Algorithms for Generating Ordered Solutions for Explicit AND/OR Structures : Extended Abstract*

Priyankar Ghosh[†], Amit Sharma[‡], P. P. Chakrabarti[†], Pallab Dasgupta[†]
[†]Department of Computer Science and Engineering,
Indian Institute of Technology Kharagpur, WB-721302, India
[‡]Department of Computer Science, Cornell University, Ithaca, NY 14850
[†]{priyankar,ppchak,pallab}@cse.iitkgp.ernet.in, [‡]asharma@cs.cornell.edu

Abstract

We present algorithms for generating alternative solutions for explicit acyclic AND/OR structures in non-decreasing order of cost. Our algorithms use a best first search technique and report the solutions using an implicit representation ordered by cost. Experiments on randomly constructed AND/OR DAGs and problem domains including matrix chain multiplication, finding the secondary structure of RNA, etc, show that the proposed algorithms perform favorably to the existing approach in terms of time and space.

1 Introduction

Traditionally AND/OR graphs [Nilsson, 1980] have been used in problem reduction search for solving complex problems, logical reasoning, theorem proving, etc. Over the last decade there has been a renewed research interest towards applying AND/OR structures in a wide variety of other areas like planning with uncertainty [Russell and Norvig, 2003], graphical models [Dechter and Mateescu, 2007], web service composition [Lang *et al.*, 2005], solution sampling and counting [Gogate and Dechter, 2008], model based programming [Elliott, 2007], etc. However, the problem of generating alternative solutions which has been studied for ordinary graphs extensively [Nilsson, 1998; Eppstein, 1998; 1990; Chegireddy and Hamacher, 1987], has not been deeply studied for AND/OR graphs and the approaches related to these topics are often not obvious.

An ordered set of solutions of an explicit AND/OR DAG can be used to develop useful variants of the AO* algorithm. Currently in AO*, only the minimum cost solution is computed whereas several variants of the A* algorithm exist, where solutions are often sought within a factor of cost of the optimal solution. These approaches were developed to adapt the A* algorithm for using inadmissible heuristics, leveraging multiple heuristics, generating solutions quickly within bounded sub-optimality, [Ebendt and Drechsler, 2009; Chakrabarti *et al.*, 1989; Pearl, 1984], etc. Similar techniques can be developed for AO* search if ordered set of potential solutions are made available. An initial algorithm for generating k-best solutions for AND/OR DAGs has been proposed in [Elliott, 2007] in the context of model based programming. For graphical models, which can be expressed using AND/OR structures, ordered sets of solutions are useful [Flerova and Dechter, 2011; 2010]. In the domain of service composition, the primary motivation behind providing a set of alternative solutions ordered by cost is to offer more choices, while trading off the specified cost criterion (to a limited extent) in favor of other 'unspecified' criteria (primarily from the standpoint of quality) and algorithms have been developed for generating a ranked set of solutions for the service composition problem [Shiaa *et al.*, 2008].

The existing method [Elliott, 2007] (will be referred to as BU henceforth) works bottom-up by computing k-best solutions for the current node. We present a best first search algorithm, named *Alternative Solution Generation* (ASG) for generating an ordered set of solutions. The proposed algorithm maintains a list of candidate solutions, initially containing only the optimal solution, and iteratively generates the next solution in non-decreasing order of cost by selecting the minimum cost solution from the list. In each iteration, this minimum cost solution is used to construct another set of candidate solutions, which is again added to the current list. We present two versions of the algorithm, namely, (*i*) ***Basic ASG (will be referred to as ASG henceforth) :*** This version of the algorithm may construct a particular candidate solution more than once; (*ii*) ***Lazy ASG or LASG :*** A variant of ASG that constructs every candidate solution only once.

We use a compact representation, named *signature*, for storing the solutions. From the signature of a solution, the actual explicit form of that solution can be constructed through a top-down traversal of the given DAG. This representation allows the proposed algorithms to work in a top-down fashion starting from the initial optimal solution. Unlike the existing approach, our proposed algorithms are interruptible in nature, i.e., our algorithms can be stopped at any point of time during the execution and the set of ordered solutions obtained so far can be observed and subsequent solutions will be generated when the algorithms are resumed again. Moreover, if an upper limit estimate on the number of solutions required is known a priori, our algorithms can be further optimized using that estimate. We have used randomly constructed trees and DAGs as well as domains including the 5-*peg Tower of*

*This paper is an extended abstract of the JAIR publication [Ghosh *et al.*, 2012]

Hanoi problem, the *matrix-chain multiplication* problem and the problem of finding the *secondary structure of RNA* as test domains. Experimental results show that our proposed algorithms outperform the existing approach by a large margin in terms of the memory used and time required for generating a specific number of ordered solutions for different domains.

2 Background and Proposed Algorithms

In this paper we use $G_{\alpha\beta} = \langle V, E\rangle$ to denote an AND/OR directed acyclic graph where V is the set of nodes, and E is the set of edges. The nodes of $G_{\alpha\beta}$ with no successors are called *terminal* nodes. The *non-terminal* nodes of $G_{\alpha\beta}$ are of two types – *i*) OR nodes and *ii*) AND nodes . The start (or root) node of $G_{\alpha\beta}$ is denoted by v_R. *OR edges* and *AND edges* are the edges that emanate from OR nodes and AND nodes respectively. We use the standard notion of *solution graph*, $S(v_q)$, rooted at any node $v_q \in V$ [Martelli and Montanari, 1973]. By a solution graph S of $G_{\alpha\beta}$ we mean a solution graph with root v_R.

We present our algorithms using the notion of *alternating* AND/OR trees which offer a succinct representation. Also, the correctness proofs are much simpler for alternating AND/OR trees. Formally, an *alternating* AND/OR tree, $\hat{T}_{\alpha\beta} = \langle V, E\rangle$, is an AND/OR tree with the restriction that there is an alternation between the AND nodes and the OR nodes. Every child of an AND node is either an OR node or a terminal node, and every children of an OR node is either an AND node or a terminal node. We have shown in [Ghosh *et al.*, 2012] that every AND/OR tree can be converted to an equivalent alternating AND/OR tree with respect to the solution space. The notion of cost is as follows. In $G_{\alpha\beta}$, every edge $e_{qr} \in E$ from node v_q to node v_r has a finite non-negative cost $c_e(\langle v_q, v_r\rangle)$ or $c_e(e_{qr})$. Similarly every node v_q has a finite non-negative cost denoted by $c_v(v_q)$. We use the notion of additive costs for given *solution graph* S [Martelli and Montanari, 1973].

Definition 2.a **[Cost of a Solution Graph]** For every node v_q in S, $C(S, v_q)$ denotes the cost of the solution graph, $S(v_q)$. The cost of a solution S is $C(S, v_R)$ which is also denoted by $C(S)$. We denote the optimal solution below every node v_q as *opt*(v_q). Therefore, the optimal solution of the entire AND/OR DAG $G_{\alpha\beta}$, denoted by S_{opt}, is *opt*(v_R). The cost of the optimal solution rooted at every node v_q in $G_{\alpha\beta}$ is $C_{opt}(v_q)$. The cost of the optimal solution S_{opt} of $G_{\alpha\beta}$ is denoted by $C_{opt}(v_R)$ or, alternatively, by $C_{opt}(S_{opt})$. ☐

The existing method, BU [Elliott, 2007], works with the input parameter k, i.e., the number of solutions to be generated have to be known a priori. Also this method is not inherently incremental in nature, thus does not perform efficiently when the solutions are needed on demand, e.g., first initial 20 solutions are needed, then the next 10 are needed. In this case the first 20 solutions will have to be recomputed again while computing next 10 solutions, i.e., from the 21st solution to the 30th solution. We address these limitations in our proposed approach. In this paper, we present our proposed algorithms for alternating AND/OR trees for brevity. The details of the changes required for handling AND/OR DAGs are presented in detail in [Ghosh *et al.*, 2012].

2.1 Core Concepts and Definitions

Our proposed ASG algorithm works top-down using edge markings, to generate the next best solutions from the previously generated solutions. ASG works iteratively – it generates the next best solution based on the previously generated solutions. Thus, we first compute the optimal solution, and subsequently generate other solutions from the already generated solutions. The following terminology and notions are used to describe the *ASG* algorithm. In the context of AND/OR trees, we use e_q to denote the edge that points to the vertex v_q. Our algorithm has been developed on the notion of – (a) marking of an OR edge, (b) swap option and swap operation and (b) the implicit representation of a solution. Next, we briefly describe the concepts.

Definition 2.b **[Aggregated Cost]** In an AND/OR DAG $G_{\alpha\beta}$, the *aggregated cost*, c_a, for an edge e_{ij} from node v_i to node v_j, is defined as : $c_a(e_{ij}) = c_e(e_{ij}) + C_{opt}(v_j)$. ☐

Definition 2.c **[V_{opt} and E_{opt}]** For any solution graph S_m of an AND/OR DAG $G_{\alpha\beta}$, we define a set of nodes, $V_{opt}(S_m)$, and a set of OR edges, $E_{opt}(S_m)$, as:

1. $V_{opt}(S_m) = \{v_q \mid v_q$ in S_m and solution graph $S_m(v_q)$ is identical to the solution graph $opt(v_q)\}$
2. $E_{opt}(S_m) = \{e_{pr} \mid$ OR edge e_{pr} in S_m, and $v_r \in V_{opt}(S_m)\}$

Clearly, for any node $v_q \in V_{opt}(S_m)$, if v_q is present in S_{opt}, then the solution graph $S_m(v_q)$ is identical to the solution graph $S_{opt}(v_q)$. Also $C(S_m, v_q) = C_{opt}(v_q)$ ☐

Marking of an OR edge. The notion of marking an OR edge is as follows. For an OR node v_q, $L(v_q)$ is the list of OR edges of v_q sorted in non-decreasing order of the *aggregated cost* of the edges. We define $\delta_{(i,i+1)}$ as the difference of the *cost* of OR edge e_i and e_{i+1}, where e_i and e_{i+1} emanate from the same OR node v_q, and e_{i+1} is the edge next to e_i in $L(v_q)$. Consider a solution, S_{cur}, containing the edge $e_i = (v_q, v_i)$, where $e_i \in E_{opt}(S_{cur})$. We mark e_i with the cost increment which will be incurred to construct the next best solution from S_{cur} by choosing another child of v_q.

Definition 2.d **[Swap Option]** A *swap option* σ_{ij} is defined as a three-tuple $\langle e_i, e_j, \delta_{ij}\rangle$ where e_i and e_j emanate from the same OR node v_q, e_j is the edge next to e_i in $L(v_q)$, and $\delta_{ij} = c_a(e_j) - c_a(e_i)$. Also, we say that the swap option σ_{ij} *belongs* to the OR node v_q. ☐

Consider the OR node v_q and the sorted list $L(v_q)$. It may be observed that in $L(v_q)$ every consecutive pair of edges forms a swap option. Therefore, if there are k edges in $L(v_q)$, $k-1$ swap options will be formed. At node v_q, these swap options are *ranked* according to the rank of their *original edges* in $L(v_q)$. We present an example of an alternating AND/OR tree in Figure 1. The terminal nodes are represented by a circle with thick outline. AND nodes are shown in the figures with their outgoing edges connected by a semi-circular curve in all the examples. The edge costs are shown by the side of each edge within an angled bracket. The cost of the terminal nodes are shown inside a box. For every non-terminal node v_q, the pair of costs, $c_v(v_q)$ and $C_{opt}(v_q)$, is shown inside a rectangle. The optimal solution below every node is shown

using by thick dashed edges with an arrow head. The marks corresponding to OR edges e_2, e_3, e_9, e_{11}, and e_{13} are $[e_2 : 5]$, $[e_3 : 1]$, $[e_9 : 3]$, $[e_{11} : 4]$, and $[e_{13} : 3]$. Similarly, the corresponding swap options are : $\sigma_{(2,3)} = \langle e_2, e_3, 5\rangle$, $\sigma_{(3,4)} = \langle e_3, e_4, 1\rangle$, $\sigma_{(9,10)} = \langle e_9, e_{10}, 3\rangle$, $\sigma_{(11,12)} = \langle e_{11}, e_{12}, 4\rangle$, and $\sigma_{(13,14)} = \langle e_{13}, e_{14}, 3\rangle$.

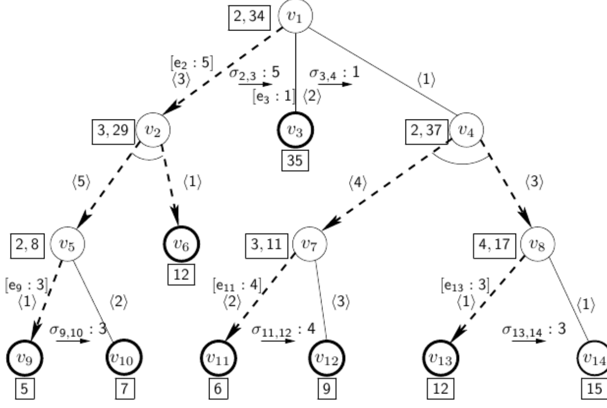

Figure 1: Example of OR-edge marking and swap option

Definition 2.e **[Swap Operation]** *Swap operation* is defined as the application of a swap option $\sigma_{ij} = \langle e_i, e_j, \delta_{ij}\rangle$ to a solution S_m that contains the OR edge e_i in the following way:

1. Remove the subtree rooted at v_i from S_m. Let the modified tree be S'_m. Edge e_i is the *original edge* of σ_{ij}.
2. Add the subtree $opt(v_j)$ to S'_m, which is constructed at the previous step. Let the newly constructed solution be S''_m. Edge e_j is the *swapped edge* of σ_{ij}.

Intuitively, a swap operation constructs a new solution S'_m from S_m and the cost of S'_m is increased by δ_{ij} compared to cost of S_m. □

Our proposed algorithms use a swap option based compact representation, named *signature*, for storing the solutions. Intuitively, any alternative solution can be described as a set of swap operations performed on the optimal solution S_{opt}. It is interesting to observe that while applying an ordered sequence of swap options, $\langle\sigma_1, \cdots, \sigma_k\rangle$, the application of each swap operation creates an intermediate alternative solution. For example, when the first swap option in the sequence, σ_1, is applied to the optimal solution, S_{opt}, a new solution, say S_1, is constructed. Then, when the 2nd swap option, σ_2, is applied to S_1, yet another solution S_2 is constructed. Let S_i denote the solution obtained by applying the swap options, $\sigma_1, \cdots, \sigma_i$, on S_{opt} in this sequence.

Suppose the swap option σ_j belongs to a node v_{p_j}. Now it is important to observe that the application of σ_j on S_{j-1} to construct S_j, invalidates the application of all other swap options that belong to an OR edge in the path from the root node to v_{p_j} in the solution S_j. This is because in S_j the application of any such swap option which belongs to an OR edge in the path from the root node to v_{p_j} would make the swap at v_{p_j} redundant. In fact, for each swap option σ_i belonging to node v_{p_i}, where $1 \leq i \leq j$, the application of all other swap options that belong to an OR edge in the path from the root node to v_{p_i} is invalidated in the solution S_j for the same reason. This condition restricts the set of swap options that can be applied on a particular solution.

Definition 2.f **[Order Relation $\hat{\mathcal{R}}$]** We define an order relation, namely $\hat{\mathcal{R}}$, between a pair of swap options as follows.

1. If there is a path from v_i to v_r in $\hat{T}_{\alpha\beta}$, where e_i and e_r are OR edges, σ_{qi} and σ_{rj} are swap options, then $(\sigma_{qi}, \sigma_{rj}) \in \hat{\mathcal{R}}$.
2. If $\sigma_{pq} = \langle e_p, e_q, \delta_{pq}\rangle$ and $\sigma_{rt} = \langle e_r, e_t, \delta_{rt}\rangle$ are two swap options such that $v_q = v_r$, then $(\sigma_{pq}, \sigma_{rt}) \in \hat{\mathcal{R}}$.

Implicit Representation of the Solutions. We use an *implicit* representation for storing every solution other than the optimal one. These other solutions can be constructed from the optimal solution by applying a set of swap options to the optimal solution in the following way. If $(\sigma_i, \sigma_j) \in \hat{\mathcal{R}}$, σ_i has to be applied before σ_j. Therefore, every solution is represented as a sequence $\hat{\Sigma}$ of swap options, where σ_i appears before σ_j in $\hat{\Sigma}$ if $(\sigma_i, \sigma_j) \in \hat{\mathcal{R}}$. Intuitively the application of every swap option specifies that the swapped edge will be the part of the solution.

Definition 2.g **[Signature of a Solution]** The minimal sequence of swap options corresponding to a solution, S_m, is defined as the *signature*, $Sig(S_m)$, of that solution. It may be noted that for the optimal solution S_{opt} of any alternating AND/OR tree $\hat{T}_{\alpha\beta}$, $Sig(S_{opt}) = \{\}$, i.e., an empty sequence. It is possible to construct more than one signature for a solution, as $\hat{\mathcal{R}}$ is a partial order. It is important to observe that all different signatures for a particular solution are of equal length and the sets of swap options corresponding to these different signatures are also equal. Therefore the set of swap options corresponding to a signature is a canonical representation of the signature. Henceforth we will use the set notation for describing the signature of a solution.

Definition 2.h **[Swap List]** The *swap list* corresponding to a solution S_m, $\mathcal{L}(S_m)$, is the list of swap options that are applicable to S_m. Let $Sig(S_m) = \{\sigma_1, \cdots, \sigma_m\}$ and $\forall i,\ 1 \leq i \leq m$, each swap option σ_i belongs to node v_{p_i}. The application of all other swap options that belong to the OR edges in the path from the root node to v_{p_i} is invalidated in the solution S_m. Hence, only the remaining swap options that are not invalidated in S_m can be applied to S_m for constructing the successor solutions of S_m.

It is important to observe that for a swap option σ_i, if the *source edge* of σ_i belongs to $E_{opt}(S_m)$, the application is not invalidated in S_m. Hence for a solution S_m, we construct $\mathcal{L}(S_m)$ by restricting the swap operations only on the edges belonging to $E_{opt}(S_m)$. Moreover, this condition also ensures that the cost of a newly constructed solution can be computed directly form the cost of the parent solution and the δ value of the applied swap option. To elaborate, suppose solution S'_m is constructed form S_m by applying σ_{jk}. The cost of S'_m can be computed directly form $C(S_m)$ and σ_{jk} as : $C(S'_m) = C(S_m) + \delta_{jk}$ if $e_j \in E_{opt}(S_m)$. □

Definition 2.i **[Successors and Predecessors of a Solution]** The set of successors and predecessors of a solution S_m is :

1. $Succ(S_m) = \{S'_m \mid S'_m$ can be constructed from S_m by applying a swap option that belongs to the swap list of $S_m\}$
2. $Pred(S_m) = \{S''_m \mid S_m \in Succ(S''_m)\}$ □

2.2 ASG algorithm

In our ASG algorithm, we maintain a list, Open, which initially contains only the optimal solution S_{opt}. At any point of time Open contains a set of candidate solutions from which the next best solution in the non-decreasing order of cost is selected. At each iteration the minimum cost solution (S_{min}) in Open is removed from Open and added to another list, named, Closed. The Closed list contains the set of ordered solutions generated so far. Then the successor set of S_{min} is constructed and any successor solution which is not currently present in Open as well as is not already added to Closed is inserted to Open. However as a further optimization, we use a sublist of Closed, named TList, to store the relevant portion of Closed such that checking with respect to the solutions in TList is sufficient to figure out whether the successor solution is already added to Closed. It is interesting to observe that this algorithm can be interrupted at any time and the set of ordered solutions computed so far can be obtained. Also, the algorithm can be resumed if some more solutions are needed. We briefly mention the key lemma and theorems.

Definition 2.j **[Solution Space DAG(SSDAG)]** The *solution space DAG* of an AND/OR tree $\hat{T}_{\alpha\beta}$ is a directed acyclic graph (DAG), $\mathcal{G}^s = \langle \mathcal{V}, \mathcal{E} \rangle$, where $\mathcal{V}$ is the set of all possible solutions of the AND/OR tree $\hat{T}_{\alpha\beta}$, and $\mathcal{E}$ is the set of edges which is defined as:

$$\mathcal{E} = \left\{ e^s_{pm} \;\middle|\; \begin{array}{l} S_p, S_m \in \mathcal{V}, \text{ and} \\ e^s_{pm} \text{ is a directed edge from } S_p \text{ to } S_m, \\ \text{and } S_m \in Succ(S_p) \end{array} \right\}$$

Clearly S_{opt} is the root node of $\mathcal{G}^s$. □

Lemma 2.1. For any alternating AND/OR tree $\hat{T}_{\alpha\beta}$, for every node of the SSDAG of $\hat{T}_{\alpha\beta}$, ASG algorithm generates the solution corresponding to that node. □

Theorem 2.1. $\forall S_j \in \mathcal{V}$, S_j is generated by ASG algorithm only once and in the non-decreasing order of costs while ties among the solutions having same costs are resolved arbitrarily in the favour of predecessor solutions. □

2.3 LASG algorithm

Although ASG algorithm adds a solution to Closed only once, a particular solution may be present in the successor set of more than one solutions. Thus it is necessary to check whether a particular solution S_i is already present in Open or TList before adding S_i to Open. We present a lazy version of ASG, LASG, which avoids this checking for duplicates using the notion of *native swap options*. Conceptually, the native swap options for a solution S_q are the swap options that become available immediately after applying σ_{ij}, but were not available in the predecessor solution of S_q.

We use the notion of *solution space tree* which is conceptually a sub-tree of a spanning tree of the solution space DAG and the *spanning trees* of the solution space DAGs are *complete* solution space trees. In LASG, instead of using the entire swap list of a solution to construct all successors at once and then add those solutions to Open, the native swap options are used for constructing a subset of the successor set. This subset consists of only those solutions that are currently not present in Open and thus can be added to Open without comparing with the existing entries in Open. The construction of each remaining successor solution S'_m of S_m and subsequently the insertion of S'_m to Open is delayed until every other predecessor solution of S'_m is added to Closed. In LASG, the solution space tree $\mathcal{T}^s$ is maintained to determine when every other predecessor of S'_m is added to Closed.

Theorem 2.2. The solution space tree constructed by LASG algorithm is *complete*. □

Complexity : We briefly compare the complexities of our proposed algorithms, ASG and LASG with the existing approach BU. For BU algorithm, the time complexity of generating the $\mathfrak{c}$ best solutions for an AND/OR tree is $O(n_{\alpha\beta}.\mathfrak{c}.\log\mathfrak{c})$ and the space complexity is $O(n_{\alpha\beta}.\mathfrak{c})$, where $n_{\alpha\beta}$ denotes the number of nodes in the AND/OR tree. Whereas, the space complexity of both ASG and LASG algorithm is $O(n_{\alpha\beta}.\mathfrak{c})$. The time complexity of LASG is $O(\mathfrak{c}.n_{\alpha\beta})$ which is $\log\mathfrak{c}$ factor better than BU. When an additional hash-map is used to reduce the time overhead of duplicate checking, the time complexity of ASG becomes $O\big(n_{\alpha\beta} + \sqrt{n_{\alpha\beta}}.(\mathfrak{c}.\lg\mathfrak{c} + \mathfrak{c}.\lg n_{\alpha\beta})\big)$, which is asymptotically lower than both LASG and BU. It is important to note that, using the additional hash map does not change asymptotic space complexity of ASG, although the exact space requirement is doubled.

3 Summary of Experimental Results

We have used randomly constructed trees and DAGs as well as some well-known problem domains including the 5-*peg Tower of Hanoi* problem, the *matrix-chain multiplication* problem and the problem of finding the *secondary structure of RNA* as test domain. For the randomly constructed trees, LASG runs upto 20 times faster than the existing approach BU. For randomly constructed DAGs, the the running time of LASG is upto 100 times lower than the running time of BU and between ASG and BU, the running times are comparable. Also, for randomly constructed DAGs, the space requirement is upto 5 times lower on average for ASG compared to BU; LASG requires around 50% of the space required by ASG. We have also empirically shown that, if the number of solutions to be generated is known a priori, ASG and LASG can be further optimized in terms of running time.

For *5-peg tower of Hanoi*, *matrix-chain multiplication* and *secondary structure of RNA*, the performances of ASG and LASG are similar. For *5-peg tower of Hanoi* problem LASG runs around two times faster on average than BU. Whereas, for *matrix-chain multiplication* and *secondary structure of RNA* the running time of LASG is two orders of magnitude better on average than the running time of BU and the space requirement of LASG is one order of magnitude better on average than that of BU. Details of the experimental results as well as the analysis based on the size of Open are presented in [Ghosh *et al.*, 2012].

References

[Chakrabarti *et al.*, 1989] P. P. Chakrabarti, Sujoy Ghose, A. Pandey, and S. C. DeSarkar. Increasing search efficiency using multiple heuristics. *Inf. Process. Lett.*, 32(5):275–275, 1989.

[Chegireddy and Hamacher, 1987] Chandra R. Chegireddy and Horst W. Hamacher. Algorithms for finding k-best perfect matchings. *Discrete Applied Mathematics*, 18(2):155–165, 1987.

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

[Ebendt and Drechsler, 2009] Rdiger Ebendt and Rolf Drechsler. Weighted A^* search - unifying view and application. *Artificial Intelligence*, 173(14):1310 – 1342, 2009.

[Elliott, 2007] Paul Elliott. Extracting the k best solutions from a valued and-or acyclic graph. Master's thesis, Massachusetts Institute of Technology, 2007.

[Eppstein, 1990] David Eppstein. Finding the k smallest spanning trees. In *Proc. 2nd Scandinavian Worksh. Algorithm Theory*, number 447 in Lecture Notes in Computer Science, pages 38–47. Springer Verlag, 1990.

[Eppstein, 1998] David Eppstein. Finding the k shortest paths. *SIAM J. Comput.*, 28(2):652–673, 1998.

[Flerova and Dechter, 2010] Natalia Flerova and Rina Dechter. M best solutions over graphical models. In *1st Workshop on Constraint Reasoning and Graphical Structures*, September 2010.

[Flerova and Dechter, 2011] Natalia Flerova and Rina Dechter. Bucket and mini-bucket schemes for m best solutions over graphical models. In *GKR 2011(a workshop of IJCAI 2011)*, 2011.

[Ghosh *et al.*, 2012] Priyankar Ghosh, A. Sharma, P. P. Chakrabarti, and Pallab Dasgupta. Algorithms for generating ordered solutions for explicit AND/OR structures. *J. Artif. Intell. Res. (JAIR)*, 44:275–333, 2012.

[Gogate and Dechter, 2008] Vibhav Gogate and Rina Dechter. Approximate solution sampling (and counting) on AND/OR spaces. In *CP*, pages 534–538, 2008.

[Lang *et al.*, 2005] Q. A. Lang, , and Y.W. Su. AND/OR graph and search algorithm for discovering composite web services. *International Journal of Web Services Research*, 2(4):46–64, 2005.

[Martelli and Montanari, 1973] A. Martelli and U. Montanari. Additive AND/OR graphs. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[Nilsson, 1980] Nils J. Nilsson. *Principles of artificial intelligence*. Tioga Publishing Co., 1980.

[Nilsson, 1998] D. Nilsson. An efficient algorithm for finding the m most probable configurations in probabilistic expert systems. *Statistics and Computing*, 8:159–173, June 1998.

[Pearl, 1984] Judea Pearl. *Heuristics: intelligent search strategies for computer problem solving*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1984.

[Russell and Norvig, 2003] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*, chapter Planning, pages 375–461. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition edition, 2003.

[Shiaa *et al.*, 2008] Mazen Malek Shiaa, Jan Ove Fladmark, and Benoit Thiell. An incremental graph-based approach to automatic service composition. *IEEE International Conference on Services Computing*, 4(2):46–64, 2008.