ELSEVIER

Artificial Intelligence 129 (2001) 91–131

Artificial Intelligence

www.elsevier.com/locate/artint

A general scheme for automatic generation of search heuristics from specification dependencies [☆]

Kalev Kask [*], Rina Dechter

Department of Information and Computer Science, University of California, Irvine, CA 92697-3425, USA

Received 15 February 2000; received in revised form 3 March 2001

Abstract

The paper presents and evaluates the power of a new scheme that generates search heuristics mechanically for problems expressed using a set of functions or relations over a finite set of variables. The heuristics are extracted from a parameterized approximation scheme called Mini-Bucket elimination that allows controlled trade-off between computation and accuracy. The heuristics are used to guide Branch-and-Bound and Best-First search. Their performance is compared on two optimization tasks: the Max-CSP task defined on deterministic databases and the Most Probable Explanation task defined on probabilistic databases. Benchmarks were random data sets as well as applications to coding and medical diagnosis problems. Our results demonstrate that the heuristics generated are effective for both search schemes, permitting controlled trade-off between preprocessing (for heuristic generation) and search. © 2001 Elsevier Science B.V. All rights reserved.

Keywords: Heuristic search; Automated reasoning

1. Introduction

Heuristic search is a general problem solving method applicable to a wide range of tasks. Its efficiency depends on the quality of the heuristic evaluation function. Therefore, one of the most important issues in heuristic search is obtaining a good heuristic function. Often there is a trade-off between the quality of the heuristic and the complexity of its computation. In this paper we will present a general scheme of mechanically generating search heuristics from a problem description. Within this scheme, the trade-off between

[☆] Preliminary versions of this paper were presented in [15,16,18]. This work was supported in part by NSF grant IIS-0086529 and by MURI ONR award N00014-00-1-0617.

[*] Corresponding author.

E-mail addresses: kkask@ics.uci.edu (K. Kask), dechter@ics.uci.edu (R. Dechter).

the quality of the heuristic function and its computational complexity is quantified and controlled by an input parameter.

The main difference between the approach presented in this paper and traditional work on mechanical generation of heuristics is in its premises. We assume that the problem is specified by a dependency model: a set of functions or relations over variables (e.g., Bayesian network, constraint network, decomposable cost functions). From a dependency model, the states and the transition rules defining a search space for the problem can be obtained. In contrast, the starting point of heuristic search is the search space description (states, operators, initial and goal states) only.

Our scheme is based on the Mini-Bucket scheme, a class of parameterized approximation algorithms based on the bucket-elimination framework [3]. The approximation uses a controlling parameter which allows adjustable levels of accuracy and efficiency [7]. It has been presented and analyzed for probabilistic tasks such as finding the most probable explanation (MPE), belief updating, and finding the maximum a posteriori hypothesis. Encouraging empirical results were reported on randomly generated Noisy-OR networks, on medical-diagnosis CPCS networks, and on coding problems [36]. However, as evidenced by the error bound produced by these algorithms, in some cases the approximation is seriously suboptimal, even when using the highest feasible accuracy level. In such cases, augmenting the Mini-Bucket approximation with search could be cost-effective.

In this paper we demonstrate our approach on two different classes of optimization tasks: solving the Max-CSP problem in Constraint Optimization and finding the Most Probable Explanation in a Bayesian network. We will show that the functions produced by the Mini-Bucket method can serve as the basis for creating heuristic evaluation functions for search. These heuristics provide either a lower bound (for minimization problems, such as Max-CSP) or upper bound (for maximization problems, such as MPE) on the cost of the best extension of a given partial assignment. Since the Mini-Bucket's accuracy is controlled by a bounding parameter, it allows heuristics having varying degrees of accuracy and results in a spectrum of search algorithms that can trade off heuristic computation and search. We evaluate the power of the Mini-Bucket heuristics within both *Branch-and-Bound*[1] and *Best-First* search.

Branch-and-Bound searches the space of partial assignments in a depth-first manner. It will expand a partial assignment only if the bound computed by the heuristic function is potentially better than the value of the current best solution. The virtue of Branch-and-Bound is that it requires a limited amount of memory and can be used as an anytime scheme; whenever interrupted, Branch-and-Bound outputs the best solution found so far. Best-First explores the search space in uniform frontiers of partial instantiations, each having the same value for the evaluation functions, while progressing by expanding nodes with the best heuristic values first. Since, as shown, the generated heuristics are admissible and monotonic, their use within Best-First search yields A* type algorithms whose properties are well understood. The algorithm is guaranteed to terminate with an optimal solution. When provided with more powerful heuristics, it explores a smaller

[1] Branch-and-Bound was proposed originally as a general search paradigm that includes Depth-First search and Best-First search as special cases when the heuristic function is admissible [14,23]. To put our work in this context, by Branch-and-Bound in this paper we mean Depth-First Branch-and-Bound.

search space, but otherwise it requires substantial space. It is also known that Best-First algorithms are optimal. Namely, when given the same heuristic information, Best-First search is the most efficient algorithm in terms of the size of the search space it explores [5]. Still, Best-First may occasionally fail because of its memory requirements. Hybrid approaches similar to those presented for A* in the heuristic search community in the past decade are clearly of potential value here as well [20].

In this paper, a Best-First algorithm with Mini-Bucket heuristics (BFMB) and a Branch-and-Bound algorithm using Mini-Bucket heuristics (BBMB) are presented and evaluated empirically. They are compared against the full bucket-elimination (whose performance is typical for other complete algorithms such as join-tree clustering), against the Mini-Bucket approximation scheme, and against iterative belief propagation (a recent popular approximation used for coding networks) for the MPE task. They are compared against some local search methods and specialized search algorithms for Max-CSP. The benchmarks are random test problems for Max-CSP, and coding networks, Noisy-OR networks, and CPCS networks when solving the MPE task.

We show that both search methods exploit heuristic's strength in a similar manner: on all problem classes, the optimal trade-off point between heuristic generation and search, as measured by total time, lies in an intermediate range of the heuristic's strength. As problems become larger and harder, this optimal point gradually increases towards the more computationally demanding heuristics. We also show that when Best-First and Branch-and-Bound have access to the same heuristic information, Best-First sometimes substantially outperforms Branch-and-Bound, provided that the heuristics were strong enough, that enough time was given, and that memory problems were not encountered. Sometimes, however, on Max-CSP problems, Branch-and-Bound somewhat outperforms Best-First, especially when provided with accurate heuristics.

Section 2 provides preliminaries and background. Section 3 describes the main idea of the heuristic function which is built on top of the Mini-Bucket algorithm, proves its properties, and embeds the heuristic within Best-First and Branch-and-Bound search. Sections 4 and 5 present empirical evaluations, while Section 6 provides related work and conclusions.

2. Background

2.1. An automated reasoning problem

The approach we propose is applicable to any optimization problem with decomposable cost functions. More formally, we assume that an automated reasoning problem (also called a dependency model) P is specified as a sixtuple $P = \langle X, D, F, \bigotimes, \Downarrow, Z\rangle$, where

(1) $X = \{X_1, \ldots, X_n\}$ is a set of variables.
(2) $D = \{D_1, \ldots, D_n\}$ is a set of finite domains.
(3) $F = \{f_1, \ldots, f_r\}$ is a set of functions or relations. The scope of function f_i, denoted *scope*$(f_i) \subseteq X$, is the set of arguments of f_i.
(4) $\bigotimes_i f_i$ is a *combination* operator defined by $\bigotimes_i f_i \in \{\prod_i f_i, \sum_i f_i, \bowtie_i f_i\}$ over a set of cost functions $\{f_i\}$.

A combination operator takes two cost functions and outputs a new cost function. For a Max-CSP problem, the combination operator is summation, while for Bayesian networks, the combination operator is multiplication. We define the cost function f to be the combination of all functions: $f = \bigotimes_{i=1}^{r} f_i$.

(5) $\Downarrow_Y f$ is a *marginalization* operator, defined by

$$\Downarrow_Y f \in \left\{ \max_{S-Y} f, \min_{S-Y} f, \prod_{S-Y} f, \sum_{S-Y} f \right\},$$

where S is the scope of function f and $Y \subseteq X$. The scope of $\Downarrow_Y f$ is Y.
A marginalization operator takes as input a cost function and generates a new function where arguments other than Y are eliminated. For example, when solving a Max-CSP problem, the marginalization operator is $\Downarrow_Y f(S) = \min_{S-Y} f(S)$. When solving the Most Probable Explanation problem in Bayesian networks, the marginalization operator is $\Downarrow_Y f(S) = \max_{S-Y} f(S)$. [2]

(6) The problem is to compute, $g(Z) = \Downarrow_Z \bigotimes_{i=1}^{r} f_i$, $Z \subseteq X$.
For the optimization tasks we consider here $Z = \emptyset$ and $S = X$. Often we also seek an assignment to all the variables that optimizes (maximizes or minimizes) the combined cost function f. Namely, we need to find $x = (x_1, \ldots, x_n)$ such that $f(x) = \Downarrow_\emptyset \bigotimes_{i=1}^{r} f_i$.

Notations. We denote variables or subsets of variables by uppercase letters (e.g., X, Y, Z, S, $R \ldots$) and values of variables by lower case letters (e.g., x, y, z, s). An assignment $(X_1 = x_1, \ldots, X_n = x_n)$ can be abbreviated as $x = (x_1, \ldots, x_n)$. For a subset of variables S, D_S denotes the Cartesian product of the domains of variables in S. x_S is the projection of $x = (x_1, \ldots, x_n)$ over a subset S.

Definition 2.1 (*Primal graph*). The primal graph of a reasoning problem has the variables X as its nodes and an arc connects any two variables that appear in the scope of the same function.

One approach for solving reasoning problems is searching the problem's search space graph which can be derived from its dependency model. Commonly, the states of the search space are the assignments of values to subsets of variables. The initial state is the empty assignment and a goal state is an assignment to all variables that optimizes the cost function, f. Each state has a subset of child states obtained by an operator that extends a partial assignment by assigning a value to an unassigned variable. Any state, $S = s$, $S \subseteq X$, can be associated with a *cost function* $f(s) = \bigotimes_{f \in F,\ scope(f) \subseteq S} f$. By searching this search space by either depth-first or breadth-first search one can find an assignment to $S = X - Z$ such that $f(s) = \Downarrow_Z \bigotimes_{i=1}^{r} f_i$.

Heuristic search algorithms use a heuristic evaluation function to guide the traversal of the search space, aiming at finding a solution path from the initial state to a goal state. This requires a heuristic evaluation function that estimates the promise of a state in leading

[2] The combination and marginalization operators can be defined axiomatically [41].

to the best cost goal state. In this paper we provide a general approach of extracting such heuristic information from the dependency model of the reasoning problem.

In the following subsections we specialize our definitions for the two dependency models we consider here, constraint networks and belief networks.

2.2. Constraint networks

Constraint satisfaction is a framework for formulating real-world problems as a set of constraints between variables. The task is to find an assignment of values to variables that does not violate any constraint, or else to conclude that the problem is inconsistent. Such problems are graphically represented by nodes corresponding to variables and edges corresponding to constraints between variables.

Definition 2.2 (*Constraint networks, constraint satisfaction problems*). A *constraint network (CN)* is defined by a triplet (X, D, C) where X is a set of variables $X = \{X_1, \ldots, X_n\}$, associated with a set of discrete-valued domains, $D = \{D_1, \ldots, D_n\}$, and a set of constraints $C = \{C_1, \ldots, C_m\}$. Each constraint C_i is a pair (S_i, R_i), where R_i is a relation $R_i \subseteq D_{S_i}$ defined on a subset of variables $S_i \subset X$ called the scope of C_i. The relation denotes all compatible tuples of values of D_{S_i} allowed by the constraint. The primal graph of a constraint network, called a *constraint graph*, has a node for each variable, and an arc between two nodes iff the corresponding variables participate in the same constraint. A *solution* is an assignment of values to variables $x = (x_1, \ldots, x_n)$, $x_i \in D_i$, such that each constraint is satisfied, namely $\forall i \ x_{S_i} \in R_i$. The *constraint satisfaction problem* (CSP) is to determine if a constraint network has a solution, and if so, to find a solution. A binary CSP is one where each constraint involves at most two variables. Sometimes (for the Max-CSP problem), we express the relation R_i as a cost function $C_i(X_{i1} = x_{i1}, \ldots, X_{ik} = x_{ik}) = 0$ if $(x_{i1}, \ldots, x_{ik}) \in R_i$, and 1 otherwise.

Example 2.1. An example of a constraint satisfaction problem is given in Fig. 1. It represents a map coloring problem and has seven variables (A, B, C, D, E, F, G), each one corresponding to a region of the map. Each variable has a domain of three colors (Red,

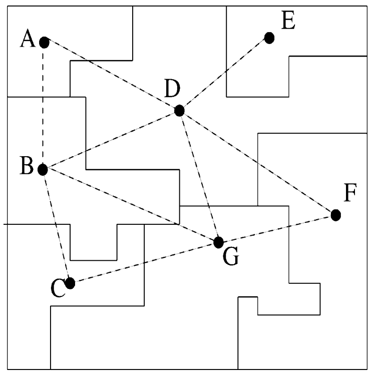


Fig. 1. A map coloring problem.

Green, Blue), and there is an inequality constraint between two variables that correspond to adjacent regions of the map. A solution is an assignment of colors to variables (nodes of the graph) such that adjacent nodes have different colors.

2.3. Max-CSP

Many real-world problems are often over-constrained and do not have a solution. In such cases, it is desirable to find an assignment that satisfies a maximum number of constraints, called a Max-CSP assignment.

Definition 2.3 (*Max-CSP*). Given a CSP, the Max-CSP task is to find an assignment that satisfies a maximum number of constraints.

Although a Max-CSP problem is a maximization problem, it can be implemented as a minimization problem. Instead of maximizing the number of constraints that are satisfied, minimizing the number of constraints that are violated.

When formalized as an automated reasoning task, its set of functions F is the set of cost functions assigning 0 to all allowed tuples and 1 to all non-allowed tuples. The marginalization operator is minimization, the combination operator is summation, and $Z = \emptyset$. Namely, $\Downarrow_{\emptyset} \bigotimes_i f_i = \min_X \sum_i f_i$.

As an optimization version of Constraint Satisfaction, Max-CSP is NP-hard. A number of complete and incomplete algorithms have been developed for Max-CSP. Stochastic Local Search (SLS) algorithms, such as GSAT [28,40], developed for Boolean Satisfiability and Constraint Satisfaction can be directly applied to Max-CSP [45]. Since they are incomplete, SLS algorithms cannot guarantee a solution, but they have been successful in practice on many classes of SAT and CSP problems. A number of search-based complete algorithms, using partial forward checking [9] for heuristic computation, have been developed for Max-CSP [22,44].

2.4. Belief networks

Belief networks provide a formalism for reasoning about partial beliefs under conditions of uncertainty. They are defined by a directed acyclic graph over nodes representing random variables of interest (e.g., the temperature of a device, the gender of a patient, a feature of an object, the occurrence of an event). The arcs signify the existence of direct causal influences between linked variables quantified by conditional probabilities that are attached to each cluster of parents-child nodes in the network.

Definition 2.4 (*Graph concepts*). A *directed graph* is a pair, $G = \{V, E\}$, where $V = \{X_1, \ldots, X_n\}$ is a set of nodes, and $E = \{(X_i, X_j) \mid X_i, X_j \in V\}$ is the set of edges. If $(X_i, X_j) \in E$, we say that X_i *points to* X_j. The degree of a variable is the number of edges incident to it. For each variable X_i, $pa(X_i)$ or pa_i, is the set of variables pointing to X_i in G, while the set of child nodes of X_i, denoted $ch(X_i)$, comprises the variables that X_i points to. The family of X_i, F_i, includes X_i and its child variables. A directed graph is acyclic if it has no directed cycles. A *poly-tree* is an acyclic directed graph whose underlying undirected graph (ignoring the arrows) has no loops.

Definition 2.5 (*Belief networks*). Given a set, $X = \{X_1, \ldots, X_n\}$ of random variables over multi-valued domains $D = \{D_1, \ldots, D_n\}$, a belief network is a pair (G, P) where G is a directed acyclic graph over X and $P = \{P_i\}$, where $P_i = \{P(X_i \mid pa(X_i))\}$ are conditional probability matrices associated with each X_i. Given a subset of variables S, we will write $P(s)$ for the probability $P(S = s)$, where $s \in D_S$. A belief network represents a probability distribution over X, $P(x_1, \ldots, x_n) = \prod_{i=1}^{n} P(x_i \mid x_{pa(X_i)})$. An evidence set e is an instantiated subset of variables. The primal graph of a belief network is called a moral graph. It can be obtained by connecting the parents of each node in G and removing the arrows. Equivalently, it connects any two variables appearing in the same family.

Definition 2.6 (*Induced width*). An *ordered graph* is a pair (G, d) where G is an undirected graph, and $d = (X_1, \ldots, X_n)$ is an ordering of the nodes. The *width of a node* in an ordered graph is the number of its earlier neighbors. The *width of an ordering* d, $w(d)$, is the maximum width over all nodes. The *induced width of an ordered graph*, $w^*(d)$, is the width of the induced ordered graph obtained by processing the nodes recursively, from last to first; when node X is processed, all its earlier neighbors are connected.

Example 2.2. An example of a belief network is given in Fig. 2(a). This belief network represents a distribution

$$P(e, d, c, b, a) = P(e \mid c, b)P(d \mid b, a)P(b \mid a)P(c \mid a)P(a).$$

In this case, $pa(E) = \{B, C\}$, $pa(B) = \{A\}$, $pa(A) = \emptyset$, $ch(A) = \{B, D, C\}$. Its corresponding moral graph is shown in Fig. 2(b). Given an ordering $d = (A, E, D, C, B)$, the ordered moral graph is depicted in Fig. 2(c) by the solid arcs. The induced ordered graph is obtained by adding the broken arcs in the figure. The width and induced width of the ordered moral graph is 4.

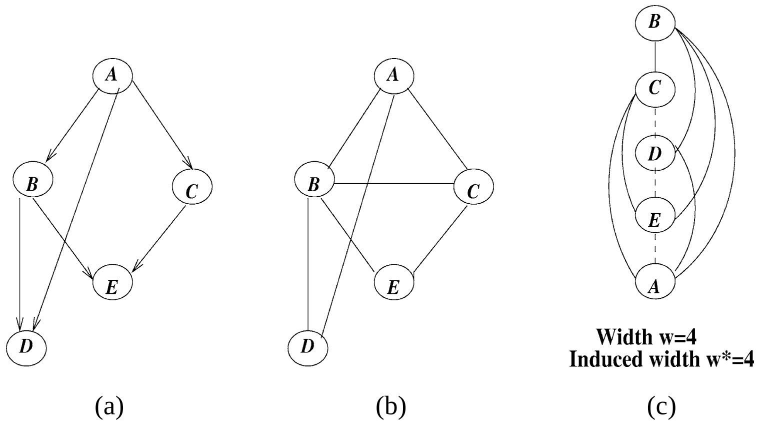


Fig. 2. (a) A belief network $P(e, d, c, b, a) = P(e \mid c, b)P(d \mid b, a)P(b \mid a)P(c \mid a) \cdot P(a)$, (b) its moral graph, (c) an ordered graph along $d = (A, E, D, C, B)$.

Definition 2.7. Given a function h defined over a subset of variables S, where $X \in S$, functions $(\min_X h)$, $(\max_X h)$, and $(\sum_X h)$ are defined over $U = S - \{X\}$ as follows: For every $U = u$, and denoting by (u, x) the extension of tuple u by assignment $X = x$, $(\min_X h)(u) = \min_x h(u, x)$, $(\max_X h)(u) = \max_x h(u, x)$, and $(\sum_X h)(u) = \sum_x h(u, x)$. Given a set of functions $h_1, \ldots, h_j$ defined over the subsets $S_1, \ldots, S_j$, the product function $\prod_j h_j$ and $\sum_j h_j$ are defined over $U = \bigcup_j S_j$. For every $U = u$, $(\prod_j h_j)(u) = \prod_j h_j(u_{S_j})$, and $(\sum_j h_j)(u) = \sum_j h_j(u_{S_j})$.

2.5. The most probable explanation

Definition 2.8 (*Most probable explanation*). Given a belief network and evidence e, the Most Probable Explanation (MPE) task is to find a complete assignment which agrees with the available evidence, and which has the highest probability among all such assignments, namely, to find an assignment $(x_1^o, \ldots, x_n^o)$ such that

$$\begin{aligned} P(x_1^o, \ldots, x_n^o) &= \max_{x_1,\ldots,x_n} \prod_{k=1}^{n} P(x_1, \ldots, x_n, e) \\ &= \max_{x_1,\ldots,x_n} \prod_{k=1}^{n} P(x_k, e \mid x_{pa_k}). \end{aligned}$$

When MPE is formalized as an automated reasoning task, the combination operator is multiplication and the marginalization operator is maximization. An MPE task is to find $\Downarrow_{\emptyset} \bigotimes_i f_i = \max_X \prod_i f_i$ where X is the set of variables and f_i is the set of conditional probability tables. It also requires an optimizing assignment.

The MPE task appears in applications such as diagnosis, abduction, and explanation. For example, given data on clinical findings, MPE can postulate on a patient's probable affliction. In decoding, the task is to identify the most likely input message transmitted over a noisy channel given the observed output. Researchers in natural language consider the understanding of text to consist of finding the most likely facts (in internal representation) that explain the given text. In computer vision and image understanding, researchers formulate the problem in terms of finding the most likely set of objects that explain the image.

It is known that solving the MPE task is NP-hard [1]. Complete algorithms use either the *cycle cutset* technique (also called *conditioning*), the *join-tree-clustering* technique [32], or the bucket-elimination scheme [3]. However, these methods work well only if the network is sparse enough to allow small cutsets or small clusters. The complexity of algorithms based on the cycle cutset idea is time exponential in the cutset size but requires only linear space. The complexity of join-tree-clustering and bucket-elimination algorithms are both time and space exponential in the cluster size that equals the *induced width* of the network's moral graph. Following Pearl's stochastic simulation algorithms [32], the suitability of Stochastic Local Search (SLS) algorithms for MPE was studied in the context of medical diagnosis applications [33] and more recently in [17]. Best-First search algorithms were proposed [42] as well as algorithms based on linear programming [37]. Various authors have worked on extending some of these algorithms to the task of finding the k most-likely explanations [24,43].

2.6. Bucket and Mini-Bucket elimination algorithms

In this subsection we summarize the main algorithms that are used as a basis for our methodology for heuristic generation. These are based on the bucket elimination scheme.

Bucket elimination is a unifying algorithmic framework for dynamic-programming algorithms applicable to probabilistic and deterministic reasoning [4]. Many algorithms for probabilistic inference, such as belief updating, finding the most probable explanation, finding the maximum a posteriori hypothesis, and calculating the maximum expected utility, as well as algorithms for constraint optimization, such as Max-CSP, can be expressed as bucket-elimination algorithms [3].

The input to a bucket-elimination algorithm is an automated reasoning dependency model, namely, a collection of functions or relations (e.g., clauses for propositional satisfiability, constraints or cost functions, or conditional probability matrices for belief networks). Given a variable ordering, the algorithm partitions the functions into buckets, each associated with a single variable. A function is placed in the bucket of its argument that appears latest in the ordering. The algorithm has two phases. During the first, top-down phase, it processes each bucket, from last to first by a variable elimination procedure that computes a new function which is placed in a lower bucket. For MPE, the variable elimination procedure computes the product of all probability matrices and maximizes over the bucket's variable. For Max-CSP, this procedure computes the sum of all cost functions and minimizes over the bucket's variable. During the second, bottom-up phase, the algorithm constructs a solution by assigning a value to each variable along the ordering, consulting the functions created during the top-down phase. Fig. 3 shows the bucket-elimination algorithm *BE* [3]. It can be shown that

Theorem 2.3 (Dechter [3]). *The time and space complexity of bucket elimination applied along order d is* $\mathrm{O}(r \cdot \exp(w^*(d)+1))$ *and* $\mathrm{O}(n \cdot \exp(w^*(d)))$ *respectively, where $w^*(d)$ is the induced width of the network's ordered primal graph along the ordering d, r is the number of functions, and n is the number of variables.*

The main drawback of bucket elimination algorithms is that they require too much space for storing intermediate functions. *Mini-Bucket Elimination* is an approximation designed to avoid the space and time problem of full bucket elimination [7] by partitioning large buckets into smaller subsets called mini-buckets which are processed independently. Here is the rationale. Let $h_1, \ldots, h_j$ be the functions in *bucket*$_p$. When *Bucket Elimination* processes *bucket*$_p$, it computes the function h^p: $h^p = \Downarrow_{U_p} \bigotimes_{i=1}^{j} h_i$, where

$$U_p = \bigcup_{i=1}^{j} S_i - \{X_p\}.$$

The Mini-Bucket algorithm, on the other hand, creates a partitioning of the bucket into mini-buckets $Q' = \{Q_1, \ldots, Q_t\}$ where the mini-bucket Q_l contains the functions

```
Algorithm BE
Input: A problem description $P = \langle X, D, F, \bigotimes, \Downarrow, \emptyset\rangle$; an ordering of the variables $d$.
Output: An assignment corresponding to an optimal solution.
1. Initialize: Partition the functions in $F$ into $bucket_1, \ldots, bucket_n$,
   where $bucket_i$ contains all functions whose highest variable is $X_i$. Let
   $S_1, \ldots, S_j$ be the scopes of functions (original and intermediate) in the
   processed bucket.
2. Backward: For $p \leftarrow n$ down-to 1, do
   for $h_1, h_2, \ldots, h_j$ in $bucket_p$, do
     • If variable $X_p$ is instantiated ($X_p = x_p$), assign $X_p = x_p$ to each $h_i$
       and put each resulting function into its appropriate bucket.
     • Else, generate the function $h^p$: $h^p = \Downarrow_{U_p} \bigotimes_{i=1}^{j} h_i$, where
       $U_p = \bigcup_{i=1}^{j} S_i - \{X_p\}$. Add $h^p$ to the bucket of the largest-index
       variable in $U_p$.
3. Forward: Assign a value to each variable in the ordering $d$ such that
   the combination of functions in each bucket is optimized.
4. Return the function computed in the bucket of the first variable and
   the optimizing assignment.
```

Fig. 3. Bucket elimination algorithm.

$h_{l_1}, \ldots, h_{l_k}$. The approximation processes each mini-bucket (by using the combination and marginalization operators) separately, therefore computing $g^p = \bigotimes_{l=1}^{t} \Downarrow_{U_p} \bigotimes_{l_i} h_{l_i}$. Clearly, g^p is a bound on h^p—for maximization problems $h^p \leqslant g^p$; for minimization problems $h^p \geqslant g^p$. Therefore, the bound computed in each bucket yields an overall bound on the cost of the solution.

The quality of the bound depends on the degree of the partitioning into mini-buckets. Given a bounding parameter i, the algorithm creates an i-partitioning, where each mini-bucket includes no more than i variables. Algorithm MBE(i),[3] described in Fig. 4, is parameterized by this i-bound. The algorithm outputs not only a bound on the cost of the optimal solution and an assignment, but also the collection of augmented buckets. By comparing the bound computed by MBE(i) to the cost of the assignment output by MBE(i), we can always have an interval bound on the error for the given instance. For example, if $\Downarrow = max$, MBE(i) provides an upper-bound on the optimal assignment in its first bucket, while the cost of the assignment generated yields a lower bound.

The algorithm's complexity is time and space O(exp(i)) where $i \leqslant n$. When the bound i is large enough (i.e., when $i \geqslant w^*$), the Mini-Bucket algorithm coincides with the full bucket elimination. In summary,

Theorem 2.4 (Dechter and Rish [7]). *Algorithm* MBE(i) *generates an interval bound on the cost of the optimal solution, and its time and space complexity are* $O(r \cdot \exp(i))$ *and* $O(r \cdot \exp(i-1))$ *respectively, where r is the number of functions.*

[3] In the original paper this algorithm was called Approx-MPE.

```
Algorithm MBE($(i)$)
Input: A problem description $P = \langle X, D, F, \bigotimes, \Downarrow, \emptyset \rangle$; an ordering of
the variables $d$; parameter i, $f = \bigotimes_{i=1}^{r} f_i$.
Output: A bound on $\Downarrow_{\emptyset} f$, the cost of the optimal solution; an assignment
to all the variables; and the ordered augmented buckets.
1. Initialize: Partition the functions in $F$ into $bucket_1, \ldots, bucket_n$,
   where $bucket_i$ contains all functions whose highest variable is $X_i$. Let
   $S_1, \ldots, S_j$ be the scopes of functions (original and intermediate) in the
   processed bucket.
2. Backward For $p \leftarrow n$ down-to 1, do
   • If variable $X_p$ is instantiated ($X_p = x_p$), assign $X_p = x_p$ to each $h_i$
     and put each resulting function into its appropriate bucket.
   • Else, for $h_1, h_2, \ldots, h_j$ in $bucket_p$, generate an $(i)$-partitioning,
     $Q' = \{Q_1, \ldots, Q_t\}$. For each $Q_l \in Q'$ containing $h_{l_1}, \ldots, h_{l_t}$ generate
     function $h^l$, $h^l = \Downarrow_{U_l} \bigotimes_{i=1}^{t} h_{l_i}$, where $U_l = \bigcup_{i=1}^{j} scope(h_{l_i}) - \{X_p\}$
     Add $h^l$ to the bucket of the largest-index variable in $U_l$.
3. Forward For $i = 1$ to $n$ do, given $x_1, \ldots, x_{p-1}$ choose a value $x_p$ of
   $X_p$ that optimizes the combination of all the functions in $X_p$'s bucket.
4. Return the ordered set of augmented buckets, an assignment
   $\bar{x} = (x_1, \ldots, x_n)$, an interval bound (the value computed in $bucket_1$ and
   the cost $f(\bar{x})$).
```

Fig. 4. Mini-Bucket elimination algorithm.

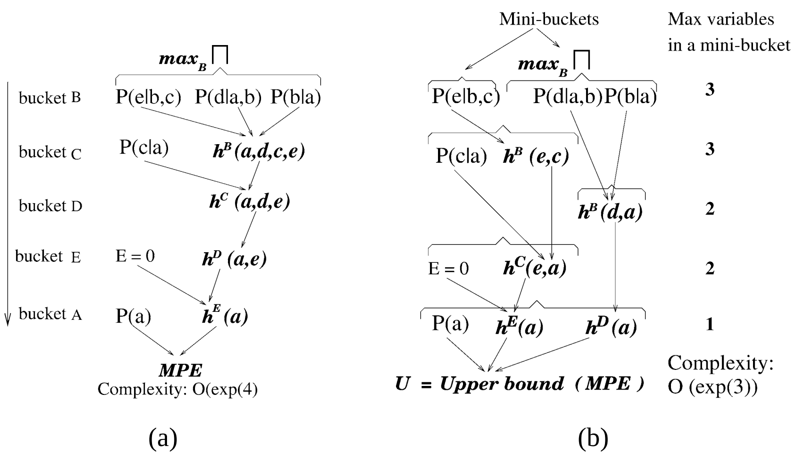


Fig. 5. Execution of BE and MBE(i). (a) A trace of BE. (b) A trace of MBE(3).

Example 2.5. Fig. 5 illustrates how algorithms BE and MBE(i) for $i = 3$ process the network in Fig. 2(a) along the ordering (A, E, D, C, B) and assumes the MPE task. Algorithm BE records the new functions $h^B(a, d, c, e)$, $h^C(a, d, e)$, $h^D(a, e)$, and $h^E(a)$. Then, in the bucket of A, it computes the probability of the MPE, $P_{t\mathrm{MPE}} =$

$\max_a P(a)h^E(a)$. Subsequently, an MPE assignment ($A = a'$, $B = b'$, $C = c'$, $D = d'$, $E = 0$) ($E = 0$ is an evidence) is computed for each variable from A to B by selecting a value that optimizes the product of functions in the corresponding bucket, conditioned on the previously assigned values. Namely, $a' = \arg\max_{a \in D_A} P(a)h^E(a)$, $e' = 0$, $d' = \arg\max_{d \in D_D} h^C(a', d, e')$, and so on. The approximation MBE(3) splits bucket B into two mini-buckets, each containing no more than 3 variables, and generates $h^B(e, c)$ and $h^B(d, a)$. An upper bound on the MPE value is computed by *upper-bound* = $\max_{a \in D_A} P(a) \cdot h^E(a) \cdot h^D(a)$. A suboptimal tuple is computed by MBE(i) similarly to the MPE tuple computed by BE, by assigning a value to each variable that maximizes the product of functions in the corresponding bucket, given the assignments to the previous variables. The value of this assignment is a lower bound on the MPE value.

3. Heuristic search with Mini-Bucket heuristics

By comparing the bound computed by MBE(i) with the cost of the assignment returned by MBE(i), we can bound the error of the assignment. If the error is large, we can increase i. Encouraging empirical results were reported for computing the MPE on randomly generated noisy-or networks, on medical-diagnosis CPCS networks, and on coding problems [7,36]. In some cases, however, the approximation was largely suboptimal, even when using the highest feasible accuracy level. Such cases call for finding a better solution by using heuristic search, such as Branch-and-Bound or Best-First search, which is the approach we explore in this paper.

A heuristic search algorithm explores the search space of partial assignments guided by a heuristic evaluation function. The heuristic function estimates the cost of the optimal completion of every partial assignment to a full assignment.

Branch-and-Bound searches the space of partial assignments in a depth-first manner. It discards any partial assignment whose heuristic value is not better than the value of the current best solution. The algorithm requires only a limited amount of memory and can be used as an anytime scheme; whenever interrupted, Branch-and-Bound outputs the best solution found so far. Best-First explores the search space in a breadth-first manner, expanding a partial assignment with the best heuristic value first. Best-First is optimal in terms of the number of nodes expanded, while it needs exponential space in the worst case.

The effectiveness of both search methods depends on the quality of the heuristic function. A heuristic function that is accurate, but not too hard to compute, is desirable. In the following section we will show how to define a heuristic function that can guide Branch-and-Bound or Best-First search, using the augmented buckets generated by the MBE(i) algorithm. We will use the task of finding the Most Probable Explanation in a Bayesian network to illustrate the idea and then proceed to the general case.

3.1. Mini-Bucket heuristic idea

Consider the Bayesian network shown in Fig. 2, and consider a given variable ordering $d = (A, E, D, C, B)$ and the bucket and mini-buckets configuration in the output, as displayed in Fig. 5. Let's assume, without losing generality, that variables A, E and D

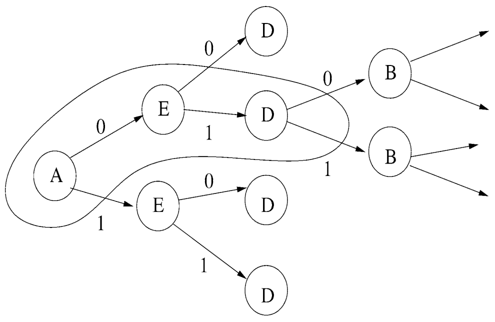


Fig. 6. Search space for $f^*(a,e,d)$.

have been instantiated during search (see Fig. 6, $a=0$, $e=1$, $d=1$). Let $f^*(a,e,d)$ be the probability of the best completion of the partial assignment $(A=a, E=e, D=d)$. By definition,

$$\begin{aligned} f^*(a,e,d) &= \max_{b,c} P(a,b,c,d,e) \\ &= P(a)\cdot \max_{b,c} P(c\mid a)P(e\mid b,c)P(b\mid a)P(d\mid a,b) \\ &= g(a,e,d)\cdot h^*(a,e,d), \end{aligned} \tag{1}$$

where

$$g(a,e,d)=P(a)$$

and

$$h^*(a,e,d)=\max_{b,c} P(c\mid a)P(e\mid b,c)P(b\mid a)P(d\mid a,b).$$

We can derive:

$$\begin{aligned} h^*(a,e,d) &= \max_{b,c} P(c\mid a)P(e\mid b,c)P(b\mid a)P(d\mid a,b) \\ &= \max_{c} P(c\mid a)\cdot \max_{b} P(e\mid b,c)P(b\mid a)P(d\mid a,b) \\ &= \max_{c} P(c\mid a)\cdot h^B(a,d,c,e) \\ &= h^C(a,d,e), \end{aligned} \tag{2}$$

where

$$h^B(a,d,c,e)=\max_{b} P(e\mid b,c)P(b\mid a)P(d\mid a,b)$$

and

$$h^C(a,d,e)=\max_{c} P(c\mid a)\cdot h^B(a,d,c,e).$$

Interestingly, the functions $h^B(a,d,c,e)$ and $h^C(a,d,e)$ are already produced by the bucket elimination algorithm BE (Fig. 5(a)). Specifically, the function $h^B(a,d,c,e)$, generated in *bucket*$_B$, is the result of a maximization operation over variable B. In practice, however, this function may be too hard to compute as it requires processing a function on five variables and recording a function on four variables. So, it can be replaced by an approximation, where the maximization is split into two parts. This yields a function, which we denote $h(a,e,d)$, that is an upper bound on $h^*(a,e,d)$ defined as follows:

$$\begin{aligned}
h^*(a,e,d) &= \max_c P(c \mid a) \cdot \max_b P(e \mid b,c) P(b \mid a) P(d \mid a,b) \\
&\leqslant \max_c P(c \mid a) \cdot \max_b P(e \mid b,c) \cdot \max_b P(b \mid a) P(d \mid a,b) \\
&= \max_c P(c \mid a) \cdot h^B(e,c) \cdot h^B(d,a) \\
&= h^B(d,a) \cdot \max_c P(c \mid a) \cdot h^B(e,c) \\
&= h^B(d,a) \cdot h^C(e,a) \\
&= h(a,e,d),
\end{aligned}$$

where we define

$$\begin{aligned}
h^B(e,c) &= \max_b P(e \mid b,c) \\
h^B(d,a) &= \max_b P(b \mid a) \cdot P(d \mid a,b) \\
h^C(e,a) &= \max_c P(c \mid a) \cdot h^B(e,c).
\end{aligned}$$

Notice now that the functions $h^B(e,c)$, $h^B(d,a)$ and $h^C(e,a)$ were already computed by the Mini-Bucket algorithm MBE(i) (Fig. 5(b)). Using the upper-bound function $h(a,e,d)$, we can now define a function $f(a,e,d)$ that provides an upper bound on the exact value $f^*(a,e,d)$. Namely, replacing $h^*(a,e,d)$ by $h(a,e,d)$ in $f^*(a,e,d)$ in Eq. (1) we get

$$f(a,e,d) = g(a,e,d) \cdot h(a,e,d) \geqslant f^*(a,e,d).$$

3.2. Mini-Bucket heuristic for MPE

In the following, we will assume that a Mini-Bucket algorithm was applied to a belief network using a given variable ordering $d = (X_1, \ldots, X_n)$, and that the algorithm outputs an ordered set of augmented buckets *bucket*$_1, \ldots,$ *bucket*$_p, \ldots,$ *bucket*$_n$, containing both the input functions and the newly generated functions. Relative to such an ordered set of augmented buckets, we use the following notations:

- P_{p_j} denotes an input conditional probability matrix in *bucket*$_p$ (namely, one whose highest-ordered variable is X_p), enumerated by j.
- h_{p_j} denotes a function residing in *bucket*$_p$ that was generated by the Mini-Bucket algorithm, enumerated by j.
- h_j^p stands for a function created by processing the jth mini-bucket in *bucket*$_p$.
- λ_{p_j} stands for an arbitrary function in *bucket*$_p$, enumerated by j. Notice that $\{\lambda_{p_j}\} = \{P_{p_j}\} \cup \{h_{p_j}\}$.

We denote by *buckets*$(1..p)$ the union of all functions in the bucket of X_1 through the bucket of X_p.

Example 3.1. In Fig. 5(b), *bucket*$_C$ (the bucket corresponding to variable C) contains two functions: $P_{C_1} = P(c \mid a)$ and $h_{C_1} = \lambda_{C_1} = h^B(e,c)$. When *bucket*$_C$ is processed, a new function $h_1^C = h^C(e,a)$ is generated. This new function is then placed in the bucket of variable E. Following the notation introduced above, $h_{E_1} = h^C(e,a)$, and it can also be denoted as λ_{E_1}.

We will now show that in general the functions recorded by the Mini-Bucket algorithm can be used to upper bound the probability of the most probable extension of any partial assignment, and therefore can serve as heuristic evaluation functions in a *Best-First* or *Branch-and-Bound* search.

Definition 3.1 (*Exact evaluation function*). Let $\bar{x}^p = (x_1, \ldots, x_p)$ be an assignment to the first p variables. The probability of the most probable extension of $\bar{x}^p$, denoted $f^*(\bar{x}^p)$ is defined by

$$f^*(\bar{x}^p) = \max_{x_{p+1},\ldots,x_n} \prod_{k=1}^{n} P(x_k \mid x_{pa_k}).$$

The above product defining f^* can be divided into two smaller products expressed by the functions in the ordered augmented buckets. In the first product all the arguments are instantiated (belong to $x_1, \ldots, x_p$), and therefore the maximization operation is applied to the second product only. Denoting

$$g(\bar{x}^p) = \left(\prod_{P_i \in \mathit{buckets}(1..p)} P_i \right)(\bar{x}^p)$$

and

$$h^*(\bar{x}^p) = \max_{(x_{p+1},\ldots,x_n)} \left(\prod_{P_i \in \mathit{buckets}(p+1..n)} P_i \right)(\bar{x}^p, x_{p+1}, \ldots, x_n)$$

we get

$$f^*(\bar{x}^p) = g(\bar{x}^p) \cdot h^*(\bar{x}^p).$$

During search, the g function can be evaluated over the partial assignment $\bar{x}^p$, while h^* can be estimated by a heuristic function h, derived from the functions recorded by the Mini-Bucket algorithm, as defined next:

Definition 3.2 (*Mini-Bucket heuristic*). Given an ordered set of augmented buckets generated by the Mini-Bucket algorithm, the heuristic function $h(\bar{x}^p)$ is defined as the product of all the h_j^k functions that satisfy the following two properties:

(1) they are generated in buckets $p+1$ through n, and
(2) they reside in buckets 1 through p. Namely, $h(\bar{x}^p) = \prod_{i=1}^{p} \prod_{h_j^k \in \mathit{bucket}_i} h_j^k$, where $k > p$ (i.e., h_j^k is generated by a bucket processed before *bucket*$_p$).

Example 3.2. In Fig. 5(b), the buckets of variables A, E and D contain a total of 4 functions generated by the Mini-Bucket algorithm: $h^B(d,a)$, $h^C(e,a)$, $h^E(a)$ and $h^D(a)$. However, when computing the heuristic functions $h(a,e,d)$, only $h^B(d,a)$ and $h^C(e,a)$ are used, yielding: $h(a,e,d) = h^B(d,a) \cdot h^C(e,a)$, because $h^E(a)$ and $h^D(a)$ were already computed in buckets D and E from $h^C(e,a)$ and $h^B(d,a)$, respectively.

The following proposition shows how $g(\bar{x}^p)$ and $h(\bar{x}^p)$ can be updated recursively.

Proposition 3.1. *Given a partial assignment $\bar{x}^p = (x_1,\ldots,x_p)$, both $g(\bar{x}^p)$ and $h(\bar{x}^p)$ can be computed recursively by*

$$g(\bar{x}^p) = g(\bar{x}^{p-1}) \cdot \prod_j P_{p_j}(\bar{x}^p), \tag{3}$$

$$h(\bar{x}^p) = h(\bar{x}^{p-1}) \cdot \frac{\prod_k h_{p_k}(\bar{x}^p)}{\prod_j h_j^p(\bar{x}^p)}. \tag{4}$$

Proof. Following Definitions 3.1 and 3.2, $g(\bar{x}^p)$ is simply the product of all input probabilities that are in buckets $1,\ldots,p$. Heuristic function $h(\bar{x}^p)$ must contain all new functions that are in buckets $1,\ldots,p$, and that were generated in buckets $p+1,\ldots,m$. Since $h(\bar{x}^{p-1})$ already contains all new functions in buckets $1,\ldots,p-1$ that were generated by buckets $p,\ldots,n$, we need to factor out all new functions that were generated in bucket p, and multiply the result with all new functions in bucket p (which must be generated in buckets $p+1,\ldots,n$), yielding $h(\bar{x}^p)$. □

Theorem 3.3. *For every partial assignment $\bar{x}^p = (x_1,\ldots,x_p)$, of the first p variables, the evaluation function $f(\bar{x}^p) = g(\bar{x}^p) \cdot h(\bar{x}^p)$ is:*
(1) *admissible—it never underestimates the probability of the best extension of $\bar{x}^p$, and*
(2) *monotonic—namely $f(\bar{x}^{p+1}) \leqslant f(\bar{x}^p)$.*

Notice that monotonicity means better accuracy at deeper nodes in the search tree.

Proof. To prove monotonicity, we will use the recursive equations (3) and (4) from Proposition 3.1. For any $\bar{x}^p$ and any value v in the domain of X_{p+1}, we have by definition

$$\begin{aligned}\frac{f(\bar{x}^p,v)}{f(\bar{x}^p)} &= \frac{g(\bar{x}^p,v)\cdot h(\bar{x}^p,v)}{g(\bar{x}^p)\cdot h(\bar{x}^p)}\\ &= \frac{(g(\bar{x}^p)\cdot \prod_j P_{(p+1)_j}(\bar{x}^p,v))\cdot \bigl(h(\bar{x}^p)\cdot \frac{\prod_k h_{(p+1)_k}(\bar{x}^p,v)}{\prod_j h_j^{p+1}(\bar{x}^p)}\bigr)}{g(\bar{x}^p)\cdot h(\bar{x}^p)}\\ &= \frac{\prod_j P_{(p+1)_j}(\bar{x}^p,v)\cdot \prod_k h_{(p+1)_k}(\bar{x}^p,v)}{\prod_j h_j^{p+1}(\bar{x}^p)}.\end{aligned}$$

Since $\{P_{(p+1)_j}\} \cup \{h_{(p+1)_k}\} = \{\lambda_{(p+1)_i}\}$, we get

$$= \frac{\prod_i \lambda_{(p+1)_i}(\bar{x}^p,v)}{\prod_j h_j^{p+1}(\bar{x}^p)}.$$

Since $h_j^{p+1}(\bar{x}^p)$ is computed by the jth mini-bucket in bucket $p+1$ by maximizing over variable X_{p+1}, (eliminating variable X_{p+1}), we get

$$\prod_i \lambda_{(p+1)_i}(\bar{x}^p, v) \leqslant \prod_j h_j^{p+1}(\bar{x}^p).$$

Thus, $f(\bar{x}^p, v) \leqslant f(\bar{x}^p)$, concluding the proof of monotonicity.

The proof of admissibility follows from monotonicity. It is well known that if a heuristic function is monotone and if it is exact for a full solution (which is the case here, since the heuristic is the constant 1 on a full solution), then it is also admissible [31]. □

In the extreme case when each bucket p contains exactly one mini-bucket, the heuristic function h equals h^*, and the heuristic function f computes the exact probability of the MPE extension of the current partial assignment.

3.3. Mini-Bucket heuristic for the general case

We will now extend this approach further. Mini-Bucket elimination can be used to generate a heuristic function for any optimization problem $P = \langle X, D, F, \bigotimes, \Downarrow, \emptyset\rangle$ with decomposable cost functions $f = \bigotimes_{i=1}^r f_i$. Next we extend Definition 3.3, Proposition 3.1 and Theorem 3.3 to a general optimization task.

Definition 3.3 (*Mini-Bucket heuristic*). Given an ordered set of augmented buckets generated by the Mini-Bucket algorithm MBE(i), and given a partial assignment $\bar{x}^p$, an evaluation function $f(\bar{x}^p) = g(\bar{x}^p) \bigotimes h(\bar{x}^p)$ is defined as follows:

(1) $g(\bar{x}^p) = (\bigotimes_{f_i \in\, buckets(1..p)} f_i)(\bar{x}^p)$ is the combination of all functions that are fully instantiated.

(2) The heuristic function $h(\bar{x}^p)$ is defined as the combination of all the h_j^k functions that satisfy the following two properties:
- they are generated in buckets $p+1$ through n,
- they reside in buckets 1 through p.

Namely, $h(\bar{x}^p) = \bigotimes_{i=1}^p \bigotimes_{h_j^k \in bucket_i} h_j^k$, where $k > p$.

Proposition 3.2. *Given a partial assignment $\bar{x}^p = (x_1, \ldots, x_p)$, both $g(\bar{x}^p)$ and $h(\bar{x}^p)$ can be computed recursively by*

$$g(\bar{x}^p) = g(\bar{x}^{p-1}) \bigotimes \left(\bigotimes_j f_{p_j}(\bar{x}^p) \right), \tag{5}$$

$$h(\bar{x}^p) = h(\bar{x}^{p-1}) \bigotimes \left[\left(\bigotimes_k h_{p_k}(\bar{x}^p) \right) \bigotimes^{-1} \left(\bigotimes_j h_j^p(\bar{x}^p) \right) \right], \tag{6}$$

where $\bigotimes^{-1}$ is the inverse of the combination operator $\bigotimes$.

Theorem 3.4. *For every partial assignment $\bar{x}^p = (x_1, \ldots, x_p)$ of the first p variables, the evaluation function $f(\bar{x}^p) = g(\bar{x}^p) \bigotimes h(\bar{x}^p)$ is admissible and monotonic.*

Next, we show how the heuristic function is incorporated in search for a general optimization task.

3.4. Search with Mini-Bucket heuristics

The tightness of the bound generated by the Mini-Bucket approximation depends on its i-bound. Larger values of i generally yield better bounds, but require more computation. Since the Mini-Bucket algorithm is parameterized by i, when using the heuristics in each of the search methods, we get an entire class of Branch-and-Bound search and Best-First search algorithms that are parameterized by i and which allow a controllable trade-off between preprocessing and search, or between heuristic strength and its overhead.

Figs. 7 and 8 present algorithms BBMB(i) (Branch-and-Bound with Mini-Bucket heuristics) and BFMB(i) (Best-First search with Mini-Bucket heuristics). Both algorithms have a preprocessing step of running the Mini-Bucket algorithm that produces a set of ordered augmented buckets.

Branch-and-Bound with Mini-Bucket heuristics (BBMB(i)) traverses the search space in a depth-first manner, instantiating variables from first to last. Throughout the search, the algorithm maintains a global bound on the cost of the optimal solution, which corresponds to the cost of the best full variable instantiation found thus far. When the

Algorithm BBMB(i)
Input: A problem description $P = \langle X, D, F, \bigotimes, \Downarrow, \emptyset\rangle$; ordering d; time bound t.
Output: An optimal assignment, or a bound and a (suboptimal) assignment (produced by MBE(i)).
1. **Initialize:** Run MBE(i) algorithm generating a set of ordered augmented buckets and a bound on the optimal cost. Initialize global bound B to the worst value (for minimization problems ∞, for maximization problems 0). Set p to 0.
2. **Search:** Execute the following procedure until variable X_1 has no legal values left or until out of time, in which case output the current best solution.
• **Expand:** Given a partial instantiation $\bar{x}^p$, compute all partial assignments $\bar{x}^{p+1} = (\bar{x}^p, v)$ for each value v of X_{p+1}. For each node $\bar{x}^{p+1}$ compute its heuristic value $f(\bar{x}^{p+1}) = g(\bar{x}^{p+1}) \bigotimes h(\bar{x}^{p+1})$ using Eqs. (5) and (6). Discard those assignments $\bar{x}^{p+1}$ for which $f(\bar{x}^{p+1})$ is not better than the global bound B. Add the remaining assignments to the search tree as children of $\bar{x}^p$.
• **Forward:** If X_{p+1} has no legal values left, goto Backtrack. Otherwise let $\bar{x}^{p+1} = (\bar{x}^p, v)$ be the best extension to $\bar{x}^p$ according to f. If $p + 1 = n$, then set $B = f(\bar{x}^n)$ and goto Backtrack. Otherwise remove v from the list of legal values. Set $p = p + 1$ and goto Expand.
• **Backtrack:** If $p = 1$, Exit. Otherwise set $p = p - 1$ and repeat the Forward step.

Fig. 7. Algorithm Branch-and-Bound with MBE(i).

```
Algorithm BFMB(i)
Input: A problem description $P = \langle X, D, F, \bigotimes, \Downarrow, \emptyset\rangle$; ordering $d$; time bound $t$.
Output: An optimal assignment, or a bound and a (suboptimal) assignment (produced by MBE($i$)).
1. Initialize: Run MBE($i$) algorithm generating a set of ordered augmented buckets and a bound on the optimal cost. Insert a dummy node $\bar{x}_0$ in the set $L$ of open nodes.
2. Search:
• If out of time, output Mini-Bucket assignment.
• Select and remove a node $\bar{x}^p$ with the best heuristic value $f(\bar{x}^p)$ from the set of open nodes $L$.
• If $p = n$ then $\bar{x}^p$ is an optimal solution. Exit.
• Expand $\bar{x}^p$ by computing all child nodes $(\bar{x}^p, v)$ for each value $v$ in the domain of $X_{p+1}$. For each node $\bar{x}^{p+1}$ compute its heuristic value $f(\bar{x}^{p+1}) = g(\bar{x}^{p+1}) \bigotimes h(\bar{x}^{p+1})$, using Eqs. (5) and (6).
• Add all nodes $(\bar{x}^p, v)$ to $L$ and goto Search.
```

Fig. 8. Algorithm Best-First search with MBE(i).

algorithm processes variable X_p, all the variables preceding X_p in the ordering are already instantiated, so it can compute $f(\bar{x}^{p-1}, X_p = v) = g(\bar{x}^{p-1}, v) \bigotimes h(\bar{x}^p, v)$ for each extension $X_p = v$. The algorithm prunes all values v whose heuristic estimate $f(\bar{x}^p, X_p = v)$ is not better (that is—not greater for maximization problems; not smaller for minimization problems) than the current global bound, because such a partial assignment $(x_1, \ldots, x_{p-1}, v)$ cannot be extended to an improved full assignment. The algorithm assigns the best value v to variable X_p and proceeds to variable X_{p+1}, and when variable X_p has no values left, it backtracks to variable X_{p-1}. Search terminates when it reaches a time-bound or when the first variable has no values left. In the latter case, the algorithm has found an optimal solution.

Algorithm Best-First with Mini-Bucket heuristics (BFMB(i)) maintains a list of open nodes. Each node corresponds to a partial assignment $\bar{x}^p$ and has an associated heuristic value $f(\bar{x}^p)$. The basic step of the algorithm consists of selecting an assignment $\bar{x}^p$ from the list of open nodes having the best heuristic value (that is—the highest value for maximization problems; the smallest value for minimization problems) $f(\bar{x}^p)$, expanding it by computing all partial assignments $(\bar{x}^p, v)$ for all values v of X_{p+1}, and adding them to the list of open nodes.

Since, as shown, the generated heuristics are admissible and monotonic, their use within Best-First search yields A* type algorithms whose properties are well understood. The algorithm is guaranteed to terminate with an optimal solution. When provided with more powerful heuristics, it explores a smaller search space, but otherwise it requires substantial space. It is known that Best-First algorithms are optimal. Namely, when given the same heuristic information, Best-First search is the most efficient algorithm in terms of the size of the search space it explores [5]. In particular, Branch-and-Bound will expand any node that is expanded by Best-First (up to some tie breaking conditions), and in many cases it explores a larger space. Still, Best-First may occasionally fail because of its memory requirements. Therefore, as we will indeed observe in our experiments, Branch-and-Bound and Best-First search have complementary properties, and both can be strengthened by the Mini-Bucket heuristics.

3.5. Selecting accuracy parameter

One of the open issues needing further future research is the best threshold point for the accuracy parameter i. For any accuracy parameter i, we can determine the space complexity of Mini-Bucket preprocessing in advance. This can be done by computing signatures (i.e., arguments) of all intermediate functions, without computing the actual functions. Based on the signatures of original and intermediate functions, we can compute the total space needed. Knowing the space complexity, we can estimate the time complexity. Thus given the time and space at our disposal, we can select the parameter i that would fit. However, the cost-effectiveness of the heuristic produced by Mini-Bucket preprocessing may not be predicted a priori. We observed that in general, as the problem graph is more dense, higher levels of Mini-Bucket heuristic become more cost-effective.

When repeatedly solving problems from a given class of problems, a preliminary empirical simulation can be informative when the problem class is not too heterogeneous. Otherwise, we can start with a small accuracy parameter (corresponding to a weak heuristic) and gradually proceed to higher accuracy parameters, until available time is up and then choose the best solution found.

4. Experimental results—Max-CSP

In order to empirically evaluate the performance of our approach, we have conducted a number of experiments on two classes of optimization problems: the Max-CSP task in Constraint Optimization and the Most Probable Explanation task in Bayesian networks.[4]

For the Max-CSP task, we tested the performance of BBMB(i) and BFMB(i) on sets of random binary CSPs. Each problem in this class is characterized by four parameters: $\langle N, K, C, T\rangle$, where N is the number of variables, K is the domain size, C is the number of constraints, and T is the tightness of each constraint, defined as the number of tuples not allowed. Each problem is generated by randomly picking C constraints out of $\binom{N}{2}$ total possible constraints, and picking T nogoods out of K^2 maximum possible for each constraint.

We used the min-degree heuristic for computing the ordering of variables. It places a variable with the smallest degree at the end of the ordering, connects all of its neighbors, removes the variable from the graph and repeats the whole procedure.

In addition to MBE(i), BBMB(i) and BFMB(i), we ran, for comparison, two state-of-the-art algorithms for solving Max-CSP: PFC-MPRDAC as defined in [22] and a Stochastic Local Search (SLS) algorithm we developed for CSPs [19].

[4] All our experiments were done on a 450 MHz Pentium II with 386 MB of RAM running Windows NT 4.0.

PFC-MPRDAC [22] is a specialized Branch-and-Bound search algorithm developed for constraint optimization. It uses a forward checking step based on a partitioning of unassigned variables into disjoint subsets of variables. This partitioning is used for computing a heuristic evaluation function that is used for determining variable and value ordering, as well as pruning. Currently, it is one of the best known complete algorithms for Max-CSP.

As a measure of performance, we used the accuracy ratio $opt = F_{\text{Max-CSP}}/F_{alg}$ between the value of the optimal solution ($F_{\text{Max-CSP}}$) and the value of the solution found by the test algorithm (F_{alg}), whenever $F_{\text{Max-CSP}}$ is available. We also record the running time of each algorithm.

We recorded the distribution of the accuracy measure *opt* over five predefined ranges: $opt \geqslant 0.95$, $opt \geqslant 0.5$, $opt \geqslant 0.2$, $opt \geqslant 0.01$ and $opt < 0.01$. However, because of space and clarity, we only report the number of problems that fall in the range $\geqslant 0.95$. Problems in this range were solved optimally.

4.1. Complete algorithms

Here we evaluate the performance of algorithms as complete ones. Tables 1–3 report results of experiments with three classes of over-constrained binary CSPs with domain sizes: $K = 10$, $K = 5$ and $K = 3$ respectively. Tables 1 and 2 contain three blocks, each corresponding to a set of CSPs with a fixed number of variables and constraints. Within each block, there are two small blocks each corresponding to a different constraint tightness, given in the first column. In columns 2 through 6 (Table 1), and columns 2 through 7 (Tables 2 and 3), we have results for MBE, BBMB and BFMB (in different rows) for different values of i-bound. In the last column we have results for PFC-MRDAC. Each entry in the table gives a percentage of the problems that were solved exactly (fall in the 0.95 range) within our time bound, and the average CPU time for these problems.

For example, looking at the second block of the middle large block in Table 1 (corresponding to binary CSPs with $N = 15$, $K = 10$, $C = 50$ and $T = 85$) we see that MBE with $i = 2$ (column 2) solved only 1% of the problems exactly in 0.02 seconds of CPU time. On the same set of problems, BBMB using Mini-Bucket heuristics, solved 20% of the problems optimally using 180 seconds of CPU time on the average, while BFMB solved 1% of the problems exactly in 190 seconds on the average. When moving to columns 3 through 6 in rows corresponding to the same set of problems, we see a gradual change caused by a higher level of Mini-Bucket heuristic (higher values of the i-bound). As expected, Mini-Bucket Elimination solves more problems, while using more time. Focusing on BBMB, we see that it solved all problems when the i-bound is 5 or 6. Its total running time as a function of i forms a U-shaped curve. At first ($i = 2$) it is high (180), then as i-bound increases the total time decreases (when $i = 5$ the total time is 28.7), but then as i-bound increases further the total time starts to increase again. The same behavior is observed in case of BFMB. For each set of problems, we have highlighted in bold the results of BBMB(i) and BFMB(i) corresponding to the optimal value of i.

This demonstrates a trade-off between the amount of preprocessing performed by MBE and the amount of subsequent search using the heuristic cost function generated by MBE. The optimal balance between preprocessing and search corresponds to the value of i-bound

Table 1
Search completion times for problems with 10 values. 100 samples

T	MBE BBMB BFMB $i=2$ %/time	MBE BBMB BFMB $i=3$ %/time	MBE BBMB BFMB $i=4$ %/time	MBE BBMB BFMB $i=5$ %/time	MBE BBMB BFMB $i=6$ %/time	PFC-MRDAC %/time
$N=10$, $K=10$, $C=45$. Time bound 180 sec. Avg $w^*=9$. Dense network.						
84	2/0.02	4/0.11	6/0.87	10/7.25	16/56.7	
	26/180	98/90.7	100/11.7	**100/10.0**	100/57.6	100/4.00
	2/189	4/184	78/65.7	**98/17.9**	100/59.3	
85	0/–	3/0.11	2/0.89	8/7.45	10/57.3	
	20/180	100/80.1	100/11.6	**100/9.62**	100/57.3	100/3.95
	0/–	5/124	82/54.4	**100/18.7**	100/58.9	
$N=15$, $K=10$, $C=50$. Time bound 180 sec. Avg $w^*=7.7$. Medium density.						
84	0/–	0/–	3/0.96	6/8.77	14/78.3	
	10/180	60/161	90/50.1	**100/26.2**	100/86.2	100/13.5
	0/–	0/–	21/70.5	65/49.8	97/89.7	
85	1/0.02	2/0.13	3/0.95	7/8.12	17/71.0	
	20/180	68/164	98/79.0	**100/28.7**	100/74.9	100/13.2
	1/190	5/184	16/82.0	63/59.6	97/82.8	
$N=25$, $K=10$, $C=37$. Time bound 180 sec. Avg $w^*=4.5$. Sparse network.						
84	0/–	7/0.10	30/0.60	84/3.41	99/9.74	
	36/114	99/4.42	**100/0.77**	100/3.70	100/9.93	100/4.16
	3/56.9	94/8.67	**100/1.28**	100/3.77	100/9.93	
85	0/–	10/0.10	34/0.60	79/3.20	99/9.36	
	31/88.6	100/7.55	**100/0.75**	100/3.31	100/9.58	100/7.51
	9/51.1	89/17.1	**100/1.34**	100/3.34	100/9.59	

at the bottom of the U-shaped curve. The added amount of search on top of MBE can be estimated by $t_{search} = t_{total} - t_{\mathrm{MBE}}$. As i increases, the average search time t_{search} decreases, and the overall accuracy of the search algorithm increases (more problems fall within higher ranges of *opt*). However, as i increases, the time of MBE preprocessing increases as well.

One crucial difference between BBMB and BFMB is that BBMB is an anytime algorithm—it always outputs an assignment, and as time increases, the solution improves.

Table 2
Search completion times for problems with 5 values. 100 samples

	MBE	MBE	MBE	MBE	MBE	MBE	MBE	
	BBMB	BBMB	BBMB	BBMB	BBMB	BBMB	BBMB	PFC-
T	BFMB	BFMB	BFMB	BFMB	BFMB	BFMB	BFMB	MRDAC
	$i=2$	$i=3$	$i=4$	$i=5$	$i=6$	$i=7$	$i=8$	
	%/time	%/time	%/time	%/time	%/time	%/time	%/time	%/time
$N=15$, $K=5$, $C=105$. Time bound 180 sec. Avg $w^*=14$. Dense network.								
18	0/–	0/–	0/–	12/0.56	13/2.27	31/11.7	34/49.7	
	10/180	32/180	64/148	96/81.4	100/33.4	**100/21.9**	100/52.5	100/9.61
	0/–	0/–	0/–	13/111	59/64.5	88/47.4	100/58.1	
19	0/–	0/–	0/–	0/–	5/2.78	12/14.6	40/60.3	
	16/180	40/180	77/155	100/76.8	100/29.7	**100/22.8**	100/60.9	100/7.69
	0/–	0/–	2/188	3/182	42/54.0	88/39.2	100/61.9	
$N=20$, $K=5$, $C=100$. Time bound 180 sec. Avg $w^*=12$. Medium density.								
18	0/–	0/–	7/0.17	10/0.71	11/3.12	23/14.4	29/68.7	
	5/180	15/180	38/170	71/132	86/82.3	**95/57.4**	96/90.6	100/18.7
	0/–	0/–	1/183	2/60.0	9/76.9	33/81.5	59/98.9	
19	0/–	0/–	0/–	4/0.70	4/3.21	4/14.9	4/70.7	
	4/180	24/180	56/160	64/121	84/97.5	**96/85.4**	92/90.0	100/17.4
	0/–	0/–	0/–	12/89.5	12/76.5	32/77.3	52/96.8	
$N=40$, $K=5$, $C=55$. Time bound 180 sec. Avg $w^*=5.1$. Sparse network.								
18	0/–	12/0.02	36/0.07	54/0.19	88/0.53	100/1.03	100/1.14	
	44/87.7	100/4.41	**100/0.21**	100/0.23	100/0.56	100/1.04	100/1.15	100/4.94
	3/4.56	92/14.9	100/0.45	**100/0.27**	100/0.57	100/1.04	100/1.16	
19	0/–	7/0.03	25/0.07	55/0.20	79/0.56	96/1.29	100/1.89	
	38/104	99/8.35	100/0.34	**100/0.25**	100/0.61	100/1.35	100/1.90	100/8.04
	1/25.4	83/14.4	100/1.28	**100/0.30**	100/0.63	100/1.36	100/1.90	

BFMB, on the other hand, only outputs a solution when it finds an optimal solution. In our experiments, if BFMB did not finish within the preset time bound, it returned the MBE assignment.

From the data in the tables we can see that the performance of BFMB is consistently worse than that of BBMB. BFMB(i) solves fewer problems than BBMB(i) and, on the average, takes longer on each problem. This is even more pronounced when a non-trivial amount of search is required (lower i-bound values). These results are in contrast to the

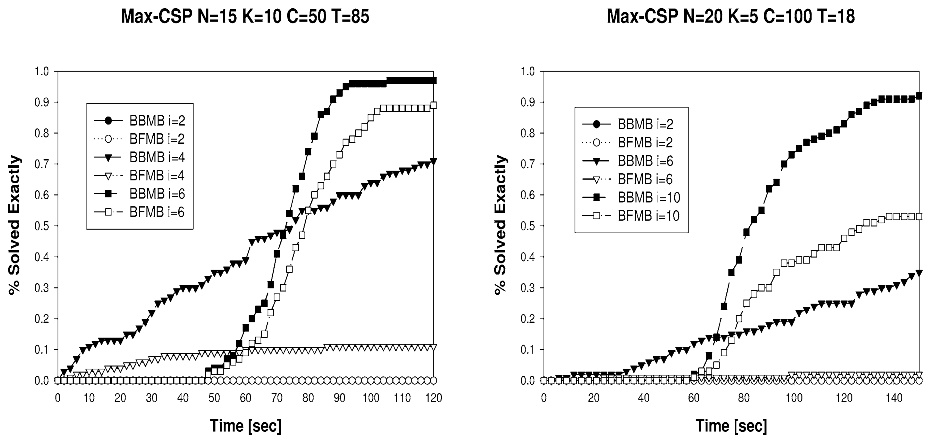
Table 3
Search completion times for problems with 3 values. 25 samples

	MBE	MBE	MBE	MBE	MBE	MBE	
	BBMB	BBMB	BBMB	BBMB	BBMB	BBMB	PFC-MRDAC
T	BFMB	BFMB	BFMB	BFMB	BFMB	BFMB	
	$i=2$	$i=4$	$i=6$	$i=8$	$i=10$	$i=12$	
	%/time	%/time	%/time	%/time	%/time	%/time	%/time
$N=100$, $K=3$, $C=200$. Time bound 1 hr. Avg $w^*=21$. Sparse network.							
1	70/0.03	90/0.06	100/0.32	100/2.15	100/15.1	100/116	
	90/12.5	**100/0.07**	100/0.33	100/2.16	100/15.1	100/116	100/0.08
	80/0.03	**100/0.07**	100/0.33	100/2.15	100/15.1	100/116	
2	0/–	0/–	4/0.35	20/2.28	20/15.6	24/123	
	0/–	0/–	96/644	**92/41**	96/69	100/125	100/757
	0/–	0/–	56/131	88/170	92/135	100/130	
3	0/–	0/–	0/–	0/–	4/14.4	4/114	
	0/–	0/–	100/996	100/326	**100/94.6**	100/190	100/2879
	0/–	0/–	16/597	60/462	88/344	84/216	
4	0/–	0/–	0/–	0/–	4/14.9	8/120	
	0/–	0/–	52/2228	88/1042	92/396	**100/283**	100/7320
	0/–	0/–	4/2934	8/540	28/365	60/866	

behavior we observed when using this scheme for optimization in belief networks as we will see in the next section. We speculate that this is because, for Max-CSP, there are large numbers of frontier nodes having the same heuristic value.

Tables 1–3 also report the results of PFC-MRDAC. When the constraint graph is dense, PFC-MRDAC is up to 2–3 times faster than the best performing BBMB. When the constraint graph is sparse, the best BBMB is up to two orders of magnitude faster than PFC-MRDAC. The superiority of our approach in sparse problems is most notable for larger problems (Table 3).

In Fig. 9 we provide an alternative view of the performance of BBMB(i) and BFMB(i). Let $F_{\mathrm{BBMB}(i)}(t)$ and $F_{\mathrm{BFMB}(i)}(t)$ be the fraction of the problems solved completely by BBMB(i) and BFMB(i), respectively, by time t. Each graph in Fig. 9 plots $F_{\mathrm{BBMB}(i)}(t)$ and $F_{\mathrm{BFMB}(i)}(t)$ for several values of i. These figures display the trade-off between preprocessing and search in a clear manner. Clearly, if $F_{\mathrm{BBMB}(i)}(t) > F_{\mathrm{BBMB}(j)}(t)$ for all t, then BBMB(i) completely dominates BBMB(j). For example, in Fig. 9(a) BBMB(4) completely dominates BBMB(2) (here BBMB(2) and BFMB(2) overlap). When $F_{\mathrm{BBMB}(i)}(t)$ and $F_{\mathrm{BBMB}(j)}(t)$ intersect, they display a trade-off as a function of time. For



(a) (b)

Fig. 9. Max-CSP: Distribution of search completion.

example, if we have less than 70 seconds, BBMB(4) is better than BBMB(6). However, when sufficient time is allowed, BBMB(6) is superior.

4.2. *Anytime algorithms*

Next we compare the anytime performance of BBMB, which returns a (suboptimal) solution any time during search, with that of Stochastic Local Search (SLS), which is inherently incomplete and can never guarantee an optimal solution for Max-CSP, but has been shown to work well on CSPs in practice.

A Stochastic Local Search (SLS) algorithm, such as GSAT [28,40], starts from a randomly chosen complete instantiation of all the variables, and moves from one complete instantiation to the next. It is guided by a cost function that is the number of unsatisfied constraints in the current assignment. At each step, the value of the variable that leads to the greatest reduction of the cost function is changed. The algorithm stops when either the cost is zero (a *global minimum*), in which case the problem is solved, or when there is no way to improve the current assignment by changing just one variable (a *local minimum*). A number of heuristics have been designed to overcome the problem of local minima [11,29,38,39]. In our implementation of SLS we use the basic greedy scheme combined with the constraint re-weighting as introduced in [29]. In this algorithm, each constraint has a weight and the cost function is the weighted sum of unsatisfied constraints. Whenever the algorithm reaches a local minimum, it increases the weights of unsatisfied constraints.

An SLS algorithm for CSPs can immediately be applied to a Max-CSP problem. When evaluating the performance of SLS, we treat it as an anytime algorithm—we report the fraction of problems solved exactly by time t as a function of t. To do that, we use the optimal cost found by BBMB.

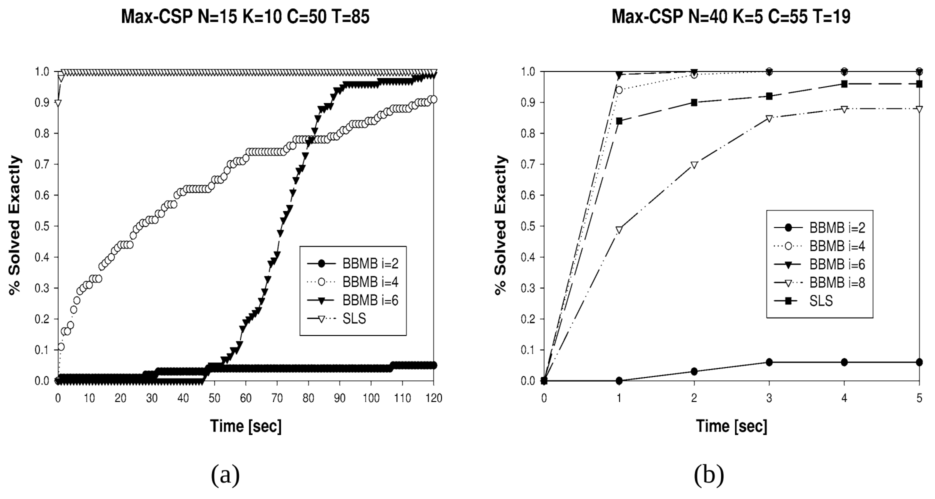


Fig. 10. Max-CSP: Distribution of anytime performance.

In Fig. 10 we present results comparing BBMB and SLS as anytime algorithms. Fig. 10(a) (Fig. 10(b)) corresponds to one row in Table 1 (Table 2). When the constraint graph is dense (Fig. 10(a)), SLS is substantially faster than BBMB. However, when the constraint graph is sparse (Fig. 10(b)), BBMB(4) and BBMB(6) are faster than SLS. We should note that, based on our experiments, on randomly generated networks, when the constraint graph has high or medium density, SLS exhibits impressive performance, often arriving at an optimal solution within a few seconds and is significantly superior to BBMB(i) as an anytime algorithm. Only on sparse networks is the performance of SLS somewhat worse than that of BBMB(i).

5. Experimental results—Bayesian networks

We also tested the performance of our scheme for solving the MPE task on four types of Bayesian networks—random coding networks, Noisy-OR networks, random Bayesian, and CPCS networks. On each problem we ran both BBMB(i) and BFMB(i) with various i-bounds. For comparison, on random coding networks we also ran the Iterative Belief Propagation (IBP) algorithm that is the best incomplete algorithm known for probabilistic decoding. The algorithm is identical to Pearl's belief updating on tree-like networks [32] but is applied iteratively on cyclic networks. The most likely value for each variable is selected for the output assignment [36].

We treat all algorithms as approximation algorithms. Algorithms BBMB and BFMB, if allowed to run until completion, will solve all problems exactly. However, since we use a time-bound, both algorithms may return suboptimal solutions, especially for harder and larger instances. As before, when interrupted, BBMB outputs its best solution, while BFMB outputs the Mini-Bucket solution.

As with Max-CSP problems, to measure performance we used the accuracy ratio $opt = P_{alg}/P_{\mathrm{MPE}}$ between the value of the solution found by the test algorithm (P_{alg}) and the value of the optimal solution (P_{MPE}), whenever P_{MPE} is available, given a fixed time bound. We also record the running time of each algorithm.

As before, we recorded the distribution of the accuracy measure *opt* over the same five predefined ranges : $opt \geqslant 0.95$, $opt \geqslant 0.5$, $opt \geqslant 0.2$, $opt \geqslant 0.01$ and $opt < 0.01$, and we only report the number of problems that fall in the range $\geqslant 0.95$.

5.1. Random coding networks

The purpose of *channel coding* is to provide reliable communication through a noisy channel. A systematic *error-correcting encoding* [27] maps a vector of K *information bits* $u = (u_1, \ldots, u_K)$, $u_i \in \{0, 1\}$, into an N-bit *codeword* $c = (u, x)$, where $N - K$ additional bits $x = (x_1, \ldots, x_{N-K})$, $x_j \in \{0, 1\}$, add redundancy to the information source in order to decrease the decoding error. The codeword, called the *channel input*, is transmitted through a noisy channel. A commonly used Additive White Noise (AWGN) channel model implies that independent Gaussian noise with variance σ^2 is added to each transmitted bit, producing the *channel output* y. Given a real-valued vector y, the *decoding* task is to restore the input information vector u [21,25,27].

The random coding networks we generated fall within the class of *linear block codes*. They can be represented as four-layer belief networks (Fig. 11). The second and third layers (from top) correspond to input information bits and parity check bits respectively. Each parity check bit represents an XOR function of input bits u_i. The first and last layers correspond to transmitted information and parity check bits respectively. Input information and parity check nodes are binary, while the output nodes are real-valued. In our experiments, each layer has the same number of nodes because we use code rate of $R = K/N = 1/2$, where K is the number of input bits and N is the number of transmitted bits.

Given a number of input bits K, number of parents P for each XOR bit, and channel noise variance σ^2, a coding network structure is generated by randomly picking parents for each XOR node. Then we simulate an input signal by assuming a uniform random

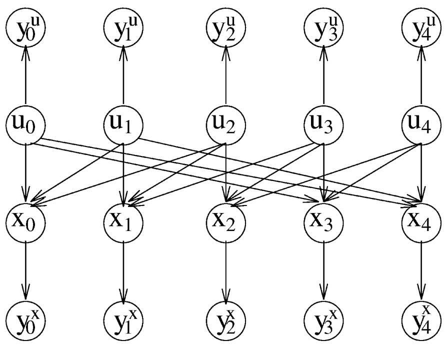


Fig. 11. Belief network for structured $(10, 5)$ block code with parent set size $P = 3$.

distribution of information bits, compute the corresponding values of the parity check bits, and generate an assignment to the output nodes by adding Gaussian noise to each information and parity check bit. The decoding algorithm takes as input the coding network and the observed real-valued output assignment and recovers the original input bit-vector by computing or approximating an MPE assignment. In our experiments, all coding networks were generated by randomly picking four parents for each XOR bit.

On random coding networks, we compare BBMB/BFMB against IBP using two different criteria: accuracy in finding the MPE tuple and *Bit Error Rate* (BER) which is the fraction of bits being decoded incorrectly. When comparing performance on solving the MPE task, we compare probabilities of the complete assignment computed by each algorithm. Since IBP does not solve the MPE problem, but instead computes a belief for each variable, we construct an output tuple for IBP by picking the most likely value for each variable. When comparing algorithms using Bit Error Rate, we need to pick a value for each variable and compare it against its correct value.

Tables 4–7 report results on random coding networks. In addition to MBE, BBMB and BFMB, we also ran IBP. For each σ we generated and tested 100 samples divided into 10 different networks, each simulated with 10 different input bit vectors.[5] We also attempted bucket elimination (BE) on this set of problems, but the induced width w^* was too large and BE failed to solve any problems.

In Table 4, there are five horizontal blocks, each corresponding to a different value of channel noise σ. Each block reports a distribution over the 95% accuracy range. Within each block we have three rows, one for each of MBE (Mini-Bucket Elimination), BBMB, and BFMB. Columns 3 through 6 report the results on various i-bounds. Column 7 reports results for IBP.

Looking at the third block in Table 4 (corresponding to $\sigma = 0.32$), we see that MBE with $i = 2$ (column 3) solved 45% of the problems exactly (*opt* $\geqslant 0.95$), in 0.05 seconds on the average. On the same set of problems, BBMB, using Mini-Bucket heuristics, solved 96% of the problems optimally using 0.94 seconds on the average, while BFMB solved all problems optimally with an average time of 0.13 seconds only. When moving to columns 4 through 6 in rows corresponding to $\sigma = 0.32$ and *opt* $\geqslant 0.95$, we see the gradual change caused by higher level of Mini-Bucket heuristic (higher values of i-bound). As expected, Mini-Bucket solves more problems, while using more time. For both BBMB and BFMB, we see that it always solved all problems with any i-bound, and its total running time as a function of i forms a U-shaped curve.

This demonstrates, once again, the trade-off between the amount of preprocessing performed by MBE and the amount of subsequent search using the heuristic cost function generated by MBE. The optimal balance between preprocessing and search corresponds to the value of i-bound at the bottom of the U-shaped curve. As problems become harder (i.e., σ increases) both search algorithms achieve their best performance for larger i, namely when the heuristic is stronger. For example, in Table 4, when σ is 0.22, the optimal performance is for $i = 2$. When σ is 0.40, the optimal point is $i = 10$.

[5] In the past [15], we have run much larger numbers of random coding experiments with different variable orderings. The results we report in this paper (with min-degree ordering) are normalized to 100 instances and are typical of all the experiments we have run.

Table 4
Random coding, $N = 100$, $K = 50$, avg $w^* = 21$. 100 samples

σ	*opt*	MBE BBMB BFMB $i = 2$ %/time	MBE BBMB BFMB $i = 6$ %/time	MBE BBMB BFMB $i = 10$ %/time	MBE BBMB BFMB $i = 14$ %/time	IBP %/time
0.22	$\geqslant 0.95$	86/0.04	89/0.06	97/0.32	99/3.26	100/
		100/0.06	100/0.14	100/0.33	100/3.26	0.09
		100/0.06	100/0.08	100/0.34	100/3.27	
0.28	$\geqslant 0.95$	74/0.04	70/0.06	86/0.34	97/3.13	100/
		99/0.38	**100/0.40**	100/0.40	100/3.14	0.09
		100/0.13	**100/0.10**	100/0.37	100/3.39	
0.32	$\geqslant 0.95$	45/0.05	56/0.06	71/0.34	81/3.34	99/
		96/0.94	100/0.78	**100/0.40**	100/3.39	0.09
		100/0.13	**100/0.10**	100/0.37	100/3.39	
0.40	$\geqslant 0.95$	14/0.04	20/0.06	44/0.32	62/3.07	90/
		95/3.13	99/2.20	**100/0.70**	100/3.11	0.07
		99/0.87	100/0.64	**100/0.48**	100/3.10	
0.51	$\geqslant 0.95$	3/0.04	8/0.06	13/0.34	18/3.38	32/
		77/12.0	92/8.15	**100/2.52**	100/4.00	0.08
		71/9.05	88/6.84	**99/2.78**	100/4.07	

As in case of Max-CSP, if BFMB did not finish within the preset time bound, it returned the MBE assignment. From Table 4 we see that when sufficient time is given (indicated by cases when both BBMB and BFMB solve all problems), the average running time of BFMB is never worse than BBMB and is sometimes better by a factor of 3–8.

In Table 5 we report the Bit Error Rate (BER) for the same problems and algorithms as in Table 4. BER is a standard measure used in the coding literature denoting the fraction of input bits that were decoded incorrectly. We observe that when the noise is very small (0.22, 0.28), BBMB and BFMB, at their most effective point i, are comparable to IBP. Both BBMB/BFMB and IBP solve all problems exactly within the time bound, and for small i they are also competitive time-wise. However, when the noise increases to 0.51, using sufficiently large i-bound, BBMB and BFMB outperform IBP in terms of BER, although they use more time. We ran 30 iterations of IBP on each problem and noticed that it usually converged to its final assignment after 5–10 iterations in a fraction of a second

Table 5
Random coding BER, $N = 100$, $K = 50$. 100 samples

σ	MBE BBMB BFMB $i = 2$ BER/time	MBE BBMB BFMB $i = 6$ BER/time	MBE BBMB BFMB $i = 10$ BER/time	MBE BBMB BFMB $i = 14$ BER/time	IBP BER/time
0.22	0.0060/0.04	0.0046/0.06	0.0010/0.32	0.0004/3.26	0.0002/
	0.0002/0.06	0.0002/0.14	0.0002/0.33	0.0002/3.26	0.09
	0.0002/0.06	0.0002/0.08	0.0002/0.34	0.0002/3.27	
0.28	0.0182/0.04	0.0212/0.06	0.0048/0.34	0.0001/3.13	0.0002/
	0.0014/0.38	0.0002/0.40	0.0002/0.40	0.0002/3.14	0.09
	0.0002/0.13	**0.0002/0.10**	0.0002/0.37	0.0002/3.39	
0.32	0.0448/0.05	0.0362/0.06	0.0256/0.34	0.0148/3.34	0.0022/
	0.0072/0.94	0.0022/0.78	0.0022/0.40	0.0022/3.39	0.09
	0.0022/0.13	0.0022/0.10	0.0022/0.37	0.0022/3.39	
0.40	0.0996/0.04	0.0996/0.06	0.0628/0.32	0.0406/3.07	0.0088/
	0.0194/3.13	0.0120/2.20	0.0088/0.70	0.0088/3.11	0.07
	0.0116/0.87	**0.0088/0.64**	0.0088/0.48	0.0088/3.10	
0.51	0.1916/0.04	0.1852/0.06	0.1630/0.34	0.1486/3.38	0.0800/
	0.0980/12.0	0.0830/8.15	**0.0762/2.52**	0.0762/4.00	0.08
	0.0974/9.05	0.0822/6.84	**0.0766/2.78**	0.0762/4.07	

only, which is generally much faster than BBMB and BFMB. However, unlike BBMB, the solution found by IBP cannot be improved, even if more time is allowed.

BER as a performance criteria is somewhat insensitive with respect to the computation accuracy. When decoding a bit, as long as the probability of the correct value is larger than the probability of the incorrect value, the error is 0, since the value with the largest probability is picked. Unlike a complete algorithm, such as BBMB or BFMB, an approximation algorithm, such as IBP, benefits from this.

These phenomena are more pronounced in Tables 6 and 7, where we present results with $K = 100$ input bits. In this set of experiments, we increased the time bound from 30 to 60 seconds (for small noise) or to 180 seconds (for large noise), while doubling the problem size. Again, we see a similar pattern of preprocessing-search trade-off as with networks of $K = 50$ bits. We observe again the superiority of BFMB over BBMB. Given the same i-bound, BFMB can solve more problems than BBMB in less time. In this set of problems,

Table 6
Random coding, $N = 200$, $K = 100$, avg $w^* = 42$. 100 samples

σ	*opt*	MBE BBMB BFMB $i = 2$ %/time	MBE BBMB BFMB $i = 6$ %/time	MBE BBMB BFMB $i = 10$ %/time	MBE BBMB BFMB $i = 14$ %/time	IBP %/time
0.22	$\geqslant 0.95$	79/0.09	84/0.12	90/0.73	95/8.00	100/
		98/0.94	98/0.65	99/0.95	100/8.04	0.16
		100/0.12	100/0.16	100/0.77	100/8.03	
0.28	$\geqslant 0.95$	41/0.09	45/0.12	59/0.72	71/7.95	100/
		84/3.72	88/4.17	96/2.50	99/8.64	0.13
		100/0.80	**100/0.56**	100/0.89	100/8.03	
0.32	$\geqslant 0.95$	18/0.09	21/0.12	31/0.73	46/8.06	99/
		63/10.2	68/9.49	87/6.33	92/10.7	0.16
		94/6.19	96/4.12	**99/2.49**	100/8.75	
0.40	$\geqslant 0.95$	N/A	0/-	5/0.73	6/7.77	77/
		N/A	28/90.5	39/51.8	58/42.7	0.15
		N/A	39/66.7	67/50.1	85/41.1	

when the noise is large, IBP is superior, although if we allowed more time, BBMB and BFMB would achieve better performance.

In Fig. 12 we provide an alternative view of the performance of BBMB(i) and BFMB(i). As before, let $F_{\mathrm{BBMB}(i)}(t)$ and $F_{\mathrm{BFMB}(i)}(t)$ be the fraction of the problems solved completely by BBMB(i) and BFMB(i), respectively, by time t. These figures display trade-off between preprocessing and search. For example, in Fig. 12(c) BBMB(10) completely dominates BBMB(6). When $F_{\mathrm{BBMB}(i)}(t)$ and $F_{\mathrm{BBMB}(j)}(t)$ intersect, we see a trade-off as a function of time, as is the case with BBMB(6) and BBMB(14). The figures also show that $F_{\mathrm{BFMB}(i)}(t)$ always dominates $F_{\mathrm{BBMB}(i)}(t)$ for each value of i.

5.2. *Random and Noisy-OR networks*

Uniform Random Bayesian networks and Noisy-OR networks are generated using parameters (N, K, C, P), where N is the number of variables, K is their domain size, C is the number of conditional probability matrices, and P is the number of parents in each conditional probability matrix.

The structure of each test problem is created by randomly picking C variables out of N and, for each, randomly selecting P parents from their preceding variables, relative to some ordering. For random Bayesian networks, each probability table is generated

Table 7
Random coding BER, $N = 200$, $K = 100$. 100 samples

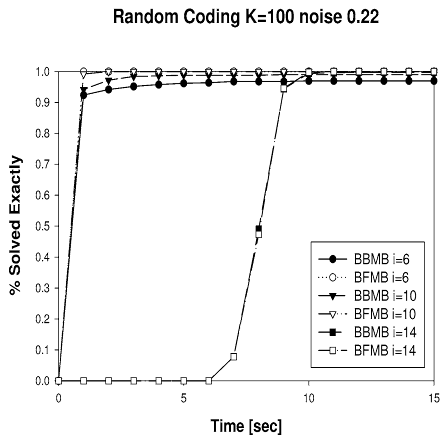
σ	MBE BBMB BFMB $i = 8$ BER/time	MBE BBMB BFMB $i = 10$ BER/time	MBE BBMB BFMB $i = 12$ BER/time	MBE BBMB BFMB $i = 14$ BER/time	IBP BER/time
0.22	0.00428/0.09	0.00378/0.12	0.00210/0.73	0.00120/8.00	0.00022/
	0.00068/0.94	0.00060/0.65	0.00034/0.95	0.00022/8.04	0.16
	0.00022/**0.12**	0.00022/0.16	0.00022/0.77	0.00022/8.03	
0.28	0.02026/0.09	0.02058/0.12	0.01532/0.72	0.00922/7.95	0.00104/
	0.00782/3.72	0.00604/4.17	0.00262/2.50	0.00136/8.64	0.13
	0.00102/0.80	**0.00102**/**0.56**	0.00102/0.89	0.00102/8.03	
0.32	0.04284/0.09	0.04374/0.12	0.03806/0.73	0.02760/8.06	0.00282/
	0.02378/10.2	0.02134/9.49	0.01098/6.33	0.00728/10.7	0.16
	0.00666/6.19	0.00606/4.12	**0.00352**/**2.49**	0.00282/8.75	
0.40	N/A	0.11520/0.12	0.10510/0.73	0.09730/7.77	0.01170/
	N/A	0.08350/90.5	0.05490/51.8	0.04740/42.7	0.15
	N/A	0.08120/66.7	0.05090/50.1	**0.02640**/**41.1**	

uniformly randomly. For Noisy-OR networks, each probability table represents an OR-function with given noise and leak probabilities P_{noise} and P_{leak}: $P(X = 0 \mid Y_1, \ldots, Y_P) = P_{leak} \times \prod_{Y_i=1} P_{noise}$.

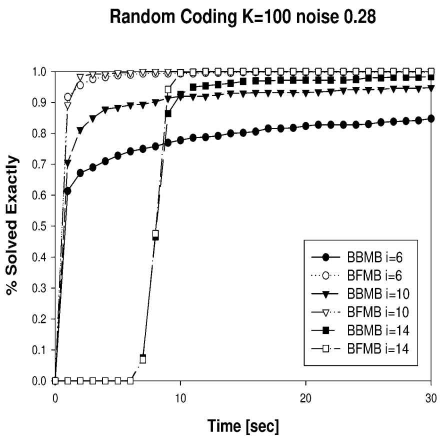
Tables 8–11 present results of experiments with Uniform Random and Noisy-OR networks. In each table, parameters N, K and P are fixed, while C, controlling network's sparseness, is changing. When running Noisy-OR experiments, we chose a number of variables as evidence variables and fixed their values.

We see again a similar pattern of trade-off between Mini-Bucket preprocessing for heuristic generation and search. For Noisy-OR networks (Tables 8–10), the Mini-Bucket algorithm can solve most of the problems exactly even when the i-bound is small, and BBMB/BFMB search serves mainly to prove optimality. We also see that here Branch-and-Bound is slightly better than Best-First, perhaps because the lower bound generated by the Mini-Bucket algorithm is often optimal. In such cases ($f = f^*$), because there are many solutions, Branch-and-Bound may find a solution quickly, while Best-First expands many nodes on the frontier, all having $f = f^*$, before reaching a complete assignment.

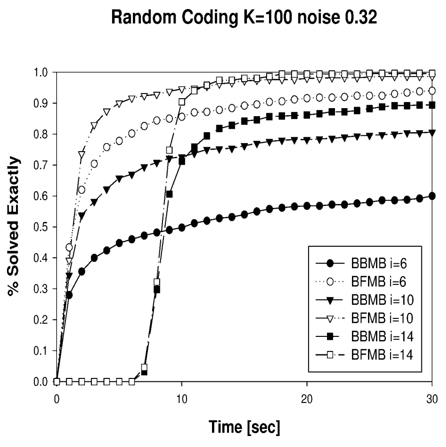
Since IBP performed so well on coding networks, we also ran IBP on Noisy-OR networks for comparison. Here IBP was quite good when the number of evidence variables



(a)



(b)



(c)

Fig. 12. Random coding with $K = 100$. $\sigma =$ (a) 0.22, (b) 0.28, (c) 0.32.

is small (Table 8). However, when we increase the number of evidence variables, IBP is quite poor (Table 10). Notice that MBE alone is quite competitive with IBP here.

In the case of Uniform Random networks (Table 11), the Mini-Bucket algorithm can solve most of the problems optimally when the i-bound is large. When the i-bound is small, a considerable amount of search is required. As with Noisy-OR networks, BBMB is slightly better than BFMB, due, we believe, to the tie-breaking rules. We also see that on Random Bayesian networks, IBP did not solve any problems (did not converge).

Figs. 13(a) and (b) correspond to one row in Tables 8 and 11, showing the performance of BBMB and BFMB as a function of time.

Table 8
MPE on Noisy-OR. $P_{noise} = 0.2$, $P_{leak} = 0.01$. 10 evidence variables. 100 samples

N *C* *P*	w^*	*opt*	MBE BBMB BFMB $i = 2$ %/time	MBE BBMB BFMB $i = 6$ %/time	MBE BBMB BFMB $i = 10$ %/time	MBE BBMB BFMB $i = 14$ %/time	IBP %/time
128	41	$\geqslant 0.95$	**100/0.05**	100/0.08	100/0.65	100/8.07	97/
85			100/0.95	**100/0.59**	100/0.98	100/8.47	1.6
4			100/1.76	100/1.19	100/1.37	100/8.56	
128	44	$\geqslant 0.95$	99/0.06	99/0.09	99/0.74	99/9.06	98/
95			**100/1.68**	100/1.68	100/1.69	100/9.70	1.7
4			100/2.68	100/2.58	100/2.47	100/9.96	
128	48	$\geqslant 0.95$	99/0.06	99/0.09	99/0.92	99/10.7	97/
105			100/2.67	**100/1.75**	100/1.93	100/11.2	1.9
4			100/5.22	100/3.88	100/2.82	100/12.0	
128	52	$\geqslant 0.95$	98/0.07	98/0.10	98/1.05	99/10.0	98/
115			100/4.47	100/4.01	**100/3.03**	100/12.8	2.1
4			100/7.37	100/6.61	100/4.85	100/13.9	

5.3. *CPCS networks*

As another realistic domain, we used the CPCS networks derived from the Computer-Based Patient Care Simulation system, and based on INTERNIST-1 and Quick Medical Reference expert systems [34]. The nodes in CPCS networks correspond to diseases and findings. Representing it as a belief network requires some simplifying assumptions,

(1) conditional independence of findings given diseases,
(2) Noisy-OR dependencies between diseases and findings, and
(3) marginal independencies of diseases.

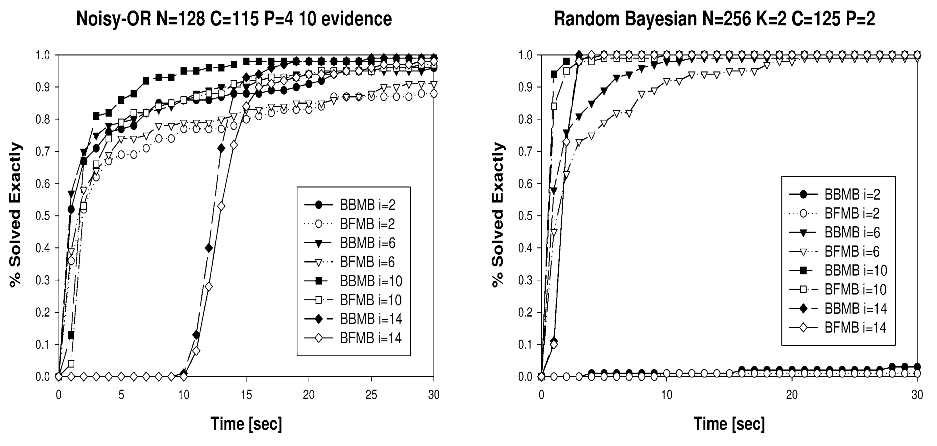
For details see [34].

In Table 12 we have results of experiments with two binary CPCS networks, cpcs360b ($N = 360$, $C = 335$) and cpcs422b ($N = 422$, $C = 348$), with 100 instances in both cases. Each instance had 10 evidence nodes picked randomly.

Our results show a similar pattern of trade-off between preprocessing and search. Since the cpcs360b network is solved quite effectively by the MBE approximation scheme, we get very good heuristics, and therefore, the added search time is relatively small, serving primarily to prove the optimality of the MBE solution. On the other hand, on cpcs422b MBE can solve less than half of the instances accurately when i is small, and more as i increases. BBMB and BFMB are roughly the same; both enhance the MBE's solution

Table 9
MPE on Noisy-OR. $P_{noise} = 0.2$, $P_{leak} = 0.01$. 20 evidence variables. 100 samples

			MBE	MBE	MBE	MBE	
N			BBMB	BBMB	BBMB	BBMB	
C	$w*$	*opt*	BFMB	BFMB	BFMB	BFMB	IBP
P			$i = 2$	$i = 6$	$i = 10$	$i = 14$	
			%/time	%/time	%/time	%/time	%/time
128	41	$\geqslant 0.95$	99/0.07	100/0.09	100/0.76	100/9.29	82/
85			100/25.4	100/3.60	**100/1.75**	100/9.82	1.45
4			100/54.8	100/8.41	100/3.04	100/10.5	
128	44	$\geqslant 0.95$	96/0.07	99/0.10	98/0.89	100/10.9	75/
95			99/47.6	100/14.9	**100/6.76**	100/12.7	1.55
4			99/82.1	100/28.9	100/10.7	100/15.2	
128	48	$\geqslant 0.95$	97/0.08	99/0.10	100/1.04	99/12.7	79/
105			100/67.0	100/30.3	**100/9.94**	100/15.8	1.7
4			98/103	99/48.4	100/17.9	100/19.9	
128	52	$\geqslant 0.95$	87/0.09	91/0.12	94/1.11	96/14.3	69/
115			97/143	100/80.0	100/37.5	**100/30.7**	1.95
4			93/194	97/125	99/60.0	98/42.5	



(a) (b)

Fig. 13. Distribution of search completion.

Table 10
MPE on Noisy-OR. $P_{noise} = 0.2$, $P_{leak} = 0.01$. 30 evidence variables. 100 samples

			MBE	MBE	MBE	MBE	
N			BBMB	BBMB	BBMB	BBMB	
C	$w*$	*opt*	BFMB	BFMB	BFMB	BFMB	IBP
P			$i = 2$	$i = 6$	$i = 10$	$i = 14$	
			%/time	%/time	%/time	%/time	%/time
128	41	$\geqslant 0.95$	90/0.07	95/0.08	97/0.70	100/8.80	57/
85			100/80.0	100/21.5	**100**/**6.33**	100/11.4	1.3
4			98/125	100/38.6	100/12.3	100/14.6	
128	44	$\geqslant 0.95$	79/0.08	89/0.11	94/0.80	93/10.3	51/
95			99/120	100/49.0	**100**/**14.1**	100/15.5	1.45
4			86/150	95/71.6	100/30.6	100/21.8	
128	48	$\geqslant 0.95$	84/0.08	92/0.10	95/0.92	95/11.9	55/
105			97/168	98/79.0	100/32.7	**100**/**27.4**	1.6
4			88/220	97/128	99/52.9	99/42.5	
128	52	$\geqslant 0.95$	73/0.09	84/0.11	88/1.08	89/13.6	51/
115			92/184	97/123	98/72.9	**98**/**48.1**	1.75
4			75/213	91/158	91/93.4	95/67.7	

quality significantly. They can solve all instances accurately for $i \geqslant 12$. For comparison, bucket elimination solved the cpcs360 network (with no evidence) in 115 sec while for cpcs422 it took 1697 sec. Processing the networks with evidence is a much more challenging task, however.

We also ran IBP on cpcs360b. After one iteration of IBP, 99 problems were solved correctly. However, IBP would diverge when more iterations were run. For example, after 30 iterations, only 62 problems were solved correctly. Because of the large family size, cpcs422b took too long for our implementation of IBP.

5.4. *Stochastic local search in Bayesian networks*

Since Stochastic Local Search (SLS) was quite successful in solving the Max-CSP task, a question arises as to its performance on solving the MPE task in Bayesian networks. In [17] we investigated the performance of a simple greedy hill-climbing algorithm combined with Stochastic Simulation on solving the MPE task in Random Bayesian networks, Noisy-OR network as well as random coding networks. This algorithm was the best version of SLS among several we tried for the MPE task. We found that its performance was poor, sometimes significantly worse than that of Mini-Bucket Elimination with an i-bound 10.

Table 11
MPE on uniform random. Time bound 30 sec. 100 samples

N *C* *P*, *K*	BE %/time w^*	*opt*	MBE BFMB BBMB $i=2$ %/time	MBE BFMB BBMB $i=4$ %/time	MBE BFMB BBMB $i=6$ %/time	MBE BFMB BBMB $i=8$ %/time	MBE BFMB BBMB $i=10$ %/time	MBE BFMB BBMB $i=12$ %/time	MBE BFMB BBMB $i=14$ %/time	IBP %/time
256	91/4.91	$\geqslant 0.95$	5/0.03	10/0.04	41/0.04	60/0.06	87/0.10	98/0.18	100/0.24	0/
100	14.6		72/12.2	100/0.79	100/0.10	**100/0.09**	100/0.12	100/0.19	100/0.25	0.37
2,2			49/11.2	99/1.19	100/0.16	**100/0.11**	100/0.14	100/0.20	100/0.26	
256	69/7.11	$\geqslant 0.95$	2/0.03	14/0.03	39/0.06	42/0.08	76/0.12	91/0.23	98/0.37	0/
105	15.8		70/15.3	100/1.49	100/0.13	**100/0.10**	100/0.14	100/0.25	100/0.40	0.38
2,2			38/12.3	98/1.85	100/0.22	**100/0.13**	100/0.16	100/0.27	100/0.42	
256	41/9.06	$\geqslant 0.95$	3/0.04	6/0.04	19/0.05	34/0.07	53/0.14	78/0.32	93/0.60	0/
110	17.5		43/17.1	99/2.37	100/0.24	**100/0.14**	100/0.17	100/0.36	100/0.65	0.39
2,2			26/18.5	96/3.46	100/0.41	**100/0.20**	100/0.20	100/0.65	100/0.67	
256	17/12.3	$\geqslant 0.95$	0/–	6/0.04	17/0.05	32/0.08	41/0.16	62/0.38	81/0.92	0/
115	19.1		23/24.8	95/5.24	100/0.45	100/0.22	**100/0.22**	100/0.45	100/1.04	0.41
2,2			6/16.4	90/7.56	100/0.90	100/0.37	**100/0.29**	100/0.48	100/1.07	
256	11/9.99	$\geqslant 0.95$	1/0.04	7/0.04	15/0.05	23/0.08	42/0.18	54/0.46	76/1.25	0/
120	20.3		18/24.9	92/9.45	100/0.80	100/0.30	**100/0.28**	100/0.53	100/1.39	0.42
2,2			4/25.7	70/9.14	100/1.54	100/0.53	**100/0.37**	100/0.59	100/1.41	
256	2/21.1	$\geqslant 0.95$	0/–	4/0.04	14/0.05	20/0.08	22/0.19	50/0.53	67/1.53	0/
125	22.4		14/26.6	83/11.1	99/1.66	100/0.87	**100/0.46**	100/0.68	100/1.69	0.43
2,2			2/21.3	61/11.5	99/3.04	100/1.24	**100/0.68**	100/0.79	100/1.73	

6. Related work

Our approach applies the paradigm that heuristics can be generated by consulting relaxed models, suggested in [10,31] and even earlier in [13]. The Mini-Bucket computation can be viewed as relaxation in the following sense. For each bucket and its partitioning into mini-buckets, a variable in the original problem is replaced by a set of new variables in the relaxed problem, each corresponding to a single mini-bucket, and each function in the original problem is associated with the copy of the variable in the relaxed problem corresponding to its mini-bucket in the original problem. For example, the Mini-Bucket trace in Fig. 5(b), corresponds to solving exactly by full bucket-elimination the following

Table 12
CPCS networks. Time 30 and 45 respectively

CPCS360b	MBE	MBE	MBE	MBE
w^*=20	BBMB	BBMB	BBMB	BBMB
100	BFMB	BFMB	BFMB	BFMB
samples	$i=4$	$i=8$	$i=12$	$i=16$
10 evid.	%/time	%/time	%/time	%/time
$\geqslant 0.95$	93/0.91	93/0.93	96/1.99	98/15.8
	100/0.93	100/0.94	100/2.00	100/15.8
	100/0.98	100/0.96	100/2.00	100/15.8
CPCS422b	MBE	MBE	MBE	MBE
$w^*=23$	BBMB	BBMB	BBMB	BBMB
100	BFMB	BFMB	BFMB	BFMB
samples	$i=4$	$i=8$	$i=12$	$i=16$
10 evid.	%/time	%/time	%/time	%/time
$\geqslant 0.95$	40/22.6	46/23.1	51/22.7	59/39.0
	96/24.8	98/24.5	100/22.9	100/39.0
	97/25.9	97/24.5	100/23.1	100/39.1

relaxation of the problem in Fig. 2. Variable B is replaced by two variables B_1 and B_2, and the functions $P(e \mid b, c)$, $P(d \mid a, b)$, and $P(b \mid a)$ are replaced by $P(e \mid b_1, c)$, $P(d \mid a, b_2)$ and $P(b_2 \mid a)$. Thus the two mini-buckets correspond to two full buckets in the relaxed problem. The relaxed problem has a smaller width and can be solved more efficiently, yielding a bound (upper or lower) as expected. Another work using this paradigm appears in [6], that generates heuristics for value ordering in constraint satisfaction problems. The idea is to count the number of solutions extending the current partial assignment using a relaxation to tree subproblem. Another line of this research was presented in [30,35]. These works assume as input only a set of search space generation rules. Therefore, while they share the same goals they use a different approach to derive a relaxed model of the problem.

Relating our work to the broader work on combinatorial optimization, we observe that our Branch-and-Bound scheme can be viewed as an extension of Branch-and-Bound algorithms for integer programming (that are restricted to linear objective functions and constraints only). These methods create lower bounds by relaxing the integrality constraints, and they are solved by efficient linear programming schemes (mostly improved versions of the simplex method) [12,46]. It would be interesting to compare and combine our approach with integer programming. Experience gained in the constraint programming community demonstrates that for many problem instances search with constraint propagation outperforms integer programming (see for instance [8,26]).

In relation to constraint processing algorithms, the Mini-Bucket heuristics can be viewed as an extension of bounded constraint propagation algorithms that were investigated in the constraint community in the last decade [2]. Rather than applying this idea to the constraints only, we extended it here to the objective function as well.

We mentioned earlier related work on specific MPE algorithms. It would be interesting to compare our algorithms with search-based algorithms for MPE. Search is normally not the method of choice for probabilistic inference. Still some search methods and integer programming approaches have been pursued and it would be good to compare against those [37,42].

7. Summary and conclusion

The paper presents and evaluates the power of a new scheme that generates search heuristics mechanically for problems that are specified by a set of dependencies. The framework can capture many classes of problems, such as those defined on belief networks, influence diagrams, constraint networks, and in general, what is often called *graphical models*. The heuristics are extracted from the Mini-Bucket approximation method applied to the dependency model. Our experiments demonstrate the potential of this scheme in improving general search, showing that the Mini-Bucket heuristic's accuracy allows the user a controlled trade-off between preprocessing and search. We demonstrate this property in the context of both Branch-and-Bound and Best-First search. Although the best threshold point for the accuracy parameter may not be predicted a priori, a preliminary empirical analysis can be informative when given a class of problems that is not too heterogeneous.

Furthermore, the experiments show that Mini-Bucket heuristics can facilitate Best-First search on relatively sizable problems, thus extending the boundaries of this search scheme which is computationally optimal (relative to search algorithms having access to the same heuristic) for achieving exact solution. Indeed, we show that Best-First sometimes outperforms Branch-and-Bound by a factor of 3–8. In other cases, however, as we observed for Max-CSP, Branch-and-Bound outperforms Best-First search.

We evaluated our scheme within two classes of optimization problems—Max-CSP on constraint networks and Most Probable Explanation in Bayesian networks. We showed that search with Mini-Bucket heuristics can be competitive with the best known algorithms for Max-CSP, such as SLS and PFC-MRDAC, especially when the networks are relatively sparse, in which case BBMB outperforms both SLS and PFC-MRDAC. We also show that search with Mini-Bucket heuristics can be competitive with the best known approximation algorithms for probabilistic decoding, such as IBP, when noise is small and when the networks are relatively small. Obviously, when problem sizes increase, BBMB and BFMB in general require much more time. However, as much as IBP is efficient, its performance will not improve with time.

Since the Mini-Bucket elimination is applicable across a variety of tasks such as probabilistic inference and decision making, the scheme proposed here has the potential of being widely applicable.

An important extension that we plan to pursue is generating the Mini-Bucket heuristics during search rather than in a preprocessing mode. This will allow a more flexible search scheme that can facilitate dynamic variable ordering and will enable exploiting different levels of Mini-Bucket accuracy in different parts of the search tree. Experience with constraint processing methods suggests that dynamic variable ordering is essential for efficiency and that the bounded computation per each node in the search tree is cost-effective.

References

[1] P. Dagum, M. Luby, Approximating probabilistic inference in Bayesian belief networks is NP-hard, in: Proc. AAAI-93, Washington, DC, 1993.
[2] R. Dechter, Constraint networks, in: Encyclopedia of Artificial Intelligence, 1992, pp. 276–285.
[3] R. Dechter, Bucket elimination: A unifying framework for probabilistic inference algorithms, in: Proc. 12th Conference on Uncertainty in Artificial Intelligence (UAI-96), Portland, OR, 1996, pp. 211–219.
[4] R. Dechter, Bucket elimination: A unifying framework for reasoning, Artificial Intelligence 113 (1999) 41–85.
[5] R. Dechter, J. Pearl, Generalized best-first search strategies the optimality of A^*, J. ACM 32 (1985) 506–536.
[6] R. Dechter, J. Pearl, Network-based heuristics for constraint satisfaction problems, Artificial Intelligence 34 (1987) 1–38.
[7] R. Dechter, I. Rish, A scheme for approximating probabilistic inference, in: Proc. 13th Conference on Uncertainty in Artificial Intelligence (UAI-97), Providence, RI, 1997, pp. 132–141.
[8] R. Dechter (Ed.), Principles and Practice of Constraint Programming (CP-2000), Lecture Notes in Computer Science, Springer, Berlin, 2000.
[9] E.C. Freuder, R.J. Wallace, Partial constraint satisfaction, Artificial Intelligence 58 (1–3) (1992) 21–70.
[10] J. Gaschnig, Performance measurement analysis of search algorithms, Technical Report CMU-CS-79-124, Carnegie Mellon University, Pittsburgh, PA, 1979.
[11] I.P. Gent, T. Walsh, Towards an understanding of hill-climbing procedures for SAT, in: Proc. AAAI-93, Washington, DC, 1993, pp. 28–33.
[12] Z. Gu, G.L. Nemhauser, M.W.P. Savelsbergh, Lifted flow covers for mixed 0–1 integer programs, Math. Programming (1999) 439–467.
[13] M. Held, R.M. Karp, The travelling salesman problem minimum spanning trees, Oper. Res. 18 (1970) 1138–1162.
[14] T. Ibaraki, Enumerative Approaches to Combinatorial Optimization—Part I, Annals of Operations Research, Vol. 10, Scientific, Basel, 1987.
[15] K. Kask, R. Dechter, Branch bound with mini-bucket heuristics, in: Proc. IJCAI-99, Stockholm, Sweden, 1999, pp. 426–433.
[16] K. Kask, R. Dechter, Mini-Bucket heuristics for improved search, in: Proc. 15th Conference on Uncertainty in Artificial Intelligence (UAI-99), Stockholm, Sweden, 1999, pp. 314–323.
[17] K. Kask, R. Dechter, Stochastic local search for Bayesian networks, in: Workshop on AI Statistics (AI-STAT-99), 1999, pp. 113–122.
[18] K. Kask, R. Dechter, New search heuristics for Max-CSP, in: Proc. Conference on Principles and Practice of Constraint Programming (CP-2000), Lecture Notes in Computer Science, Springer, Berlin, 2000, pp. 262–277.
[19] K. Kask, R. Dechter, GSAT local consistency, in: Proc. IJCAI-95, Montreal, Quebec, 1995, pp. 616–622.
[20] R. Korf, Linear-space best-first search, Artificial Intelligence 62 (1993) 41–78.
[21] F.R. Kschischang, B.H. Frey, Iterative decoding of compound codes by probability propagation in graphical models, IEEE J. Selected Areas in Communication 16 (2) (1998) 219–230.
[22] J. Larossa, P. Meseguer, Partition-based lower bound for Max-CSP, in: Proc. Conference on Principles and Practice of Constraint Programming (CP-1999), Alexandria, VA, 1999, pp. 303–315.

[23] E.L. Lawler, D.E. Wood, Branch-and-bound methods: A survey, Oper. Res. 14 (1966) 699–719.

[24] Z. Li, B. D'Ambrosio, An efficient approach for finding the MPE in belief networks, in: Proc. 10th Conference on Uncertainty in Artificial Intelligence (UAI-93), Amherst, MA, 1993, pp. 342–349.

[25] D.J.C. MacKay, R.M. Neal, Near shannon limit performance of low density parity check codes, Electronic Letters 33 (1996) 457–458.

[26] M. Mahler, J. Puget (Eds.), Principles and Practice of Constraint Programming (CP-98), Lecture Notes in Computer Science, Springer, Berlin, 1998.

[27] R.J. McEliece, D.J.C. MacKay, J.-F. Cheng, Turbo decoding as an instance of Pearl's belief propagation algorithm, IEEE J. Selected Areas in Communication 16 (2) (1998) 140–152.

[28] S. Minton, M.D. Johnston, A.B. Philips, P. Laird, Solving large scale constraint satisfaction scheduling problems using heuristic repair methods, in: Proc. AAAI-90, Boston, MA, 1990, pp. 17–24.

[29] P. Morris, The breakout method for escaping from local minima, in: Proc. AAAI-93, Washington, DC, 1993, pp. 40–45.

[30] A.E. Mayer, O. Hansson, M.M. Yung, Criticizing solutions to relaxed models yields powerful admissible heuristics, Inform. Sci. 63 (3) (1992) 207–227.

[31] J. Pearl, Heuristics: Intelligent Search Strategies, Addison-Wesley, Reading, MA, 1984.

[32] J. Pearl, Probabilistic Reasoning in Intelligent Systems, Morgan Kaufmann, San Mateo, CA, 1988.

[33] Y. Peng, J.A. Reggia, A connectionist model for diagnostic problem solving, IEEE Trans. Systems Man Cybernet. (1989).

[34] M. Pradhan, G. Provan, B. Middleton, M. Henrion, Knowledge engineering for large belief networks, in: Proc. 10th Conference on Uncertainty in Artificial Intelligence, 1994.

[35] A.E. Prieditis, Machine discovery of effective admissible heuristics, Machine Learning 12 (1993) 117–141.

[36] I. Rish, K. Kask, R. Dechter, Empirical evaluation of approximation algorithms for probabilistic decoding, in: Proc. Conference on Uncertainty in Artificial Intelligence (UAI-98), 1998.

[37] E. Santos, On the generation of alternative explanations with implications for belief revision, in: Proc. 7th Conference on Uncertainty in Artificial Intelligence (UAI-91), Los Angeles, CA, 1991, pp. 339–347.

[38] B. Selman, H. Kautz, An empirical study of greedy local search for satisfiability testing, in: Proc. AAAI-93, Washington, DC, 1993, pp. 46–51.

[39] B. Selman, H. Kautz, B. Cohen, Noise strategies for local search, in: Proc. AAAI-94, Seattle, WA, 1994, pp. 337–343.

[40] B. Selman, H. Levesque, D. Mitchell, A new method for solving hard satisfiability problems, in: Proc. AAAI-92, San Jose, CA, 1992, pp. 440–446.

[41] P.P. Shenoy, G. Shafer, Axioms for probability and belief-function propagation, in: Uncertainty in Artificial Intelligence, Vol. 4, North-Holland, Amsterdam, 1990, pp. 169–198.

[42] S.E. Shimony, E. Charniak, A new algorithm for finding map assignments to belief networks, in: P. Bonissone, M. Henrion, L. Kanal, J. Lemmer (Eds.), Uncertainty in Artificial Intelligence, Vol. 6, North-Holland, Amsterdam, 1991, pp. 185–193.

[43] B.K. Sy, Reasoning MPE to multiply connected belief networks using message-passing, in: Proc. AAAI-92, San Jose, CA, 1992, pp. 570–576.

[44] G. Verfaillie, M. Lemaitre, T. Schiex, Russian doll search, in: Proc. AAAI-96, Portland, OR, 1996, pp. 181–187.

[45] R. Wallace, Analysis of heuristic methods for partial constraint satisfaction problems, in: Proc. Conference on Principles and Practice of Constraint Programming (CP-1996), Cambridge, MA, 1996, pp. 482–496.

[46] L.A. Wolsey, Integer Programming, Wiley, New York, 1998.