Journal of Machine Learning Research 14 (2013) 1387-1415 Submitted 1/11; Revised 12/12; Published 5/13

Finding Optimal Bayesian Networks Using Precedence Constraints[*]

Pekka Parviainen PEKKAPA@KTH.SE
Science for Life Laboratory
School of Computer Science and Communication
Royal Institute of Technology (KTH)
Tomtebodavägen 23A
17121 Solna, Sweden

Mikko Koivisto MIKKO.KOIVISTO@CS.HELSINKI.FI
Helsinki Institute for Information Technology
Department of Computer Science
University of Helsinki
Gustaf Hällströmin katu 2b
00014 Helsinki, Finland

Editor: Chris Meek

Abstract

We consider the problem of finding a directed acyclic graph (DAG) that optimizes a decomposable Bayesian network score. While in a favorable case an optimal DAG can be found in polynomial time, in the worst case the fastest known algorithms rely on dynamic programming across the node subsets, taking time and space 2^n, to within a factor polynomial in the number of nodes n. In practice, these algorithms are feasible to networks of at most around 30 nodes, mainly due to the large space requirement. Here, we generalize the dynamic programming approach to enhance its feasibility in three dimensions: first, the user may trade space against time; second, the proposed algorithms easily and efficiently parallelize onto thousands of processors; third, the algorithms can exploit any prior knowledge about the precedence relation on the nodes. Underlying all these results is the key observation that, given a partial order P on the nodes, an optimal DAG compatible with P can be found in time and space roughly proportional to the number of ideals of P, which can be significantly less than 2^n. Considering sufficiently many carefully chosen partial orders guarantees that a globally optimal DAG will be found. Aside from the generic scheme, we present and analyze concrete tradeoff schemes based on parallel bucket orders.

Keywords: exact algorithm, parallelization, partial order, space-time tradeoff, structure learning

1. Introduction

During the last two decades, Bayesian networks (BNs) have become one of the most popular and powerful frameworks for modeling various aspects of intelligent reasoning, such as degrees of belief, causality, and responsibility (Pearl, 1988, 2000; Chockler and Halpern, 2004). While the conceptual basis of BNs can be regarded as satisfactory to a large extent, there remain computational bottlenecks that currently limit the utilization of BNs in large and combinatorially complex do-

[*]. A preliminary version of this work appeared in UAI 2009 (Parviainen and Koivisto, 2009). Part of the work was done while the first author (P.P.) was affiliated with Helsinki Institute for Information Technology at University of Helsinki. All correspondence should be addressed to the first author.

mains. Underlying many of these bottlenecks is the graphical structure of the model, a *directed acyclic graph* (DAG). In particular, when one is supposed to learn a BN from data (Verma and Pearl, 1990; Spirtes and Glymour, 1991; Cooper and Herskovits, 1992; Heckerman et al., 1995), in principle, one has to exhaust the space of all possible DAGs, which can be enormous and does not easily factorize due to the acyclicity constraint. Indeed, when formalized in a natural manner, the problem is known to be NP-hard (Chickering, 1996; Chickering et al., 2004). Consequently, much of the machine learning research on BNs has focused on tractable special cases or other restrictive assumptions and, of course, on various heuristics.

However, the continued increase in computational resources and the advances in algorithmic techniques have recently turned many researchers' attention to exact means for learning BNs from data. Common to such endeavors is that the problem is cast as optimization (or sometimes as integration) of a scoring function that assigns each possible DAG a real number reflecting how well the DAG fits the given data. Furthermore, it is assumed that the scoring function decomposes into a sum of local terms, each local term depending on a child node and its parent nodes. For this optimization problem, techniques similar to the classic dynamic programming (DP) treatment of the traveling salesman problem (Bellman, 1962; Held and Karp, 1962) have yielded exponential-time algorithms that solve instances of up to around 30 nodes with feasible worst-case runtime guarantees (Ott and Miyano, 2003; Koivisto and Sood, 2004; Silander and Myllymäki, 2006; Singh and Moore, 2005). Note that, while heuristic search algorithms may *often* find an optimal DAG in instances of this size, that certainly does not happen *always* and there is usually no practical way to verify or falsify a claim of optimality. Exact algorithms completely avoid that major uncertainty concerning the quality of the algorithm's output.[1] Besides, they provide tools for the design and analysis of heuristic methods that better scale up to larger instances. There are obvious interests in extending the scope of exact algorithms.

The existing DP algorithms suffer from two major shortcomings: they have huge memory requirements and they do not parallelize efficiently. For example, a streamlined implementation of the DP algorithm, by Silander and Myllymäki (2006), is able to handle 29 nodes in around 10 hours, but, needing almost 100 gigabytes of hard disk aside a main memory of a few gigabytes. As such, the algorithm cannot handle larger instances. However, supposing the memory requirement was not an issue and that the algorithm could be run in parallel on thousands of processors, much larger instances could be solved. For example, a 40-node instance would take just about the same 10 hours if given $2^{11} = 2048$ processors, or about a week if given one hundred processors. Unfortunately, it is notoriously difficult to save space in related DP algorithms without an increase in the running time (Bellman, 1962; Bodlaender et al., 2006, 2012). Thus, a plausible goal is to trade as little time as possible for a fair amount of space and parallelization capacity.

In this article, we introduce new algorithmic schemes that address the issues of memory requirement and parallelization of the DP algorithms for the BN learning problem. Our contributions stem from the following observation: Suppose we know a priori some *precedence constraints* that an optimal DAG we are searching for must obey; that is, if node u is constrained to precede node

1. The reader may wonder whether the search for an optimal DAG is important at all. Might not noise in the data, limitations of the score function, etc., dampen any advantage of the optimality procedure over, say, heuristic local search? Yes, they might. When that is the case, the research, indeed, ought to focus on removing the corresponding bottlenecks, that is, improve data quality, the model, etc., which is a matter of more practical case-by-case studies. But we also believe there is room for research in settings where "optimal" is truly more desirable than "arbitrary", and research on BNs is not an exception.

v, then the DAG must not contain a directed path from v to u. In an extreme case, the precedence constraints specify a linear order on the nodes, and more generally, the constraints specify a partial order. The key observation is that the DP algorithms can be extended to exploit the precedence constraints to save both time and space. Specifically, the time and space requirements can be made to grow roughly linearly in the number of node subsets that are closed under taking predecessors, called *ideals* (or downsets) of the partial order. As the number of ideals can be much smaller than the number of all node subsets, depending on the width of the partial order, there is potential for a significant improvement over the existing DP algorithms, which are ignorant of precedence constraints. To realize this potential, we need to consider partial orders that have only few ideals. The second part of our observation addresses this requirement: Instead of assuming a single given partial order, we may consider multiple partial orders that together cover all possible DAGs, that is, every DAG is compatible with at least one of the partial orders. Each partial order amounts to a subproblem that can be solved independently of the others, until only finally the best of the solutions to the subproblems is returned. Because of the independence of the subproblems, the computations can be run in parallel on as many processors as there are partial orders. We are left with the freedom to choose either many partial orders, each with only few ideals (the extreme is to consider all linear orders), or just a few partial orders, each with many ideals, or something in between. This freedom allows us to trade time for space and parallelization in a smooth fashion, to adapt to the available resources.

We begin the remainder of this article in Section 2 by formulating the optimization problem in question more carefully and by reviewing the basic DP algorithm. Section 3 illustrates the problem setting by describing a simple scheme, called the *two-bucket scheme*, that enables trading time for space and parallelization. While that scheme is *per se* rather inefficient, it serves as a base for the developments that follow in later sections. Specifically, Section 4 combines the two-bucket scheme with divide and conquer to get efficient tradeoff in the small-space regime. While the resulting algorithms are, admittedly, of merely theoretical interest, they connect to and extend what is known about the polynomial-space solvability of related classic problems, such as the traveling salesman problem (Savitch, 1970; Gurevich and Shelah, 1987; Björklund and Husfeldt, 2008; Bodlaender et al., 2006, 2012). Section 5, on the other hand, generalizes the two-bucket scheme into a generic partial order approach, constituting the main conceptual and technical contribution of this work. The generic approach being quite abstract, its more concrete implications are derived in Section 6 for a particular class of partial orders, namely (parallel compositions of) bucket orders. Some of the combinatorial analyses concerning bucket orders build on our on-going work on related permutation problems—some results have been published in a preliminary form in a conference proceedings (Koivisto and Parviainen, 2010) that we will cite in Section 6. Finally, in Section 7, we summarize our main findings and discuss how they advance the state of the art in learning BNs from data.

After the publication of a preliminary version of this work (Parviainen and Koivisto, 2009), several notable related results have been announced by other groups. Branch-and-bound methods exploiting local constraints are proposed by de Campos et al. (2009, 2011) and Etminani et al. (2010). Jaakkola et al. (2010) and Cussens (2011) have developed integer linear programming techniques and achieved rather encouraging empirical results. Malone et al. (2011) and Yuan et al. (2011) propose tighter implementations of the DP algorithm, yielding moderate savings in memory and time usage. Finally, Tamada et al. (2011) present a parallelization scheme with somewhat involved communication routines. We discuss the merits and limitations of these methods and their relation to our work more thoroughly in Section 7.

2. Preliminaries

This section presents the basic terminology and formulations needed in later sections. In particular, we formalize the computational problem of finding an optimal Bayesian network, accompanied with some remarks concerning the representation of the input. Then we review a dynamic programming algorithm that sets the technical and conceptual baseline for the developments in Sections 3–6.

2.1 The Optimal Bayesian Network Problem

A Bayesian network is a multivariate probability distribution that obeys a structural representation in terms of a directed acyclic graph (DAG) and a corresponding collection of univariate conditional probability distributions. For our purposes, it is crucial to treat the DAG, that is, the *network structure*, explicitly, whereas the conditional probabilities will enter our formalism only implicitly. Formally, a DAG is a pair (N, A), where N is the *node set* and $A \subseteq N \times N$ is the *arc set*. A node u is said to be a *parent* of a node v if the arc uv is in A. The *parent set* of a node v consists of the parents of v and is denoted by A_v. When there is no ambiguity about the node set we identify the DAG with its arc set. We denote the cardinality of N by n. The *number of parents* or *indegree* of node v in A is simply $|A_v|$. A node that is not a parent of any node is called a *sink* of the DAG.

We formulate the task of learning a Bayesian network from data as a generic optimization problem over DAGs on a given node set N. Specifically, we assume that each DAG A is associated with a real number $f(A)$ that specifies how well the Bayesian networks with structure A fit the given data. In particular, the *scoring function* f can take any of the various forms derived under different paradigms for statistical inference, chiefly, the Bayesian, the maximum-likelihood, and the minimum description length paradigms. For examples of concrete scoring functions and their justifications, see, for instance, de Campos's (2006) review and the next paragraph. The optimization problem becomes algorithmically interesting when the function f has some structure. To this end, we make the usual assumption that the scoring function is *decomposable*, that is, for each node v there exist a *local scoring function* f_v such that

$$f(A) = \sum_{v \in N} f_v(A_v)\,,$$

for all DAGs A on N; each f_v is a function from the subsets of $N \setminus \{v\}$ to real numbers. We call $f(A)$ and $f_v(A_v)$ the *score* of A and the *(local) score* of A_v, respectively.

Example 1 (the Bayesian Dirichlet (BD) score) Heckerman et al. (1995) define the *Bayesian Dirichlet (BD) metric* as the joint probability distribution over the DAG and the data:

$$p(A) \prod_{v \in N} \prod_{x=1}^{q_v} \frac{\Gamma(m'_{ux})}{\Gamma(m'_{ux} + m_{ux})} \prod_{y=1}^{r_v} \frac{\Gamma(m'_{uxy} + m_{uxy})}{\Gamma(m'_{uxy})}\,,$$

where $p(A)$ is the prior of the structure A, r_v is the number of possible values of node v and q_v the number of possible value configurations of the parents of v in A. Furthermore, m_{uxy} is the number of data records in which u has value x and the parents of u in A have value configuration y. Here we assume some arbitrary but fixed labeling of the values and value configurations by numbers $1, 2, 3, \ldots$. The numbers m'_{uxy} are nonnegative reals specified by the modeler, and m_{ux} and m'_{ux} are obtained by taking the sum of the m_{uxy} and m'_{uxy}, respectively, over the range of y. The gamma function Γ appears in the expression as Dirichlet priors on the local conditional distributions are marginalized out.

The *BD score* is obtained by taking a logarithm of the BD metric. Whether the BD score is decomposable or not depends on the choice of the prior $p(A)$. Often the prior is assigned such that it factorizes into a product $\prod_v \rho_v(A_v)$, with, for instance $\rho_v(A_v) = c\kappa^{|A_v|}$ for some constants c and κ independent of A_v but possibly dependent on v (Heckerman et al., 1995). With such a prior, the BD score is easily seen to yield a decomposable scoring function.

Definition 1 (the OBN problem) Given a decomposable scoring function f as input, the *optimal Bayesian network* problem is to output a DAG A that maximizes $f(A)$.

Concerning the representation of the problem input, some remarks are in order. First, while the notion of decomposability concerns the mere existence of local scoring functions, we naturally assume that the functions f_v are given explicitly as input. In practice, the values $f_v(A_v)$ are computed for every relevant A_v based on the data and the chosen scoring function. Second, we allow $f_v(A_v)$ to take the value $-\infty$ to indicate that A_v cannot be the parent set of v. In fact, we assume that a collection of *potential parent sets*, denoted as $\mathcal{F}_v$, is given as input, with the understanding that outside that collection the values are $-\infty$. Third, we remark that the size of $\mathcal{F}_v$ may often be much less than the theoretical maximum 2^{n-1}, due to several potential reasons:

(a) The *maximum number of parents* is set to k, a parameter specified by the modeler.

(b) The parents are assumed to be contained in a predetermined (small) *set of candidates*.

(c) One can safely *ignore a parent set that has a subset with a better score* (observed by computing the score or by analytical bounds specific to the scoring function).

We note that (a) and (b) are often assumed by the modeller but do not hold in general, and are not assumed in the sequel, whereas (c) always holds. Common to (a) and (b) is that they yield a *downward closed* collection of parent sets, that is, a collection that is closed with respect to set inclusion. While the same does not hold for (c), it is plausible to expect that the pruned collection is not much smaller than the *downward closure* of the collection, obtained by taking all members of the collection and their subsets: In theory, the *downward closure* can be larger than the pruned collection by a factor of about 2^k, where k is the size of the largest parent set; for a small constant k, this factor is not large. In practice, however, we have observed significantly smaller factors, typically not exceeding 2.[2] Motivated by this proximity, we for technical ease take the downward closure of the pruned collection as the collection $\mathcal{F}_v$. For the state of the art in pruning parent sets we refer to the results of de Campos and Ji (2011).

These issues become relevant when it comes to representing the input in a data structure that takes relatively little space but enables fast fetching of local scores. If only (a) and (b) hold, then a

2. We have examined the pruned parent set collections of 52 data sets made available by James Cussens at `www.cs.york.ac.uk/aig/sw/gobnilp/data/`. In 21 of the data sets, the maximum number of parents was set to 3, and in the rest it was set to 2. We found that the factor—defined as the size of the downward closure of the pruned collection divided by the size of the pruned collection—varied (over the data sets and nodes) from 1.0 to 4.2. The median value over the nodes varied (over the data sets) from 1.0 to 1.8. The maximum over the nodes varied from 1.0 to 4.2 and was below 2.7 for all but one data set; the value 4.2 was due to having 5 members in the pruned collection while 21 members in the closure. This suggests that larger factors occur only when the closure is small, in which case the size of the closure is not a bottleneck in the computations.

simple array representation would suffice, because the regularity enables efficient indexing. However, to allow for general downward-closed collections we will work with the following *augmented representation*. We assume that each f_v is provided as an array of tuples $(Z, f_v(Z), U)$, where $Z \in \mathcal{F}_v$ and U consists of the nodes u outside Z satisfying $Z \cup \{u\} \in \mathcal{F}_v$. We further assume that the array is ordered lexicographically by Z (with respect to an arbitrary but fixed ordering of the nodes). This representation is motivated by the following observation concerning set interval queries, which we will need in Section 5.2.

The *interval* from set X to set Y, denoted as $[X, Y]$, is the collection $\{Z : X \subseteq Z \subseteq Y\}$.

Proposition 2 (interval queries) *Given a downward closed collection $\mathcal{F}_v$ in the augmented representation and sets $X, Y \subseteq N \setminus \{v\}$, the scores $f_v(Z)$ for all $Z \in \mathcal{F}_v$ in the interval $[X, Y]$ can be listed in $O(n)$ time per score.*

Proof First, search for the tuple $(X, f_v(X), U)$. If no tuple is found, then stop and list no sets—this is correct, because a downward closed collection intersects the interval $[X, Y]$ only if X belongs to the collection. Otherwise let $Z = X$, list $f_v(Z)$, and proceed recursively to the tuples of $Z' = Z \cup \{u\}$ for each $u \in U$ that belongs to $Y \setminus Z$ and succeeds the maximum node in $Z \setminus X$. Clearly, the algorithm visits all the desired tuples exactly once. For each visited tuple, locating it takes $O(\log |\mathcal{F}_v|)$ time using binary search, and traversing through the nodes in U takes $O(n)$ time. Since $\log |\mathcal{F}_v| = O(n)$, the claimed time bound follows. ■

It is worth noting that the simple array representation, for cases (a) and (b), is more efficient. Compared to the augmented representation, it takes less space and enables faster interval queries, both by a factor linear in n. We leave the verification of this to the reader.

As suggested by the above discussion, we gauge the time and space requirements of an algorithm by the number of basic operations it executes and the maximum storage size it needs at any point of its execution, respectively. More specifically, by basic operations we refer not only to addition and comparison of real numbers but also to indexing arrays by nodes or node subsets. The storage size is assumed to be constant for real numbers and individual nodes.

2.2 Dynamic Programming

The OBN problem can be solved by dynamic programming across the node subsets (Ott and Miyano, 2003; Koivisto and Sood, 2004; Silander and Myllymäki, 2006; Singh and Moore, 2005). The idea of the algorithm can be described in terms of node orderings, as follows. Suppose $\hat{A}$ is an optimal DAG, that is, it maximizes the scoring function f. Because $\hat{A}$ is acyclic, there is at least one topological ordering of the nodes that is compatible with $\hat{A}$, that is, if uv is an arc in $\hat{A}$, then u precedes v in the topological ordering. We will call any such ordering an *optimal linear order* on the nodes. Now, vice versa, if an optimal linear order is given, finding an optimal DAG is relatively simple: for each node v, independently, find a best-score parent set[3] among its predecessors in $O(|\mathcal{F}_v|)$ time. Now, when such an optimal linear order is not given, we basically need to consider all possible linear orders on the nodes. The key observation is that two linear orders imply the same best-score

3. Finding a best-score parent set is computationally hard for the usual local scoring functions, assuming the scoring function is not given by an explicit array of numbers but by an implicit expression, like the one in Example 1, that is efficient to evaluate for a given data set (Koivisto, 2006). In the worst case, there is little hope for doing better than examining all possible sets up to size logarithmic in the number of data points.

for node v whenever the set of predecessors of v are the same in the orders. Thus the algorithm only needs to tabulate the best cumulated scores for the node subsets. In the following paragraphs we present a dynamic programming algorithm that proceeds in two phases. We omit a rigorous proof of correctness. See Section 5 for a proof concerning a generalization of the algorithm.

In the *first phase*, the algorithm computes the best-score for each node v and set of predecessors $Y \subseteq N \setminus \{v\}$, defined as

$$\hat{f}_v(Y) = \max_{X \subseteq Y} f_v(X) .$$

The direct computation of $\hat{f}_v(Y)$ for any fixed v and Y requires $2^{|Y|}$ basic operations. Thus, the total number of basic operations scales as $n \sum_{i=0}^{n-1} \binom{n-1}{i} 2^i = n3^{n-1}$. However, this can be significantly lowered by the following observation.

Lemma 3 (Ott and Miyano 2003) *If $v \in N$ and $Y \subseteq N \setminus \{v\}$, then*

$$\hat{f}_v(Y) = \max \Big\{ f_v(Y), \max_{u \in Y} \hat{f}_v(Y \setminus \{u\}) \Big\} .$$

This recurrence allows us to proceed levelwise, that is, in increasing cardinality of Y. If the values $\hat{f}_v(X)$ have already been computed for all $X \subset Y$ and stored in an array, computing $\hat{f}_v(Y)$ takes no more than n comparisons. Recall we assume that indexing by subsets takes only constant time. Thus, the values $\hat{f}_v(Y)$ for all v and Y can be computed in $O(n^2 2^n)$ time. We discuss the space requirement later.

In the *second phase*, the algorithm goes through all node subsets $Y \subseteq N$, tabulating the maximum score over all DAGs on Y, denoted as $g(Y)$. In particular, $g(N)$ is the maximum score over all DAGs on N. One easily finds the recurrence

$$g(Y) = \max_{v \in Y} \Big\{ g(Y \setminus \{v\}) + \hat{f}_v(Y \setminus \{v\}) \Big\} , \tag{1}$$

with $g(\emptyset) = 0$. In other words, $g(Y \setminus \{v\}) + \hat{f}_v(Y \setminus \{v\})$ is the maximum score over all DAGs on Y such that v is the "last node", that is, the parents of v are selected from $Y \setminus \{v\}$. Given the values $\hat{f}_v(Y)$ computed in the first phase, the values $g(Y)$ can be computed in $O(n2^n)$ basic operations. So the time requirement of the whole algorithm is $O(n^2 2^n)$.

Like the time requirement, also the space requirement is dominated by the first phase, the management of the values $\hat{f}_v(Y)$. If all intermediate results are kept in memory, the space requirement is $O(n2^n)$. However, this can be reduced to $O(\sqrt{n}2^n)$ by merging the two phases of the algorithm (Bellman, 1962; Ott and Miyano, 2003; Malone et al., 2011): Note that the computation of both $g(Y)$ and $\hat{f}_v(Y)$ requires information only about the sets of size $|Y| - 1$. Therefore, we can proceed levelwise and compute both phases for one level at a time. Thereby, at any level ℓ we need to keep in memory the values $g(Y)$ and $\hat{f}_v(Y)$ only for $O\big(\binom{n}{\ell} + \binom{n}{\ell-1}\big)$ sets Y of size ℓ and $\ell - 1$. Hence the space requirement is $O\big(n\binom{n}{\lceil n/2 \rceil}\big)$.

Once the optimal score has been computed, an optimal DAG can be constructed with negligible time and space overheads by back tracing, as follows. Starting with $Y = N$, first a node $v \in Y$ such that $g(Y) = g(Y \setminus \{v\}) + \hat{f}_v(Y \setminus \{v\})$ can be found in $O(n)$ time, since the required values of g and $\hat{f}_v$ have already been computed and are available. An optimal parent set for v can then be found by brute-force in $O(|\mathcal{F}_v|)$ time. If the values of g and $\hat{f}_v$ are kept in memory for all node subsets, then this step can be just repeated n times. Otherwise, if the values of g and $\hat{f}_v$ are kept in memory only for the node subsets at the last two levels, one can resolve the entire problem on the smaller node subset $Y \setminus \{v\}$ to recompute the needed values of g and $\hat{f}_v$. While this is repeated n times, the overall running time becomes not more than roughly two-fold, since the smaller instances are solved exponentially faster.

3. Two-Bucket Scheme

In this section we present a simple scheme for solving the OBN problem with less space, albeit more slowly. The idea is to guess the set of, say, s first nodes of an optimal linear order and then solve separately the independent subproblems on those s first nodes and on the last $n-s$ nodes. More formally and in terms of DAGs, the key observation is the following: Let $\hat{A}$ be an optimal DAG on the node set N. Fix an integer s with $n/2 \le s \le n$. Since $\hat{A}$ is acyclic, there exists a partition of N into two sets N_0 and N_1 of size s and $n-s$, respectively, such that every arc between N_0 and N_1 in $\hat{A}$ is directed from N_0 to N_1. In other words, the parents of any node in N_0 are from N_0, while a node in N_1 may have parents from both N_0 and N_1. Thus, one can find $\hat{A}$—strictly speaking, the associated optimal score $f(\hat{A})$—by trying out all possible ordered partitions (N_0, N_1) of N, with $|N_0| = s$ and $|N_1| = n-s$, and solving the recurrences

$$g_0(Y) = \max_{v \in Y} \left\{ g_0(Y \setminus \{v\}) + \hat{f}_v(Y \setminus \{v\}) \right\}, \tag{2}$$

for $\emptyset \subset Y \subseteq N_0$ with $g_0(\emptyset) = 0$, and

$$g_1(Y) = \max_{v \in Y} \left\{ g_1(Y \setminus \{v\}) + \hat{f}_v(N_0 \cup Y \setminus \{v\}) \right\}, \tag{3}$$

for $\emptyset \subset Y \subseteq N_1$ with $g_1(\emptyset) = 0$. The score of $\hat{A}$ is obtained as the maximum of $g_0(N_0) + g_1(N_1)$ over all the said partitions (N_0, N_1).

We notice that the two subproblems are independent of each other given the partition (N_0, N_1), and thus they can be solved separately. Applying the algorithm of the previous section, the computation of g_0 takes $O(2^s n^2)$ time and $O(2^s n)$ space.

Computing g_1 can be more expensive, since evaluating the term $\hat{f}_v(N_0 \cup Y \setminus \{v\})$ requires considering all possible subsets of N_0 as parents of v, in addition to a subset of $Y \setminus \{v\}$. To this end, at each $X_1 \subseteq N_1 \setminus \{v\}$ define

$$f'_v(X_1) = \max\left\{ f_v(X) : X \cap N_1 = X_1 \,, X \in \mathcal{F}_v \right\}$$

and

$$\hat{f}'_v(Y_1) = \max_{X_1 \subseteq Y_1} f'_v(X_1) \,.$$

Observe that computing $f'_v(X_1)$ for all X_1 takes $O((F + 2^{n-s})n)$ time in total, where F is the size of $\mathcal{F}_v$. Now, because $\hat{f}_v(N_0 \cup Y_1) = \hat{f}'_v(Y_1)$, the algorithm of the previous section again applies to compute g_1, running in $O((F + 2^{n-s})n^2 + 2^{n-s}n^2)$ time and $O(2^{n-s}n)$ space in total. Since there are $\binom{n}{s}$ possible partitions (N_0, N_1), we have the following:

Proposition 4 *OBN can be solved in $O(\binom{n}{s}(2^s + F)n^2)$ time and $O((2^s + F)n)$ space for any $s = n, n-1, \ldots, n/2$, assuming each node has at most F potential parent sets.*

When F is $O(2^s)$ we can derive simpler asymptotic bounds. For instance, putting $s = 4/5n$ yields $O(2.872^n)$ time and $O(1.742^n)$ space. In general, this approach yields a smooth time-space tradeoff for space bounds between $O(2^{n/2}n)$ and $O(2^n n)$ (see Figure 4 in Section 7). However, within this space complexity range a more efficient scheme exists, as we will show in Section 6. Furthermore, with space less than $O(2^{n/2}n)$ the above scheme is not applicable.

4. Divide and Conquer Scheme

The partitioning idea from the previous section can be applied recursively, as described next. To solve the subproblems, namely computing $g_0(N_0)$ and $g_1(N_1)$ via the recurrences (2–3), with less space, we apply the partitioning technique again. The problem of computing $g_0(N_0)$ is of the same form as the original problem, and can thus be treated in a straightforward manner. The computation of $g_1(N_1)$ is only little more involved. Like before, for any fixed integer s satisfying $0 \leq s \leq |N_1|$, there exists a partitioning of the node set N_1 into subsets N_{10} and N_{11} of size s and $|N_1| - s$, respectively, such that every arc between N_{10} and N_{11} in an optimal DAG $\hat{A}$ is directed from N_{10} to N_{11}. So, one can compute $g_1(N_1)$ by trying out all possible partitions (N_{10}, N_{11}) of N_1, with $|N_{10}| = s$ and $|N_{11}| = |N_1| - s$, and solving the recurrences

$$g_{10}(Y) = \max_{v \in Y} \Big\{ g_{10}(Y \setminus \{v\}) + \hat{f}_v((N_0 \cup Y \setminus \{v\}) \Big\},$$

for $\emptyset \subset Y \subseteq N_{10}$ with $g_{10}(\emptyset) = 0$, and

$$g_{11}(Y) = \max_{v \in Y} \Big\{ g_{11}(Y \setminus \{v\}) + \hat{f}_v(N_0 \cup N_{10} \cup Y \setminus \{v\}) \Big\},$$

for $\emptyset \subset Y \subseteq N_{11}$ with $g_{11}(\emptyset) = 0$. The score $g_1(N_1)$ is obtained as the maximum of $g_{10}(N_{10}) + g_{11}(N_{11})$ over all the said partitions (N_{10}, N_{11}). In general, one can apply partitioning recursively, say to depth d, and then solve the remaining subproblems by dynamic programming.

For an analysis of the time and space requirements, it is convenient to assume a balanced scheme: in every step of the recursion, the node set in question is partitioned into two sets of about equal sizes. For simplicity, assume n is a power of 2. Then, at depth $d \geq 0$ of the recurrence, the node set in each subproblem in question is of size $s = n/2^d$. Hence, each subproblem can be solved in time and space within a polynomial factor to 2^s, assuming the number of parents per node is polynomial in n; more precisely, if each node has at most F potential parent sets, the routines described in the previous sections take $O((2^s n + F)n)$ time and $O((2^s + F)n)$ space. Because each subproblem of size $2s$ is divided into $2\binom{2s}{s} \leq 2^{2s}$ subproblems of size s, the total number of subproblems of size s that need to be solved is at most $2^n 2^{n/2} 2^{n/4} \cdots 2^{2s} = 2^{2n-2s}$.

Theorem 5 *OBN can be solved in $O\big(2^{2n-2s}(2^s + F)n^2\big)$ time and $O((2^s + F)n)$ space for any $s = n, n/2, n/4, \ldots$, assuming each node has at most F potential parent sets.*

Choosing an $s \geq 0$ such that $2^s \leq F < 2^{s+1}$, gives a theoretically interesting implication:

Corollary 6 *OBN can be solved in $O(4^n n^2/F)$ time and $O(nF)$ space, assuming each node has at most F potential parent sets.*

In particular, when F grows polynomially in n, we have a polynomial-space algorithm whose running time scales roughly as 4^n. Analogous results are known for a number of related permutation problems, such as the traveling salesman problem, the minimum fill-in problem, the pathwidth problem, the cutwidth problem, the optimal linear arrangement problem, and the feedback arc set problem (Gurevich and Shelah, 1987; Björklund and Husfeldt, 2008; Bodlaender et al., 2012). The divide and conquer technique underlying all these results can be attributed to Savitch (1970).

5. The Partial Order Approach

This section generalizes the two-bucket scheme in a different direction than the divide and conquer scheme. Informally speaking, we replace a two-bucket partition by a partial order and, accordingly, the consideration of all fixed-size partitions by the consideration of sufficiently many partial orders so as to "cover" the linear orders.

We begin by introducing the needed concepts. Then the following three subsections generalize the two-phase dynamic programming algorithm. While Sections 5.2 and 5.3 present, respectively, the first and the second phase of the dynamic programming algorithm in a logical order, the treatment of the former is strongly motivated by the latter. The reader may prefer taking a look at the second phase first. We end this section with some remarks about choosing an efficient system of partial orders and about implications to parallel computation.

5.1 Partial Order Concepts

The following paragraphs introduce some concepts related to partial orders that will be needed in the remainder of this paper; for a more thorough treatment, an interested reader may refer, for instance, to the book by Davey and Priestley (2003).

Partial orders are binary relations that can be viewed as specializations of DAGs, for they inherit the acyclicity property of DAGs but also require additional properties of reflexivity and transitivity. But partial orders can also be viewed as extensions of DAGs, for the additional properties can be achieved by augmenting a DAG with appropriate arcs. Formally, a *partial order* P on *ground set* M is a subset of $M \times M$ such that for all $x, y, z \in M$ it holds that $xx \in P$ (reflexivity), $xy \in P$ and $yx \in P$ implies $y = x$ (antisymmetry), $xy \in P$ and $yz \in P$ implies $xz \in P$ (transitivity). A partial order P is a *linear order* if, in addition, for all $x, y \in M$ it holds that $xy \in P$ or $yx \in P$ (totality). A linear order Q is a *linear extension* of a partial order P if $P \subseteq Q$. If $xy \in P$ and $x \neq y$, we say that x is *smaller* than y and that y is *larger* than x; we denote by P_y the set of all elements that are smaller than y. An element x is *maximal* if no element is larger than x, and *minimal* if no element is smaller than x. The *trivial order* on M is the "diagonal" partial order $\{xx : x \in M\}$. If the ground set consists of a single element, we call the partial order a *singleton order*. Because a partial order P determines its ground set M, there will usually be no need to refer to the structure (M, P) known as a *partially ordered set (poset)*. By writing $xy \in P$ instead of xPy we emphasize the similar treatment of DAGs and partial orders. A notion of compatibility between DAGs and partial orders will be central in our developments:

Definition 7 (compatibility) A DAG A and a partial order P are said to be *compatible* with each other if there exists a partial order Q that is a superset of both A and P.

One of the key concepts we need is an *ideal* of a partial order. Informally, an ideal is a subset of the ground set that is closed under taking smaller elements. In the literature, ideals are also

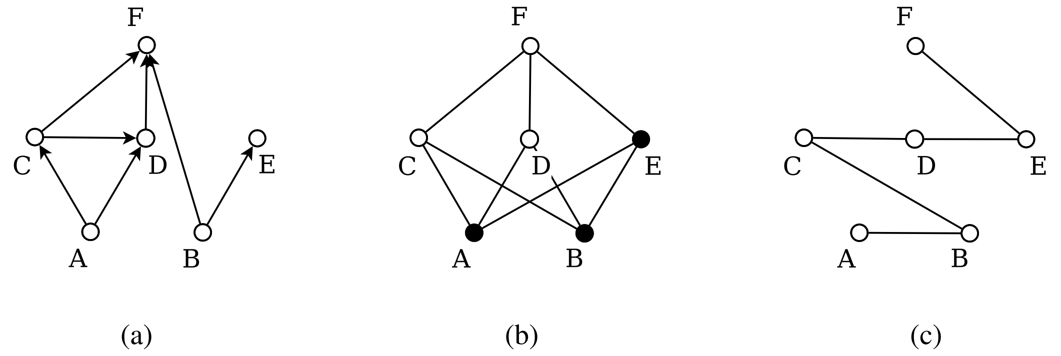

Figure 1: Three binary relations on the set $\{\mathrm{A,B,C,D,E,F}\}$. (a) A DAG. (b) A Hasse diagram of a partial order compatible with the DAG. The partial order has 12 ideals, namely $\emptyset$, $\{\mathrm{A}\}$, $\{\mathrm{B}\}$, $\{\mathrm{A,B}\}$, $\{\mathrm{A,B,C}\}$, $\{\mathrm{A,B,D}\}$, $\{\mathrm{A,B,E}\}$, $\{\mathrm{A,B,C,D}\}$, $\{\mathrm{A,B,C,E}\}$, $\{\mathrm{A,B,D,E}\}$, $\{\mathrm{A,B,C,D,E}\}$, and $\{\mathrm{A,B,C,D,E,F}\}$. One of the ideals is marked by black dots. (c) A Hasse diagram of a linear order that is an extension (superset) of both the DAG and the partial order.

called *order ideals* or *downsets*. We will show that, when maximizing a decomposable scoring function over DAGs that are compatible with a given a partial order, the basic dynamic programming algorithm can be trimmed to run only across the ideals. We illustrate this soon in Example 2 and give a general treatment in Sections 5.2 and 5.3.

Definition 8 (ideal) Let P be a partial order on M. A subset I of M is called an *ideal* of P if $y \in I$ and $xy \in P$ imply that $x \in I$. We denote the set of all ideals of P by $\mathcal{I}(P)$.

Figure 1 illustrates some of the above defined concepts. Note that we visualize a partially ordered set (M, P) by its *transitive reduction*, that is, the graph obtained by removing from P all pairs xy for which $x = y$ or there exist a z that is larger than x and smaller than y. We draw a transitive reduction on the plane using a Hasse diagram: a pair xy in the reduction corresponds to a line segment such that x is positioned below or to the left of y.

Example 2 (dynamic programming across ideals) Consider maximizing a decomposable scoring function over DAGs that are compatible with the partial order given in Figure 1(b). Because the score is decomposable, an optimal DAG $\hat{A}$ must have a sink node $v \in N = \{\mathrm{A,B,C,D,E,F}\}$ such that the parent set $\hat{A}_v$ of v maximizes the local score over all subsets of $N \setminus \{v\}$, and the remainder of the DAG maximizes the score over DAGs on $N \setminus \{v\}$—this is the essence of the basic dynamic programming recurrence (1). For concreteness, the reader may think of the DAG of Figure 1(a) as the optimal DAG. The key observation we can make here is that, due to the precedence constraints given by the partial order, not all nodes $v \in N$ need to be considered as the possible sink node. Indeed, since F is the only maximal element in the partial order, it has to be a sink of any DAG compatible with the partial order. Thus, in the dynamic programming recurrence it suffices to consider only one out of the six cases, namely recursing on the subproblem of maximizing the score over DAGs on $N \setminus \{\mathrm{F}\} = \{\mathrm{A,B,C,D,E}\}$. It is not a coincidence that this is the only ideal of the partial order of size 5. Continuing the same reasoning to smaller subproblems shows that it is sufficient to tabulate the optimal scores (DAGs) for the ideals of the partial order.

We will work with collections of partial orders that share the same ground set. We call such a collection a *partial order system* on the ground set, or *POS* for short. Our interest is particularly in partial order systems that exhibit sufficient diversity so as to "cover" all linear orders on the ground set. This idea is formalized in the following:

Definition 9 (cover) A POS $\mathcal{P}$ on M is said to be a *cover* on M if any linear order on M is an extension of at least one partial order in $\mathcal{P}$.

An extreme example of a cover on a ground set M is the collection $\{T\}$ formed by the trivial order T on M. At the other extreme we have the cover that consists of all linear orders on M. For yet another example, consider the three partial orders defined by $P = \{\mathrm{AA}, \mathrm{BB}, \mathrm{CC}, \mathrm{AB}, \mathrm{AC}\}$, $Q = \{\mathrm{AA}, \mathrm{BB}, \mathrm{CC}, \mathrm{BA}, \mathrm{BC}\}$, and $R = \{\mathrm{AA}, \mathrm{BB}, \mathrm{CC}, \mathrm{CA}, \mathrm{CB}\}$. We see that the system $\{P, Q, R\}$ is a cover on the ground set $\{\mathrm{A}, \mathrm{B}, \mathrm{C}\}$.

5.2 Dynamic Programming: First Phase

We aim at an algorithm that maximizes $f(A)$ over all DAGs A on the node set N subject to the constraint that A is compatible with a given partial order P. In this subsection, we modify the first phase of the basic dynamic programming algorithm, described in Section 2.2, accordingly. In the next subsection, we will modify the second phase.

Recall that in the first phase, the task is to compute the values $\hat{f}_v(Y)$ for all nodes v and node subsets $Y \subseteq N \setminus \{v\}$. However, it turns out that, in the second phase, we need these values only for the ideals Y of the given partial order P. This gives us an opportunity to save space, provided that we have an appropriate "sparse" variant of the recurrence of Lemma 3 at hand. Next we present such a variant.

Our key insight is the following simple observation concerning collections of subsets. We leave the proof to the reader.

Lemma 10 *Let X and Y be sets with $X \subseteq Y$. Let*

$$\mathcal{A} = [X, Y] \quad \textit{and} \quad \mathcal{B} = \{Z \subseteq Y : x \notin Z \textit{ for some } x \in X\}\,.$$

Then (i) $2^Y = \mathcal{A} \cup \mathcal{B}$ and (ii) $\mathcal{B} = \bigcup_{x \in X} 2^{Y \setminus \{x\}}$.

In terms of the set functions f_v and $\hat{f}_v$, for an arbitrary v, Lemma 10 amounts to the following generalization of Lemma 3 (one obtains Lemma 3 at $X = Y$):

Lemma 11 *Let X and Y be subsets of $N \setminus \{v\}$ with $X \subseteq Y$. Then*

$$\hat{f}_v(Y) = \max\Big\{ \max_{X \subseteq Z \subseteq Y} f_v(Z),\ \max_{u \in X} \hat{f}_v(Y \setminus \{u\}) \Big\}\,.$$

Proof By Lemma 10(ii) and the definition of $\hat{f}_v$, the maximum of $f_v(Z)$ over $Z \in \bigcup_{u \in X} 2^{Y \setminus \{u\}}$ equals $\max_{u \in X} \hat{f}_v(Y \setminus \{u\})$. By Lemma 10(i), the larger of $\max_{X \subseteq Z \subseteq Y} f_v(Z)$ and $\max_{u \in X} \hat{f}_v(Y \setminus \{u\})$ equals $\max_{V \subseteq Y} f_v(V)$, which by definition is $\hat{f}_v(Y)$. ■

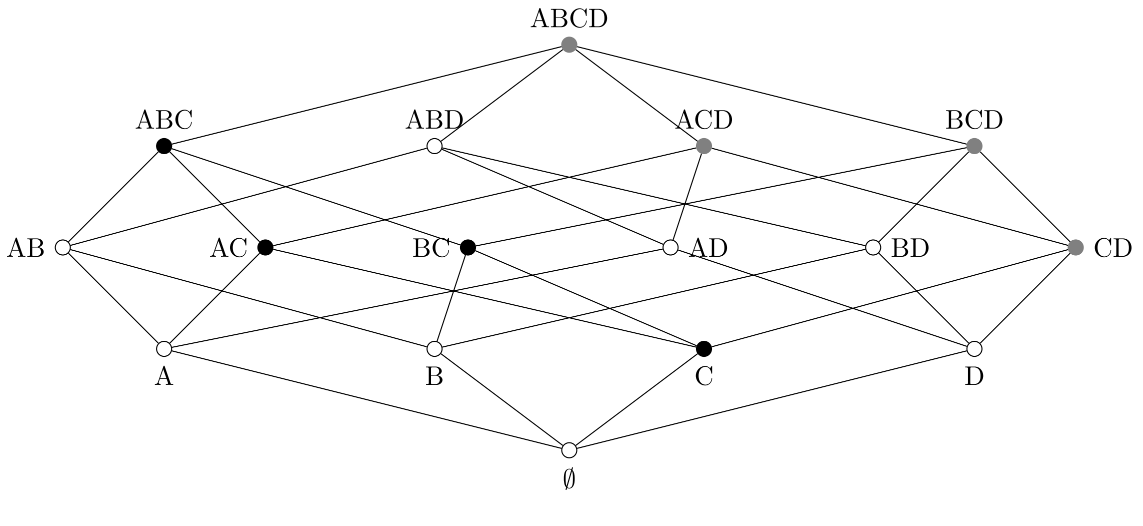

Figure 2: Tails of the ideals $\{\mathrm{A},\mathrm{B},\mathrm{C}\}$ and $\{\mathrm{A},\mathrm{B},\mathrm{C},\mathrm{D}\}$ of the partial order of Figure 1(b), marked in black and gray, respectively, on the subset lattice of $\{\mathrm{A},\mathrm{B},\mathrm{C},\mathrm{D}\}$. An interval from one set to another consists of the sets that are along a shortest path between the two sets in a Hasse diagram of the lattice. For example, the interval $\big[\{\mathrm{C}\},\{\mathrm{A},\mathrm{B},\mathrm{C}\}\big]$ consists of the sets $\{\mathrm{C}\}$, $\{\mathrm{A},\mathrm{C}\}$, $\{\mathrm{B},\mathrm{C}\}$, and $\{\mathrm{A},\mathrm{B},\mathrm{C}\}$. In the figure, each subset is referred to by a sequence of its elements.

Lemma 11 leaves us the freedom to choose a suitable node subset X for each set of interest Y. For this choice, we make use of the fact that in the second phase of dynamic programming, as it will turn out in the next subsection, we need the values $\hat{f}_v(Y)$ only for sets Y that are ideals of P. Thus, our goal is to choose X such that $Y \setminus \{u\} \in \mathcal{I}(P)$ for all $u \in X$. To this end, we let X consist of all such nodes in Y that have no larger node in Y (w.r.t. P). Accordingly, for $Y \in \mathcal{I}(P)$ define

$$\check{Y} = \{u \in Y : uv \notin P \text{ for all } v \in Y \setminus \{u\}\}\,.$$

Furthermore, define the *tail* of Y as the interval

$$\mathcal{T}_Y = [\check{Y}, Y]\,;$$

see Figure 2 for an illustration. By letting $X = \check{Y}$ and noting that $f_v(Z) = -\infty$ for $Z \notin \mathcal{F}_v$, we may rephrase the equation in Lemma 11 as

$$\hat{f}_v(Y) = \max\Big\{ \max_{Z \in \mathcal{T}_Y \cap \mathcal{F}_v} f_v(Z),\, \max_{u \in \check{Y}} \hat{f}_v(Y \setminus \{u\}) \Big\}\,. \tag{4}$$

The next two lemmas show that $\check{Y}$ indeed has the desired property (in a maximal sense) and that the tails of different ideals Y are pairwise disjoint and thus optimally cover the subsets of N.

Lemma 12 *Let $Y \in \mathcal{I}(P)$ and $u \in Y$. Then $Y \setminus \{u\} \in \mathcal{I}(P)$ if and only if $u \in \check{Y}$.*

Proof "If": Let $u \in \check{Y}$. Let $st \in P$. By the definition of $\mathcal{I}(P)$ we need to show that $t \in Y \setminus \{u\}$ implies $s \in Y \setminus \{u\}$. So, suppose $t \in Y \setminus \{u\}$, hence $t \in Y$. Now, since $Y \in \mathcal{I}(P)$, we must have $s \in Y$. It remains to show that $s \neq u$. But this holds because $ut \notin P$ by the definition of $\check{Y}$.

"Only if": Let $u \notin \check{Y}$. Then we have $uv \in P$ for some $v \in Y \setminus \{u\}$. But $u \notin Y \setminus \{u\}$ and $v \in Y \setminus \{u\}$, implying $Y \setminus \{u\} \notin \mathcal{I}(P)$ by the definition of $\mathcal{I}(P)$. ■

Lemma 13 *Let Y and Y' be distinct ideals of P. Then the tails of Y and Y' are disjoint.*

Proof The lemma states that there does not exist any nonempty Z such that $Z \in \mathcal{T}_Y \cap \mathcal{T}_{Y'}$. Suppose the contrary that $Z \in \mathcal{T}_Y \cap \mathcal{T}_{Y'}$. By symmetry we may assume that $Y \setminus Y'$ contains an element w. Thus $w \notin Z$, because $Z \subseteq Y'$. Because $\check{Y} \subseteq Z$, we have $w \notin \check{Y}$. By the definition of $\check{Y}$ we conclude that for every $u \in Y \setminus \check{Y}$ there exists $v \in \check{Y}$ such that $uv \in P$. Therefore, in particular there exists $v \in \check{Y}$ such that $wv \in P$. Since $w \notin Y'$ and Y' is an ideal of P it follows by definition of an ideal that $v \notin Y'$. On the other hand, $v \in \check{Y}$ and $\check{Y} \subseteq Z \subseteq Y'$ implies that $v \in Y'$: contradiction. ■

We we will use these lemmas in Section 5.4 to perform the recurrence of Lemma 11 over the ideals of the partial order and their tails.

5.3 Dynamic Programming: Second Phase

Recall that the second phase of the basic dynamic programming algorithm is captured by the recurrence (1), which concerns all subsets of N that can begin some linear order on N, that is, all the 2^n subsets. Now that we restrict our attention to DAGs that are compatible with the given partial order P, we may restrict the recurrence to only those sets that can begin some linear extension of P, that is, to the ideals of P. Formally, define the function g^P by $g^P(\emptyset) = 0$ and for nonempty $Y \in \mathcal{I}(P)$ recursively:

$$g^P(Y) = \max_{\substack{v \in Y \\ Y \setminus \{v\} \in \mathcal{I}(P)}} \left\{ g^P(Y \setminus \{v\}) + \hat{f}_v(Y \setminus \{v\}) \right\}. \tag{5}$$

We can show that $g^P(N)$ equals the maximum score over the DAGs compatible with P:

Lemma 14 *Let P be a partial order on N. Then*

$$g^P(N) = \max\{f(A) : A \text{ is compatible with } P\}.$$

Furthermore, if $\mathcal{P}$ is a cover on N, then

$$\max_{P \in \mathcal{P}} g^P(N) = \max_A f(A),$$

where A runs through all DAGs on N.

Proof Let P be a partial order on N. For any subset $Y \subseteq N$ denote by $P[Y]$ the induced partial order $\{xy \in P : x, y \in Y\}$. We show by induction on the size of Y that $g^P(Y)$ equals the maximum score $f(A')$ over the DAGs $A' \subseteq Y \times Y$ compatible with $P[Y]$, assuming Y is an ideal of P. Here the score $f(A')$ is naturally defined as $\sum_{v \in Y} f_v(A'_v)$.

For the base case, consider an arbitrary singleton $\{v\} \in \mathcal{I}(P)$. Clearly, there is exactly one DAG A' on $\{v\}$ and it is compatible with $P[\{v\}] = \{vv\}$; the score of the DAG is $f(A') = f_v(\emptyset) = \hat{f}_v(\emptyset)$. This is precisely what the recurrence (5) gives, as $g^P(\emptyset) = 0$.

Suppose then that the recurrence (5) holds for all proper subsets of a subset $Y \subseteq N$. Without any loss of generality, we assume $Y = N$ for notational convenience. Now, write

$$\begin{aligned}
\max_{A \text{ is compatible with } P} f(A) &= \max_{L \supseteq P} \max_{A \subseteq L} \sum_{v \in N} f_v(A_v) \\
&= \max_{L \supseteq P} \sum_{v \in N} \max_{A_v \subseteq L_v} f_v(A_v) \\
&= \max_{L \supseteq P} \sum_{v \in N} \hat{f}_v(L_v) \\
&= \max_{v \in N} \Big\{ \hat{f}_v(N \setminus \{v\}) + \max_{L' \supseteq P[N \setminus \{v\}]} \sum_{u \in N \setminus \{v\}} \hat{f}_u(L'_u) \Big\} \\
&= \max_{v \in N} \Big\{ \hat{f}_v(N \setminus \{v\}) + g^{P[N \setminus \{v\}]}(N \setminus \{v\}) \Big\} .
\end{aligned}$$

Here L and L' run trough the respective linear extensions and A through the DAGs satisfying the mentioned condition. The first identity holds by the definition of compatibility and the decomposability of the scoring function f; the second one by the distributive law; the third one by the definition of $\hat{f}_v$; the fourth one because addition distributes over maximization and the fact that some node v is the last one in the linear order L and that the induced linear order L' on the remaining nodes is an extension of the induced partial order $P[N \setminus \{v\}]$; the fifth one by the induction assumption (and the third identity). Finally, it suffices to notice that $g^P(Y) = g^{P[Y]}(Y)$ for $Y \in \mathcal{I}(P)$, since clearly a subset X of Y is an ideal of P if and only if X is an ideal of $P[Y]$.

For the second statement, it suffices to observe that

$$\max_{P \in \mathcal{P}} \max_{L \supseteq P} \sum_{v \in N} \hat{f}_v(L_v) = \max_{L} \sum_{v \in N} \hat{f}_v(L_v) = \max_{A} f(A) ,$$

since $\mathcal{P}$ is a cover on N. ■

5.4 Dynamic Programming: First and Second Phase Merged

We now merge the ingredients given in the previous two subsections into an algorithm for evaluating g^P using the recurrence (5) and Lemma 11, for a fixed $P \in \mathcal{P}$. In Algorithm 1 below, $g^P[Y]$ and $\hat{f}_v[Y]$ denote program variables that correspond to the respective target values $g^P(Y)$ and $\hat{f}_v(Y)$ to be computed. Also, recall that $\mathcal{F}_v$ denotes the collection of potential parent sets for node v.

Algorithm 1:

```
1. Let $g^P[\emptyset] \leftarrow 0$.
2. For each $v \in N$, let $\hat{f}_v[\emptyset] \leftarrow f_v(\emptyset)$.
3. For each nonempty $Y \in \mathcal{I}(P)$, in increasing order of cardinality:
    (a) let
        $$g^P[Y] \leftarrow \max_{v \in \check{Y}} \Big\{ g^P[Y \setminus \{v\}] + \hat{f}_v[Y \setminus \{v\}] \Big\};$$
    (b) for each $v \in Y$, let $\hat{f}_v[Y]$ be the larger of
        $$\max_{Z \in \mathcal{T}_Y \cap \mathcal{F}_v} f_v(Z) \text{ and } \max_{u \in \check{Y}} \hat{f}_v[Y \setminus \{u\}].$$
```

Lemma 15 *Algorithm 1 correctly computes g^P, that is, $g^P[Y] = g^P(Y)$ for all $Y \in \mathcal{I}(P)$.*

Proof Observe first that the algorithm correctly computes $\hat{f}_v$ for each $v \in N$, that is, after the execution of the algorithm we have $\hat{f}_v[Y] = \hat{f}_v(Y)$ for all $v \in Y \in \mathcal{I}(P)$. To this end, it suffices to note that step 3(b) implements the recurrence of Lemma 11 as rephrased in (4).

Then notice that step 3(a) implements the recurrence (5). Indeed, by Lemma 12, the condition "$v \in Y$ and $Y \setminus \{v\} \in \mathcal{I}(P)$" of (5) is equivalent to "$v \in \check{Y}$" of step 3(a), given that $Y \in \mathcal{I}(P)$ (guaranteed in step 3). This completes the proof. ■

To solve the OBN problem it suffices to run Algorithm 1 for every partial order in a system that is a cover on N. The appropriate partial order system of course varies with the problem instance, particularly with the number of nodes n. In the following statement of the time and space complexity of OBN, we do not fix any particular way to choose and construct the needed partial order system, but we simply refer to any appropriate system. Thus, the complexity results are nonuniform with this respect.

Theorem 16 (main) *OBN can be solved in $O([\sum_{P \in \mathcal{P}}(|\mathcal{I}(P)| + F)]n^2)$ time and $O([\max_{P \in \mathcal{P}} |\mathcal{I}(P)| + Fn]n)$ space, assuming $\mathcal{P}$ is a cover of N and each $\mathcal{F}_v$ is downward closed and of size at most F.*

Proof By Lemmas 14 and 15, it suffices to run Algorithm 1 for each $P \in \mathcal{P}$.

The time requirement of Algorithm 1 is dominated by steps 3(a) and 3(b). Given Y, the set $\check{Y}$ can be constructed in time $O(n^2)$ by removing from Y each element that is not maximal in Y. Thus, the contribution of step 3(a) in the total time requirement is $O(|\mathcal{I}(P)|n^2)$.

We then analyze the time requirement of step 3(b), for fixed v. By Proposition 2, the maximization of the local scores over $\mathcal{T}_Y \cap \mathcal{F}_v$ can be done in $O(|\mathcal{T}_Y \cap \mathcal{F}_v|n)$ time. By Lemma 13 the collections $\mathcal{T}_Y \cap \mathcal{F}_v$ are disjoint for different $Y \in \mathcal{I}(P)$. Thus the total contribution to the time requirement is proportional to $|\mathcal{F}_v| \leq F$, for each v. Because step 3(b) is executed $|\mathcal{I}(P)|$ times, the total time requirement of step 3(b) is $O(|\mathcal{I}(P)|n^2 + Fn^2)$. Combining the time bounds of steps 3(a) and 3(b) and summing over all members of $\mathcal{P}$ yields the claimed bound $O([\sum_{P \in \mathcal{P}} |\mathcal{I}(P)| + F]n^2)$.

The space requirement for a fixed partial order P is $O(|\mathcal{I}(P)|n)$, since by Lemma 12 the values $g^P[Y]$ and $\hat{f}_v[Y]$ are only needed for $Y \in \mathcal{I}(P)$. In addition, storing the local scores requires $O(Fn)$ space for each node v. Therefore, the total space requirement is $O\big(\big[\max_{P\in\mathcal{P}} |\mathcal{I}(P)| + Fn\big]n\big)$. ■

Remark 17 Like in the basic DP algorithm, step 3 of Algorithm 1 can be implemented so that, compared to the bound in Theorem 16, a space saving proportional to $\sqrt{n}$ is obtained.

5.5 Notes

We end this section with a couple of remarks on the general partial order approach.

The results in this section apply to any POS that is a cover on the node set. However, they do not tell us how to choose a POS that yields, in some sense, the most efficient tradeoff between space and time. Ideally, we would like to have a scheme that given a space bound as a function of the number of nodes n, say s_n, gives us a POS $\mathcal{P}_n$ such that the space requirement implied by $\mathcal{P}_n$ is $O(s_n)$ while the time requirement bound is minimized. Unfortunately, designing such an optimal scheme seems difficult due to the combinatorial challenge of simultaneously controlling the required covering property of the partial order system and the number of ideals of the members of the system. Note that both counting the linear extensions and the ideals of a given partial order are #P-hard computational problems (Brightwell and Winkler, 1991; Provan and Ball, 1983), suggesting that their mathematical analysis is also not easy. In the next section we give a partial and practical solution to this issue by studying a subclass of series-parallel partial orders, which enables a derivation of concrete, quantitative space-time tradeoff results.

As another remark, we note that the algorithms can be easily parallelized onto $|\mathcal{P}|$ processors, each with its own memory, with negligible communication costs. This is in sharp contrast with the basic dynamic programming algorithm that does not enable such large-scale parallelization. We discuss the recent work by Tamada et al. (2011) in this light in Section 7. The ease of the parallelization stems from the fact that dynamic programming over the ideals can be done independently for each partial order P in $\mathcal{P}$. More precisely, each processor gets a dedicated partial order P (and the local scores) as input and outputs the score $g^P(N)$ and a respective DAG. It then remains to communicate these outputs to a central unit that chooses an optimal one among them.

6. Bucket Order Schemes

To apply the partial order approach in practice, it is essential to find partial order systems that provide a good tradeoff between time and space requirements. This section concentrates on a class of partial orders we call *parallel bucket orders*, which turn out to be relatively easy to analyze and seem to yield good tradeoff in practice. They also subsume, for instance, the two-bucket construction of Section 3.

6.1 Bucket Orders and Reorderings

Let P and Q be partial orders on disjoint ground sets M and N, respectively. We say that a partial order R is a *series composition* of P and Q if $R = P \cup Q \cup \{xy : x \in M, y \in N\}$, and a *parallel composition* of P and Q if simply $R = P \cup Q$. Note that both compositions always yield a partial order, since the ground sets are disjoint. As both composition operations are associative, they readily extend to any finite number of partial orders. Because the series composition operation is not commutative, it is, of course, applied to a sequence of partial orders. A *series-parallel* partial order is defined recursively: a partial order is a series-parallel partial order if it is either a singleton order or a parallel or series composition of two or more series-parallel partial orders. For example, the partial order $\{\mathrm{AA}, \mathrm{BB}, \mathrm{CC}, \mathrm{AB}, \mathrm{AC}\}$ is a series composition of $\{\mathrm{AA}\}$ and $\{\mathrm{BB}, \mathrm{CC}\}$, of which the former is a singleton order and the latter is a parallel composition of two singleton orders. Note that any trivial order is a parallel composition of singleton orders.

We study two special classes of series-parallel partial orders, namely bucket orders and parallel bucket orders. A *bucket order* is a series composition of the trivial orders on some ground sets $B_1, B_2, \ldots, B_\ell$, called *buckets*. The bucket order is said to be of *length* ℓ and *type* $|B_1| * |B_2| * \cdots * |B_\ell|$. We may denote the bucket order by $B_1 B_2 \cdots B_\ell$. For instance, the series-parallel partial order in the previous paragraph is a bucket order $\{\mathrm{A}\}\{\mathrm{B}, \mathrm{C}\}$, thus of length two and type $1 * 2$. Likewise, the partial order in Figure 1(b) is a bucket order of length three and type $2 * 3 * 1$, the trivial order on some ground set M is of length one and type $|M|$, and a linear order on M is of length $|M|$ and type $1 * 1 * \cdots * 1$.

By taking a parallel composition of some number of bucket orders we obtain a *parallel bucket order*. Note that the same parallel bucket order may be obtained by different collections of bucket orders. It is, however, easy to observe that for each parallel bucket order P there is a unique collection of bucket orders $P_1, P_2, \ldots, P_p$, called the bucket orders of P, such that their parallel composition is P and they are the connected components of P.

The following lemma states a well-known result concerning the number of ideals of a series-parallel partial order (see, for example, Steiner's article, 1990, and references therein).

Lemma 18 *Let P_1 and P_2 be partial orders on disjoint ground sets. Then (i) the series composition of P_1 and P_2 has $|\mathcal{I}(P_1)| + |\mathcal{I}(P_2)| - 1$ ideals and (ii) the parallel composition of P_1 and P_2 has $|\mathcal{I}(P_1)||\mathcal{I}(P_2)|$ ideals.*

The next two lemmas state the number of ideals of a parallel bucket order.

Lemma 19 *Let B be a bucket order $B_1 B_2 \ldots B_\ell$. Then the number of ideals of B is given by $|\mathcal{I}(B)| = 1 - \ell + 2^{|B_1|} + 2^{|B_2|} + \cdots + 2^{|B_\ell|}$.*

Proof Any singleton order has two ideals, namely the empty set and the ground set. Thus, by Lemma 18(ii), the trivial order on the bucket B_i has $2^{|B_i|}$ ideals, for each i. Hence, by Lemma 18(i), B has $1 - \ell + 2^{|B_1|} + 2^{|B_2|} + \cdots + 2^{|B_\ell|}$ ideals. ■

We note that the order of buckets does not affect the number of ideals.

Lemma 20 *Let P be the parallel composition of bucket orders $P_1, P_2, \ldots, P_p$. Then the number of ideals of P is given by $|\mathcal{I}(P)| = |\mathcal{I}(P_1)||\mathcal{I}(P_2)| \cdots |\mathcal{I}(P_p)|$.*

Proof Follows immediately from Lemma 18(ii). ■

Let us return to the issue of choosing a POS that is a cover of the node sets and yields a good space-time tradeoff. To this end, we will consider systems of parallel bucket orders of a certain kind, namely systems obtained via "reordering" a fixed parallel bucket order:

Definition 21 (reordering) We say two bucket orders are *reorderings* of each other if they have the same ground set and they are of the same type. Furthermore, we say two parallel bucket orders are *reorderings* of each other if their bucket orders can be labelled as $P_1, P_2, \ldots, P_p$ and $Q_1, Q_2, \ldots, Q_p$, respectively, such that P_i is a reordering of Q_i for each i. We denote the collection of reorderings of a parallel bucket order P by $\mathcal{R}(P)$.

We note that two bucket orders are reorderings of each other if and only if they are automorphic to each other. However, this equivalence does not hold for parallel bucket orders in general, for reordering only allows shuffling within each bucket order, but not between different bucket orders.

It can be shown that $\mathcal{R}(P)$ is a cover on the ground set of P. The proof below slightly simplifies the one we have given earlier (Koivisto and Parviainen, 2010, Theorem 3.1).

Theorem 22 *Let P be a parallel composition of bucket orders. Then $\mathcal{R}(P)$ is a cover on the ground set of P.*

Proof Let L be a linear order on M. It suffices to construct a parallel bucket order $Q \in \mathcal{R}(P)$ such that L is an extension of Q. To this end, let $P_1, P_2, \ldots, P_p$ be the bucket orders of P, with respective ground sets $M_1, M_2, \ldots, M_p$. For each $i = 1, 2, \ldots, p$, construct a bucket order Q_i on M_i as follows: Let $m_1 * m_2 \cdots * m_\ell$ be the type of P_i. Let L' be the induced order $L[M_i]$. Observe that L' is a linear order. Now, put $Q_i = C_1 C_2 \cdots C_\ell$ where C_1 consists of the first m_1 elements in the order L', C_2 consists of the next m_2 elements in the order, and so forth. Observe that L' is an extension of Q_i and that Q_i is a reordering of P_i. Finally, let Q be the parallel composition of $Q_1, Q_2, \ldots, Q_p$. Clearly, $Q \in \mathcal{R}(P)$. To complete the proof, note that L is an extension of Q, since $xy \in Q$ implies $xy \in Q_i$ for some i, whence $xy \in L[M_i] \subseteq L$. ■

When P is a parallel composition of p bucket orders, each of type $b_1 * b_2 * \cdots * b_\ell$, we find it convenient to denote the POS $\mathcal{R}(P)$ by $(b_1 * b_2 * \cdots * b_\ell)^p$. This notation is explicit about the combinatorial structure of the POS, while it ignores the arbitrariness of the labeling of the ground set. When the size of the ground set, n, is clear from the context, we may extend the notation $(b_1 * b_2 * \cdots * b_\ell)^p$ to refer to a system $\mathcal{R}(P)$, where P is a parallel composition of p bucket orders of type $b_1 * b_2 * \cdots * b_\ell$ and one trivial order on the remaining $n - p(b_1 + b_2 + \cdots + b_\ell)$ elements. It is instructive to notice that a POS $(b_1 * b_2 * \cdots * b_\ell)^p$ consists of $\binom{b_1+b_2+\cdots+b_\ell}{b_1 \quad b_2 \quad \cdots \quad b_\ell}^p$ different partial orders, since any bucket order of type $b_1 * b_2 * \cdots * b_\ell$ has exactly $\binom{b_1+b_2+\cdots+b_\ell}{b_1 \quad b_2 \quad \cdots \quad b_\ell} = \frac{(b_1+b_2+\cdots+b_\ell)!}{b_1! b_2! \cdots b_\ell!}$ different reorderings.

For an illustration of these concepts, consider a node set $N = \{\mathrm{A,B,C,D,E,F,G,H}\}$ partitioned into $N_1 = \{\mathrm{A,B,C,D,E,F}\}$ and $N_2 = \{\mathrm{G,H}\}$. Let the bucket order on N_1 be the one shown in Figure 1(b), denoted as P_1, and let the bucket order on N_2 be the trivial order $\{\mathrm{GG,HH}\}$, denoted as P_2. Now the reorderings of the parallel composition of P_1 and P_2 form a POS $(2*3*1)^1$. By Lemma 19, P_1 has $1 - 3 + 2^2 + 2^3 + 2^1 = 12$ ideals and P_2 has $1 - 1 + 2^2 = 4$ ideals. By Lemma 20, the total number of ideals of the parallel composition of P_1 and P_2 is $12 \times 4 = 48$. The number of the partial orders in $(2*3*1)^1$ is $\binom{6}{2 \quad 3 \quad 1}^1 = 60$.

6.2 Bucket Order Schemes

A partial order system $(b_1 * b_2 * \cdots * b_\ell)^p$ is associated with some natural *parameters*, such as the number of parallel bucket orders p. When all or some of the parameters are treated as variables that can take different values, we refer to the implied family of partial order systems as a *bucket order scheme*. Furthermore, we denote the scheme simply by the expression $(b_1 * b_2 * \cdots * b_\ell)^p$, where the parameters and their ranges are assumed to understood from the context. For instance, we may talk about the scheme $(5*5)^p$, understanding that p takes values over its natural range, that is, $p = 1, 2, \ldots, \lfloor n/(5+5) \rfloor$. Another example of a bucket order scheme is the *two-bucket scheme* of Section 3, which corresponds to partial order systems $(s*(n-s))^1$, with s treated as the parameter. Other examples are the *pairwise scheme*, corresponding to systems $(1*1)^p$, with p as the parameter, and the *generalized two-bucket scheme* defined by the systems $(\lceil m/2 \rceil * \lfloor m/2 \rfloor)^{\lfloor n/m \rfloor}$, with m as the parameter.

Via Theorem 16, any fixed bucket order scheme implies parameterized time and space complexity bounds for the OBN problem. Moreover, one scheme may dominate another scheme in the sense of yielding an equal or smaller time bound at any space bound. While it is currently an open problem to characterize bucket order schemes that dominate all other bucket order schemes, our analysis of the so-called space-time product (Koivisto and Parviainen, 2010) suggests that the most efficient trade-off is achieved with the bucket order scheme $(\lceil m/2 \rceil, \lfloor m/2 \rfloor)^p$. The scheme guarantees that the product of the time and space requirements scales roughly as C^n, where C equals 4 or is slightly below 4 depending on the space bound. If some other scheme resulted in a significantly faster algorithm at any space bound, then that scheme would make a new record also in terms of the space-time product, albeit possibly at a single point. Our prior study (Koivisto and Parviainen, 2010) shows that no such scheme exists among bucket order schemes. When measured by the space-time product, the tradeoff of the scheme $(\lceil m/2 \rceil, \lfloor m/2 \rfloor)^p$ slowly improves when m increases, until $m = 26$, after which the tradeoff starts slowly getting worse. We next examine this scheme in more detail focusing on the range $2 \leq m \leq 26$. See Figure 4 (in Section 7) for the tradeoff curve.

6.3 Practical Bucket Order Schemes

A partial order system $(\lceil m/2 \rceil * \lfloor m/2 \rfloor)^p$ consists of $\binom{m}{\lfloor m/2 \rfloor}^p$ partial orders, each having $2^{n-mp}(2^{\lfloor m/2 \rfloor} + 2^{\lceil m/2 \rceil} - 1)^p$ ideals. Plugging these numbers into Theorem 16 gives us the following time and space bounds.

Corollary 23 *OBN can be solved in $O\big([\binom{m}{\lfloor m/2 \rfloor}^p(I+F)]n^2\big)$ time and $O\big([I+Fn]n\big)$ space, where $I = 2^{n-mp}(2^{\lfloor m/2 \rfloor} + 2^{\lceil m/2 \rceil} - 1)^p$, for any $m = 2, \ldots, n$ and $p = 0, \ldots, \lfloor n/m \rfloor$, assuming each $\mathcal{F}_v$ is downward closed and of size at most F.*

We observe that the number of ideals, I, dominates both the time and space requirements as long as the input size, roughly F, is not too large. Note that pruning the parent sets affects F but has no effect on I. Since in our case, I usually grows exponentially in n, exponentially large families of potential parent sets can be handled with negligible extra cost. To investigate this issue more carefully, we focus on the case where each node is allowed to have at most $k = \alpha n$ parents, with some *slope* $\alpha \leq 1/2$. How large can the slope α be, yet guaranteeing that the size of the input does not dominate the time and space requirements? Recall that now the term Fn in the space bound can be replaced by F.

Next we present some lower bounds for α. We note that the largest such slope varies depending on the scheme used. For a moment, let us focus on the $(\lfloor m/2 \rfloor * \lceil m/2 \rceil)^{\lfloor n/m \rfloor}$ scheme, and to simplify calculations, assume m is even and n divisible by m. For any fixed m, we bound the largest slope by α_m, as follows. It is well-known that $\sum_{i=0}^{\alpha_m n}\binom{n}{i}$, an upper bound for the number of potential parent sets per node, is at most $2^{H(\alpha_m)n}$, where H is the binary entropy function (for a proof, see for example Flum and Grohe (2006, p. 427)). On the other hand, every partial order in the system $(m/2 * m/2)^{n/m}$ has $((2^{m/2+1}-1)^{1/m})^n$ ideals. Thus, the number of ideals dominates the space and time requirements if $2^{H(\alpha_m)n} \leq ((2^{m/2+1}-1)^{1/m})^n$, equivalently, $H(\alpha_m) \leq 1/m \log_2(2^{m/2+1}-1)$. Solving this inequality numerically gives us a bound α_m. Table 1 shows the α_m for each even $m \leq 26$.

m	α_m	m	α_m	m	α_m
2	0.238	12	0.139	20	0.127
4	0.190	14	0.135	22	0.125
6	0.167	16	0.131	24	0.124
8	0.153	18	0.129	26	0.123
10	0.145				

Table 1: Bounds on the maximum indegree slopes for the scheme $(m/2 * m/2)^{n/m}$.

Since the partial orders in $(m/2 * m/2)^{n/m}$ have as many or fewer ideals than the partial orders in $(m/2 * m/2)^p$, for $p \leq n/m$, we have the following characterization.

Corollary 24 *OBN can be solved in* $O(((\binom{m}{m/2})2^{n-mp}(2^{m/2+1}-1))^p n^2)$ *time and* $O(2^{n-mp}(2^{m/2+1}-1)^p n)$ *space for any* $p = 0, 1, 2, \ldots, \lfloor n/m \rfloor$ *and* $m = 2, 4, 6, \ldots 26$, *provided that each node has at most* $\alpha_m n$ *parents, with* α_m *as given in Table 1.*

For example, the maximum indegree in 30-node DAGs can be set to $\lfloor 0.238 \times 30 \rfloor = 7$ in the pairwise scheme $(1 * 1)^p$ and to $\lfloor 0.139 \times 30 \rfloor = 4$ in the scheme $(6 * 6)^p$. A larger maximum indegree may render the input size dominate the number of ideals in the time and space requirements. In the next subsection we provide some empirical results, which suggest that the bounds in Table 1 are only slightly conservative.

6.4 Empirical Results

We have implemented the presented algorithm in the C++ language into a publicly available computer program BOSON (Bucket Order Scheme for Optimal Networks).[4] As the implementation is not fully optimized, the empirical results reported below should be viewed rather as a proof of concept.[5] We tested our implementation varying the number of nodes n, bucket order sizes m, and the number of parallel bucket orders p. The experiments were run on Intel Xeon R5540 processors, each with 32 GB of RAM.

We examined the running time for the limit of 16 GB of memory, letting the number of nodes n vary from 25 to 34, with maximum indegree set to 3. The local scores were taken as given, so

4. BOSON is available at `www.csc.kth.se/~pekkapa/code/boson-1.0.tar.gz`.
5. We found the Silander-Myllymäki implementation (Silander and Myllymäki, 2006) about five times faster when the algorithms were run on the same setting, that is, running our algorithm on the trivial order on the nodes. It is likely that also the space usage can be lowered by a similar small factor by implementing Remark 17. Taken together, such improvements would allow us to deal with networks with two to three additional nodes compared to the present implementation, with the same time and space resources.

computing them is not included in the running time estimates. Since we do not prune potential parent sets based on the local scores, we have them for all possible parent sets of size at most the maximum indegree. Note that the actual scores are irrelevant when measuring the time and space usage. First we estimated the smallest bucket order size m that yields a memory requirement of 16 GB or less. Then we ran Algorithm 1 for a partial order from the POS $(\lceil m/2 \rceil, \lfloor m/2 \rfloor)^1$ and gauged the running time. Finally, the measured running time was multiplied by the cardinality of the POS to get an estimate of the total running time. As all partial orders in the POS yield identical time and space requirements, there is no issue with estimation error or variance. Table 2 shows the results. We observe that, as expected, the time requirement grows rapidly with n: An optimal 25-node DAG can be found in about 25 minutes, while a 30-node DAG requires over 16 days of CPU time. Finding an optimal 34-node DAG is feasible using large-scale parallelization: with 1000 processors it takes about 6 days.

n	p	m	Time per PO	Cover size	Total time
25	0	0	0.42	1	0.42
26	1	3	0.44	3	1.34
27	1	5	0.49	10	4.9
28	1	8	0.35	70	24.8
29	1	10	0.39	252	97.7
30	1	12	0.43	924	394
31	1	14	0.49	3432	1671
32	1	16	0.53	12 870	6784
33	1	18	0.83	48 620	40 332
34	1	20	0.78	184 756	144 930

Table 2: Time requirements for finding optimal Bayesian networks, for varying number of nodes when space is limited to 16 GB. Columns: n is the number of nodes; p is the number of parallel bucket orders; m is the size of the balanced bucket order; *time per PO* is the running time (in CPU hours) per partial order; *cover size* is the number of partial orders in the cover; *total time* is the total running times (in CPU hours), that is, time per partial order multiplied by the size of the cover.

Next we studied how different schemes affect the running times and the space usage in practice. We analyzed two specializations of the generalized bucket order scheme $(\lceil m/2 \rceil * \lfloor m/2 \rfloor)^p$: the *practical scheme*, where $p = 1$, and the pairwise scheme, where $m = 2$. The results are shown in Tables 3 and 4. As expected, the practical scheme yields a clearly better space-time tradeoff than the pairwise scheme. This is perhaps even more clearly pronounced in Figure 3, which shows the empirical tradeoffs in the space-time plane along with the analytical bounds (Corollary 23). We see that, in general, the empirical and analytical bounds are in a good agreement, the analytical bounds being slightly conservative for medium m (the practical scheme) and medium p (the pairwise scheme).

We also investigated the influence of the maximum indegree k on various characteristics of our implementation. Note that, in effect, k parameterizes the number of potential parent sets, which in turn determines the computational complexity. Thus the observations readily extend to the case when the number of potential parent sets is reduced using an appropriate pruning proce-

m	p	Space	Total time	m	p	Space	Total time
0	0	21248	1.01	14	1	331	32.51
2	1	15936	1.13	15	1	248	43.97
3	1	13280	1.41	16	1	166	64.35
4	1	9296	1.75	17	1	124	87.79
5	1	7304	2.13	18	1	83	129.65
6	1	4980	2.80	19	1	62	171.93
7	1	3818	3.57	20	1	41	256.61
8	1	2573	4.96	21	1	31	342.92
9	1	1950	6.88	22	1	21	489.88
10	1	1307	9.06	23	1	16	638.48
11	1	986	12.15	24	1	10	976.50
12	1	659	17.43	25	1	8	1444.53
13	1	495	24.31	26	1	5	2600.15

Table 3: Running times (in CPU hours) and space usage (in MB) of the practical scheme $(\lceil m/2 \rceil * \lfloor m/2 \rfloor)^1$ with $2 \leq m \leq 26$ and $n = 26$.

m	p	Space	Total time	m	p	Space	Total time
0	0	21248	1.01	2	7	2836	12.79
2	1	15936	1.13	2	8	2127	20.25
2	2	11952	1.48	2	9	1595	32.36
2	3	8964	2.09	2	10	1197	51.66
2	4	6723	3.19	2	11	897	78.96
2	5	5042	4.97	2	12	673	126.52
2	6	3782	7.90	2	13	505	198.88

Table 4: Running times (in CPU hours) and space usage (in MB) of the pairwise scheme $(1 * 1)^p$ with $1 \leq p \leq 13$ and $n = 26$.

dure (de Campos and Ji, 2011). We analyzed the scheme $(10 * 10)^1$ with $n = 20$ nodes, varying k from 1 to 8. For interpretation of the results, shown in Table 5, it is useful to note that the partial orders in question have 2047 ideals. For comparison, the (worst-case) input size is 1160 for $k = 3$ and 5036 for $k = 4$. So we conclude that the number of ideals dominates the input size precisely when $k \leq 3$. Now, recall that the bounds in Corollary 24 guarantee this only for $k \leq 2$, indicating that the analytical bounds are not tight but slightly pessimistic. Table 5 shows also, perhaps somewhat surprisingly, that even if the input size for $k = 5$ is more than 3 times the input size for $k = 4$, the total running time less than doubles. This can be explained by the fact that the respective increase in the tail accesses, from 46520 to 160260, does not yet pay off, since the number of computation steps that are not related to the input (nor the maximum indegree) appears to be as large as 358300. For larger k, the running time will grow about linearly with the number of tail accesses.

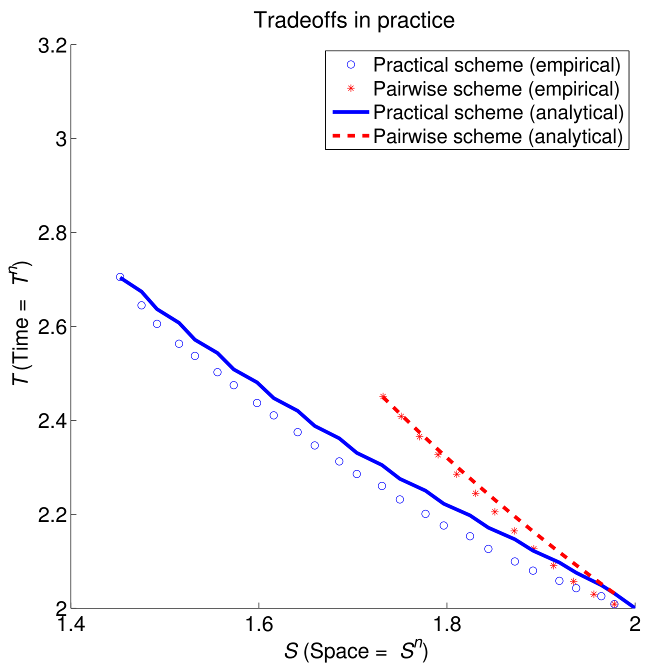

Figure 3: Empirical and analytical time and space requirements of the pairwise scheme and the practical scheme, for $n = 26$ nodes. The empirical time and space requirements were normalized by dividing them, respectively, by the empirical time and space requirement of the basic dynamic programming algorithm (the case of the trivial order). We took the 26th root of the ratio, multiplied by 2, as the normalized value. The analytical bounds were normalized analogously.

7. Discussion

While this work is largely theoretical, it is mainly motivated by practical needs for solving larger problem instances to guaranteed optimum. The presented methods address those needs by two means. First, the algorithms truly decrease the space requirement of existing dynamic programming algorithms, which makes it possible to process larger instances on a typical modern computer with some limited amount of RAM, say 16 GB. Second, the algorithms can be easily and efficiently implemented to run in parallel on practically as many processors as available. As large computer clusters and grid computing with thousands of processing units are becoming more common, we believe the presented schemes bring larger networks to within the reach of exact algorithms.

Figure 4 summarizes the relationship of the time and the space requirement of different schemes presented in this work. Compared to the naive two-bucket scheme, the more general bucket order schemes yield significantly more efficient exchange of time and space resources. Our most efficient schemes enable saving space by practically any factor, specified by the user, at the cost of increasing the runtime by about the same factor. It should be noted, though, that none of the schemes enables saving space without an increase in the runtime. Whether more efficient schemes exist or whether considerable space savings are possible even at no increase in the runtime, is a difficult open ques-

k	Input size	Regular accesses	Tail accesses	Total time
1	20	210	90	2.4
2	191	1020	1350	2.5
3	1160	3060	9840	2.8
4	5036	6420	46500	4.3
5	16664	10200	160260	8.5
6	43796	13140	429480	18.3
7	94184	14700	932160	35.5
8	169766	15240	1687530	60.4

Table 5: Characteristics of the practical scheme $(10*10)^1$ with $n = 20$ nodes, for varying maximum indegree k. Columns: *input size* is the number of parent sets per node; *regular accesses* is the number of accesses to input when the parent set is an ideal; *tail accesses* is the number of accesses to input when the parent set is in the tail of an ideal; *total time* is the running time in hours. Regular and tail accesses are per partial order.

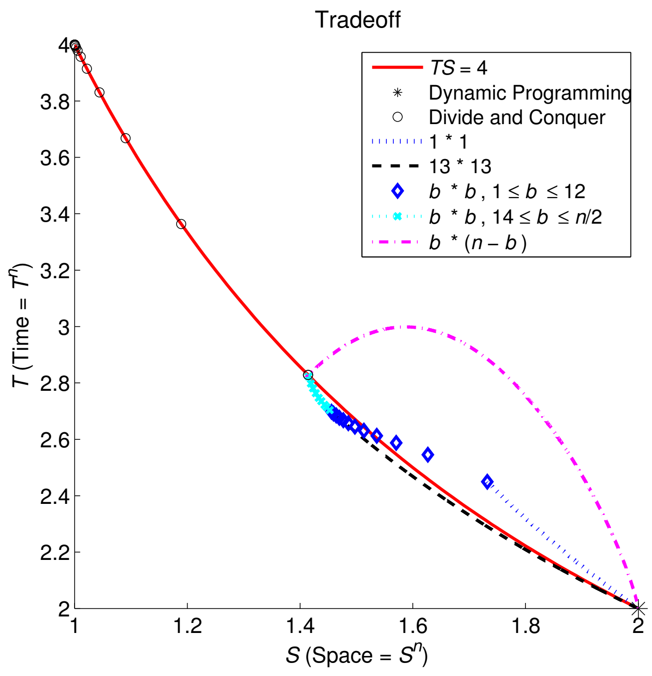

Figure 4: Comparison of time and space requirements of different schemes. The base of the exponential time and space requirement is shown on the vertical and horizontal axis, respectively.

tion. The difficulty is suggested by the fact that, no more efficient schemes are known for the classic traveling salesman problem, a simpler cousin of the BN problem studied here.

We presented the results under the assumption that the local scores are given as input and stored in the memory. While this approach sometimes yields an unnecessary large lower bound for the space requirement, we find the assumption both convenient and most relevant for practical implementations. Namely, the number of potential parent sets, particularly after pruning, is typically relatively small compared to the amount of memory available—the bottleneck regarding memory consumption is the number of ideals of the partial orders. From a theoretical point of view, it is, however, interesting to also consider the alternative approach, in which the local scores are always computed anew from given data (in time polynomial in n) when needed in the calculations. For example, this approach combined with the divide and conquer scheme yields an algorithm that takes only polynomial space, albeit the runtime is within a polynomial factor to 8^n (with no restrictions on the number of potential parent sets, cf. Corollary 6). It is an intriguing open question whether much faster polynomial-space algorithms exist.

Recently, also other approaches to save space and enhance parallelization have been proposed. Specifically, Malone et al. (2011) and Yuan et al. (2011) present a tighter implementation of the basic DP algorithm and demonstrate moderate space savings (proportional to the squareroot of the number of nodes). Compared to the basic DP algorithm, it enables finding optimal BNs with a couple of more nodes. Tamada et al. (2011) present yet a different approach. They divide the work of the DP algorithm into several subtasks that overlap only little. The subtasks can be solved in parallel, which enables efficient parallelization with a tolerable communication overhead. Compared to our approach, their algorithm has two notable drawbacks, however: First, it has to be run on a supercomputer that enables fast (and synchronized) communication between the processing units, whereas in our approach communication is not an issue. Second, the total amount of memory needed remains large: a 32-node instance was solved using over 800 GB of memory in total (with 256 CPU cores in less than six days) (Tamada et al., 2011).

Whether typical instances met in practice can be solved significantly faster than the present worst-case bounds would suggest, is one of the most central open questions in the research area. There is some hope for an affirmative answer. For an extreme example, consider an instance for which the empty DAG (that is, no arcs) happens to be optimal. That particular instance can be solved by just letting each node take its best-scoring parent set, namely the empty set, with no worries about the acyclicity constraint. This can be done relatively fast if the number of potential parent sets is small, for instance, due to an assumed bound on the number of parents per node. Quite recently, this observation has been taken further using branch-and-bound (de Campos et al., 2009; de Campos and Ji, 2011; Etminani et al., 2010) and linear programming ideas (Jaakkola et al., 2010; Cussens, 2011). The reported results show promising scalability of these methods under some favorable conditions, of which nature has, unfortunately, not yet been satisfactorily characterized. In particular, no worst-case upper bounds for their runtime are known that would be competitive to the bounds of the DP algorithms. What is worse, the methods are known to be unfeasibly slow even for some small benchmark instances that DP algorithms solve in a few minutes. Indeed, an illustrative example is the 19-variable image segmentation data set (from the UCI machine learning repository), on which Cussens's (2011) implementation of the linear programming method does not terminate within two hours, using the BIC scoring that results in a total of 8164 potential parent sets after pruning (B. Malone, personal communication). In comparison, the DP algorithms solve the instance in a few seconds. Note that our schemes further enable solving that instance in very small space, say, using about 2^{15} bytes, by increasing the total runtime by a factor around $2^{19-15} = 16$, or by running the algorithm on that many processors in parallel. Nevertheless, the branch-and-bound and linear programming techniques complement the DP methodology—hybrids between them is a plausible topic of future research.

Finally, it is worth noting that sometimes a "good" partial order of the nodes is actually known to the modeller. By "good" we mean one that encodes prior knowledge about the DAG to be found and that has a relatively small number of ideals. For example, in causal discovery and dynamic Bayesian networks it is quite expected that the precedence order is known for many pairs of nodes. In such a case, one needs to run the presented DP algorithm on just that partial order, thereby using much less time and space than the basic DP algorithm, which essentially cannot exploit given precedence constraints (besides excluding some parent sets for some nodes). In other words, the presented DP algorithm generalizes the basic DP algorithm to fully exploit given precedence constraints, if any.

Acknowledgments

The authors thank Kustaa Kangas, Petteri Kaski, Brandon Malone, and Teppo Niinimäki for valuable discussions. We also thank the anonymous reviewers of earlier versions of the manuscript for concrete suggestions that enabled us to improve the presentation and clarity of this paper substantially. This work was supported in part by the Academy of Finland, grants 125637, 218153, and 255675.

References

R. Bellman. Dynamic programming treatment of the travelling salesman problem. *Journal of the ACM*, 9(1):61–63, 1962.

A. Björklund and T. Husfeldt. Exact algorithms for exact satisfiability and number of perfect matchings. *Algorithmica*, 52:226–249, 2008.

H. L. Bodlaender, F. V. Fomin, A. M. C. A. Koster, D. Kratsch, and D. M. Thilikos. On exact algorithms for treewidth. In *Proceedings of the 14th Annual European Symposium on Algorithms (ESA)*, pages 672–683, 2006.

H. L. Bodlaender, F. V. Fomin, A. M. C. A. Koster, D. Kratsch, and D. M. Thilikos. A note on exact algorithms for vertex ordering problems on graphs. *Theory of Computing Systems*, 50(3): 420–432, 2012.

G. Brightwell and P. Winkler. Counting linear extensions. *Order*, 8:225–242, 1991.

D. M. Chickering. *Learning from Data: Artificial Intelligence and Statistics V*, chapter Learning Bayesian networks is NP-Complete, pages 121–130. Springer-Verlag, 1996.

D. M. Chickering, D. Heckerman, and C. Meek. Large-sample learning of Bayesian networks is NP-hard. *Journal of Machine Learning Research*, 5:1287–1330, 2004.

H. Chockler and J. Y. Halpern. Responsibility and blame: A structural-model approach. *Journal of Artificial Intelligence Research*, 22:93–115, 2004.

G. F. Cooper and E. Herskovits. A Bayesian method for the induction of probabilistic networks from data. *Machine Learning*, 9(4):309–347, 1992.

J. Cussens. Bayesian network learning with cutting planes. In *Proceedings of the 27th Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 153–160, 2011.

B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 2003.

C. P. de Campos and Q. Ji. Efficient structure learning of Bayesian networks using constraints. *Journal of Machine Learning Research*, 12:663–689, 2011.

C. P. de Campos, Z. Zeng, and Q. Ji. Structure learning of Bayesian networks using constraints. In *Proceedings of the 26th International Conference on Machine Learning (ICML)*, pages 113–120, 2009.

L. M. de Campos. A scoring function for learning Bayesian networks based on mutual information and conditional independence tests. *Journal of Machine Learning Research*, 7:2149–2187, 2006.

K. Etminani, M. Naghibzadeh, and A. R. Razavi. Globally optimal structure learning of Bayesian networks from data. In *Proceedings of the 20th International Conference on Artificial Neural Networks (ICANN)*, pages 101–106, 2010.

J. Flum and M. Grohe. *Parametrized Complexity Theory*. Springer, 2006.

Y. Gurevich and S. Shelah. Expected computation time for Hamiltonian path problem. *SIAM Journal of Computation*, 16(3):486–502, 1987.

D. Heckerman, D. Geiger, and D. M. Chickering. Learning Bayesian networks: The combination of knowledge and statistical data. *Machine Learning*, 20(3):197–243, 1995.

M. Held and R. M. Karp. A dynamic programming approach to sequencing problem. *Journal of the Society for Industrial and Applied Mathematics*, 10(1):196–210, 1962.

T. Jaakkola, D. Sontag, A. Globerson, and M. Meila. Learning Bayesian network structure using LP relaxations. In *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics (AISTATS)*, JMLR: W&CP, pages 358–365, 2010.

M. Koivisto. Parent assignment is hard for the MDL, AIC, and NML costs. In *Proceedings of the 19th Annual Conference on Learning Theory (COLT)*, pages 289–303, 2006.

M. Koivisto and P. Parviainen. A space–time tradeoff for permutation problems. In *Proceedings of the 21st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 484–492, 2010.

M. Koivisto and K. Sood. Exact Bayesian structure discovery in Bayesian networks. *Journal of Machine Learning Research*, 5:549–573, 2004.

B. Malone, C. Yuan, E. A. Hansen, and S. Bridges. Improving the scalability of optimal Bayesian network learning with external-memory frontier breadth-first branch and bound search. In *Proceedings of the 27th Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 479–488, 2011.

S. Ott and S. Miyano. Finding optimal gene networks using biological constraints. *Genome Informatics*, 14:124–133, 2003.

P. Parviainen and M. Koivisto. Exact structure discovery in Bayesian networks with less space. In *Proceedings of the 25th Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 436–443, 2009.

J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann, San Francisco, 1988.

J. Pearl. *Causality: Models, Reasoning, and Inference*. Cambridge University Press, Cambridge, 2000.

S. Provan and M. O. Ball. On the complexity of counting cuts and of computing the probability that a graph is connected. *SIAM Journal of Computing*, 12:777–788, 1983.

W. Savitch. Relationship between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4:177–192, 1970.

T. Silander and P. Myllymäki. A simple approach for finding the globally optimal Bayesian network structure. In *Proceedings of the 22th Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 445–452, 2006.

A. P. Singh and A. W. Moore. Finding optimal Bayesian networks by dynamic programming. Technical Report CMU-CALD-05-106, Carnegie Mellon University, June 2005.

P. Spirtes and C. Glymour. An algorithm for fast recovery of sparse causal graphs. *Social Science Computer Review*, 9:62–72, 1991.

G. Steiner. On the complexity of dynamic programming with precedence constraints. *Annals of Operations Research*, 26:103–123, 1990.

Y. Tamada, S. Imoto, and S. Miyano. Parallel algorithm for learning optimal Bayesian network structure. *Journal of Machine Learning Research*, 12:2437–2459, 2011.

T. S. Verma and J. Pearl. Equivalence and synthesis of causal models. In *Proceedings of the Sixth Annual Conference on Uncertainty in Artificial Intelligence (UAI)*, pages 255–270, 1990.

C. Yuan, B. Malone, and X. Wu. Learning optimal Bayesian networks using A* search. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI)*, pages 2186–2191, 2011.