Static and dynamic speedup techniques for the Junction Tree Algorithm

Statische und dynamische Optimierungsverfahren für den Junction Tree Algorithmus

Bachelor-Thesis von Michael Kutschke

August 2011

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Informatik
Fachgebiet Softwaretechnik

Eingereicht am 30. August 2011

Prüfer: Prof. Dr. Mira Mezini

Betreuer: Dipl.-Inf. Marcel Bruch, MSc. Johannes Lerch

Eigenständigkeitserklärung

Hiermit versichere ich die vorliegende Bachelor-Thesis ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 30. August 2011.

Michael Kutschke

Abstract

Uncertainty exists in many domains for different reasons. Therefore, expert systems often have to rely on probabilistic reasoning techniques to solve their tasks. This uncertainty is often encoded in bayesian networks. The Junction Tree Algorithm [LS88] is one of the algorithms for exact inference in such networks. Huang and Darwiche [HD96] presented a very detailed guide on that algorithm. However, some points were left unspecified. This thesis elaborates on their work, presenting several static and dynamic speedup techniques, of which some originate from ideas found in Huang and Darwiche's paper, while others don't. We also evaluate our implementation by comparing it to SMILE [Dru99], a bayesian reasoning library which also implements the Junction Tree Algorithm based on Huang and Darwiche's guide.

Contents

1 Introduction

Uncertain knowledge exists in many problem domains and is commonly addressed through the modeling as bayesian network. Several algorithms for inference in such networks exist, one of which is the Junction Tree Algorithm by Lauritzen and Spiegelhalter[LS88]. Exact inference in bayesian networks is an NP-hard problem, as it includes SAT as special case [Pum01]. Still, for many bayesian networks, exact inference is tractable. Among the algorithms for exact inference, the Junction Tree Algorithm is one of the most efficient[Bis06]. As we will show in

this thesis, the algorithm leaves a lot of potential for optimization in implementations. Huang and Darwiche [HD96] presented a detailed, self-contained guide to the algorithm for helping developers wanting to implement the algorithm. However, several issues were left unspecified or did not appear at all in their document. This thesis elaborates on their work. We present an implementation that incorporates several ideas found in Huang and Darwiches guide. Further optimizations are discussed, presented, and evaluated. These optimizations include both static and dynamic techniques. The implementation is then compared to other implementations, of which SMILE [Dru99] is as well based on the guide of Huang and Darwiche.

1.1 Probability Theory

We will start by introducing the notions and terms needed to understand the Junction Tree Algorithm.

Given a random variable X, which can take values x from some arbitrary set, we call that set the *domain* of X and denote it by $dom(X)$. $P(X) : dom(X) \rightarrow \mathbb{R}$ is called the probability distribution of X.
All probability distributions P(X) have the following properties:

$$P(x) \geq 0 \tag{1}$$

$$\int P(x)\,\mathrm{d}x = 1 \tag{2}$$

A probability distribution P(X,Y) over more than one random variable is called *joint probability distribution* of X and Y. Note that $dom(X)$ is not necessarily equal to $dom(Y)$. For a joint Probability Distribution P(X,Y) it holds that

$$P(X) = \int P(X, y)\,\mathrm{d}y \tag{3}$$

This operation is also called *marginalization* over Y.

In the following, if a Probability Distribution P(**X**) appears, **X** stands for a set of random variables.

The *conditional probability distribution* P(**X** | **Y**) ('X given Y') is defined as

$$P(\mathbf{X}|\mathbf{Y}) = \frac{P(\mathbf{X}, \mathbf{Y})}{P(\mathbf{Y})} \tag{4}$$

Corollary, it is

$$P(\mathbf{X}, \mathbf{Y}) = P(\mathbf{X}|\mathbf{Y})P(\mathbf{Y}) \tag{5}$$

From equation (5), if extended to more than two random variables, we see that any joint probability distribution can be factorized by other (conditional) distributions over subsets of the original variable set. For example, it is

$$P(X,Y,Z) = P(X|Y,Z)P(Y,Z) = P(X|Y,Z)P(Y|Z)P(Z) \tag{6}$$

1.2 Bayesian Networks

Bayesian networks are a way to represent joint probability distributions, using the factorized form.

Definition 1.1 *A Graph $G = (V,E)$ consists of a set of vertices V and a set of edges $E \subseteq V \times V$. If the edges have a direction, G is directed.*

Definition 1.2 *Given a directed Graph G = (V,E), $(u,v) \in E$ (there is a directed edge from u to v). Then u is called parent of v, and v is called child of u. It is also parents(u) = $\{ w \mid w$ is parent of u $\}$*

Bayesian Networks are acyclic directed graphs, augmented with *belief potentials*. Such potentials are nothing more than real-valued functions over a set of variables, in this special case the conditional probability distributions P(v | parents(v)) . In a bayesian network, every vertex is associated to exactly one random variable and vice versa. We will therefore treat the vertices of a bayesian network as if it was the random variable it represents. Every vertex v has an assigned belief potential P(v | parents(v)). The bayesian network and its potentials are further defined by the network's global semantic.

Definition 1.3 *Global semantic of a bayesian network: Let* **U** *be a set of random variables. Then a bayesian Network G = (***U***,E) has to fulfill the following equation:*

$$P(\mathbf{U}) = \prod_{v \in \mathbf{U}} P(v|parents(v)) \tag{7}$$

In the context of this thesis, we will assume discrete random variables. The conditional probability distributions then become tables, having one entry for each combination of variable assignments. Figure 1 shows an example bayesian network with it's conditional probability tables (the potentials). Bayesian networks encode the dependencies, or more precise the independencies between random variables. This becomes interesting later on for some of the optimizations we present. We first need to introduce the notion of conditional independence:

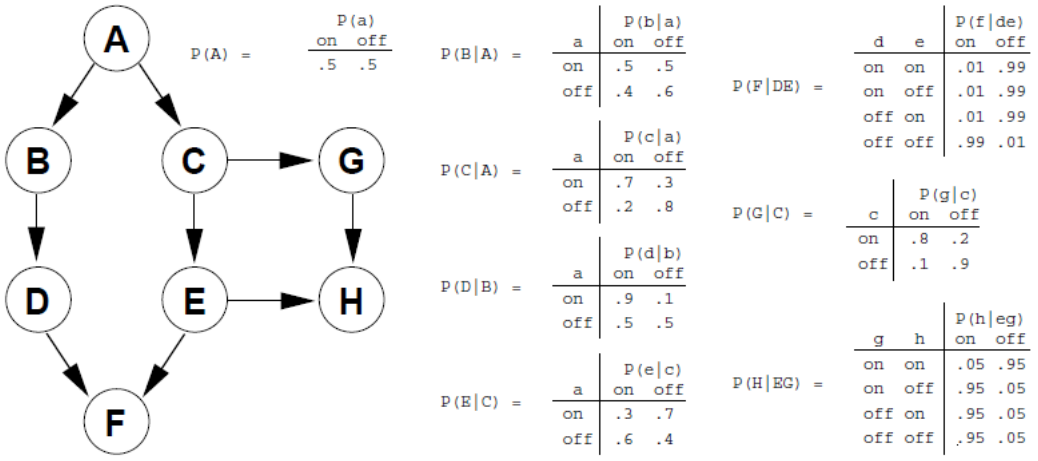


Figure 1: An example bayesian network [HD96]

Definition 1.4 *Two random variables X and Y are called* **conditionally independent given Z** *if*

$$P(X|Y,Z) = P(X|Z) \tag{8}$$

This is important as it allows the bayesian network to have a sparser structure (compared to the full factorization as, for example, in equation (6)). And this sparser structure leads to smaller conditional probability tables, hence less memory needed to save that network.

Given this notion of conditional independence, we can define an alternative (local) semantic for our bayesian network which can be shown to be equivalent with the global semantic, given the conditional probability tables. This local semantic will later help us to understand the discussed optimizations better.

First, we need to define *moralization* as it will make the definition of the local semantic easier, and will be needed anyway for the algorithm.

Definition 1.5 *The* **moral graph** *of a directed graph* $G = (V,E)$ *is an undirected graph* $G' = (V,E')$*, where*
$E' = E \bigcup_{u \in V} \{(v,w) | v \text{ and } w \text{ are parents of } u \text{ in } G\}$.
The process of turning a directed graph into it's moral graph is called **moralization**.

In other words, moralization means for every node, we connect it's parents to make them pairwise adjacent, and turn the graph undirected. Apart from the moral graph being an intermediate step in the Junction Tree Algorithm, it is useful as conditional independencies can easily be derived from it.

Definition 1.6 *Let Graph $G = (V, E)$. Let $A, B, C \subset V$ be non-empty and disjunct. Then B* **separates** *A and C if the removal of B makes it such that there are no two nodes $v \in A$ and $w \in C$ such that there is a path connecting v and w.*

Definition 1.7 *Local semantic: Assume we have a bayesian network. Two sets of random variables* **X** *and* **Y** *from the network are conditionally independent given* **Z** *if* **Z** *separates* **X** *and* **Y** *in the moral graph of the network. [Bis06]*

2 Definitions - Operations on belief potentials

As we see from equations (3) and (9), the only operations we need to compute the probability distribution of a random variable given only the potentials are marginalization and multiplication. In the following, as we only consider random variables with finite domains, we will substitute the integrals for marginalizations with sums. With finite domains, we can represent the potentials as tables having one entry for every function value. Except for marginalization and multiplication of potentials, all operations used in formulas in this thesis are assumed to be entry-wise.

Recall that the bayesian network is a representation of the factorized joint probability distribution (see equation (7)). Of course, a joint probability distribution can also be factorized by other functions on the random variables that, by themselves, are not necessarily probability distributions. In the following, we will not assume the belief potentials to be probability distributions. Also we will denote these belief potentials by $\phi_{\mathbf{X}} : dom(\mathbf{X}) \to \mathbb{R}$, meaning they are tables containing one real value for every combination of variable assignments of the variable set **X**.

$$P(\mathbf{X}) = \prod_{\mathbf{Y} \subseteq \mathbf{X}} \phi_{\mathbf{Y}} \tag{9}$$

In the following examples, A, B and C are random variables, and $a \in dom(A)$, $b \in dom(B)$ and $c \in dom(C)$.

2.1 Marginalization

Given a potential ϕ_{AB} over A and B, if we marginalize over A, we get a potential only dependent on B. Every table entry in that new potential ϕ_B is the sum over all the table entries of ϕ_{AB} where B has the value corresponding to the table cell.

$$\left(\sum_{A} \phi_{AB} \right)(b) = \sum_{a \in dom(A)} \phi_{AB}(a, b) \tag{10}$$

Example:

ϕ_{AB} =

A	B	
	f	t
f	0.1	**0.9**
t	0.2	**0.8**

$\sum_A \phi_{AB}$ =

B	
f	0.3
t	**1.7**

$$\left(\sum_A \phi_{AB}\right)(B=t)$$

$$= \phi_{AB}(A=f, B=t) + \phi_{AB}(A=t, B=t)$$

$$\mathbf{1.7 = 0.9 + 0.8}$$

2.2 Multiplication of potentials

When we multiply two potentials $\phi_{\mathbf{X}}$ and $\phi_{\mathbf{Y}}$, we get a new potential over the union of $\mathbf{X}$ and $\mathbf{Y}$. To compute the value of a table cell in $\phi_{\mathbf{X}}\phi_{\mathbf{Y}}$, we need to multiply those values of $\phi_{\mathbf{X}}$ and $\phi_{\mathbf{Y}}$ where the variables take the values corresponding to the cell.

$$\left(\phi_{AB}\phi_{BC}\right)(a,b,c) = \phi_{AB}(a,b)\phi_{BC}(b,c) \tag{11}$$

Example:
Let dom(A) = dom(B) = dom(C) = { t, f }.

ϕ_{AB} =

A	B	
	f	t
f	0.1	**0.9**
t	0.5	0.5

ϕ_{BC} =

B	C	
	f	t
f	0.5	0.5
t	0.0	**1.0**

$\phi_{AB}\phi_{BC}$ =

A	B	C	
		f	t
f	f	0.05	0.05
f	t	0.0	**0.9**
t	f	0.25	0.25
t	t	0.0	0.5

$$(\phi_{AB}\phi_{BC})(A=f, B=t, C=t)$$

$$= \phi_{AB}(A=f, B=t)\phi_{BC}(B=t, C=t)$$

$$0.9 = 0.9 * 1.0$$

3 The Junction Tree Algorithm

As we said in the introduction, exact inference in bayesian networks is an NP-hard problem. However, inference in (poly-)trees can be done in time linear to the size of the bayesian network. A polytree is a Graph where each pair of vertices is connected by at most one path.

To efficiently perform inference in arbitrary networks, one can restructure the network to get a (poly-)tree. Then, the inference can be done in time linear to the size of that network. The NP-hardness has not disappeared, as the new network's size is in the worst case exponential in size of the original network.

The Junction Tree Algorithm does this restructuring. From the bayesian network, a secondary structure, the **Junction Tree**, is built. The Junction Tree is a tree-shaped network, in which inference can be performed efficiently.

3.1 Properties of the junction tree

The algorithm constructs a so-called junction tree, which is a tree with the following properties:

clusters: each vertex stands for a set of variables of the original bayesian network. We call them **clusters**

sepsets: the edges between two vertices are marked with the intersection of both variable sets, and are called separating sets, or **sepsets**

running-intersection property: for every pair of clusters u and v, it holds that any clusters on the path between u and v contain $u \cap v$. This can be seen in figure 2. For example, D is contained in every cluster on the path between DE and DF. This property is directly related to the local semantic of bayesian networks, as we will explain later. running-intersection is therefore important to conserve the network semantics.

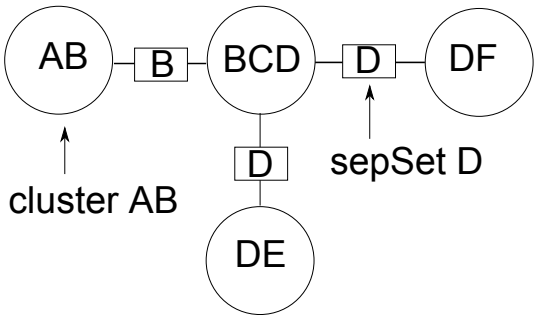


Figure 2: Example junction tree. We can see the running intersection property: For example, D is contained in every cluster on the path between DE and DF

For each cluster **X** and each sepset **S** in the junction Tree, there is an assigned belief potential $\phi_{\mathbf{X}}$ or $\phi_{\mathbf{S}}$ over the variables in **X** or **S** respectively. These are chosen in a way to satisfy the following equations (where **U** denotes all variables, **X** denotes a cluster, and **S** a sepset incident to that cluster, and $\phi_{\mathbf{X}}$, $\phi_{\mathbf{S}}$ the assigned potentials, respectively)[HD96]:

$$\sum_{\mathbf{X}\setminus\mathbf{S}} \phi_{\mathbf{X}} = \phi_{\mathbf{S}} \tag{12}$$

$$P(\mathbf{U}) = \frac{\prod_i \phi_{\mathbf{X}_i}}{\prod_j \phi_{\mathbf{S}_j}} \tag{13}$$

Also, for each cluster (or sepset) **X** it needs to hold that $\phi_{\mathbf{X}} = P(\mathbf{X})$. If this holds, we can easily compute the probabilities of any variable V as

$$P(V) = \sum_{\mathbf{X}\setminus\{V\}} \phi_{\mathbf{X}} \tag{14}$$

using any cluster or sepset **X** that contains V [HD96].

3.2 Building the junction tree

To build the junction tree, the bayesian network B needs to be moralized first. Moralization means, for every node, connecting its parents in the bayesian network pairwise. Note that while the bayesian network is a directed graph, the moralized graph is undirected.

After moralization, the graph needs to be triangulated, which means adding edges such that there is no cycle in the graph with more than three edges that does not contain a smaller cycle (a 'shortcut', so to say). Figure 3 shows an example of a moralization and subsequent triangulation. In the triangulated graph, the maximal cliques will form the junction tree. Or, in other words, the junction tree is a tree of cliques, and the sepsets are the nodes shared by 'adjacent' cliques. This last sentence will hold if and only if the running intersection property holds, therefore if we lose that property, we lose the original structure at the same time and cannot perform inference in a consistent fashion anymore.

Definition 3.1 *A Graph $G' = (V', E')$ with $V' \subseteq V$ and $E' \subseteq E$ is called subgraph of G. G is called fully-connected if $E = V \times V$. A clique in G is a maximal fully-connected subgraph of G.*

To perform the triangulation, we delete vertices from the Graph one by one, connecting all their neighbors pairwise at the same time. As we thereby create cliques (including the deleted vertex), the triangulation and the clique identification can be performed simultaneously. Such a created clique is maximal if it is not contained in a formerly identified clique. We will formulate this as lemma:

Proposition 3.2 *Given Graph G = (V,E). Suppose for every vertex $v_i \in V$ we do the following:*

1. *connect the neighbors of v_i such as to form a clique c_i*
2. *delete vertex v_i*

c_i is a maximal clique in the triangulated graph of G if and only if it is not contained in a formerly identified clique c_j $(j < i)$.

Proof Note a clique will be maximal if it is not contained in *any* other clique. Note also that besides the c_i, there can not be other maximal cliques. This is because after step i, no edges are added to vertices $1 - i$, thus no new maximal cliques can be formed that are not detected.

Now, it is trivial that if c_i is contained in c_j $(j < i)$, then c_i is not maximal. The point is that no c_i can be contained in a c_k with $k > i$. If v_k is not adjacent to v_i at step i of the algorithm, then clearly c_k will not contain v_i and therefore does not contain c_i either. If v_k is a neighbor of v_i, then either c_k is contained in c_i or there is a node w that is neighbor to v_k but not to v_i. It is therefore not in c_i. Then c_k does not contain c_i. □

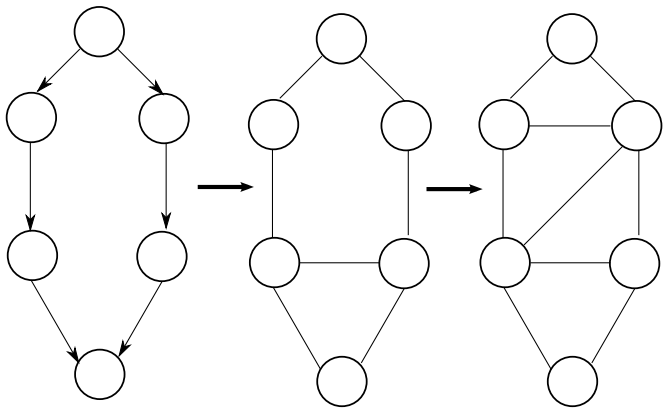


Figure 3: the original network, the corresponding moralized graph and the graph after triangulation

We left the order in which the vertices are to be eliminated unspecified. However, the ordering is crucial to minimize the amount of additional edges added for the triangulation and thus has an impact on how big the cliques will become. The size of the cliques, on their turn, have direct impact on how big our potential's tables will become. As finding the perfect elimination ordering is NP-complete [Yan], we need heuristics for this subproblem. As in the course of this algorithm, the preparation of the junction tree is a one-time effort, we choose a rather expensive heuristic and in each step, find the node leading to the least added edges if eliminated [HD96]. (Other heuristics would be possible)

We now have the vertices of our junction tree. To choose the sepsets in a way fulfilling the running-intersection property, we compute a maximal spanning tree, using the number of variables as edge cost measure, breaking ties by considering the product of the domain sizes of the variables of the concerning sepset (which would be the size of our sepset potential table if we chose that sepset). It has been shown in [Jen88] that a maximal weighted spanning tree of the sepsets is also a Junction Tree. Figure 4 is an example to motivate this. We see three (maximal) cliques: ABC, BCD and BDE. If we choose BD and BC as sepsets, we end up with a valid junction tree. However, if we chose to connect ABC and BDE first, by adding B as sepset, we will either violate the tree property or the running intersection property.

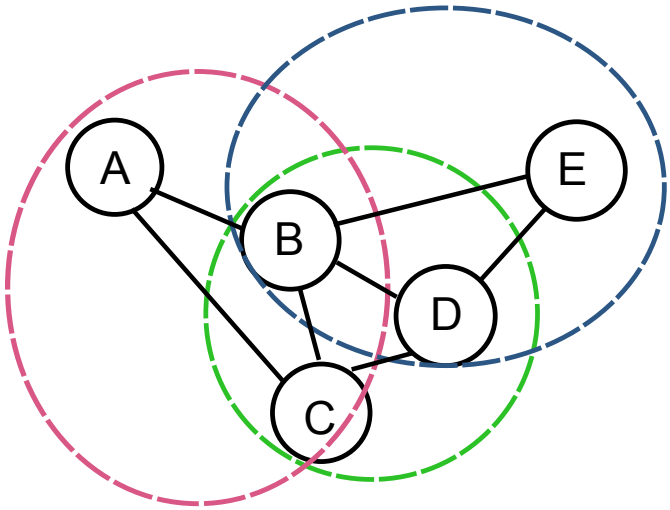


Figure 4: Example triangulated graph with three cliques: ABC, BCD, BDE.

3.3 Initializing the junction tree

To initialize the junction tree, we first initialize all potentials (clusters and sepsets).

$$\phi_{\mathbf{X}} \leftarrow 1 \tag{15}$$

Then, we multiply each conditional probability table of the original bayesian nodes into the potential of a cluster that contains the node's variable as well as it's parents. We call one these cluster the home cluster of that node (or it's corresponding random variable). $(V \cup parents(V)) \subseteq \mathbf{X}$:

$$\phi_{\mathbf{X}} \leftarrow \phi_{\mathbf{X}} P(V|parents(V)) \tag{16}$$

After this, equation (13) is satisfied, as we have

$$P(\mathbf{U}) = \frac{\Pi_i \phi_{\mathbf{X}_i}}{\Pi_j \phi_{\mathbf{S}_j}} = \prod_{v \in \mathbf{U}} P(v|parents(v)) \tag{17}$$

Equations (12) and (14) are not satisfied. For Equation (14) and (12) to be satisfied, we need local message passes or belief propagation. This is done as follows:

One cluster X is chosen arbitrarily (we will discuss this choice more in detail later), from which we start the procedure COLLECT-EVIDENCE[HD96]. We call X the **propagation root**.

COLLECT-EVIDENCE(X):

1. mark X
2. for every unmarked neighbor Y of X, call COLLECT-EVIDENCE(Y) and pass a message from Y to X

Then, we unmark all clusters and the procedure DISTRIBUTE-EVIDENCE is called on X [HD96]:

DISTRIBUTE-EVIDENCE(X):

1. mark X
2. for every unmarked neighbor Y of X, pass a message from X to Y and call DISTRIBUTE-EVIDENCE(Y)

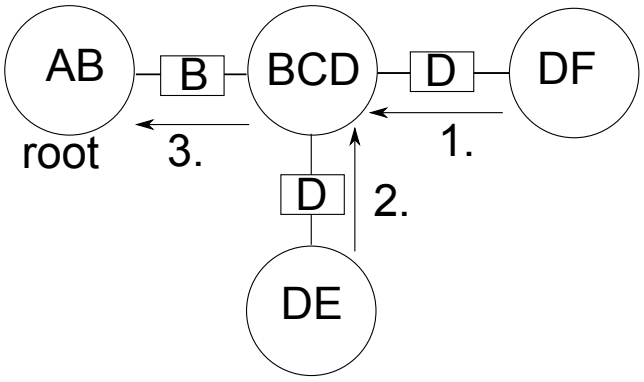


Figure 5: COLLECT-EVIDENCE(AB): Messages from the leaves are propagated towards the root

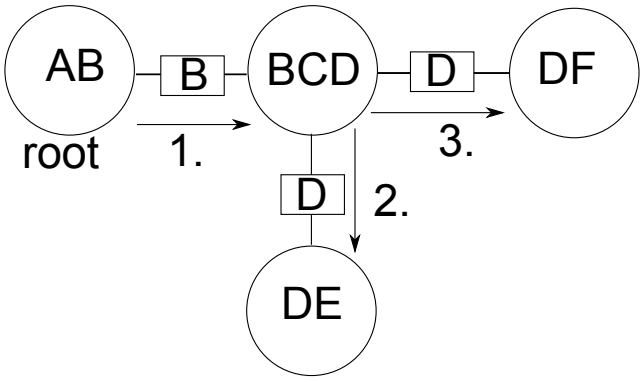


Figure 6: DISTRIBUTE-EVIDENCE(AB): Messages from the root are propagated towards the leaves

An example of this scheme is shown in Figures 5 and 6: Assuming we chose AB as the propagation root, we start by calling COLLECT-EVIDENCE(AB). This leads to messages being passed from the leaves of the tree towards the root. After that, we call DISTRIBUTE-EVIDENCE(AB),

propagating messages from the root to the leaves. After both calls, every cluster has received the necessary informations from every other cluster in the tree.

A single message pass from X to Y is performed as follows: Let S_{XY} be the sepset connection X and Y.

$$\phi_{S_{XY}}^{old} \leftarrow \phi_{S_{XY}} \tag{18}$$

$$\phi_{S_{XY}} \leftarrow \sum_{X \setminus S_{XY}} \phi_X \tag{19}$$

$$\phi_Y \leftarrow \phi_Y \frac{\phi_{S_{XY}}}{\phi_{S_{XY}}^{old}} \tag{20}$$

Whenever $\phi_{S_{XY}}^{old}(r) = 0$, it can be shown that $\phi_{S_{XY}}(r) = 0$, so we simply set $0/0 = 0$ in this case.

A message pass conserves the invariant of equation (13).

After the calls of COLLECT-EVIDENCE(X) and DISTRIBUTE-EVIDENCE(X), equation (14) holds and we can thus compute the beliefs we are interested in.

3.4 Evidence

Most of the time, we are not only interested in some P(X), but we observe assignments for some variables **E**, the 'evidence', and want to know P(X|**E**). In order for evidence to be taken into account, we make sure that in the message passes and the marginalization, only the entries in the potential tables are used that would not become zero if we multiplied the potentials with the evidence vectors or likelihoods defined as [HD96]:

$$\Delta_X(r) = \begin{cases} 1 & if\ evidence(X) = r \\ 0 & else \end{cases}$$

That way, we compute P(X,**E**), which is just an unnormalized version of the distribution P(X|**E**) we are really interested in.

For purposes explained in a later subsection, instead of integrating this into the initialization, we initialize the junction tree, store the resulting state and use it as starting point for the belief updates, which themselves again consist of a call to collect-evidence and distribute evidence after including the evidence in the potentials. Due to the nature of the message pass procedure, this works, as once $\sum_{X \cap Y} \phi_X = \sum_{X \cap Y} \phi_Y$, a message pass does not change any state. The junction

tree therefore has some kind of ‘equilibrium’ state to which it will converge as long as only evidence gets added. If evidence is removed or changed, we need to reinitialize the potentials and start over [HD96].

The next section explains the algorithm used to determine the necessary entries, making it unnecessary to actually multiply the likelihoods into the potentials.

4 Static and dynamic optimizations

The Junction Tree Algorithm not only is efficient by itself, it also leaves potential for further optimizations. In this section, we present different optimizations. Some are static, meaning independent of the current evidence. Others are dynamic, and exploit information about the current evidence. Huang and Darwiche [HD96] also describe optimizations based on single queries, but we did not investigate those further as we assumed an use case that makes querying of multiple variables given the same evidence necessary, in which case query-based optimization only helps in special cases.

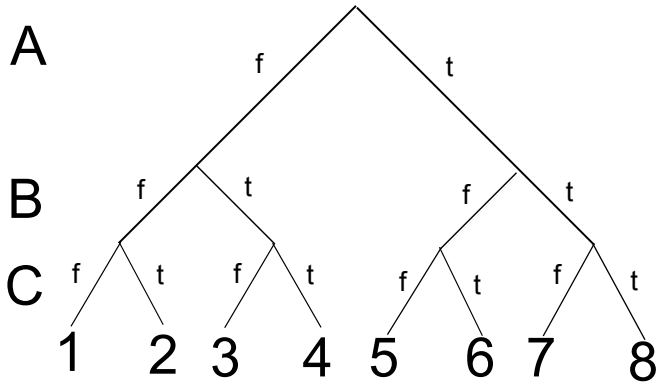
4.1 Evidence shrinking

Huang and Darwiche [HD96] describe an optimization they call ‘evidence shrinking’. The idea is to only use those elements of the potential table which are consistent with the current evidence, as the others are necessarily zero. We present, after introducing an alternative representation of the belief potentials, an algorithm to perform that form of optimization, along with a data structure to efficiently store which entries need to be visited. This works dynamically, dependent on the evidence, with time and space complexity linear to the number of variables in the potential.

A potential is nothing more than a table having one dimension per variable in the cluster, as we saw in the introduction. While the table is a useful way of thinking about the potential, we can also think of it as a decision tree as is shown in Figure 7. When we perform an operation on the potential, we typically visit the leaves of the decision tree in a sequential order. In the following, we describe how we can determine which leaves of the tree need to be visited. We call this the *valid* leaves.

One of the properties of such a decision tree that can be used to efficiently compute the entries that are necessary to visit during computation is that on any stage, all subtrees have the same structure.

Our goal will be to develop a data structure that efficiently stores which leaves of the decision tree need to be visited. We call this data structure a **cut** and call the algorithm for determining the valid leaves cutting.



A	B	C	
		f	t
f	f	1	2
f	t	3	4
t	f	5	6
t	t	7	8

Figure 7: conditional probability table along with it's representation as decision tree

Let us consider some examples first. Look at Figure 8 now. Suppose we observe $A = t$. Then we only need to use leaves 5-8 in operations.

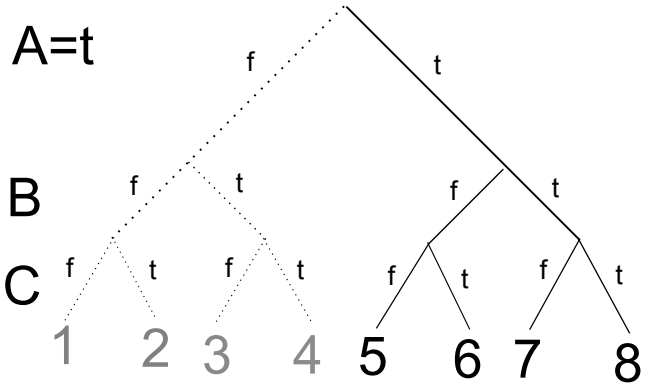


Figure 8: Valid leaves if $A = t$ is observed (5-8)

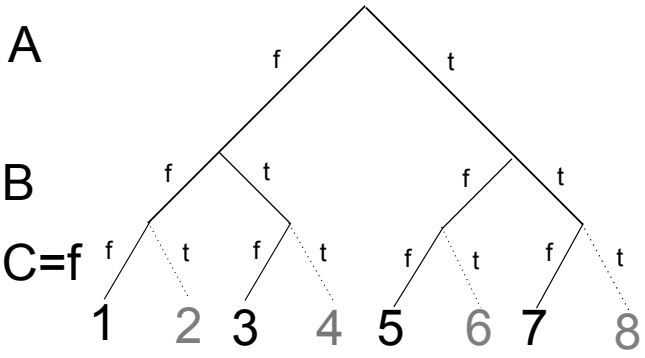


Figure 9: Valid leaves if $C = f$ is observed. To get the valid leaves, we just need to skip every other leaf.

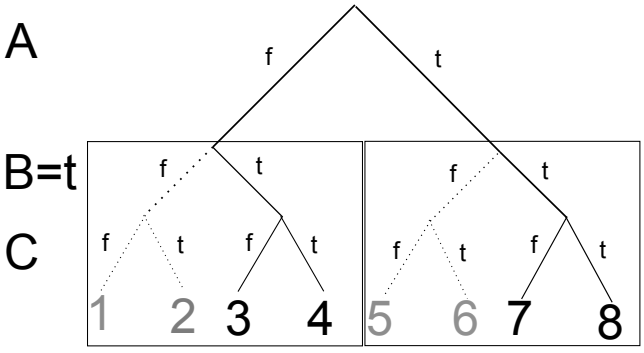


Figure 10: Valid leaves if $B = t$. The boxes denote how the tree is decomposed and recursively cut.

Suppose we now additionally observe $B = f$. We then only need leaves 5-6. We see from these two examples, that if, starting at the root, subsequent dimensions of our decision tree are

observed, only the range of the valid leaves changes. A data-structure should therefore save which **range** of leaves is valid.

Suppose now we don't observe A and B anymore, but we observe $C = f$ (see figure 9). Then leaves 1,3,5 and 7 are valid. If we now also observe $A = f$, it is only 1 and 3. We see, if we observe subsequent dimensions *from the leaves*, only the number of leaves we need to skip in between valid leaves changes. In addition to the range of valid leaves, the data-structure should therefore also know how many leaves can be **skipped** in between valid leaves when traversing sequentially.

While these two things are already very expressive and powerful, they can not reflect every possible situation we can encounter. Consider now we only observe $B = t$ as shown in Figure 10. There is no combination of range and skip specification that will yield all and only valid leaves. We therefore recursively decompose the decision tree (shown as boxes in figure 10) and cut the subtrees as described above. We decompose the tree starting at the inner observed dimension. Note how each subtree has exactly the same structure. Therefore we only need to cut one of them. Later, when we want to know which entries are valid, we just use the computed cut for every one of the equivalent subtrees. A cut therefore also needs to know the cut of it's subtrees, and how many subtrees there are, so we can recover the information lost because of only storing the cut of one subtree.

For example (again figure 10), the two cuts would store following information:

Main Cut	
range	1-8
skip	0
subtrees	2

Subtree cut	
range	3-4
skip	0
subtrees	-

These three concepts or **range**, **skipping** and recursive decomposition / cutting are expressive enough to handle all possible situations.

The algorithm thus works as follows:

1. set how many leaves have to be skipped by considering observed dimensions, starting at the leaves, until an unobserved dimension is hit. Also set the range accordingly if needed (for example, if we need to skip all odd leaves, our range has to start at 2 instead of 1)

2. compute the range of valid leaves, by considering observed dimensions, starting at the root, until an unobserved dimension is hit.

3. descend until an observed dimension is hit again. This only increases the number of subtrees.

4. if the dimension we stopped at in step (3) has not been considered in step (1), decompose: cut the subtree by setting the skipped leaves (don't change) and the range corresponding to the subtree rooted at the found observed dimension (from (3)) and for that subtree, repeat from (2)

This cut algorithm is linear in the number of variables in the cluster, that is logarithmic in the total number of table entries. It is functionally equivalent (for hard evidence) to the SELECT-operator proposed in [BF05], and shares its benefits, except for not being able to handle soft evidence. In the same way as the SELECT-operator does, with cuts we don't need to multiply our potentials with evidence vectors as described in section 3.4. It is also an efficient way to store which leaves are necessary for computation, as it takes only space linear in the number of variables, instead of linear in the (reduced) table size as [HD96] propose.

4.2 Numerical stability

Numerical stability can become an issue when dealing with small probabilities, which appear for example when the network is dense and big clusters in terms of number of variables are generated, but also if there exist long chains in the junction tree or when dealing with highly improbable evidence. To overcome this, one can also compute the message passes, marginalization etc. in the log domain. Let's consider the most important function, for example, the message pass. When using log-values, the formulas become:

$$\phi_{S_{XY}}^{old} \leftarrow \phi_{S_{XY}} \tag{21}$$

$$\phi_{S_{XY}} \leftarrow log \sum_{X \setminus S_{XY}} exp(\phi_X) \tag{22}$$

$$\phi_Y \leftarrow \phi_Y + (\phi_{S_{XY}} - \phi_{S_{XY}}^{old}) \tag{23}$$

Here, instead of handling the special case 0/0 as in equation (20), we set $-\infty - (-\infty) = -\infty$.

The log sum exp does not already provide the needed numerical stability by itself, but it can be easily transformed to yield the more stable formulation:

$$max \leftarrow max(\phi_X) \tag{24}$$

$$\phi_{S_{XY}} \leftarrow max + log \sum_{X \setminus S_{XY}} exp(\phi_X - max) \tag{25}$$

Because of the log sum exp, this is a lot slower in terms of computation than the original formulation without logarithms. To overcome this, one can combine both forms in the same

junction tree. The only care that needs to be taken is to transform the values of the sepsets at points where one cluster is computed in the log domain and it's neighboring cluster in the normal domain. As a heuristic, we chose the cluster size to determine whether a cluster should be computed in the log domain. It should be noted, however, that as said above, the cluster size is not the only source of numerical instability. Improbable evidence may also cause numerical instability, as throughout the algorithm, we work with the unnormalized P(X,**E**) distribution, whose values will get very small when P(**E**) is small. The threshold for the log computation is therefore network-specific.

4.3 Optimization of the propagation

For propagation, we need to choose a node from which we call COLLECT-EVIDENCE and DISTRIBUTE-EVIDENCE. We call this the propagation root and consider it, for the time of the propagation, as the root of the junction tree. It's children are then roots of their respective subtrees, and so on.

As we wrote in section 3.4, we can save the initial state of the potentials and use it as a starting point for further belief updates. It can be seen that because of this alone, we do not need to call COLLECT-EVIDENCE on a cluster X if and only if the unmarked subtree rooted at X does not contain an observed variable. For example, see figure 11. Bold characters stand for observed variables. If we observe neither D nor E, we do not need to collect information from DE, as nothing has changed in that subtree compared to the initial state where we observed no variables at all.

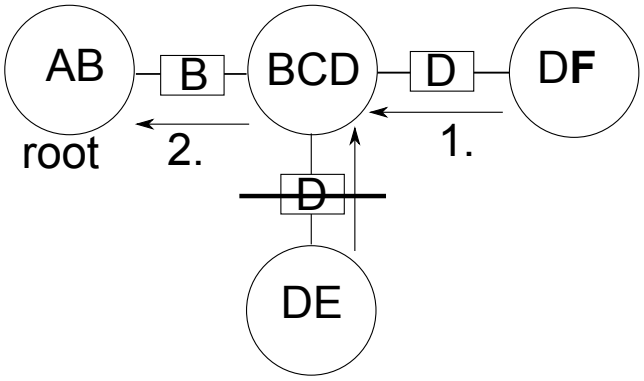


Figure 11: Situation where collection can be skipped for cluster DE if neither D nor E are observed (bold font means observed).

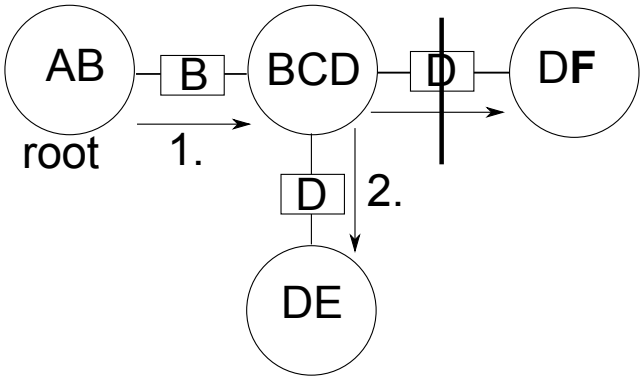


Figure 12: Situation where distribution can be skipped for cluster DF if F is observed and DF is not the query cluster for variable D

Remember equation (14), which tells us we can get the probabilities of a variable V by marginalizing of any potential that contains V. Assume now that for every random variable

V, we select the cluster from which we will query the marginals. We call this the **query cluster** of V. Then, we do not need to call DISTRIBUTE-EVIDENCE on a cluster X if and only if there is no cluster in the unmarked subtree rooted at X which is a query cluster for an unobserved variable. It is obvious that as we know the probability of the observed variables, we do not need to propagate information into a subtree from where we will never retrieve it (in the form of querying a potential in that subtree). Note that this destroys the consistency of the junction tree, but at no cost for us, except choosing the query clusters beforehand. Looking at figure 12, we see that if the query cluster of D is DE, and we observe F, then we do not need to propagate information back to DF when executing DISTRIBUTE-EVIDENCE.

Because of these optimizations, it makes sense to choose a cluster as propagation root that contains observed variables, as this choice makes it more likely that some subtree does not need to be visited during COLLECT-EVIDENCE. Note that we can not chose two different roots for collection and distribution. For example, looking at figure 11 again, we would not have needed any message passes for COLLECT-EVIDENCE had we chosen DF as propagation root.

Furthermore, any message pass from a cluster X to cluster Y can be skipped if all variables in $S_{XY} = X \cap Y$ are observed as we can easily introduce the evidence directly in clusters X and Y. The marginalization would then just turn the sepset S_{XY} into a single number that would only rescale the valid entries. To give this a better theoretical foundation, note how the running intersection property of the junction tree and the local semantic of the bayesian network have to do with each other: If we consider any sepset S_{XY}, it separates the variables in the subtree of X from those in the subtree of Y (without the variables of S_{XY}, of course). This can also be seen in figure 4, where the sepSet BC separates ABC from BCD and BDE. The variables in the different subtrees are therefore conditionally independent, given S_{XY}. If we therefore observe S_{XY}, we do not need to pass messages from X to Y and vice versa.

Note that this last optimization assumes we are only interested in $P(X|\mathbf{E})$. If we wanted to compute $P(X, \mathbf{E})$, we would still need to pass a message from X to Y if S_{XY} is observed, as we again destroy the consistency of the junction tree by violating equation (12).

4.4 Cluster-sepset mappings

The critical operations in terms of runtime, which are the multiplications and marginalizations, can be significantly sped up by precomputing a mapping between the node factors and the sepset factors which statically maps the node factor array indices to the corresponding index in the sepset factor array that is used when performing multiplication or to which is written when doing marginalization of the node factor. This is already described in [HD96] and called *cluster-sepset mapping*.

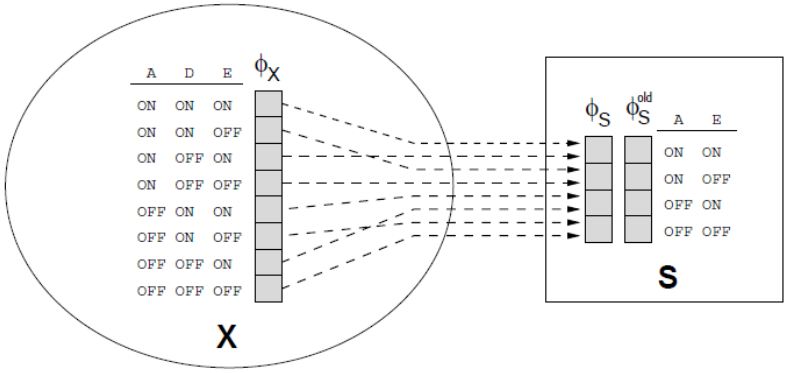


Figure 13: cluster-sepset mapping [HD96]

These mappings have two appealing properties. First, they work bidirectional: the same entries used when multiplying are exactly the ones where the corresponding elements need to be summed when marginalizing. Second, it is possible to share the mappings. This greatly reduces the actual number of mapping instances due to the natural structural coincidences that occur in most bayesian networks (many variables have the same domain sizes etc.). Sharing the prepared operations reduced the amount of prepared operations on most of the bayesian networks we used for evaluation by more than 50%, for some even more than 80% (see evaluation section), greatly reducing the impact this optimization has on the potential to work on big networks with many nodes.

As for querying a variable, we also need to marginalize, we can also store mappings for these operations. In the following, we will not separate between those two kinds of mappings.

Note that not only can we store these mappings, but we also need to compute the **cut** only once per factor per propagation and reuse it in both marginalization and multiplication.

5 Evaluation

This section deals with the evaluation of a java implementation of the algorithms presented in this paper. We call this implementation *Jayes*. For this purpose, we compared our implementation with two other bayesian reasoning libraries, SMILE [Dru99] and SamIam[Dar11].

The reason for choosing SMILE for performance comparison is that their implementation of the Junction Tree Algorithm is also based on [HD96], therefore we see it as some kind of reference implementation, even if we don't have access to the sources. SMILE is in fact a native C++ library, we use it together with a Java wrapper.

SamIam, in contrast, is a pure Java library written by Adnan Darwiche, one of the authors of [HD96]. This makes it interesting for us to compare to. We compare to the

il2.inf.jointree.UnindexedHuginAlgorithm class, which is a basic implementation of the Junction Tree Algorithm. It can be seen as baseline for our approach.

Also, we evaluate a variant of our algorithm not performing propagation optimization (see section 4.3) to be able to make a statement about the performance impact this optimization has. Note that since we did not encounter numerical instabilities, we set the threshold for log computation to 100, effectively disabling it altogether.

The goal of our evaluation is to check whether

1. Jayes computes exact beliefs in the sense that, assuming SMILE computes correct, our deviation from SMILE's computed beliefs are insignificant
2. Jayes is an improvement over the basic Junction Tree Algorithm by showing significant performance benefit compared to SamIam
3. Jayes is competitive in the sense that it performs as well or better than SMILE and SamIam in most cases
4. our proposed propagation scheme optimizations significantly improve performance
5. sharing cluster-sepset mappings significantly reduces memory requirements

5.1 Test setup

As the complexity of exact inference is highly dependent on the bayesian network's structure, we chose to evaluate on 8 different networks (see 1), 7 of which are freely available on the internet and commonly used for evaluation (among others). These are alarm, andes, diabetes, hailfinder, Hepar II and win95pts. The eighth is called Lorg.eclipse.swt.widgets.Button, or short Button, and is a code recommendation model from the code recommenders project [ea11].

In Table 1, we list the networks' names, the number of nodes (variables) in each bayesian network, and the number of edges that connect them. We also give a short description from what domain those nets come from. The node and edges amount give an impression on how big and dense the network is. The number of edges divided by the number of nodes gives an average in-degree of the nodes. This can be an indicator whether exact inference is difficult or not, as the in-degree has impact on the potential table sizes. However, the structure, for example the number of states within variables, is more important than this kind of statistics, and this can not be reflected properly in the table.

To give an extreme example, the Button network is quite special. Actually, our algorithm was optimized to work well on that particular network (and similar networks). It's structure is shown in figure 14. It consists of three main variables, *Context*, *Availability* and *Pattern*. Context has 95

Name	nodes	edges	description
alarm [BSCC89]	37	46	alarm monitor for intensive care patients
andes [CGVD97]	223	338	coached problem solving
diabetes [AHB+91]	413	602	insulin adjustment
hailfinder [Abr96]	56	66	weather model for predicting hail
Hepar II [Oni03]	70	123	diagnosis of liver disorders
Button	85	85	java code recommendation system
pathfinder [HHN92]	109	195	diagnosis of lymph-node diseases
win95pts [smi11]	76	112	printer troubleshooting in Windows 95 (by Jack Breese)

Table 1: Evaluation Networks

states, Availability only 2, and Pattern has 748 states. Additionally, there are 82 variables corresponding to methods, each being binary. As one would expect, there is a significant difference in inference speed when Pattern and Context are observed opposed to when they are not. It is an extreme case of a shallow tree-like network that contains a cycle, and has a very dominant root(-cluster). The Junction Tree is dominated by the Pattern node which appears in every cluster and every sepset. However, even if the Button network is an extreme and somewhat special case, many networks have such dominant nodes, with either an extraordinary amount of states, number of connections to other variables, or both. We will discuss this more in-depth when we present our evaluation results.

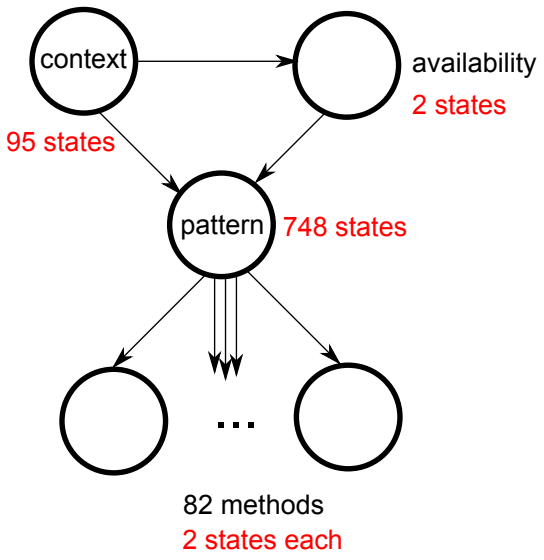


Figure 14: Button network's structure

As we wanted to test the performance of the algorithms with regard to the amount of observed variables, we chose to evaluate with test cases having 10-90% observed variables, with steps of 10%.

Because there are these dominant nodes, we wanted to make sure that when evaluating, we cover a broad range of observed variable combinations. To this end, we chose to evaluate on 1000 random variable assignments, sampled from the respective joint distribution. We then chose the observed variables randomly, with each variable having the same probability of being observed (equal to the percentage of variables we wanted to have observed). This way, if there are one or two dominant variables, we will get test cases where those variables are observed and others where they are not. In the following, whenever we mention a *test suite*, we mean the 1000 test cases associated to one network / observation probability combination.

Our experiments were performed on a 4-core/8-thread Intel Core i7-870 processor clocked at 2.93 GHz with $4 \times 32KB$ L1 data and instruction caches, $4 \times 256KB$ L2 cache, 8MB L3 cache, and 4096MB RAM running a 64-bit version of GNU/Linux (Kernel 2.6.32). The Java VM was a 64-bit Hotspot Server VM (build 20.1-b02) with Java version 1.6.

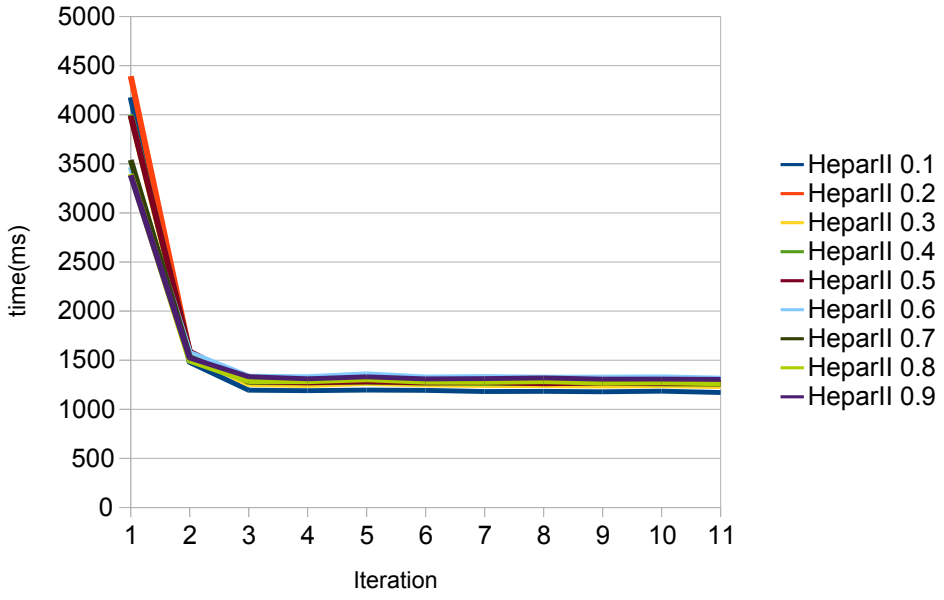


Figure 15: Time taken by the complete test suite on Hepar II

Because the Hotspot VM compiles bytecode over time, early executions are more expensive than later executions. We therefore executed our complete test suite 11 times in a row during each VM invocation, and only use the results of the last 8. This is motivated by figure 15, which shows that the first few iterations take significantly longer than the latter ones, and the latter

ones show amazingly constant execution times. Other networks than Hepar II show similar pattern. To avoid outliers, we evaluate the median of these 8 iterations for every test case. Since we have 1000 test cases per network/observation probability combination, it is highly unlikely that outside effects influence one test case more than half of the time.

5.2 Evaluation - Exact computation

For each test suite, we kept track of the maximal deviations the algorithms showed in comparison to SMILE. For the purpose of evluation, we assume SMILE to compute correct, that is exact, marginals. Of course for any algorithm, this cannot be asserted beyond doubt, due to rounding errors etc. All the algorithms we evaluated use double-precision floating point numbers.

5.2.1 Results

For diabetes, HeparII and pathfinder, Jayes' beliefs maximally deviates from SMILE's by an absolute 10^{-8}. For the other networks, this is in the range of $10^{-15} - 10^{-16}$. SamIam, however, showed serious numerical instabilities on the following configurations:

- alarm, 10%-90% observed variables: max. Error 0.99
- diabetes, 10% - 40% observed variables: max. Error 1.0
- Hepar II, 10% - 90%: max Error 0.9
- pathfinder, 10% - 90%: max. Error 1.0
- win95pts, 10% - 90%: max. Error 1.0

5.2.2 Interpretation

The results indicate our algorithm computes exact marginals, with errors which we feel are acceptable to the point that we would call them insignificant.

The errors shown by SamIam were unexpected, and odd especially in the case of the diabetes network. On diabetes, severe errors occurred when observing 10-40% of the variables, but were zero for the 50-90% test cases. This is odd because more observed variables lower the probability of the evidence, which would make computations more prone to numerical instability. Also, errors do not coincide with big networks, as there were no (significant) errors on the andes network at all. We did not have time to study this phenomenon closer, but as we only experienced similar numerical instabilities in early stages of our development, it might be interesting to investigate whether the sole fact of Jayes starting with an already initialized junction tree and entering the evidence in that tree as opposed to starting with an uninitialized tree avoids

numerical instability to a certain level. Also it would be interesting to evaluate whether our proposed log-scale computation would, for SamIam, result in correct results or if there is a deeper flaw in SamIam's algorithm or our use of it.

5.3 Evaluation - runtime

On order to evaluate performance, we measured the time needed for the libraries to update their beliefs and query each variable once. We could have included the time needed to set the evidence, but SMILE's wrapper induced significant overhead for th setting of the evidence. We therefore chose to not include it in what we see as the runtime of the algorithm, because what we were really interested in is the performance of the belief update. As an example, figure 16 shows that on the alarm network, the time for setting the evidence grows when more variables are observed, to the point of exceeding the time needed for the belief update. We suggest that this is due to the JNI overhead.

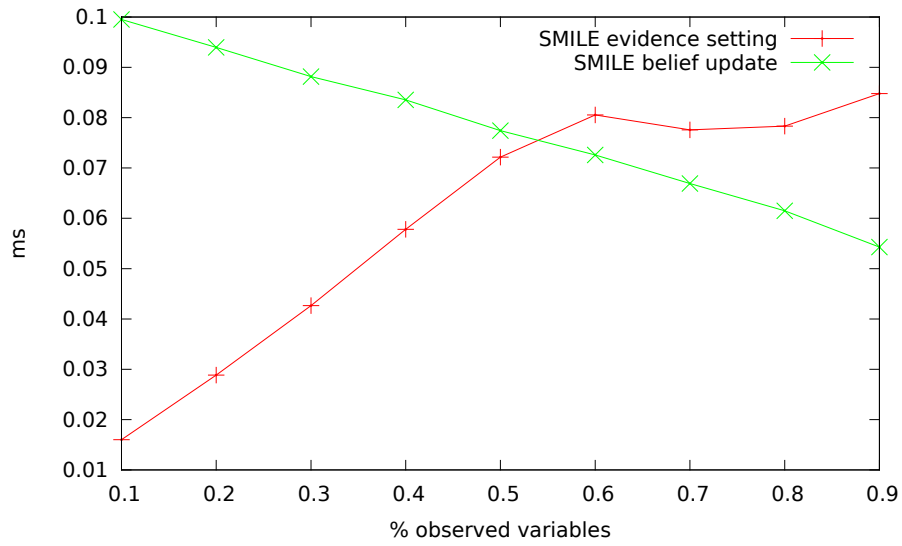


Figure 16: Comparison of time needed to set the evidence and performing the belief update for SMILE, on the alarm network.

5.3.1 Results

Figures 17 through 19 show some of the results of our evaluation. For every network, we plotted the mean execution times against the percentage of observed variables. We also show the mean amount of skipped message passes due to the different propagation scheme optimizations described in section 4.3. For clarity, we also show the execution time diagram a second time without the SamIam results to be able to have a higher resolution of the other results. The remainder of our evaluation results can be found in the appendix A.

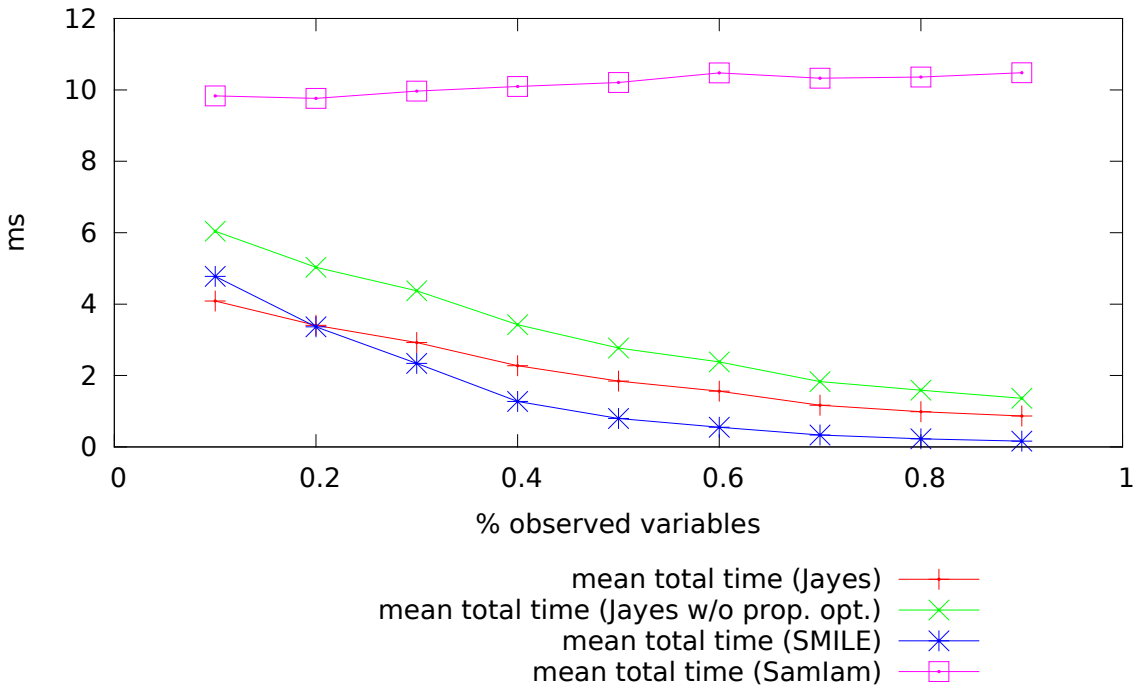


Figure 17: Evaluation results for the Button network

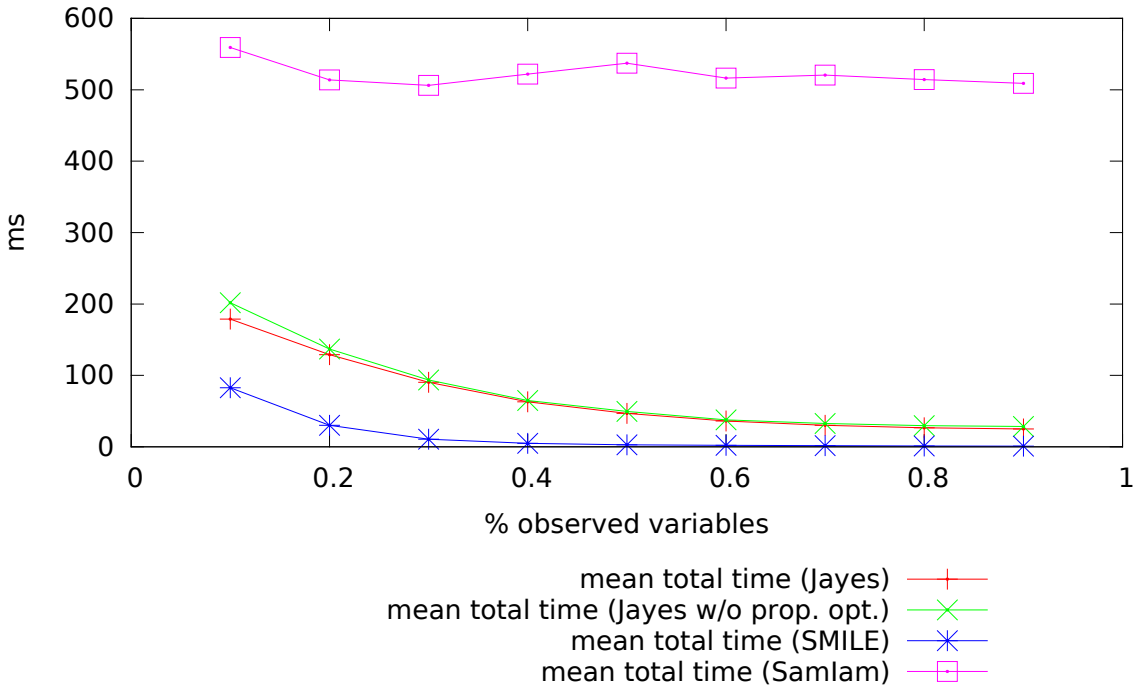


Figure 18: Evaluation results for the diabetes network

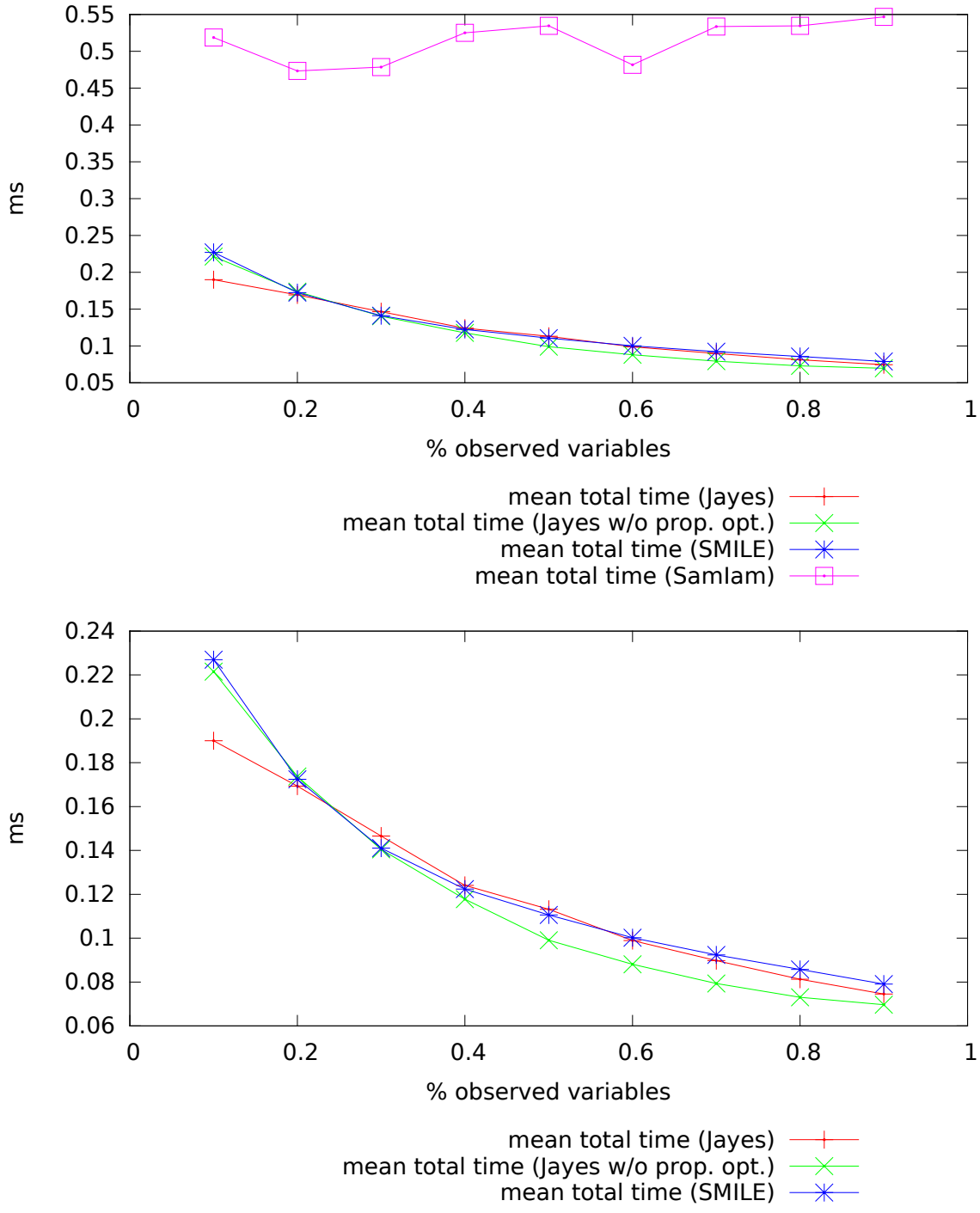


Figure 19: Evaluation results for the hailfinder network

On all networks, we see the Jayes variants are significantly faster than SamIam. This ranges from factor ~ 2 on smaller networks like alarm, HeparII and win95pts to factor $\sim 3-\sim 5$ on andes and diabetes. SMILE as well as Jayes show decreasing execution times with growing amount of observed variables while SamIam, due to lacking optimizations, exposes nearly constant execution times on almost all networks. As expected, Jayes (and SMILE) show decreasing execution times with growing number of observed variables.

Comparing SMILE and Jayes, we see that on alarm, Hepar II and win95pts, Jayes performs significantly better than SMILE. On hailfinder, it performs equally well. Note that on these networks, inference only takes less than a millisecond for all algorithms. On the other networks, SMILE performs better, by an approximately constant absolute value. This value falls slightly with increasing number of observed variables.

Analyzing further, we find that as expected, we find that execution times are highly dependent on *which* variables are observed. Figure 20 shows the extreme example Button. We find there are multiple "levels", corresponding to observed variable combinations. The lowest "level" coincides with observing the Pattern variable. Jayes does not show another level for observing both pattern and context, but SMILE and SamIam show such a level. Similar levels can also be observed in the pathfinder network (see figure 21). The diabetes network exposes a hinge that gets flatter with more observed variables, but since the hinge moves, this still indicates a dominant node. For comparison, see figure 22 for the Hepar II network, which clearly does not show this characteristic.

5.3.2 Interpretation

The results show that our algorithm performs significantly better than the basic Junction Tree Algorithm used by SamIam. They also show we come close to the performance of the well-established, native SMILE library. Especially interesting is the similarity of the mean execution time graphs of SMILE and Jayes. While this would suggest a constant overhead of Jayes compared to SMILE, the close look at the distribution of execution times (like in figures 20 and 22) shows this is not the case. Especially figure 20 shows that our algorithm performs significantly better than SMILE in special cases. This special case is non-observance of the major nodes in the Button network. This network, however, is quite special. On the pathfinder network, which only has one dominant node, Jayes is not faster than SMILE when the dominant node is unobserved, however, less performance is lost (relatively) than for SMILE.

Our results show that Jayes is indeed competitive and, on small networks or in special cases, even outperforms SMILE.

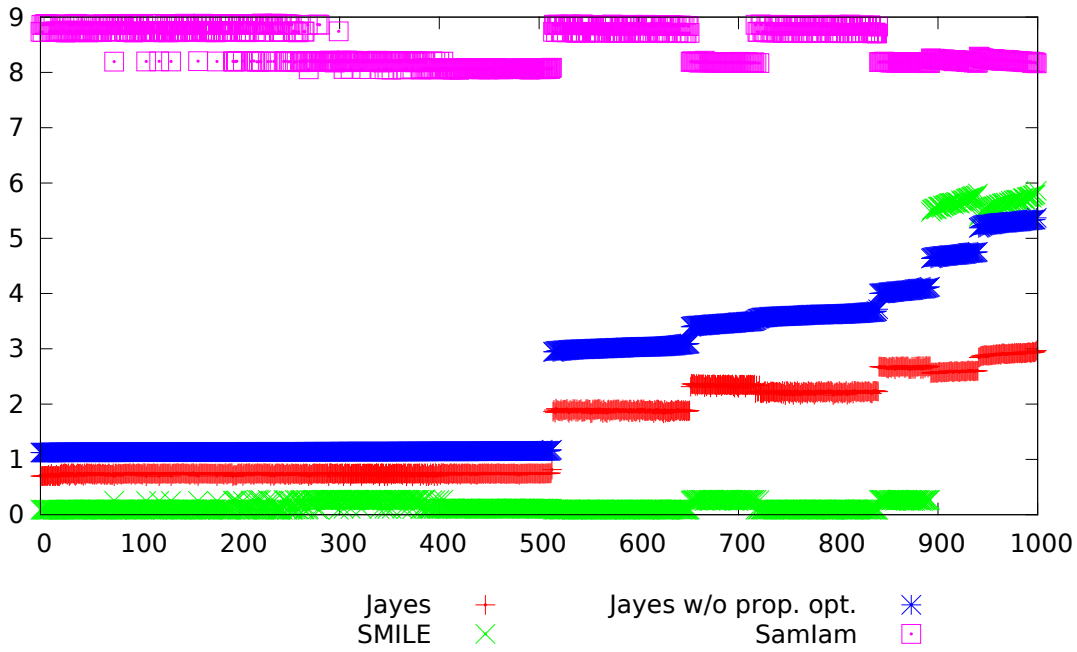


Figure 20: Execution time of the single scenarios, for the Button network with 50% observed variables, sorted by Jayes without propagation optimization

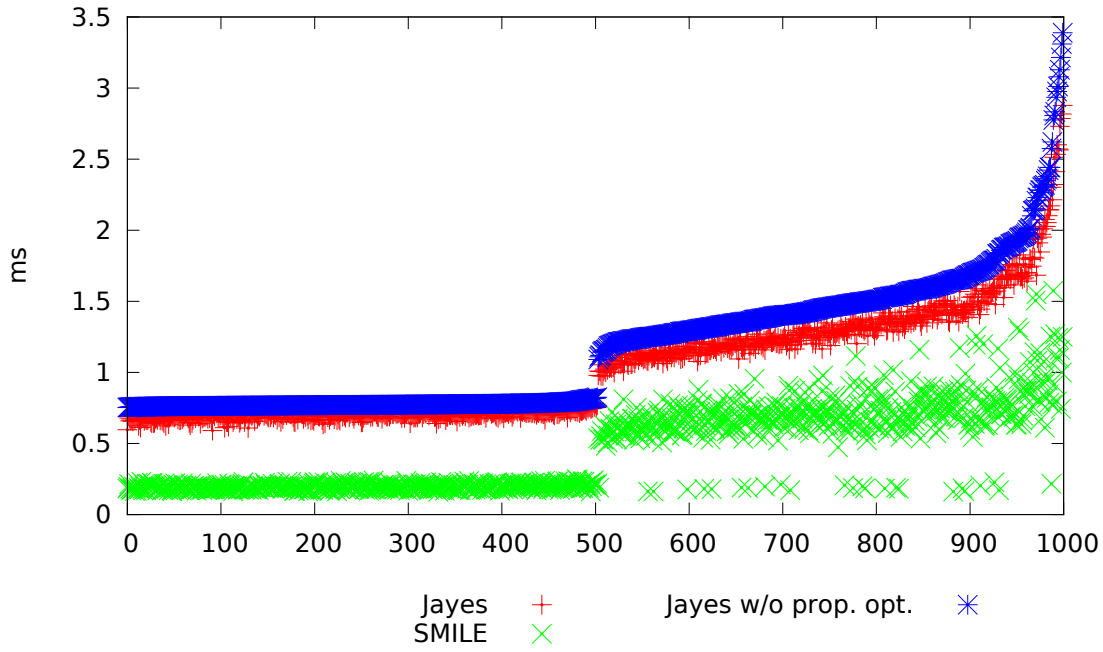


Figure 21: Execution time of the single scenarios, for the pathfinder network with 50% observed variables, sorted by Jayes without propagation optimization

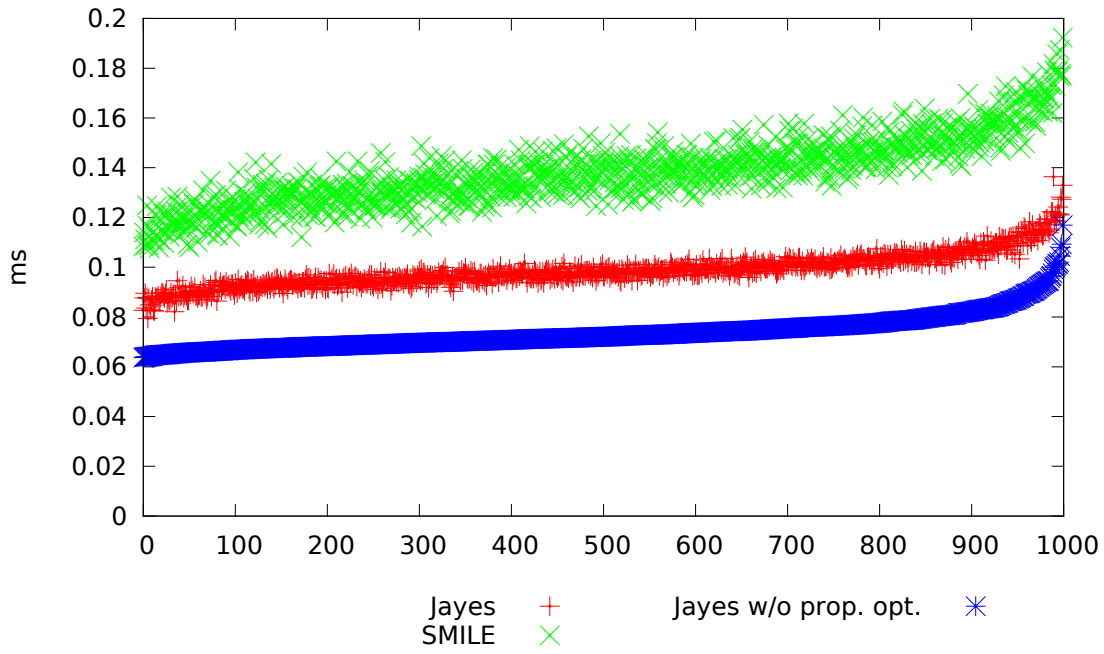


Figure 22: Execution time of the single scenarios, for the Hepar II network with 50% observed variables, sorted by Jayes without propagation optimization

5.4 Evaluation - Impact of propagation scheme optimizations

To get an image of the impact propagation optimization has, it is important to see how many messages actually get skipped. For every test case, we measured how many messages were skipped

- due to the optimizations of COLLECT-EVIDENCE,
- due to the optimizations of DISTRIBUTE-EVIDENCE,
- due to fully observed sepsets.

The amount of messages that would be sent if no optimizations are done is clearly twice the number of sepsets, as for every sepset, one message is sent for COLLECT-EVIDENCE, and one is sent for DISTRIBUTE-EVIDENCE.

We then compare this to the actual execution times that we measured.

5.4.1 Results

On all networks, the (mean) amount of messages that can be skipped grows strongly with increasing number of observed variables. As can be seen from the diagrams (see figure 23), this can be attributed to both the Optimization of DISTRIBUTE-EVIDENCE, and the skipping of

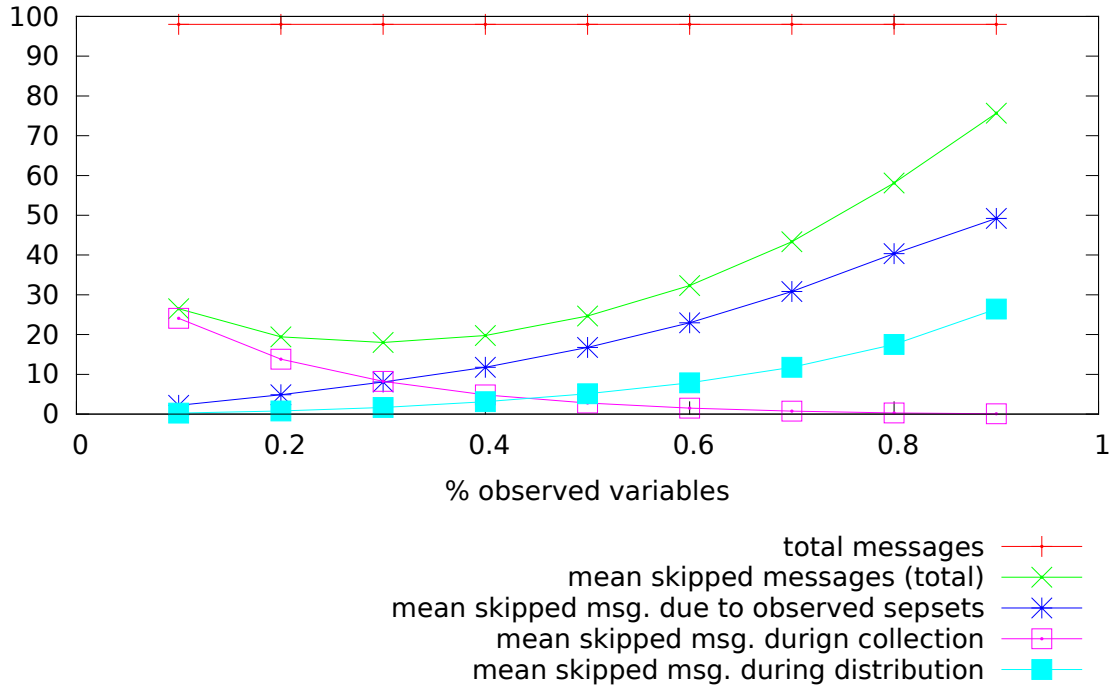


Figure 23: mean skipped messages for the win95pts network

messages due to fully observed sepsets (see section 4.3). The minimum of skipped messages is around 20%−30% of observed variables for all networks except Button, which does not show an explicit minimum. This minimum is $\sim 15\%$ of skipped messages for andes and diabetes, $\sim 20\%$ for win95pts, $\sim 25\% - 35\%$ for alarm, hailfinder, Hepar II and pathfinder. For all networks, this goes up to $\geq 70\%$ when we have 90% observed variables.

However, we don't see this result reflected very well in the actual execution time measurements. The Jayes variant not using the propagation scheme optimization in most cases was not significantly slower, sometimes even faster than the variant using the optimization (see, for example figure 22). The propagation optimization performed best, compared to the variant without this optimization, when there were only few variables observed.

5.4.2 Interpretation

We have seen that our propagation scheme optimization, while not showing significant impact on the execution time, covers a significant percentage of the messages. As message passing is the most costly operation of the belief update, an attempt for explanation would be that this indicates that checking whether a message needs to be sent takes similar time to actually sending the message. The reason for the message passes being as expensive as checking whether the message needs to be sent is, that with a growing number of observed variables, the costs of messages greatly decrease due to evidence shrinking. Especially, messages that are passed via

fully observed sepsets tend to be cheap. A noteable exception is the Button network: It's tree-like structure with many leaf nodes, one dominant node with hundreds of states and big sepsets is favored by the propagation scheme optimization, thus we see significant performance gain. This special structure is also reflected in the diagram showing how many messages are skipped, which differs greatly from the others.

Considering figure 20 again, it is interesting to see that apart from the first, big level which corresponds to observing pattern, there are at least six further levels. This clearly indicates that there is at least one method variable that significantly impacts the performance of the algorithms. It is there where Jayes and the variant not using propagation optimization show obvious differences.

On the other hand, these results at the same time show the potential of the propagation scheme optimization. If the costs needed to determine which messages can be skipped can be reduced significantly, this potentially would mean significant speed-up of the algorithm at the same time.

5.5 Evaluation - Sharing cluster-sepset-mappings

For each network, we measured how many cluster-sepset mappings can be saved if they are shared among sepsets. As we also use extra mappings for the querying of variables (because it accelerates querying), we also share those together with the mappings. We measured both together and won't separate between cluster-sepset mappings and „cluster-query mappings" from here.

5.5.1 Results

As mentioned before, cluster-sepset-mappings have the appealing property to be able to be easily shared via the flyweight pattern. For the networks from table 1, this results in a median 54% reduction in the number of distinct mappings. This is shown in figure 24. The figure also shows that this is very network dependent, and for certain networks, the reduction can be significantly higher.

5.5.2 Interpretation

Sharing cluster-sepset mappings comes at no cost, apart from initialization, and is able to reduce the amount of needed mappings drastically, dependent on the network. While the cluster-sepset mappings themselves are only a part of the total memory needed, the optimization itself becomes much more valuable as it's cost memorywise decreases. We have shown that on certain networks, the reduction can be much higher than 50%. Note that we only measured the mapping instances - as different sepsets have different sizes, so do the mappings. Therefore, a 90%

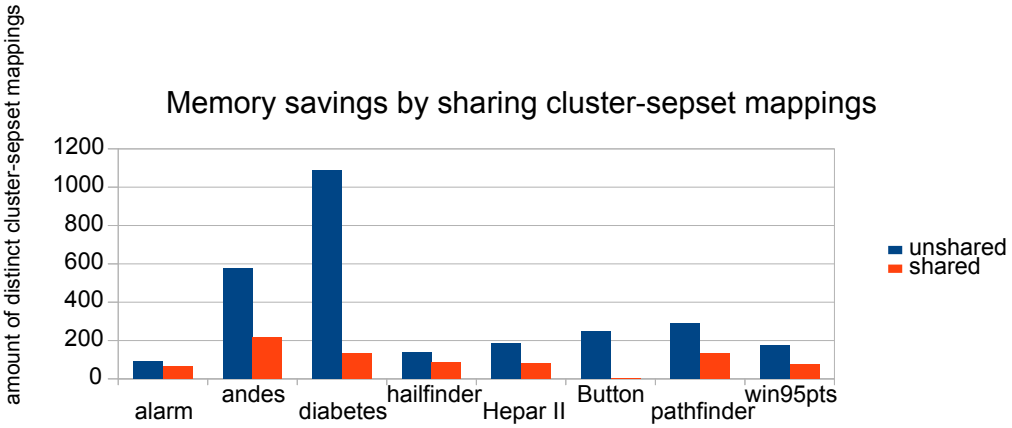


Figure 24: impact of sharing on the number of cluster-sepset-mappings

reduction of the amount of sepsets may not result in a 90% reduction of memory if big mappings cannot be shared.

6 Future Work

During evaluation, we have seen our implementation still has potential for improvement. We believe, however, that our propagation optimization is a first step for further optimization. In the following we present some of the ideas we have for potential future research.

Reducing the costs of propagation scheme optimization

We have seen that a significant portion of the messages can be **completely skipped**. However, our algorithm seemed to suffer from having similar costs of a message pass and the costs to determine whether it could be skipped. Further improvements should include reducing the costs of determining which messages can be skipped.

Leveraging a-priori knowledge

More research could also answer the question how to leverage **a-priori knowledge** about observed variables: in many applications, there is beforehand knowledge about which variables will be observed always or never, which variables are queried and which are not. Experiments we have performed show small, almost insignificant impact on performance when reordering the dimensions of the potential tables according to such prior knowledge. The impact gets significant (for the worse), however, if the ordering is intentionally done ‘wrong’. Further evaluation is needed to see if reordering variables according to prior knowledge leads to performance benefits (for example, through better cache using etc.). Another simple approach to leverage such knowledge in combination with our propagation optimization is use HornSAT reasoning for de-

termining which messages can be skipped. Every bayesian variable would be represented by a boolean variable stating whether it is observed or not. As one would know some of the boolean assignments beforehand, this would speed up the reasoning process because one would be able to eliminate a few clauses and literals beforehand. Note, however, that this again would be a time-space trade-off, as for every propagation root, the Horn clauses would look slightly different, and for pre-elimination, one would need to store the clauses for every possible propagation root. This means quadratic space overhead. We still feel it would be worth it to investigate this further.

Search in the space of junction trees

Another approach could include a search in the space of junction trees. For a given triangulated graph, there are **multiple valid junction trees**. Some of these may have properties favorable for our presented optimizations. First tests by just shuffling the sepsets before we compute the spanning tree showed that at least, as with the variable orderings mentioned before, there is potential for the worse. This suggests there may also be potential for the good. As our propagation optimizations highly depend on the structure of the junction tree, it may be worthwhile to investigate the potential alternate junction trees may offer. Combining this with a-priori knowledge about observations may allow for even better heuristics.

Further optimization of the message passing

Furthermore, especially DISTRIBUTE-EVIDENCE could easily be **parallelized**. First experiments have not shown much gain, but further investigation might be interesting. [ZMC11] have shown a way for parallelization single message passes on a GPU. It would be interesting to see whether and how some of our optimizations could accelerate this even further, especially whether evidence shrinking is still useful in a GPU-context and how it can be implemented and used on a GPU. Also, when calling DISTRIBUTE-EVIDENCE on a node that is connected to multiple sepsets that contain exactly the same variables, redundant marginalizations are done. This happens especially in networks like Button, but also in other networks. It should be researched how common this problem is and then evaluated if a change to the message passing to only marginalize once in such cases grants significant performance benefit.

Numerical stability

We did not evaluate the heuristic we chose for numerical stability - because we did not encounter numerical instabilities in the final version of Jayes, and did not have the time needed to find examples where this was an issue nor to properly evaluate and analyze the performance

impact of log-scale computation. It should be evaluated in which cases numerical instabilities do occur, and if our heuristic in these cases is useful to trade-off time for more stable computation. Also it would be interesting to see what impact log-scale computation really has. First tests have indicated that full log-scale computation takes between two and four times longer than computing with non-log-scale numbers, but these numbers need further evaluation.

7 Summary

We have presented the Junction Tree Algorithm and showed the optimization potential implementations can have. We also showed several optimizations, which span both static techniques, independent of evidence, and dynamic techniques, which depend on the evidence and leverage the information derived from knowing which variables are observed. These dynamic techniques can greatly enhance performance for certain queries by skipping certain nodes during collect/distribute-evidence (see section 4.3). We presented an alternative representation of the belief potentials to allow efficient storage of the informations needed to only visit relevant entries in the potential table as dynamic technique. On the other hand, static cluster-sepset-mappings help to find corresponding table entries in different tables, which greatly increases the speed of the basic marginalization and multiplication operations. Third, we presented and evaluated optimizations of the message propagation scheme itself, which allow skipping of a significant portion of the messages.

We evaluated a Java implementation of our presented algorithms, Jayes, by comparing it to SMILE and SamIam, other bayesian reasoning libraries. This showed our implementation is a major improvement over the basic Junction Tree Algorithm and can compete with well-established libraries for exact inference. The test results also show further potential for improvement, especially the propagation scheme optimizations. If these can be made faster, they may be able to provide significant performance increase.

References

[Abr96] J. Brown W. Edwards A. Murphy R. Winkler Abramson, B. Hailfinder: A bayesian system for forecasting severe weather. *International Journal of Forecasting*, 12(1):57–72, 1996.

[AHB+91] Steen Andreassen, Roman Hovorka, Jonathan Benn, Kristian G. Olesen, and Ewart R. Carson. A model-based approach to insulin adjustment. In M. Stefanelli, A. Hasman, M. Fieschi, and J. Talmon, editors, *Proceedings of the Third Conference on Artificial Intelligence in Medicine*, pages 239–248. Springer-Verlag, 1991.

[BF05] C. J. Butz and F. Fang. Incorporating evidence in bayesian networks with the select operator. In *In Proceedings of the 18 th Canadian Conference on Artificial Intelligence, Springer-Verlag, Victoria British-Columbia*, pages 297–301, 2005.

[Bis06] Christopher M. Bishop. *Pattern Recognition and Machine Learning*, pages 359–386,416f. Springer-Verlag, 2006.

[BSCC89] I. A. Beinlich, H. J. Suermondt, R. M. Chavez, and G. F. Cooper. The ALARM Monitoring System: A Case Study with Two Probabilistic Inference Techniques for Belief Networks. In J. Hunter, J. Cookson, and J. Wyatt, editors, *Second European Conference on Artificial Intelligence in Medicine*, volume 38, pages 247–256, Berlin, Germany, 1989. Springer-Verlag.

[CGVD97] C. Conati, A. S. Gertner, K. VanLehn, and M. J. Druzdzel. On-line student modeling for coached problem solving using Bayesian networks. In *Proceedings of the Sixth International Conference on User Modeling (UM–96)*, pages 231–242, Vienna, New York, 1997. Springer Verlag.

[Dar11] Adnan Darwiche. Samiam. `http://reasoning.cs.ucla.edu/samiam/`, July 2011.

[Dru99] M J Druzdzel. *SMILE: Structural modeling, inference, and learning engine and GeNIe: A development environment for graphical decision-theoretic models*, pages 902–903. American Association for Artificial Intelligence, 1999.

[ea11] Marcel Bruch et al. Code recommenders. `http://www.eclipse.org/recommenders/`, July 2011.

[HD96] Cecil Huang and Adnan Darwiche. Inference in belief networks: A procedural guide. *International Journal of Approximate Reasoning*, 15:225–263, 1996.

[HHN92] D. E. Heckerman, E. J. Horvitz, and B. N. Nathwani. Toward normative expert systems: Part i the pathfinder project. *Methods of Information in Medicine*, 31:90–105, 1992.

[Jen88] Finn V. Jensen. Junction trees and decomposable hypergraphs. Technical report, Judex Datasystemer, Aalborg, Denmark., 1988.

[LS88] S. L. Lauritzen and D. J. Spiegelhalter. Local Computations with Probabilities on Graphical Structures and Their Application to Expert Systems. *Journal of the Royal Statistical Society. Series B (Methodological)*, 50(2), 1988.

[Oni03] Agnieszka Onisko. *Probabilstic Causal Models in Medicine: Application to Diagnosis of Liver Disorders*. PhD thesis, Institute of Biocybernetics and Biomedical Engineering, Ploish Academy od Science, March 2003.

[Pum01] Sam J. Pumphrey. Solving the satisfiability problem using message-passing techniques, 2001.

[smi11] Genie/smile network repository. `http://genie.sis.pitt.edu/networks.html`, July 2011.

[Yan] Mihalis Yannakakist. Computing the minimum fill-in is NP-complete.

[ZMC11] Lu Zhen, Ole J. Mengshoel, and Jike Chong. Belief propagation by message passing in junction trees: Computing each message faster using GPU parallelization, 2011.

A Appendix - Evaluation results (runtime)

A.1 alarm

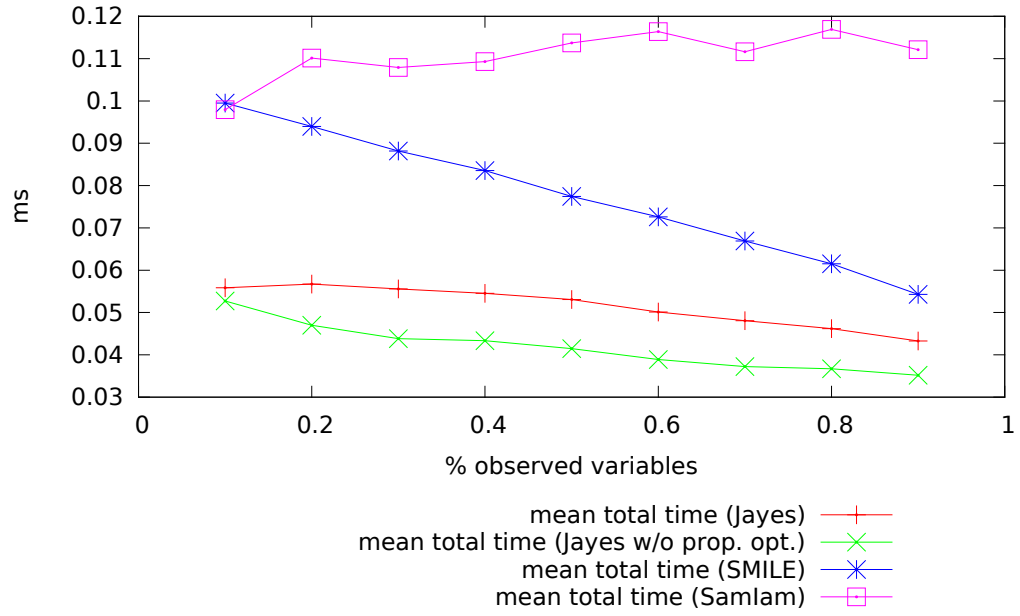


Figure 25: Evaluation results for the alarm network

A.2 andes

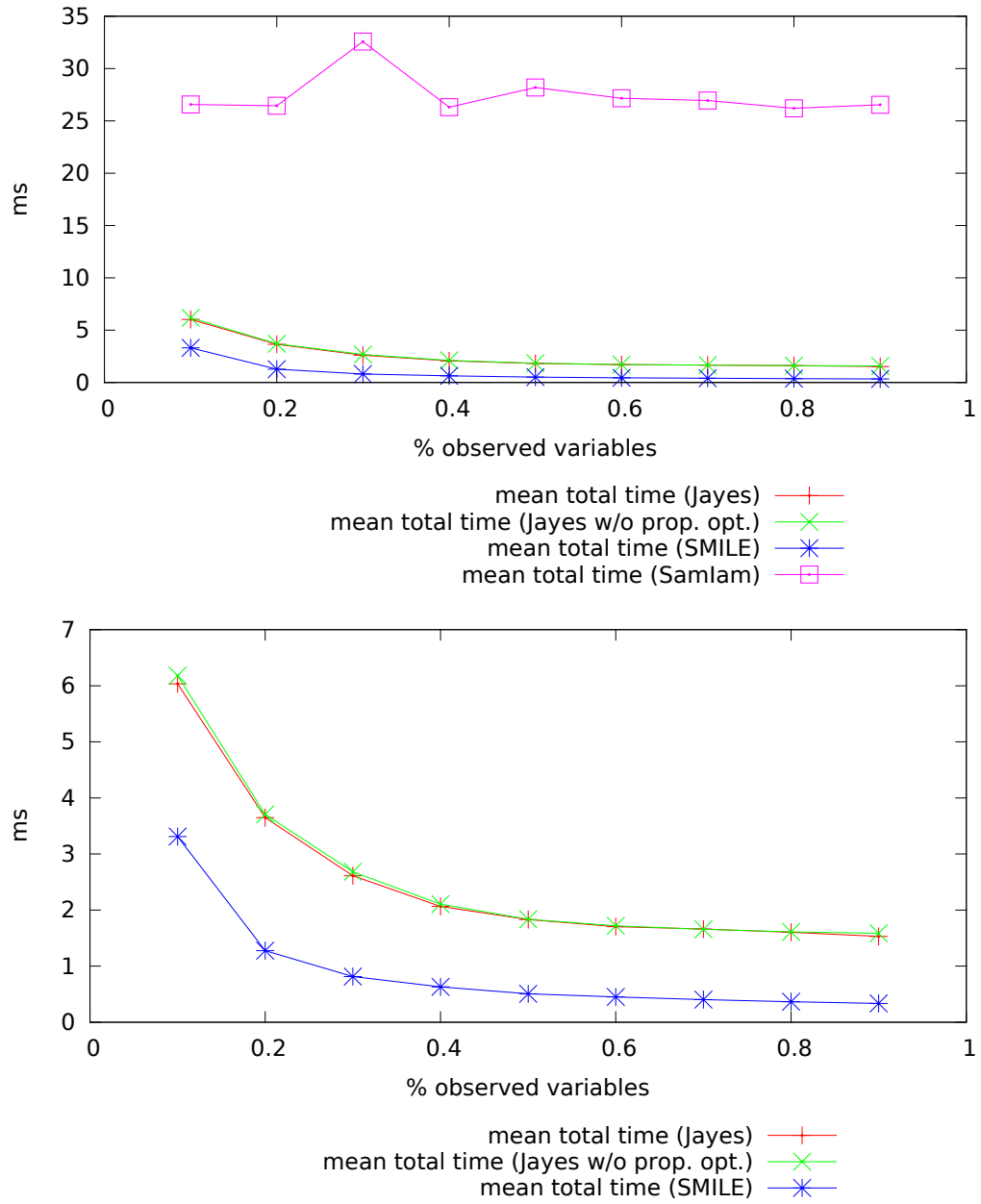


Figure 26: Evaluation results for the andes network

A.3 Button

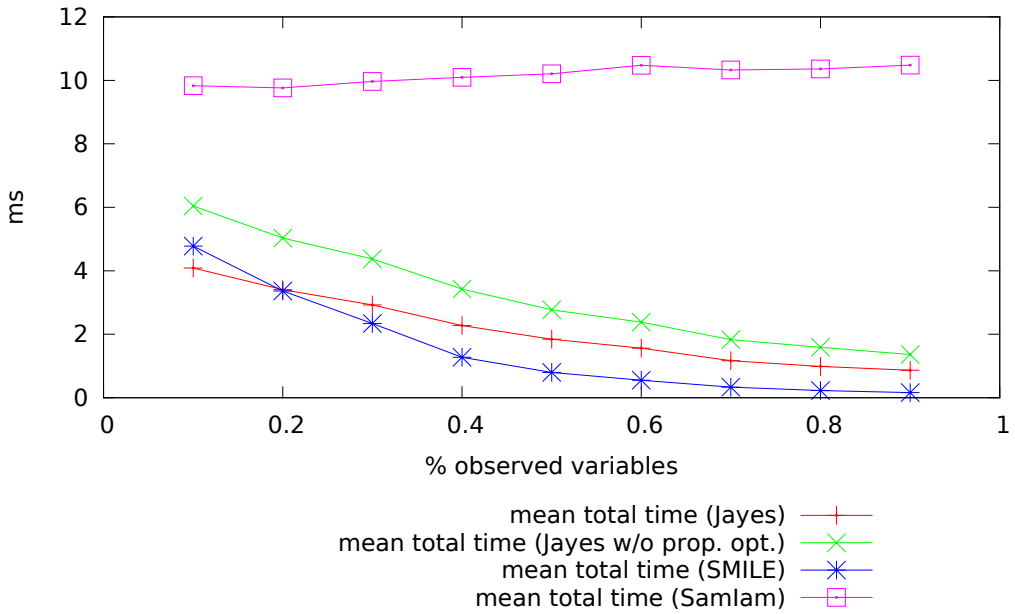


Figure 27: Evaluation results for the Button network

A.4 diabetes

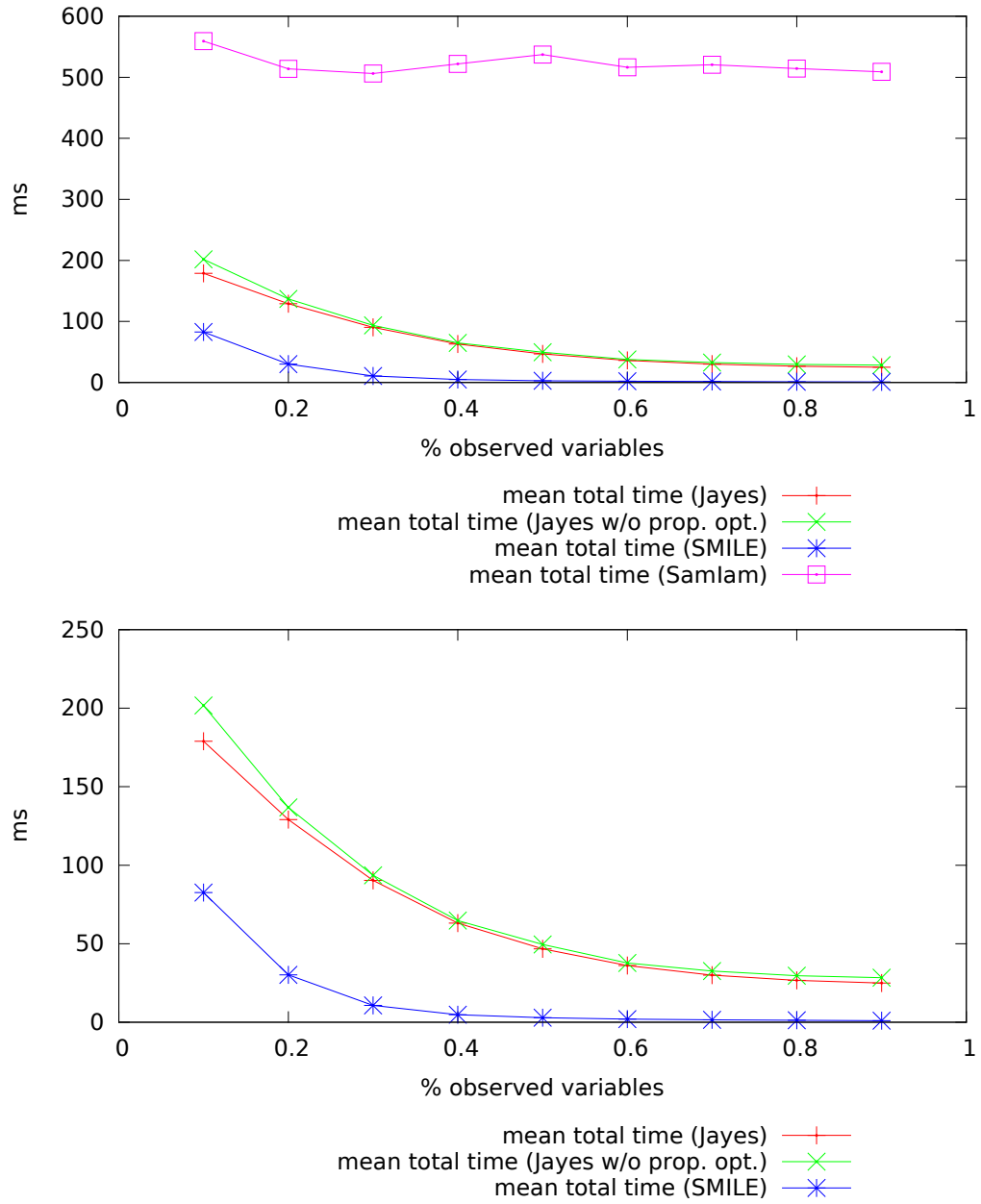


Figure 28: Evaluation results for the diabetes network

A.5 hailfinder

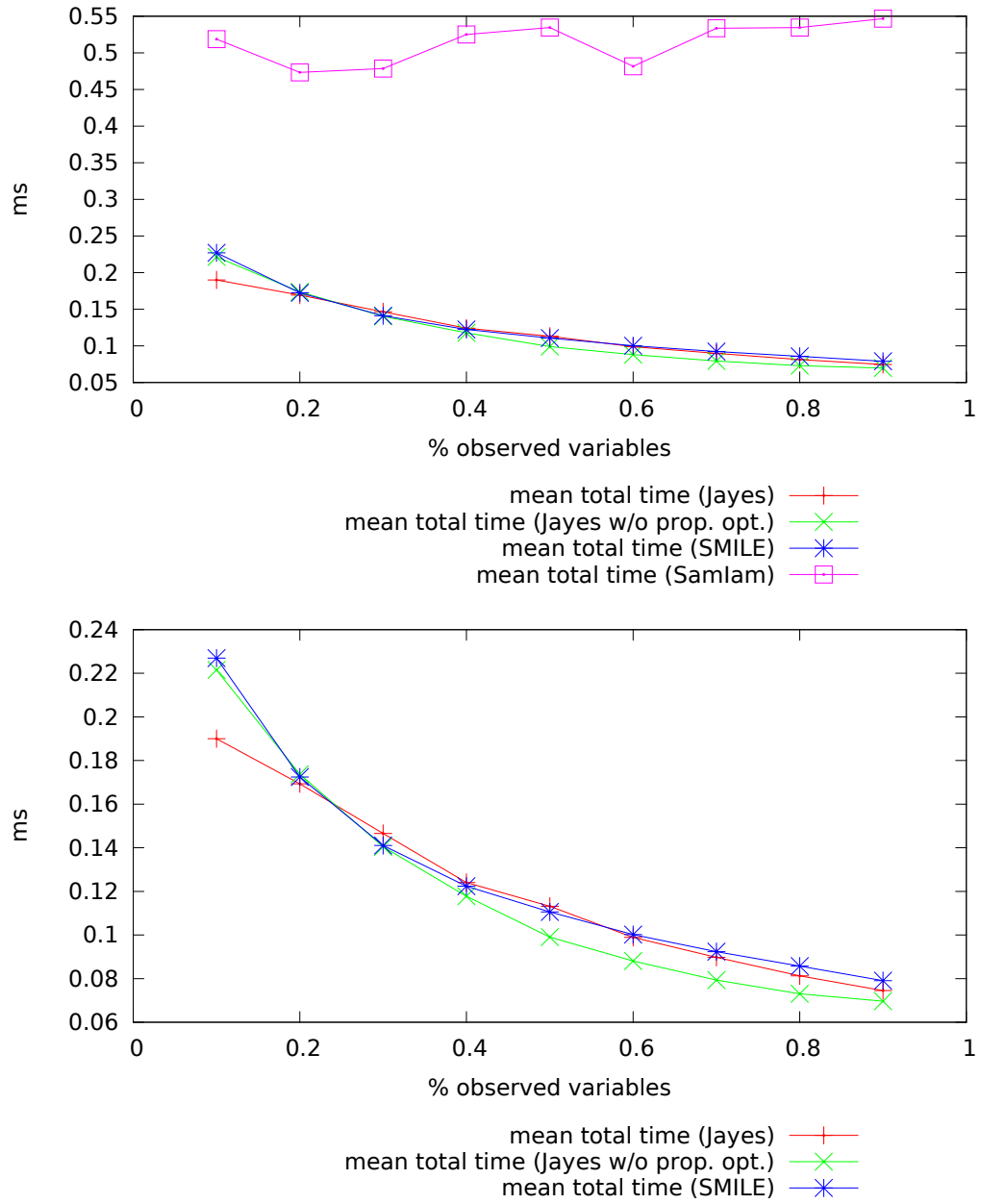


Figure 29: Evaluation results for the hailfinder network

A.6 Hepar II

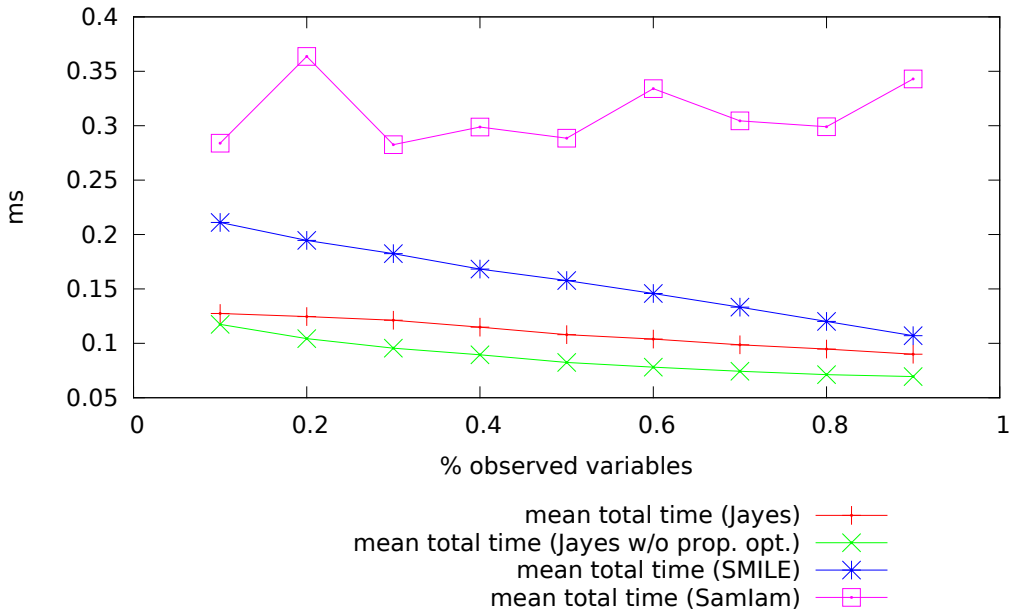


Figure 30: Evaluation results for the Hepar II network

A.7 pathfinder

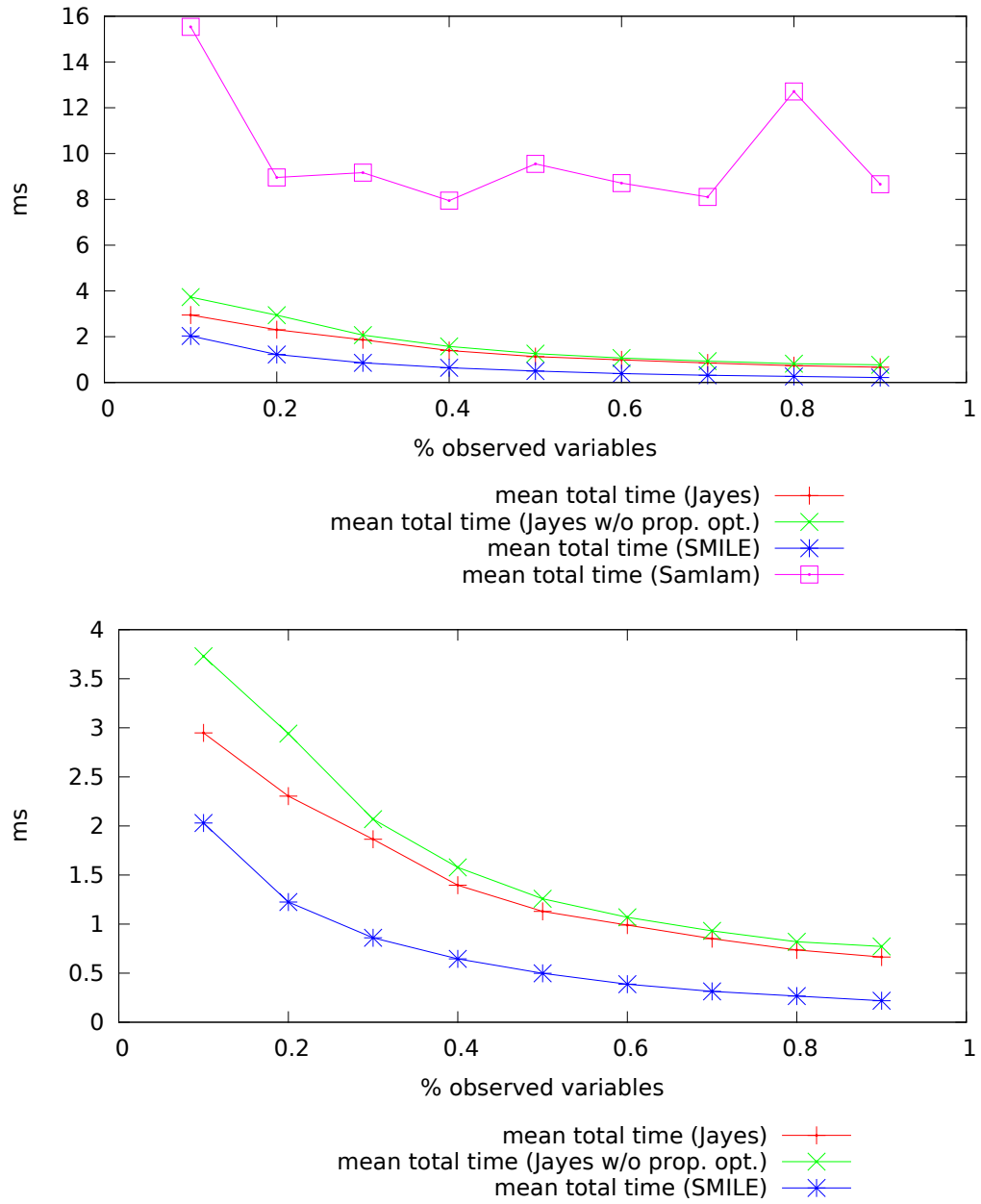


Figure 31: Evaluation results for the pathfinder network

A.8 win95pts

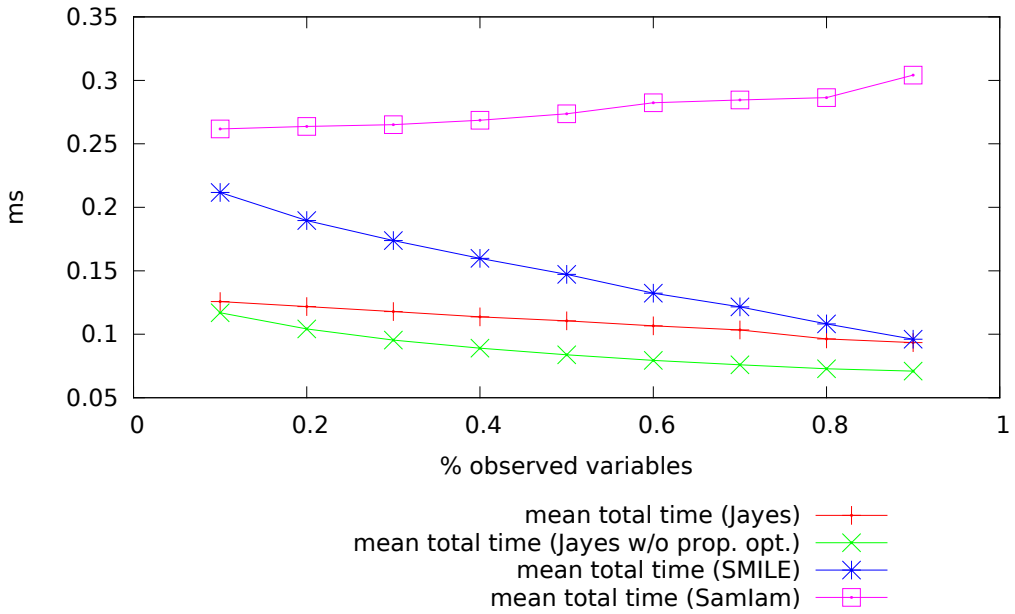


Figure 32: Evaluation results for the win95pts network

B Appendix - Evaluation results (skipped messages)

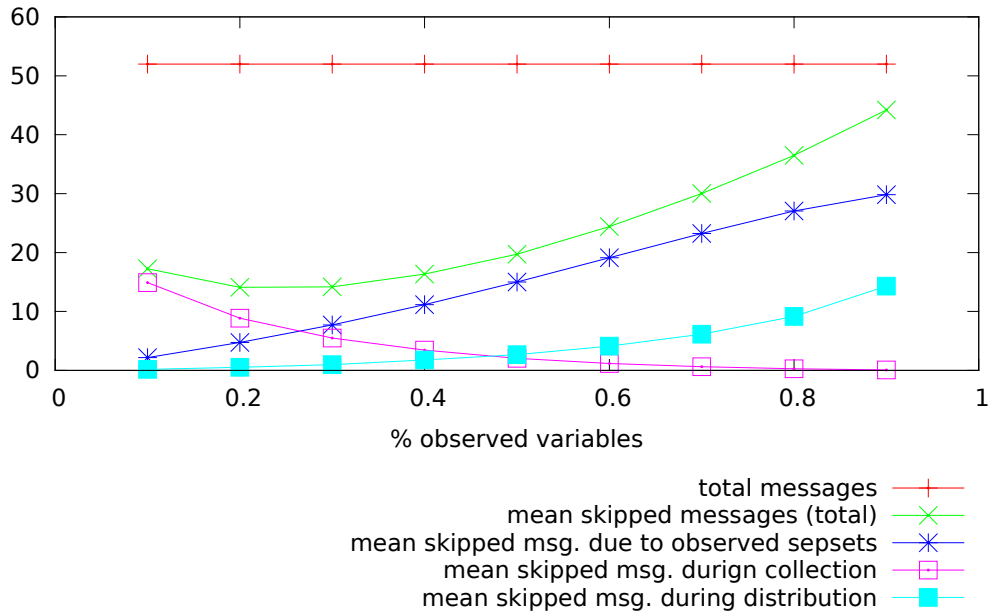


Figure 33: Skipped messages (alarm network)

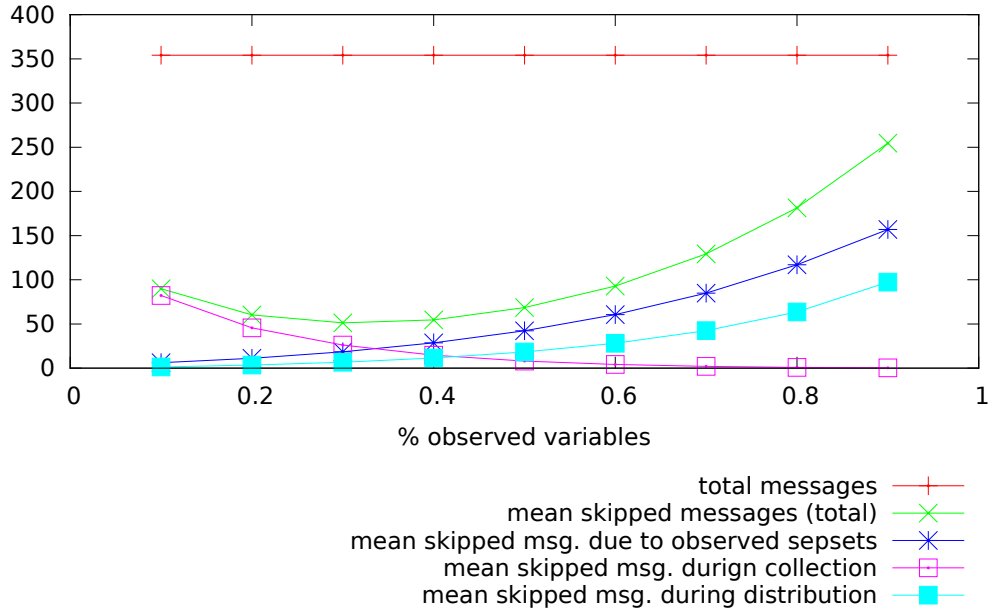


Figure 34: Skipped messages (andes network)

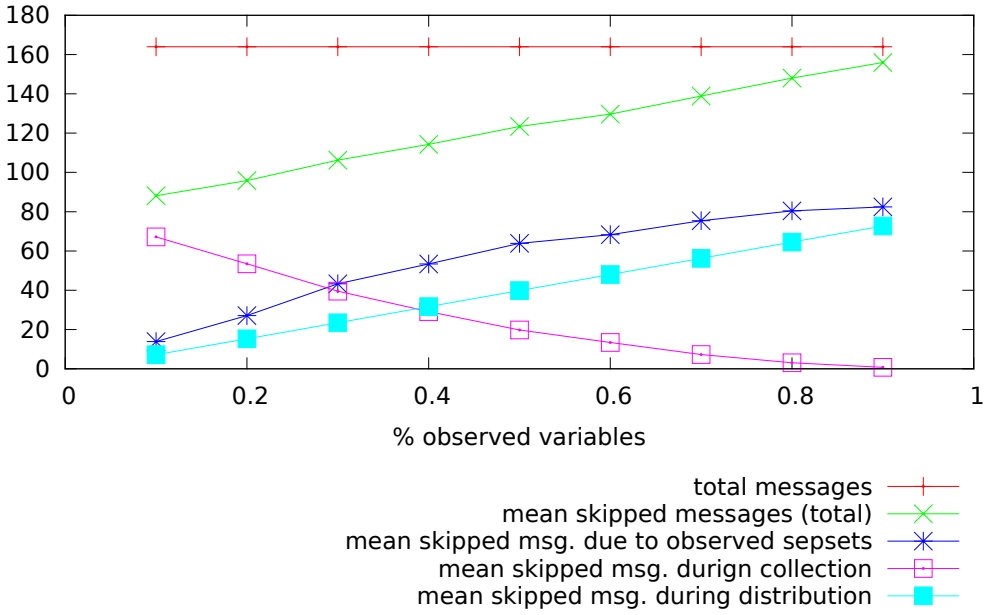


Figure 35: Skipped messages (Button network)

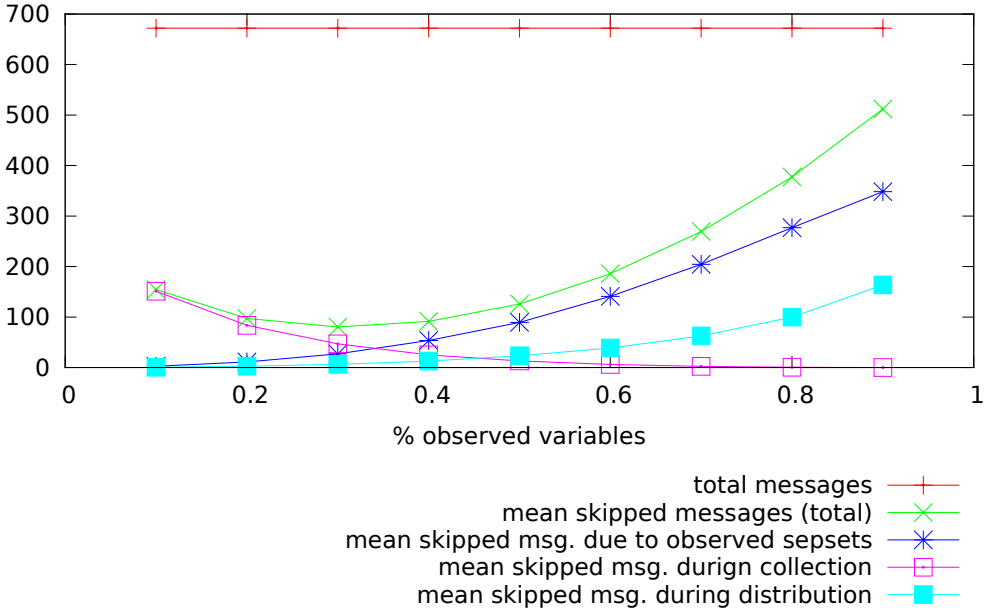


Figure 36: Skipped messages (diabetes network)

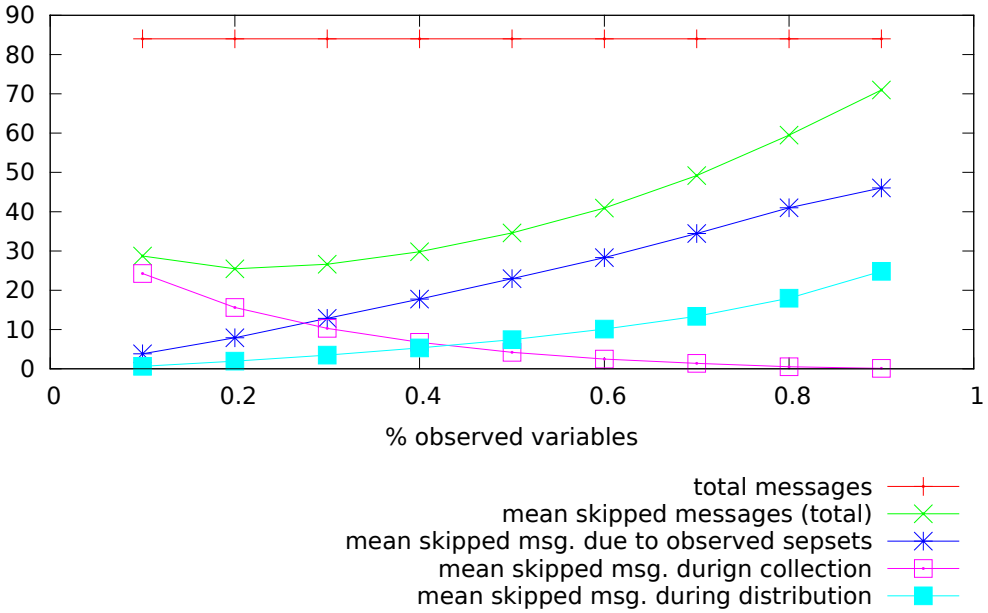


Figure 37: Skipped messages (hailfinder network)

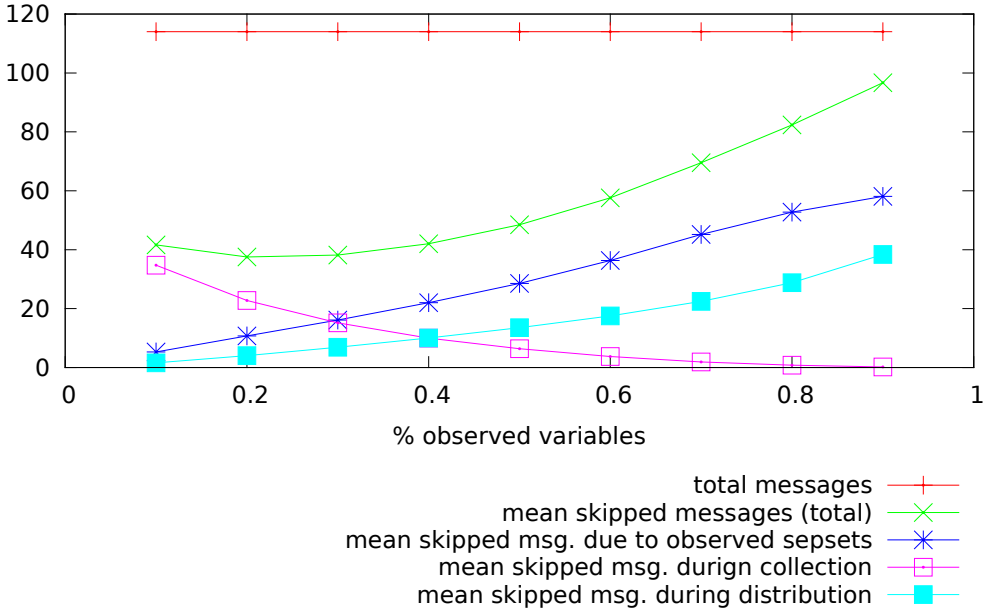


Figure 38: Skipped messages (Hepar II network)

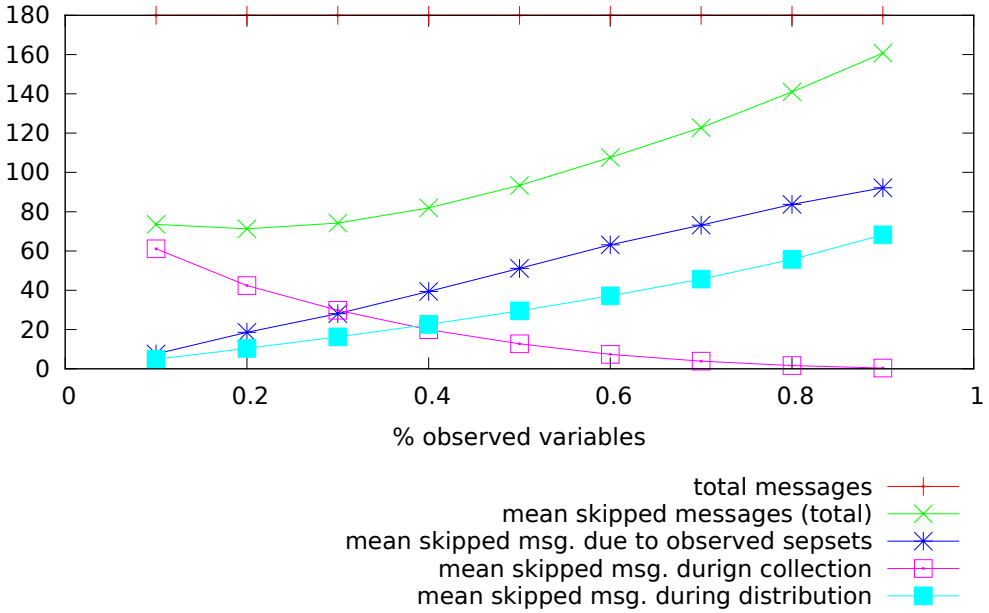


Figure 39: Skipped messages (pathfinder network)

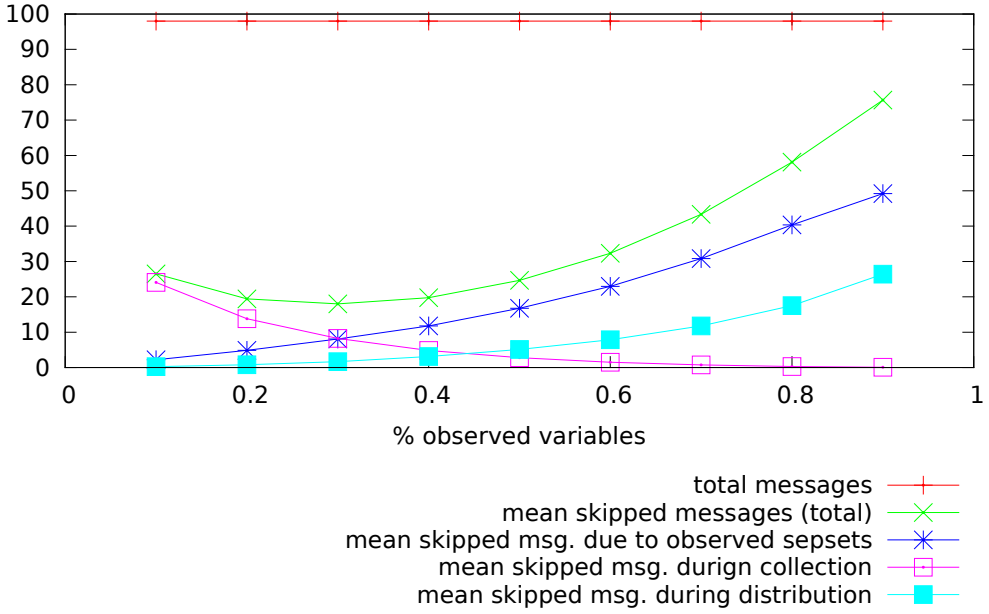


Figure 40: Skipped messages (win95pts network)