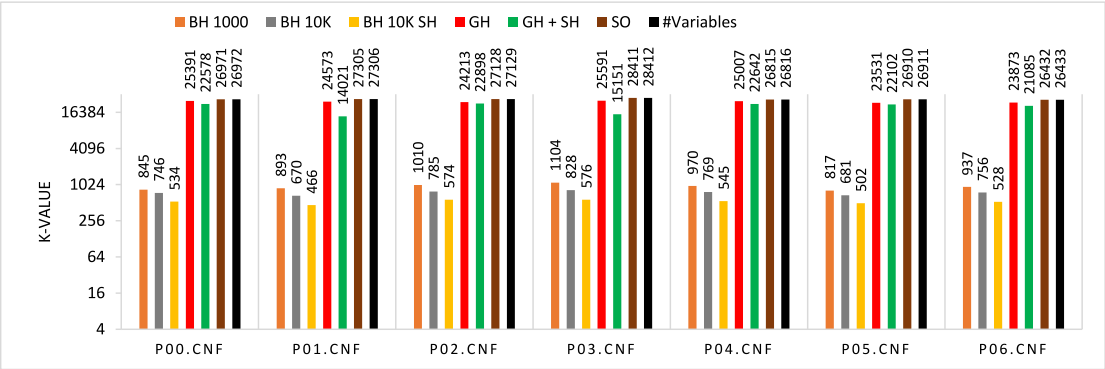

Figure 5.4: Results over two car sequencing problem instances of 100 cars encoded in E1. The Horizontal axis represents the instances tested, while the vertical axis represents the *k-value* found by various algorithms. The black bar in the chart represents an upper boundary for the k-value, in terms of the number of variables in each instance.

When benchmarking some original instances, we see that the *k-value* increases even further, giving us little hope for achieving low k-value for these encodings in general. Instead, we continue with focusing on sub-formulas of these encodings in the next section, hoping that this achieves better results.

5.3 Capacity and Cardinality constraints

In the two first figures in this section we have translated the capacity and cardinality constraints for all of the instances we benchmarked in figures 5.1 and 5.2. Given that we achieved good results on these, we decided to see if this can be scaled to larger instances. For this we chose the two first instances in figure 5.4.

The following naming convention is used in the figures.

- CA are capacity constraint clauses.
- CNT are cardinality constraint clauses.
- CNT+CA are the two above combined.

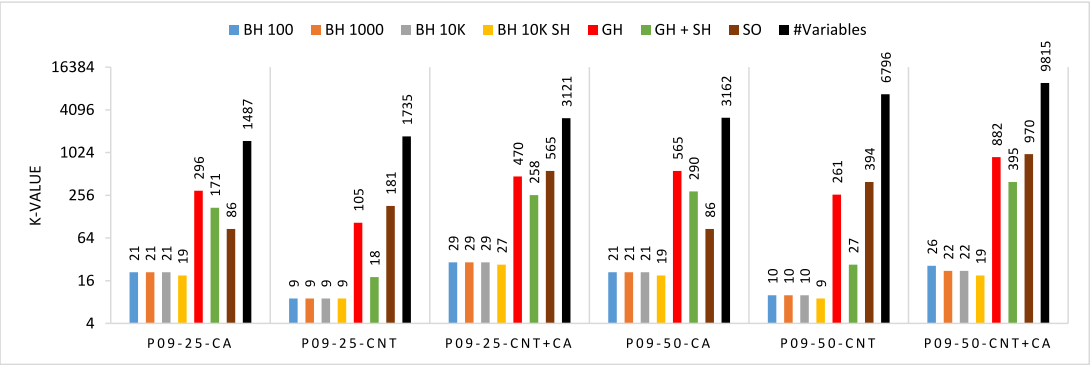

Figure 5.5: Results over cardinality and capacity constraints of car sequencing instances with 25 and 50 cars encoded in E1. The Horizontal axis represents the instances tested, while the vertical axis represents the *k-value* found by various algorithms. The black bar in the chart represents an upper boundary for the *k-value* , in terms of the number of variables in each instance.

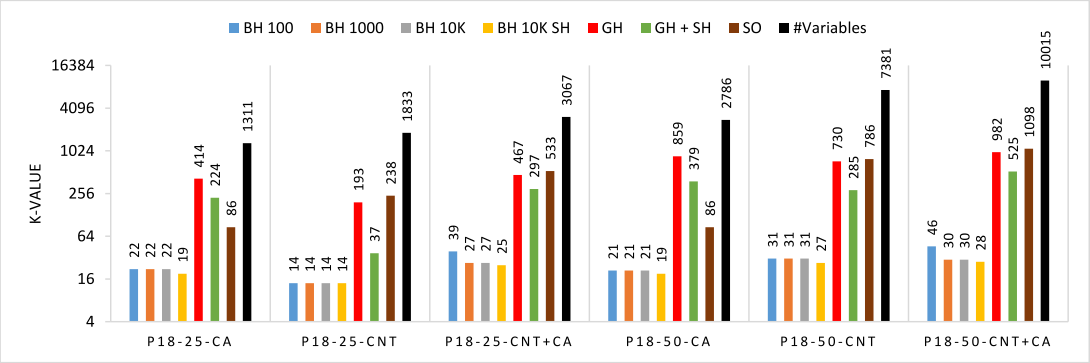

Figure 5.6: Results over cardinality and capacity constraints of car sequencing instances with 25 and 50 cars encoded in E1. The Horizontal axis represents the instances tested, while the vertical axis represents the *k-value* found by various algorithms. The black bar in the chart represents an upper boundary for the *k-value* , in terms of the number of variables in each instance.

The two figures above show that these types of sub formulas can achieve a very low k-value for small instances. Any combination with BH works best. GH+Mk, GH+Mk+SH and SO perform poorly for most of these formulas. In the next figure we increase the size of formulas, but only use the best algorithms from the above figures.

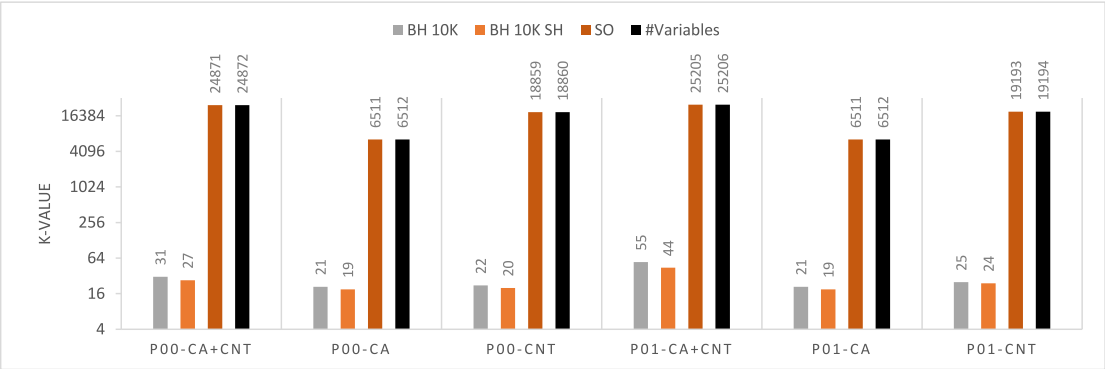

Figure 5.7: Results over cardinality and capacity constraints of car sequencing instances 100 cars encoded in E1. The Horizontal axis represents the instances tested, while the vertical axis represents the *k-value* found by various algorithms. The black bar in the chart represents an upper boundary for the *k-value* , in terms of the number of variables in each instance.

For capacity and cardinality constraints the k-values are very low when using BH+Mk and BH+Mk+SH. This proves that our heuristics work very well for some types of CNF formulas, especially these.

5.4 Conclusion

As the results reveal in this chapter, the k-values found by various algorithms for each of the complete instances is way too high for us to efficiently solve them by the DP algorithm for solving MaxSAT and #SAT. The reason for this is likely due to: 1) the heuristics we developed for minimizing the k-values are not good enough for these CNF formulas, and 2) these CNF formulas cannot achieve low k-value because the interaction between the clauses and variables are not very linear in respect of k-value. For capacity and cardinality formulas the variables and clauses interact very tightly, i. e. any clause in the ordering has mostly consecutive variables. In this way the heuristics works great. We considered to check what types of instances of the car sequencing problem is typical in the industry, but given we did not achieve low k-value for any of the complete instances we decided it was not valuable. We hope that the cardinality and capacity formulas have some practical use in some other situations.

Chapter 6

Conclusions

In this chapter we summarize the thesis results, and present open problems for future work.

6.1 Summary

In this thesis we have studied a class of propositional boolean formulas for which a DP algorithm [3] may be used to solve MaxSAT and #SAT efficiently. To be more precise, we implemented various algorithms for testing CNF formulas for linearity, and then experimented with these algorithms on various CNF formulas hoping to find practical applications for which this method is favourable.

In chapter 2 we have presented most of the algorithms used for experimentation throughout this thesis. We gave two new algorithms for recognizing CNF formulas that have an interval ordering given restrictions on the variables, and/or for the clauses. Since we do not expect that there exists many practical applications that have an interval ordering we provided various heuristics for computing a *k-interval ordering* on the CNF formula, where hopefully, the k-value is low enough to be efficiently solved by the dynamic programming algorithm for solving MaxSAT and #SAT. In the end of the chapter we prove a theoretical result that the ps-width of a CNF formula F is at most $m + 1$. This result was given implicit in [3], but we have provided a full proof.

In chapter 3 we first gave the SH algorithm for optimizing the linear ordering that was produced by the Mk algorithm. We observed that SH can make a huge difference when solving MaxSAT and #SAT with the DP algorithm in [3]. It also had a nice side effect of reducing the *k-value* when we allowed it to shift passed clauses, and thereby reordering the clauses. Through the experiments we conducted we observed that the combined algorithms BH10000+Mk+SH was generally the best algorithm in terms of achieving lowest possible *k-value* for instances taken from SAT competitions. By using this algorithm we where able to discover a few problems for which the *k-value* was very low. The BH10000+Mk+SH algorithm does not perform as well for the artificially constructed formulas having an interval ordering as the GH+Mk+SH algorithm. The conclusion we draw from this is that none of the algorithms we tested are overall the best algorithm for minimizing the *k-value* on all instances. We suggest an improvement for this in the open problems section.

The results from the experiments conducted in chapter 5 show that the particular encodings we presented for the car sequencing problem does not seem to have a low *k-value* for the car sequencing problem in general. Even for very small instances the *k-value* we achieved by the various algorithm is much too high for DP algorithm for solving MaxSAT and #SAT to handle. However, when applying the various algorithm for sub-formulas of these encodings, particularly the cardinality and capacity constraints clauses, we observed very good results. The best algorithm for minimizing the *k-value* on all tested instances in this chapter is the BH+Mk+SH algorithm, while the algorithms GH+Mk and GH+Mk+SH performed very poorly, often the *k-value* was as high as the number of variables in the instance.

6.2 Open Problems

In this section we list open problems that came up while writing this thesis:

- In chapter 2 we developed an exact merging algorithm for finding an interval ordering in q-CNF formulas when given a linear ordering on the variables only. This algorithm runs in time $\mathcal{O}(q \cdot m^3)$, but we believe that this can improved in the future with introducing a more sophisticated data structure and new techniques.

- In the experimental section we found a few instances which had a low k-value found by the various algorithms. These instances encode two different problems called *Automata synchronization* and *Symbolic simulation* for which the DP algorithm for solving MaxSAT and #SAT may be of practical use. One could compare our approach against other solvers to see if our approach is more favourable.

- We noticed that for the algorithms we have presented in this thesis the results varies much on the various CNF formulas tested (e. g. any combination with GH performs very well on the artificially generated formulas, while BH+Mk+SH is generally the best for SAT and car sequencing problems). To achieve better results in general, one could try to make a hybrid algorithm out of GH and BH, i. e. let GH produce a linear ordering of the clauses and variables for which we extract the two linear orderings on the clauses and variables and run BH+Mk+SH on them. This could possibly be a better input to the BH algorithm instead of creating the orders as they appear in the CNF formula.

- In the results from car sequencing problems we experienced that the BH+Mk+SH algorithm performs very well on formulas encoding capacity and cardinality constraints. These formulas may be of further interest, as they might be used in other problems where we can achieve low k-value over the complete instance. One could go into the details of this encoding and study for

which kind of problems they are used (i. e. the sequential counter clauses), and then perform similar experiments as we have done in this thesis.

- It would be interesting to see if some other encoding could be given for the car sequencing problem, maybe one geared more directly to maintaining the linear constraints.
- It would be interesting to perform the same tests as we have done on CNF formulas where the MaxSAT and #SAT problem is more relevant than instances from the SAT competition.
- The memory issues we have with the DP algorithm for solving MaxSAT and #SAT prohibited us for measuring the k-value of CNF instances much larger than 1MB. One could create a dedicated program for this purpose only, and thereby avoiding the large data structures necessary for solving MaxSAT and #SAT.

Appendix A

Code repository

The algorithms proposed in this thesis have been implemented into the code developed by Sigve Hortemo Sæther for solving MaxSAT and #SAT by dynamic programming on structured CNF formulas. The extended code can be found on GitHub [14], while the code for the dynamic programming algorithm for solving MaxSAT and #SAT was available at `http://people.uib.no/ssa032/pswidth/`.

Bibliography

[1] H. Kautz and B. Selman. Planning as Satisability. *Proceedings ECAI-92*, 1992.

[2] Edmund Clarke Armin Biere, Alessandro Cimatti and Yunshan Zhu. Symbolic model checking without BDDs. *TACAS'99*, 1999.

[3] Jan Arne Telle Sigve Hortemo Sæther and Martin Vatshelle. Solving #sat and MaxSAT by dynamic programming. 2015.

[4] Sigve Hortemo Sæther Serge Gaspers, Christos Papadimitriou and Jan Arne Telle. On satisfiability problems with a linear structure. 2015.

[5] S. Tagawa K. Sugiyama and M Toda. Methods for visual understanding of hierarchical systems. *IEEE Transactions on Systems, Man, and Cybernetics, SMC- 11(2)*, pages 109–125, 1981.

[6] Valentin Mayer-Eichberger and Toby Walsh. SAT Encodings for the Car Sequencing Problem. 2013.

[7] Haiko Muller. Recognizing interval digraphs and interval bigraphs in polynomial time. 1997.

[8] Arash Rafiey. Recognizing interval bigraphs by forbidden patterns. 2012.

[9] Charles E.; Rivest Ronald L.; Stein Clifford (2001) Cormen, Thomas H.; Leiserson. Section 22.4: Topological sort. pages 549–552, 2001.

[10] Michael Sipser. Introduction to the Theory of Computation. 2013.

[11] Waldo C. Kabat Bruce D. Parello and L. Wos. Job-shop scheduling using automated reasoning: a case study of the car sequencing problem. *Journal of Automated Reasoning 2*, pages 1–42, 1986.

[12] Carsten Sinz. Towards an Optimal CNF Encoding of Boolean Cardinality Constraints. *CP*, pages 827–831, 2005.

[13] H. Simonis M. Dincbas and Pascal Van Hentenryck. Solving the car sequencing problem in constraint logic programming. *European Conference on Artificial Intelligence (ECAI-88)*, 1988.

[14] Christian Egeland. Extensions for the dp algorithm for solving MaxSAT and #sat developed by sigve hortemo sæther, 2016.