
Learning Subproblem Complexities in Distributed Branch and Bound

Lars Otten

Department of Computer Science
University of California, Irvine
lotten@ics.uci.edu

Rina Dechter

Department of Computer Science
University of California, Irvine
dechster@ics.uci.edu

Abstract

In the context of distributed Branch and Bound Search for Graphical Models, effective load balancing is crucial yet hard to achieve due to early pruning of search branches. This paper proposes learning a regression model over structural as well as cost function-based features to more accurately predict subproblem complexity ahead of time, thereby enabling more balanced parallel workloads. Early results show the promise of this approach.

1 Introduction

This paper explores the application of learning for improved load balancing in the context of distributed search for discrete combinatorial optimization over graphical models (e.g., Bayesian networks, weighted CSPs). Specifically, we consider one of the best exact search algorithms for solving the MPE/MAP task over graphical models, AND/OR Branch and Bound (AOBB) [9], ranked first or second in the UAI'06 and '08 evaluations. We adapt the established concept of parallel tree search [5], where a search tree is explored centrally up to a certain depth and the remaining subtrees are solved in parallel. In the graphical model context we explore the search space of partial instantiations up to a certain point and solve the resulting conditioned subproblems in parallel.

The distributed framework is built with a grid computing environment in mind, i.e., a set of autonomous, loosely connected systems – notably, we cannot assume any kind of shared memory which many parallel algorithms build upon [4, 5, 1]. The primary challenge – and focus of this paper – is therefore to find a set of subproblems with balanced complexity, so that the overall parallel runtime will not be dominated by just a few of them. In the optimization context, however, the use of cost and heuristic functions for pruning makes it very hard to reliably predict and balance subproblem complexity ahead of time, even if structural parameters like induced width are known.

Our suggested approach and the main contribution of this paper is to estimate subproblem complexity by learning a regression model over the subproblems' parameters, structural as well as with respect to the cost function. This model is trained during preprocessing on a small number of subproblem samples and then used to predict the size of each subproblem's search space in advance, merging/splitting accordingly.

Prior work on estimating search complexity goes back to [8] and more recently [6], which predict the size of general backtrack trees through random probing. Similar schemes were devised for Branch and Bound algorithms [2], where search is run for a limited time and the partially explored tree is extrapolated. Our approach differs by sampling and learning entirely during preprocessing, allowing very fast repeated estimates when the parallelization frontier is iteratively computed.

We present some early results of this ongoing work, running with a varying degree of parallelism on haplotyping problems from the domain of genetic linkage analysis. While limited in scope at this point, results are promising, with good parallel speedups in particular.

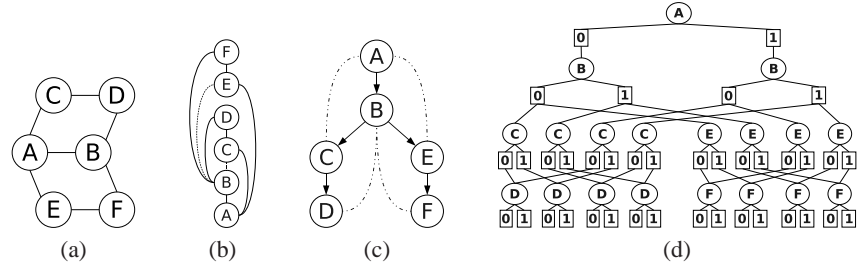


Figure 1: (a) Example primal graph with six variables, (b) its induced graph along ordering $d = A, B, C, D, E, F$, (c) a corresponding pseudo tree, and (d) the resulting context-minimal AND/OR search graph.

2 Background

We assume the usual definitions of a *graphical model* as a set of functions $F = \{f_1, \dots, f_m\}$ over discrete variables $X = \{X_1, \dots, X_n\}$, its *primal graph*, *induces graph*, and *induced width*. Figure 1(a) depicts the primal graph of an example problem with six variables. The induced graph for the example problem along ordering $d = A, B, C, D, E, F$ is depicted in Figure 1(b), its induced width is 2. Note that different orderings will vary in their implied induced width; finding an ordering of minimal induced width is known to be NP-hard, in practice heuristics like *minfill* [7] are used to obtain approximations.

2.1 AND/OR Search Spaces

The concept of AND/OR search spaces has been introduced as a unifying framework for advanced algorithmic schemes for graphical models to better capture the structure of the underlying graph [3]. Its main virtue consists in exploiting conditional independencies between variables, which can lead to exponential speedups. The search space is defined using a *pseudo tree*, which captures problem decomposition:

DEFINITION 1 (pseudo tree) *Given an undirected graph $G = (X, E)$, a pseudo tree of G is a directed, rooted tree $\mathcal{T} = (X, E')$ with the same set of nodes X , such that every arc of G that is not included in E' is a back-arc in \mathcal{T} , namely it connects a node in \mathcal{T} to an ancestor in \mathcal{T} . The arcs in E' may not all be included in E .*

AND/OR Search Trees : Given a graphical model instance with variables X and functions F , its primal graph (X, E) , and a pseudo tree \mathcal{T} , the associated *AND/OR search tree* consists of alternating levels of OR and AND nodes. OR nodes are labeled X_i and correspond to the variables in X . AND nodes are labeled $\langle X_i, x_i \rangle$, or just x_i and correspond to the values of the OR parent's variable. The structure of the AND/OR search tree is based on the underlying pseudo tree \mathcal{T} : the root of the AND/OR search tree is an OR node labeled with the root of \mathcal{T} . The children of an OR node X_i are AND nodes labeled with assignments $\langle X_i, x_i \rangle$; the children of an AND node $\langle X_i, x_i \rangle$ are OR nodes labeled with the children of X_i in \mathcal{T} , representing conditionally independent subproblems. It was shown that, given a pseudo tree \mathcal{T} of height h , the size of the AND/OR search tree based on \mathcal{T} is $\mathcal{O}(n \cdot k^h)$, where k bounds the domain size of variables [3].

AND/OR Search Graphs : Different nodes may root identical subproblems and can be merged through *caching*, yielding an *AND/OR search graph* of smaller size, at the expense of using additional memory during search. A mergeable node X_i can be identified by its *context*, the partial assignment of the ancestors of X_i which separates the subproblem below X_i from the rest of the network. Merging all context-mergeable nodes yields the *context minimal* AND/OR search graph. Given a graphical model, its primal graph G , and a pseudo tree \mathcal{T} , the size of the context-minimal AND/OR search graph is $\mathcal{O}(n \cdot k^{w^*})$, where w^* is the induced width of G over a depth-first traversal of \mathcal{T} and k bounds the domain size [3].

Figure 1(c) depicts a pseudo tree extracted from the induced graph in Figure 1(b) and Figure 1(d) shows the corresponding context-minimal AND/OR search graph. Note that the AND nodes for

B have two children each, representing independent subproblems and thus demonstrating problem decomposition. Furthermore, the OR nodes for D (with context $\{B, C\}$) and F (context $\{B, E\}$) have two edges converging from the AND level above them, signifying caching.

Weighted AND/OR Search Graphs : Given an AND/OR search graph, each edge from an OR node X_i to an AND node x_i can be annotated by *weights* derived from the set of cost functions F in the graphical model: the weight $l(X_i, x_i)$ is the sum of all cost functions whose scope includes X_i and is fully assigned along the path from the root to x_i , evaluated at the values along this path. Furthermore, each node in the AND/OR search graph can be associated with a *value*: the value $v(n)$ of a node n is the minimal solution cost to the subproblem rooted at n , subject to the current variable instantiation along the path from the root to n . $v(n)$ can be computed recursively using the values of n 's successors [3].

AND/OR Branch and Bound : AND/OR Branch and Bound is a state-of-the-art algorithm for solving optimization problems over graphical models. Assuming a maximization task, it traverses the context-minimal AND/OR graph in a depth-first manner while keeping track of a current lower bound on the optimal solution cost. During expansion of a node n , this lower bound l is compared with a heuristic upper bound $u(n)$ on the optimal solution below n – if $u(n) \leq l$ the algorithm can prune the subproblem below n [3].

Distributed AND/OR Branch and Bound :

Our distributed implementation of AND/OR Branch and Bound is based on the notion of parallel tree search [5], where a search tree is explored centrally up to a certain depth and the remaining subtrees are solved in parallel. In the context of graphical models we explore the search space of partial instantiations up to a certain point and solve the resulting conditioned subproblems in parallel. Applied to the search graph from Figure 1(d), for instance, we could obtain eight independent subproblems as shown in Figure 2, with a conditioning search space (in gray) spanning the first two levels (variables A and B). In the following we will outline our approach to finding a balanced set of subproblems.

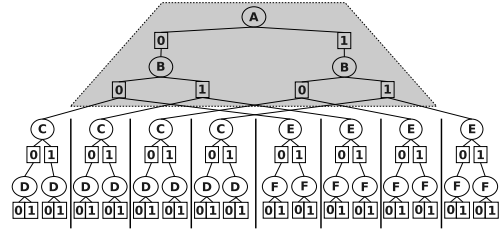


Figure 2: Parallelization applied to the example problem from Figure 1, resulting in eight independent subproblems, with conditioning search space in gray.

3 Load Balancing through Complexity Prediction

Our general scheme to find a balanced set of subproblems is outlined in Algorithm 1: starting with just the search space root node (corresponding to a single, large subproblem), we iteratively pick the subproblem with largest estimated complexity and condition it further, until the desired level of parallelism (measured by the number of subproblems, typically chosen to be ten times the number of available CPUs) is obtained. Note that each node generated needs to have its subproblem complexity estimated, resulting in many such queries; it is therefore advisable to keep most of the estimation complexity to an offline preprocessing phase in order to ensure fast prediction queries.

The following section derives our principled approach to subproblem complexity prediction, which we formulate as a regression learning problem. We note that, while the goal in the present context is

Algorithm 1 Pseudo code for subproblem generation

Input: Pseudo tree \mathcal{T} with root X_0 , minimum subproblem count p , complexity estimator \hat{N} .

Output: Set F of subproblem root nodes with $|F| \geq p$.

- 1: $F \leftarrow \{X_0\}$
 - 2: **while** $|F| < p$:
 - 3: $n' \leftarrow \arg \max_{n \in F} \hat{N}(n)$
 - 4: $F \leftarrow F \setminus \{n'\}$
 - 5: $F \leftarrow F \cup \text{children}(n')$
-

to ensure balanced subproblem complexity in the distributed scheme, accurate complexity prediction is a worthwhile issue in itself and a possible subject of future research.

3.1 Subproblem Complexity Prediction

Given a search node n , we propose to model the complexity of the subproblem below it (measured by the number of node expansions required for its solution) as an exponential function of various subproblem features $x_j(n)$, capturing the exponential nature of the search space size as follows:

$$N(n) = b^{\sum_j \lambda_j x_j(n)} \quad (1)$$

The subproblem features $x_j(n)$ we consider can be divided into two groups:

- Structural: $d(n)$, depth of n in the conditioning search space; $h(n)$, height of the subproblem pseudo tree below n ; $w(n)$, induced width of the conditioned subproblem below n ; $c(n)$, the number of problem variables in the subproblem below n .
- Cost-function related: $U(n)$, heuristic upper bound on the subproblem solution cost below n , as used by the Branch and Bound algorithm for pruning; $L(n)$, lower bound on the subproblem solution cost as derived from the current best (overall) solution.

If we instead consider the log complexity, Equation 1 becomes the following:

$$\log N(n) = \sum_j \lambda_j x_j(n) \quad (2)$$

Finding suitable parameter values λ_j can thus be formulated as a well-known *linear regression* problem. In other words, given a set of m sample subproblems n_k and their respective complexities $N(n_k)$, $1 \leq k \leq m$, we aim to find parameters λ_j that minimize the mean squared error:

$$MSE = \frac{1}{m} \sum_{k=1}^m \left(\sum_j \lambda_j x_j(n_k) - \log N(n_k) \right)^2 \quad (3)$$

The optimal selection of λ_j 's can be computed using the *ordinary least squares* method: We take each sample subproblem's features as a row of the design matrix X (i.e., $X_{i,j} = x_j(n_i)$) and let Y with $Y_i = \log(N(n_i))$ denote the column vector of log subproblem sizes. With $\|\cdot\|$ as the Euclidean norm, we then minimize $\|X\Lambda - Y\|^2$ through the closed-form expression $\hat{\Lambda} = (X^T X)^{-1} X^T Y$. Optionally applying *ridge regression* for regularization we get:

$$\hat{\Lambda} = (X^T X + \alpha I)^{-1} X^T Y \quad (4)$$

where I is the identity matrix and $\alpha \in \mathbb{R}$ a small constant (e.g. $\alpha = 0.01$). Given the learned parameters $\hat{\lambda}_j$ we can then predict the (log) complexity of a new subproblem n' as:

$$\log \hat{N}(n') = \sum_j \hat{\lambda}_j x_j(n') \quad (5)$$

In the following we briefly describe how we can obtain the sample set of subproblems required to learn the parameter values $\hat{\lambda}_j$ of the regression model.

3.2 Subproblem Sampling

Recall first that AND/OR Branch and Bound is a depth-first search scheme. Second, we realize that any leaf node that is generated is a subproblem in itself (of size 1) and as the algorithm backtracks from the leaf the solved subproblem expands in size. Hence obtaining a single sample subproblem is straightforward: we run the search for a limited number of operations and take the largest solved subproblem as the sample. To obtain multiple different samples, we introduce randomness in the value choices of the search up to a certain depth, below which the algorithm continues to use the heuristic upper bound as a value selection heuristic as before.

For the parallel results in the following section, for a given problem instance we sampled 10 subproblems each of size approx. 10,000, 40,000, 80,000, 120,000, 160,000, and 200,000 nodes, for a total of 60 samples. This sampling process takes around 10 minutes per problem instance and is performed offline at this stage – eventually it will be fully integrated into the parallel scheme.

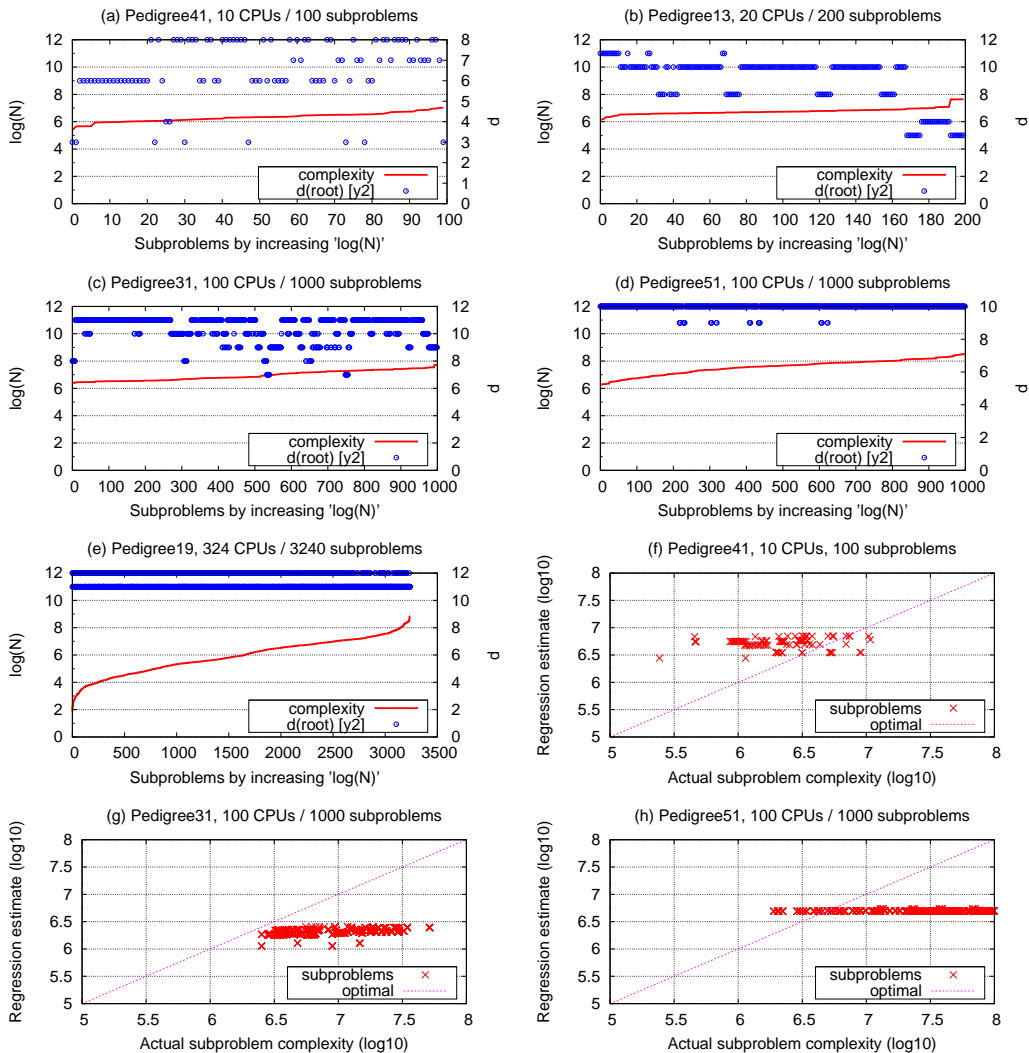


Figure 3: **(a)–(e)**: Subproblem statistics for select runs. Shown are each subproblem’s complexity as well as the depth of its root in the central search space (plotted against second y-axis). **(f)–(h)**: Scatter plots of subproblem complexities against regression estimates.

4 Empirical Results

This line of work is an ongoing effort, with a more comprehensive study to be carried out in the future; here we present some early results. We note that finding suitable benchmark problems is not an easy feat: if problems are too simple, parallelization is most likely detrimental overall, due to preprocessing and overhead from the distributed scheme. Many very hard problems, on the other hand, will still remain infeasible for practical purposes, in particular since every experiment binds a potentially large number of shared parallel resources for an extended period of time.

We report on five haplotyping problems from the domain of genetic linkage analysis, which correspond to MPE queries over Bayesian networks. With sequential runtimes from under one hour to over six days, we varied the number of CPUs depending on the problem complexity, since massive parallelism is futile for relatively simple problems. For each instance below, n is the number of problem variables, k its maximum domain size, w the induced width, and h the pseudo tree height.

Table 1 contains results in terms of parallel runtime with varying levels of parallelism, depending on overall problem complexity. Also included are sequential solution times (using plain AOBB)

| | sequential | CPU | time | spd | CPU | time | spd |
|------------|------------|-----|--------------|-----|-----|--------------|-----|
| pedigree41 | 2,247 | 5 | 556 | 4 | 10 | 326 | 7 |
| pedigree13 | 12,662 | 10 | 1,243 | 10 | 20 | 748 | 17 |
| pedigree31 | 92,078 | 50 | 2,078 | 44 | 100 | 1,134 | 81 |
| pedigree51 | 570,411 | 100 | 6,382 | 89 | 324 | 2,275 | 251 |
| pedigree19 | 1,659,324 | 324 | 8,396 | 198 | | | |

Table 1: Sequential and parallel runtime results for different number of CPUs (number of subproblems is always ten times number of CPUs). “spd” is speedup vs. sequential. All times in seconds.

and relative speedup. Figure 3 shows detailed subproblems statistics: (a) through (e) depict the complexity of the individual subproblems for select runs and allow us to assess load balancing; (f) through (h) directly contrast the complexity estimates from the learned regression model with the actual subproblem complexities, enabling evaluation of the prediction quality.

Pedigree41 ($n = 1062, k = 5, w = 33, h = 100$) : Solved sequentially in under one hour, this problem doesn’t leave room for much parallelism and using more than 10 CPUs would add little benefit. The subproblem complexities, however, seem fairly balanced (Fig. 3(a)).

Pedigree13 ($n = 1077, k = 3, w = 32, h = 102$) : Sequential AOBB takes about 3 1/2 hours on this problem, using 10 CPUs yields almost perfect linear speedup. The effect doesn’t hold as strongly for 20 CPUs, but the time improvement is still pronounced. The load balancing is acceptable as well (Fig. 3(b)), however we found that the underlying model predictions (not pictured) are many orders of magnitude different from the actual complexities, which will need further investigation.

Pedigree31 ($n = 1183, k = 5, w = 30, h = 85$) : We obtain favorable speedups with up to 100 processors. Similarly, the distribution of subproblem complexities seems relatively balanced (Fig. 3(c)). The underlying estimates, however, are again not very accurate (Fig. 3(g)).

Pedigree51 ($n = 1152, k = 5, w = 39, h = 98$) : One of the two problems where using all available 324 CPUs makes sense, yielding a substantial speedup. The subproblem complexities are not quite as balanced (Fig. 3(d)) and the underlying predictions quite crude (Fig. 3(h)), but it does not appear to impact the overall solution time negatively.

Pedigree19 ($n = 693, k = 25, w = 25, h = 98$) : The hardest problem in this set, its parallel speedup is a bit less distinct. We think this is due to the considerable imbalance across subproblems (Fig. 3(e)); in fact the overall time is dominated by a few long-running subproblems.

Overall, however, we observe very reasonable parallel speedups across instances; load balancing seems fairly effective, with the exception of pedigree19, which we need to investigate more closely. However, while our framework appears to generally enable effective load balancing, the accuracy of the underlying estimates is mediocre at best; further research is needed here as well.

5 Conclusion & Future Work

We have developed a distributed Branch and Bound framework that uses learning to achieve better load balancing on a computational grid. In particular, we proposed to train a linear regression model on structural and cost-function based features of sample subproblems. The resulting model is then used to determine a suitable parallelization of the entire problem search space.

Early results shown in Section 4 have shown promise, but also highlighted areas for future work. We were able to obtain very good parallel speedups for very complex as well as (using fewer CPUs) relatively simple problems and generally observed effective load balancing. However, we also found that in most cases the learned regression model didn’t predict complexities very accurately, sometimes leading to imbalanced subproblem complexities. The most notable example in this regard was pedigree19, which will require more in-depth analysis.

Future work will involve identifying additional features to add to the model, experimenting with varying sample sizes, and, most importantly, a more extensive empirical evaluation of the various aspects of our scheme on more problem instances. Another path worth investigating might be using non-linear models and to compare their performance to the current linear regression.

References

- [1] Geoffrey Chu, Christian Schulte, and Peter J. Stuckey. Confidence-based work stealing in parallel constraint programming. In Ian Gent, editor, *CP*, volume 5732 of *Lecture Notes in Computer Science*, pages 226–241, Lisbon, Portugal, September 2009. Springer-Verlag.
- [2] Gérard Cornuéjols, Miroslav Karamanov, and Yanjun Li. Early estimates of the size of branch-and-bound trees. *INFORMS Journal on Computing*, 18(1):86–96, 2006.
- [3] Rina Dechter and Robert Mateescu. AND/OR search spaces for graphical models. *Artif. Intell.*, 171(2-3):73–106, 2007.
- [4] Bernard Gendron and Teodor Gabriel Crainic. Parallel branch-and-bound algorithms: Survey and synthesis. *Operations Research*, 42(6):1042–1066, 1994.
- [5] Ananth Grama and Vipin Kumar. State of the art in parallel search techniques for discrete optimization problems. *IEEE Trans. Knowl. Data Eng.*, 11(1):28–35, 1999.
- [6] Philip Kilby, John Slaney, Sylvie Thiébaux, and Toby Walsh. Estimating search tree size. In *AAAI*, pages 1014–1019. AAAI Press, 2006.
- [7] Uffe Kjaerulff. Triangulation of graphs – algorithms giving small total state space. Technical report, Aalborg University, 1990.
- [8] Donald E. Knuth. Estimating the efficiency of backtrack programs. *Mathematics of Computation*, 29(129):121–136, 1975.
- [9] Radu Marinescu and Rina Dechter. AND/OR Branch-and-Bound search for combinatorial optimization in graphical models. *Artif. Intell.*, 173(16-17):1457–1491, 2009.