Hashing-based delayed duplicate detection as an approach to improve the scalability of optimal Bayesian network structure learning with external memory frontier breadth-first branch and bound search

Niklas Jahnsson
Bayesian networks are graphical models used to represent the joint probability distribution for all variables in a data set. A Bayesian network can be constructed by an expert, but learning the network from the data is another option. This thesis is centered around exact score-based Bayesian network structure learning. The idea is to optimize a scoring function measuring the fit of the network to the data, and the solution represents an optimal network structure.

The thesis adds to the earlier literature by extending an external memory frontier breadth-first branch and bound algorithm by Malone et al. [MYHB11], which searches the space of candidate networks using dynamic programming in a layered fashion. To detect duplicates during the candidate solution generation, the algorithm uses efficiently both semiconductor and magnetic disk memory. In-memory duplicate detection is performed using a hash table, while a delayed duplicate detection strategy is employed when resorting to the disk. Delayed duplicate detection is designed to work well against long disk latencies, because hash tables are currently still infeasible on disk. Delayed duplicate detection allows the algorithm to scale beyond search spaces of candidate solutions fitting only in the comparatively expensive and limited semiconductor memory at disposal.

The sorting-based delayed duplicate detection strategy employed by the original algorithm has been found to be inferior to a hashing-based delayed duplicate strategy in other application domains [Kor08]. This thesis presents an approach to use hashing-based delayed duplicate detection in Bayesian network structure learning and compares it to the sorting-based method. The problem faced in the hashing of candidate solutions to disk is dividing the candidate solutions in a certain stage of the search in an efficient and scalable manner to files on the disk. The division presented in this thesis distributes the candidate solutions into files unevenly, but takes into account the maximum number of candidate solutions that can be held in the semiconductor memory. The method works in theory as long as operating system has free space and inodes to allocate for new files.

Although the hashing-based method should in principle be faster than the sorting-based, the benchmarks presented in this thesis for the two methods show mixed results. However, the analysis presented also highlights the fact that by improving the efficiency of, for example, the distribution of hashed candidate solutions to different files, the hashing-based method could be further improved.
## Contents

1 Introduction 1

2 Background 3
   2.1 Bayesian networks ............................................. 3
   2.2 Score-based learning ........................................... 4
      2.2.1 Scoring functions ........................................ 5
      2.2.2 Dynamic programming ..................................... 6
      2.2.3 Graph-based search approach ............................ 7
      2.2.4 Order graph ................................................. 7
      2.2.5 Sparse parent graph ....................................... 8
      2.2.6 Heuristic function ......................................... 11

3 Delayed duplicate detection 12
   3.1 Sorting-based delayed duplicate detection .................. 12
   3.2 Hashing-based delayed duplicate detection ................. 13
   3.3 Hybrid duplicate detection .................................. 14

4 Using delayed duplicate detection with hashing in Bayesian network structure learning 15
   4.1 First-\(k\) subcombinations .................................... 16
   4.2 Selecting the number of variables used in a first-\(k\) subcombination .. 18
   4.3 Dividing candidate order graph nodes to files on the disk .... 19
   4.4 Distribution of candidate order graph nodes to different files on the disk ................................................. 20

5 Experimental results 24
   5.1 Implementation details ......................................... 24
   5.2 Environment ..................................................... 24
   5.3 Data used ....................................................... 25
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.4 Overall performance</td>
<td>26</td>
</tr>
<tr>
<td>5.5 Scenario with 20 variables</td>
<td>26</td>
</tr>
<tr>
<td>5.5.1 Locality</td>
<td>29</td>
</tr>
<tr>
<td>5.6 Scenario with 29 variables</td>
<td>30</td>
</tr>
<tr>
<td>5.6.1 First set of results under light load</td>
<td>31</td>
</tr>
<tr>
<td>5.6.2 Second set of results under heavy load</td>
<td>39</td>
</tr>
<tr>
<td>5.6.3 Locality</td>
<td>47</td>
</tr>
<tr>
<td>6 Conclusions</td>
<td>49</td>
</tr>
<tr>
<td>References</td>
<td>51</td>
</tr>
</tbody>
</table>
1 Introduction

Bayesian networks are probabilistic graphical models used to represent relationships between variables in data sets [Pea88]. A Bayesian network is a directed acyclic graph: its nodes represent the variables in the dataset and its edges represent the direct dependencies among the variables. In other words, a Bayesian network represents the joint probability distribution of all the variables in it.

To use Bayesian networks in practice, a Bayesian network representing the data needs to be constructed. When lacking experts to construct the network, an option is to learn the network from data. There are different ways to learn a network from the data, and this thesis is centered around score-based Bayesian network structure learning [HGC95], in which a scoring function is used to measure how well a certain Bayesian network structure fits the data. The goal in score-based learning is to find a network structure that optimizes the value of the scoring function.

With the score-based learning framework, the earlier literature has utilized both approximate [Bou94, Hec96] and exact methods [KS04, SM06, JSGM10, dCJ11] to learn a Bayesian network. One of the major motivations for exact methods is that the solutions found by the approximate learning methods have an unknown quality whereas the exact methods are guaranteed to provide an optimal solution.

This thesis adds to the earlier literature by extending an external memory frontier breadth-first branch and bound algorithm by Malone et al. [MYHB11]. The algorithm by Malone et al. will be later on referred to as MYHB algorithm, and it searches the space of candidate subnetworks using dynamic programming in a layered fashion. This enables the search to efficiently use both semiconductor memory and also to resort to magnetic disk memory if needed. When searching, duplicates corresponding to subnetworks over the same set of variables are created, and they are detected using an in-memory hash table. Only the best subnetwork is retained. This process is called duplicate detection, and it is a key operation in the algorithm. However, this also means that when resorting to disk, the algorithm needs to employ a duplicate detection strategy to make sure that duplicates on disk are detected.

Delayed duplicate detection is a process utilized in artificial intelligence algorithms employing external memory to detect duplicates in generated candidate solutions during, for example, a brute force search. It replaces a hash table in semiconductor memory, and it is designed to work well despite long disk latencies when searching on a magnetic disk storage. This is important because hash tables are currently
infeasible on disk. In other words, delayed duplicate detection enables algorithms to search spaces of candidate solutions which would not fit the semiconductor memory at disposal.

The MYHB algorithm uses a sorting-based delayed duplicate detection algorithm, which sorts and writes the candidate solutions to disk every time the memory fills up. In the end of the search for a single layer the duplicates can be detected by reading the candidate solutions in an orderly fashion from the files on the disk into memory. However, in the earlier literature a hashing-based delayed duplicate detection has outperformed the sorting-based method, for example, in a breadth-first search of Towers of Hanoi [Kor08]. In principle, the hashing-based method runs in $O(n)$ time in the number of candidate solutions, while the sorting based method requires $O(n \log n)$ time. This suggests that it might be beneficial to apply the hashing-based method also in the structural learning framework for Bayesian networks.

A version of the algorithm using a hashing-based delayed duplicate detection is presented in this thesis and compared against the MYHB algorithm. The problem faced in the hashing of candidate solutions to disk is dividing the candidate solutions in a certain stage of the search in efficient and scalable way to files on the magnetic disk. The division presented in this thesis distributes the candidate solutions into files unevenly, but takes into account the maximum number of candidate solutions that can be held in the semiconductor memory. In other words, the method should in theory be able to scale up as long as the operating system has free space and inodes to allocate for new files. Although hashing-based method should in principle be faster than sorting-based, the benchmarks presented for the hashing and sorting-based algorithms show mixed results.

The thesis is organized as follows. The next section will present the relevant background in Bayesian networks and structure learning. After that, delayed duplicate detection is introduced, and its application to structure learning is outlined. Section 4 defines one way to implement hashing-based delayed duplicate detection in Bayesian network structure learning context and analyzes briefly the way the chosen method divides to candidate solutions to the disk in practice. Section 5 will introduce the experimental setup and showcase the results. The last section concludes the thesis.
2 Background

This section briefly summarizes the earlier work related to structure learning of Bayesian networks and introduces the concepts utilized later on. First, a random variable is a variable that may take on one of several possible outcomes, or values, from a specific domain. The following will assume that random variables refer to categorized random variables, i.e., random variables taking values from finite unordered domains.

Notation will follow mostly the standards in the literature as presented, for example, by Heckerman [Hec96]. Random variables will be referred with upper-case letters, for example $X_1$, and their states with same letter in lower case, for example $x_1$. A set of random variables will be referred by a bold-face, upper-case letter, for example $V$, and a state of a set with each variable assigned to a particular value in the set will be referred to with a corresponding bold-face, lower-case letter, for example, $v$. In the previous, it is assumed that the set of random variables $V$ is finite with size $n \in \mathbb{N}$.

The probability of seeing random variable $X_1$ having value $x_1$ will be denoted by $p(X_1 = x_1)$, and as a shorthand for this $p(x_1)$ will be used. Similarly, notation $p(x_1|\xi)$ will be used as a shorthand of $p(X_1 = x_1|\xi)$ to denote the conditional probability that $X_1 = x_1$ given we know state information $\xi$. Note that in addition to using $p(\cdot)$ as a shorthand for probability, it will also be used to denote probability distribution and probability mass function; the usage will be clear from the context.

2.1 Bayesian networks

A Bayesian network is defined as a directed acyclic graph (DAG) $G$ as depicted by Figure 1. The vertices of the graph $G$ represent single random variables from a set of random variables $V = \{X_1, X_2, ..., X_n\}$ with $n \in \mathbb{N}$. The edges of the graph $G$ represent dependencies between two variables: for $i, j \in [1, n]$ a directed edge from node $X_i$ to node $X_j$ means that node $X_i$ is a parent of node $X_j$ or, in other words, node $X_j$ is a child of node $X_i$. Parent set $PA_i$ for node $X_i$ is defined to contain all the parents of node $X_i$. Bayesian network factorization assumes that given its parent set $PA_i$, node $X_i$ is independent from its non-descendants [Pea88].

Relations between the variables in the network are quantified by a set of conditional probability distributions. In a Bayesian network each variable $X_i$ has its own con-
Additional probability distribution \( p(X_i|\text{PA}_i) \) conditioning on its parents \( \text{PA}_i \). As a whole, a Bayesian network is used to represent a joint probability distribution over a set of random variables \( \mathcal{V} \). Due to the independence of a single node from its non-descendants given its parents, the joint probability distribution can be factorized as the product of all the individual nodes’ conditional distributions as follows

\[
p(x_1, x_2, ..., x_n) = \prod_{i=1}^{n} p(x_i|\text{pa}_i).
\]

(1)

Overall, Bayesian networks have been used earlier in the literature to solve various inference tasks. There are roughly three approaches to tackle a Bayesian network structure learning task: score-based learning, constraint-based learning and hybrid methods [YM13]. Score-based learning measures the different network structures using a scoring function and selects the one having the best score [HGC95, Hec96]. On the other hand, constraint-based learning methods try to find conditional independence relationships from the data using statistical tests, for example, for causality, and then use the found relationships to build up the network structure [Pea88, Pea00]. Hybrid methods are combinations of the two earlier methods. Next score-based learning will be explained in more detail as it is the paradigm followed in this thesis.

### 2.2 Score-based learning

Learning a Bayesian network is defined as a quest to find a Bayesian network structure that best fits the data. In other words, given a scoring function \( \text{score}(\cdot) \), the task is to find a network structure \( G^* \) fitting the data best

\[
G^* = \arg \min_G \text{score}(G).
\]

(2)
It should be noted that whether the argument in Equation 2 is minimized or maximized depends on the scoring function used.

Overall, Chickering [Chi96] has shown that score-based learning with a typical scoring function is NP-hard, and early research in this area mainly focused in using approximation algorithms. However, recently also multiple exact learning algorithms have been developed based on, for example, dynamic programming [SM06] and linear integer programming paradigms [JSGM10]. The benefit of exact learning algorithms is that the quality of the solution they provide is by construction known to be optimal whereas the quality of the solution provided by the approximation algorithms is unknown.

Dataset $D$ used to learn a network structure can be defined as a set of vectors $D = \{D_1, D_2, ..., D_N\}$ with $N \in \mathbb{N}$ [YM13]. This definition assumes that each datapoint $D_k$ with $k \in \mathbb{N}$ is a vector of values over variables in set $V$. In addition, it is assumed that each variable $X_i$ in set $V$ is categorical and has only a finite number of possible values. It is also assumed that no data point is missing values.

Score-based learning can be divided to two elements: scoring function and search strategy [YM13]. A scoring function defines the quality of the solution while the search strategy decides how the candidate solutions are generated and how the scoring function is optimized.

### 2.2.1 Scoring functions

A scoring function $\text{score}(\cdot)$ measures how well a network structure fits the given data. Effectively, it takes as parameters the observed data and the Bayesian network structure. Given its parameters, a scoring function returns a score, which reflects how well the data fits the given network structure. For any decomposable scoring function the score of the network can be defined as the sum of the scores for the nodes in the network [Hec96]

$$\text{score}(G) = \sum_{i=1}^{n} \text{score}(X_i|\text{PA}_i).$$  \hspace{1cm} (3)

Two network structures are equivalent if the set of distributions that can be represented using one of the networks is identical to the set of distributions that can be represented using the other [Chi95]. If this is the case, then the two network structures belong to the same equivalence class. Scoring functions can be score
equivalent, which means that they assign the same score to each network structure in the same equivalence class. We now define a scoring function using minimum description length principle (MDL) as its metric, and this scoring function is known to be score equivalent [Chi95].

Given the number of states for \( X_i \) as \( r_i \), the number of data points consistent with \( PA_i = pa_i \) as \( N_{pa_i} \), and number of data points further having \( X_i = x_i \) as \( N_{x_i,pa_i} \), we can define the MDL scoring function as follows [LB94]

\[
\text{MDL}(G) = \sum_{i=1}^{n} \text{MDL}(X_i|PA_i),
\]

where

\[
\text{MDL}(X_i|PA_i) = H(X_i|PA_i) + \frac{\log(N)}{2} K(X_i|PA_i),
\]

\[
H(X_i|PA_i) = - \sum_{x_i,pa_i} N_{x_i,pa_i} \log(\frac{N_{x_i,pa_i}}{N_{pa_i}}),
\]

\[
K(X_i|PA_i) = (r_i - 1) \prod_{X_i \in PA_i} r_l.
\]

With MDL the goal is to find the network that minimizes the MDL score. Note that for other scoring functions the task might be maximization, but it can be transformed to minimization by just by switching the sign of the score.

The idea behind the MDL principle is to find a model which minimizes the sum of the encoding lengths of the data and the model itself [LB94]. In other words, the MDL principle is a formalization of the well known Occam’s razor principle, which states that among competing hypotheses, the one with the simplest explanation should be selected. In the above equations function \( H(\cdot) \) represents the encoding of the data, while the function \( K(\cdot) \) represents the encoding of the model.

### 2.2.2 Dynamic programming

To find an optimal Bayesian network for a set of variables \( V \), it is sufficient to find the best choice of a variable \( X_i \) as a leaf [YM13]. This effectively means that for any variable \( X_i \) chosen as the best leaf, the best possible Bayesian network can be created by choosing its optimal parent set \( PA_{X_i} \) from the rest of the variables.
and constructing an optimal subnetwork for the rest of the variables $V \setminus \{X_i\}$. This means that the optimal leaf is such that the sum of $\text{score}(X_i, \text{PA}_{X_i})$ and $\text{bestscore}(V \setminus \{X_i\})$ are minimized, where $\text{bestscore}(\cdot)$ is used here to select the optimal parent set for variable $X_i$. In other words, we have the following relation

$$\text{score}(V) = \min_{X_i \in V} \{\text{score}(V \setminus \{X_i\}) + \text{bestscore}(X_i, V \setminus \{X_i\})\},$$

where

$$\text{bestscore}(X_i, V \setminus \{X_i\}) = \min_{\text{PA}_{X_i} \subseteq V \setminus \{X_i\}} \text{score}(X_i, \text{PA}_{X_i}).$$

### 2.2.3 Graph-based search approach

The idea in the following is to formulate the learning task as a graph search problem. The search graph naturally decomposes into layers, which can be searched one at a time. This enables the algorithm to use memory and disk in an orchestrated manner without keeping all the nodes in memory at any point in time. Theoretically, this also means that the algorithm can scale up to any number of nodes if there is enough disk available. In the following, a definition of the order graph is given. Effectively, an order graph is the search graph. After that, a short review of the sparse parent graph is given as it is used to obtain optimal scores for different parent sets during the search in the order graph. As noted by Malone et al. [MYHB11], the formulation of the problem of finding an optimal Bayesian network as a graph search problem, where the search graph is the order graph, opens up the possibility to use any graph search algorithm to find the best path from the start node to the goal node. This thesis is centered around breadth-first search, but, for example, Yuan and Malone [YM13] use an A* algorithm to search for an optimal Bayesian network.

### 2.2.4 Order graph

As defined by Malone et al. [MYHB11], an order graph is a Hasse diagram. Each of its nodes $N_k$, with $k = 1, \ldots, m$ and $m \in \mathbb{N}$, contains the score for the associated optimal subnetwork for one subset of variables. Effectively, the order graph is a powerset for the set of variables $V$ as it contains all the subsets of the set of variables $V$. This also straight away yields that the number of the nodes $m$ in the order graph equals $2^n$. One can derive it from the fact that any subset of $V$ can be represented
as \{\omega_1, \omega_2, ..., \omega_n\} where each item \omega_i in the subset, \(1 \leq i \leq n\), can have either value one or zero based on whether a variable \(X_i\) belongs to the subset. As there are two choices for each \(n\) variables in the variable set \(V\), there are together \(2^n\) unique subsets. Figure 2 shows an example of an order graph for four variables.

The binomial coefficient \(C(n, k)\), where \(n, k \in \mathbb{N}\), gives the number of distinct \(k\)-element subsets for any set containing \(n\) elements. For an order graph, binomial coefficient can be used to calculate the number of nodes on a certain layer. For an order graph with \(n\) variables in the set of variables \(V\), the number of nodes on layer \(l\) is given by \(C(n, l)\).

A directed path in the order graph starts from the top of the graph from the start node and ends at the bottom of the graph at the goal node. A path effectively induces an ordering for the variables on the path. For example, a path \(\emptyset, \{X_1\}, \{X_1, X_2\}, \{X_1, X_2, X_3\}, \{X_1, X_2, X_3, X_4\}\) produces an ordering of the variables where variable \(X_1\) is first, variable \(X_2\) is second, variable \(X_3\) third and variable \(X_4\) is last. All variables preceding a variable in the ordering are candidate parents for the variable. Each edge on the path has a cost equal to \(\text{bestscore}(\cdot)\), where the new variable is the child node and its parents are a subset of the preceding variables. For example, the cost of the edge edge between node \(\{X_1, X_2\}\) and node \(\{X_1, X_2, X_3\}\) is given by \(\text{bestscore}(X_3, \{X_1, X_2\})\).

![Figure 2: An example of an order graph for a Bayesian network of four variables \(V = \{X_1, X_2, X_3, X_4\}\).](image)

### 2.2.5 Sparse parent graph

A parent graph is a data structure used to compute the costs for the arcs in the order graph [YM13]. In other words, for variable \(X_i\) and candidate parents \(U \subset V \setminus \{X_i\}\)
it will be used to find bestscore($X_i, U$) for the edge from $U \rightarrow U \cup \{X_i\}$. Each variable in the dataset has its own parent graph. A full parent graph is a Hasse diagrams containing all subsets of the variables in set of variables $V$ except the variable itself in question, $X_i$. Figure 3a gives an example of the parent graph for variable $X_1$ with a set of variables $V = \{X_1, X_2, X_3, X_4\}$.

Figure 3: A sample parent graph for variable $X_1$ for a Bayesian network of four variables $V = \{X_1, X_2, X_3, X_4\}$. Figure (a) shows the raw scores for all parent sets of $X_1$. The first line specifies the parent set and the second line gives the score of using that set as the parent set for $X_1$. Figure (b) gives the optimal parent set scores, i.e., bestscore($X_1, ...$) for each candidate parent set. Note that the second line here gives the optimal score of using this parent set as the parent set for $X_1$ and it is propagated from a predecessor parent set if the predecessor has better score than the parent set itself. Last Figure (c) shows the effect of pruning the parent sets by leaving away pruned candidate parent sets in white. A parent set can be pruned if any of parent set’s predecessors have better score than the parent set itself.

Each node in the parent graph stores the optimal parent set $PA_{X_i}$ formed using the predecessors $U$ to minimize score($X_i, PA_{X_i}$). Figure 3a shows the scores for different parents of node $X_1$. In practice, Yuan et al. collect the counts from a dataset using sparse AD-trees and compute scores based on the counts [YM13, ML98]. Score calculation would easily become a problem with exponential amount of scores to calculate unless some of the parent sets could be pruned. Actually, some parent sets can be discarded without even calculating their scores at all due to the following theorems by Tian [Tia00]. Although the following theorems concern only MDL scores, other scoring functions have similar pruning rules which could be used
We begin with Theorem 1 from Tian, which gives an upper bound to the number of parents in an optimal Bayesian network measured by the MDL scoring function.

**Theorem 2.1.** In an optimal Bayesian network based on the MDL scoring function, each variable has at most \( \lfloor \log_2(\frac{2N}{\log N}) \rfloor \) parents, where \( N \) is the number of data points.

Another theorem used in pruning candidate parent sets is the equation 13 from Tian [Tia00] and it will be presented next.

**Theorem 2.2.** Let \( U \) and \( S \) be two candidate parent sets for \( X_i \), \( U \subseteq S \), and \( K(X_i|S) - \text{MDL}(X_i|U) > 0 \). Then \( S \) and all supersets of \( S \) cannot possibly be optimal parent sets for \( X_i \).

One more pruning theorem for the parent graph has appeared in the literature [Tey05] and is presented next. It effectively means that a parent set is not optimal when a subset has better score.

**Theorem 2.3.** Let \( U \) and \( S \) be two candidate parent sets for \( X_i \) such that \( U \subseteq S \), and \( \text{score}(X_i, U) \) is better than \( \text{score}(X_i, S) \). Then \( S \) is not the optimal parent set of \( X_i \) for any candidate set.

This theorem can be seen in action in Figure 3b, where the optimal parent set propagates to successor nodes having worse scores than their predecessors. Due to this propagation, the bestscore-function must have the following property [Tey05].

**Theorem 2.4.** Let \( U \) and \( S \) be two candidate parent sets for \( X_i \) such that \( U \subseteq S \). Then we have \( \text{bestscore}(X_i, S) \leq \text{bestscore}(X_i, U) \).

The full parent graph for each variable \( X_i \) would enumerate all the powerset of \( V \setminus \{ X_i \} \) and store \( \text{bestscore}(X_i, U) \) values for all the powersets’ constituents. However, Theorem 2.3 shows that the number of optimal parent sets is often less than the whole powerset and that an optimal parent set might be shared by several candidate parent sets. As a response to this, a sparse representation of the parent graph can be used instead [YM13].

First of all, Theorems 2.1 and 2.2 allow the algorithm to prune some of the parent sets without evaluating their quality. This effectively means that whole powerset will never be created. Second, instead of creating Hasse diagrams to represent parent
sets, the optimal parent scores are sorted for each variable $X_i$ in a list along with another list containing the corresponding optimal parent sets. By using these lists effectively, Yuan and Malone [YM13] are able to largely alleviate the problems with full parent graphs’ size increasing exponentially with input variable count.

2.2.6 Heuristic function

In order to prune candidate solutions during the search in the order graph, a heuristic function is needed to assess the quality of subnetworks [YM13]. The idea is based on the fact that an admissable heuristic function can be used to guide the search. The optimal solution to a relaxed problem can be used as an admissable bound for the original problem. However, as this thesis is centered around the performance of delayed duplicate detection explained in further detail later, the use of heuristic function to guide the search in the order graph is left aside. All the results presented in this thesis are done without any pruning using a heuristic function during the order graph search.
3 Delayed duplicate detection

Many search algorithms are often limited by the amount of semiconductor memory available. At the same time disk storage is a few orders of magnitudes cheaper than semiconductor memory. One of the main challenges with using disk instead of memory in search algorithms is the detection of duplicates during the generation of the candidate nodes in the search graph, in other words, duplicate detection. When working with semiconductor memory, duplicate detection can be done with an in-memory hash table. However, due to long disk latencies, randomly accessed hash-tables have been infeasible using disk. According to Korf [Kor08], the main reason why in-memory hash tables using virtual memory on disk are infeasible is that the access to memory would in many cases result in a page fault.

Search algorithms can address the problem of duplicate detection when the search graph does not fit into semiconductor memory by using a mechanism called delayed duplicate detection (DDD) [Kor08, Kor04, Ros94]. It can be generically defined as a process in which the duplicates are detected after they have been written to disk. According to Korf [Kor08], Jerry Brian was among the first to propose the use of delayed duplicate detection to perform brute-force breadth first searches of Rubik’s Cube using external memory. However, the work was never published and only briefly described in a series of posts to the Cube-Lovers mailing list starting in December 1993. It is also worth noting that although the price of semiconductor memory has gone down, there has still been interest in doing research in delayed duplicate detection by making the delayed duplicate detection faster, for example, by moving the calculations to GPU [ES11].

3.1 Sorting-based delayed duplicate detection

There have not been too many deviations from Jerry Brian’s idea until quite recently. The general idea of delayed duplicate detection until Korf’s paper in 2008 [Kor08] was to push the candidate solutions from memory to disk as soon as they are generated [Kor04]. Then at the end of the search the candidate solutions are sorted on the disk and duplicates detected. The sorting part of this runs in principle in $O(n \log n)$ time, which is the bottleneck of the method.

A variant of this approach was used by Malone et al. [MYHB11] in the MYHB algorithm. This approach works in a way that the candidate solutions are sorted first in memory before writing them to disk to separate files. Once the candidate
solutions for a one layer in the search graph have been generated, they are in files on the disk. To create the global order between all of the candidate solutions in the files, the algorithm then reads the topmost candidate solution from each file in memory, and so creates an ordered set of the topmost candidate solutions in memory. By ordering this set and then selecting the first candidate solution according to the ordering, the procedure finds the first candidate solution in the global ordering. After that the set of candidate solutions needs to be updated to again contain the topmost candidate solutions left in the files on the disk, and then the next candidate solution in the global ordering can be selected. This process finds a global ordering for the nodes on the disk, and the procedure allows the detection of duplicates as the ordering will present duplicates contiguous to each other. The global ordering ensures that after a new candidate solution with certain score has been read from the disk, all of the duplicates of the candidate solution will be read from the disk before reading a new unseen candidate solution.

After Jerry Brian’s initial work, there has been two major changes to delayed duplicate detection. One of them is the structured duplicate detection [ZH04], but it will not be considered more extensively here. The second one is the hashing-based delayed duplicate detection introduced by Korf [Kor08] and it will be next discussed in more detail.

3.2 Hashing-based delayed duplicate detection

Hashing-based delayed duplicate detection runs in linear time in the number of candidate solutions being processed. This makes it faster than the sorting-based method, which runs in $O(n \log n)$ time in the number number candidate solutions processed.

The main idea in the hashing-based delayed duplicate detection is to partition the search space into non-overlapping subsets of unique candidate solutions. These sets define files on the disk so that a single file only contains candidate solutions belonging to one of the sets. Next, the candidate solutions are written to the files as they are generated. The delayed duplicate detection is done by employing an in-memory hash table: the candidate solutions from a single file are read into the hash table and duplicates are detected one at a time. The definition of the subsets partitioning the search space must guarantee that the hash table will not contain more than a given amount of unique candidate solutions. This ensures that memory requirements of any size can be met.
Korf [Kor08] presents as an example the 30-disc four-peg Towers of Hanoi Problem. He divides the discs into the 16 largest and the 14 smallest, and uses the 16 largest discs to decide the file into which a candidate solution is written in. The division of candidate solutions based on the 16 largest discs also reduces the memory requirement, because in a single file the 16 largest discs are the same for each candidate solutions. This means that it is actually enough to write only the 14 smallest discs per candidate solution into each file.

The method can also be parallelized without problems [Kor08]. Both reading and writing to separate files are not subject to concurrent accesses that would need to be synchronized on the program level. The candidate solutions can be generated in separate threads without the two threads accessing the same resources because a single thread can generate candidate solutions without affecting others. In a layered search, the files containing the candidate solutions from the previous layer can be distributed between the worker threads so that each thread only needs to access one file at a time. Similarly, writes to files on the disk need not to be synchronized, because the writes to the same file by separate threads are temporarily stored to separate output buffers [Kor08]. In other words, it is enough to allow the operating system to serialize the writes to the files on the disk.

### 3.3 Hybrid duplicate detection

The MYHB algorithm by Malone et al. [MYHB11] utilizes in-memory duplicate detection as long as possible and resorts to delayed duplicate detection only when necessary. This behavior can be named as hybrid duplicate detection, and it is done to reduce the number of writes done to the disk. The barrier where duplicate detection changes to delayed duplicate detection is binary, which means that the in-memory hash table is flushed to disk if it contains more nodes than a given predefined limit.
4 Using delayed duplicate detection with hashing in Bayesian network structure learning

Since the size of the largest layer of the order graph with $n$ variables is order of $C(n, \frac{n}{2})$, even for modest $n$ the number of unique order graph nodes generated for that layer is too large to fit into semiconductor memory available for the algorithm. The generation of the nodes during the search through the order graph also produces duplicates, and only the duplicate with the best score should be used in node generation for the next order graph layer. Overall, this means that the algorithm needs to utilize disk to manage all generated order graph nodes.

The MYHB algorithm by Malone et al. [MYHB11] performs delayed duplicate detection using the sorting-based hybrid method. The idea is to perform the delayed duplicate detection using multiple sorted files and reading nodes from them one at a time to detect possible duplicates as described in Section 3.1. However, in the work by Korf [Kor08], disk-based hashing was superior to the sorting-based approach.

The main idea in this thesis is to study whether this is also the case for Bayesian network structure learning. In principle, the hashing-based DDD algorithm runs in linear time in the number of candidate solutions generated while the sorting based DDD algorithm has a theoretical speed limit of $O(n \log n)$.

The main concern when using hashing to deal with external memory is dividing the generated nodes to different files when writing them to the disk. In the following a concept of file-set is used to refer to a set of candidate order graph nodes which are laid into the same file on the disk. The delayed duplicate detection is performed on a file basis, so a practical requirement is that each file-set should not contain more unique nodes than fits into memory. Otherwise, all the nodes will not fit into the memory, and the duplicate detection would fail. For the implementation, a natural starting point is to first satisfy the constraint with memory, and that will be further discussed next.

To hash a single candidate order graph node to certain file on the disk a file hash-function needs to be defined. It divides a single layer of the order graph into subsets at most of a given size. One approach to do the division will be presented next. The idea used in dividing the nodes to different files is very similar to the idea Korf [Kor08] uses when dividing the candidate solutions to different files based on the certain number of bits in the bitstring representing a single candidate solution. Such division allows the algorithm to control the number of files and the number
of unique candidate solutions per file, which is crucial to ensure that the delayed
duplicate detection respects given memory requirements.

4.1 First-\(k\) subcombinations

The basic idea in dividing order graph nodes into different files is based on the use
of combinations and subcombinations similar to Tamada et al. [TIM11].

**Definition 4.1** (Combination). An ordered set of variables \(U \subset V\) is referred
to as a *combination*, and it uniquely identifies a node in the order graph. With
the number of variables in the set of variables \(V\) denoted by \(n\), a combination is
represented using an indicator set \(I = \{\omega_1, \omega_2, ..., \omega_n\}\) where each item \(\omega_i, 1 \leq i \leq n\)
in the indicator set \(I\) can have either value one or zero indicating whether a variable
\(X_i \in V\) belongs to subset \(U\).

It should be noted that the index \(i\) is used in indexing the set of variables \(V\), but also
the indicator set \(I\) for the combination \(U\). This way the index \(i\) creates a *precedence
order* for the variables belonging to the combination and is important to keep in
mind in the following definitions. It is also worth noting that such precedence order
is arbitrary in the sense that variable \(X_1\) has highest precedence solely because it
happens to be the first variable in the set of variables \(V\).

**Definition 4.2** (Subcombination). An ordered subset \(S \subset U \subset V\) of a combination
\(U\) is called a *subcombination*.

**Example 4.1.** To highlight the importance of the indicator set, we showcase it
using set notation. Define an element of \(V\) as \(X_i\). Now a subcombination \(S\) of
combination \(U\) can be represented using an indicator set

\[
I = \{ \omega_i \mid \text{if } X_i \in S, \text{ then } \omega_i = 1, \text{ otherwise } \omega_i = 0 \}.
\]

Based on theses two definitions we can define the first-\(k\) subcombination as fol-
lows:

**Definition 4.3** (First-\(k\) subcombination). A *first-\(k\) subcombination* of combination
\(U\) is a subcombination having the first \(k\) variables in common with combination \(U\).

First-\(k\) subcombinations are used in the following to define file-set. This means that
we need to define when two combinations have a common first-\(k\) subcombination.
**Definition 4.4** (Indicator set for a common first-\(k\) subcombination). Define the index of the \(k^{th}\) variable appearing in the first-\(k\) subcombination \(S\) in the variable set \(V\) as \(l\). Given two combinations \(U_1\) and \(U_2\) from variable set \(V\), the indicator set \(I_k\) for the common first-\(k\) subcombination is defined as

\[
I_k = \{ \omega_i \mid \omega_i \in I_{U_1}, \lambda_i \in I_{U_2}, \omega_i = \lambda_i \text{ when } i \leq l \leq n, \text{ otherwise } \omega_i = 0 \}.
\]

The following example illustrates definition of a common first-\(k\) subcombination.

**Example 4.2.** If we have combination \(U_1 = \{X_1, X_4, X_6, \ldots\}\) and another combination \(U_2 = \{X_1, X_4, X_7, \ldots\}\), we can define knowing only the first four variables of the both combinations their common first-2 subcombination \(S = \{X_1, X_4\}\).

It is important to note that from the definition of the first-\(k\) subcombination it follows that each possible combination from the set of variables \(V\) belongs to exactly one first-\(k\) subcombination for each \(k\). This is built in to the definition of the first-\(k\) subcombination. We can see this by defining \(l\) as the index of the \(k^{th}\) variable. Then the first \(l\) indicators in the indicator set \(I = \{\omega_1, \omega_2, \ldots, \omega_n\}\) of a combination \(U\) identify uniquely the first-\(k\) subcombination in question. The following example will explain this in further detail.

**Example 4.3.** In the following, it is assumed that there are ten variables in the set of variables \(V = \{X_1, X_2, \ldots, X_{10}\}\), the combinations formed from the set of variables have five variables in them, and we are considering a first-4 subcombination. We take as an example first-4 subcombination \(S = \{X_1, X_2, X_3, X_7\}\). From the definition of first-4 subcombination \(S\) we know that the index, \(l\), of the fourth variable, \(X_7\), is seven. This means that any combination belonging to this first-4 subcombination will have the first seven indicators in its indicator set having values \(\{1, 1, 1, 0, 0, 0, 1\}\). If we assumed that there were two first-\(k\) subcombinations with the same indicator sets, then the two would be by definition the same first-\(k\) subcombination. In other words, any combination having these seven indicators set as defined above, would have the same first-4 subcombination. Two examples are combinations \(C_1 = \{X_1, X_2, X_3, X_7, X_9\}\) and \(C_2 = \{X_1, X_2, X_3, X_7, X_8\}\) which both have same first seven values in their indicator sets \(I_{C_1} \{1, 1, 1, 0, 0, 0, 1, 0, 1, 0\}\) and \(I_{C_2} \{1, 1, 1, 0, 0, 0, 1, 1, 0, 0\}\).

Based the definition of a first-\(k\) subcombination, we define a set of combinations having a common first-\(k\) subcombination. A set of combinations is a set of sets: it will be called a family of combinations and referred with italic capital font like \(\mathcal{F}\).
**Definition 4.5** (Family of combinations). A set of combinations is a family of sets containing combinations having a certain common subcombination. In other words, given a first-\(k\) subcombination \(S\) with \(k \leq l\), for each combination \(U_j, j \in \mathbb{N}\) in the family of combinations \(\mathcal{F}\), the indicator set of the \(j\)th combination must satisfy

\[
I_{U_j} = \{ \omega_i \mid \omega_i \in I_S \text{ when } i \leq l \leq n \}.
\]

Not all of the possible first-\(k\) subcombinations correspond to combinations. For example, if we have ten variables in the set of variables \(V = \{X_1, X_2, ..., X_{10}\}\) and a combination has five variables, then we can see that a first-2 subcombination having variables \(X_9\) and \(X_{10}\) will correspond to zero combinations. This is because a node having variables \(X_9\) and \(X_{10}\) must have three other variables in it as well, and all of these three variables must have indices smaller than 9 and 10, because 9 and 10 are the largest indices.

### 4.2 Selecting the number of variables used in a first-\(k\) subcombination

Binomial coefficients can be used to calculate how many distinct variable combinations can be generated into the family of combinations \(\mathcal{F}\) for a first-\(k\) combination. With the number of variables in the set of variables \(V\) denoted by \(n\), the number of distinct order graph nodes having \(m\) variables are given by binomial coefficient \(C(n, m)\).

Similarly, the same method can be used to solve for the number of variables used in the first-\(k\) subcombination defining the family of combinations for each file, i.e. a file-set. The number \(k\) of first-\(k\) variables used to create the family of combinations \(\mathcal{F}\) can be calculated by enforcing that the number of distinct variable combinations per file is at most the number of nodes that fit into memory. In practice, the solving is implemented by guessing that the number of variables used in the first-\(k\) subcombination is not larger than \(m\). Then, \(k\) is decremented one-by-one until the number of order graph nodes in the family of combinations per file is as large as possible still fitting into memory. The implementation selects \(k\) to be as small as possible to avoid creating more files than needed because more files increase overhead.

We will now present an example showing how to calculate the number of unique combinations for a single file-set based on first-\(k\) subcombination. This example mimics...
the process used in the implemented algorithm to find the number of variables used in first-$k$ subcombination, which defines the file-sets on the disk.

**Example 4.4.** In the following, it is assumed that there are ten variables in the set of variables $V = \{X_1, X_2, \ldots, X_{10}\}$ and we are on level 5 of the order graph, i.e., we have five variables in a candidate order graph node. This means that there are $C(10, 5) = 252$ unique order graph nodes. Assume now that we can fit into memory at most 50 unique order graph nodes at a time. This means that a single file-set can at most contain unique 50 nodes. We can now start the iteration, where we decrement the number of variables used in the first-$k$ subcombination one-by-one until a first-$k$ subcombination will at most match to 50 unique order graph nodes. If we use five variables for the first-$k$ subcombination, then there will be exactly one unique order graph node in each of the files, and 252 different files. If we use four variables for the first-$k$ subcombination, then there will be $C(10 - 4, 5 - 4) = C(6, 1) = 6$ unique combinations in each file-set. We can decrease the number of variables used in the first-$k$ subcombination to three, which means that there will be $C(10 - 3, 5 - 3) = C(7, 2) = 21$ unique combinations in each file-set. If we would decrease the number of variables used in the first-$k$ subcombination to two, then there would be $C(10 - 2, 5 - 2) = C(8, 3) = 56$ unique combinations in each file-set, and that would violate the limit for memory set earlier. When we use three as the number of variables in the first-$k$ subcombination, the possible number of files used on the disk can be calculated by noticing that there are three variables that we select from a set of ten variables. Thus, we can have $C(10, 3) = 120$ different file-sets based on this first-$k$ subcombination.

It is worth noting here that the number of files used on the disk is far from perfect. When we have 21 unique order graph nodes in a single file, and altogether 252 unique order graph nodes, it would make sense to have only $\frac{252}{21} = 12$ files on the disk. However, the definition of the first-$k$ subcombinations means that some of the files will contain only a few unique order graph nodes. This property of the first-$k$ subcombinations will be later assessed in more detail.

### 4.3 Dividing candidate order graph nodes to files on the disk

After selecting the number of variables used in the first-$k$ subcombination, the next step is to divide the candidate order graph nodes to the files on the disk based on the file-set to which they belong. Each first-$k$ subcombination will have its own file,
and the definition of first-\(k\) subcombination guarantees that each one of these files will not contain more unique nodes than what can fit into memory.

We will next present an example showing how the candidate order graph nodes are divided to files on the disk based on the first-\(k\) subcombination defining the different file-sets. This is the process used in the algorithm when writing the candidate order graph nodes to the disk.

**Example 4.5.** In the following, it is assumed that there are ten variables in the set of variables \(V = \{X_1, X_2, ..., X_{10}\}\) and we are on the level 5 of the order graph, i.e., we have five variables in a candidate order graph node. If we now use two variables as the \(k\) for the first-\(k\) subcombinations, we can see that we have, for example, the following first-\(k\) subcombinations: \(\{X_1, X_2\}, \{X_1, X_3\}\). Similarly, we have, for example, the following candidate order graph nodes which need to be written to the files on the disk: \(\{X_1, X_2, X_3, X_5, X_6\}, \{X_1, X_2, X_5, X_6, X_9\}\). Both of these nodes would be written to the file corresponding to the first-\(k\) subcombination \(\{X_1, X_2\}\) because the first two variables in for the two combinations are \(X_1\) and \(X_2\).

### 4.4 Distribution of candidate order graph nodes to different files on the disk

The definition of first-\(k\) subcombination means that the distribution of unique order graph nodes to files is uneven. The maximum number of nodes per file will also be reached at most in one file. In other words, there is still room for improvement in the way this hashing function divides order graph nodes into different files. To quantify the distribution of combinations to file-sets we next define a union of combinations.

**Definition 4.6** (Union of combinations). A union of combinations is a union of the variables belonging to the combinations that are part of the union. For set combinations \(U_j, j \in \mathbb{N}\) the union is defined as \(\bigcup_{j \in \mathbb{N}} U_j\). In other words, for combinations with indicator sets \(I_j\) and components \(\omega_{i,j}\), the indicator set for the union, \(I_U\), is defined as

\[
I_U = \{ \omega_i \mid \text{If } \omega_{i,j} = 1 \text{ for at least one } j, \text{ then } \omega_i = 1, \text{ otherwise } \omega_i = 0 \}\.
\]

We now consider an example showing how first-\(k\) subcombinations lead to uneven distribution of candidate order graph nodes to files on the disk.

**Example 4.6.** In the following, it is assumed that there are ten variables in the set of variables \(V = \{X_1, X_2, ..., X_{10}\}\) and we are on the level 3 of the order graph. This
means that there are $C(10, 3) = 120$ unique order graph nodes. In addition, there are $C(9, 2) = 36$ nodes containing for example $X_1$, because if we take one of the variables as given, i.e. $X_1$, then we have nine variables left and we need to choose two of them. Similarly, we have $C(8, 1) = 8$ nodes containing sub-combination \(\{X_1, X_2\}\) or sub-combination \(\{X_1, X_3\}\). Now if we look at the union of the two earlier combinations \(\{X_1, X_2, X_3\}\), we can see that there are $C(7, 0) = 1$ nodes containing this combination. This means effectively that this one node will be laid to the file corresponding to sub-combination \(\{X_1, X_2\}\), because variables $X_1$ and $X_2$ precede variable $X_3$ in the ordering of the variables in the set of variables $V$. This means that instead of having eight nodes, the file for sub-combination \(\{X_1, X_3\}\) will have one node less. The same will also happen for sub-combinations \(\{X_1, X_3\}\) and \(\{X_1, X_4\}\), whose union is \(\{X_1, X_3, X_4\}\). The file for subcombination \(\{X_1, X_3\}\) will contain only six nodes, because both subcombinations \(\{X_1, X_2\}\) and \(\{X_1, X_3\}\) will both steal one node from it. The analysis can be continued in this case up to the super-combination \(\{X_1, X_9, X_{10}\}\) and it would show that file for sub-combination \(\{X_1, X_9\}\) will have exactly one node and file for sub-combination \(\{X_1, X_{10}\}\) will not contain any nodes. This means that combinations will be divided unevenly between the file-sets and only one of the file-sets, in this case the file-set for first-sub-combination \(\{X_1, X_2\}\), will contain the maximum number of nodes.

Table 1 shows the counts and percentages for five different subcombinations for three different variable sets on different levels of the order graph. It also shows what happens when the number of nodes on the layer is increased or the number of variables in the set of variables is increased. The main take from the table is that as the number of variables is increased also the number of variables in the subcombinations used needs to increase as well to allow hashing to properly divide the order graph nodes to files on the disk. It also highlights the fact that the hardest part for the hashing is always the layer in the order graph having half of the variables in the set of variables. For example, if we have twenty variables in the variable set and the sub-combination used uses four variables, then the most combinations are created on layer twelve and thirteen of the order graph. This can be explained by the fact that the the number of unique combinations from a set of variables $V$ with $n$ variables is largest when we select a subset with $\frac{n}{2}$ variables. In other words, when we fix four of the 20 variables, then we have $20 - 4 = 16$ variables remaining. This means that the most combinations are created when we are on level twelve of the order graph, because there we are fixing four variables from 12. This means that we have left $12 - 4 = 8$ variables, which is half of sixteen.
Table 1 also shows that as the number of variables in the order graph node is increased, the effect of stealing from other nodes increases. This can be seen from the fact that, for example, the number of nodes belonging to family of first-2 subcombination \{1, 2\} increase until about half of the nodes belong to its family. The problem with stealing is not very apparent with ten nodes in the set of variables. However, there are 24310 nodes in the family of first-3 subcombination \{1, 2, 3\} on layer eleven of the order graph for twenty variables, so the problem comes into focus. However, by increasing the size of the first-\(k\) sub-combination, the problem can be alleviated: the sub-combination \{1, 2, 3, 4, 5\} has only 5005 nodes on layer eleven and at most 6435 nodes on layer 12 of the order graph for twenty variables.
Table 1: Counts and percentages in parentheses of the order graph nodes per layer $l$ for five sub-combinations \{1\}, \{1, 2\}, \{1, 2, 3\}, \{1, 2, 3, 4\} and \{1, 2, 3, 4, 5\} with variable set $V = \{1, 2, 3, \ldots\}$ having 10, 15 and 20 number of variables ($n$) on all layers of the order graph.
5 Experimental results

5.1 Implementation details

The hashing-based DDD was implemented on top of the codebase for structure learning created by Malone for a series of published papers [MYHB11, YM13]. The codebase\(^1\) is written in C++ and can be obtained from the original author. Thus, a natural experimental setup is to compare the effect of the changes against the original codebase. The changes are available as a fork\(^2\) of the original codebase.

The implementation allows the program to have at most 500 files open at the same time while dividing the order graph nodes. This is due to the fact that the operating system can only have a limited number of files open at any time. The operating system used in the tests had a hard limit of 1024 open files per process, so 500 files was selected to ensure no operating system related problems affect the tests.

Implementation uses hybrid duplicate detection in the same manner as the MYHB algorithm. The algorithm takes as a parameter the maximum number of nodes kept in memory, and once this limit is reached all of the nodes in memory at that point in time are written to disk.

5.2 Environment

Experiments were performed on a cluster consisting of 240 Dell PowerEdge M610 blade servers. The operating system on the nodes was Ubuntu 12.04.5 LTS (GNU/Linux 3.2.0-60-generic x86i_64). Each node consists of 32 gigabytes of RAM and two four-core Intel Xeon E5540 processors supporting hyper threading. Each node had 16 gigabytes of local disk for all users. This meant that the running program could write at most 16 gigabytes on the disk. In a more realistic situation the amount of disk is multiple times the amount of RAM.

The code is single-threaded, but the experiments were run parallel to each other on separate nodes with each node running only a single instance at a time. None of the experiments were run on a dedicated node, so other users’ access to a node’s resources might also reflect in the results in nondeterministic ways. Due to the fact that other users could affect the results, all of the timing tests were run ten times.

\(^1\)https://bitbucket.org/bmmalone/urlearning-cpp
\(^2\)https://bitbucket.org/nikkekin/urlearning-cpp
to minimize their impact.

Measurements for the runtime were taken by adding logging statements into the code. The Boost Timer library\(^3\) was used to log time, and the results reported in the following utilize the concepts introduced in the Timer library. Wall-time measures time as if it was measured by a clock on the wall. User time measures the time charged by the operating system as user cpu time and system time measures the time charged by the operating system as system cpu time. Cpu time should be a sum of the two latter, and any deviations between the wall and cpu times suggest that either the code runs parallel in multiple cores or that there has been waiting time, for example, on input and output. The code used is single-threaded, so the deviations seen in our case result from congestion on input and/or output.

### 5.3 Data used

The experiments consist of two different scenarios. The first scenario is the same used by Malone and Yuan [MY13] to allow testing of sensitivity of the algorithm used to the network and dataset generation. The scenario consists of networks of 29 variables with 2, 4, 6 or 8 maximum parents per variable. In addition, the generated dataset sizes are varied with 1k, 5k, 10k and 20k data points. The different dataset names are given as follows: <number of variables>.<maximum number of parents>.<number of data points>.

The second scenario was generated using the same ideas to test the sensitivity of the algorithm to the network and dataset generation, but this time with a dataset with fewer variables to allow for a more fine-grained understanding of the dataset. Only 20 variables were used in the network to make sure that the runtimes of the algorithms used would stay manageable. The maximum parents were again 2, 4, 6 or 8. The dataset sizes were again selected to be 1k, 5k, 10k and 20k data points. The naming of the different datasets follows the same convention as already explained above. The results will be first presented for the dataset with 20 variables, as it introduces the concepts used in the analysis.

For the scenario with 20 variables, the datasets were generated. The generation mechanism used was the same as used by Malone and Yuan [MY13] and utilizes C++ library already referenced above and a Java-library provided by Malone and

\(^3\)http://www.boost.org
Yuan\(^4\). First a random Bayesian network was generated using Ide and Cozman Bayesian network structure generation algorithm [IC02] given the structural constraint with maximum number of parents. Then logic sampling [Hen88] was used to sample data points from the generated network. Logic sampling is a process where the network is first topologically sorted and then sampled in the order variable by variable. The sampling works correctly because the topological sorting guarantees that once a certain node is reached its parents have already been sampled. This means that it is enough to sample appropriately from the conditional distribution of the variable given its parents. After the dataset for a certain network was generated, the parent scores were calculated and stored to a file which was used as an input for the algorithm.

5.4 Overall performance

Overall, user time is the bulk of the time spent running the learning algorithm, and the difference between the wall time and whole used cpu time is not large. This suggests that there is not too much waiting for disk. This is true for both datasets and suggests that writing to disk is not a bottleneck. It is also clear that as the maximum number of nodes kept in memory is increased, less delayed duplicate detection is needed. This is natural as the more duplicate detection takes place in memory, the less delayed duplicate detection is needed.

5.5 Scenario with 20 variables

Overall, the results reported in Table 2 and Tables 12-26 in online Appendix A\(^5\) show that the runtime results for the two algorithms with 20 variables are not sensitive to either the amount of parents in the generating network or the amount of data. For brevity, only Table 2 is with the main-part of the thesis. The rest of the tables are in online Appendix A.

The results in, for example, Table 2, show that the results for the sorting codebase with 1000 maxnodes are empty. This is related to the fact that the algorithm died in the middle of the run. The reason for the original program dying is that the amount of file handles open exceeds the limit on the operating system of 1024 file handles.

\(^5\)https://bitbucket.org/nikkekin/urlearning-cpp/src/original_with_timer/tables.pdf
handles.

Another point to make is that when reducing the maximum number of nodes kept in memory from 5000 to 1000, the performance of the codebase using hashing deteriorates significantly. The mean of the time used for hashing-based delayed duplicate detection jumps from 6.48 seconds to 21.15 seconds. This deterioration is probably related to the fact that number of files used in the hashing-based codebase increase dramatically. The important point to make here is that the maximum number of nodes to keep in the semiconductor memory governs two things in the hashing codebase: the number of files on the disk and the number of times the nodes in the memory are divided to those files. In the MYHB algorithm the idea is similar, but the difference is that the number of files used on the disk does not depend on the way the nodes can be divided to them: each time the nodes are written to disk a new file is created.

With a small number for the maximum number of nodes kept in memory it does seem overall that the sorting-based codebase is able to slightly outperform the hashing-based codebase. However, as the number for maximum number of nodes kept in memory is increased, the hashing-based codebase seems overtakes the sorting-based codebase. However, in both cases the differences are not significant and might be related to, for example, the different data structures used by the codebases or the different ways the two write the nodes on the disk.

It is also worth noting that when using the 1000 as the number of maximum nodes kept in memory, the results do show 0.75 seconds of delay from the disk when calculated as a difference between the mean times for wall and CPU for the hashing-based codebase. This is consistent over all of the results reported for networks with 20 variables. However, it is still worth noting that even in this case the disk takes only around $0.75/21.15 = 3.5\%$ of the runtime, which is not a large percentage. Overall, the results show that both sorting and hashing codebases seem to be able to utilize the disk quite effectively without suffering from long waiting times for input and output.
Table 2: Timing results in seconds for the dataset with 20 variables, maximum 2 parents per variable and 1000 observations from the runs with varying number of nodes kept in semiconductor memory (maxnodes). Timing results for delayed duplicate detection report the time used for delayed duplicate detection while the timing results for the whole run report time used running the algorithm altogether.
5.5.1 Locality

Due to hybrid duplicate detection, many duplicates are detected in semiconductor memory. In this context, locality refers to the amount of nodes detected in this way. As the algorithms used in this thesis are deterministic, locality can be measured by running the codebase only once. In addition, as we have restricted the search of the order graph not to use any pruning, it happens in exactly the same way regardless of the dataset generation mechanism, for example, the amount of maximum parents per variable or the number of observations do not impact the results. However, both the maximum number of nodes kept in memory, i.e. maxnodes, and the method used to write and read the candidate order graph nodes from the disk affect the duplicate detection behavior as they change the order in which candidates are generated.

Table 3 shows the expanded order graph nodes and candidate order graph nodes written to disk during three runs of the hashing codebase as maxnodes varies for the codebase using hashing. It effectively shows how much more work the delayed duplicate detection has to do on each layer, and as whole, when the maximum number of nodes is increased from 5,000 to 150,000. In the optimal scenario no duplicate detection is needed and the number of nodes written to disk on layer \( l \) equals the number of nodes expanded on layer \( l + 1 \).

Similarly, Table 3 also shows the same information for the codebase using sorting. The differences in the amount of nodes written to disk between hashing and sorting are explained by the fact that the two methods read the nodes from the previous layers from the disk in a different order. Thus, they also expand nodes in different orders. This means that a different amount of duplicates will be detected in memory. Surprisingly, there is quite a big a difference of almost one million nodes between the two codebases when only 5000 candidate nodes are kept in the memory. Interestingly, the results for twenty variables also show that for these three sizes of maximum number of nodes kept in memory, the sorting codebase always writes less order graph nodes to disk. One plausible explanation for this would be that the sorting codebase implicitly benefits from the order in which order graph nodes are generated and does not distort it in the same way as the hashing codebase does. This is based on the assumption that the duplicates for the nodes expanded are found more likely to be found from the in-memory hash table for the sorting-based codebase than for the hashing-based. This is interesting because the sorting-based codebase writes nodes to the disk in groups that are generated and then written to disk as the number of nodes in the hash table exceeds maxnodes, while the hashing-
Table 3: Order graph nodes expanded and written to disk for dataset with twenty variables when maximum number of nodes kept in memory (maxnodes) varies with values 5000, 50000 and 150000 for the codebases using hashing and sorting.

5.6 Scenario with 29 variables

Due to the exponential size of the order graph with respect to the number of variables, the scenario with 29 variables creates a much larger learning problem than the scenario with 20 variables. A small python script was used to execute each algorithm ten times on a given dataset on a single server node. Some of the server nodes failed during the experiments, so the node running the experiment was not
the same for all of the runs even for the same algorithm with the same parameters. Due to the problems running the experiments, two sets of experiments are presented. They are in every other way the similar to each other, but in the latter a bug in the timing of the delayed duplicate detection was fixed so that the times for user, system and cpu for delayed duplicate detection were also reported. The decision to show both experiments was done because of the variation in the results between the two experiments. Probably, the variation is related to how the load from other users affected the running of the algorithm. One possible source of variation for the load on the servers between the two experiments is the time of the year the experiments were run. The first set of results was run during the two weeks starting from the end of July 2014, which is a holiday season in Finland and so the load on the server nodes would likely be low compared to other times during the year. However, the second set of experiments was run during the middle of September 2014, which is a period when holidays are over.

5.6.1 First set of results under light load

The results reported in Tables 5 and 6 show that the mean run times for the hashing and sorting based algorithms are usually close to each other. There is variation between the two depending on the dataset. Overall, as the number for the maximum amount of order graph nodes kept in memory is increased, the amount of delayed duplicate detection decreases due to the fact that more duplicate detection is done in memory. Interestingly the results also suggest that both the hashing and sorting based seem to be, at least in these tests, sensitive to the maximum amount of parent nodes and to the amount of data available as also noticed earlier by Malone and Yuan [MY13]. This can be seen from the fact that the runtimes for the algorithms are clearly different when these parameters are varied. This can be at least partly explained by the number of scores in a sparse parent lists for the dataset, which are given in Table 4. These lists need to be scanned during the search to find optimal parents, so their size has an effect on the performance of the algorithm.
Figures 4, 5 and 6 show the kernel density estimates\(^6\) for the runtime of the delayed duplicate detection and for the runtime of the algorithm for the first set of results conditioned on the codebase, maximum number of nodes kept in memory, number of data points and maximum number of parents. These figures together show that both of the algorithms behaved in a consistent manner with their mean runtimes close to each other over the tests and the timing results contain only a few outliers which do not affect the interpretation of the results.

\(^6\)The kernel density functions presented in this thesis were estimated using a standard library routine density from the statistical package R. Density functions have an advantage over histograms when presenting data, because they require less parameters. For a histogram, one has to select the number of subintervals dealing the data, the size of the intervals and the locations of the intervals [TK76]. However, with non-parametric kernel density estimates, it is enough to only select the size of the intervals. The kernel function is situated at each observation instead of grouping observations [TK76]. The size of the intervals corresponds to the standard deviation of the kernel density and is called bandwidth. Another parameter needed is the smoothing kernel. R package uses by default normal distribution as the smoothing kernel and selects bandwidth using Silverman’s rule of thumb. These parameters are sufficient for comparisons presented in this thesis, but especially the bandwidth can be chosen using more advanced methods [JMS96].

<table>
<thead>
<tr>
<th>Dataset</th>
<th>number of scores</th>
</tr>
</thead>
<tbody>
<tr>
<td>29.2.1000</td>
<td>1,081</td>
</tr>
<tr>
<td>29.2.5000</td>
<td>5,798</td>
</tr>
<tr>
<td>29.2.10000</td>
<td>11,567</td>
</tr>
<tr>
<td>29.4.10000</td>
<td>27,138</td>
</tr>
<tr>
<td>29.4.5000</td>
<td>2,837</td>
</tr>
<tr>
<td>29.4.10000</td>
<td>28,165</td>
</tr>
<tr>
<td>29.4.20000</td>
<td>81,128</td>
</tr>
<tr>
<td>29.6.10000</td>
<td>258,056</td>
</tr>
<tr>
<td>29.6.5000</td>
<td>2,349</td>
</tr>
<tr>
<td>29.6.10000</td>
<td>29,472</td>
</tr>
<tr>
<td>29.8.10000</td>
<td>104,128</td>
</tr>
<tr>
<td>29.8.1000</td>
<td>787</td>
</tr>
<tr>
<td>29.8.5000</td>
<td>11,002</td>
</tr>
<tr>
<td>29.8.10000</td>
<td>38,611</td>
</tr>
</tbody>
</table>

Table 4: Number of scores in the sparse parent lists for the datasets with 29 variables.
Table 5: Timing results in seconds for the first seven datasets from the first set of runs. Timing results for delayed duplicate detection (ddd) report the time used for delayed duplicate detection while the timing results for the whole run report time used running the algorithm altogether.
Table 6: Timing results in seconds for the last seven datasets from the first set of runs. Timing results for delayed duplicate detection (ddd) report the time used for delayed duplicate detection while the timing results for the whole run report time used running the algorithm altogether.
Figure 4: Kernel density estimates and data points plotted for the first set of results for the runtime of the delayed duplicate detection conditioned on the codebase and the number of data points in the dataset are presented Figure (a). In Figure (b) runtime of the delayed duplicate detection is conditioned on the codebase and the number of data points.
Figure 5: Kernel density estimates and data points plotted for the first set of results for the runtime of the delayed duplicate detection conditioned on the codebase and the maximum number of nodes kept in memory are presented Figure (a). In Figure (b) kernel density estimates and data points are plotted for the first set of results for the runtime of the algorithm conditioned on the codebase and the number of data points in the dataset.
Figure 6: Kernel density estimates and data points plotted for the first set of results for the runtime of the algorithm conditioned on the codebase and the maximum number of parents are presented Figure (a). In Figure (b) runtime of the algorithm is conditioned on the codebase and the maximum number of nodes kept in memory.
To analyze statistically the differences between the hashing-based and sorting-based codebases, Mann-Whitney-Wilcoxon [MW47, Wil45] tests\(^7\) were executed for each input file used for time used for delayed duplicate detection and for the whole run. The results for the first set of results are given in Table 7. It shows that in most of the cases the null hypothesis stating that the hashing-based and sorting-based populations are the same can be rejected at a confidence lower than 1 percent for both timing results. This suggests that the difference in times used for delayed duplicate detection and for the whole run for the two methods are for most of the input files statistically significantly different.

\(^7\)Mann-Whitney-Wilcoxon [MW47, Wil45] tests presented in this thesis were executed using a standard library function wilcox.test from the statistical package R. Mann-Whitney-Wilcoxon test is a non-parametric test, which assesses whether two samples, e.g. A and B, are from the same distribution. If this were the case, then the two samples should have values, which are distributed similarly. In practice the test assigns a rank for each observation in the two samples based on the numerical value of an observation in relation to all the observations in the two samples. Then a rank sum is calculated for one of the samples, in this example for sample A. Under the null hypothesis, each of the observations is equally likely to be part of the rank sum calculated, and based on this assumption a distribution for the rank sums can be calculated. Based on the distribution of the rank sums, the test assesses what is the probability to see a rank sum smaller or equal to the one observed for sample A.
Table 7: Results from Mann-Whitney-Wilcoxon tests for first set of results for time used for delayed duplicate detection (DDD) and for the time used for the whole run. The tests were executed for each individual file used as an input for the two algorithms. P-values significant at 1 percent level have been marked with asterisk (*).

5.6.2 Second set of results under heavy load

Tables 8 and 9 report the results for the second set of runs. They confirm the earlier results that a larger number as the maximum number of nodes kept in memory means less writing to disk and a faster runtime. At the same time, the results from the second runs turn upside down the performance of the hashing based codebase against the sorting based codebase; in these tests, sorting-based codebase was typically faster. Due to the difference in testing conditions, the results between the two sets
of runs are inconsistent. That is, the best choice of algorithm depends upon the environment in which it is executed.

Another interesting observation relates to timing of the delayed duplicate detection and the whole run. The whole runtime includes delayed duplicate detection, but still the delayed duplicate detection shows larger differences between the used CPU time and the wall time than the whole run’s used CPU and wall time. For example, for the dataset 29.2.1000 with sorting and 25 million as the maximum number of nodes kept in memory, the difference between the used cpu time for hashing and wall time is about 25 seconds. However, for the same run the difference between the whole run wall time and the used cpu time is only six seconds. In a normal situation the differences here could be used to judge how much time the algorithm has used doing input and output.

Figures 7, 8 and 9 show kernel density estimates for the runtime of the delayed duplicate detection runtime and for the runtime of the algorithm conditioned on the codebase, maximum number of nodes kept in memory, number of data points and maximum number of parents. Figure 7 shows how the variation in the delayed duplicate detection’s runtime is clearly more variant with a larger maximum value for the hashing-based codebase. Figure 8 shows how delayed duplicate detection for the hashing-based codebase is clearly slower than the sorting-based codebase when maximum number of parents is 25M or 50M.
Table 8: Timing results in seconds for the first seven datasets from the second set of runs. Timing results for delayed duplicate detection (ddc) report the time used for delayed duplicate detection while the timing results for the whole run report time used running the algorithm altogether.
<table>
<thead>
<tr>
<th>System</th>
<th>Task</th>
<th>Time (ms)</th>
<th>Mean (ms)</th>
<th>SD (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>29.4.5000</td>
<td>sorting</td>
<td>29.4.5000</td>
<td>10</td>
<td>47.90</td>
</tr>
<tr>
<td>29.6.1000</td>
<td>sorting</td>
<td>29.6.1000</td>
<td>10</td>
<td>35.90</td>
</tr>
<tr>
<td>29.6.5000</td>
<td>sorting</td>
<td>29.6.5000</td>
<td>10</td>
<td>34.90</td>
</tr>
<tr>
<td>29.8.1000</td>
<td>sorting</td>
<td>29.8.1000</td>
<td>10</td>
<td>28.90</td>
</tr>
<tr>
<td>29.8.5000</td>
<td>sorting</td>
<td>29.8.5000</td>
<td>10</td>
<td>26.90</td>
</tr>
<tr>
<td>30.0.1000</td>
<td>sorting</td>
<td>30.0.1000</td>
<td>10</td>
<td>23.90</td>
</tr>
<tr>
<td>30.0.6000</td>
<td>sorting</td>
<td>30.0.6000</td>
<td>10</td>
<td>20.90</td>
</tr>
<tr>
<td>30.0.9000</td>
<td>sorting</td>
<td>30.0.9000</td>
<td>10</td>
<td>18.90</td>
</tr>
<tr>
<td>30.0.1200</td>
<td>sorting</td>
<td>30.0.1200</td>
<td>10</td>
<td>16.90</td>
</tr>
</tbody>
</table>

Table 9: Timing results in seconds for the last seven datasets, ordered by system and task.
Figure 7: Kernel density estimates and data points plotted for the second set of results for the runtime of the delayed duplicate detection conditioned on the codebase and the number of data points in the dataset are presented Figure (a). In Figure (b) runtime of the delayed duplicate detection is conditioned on the codebase and the maximum number of parents.
Figure 8: Kernel density estimates and data points plotted for the second set of results for the runtime of the delayed duplicate detection conditioned on the codebase and the maximum number of nodes kept in memory are presented Figure (a). In Figure (b) kernel density estimates and data points are plotted for the second set of results for the runtime of the algorithm conditioned on the codebase and the number of data points in the dataset.
Figure 9: Kernel density estimates and data points plotted for the second set of results for the runtime of the algorithm conditioned on the codebase and the maximum number of parents are presented Figure (a). In Figure (b) runtime of the algorithm is conditioned on the codebase and the maximum number of nodes kept in memory.
The results for the second set of results from the Mann-Whitney-Wilcoxon tests for the time used for delayed duplicate detection and for the whole run are given in Table 10. Also this time the results show that in most of the cases the null hypothesis stating that the hashing-based and sorting-based populations are the same can be rejected at a confidence lower than 1 percent for both timing results. This means that the difference in times used for delayed duplicate detection and for the whole run for the two methods are for most of the input files statistically significantly different for both sets of results. This means that statistically the distributions generating the timing results for the two algorithms are not in most of the cases same.
<table>
<thead>
<tr>
<th>maxnodes</th>
<th>maxparents</th>
<th>data points</th>
<th>n</th>
<th>DDD p-value</th>
<th>run p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>25M</td>
<td>2</td>
<td>1000</td>
<td>20</td>
<td>0.000*</td>
<td>0.000*</td>
</tr>
<tr>
<td>50M</td>
<td>2</td>
<td>1000</td>
<td>20</td>
<td>0.005*</td>
<td>0.009*</td>
</tr>
<tr>
<td>75M</td>
<td>2</td>
<td>1000</td>
<td>20</td>
<td>0.000*</td>
<td>0.000*</td>
</tr>
<tr>
<td>25M</td>
<td>2</td>
<td>10000</td>
<td>20</td>
<td>0.000*</td>
<td>0.000*</td>
</tr>
<tr>
<td>50M</td>
<td>2</td>
<td>10000</td>
<td>20</td>
<td>0.000*</td>
<td>0.000*</td>
</tr>
<tr>
<td>75M</td>
<td>2</td>
<td>10000</td>
<td>20</td>
<td>0.000*</td>
<td>0.000*</td>
</tr>
<tr>
<td>25M</td>
<td>2</td>
<td>20000</td>
<td>20</td>
<td>0.000*</td>
<td>0.000*</td>
</tr>
<tr>
<td>75M</td>
<td>2</td>
<td>20000</td>
<td>20</td>
<td>0.393</td>
<td>0.015</td>
</tr>
<tr>
<td>25M</td>
<td>2</td>
<td>5000</td>
<td>20</td>
<td>0.000*</td>
<td>0.000*</td>
</tr>
<tr>
<td>50M</td>
<td>2</td>
<td>5000</td>
<td>20</td>
<td>0.000*</td>
<td>0.000*</td>
</tr>
<tr>
<td>75M</td>
<td>2</td>
<td>5000</td>
<td>20</td>
<td>0.000*</td>
<td>0.000*</td>
</tr>
<tr>
<td>25M</td>
<td>4</td>
<td>1000</td>
<td>20</td>
<td>0.000*</td>
<td>0.000*</td>
</tr>
<tr>
<td>50M</td>
<td>4</td>
<td>1000</td>
<td>20</td>
<td>0.000*</td>
<td>0.000*</td>
</tr>
<tr>
<td>75M</td>
<td>4</td>
<td>1000</td>
<td>20</td>
<td>0.000*</td>
<td>0.000*</td>
</tr>
<tr>
<td>25M</td>
<td>4</td>
<td>10000</td>
<td>20</td>
<td>0.000*</td>
<td>0.075</td>
</tr>
<tr>
<td>50M</td>
<td>4</td>
<td>10000</td>
<td>20</td>
<td>0.005*</td>
<td>0.015</td>
</tr>
<tr>
<td>75M</td>
<td>4</td>
<td>10000</td>
<td>20</td>
<td>0.019</td>
<td>0.000*</td>
</tr>
<tr>
<td>25M</td>
<td>4</td>
<td>20000</td>
<td>20</td>
<td>0.000*</td>
<td>0.000*</td>
</tr>
<tr>
<td>50M</td>
<td>4</td>
<td>20000</td>
<td>20</td>
<td>0.000*</td>
<td>0.000*</td>
</tr>
<tr>
<td>75M</td>
<td>4</td>
<td>20000</td>
<td>20</td>
<td>0.000*</td>
<td>0.000*</td>
</tr>
<tr>
<td>25M</td>
<td>4</td>
<td>5000</td>
<td>20</td>
<td>0.000*</td>
<td>0.000*</td>
</tr>
<tr>
<td>50M</td>
<td>4</td>
<td>5000</td>
<td>20</td>
<td>0.000*</td>
<td>0.001*</td>
</tr>
<tr>
<td>75M</td>
<td>4</td>
<td>5000</td>
<td>20</td>
<td>0.393</td>
<td>0.019</td>
</tr>
<tr>
<td>25M</td>
<td>6</td>
<td>1000</td>
<td>20</td>
<td>0.000*</td>
<td>0.000*</td>
</tr>
<tr>
<td>50M</td>
<td>6</td>
<td>1000</td>
<td>20</td>
<td>0.000*</td>
<td>0.000*</td>
</tr>
<tr>
<td>75M</td>
<td>6</td>
<td>1000</td>
<td>20</td>
<td>0.000*</td>
<td>0.000*</td>
</tr>
<tr>
<td>25M</td>
<td>6</td>
<td>10000</td>
<td>20</td>
<td>0.000*</td>
<td>0.000*</td>
</tr>
<tr>
<td>50M</td>
<td>6</td>
<td>10000</td>
<td>20</td>
<td>0.000*</td>
<td>0.000*</td>
</tr>
<tr>
<td>75M</td>
<td>6</td>
<td>10000</td>
<td>20</td>
<td>0.000*</td>
<td>0.000*</td>
</tr>
<tr>
<td>25M</td>
<td>6</td>
<td>5000</td>
<td>20</td>
<td>0.000*</td>
<td>0.000*</td>
</tr>
<tr>
<td>50M</td>
<td>6</td>
<td>5000</td>
<td>20</td>
<td>0.000*</td>
<td>0.001*</td>
</tr>
<tr>
<td>75M</td>
<td>6</td>
<td>5000</td>
<td>20</td>
<td>0.075</td>
<td>0.000*</td>
</tr>
<tr>
<td>25M</td>
<td>8</td>
<td>1000</td>
<td>20</td>
<td>0.000*</td>
<td>0.000*</td>
</tr>
<tr>
<td>50M</td>
<td>8</td>
<td>1000</td>
<td>20</td>
<td>0.000*</td>
<td>0.000*</td>
</tr>
<tr>
<td>75M</td>
<td>8</td>
<td>1000</td>
<td>20</td>
<td>0.000*</td>
<td>0.000*</td>
</tr>
<tr>
<td>25M</td>
<td>8</td>
<td>10000</td>
<td>20</td>
<td>0.000*</td>
<td>0.000*</td>
</tr>
<tr>
<td>50M</td>
<td>8</td>
<td>10000</td>
<td>20</td>
<td>0.000*</td>
<td>0.000*</td>
</tr>
<tr>
<td>75M</td>
<td>8</td>
<td>10000</td>
<td>20</td>
<td>0.004*</td>
<td>0.000*</td>
</tr>
<tr>
<td>25M</td>
<td>8</td>
<td>5000</td>
<td>20</td>
<td>0.000*</td>
<td>0.000*</td>
</tr>
<tr>
<td>50M</td>
<td>8</td>
<td>5000</td>
<td>20</td>
<td>0.000*</td>
<td>0.000*</td>
</tr>
<tr>
<td>75M</td>
<td>8</td>
<td>5000</td>
<td>20</td>
<td>0.000*</td>
<td>0.000*</td>
</tr>
</tbody>
</table>

Table 10: Results from Mann-Whitney-Wilcoxon tests for the second set of results for time used for delayed duplicate detection (DDD) and for the time used for the whole run. The tests were executed for each individual file used as an input for the two algorithms. P-values significant at 1 percent level have been marked with asterisk (*).

### 5.6.3 Locality

Table 11 shows the number of order graph nodes expanded and written to disk during the runtime of the algorithms for 29 variables. It shows the same result as the tables with 20 variables: the algorithm using hashing in delayed duplicate detection writes to disk clearly a larger amount of order graph nodes.
<table>
<thead>
<tr>
<th>maxnodes</th>
<th>25M</th>
<th>50M</th>
<th>75M</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>codebase</td>
<td>expanded</td>
<td>written</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>29</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>29</td>
<td>406</td>
<td>29</td>
</tr>
<tr>
<td>2</td>
<td>406</td>
<td>3,654</td>
<td>406</td>
</tr>
<tr>
<td>3</td>
<td>3,654</td>
<td>23,751</td>
<td>3,654</td>
</tr>
<tr>
<td>5</td>
<td>118,755</td>
<td>475,020</td>
<td>118,755</td>
</tr>
<tr>
<td>6</td>
<td>475,020</td>
<td>1,560,780</td>
<td>475,020</td>
</tr>
<tr>
<td>7</td>
<td>1,560,780</td>
<td>4,292,145</td>
<td>1,560,780</td>
</tr>
<tr>
<td>8</td>
<td>4,292,145</td>
<td>1,001,005,005</td>
<td>4,292,145</td>
</tr>
<tr>
<td>9</td>
<td>10,015,005</td>
<td>20,030,010</td>
<td>10,015,005</td>
</tr>
<tr>
<td>10</td>
<td>20,030,010</td>
<td>63,099,947</td>
<td>20,030,010</td>
</tr>
<tr>
<td>11</td>
<td>34,597,290</td>
<td>121,474,189</td>
<td>34,597,290</td>
</tr>
<tr>
<td>12</td>
<td>51,895,935</td>
<td>192,163,998</td>
<td>51,895,935</td>
</tr>
<tr>
<td>13</td>
<td>77,558,760</td>
<td>248,842,458</td>
<td>77,558,760</td>
</tr>
<tr>
<td>14</td>
<td>77,558,760</td>
<td>190,639,480</td>
<td>77,558,760</td>
</tr>
<tr>
<td>15</td>
<td>67,863,915</td>
<td>137,836,117</td>
<td>67,863,915</td>
</tr>
<tr>
<td>16</td>
<td>77,558,760</td>
<td>248,842,458</td>
<td>77,558,760</td>
</tr>
<tr>
<td>17</td>
<td>51,895,935</td>
<td>71,474,182</td>
<td>51,895,935</td>
</tr>
<tr>
<td>18</td>
<td>34,597,290</td>
<td>20,030,010</td>
<td>34,597,290</td>
</tr>
<tr>
<td>19</td>
<td>20,030,010</td>
<td>10,015,005</td>
<td>20,030,010</td>
</tr>
<tr>
<td>20</td>
<td>10,015,005</td>
<td>4,292,145</td>
<td>10,015,005</td>
</tr>
<tr>
<td>21</td>
<td>4,292,145</td>
<td>1,560,780</td>
<td>4,292,145</td>
</tr>
<tr>
<td>22</td>
<td>1,560,780</td>
<td>475,020</td>
<td>1,560,780</td>
</tr>
<tr>
<td>25</td>
<td>23,751</td>
<td>3,654</td>
<td>23,751</td>
</tr>
<tr>
<td>26</td>
<td>3,654</td>
<td>406</td>
<td>3,654</td>
</tr>
<tr>
<td>27</td>
<td>406</td>
<td>29</td>
<td>406</td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>1</td>
<td>29</td>
</tr>
</tbody>
</table>

All: 536,870,911, 1,329,244,743, 536,870,911, 962,536,992, 536,870,911, 642,516,295

Table 11: Order graph nodes expanded and written to disk for maximum number of nodes taking values 25M, 50M and 75M for hashing and sorting codebases.
6 Conclusions

This thesis has presented a way to use hashing-based delayed duplicate detection in structure learning of Bayesian networks. The method developed divides the candidate order graph nodes to files on the disk by using the first-\(k\) variables of the candidate node. This way a single layer of the order graph can be divided so that a single file on the disk contains at most a given amount of unique order graph nodes.

The hashing-based delayed duplicate detection has proven to be a more efficient method than sorting-based methods in other application areas, but in structure learning of Bayesian networks the results are mixed. The results presented in this thesis show that the order in which the candidate solutions are read from the earlier layer during the search through the order graph affects the number of duplicates detected in memory. The locality measure presented shows that this order can have a significant impact on the number of candidate solutions written to disk during delayed duplicate detection. Overall, the timing results presented for the different datasets show mixed results. The disparities between otherwise identical experiments suggests that using a shared environment with other users can have a major effect on the runtime of the algorithm, and hashing-based delayed duplicate detection seems to be affected differently by this than sorting-based delayed duplicate detection algorithm.

There are several areas for future work. Different division of combinations into files on the disk per first-\(k\) subcombinations could improve the efficiency of hashing the candidate solutions to disk; the current method unnecessarily increases the number of files because the combinations are unevenly distributed between the created files. A more efficient division of combinations to files could decrease the number of files needed during the search. However, it is unclear whether there is a possibility to find such division generically without enumerating the different candidate solutions during the search.

The algorithm presented here could also be parallelized in the same manner as suggested by Korf [Kor08]. He parallelized the writing and reading of the candidate solutions to the files on the disk. This could speed up the search by allowing both candidate generation and delayed duplicate detection phases to run parallel on multiple cores. After all, the results clearly suggest that the delayed duplicate detection with hashing is still only a minor portion of the runtime of the whole algorithm. The algorithm could be created using an idea of a "stop the world" pause between
the writing and reading phases. This pause is necessary because the delayed duplicate detection cannot be performed before all of the candidates have been generated. However, a clear problem with parallelization is that the in-memory duplicate detection utilizes a hash table, which means that the parallelization would slow down the in-memory duplicate detection since multiple threads would be accessing the same hash table at the same time. Korf did not experience this problem as his solution uses only delayed duplicate detection. Of course, another possibility is to test the speedup that the hybrid duplicate detection gives to the whole algorithm compared to the parallelized algorithm using only delayed duplicate detection.

One more possible venue of future research would be to investigate how the order of reading the order graph nodes from the previous layer can affect the number of nodes written to disk during delayed duplicate detection. This could be an interesting future research area especially, because right now the candidate order graph nodes are not deliberately generated in a certain order, but instead just created in a brute force manner. If there was a certain way of generating the candidate solutions, which would allow for an efficient in memory duplicate detection, this would of course make the generation of candidate solutions more efficient than what it is right now. Based on the results presented in this thesis it can be said that the order in which the sorting-based algorithm generates the candidate solutions implicitly generates the candidate solutions in a more suitable order for in-memory duplicate detection than the hashing-based algorithm presented in this thesis. This observation has not been made previously and it is also important in other application domains of delayed duplicate detection than only Bayesian network structure learning.
References


