Search Algorithms for Solving Queries on Graphical Models & the Importance of Pseudo-trees in their Complexity.

University of California, Irvine

CS199: Individual Study with Rina Dechter

Héctor Otero Mediero
hoterome@uci.edu

June 16, 2017

Contents

1 Introduction

Traditional search algorithms work on problems that can be modeled in a tree-like shape, this is, solving a problem corresponds to traversing the tree from the root to one of the leaves of the tree that represents a solution. Each one of the edges can be seen as a transition between different states of the problem where each state depends on the previous and the decision taken.

There's a subset of these problems where we can identify subproblems which are independent from each other, in other words, the solutions of one don't depend on the other. Sometimes we can also identify equivalent subproblems that can be solved interchangeably. Consequently, it's interesting to have a graph than can model these relations and, later, algorithms that exploit them to solve the given problems. The models that capture these independencies are called AND/OR trees (or graphs depending on their connectivity) and yield solutions in the form of a sub-tree.

An example modeled with a common search tree (which corresponds to an OR tree) has inherently larger size than an AND/OR tree since the later as it takes into account the subproblems' independency. Intuitively we can think that an OR tree solves subproblems in a chain-like manner while AND/OR trees branches them out yielding a tree with a smaller height.

As it'll be seen later, these graphs can be used to model inference problems (graphical models, in general) and exact and approximated algorithms have been implemented used to obtain the answers compute the answers to the most common queries. The modeling, solution and complexity of finding solutions to these problems are the subject of study of the present work. Orderings of variables and pseudo-trees, the underlying structures needed to construct the problem's space, will be studied as well due to their impact in the complexity of said algorithms. The new contributions of this work are the analysis of the trade-offs between the width and height of the pseudo-trees and the implementation of AND/OR Beam Search and Anytime AND/OR Beam Search as an attempt to tackle the weaknesses of some of the state-of-the-art algorithms.

2 Graphical Models

There's a broad variety of problems that can be solved by using search algorithms that work on AND/OR spaces, including probabilistic reasoning and constraint satisfaction problems. More generally, any problem that can be reduced to a graphical model can be solved with such algorithms.

This section is dedicated to the definitions of different concepts that form the necessary background for the following section as following the notation in [Dechter, 2013].

A **graphical model** is a tuple $\mathcal{M} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \otimes \rangle$ where:

1. $\mathbf{X} = X_1,...,X_n$ is a finite set of variables.

2. $\mathbf{D} = D_1,...,D_n$ is the corresponding set of finite domains of values the variables can take.

3. $\mathbf{F} = f_1,...,f_r$ is a set of functions, defined over scopes of variables $\mathcal{S} = S_1,...,S_r$, where $S_i \subseteq$ X.

4. $\otimes$ is a *combination* operator. It defines a global function for the graphical model joining together the functions defined in F. This is, a function $\otimes_{i=1}^{r} f_i$ whose scope is $\mathbf{X}$.

Queries on this graphical model are defined with respect to this *global function*. The simplest query that can be found is the result of an assignment of values to the set X. More interesting and complex ones include the maximization/minimization of this value or counting the amount of assignments that yield a non-zero result (useful for CSPs). These queries, define an added operator (max, min, sum respectively) on the global function called marginalization operator. A combination of a marginalization operator and a graphical model form a reasoning problem.

In the following sections, we'll see how different well-known problems of the literature are mapped to a reasoning problem.

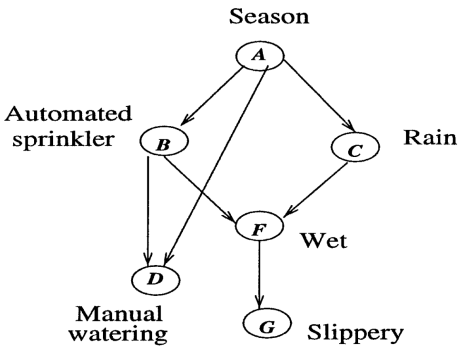


Figure 1: Example of a Bayesian Network. [Pearl, 1988]

3 Bayesian Networks

Bayesian Networks, as introduced in [Pearl, 1988], model the conditional relations between a set of random variables. These networks are represented as directed acyclic graphs which intuitively can be thought as capturing the causal relationship between the variables. Additionally we include in the definition a set of Conditional Probability Tables (CPTs) that determine the values for the probability of a variable (node in the graph) given its parents.

It can be easily seen that Bayesian networks can be mapped to a graphical model where:

- $\mathbf{X} = \{X_1, X_2, ..., X_n\}$, the set of random variables in the network.

- $\mathbf{D} = \{D_1, D_2, ..., D_n\}$, the corresponding set of values they can take.

- $\mathbf{F} = \{F_1, F_2, ..., F_p\}$ is the set of conditional probabilities included in the CPTs, and their corresponding scopes are denoted as $\mathrm{S} = \{S_1, S_2, ..., S_p\}$ and are formed by a given variable and their parents.

- The *global function* defines the joint probability distribution of the Bayesian network. In other words, the combination operator is defined to be the product of the defined probabilities (F).

$$P(X) = \prod_{fi \in F} f_i$$

As we can define a Bayesian network as a graphical model, we can give meaning to the queries that can be formulated over the global function and their respective marginalization operators.

It is obvious to see, that a full assignment of values to X, gives us the corresponding probability of the assignment to happen. More complex queries are defined next. In some cases, they include a set of assignments, E = e, called evidence $\{X_i = e_i \mid X_i \in E, E \subseteq X, e_i \in D_i\}$, that represent fixed assignments to certain variables

- **Probability of evidence**, $P(E = e)$, represents the probability of a certain assignment happening. As in the example before the marginalization operator is the sum but this time it's defined over all the variables.

$$P(E = e) = \sum_{X} \prod_{j} P(X_j | X_{paj}, E = e)$$

- **Belief updating** or **posterior marginals**: the updated probabilities of the random variables given some evidence, $P(X_i|E = e)$. It's computed summing out all the variables that are not X_i.

$$P(X_i|E = e) = \sum_{X - Xi} \prod_{j} P(X_j | X_{paj}, E = e)$$

where X_{paj}, represents the set of parents of X_j. This is equivalent to a reasoning problem where the marginalization operator is the sum.

- **Most probable explanation** (MPE) corresponds to the assignment $x^0 = \{X_i = x_i | X_i \in \bar{E}\}$ that maximizes the global function with respect to the variables not in the evidence, $\bar{E}$. Namely,

$$x^0 = \underset{x}{\text{argmax}} \prod_{X_j \in \bar{E}} P(X_j = x_j | X_{paj}, E = e)$$

In this case, $\otimes = argmax$. The value of this assignment can be found with a *max* marginalization operator.

Maximum a posteriori hypothesis (MAP). Given a set of variables A $\subseteq \bar{E}$, finding an assignment, $a^0 = \{X_i = a_i | X_i \in A\}$ that maximizes its posterior marginals.

$$a^0 = \underset{a}{\text{argmax}} \sum_{X-A} \prod_{j} P(X_i = a_i | X_{paj}, E)$$

There are two important remarks that should be made to conclude the explanation of the queries on Bayesian Networks. Firstly, in terms of naming convention, MPE is sometimes called MAP (leading to confusion) and MAP is consequently named marginal MAP. During the rest of the paper we will use *MPE* to refer to the Most Probable Explanation and *marginal MAP* to the Maximum a posteriori hypothesis. Secondly, MPE and marginal MAP are really similar in terms of formulation (the only difference being that $A \subseteq \bar{E}$) but they don't compute the same set of solutions. Nevertheless, MPE is sometimes used to give an approximate result to a marginal MAP query as it is easier to compute as we will see later.

Example: Given the Bayesian network in Figure 1, and a bivalued domain (True, False) for each one of the variables and A = {summer, autumn, winter, spring }, we can exemplify the previous queries as:

- *Probability of Evidence*: What is the probability that the floor is wet? P(F = True).
- *Belief Updating*: Knowing that the floor is wet, what is the probability that it rained? P(C = True | F = True)
- *MPE*: Knowing that we're in summer and that neither the sprinklers or manual watering are True. What's the most probable state of the rest of the variables?
- *Marginal MAP*: Knowing that the floor is wet due to manual watering, what is the most probable combination of season and rain, values?

4 Bucket Elimination

4.1 Bucket Elimination: Belief Updating

It's important to analyze inference methods before delving into search algorithms as it will give us a better understanding of why the later were developed and are preferred in order to solve certain problems. The main algorithm used is Bucket Elimination which, with a small set of changes, can be used to solve all the queries presented previously as explained in [Dechter, 1996]. During this section, we will describe said algorithm including an analysis on its complexity.

We will begin by describing the version used to solve Belief Updating and Probability of evidence queries, named BE-BEL.

Given a Bayesian Network, an ordering and, optionally, evidence, the algorithm can be described as the two following processes:

- **Function partitioning** in the different buckets. We will place a function in the bucket of the variable that, belonging to the function's scope, appears the latest in the ordering. We call this variable the *highest-ordered variable* in the scope of the function.

- **Bucket processing**: if the bucket's variable is part of the evidence provided, we will assign the value to each of the functions in the bucket and place each one of them in the bucket of the first variable that belongs to the scope of the function and appears before the current bucket. Otherwise, we will produce a new function, λ, as the result of the multiplication of all the functions in the bucket summed over the bucket's variable (eliminating, this way,

the variable from the function scope). Afterwards, we will place this *message* in the bucket of the highest-ordered variable of the message.

An schematic execution (with evidence G = 1) along the ordering d = {A, C, B, F, D, G} and the pseudocode for the algorithm can be seen below.

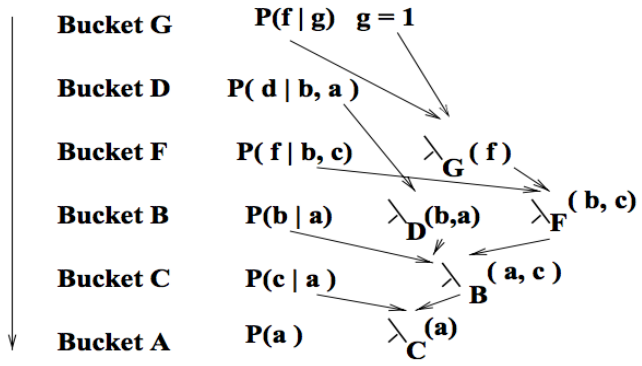


Figure 2: Execution of Bucket Elimination. [Dechter, 1996]

ALGORITHM BE-BEL
Input: A Bayesian network B = (X, D, P, $\prod$), an ordering d = (X_1, X_2, ..., X_n) and evidence E.
Denote B_i the bucket corresponding to X_i.
Denote nearest(F_i, X_j) as the bucket B_k, s.t. X_k is the first variable in S_i that appears before X_j in d
Partition functions into buckets:

1. **for** i = 1 to $\|P\|$
2. **for** j = n to 1
3. **if** X_j appears in P_i
4. place P_i in B_j
5. break

Denote Ψ_i as the product of the functions in B_i
Backward: process buckets in inverse order, generating messages λ_p:

6. **for** i = n to 2
7. **if** X_i in E
8. **for** F_j = {$\lambda_1, \lambda_2, ..., \lambda_p$ and Ψ_i} in B_i
9. assign $X_i = x_i$ to F_j
10. place the function, F_j, in nearest(F_j, X_i)
11. **else**
12. $\lambda_i \leftarrow \sum_{X_i} \Psi_i \prod_{j=1}^{p} \lambda_j$
13. place λ_i in nearest(λ_i, X_i)

14. P(E) = α = $\sum_{X_1} \Psi_1 \prod_{\lambda \in B_1} \lambda$
15. P(X_1|E) = $\frac{1}{\alpha} \Psi_1 \prod_{\lambda \in B_1} \lambda$

4.2 Bucket Elimination: MPE & Marginal MAP

The implementations of the Bucket Elimination for the Most Probable Explanation and Maximum A Posteriori queries is pretty straightforward and requires only two modifications with respect to BE-Bel:

- Line 12: The computation of the messages is different for each one of the tasks, therefore, there's a mandatory modification in this line of the code. For MPE, it consists of changing the marginalization operator for maximization.

For Marginal MAP, we need to distinguish between the variables (buckets) that are hypothesized that will use a maximization operator and the rest, which will use the sum for marginalizing.

Forward phase: Both MPE and Marginal MAP return an assignment (besides its value), therefore we need an extra phase that traverses the buckets computes the value of each variable. In order to select a value for a variable ($X_n = x_n$), we first assign the value selected for the variables evaluated before X_n, (X_1, X_2, ..., X_n). Then, we select the value of the assignment which maximizes the functions in the evaluated bucket (argmax_{x_i}). MPE will return an assignment to all of the variables that are not evidence and Marginal MAP an assignment to the hypothesized variables.

Below we can see the pseudo-code for BE-mpe.

ALGORITHM BE-MPE
Input: A Bayesian network B = (X, D, P, $\prod$), an ordering d = (X_1, X_2, ..., X_n) and evidence E.
Denote B_i the bucket corresponding to X_i.
Denote nearest(F_i, X_j) as the bucket B_k, s.t. X_k is the first variable in S_i that appears before X_j in d
Partition functions into buckets:

```
1. for i = 1  to ||P||
2.  for j = n to 1
3.    if X_j appears in P_i
4.      place P_i in B_j
5.      break
```

Denote Ψ_i as the product of the functions in B_i
Backward: process buckets in inverse order, generating messages λ_p:

```
6. for i = n to 2
7.  if X_i in E
8.    for F_j = {λ_1, λ_2, ..., λ_p and Ψ_i} in B_i
9.      assign X_i = x_i to F_j
10.     place the function, F_j, in
        nearest(F_j, X_i)
11. else
12.   λ_i ← max_{X_i} Ψ_i ∏_{j=1}^{p} λ_j
13.   place λ_i in nearest(λ_i, X_i)
```

Forward: Generate the value and assignment for the MPE tuple.

```
14. mpe = max_{X_1} Ψ_1 ∏_{λ∈B_1} λ
15. for i = 1 to n:
16.   Assign the variables evaluated
      before to λ_j where they appear
17.   x_i^o = argmax_{x_i} Ψ_i ∏_{j=1}^{p} λ_j
18. return mpe, x^0 = {x_1^o, x_2^o, ..., x_n^o}
```

ALGORITHM BE-MAP

Input: A Bayesian network B = (X, D, P, $\prod$), a set of hypothesized variables A = (A_1, A_2, ..., A_k), an ordering d = (X_1, X_2, ..., X_n) where the hypothesized variables are the first in the ordering, evidence E.
Denote B_i the bucket corresponding to X_i.
Denote nearest(F_i, X_j) as the bucket B_k, s.t. X_k is the first variable in S_i that appears before X_j in d
Partition functions into buckets:

```
1. for i = 1  to ||P||
2.  for j = n to 1
3.    if X_j appears in P_i
4.      place P_i in B_j
5.      break
```

Denote Ψ_i as the product of the functions in B_i
Process buckets in inverse order, generating messages λ_p:

```
6. for i = n to 2
7.  if X_i in E
8.    for F_j = {λ_1, λ_2, ..., λ_p and Ψ_i} in B_i
9.      assign X_i = x_i to F_j
10.     place the function, F_j, in nearest(F_j, X_i)
11. else
12.   if X_i ∉ A
13.     λ_i ← Σ_{X_i} Ψ_i ∏_{j=1}^{p} λ_j
14.   else
15.     λ_i ← max_{X_i} Ψ_i ∏_{j=1}^{p} λ_j
16.   place λ_i in nearest(λ_i, X_i)
```

Forward: Generate the value and assignment for the Marginal MAP tuple.

```
17. map = max_{X_1} Ψ_1 ∏_{λ∈B_1} λ
18. for i = 1 to |A|:
19.   Assign the variables evaluated
      before to λ_j where they appear
20.   x_i^o = argmax_{x_i} Ψ_i ∏_{j=1}^{p} λ_j
21. return map, x^0 = {x_1^o, x_2^o, ..., x_k^o}
```

4.3 Complexity

If we analyze BE-BEL algorithm, we can see how the most costly computation in the inner loop is generating the message function (line 12). To compute a single message, we have to generate a value for each possible combination of variables in the message scope. This calculation is exponential in the amount of variables in the bucket's scope. More specifically, if we denote k as the largest domain and the bucket's scope size s (not including the bucket's variable), k^s.

Consequently, in terms of space, the function's complexity is O($n \cdot k^s$) as we will generate a maximum of k^s for each of the n buckets. A bit more elaborately we can deduce that the time complexity is O($r \cdot k^{s+1}$). Generating the previous messages takes k^{s+1} since this s+1 the amount of variables in the bucket. The number of functions to be processed for each bucket (line 7) is the union of the messages received from its children, deg_i and the functions originally in the bucket, r_i. Thus, for the totality of the buckets we have $\Sigma_i(r_i + deg_i)k^{s_i+1}$. We can easily see that the sum over r_i gives r. Moreover, we can observe that the sum of the degrees of each node (the number of messages it receives) is bounded by the number of variables n, as each variable can generate as much as one message. Therefore we get that the time complexity is $O((r+n)k^{s+1})$. And assuming r > n[1], we get $O(r \cdot k^{s+1})$

[1]For Bayesian Networks r = n

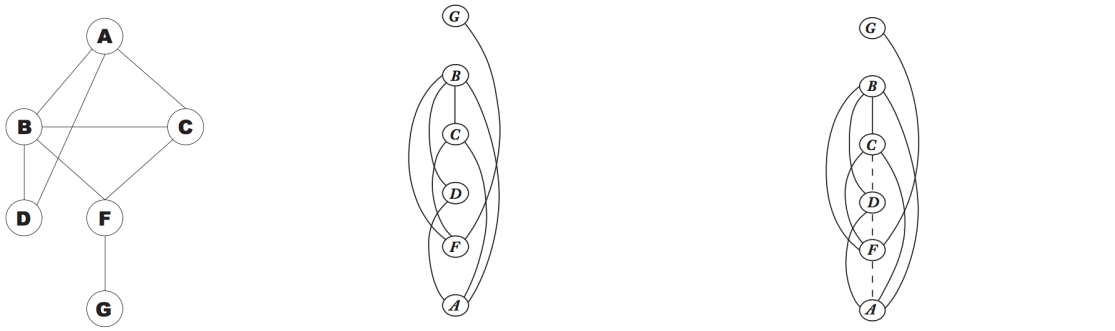
The size of the largest scope, s, can't be directly computed from the input graph. In order to characterize it with respect to it, we introduce the following definitions:

- **Primal graph** (also moral graph): Given a directed graph G = (V, E) its moral graph is a undirected graph G' = (V, E ∪ E') where E' is the set of edges that links nodes with common children. This process is also called moralizing a graph.

- **Induced graph** : Given an undirected graph G = (V, E) and an ordering of the vertices, d, an induced graph, G' = (V, E'), includes an edge between two vertices if it's present in the original edges, E, or if two vertices have a neighbor in common that appears after them in the ordering. In other words, if we construct the graph by processing the nodes in an inverse manner, we connect all the node's neighbors that precede it in the ordering. We also call this set of nodes, the parents of the node.

- **Induced width of an induced graph** (also treewidth), w(d), given a graph and an ordering, its induced-width is the maximum number of parents a node has in the induced graph.

- **Induced width of a graph**, w^*, we define it as the minimum induced-width across all the possible orderings of the vertices.



(a) The moral graph of Figure 1.

(b) The moral graph along the ordering d = {A, F, D, C, B, G}.

(c) Its induced graph.

Figure 3: [Dechter, 2013]

Above we can see the moral graph corresponding to the Bayesian Network shown in section 1. Then we can see it represented upside-down along the ordering d = (A, F, D, C, B, G). In order to generate the induced graph, we process it from top to bottom; for each variable we check if it has two or more neighbors appearing before it in the ordering (lower in the representation), if this is the case, we connect them all. For the given example, B has four parents (C, D, F, A) so we connect C-D, D-F and F-A (note that the rest of connections are already present) with a dotted line. The induced width of this graph is 4 as B has four parents. Only edges that go from a node appearing above to one below are counted as parents, for example, C has 3 parents (D, F, A).

If we follow the execution of Bucket Elimination in the representation shown below, we can see that a node with a single parent will always send it a message. A node with several parents will first pass it to one of them which will afterwards pass it onto the next one, until all of them are traversed. If we connect all the parents, as if to symbolize the flow of information in the graph, we can see that we have generated an induced graph along the ordering given. More importantly, the arcs from a node to its parents represent the amount of variables that are considered in that bucket in particular. Due to this, we can rewrite the largest scope size as $w^*(d) + 1$ as in a single bucket, we will consider as many variables as parents a variable has, plus the variable itself.

Another relevant observation we can make to the algorithm is related to the evidence present. It is simple to see that processing a bucket whose variable has been instantiated helps reducing the computational complexity as the functions generated will see their scope reduced in one (the bucket's variable) avoiding a potential increment generated by the combination of the different functions as in line 12. This can be reflected graphically if, while

constructing the graph we don't include an edge among parents of instantiated bucket variables. The induced-width of this graph is referred as **conditional induced-width**, w^*_E(d), and represents a tighter bound for the complexity of the Bucket Elimination algorithm. This observation is the base for the algorithms that combine search and inference to solve queries, for example, **cutset-conditioning**. This scheme generates a graph for each of the possible assignments of a given subset of the variables (cutset) with the aim of generating problems with a reduced (conditional). In this scheme there's a trade-off between the amount of problems to be solved (exponential in the size of the cutset) and the time/space complexity of the subproblems generated. Special cases for this scheme include cycle-cutset, which looks for a cutset that yields problems whose graph is acyclic, and q-cutset, that reduces the problem to one whose induced-width equal to q.

Therefore, all of the Bucket Elimination algorithms possess the same time and space complexity which is specified through the concept of conditional induced-width as follows:

Time complexity: $O(r \cdot k^{w^*_E(d)+1})$
Space complexity: $O(n \cdot k^{w^*_E(d)})$

To sum up, given a Bayesian Network, we can compute the complexity of executing Bucket Elimination over a given ordering by moralizing the network, generating its induced graph along the ordering and finally computing its conditional induced-width.

4.4 Bucket-Tree Elimination and Cluster-Tree Schemes

In this section we'll talk about extensions of the Bucket Elimination algorithm without diving in too deeply in the details of the algorithms referenced. Bucket Elimination can be interpreted as an algorithm that sends messages along a tree of buckets, this is, if each bucket is a node in the tree and we connect every two buckets that send a message to each other yielding a directed tree.

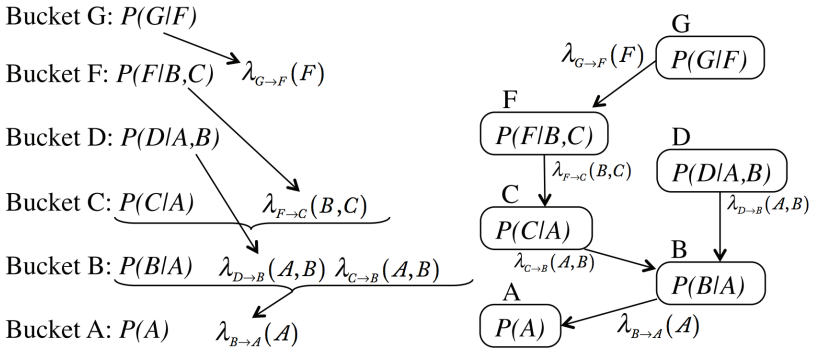


Figure 4: A Bucket Elimination execution represented as a Bucket Tree. [Dechter, 2013]

We know that for Bucket Elimination to compute the beliefs for a bucket variable we need to place this bucket as the first variable in the ordering. It can be shown in the bucket tree representation that once the original messages λ_i are generated we can propagate this information to calculate beliefs for other variables by simply redirecting the necessary edges that make its bucket the last in the ordering. The buckets for which an edge has been redirected generate a new set of messages, denoted π_i. For example, if a bucket A received a message from bucket B and their edge is redirected, A calculates a new message π_i by multiplying all functions in its augmented bucket (the original functions in the bucket the messages received) excluding the message received from B and summing out the variables in bucket A that are not in bucket B. Once all of the π_i messages have been computed, each bucket can answer any belief query on a subset of the variables of its scope. This algorithm called Bucket-Tree Elimination and its asynchronous version Bucket-Tree Propagation can be used to speed-up the calculation of tasks that include several reasoning tasks and can be found more described in larger detail in Chapter 5 of [Dechter, 2013].

A generalization of the bucket-tree scheme is the result of collapsing adjacent buckets into a single cluster. Each one of this clusters can be collapsed with other adjacent clusters or buckets at our own discretion, resulting in a spectrum of cluster-trees that range from the original bucket tree (each cluster formed by a single variable) to a tree

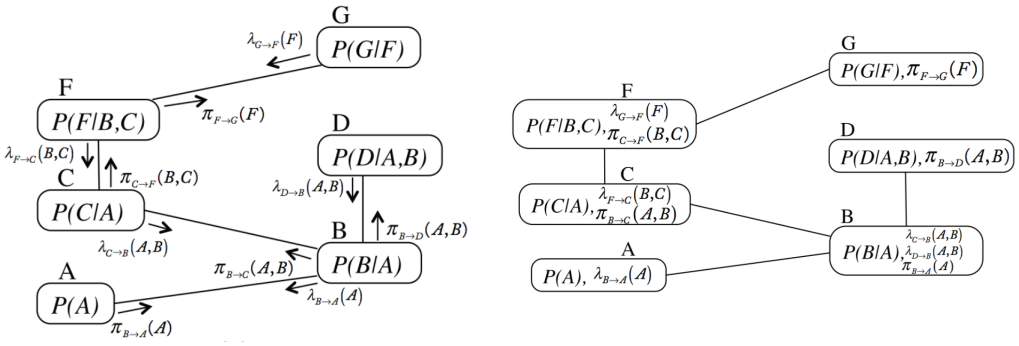


Figure 5: An example of Bucket Tree Elimination and the Augmented Bucket Tree [Dechter, 2013]

with a single cluster (every variable in a single bucket). These cluster-trees, also known as tree-decompositions, can compute messages in a similar way to Bucket-Tree Elimination with an algorithm called Cluster-Tree Elimination.

More generally, legal tree-decompositions can be generated from triangulations of the graphical model with an algorithm called Join-Tree Clustering. As we'll see in a later section, triangulations can also be used to compute variable orderings.

5 Finding an ordering for optimal treewidth

As we can tell by the way the complexity is calculated, finding an ordering that yields a graph with the minimum induced width is desired. This task has been shown to be NP-complete [Arnborg, 1985], notwithstanding, several greedy algorithms have been developed that achieve results that can be considered good enough for the cost of the calculation.

Among them we can highlight:

1. **Min-degree** (also Min-width): arranges the nodes in terms of their degree, this is, the number of neighbors. Selects the one with the largest degree, remove it from the graph and places it in position 1 of the ordering. Repeat this process until all vertices have been assigned. For a given node, min-degree can be interpreted as a lower bound of the treewidth of the subtree that includes it and the remaining nodes.

2. **Max-cardinality**: the first node is selected at random and placed in position n. Afterwards, we arrange the nodes in terms of the amount of neighbors that have been labeled, the one with the most neighbors is chosen and placed it in position n-1. We repeat this process until all vertices are assigned. Max-cardinality gives a lower bound on the treewidth of the tree that includes the given node and the previously selected ones.

3. **Min-fill**. This algorithm selects the nodes from 1 to n. We choose the node whose fill set (amount of edges that need to be added for the parent set to be fully connected) is the smallest. Once a node has been evaluated, we remove it from the graph along with its edges and connect the parent set. We repeat this process until all the vertices have been evaluated.

4. **Min-complexity**: nodes are arranged in terms of the complexity of solving variable elimination in the neighboring graph. This is, the product of the domain sizes of the neighbors to a given node. We select the node with the smallest cost and remove it from the graph.

In the case of two nodes having the same value for one of these metrics, the most common tie-breaking scheme is selecting one of them at random. Nevertheless, more sophisticated schemes could be developed such as using another metric or computing the same metric for the resulting graph after removing the evaluated node.

5.1 Advanced Greedy Algorithms for Optimal Treewidth

IGVO (Iterative Greedy Variable Ordering) [Kask et al., 2011] is one of the state-of-the-art schemes for generating pseudo-trees; it embeds one of the previous functions as a heuristic inside of a larger algorithm. IGVO, generates

iteratively pseudo-trees using a random tie-breaking model (it selects a variable with a probability proportional to its cost among the pool of lowest cost variables). This model aims to explore solutions that locally may be suboptimal but that may lead to a better global solution than the greedy one. Other characteristics that make this scheme interesting are the early termination of orderings of inferior quality, its optimized implementation when using the min-fill heuristic, the possibility of having multiple objective functions for the ordering and the possibility of parallelizing the process (for n threads we could potentially generate n times more pseudo-trees than sequentially) The results shown in [Kask et al., 2011] confirm the intuition as IGVO produces consistently better solutions than any of the heuristics previously presented by themselves.

5.2 Complete Algorithm for Treewidth

We can model the problem of finding the treewidth as a complete search problem (taking into account that it may not end due to its complexity). At each level, we would select a vertex of the graph yielding a search tree for each one of the possible starting nodes; an admissible heuristic can be given by min-width or max-cardinality since they offer lower bounds on the solutions rooted at each node.

In [Gogate and Dechter, 2004] Minor-min-width is introduced and evaluated. The minor G' of a graph G = (V, E) given nodes u, v ∈ V is obtained by contracting u and v into a single node w such that every neighbor of u and v in G is a neighbor of w in G'. The minor-min-width of a graph is the maximum min-width of all its possible minors formed by contracting any two neighbors u, v s.t. u is a minimum degree node in G and the intersection between neighbors of u and v is minimal. Since the treewidth of a graph is never less than the treewidth of one of its minors, MMW clearly gives a lower bound on the solution. The branch-and-bound algorithm described in the paper works as follows:

- The min-fill value of the graph (width of the ordering given by min-fill) is an upper bound on the graph's treewidth; the graph minor-min-width value, is a lower bound on the treewidth.

- If both values are equal, we return the value as the tree's treewidth.

Otherwise, we start a branch-and-bound search, pruning solutions whose heuristic is worse than the upper bound. The heuristic for a node is defined by two functions: g(s) which is the maximum degree of a node selected along the partial ordering ending in the node and h(s) the minor-min-width of the graph obtained by contracting the node u and any of its neighbors, v, with minimum degree in the neighborhood of the u. The heuristic function is f = max(g(s), h(s)). This is an admissible heuristic on the treewidth as it's calculated as the maximum of two valid lower bounds, minor-min-width and min-degree.

Once all the nodes have been either pruned or explored we have obtained the value of the treewidth. This algorithm is developed in an anytime fashion, such that, if stopped during the execution we obtain a lower bound on the graph's treewidth.

5.3 Pseudo-Trees

As we'll see later, to construct an AND/OR tree we need a valid *guiding spanning tree* that transforms the graphical model into a tree-like shape traversable by search algorithms.

One of the most common representations is that obtained by a depth-first search traversal of the model. The resulting tree is thus called DFS spanning tree of the model. There's a more broad set of trees which are legal spanning trees of the models called pseudo-trees. As defined in [Dechter 2013]:

Pseudo-tree: Given an undirected graph $\mathcal{G}$ = (V, E), a directed rooted tree $\mathcal{T}$ = (V, E') defined on all its nodes is a pseudo-tree if any arc in E which is not 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. Given a pseudo-tree $\mathcal{T}$ of G, the extended graph of G relative to $\mathcal{T}$ includes also the arcs in E' that are not in E. That is, the extended graph is defined as $G^{\mathcal{T}} = (V, E \cup E')$.

The bucket-tree induced by the execution of the bucket-elimination algorithm yields a valid pseudo-tree for the graphical model. Different orderings result in AND/OR Search Trees of different height or width, reason why, one of the approaches followed in order to generate good pseudo-trees consists on finding a good ordering with a good induced-width.

As mentioned in section 4.4, there's a relation between triangulations and legal tree-decompositions of a graphical model. Said triangulations also induce an ordering of the nodes in the graph that can be used to generate pseudo-trees. This property is used in the alternative approach to generate good orderings of a graph that we'll analyze in the following section.

5.4 Enumerating Minimal Triangulations

[Carmeli et al., 2017] tackles the task of finding a good ordering as an enumeration problem, this is, constructing all the valid guiding pseudo-trees. The amount of pseudo-trees for a given graphical model is exponential in the amount of variables but the algorithm described guarantees to generate decompositions in incremental polynomial time. This way, if we generate a large pool of pseudo-trees to choose from we may generate the optimal one, as well as various pseudo-trees whose characteristics can be a subject of study.

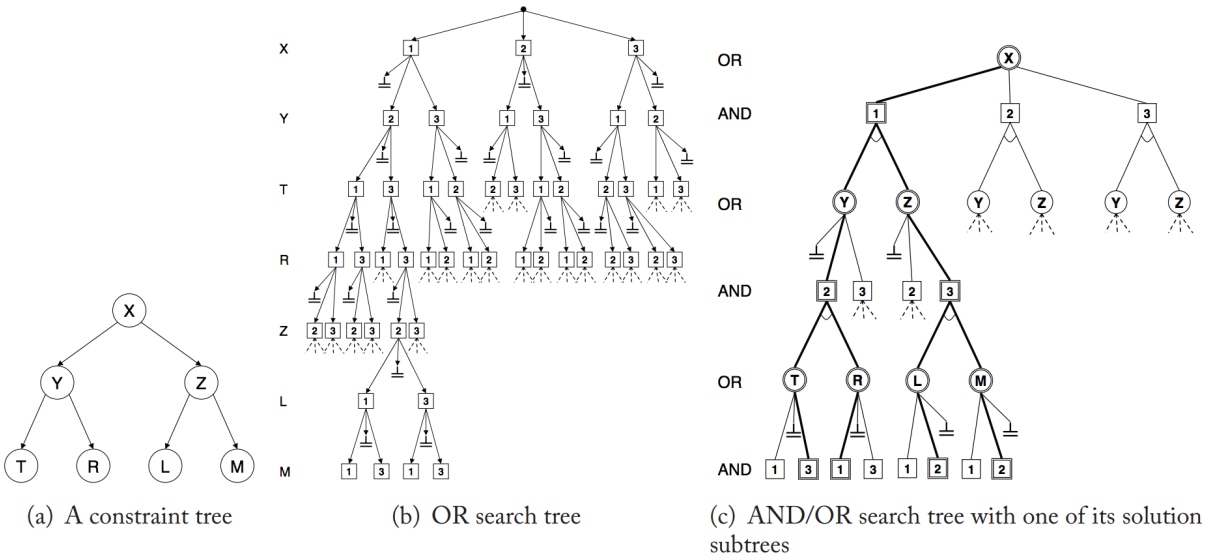
There's a set of preliminary concepts needed to understand the paper whose definition are also included in the paper:

- Given a graph G = (V, E), a **clique** $C \subseteq V$ is a subset of nodes C = {u | $\exists\ e_{u,v} \in E\ \forall$ u, v $\in C$} (every pair of nodes is connected in the graph). A maximal clique is a clique that's not contained in any other clique of the graph. We say that we saturate a set of nodes U, if we convert them in a clique, this is, we add all the necessary edges so that they are connected pair-wise.
- An **independent set of nodes** I of a graph G = (V, E), I = {u | $\nexists\ e_{u,v}\ \forall$ v $\in$ V}. In other words, there is not an edge that connects two nodes in the set. A maximal independent set is one that is not included in any other independent set.
- A set of nodes U is a **(u,v)-separator** if u and v are disconnected in $G_{|U}$ ($\nexists\ e_{u,v}$). A (u,v)-separator, S, is said to be minimal if no strict subset of the nodes in S is also a (u,v)-separator. We say a separator S_1 *crosses* another separator S_2 if S_1 is a (u,v)-separator for any u,v in S_2; otherwise they are called *parallel* separators.
- A graph is **chordal** if any cycle in the graph that includes 3 or more nodes has an edge between two non-consecutive nodes. A chordal graph is a graph whose minimal separators are cliques; therefore, a chordal graph has less minimal separators than vertices.
- A **triangulation** G' = (V, E') of a graph G = (V, E) is obtained by adding edges to G such that it becomes chordal. A triangulation is said to be minimal if no subset of E' makes G' chordal.
- A **tree decomposition** D = (V^t, E^t) of G = (V, E) is a pair (t, β), where t is a tree and $\beta : V^t \rightarrow 2^V$ is a function that maps every node of t (a bag) into a set of nodes of G, so that all of the following holds:
  1. Nodes are covered: for every node u $\in$ V there is a node v $\in V^t$ such that u $\in \beta$(v).
  2. Edges are covered: for every edge e $\in$ E there is a node v $\in V^t$ such that e $\subseteq \beta$(v).
  3. Junction-tree (or running-intersection) property: for all nodes u,v,w $\in V^t$, if v is on the path between u and w, then β(v) contains β(u) $\cap$ β(w).

The enumeration of orderings or proper tree decompositions is done through the existing relation between minimal triangulations of the graphical model whose induced width (and induced width of the pseudo-tree) is the size of the largest clique in the triangulations. It is decomposed in four phases:

-**Succinct Graph Representation** (SGR): an alternative representation of the given graphical model whose nodes are the minimal separators of the original graph (two nodes will be connected if the separators *cross* each other, call the function that computes this A_E). This representation allows us to calculate the Maximal Independent Sets of a graph without explicitly generating the necessary graph whose size might be exponential in the size of the original graph (which may already be large). The paper also describes a function A_V to enumerate the nodes in the SGR and. A_E is computed in polynomial time and A_V computes nodes with polynomial delay.

Maximal Independent Sets (MIS): once the SGR is constructed we can enumerate the maximal independent sets of the original graph in incremental polynomial time. This task defines a subroutine `Extend` that, given an independent set, computes a maximal independent set (this calculation can be guided by diverse heuristics as we'll see later). Each MIS is computed through the *extension* of a previously-computed MIS, J, in the direction of a vertex, v, of the SGR. This extension is computed by generating a new independent set as the subset of nodes in



(a) A constraint tree (b) OR search tree (c) AND/OR search tree with one of its solution subtrees

Figure 6: [Dechter, 2013]

J that are not connected to v in the SGR. This algorithm generates new nodes in the SGR when it has evaluated all of the MIS generated previously.

Minimal triangulations: there's a bijection between maximal independent sets and minimal triangulations of a graph that can be calculated in polynomial time. Given, I, a maximal independent set of G^{ms}, it corresponds to the set of pairwise-parallel minimal separators in h, where h is the graph obtained by saturating each separator S in I in the original graph G, $G_{[I]}$. Each set of nodes S is a clique in h and h is a minimal triangulation of the original graph.

- **Tree decompositions** or **Pseudo-tree**: given a minimal triangulation, its corresponding decompositions can be enumerated with polynomial delay. A minimal ordering (one that doesn't add any edges to the graph) can also be obtained by applying Maximum-Cardinality Search since it's guaranteed when applying it over a chordal graph.

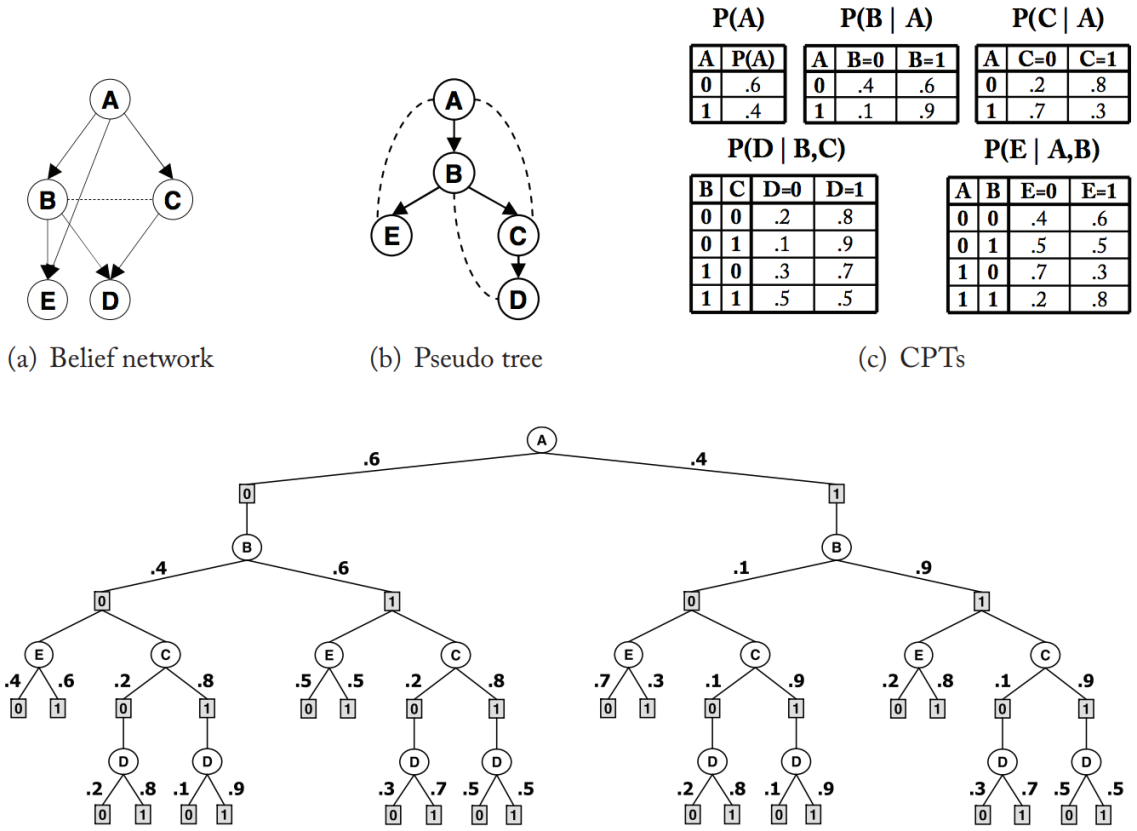
6 AND/OR Search Spaces

Given a graphical model and an ordering of the nodes we can, instead of constructing an induced graph, represent all the possible assignments of the different variables in the shape of a tree. The root of the tree being the first variable in the ordering, afterwards, a child for each possible value it can take and so on and so forth. With this representation, a path from the root to a leaf node would give us a full assignment of the variables. If we place in the edges the probabilities of a node given its ancestors, by multiplying all of them we will compute the probability of the assignment.

Most common queries would require traversing the whole tree, partly because the independencies present in the original model are ignored. The solution for capturing independencies in a tree-like form is available in AND/OR search spaces which also reduces effectively the size of tree.

6.1 AND/OR Search Trees

AND/OR trees receive their name from the two types of nodes they're made from, OR nodes represent the variables, X_i, and AND nodes, $\langle X_i, x_i \rangle$ or simply x_i, correspond to the assignment of a value to said variables. They are constructed according to a tree representation of the graphical model called *guiding spanning tree* which we will look into in this section. An ordinary search tree like the one shown before is called OR-tree.



(a) Belief network (b) Pseudo tree (c) CPTs

(d) Labeled AND/OR tree

Figure 7: [Dechter, 2013]

AND/OR Tree: Given a guiding spanning tree $\mathcal{T}$ of a graphical model $\mathcal{M} = \langle X, D, F, \otimes \rangle$, an AND/OR tree for a graphical model is constructed by alternating OR and AND nodes. The tree's root is an OR node for the root variable in $\mathcal{T}$. An AND node $\langle X_i, x_i \rangle$ is a child of an OR node X_i if $x_i \in D_i$. An OR node X_i is a child of an AND node $\langle X_j, x_j \rangle$ if X_i is a child of X_j in $\mathcal{T}$.

A full assignment to the variables in a graphical model takes the form a subtree in the AND/OR tree (a path for an OR-tree). For each OR node in the solution subtree, only one of its AND children will be in the solution (a single assignment per variable); every child of an AND node in the solution subtree belongs to the solution itself.

The only thing yet to define in the search space is the weights of the arcs. For the queries defined in section 3, the edges leaving AND nodes won't have an assigned weight and the OR nodes will have weights obtained from the CPTs. In particular, the weight of an edge going from the OR node X_i to the AND node $\langle X_i, x_i \rangle$ would correspond to the product of all the functions whose scope is fully assigned for the first time at node X_i. If no function fulfill this condition, its weight will be 1.

This model can be extended to other queries like solution counting on constraint satisfiability by changing the weights of the arcs.

6.2 AND/OR Search Graphs

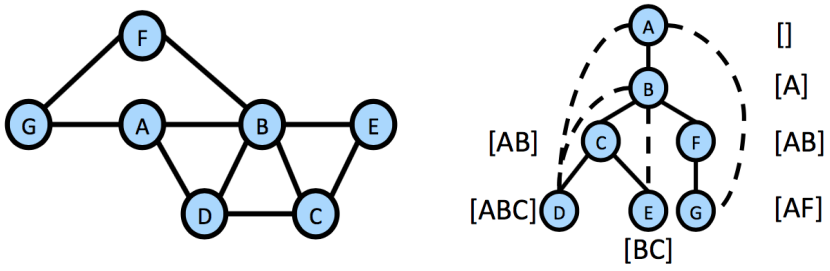
If we analyze a traditional path-finding problem, we can see that several paths may lead us to the same position in the graph and, although they may have different costs, the problems rooted at the end of both paths are equivalent. This is, optimally solving that subproblem is the same for both paths. If we're able to identify this subproblems, we can effectively reduce the search space for a faster search.

This same property can be observed in graphical models. If we detect a subtree whose solution depends solely on the assignment to its root, we can merge all the different problems. More generally, two subtrees (graphical models representations) are equivalent if they have the same set of solutions, each one of them with the same cost. More formally, they are conditioned models which root subproblems with the same cost. Below, a formal definition of the concepts from [Dechter, 2013].

Cost of an assignment, conditioned model. Given a graphical model $\mathcal{R} = \langle \mathbf{X}, \mathbf{D}, \mathbf{F}, \otimes \rangle$

1. The cost of a full assignment x = $(x_1, ..., x_n)$ is defined by c(x) = $\otimes_{f \in F} f(x_f)$. The cost of a partial assignment y, over $Y \subseteq X$ is the combination of all the functions whose scopes are included in Y (denoted F_y) evaluated at the assigned values. Namely, c(y) = $\otimes_{f \in F_Y} f(y_f)$.
2. The graphical model conditioned on Y = y is $\mathcal{M}|_y = \langle X - Y, D|_{X-Y}, F|_y, \otimes \rangle$ where $F|_y = \{f|_{Y=y},\ f \in F\}$.

In practice, finding subproblems that only depend on its root is more difficult that in a path-finding problem as the dependencies of the variables in the subtree and the ones in partial assignment leading to the node have to be analyzed. For this purpose, we define merge operations that depend only on the graph properties of the graphical model.



(a) A graphical model.

(b) Context-labeled pseudo-tree.

Figure 8: [Lam, 2017]

The set of ancestors that condition the problem rooted at a variable is denominated **context**. In order to define it more formally we introduce the definitions of two sets of nodes:

1. Parents, pa_{X_i}: Given an OR node labeled as X_i in an AND/OR tree, the set of parents of the node is defined as the ancestors in of X_i that are connected to X_i or to any of its descendants in the pseudo-tree of the graphical model. This is called OR-context of a node.
2. Parents-separators, pas_{X_i}: Given an OR node, X_i, or an AND node, $\langle X_i, x_i \rangle$, the set of parent-separators is defined as $pas_{X_i} = pa_{X_i} \cup X_i$. This is called AND-context of a node.

Given two partial paths, path(n_1) and path(n_2), whose last node in the path is n_1 and n_2 respectively, and we define val($path_i$)[X] as the value of the variables in the set X in the $path_i$, we can define AND merging and OR merging as pointed out in [Dechter, 2013]:

- AND-merging: If n_1 and n_2 are AND nodes annotated by $\langle X_i, x_i \rangle$ and

  $$val(path(n_1))[pas_{X_i}] = val(path(n_2))[pas_{X_i}]$$

  then the AND/OR search subtrees can rooted at n_1 and n_2 can be merged.
- OR-merging: 1. If n_1 and n_2 are OR nodes annotated by $\langle X_i, x_i \rangle$ and

  $$val(path(n_1))[pa_{X_i}] = val(path(n_2))[pa_{X_i}]$$

  then the AND/OR search subtress can rooted at n_1 and n_2 can be merged.

Consequently, we can define a **Context-minimal AND/OR search graph** as the AND/OR search graph that can't be subject to any AND-merging or OR-merging with respect to its guiding pseudo-tree $\mathcal{T}$. In other words, a graph where each context only appears once.

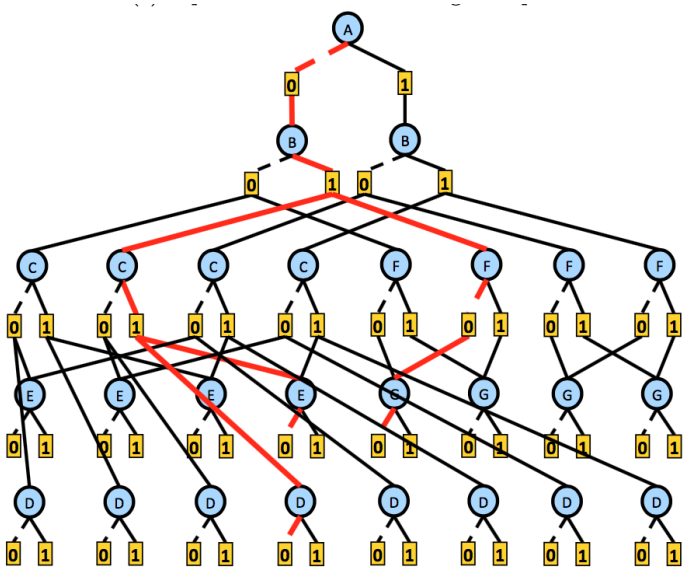


Figure 9: Context-minimal AND/OR Graph for the graphical model in Figure 8. [Lam, 2017]

We can now bound the space complexity of a Context-minimal AND/OR Search graph with respect to its guiding pseudo-tree and the context's size in the graph. The width of a pseudo-tree is the largest size for a context (set of *parents*) of a variable in the pseudo-tree.

Each variable appears only once per possible assignment to its context. The amount of combinations is bounded by the largest scope size of variable in the context that, as stated before, is bounded by the induced width of the pseudo tree. Since there are n variables in the tree we get $O(n \cdot k^w)$.

We can compare this size with the one of a classical OR search tree where. An OR-tree would have n levels (one per variable), and a maximum branching factor of k a size, making its size bounded by $O(n \cdot k^{n-1})$. If we observe the biggest OR context of a variable would include every other variable ($w \leq n-1$), we see that the previous expression tightens its size exponentially.

6.3 Searching an AND/OR tree or graph

We will now proceed to explain how the most common queries (described in section 3) can be computed through traversing an AND/OR space. Computing the cost of a solution in a search tree where the solutions lay in the leaves of the tree consists on adding up the cost of the different edges. In a similar way, the cost of a solution subtree is computed according to the combination operators of each reasoning problem. The combination operator will be applied will be applied over the AND nodes and the marginalization operator over the OR nodes. In case of evidence $X_i = x_i$, we will only traverse the subtree rooted by the AND node $\langle X_i, x_i \rangle$.

Belief updating: product over AND nodes and summation over OR-nodes. This is, the value (probability for Bayesian Networks) of an AND node is the product of the value of its children and the value of an OR-node the summation of the value of its children weighted by the arc that joins them.

The algorithm AO-Belief-updating, performs a DFS search of the graph keeping in a stack the fringe of the search (nodes that have been open but not yet expanded) which is initialized with the root of the tree. We differentiate two processes inside of the execution loop:

1. EXPAND: In this step, we initialize the children of the current node, popped from the top of the stack. AND nodes, $\langle X_i, x_i \rangle$ will be initialized with the value of the product of the functions of X_i OR-context. In the case of a OR node, it will be initialized to 0. We then push the successors of current node to the top of the stack.
2. PROPAGATE: When a node has an empty set of successors, we propagate this information up the tree. For an AND node, we have encountered a leaf node, so we sum the node's value to its parent's value. For an OR

node, we multiply the value of its parent by the node's value; if this is zero, we will remove the siblings from the fringe since we've found a dead end (an AND node for which not all of its children can be solved). We will continue this process with the parent of the current node, until we find a non-empty set of successors. The execution will end once we evaluate the root node in the propagate step.

Once the execution terminates, we can find in the first node the updated belief of the root variable. After processing the root node, we can also find the probability of evidence. In practice, the implementation of the AND/OR graph is in the form of a cache. Each time we find the value of a node (in the *propagate* step), we save its value in memory. This way, each time we come across the same context in the *expand* state we can retrieve its value directly without adding its successors to the fringe and we can directly propagate its value up.

The rest of the queries can be implemented similarly by simply changing the combination and marginalization operator in the EXPAND and PROPAGATE steps.

6.4 Complexity

The execution of the algorithm in AND/OR trees (without caching any values) has space complexity linear in the number of variables in the problem O (n), as it's the maximum size the fringe stack can get to be. In terms of time, the computation is dominated by the propagation of the values since we have to compute the value of each node. The amount of nodes is bounded by $O(n \cdot k^h)$, where h is the height of the pseudo-tree guiding the graphical model. We can establish a relation between the height and the induced width of the graphical model; the height of the tree is bounded by the induced width according to the formula $h \leq w \cdot log\ n$. This can be proven as follows ([Hasfsteinsson, 1991]):

Given a tree decomposition of the graphical model (where each one of the nodes represents a superset of the scope of one or more functions), we can find a node that divides the nodes in two groups of approximately the same size. This node will be the root of our pseudo-tree. We repeat the process for each branch until they are only composed by a node yielding a tree with height log n (where n is the amount of nodes in the tree decomposition). Each node in the tree decomposition formed by more than one variable needs to be decomposed in a chain of maximum length w (w bounds the maximum scope size of a function), giving a pseudo-tree of a total height bounded by $w \cdot log\ n$.

For AND/OR graphs, as we've mentioned before, there's a need to cache the values of the subproblems that have already been solved. For a single variable, we can have k^w possible assignments of its context, giving us a space complexity of $O(n \cdot k^w)$. In practice, there are certain values that will never be consulted in the cache called *dead-caches* [Allen and Darwiche, 2003]. These values have as context every variable that appears in the path from the root node of the pseudo-tree to them. In Figure X, A, B, C, D and F are dead-caches.

This brings us back to the initial problem of space and time complexities in the induced width of the graphical model, but allows us to describe a generalization of this algorithms, called AO(i). The argument i describes the maximum context size for which we will cache results. In one end of the spectrum we can find AO(0), which is equivalent to AOT, where no functions are cached; in the other end we find AO(w), which is the equivalent of AOG where every function is cached. Consequently, we have a tool to control the balance between the time and space complexities.

7 Approximation algorithms

In the last section we've talked about an alternative for the cases where the induced width of the problem is too big to generate the functions completely. One of them, suffers in terms of time and the other presents the same problems in memory as Bucket Elimination. In the lookout for a good time and memory complexities, approximation algorithms were born that tackle the complexity of computing the value for a message.

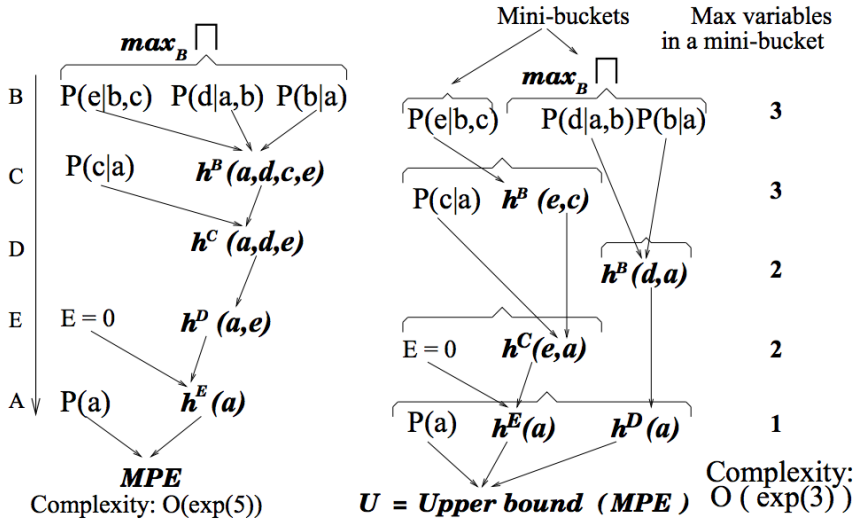


Figure 10: Bucket Elimination vs Mini-Bucket Elimination [Dechter and Rish, 2003]

7.1 Power sum

Firstly, we will define an auxiliary operator that will help us later define the relaxation of the problem as explained in [Liu and Ihler, 2011]. This operator is called power sum and is defined as[2]:

$$\varepsilon_x^w f(x) = (\sum f(x)^{\frac{1}{w}})^w$$

There are three values of w that will be considered, w = 1 which gives us the summation $\sum f(x)$, $w \to 0^+$ which gives us $\max f(x)$ and $w \to 0^-$, gives us $\min f(x)$.

Theorem: **Generalization of Holder's inequality**: Assume that r $\in (0, \infty)$ and $p_1, p_2, ..., p_n \in (0, \infty]$ such that $\sum_{k=1}^n \frac{1}{p_k} = \frac{1}{r}$ then, for all measurable functions $f_1, ..., f_n$ defined on a probability space (S, Σ, μ),

$$\left\| \prod_{k=1}^n f_k \right\|_r \leq \prod_{k=1}^n \|f_k\|_{p_k}.$$

By substituting in this inequality for the power sum expression for $\frac{1}{p_k} = w_k$ and $\frac{1}{r} = w$ we get:

$$\varepsilon_x^w \prod_{i=1}^r f(x) \leq \prod_{i=1}^r \varepsilon_x^{w_k} f(x)$$

7.2 Mini-Buckets

Mini-buckets are created with the intention of reducing the computation and space needed to calculate the messages in each bucket. Each function is assigned to one of the mini-buckets that the bucket is subdivided in. The messages will be generated independently for each mini-bucket generating thus an approximation of the initial message. The calculations required vary across the different queries as we'll see in the following subsections.

The mini-bucket partitioning is characterized by two different parameters, i the maximum scope size of a function in the mini-bucket and m, the number of functions in the mini-bucket. The two parameters are independent but not all combinations yield a possible partitioning of the functions. In specific, any combination with a value of i

[2]In the paper, the power sum is expressed as Σ_x^w but it has been substituted as it may be confused with a regular sum with w as upper value for x.

smaller than the maximum scope size of a function is not legal. As for a given i-m combination there can be several valid partitionings, we'll see later how to select the appropriate bucket for each one of the functions.

Mini-Bucket approximation is not monotonic with respect to the i-bound (we can't guarantee that the approximation will be better with a larger i-bound). More generally, we can only guarantee one approximation will yield better results if and only if one of the partitionings is a refinement of the other.

Refinement [Dechter and Rish, 2003]: Given two partitionings Q' and Q" over the same set of elements, Q' is a refinement of Q" if and only if for every set A ∈ Q' there exists a set B ∈ Q" such that A ⊆ B.

7.3 MPE

Given an MPE task defined over a set of variables X, we have to calculate a message h such that:

$$\lambda_i = \max_X \prod_{f \in b_i} f \quad s.t. \quad scope(f) \subseteq X$$

where f represents both messages and functions in the bucket. In other words, finding the assignment that maximizes the function λ_i, which consists of the computation of k^n values, where $n = \|X\|$.

If we subdivide each bucket, b_i into n mini-buckets, mb_i, we can write the equivalent expression:

$$\lambda_i = \max_X \prod_i^n \prod_{f \in mb_i} f \quad s.t. \quad scope(f) \subseteq X$$

We can derive a new inequality from the following reasoning: Given any two non-negative functions F(x) and G(x)

$$\max_x F(x)G(x) \leq \max_x [F(x) \max_x G(x)]$$

Since $\max_x G(x)$ is a constant we can take it out of the maximization giving us the expression we're looking for

$$\max_x F(x)G(x) \leq \max_x F(x) \cdot \max_x G(x)$$

We can apply this reasoning to the formula of λ_i, since the expressions $f(X_i)$ are probabilities (non-negative functions), moving the maximization to every mini-bucket getting:

$$\lambda_i \leq g_i = \prod_i^n \max_X \prod_{f \in mb_i} f$$

Maximizing over every mini-bucket needs less computation g_i values (as we maximize over a subset of X) and gives us an upper bound on λ_i. This is equivalent to the power sum of the mini buckets with all the weights equal to 0.

The execution of MPE for its mini-bucket version is divided as well into two phases, the backward phase where a message is generated independently for each mini-bucket from the last bucket ordering to the first which gives us an upper bound, U, of the probability of the MPE. In the forward phase, the buckets are treated as if they weren't divided, this is, we maximize over the product of all the functions independently of their mini-bucket. This way, we obtain the most-probable assignment for the relaxed modeled we've constructed with mini-buckets. We can also obtain a lower bound, L, of the MPE query as the probability of the MPE-tuple found, since the assignment we've found is not necessarily maximal (generally, the probability of any suboptimal assignment gives a lower bound).

This execution finds the joint probability of the assignment and the evidence ($\max_X P(X, e)$) rather than the probability conditioned on the evidence. The task is equivalent, since $\frac{P(X,e)}{P(e)} = P(X|e)$ and P(e) is a constant that can be ignored in a optimization task. If we want the actual value for the MPE, we need the probability of the evidence to calculate $P(X|e)$ which is not available as it requires an exact computation. Normally this is not a problem, as most of the time we're interested in the tuple rather than in the value of the MPE.

To summarize, we generate an interval bounded by the following expression:

$$\frac{L}{P(e)} \leq P(x_i|e) \leq \frac{U}{P(e)}$$

A couple remarks worth mentioning are:

- If $\hat{P}(e) = U$ we get a trivial upper bound $MPE \leq 1$.
- If $U = L$ we've found the exact value of the MPE.
- The quality of the bounds found can be calculated by their ratio.
- For an unlikely probability of evidence the interval would become really large but this doesn't change the ratio.

7.4 Belief Updating

These mini-bucket approximation can be extended to the other tasks/queries presented in section 3. A similar reasoning can be followed for belief updating by modifying the computation of the messages as shown below:

$$\lambda_i = \sum_{X_p} \prod_{i}^{n} \prod_{f \in mb_i} f$$

If we move the summation into each one of the mini-buckets, we would obtain

$$\lambda_i \leq g_i = \prod_{i}^{n} \sum_{X_p} \prod_{f \in mb_i} f$$

This is clearly a valid upper bound where we are bounding each mini-bucket with its sum over X_p. A tighter upper bound for every mini-bucket would be its maximum, we can find a valid inequality if we reformulate the initial expression to:

$$\lambda_i = \sum_{X_p} (\prod_{f \in mb_1} f) \cdot (\prod_{mb_j \neq mb_1} \prod_{f \in mb_j} f)$$

and then we maximize over each mini-bucket, which becomes a constant over X_p that can now go out of the sum:

$$\lambda_i \leq g_i = (\sum_{X_p} \prod_{f \in mb_1} f) \cdot \prod_{mb_j \neq mb_1} \max_{X_p} \prod_{f \in mb_j} f$$

Therefore we can obtain an upper bound on the message generated by summing over the bucket variable in one of the mini-buckets and maximizing over the rest. This is equivalent to the power sum of the mini buckets with one weight equal to 1 and the rest equal to 0. We can obtain a lower bound by minimizing or if we want a closer estimate of the belief with no guarantees we can use the mean over X_p in the rest of the mini-buckets.

The same normalization explained for mbe-MPE needs to be present for mbe-bel as we're computing the joint probability of the marginals $P(x_i, e)$ not the marginals themselves $P(x_i|e)$. In this case, we would need to divide the probabilities found by the lower bound and upper bound of the probability of the evidence as found by mbe-bel-min and mbe-bel-max respectively. This normally generates a really weak bound for the beliefs, even more in cases of rare evidence due to accumulated error.

$$\frac{L(P(x_i, e))}{U(P(e))} \leq P(x_i|e) \leq \frac{U(P(x_i, e))}{L(P(e))}$$

7.5 Marginal MAP

The definition of Marginal MAP restricts the possible orderings to those where the hypothesized variable appear first; these variables are eliminated by summation and the rest are eliminated by the maximization operator. It could be said that Marginal MAP consists of applying Belief Updating over the first variables and MPE over the rest.

It we translate this idea to the power sum we would have those mini-buckets from a hypothesized variable ($X_i \notin A$) with a weight equal to 1 and the rest equal to 0, yielding the following expression:

$$\lambda_i = \prod_{i=1}^{r} \varepsilon_x^{w_k} f \quad s.t. \begin{cases} 0, & \text{if } X_i \in A \\ 1, & \text{otherwise} \end{cases}$$

As in the previous cases, MBE-map computes an upper bound on the map value in the backward phase and the tuple in the forward phase. The probability of the assignment found is a lower bound on the real value but calculating this value for MAP is not a simple task as we can't use the values computed by MBE in the forward phase as for the summation buckets the values computed are upper bounds rather than lower bounds. A possibility would be computing the values for those buckets with MBE-bel-min and then proceed as in MPE calculating the probability of the assignment found. In any case, most of the time we're looking for the tuple rather than for its probability, therefore this is a minor issue for the implementation.

7.6 Weighted Mini-Bucket Elimination & Complexity

The definition of the power sum operator allows us to define a generalization of the Mini-Bucket Elimination algorithm for all the different queries described before. The pseudo-code for the algorithm is shown below:

ALGORITHM Weighted-MBE

Input: A Bayesian network B = (X, D, P, $\prod$), an ordering d = (X_1, X_2, ..., X_n), evidence E, W = {w_1, w_2, ..., w_n} the power-sum weight associated with each variable in X, the (i,m) parameters for the mini-bucket partitioning and, optionally, a set of hypothesized variables A = (A_1, A_2, ..., A_p).

Denote:
- B_i the bucket corresponding to X_i;
- mb_{ij} as the *jth* mini-bucket in B_i;
- nearest(F_i, X_j) as the bucket B_k, s.t. X_k is the first variable in S_i that appears before X_j in d;
- Ψ_{ij} as the product of the functions in mb_{ij};

Partition input functions into buckets

Backward: process buckets in inverse order, generating messages $\lambda_i j$:

```
1. for i = n to 2
2.    Generate a legal (i,m)-partitioning for the functions in B_i generating mini-buckets, mb_ij
3.    for mb_ij in B_i
4.      if X_i in E
5.        for F_j = {λ_1, λ_2, ..., λ_p and Ψ_i} in mb_ij
6.          assign X_i = x_i to F_j
7.          place the function, F_j, in nearest(F_j, X_i)
8.      else
9.        //Compute a message as the power-sum of the functions inside of each bucket
10.       λ_ij ← ε_{X_i}^{w_i} Ψ_ij ∏_{k=1}^{p} λ_k
11.       place λ_ij in nearest(λ_ij, X_i)
```

Calculate P(E), map or mpe value with the content in B_1
If the task is MPE or MAP, **Forward:** Generate the value and assignment for the MPE/MAP tuple.

```
12. k ← amount of variables to maximize over (n for MPE, |A| for MAP)
13. for i = 1 to k:
14.   Assign the variables evaluated before to λ_j where they appear
15.   x_i^o = argmax_{x_i} Ψ_i ∏_{j=1}^{p} λ_j
16. return mpe/map assignment, x^0 = {x_1^o, x_2^o, ..., x_k^o}
```

The process followed in WMBE is really similar to that in the original Bucket-Elimination algorithm. It starts by assigning each input function to their corresponding bucket; then, we process the buckets in inverse order as part of the backward phase of the algorithm. We divide the bucket into mini-buckets that will be processed independently to one another generating a message for each one of them. In the case that the bucket is not part of the evidence, we will generate a message (line 10) according to the weight assigned to the bucket. This line shows the generalized version of the mini-bucket elimination as the power-sum computation. It's in this point that the execution will be different depending on the query.

Once we've processed all the buckets and mini-buckets. By processing the first bucket in the ordering we will generate the P(E), map or mpe correspondingly. If the query requires to generate a tuple, we will proceed with the forward phase. The assignments are computed exactly as in BE-mpe and BE-map, this is, without taking into account the subdivision in mini-buckets.

The complexity of the mini-bucket algorithms is easily characterized with the (i, m) parameters. Given a valid (i,m) partitioning, r probability functions given as input, with k bounding the domain size of its variables, its space and time complexity is $O(r \cdot k^i)$ as the maximum variables to be processed in a bucket to generate a message is i which is also equal to the scope of the largest function that's stored. For the special case of $m = 1$, we get $O(r \cdot k^{|S|})$ where S is the largest scope of the input functions.

7.7 As an anytime scheme

We can convert the mini-bucket computation into an anytime scheme as follows. Each time we compute a solution, we increase the i-bound parameter. Our stopping condition can be the ratio between the bounds or, if we calculate the probability of evidence, the absolute difference between the bounds. Two important remarks to be made are: 1) It's not guaranteed to terminate as i-bound needed to achieve it may be larger than the one the memory available can allow. 2) For an i-bound equal to the induced width, it's equivalent to executing Bucket Elimination.

7.8 Partitioning heuristics

As mentioned in the introduction to this section, for each bucket there are multiple mini-bucket configurations valid for each bucket that fulfills the (i,m) parameters. Due to this, rather than rely on a random assignment of the functions two schemes were developed to order them:

1. **Scope-based Partitioning**: starting with every function in a mini-bucket, we order them in decreasing order of scope size (arity). Afterwards, we go over the mini-buckets in this order and merge every bucket with the one with largest arity so that the (i, m) partitioning is still valid. generated without considering the content of the functions. As we mentioned in section 6.2, there are no guarantees that a partitioning with less buckets will be a better approximation unless one is a refinement of the other.
2. **Content-based Partitioning**: as it names points out, this heuristic focuses on the information included in the functions. It will try to reduce the error between the function computed by the mini-buckets and the original bucket function. The error measure can be any available (absolute error, relative error, log relative error, max log relative error...). Finding the optimal partitioning is computationally hard so the merging is done in a greedy manner. We will select merging the two mini-buckets that minimize the error selected and repeat this process until we can't merge anymore; as we are computing refinements of a mini-bucket, the approximation will improve with each merge even though its optimality is not guaranteed.

8 Mini-Bucket-guided Search for optimization tasks

To model a problem as heuristic search, we need a function that computes a lower bound or an upper bound on the optimal cost of the solution rooted at a given node. In graphical models we can distinguish between: dynamic heuristics[3] which are computed during search and static heuristics which are computed before and independently of the search task; an example of the later is the one given by the mini-bucket evaluation of the nodes. Given this heuristic, we can find better solutions than the one generated with MBE-mpe which is simply a greedy assignment

[3]The use of dynamic heuristics in AND/OR graphs is part of [Lam, 2017]

of the variables leading often to a suboptimal solution. Later, we will describe the adaptations of depth-first Branch-and-Bound and Best-first Search on graphical models.

8.1 Mini-bucket heuristic

In the present section we introduce the use of the mini-bucket computations as a valid heuristic for optimization problems as described in [Kask and Dechter, 2001].

Given a partial assignment over variables $Z = \{X_1 = x_1, X_2 = x_2, ..., X_p = x_p\}$ we define the probability of the best assignment, f^*, that includes it as as the maximization over the variables that haven't been assigned $Y = \{X_{p+1}, X_{p+2}, ..., X_n\}$.

$$f^*(Z) = \max_Y P(Z \cup Y)$$

The product of the functions that are fully assigned is denoted as $g(Z)$. The maximization over the product of the ones that include at least one of the variables that hasn't been assigned is labeled $h^*(Z)$, such that:

$$f^*(Z) = g(Z) \cdot h^*(Z)$$

As we know, the functions needed to compute $h^*(Z)$ are generated by the Bucket Elimination algorithm. We also know, that if we apply the mini-bucket algorithm we obtain an upper bound of these functions as a result, which is denoted $h(Z)$. Thus we can state that:

$$f(Z) = g(Z) \cdot h(Z) \geq f^*(Z)$$

If we denote Z_t as set of variables including all variables from 1 to t, the $h(Z_t)$ function is a product of all input probability functions that reside in the buckets $P_{1,t}$ and messages residing in buckets from 1 to t generated by buckets from t+1 to n, $h_{1,t}^{t+1,n}$. We will abuse notation by denoting P_i as the product of all probability functions in bucket i, $h_i^{a,b}$ as the product of all messages in bucket i and $h_{a,b}^i$ as the messages generated by bucket i. By using this notation we can say that:

$$g(Z_t) = P_{1,t}$$
$$h(Z_t) = h_{1,t}^{t+1,n}$$

We can update this functions recursively as follows:

$$g(Z_t) = g(Z_{t-1}) \cdot P_t$$
$$h(Z_t) = h(Z_{t-1}) \cdot \frac{h_t^{t+1,n}}{h_{1,t}^t}$$

In other words, we multiply g by the input probability functions that reside in bucket X_t; we multiply h by the messages that reside in X_t which are generated by a later bucket and divide by the messages generated by X_t that belong to the buckets 1 to t.

The resulting heuristic function is both admissible (provides always an upper bound on the optimal solution) and monotonic ($f(Z_t) \geq f(Z_{t+1})$) which makes it more precise for deeper nodes in the tree. A proof can be found in [Kask and Dechter, 2001] page 17-18.

8.2 Depth-first Branch-and-Bound

Depth-first Branch-and-Bound [Kask and Dechter, 2001] keeps the value of the best solution found so far, and while traversing the graph in a depth-first manner if a partial solution has an heuristic value (f) smaller than the one of the solution found, we prune the tree at that point as any solution rooted at that subtree is worse. This algorithm can be run in an anytime manner since at any point in time in the execution we have a valid solution for the task and if we allow it to run for longer this solution can be improved. With an admissible heuristic Depth-first Branch-and-Bound finds the optimal solution in an exponential time using only linear memory.

The algorithm is parameterized by the i-bound selected for the mini-bucket message generation and thus denoted as BBMB(i). Its execution proceeds as follows:

ALGORITHM BBMB(i)
Input: A reasoning problem $\mathcal{P}$, an ordering $d = \{X_1, X_2, ..., X_n\}$ and a time bound t
STEP 1: Compute the backward phase of WMBE(i) (as if for MPE) along the given ordering.
STEP 2: Search the graph.
Cost of the best solution to $-\infty$, n = X_1.
EXPAND(n): Compute the heuristic value (f) for all legal assignments of n. If any of them has a smaller value than the cost of the best solution, removed them from the legal values. Forward(n).
FORWARD(n): If n has no legal children, backtrack. Otherwise, if n corresponds to the last variable record the best f value as the cost of the best solution. If n corresponds to a different variable, EXPAND(best_child).
BACKTRACK(n): If n is the root node, return the best solution. Else, set n = parent(n).

8.3 Best-first Search

Since we have an admissible heuristic function for the graph, we can also perform other types of search. One of the most famous type of algorithms are those who expand first the most-promising node of the fringe of the graph (nodes that have been opened but haven't been expanded yet). A^*, keeps track of a sorted list of the fringe nodes, opening at each step the node with the best evaluation function (f). Each node is associated with a partial assignment (the values assigned to the variables appearing before it in the ordering). The optimal solution is found when a goal node is expanded, in our case, a leaf node (since all the solutions lay at the same height). This algorithm expands the least amount of nodes for reaching this solution but requires memory and time exponential in the height of the tree searched. A negative remark for this algorithm is that it can't be run in an anytime fashion.

The pseudo code of the algorithm BFMB(i) as presented in [Kask and Dechter, 2001]:

ALGORITHM BFMB(i)
Input: A reasoning problem $\mathcal{P}$, an ordering $d = \{X_1, X_2, ..., X_n\}$ and a time bound t
STEP 1: Compute the backward phase of WMBE(i) (as if for MPE) along the given ordering.
STEP 2: Search the graph.
FRINGE = X_1
Select the node in the fringe with the best f.
If n corresponds to an assignment of the last variable, $X_n = x_n$, return the full assignment and the value of the solution.
Otherwise, add to the fringe all of its legal children, value assignments to the next variable X_{i+1} Repeat.

8.4 Searching AND/OR graphs

The previous algorithms can be easily expanded to AND/OR graphs., since each legal assignment for a variable is represented by an AND node $\langle X_i, x_i \rangle$, child of its parent OR node $\langle X_i \rangle$. There'd be a needed modification in the forward step, since we would expand all the OR nodes that are children to the best assignment. In the following sections we will present the pseudocode for both AND/OR Branch-and-Bound and AND/OR Best First Search [Marinescu and Dechter, 2009].

As pointed when we first introduced AND/OR space, the dimensions of the search space is much smaller in an AND/OR tree or graph than in a regular tree which intuitively will help us doing a more efficient (less nodes will be open for finding a solution of the same quality) and effective search (if the algorithm is anytime, we take shorter to find a solution so we can expect a sooner improvement in it).

An important concept to be defined prior to the introduction of this algorithms is that of *partial solution tree*.

Partial solution tree [Marinescu and Dechter, 2009]. A partial solution tree T' of a context minimal AND/OR search graph $C_{\mathcal{T}}(\mathcal{R})$ is a subtree which: (1) contains the root node s of $C_{\mathcal{T}}(\mathcal{R})$; (2) if n in T' is an OR node then it contains one of its AND child nodes in $C_{\mathcal{T}}(\mathcal{R})$, and if n is an AND node it contains all its OR children in $C_{\mathcal{T}}(\mathcal{R})$. A node in T' is called a tip node if it has no children in T'. A tip node is either a terminal node (if it has no children in $C_{\mathcal{T}}(\mathcal{R})$), or a non-terminal node (if it has children in $C_{\mathcal{T}}(\mathcal{R})$).

In this definition $C_{\mathcal{T}}(\mathcal{R})$ denotes the context minimal AND/OR search graph of a graphical model $\mathcal{R}$ constructed using the pseudo-tree $\mathcal{T}$.

8.5 AND/OR Branch-and-Bound

As explained before, Branch-and-Bound keeps an upper bound or a lower bound (for minimization or maximization tasks, correspondingly) of the optimal solution and prunes those subtrees whose evaluation function is worse than the upper bound found and explores those who are better, updating the bounds if a complete solution found is better than the bound.

Given a node evaluation function h(n) (in our case the mini-bucket heuristic), we can calculate the heuristic evaluation function f of a partial solution tree T recursively as follows:

- If T consists of a single node n, f(T) = h(n)

- If the tree is rooted at n an OR node with a child AND node m, f(T) = w(n,m) + f(T_m), where T_m is the partial solution tree rooted at m.

- If n is an AND node with OR nodes M = {m_1, m_2, ..., m_n} as children, h(n) = $\sum_i f(T_{m_i})$.

Below we can see the detailed pseudo-code for AND/OR Branch-and-Bound that includes caching for the AND nodes:

Algorithm AND/OR Branch-and-Bound

```
1. v(s) = ∞
2. ST(s) = ∅
3. OPEN = {s}
4. Initialize empty Cache
5. while OPEN ≠ ∅:
6.  n = OPEN.pop()
7.  if n is an OR node:
8.    for all AND children m_i:
9.      w(n, m_i) = ∑ f(assn(π_n))
10.     succ(n) = succ(n) ∪ {m_i}
11. else if n is an AND node:
12.   context = assn(π_n)[pas_n]
13.   if context in Cache:
14.     v(n) = Cache[context].val
15.     ST(n) = Cache[context].assn
16.     cached = True
17.   for all OR ancestors m_i:
18.     if f(T_{m_i}) ≤ v(s):
19.       deadend = True
20.       break
21.   if not deadend and not cached:
22.     for all OR children m_i:
23.       v(m_i) = ∞
24.       ST(m_i) = ∅
25.       succ(n) = succ(n) ∪ {m_i}
26.   else if deadend:
27.     p = parent(n)
28.     succ(p) = succ(p) - {n}
29.
30. OPEN.push(succ(n))
31. while succ(n) == ∅:
32.   p = parent(n)
33.   if n is an OR node:
34.     if n is X_1:
35.       return v(n), ST(n)
36.
37.     v(p) = v(p) + v(n)
38.     ST(p) = ST(p) ∪ ST(n)
39.   else if n is an AND node:
40.     context = assn(π_n)[pas_n]
41.     Cache[context].val = v(n)
42.     Cache[context].assn = ST(n)
43.
44.     if v(p) < (v(n) + w(p, n)):
45.       v(p) = v(n) + w(p, n)
46.       ST(p) = ST(n) + {(X_n, x_n)}
47.
48.     succ(p) = succ(p) - {n}
49.     n = p
```

The algorithm is similar to that described in section 8.2, in the way we expand the nodes and propagate the values found but includes the necessary lines to prune those subtrees who offer worse solutions than the one found and to continue traversing the tree until every node has been explored or pruned. In concrete, the code provided serves

for a maximization task but few changes it could be adapted to a minimization task.

The algorithm keeps track of the value of the best solution found so far, v(s), and the solution subtree that provides it, ST(s). We keep track of a list of the nodes in the fringe of the search in a stack called OPEN. As in the code described in section 8.2, AOBB can be described in two independent tasks:

EXPAND lines 6-30: In this section of the code we open a node from the stack. For an OR node (lines 7-10), we initialize the value for the edges going to its children as the sum of the functions that become fully assigned at that level and calculate their cost for the assignment that leads to the current node, π_n. For an AND node (lines 11-28), we first check if it has been solved before and thus exists in the cache (we do so by calculating its context, the value assignment to its parent-separators). If it exists in the cache, we retrieve the value and assignment of the solution rooted at that node. We also calculate the heuristic value of all the OR nodes (that root a partial solution tree) in the path from the root to the node. If any of them, has a value worse than the recorded best solution, we stop expanding the current subtree (lines 17-20). For a node which is a deadend, we remove it from its parent's legal successors (lines 26-28). If we still need to solve the tree rooted at the AND node (it wasn't in the cache or a deadend), we initialize its OR children (lines 22-25) and add them to the top of the OPEN stack.

PROPAGATE lines 31-49: This process is triggered when a node has an empty set of successors, which may mean that the node is a leaf (we've found a solution and its value needs to be propagated to its ancestors) or we've pruned it. Lines 33-38 handle the case of an OR node, if the node evaluated is the root, we return its value and solution tree. For any other OR node, we add its value to its parent AND node and add the solution subtree found to its parent's. For an AND node (lines 40-49), we save the value of the solution to the cache. We calculate the best solution for its parent OR node (line 44) and if the rooted at the AND node is better, we change the parents assignment to the evaluated AND node. Afterwards, we remove the node from the parent's successors as either this is the best solution so far or the solution is worse than one found (no need to evaluate it any further in any of the two cases. We continue this process with the node's parent until one of the nodes has successors that haven't been evaluated yet.

8.6 Best-First AND/OR Search

Best-First AND/OR Search is the adaptation of A^* for AND/OR search spaces. As in the implementation for OR-trees, at each step we expand the most promising nodes in terms of the evaluation function, f. We label nodes as SOLVED when we've found the optimal solution for the subtree that they root; after, we propagate this information to its ancestors until we label the root node as SOLVED. Only then we've found the optimal solution. As in the case for A^*, this algorithm is guaranteed to expand the least amount of nodes for finding the optimal solution with the given heuristic function. Below we can see its detailed pseudo-code.

The algorithm keeps track of the partial solution found so far in the structure CT and its value in v(s). First we create the partial solution tree (lines 4-10) by selecting the marked successors for each node. After, we can distinguish two different processes:

EXPAND lines 11-28: we select a node among the non-terminal nodes of the partial solution tree[4]. If the selected node is an OR node, we initialize all of its AND children; we can either do this by obtaining it from the cache (line 14) or by calculating its value and weight (lines 15-17). If any of the AND nodes is terminal, we label it as SOLVED. For an AND node, we initialize the value to their heuristic value. We add all of the new tip nodes to the current solution (in the next iteration, once of them will be evaluated).

REVISE lines 29-44: this process updates the partial solution tree in a bottom-up manner, starting with the node that has just been expanded. If the current node is an AND node, we calculate its value as the sum of the value of all of its OR nodes and we mark all the AND-to-OR edges. If all of its children are labeled as SOLVED, we label the current node as solved. For an OR-node, we calculate its cost as the minimal cost for the sum of the value of a child and the edge between them. If the node being currently evaluated has been solved or his value has changed (only a cost-increment is possible since the mini-bucket heuristic used is a monotonic function as shown before) we add the ancestors that have it as marked successors to propagate this change up the tree.

[4]In a naïve implementation this node is selected randomly. In anytime implementations, there's a secondary heuristic, called ordering heuristic, for prioritizing those subproblems that will tighten the most the bounds of the solution. [Lam et al., 2016]

ALGORITHM AND/OR Best-First Search

```
1.  v(s) = h(s)
2.  CT = {s}
3.  while s is not SOLVED:
4.   S = {s}
5.   PST = {}
6.   while S ≠ ∅:
7.     n = S.pop()
8.     PST = PST ∪ n
9.     if marked_succ(n) ≠ ∅:
10.      S.push(marked_succ(n))
11.  select n among tip nodes
12.  if n is OR node:
13.    for all AND children x_i:
14.      let n' be Cache.assn(π_n)[pas_i]
15.      if n' == NULL:
16.        v(n') = h(n')
17.        w(n, n') = ∑ f(assn(π_n))
18.        if n' is TERMINAL:
19.          label n' as SOLVED
20.      succ(n) = succ(n) ∪ {n'}
21.
22.  else if n is AND node:
23.    for all OR children m_i:
24.      v(m_i) = h(m_i)
25.      succ(n) = succ(n) ∪ {n'}
26.
27.  CT = CT ∪ {succ(n)}
28.  S = {n}
29.  while S ≠ ∅:
30.    let m be a node in S without descendants in CT still in S
31.    remove m from S
32.    if m is AND node:
33.      v(m) = ∑_{m_j∈succ(m)} v(m_j)
34.      mark all arcs to the successors
35.      label m as SOLVED if all successors are labeled SOLVED
36.
37.    else if m is an OR NODE:
38.      v(m) = min_{mj∈succ(m)} (w(m,mj) + v(mj))
39.      mark the arc of the child with minimal value
40.      label m as SOLVED if the child is SOLVED
41.
42.    if m changes its value or m is labeled SOLVED:
43.      add to S all parents who have m as a marked successor
44.
45. return v(s)
```

8.7 DAOOPT

DAOOPT (Distributed AND/OR Optimization) is a search framework developed in Rina Dechter's group at the University of California, Irvine that includes an implementation of AND/OR Branch-and-Bound DFS and AND/OR Best-First Search and that can be configured to execute with and without caching (executing the algorithms through a AOG or an AOT) as well as the possibility of parallelizing independent subproblems. Both the AND/OR Beam

Search and Anytime AND/OR Beam Search (explained in section 10) were added to this framework whose C++ implementation was made by William Lam and Lars Otten and is available in their Github repository[5].

DAOOPT answers MPE queries over graphical models and includes:

- Min-fill algorithm to find variable orderings.

- Context-based caching to implement AND/OR graphs.

- Mini-Bucket Elimination algorithm, needed as heuristics in the previous ones.

- AO* (AOBF), Anytime AO* (AAOBF) and AND/OR Branch-and-Bound (AOBB).

- Parallelization of AND/OR Branch-and-Bound [Otten and Dechter, 2010].

- Dynamic Heuristics [Lam, 2017].

9 Trade-offs between the width and height of pseudo-trees

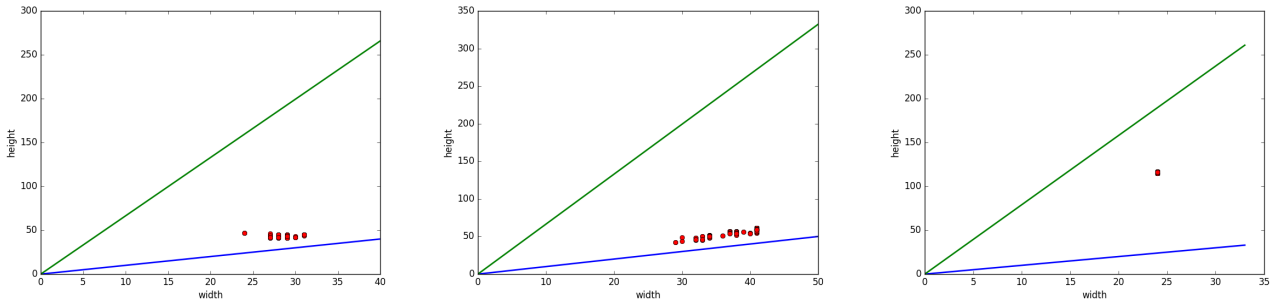
In this section we'll present an analysis of the results found when studying the trade-offs between pseudo-trees generated by the code in [Carmeli et al., 2017]. The time measurements corresponds to the execution of different problems of UAI probabilistic inference challenges[6].

There are several heuristics available to guide the triangulations (MCS, Min-degree, initialDegree, min-fill, initialFill, ...). Empirically, we can say that heuristics that offer better orderings in terms of induced width (MCS, min-degree and min-fill) show less variability in terms of the induced width (Figure 11c, executed with MCS). Since the aim of this work was to visualize the trade-offs between height and width of the orderings we opted for heuristics that, although offering worse width, show more variability in the width (and consequently height) of the orderings. In particular, the rest of the plots below were obtained with initialDegree.

The height of the pseudo-tree and the induced-width of a graphical model are not independent factors but they are related to one another. To be specific, for a given width we can find a pseudo-tree with a height h that fulfills the expression $h \leq w \cdot log\ n$, where n is the number of variables/nodes of the problem. We also know that $h \geq w$ (since in a chain-like model, the largest width a variable can have includes all the previous variables, value bounded by h - 1), therefore if we measure the ratio between height and width we can generate values of the ratio:

$$1 \leq \frac{h}{w} \leq logn$$

In the plots, the blue line represents equal height and width (ratio = 1) and the upper bound h = w log n (ratio = log n).

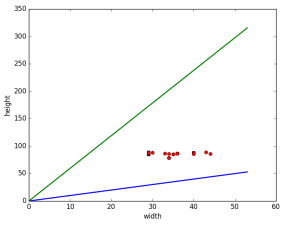


(a) BN (b) BN (c) WCSP

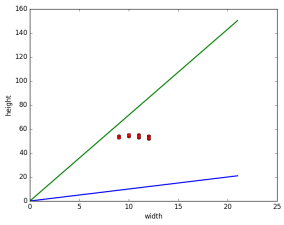
Figure 11: Width-Height plots

[5]https://github.com/willmlam/daoopt-exp

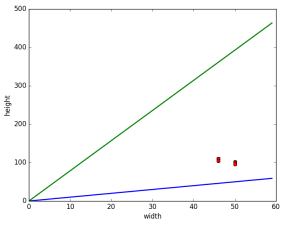
[6]In particular, the source for the problem was Alex Ihler's website http://sli.ics.uci.edu/ ihler/uai-data/



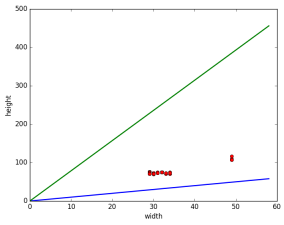
(a) Pedigree



(b) WCSP

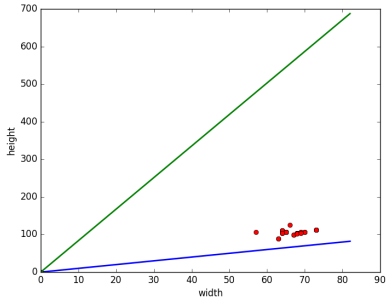


(c) Image Segmentation

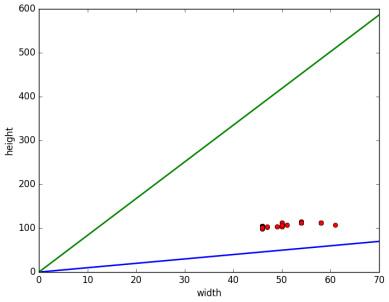


(d) Image Segmentation

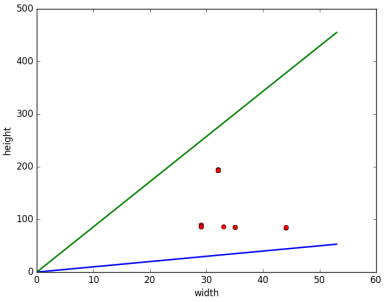
Figure 13: Width-Height plots



(a) CPD



(b) CPD



(c) Pedigree

Figure 12: Width-Height plots

To get a better grasp of the measure we've established, let's remember what this metrics measure to try to give the previous ratio a meaning:

Width: The width of the pseudo-tree is equal to the size of the largest context, this is, for a given variable the amount of variables that appear before it and are connected to it or its descendants. This measure is relevant since it represents the size of the largest function that has to be recorded when executing Bucket elimination. We also know that the amount of times a subproblem appears corresponds to the amount of possible instantiations of its context k^w.

Height: The height of a pseudo-tree determines the size of the AND/OR space, which is exponential in this measure. A smaller height, will yield a smaller search space and intuitively a faster search.

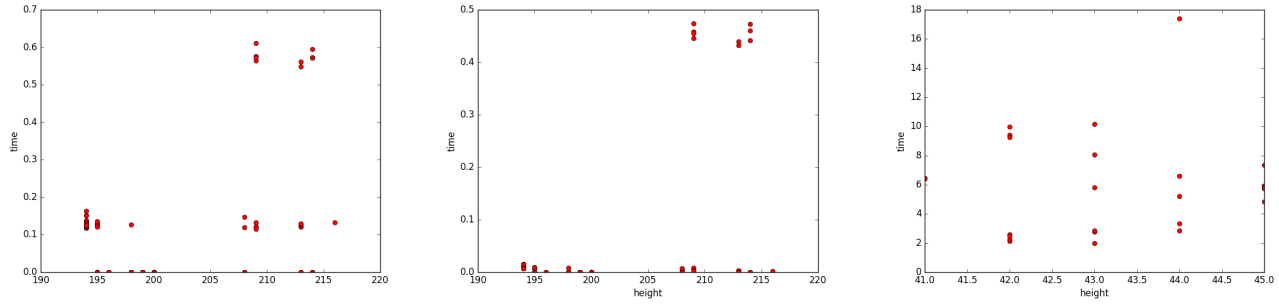
Another important concept is that of **dead caches**, that establishes that when the context of a node is formed by all the nodes that form a path from the root to it, this subproblem is never revisited. In other words, the height of this node is equal to its induced width. If the amount of variables shared between the path and the context is small (most variables in the path don't appear in the context), the problem will appear a large amount of times. This relation could be generalized by the ratio, for a really large value we can intuitively think that problems will appear in several occasions. How does this translate to the execution of AND/OR Branch-and-Bound or AND/OR Best First search? If the ratio is large, either of the algorithms should benefit greatly if executed with caching (shorter execution times) Otherwise, the difference shouldn't be great and the algorithms may benefit from their parallelization. Moreover, for the same value of width, there's shouldn't be much of a difference in execution times between those pseudo-trees with a larger height and those with a smaller height if we're caching the results.

We will now analyze two types of plots:

- For a given problem and width, we will execute all of its possible heights to check if smaller heights mean better execution times without caching. After this, for caching to check the effect disappears once we cache the results.

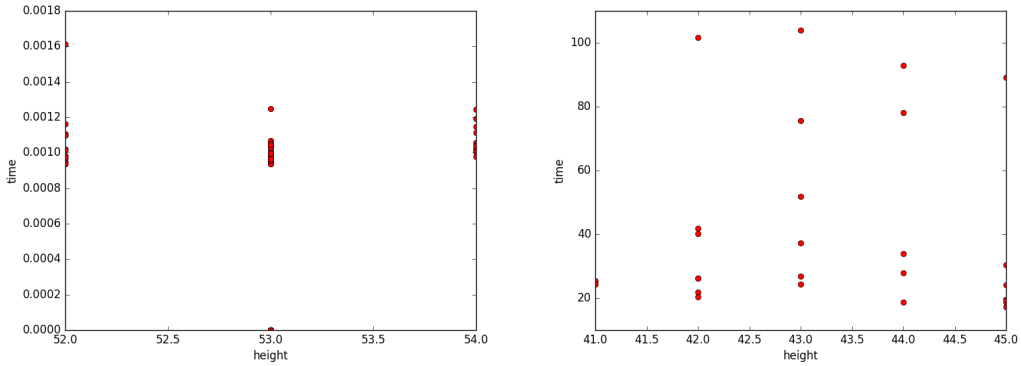
- To check if the ratio generalizes the idea, we will plot for a given problem all of its orderings with caching and without caching and plot the execution times against the ratio of height and width.

9.1 Same Width-Different Height & Execution time Plots



(a) Pedigree (b) Pedigree (c) BN

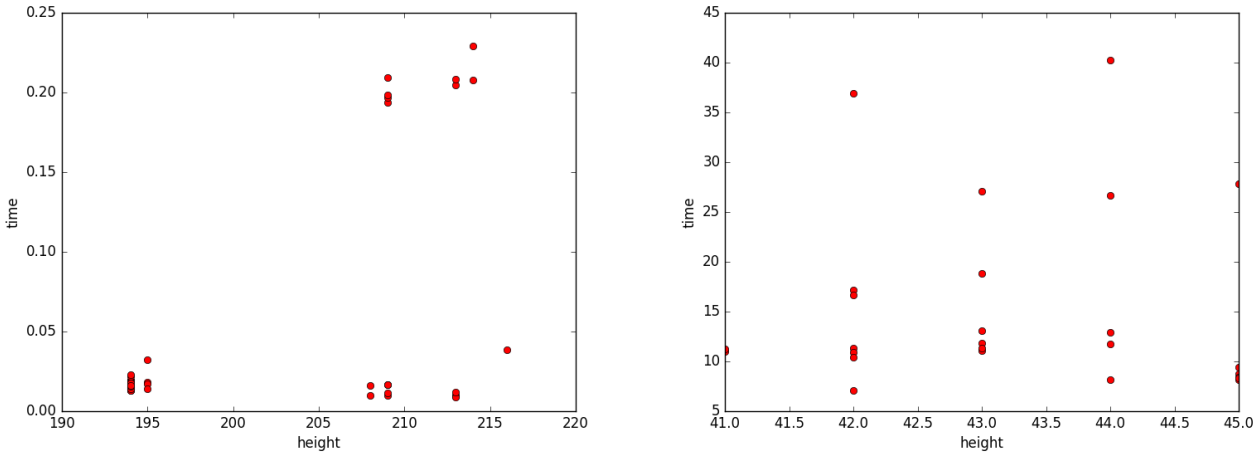
Figure 14: Height-time plots for orderings with the same width (AOBF no-caching)



(a) WCSP (b) BN

Figure 15: Height-time plots for orderings with the same width (AOBF no-caching)

From the results we can say that the hypothesis seems to be architecture-dependent, this is, some domains of networks respond better to the hypothesis than others. The Pedigree network seem to confirm the hypothesis since in the plot we can see how larger times are only achieved by larger heights. This being said, shorter times are also achieved by those heights which leads us to say that, for Pedigree networks, if we compare two orderings with the same width a smaller height guarantees to have a shorter or equal time than the larger heights. For some problems from the BN domain, we can see some outliers for smaller heights confirming the idea that there's a dependence on the architecture of the network built (although the probability of having a longer execution time is higher for a taller trees). In Figure 15a, we can see an example of a WCSP (representative of its domain in term of the results) that seems to be unaffected by the height of the ordering. From these plots, we can conclude that it's a good heuristic to follow to choose those trees with a smaller height if they share the same width, although it's not the only case for which those times may be achieved. Below we can see the execution times of the problems in Figure 14b and 15b caching the results. We can observe that the plots look really similar to those without caching but with times cut in half. This points to the idea that reducing the search space is effective but doesn't vanish the effect of the height in the search tree.



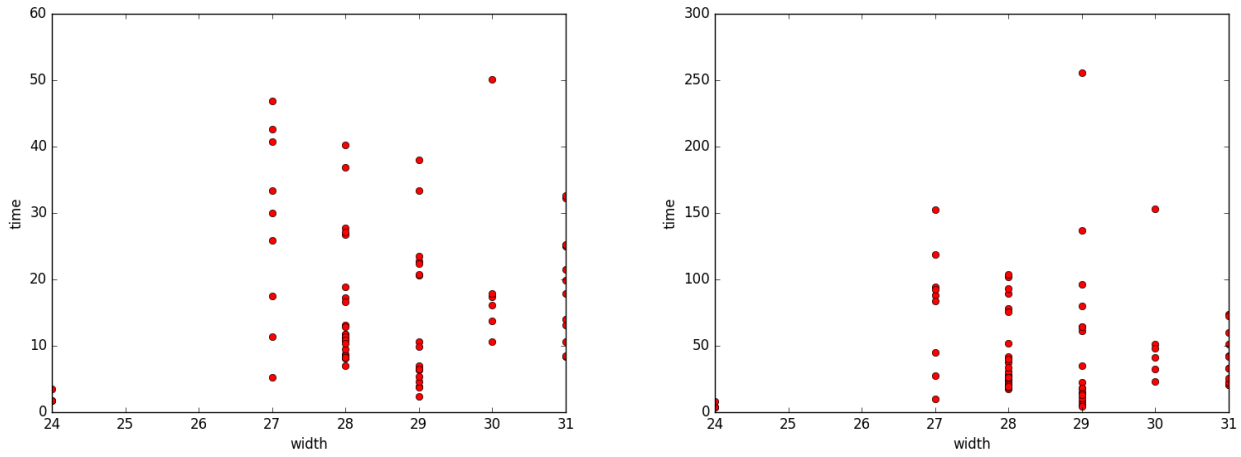
(a) Pedigree

(b) BN

Figure 16: Height-time plots for orderings with the same width (AOBF caching)

9.2 Height-Width ratio & Execution time Plots

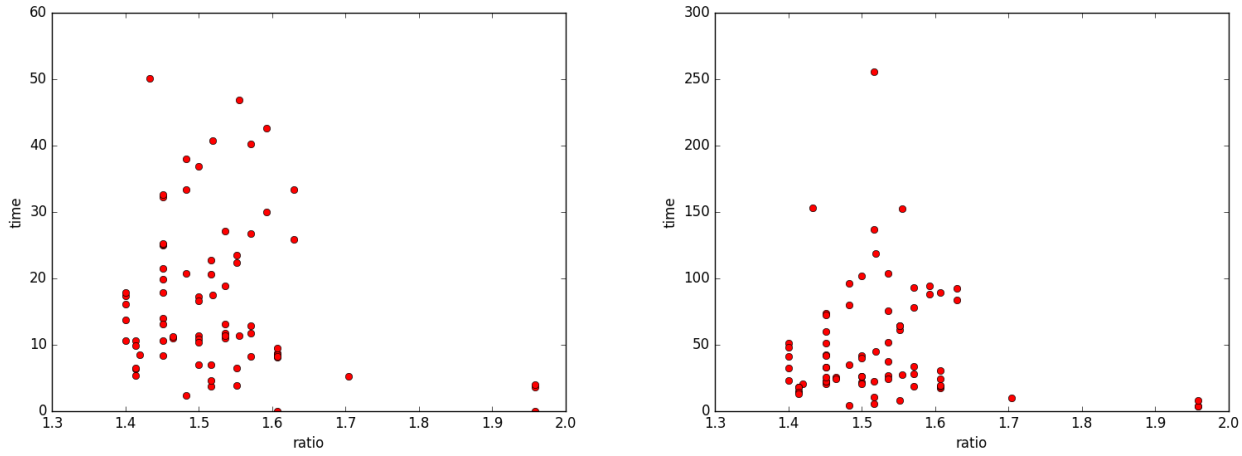
In order to analyze if the ratio is a good heuristic for choosing a pseudo-tree we've plotted two types of graphs. The first one corresponds to the width-time relation, this is, for a single problem we plot the widths and execution times of all the orderings we've found with the triangulation algorithm. The second type, is the plot of the ratio height-width of the orderings against the execution times. With these plots we try to discern how valuable the information given by the ratio is and whether this information is simply a re-statement of the one given by the width. Throughout the results we'll see that the results are domain-dependent.



(a) AOBF caching

(b) AOBF no-caching

Figure 17: Width-time plots for BN example

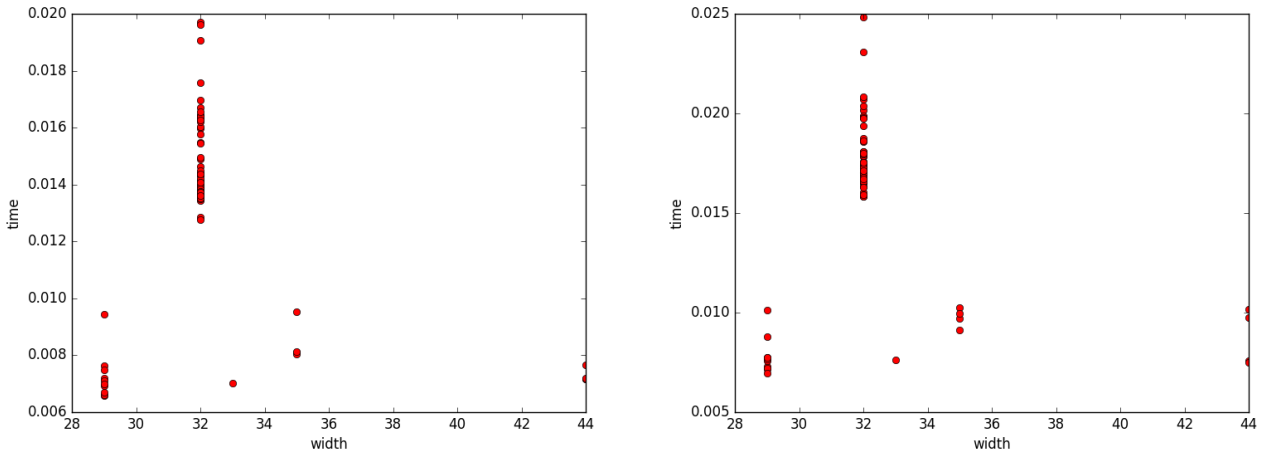


(a) AOBF caching (b) AOBF no-caching

Figure 18: Ratio-time plots for BN example

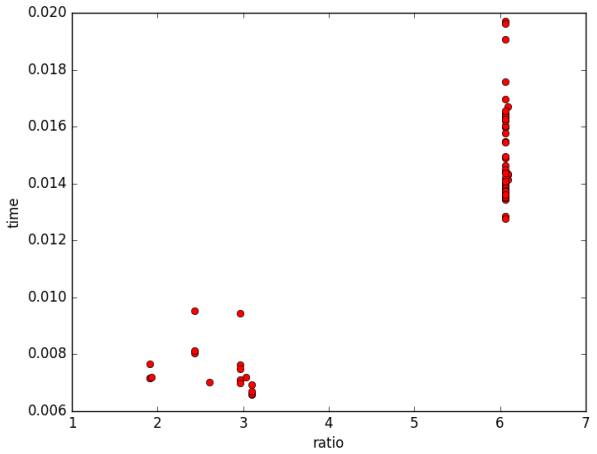
In Figure 17, we can see the distribution for the execution times of AOBB with and without caching. Both plots look really similar to one another, the main difference being the scale of the y-axis (smaller values for the caching execution). An important remark to be made is that these execution times don't seem to be a direct function of the width. In other words, the quality of the execution doesn't seem to be guided by the width.

In Figure 18, we see the plot of the height-width ratio. The first thing to notice is that the distribution changes radically. A more compact tree (smaller ratio between width and height) seems to yield better results than larger ratios. This is a key example where the ratio could be used. Whenever the width-time plot doesn't clearly show an exponential relation we can make use of the second type of plot to choose a better ordering.

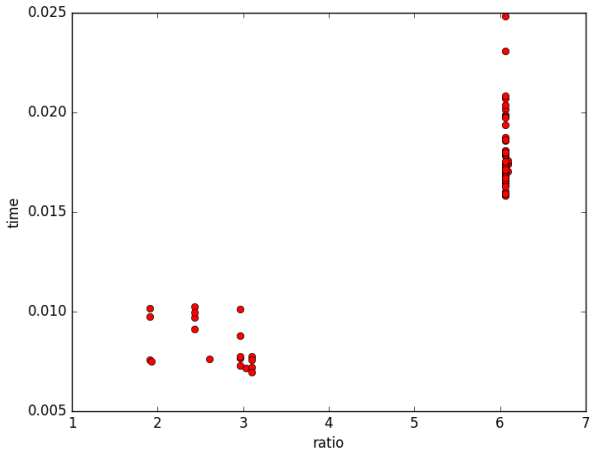


(a) AOBF caching (b) AOBF no-caching

Figure 19: Width-time plots for Pedigree example



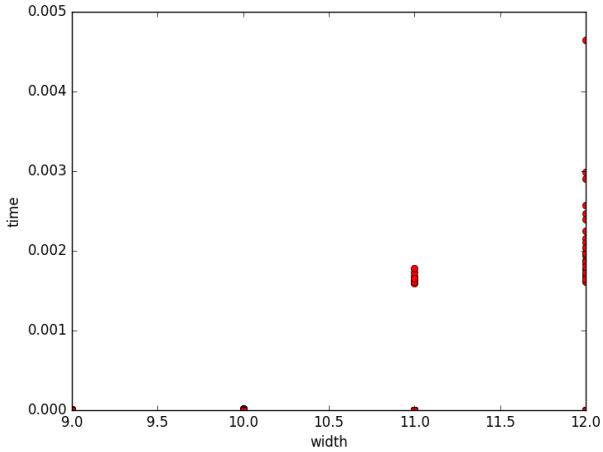
(a) AOBF caching



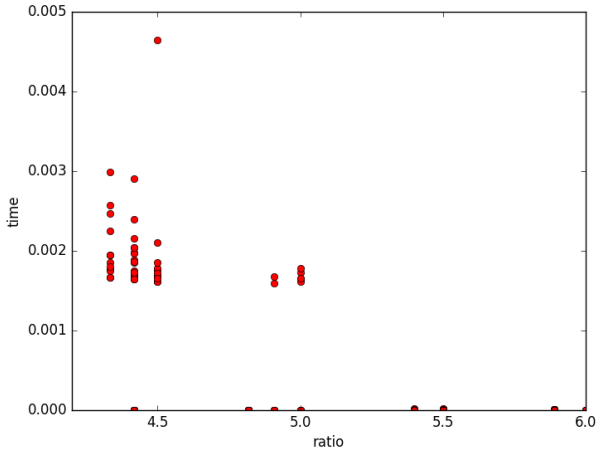
(b) AOBF no-caching

Figure 20: Ratio-time plots for Pedigree example

The plots above for a Pedigree example, follow the same pattern shown for the BN example. The width-time plots don't show a clear relation between them. On the other hand, ratio-time plots point out that a more compact tree in general yields a better execution time and the difference between caching and no-caching is simply in the shape of reduced times.

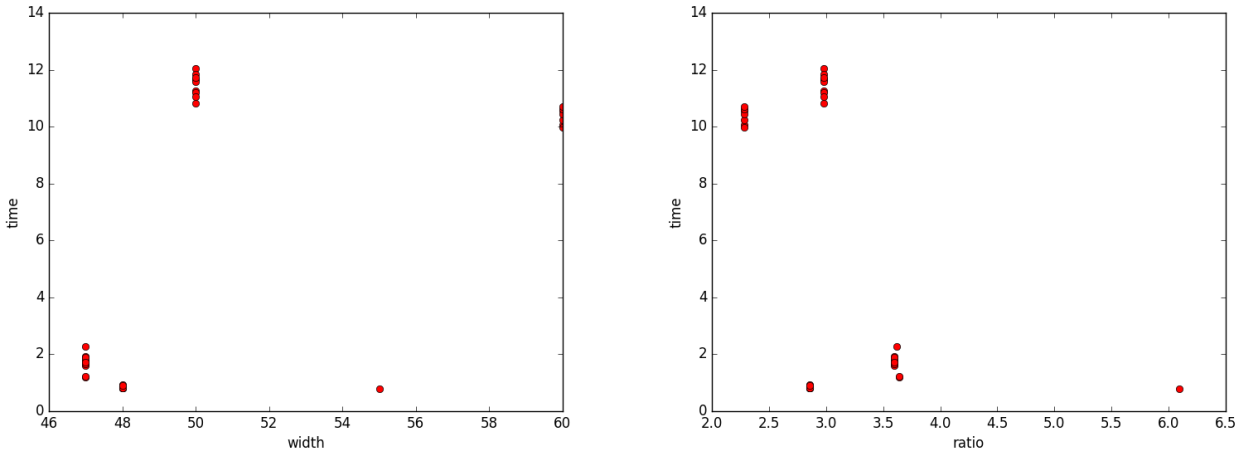


(a) Width-time plot



(b) Ratio-time plot

Figure 21: WCSP example



(a) Width-time plot

(b) Ratio-time plot

Figure 22: Promedas example

The examples shown above, taken from the WCSP and Promedas domain, are the mirror image of the ones shown above. The width-time plots show that, on average, a large width causes longer executino times. Whenever we plot the ratio against the execution times, we can see that smaller ratios (corresponding to a larger width) yield a worse-quality solution which points out that the guiding metric for selecting orderings should be the width only rather than the ratio.

From the plots in this section we can think that there are two types of models: those for which we would select the orderings that produce a pseudo-tree with a smaller ratio (more compact tree) and those who respond drastically to the increment in the width of the model for which we would choose a larger ratio (choose orderings with smaller widths independently of their height). The problem is that, for doing so, we need to execute a graphical model with distinct orderings to see which type we're working with.

In an effort to distinguish between them prior to this execution we show the characteristics of the examples run above as shown in Alex Ihler's repository for UAI problem files:

Model	Width	Depth	Number of variables	Average domain size	Max. domain size
BN	14	21	100	2	2
Pedigree	19	61	385	2.06	3
WCSP	9	33	143	2.81	4
Promedas	21	60	1005	2	2

There doesn't seem to be a direct correlation between the characteristics of the graphical model shown above and the results obtained in the previous section. It could be the case that the relation is merely an issue of the connectivity of the graphs and could be modeled through other parameters.

In the ending section, we'll talk about future work that could be done in this research topic to help drive orderings and derive theories that could make it useful to the community.

10 Beam Search

Beam Search is a well-known algorithm for heuristic search whose execution depends on a β parameter called beam-width. At each point in time, Beam Search arranges the open nodes in terms of the evaluation function and opens a total of β nodes and forgets about the rest. Beam Search does not guarantee optimality or completeness (it may not find a solution even if one exists). The interesting points about this algorithm are that it offers a pruning policy that it improves the memory requirements from a Best-First Search algorithm and that, in a single pass of the algorithm, it explores more solutions than Branch-and-Bound (the first solution found by Branch-and-Bound is equivalent to Beam Search with $\beta = 1$).

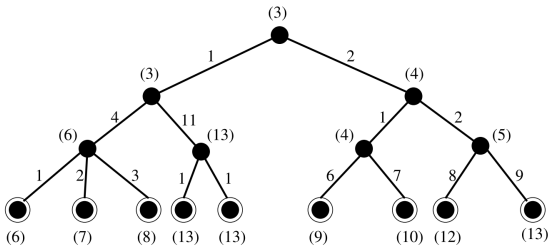


Figure 23: A search tree: edges labeled with the assignment costs and nodes with heuristic values. [Vadlamudi et al. 2013]

An important remark to be made about Beam Search is that it's not monotonic with respect to the beam-width, this is, a larger β doesn't guarantee a better solution, as it can be seen in the image above. For a beam width of value 1 we find a solution with cost 6 and for a value of 2, we find one with a cost of 9. Due to this fact, different version of Beam Search have been developed including an anytime version [Zhang, 1998] and an incremental one [Vadlamudi et al. 2013]. Minding the fact that it's not guaranteed it's intuitive to think that the overall trend will be finding a better solution for a larger β (for a value of ∞ we get the equivalent of Breadth First Search in OR trees).

In this section we will present an adaptation of the classic Beam Search for AND/OR spaces and afterwards a more interesting AND/OR Anytime Beam Search.

10.1 AND/OR Beam Search

The solution of a problem in an OR-tree takes the shape of a path, therefore all of the nodes in the fringe can be considered independently from one another. This is not the case for an AND/OR tree since a solution takes the shape of a subtree (for an AND node in the solution, all of its children are in the solution and for an OR node only one of the children is in the solution). This limits the way the pruning in Beam Search can be adapted to AND/OR search; we can't select the best β nodes from the tip nodes, as we may eliminate all of the children for an OR node or at least one children of an AND node. Therefore, the pruning of the graph won't be done with respect to the list of fringe nodes but rather before initializing the children of a node. For an AND node, we will initialize all of its children as in AOBF; for an OR node, rather than adding all of its children to the successors (line 20) we will arrange the nodes in terms of its heuristic function and add to its successors only its β best children. This way, the children not included at this point won't ever be considered reducing effectively the search space. On the other hand, this pruning may eliminate the optimal solution (Beam Search doesn't guarantee optimality) or, even, prune all available solutions (Beam Search doesn't guarantee completeness) in the case of a Constraint Satisfaction Problem. An important remark to be made is that for a value of the beam width equal to the largest domain size, AND/OR Beam Search will be equivalent to AOBF.

The rest of the algorithm remains the same so we won't show the pseudo-code for the implementation. We will call this implementation AOBeam.

10.2 Stochastic AND/OR Beam Search

In an effort of improving the algorithm in its weakest points, we introduce an improvement to the implementation of AND/OR Beam Search with ideas from the literature in stochastic local search by pruning the nodes according to a probability distribution.

For each OR node, we will keep besides the β best nodes all the rest of the nodes with a probability proportional to its heuristic function. In specific, if the heuristic function is a lower bound of the solution (the smaller the value the better) we will keep that node, n, with a probability of:

$$p(n) = 1 - \frac{h(n)}{\sum_{n' \in N} h(n')}$$

For a heuristic function that's the upper bound:

$$p(n) = \frac{h(n)}{\sum_{n' \in N} h(n')}$$

where N denotes n and all of its sibling AND nodes.

We've also introduced a new parameter α that adds up to this probability. This way we will keep a node in the fringe nodes with a probability of p'(n) = p(n) + α, increasing its value every iteration. This method of weakening the pruning policy is inspired by [Zhang, 1998].

An implementation of *Stochastic* AND/OR Beam Search and its subroutine `prune_nodes` is shown below. We show Stochastic AND/OR Beam Search it in its entirety for completeness of the work but the only lines that change with respect to AOBF are highlighted.

In order to prune the nodes in the successors of an OR node, we first sort the nodes from better to worse according to their heuristic value. We will keep the first β nodes; for the rest, we will keep a node n with a probability p(n) computed as shown previously (this process is simulated with a biased coin toss).

ALGORITHM prune_nodes(succ, β, α)

```
1. sort succ w.r.t their heuristic
2. for i = β + 1 to |succ|:
3.    r = biased coin toss with p(succ(i))
          probability of heads
4.    if r == tails:
5.      succ = succ \ succ(i)
6. return succ
```

ALGORITHM StochasticAOBeam(β, α)

```
1.  v(s) = h(s)
2.  CT = {s}
3.  while s is not SOLVED:
4.   S = {s}
5.   PST = {}
6.   while S ≠ ∅:
7.     n = S.pop()
8.     PST = PST ∪ n
9.     if marked_succ(n) ≠ ∅:
10.      S.push(marked_succ(n))
11.  select n among tip nodes
12.  if n is OR node:
13.    succ_beam = ∅
14.    for all AND children x_i:
15.      let n' be Cache.assn(π_n)[pas_i]
16.      if n' == NULL:
17.        v(n') = h(n')
18.        w(n, n') = ∑ f(assn(π_n))
19.        if n' is TERMINAL:
20.          label n' as SOLVED
21.      succ_beam = succ_beam ∪ n'
22.    succ(n) = prune_nodes(succ_beam, β, α)
23.  else if n is AND node:
24.    for all OR children m_i:
25.      v(m_i) = h(m_i)
26.      succ(n) = succ(n) ∪ {n'}
27.
28.  CT = CT ∪ {succ(n)}
29.  S = {n}
30.  while S ≠ ∅:
31.    let m be a node in S without descendants in CT still in S
32.    remove m from S
33.    if m is AND node:
34.      v(m) = ∑_{m_j∈succ(m)} v(m_j)
35.      mark all arcs to the successors
36.      label m as SOLVED if all successors are labeled SOLVED
37.
38.    else if m is an OR NODE:
39.      v(m) = min_{mj∈succ(m)} (w(m,mj) + v(mj))
40.      mark the arc of the child with minimal value
41.      label m as SOLVED if the child is SOLVED
42.
43.    if m changes its value or m is labeled SOLVED:
44.      add to S all parents who have m as a marked successor
45.
46. return v(s)
```

10.3 Anytime AND/OR Beam Search

An interesting feature of AOBB is that it's anytime. This is, at any point in the execution we have a solution that can improve with time if we keep executing the algorithm. We can convert Stochastic Beam Search into an

iterative version by one of these two methods:

a) increasing the beam width each iteration.

b) increasing the α value.

Option a) is more aggressive as we would consider one more value for all of the OR nodes in the graph. Option b), on the other hand, allows us to tune the looseness of the pruning policy more finely since we will include more slowly in the search space solutions that weren't previously considered[7]. Whatever the policy selected is, we could include it in the subroutine `weaken_pruning_policy()`.

The only thing left to configure would be the stopping condition for the algorithm. We can't stop it when it finds the optimal solution as we don't know when this solution is found. On the other hand, we can set a timeout for the algorithm, an upper bound for the β parameter, a number of iterations increasing the α value, a number of iterations without the solution changing or a combination of all the previous ones.

An implementation of Anytime AND/OR Beam Search (AAOBeam) is shown below. Along the execution we keep the value of the best solution found so far in a variable v and its assignment in s, if a new solution improves[8] the best found so far we record it. We weaken the pruning policy and if the stopping criteria is not met, we execute AOBeam with the new values of β and α.

ALGORITHM AAOBeam(β, α)

```
1. v = -∞
2. s = ∅
2. while stopping condition not met:
3.    v', s' = StochasticAOBeam(β, α)
4.    if v' > v:
5.      v = v'
6.      s = s'
7.    weaken_pruning_policy()
```

This implementation is purely theoretical and hasn't been tested experimentally.

11 Evaluation and Test Results

The problems used to test AOBeam and Anytime AOBeam were taken from the same repository as the ones for testing the trade-offs of the width and height of the pseudo-trees. As we mentioned before, for a value of the beam width equal to the maximum size of a domain in the network AOBeam is equivalent to AOBF (leading to a useless comparison); due to this, we selected problems with large domain sizes (at least larger than 2) as that characterizes a problem that can make using AOBeam interesting to use and serve as an useful comparison with AOBB, AOBF.

The task evaluated is MPE and the plots will show the logarithm base 10 of the bounds generated rather than the bounds itself for a better representation. Therefore, any difference in the solutions that's linear in the plots it's in reality exponential in the probability values. g Below, we show the characteristics of the problems chosen:

[7]Obviously for a value of $\alpha = 1$, we would consider every possible value in the domain of each OR node.

[8]The example considers an optimization task where we want to maximize the value of the function.

Domain	Name	Width	Height	#variables	avg. dom. size	max. dom. size
BN	BN_101	16	23	58	9	50
BN	BN_103	17	25	76	11.86	50
WCSP	CELAR6-SUB0.wcsp	7	11	16	40	44
WCSP	CELAR6-SUB1.wcsp	9	10	14	44	44
WCSP	CELAR6-SUB2.wcsp	15	15	16	43	44
Object Detection	0034.K10.F2.model	7	25	60	11	11
Object Detection	0034.K10.F100.model	59	59	60	11	11
CPD	1b2v	9	37	132	9.23	36
CPD	1czs	20	37	143	22.8	81
CPD	1e1m	16	64	374	20.12	81
Segmentation	11_17_s.21	19	50	228	21	21
Segmentation	10_16_s.21	17	50	233	21	21

11.1 AOBeam vs AOBF vs AOBB

Below we see the results of the tests recorded when testing AND/OR Beam Search (with $\beta = 2$), AOBF and AOBB with the problems in the table mentioned before. For some of the problems above we show the bound on the solution generated by each one of the algorithms: upper bound for AND/OR Beam Search (in gold) and AOBF (in blue) and lower bound for AOBB (in green). When available in Alex Ihler's website, the value of the optimal solution is in the middle of the values found represented as a red horizontal line. Even if this solution is not represented we know that it's located in between both bounds as they all offer legal bounds.

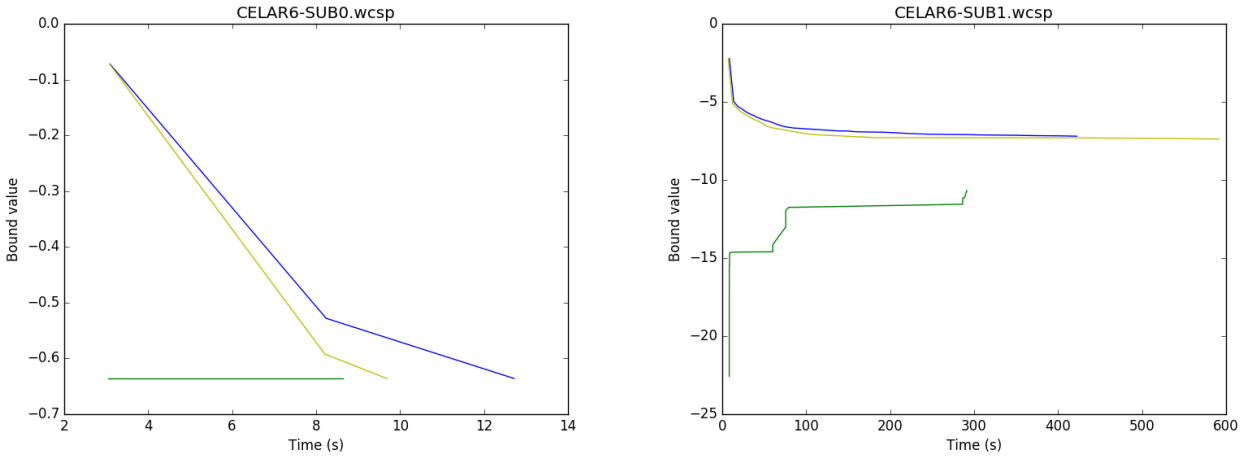
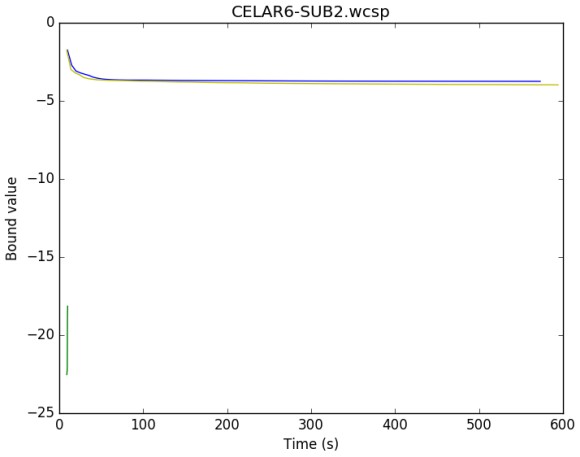
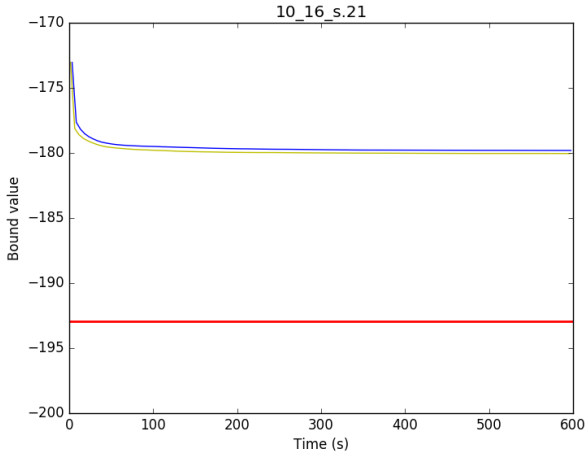


Figure 24: Plots for problems in the WCSP domain.

Above we can see the results for two WCSPs. For the one on the left, all of the algorithms find the optimal solution pretty quickly, 8-13 seconds, the faster being AOBB. In this first example, we can clearly see one of the benefits that we can obtain from executing AOBeam: aggressive pruning. Although AOBF is guaranteed to obtain the optimal solution (and AOBeam not) it does it by not discarding any subtrees along the way and this, generally, will make the generations of solutions slower than for AOBeam. In the second example, we can see an execution for a problem where none of the algorithms find the optimal solution. Overall, we can see that the curve for AOBeam is slightly smaller than the one plotted by AOBF although converging at certain points in the curve. Here, we can see the other advantage that AOBeam may have over the other algorithms. Since it considers a overall smaller amount of subtrees, this allows AOBeam to traverse deeper parts of the tree that may bound closer the solution even if they don't lead to the optimal solution. This is reflected in AOBeam's line being the one that generates the latest bound in the plot (its bound continues to decrease until almost the end of the time interval shown). AOBB, although traversing the tree in a depth-first needs to evaluate all the possible nodes in the tree. This exhaustive evaluation of the space is the reason why, after a promising start, from 300 seconds to the end of the execution AOBB doesn't generate any tighter bound.



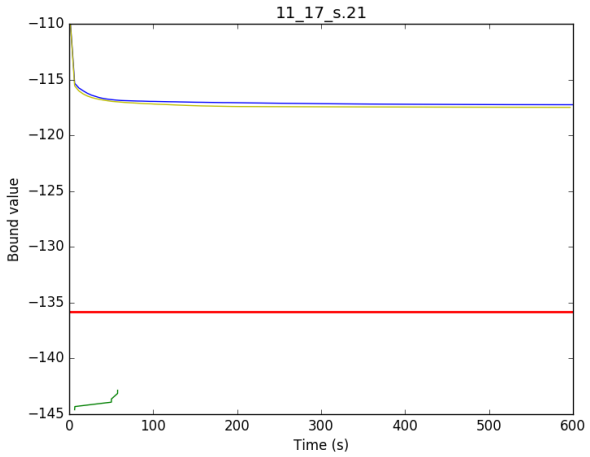
(a) WCSP domain



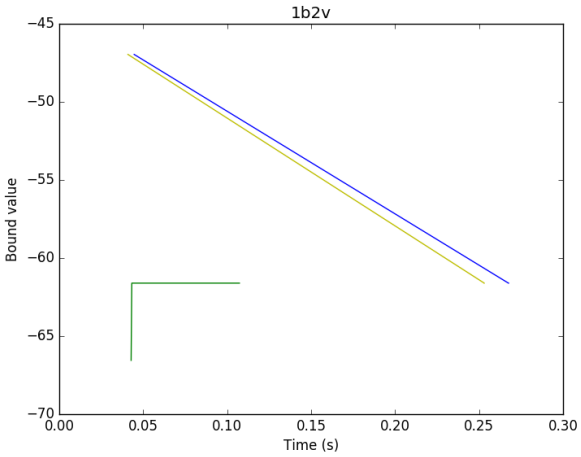
(b) Segmentation domain

Figure 25: Bound curves for different domains

For the plots above, we can see again AOBeam generating an overall smaller bound that AOBF. The important comment to be made about this plots is that, once again, AOBB is incapable of generating a tighter bound than the one found in the initial seconds of the execution. For the Figure 25a, AOBB only finds two solutions and for the Figure 25b, a single one (with value around 198). On the other hand, if we measure absolute error with respect to the solution AOBB generates much more tighter bounds of the solution than any of the other two. The same observations can be made for Figure 26a and Figure 27.



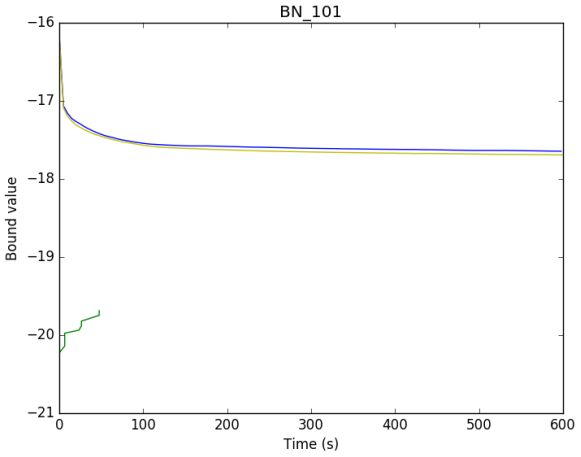
(a) Segmentation domain



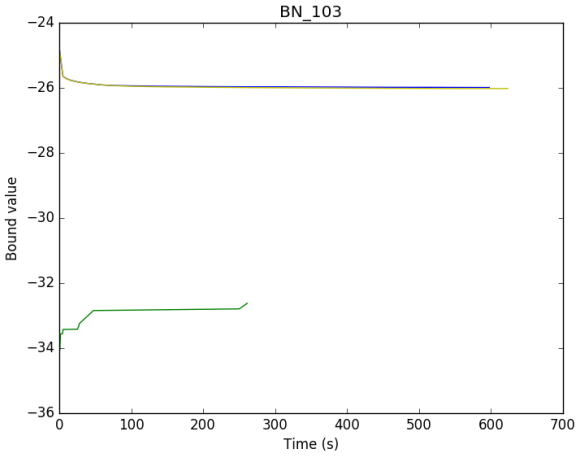
(b) CPD domain

Figure 26: Bound curves for different domains

Figure 26b is an example where all three algorithms find the optimal solution, AOBB being the fastest algorithm.

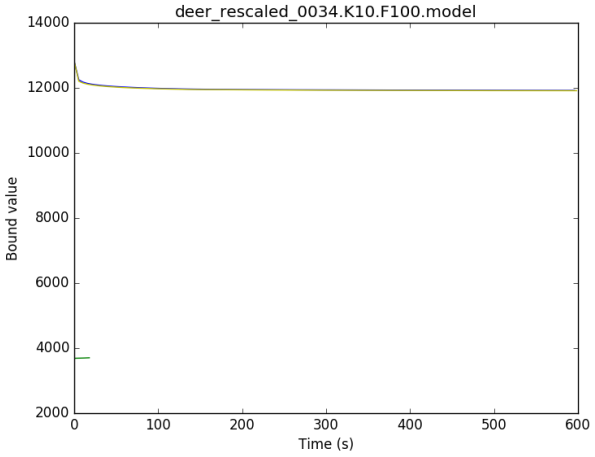


(a) BN domain

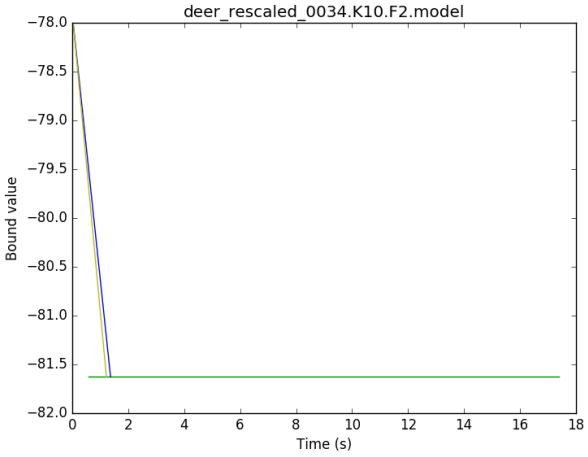


(b) BN domain

Figure 27: Bound curves for examples in the BN domain



(a) Object detection domain



(b) Object detection domain

Figure 28: Bound curves for different domains

The last example shown, Figure 28, shows an example where, again, all three algorithms find the optimal solution. This time, AOBB is the slowest one. This is an atypical example among the test executed; we can see how AOBB has a great bound from the start (really close to the solution) but continues to search the space to locate the optimal value. This case may happen when there are several solutions really close to the optimal one. AOBB may find a suboptimal solution and, since it evaluates the nodes in a DFS manner rather than in terms of its heuristic (only used for pruning) it's required to search a bigger section of the space. For the case of an optimal solution and solutions close to it, AOBF will consider all of them since the beginning (due to their heuristic value) opening a subset of them opening a subset of them until the optimal solution is expanded. For AOBeam it can be one of two cases, if the heuristic value for the subtree with the optimal solution is in the β best nodes, the results will be the same as with AOBF; otherwise, we'll find a suboptimal solution.

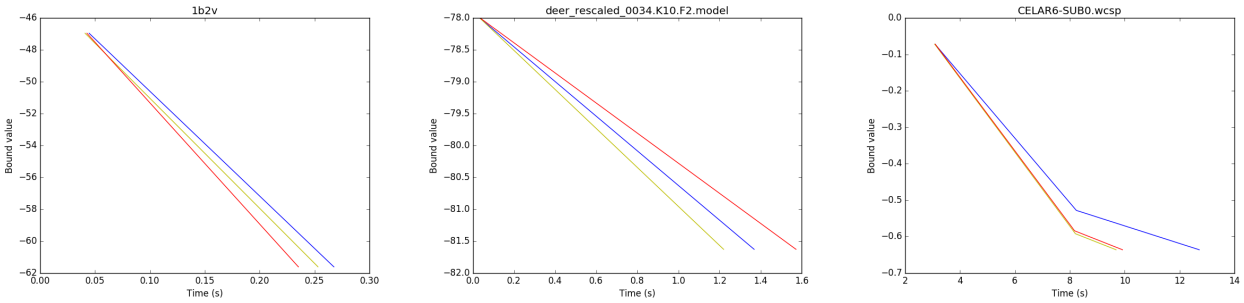
The examples plotted show a range of examples where AOBeam always performs better than AOBF. This is, a biased set of examples. Full executions of examples or simply longer executions, will show how the bound generated by AOBF merges and improves the ones generated by AOBeam. More importantly, AOBeam may stop searching with a suboptimal solution, a guarantee that AOBF will improve its bounds. Other examples where AOBeam may not be as useful, are those with smaller domain sizes as it execution will mimic that of AOBF mitigating any

of its advantages. Overall, we can say that a better bound than may be generated by AOBeam than AOBF if the execution is done in a limited time. In comparison with AOBB, we can say that its priority search will make AOBeam open a smaller set of nodes which for problems where the first solution found by AOBB is far from the optimal one in a DFS traversal.

11.2 Stochastic AOBeam vs AOBeam vs AOBF

In the following plots we center our comparison among StochasticAOBeam, AOBeam and AOBF. As we've seen before, the problem with AOBF is that, as a result of not pruning any node that may lead to a better solution than the actual partial solution it takes longer to tighten the solution bound. Our goal with AOBeam was reducing the search space (sacrificing optimality and completeness) in order to generate tighter bounds earlier in the execution. Afterwards, we introduced the idea of pruning nodes according to its probability as an improvement to AOBeam for it to consider with a larger probability those solutions that are better. This allows us to explore a larger parter of the space when it's as promising as the β first nodes, this is, when its heuristic value is close to the selected ones. We also included the α parameter to tune this probability that will be analyzed later.

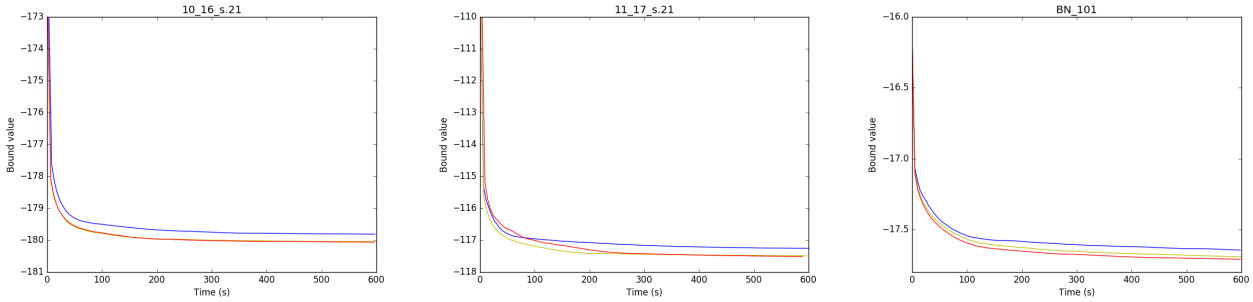
Below we can see some examples of an execution of StochasticAOBeam($\beta = 2$, $\alpha = 0$) in red, in comparison with AOBeam($\beta = 2$) in yellow and AOBF in blue.



(a) CPD domain (b) Object detection domain (c) WCSP domain

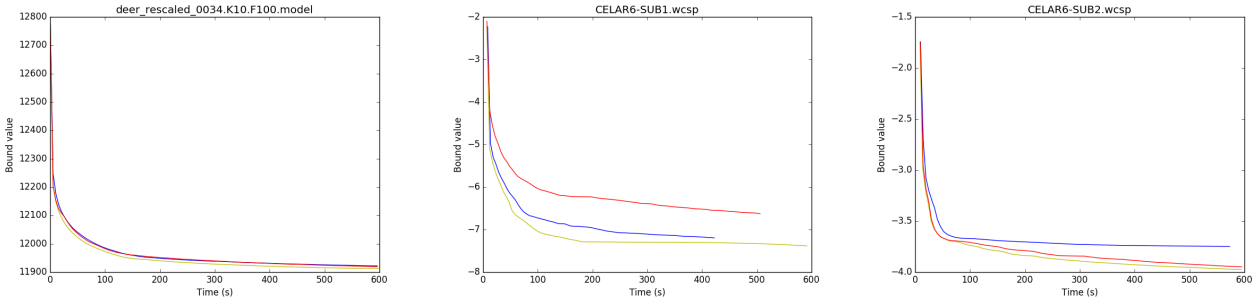
Figure 29: Curves for small problems where all three algorithms find a solution

These first three examples show the performance of the three algorithms in problems whose solution is found quickly. With these examples we try to evaluate how much overhead the calculations added to StochasticAOBeam suppose. This is easier to see for small problems where the calculations would be a large part of the execution time. Overall, we can see that the execution times of StochasticAOBeam are really close to those of AOBF and AOBeam, finding the solution the first on the shortest example and the second on Figure 29c.



(a) Segmentation domain (b) Segmentation domain (c) WCSP domain

Figure 30

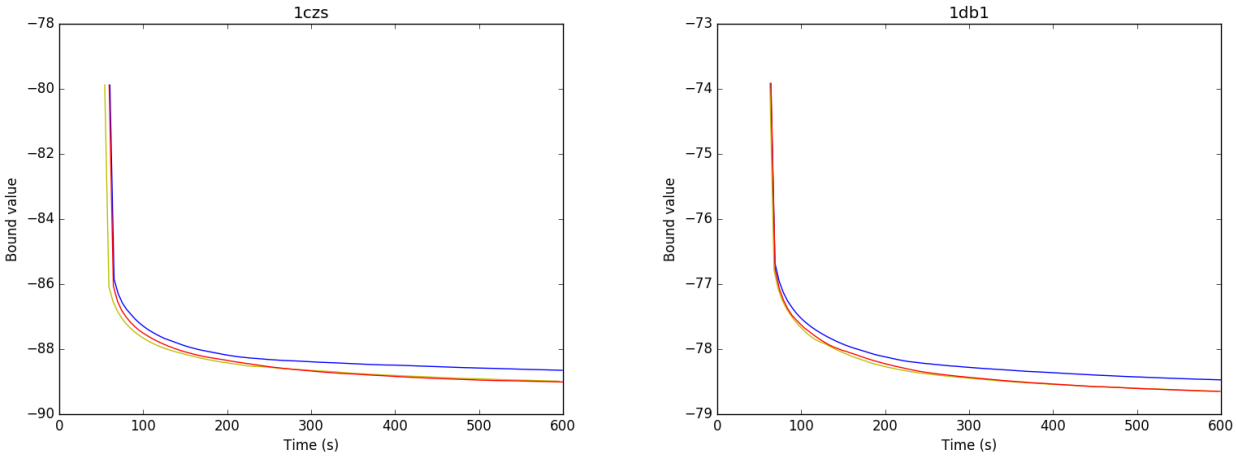


(a) Object Detection domain (b) WCSP domain (c) WCSP domain

Figure 31

In Figure 30 and 31, we can see the first 10 minutes of the executions of different problems. Overall, we can see three really similar curves with both versions of AOBeam having a steeper slope than AOBF. It's interesting to see how the randomness of StochasticAOBeam is shown in Figure 30b is shown as a not-smooth curve, starting as the worst bound and evolving to be the best bound out of the three. This is the double-edge sword that's brought by introducing a probabilistic pruning into play. On the one hand, it may lead us with a larger probability to explore good solutions but on the other hand, it may the algorithm check less-promising parts of the space (i.e. Figure 31b).

The next experiments are with $\beta = 1$, $\alpha = 0.05$ for StochasticAOBeam and $\beta = 3$ for AOBeam.



(a) CPD domain (b) CPD domain

Figure 32

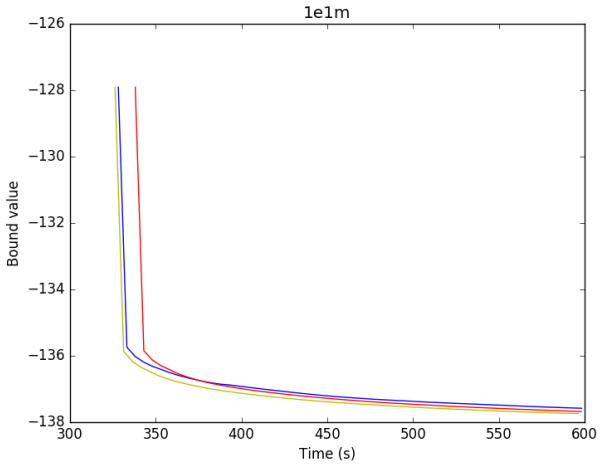


Figure 33: CPD domain

We can see in the first two figures that StochasticAOBeam with a β value that's a third of the one for AOBeam is able to keep up with the bounding of the solution. In figure 35, we can see again an example where the stochasticity of the algorithm executed comes into play, being the last one to start bounding the solution but after that (due to its smaller value of β) it's able to reach the other two algorithms surpassing AOBF.

A problem these last executions make obvious is that the pruning is not as effective as it could be firstly thought. This is due to the fact that opening β nodes per level, doesn't keep β alternatives open but rather, β times the amount of OR nodes we've traversed. For example, if we imagine a chain of OR nodes with domain size equal to 4, the amount of nodes at depth n AOBF will have open are bounded by 4^n. If we execute AOBeam with $\beta = 2$, the amount of nodes is bounded by 2^n which, even if quadratically smaller than the one for AOBF, is far larger than β. If to this we add that for a good heuristic, AOBF will open only few nodes per level that makes AOBeam and AOBF pretty close in execution time. In the last section, we will propose alternatives to solve this issue.

12 Conclusions & Future Work

Throughout this work we've gathered knowledge regarding inference tasks on graphical models. It's its generality what makes the field of graphical models such an interesting one. We've shown the implementation for probabilistic models in the form of Bayesian Networks but the techniques presented can be easily adapted to constraint networks, propositional CNF formulas, influence diagrams...

We've presented algorithms that can be divided in three sections: exact inference algorithms (Bucket Elimination), approximation inference algorithms (Mini-Bucket Elimination) and search algorithms in both its exact and approximation flavors. Their space and time complexities have also been a subject of study, understanding along the different sections how the underlying structures used to model the space for this algorithms affect its complexity. State-of-the-art algorithms for finding good orderings of pseudo-trees have been analyzed, including one of the most current papers on the topic (published in 2017) that offers a different approach to the task.

Proof of this understanding is one of the contributions of this work: an analysis of the trade-offs between the width and height of the pseudo-trees. Although inconclusive, this work is one of the first (if not the first) to analyze the trade-offs generated between the height and width of the pseudo-trees and it's a first step in the path to possibly proving an existing relation that can be abstracted for its use in selecting among them. A more in-depth study needs to be made in the different domains to see if any generalization can be made. The examples chosen for the study are small examples, execution times range from 1 to 5 minutes, where the differences may not be so easy to appreciate as in larger execution times in more complex problems (a case-per-case study of the trade-offs for width and height could be done with larger examples as they would mitigate the effect of OS jitter and randomness). Finally, the study was only made with AND/OR Best First Search, so the obvious extension would be: a) Testing the same orderings with other algorithms and seeing if all of them are affected equally or if they have to be evaluated

separated due to the presence of different patterns. b) Executing the same orderings with parallelization of the code rather than with caching.

With respect to search algorithms, special focus has been given to Best-First AND/OR Search and AND/OR Branch and Bound as they are the state-of-the-art algorithms to answer optimization tasks on graphical models. Their implementation in DAOOPT has been understood and has yielded the development of two new search algorithms for optimization in AND/OR search spaces that try to tackle the weaknesses of the previously mentioned ones. As for this implementation, there's a necessary improvement in term of the amount of nodes open. One possible alternative would be decreasing the values of β and α along the execution. A more complex one, would be making sure that we have as many as β alternatives open for problems at a given level in the tree. There's also a need to study the algorithms' performance with weaker/stronger heuristics than MBE as it will give us a better insight of which cases favor each of them. For the anytime implementation, we could devise an imlpementation to make it incremental. Namely, right now each iteration of the algorithm would be performed independently without taking into account previous results or calculations which leads to an obvious redundancy in the calculations and makes Anytime AND/OR Beam Search take longer than necessary. This would also allow us to check if all the nodes in the graph have been traversed evaluated, thus, giving us a guarantee of completeness and optimality for those cases. Once this implementation is complete, we could evaluate its performance against another anytime scheme, Anytime-AND/OR Best First Search. Intuitively, cases where the calculation of a single iteration is short (both algorithms find a quick solution each time they start a search process) AAOBF will find better solutions as it will consider the complete search space. On the contrary, examples that require a lengthy calculation per iteration may favor the implementation of Beam Search since it prunes the search-spaces more aggressively and the accumulated results may be better due to a larger amount of iterations.

Finally, both studies will be helped by executing the different algorithms in CPU clusters as programmed tasks whose results can be obtained as many hours later as needed for the termination and whose memory resources are larger than those of a personal computer.

I would like to personally thank my research tutor at UCI, Rina Dechter, for giving me the opportunity of working in my Bachelor's Thesis with her, a referent in the field of graphical models, as well as for the guidance and help provided through the development of this work. I would also like to thank William Lam, part of Rina Dechter's research group at UCI, for his advise on adhering the implementations to DAOOPT. I would also like to thank Carlos Linares, professor at UC3M, for his introduction to heuristic search that initiated my interest in the topic and for encouraging learning inside and outside of a classroom setting. Finally, special thanks to the University of California and Carlos III University for their agreement in the Study Abroad program that allows students to have such an irreplaceable experience.

13 References

[Vadlamudi et al. 2013] S.G. Vadlamudi, Sandip Aine and P.P. Chakrabarti. Incremental Beam search. Information Processing Letters 113 (2013) 888–893

[Zhang, 1998] Complete Anytime Beam Search. Weixiong Zhang. In AAAI-98 Proceedings.

[Liu and Ihler, 2011] Q. Liu and A. Ihler. Bounding the partition function using Hölder's inequality. In International Conference on Machine Learning (ICML), pages 849–856, 2011

[Marinescu et al., 2014] Radu Marinescu, Rina Dechter, and Alexander Ihler. AND/OR Search for Marginal MAP Search. In UAI 2014.

[Marinescu and Dechter, 2009] Radu Marinescu and Rina Dechter. Memory intensive and/or search for combinatorial optimization in graphical models. Artif. Intell., 173(16-17):1492–1524, 2009.

[Rollon and Dechter, 2010] Emma Rollon and Rina Dechter. New mini-bucket partitioning heuristics for bounding the probability of evidence. In AAAI, 2010.

[Marinescu and Dechter, 2005] R. Marinescu and R. Dechter. And/or branch-and-bound for graphical models. In International Joint Conference on Artificial Intelligence (IJCAI), pages 224–229, 2005.

[Kask and Dechter, 2001] K. Kask and R. Dechter. A general scheme for automatic generation of search heuristics from specification dependencies. Artificial Intelligence, 2001.

[Arnborg, 1987] S. Arnborg, D. Corneil and A. Proskourowski. Complexity of finding embeddings in a k-tree. In SIAM Journal of Discrete Mathematics, 1987. 8:277-284

[Arnborg, 1985] S. A. Arnborg. Efficient algorithms for combinatorial problems on graphs with bounded decomposability - a survey. BIT Computer Science and Numerical Mathematics. Volume 25 Issue 1 1985, pp. 2-23.

[Kask et al., 2011] Kalev Kask, Andrew Gelfand, Lars Otten and Rina Dechter. Pushing the Power of Stochastic Greedy Ordering Schemes for Inference in Graphical Models. In AAAI 2011.

[Gogate and Dechter, 2004] Vibhav Gogate and Rina Dechter. A Complete Anytime Algorithm for Treewidth. Proceedings of the Twentieth Conference on Uncertainty in Artificial Intelligence (UAI2004)

[Dechter, 1996] Rina Dechter. Bucket Elimination: A unifying framework for probabilistic inference. In "Uncertainty in Artificial Intelligence", UA196, 1996, pp. 211-219

[Lam et al., 2016] William Lam, Kalev Kask, Rina Dechter and Javier Larrosa. On the Impact of Subproblem Orderings on Anytime AND/OR Best-First Search for Lower Bounds. In Proceedings of the European Conference on Artificial Intelligence 2016 (ECAI 2016).

[Otten and Dechter, 2010] Lars Otten and Rina Dechter. Towards Parallel Search for Optimization in Graphical Models. In Proceedings of ISAIM'10, Fort Lauderdale, FL, USA, January 2010

[Dechter, 2013] Rina Dechter. Reasoning with Probabilistic and Deterministic Graphical Models: Exact Algorithms.Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan & Claypool Publishers 2013, pp. 21-22.

[Allen and Darwiche, 2003]. D. Allen and A. Darwiche. New advances in inference by recursive conditioning. In Proceedings of the Nineteenth Conference on Uncertainty in Artificial Intelligence, 2–10. 2003.

[Pearl, 1988] Judea Pearl. Probabilistic Reasoning in Intelligent Systems. Morgan Kaufmann, 1988.

[Dechter and Rish, 2003] Dechter, R. and Rish, I., "Mini-Buckets: A General Scheme for Bounded Inference" In "Journal of the ACM", Vol. 50, Issue 2: pp. 107-153, 2003.

[Lam, 2017] William Lam. "Advancing Heuristics for Search over Graphical Models" Ph.D Thesis, 2017. pp. 130-163

[Hasfsteinsson, 1991]. H.Hasfsteinsson, H.L Bodlaender, J.R. Gilbert and T. Kloks. Approximating treewidth, path-width and minimum elimination tree-height. In Technical report RUU-CS-91-1, Utrecht University, 1991.

[Carmeli et al., 2017]. Nofar Carmeli, Batya Kenig and Benny Kimelfeld. Efficiently Enumerating Minimal Triangulations. PODS'17, May 14-19, 2017.