Extracting the K Best Solutions from a Valued And-Or Acyclic Graph

by

Paul Harrison Elliott

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Masters of Engineering in Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2007

Author . . . .
Department of Electrical Engineering and Computer Science
May 29, 2007

Certified by . . . .
Howard Shrobe
Principal Research Scientist in Electrical Engineering and Computer Science
Thesis Supervisor

Certified by . . . .
Brian C. Williams
Professor in Aeronautical and Astronautical Engineering
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Professor of Electrical Engineering
Chairman, Department Committee on Graduate Theses

Extracting the K Best Solutions from a Valued And-Or Acyclic Graph

by

Paul Harrison Elliott

Submitted to the Department of Electrical Engineering and Computer Science
on May 29, 2007, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in Computer Science

Abstract

In this thesis, we are interested in solving a problem that arises in model-based programming, specifically in the estimation of the state a system described by a probabilistic model. Some model-based estimators, such as the MEXEC algorithm and the DNNF-based Belief State Estimation algorithm, use a valued and-or acyclic graph to represent the possible estimates. These algorithms specifically use a valued *smooth deterministic decomposable negation normal form* (sd-DNNF) representation, a type of and-or acyclic graph.

Prior work has focused on extracting either all or only the best solution from the sd-DNNF. This work develops an efficient algorithm that is able to extract the k best solutions, where k is a parameter to the algorithm. For a graph with $|E|$ edges, $|V|$ nodes and $|E_v|$ children per non-leaf node, the algorithm presented in this thesis has a time complexity of $O(|E|k \log k + |E| \log |E_v| + |V|k \log |E_v|)$ and a space complexity $O(|E|k)$.

Thesis Supervisor: Howard Shrobe
Title: Principal Research Scientist in Electrical Engineering and Computer Science

Thesis Supervisor: Brian C. Williams
Title: Professor in Aeronautical and Astronautical Engineering

Acknowledgments

I would like to thank my wife Joelle Brichard for her patience. I would like to thank Dr. Howie Shrobe for accepting this project on short notice. I would like to thank my adviser Prof. Brian Williams and my co-worker Seung Chung who helped with the initial versions of the algorithm.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

In this thesis, we are interested in solving a problem that arises in model-based programming[15]. In model-based programming, the models written by the system engineers can be used to diagnose and reconfigure the system online. The main component of a model-based program is a generic software engine that is validated for correctness once and re-used on multiple projects, changing only the engine data, the system models.

The problem of interest to this thesis occurs in the diagnosis portion of the engine, called a mode estimator. The mode estimator is capable of automatically doing system-wide diagnostic reasoning, inferring the likely hidden state of the system. An estimator infers the current state by reasoning over a probabilistic model of the system dynamics, the commands that have been executed, and the resulting sensory observations. To support real-timed interaction of the engine with the world, mode estimators must approximate the probability distribution of the hidden state. One type of approximation commonly used is fixing the number of states or trajectories tracked simultaneously by the estimator, such as in the *Best-First Trajectory Enumeration* (BFTE) algorithm [15], the *Best-First Belief State Update* (BFBSU) algorithm

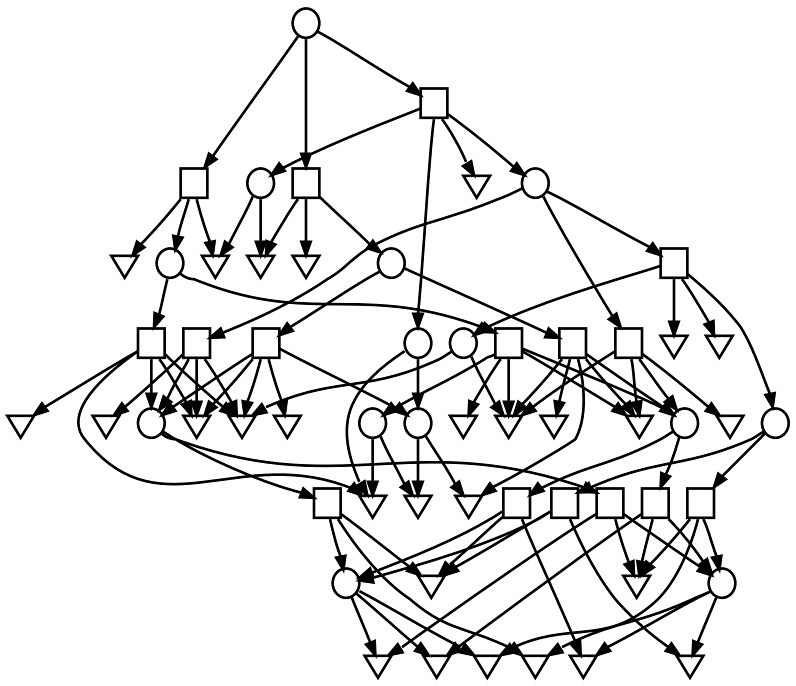


Figure 1-1: This is an example sd-DNNF from which we would like to extract solutions. This example contains 180 solutions within the 58 nodes, where a solution typically includes about 20 nodes. We use circles to represent **Or** nodes, squares for **And** nodes, and triangles for leaf nodes.

[14, 13], the MEXEC algorithm [1], and the *DNNF-based Belief State Estimation* (DBSE) algorithm [12].

For the mode estimator algorithms DBSE and MEXEC, the underlying representation of the estimator is a valued *smooth deterministic decomposable negation normal form* (sd-DNNF) [8] representation. The valued sd-DNNF representation is related to AND/OR Search Spaces for Graphical Models [11]. Prior work on *valued* sd-DNNF representations have only shown how to extract the best solution, all solutions, or all solutions that have the same value as the best solution, where this last type of extraction is a composition of the first two.

The sd-DNNF representation is a directed acyclic graph, with a single root node.

The internal nodes are labeled as either **And** or **Or** nodes and the leaves of the graph are partial solutions we wish to combine into a complete solution. Our sd-DNNF is valued in that our leaves are labeled with probabilities. We expect to repeatedly extract solutions from the same valued sd-DNNF, where we will be varying only the values. Fig. 1-1 is an example of an sd-DNNF from which we would like to extract solutions. The DBSE and MEXEC algorithms use an sd-DNNF as their representation because an sd-DNNF is a compact encoding of solutions. An sd-DNNF is compact through the use of decomposition and memoization [9].

1.1 Problem Statement

The problem we're interested in solving is to find the k most probable solutions of a valued sd-DNNF, a type of acyclic and-or graph with valued leaves. We will define our sd-DNNF, a solution of an sd-DNNF, and the probability of a solution in the next section followed by a formal definition of this problem statement in Section 1.3.

1.2 Definitions

In this section we will formally define our valued sd-DNNF, followed by the definition of a selection of the sd-DNNF, the correspondence between selections and solutions, and then the probability of a solution.

1.2.1 Valued sd-DNNF

Formally, our valued sd-DNNF is the tuple $\langle V, E, \mathcal{L}_L, \mathcal{L}_{\mathbf{P}}, \times, > \rangle$:

- V is the set of nodes of the directed, acyclic graph. V is partitioned into three sets, A, O, and L, corresponding to the **And**, **Or**, and **Leaf** nodes, respectively.

r is the root node of this acyclic graph such that $r \in V$. Our graph has a single root.

- E is the set of edges of the graph. An edge $e \in E$ is an ordered pair of nodes $\langle m, n \rangle$. Edges are directed: $m \rightarrow n$. Since leaves necessarily have no out-going edges, $m \in A \cup O$ and $n \in V = A \cup O \cup L$. We define a *path* $v_1, v_2, \ldots, v_p$ in the normal way: a path is such that for every successive pair of nodes v_i and v_{i+1}, there is an edge $\langle v_i, v_{i+1} \rangle \in E$. The graph is acyclic, so there does not exist a path such that $v_1 = v_p$, for any p. Thus, for each edge $\langle m, n \rangle$, we designate n as a *child* of m, and m as a *parent* of n. All **And** and **Or** nodes have at least one child.

- $\mathcal{L}_L$ is a function that labels L with a unique symbol or the empty symbol $\emptyset$. This symbol can be thought of as the meaning of this leaf. This algorithm assumes these symbols are partial solutions to a problem that we care about, and we will explain how they are used in the algorithm momentarily when we define a solution.

- $\mathcal{L}_{\mathbf{P}} : L \rightarrow [0, 1]$ is a function that labels L with a probability, or more generally, with a cost or reward. The meaning of this probability will be explained momentarily.

- $\times$ is a binary function that combines probabilities into a new probability. This function is expected to combine the labels of $\mathcal{L}_{\mathbf{P}}$.

- $>$ is a total ordering of the probabilities of $\mathcal{L}_{\mathbf{P}}$. If $a > b$, then we prefer solutions with probability a over solutions with probability b. We will define solutions and probabilities of solutions momentarily.

In our definition of our valued sd-DNNF, we are assuming that the most probable solution is defined by a maximum-product, thus we use $>$ and $\times$, respectively, to find the probability of a solution. This work can be equivalently framed as a maximum-sum, using $>$ and $+$, should we want to use rewards instead of probabilities. Likewise, we can minimize instead of maximize. The important part to this algorithm is that $>$ be a choice function and $\times$ be a function that combines independent choices.

1.2.2 Selection

A *selection* is a set of nodes that obey these rules:

1. A selection always includes the root node.
2. For every **And** node a selected, every child of a is also selected.
3. For every **Or** node o selected, one and only one child of o is also selected.

For example, consider the selection of Fig. 1-2. The selection shown is {o1, a2, l4}. We denote the root o1 with a double line. In this case, our root is an **Or** node. The other two valid selections are {o1, o3, a5, l7} and {o1, o3, a6, l8}. The labels of $\mathcal{L}_L$ and of $\mathcal{L}_\mathbf{P}$ are designated in the figure within the $L[\ldots]$ and the $P[\ldots]$ leaf labels, respectively, of nodes l4, l7, and l8. For example, $\mathcal{L}_L(\text{l4}) =$ “Switch=Off” and $\mathcal{L}_\mathbf{P}(\text{l4}) = 0.5$.

1.2.3 sd-DNNF Properties

An sd-DNNF[8] imposes three properties on an and-or acyclic graph, the properties of *smooth*, *deterministic*, and *decomposable*. All three properties assume that the symbols we use to label the leaves represent assignments to variables, either binary or multi-valued. The *smooth* property states that every variable x that labels a descendant of one child of an **Or** node must label a descendant of every child of the **Or** node. Said another way, for an **Or** node o, every selection rooted at o will define

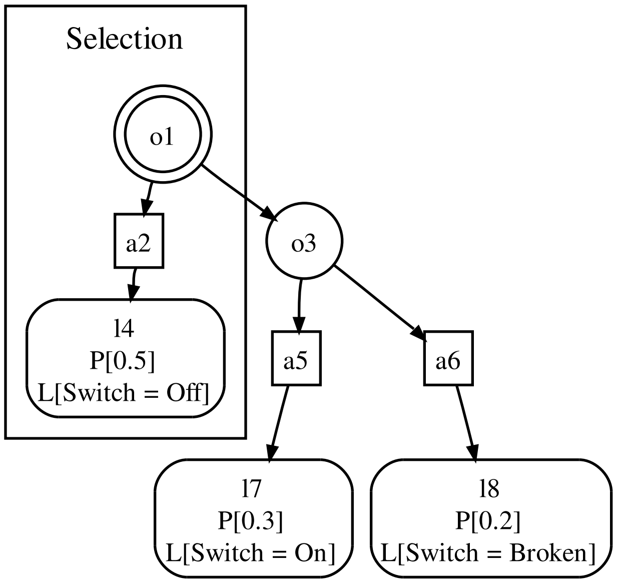


Figure 1-2: This figure shows a selection of this simple sd-DNNF. The node o1 is the root of the tree. Our selection consists of o1, a2, and l4. This is a correct selection as it includes the root o1; it includes exactly one child of o1, namely a2; and it includes all of the children of a2, namely l4.

a solution with exactly the same variables as every other selection rooted at *o*. Fig. 1-2 is smooth because the only variable Switch appears on a leaf of some descendant of both children of both or nodes, o1 and o3.

The *deterministic* property also applies to **Or** nodes. This property requires that each selection, as specified by a selection per **Or** node, must represent a different set of assignments to the variables. For the purpose of this thesis, this means that the set of symbols on the leaves of a selection are unique among all selections. For Fig. 1-2, the deterministic property trivially holds as each selection has a different leaf and each leaf has a unique assignment.

The *decomposable* property applies to **And** nodes. This property requires that the variables of the leaves of a descendant of a child of the **And** node are disjoint from the variables of any other descendant of every other child; an **And** node partitions variables among its children. In this way, contradictory assignments are never included in the same selection. For this thesis, this property ensures that if a symbol only appears on one leaf, a selection will never include the same symbol twice. This again holds trivially for Fig. 1-2, as every **And** node has only one child.

1.2.4 Solution

A *solution* is constructed by creating a *selection* of nodes, and then applying $\mathcal{L}_L$ to all of the leaf nodes in the selection. The set of resulting leaf symbols is a solution. We omit the empty symbol $\emptyset$ from our solution.

For example, the solution of the selection {o1, a2, l4} shown in Fig. 1-2 is {“Switch=Off”}. Since the only leaf node of this selection is l4, our solution is the set containing only $\mathcal{L}_L$ (l4). As stated above, this is “Switch=Off”.

We are assuming a deterministic and-or graph, so each selection will have a unique set of leaf symbols. We further require that these set of leaf symbols are unique even after omitting the empty symbol $\emptyset$, and thus each selection will have a unique solution. In the DBSE algorithm, $\emptyset$ is used to add leaves with probabilities that can alter which solutions are the best solutions while not changing the symbols included in the best solutions.

1.2.5 Solution Probability

To compute the *probability* of a solution, we apply $\mathcal{L}_\mathbf{P}$ to all of the leaf nodes of the selection of the solution and then combine them with $\times$. Since we assume there is a one-to-one correspondence between solutions and selections, this selection is unique.

In the example of Fig. 1-2, the probability of the selection and solution is 0.5. Since we only have one leaf, we need not apply $\times$.

1.3 Formal Problem Statement

For the valued sd-DNNF $\langle V, E, \mathcal{L}_L, \mathcal{L}_{\mathbf{P}}, \times, > \rangle$, we want to find the k most probable solutions. Equivalently, we order all selections from most probable to least probable and keep the first k selections. In this thesis, we want the k solutions of the k best selections and the probability of these solutions[1].

1.4 Innovative Claims

Given an sd-DNNF with $|E|$ edges, $|V|$ nodes, $|E_v|$ children per node, and trying to extract the k best solutions, this work provides a novel algorithm that solves our problem with a running time of $O(|E|k \log k + |E| \log |E_v| + |V|k \log |E_v|)$ and a space of $O(k|E|)$. Prior work [8] was only able to extract the k best solutions, for $k > 1$, by extracting all solutions and then keeping the best k. Since the number of solutions is expected to be much larger than $|E|$, this is a substantial improvement.

At the core of the k best solutions algorithm is a novel algorithm that can find the k best combinations of n sorted lists of k elements each. A combination is an element from each of the n lists combined together using $\times$, and thus there are k^n combinations. This novel algorithm has a complexity of $O(nk \log k)$ time and $O(nk + k \log k)$ space.

1.5 Related Work

This work builds on the Minimum Cardinality (MCard) work by Darwiche [8] and valued sd-DNNF work by Barret [1]. Both provide a specification for the sd-DNNF

[1]We can assume that solutions can be found from selections because we assume that there is at most one selection in an sd-DNNF that generates a particular solution. The deterministic property of an sd-DNNF ensures this is a correct assumption. Thus, we need never combine multiple selections, typically with $+$, to find the best solution.

$\langle V, E, \mathcal{L}_L, \mathcal{L}_{\mathbf{P}}, \times, > \rangle$ and an algorithm to extract the best solution (minimal or maximal, respectively). [8] also provides two more algorithms, one to extract all solutions and one to extract all solutions of minimum cardinality (value). The latter algorithm first restricts the tree to have minimum value and then extracts all solutions. This thesis generalizes the algorithm to extract the maximal solution by allowing more than one solution to be extracted. We present a version of the algorithm to extract the maximal value in Chapter 2.

This work is inspired by the acyclic join-tree algorithm [10] and employs dynamic programming [2]. A join-tree is a tree where the nodes are constraints on variables and the edges connect together nodes whose constraints share variables. The edges are labeled with the shared variables. A join-tree requires that every node with a common variable a be connected through a path of edges labeled with a to every other node that also shares the common variable a. The acyclic join-tree algorithm looks for a solution to this constraint tree by having each node, starting at the leaves, inform their parent as to which values of their shared variables are possible. The parent then constrains itself with this information (by performing a join operation). This continues to the root, at which point the root has enough information to know if there exists a solution and which assignments are possible at the root. The root can then make a final decision as to which assignment it chooses and this information is then propagated back to the leaves, where a full solution can be assembled.

1.6 Approach

Qualitatively, our sd-DNNF consists of a number of local constraints at **And** and **Or** nodes that describe how an internal node's value depends on its children's values. Leaves always have a constant value. Since a node may have multiple parents, we use dynamic programming to remember each node's value and annotations about why it has that value. We want to select the final k selections at the root, as the root will have all the information necessary to make the decision. Thus, our algorithm propagates up the impact of the choice of leaves to the root. Once the root has made its decision, our algorithm finds the k best solutions by traversing the tree back down from the root to the leaves, based on the annotations. Once we reach the leaves, we can assemble our solution from the symbols of each leaf.

This thesis will next present an algorithm for extracting the most probable solution from the valued sd-DNNF in chapter 2. Chapter 3 presents our new algorithm that extracts the k most probable solutions from the sd-DNNF and then results of running this algorithm on seven graphs is presented in chapter 4. We will then conclude in chapter 5.

Chapter 2

Best-Solution Algorithm

This chapter introduces an algorithm to extract the best solution from an sd-DNNF. To extract the best solution, we need to find the best selection. Intuitively, since two selections differ based on the choices made at the **Or** nodes, we want to choose the best child for every **Or** node.

To find this best selection, we apply three rules:

1. For each leaf node l, the probability of the leaf node is $\mathcal{L}_{\mathbf{P}}(l)$.

2. For each **And** node a, the probability of the **And** node is the combination of the probability of all of its children using $\times$.

3. For each **Or** node o, we choose the best child v of o using $>$ and the probability of o is the probability of v.

The best selection is then the selection that includes the best child of each **Or** node visited from the root. We visit the best child of an **Or** node and all children of **And** nodes. Fig. 2-1 shows an example of a best selection for Fig. 1-2. In Fig. 2-1, we have highlighted the best subgraphs for each node with a solid line. The subgraph that starts at the root node o1 is the best overall selection.

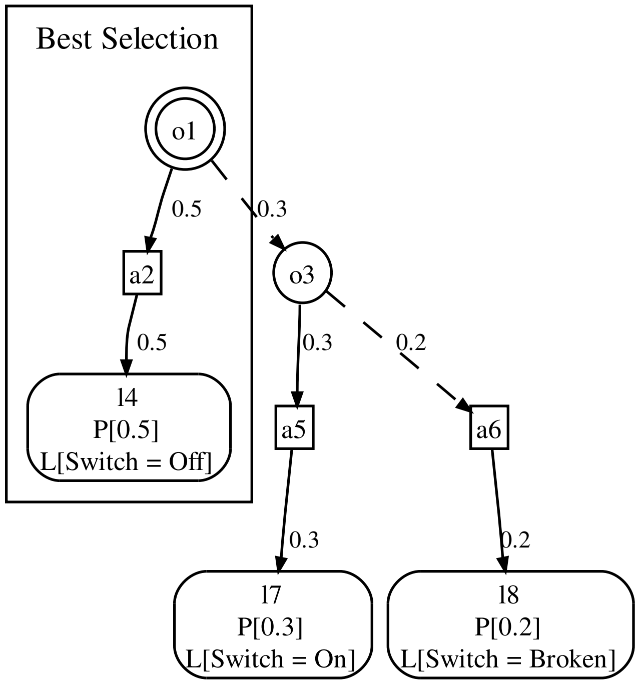


Figure 2-1: This figure shows the best selection of this simple sd-DNNF. Solid arcs represent the best choice for each node locally. Starting at the root, the overall best choice is o1, a2, and then l4, which is our best selection. We label the arcs with the probability of choosing that child.

In following these three rules, if we cache at each node the probability of the best choice, then the parents of the node can make use of this probability locally to compute their own best probability. We will visit each edge once: for **And** nodes we apply $\times$ to each child and for **Or** nodes we select the largest child with $>$. Since each parent needs their children's values to evaluate their own rule, we need a way of visiting all children before their parents. We've chosen to pre-order our nodes from 1 to $|V|$, such that the order of a node is greater than all parents of the node and less than all children of the node[1]. This is called a topological sort[3]. This ordering

[1]Recall that we're interested in solving the same problem multiple times, varying only the prob-

must exist because there are no cycles in the graph, though it is not in general unique. Using the ordering, the algorithm can walk over the nodes from $|V|$ to 1 and guarantee that every child is visited before its parent. We designate our ordered nodes V_O. This ordering always places the root r at position 1.

Once we have the best selection, we need to extract the corresponding solution. For each **Or** node o in the graph, we record a decision $\eta(o) \in \text{Children}(o)$. The solution of the selection is the set of $\mathcal{L}_L(l)$ for each l that has a path from the root r to l such that every **Or** node o_i in the path at position i is followed by $\eta(o_i)$ at position $i+1$.

Since we're looking for all leaves connected to the root by some path, this problem is naturally related to the transitive closure[4] of the sd-DNNF graph. A transitive closure of a directed graph is a new graph where the nodes are the same, but there is an edge $\langle m, n\rangle$ in the new graph if there is a *path* in the original graph from m to n. The problem of determining the leaves of the selection is equivalent to examining the leaves that are directly connected to the root node r in the transitive closure graph of a modified sd-DNNF, where the sd-DNNF is "modified" such that the only out-going edge of an **Or** node is the one specified by η. We are only interested in the edges of the root node in the transitive closure graph, so we need not compute the full transitive closure of the graph. Since our graph is acyclic, we can use our topological ordering to walk once over the nodes and be sure to visit all nodes along any path from the root to the leaves before their children. This lets us avoid adding edges explicitly and instead only mark nodes that would be connected to the root in the transitive closure graph.

A node is always connected to itself, so we always mark the root. For connected **And** nodes, we mark all children as also connected to the root, as all of them are part of at least one path from the root to a leaf. For connected **Or** nodes, we mark only

abilities, not the structure, so we can omit this sorting cost from our calculations.

the child specified by η. Once we have marked our sd-DNNF, we can apply $\mathcal{L}_L$ to all of the marked leaves, as these are all part of the solution. In Alg. 2.3, we make the optimization of applying $\mathcal{L}_L$ to a leaf when it is visited, rather than making a second pass of the marked sd-DNNF.

2.1 Find-Best-Solution Algorithm

Algorithm 2.1: FindBestSolution(V_O, E, $\mathcal{L}_L$, $\mathcal{L}_{\mathbf{P}}$, $\times$, $>$)

```
1 $\eta \leftarrow$ FindBestSelection($V_O$, $E$, $\mathcal{L}_{\mathbf{P}}$, $\times$, $>$) ;
2 $S \leftarrow$ GetSolutionFromSelection($V_O$, $E$, $\mathcal{L}_L$, $\eta$) ;
3 return $S$;
```

The algorithm that computes the best solution is shown in Alg. 2.1. The algorithm is broken into the two passes specified above, a pass from the leaves to the root that computes the best selection and a second pass from the root to the leaves that extracts the solution of the best selection. The first pass returns $\eta : O \rightarrow V$, a function that records the best child node for each **Or** node. η defines a superset of a selection, as it contains decisions for **Or** nodes that are not part of the selection. The parts of η that are not part of the best selection will be ignored by GetSolutionFromSelection as the irrelevant **Or** nodes will not be connected to the root node. The second pass returns the best solution, a set of labels, corresponding to the selection.

2.1.1 Find-Best-Selection Algorithm

The first part of Alg. 2.1 is shown in Alg. 2.2. This function is propagating the probabilities of the leaves of the valued sd-DNNF to the root, making decisions at each **Or** node as to which child is best. This algorithm applies the three rules on page 27. Line 5 applies rule 1. Lines 8-13 apply rule 2. These lines combine the

Algorithm 2.2: FindBestSelection(V_O, E, $\mathcal{L}_{\mathbf{P}}$, $\times$, $>$)

```
for $i = |V|$ to $1$ do
    $v \leftarrow V_O\left[i\right]$ ;
    switch $v$ in
        case $v \in L$
            $\mathbf{P}_V\left(v\right) \leftarrow \mathcal{L}_{\mathbf{P}}\left(v\right)$ ;
        end
        case $v \in A$
            // Combine the probability of the children of $v$
            $e \leftarrow$ some $\langle v, n\rangle \in E$ ;
            $p \leftarrow \mathbf{P}_V\left(n\right)$ ;
            foreach $\langle v, n\rangle \in E \setminus e$ do
                $p \leftarrow p \times \mathbf{P}_V\left(n\right)$ ;
            end
            $\mathbf{P}_V\left(v\right) \leftarrow p$ ;
        end
        case $v \in O$
            // Find the best child of $v$
            $e \leftarrow$ some $\langle v, n\rangle \in E$ ;
            $\langle b, p\rangle \leftarrow \langle n, \mathbf{P}_V\left(n\right)\rangle$ ;
            foreach $\langle v, n\rangle \in E \setminus e$ do
                if $\mathbf{P}_V\left(n\right) > p$ then
                    $\langle b, p\rangle \leftarrow \langle n, \mathbf{P}_V\left(n\right)\rangle$ ;
                end
            end
            $\mathbf{P}_V\left(v\right) \leftarrow p$ ;
            $\eta\left(v\right) \leftarrow b$ ;
        end
    end
end
return $\eta$ ;
```

probability of each child of the **And** node v using $\times$ and this is the probability of v. Lines 16-23 apply rule 3. These lines look for the most probable child of the **Or** node v, using $>$. Line 24 then records the **Or** node's most probable child in η. Finally, we return η on line 28.

Runtime Analysis This algorithm visits every node once and every edge once. Our nodes are stored sorted in an array, making $V_O\left[i\right]$ an $O(1)$ operation. We perform $\times$ or $>$ per edge, which for our problem are both $O(1)$ operations. We store edges with the parent node, which means we can directly access the list of edges $v \to n$ on lines 10 and 18 in $O(1)$ time. We store $\mathbf{P}_V$ with each node, and thus as a space optimization, we can use $\mathcal{L}_{\mathbf{P}}\left(l\right)$ as $\mathbf{P}_V\left(l\right)$ for all the leaf nodes. Storing $\mathbf{P}_V$ with each node makes looking up and updating $\mathbf{P}_V$ also an $O(1)$ operation. Finally, we also store η with the **Or** nodes, likewise giving us $O(1)$ access. We can return η to the second part of Alg. 2.1, Alg. 2.3, by just passing Alg. 2.3 our annotated sd-DNNF. Thus, for each edge and each node, we perform an $O(1)$ operation, giving Alg. 2.2 a time complexity of $O(|E| + |V|)$. Since every node v in the sd-DNNF has a path from r to v, the sd-DNNF has at least as many edges as a tree. A tree has one more node than edge, so for the sd-DNNF $|E| + 1 \geq |V|$. This constraint lets us simplify our complexity bound to $O(|E|)$.

Space Analysis The sd-DNNF itself requires $O(|E| + |V|)$ space. The algorithm stores a probability $\mathbf{P}_V$ per node and a reference to a node for η per **Or** node. This is an $O(|V|)$ additional space requirement.

2.1.2 Get-Solution-From-Selection Algorithm

The second part of Alg. 2.1 is shown in Alg. 2.3. This function extracts the best solution that corresponds to the best selection η we found in the first part. Line 1

Algorithm 2.3: GetSolutionFromSelection(V_O, E, $\mathcal{L}_L$, η)

```
Marked ← $\{r\}$ ;                    // Initially just the root is marked
$S \leftarrow \emptyset$ ;
for $i = 1$ to $|V|$ do
    $v \leftarrow V_O[i]$ ;
    if $v \in$ Marked then
        switch $v$ do
            case $v \in L$
                $S \leftarrow S \cup \{\mathcal{L}_L(v)\}$ ;
            end
            case $v \in A$
                foreach $\langle v, n \rangle \in E$ do
                    Marked ← Marked $\cup \{n\}$ ;
                end
            end
            case $v \in O$
                // Mark the choice made in $\eta$
                Marked ← Marked $\cup \{\eta(v)\}$ ;
            end
        end
    end
end
return $S$ ;
```

initially marks the root node, in preparation for finding the leaves connected to the root in the modified sd-DNNF. Line 2 initially sets our solution to empty. Lines 3-20 then loop over the nodes from the root to the leaves. Line 5 ensures we only extend our paths from marked nodes, nodes that are already part of some path from the root. Line 8 adds to our solution by applying $\mathcal{L}_L$ to a marked leaf. Lines 11-13 marks all the children of a marked **And** node. Line 16 marks the one selected child of a marked **Or** node. Once lines 3-20 have visited all the nodes, all of the marked leaves will have been visited, so S represents the solution corresponding to the selection and we return this on line 21.

Runtime Analysis This algorithm visits every node once and every edge of every marked node once. We store a flag with each node indicating whether or not it is marked, thus setting and checking this flag is $O(1)$. Since we store the flag per node, line 1 is an $O(|V|)$ operation, as we must clear the marks on every node except the root, which must be set. We assume solution labels are unique and that the order in which they need to be returned is unimportant, so adding $\mathcal{L}_L(v)$ to S on line 8 just involves appending the symbol to a list, an $O(1)$ operation. We only mark those nodes that are part of the selection, so this append operation is performed $O(|\text{Leaves in the Selection}|)$ times by this algorithm. As stated in Section 2.1.1, we store η with the **Or** nodes and edges with the parent node, thus all the operations performed per edge and per node are $O(1)$. Since we only mark nodes that are part of the selection, we only visit those edges that are part of the selection. Thus, the time complexity of this algorithm is $O(|\text{Edges in the Selection}| + |V|)$. Since the number of edges required to define a selection varies widely from sd-DNNF to sd-DNNF, we cannot further simplify this bound.

An alternative formulation of this algorithm would be a recursive depth-first walk from the root to the leaves, visiting all the marked nodes. Due to the decomposition and determinism of the sd-DNNF, a node that is part of a selection will always have exactly one parent that is part of a selection (except the root, which has none). Thus, we would not visit the same node more than once. This formulation would only visit those nodes that are part of the selection, reducing the complexity of the algorithm to $O(|\text{Edges in the Selection}| + |\text{Nodes in the Selection}|)$. Since this forms a tree, $|\text{Edges in the Selection}| + 1 = |\text{Nodes in the Selection}|$, so this simplifies to $O(|\text{Nodes in the Selection}|)$.

Space Analysis The sd-DNNF itself requires $O(|E| + |V|)$ space. The algorithm stores a flag per node for Marked. We also store a list of symbols (or references to symbols) in S, our solution. The flags require $O(|V|)$ space and S requires $O(|\text{Leaves in the Selection}|)$ space.

The alternative formulation requires a stack for the **And** nodes along the current path, recording which child is currently being visited. This stack will contain at most the number of **And** nodes along the path with the most **And** nodes. This is clearly no more than $|A|$ as opposed to storing $|V|$ flags.

Putting together the runtime and space analysis for Algorithms 2.2 and 2.3, we can now state the requirements for Algorithm 2.1. The time required is dominated by Alg. 2.2, requiring $O(|E| + |V|)$ time, and thus this is also the time required by Alg. 2.1. The space required is proportional to the number of nodes in the graph, plus the graph itself, so $O(|E| + |V|)$ total space is used.

2.2 Find-Best-Solution Example

We will now show, as an example, the two parts of Alg. 2.1 run on the example shown in Fig. 2-1. The progression of Alg. 2.2 is show in figures 2-2, 2-3, and 2-4. Figures 2-5, 2-6, and 2-7 show progressively how Alg. 2.3 operates on Fig. 2-1.

The example shown in Fig. 2-1 is defined by the following sd-DNNF:

- The nodes $V = \{\text{o1, a2, o3, l4, a5, a6, l7, l8}\}$, where $A = \{\text{a2, a5, a6}\}$, $O = \{\text{o1, o3}\}$, and $L = \{\text{l4, l7, l8}\}$. The root node $r = \text{o1}$. The number at the end of each node's name in V is its ordering by V_O.
- The edges E are $\langle \text{o1}, \text{a2} \rangle$, $\langle \text{o1}, \text{o3} \rangle$, $\langle \text{a2}, \text{l4} \rangle$, $\langle \text{o3}, \text{a5} \rangle$, $\langle \text{o3}, \text{a6} \rangle$, $\langle \text{a5}, \text{l7} \rangle$, and $\langle \text{a6}, \text{l8} \rangle$.
- The symbols $\mathcal{L}_L$ are:
  - $\mathcal{L}_L\,(\text{l4}) =$ "Switch = Off"
  - $\mathcal{L}_L\,(\text{l7}) =$ "Switch = On"
  - $\mathcal{L}_L\,(\text{l8}) =$ "Switch = Broken"
- The probabilities $\mathcal{L}_{\mathbf{P}}$ are: $\mathcal{L}_{\mathbf{P}}\,(\text{l4}) = 0.5$, $\mathcal{L}_{\mathbf{P}}\,(\text{l7}) = 0.3$, and $\mathcal{L}_{\mathbf{P}}\,(\text{l8}) = 0.2$.
- Arithmetic multiplication for $\times$.
- Arithmetic greater-than for $>$.

2.2.1 Find-Best-Selection Example

The FindBestSelection algorithm is employing dynamic programming to ensure that the probability of each node is only computed once. We store the computed probability in the variable $\mathbf{P}_V$. We are trying to decide the best selection locally at each **Or** node, specifically o1 and o3 for our example, based on the probabilities of its children. This selection is stored in the variable η (Eta). The initially empty state of these variables and the graph are shown in Fig. 2-2.

Alg. 2.2 consists of one loop that runs from the leaves to the root of the valued sd-DNNF. The first node assigned to v on line 2 is l8. This is a leaf node, so we execute line 5. This sets $\mathbf{P}_V\,(\text{l8}) = \mathcal{L}_{\mathbf{P}}\,(\text{l8}) = 0.2$. The next v is l7, which sets $\mathbf{P}_V\,(\text{l7}) = 0.3$.

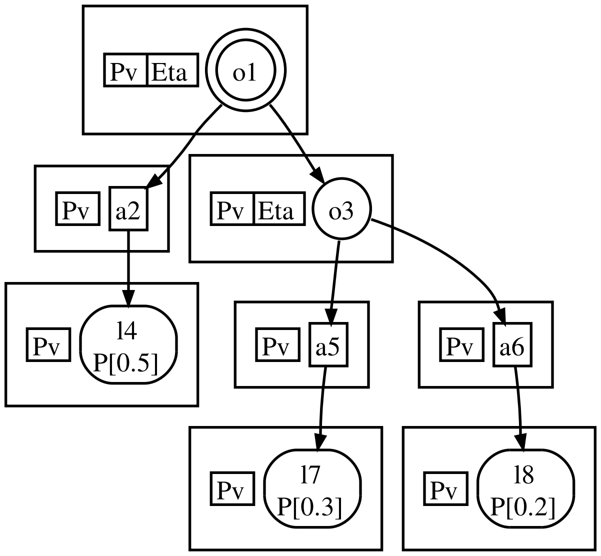


Figure 2-2: This figure shows the initial state of the FindBestSelection function for this simple sd-DNNF. The values of $\mathbf{P}_V$ are initially unknown and η is initially undecided.

We then visit $v = \text{a6}$, which executes lines 8 to 13. The only edge of the form $\langle \text{a6}, * \rangle$, that is to say the only out-going edge of a6, is the edge $\langle \text{a6}, \text{l8} \rangle$. Since $\mathbf{P}_V(\text{l8}) = 0.2$, we set $p = 0.2$ and then set $\mathbf{P}_V(\text{a6}) = 0.2$. We continue, visiting $v = \text{a5}$ and setting $\mathbf{P}_V(\text{a5}) = 0.3$; and visiting $v = \text{l4}$ and setting $\mathbf{P}_V(\text{l4}) = 0.5$. Fig. 2-3 shows the state of $\mathbf{P}_V$ and η at this point.

We then visit o3. The node o3 is our first **Or** node, and visiting this node executes lines 16 to 24. The node o3 has two children, a5 and a6. Lets assume that $n = \text{a5}$ is first, so we set $b = \text{a5}$ and $p = \mathbf{P}_V(\text{a5}) = 0.3$ on line 17. We then visit $n = \text{a6}$ and we skip this node because $\mathbf{P}_V(\text{a6}) = 0.2$ is less than 0.3. Line 23 then sets $\mathbf{P}_V(\text{o3}) = 0.3$, the value of o3's best child. Finally, line 24 sets $\eta(\text{o3}) = \text{a5}$, recording the best choice. The algorithm then continues on to the last two nodes, a2 and o1. Visiting the node a2 sets $\mathbf{P}_V(\text{a2}) = 0.5$, and visiting the node o1 sets $\mathbf{P}_V(\text{o1}) = 0.5$ and $\eta(\text{o1}) = \text{a2}$.

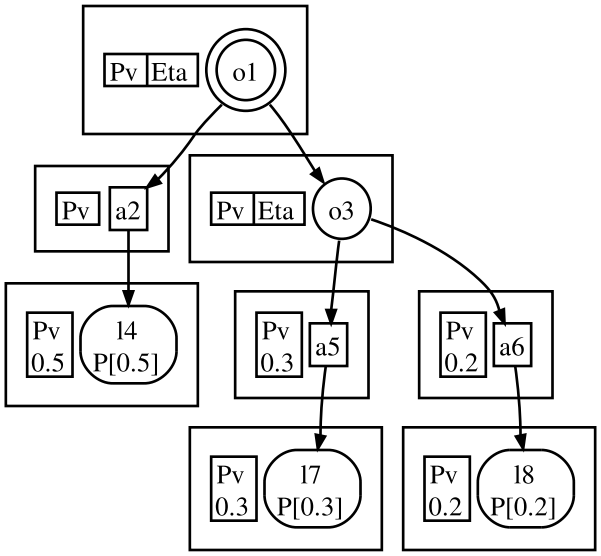


Figure 2-3: This figure shows the intermediate state of the FindBestSelection function for this simple sd-DNNF. We propagated the leaves to the **And** nodes using lines 8-13 of Alg. 2.2.

This is the final state of the algorithm, shown in Fig. 2-4. We now return η on line 28, where $\eta\,(\text{o1}) = \text{a2}$ and $\eta\,(\text{o3}) = \text{a5}$.

2.2.2 Get-Solution-From-Selection Example

The GetSolutionFromSelection algorithm, Alg. 2.3, is determining the leaves connected to the root in the modified sd-DNNF. We mark all the nodes that have a path from the root to themselves, and we record which nodes are marked in the Marked variable. We store the set of symbols of our solution in the variable S. Initially, the root o1 is marked, so Marked$= \{\text{o1}\}$. The initial state at the start of the main loop on line 3 is shown in Fig. 2-5. We denote membership in Marked by coloring the marked nodes black.

The main loop runs from the root down to the leaves, so the first node visited

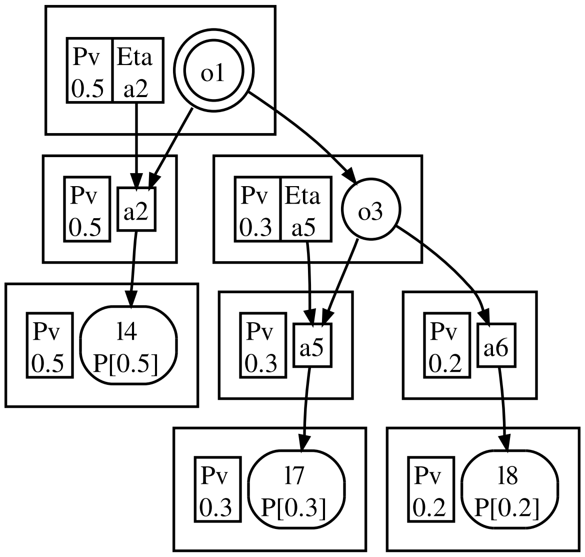


Figure 2-4: This figure shows the final state of the FindBestSelection function for this simple sd-DNNF, just prior to returning η. We propagated the probabilities of $\mathbf{P}_V$ to the root using lines 16-23 of Alg. 2.2. We also set η for both **Or** nodes using line 24 of Alg. 2.2.

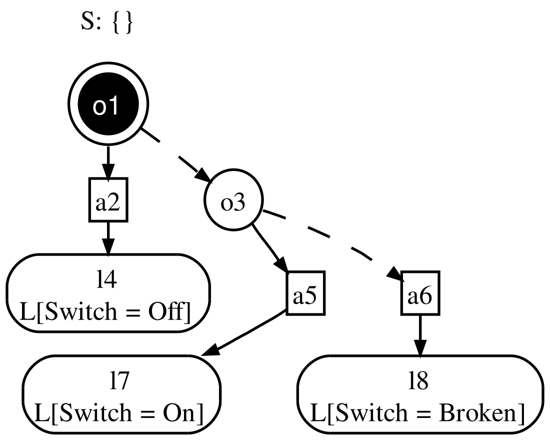


Figure 2-5: This figure shows the initial state of the GetSolutionFromSelection function for this simple sd-DNNF, just after executing lines 1 and 2 of Alg. 2.3. Initially the only marked node is the root o1. Marked nodes are black, while the remaining white nodes are not marked. The solid lines connecting the nodes represent the edges that are part of the modified sd-DNNF, while the dashed lines are currently suppressed by the **Or** node choice stored in η.

is the root o1. The node o1 is marked, as we stated initially, and is an **Or** node. We thus execute line 16. Since $\eta\,(\text{o1}) = \text{a2}$, we add a2 to Marked. Marked is now {o1, a2}. This state is shown in Fig. 2-6.

We then visit the node a2, which is marked, and execute the lines 11 to 13. This marks all of the children of a2, in this case only l4. Thus, after executing lines 11 to 13, Marked is now {o1, a2, l4}. We then visits o3, but o3 is not marked, so we skip over o3. The node l4 is then visited, executing line 8. Since $\mathcal{L}_L\,(\text{l4}) =$ "Switch = Off", we add this symbol to S: $S = \{$"Switch = Off"$\}$. The nodes a5, a6, l7, and l8 are then visited in that order, but none of them are marked. The main loop is now done and the algorithm is ready to return S on line 21. This state is shown in Fig. 2-7.

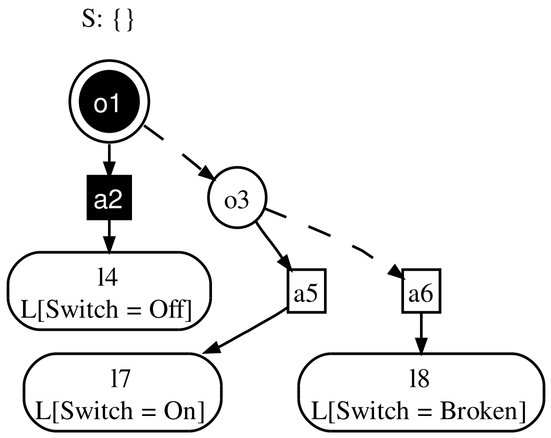


Figure 2-6: This figure shows the state of the GetSolutionFromSelection function for this simple sd-DNNF after executing line 16 of Alg. 2.3 with v = o1 on Fig. 2-5. This marks a2 as $\eta\,(\text{o1}) = \text{a2}$. Marked nodes are black, while the remaining white nodes are not marked. The solid lines connecting the nodes represent the edges that are part of the modified sd-DNNF, while the dashed lines are currently suppressed by the **Or** node choice stored in η.

2.3 Summary

This chapter described the prior work of [8] and [1], an algorithm for extracting the best solution from an sd-DNNF. The algorithm requires $O(|E| + |V|)$ time and space. The algorithm works in two parts, the first part passes from the leaves to the root, deciding along the way which sub-tree of **Or** nodes is the optimal choice while propagating the probability of the sub-trees to the root. The second part uses the selection of the first part, which is defined by η, to extract a solution.

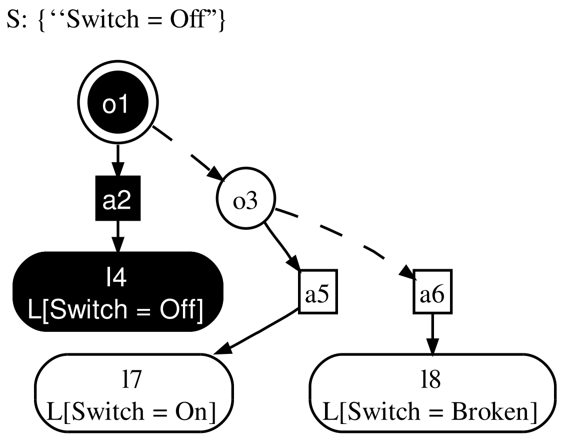


Figure 2-7: This figure shows the final state of the GetSolutionFromSelection function for this simple sd-DNNF. After Fig. 2-6, we have executed line 12 of Alg. 2.3 with $v =$ a2 and $n =$ l4, thus marking l4. We then executed line 8 with $v =$ l4, adding “Switch = Off” to our solution S. This S is then returned on line 21. Marked nodes are black, while the remaining white nodes are not marked. The solid lines connecting the nodes represent the edges that are part of the modified sd-DNNF, while the dashed lines are currently suppressed by the **Or** node choice stored in η.

Chapter 3

K-Best-Solutions Algorithm

We are interested in finding the k best solutions of the sd-DNNF $\langle V, E, \mathcal{L}_L, \mathcal{L}_{\mathbf{P}}, \times, > \rangle$. We will now extend the work algorithm developed in Chapter 2 to support k solutions. The basic principle of this extension is that we can record at each node the k best predecessors, rather than just the best predecessor. Each node will have up to k selections recorded. This requires recording significantly more information than η required before, specifically we will need to record predecessor information for both **And** and **Or** nodes, as each node will be able to choose among the k selections of each child. We extend our rules from page 27 to support k solutions:

1. For each leaf node l, the probability of the leaf node is $\mathcal{L}_{\mathbf{P}}(l)$.

2. For each **And** node a, we want the k best combinations of its children using $\times$, where a combination includes a selection from each child of a. Lets assume that a has p children, v_1, …, v_p. Each child will have between 1 and k selections recorded. If we denote the selections of a child as $\mathrm{Sel}(v_i)$, then there will be $\prod_{i=1}^{p} |\mathrm{Sel}(v_i)|$ combinations. The probability of a combination is computed by using $\times$ to combine the probabilities of all of the selections included in the combination. The k best selections for a are the k most probable combinations,

ordered by $>$.

3. For each **Or** node o, we also want the k best selections from among its children. Since an **Or** node is making a choice among its children, we consider all the selections of the p children of o and choose the k best selections among all of them, ordered by $>$.

Lets first start with an example of rule 2. For our example, the **And** node a has 2 children, v_1 and v_2 and we have $k = 3$. Node v_1 has 3 selections, with probabilities 0.3, 0.2, and 0.1. Node v_2 has 2 selections, with probabilities 0.5 and 0.2. We will denote a combination as $\langle i, j \rangle$, where we have numbered $\mathrm{Sel}(v_i)$ from 1 to k, and so i is the i^{th} selection of v_1 and j is the j^{th} selection of v_2. There are 6 combinations of the selections of a: $\langle 1, 1 \rangle$, $\langle 1, 2 \rangle$, $\langle 2, 1 \rangle$, $\langle 2, 2 \rangle$, $\langle 3, 1 \rangle$, and $\langle 3, 2 \rangle$. Assuming multiplication for $\times$, the probability of these combinations are 0.15, 0.06, 0.1, 0.04, 0.05, and 0.02, respectively. The 3 best combinations, since $k = 3$, are 0.15, 0.1 and 0.06, corresponding to $\langle 1, 1 \rangle$, $\langle 2, 1 \rangle$, and $\langle 1, 2 \rangle$, respectively. The selections $\mathrm{Sel}(a)$ are set to these three combinations.

Now consider a similar example of rule 3 for an **Or** node o. The node has 2 children, v_1 and v_2, where v_1 has 3 selections and v_2 has 2 selections. We let $k = 3$. We will use the probabilities of 0.3, 0.2, and 0.1 for v_1's selections, and 0.2 and 0.1 for v_2's selections. The top 3 selections for o are selection 1 and 2 of v_1 and selection 1 of v_2, and so $\mathrm{Sel}(o)$ are set to these three selections. Note that we assume a total ordering with $>$, so, for example, we can us the index of v_i in V_O to break ties when the probabilities are equal.

Rules 2 and 3 both require a probability per each of their children's selections, so we extend the probability recording $\mathbf{P}_V$ of Chapter 2 to include both the node and the selection: $\mathbf{P}_V(v, i)$. We need a way to know how many selections a node actually has, as it may be less than k, so we define a new function#Sel that returns the number of selections of a node: $\#\text{Sel}(v)$.

As with Chapter 2, we are generating a modified graph that describes the selections we want to find. In the modified graph, every node is effectively replicated once for every selection it records, thus nodes are indexed by the sd-DNNF's node and the selection number. We denote this $\langle v, j\rangle$ for a node v and the j^{th} selection. A leaf always has exactly 1 selection, so for a leaf node l, our modified sd-DNNF graph has the one node $\langle l, 1\rangle$.

To efficiently extract the k best solutions, given we've computed the k best selections, we recorded more information than the algorithm of Chapter 2. For each selection of an **Or** node o, we need to know which child's selection was chosen for each of o's selections. We thus extend η to be a function $O \times \{1, \ldots, k\} \to V \times \{1, \ldots, k\}$, where we constrain $\eta(o, i) = \langle v, j\rangle$ such that $v \in \text{Children}(o)$, $i \leq \#\text{Sel}(o)$, and $j \leq \#\text{Sel}(v)$. This function connects selections of o to selections of v.

For **And** nodes, we now need to know which combination of its children was chosen for each of the **And** node's selections. We will record this information in ξ. Recall that for an **And** node a, the i^{th} best selection of a is a combination of the selections of the children of a. We define ξ as the function $A \times \{1, \ldots, k\} \times V \to \{1, \ldots, k\}$, where we constrain $\xi(a, i, v) = j$ such that $v \in \text{Children}(a)$, $i \leq \#\text{Sel}(a)$, and $j \leq \#\text{Sel}(v)$. ξ records the selection of the child v corresponding to the i^{th} best selection of a, specifically the selection $\langle v, j\rangle$. In Chapter 2 where $k = 1$, a, as well as every child of a, only has one selection. There is only one combination possible when $k = 1$, which is the first and only selection of every child of a. Thus, the function ξ evaluates to 1 for every node and is omitted from the algorithms of Chapter 2.

To illustrate ξ, lets look back at the **And** node example given above, where a has two children, v_1 and v_2. The best 3 selections were $\langle 1, 1\rangle$, $\langle 2, 1\rangle$, and $\langle 1, 2\rangle$, in that order. Then for this fixed a, $\xi(a, i, v)$ defines a 3×2 table:

$$\begin{array}{cc|c|c|}
 & & \multicolumn{2}{c}{v} \\
 & & v_1 & v_2 \\
\hline
 & 1 & 1 & 1 \\
\hline
i & 2 & 2 & 1 \\
\hline
 & 3 & 1 & 2 \\
\hline
\end{array}$$

where, for instance, the entry at $(3, v_2)$ has a value of 2, the value of $\xi(a, 3, v_2)$. This entry means that $\langle a, 3\rangle$ is connected to $\langle v_2, 2\rangle$. $\langle a, 3\rangle$ is also connected to $\langle v_1, 1\rangle$.

The fully computed η and ξ functions define up to k selections, where the number of selections defined is the number of selections of the root node, $\#\mathrm{Sel}(r)$. Given the up to k selections defined by η and ξ, we can extract the corresponding solutions. As in 2, the solutions are defined by paths from the root to the leaves, in the graph modified by η and ξ. The i^{th} solution is the set of $\mathcal{L}_L(l)$ of all the leaves that have a path from the i^{th} selection of the root node. A path for the i^{th} selection of the root starts at the node $\langle r, i\rangle$. For every **And** node $\langle a_j, i_1\rangle$ along the path at position j, the node $\langle v_{j+1}, i_2\rangle$ at position $j+1$ in the path must be such that $\xi(a_j, i_1, v_{j+1}) = i_2$. That is to say that $\langle a_j, i_1\rangle$ connects to $\langle v_{j+1}, i_2\rangle$ in the modified graph. For every **Or** node $\langle o_j, i_1\rangle$ along the path at position j, the node $\langle v_{j+1}, i_2\rangle$ at position $j+1$ in the path must be such that $\eta(o_j, i_1) = \langle v_{j+1}, i_2\rangle$. That is to say that $\langle o_j, i_1\rangle$ connects to $\langle v_{j+1}, i_2\rangle$ in the modified graph.

We can extend the notion of marking developed in Chapter 2 by noting that each node's selection may be part of any subset of the k root selections, but that each root selection i, will include either one or zero selections of the node. There will never be more than one selection, as we noted before, due to the decomposition property of an sd-DNNF, as a selection necessarily forms a tree in the modified graph. For each node, rather than storing just a marking as before, we store k markings. Each marking $m(v, i)$ either takes on the special value $\perp$ or a value j that specifies the

j^{th} selection of the node v is part of the i^{th} root selection. Initially all the marks are $\perp$, except the root markings, for which $m(r,i) = i$ for each i from 1 to $\#\text{Sel}(r)$. To extract the k solutions, we walk over the original sd-DNNF from the root to the leaves, propagating the k markings to the children.

For an **And** node a, for each i such that $m(a,i) \neq \perp$, and for each child v of a, we set $m(v,i) = \xi(a, m(a,i), v)$. If $\xi(a, m(a,i), v) = j$, then this records that $\langle v, j\rangle$ is part of the i^{th} solution.

For an **Or** node o and for each i such that $m(o,i) \neq \perp$, let $\langle v, j\rangle = \eta(o, m(o,i))$. Then we set $m(v,i) = j$, again noting that $\langle v, j\rangle$ is part of the root selection i.

For a leaf node l, $m(l,i)$ is either $\perp$ or 1 as the leaf always has exactly one selection. Thus, for each $m(l,i) = 1$, the i^{th} solution includes the symbol of l, $\mathcal{L}_L(l)$.

We will now present an algorithm to compute the k most probable solutions. The sub-routines of this algorithm have the hierarchy shown in Fig. 3-1.

3.1 Find-K-Best-Solutions Algorithm

Algorithm 3.1: FindKBestSolutions(V_O, E, $\mathcal{L}_L$, $\mathcal{L}_\mathbf{P}$, $\times$, $>$, k)

```
1 $\langle \eta, \xi, \#_r \rangle \leftarrow$ FindKBestSelections($V_O$, $E$, $\mathcal{L}_\mathbf{P}$, $\times$, $>$, $k$) ;
2 $\mathcal{S}_k \leftarrow$ GetKSolutionsFromSelections($V_O$, $E$, $\mathcal{L}_L$, $\eta$, $\xi$, $\#_r$) ;
3 return $\mathcal{S}_k$;
```

As with the $k = 1$ algorithm, we break this algorithm down into two parts. The first part makes a pass from the leaves to the root, computing η and ξ. This involves internally computing up to k probabilities per node. Together, η and ξ define a superset of between 1 and k selections. The value $\#_r$ specifies the number of selections defined by η and ξ, between 1 and k. The second part of the algorithm makes a pass from the root to the leaves, extracting the $\#_r$ best solutions from the

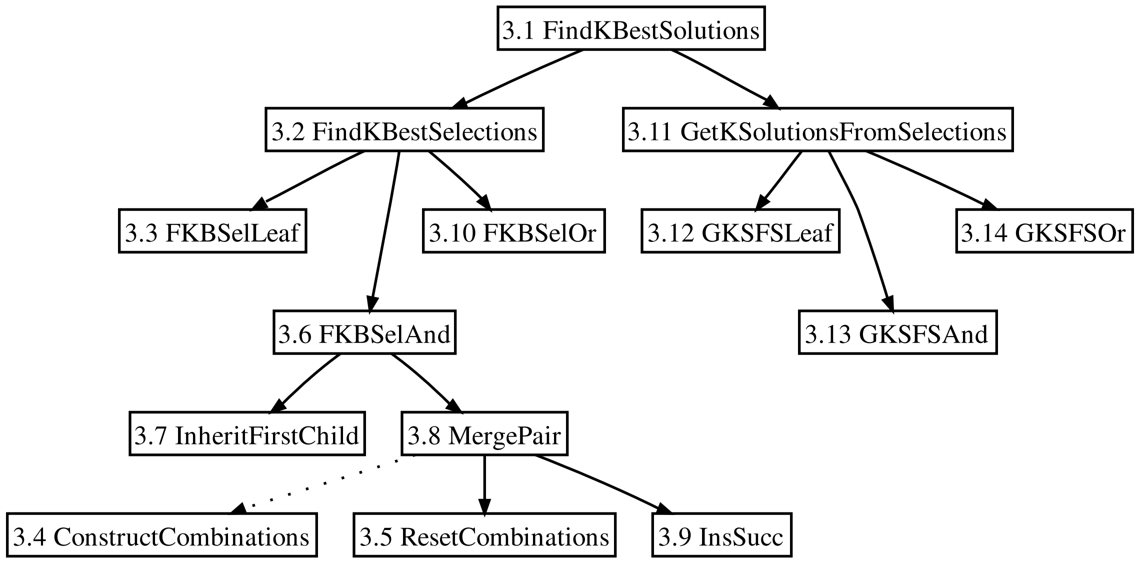


Figure 3-1: This diagram shows how the various algorithms of this chapter are related. The top-level algorithm is FindKBestSolutions. The ConstructCombinations function is the only unusual item in this diagram, as it is expected that it will be run prior to running FindKBestSolutions so that its output can be used by MergePair. ConstructCombinations only depends on k.

selections. These are then returned.

3.2 Find-K-Best-Selections Algorithm

Algorithm 3.2: FindKBestSelections(V_O, E, $\mathcal{L}_\mathbf{P}$, $\times$, $>$, k)

```
for $i = |V|$ to 1  do
    $v \leftarrow V_O[i]$ ;
    switch $v$ in
        case $v \in L$
            $\langle \mathbf{P}_V, \#\text{Sel} \rangle \leftarrow$ FKBSelLeaf($v, \mathbf{P}_V, \#\text{Sel}, \mathcal{L}_\mathbf{P}$) ;
        end
        case $v \in A$
            $\langle \mathbf{P}_V, \#\text{Sel}, \xi \rangle \leftarrow$ FKBSelAnd($v, \mathbf{P}_V, \#\text{Sel}, \xi, \times, >$) ;
        end
        case $v \in O$
            $\langle \mathbf{P}_V, \#\text{Sel}, \eta \rangle \leftarrow$ FKBSelOr($v, \mathbf{P}_V, \#\text{Sel}, \eta, >$) ;
        end
    end
end
return $\langle \eta, \xi, \#Sel(1) \rangle$ ;
```

The first part of Alg. 3.1, as with the $k = 1$ algorithm, is processing the sd-DNNF from the leaves to the root. This algorithm, Alg. 3.2, is computing the k best selections using dynamic programming. At each node we compute and cache the k most probable selections rooted at that node, where a selection is summarized at each node based on its children's summaries. For **And** nodes, a selection is summarized by specifying, for each child, one of the child's selections. For **Or** nodes, a selection is summarized by specifying a child and a selection of that child.

The algorithm has three update rules, one for each type of node: L, A, and O. These three rules are shown on page 43. We have broken these three rules into three functions: rule 1 is implemented in Alg. 3.3, rule 2 is implemented in Alg. 3.6, and rule 3 is implemented in Alg. 3.10.

Alg. 3.2, between lines 1-14, is iterating over the nodes of the sd-DNNF, from the leaves to the root, invoking the appropriate rule. These rules update $\mathbf{P}_V$, #Sel, η, and ξ, as appropriate. $\mathbf{P}_V$ stores the probability of each node's k best selections. For leaves, this always has one entry, from $\mathcal{L}_\mathbf{P}$. For **And** and **Or** nodes, this stores the probability of each selection, sorted from most probable to least probable. Keeping this list sorted makes both **And** and **Or** node computations much more efficient. #Sel stores exactly how many selections are available at each node. This will be between 1 and k. η records a selection summary for **Or** nodes, with one summary per selection. ξ records a selection summary for **And** nodes, with one summary per selection and child pair. We store each of these variables per node, for efficient access, passing ξ and η to the second step by passing our annotated graph.

This algorithm returns on line 15, where we return our selection summaries collectively in ξ and η along with the number of selections found at the root, which is exactly the number of corresponding solutions. In general, unless k is greater that the total number of solutions in the sd-DNNF, or unless we suppress solutions with value less than a certain amount, the number of solutions found at the root will be exactly k. In our motivating domain of estimation, for example, we suppress solutions with 0 probability.

We will now present algorithms 3.3, 3.6, and 3.10 in sections 3.2.1, 3.2.2, and 3.2.9, respectively.

3.2.1 Find-K-Best-Selections Leaf-case Algorithm

The leaf node case for finding the k best selections, Alg. 3.3, turns out to be nearly identical to that of Alg. 2.2. We set $\mathbf{P}_V(l,1) = \mathcal{L}_L(l)$ on line 1 and record that we only have 1 selection on line 2. That's all that we need to do for leaf nodes.

```
Algorithm 3.3: FKBSelLeaf($l$, $\mathbf{P}_V$, #Sel, $\mathcal{L}_\mathbf{P}$)
1 $\mathbf{P}_V(l, 1) \leftarrow \mathcal{L}_\mathbf{P}(l)$ ;
2 $\#\mathrm{Sel}(l) = 1$ ;
3 return $\langle \mathbf{P}_V, \#Sel \rangle$ ;
```

	0.4	0.3	0.2	0.1
0.4	**0.16**	**0.12**	0.08	0.04
0.3	**0.12**	**0.09**	0.06	0.03
0.2	0.08	0.06	0.04	0.02
0.1	0.04	0.03	0.02	0.01

Table 3.1: This table illustrates the combinations of the children of a hypothetical **And** node with two children when $k = 4$. The two children have identical distributions of 0.4, 0.3, 0.2, and 0.1 for selections 1, 2, 3, and 4, respectively. The upper-left region circumscribes all combination of the two children that could ever be part of the best 4 selections of the **And** node. The four bold values forming a square in the upper left are the 4 best selections for this example.

Time and Space Analysis We store all variables indexed by an sd-DNNF node with the sd-DNNF node itself, so all look-up times are $O(1)$. Since a leaf always stores exactly one answer, a leaf need only have $O(1)$ space to store the two values. Thus, FKBSelLeaf requires $O(1)$ time and space.

3.2.2 Find-K-Best-Selections And-case Algorithm

The **And** node case for finding the k best selections, Alg. 3.6, requires finding the k best combinations of its children's selections. If this node a has $|E_a|$ children and each has k solutions, then there are $k^{|E_a|}$ combinations; however, we are only interested in k of them. Much less work is required to extract only k solutions, which we will quantify momentarily.

Lets start with an example. Let $k = 4$, $c = 2$, and $\mathbf{P}_V$'s entries 1 through 4 are 0.4, 0.3, 0.2, and 0.1, respectively, for both children. The combination of these pair

	0.4	0.3	0.2	0.1
0.4	**0.16**	**0.12**	0.08	0.04
0.3	**0.12**	0.09	-	-
0.25	**0.10**	-	-	-
0.05	0.04	-	-	-

Table 3.2: This table illustrates a second possible combination of the 4 best children, again in bold. We have omitted those entries that could never be optimal.

	0.5	0.2	0.2	0.1
0.35	**0.175**	0.07	0.07	0.035
0.3	**0.15**	0.06	-	-
0.2	**0.1**	-	-	-
0.15	**0.075**	-	-	-

Table 3.3: This table illustrates a third possible combination of the 4 best children, again in bold. We have omitted those entries that could never be optimal.

of children is illustrated in Table 3.1. The biggest combinations of the children are $\langle 1, 1 \rangle$, $\langle 1, 2 \rangle$, $\langle 2, 1 \rangle$, and $\langle 2, 2 \rangle$. The double-edge region defined around the upper left section of the matrix illustrates the region in which all k-best combinations reside. We illustrate two other combinations in tables 3.2 and 3.3, which with their reflections, represent all k-best combinations of 4.

The key to realize here is that, since our probabilities are sorted by $>$, and we only want the first k of them, we can start with the guaranteed best pair, the combination $\langle 1, 1 \rangle$. We will now show why this is the guaranteed best pair. The product of two numbers is monotonically increasing (unless one value is 0); when you increase either value of the product, the value of the product increases. Thus, the product of the two largest values will be the largest value among all products. The next best product will be a combination of the largest value of one of the two children and the second largest of the other child. Again, if we select the second best value for both children, the result will be smaller than if we only decrease one of the values.

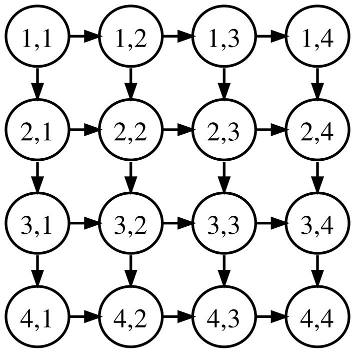


Figure 3-2: This figure shows which combinations of an **And** node is enabled in the case where $k = 4$ and the **And** node has two children. A node is enabled if all of its parents have been included in the solution. Thus the root node, $1, 1$, is always enabled.

For a combination $\langle i, j\rangle$, the children of this combination are the combinations $\langle i+1, j\rangle$ and $\langle i, j+1\rangle$, subject to neither child index exceeding k. The parent/child relationship between combinations is illustrated for $k = 4$ in Fig. 3-2. The value of a combination $\langle i, j\rangle$ is always greater than that of its children. Again, this trivially holds as one of the two values of the child is equal to one of this combination's values and the other child's value is less than this combination's other value.

As a corollary, a combination $\langle i, j\rangle$ need not be considered until all of its parents are considered. Since $\langle 1, 1\rangle$ is the only node with no parents, this is the only possible maximal node, as we said above. We use this fact in Alg. 3.6 to pre-build a structure C_a to hold all possible combinations of k and then only consider among those combinations that have had all their parents selected.

To bound the number of combinations needed in C_a, we note that to ever consider the candidate at $\langle i, j\rangle$, both its parents along with their parents and so on back to the first candidate $\langle 1, 1\rangle$ must have all be already accepted as part of the **And** node's selection. This means there has been less than k combinations accepted, and $\langle i, j\rangle$ may be the k^{th} combination, so $i * j \leq k$. Since everything is positive, $j \leq \frac{k}{i}$. All the

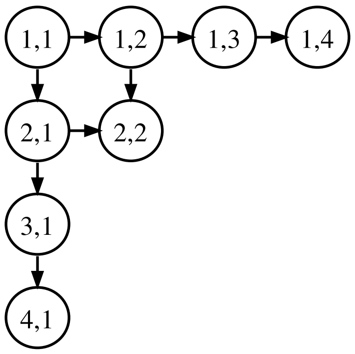


Figure 3-3: This figure shows the set of combinations that are part of the C_a structure when $k = 4$. A combination will be enabled if both of its parents are accepted as part of the k best selections. $1, 1$ is always enabled.

values are integers, so $j \leq \left\lfloor \frac{k}{i} \right\rfloor$. i varies from 1 to k, so the total number of possible candidates is:

$$\sum_{i=1}^{k} \left\lfloor \frac{k}{i} \right\rfloor$$

Each term of this equation is the floor of k times a term in the harmonic series[5]. The sum of the first k terms of the harmonic series is upper-bounded by $(\log k) + 1$, and thus the number of possible candidates is upper-bounded by $(k \log k) + k$ or $O(k \log k)$.

Fig. 3-3 shows the combinations of C_a trimmed down from Fig. 3-2. Combinations are all indexed to allow for $O(1)$ look-up. Specifically, a combination c_a in C_a is a tuple $\langle i, j_1, j_2, i_1, i_2, \#_P, \#_E \rangle$. The combination is located at $C_a[i]$. The pair $\langle j_1, j_2 \rangle$ is the combination of c_a. The two values i_1 and i_2 are indices in C_a referring to the two children of c_a. These can have the special value $\perp$ if the combination has only 0 or 1 child. $\#_P$ is the number of parents of c_a. $\#_E$ is set when we use C_a, and correspond to the current number of un-accepted parents. When we reset C_a, we set $\#_E = \#_P$. Each time a parent is accepted, it decrements its two children's $\#_E$ value. When

$\#_E$ reaches 0, the combination is enabled and can be added to the queue of enabled combinations. We require that the combination $\langle j_1, j_2\rangle = \langle 1, 1\rangle$ have a known index so we can start Alg. 3.6. Our algorithm for indexing C_a currently indexes the $\langle 1, 1\rangle$ combination as the last index of C_a.

3.2.3 Construct-Combinations Algorithm

Algorithm 3.4: ConstructCombinations(C_a, k, j_1, j_2)

```
1  if $j_1 * j_2 > k$ then
2      return $\langle C_a, \perp\rangle$ ;
3  end
4  if $\langle j_1, j_2\rangle$ is in $C_a$ then
5      $i \leftarrow$ Index of $\langle j_1, j_2\rangle$ in $C_a$ ;
6      return $\langle C_a, i\rangle$ ;
7  end
8  $\langle C_a, i_1\rangle \leftarrow$ ConstructCombinations($C_a$, $k$, $j_1 + 1$, $j_2$) ;
9  $\langle C_a, i_2\rangle \leftarrow$ ConstructCombinations($C_a$, $k$, $j_1$, $j_2 + 1$) ;
10 $\#_P \leftarrow 0$ ;
11 if $j_1 > 1$ then
12     $\#_P \leftarrow \#_P + 1$ ;
13 end
14 if $j_2 > 1$ then
15     $\#_P \leftarrow \#_P + 1$ ;
16 end
17 $i \leftarrow |C_a| + 1$ ;
18 $C_a[i] \leftarrow \langle i, j_1, j_2, i_1, i_2, \#_P, \#_P\rangle$ ;
19 return $\langle C_a, i\rangle$ ;
```

We show the code used to construct and initialize C_a in Alg. 3.4 and 3.5, respectively. Alg. 3.4 is assumed to run prior to the algorithm of this chapter, Alg. 3.1, as the data of C_a with the exception of $\#_E$ is constant for a constant k. The recursive Alg. 3.4 is called as ConstructCombinations(C_a, k, 1, 1), with an empty C_a, and returns a constructed C_a along with the location i of our entry $c_a = \langle i, 1, 1, *, *, 0, 0\rangle$.

In general, this algorithm returns the updated C_a and the index of the entry with the combination $\langle j_1, j_2 \rangle$, or $\perp$ if it isn't one of the possible combinations. This is done recursively, where we return the index of a entry if it has already been inserted into C_a and we insert a new entry into C_a, otherwise. An entry is inserted after all of its children have been inserted. If we assume $k \log k$ entries are generated, this algorithm looks through this list once for each edge in the graph to be sure the entry has not yet been created, where there are two edges per entry. Otherwise, it only performs $O(1)$ steps computing the elements of the new entry.

Line 1 is the base case of Alg. 3.4. We return the non-entry index $\perp$ if there is no way for both parents of this entry to be accepted at the same time. Line 4 makes sure that we do not insert a combination more than once. If the combination already exists, we return the index of the combination's entry. As stated above, this is an $O(k \log k)$ operation in general[1]. Lines 8 and 9 recursively look-up or construct the two children of this combination.

Lines 10-16 set the number of parents of the entry. This is easily computed as most entries have two parents. An entry that has a value of 1 for one of its two combination values has only 1 parent, as the parent in the value-of-1 direction would have a 0 value, and 0 is an invalid value (our values start at 1). The combination $\langle 1, 1 \rangle$ is the only combination where both values of the combination are 1, and so it has 0 parents.

Line 17 computes the index of this new entry. We insert this entry at the end of C_a. We then add our new node on line 18 and return it on line 19.

Time and Space Analysis We do intend ConstructCombinations to be an off-line algorithm, as it generates a constant structure that depends only on k, thus the time

[1]We could speed this up to $O(\log(k \log k))$ if we add an explicit indexing map or $O(1)$ if we used an appropriate hashing function of $\langle j_1, j_2 \rangle$. These optimizations are ignored in this thesis because this is a pre-processing step and is fast enough for all our values of k.

it takes to generate C_a is not included in the other algorithms, just the space. Lines 1 and 10-19 are all $O(1)$ operations: reading or setting a field, or appending to the end of a vector. We stated above that line 4 is currently just a linear search through a vector of length $O(k \log k)$, which is thus an $O(k \log k)$ operation. We could create an index that maps $\langle j_1, j_2 \rangle$ to an index in C_a or $\perp$ to reduce this search cost, using a map or hash map. Lines 8 and 9 are recursive calls. We construct at most $O(k \log k)$ entries and we only recurse twice for constructed entries, so ConstructCombinations is called at most twice as many times as there are entries, still $O(k \log k)$. We only run line 4 for entries that we would otherwise construct. Since an entry has at most two parents, line 4 is run at most twice per entry constructed, an $O(n^2)$ step for the algorithm, where $n = k \log k$. Thus, the overall complexity of ConstructCombinations is $O(k^2 \log^2 k)$ time. We generate only enough space to hold the entries we want, so the space required is the space of C_a, which we explained above is bounded by $O(k \log k)$.

3.2.4 Reset-Combinations Algorithm

Algorithm 3.5: ResetCombinations(C_a)

```
1 foreach  $\langle i, j_1, j_2, i_1, i_2, \#_P, \#_E \rangle = C_a[i]$ do
2     $C_a[i] \leftarrow \langle i, j_1, j_2, i_1, i_2, \#_P, \#_P \rangle$ ;
3 end
4 return $C_a$ ;
```

The Alg 3.5 is just responsible for setting $\#_E = \#_P$ for each entry in C_a, specifically on line 2. This is done iteratively. Thus the complexity of this algorithm is $O(|C_a|)$ time where $|C_a|$ is $O(k \log k)$.

3.2.5 The Find-K-Best-Selections And-case Algorithm

Algorithm 3.6: FKBSelAnd(a, $\mathbf{P}_V$, #Sel, ξ, $\times$, $>$)

```
1  $e \leftarrow$ some $\langle a, n_{\text{Prev}} \rangle \in E$ ;
2  $\langle \#_a, \mathbf{P}_a, \beta_\xi \rangle \leftarrow$ InheritFirstChild($n_{\text{Prev}}$, #Sel, $\mathbf{P}_V$) ;
3  foreach $\langle a, n \rangle \in E \setminus \{e\}$ do
4      $\langle \#_a, \mathbf{P}_a, \beta_\xi, n_{\text{Prev}} \rangle \leftarrow$ MergePair($\beta_\xi$, $n_{\text{Prev}}$, $n$, $\mathbf{P}_a$, $\mathbf{P}_V$, $\#_a$, #Sel$(n)$) ;
5  end
6  for $i = 1$ to $\#_a$ do
7      $\mathbf{P}_V(a, i) \leftarrow \mathbf{P}_a(i)$ ;
8      $\langle n, j_1 \rangle \leftarrow \langle n_{\text{Prev}}, i \rangle$ ;
9      while $n \neq \perp$ do
10         $\langle n', j_1', j_2 \rangle \leftarrow \beta_\xi(n, j_1)$ ;
11         $\xi(a, i, n) \leftarrow j_2$ ;
12         $\langle n, j_1 \rangle \leftarrow \langle n', j_1' \rangle$ ;
13     end
14 end
15 #Sel$(a) = \#_a$ ;
16 return $\langle \mathbf{P}_V, \#Sel, \xi \rangle$ ;
```

Lets now explain the parts of Alg. 3.6. This algorithm is pair-wise combining all of the children's selections into this **And** node a's best k selections. The algorithm starts out by inheriting the selections of one of its children. Then, for all the other children, it computes the best k selections from the combination of a's current selections and the next child's selections. Once all children have been combined, the algorithm's current k best selections are the actual k best selections and the algorithm is done.

In Alg. 3.6, line 1 starts out by getting some out-going edge of the **And** node a, with some child n. Line 2 inherits the best selections of the child n as a's best selections, noting which child these selections came from, using Alg. 3.7. The variable $\#_a$ stores the current number of selections, between 1 and k. The variable $\mathbf{P}_a$ is a local version of $\mathbf{P}_V$ specific to a.

The variable β_ξ is used to compute the entries for ξ. β_ξ is an acyclic graph that

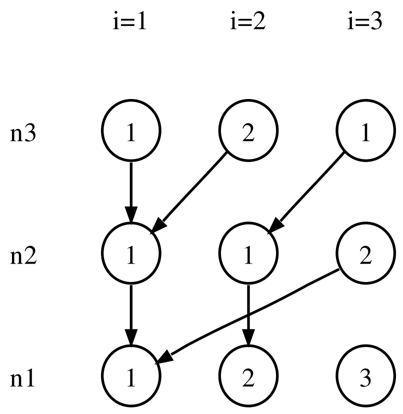


Figure 3-4: This is an example of a possible configuration of β_ξ for an example that assumes $k = 3$ and $|E_v| = 3$. The labels on the nodes are the value j for the entry $\beta_\xi(n, i) = \langle n_2, i_2, j\rangle$.

captures the best combinations of a child n of a with all other children that have already been combined. β_ξ is a function $V \times \{1, \ldots, k\} \rightarrow (V \cup \{\perp\}) \times \{1, \ldots, k\} \times \{1, \ldots, k\}$. For a particular entry $\beta_\xi(v, i) = \langle v_2, i_2, j\rangle$, the entry means that the modified node $\langle v, j\rangle$ is part of the i^{th} best combination of a and that the i_2^{th} best entry for v_2 is also part of the i^{th} best combination of a. A leaf of this graph an entry of the form $\langle \perp, 1, j\rangle$, for some j.

Consider an example where $k = 3$ and $|E_v| = 3$. In this example, the children are n1, n2, and n3 and all of them have three selections. These selections have probabilities such that the three best combinations of n1 and n2 are $\langle 1, 1\rangle$, $\langle 2, 1\rangle$, and $\langle 1, 2\rangle$. Given these three best combinations of n1 and n2, the probabilities are such that the three best combinations of these combinations and n3 are $\langle 1, 1\rangle$, $\langle 1, 2\rangle$, and $\langle 2, 1\rangle$, where the first number is the i^{th} best combination of n1 and n2. We can rewrite these combinations without indices as $\langle\langle 1, 1\rangle, 1\rangle$, $\langle\langle 1, 1\rangle, 2\rangle$, and $\langle\langle 2, 1\rangle, 1\rangle$, respectively. This example is depicted in Fig. 3-4. The three best combinations of a can be read from Fig. 3-4 by looking at the three sequences that start at the three top nodes $\langle \text{n3}, 1\rangle$, $\langle \text{n3}, 2\rangle$, and $\langle \text{n3}, 3\rangle$, respectively. Reading off the three sequences, in reverse – from n1 to n3, we get the same three best combinations $\langle 1, 1, 1\rangle$, $\langle 1, 1, 2\rangle$, and $\langle 2, 1, 1\rangle$. The nine entries of β_ξ that correspond to Fig. 3-4 are:

$$\begin{array}{lll}
\beta_\xi\left(\text{n3}, 1\right) = \langle \text{n2}, 1, 1\rangle & \beta_\xi\left(\text{n3}, 2\right) = \langle \text{n2}, 1, 2\rangle & \beta_\xi\left(\text{n3}, 3\right) = \langle \text{n2}, 2, 1\rangle \\
\beta_\xi\left(\text{n2}, 1\right) = \langle \text{n1}, 1, 1\rangle & \beta_\xi\left(\text{n2}, 2\right) = \langle \text{n1}, 2, 1\rangle & \beta_\xi\left(\text{n2}, 3\right) = \langle \text{n1}, 1, 2\rangle \\
\beta_\xi\left(\text{n1}, 1\right) = \langle \perp, 1, 1\rangle & \beta_\xi\left(\text{n1}, 2\right) = \langle \perp, 1, 2\rangle & \beta_\xi\left(\text{n1}, 3\right) = \langle \perp, 1, 3\rangle
\end{array}$$

Alg. 3.6 constructs β_ξ one row at a time, where Alg. 3.7 constructs the bottom row of β_ξ, and each subsequent row is added by Alg. 3.8. After calling Alg. 3.7 on line 2, β_ξ contains $\#_a$ entries, where the i^{th} entry is $\beta_\xi\left(n_{\text{Prev}}, i\right) = \langle \perp, 1, i\rangle$.

Lines 3-5 loop over all the remaining children of a, taking the k best combinations of a's current k best combinations and the child's k best combinations, inserting another row in β_ξ. This loop utilizes the function MergePair, Alg. 3.8. Finally, lines 6-15 copy the local versions of these variables over to the final version. Lines 8-13 copies the i^{th} best combination of a from β_ξ into $\xi\left(a, i, *\right)$. Line 8 sets our current node in β_ξ to the root node of the i^{th} best combination in β_ξ; the root is the i^{th} position of the top row. Lines 9-13 loop from the root node in β_ξ to the leaf, where at the end $n = \perp$. For each node in β_ξ visited, the algorithm grabs the node's data on line 10. This data specifies the next node in the sequence as well as a modified node $\langle n, j_2\rangle$ that belongs to i^{th} combination of a. The algorithm connects this modified node to $\langle a, i\rangle$ on line 11 and then the loop moves on to the next β_ξ node in the sequence on line 12.

3.2.6 Inherit-First-Child Algorithm

The first subroutine of Alg. 3.6 is InheritFirstChild, Alg. 3.7. This algorithm is responsible for initializing the **And** node's local versions of the selection variables based on the child's best selections. The **And** node inherits the selections of the child

Algorithm 3.7: InheritFirstChild(n, #Sel, $\mathbf{P}_V$)

```
1 $\#_a = \#\mathrm{Sel}\,(n)$ ;
2 for  $i = 1$ to $\#_a$ do
3     $\mathbf{P}_a\,(i) = \mathbf{P}_V\,(n, i)$ ;
4     $\beta_\xi\,(n, i) \leftarrow \langle \perp, 1, i \rangle$ ;
5 end
6 return $\langle \#_a, \mathbf{P}_a, \beta_\xi \rangle$ ;
```

node, so line 1 sets $\#_a$ to the number of selections of the child. The probabilities are the same and in the same order, so these can also be copied. The variable β_ξ is initialized for the child n to the selection number that was borrowed. These initialized values are returned on line 6.

Time and Space Complexity This algorithm performs an $O(1)$ step on line 1, copying the number of selections of a's first child. We then copy up to k values on lines 3 and 4. Both take $O(1)$ time. Thus, the time complexity of this algorithm is $O(k)$. The space required is dominated by β_ξ, requiring $O(k|E_a|)$ space, though this space is not specific to InheritFirstChild, as it is space returned to the calling function FKBSelAnd, Alg. 3.6.

3.2.7 Merge-Pair Algorithm

The other subroutine of Alg. 3.6 is MergePair, Alg. 3.8. This algorithm is responsible for computing the k best pairings between the current k best combinations of the **And** node's processed children and the next child's k best combinations. The variable β_ξ summarizes the combinations of the processed children. We use our combination structure C_a from Alg. 3.4 and 3.5 to help decide which combinations are available as we select our k best combinations. MergePair will extract up to k values and store them in β_ξ and a second probability vector. It is assumed these two prob-

Algorithm 3.8: MergePair(β_ξ, n_{Prev}, n, $\mathbf{P}_a$, $\mathbf{P}_V$, $\#_a$, $\#_n$)

```
1  Let $C_a$ be the pre-constructed combinations structure and $\#_{1,1}$ be the index of
   the combination $\langle 1, 1 \rangle$ ;
2  $C_a \leftarrow$ ResetCombinations($C_a$) ;
3  $\#'_a \leftarrow 0$ ;
4  $p \leftarrow \mathbf{P}_a(1) \times \mathbf{P}_V(n, 1)$ ;
5  Insert $\langle p, \#_{1,1} \rangle$ into $Q$ ;
6  while $\#'_a < k$ and $|Q| > 0$ do
7      $\langle p, i \rangle \leftarrow$ Remove best element of $Q$ ordered by $p$ using $>$ ;
8      $\langle i, j_1, j_2, i_1, i_2, \#_P, \#_E \rangle \leftarrow C_a[i]$ ;
9      $\#'_a \leftarrow \#'_a + 1$ ;
10     $\mathbf{P}'_a(\#'_a) = p$ ;
11     $\beta_\xi(n, \#'_a) \leftarrow \langle n_{\text{Prev}}, j_1, j_2 \rangle$ ;
12     $\langle Q, C_a \rangle \leftarrow$ InsSucc($Q$, $C_a$, $\mathbf{P}_a$, $\mathbf{P}_V$, $n$, $\times$, $>$, $\#_a$, $\#_n$, $i_1$) ;
13     $\langle Q, C_a \rangle \leftarrow$ InsSucc($Q$, $C_a$, $\mathbf{P}_a$, $\mathbf{P}_V$, $n$, $\times$, $>$, $\#_a$, $\#_n$, $i_2$) ;
14 end
15 return $\langle \#'_a, \mathbf{P}'_a, \beta_\xi, n \rangle$ ;
```

ability vectors, $\mathbf{P}_a$ and $\mathbf{P}'_a$, are swapped between pairings, so the probability vector is only copied once, when it is copied to $\mathbf{P}_V$. This algorithm uses a priority queue of possible candidate combinations to efficiently insert combinations and extract the best combination. Combinations are sorted by their value using $>$.

The algorithm starts out on line 2 by setting $\#_E = \#_P$ for all combinations in C_a. This marks each node as initially having none of their parents accepted. We initialize our next local best selection count to 0 on line 3. The best combination is $\langle 1, 1 \rangle$, and we compute the probability of this combination on line 4. We then insert this best value into the queue Q on line 5. We're now ready to extract the best k combinations between the **And** node's current best combinations and the new nodes best selections. Lines 6-14 are responsible for getting the next best combination, recording it, and the adding that combinations successors to Q, as appropriate. Successors, here, are defined by the relationship stored in C_a.

Within this loop, line 7 finds the next most probable combination, ordered with $>$.

Since a candidate is only inserted in the queue once all of its parents have already been added to $\mathbf{P}'_a$ and β_ξ, the maximal node in Q is the next most maximal combination of a. Line 8 looks up the combination associated with the index we got from Q. This gives us the combo $\langle j_1, j_2 \rangle$ and up to two successors i_1 and i_2. Line 9 increments our number of accepted combinations by one as we just got a new one off the queue. Line 10 sets the probability of our next accepted combination to p, the value we computed for sorting in Q. Line 11 points our next combination to the previous combination starting at $\langle n_{\text{Prev}}, j_1 \rangle$ and adds an entry for the child n based on its selection of j_2. Lines 12 and 13 call InsSucc on the first and second possible successors, respectively. InsSucc will update C_a by recording that one more parent of i_1 and i_2 has been accepted. If all of the child's parents have been accepted, InsSucc will compute the probability of the combination of the child and add it to the queue. Once the loop is done getting up to k combinations, the function returns the new local variables on 15.

3.2.8 Insert-Successor Algorithm

The last subroutine used by the **And** node case is the InsSucc algorithm. As was just stated, this routine is updating the value of $\#_E$ of the entry i in C_a. This involves subtracting one, as this function is called whenever a parent of this entry has been accepted. If $\#_E$ is then zero, then the entry's combination becomes enabled and we insert it in the queue. This requires first computing the probability of the combination.

There are two special cases for this routine. First, i may be equal to $\perp$, in which case this isn't actually referring to an entry and there isn't anything to do. Recall that this means that it was not possible for this child of the parent to have ever been enabled, so this child reference was set to $\perp$. The other case is that

Algorithm 3.9: InsSucc(Q, C_a, $\mathbf{P}_a$, $\mathbf{P}_V$, n, $\times$, $>$, $\max_1$, $\max_2$, i)

```
1  if $i = \perp$ then
2      return $\langle Q, C_a \rangle$ ;
3  end
4  $\langle i, j_1, j_2, i_1, i_2, \#_P, \#_E \rangle \leftarrow C_a[i]$ ;
5  if $j_1 > \max_1$ or $j_2 > \max_2$ then
6      return $\langle Q, C_a \rangle$ ;
7  end
8  $\#_E \leftarrow \#_E - 1$ ;
9  $C_a[i] \leftarrow \langle i, j_1, j_2, i_1, i_2, \#_P, \#_E \rangle$ ;
10 if $\#_E = 0$ then
11     $p \leftarrow \mathbf{P}_a(j_1) \times \mathbf{P}_V(n, j_2)$ ;
12     Insert $\langle p, i \rangle$ into $Q$ ordered by $p$ using $>$ ;
13 end
14 return $\langle Q, C_a \rangle$ ;
```

one or both of the nodes a and n may not have a full k selections. This matters because we cannot compute the probability of a combination if one of the values of the combination exceeds the number of selections of the corresponding node. We thus prune combinations that exceed our actual number of selections.

The algorithm starts out on line 1 by returning if i is $\perp$. Line 4 gets the entry in C_a corresponding to i so we can update $\#_E$. Before updating i, the algorithm returns if the combination of i exceeds the number of selections of either sd-DNNF node on line 5. Lines 8 and 9 update $\#_E$ in C_a. Lines 10-13 add the combination to Q if $\#_E$ is 0, which is to say if the combination is enabled. Line 11 computes the probability of the combination, while line 12 adds the combination to Q.

Runtime Analysis We will start with the InsSucc routine and work up to the Find-K-Best-Selections, And-case algorithm. Every time the InsSucc routine is called, it looks up an entry in C_a, decrementing the value $\#_E$. Lines 1-9 are all $O(1)$, as we assume we update in-place and that we access directly by index. Lines 11 and 12 are only run once per combination inserted into the queue. Line 11 applies $\times$ once, an $O(1)$ operation. Line 12's complexity depends on the size of the queue. For a queue of length $|Q|$, line 12 has a time complexity of $O(\log |Q|)$. This complexity arises from $O(\log |Q|)$ applications of $>$ to determine where in the heap the combination belongs. We will show that $|Q| \leq \left\lfloor \sqrt{2k} \right\rfloor + 1$, so line 12's complexity is $O(\log k)$. Thus, if the candidate is not added to the queue, InsSucc has $O(1)$ time complexity, and if it is added to the queue, InsSucc has $O(\log k)$ complexity. InsSucc requires $O(1)$ space for everything but Q. The queue, as we said, is of size $O(\sqrt{k})$.

We will now justify the claim that the queue will never be larger that $\left\lfloor \sqrt{2k} \right\rfloor + 1$. First, note that the algorithm accepts at most k combinations before terminating. Also note that we only add a combination $\langle i, j \rangle$ to the queue if all their parents have been accepted, and they will have only been accepted if all their parents have been accepted, etc. For $\langle i, j \rangle$ to be added to the queue, $(i * j) - 1 \leq k$ combinations have been accepted. Additionally, note that at most one combination per row and per column is enabled and thus in the queue. Lets assume without loss of generality that there are two in the same row i. Let the column of the two combinations be j_1 and j_2, such that $j_1 < j_2$. The combination $\langle i, j_1 \rangle$ is an ancestor of $\langle i, j_2 \rangle$, and both are only enabled. This violates our constraint that a combination be accepted prior to any of its children being enabled, and thus any of their descendants being enabled. So there is at most one enabled combination per row and column. The configuration with k combinations accepted that has the maximal number of combinations enabled has one enabled combination per row and per column, lets say w rows and h columns. This is maximal for a fixed area k as inserting just a row or column anywhere would increase the number of combinations accepted while not changing the number of enabled combinations. To compensate for the extra area added, we would need to remove a column or row, which would in turn reduce the number of enabled combinations. This maximal form is square, so $w = h$, and forms a triangle, which has area $\frac{1}{2}w^2$. The area of the triangle is equal to the number of accepted combinations, so $\frac{1}{2}w^2 \leq k$. Solving for w, we get $w \leq \sqrt{2k}$. Since w is an integer, a tighter bound is $w \leq \left\lfloor \sqrt{2k} \right\rfloor$. If we have at most one enabled combination per column, at most one enabled combination to the right of the right most accepted combination, and w columns of accepted combinations, then we can have at most $w + 1$ enabled combinations, or $\left\lfloor \sqrt{2k} \right\rfloor + 1$.

We showed previously that ResetCombinations has a complexity of $O(k \log k)$ time and $O(1)$ space, required to reset the $\#_E$ values of each entry of C_a. Thus, with the complexity of InsSucc, we can now analyze the complexity of the MergePair algorithm. The MergePair algorithm calls ResetCombinations once on line 2, requiring $O(k \log k)$ time. Lines 3-5 are $O(1)$ time operations. The queue Q is needed only for this subroutine, starting at line 5, and has a maximal space requirement of $O(\sqrt{k})$. Our loop from lines 6-14 will run at most k times. Within the loop, we dequeue an element from Q once on line 7 with complexity $O(\log k)$. Lines 8-11 perform $O(1)$ operations, setting values. Lines 12 and 13 each call InsSucc. InsSucc will insert at most $k + \left\lfloor \sqrt{2k} \right\rfloor$ combinations into Q, with an $O(\log k)$ time complexity each time something is inserted into Q. The remaining times it is called it has an $O(1)$ time complexity. Thus, the loop excluding the InsSucc part has an $O(k \log k)$ time complexity. The InsSucc part has $O((k + \left\lfloor \sqrt{2k} \right\rfloor) \log k) = O(k \log k)$ time complexity. Thus, together the whole loop has time complexity $O(k \log k)$. The final line 15 can have an $O(k)$ time complexity if the data $\mathbf{P}'_a$ is copied to $\mathbf{P}_a$ or $O(1)$ if the data is swapped. Given all three parts, the initial part, the loop, and the return (either version), the overall time complexity of MergePair is $O(k \log k)$. We will now summarize the space required. The C_a uses $O(k \log k)$ space. The Q uses $O(\sqrt{k})$ space. Our local copy of $\mathbf{P}'_a$ uses $O(k)$ space. The β_ξ uses $O(|E_a|k)$ space, where each call to MergePair adds $O(k)$ data to β_ξ. Thus, the total space required for this step is $O(|E_a|k + k \log k)$.

We can now finally determine the complexity of the Find-K-Best-Selections, And-case (FKBSelAnd) algorithm. First recall that the complexity of InheritFirstChild was $O(k)$ time and $O(|E_a|k)$ space. We assume that out-going edges of a node are stored with the node, so line 1 of Alg. 3.6 just involves selecting the first out-going edge, an $O(1)$ operation. Line 2 invokes InheritFirstChild once, returning our local variables $\langle \#_a, \mathbf{P}_a, \beta_\xi \rangle$. The local variables take $O(|E_a|k)$ space, where $|E_a|$ is the number of children of a, or equivalently the number of out-going edges. The algorithm then iterates over the remaining $|E_a| - 1$ edges on line 3, and for each edge, it invokes MergePair. MergePair requires $O(k \log k)$ time and space, so the loop requires $O(|E_a|k \log k)$ time and, as we need not keep the previous local variable copies, only $O(|E_a|k + k \log k)$ space. Lines 6-15 copy the local variables to their final location on the node, an $O(|E_a|k)$ time operation. All together, this algorithm has an $O(|E_a|k \log k)$ time and an $O(|E_a|k + k \log k)$ space complexity.

3.2.9 Find-K-Best-Selections Or-case Algorithm

The **Or** node case for finding the k best selections, Alg. 3.10, requires grabbing the best k selections from all its children. This can be likened to performing the traditional merge step of a merge-sort[6], with two modifications. First, there are multiple lists, not just two. There is a list per child, so since there are $|E_o|$ children, there are $|E_o|$ lists. Second, while each list may contain k elements, we are only interested in the first k merged elements, not all $k|E_o|$. We solve this problem by keeping a priority queue of the leading selections for each list. The algorithm repeatedly takes the best option from the queue and then adds the associated lists next best element to the queue, if any.

Alg. 3.10 starts out by getting a reference to all of its edges on line 1. We record the number of edges on line 2. Lines 3-7 setup our $|E_o|$ lists for merging, inserting each one into the priority queue Q. Since each child's selections are sorted, to get the

Algorithm 3.10: FKBSelOr(o, $\mathbf{P}_V$, #Sel, η, $>$)

```
1  $E_o \leftarrow$ All edges matching $\langle o, n \rangle \in E$, in any order ;
2  $\#_E \leftarrow |E_o|$ ;
3  for $i = 1$ to $\#_E$ do
4      $\langle o, n \rangle \leftarrow E_o[i]$ ;
5      $p \leftarrow \mathbf{P}_V(n, 1)$ ;
6      Insert $\langle p, n, 1, \#\mathrm{Sel}(n) \rangle$ in $Q$ ordered by $p$ using $>$ ;
7  end
8  $\#_o \leftarrow 0$ ;
9  while $\#_o < k$ and $|Q| > 0$ do
10     $\langle p, n, j, \max_j \rangle \leftarrow$ Remove best element of $Q$ ordered by $p$ using $>$ ;
11     $\#_o \leftarrow \#_o + 1$ ;
12     $\mathbf{P}_V(o, \#_o) \leftarrow p$ ;
13     $\eta(o, \#_o) \leftarrow \langle n, j \rangle$ ;
14     if $j + 1 \leq \max_j$ then
15         $p \leftarrow \mathbf{P}_V(n, j + 1)$ ;
16         Insert $\langle p, n, j + 1, \max_j \rangle$ in $Q$ ordered by $p$ using $>$ ;
17     end
18 end
19 $\#\mathrm{Sel}(o) = \#_o$ ;
20 return $\langle \mathbf{P}_V, \#Sel, \eta \rangle$ ;
```

next best answer for the child we need only look at the next selection for the child. Thus, we record in Q the probability of this list's best option as well as its index so we can easily compute the next best index. The probability of the best selection of each child is always the first one, $\mathbf{P}_V(n, 1)$, and this is looked-up on line 5. We record 4 elements in an entry in Q on line 6: $\langle p, n, j, \max_j \rangle$. p is the probability of the entry, n is the child node of o, j is the selection number of node n that has probability p, and $\max_j$ is the number of selections that n has. Only p, n, and j are necessary, as $\max_j$ can be looked-up based on n, but it is convenient to include $\max_j$. Line 8 sets the number of selections gathered to 0.

Lines 9-18 grab the next best selection from among the current selections of each list from the Q and records it. If there is another selection after this current selection, it is re-inserted in Q. Line 10 gets our best element from the queue. Lines 11-13 record this next best entry into o's variables and increments the number of selections found. Lines 14-17 check to see if there is another selection for node n at position $j+1$. If so, these lines get the probability of this next selection and inserts an entry for this next selection in Q. Line 19 sets the number answers we found in this node's local variable and then the algorithm returns on line 20.

Runtime Analysis The FKBSelOr algorithm has two loops, one that generates an initial set of candidates among its $|E_o|$ children. The other extracts up to k selections. We assume out-going edges are stored with the node, so lines 1 and 2 are $O(1)$ operations. The loop from lines 3-7 performs $|E_o|$ iterations. Each iteration, we perform two $O(1)$ operations and then insert a fixed-size entry in Q. The enqueue operation, assuming a heap implementation, requires $O(\log |Q|)$ time to insert. We insert $|E_o|$ items for a total complexity of $O(|E_o| \log |E_o|)$ time and $O(|E_o|)$ space. The loop from lines 9-18 perform $O(k)$ iterations. For each iteration, we dequeue one element of Q on line 10. We then perform $O(1)$ operations between lines 11-13, setting some constant-size data. Finally, we sometimes add one element back into Q on line 16. If all of o's children have k selections, then we will insert a new element in Q either k or $k-1$ times. The enqueue and dequeue operations both have an $O(\log |E_o|)$ time complexity, as the queue will have at most $|E_o|$ elements in it at all times. Thus, this loop has complexity $O(k \log |E_o|)$. Combined with the first loop, the overall complexity of this algorithm is $O(|E_o| \log |E_o| + k \log |E_o|)$ time and $O(|E_o| + k)$ space.

3.2.10 Overall Find-K-Best-Selections Complexity

Given that we've now determined the complexity of all three sub-routines of Find-KBestSelections, we're now able to compute the complexity of FindKBestSelections. This entire algorithm consists of looping over all the nodes once, calling the appropriate sub-routine based on the type of node. Thus, the complexity is just the number of each type of node times the complexity of that type of node. Specifically,

$$O(|L| + |A||E_a|k \log k + |O||E_o| \log |E_o| + |O|k \log |E_o|)$$

time, where $|E_a|$ and $|E_o|$ are the number of children of an average **And** and **Or** node, respectively. If one assumes that there are approximately an equal number of each type of node and about the same number of children on average, this simplifies to

$$O(|V||E_v|k \log k + |V||E_v| \log |E_v| + |V|k \log |E_v|).$$

We can substitute $|E|$ for $|V||E_v|$ as the latter just represents the number of edges in the graph, giving us

$$O(|E|k \log k + |E| \log |E_v| + |V|k \log |E_v|).$$

The space required by this algorithm is dominated by two structures, C_a and ξ. The former requires $O(k \log k)$ space. The latter requires $O(|E|k)$ space. So the FindKBestSelections algorithm requires $O(k \log k + |E|k)$ space. Note that η requires $O(|V|k)$ space, but this is never more than $O(|E|k)$ space. Likewise, β_ξ requires $O(|E_v|k)$ space, but $|E_v| < |E|$, so we can ignore this term.

3.3 Get-K-Solutions-From-Selections Algorithm

Algorithm 3.11: GetKSolutionsFromSelections(V_O, E, $\mathcal{L}_L$, η, ξ, $\#_r$)

```
m(∗,∗) ←⊥ ;
for $i = 1$ to $\#_r$ do
    $m(r,i) = i$ ;
    $\mathcal{S}_k[i] = \emptyset$ ;
end
for $i = 1$ to $|V|$ do
    $v \leftarrow V_O[i]$ ;
    switch $v$ in
        case $v \in L$
            $\mathcal{S}_k \leftarrow$ GKSFSLeaf($v$, $m$, $\#_r$, $\mathcal{S}_k$, $\mathcal{L}_L$) ;
        end
        case $v \in A$
            $m \leftarrow$ GKSFSAnd($v$, $m$, $\#_r$, $\xi$) ;
        end
        case $v \in O$
            $m \leftarrow$ GKSFSOr($v$, $m$, $\#_r$, $\eta$) ;
        end
    end
end
return $\mathcal{S}_k$ ;
```

The premise of this algorithm is that we've described a modified graph with ξ and η built upon the sd-DNNF nodes. This modified graph has up to k root nodes, each of which is a tree that defines a selection. In our modified graph, a node is a pair: $\langle n, i\rangle$. The node n is an sd-DNNF node, and i corresponds to the i^{th} best selection for n. Thus, if $k = 3$ and the root has three selections, the best three selections start at $\langle r, 1\rangle$, $\langle r, 2\rangle$, and $\langle r, 3\rangle$.

We use the marking system described at the start of the chapter, namely each node has a list of k markings $m(v, i)$, $i \in \{1, \ldots, k\}$. The i^{th} marking records which of v's local modified nodes, or local selections, belong to the selection that starts at

$\langle r, i \rangle$. If there isn't such a local selection, the marking is set to $\perp$.

Otherwise, the marking rules are the same as Chapter 2 in the modified graph. Namely, the leaves of the tree rooted by the i^{th} root node include their symbols in the i^{th} solution. The modified **And** nodes mark all of their modified children. The modified **Or** nodes mark their only modified child. The implementation of this algorithm is shown in Alg. 3.11.

Alg. 3.11 starts out on line 1 clearing all the marks by setting them to $\perp$. There are $O(|V|k)$ markings, of which we must clear $O(|V|\#_r)$ markings. Lines 2-5 sets the root markings and clears our solutions. A modified root node $\langle r, i \rangle$ is marked as part of the i^{th} solution. Lines 6-19 loop over all the sd-DNNF nodes, from the root to the leaves. For each node, we call the appropriate GKSFS function. These functions move the markings down the modified graph and set the solutions. The algorithm returns the set of solutions $\mathcal{S}_k$ on line 20, once every modified leaf has had the opportunity to add itself to the appropriate solutions.

3.3.1 Get-K-Solutions-From-Selections Leaf-case Algorithm

Algorithm 3.12: GKSFSLeaf(l, m, $\#_r$, $\mathcal{S}_k$, $\mathcal{L}_L$)

```
for $i = 1$ to $\#_r$ do
    if $m(l, i) \neq \perp$ then
        $\mathcal{S}_k[i] \leftarrow \mathcal{S}_k[i] \cup \{\mathcal{L}_L(v)\}$ ;
    end
end
return $\mathcal{S}_k$ ;
```

The leaf case, like all cases, must process all the possible root markings to see if any of them include this leaf. Since the leaf only has one modified node, $\langle l, 1 \rangle$, $m(l, i)$ will always be 1 or $\perp$. When $m(l, i) = 1$, the root node includes this leaf, so we add this leaf's symbol to the appropriate solution.

Time and Space Complexity The algorithm will always loop $\#_r$ times, where $\#_r$ is the actual number of selections found, between 1 and k. In general, this will be k. We again assume appending symbols is an $O(1)$ operation, so for each loop, this algorithm does an $O(1)$ operation. Thus, the time complexity of this algorithm is $O(\#_r)$. The algorithm requires only $O(1)$ local space.

3.3.2 Get-K-Solutions-From-Selections And-case Algorithm

Algorithm 3.13: GKSFSAnd(a, m, $\#_r$, ξ)

```
for $i = 1$ to $\#_r$ do
    if $m\,(a, i) \neq \perp$ then
        $j_a \leftarrow m\,(a, i)$ ;
        foreach $\langle a, n \rangle \in E$ do
            $j_n \leftarrow \xi\,(a, j_a, n)$ ;
            $m\,(n, i) \leftarrow j_n$ ;
        end
    end
end
return $\mathcal{S}_k$ ;
```

The **And** node case is pushing each root marking that includes one of this **And** node's modified nodes to the appropriate combination of modified child nodes. If $m\,(a, i)$ is not $\perp$ on line 2, then it specifies which modified node of a is marked for the i^{th} solution, specifically $\langle a, j_a \rangle$. Given that $\langle a, j_a \rangle$ is part of the i^{th} solution, we mark each modified member of the j_a combination of a, specified by ξ. For a child n of a, j_n on line 5 is the index of the modified child node $\langle n, j_n \rangle$. ξ captures the relation that $\langle a, j_a \rangle$ is connected to $\langle n, j_n \rangle$ in our modified graph. We thus mark $\langle n, j_n \rangle$ as also being part of the solution i by setting $m\,(n, i) = j_n$.

Time and Space Complexity All of the operations of this algorithm are $O(1)$ within the double loop. Thus, the time complexity of the double-loop is $O(|E_a|\#_r)$, where $|E_a|$ is the number of out-going edges of a and the number of iterations of the inner loop. The algorithm requires only $O(1)$ space locally.

3.3.3 Get-K-Solutions-From-Selections Or-case Algorithm

Algorithm 3.14: GKSFSOr(o, m, $\#_r$, η)

```
1 for $i = 1$ to $\#_r$ do
2     if $m(o, i) \neq \perp$ then
3         $j_o \leftarrow m(o, i)$ ;
4         $\langle n, j_o \rangle \leftarrow \eta(o, j_o)$ ;
5         $m(n, i) \leftarrow j_n$ ;
6     end
7 end
8 return $\mathcal{S}_k$ ;
```

The **Or** node case is also pushing each root marking that includes one of this **Or** node's modified nodes to the appropriate modified child node. If $m(o, i)$ is not $\perp$ on line 2, then it specifies which modified node of o is marked for the solution i, specifically $\langle o, j_o \rangle$. The modified node $\langle o, j_o \rangle$ is connected to exactly one modified child node, namely $\langle n, j_n \rangle = \eta(o, j_o)$. So line 5 marks this modified child by setting $m(n, i) = j_n$.

Time and Space Complexity This algorithm has only one loop and everything else is looked up by index, an $O(1)$ operation, so the overall complexity is $O(\#_r)$ time. The algorithm only requires $O(1)$ local space.

3.3.4 Overall Get-K-Solutions-From-Selections Complexity

We can now analyze the complexity of the whole GetKSolutionsFromSelections algorithm. The algorithm starts by clearing $O(|V|\#_r)$ markings on line 1, requiring $O(|V|\#_r)$ time and space. Lines 2-5 set and additional $O(\#_r)$ terms, each of which is of size $O(1)$. Finally, lines 6-19 loop over all the nodes once from the root to the leaves. If $|S|$ is the size of an average solution, this loop will generate $\#_r$ solutions of size $|S|$. The time complexity of the loop is

$$O(|L|\#_r + |A||E_a|\#_r + |O|\#_r).$$

The term $|A||E_a|$ represents the total number of out-going edges that have **And** node parents. If this is $O(|E|)$, then we can simplify our time complexity to $O(|E|\#_r)$. The space complexity is dominated by the space required to store the $O(|V|\#_r)$ markings.

As was the case with the FindBestSolutionFromSelection algorithm, this can be re-framed as a depth-first search, where our first step is to iterate over the $\#_r$ root nodes and then keep track of which modified **And** node child we're visiting along the path. This change would reduce the complexity of this part to $O(|\text{Sel}|\#_r)$, where $|\text{Sel}|$ is the number of nodes in a selection. The amount of space can be reduced substantially to $O(|a_{\text{Sel}}|)$, where $|a_{\text{Sel}}|$ is the largest number of **And** nodes along any path from the root to a leaf. This is the same space required to store the $k = 1$ case, as we can perform our depth-first search $\#_r$ times with the same stack.

3.4 Find-K-Best-Solutions Complexity

The overall complexity of the FindKBestSolutions algorithm is dominated by the first part of the algorithm. This chapter's algorithm has a time complexity of $O(|E|k \log k + |E| \log |E_v| + |V|k \log |E_v|)$ and a space complexity of $O(|E|k)$.

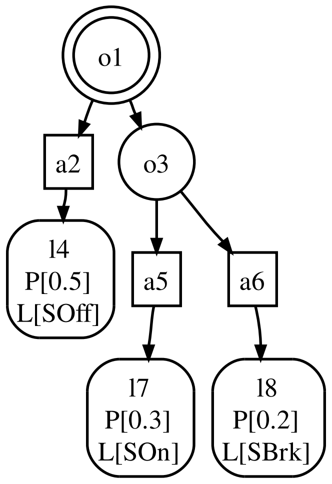


Figure 3-5: Our simple sd-DNNF example, identical to previous sections except with shorter labels $\mathcal{L}_L$.

3.5 Find-K-Best-Solutions Example

In this section we demonstrate the sub-routines of Alg. 3.1 FindKBestSolutions: how they generate the modified graph, and then how they read out the k best solutions. We present the algorithms twice, first on the simple switch example from the previous chapter, Fig. 3-5, and then on a more complicated A-B example, Fig. 3-12.

3.5.1 Switch Example

The switch example exercises the sub-routines of Alg. 3.1 with the exception of Alg. 3.4, 3.5, 3.8, and 3.9, all of which are related to the MergePair sub-routine, Alg. 3.8. MergePair is never run for the simple example because all of the **And** nodes have only one child.

Recall that Alg. 3.1 is first generating a modified graph, described by η and ξ, and then extracting up to k solutions. The modified graph has up to k replicas of each internal node, A and O. We denote each of these modified nodes $\langle v, i\rangle$. Fig. 3-6

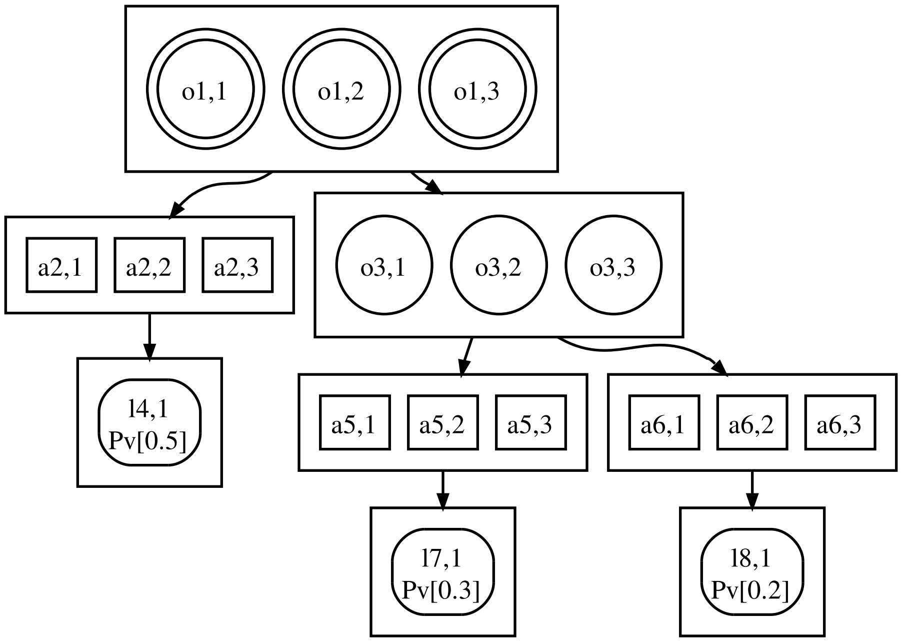


Figure 3-6: These are the modified nodes of the sd-DNNF for $k = 3$; there are 3 copies of all of the internal nodes. The modified nodes in this figure do not yet have any edges, which is the form of the modified graph when first starting the FindKBestSolutions algorithm, Alg. 3.1

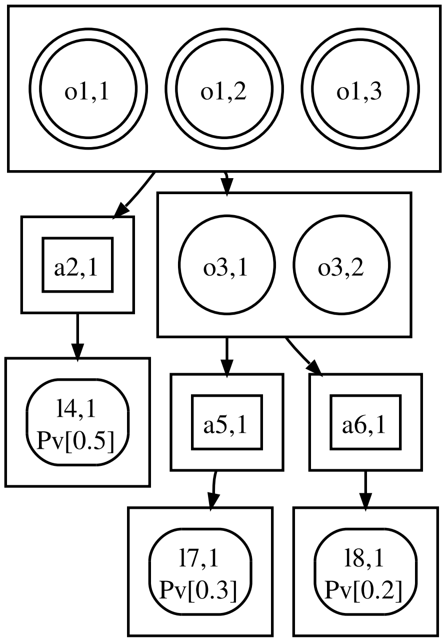


Figure 3-7: This figure is the same as Fig. 3-6 except that nodes in Fig. 3-6 that will never have edges, or be part of a selection, have been omitted. These nodes are unnecessary and need not be allocated.

shows the modified nodes of Fig. 3-5 explicitly for $k = 3$. Initially, when Alg. 3.1 begins, all of these modified nodes exist but have no edges. The objective of Alg. 3.2 is to add the appropriate edges in ξ and η, to define the solutions we want.

For the switch problem, when $k = 3$, some modified nodes will never have edges because there are less than 3 selections rooted at the unmodified node. For example, a2 has only one selection, which includes itself and l4. We thus omit these nodes from the figure to save space. We omit $\langle \text{a2}, 2\rangle$, $\langle \text{a2}, 3\rangle$, $\langle \text{o3}, 3\rangle$, $\langle \text{a5}, 2\rangle$, $\langle \text{a5}, 3\rangle$, $\langle \text{a6}, 2\rangle$, and $\langle \text{a6}, 3\rangle$ from Fig. 3-6 in Fig. 3-7.

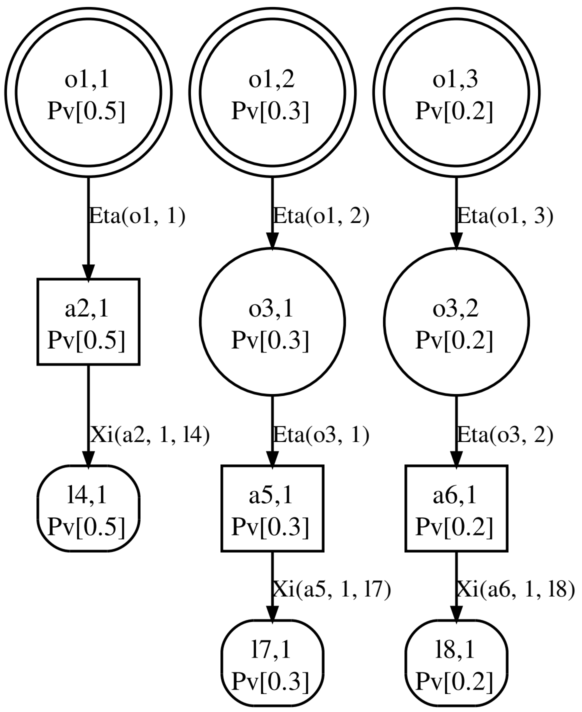


Figure 3-8: For the simple switch example and $k = 3$, this figure shows the modified graph with the edges added by FindKBestSelections, Alg. 3.2.

Find-K-Best-Solutions Switch Example

Alg. 3.1 is broken down into the same two steps as Chapter 2, namely a step that finds the k best selections, Alg. 3.2, and a step that extracts the k solutions of the k selections, Alg. 3.11. We now show how the first step, Alg. 3.2, generates the modified graph defined by ξ and η. This algorithm adds the appropriate edges to Fig. 3-7 to arrive at Fig. 3-8 for $k = 3$. Fig. 3-8 has three selections with probabilities 0.5, 0.3, and 0.2. We will then describe Alg. 3.11, which will find that the top three solutions are {“SOff”}, {“SOn”}, and {“SBrk”}, in that order.

Find-K-Best-Selections Switch Example

Alg. 3.2 loops over all the sd-DNNF nodes from the leaves to the root, adding edges to ξ and η. For the switch example, the nodes are processed in the order: l8, l7, a6, a5, l4, o3, a2, and then o1. The first iteration of Alg. 3.2 will invoke Alg. 3.3, FKBSelLeaf, on the leaf l8, on line 5.

In Alg. 3.3, $\#\mathrm{Sel}\left(\mathrm{l8}\right)$ is set to 1 and $\mathbf{P}_V\left(\mathrm{l8},1\right)$ is set to $\boldsymbol{\mathcal{L}}_{\mathbf{P}}\left(\mathrm{l8}\right)=0.2$. The next node processed by Alg. 3.2 is another leaf, l7, and Alg. 3.3 sets $\#\mathrm{Sel}\left(\mathrm{l7}\right)=1$ and $\mathbf{P}_V\left(\mathrm{l7},1\right)=0.3$.

Alg. 3.2 continues and visits a6. This **And** node is the first internal node we have examined and Alg. 3.2 invokes Alg. 3.6 FKBSelAnd on this node, on line 8. Alg. 3.6 sets $\#\mathrm{Sel}\left(\mathrm{a6}\right)=1$ and $\mathbf{P}_V\left(\mathrm{a6},1\right)=0.2$. It also adds our first edge $\xi\left(\mathrm{a6},1,\mathrm{l8}\right)=1$, which is to say $\langle\mathrm{a6},1\rangle\rightarrow\langle\mathrm{l8},1\rangle$. We will go into the details of how Alg. 3.6 computes these values after we have finished with Alg. 3.2.

The next node visited is a5, and this sets $\#\mathrm{Sel}\left(\mathrm{a5}\right)=1$, $\mathbf{P}_V\left(\mathrm{a5},1\right)=0.3$, and $\xi\left(\mathrm{a5},1,\mathrm{l7}\right)=1$, or equivalently $\langle\mathrm{a5},1\rangle\rightarrow\langle\mathrm{l7},1\rangle$. Visiting the node l4 sets $\#\mathrm{Sel}\left(\mathrm{l4}\right)=1$ and $\mathbf{P}_V\left(\mathrm{l4},1\right)=0.5$.

Alg. 3.2 then visits the **Or** node o3. Alg. 3.2 invokes Alg. 3.10, FKBSelOr, on o3, on line 11. This sub-routine sets several values. It sets $\#\mathrm{Sel}\left(\mathrm{o3}\right)=2$, indicating there are two selections. For the first selection, it sets $\mathbf{P}_V\left(\mathrm{o3},1\right)=0.3$ and $\eta\left(\mathrm{o3},1\right)=\langle\mathrm{a5},1\rangle$, or equivalently $\langle\mathrm{o3},1\rangle\rightarrow\langle\mathrm{a5},1\rangle$. For the second selection, the sub-routine sets $\mathbf{P}_V\left(\mathrm{o3},2\right)=0.2$ and $\eta\left(\mathrm{o3},2\right)=\langle\mathrm{a6},1\rangle$, or equivalently $\langle\mathrm{o3},2\rangle\rightarrow\langle\mathrm{a6},1\rangle$. We will go into the details of how Alg. 3.10 computes these values after we have finished with Alg. 3.2.

Alg. 3.2 then moves on to the **And** node a2 and sets $\#\mathrm{Sel}\left(\mathrm{a2}\right)=1$, $\mathbf{P}_V\left(\mathrm{a2},1\right)=0.5$, and $\xi\left(\mathrm{a2},1,\mathrm{l4}\right)=1$, or equivalently $\langle\mathrm{a2},1\rangle\rightarrow\langle\mathrm{l4},1\rangle$. The final node visited by Alg. 3.2 is the root o1. Alg. 3.2 invokes Alg. 3.10 on this **Or** node. Alg. 3.10 sets

$\#\mathrm{Sel}\,(\mathrm{o1}) = 3$, indicating there are 3 selections. For the first selection, the sub-routine sets $\mathbf{P}_V\,(\mathrm{o1}, 1) = 0.5$ and $\eta\,(\mathrm{o1}, 1) = \langle \mathrm{a2}, 1\rangle$. For the second selection, the sub-routine sets $\mathbf{P}_V\,(\mathrm{o1}, 2) = 0.3$ and $\eta\,(\mathrm{o1}, 2) = \langle \mathrm{a5}, 1\rangle$. For the third and last selection, the sub-routine sets $\mathbf{P}_V\,(\mathrm{o1}, 3) = 0.2$ and $\eta\,(\mathrm{o1}, 3) = \langle \mathrm{a6}, 1\rangle$.

This concludes Alg. 3.2. The algorithm generated 3 selections starting at $\langle \mathrm{o1}, 1\rangle$, $\langle \mathrm{o1}, 2\rangle$, and $\langle \mathrm{o1}, 3\rangle$, respectively. The probabilities of these three selections are 0.5, 0.3, and 0.2, respectively. The five edges added to the modified graph with η and the three edges added to the modified graph with ξ are shown in Fig. 3-8.

Find-K-Best-Solutions, And Case, Switch Example

In this section, we show the FKBSelAnd algorithm, Alg. 3.6, running on node a6 of the simple switch example. In this example, only the sub-routine Alg. 3.7 InheritFirstChild is called. FKBSelAnd sets $\#\mathrm{Sel}\,(\mathrm{a6}) = 1$, $\mathbf{P}_V\,(\mathrm{a6}, 1) = 0.2$, and $\xi\,(\mathrm{a6}, 1, \mathrm{l8}) = 1$.

Alg. 3.6 starts by getting its only child l8 and putting it in e on line 1. The algorithm then calls Alg. 3.7 on line 2, which sets the local variables $\#_a = 1$, $\mathbf{P}_a\,(1) = 0.2$, and $\beta_\xi\,(\mathrm{l8}, 1) = \langle \perp, 1, 1\rangle$. Since the algorithm has no other children, lines 3-5 are skipped.

Since $\#_a = 1$, lines 6-14 are only run once. The algorithm sets $\mathbf{P}_V\,(a, 1) = 0.2$ on line 7. Line 8 sets $n = \mathrm{l8}$ and $j_1 = 1$. Within the inner loop, line 10 gets $\beta_\xi\,(\mathrm{l8}, 1) = \langle \perp, 1, 1\rangle$. Line 11 sets $\xi\,(\mathrm{a6}, 1, \mathrm{l8}) = 1$ and line 12 sets $n = \perp$ and $j_1 = 1$, ending the loop. Line 15 sets $\#\mathrm{Sel}\,(\mathrm{a6}) = 1$.

Inherit-First-Child Switch Example

Continuing our simple switch example from within the FKBSelAnd algorithm, Alg. 3.6, this function sets $\#_a = 1$, $\mathbf{P}_a\,(1) = 0.2$, and $\beta_\xi\,(\mathrm{l8}, 1) = \langle \perp, 1, 1\rangle$ when invoked on the child l8 of the node a6.

On line 1 of the algorithm looks up the number of selections of l8. The node l8 has only 1 selection, the modified node $\langle \text{l8}, 1\rangle$. For this one selection, line 3 copies $\mathbf{P}_V(\text{l8}, 1) = 0.2$ into $\mathbf{P}_a(1)$. Line 4 sets $\beta_\xi(\text{l8}, 1) = \langle \bot, 1, 1\rangle$, which records both that the modified node $\langle \text{a6}, 1\rangle$ gets its probability from the modified node $\langle \text{l8}, 1\rangle$, and thus it has the edge $\langle \text{a6}, 1\rangle \rightarrow \langle \text{l8}, 1\rangle$, and it also records that l8 is the last variable, as its successor is $\bot$. This sub-routine then returns, concluding the FKBSelAnd sub-routines.

Find-K-Best-Selections, Or Case, Switch Example

We demonstrate Alg. 3.10, in this section, running on node o1 of the switch example. In this sub-routine, three selections are identified with the three modified nodes $\langle \text{o1}, 1\rangle$, $\langle \text{o1}, 2\rangle$, and $\langle \text{o1}, 3\rangle$. These modified nodes are assigned probabilities in $\mathbf{P}_V$ and children in η.

The node o1 has three children, a2, a5, and a6. It thus sets $\#_E = 3$ on line 2. For each of these three children, the algorithm enqueues an entry of the form $\langle p, n, j, \max_j\rangle$. The children of o1 have the entries:

- a2: $\langle 0.5, \text{a2}, 1, 1\rangle$
- a5: $\langle 0.3, \text{a5}, 1, 1\rangle$
- a6: $\langle 0.2, \text{a6}, 1, 1\rangle$

These are all inserted into Q on line 6. Thus the queue has these three entries on line 8. Line 8 sets our current number of selections, $\#_o$, equal to 0.

The first iteration of lines 9-18 starts out on line 10 by removing the best entry, $\langle 0.5, \text{a2}, 1, 1\rangle$ from Q. After this, Q contains only two entries: $\langle 0.3, \text{a5}, 1, 1\rangle$ and $\langle 0.2, \text{a6}, 1, 1\rangle$. Line 11 sets the number of selections to 1. Line 12 sets $\mathbf{P}_V(\text{o1}, 1) = 0.5$.

Line 13 records the modified child of o1 that had this probability, namely $\langle \text{a2}, 1\rangle$. Thus, η now contains the edge $\langle \text{o1}, 1\rangle \rightarrow \langle \text{a2}, 1\rangle$.

Line 14 checks if $j + 1 \leq 1$, but, since $j = 1$, it does not. Thus, the loop from lines 9-18 starts again at the beginning. If $\max_j$ was 2, line 16 would have inserted $\langle p, \text{a2}, 2, 2\rangle$ into Q, where $p = \mathbf{P}_V (\text{a2}, 2)$.

The second iteration of the loop starts out by again removing the best entry from Q. In this case the best entry is $\langle 0.3, \text{a5}, 1, 1\rangle$. The loop sets $\#_o = 2$, $\mathbf{P}_V (\text{o1}, 2) = 0.3$, and $\eta (\text{o1}, 2) = \langle \text{a5}, 1\rangle$. Again, this loop skips lines 14-17. The third and final iteration of the loop starts out by removing the last entry from Q, the entry $\langle 0.2, \text{a6}, 1, 1\rangle$. This iteration sets $\#_o = 3$, which is also k, $\mathbf{P}_V (\text{o1}, 3) = 0.2$, and $\eta (\text{o1}, 3) = \langle \text{a6}, 1\rangle$. The loop then exits, setting the final number of selections of o1, #Sel (o3), to 3 on line 19.

This concludes the activities of Alg. 3.2, FindKBestSelections, operating on the simple switch example. This algorithm generates the modified graph with edges show in Fig. 3-8. For comparison, figures 3-9 and 3-10 show the modified graph generated by Alg. 3.2 on the same switch example, but with $k = 2$ and $k = 1$, respectively. When $k = 1$, Alg 3.2 is expected generate the same modified graph as Alg. 2.2, so it is important to observe that their modified graphs are in fact the same. The modified graph of Alg. 2.2 is shown in figures 2-4 and 2-5. The modified graph of Alg. 3.2 is shown in Fig. 3-10.

Get-K-Solutions-From-Selections Algorithm

Alg. 3.11, GetKSolutionsFromSelections, is responsible for extracting the 3 solutions corresponding to the 3 selections we found in the proceeding section by running Alg. 3.2, FindKBestSelections. We have boxed these three selections in Fig. 3-11 as well as reporting the probability of each selection. The three best solutions are, in order, {"SOff"}, {"SOn"}, and {"SBrk"}.

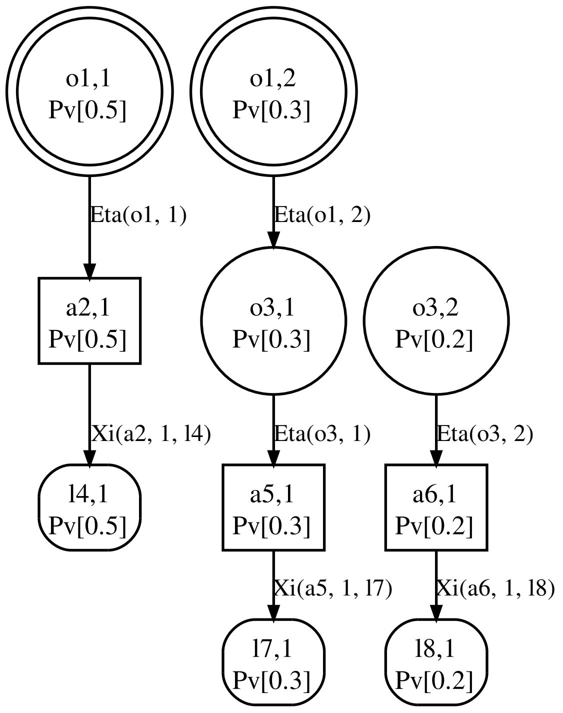


Figure 3-9: For the simple switch example and $k = 2$, this figure shows the modified graph with the edges added by FindKBestSelections, Alg. 3.2. The nodes and edges in the modified graph are a proper subset of the nodes and edges of Fig. 3-8.

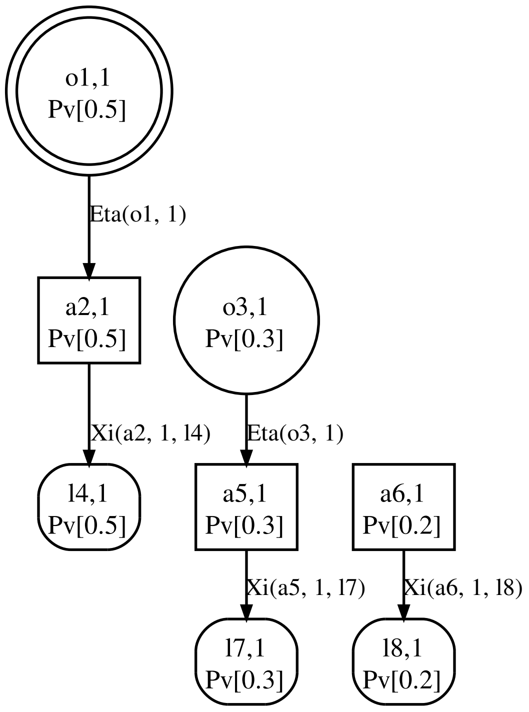


Figure 3-10: For the simple switch example and $k = 1$, this figure shows the modified graph with the edges added by FindKBestSelections, Alg. 3.2. The nodes and edges in the modified graph are a proper subset of the nodes and edges of Fig. 3-9. This figure contains the same edges as those created by the FindBestSelection algorithm, Alg. 2.2. The modified graph produced by FindBestSelection is shown in Fig. 2-4

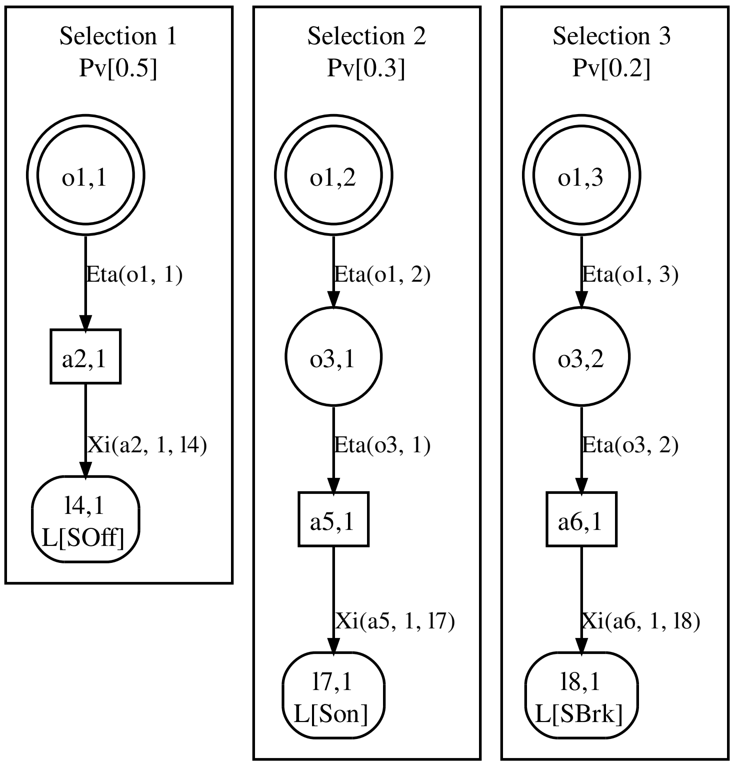


Figure 3-11: This figure boxes the 3 selections of the modified graph produced by FindKBestSelections, Alg. 3.2, along with their probability. For each selection, the GKSFS algorithm, Alg. 3.11, gathers the labels $\mathcal{L}_L$ of each leaf in the selection box.

Alg. 3.11 starts out on line 1 by clearing all of our markings. Lines 2-5 loop once for each of the root node o1's modified nodes: $\langle \mathrm{o1}, 1\rangle$, $\langle \mathrm{o1}, 2\rangle$, and $\langle \mathrm{o1}, 3\rangle$. For each modified node $\langle \mathrm{o1}, i\rangle$, the algorithm marks that modified node as part of the i^{th} selection, and thus $m\left(\mathrm{o1}, 1\right) = 1$, $m\left(\mathrm{o1}, 2\right) = 2$, and $m\left(\mathrm{o1}, 3\right) = 3$. The algorithm also clears $\mathcal{S}_k[i]$ for each i.

Alg. 3.11 then continues by iterating over all the sd-DNNF nodes from the root to the leaves between lines 6 and 19. The algorithm starts with the root node o1, and calls Alg. 3.14, GKSFSOr on o1. Alg. 3.14 sets $m\left(\mathrm{a2}, 1\right) = 1$, $m\left(\mathrm{o3}, 2\right) = 1$, and $m\left(\mathrm{o3}, 3\right) = 2$. These three values mean that $\langle \mathrm{a2}, 1\rangle$ is part of the first selection, $\langle \mathrm{o3}, 1\rangle$ is part of the second selection, and $\langle \mathrm{o3}, 2\rangle$ is part of the third selection, respectively.

For the next node, the **And** node a2, the algorithm calls Alg. 3.13, GKSFSAnd. Alg. 3.13 sets $m\left(\mathrm{l4}, 1\right) = 1$. Alg. 3.11 continues onto the next node o3, and Alg. 3.14, GKSFSOr, sets the markings $m\left(\mathrm{a5}, 2\right) = 1$, $m\left(\mathrm{a6}, 3\right) = 1$, meaning $\langle \mathrm{a5}, 1\rangle$ is part of the second selection and $\langle \mathrm{a6}, 1\rangle$ is part of the third selection.

Alg. 3.11 then visits node l4. For this node, the algorithm calls Alg. 3.12, GKSFSLeaf. This function observes that $m\left(\mathrm{l4}, 1\right) = 1$, and thus adds $\mathcal{L}_L\left(\mathrm{l4}\right) =$ "SOff" to the first solution, $\mathcal{S}_k[1]$.

Continuing with Alg. 3.11, the nodes a5 and a6 are visited, setting $m\left(\mathrm{l7}, 2\right) = 1$ and $m\left(\mathrm{l8}, 3\right) = 1$, respectively. Visiting the nodes l7 and l8 add "SOn" to $\mathcal{S}_k[2]$ and "SBrk" to $\mathcal{S}_k[3]$, respectively. The algorithm is then done and returns the solutions:

- $\mathcal{S}_k[1] = \{$"SOff"$\}$
- $\mathcal{S}_k[2] = \{$"SOn"$\}$
- $\mathcal{S}_k[3] = \{$"SBrk"$\}$

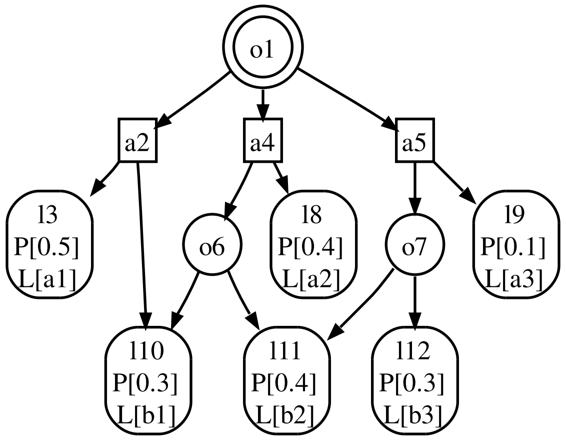


Figure 3-12: This figures shows a more complicated valued sd-DNNF than that of Fig 3-5. This graph has 5 solutions: {“a1”, “b1”}, {“a2”, “b1”}, {“a2”, “b2”}, {“a3”, “b2”}, and {“a3”, “b3”}.

3.5.2 A-B Example

For a more complicated example of this algorithm, we now present a contrived example with two types of labels, A and B. Each type of label has three values, for example a1, a2, and a3. The truth table for these two variables is:

	a1	a2	a3
b1	1	1	0
b2	0	1	1
b3	0	0	1

The sd-DNNF shown in Fig. 3-12 represents this truth table. There are six leaves, three of which are l3, l8, and l9 with labels “a1”, “a2”, and “a3” from A, respectively. The other three are l10, l11, and l12 with labels “b1”, “b2”, and “b3” from B, respectively. The complete list of nodes is: o1, a2, l3, a4, a5, o6, o7, l8, l9, l10, l11, and l12. The probabilities $\mathcal{L}_{\mathbf{P}}$ of these six leaves are $\mathcal{L}_{\mathbf{P}}(\text{l3}) = 0.5$, $\mathcal{L}_{\mathbf{P}}(\text{l8}) = 0.4$, $\mathcal{L}_{\mathbf{P}}(\text{l9}) = 0.1$, $\mathcal{L}_{\mathbf{P}}(\text{l10}) = 0.3$, $\mathcal{L}_{\mathbf{P}}(\text{l11}) = 0.4$, and $\mathcal{L}_{\mathbf{P}}(\text{l12}) = 0.3$. There are 13 edges

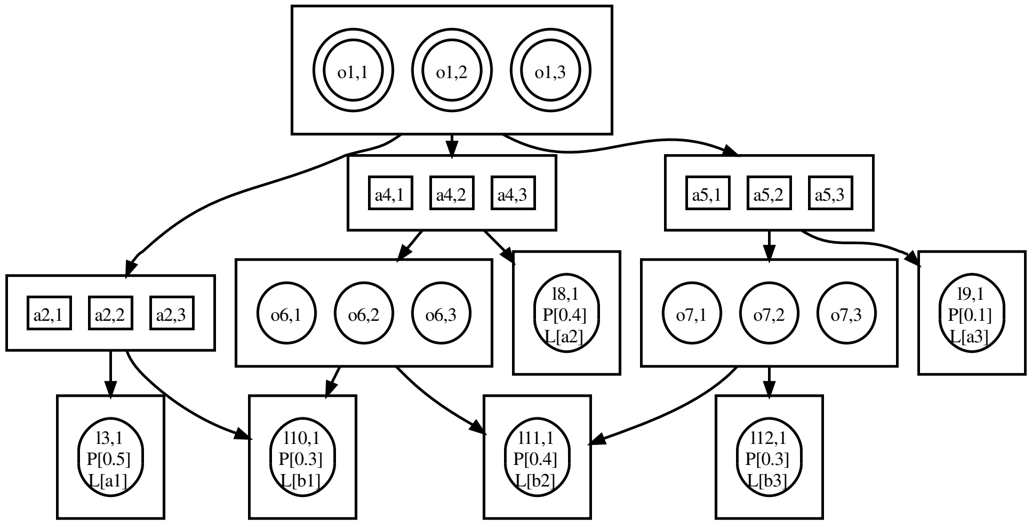


Figure 3-13: This figure shows the modified nodes of the A-B example when $k = 3$.

E, all of which are drawn in Fig. 3-12 and will thus be omitted from this textual description. We are again assuming a maximum-product.

Find-K-Best-Solutions A-B Example

We will again highlight running Alg. 3.1, FindKBestSolutions, on this new A-B example. We will focus primarily on the **And** node case of Alg. 3.2 FindKBestSelections, Alg. 3.6 FKBSelAnd, since in the previous example, Alg. 3.8 MergePair and its sub-routines were unnecessary.

As with the switch example, we have shown the modified nodes of Fig. 3-12 in Fig. 3-13 for $k = 3$ and then pruned them down to only those that will have edges in Fig. 3-14. Fig. 3-14 represents the starting point of Alg. 3.1.

For this example and $k = 3$, Alg. 3.1, FindKBestSolutions, calls Alg. 3.2, Find-KBestSelections, which generates the edges shown in Fig. 3-15. Due to space constraints, we have abbreviated ξ by omitting the last term n in $\xi(a, j, n)$, as the last

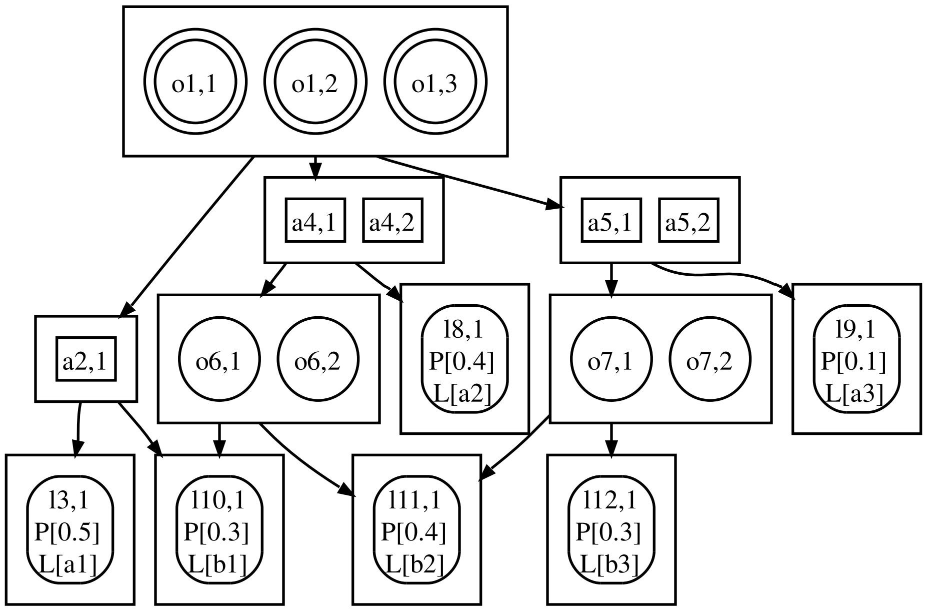


Figure 3-14: This figure shows just the modified nodes of the A-B example, when $k = 3$, that will have an edge after running FindKBestSolutions, Alg. 3.1.

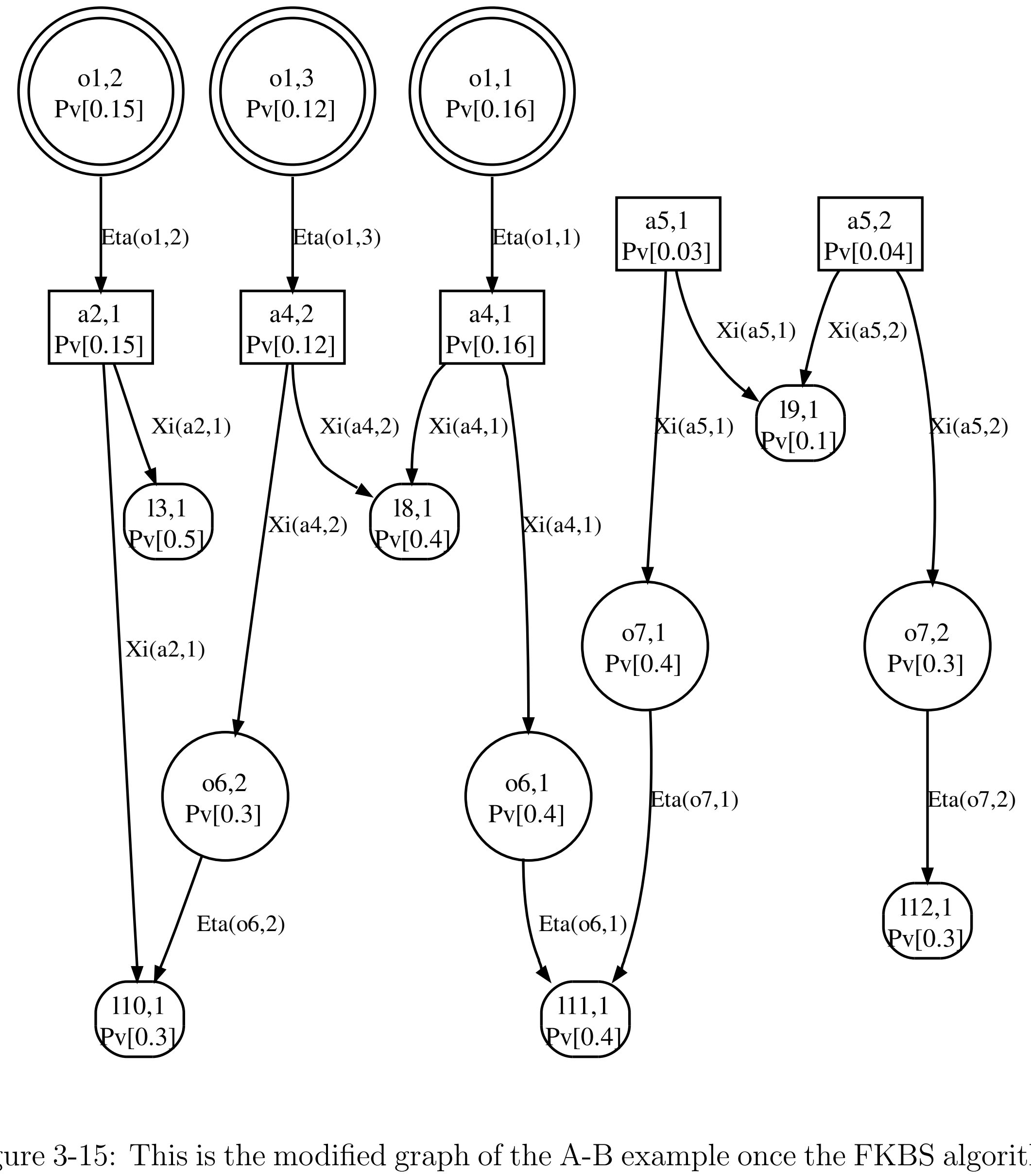


Figure 3-15: This is the modified graph of the A-B example once the FKBS algorithm, Alg. 3.2 has run.

term is the node to which the arc points. Alg. 3.1 then calls Alg. 3.11, GetKSolutionsFromSelections, which extracts the three solutions of the selections rooted at o1. The three solutions are, in order, {"a2", "b2"}, {"a1", "b1"}, and {"a2", "b1"}. The first two selections are highlighted in Fig. 3-19. The third selection overlaps with the first two, so it was not drawn.

Find-K-Best-Selections A-B Example

Alg. 3.2, FindKBestSelections, adds the edges shown in Fig. 3-15 to the A-B example, for $k = 3$. For the purpose of this example, we will focus on the edges added to o6 and a4.

When Alg. 3.2 visits o6, the algorithm calls the sub-routine FKBSelOr, Alg. 3.10. This sub-routine sets $\#\mathrm{Sel}\,(\mathrm{o6}) = 2$. For the first selection, Alg. 3.10 sets $\mathbf{P}_V\,(\mathrm{o6}, 1) = 0.4$ and $\eta\,(\mathrm{o6}, 1) = \langle \mathrm{l11}, 1\rangle$, where the modified child node $\langle \mathrm{l11}, 1\rangle$ has probability 0.4. For the second selection, Alg. 3.10 sets $\mathbf{P}_V\,(\mathrm{o6}, 2) = 0.3$ and $\eta\,(\mathrm{o6}, 2) = \langle \mathrm{l10}, 1\rangle$.

When Alg. 3.2 visits a4, the algorithm calls the sub-routine FKBSelAnd, Alg. 3.6. This sub-routine sets $\#\mathrm{Sel}\,(\mathrm{a4}) = 2$. For the first selection, Alg. 3.6 sets $\mathbf{P}_V\,(\mathrm{a4}, 1) = 0.16$, $\xi\,(\mathrm{a4}, 1, \mathrm{l8}) = 1$, and $\xi\,(\mathrm{a4}, 1, \mathrm{o6}) = 1$. For the second selection, Alg. 3.6 sets $\mathbf{P}_V\,(\mathrm{a4}, 2) = 0.12$, $\xi\,(\mathrm{a4}, 2, \mathrm{l8}) = 1$, and $\xi\,(\mathrm{a4}, 2, \mathrm{o6}) = 2$. Thus we have $\langle \mathrm{a4}, 1\rangle \rightarrow \langle \mathrm{o6}, 1\rangle$ and $\langle \mathrm{a4}, 2\rangle \rightarrow \langle \mathrm{o6}, 2\rangle$ along with $\langle \mathrm{a4}, 1\rangle \rightarrow \langle \mathrm{l8}, 1\rangle$ and $\langle \mathrm{a4}, 2\rangle \rightarrow \langle \mathrm{l8}, 1\rangle$.

We will now delve into the **And** case for this example.

Find-K-Best-Selections, And Case, A-B Example

Within Alg. 3.6, FKBSelAnd, we will assume that the first edge we choose was towards l8. Then line 2 initializes our three local **And** node variables to: $\#_a = 1$, $\mathbf{P}_a\,(1) = 0.4$, and $\beta_\xi\,(\mathrm{l8}, 1) = \langle \perp, 1, 1\rangle$. The function MergePair, Alg. 3.8, is then run

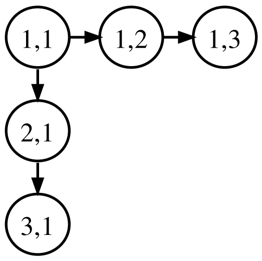


Figure 3-16: This figure shows the set of combinations that are part of the C_a structure when $k = 3$. A combination will be enabled if both of its parents are accepted as part of the k best selections. $1, 1$ is always enabled.

on the only pairing, between l8 and o6. This pairing will set $\beta_\xi\,(\text{o6}, 1) = \langle \text{l8}, 1, 1\rangle$ and $\beta_\xi\,(\text{o6}, 2) = \langle \text{l8}, 1, 2\rangle$. These three entries summarize two combinations, $\langle 1, 1\rangle$ and $\langle 1, 2\rangle$, with two edges each, thus describing the four edges presented just previously for ξ.

Merge-Pair A-B Example

MergePair, Alg. 3.8, requires that we have already constructed C_a for $k = 3$. C_a for $k = 3$ is shown in Fig. 3-16. The entries $\langle i, j_1, j_2, i_1, i_2, \#_P, \#_E\rangle$ of C_a are:

- $C_a[1] = \langle 1, 3, 1, \perp, \perp, 1, 1\rangle$
- $C_a[2] = \langle 2, 2, 1, 1, \perp, 1, 1\rangle$
- $C_a[3] = \langle 3, 1, 3, \perp, \perp, 1, 1\rangle$
- $C_a[4] = \langle 4, 1, 2, \perp, 3, 1, 1\rangle$
- $C_a[5] = \langle 5, 1, 1, 2, 4, 0, 0\rangle$

The first step of Alg. 3.8 sets the number of remaining parents $\#_E$ equal to the number of actual parents $\#_P$ for each entry by calling ResetCombinations Alg. 3.5.

The entries listed above already have the two values equal. MergePair then sets our new number of solutions $\#'_a = 0$ and computes the probability of the first combination $\langle 1, 1\rangle$. The probability of the first combination is $0.4 \times 0.4 = 0.16$. Line 5 inserts the entry $\langle 0.16, 5\rangle$ into Q, where 5 is the index of the $\langle 1, 1\rangle$ combination in C_a.

Alg. 3.8 then loops over lines 6-14. In the first iteration, the only element in Q is removed, the entry $\langle 0.16, 5\rangle$. The loop records that $\#'_a = 1$, sets $\mathbf{P}'_a(1) = 0.16$, and sets $\beta_\xi(\text{o6}, 1) = \langle \text{l8}, 1, 1\rangle$.

Line 12 then calls InsSucc, Alg. 3.9, for the combination $\langle 2, 1\rangle$, but this isn't a valid combination as l8 does not have 2 selections, so InsSucc does nothing. Line 13 then calls InsSucc for the combination $\langle 1, 2\rangle$ and this both exists and is now enabled, so InsSucc computes the probability of this combination $0.4 \times 0.3 = 0.12$ and inserts $\langle 0.12, 4\rangle$ into Q.

In the second iteration of Alg. 3.8, the entry $\langle 0.12, 4\rangle$ is dequeued from Q. The iteration records that $\#'_a = 2$, sets $\mathbf{P}'_a(2) = 0.12$, and sets $\beta_\xi(\text{o6}, 2) = \langle \text{l8}, 1, 2\rangle$. Alg. 3.8 then calls InsSucc on $i_1 = \perp$ in $C_a[4]$, so InsSucc immediately returns. Alg. 3.8 then calls InsSucc on $i_2 = 3$, which has the combination $\langle 1, 3\rangle$. Since $3 > \#\text{Sel}(\text{o6})$, InsSucc also immediately returns. The queue Q is then empty, with only 2 selections, and the algorithm returns with just these two selections.

Figures 3-17 and 3-17 illustrate the modified graph of the A-B example with $k = 2$, and $k = 1$, respectively. These figures show how the modified nodes in Fig. 3-15 are eliminated as the number of selections we seek is reduced.

Get-K-Solutions-From-Selections

For the A-B example, when $k = 3$, the GetKSolutionsFromSelections algorithm, Alg. 3.11, is performing much the same steps as in the switch example. The one interesting variation to the switch example is that two nodes are part of more than one solution,

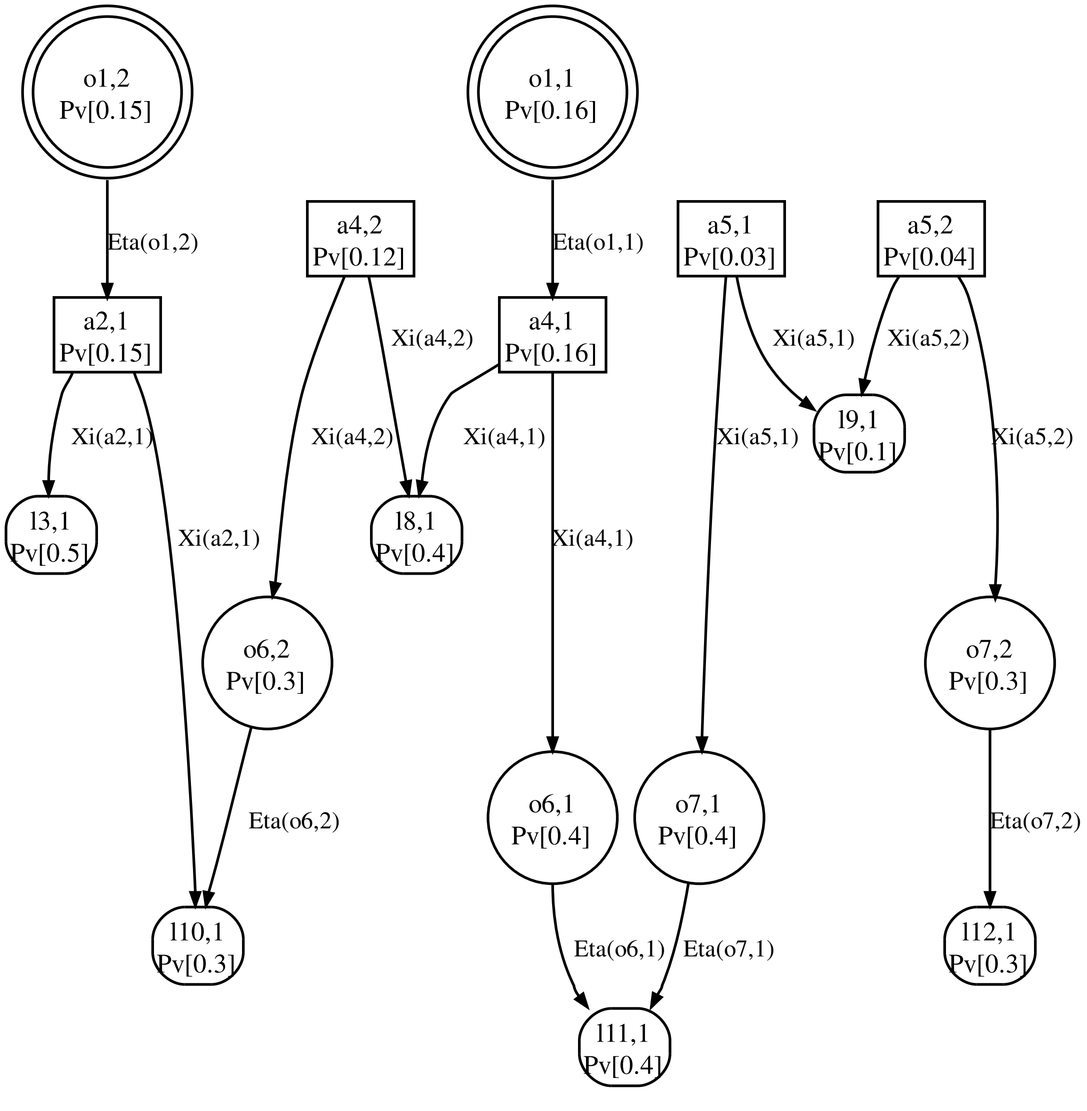


Figure 3-17: This figure shows how the modified nodes and edges change when $k = 2$ as opposed to $k = 3$.

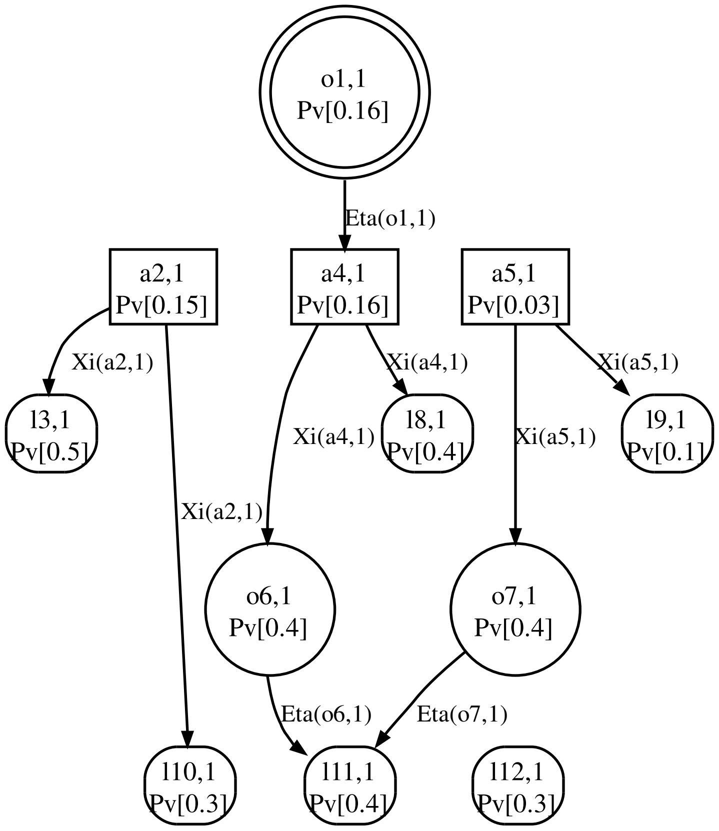


Figure 3-18: This figure shows how the modified nodes and edges change when $k = 1$ as opposed to $k = 2$.

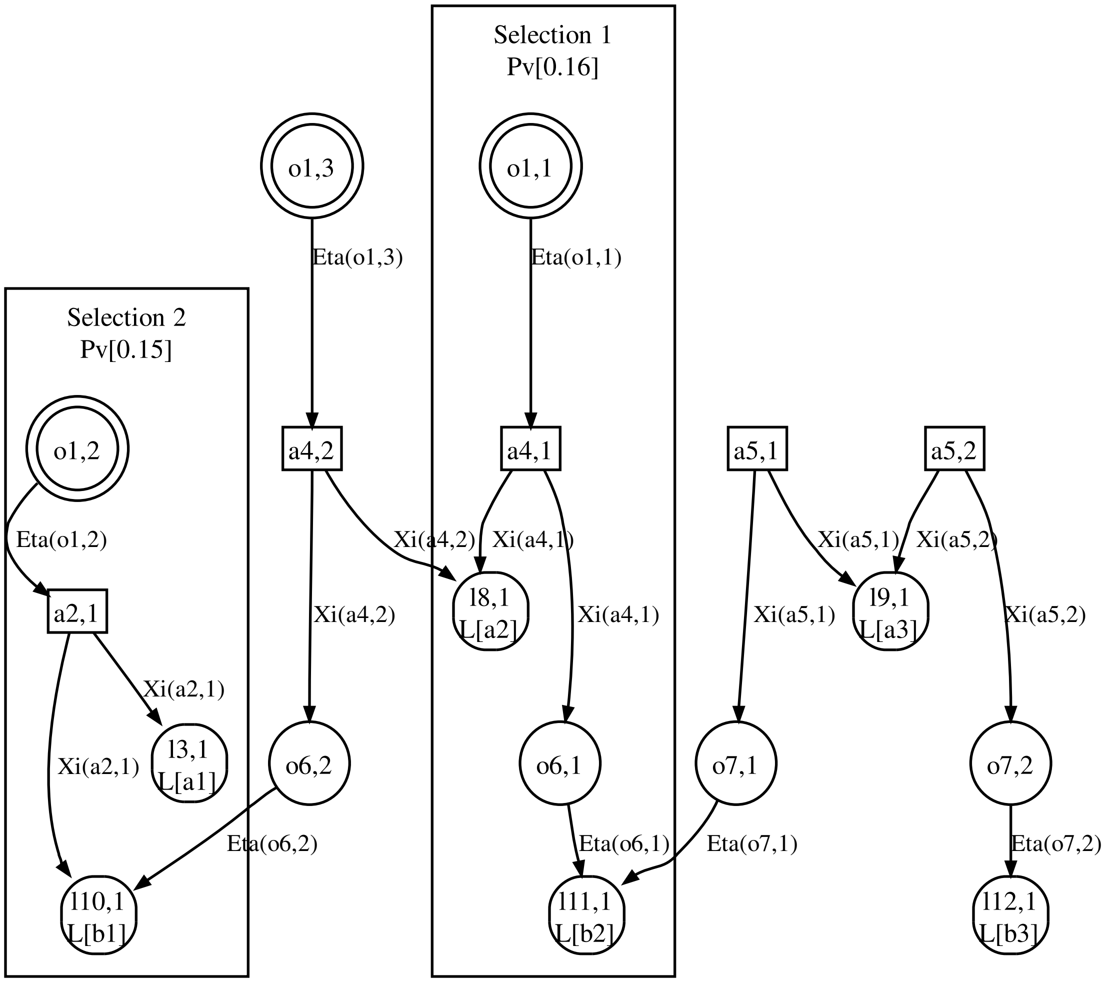


Figure 3-19: This figure shows the result of applying the GKSFS algorithm to the A-B example. We have highlighted the best 2 selections out of the 3 selections generated.

namely l8 and l10. For the node l8, for example, both $m\,(\text{l8}, 1) = 1$ and $m\,(\text{l8}, 3) = 1$, which means that l8 is part of the first and third solution. The label "a2" of l8 is thus added to $\mathcal{S}_k[1]$ and $\mathcal{S}_k[3]$. The three solutions returned are {"a2", "b2"}, {"a1", "b1"}, and {"a2", "b1"}.

3.6 Summary

In this chapter, we presented an extension of the find-best-solution algorithm of Chapter 2 that is able to find up to k solutions. The extension, Alg. 3.1, has a time complexity of $O(|E|k \log k + |E| \log |E_v| + |V|k \log |E_v|)$ and a space complexity of $O(|E|k)$. We also demonstrated this algorithm on two examples, first on the simple switch example of Chapter 2 and then on the A-B example.

Chapter 4

Results

We implemented the algorithm presented in Chapter 3 in C++. We used a visitor pattern for the switch statements in Algs. 3.2 and 3.11. The implementation was compiled with the g++ v3.4.4 package that comes with cygwin. We used the Windows built-in *QueryPerformanceCounter* function to obtain timing data, which reports the real-time elapsed. The results were gathered on a 1.7GHz Intel Pentium M computer with 1.5GB of RAM running Windows XP. All data points, unless otherwise noted, are the average of 200 runs of the algorithm, where, for each run, we vary the probability labels $\mathcal{L}_{\mathbf{P}}$ by choosing pseudo-random values with the C language built-in *rand* function.

The implementation was written for comprehensibility and correctness, not performance, in terms of both time and space. For example, standard template library vectors were used in several places to store edges and entries for ξ and η. The algorithm uses doubles, as opposed to floats, to store probabilities, and uses integer indices or pointers everywhere else.

Table 4.1 summarizes the seven graphs presented in this section. For example, G1 has 359 nodes and 863 edges, where 118 nodes are leaves, 169 are **And** nodes,

G	$\|E\|$	$\|V\|$	$\|L\|$	$\|A\|$	$\|O\|$
G1	863	359	118	169	72
G2	2,154	750	154	423	173
G3	2,559	809	190	434	185
G4	39,247	4,832	1,018	2,590	1,224
G5	6,992	1,549	794	504	251
G6	13,324	4,141	508	2,476	1,157
G7	308,084	80,754	866	55,115	24,773

G	|Solutions|	$\frac{\#\text{Symbols}}{\text{Solution}}$	$\frac{\#\text{Edges}}{\text{Node}}$		
			Avg	σ	Max
G1	6,800	21	3.58	2.87	14
G2	226,800	26	3.61	3.39	19
G3	1.5×10^8	34	4.13	3.80	16
G4	$> 10^{19}$	172	10.3	15.2	59
G5	1.7×10^9	62	9.26	12.7	78
G6	2.3×10^9	67	3.67	4.79	36
G7	$> 10^{19}$	87	3.86	8.37	103

Table 4.1: This table shows the attributes of the graphs used in this section.

and the remaining 72 nodes are **Or** nodes. A solution to G1 has 21 symbols; G1 has 6,800 solutions. The internal nodes have 3.58 children, on average, with a standard deviation of 2.87. No internal node has more than 14 children.

The graphs vary in size between 900 and 308,000 edges, with between 3.5 and 10 children per internal node, on average. These are the terms $|E|$ and $|E_v|$, respectively, in the time complexity of the Get-K-Best-Solutions algorithm:

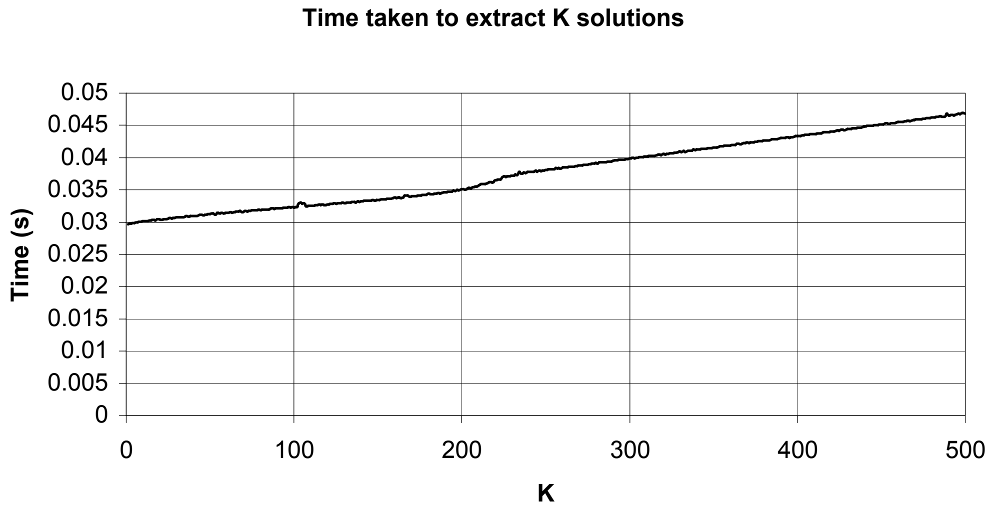
$$O(|E|k \log k + |E| \log |E_v| + |V|k \log |E_v|)$$

In the rest of this section, we empirically show how much time and memory it takes to extract k solutions from these seven graphs. We vary k between 1 and 10,000 for G1 and vary k between 1 and 500 for the remaining graphs except the last one, for which we only vary k between 1 and 100.

The performance data for the graph G1 is shown in Fig. 4-1. G1 is taken from a simple switched or-gate propagation example. Fig. 4-1 (top) shows the time it takes to extract a solution from G1. We believe the slight increase at around $k = 200$ is caused by a partial loss of locality in the algorithm, and is thus related to the processor cache size. The line itself is otherwise basically linear, implying the linear parts are more significant than the $k \log k$ part. The time complexity grows at a rate of 0.036 ms per k with a time of 28.919 ms for $k = 1$, using least-square fitting.

Fig. 4-1 (bottom) shows the memory used by the algorithm. The graph plots the amount of memory that all of the data structures are calculated to use and includes the space required to store the graph itself. As expected, this is linear in k. The memory required grows at a rate of 7.0 KB per k with a minimum requirement of 35.4 KB for $k = 1$, using least squares fitting.

Fig. 4-2 shows the same graph G1 as Fig. 4-1 for values of k up to 10,000. Since G1 has only 6,800 solutions, the actual number of solutions extracted peaks at



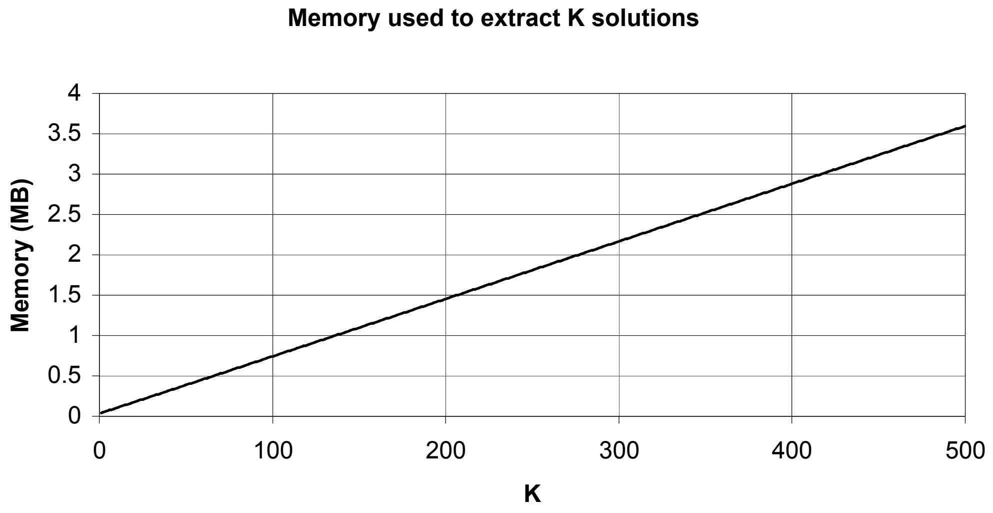
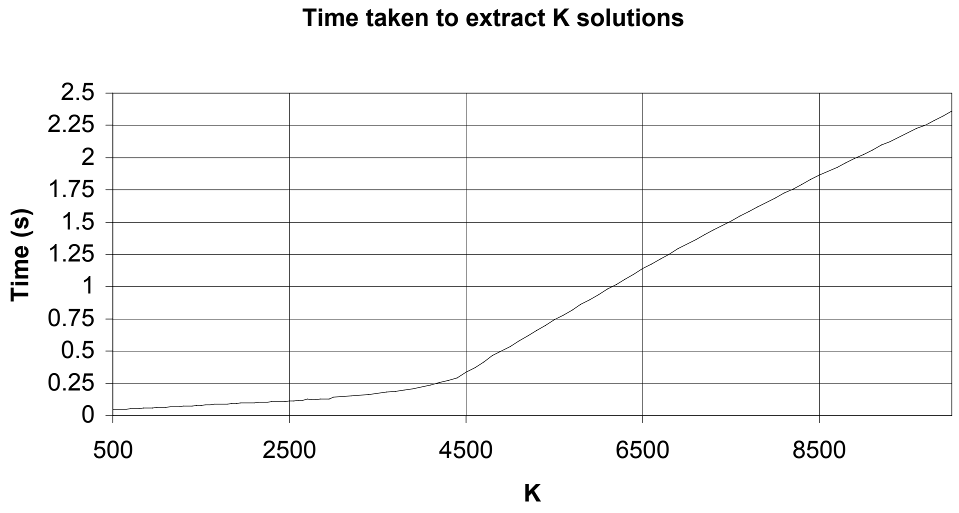


Figure 4-1: This figure shows the amount of time taken (top) and memory required (bottom) to extract k solutions from this first graph, G1. G1 has 359 nodes and 863 edges. k varies from 1 to 500.



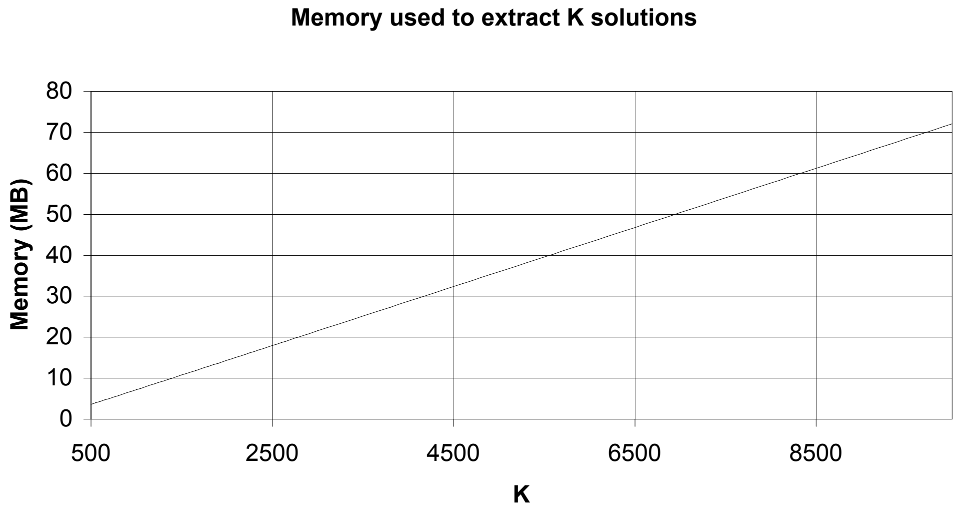


Figure 4-2: This figure shows the amount of time taken (top) and memory used (bottom) to extract k solutions from G1. k varies from 500 to 10,000 in increments of 50 for the first 3,000 and increments of 100 for the rest.

6,800. The implementation presently pre-allocates based on k alone, so the memory required continues to grow past this point, as before. The algorithm could be modified to first determine the maximal number of solutions rooted at each node and then only allocate enough space for that number of solutions. The algorithm already makes this optimization for the leaves, allocating only one copy.

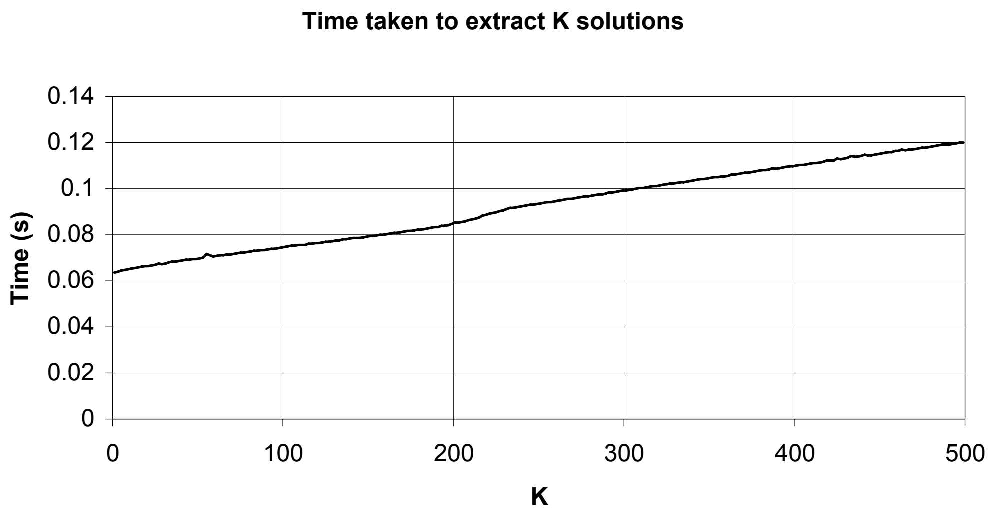
The time required starts growing at a much larger rate, though still linear, at about $k = 4,400$. We speculate this shift is also due to cache size, as at this size, the amount of memory needed *per* node exceeds the processor's cache size.

Fig. 4-3 shows the time and memory required to extract k solutions from G2. G2, like G1, is taken from a simple switched or-gate propagation example, but for a propagation breadth of 2. G2 has almost two and a half times more edges than G1 and twice as many nodes. Extracting solutions from G2 requires 0.12 ms per k with a minimum of 63.38 ms for $k = 1$. The algorithm uses 16.3 KB per k and 86.5 KB for $k = 1$.

Fig. 4-4 shows the time and memory required to extract k solutions from G3. G3 is also taken from a simple switched or-gate propagation example, but for a propagation breadth of 4. G3 is about 20% larger than G2, but it has significantly more solutions and the time complexity grows more than twice as fast as the time complexity of G2. Extracting solutions from G3 requires 0.28 ms per k and 66.08 ms for $k = 1$. The algorithm uses 18.3 KB per k and 98.9 KB for $k = 1$, only about 20% more than G2.

Fig. 4-5 shows the time and memory required to extract k solutions from G4. G4 is also taken from a simple switched or-gate propagation example, but for a propagation breadth of 50. G4 is more than an order of magnitude larger than G3 and has more solutions than fit in a 64-bit number (10^{19}). Extracting solutions from G4 requires 5.4 ms per k and 423.9 ms for $k = 1$. The algorithm uses 0.20 MB per k and 1.14 MB for $k = 1$.

Fig. 4-6 shows the time and memory required to extract k solutions from G5. G5



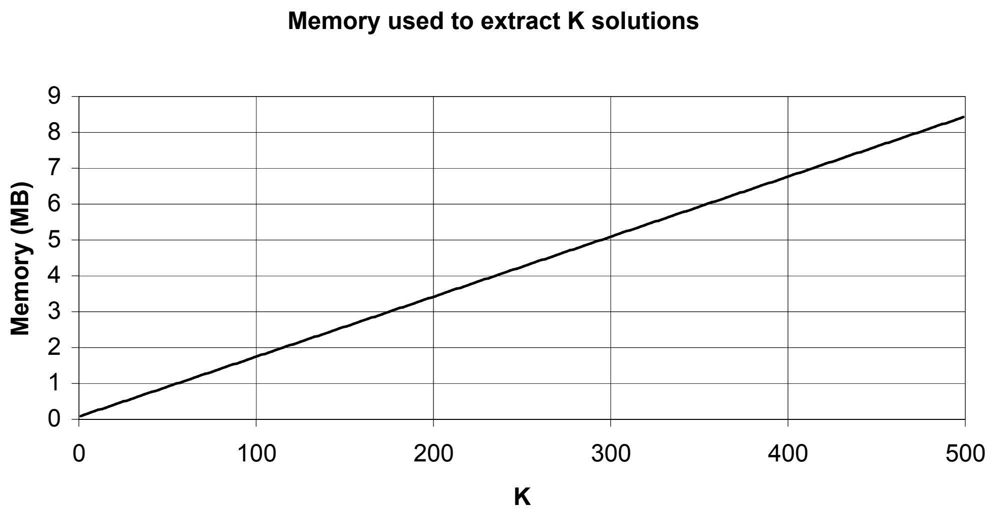
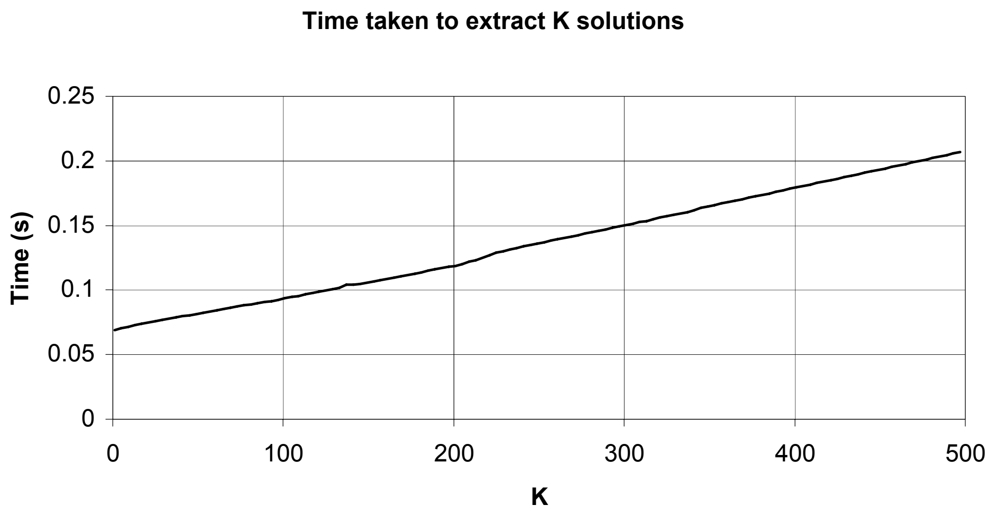


Figure 4-3: This figure shows the amount of time taken (top) and amount of memory required (bottom) to extract k solutions from G2. G2 has 750 nodes and 2,154 edges. k varies from 1 to 500 in increments of 2.



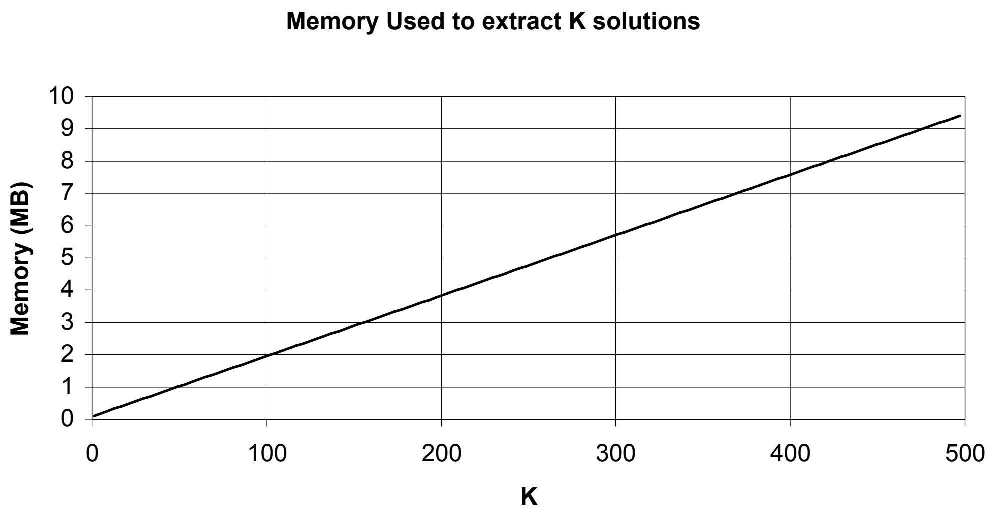
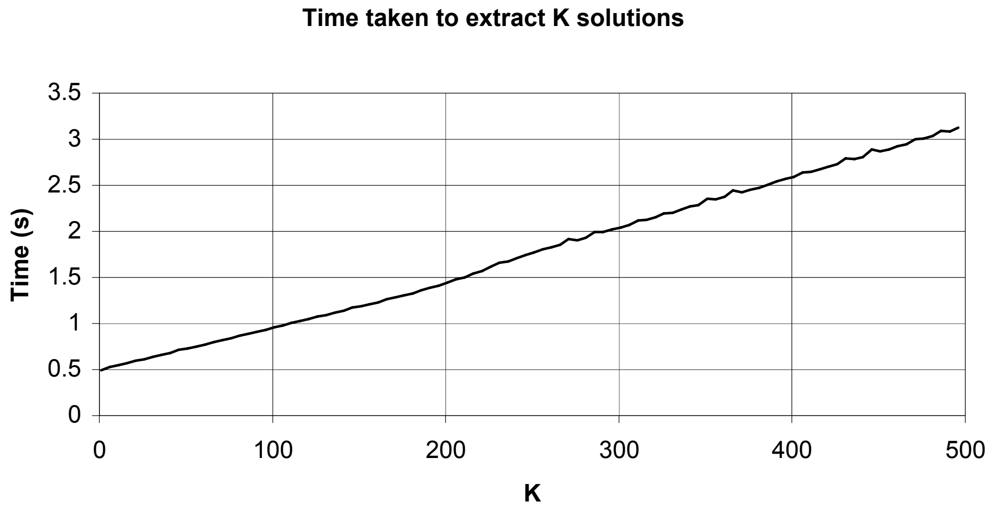


Figure 4-4: This figure shows the amount of time taken (top) and amount of memory required (bottom) to extract k solutions from G3. G3 has 809 nodes and 2,559 edges. k varies from 1 to 500 in increments of 4.



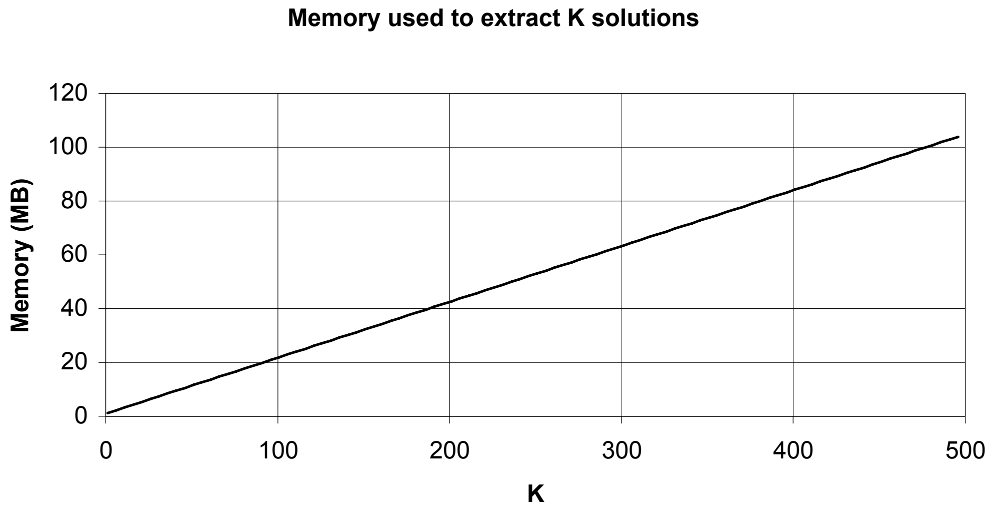
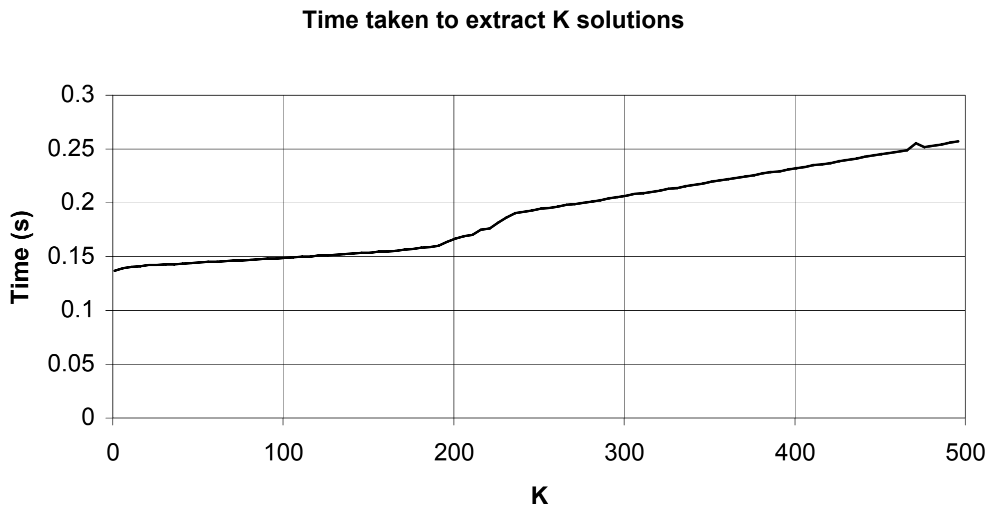


Figure 4-5: This figure shows the amount of time taken (top) and amount of memory required (bottom) to extract k solutions from G4. G4 has 4,832 nodes and 39,247 edges. k varies from 1 to 500 in increments of 5.



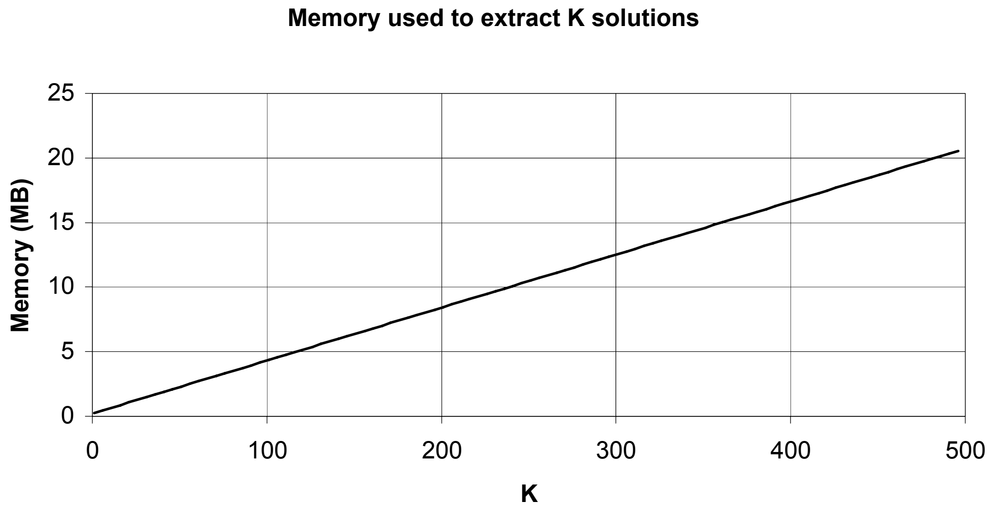


Figure 4-6: This figure shows the amount of time taken (top) and amount of memory required (bottom) to extract k solutions from G5. G5 has 1,549 nodes and 6,992 edges. k varies from 1 to 500 in increments of 10.

G	Time	$\frac{\text{Time}}{k}$	$\frac{\text{Time}}{k\|E\|}$	$\frac{\text{Time}}{\|O\|k \log \|E_v\|}$
G1	28.919 ms	0.036 ms	41.7 ps	272 ps
G2	63.38 ms	0.12 ms	53.9 ps	362 ps
G3	66.08 ms	0.28 ms	109.0 ps	740 ps
G4	423.9 ms	5.4 ms	137.6 ps	620 ps
G5	124.92 ms	0.26 ms	37.2 ps	323 ps
G6	357.82 ms	0.90 ms	67.5 ps	415 ps
G7	7.473 s	21 ms	68.2 ps	435 ps

G	Space	$\frac{\text{Space}}{k}$	$\frac{\text{Space}}{k\|E\|}$	$\frac{\text{Space}}{k\|V\|}$
G1	35.4 KB	7.0 KB	8.3 Bytes	20.0 Bytes
G2	86.5 KB	16.3 KB	7.7 Bytes	22.3 Bytes
G3	99.1 KB	18.3 KB	7.3 Bytes	23.2 Bytes
G4	1.14 MB	0.20 MB	5.3 Bytes	43.4 Bytes
G5	242 KB	40 KB	5.9 Bytes	26.4 Bytes
G6	525 KB	97 KB	7.5 Bytes	24.0 Bytes
G7	11.2 MB	2.1 MB	7.1 Bytes	27.3 Bytes

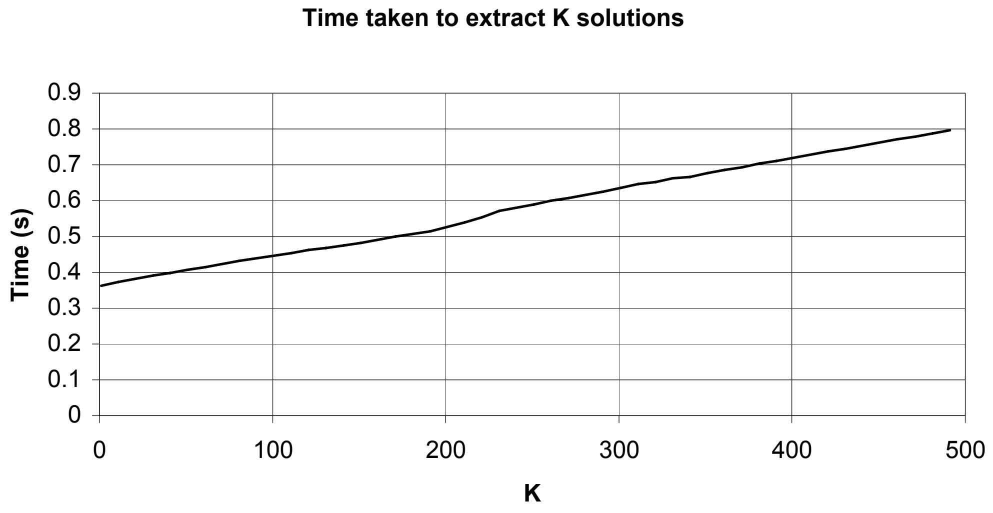
Table 4.2: This table summarizes the linear trends of the data presented in this section.

is taken from an automotive cruise control propagation example.

Fig. 4-7 shows the time and memory required to extract k solutions from G6. G6 is taken from a data transmission via a dual-band antenna example.

Fig. 4-8 shows the time and memory required to extract k solutions from G7. G7 is taken from a orbital entry propagation example.

Table 4.2 shows a summary of the linear trends demonstrated by the Find-K-Best-Solutions algorithm on the four graph examples. We used the average number of edges per node from Table 4.1 for $|E_v|$. The time term appears to be most consistent with $O(k|O| \log |E_v|)$, though there isn't enough data to confirm this. It appears different factors contribute to the actual amount of time it takes, as G2 and G3 are similar in size and come from a similar problem but G3 is many more solutions than G2



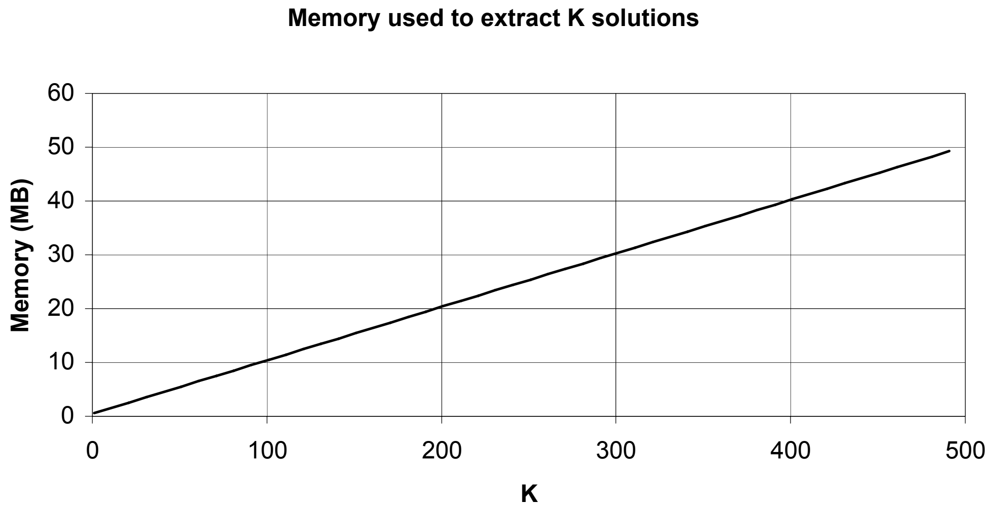
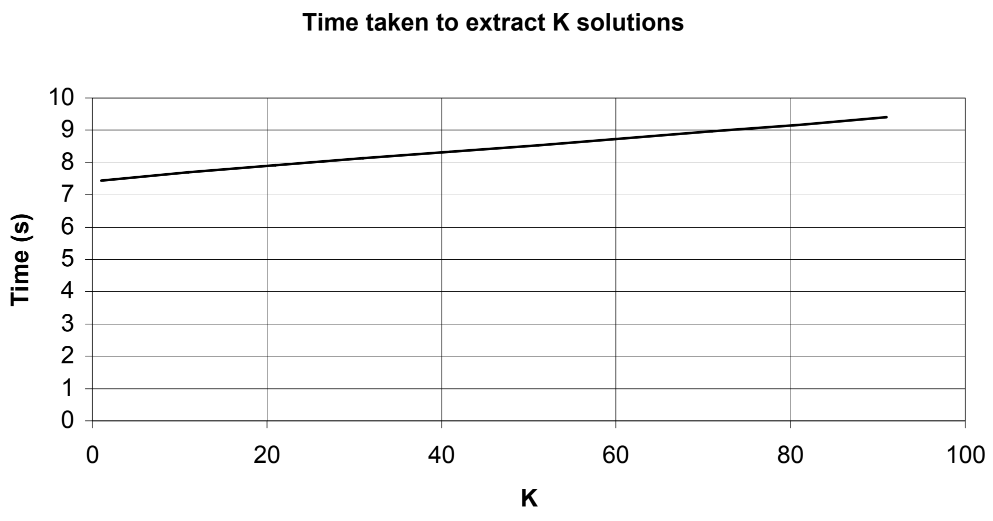


Figure 4-7: This figure shows the amount of time taken (top) and amount of memory required (bottom) to extract k solutions from G6. G6 has 4,141 nodes and 13,324 edges. k varies from 1 to 500 in increments of 10.



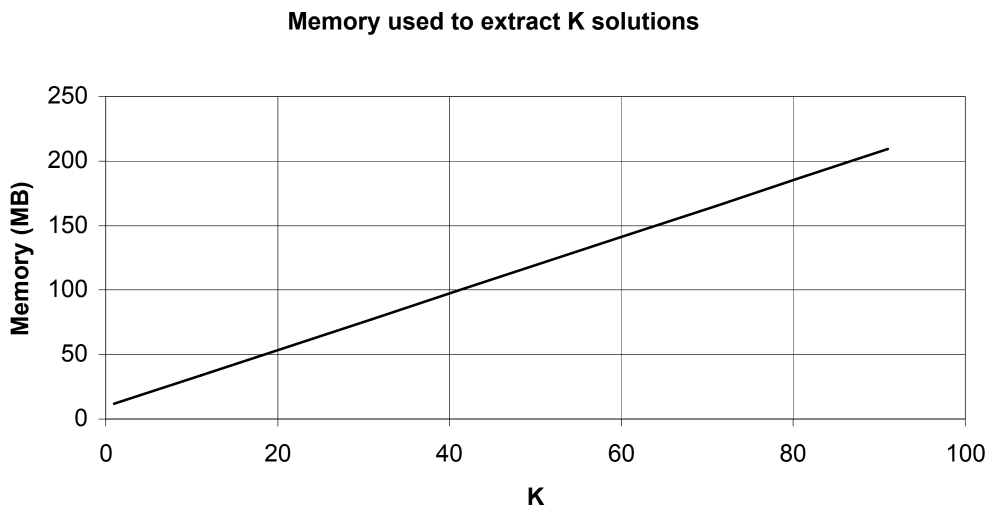


Figure 4-8: This figure shows the amount of time taken (top) and amount of memory required (bottom) to extract k solutions from G7. G7 has 80,754 nodes and 308,084 edges. k varies from 1 to 100 in increments of 10. Moreover, unlike other examples, a data point only represents 20 runs of the algorithm instead of the normal 200 runs.

and the highest value for the ratio of time to $k|O|\log|E_v|$, but this difference may be related to the actual $|E_o|$, which we did not measure. The memory required is much better behaved. The number of bytes needed grows slower than $O(k|E|)$ but faster than $O(k|V|)$.

Chapter 5

Conclusion

We now present some promising future work and then conclude.

5.1 Future Work

5.1.1 Depth-first search for solution extraction

As was stated on pages 34 and 75, the second phase of the algorithm currently needlessly walks over the whole graph pushing around markings to extract all the solutions. Since the end algorithm ended up being mostly linear, this linear cost is expected to be significant in the total time cost, as well as the space cost.

Since we're interested in all leaves connected to each root, we can just walk down the trees defined at each root copy and only store a next-child stack for **And** nodes. This reduces the time complexity of the extraction step to $O(k|\text{Sel}|)$ time, where $|\text{Sel}|$ is the number of nodes in a selection. Since each root copy is the root of a tree, and the number of leaves in a tree is one more than the number of non-leaves, we assume that $|\text{Sel}|$ is proportional to the number of symbols in a solution[1]. The space required

[1]Note that extensive use of the empty label $\emptyset$ will make the proportionality constant very large,

will reduce from $O(k|V|)$ to $O(|\mathrm{Sel}|)$ space for the data structure plus $O(k|\mathrm{Sel}|)$ for the solution itself, because we need to store at most a value per **And** nodes in the selection.

5.1.2 Memory

Memory has two problems that can be addressed. First, it lacks locality. Nodes are allocated in three blocks, one for each type, and then edges and the edge functions ξ and η are allocated inter-mixed. We speculate that the algorithm will have better time performance if each node contained all of its own data locally. Since the total memory needed for the whole algorithm can be pre-computed, the memory required can be allocated in one block and then each node can be placed in sequence based on the node ordering, including all of its edge data. To improve locality between nodes and their children, the graph can be sorted so as to minimize the average number of nodes between a parent and its children.

The second problem with memory is that the algorithm allocates more node copies than it needs to. Each internal node currently allocates a full set of k copies of itself. As we demonstrated in the examples of Chapter 3, a number of node copies will never have edges and will never take part in a final solution. The number of nodes that could ever have edges is equal to the number of selections rooted at that node (assuming this count is less than k). Counting the number of selections is a linear operation and an algorithm for doing so is provided in [8].

as it does not contribute to the number of symbols in the solution but it does contribute to the number of nodes in the selection.

5.2 Summary

This thesis has presented an novel algorithm for extracting the k best solutions from a valued and-or acyclic graph, where prior work did not exist in this area. The algorithm has a time complexity of $O(|E|k \log k + |E| \log |E_v| + |V|k \log |E_v|)$ and a space complexity of $O(k|E|)$. We then present experimental results confirming our complexity on a set of seven graphs.

The algorithm works by incrementally generating a modified graph in which every node has up to k copies, sorted by value. In the modified graph, each copy of the root node is a tree that represents a selection in the unmodified graph. The k best solutions are then the solutions of these k trees rooted at each root node copy.

Bibliography

[1] Anthony Barrett. Model compilation for real-time planning and diagnosis with feedback. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, *IJCAI*, pages 1195–1200. Professional Book Center, 2005.

[2] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*, pages 301–328. In [7], 2000. Dynamic Programming.

[3] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*, pages 549–551. In [7], 2000. Topological Sort.

[4] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*, pages 632–633. In [7], 2000. Transitive Closure.

[5] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*, pages 1060,1066. In [7], 2000. Harmonic Series.

[6] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*, pages 29,142. In [7], 2000. Merge Sort.

[7] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, 2000.

[8] Adnan Darwiche. Decomposable negation normal form. *J. ACM*, 48(4):608–647, 2001.

[9] Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *J. Artif. Intell. Res. (JAIR)*, 17:229–264, 2002.

[10] Rina Dechter. *Constraint Processing*, pages 247–249. Morgan Kaufmann, May 5 2003.

[11] Rina Dechter and Robert Mateescu. And/or search spaces for graphical models. *Artif. Intell.*, 171(2-3):73–106, 2007.

[12] Paul Elliott and Brian Williams. DNNF-based Belief State Estimation. In *Proceedings of the AAAI*, 2006.

[13] Oliver Martin. Accurate belief state update for probabilistic constraint automata. Master's thesis, Massachusetts Institute of Technology, MIT MERS, June 2005.

[14] Oliver Martin, Michel Ingham, and Brian Williams. Diagnosis as Approximate Belief State Enumeration for Probabilistic Concurrent Constraint Automata. In *Proceedings of the AAAI*, 2005.

[15] Brian C. Williams, Michel Ingham, Seung H. Chung, and Paul H. Elliott. Model-based Programming of Intelligent Embedded Systems and Robotic Space Explorers. In *Proceedings of the IEEE*, volume 9, pages 212–237, Jan 2003.