Stochastic Local Search for Solving the Most Probable Explanation Problem in Bayesian Networks

Diplomarbeit in englischer Sprache

Eingereicht am Fachbereich Informatik

der Technischen Universität Darmstadt

von **Frank Hutter**

Betreuer:	Dr. Thomas Stützle
Externer Betreuer:	Dr. Holger H. Hoos (The University of British Columbia, Canada)
Tag der Einreichung:	30. September 2004

Abstract

In this thesis, we develop and study novel Stochastic Local Search (SLS) algorithms for solving the Most Probable Explanation (MPE) problem in graphical models, that is, to find the most probable instantiation of all variables $\mathbf{V}$ in the model, given the observed values $\mathbf{E} = \mathbf{e}$ of a subset $\mathbf{E}$ of $\mathbf{V}$. SLS algorithms have been applied to the MPE problem before [KD99b, Par02], but none of the previous SLS algorithms pays sufficient attention to such important concerns as algorithmic complexity per search step, search stagnation, and thorough parameter tuning. We remove these shortcomings of previous SLS algorithms for MPE, improving their efficiency by up to six orders of magnitude. In a thorough experimental analysis, we demonstrate how each of the novel components of our algorithms substantially contributes to their high performance. A comparison with an anytime version of the prominent Mini-Buckets algorithm [DR03] and the exact algorithm Branch-and-Bound with static Mini-Buckets heuristic (BBMB) [KD99a, MKD03] shows that our best algorithm outperforms these approaches on most MPE instances we study. We also show that our SLS algorithms scale much better in terms of a number of important instance characteristics, namely the number of variables, domain size, node degree, and induced width of the underlying graphical model.

Zusammenfassung

In dieser Diplomarbeit entwickeln und untersuchen wir neuartige Stochastische Lokale Suchverfahren, um das Problem der wahrscheinlichsten Erklärung (*most probable explanation*, MPE) in Graphischen Modellen zu lösen, d. h., die wahrscheinlichste Instantiierung aller Modellvariablen $\mathbf{V}$ zu finden, wenn eine partielle Instantiierung $\mathbf{E} = \mathbf{e}$ der Variablen $\mathbf{E} \subseteq \mathbf{V}$ als Evidenz gegeben ist. Stochastische Lokale Suche (SLS) wurde schon zuvor zur Lösung dieses Problems angewendet, aber keiner der bisherigen SLS Algorithmen [KD99b, Par02] befasst sich detailliert genug mit den zentralen Problematiken der Komplexität pro Suchschritt, der Stagnation der Suche und dem sorgfältigen Abstimmen der Parameter. Wir entfernen diese Schwächen früherer SLS Algorithmen und erreichen so eine Geschwindigkeitssteigerung von bis zu sechs Größenordnungen. In einer umfassenden experimentellen Analyse zeigen wir, wie jede neue Komponente unserer Algorithmen zu deren hohen Performanz beiträgt. Ein Vergleich mit einer "Anytime"-Variante des prominenten Mini-Buckets Algorithmusses [DR03], sowie mit dem exakten Algorithmus Branch and Bound mit Mini-Buckets Heuristik (s-BBMB) [KD99a, MKD03] zeigt, dass unser bester Algorithmus für die meisten Probleminstanzen eine höhere Performanz zeigt als diese Algorithmen. Wir zeigen auch, dass unsere SLS Algorithmen deutlich besser mit einer Reihe wichtiger Instanzmerkmalen skalieren als die anderen Ansätze. Diese Merkmale umfassen die Anzahl an Variablen, die Domänengröße, den Knotengrad, sowie die induzierte Weite des zugrundeliegenden Graphischen Modells.

Ehrenwörtliche Erklärung

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

Darmstadt, September 2004

Frank Hutter

Acknowledgements

I am deeply grateful to my thesis supervisor Thomas Stützle at Darmstadt University of Technology (TUD) and my cosupervisor Holger Hoos at The University of British Columbia (UBC) for giving me the freedom to explore my areas of interest and for their precise feedback that led to substantial improvements of this thesis.

My interest in Stochastic Local Search methods has its roots in a graduate course by Holger Hoos I attended during my visit to UBC in 2001/02. In this context, I am indebted to Prof. Wolfgang Bibel and Thomas Stützle for founding and supervising the Exchange Program between TUD and UBC, to all my professors and fellow students at UBC for a seamless integration and the great times we shared, to Holger Hoos for his continued advice and guidance during this time, as well as to the German Academic Exchange Service for sponsorship. I thank everybody who contributed to making Vancouver a home away from home.

I would also like to express my gratitude to Richard Dearden for hiring me as an intern at the NASA Ames Research Center in the summers of 2002 and 2003. These internships have boosted my interest in probabilistic inference and also gave me the chance to enjoy work and fun with top researchers.

During the course of my research, I have further enjoyed joint research with and received much valuable input from the members of the Intellectics group at TUD, the members of the Bioinformatics, and Empirical & Theoretical Algorithmics Laboratory at UBC, and many members of the Laboratory for Computational Intelligence at UBC. I am especially grateful for the stimulating environment in these groups and the pleasant discussions I had with students and researchers of so different academic and personal backgrounds.

From a technical point of view, I would like to thank my office mate Christian Bang at Darmstadt University of Technology for his continuous help in finding my way in the world of the Ruby programming language, which paved the way to an invaluable automization of my experiments and the development of ParamILS. I gratefully acknowledge James Park for providing his implementation of GLS and Radu Marinescu for providing implementations of the deterministic algorithms s-BBMB and d-BBMB, as well as useful advice.

Finally, I would like to thank my loving partner Diana for putting up with my nights and weekends spend in the office and for bringing sunshine into my life when all else failed. I am extremely grateful to her and to my family, especially my parents and my brother, for their unconditional love, care, advice, and emotional support. I also would like to thank them and all my friends for the peaceful and enjoyable environment I can always be certain to find at home. I especially thank my parents and Diana for supporting my choice to follow the path laid out for me far away from home.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Since the early days of Artificial Intelligence, different representations of knowledge in "intelligent" computer systems have been advocated. Any such knowledge representation must enable the system to draw conclusions given some data. In this thesis, we deal with the problem of finding the *Most Probable Explanation (MPE)* for the data when reasoning under uncertainty. More specifically, in the light of uncertain knowledge represented as a probabilistic graphical model, such as a Bayesian network, this problem is cast as finding the most probable instantiation of all the model's variables $\mathbf{V}$ given the observed values of a subset of $\mathbf{V}$.

The MPE problem in graphical models is of considerable interest to researchers in such heterogeneous fields as medical diagnosis [JJ99], fault diagnosis [RBM02a], computer vision [TF03], and prediction of side-chains in protein folding [YW03], to name just a few. Consequently, many algorithms have been suggested to solve this problem, but since it is $\mathcal{NP}$-hard [SD03], the search for efficient algorithms is far from over. Prominent exact algorithms, such as Bucket Elimination [Dec96] or Junction Trees [CDLS99] can solve the MPE problem in many practically relevant sparse Bayesian networks [CDLS99]. However, these algorithms degrade rapidly as networks become denser, exhibiting exponential time- and space-complexity in the induced width of the network's independence graph [CDLS99, Dec96]. Another type of exact algorithms for MPE that is more suitable for networks with high induced width is based on systematic search, such as Branch-and-Bound (BnB), guided by the approximate Mini-Buckets algorithm [KD99a, DR03]. BnB algorithms have recently been claimed to be the state-of-the art method in MPE solving [MKD03], but as we show in this thesis, their performance also degrades quite rapidly with increasing induced width. We attribute this to the impaired guidance of the underlying Mini-Buckets heuristic.

One way to cope with this restriction of exact algorithms is to avoid networks with high induced widths and approximate them by networks of lower induced width [Kjæ94, BJ02]. In this thesis, we study a different approach to deal with networks of high induced width, namely to employ incomplete MPE algorithms which quickly solve most problem instances to optimality, but which are not able to proof this optimality. We show that these algorithms clearly outperform the best available BnB algorithm s-BBMB for a wide range of instances. Even more importantly, we demonstrate empirically that the efficiency of the novel algorithms we introduce does not depend on the problem's induced width; and that for each setting of s-BBMB's so-called i-bound, s-BBMB scales poorly with one or more of the instance characteristics "number of variables", "maximal domain size", and "maximal degree of the independence graph".

More specifically, the MPE algorithms we study are Stochastic Local Search (SLS) algorithms [HS04]. SLS algorithms are amongst the state-of-the-art in such heterogeneous research areas as propositional satisfiability (SAT) [SLM92, HTH02], weighted Max-SAT [MT00, SHS03], graph colouring [PS02], the travelling salesperson problem (TSP) [LK73, ACR03], scheduling [dBSD01], RNA secondary structure design [AFH$^+$04] and protein folding [SAHH02]. They are incomplete search methods rapidly moving through the space of possible solutions, changing single solution components at a time given only the local information available in a search state.

In this thesis, we concentrate on two particular subclasses of SLS algorithms, namely Dynamic Local Search (DLS) and Iterated Local Search (ILS) algorithms. DLS algorithms employ an evaluation function guiding the search that differs from the actual objective function. When a local optimum s of this evaluation function is reached, the function is modified in order to steer away from s in the future. DLS algorithms have been applied with especially great success for SAT [HTH02] and weighted Max-SAT [MT00]. Here, we focus on a particular DLS algorithm, namely Guided Local Search (GLS) [Vou97]. ILS is a general framework for achieving high coverage of the search space as well as intensified search in local optima regions. ILS iterates a three-phase process, in which a locally optimal solution is slightly modified followed by a mostly greedy local search yielding a new locally optimal solution. An acceptance criterion then decides about continuing the search from the old or the new local optimum. ILS algorithms are especially often used in the TSP domain and other domains prominent in Operations Research [dBSD01, LMS02, ACR03], but ILS is also amongst the best-performing approaches for Max-SAT [SHS03] and graph colouring [PS02].

SLS algorithms have been applied to the MPE problem before [KD99b, Par02], but none of the previous SLS algorithms pays sufficient attention to such important concerns as algorithmic complexity per search step, search stagnation, and thorough parameter tuning. Consequentially, these previous SLS algorithms have been shown to be outperformed by systematic search algorithms like Branch-and-Bound with Mini-Buckets heuristic [MKD03]. In this thesis, we remove the shortcomings of previous SLS algorithms for MPE, improving their speed by up to six orders of magnitude (i.e. by up to a factor of $1,000,000$); this yields a performance significantly higher than the one of the systematic search algorithms s-BBMB (with optimal i-bound) and Anytime MB, an anytime variant of the prominent Mini-Buckets algorithm [DR03]. Our novel algorithms also find variable instantiations which are up to 280 orders of magnitude more likely than the ones found by the previously best-performing SLS algorithms. These enormous performance gains are due to a number of components we develop and study in this thesis. Amongst the main contributions of this thesis are:

- Two novel caching schemes and the first detailed complexity analysis of single search steps of SLS algorithms for MPE. This yields a speedup for all SLS algorithms that lies between three and 116 in our experiments and increases for harder problem instances.

- The first ILS algorithm for MPE. This algorithm outperforms the previously best-performing non-penalty based algorithm G+StS [KD99b] by up to 4 orders of magnitude and finds variable instantiations of up to 20 orders higher probabilitiy than the ones found by G+StS.

- A simple parameter tuning for GLS [Par02], the previously best-performing SLS algorithm for MPE. This enables GLS to find variable instantiations that are up to 100 orders of magnitude more probable, and further enables it to quickly solve many problem instances previously unsolvable.

- An improved version of GLS, called GLS$^+$, that incorporates the logarithmic objective function into the evalutation function and outperforms our already much improved version of GLS by orders of magnitude, both in terms of solution quality found and runtime to find the optimal solution.

- A hybrid algorithm combining ILS, GLS$^+$, and a new variant of Mini-Buckets we call MB*. This exploits the fact that the performance of GLS$^+$ and ILS is not highly correlated, and that MB* yields very strong results for networks of low induced width. We demonstrate this hybrid algorithm to consistently perform very well and show that, on average, it outperforms all other algorithms, including systematic search algorithms.

- A systematic empirical study of the impact various characteristics of MPE problem instances have on problem hardness for a number of MPE algorithms. Separate experiments for each characteristic show that s-BBMB with low i-bounds scales poorly with the number of variables as well as the degree of the independence graph; and that Anytime MB and s-BBMB with high i-bounds scale poorly with the domain size. Finally, for an increasing induced width, the empirical runtime of SLS algorithms is not affected at all, while the performance of s-BBMB and Anytime MB degrades considerably.

We compare our new algorithms GLS$^+$, ILS, and the hybrid algorithm against the best performing available complete algorithm s-BBMB [KD99a, MKD03]) and against Anytime MB [DR03]. For this comparison, we employ real-world MPE problem instances from the Bayesian network repository[1] and random instances created with BNGenerator [IC02, IC03]. On all real-world instances but one with high induced width, Anytime MB performs very well, only matched in performance by our hybrid algorithm. However, on the randomly generated instances, we show Anytime MB's very poor performance on networks of maximal induced width 20. The performance of s-BBMB differs much from instance to instance. For many instances, especially such with high induced width, s-BBMB is only feasible with low i-bounds but usually yields poor results in this case. For the instances for which higher i-bounds are feasible, it generally yields better results. Also, for structured instances, s-BBMB performs much better than for randomly generated ones.

The remainder of this thesis is structured as follows. In Chapter 2, we introduce our notation and formally define the MPE problem. Chapter 3 covers some general principles of Stochastic Local Search and Chapter 4 introduces previous SLS algorithms and other important algorithms for MPE. Then, we cover the development of our novel algorithms: Chapter 5 introduces all algorithms we implemented, Chapter 6 shows how they can be implemented efficiently, and Chapter 7 focusses on maximizing their performance by a thorough parameter tuning. This is followed by the experimtal evaluation of our novel algorithms. Chapter 8 demonstrates the enormous improvements our SLS algorithms yield over the previously best-performing SLS algorithms and studies the contribution of each novel component in our algorithms to this effect. Chapter 9 demonstrates our algorithms to outperform the previous state-of-the-art in MPE solving, and shows preferable scaling behaviour with an increasing number of variables, domain size, degree, and induced width of the independence graph. Chapter 10 concludes the thesis and indicates promising directions for future work.

[1]http://www.cs.huji.ac.il/labs/compbio/Repository/

Chapter 2

Problem Statement

In this chapter, we lay the foundation for the remainder of the thesis. We formally introduce Bayesian networks and the MPE problem, cover applications of MPE and similar tasks in Bayesian networks, and describe the problem instances we use in our experimental analysis.

2.1 Bayesian Networks

Since the early days of Artificial Intelligence (AI) research, many different representations of knowledge in computer systems have been advocated [Bib93]. One classical representation are rule-based systems which encode knowledge in a set of deterministic if-then rules. Although this approach had some success, researchers became aware of the fact that many domains can not be modeled in a purely deterministic fashion and require the capability of dealing with uncertainty [RN03]. The use of probability theory was considered by AI researchers, but at the time the dominant interpretation of probability was the frequentist approach [CDLS99, RN03] which did not allow an application to the kind of uncertain events AI research was interested in. Moreover, there were no efficient algorithms to compute the probabilities of interest and so probability theory was not incorporated into early AI systems. Instead, the AI community was lead to develop alternative approaches like rule-based systems with certainty factors [SDA$^+$75] or fuzzy logic [Zad83].

The last two decades, however, have seen an increasingly strong support for the probabilistic point of view in the AI community, boosted by three developments:

- the growing support for the subjectivist approach to probability as opposed to the frequentist interpretation [CDLS99, RN03];

- the development of modular representations of high-dimensional joint probability distributions, such as Bayesian (belief) networks [Pea88]; and
- the development of efficient algorithms to manipulate probabilistic networks and query probabilities of interest from them [Pea88, LS88, Dec96, CDLS99].

Bayesian (belief) networks [Pea88] have meanwhile become the prime representation for uncertainty in AI. They are at the core of this thesis and we will define them formally later in this section.

In the subjectivist probabilistic point of view, variables in the real world can be modeled as random variables even if their value is perfectly deterministic but we simply do not *know* it. Informally, a *random variable V* is an unknown variable which can take on one of a domain D_V of values.[1] In this thesis, we will concentrate on purely discrete variables V with finite domain D_V. Each random variable V has a *prior probability distribution* $P(V)$ which defines the probability $P(V = v)$ for each possible value $v \in D_V$. The probabilities $P(V = v)$ are required to sum to 1 over all values $v \in D_V$. Assigning a value $v \in D_V$ to V results in the *variable instantiation* $V = v$. If no confusion is possible, we will simply use the lower case letters to refer to variable instantiations.

A simple example for a random variable is the variable $Rain$ with the domain $\{yes, no\}$ and the prior $P(Rain = yes) = 0.5$.[2] The prior $P(Rain)$ defines the probability of rain in the absence of any other information. If we look up to the sky and observe that it is cloudy (the random variable $Cloudy$ is instantiated to yes), our belief about $Rain$ is influenced, which can be represented by the *conditional probability* $P(Rain|Cloudy = yes)$; this might be significantly different from the prior, for example $P(Rain = yes|Cloudy = yes) = 0.8$. Since we are only dealing with discrete random variables, we can represent conditional probability distributions as tables. A *conditional probability table (CPT)* $P(V_n|V_1, \ldots, V_{n-1})$ holds for each variable instantiation $v_1, \ldots, v_{n-1}$ the conditional probabilities $P(V_n = v_n|v_1, \ldots, v_{n-1})$ for all values $v_n \in D_{V_n}$ of the dependent variable V_n;[3] for each instantiation $v_1, \ldots, v_{n-1}$, the probabilities $P(V_n = v_n|v_1, \ldots, v_{n-1})$ are required to sum to 1 over all possible values v_n of V_n.

[1] Formally, a random variable is defined as a measurable function from a probability space to some measurable space, usually the real numbers with the Borel σ-algebra. Since these concepts do not aid our discussion, we refer the interested reader to the mathematical literature [LW99].

[2] Since the probabilities over all the possible values of a random variable always sum to one, this implies $P(Rain = no) = 0.5$. The rain example is based on the sprinkler network in [RN03].

[3] Technically, it suffices for the CPT to hold all but one such probabilities since the last one computes as the difference of the sum over the others to 1.

C	P(C)
f	0.5
t	0.5

×

C	R	P(R|C)
f	f	0.8
f	t	0.2
t	f	0.2
t	t	0.8

=

C	R	P(R, C)
f	f	0.4
f	t	0.1
t	f	0.1
t	t	0.4

Figure 2.1: Prior probability table $P(Cloudy)$ (left), conditional probability table $P(Rain|Cloudy)$ (middle) and joint probability table $P(Rain, Cloudy)$ (right).

The *joint probability distribution* $P(V_i, V_j)$ is a function which assigns a probability to each instantiation v_i, v_j. Like prior and conditional probability distributions, it can be represented in table form, in this case all entries summing to one. Figure 2.1 shows the prior probability table $P(Cloudy)$, the conditional probability table $P(Rain|Cloudy)$ and the joint probability table $P(Rain, Cloudy)$ which computes as $P(Cloudy) \times P(Rain|Cloudy)$. The general concept of potentials subsumes both conditional and joint probability tables in that no restriction applies to the sum over a potential's entries:

Definition 2.1.1 (Potential). Given a set of variables $\mathcal{V}_\phi = \{V_1, \ldots, V_m\}$, a *potential* ϕ over the variables $\mathcal{V}_\phi$ is a function that assigns some real number $\phi[\mathcal{V}_\phi = v_\phi] \geq 0$ to each instantiation v_ϕ of its variables. The potential has *scope* $\mathcal{V}_\phi$ and is said to *span* the variables in its scope. A potential's *size* S_ϕ is the product of its variables' domain sizes:

$$S_\phi = \prod_{V_i \in \mathcal{V}_\phi} |D_{V_i}|.$$

We use potentials in this thesis in order to prevent having to distinguish the different normalizations of conditional and joint probabilities. Potentials are heavily used in the literature on junction trees (see e.g. [CDLS99]); they have also been called factors [RN03] or simply functions [Dec96] in other areas of AI.

Every probabilistic query about a set of random variables $\mathbf{V}$ can be answered with the help of a joint probability distribution over $\mathbf{V}$. The problem with joint probability tables – or potentials in general – is that their number of entries (i.e. their size), grows exponentially in their number of variables.

Since we can neither store, nor efficiently manipulate, nor intuitively grasp the meaning of huge potentials, a modular representation in form of smaller potentials becomes necessary. This modularity can, for example, be achieved by organizing the random variables in a structure called a Bayesian (belief) network [Pea88], which we will define after the introduction of a few concepts from graph theory that are at the core of Bayesian networks.

Definition 2.1.2 (Graph Concepts). A *graph* $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is a tuple consisting of a set of vertices or nodes $\mathcal{V}$ and a set of edges $\mathcal{E}$. Graphs can be directed or undirected. In this thesis, we concentrate on the directed version, where an edge $E = (V_i, V_j) \in \mathcal{E}$ represents a directed connection from node V_i to node V_j. *Acyclicity* of a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ means that no path $V_1, \ldots, V_n$ exists with $V_i \in \mathcal{V}$ for $i = 1, \ldots, n$, $(V_i, V_{i+1}) \in \mathcal{E}$ for $i = 1, \ldots, n-1$, $n > 1$, and $V_1 = V_n$. The *parents* $\mathrm{pa}(V_i)$ of a node $V_i \in \mathcal{V}$ are the variables with direct edges to V_i: $pa(V_i) = \{V_j \in \mathcal{V} | (V_j, V_i) \in \mathcal{E}\}$. A node V_i's family *fam*(V_i) contains V_i itself and its parents *pa*(V_i); and its *children* $\mathrm{ch}(V_i)$ are all nodes V_i has a direct edge to: *ch*$(V_i) = \{V_j \in \mathcal{V} | (V_i, V_j) \in \mathcal{E}\}$. The set of *neighbours* of a node is the union of its children and parents. The *descendants* of a variable V are – in analogy to human relationships – recursively defined as the union of the children of V and the children's descendants. Finally, the *Markov blanket* $\mathrm{mb}(V_i)$ of a node V_i is the union of its parents, its children and its children's parents in $\mathcal{G}$: *mb*$(V_i) =$ *pa*$(V_i) \cup$ *ch*$(V_i) \cup$ *pa*$($*ch*$(V_i))$.

Bayesian networks use directed acyclic graphs to describe qualitative interactions between a set $\mathbf{V}$ of random variables of interest. Each variable $V_i \in \mathbf{V}$ is associated with a node in the graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. For sake of a light notation, we identify graph nodes and variables here and in the rest of the thesis, denoting the graph directly by $\mathcal{G} = (\mathbf{V}, \mathcal{E})$. An edge $E = (V_i, V_j) \in \mathcal{E}$ then represents a direct dependency of variable V_j on variable V_i.

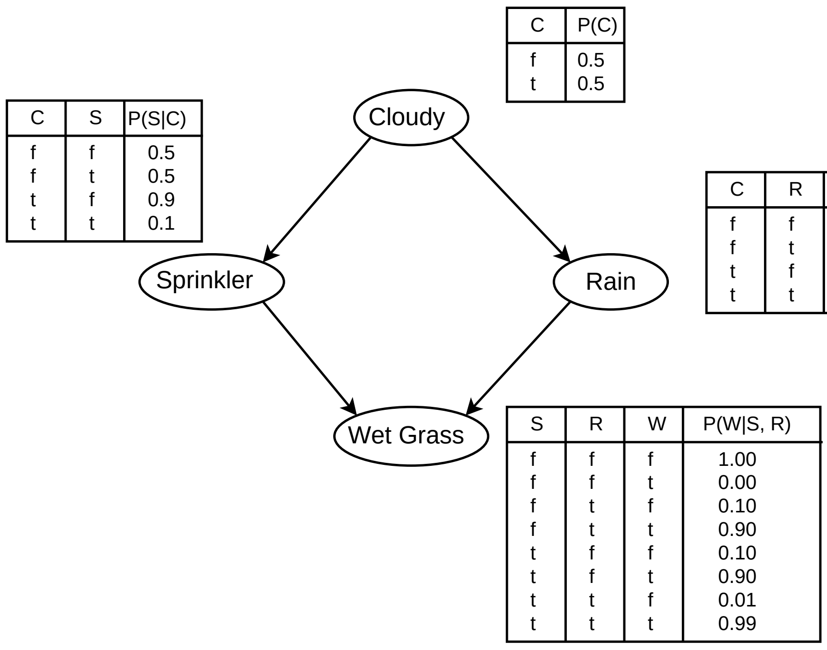
Definition 2.1.3 (Bayesian Network). A *Bayesian network* $\mathcal{B}$ is a quadruple $\langle \mathbf{V}, \mathbf{D}, \mathcal{G}, \Phi \rangle$, where

- $\mathbf{V}$ is an ordered set of random variables,
- $\mathbf{D}$ is an ordered set of finite domains D_{V_i} for each $V_i \in \mathbf{V}$,
- $\mathcal{G} = (\mathbf{V}, \mathcal{E})$ is a directed acyclic graph (DAG), also called the network's *independence graph*, and
- Φ is an ordered set of CPTs $\phi_{V_i} = P(V_i|$*pa*$(V_i))$, specifying the conditional probability distribution of each $V_i \in \mathbf{V}$ given its parents in $\mathcal{G}$.

The semantic of a discrete Bayesian network $\mathcal{B} = \langle \mathbf{V}, \mathbf{D}, \mathcal{G}, \Phi \rangle$ is that it compactly represents a joint probability table ϕ over its variables $\mathbf{V}$ in a factored way:

$$\phi = P(\mathbf{V}) = \prod_{V_i \in \mathbf{V}} P(V_i | pa(V_i)) = \prod_{V_i \in \mathbf{V}} \phi_{V_i}.$$

Figure 2.2 on the next page shows the sprinkler network, a simple Bayesian network from [RN03], part of which we used in our previous $Rain$ example. Note



C	P(C)
f	0.5
t	0.5

C	S	P(S|C)
f	f	0.5
f	t	0.5
t	f	0.9
t	t	0.1

C	R	P(R|C)
f	f	0.8
f	t	0.2
t	f	0.2
t	t	0.8

S	R	W	P(W|S, R)
f	f	f	1.00
f	f	t	0.00
f	t	f	0.10
f	t	t	0.90
t	f	f	0.10
t	f	t	0.90
t	t	f	0.01
t	t	t	0.99

Figure 2.2: Simple Bayesian network `sprinkler`. Adapted from [RN03].

that the probability for the grass being wet is not directly affected by whether it is cloudy or not; if we already know the status of variables *Rain* and *Sprinkler*, learning the status of variable *Cloudy* does not change our belief on *Wet*; this is because *Cloudy* can only directly affect our beliefs on *Rain* and *Sprinkler*, but we already know their instantiations with certainty. This characteristic is formally captured by the concept of *conditional independence*: *Wet* is conditionally independent of *Cloudy* given *Rain* and *Sprinkler*.

The independence graph $\mathcal{G}$ of a Bayesian network $\mathcal{B} = \langle \mathbf{V}, \mathbf{D}, \mathcal{G}, \Phi \rangle$ compactly encodes a set of conditional independence relationships among $\mathcal{B}$'s variables (see, e.g., [RN03] for a more detailed explanation of these results):

- A random variable in a Bayesian network is conditionally independent of its non-descendants given its parents.

- A random variable in a Bayesian network is conditionally independent of all the other variables given its Markov Blanket.

In a Bayesian network $\mathcal{B} = \langle \mathbf{V}, \mathbf{D}, \mathcal{G}, \Phi \rangle$ the set of variables which occur together with a variable $V_i \in \mathbf{V}$ in any CPT $\phi \in \Phi$ is exactly its Markov blanket *mb*(V_i). This will become important in our discussion of caching in local search algorithms (see Chapter 6) since the effects of changing the value of a variable V_i remain local in the sense of directly affecting only the variables in *mb*(V_i).

2.2 The Most Probable Explanation Problem

Bayesian networks are used to represent joint probability distributions over many variables in a compact way. Although this representation might already help researchers in their work, this is not their main purpose. Rather, they are constructed in order to answer various probabilistic queries about their variables efficiently. They are very flexible in that they can incorporate evidence that is acquired for any of their variables and answer questions about any subset of their variables conditional on this evidence. One particular problem in Bayesian networks is to determine the most likely instantiation of all variables that is consistent with some fixed evidence variables.

Formally, two partial variable assignments $\mathbf{v_1} = v_{i,1}, \ldots, v_{j,1}$ and $\mathbf{v_2} = v_{k,2}, \ldots, v_{l,2}$ are *consistent* if and only if $\mathbf{v_1}$ and $\mathbf{v_2}$ agree on their shared variables.[4] A variable assignment $\mathbf{V} = \mathbf{v}$ which assigns a value to all variables $\mathcal{V}_\phi$ of a potential ϕ (i.e. $\mathcal{V}_\phi \subseteq \mathbf{V}$) is consistent with exactly one instantiation of ϕ's variables. We denote ϕ's entry for this consistent instantiation by $\phi[\mathbf{V} = \mathbf{v}]$. This is equal to $\phi[\mathcal{V}_\phi = v_\phi]$ where v_ϕ is the unique instantiation of $\mathcal{V}_\phi$ that is consistent with $\mathbf{v}$.

Now we have the necessary ingredients to define the Most Probable Explanation Problem formally:

Definition 2.2.1 (Most Probable Explanation Problem). Given a Bayesian network $\mathcal{B} = \langle \mathbf{V}, \mathbf{D}, \mathcal{G}, \Phi \rangle$ and a set of evidence variables $\mathbf{E} = \mathbf{e}$, the *Most Probable Explanation (MPE)* problem is to find an instantiation $\mathbf{V} = \mathbf{v}$ with maximal probability $\prod_{\phi \in \Phi} \phi[\mathbf{V} = \mathbf{v}]$ over all variable instantiations $\mathbf{v}$ consistent with the evidence $\mathbf{e}$.[5]

The MPE problem can also be defined for other graphical models like Markov networks and chain graphs [CDLS99]. Basically, all that one needs is a set of variables $\mathcal{V}$ and a set of potentials spanning subsets of $\mathcal{V}$. This generalization does not

[4]Here and in the rest of this thesis, we use bold font for sets of variables and variable instantiations; we continue to use capital letters for variables and lower-case letters for variable instantiations.

[5]The MPE problem has also been defined as finding the best instantiation of all non-evidence variables $\mathbf{V} \setminus \mathbf{E}$ given the evidence $\mathbf{E} = \mathbf{e}$, but since $P(\mathbf{V} \setminus \mathbf{E}, \mathbf{e}) = P(\mathbf{V} \setminus \mathbf{E}, \mathbf{e} | \mathbf{E} = \mathbf{e}) \times P(\mathbf{E} = \mathbf{e}) \propto P(\mathbf{V} \setminus \mathbf{E} | \mathbf{E} = \mathbf{e})$, these formulations are equivalent.

conflict with our approaches for solving the problems and all the methods suggested in this thesis also apply for MPE in general graphical models.

Regarding computational complexity, MPE is a hard combinatorial optimization problem in the sense that finding the best instantiation is $\mathcal{NP}$-hard. The decision version, to decide whether there exists an instantiation with probability greater or equal to a given bound, is $\mathcal{NP}$-complete [SD03].

2.3 Applications of MPE and Similar Tasks

Interesting problems in a variety of heterogeneous research areas can be modeled as Bayesian networks, or, more generally, as graphical models. Often, the problem at hand can then be formulated as solving the Most Probable Explanation problem in the graphical model. We shortly cover two interesting examples for this.

At IBM Research, Fault Diagnosis in computer networks has been rephrased as inference in Bayesian networks [RBM02b, RBM02a]. In their model, each router, server, or workstation can be either operational or malfunctioning, and each of these is represented by an unobserved node in a Bayesian network. A number of routes through the network are probed and failure or success of these probes represent the network's Boolean evidence variables. The probability for a probe to succeed is taken to be a noisy-OR of its parents in the Bayesian network, i.e. the computer network nodes the probe had to pass. Together with prior probability distributions for the operational status of the network nodes and a two-layer independence graph, the Bayesian network is fully specified and finding the most likely faults is then equivalent to solving the MPE problem.

Recently, it has been shown that an important subproblem in Protein Folding can also be cast as finding the MPE in a graphical model [YW03]. A protein consists of a backbone of amino acid units connected by peptide bonds. Each amino acid unit, also called *residue*, has a *side-chain* attached to it. In the side-chain prediction problem, the protein backbone structure is given, and for each residue, the spatial conformation of its side-chain is to be determined. Each conformation consists of a quadruple of continuous angles, which is discretized to a small number of preferred energy conformations, so-called *rotamers*, by means of a standard library. The protein's residues then make up the variables of a graphical model with the rotamers as possible values. The standard van der Waals energy function yields the free energy achieved by complete assignments of rotamers to the residues and computes as a sum of pairwise interactions between residues which are close in 3D space. Denoting the interaction between two residues i and j as a potential with scope $\{i, j\}$, this yields a sparse graphical model with many small loops. Finding the most probable side-chain for the fixed backbone structure is then equivalent to solving the MPE task in this model.

Other application areas of MPE include *medical diagnosis* to find a patient's most likely disease given some clinical findings [JJ99]; *probabilistic decoding* to find the most probable message transmitted over a noisy channel given the channel output [DR03]; computer vision to calculate stereo disparities [TF03]; biological sequence analysis to find the most likely alignment of two sequences[DEKM98].

A straightforward generalization of MPE is *M-MPE* [CDLS99, YW04], the task to find the M most probable instantiations consistent with the evidence. The algorithms we introduce for MPE in this thesis are easily generalizable to M-MPE and perfectly set to challenge the state-of-the art for solving this problem since the only additional complexity in generalizing our algorithms would lie in some book-keeping of the M best solutions found thus far. We plan to implement this in future work.

Another generalization of MPE is the *Maximum a Posteriori Hypothesis (MAP)* problem, in which the task is to compute the most probable partial instantiation of a subset $\mathbf{W} \subseteq \mathbf{V}$ of variables, summed over all instantiations of $\mathbf{V} \setminus \mathbf{W}$. Note that the MAP assignment $\mathbf{W} = \mathbf{w}$ is not necessarily consistent with the MPE assignment $\mathbf{V} = \mathbf{v}$, but may have a probability considerably higher than the partial assignment to $\mathbf{W}$ that is consistent with $\mathbf{v}$. The novel algorithms we introduce in this thesis can also be applied to search for a MAP assignment $\mathbf{W} = \mathbf{w}$. However, especially when there are many variables in $\mathbf{V} \setminus \mathbf{W}$, efficient methods need to be employed to sum over all these variables in order to compute the probability of a MAP assignment.

A problem in Bayesian networks for which we cannot apply the algorithms introduced in this thesis is *belief updating*, the task to compute the joint probability distribution over arbitrary subsets $\mathbf{W} \subseteq \mathbf{V}$ of variables given evidence $\mathbf{E} = \mathbf{e}$. High performing SLS algorithms are generally not suitable to compute unbiased probability distributions since for optimization problems like MPE, they usually strive to predominantly sample very high-quality solutions. Thus, if used in a sampling-based approximation they will always yield highly biased estimates. Unbiased approaches like Stochastic Simulation [Pea88] perform well for this kind of tasks, but show extremely weak performance for optimization problems (cf. Section 4.1 on page 25).

2.4 Benchmark Instances

In order to evaluate our approaches for solving the MPE problem and to compare them to previous algorithms, we created various sets of problem instances.

Firstly, we use a set of real-world instances, `bnrep`, from the Bayesian network repository[6]. We employ `bnrep` for our experimental analysis since it is the closest to a standard benchmark set among all sets of MPE problem instances researchers use to evaluate their algorithms, and since it comprises real-world instances of heterogeneous problem domains. Many other types of networks have been used for experimental analysis. Amongst those are networks from medical diagnosis [Hec90], protein folding [YW03, YW04], networks from fault diagnosis in computer networks [RBM02a], and coding networks (see, eg, [DR03]).[7] A priori, it is not clear that the insights gained for some of these domains carry over to domains with fundamentally different characteristics; we plan to study this in future work by applying our algorithms to networks in a variety of the aforementioned areas.

As a first step, in this thesis, we systematically study the impact single network characteristics have on problem hardness for the various algorithms. We control these characteristics by generating appropriate networks with BNGenerator [IC02, IC03], a tool which creates random Bayesian networks with a predefined number of variables and constraints on maximal domain size, degree, and induced width of the network's independence graph. In Chapter 9, we perform scaling studies with sets of random networks in which only one of these parameters changes at a time. In separate experiments, we study the impact of a network's number of variables, its maximal domain size, degree, and induced width of the network's independence graph. In order to be able to relate these scaling studies to the results on problem set `bnrep`, we perform the same experiments as carried out on `bnrep` on a representative set of randomly generated instances which we refer to as problem set `gen`.

We denote networks from `bnrep` by their name in the repository, for example `mildew`. For the networks in `gen`, we put all constraints used for their generation into their name; e.g., network `z100v3d5iw5` denotes a network with 100 vari-

[6]http://www.cs.huji.ac.il/labs/compbio/Repository/

[7]Unfortunately, many MPE problem instances are not freely available. Also, often random MPE instances are newly generated on the fly for each experiment from a probability distribution. While on the positive side this allows the researcher to focus on distributions of problem instances rather than on just a few instances, it also much complicates the comparison of results. Another point that complicates comparison is that there exists an abundance of formats for Bayesian networks. One can only hope that some standard format like for example the Bayesian interchange format will displace the other formats someday.

ables, maximal domain size 3, maximal degree 5 and maximal induced width 5. All networks in `gen` have maximal degree 5, but vary in their number of variables z, their maximal domain size v, and their maximal induced width iw. Problem set `gen` contains exactly one instance for each combination of $z \in \{100, 200, 400\}$, $v \in \{3, 6\}$, and $iw \in \{10, 20\}$. In Table 2.1 on the next page and Table 2.2 on page 16, we give an overview of several instance characteristics for all instances in problem sets `bnrep` and `gen`, respectively. Note that column "Min-Degree Width" in these tables gives the induced width as computed by our implementation of the Anytime Mini-Buckets algorithms described later in Section 4.3 on page 32. For the generated instances in problem set `gen`, this width in some cases grows slightly larger than the constraint on induced width used for its generation. We attribute this to a different tie-breaking in otherwise equivalent algorithms for computing the induced width.[8]

For each network in `bnrep`, we consider the original network as well as a network with the original independence graph but randomly sampled CPTs. We denote these modified variants by attaching `-rand` to the network's name. For the networks in `gen`, we either use random CPTs (attaching `-rand` to the name) or, in order to approximate quantitative structure, sample the CPTs from the original networks in `bnrep` (in this case attaching `-struc` to the network name). In order to sample the CPT ϕ_{V_i} for a variable V_i with domain size x, we independently sample the 1-dimensional probability distributions $P(V_i|\mathit{pa}(V_i))$ for each instantiation of $\mathit{pa}(V_i)$. We call such a 1-dimensional probability distribution a *row* of ϕ_{V_i}. Each such row is sampled uniformly from all rows of the CPTs ϕ_V of all variables V with domain size x in all original networks of `bnrep`.

Problem instances for MPE are created for each of the aforementioned Bayesian networks by topologically sampling all the network's variables given their parents and then fixing 10 of these sampled variables as evidence. This procedure guarantees the evidence to be strictly positive.

Having introduced the MPE problem and the instances we employ for our experimental evaluation, in the next Chapter we move on to describe our favourite approach for solving the MPE problem, namely Stochastic Local Search methods.

[8]Note that computing the exact induced width of a graph is an $\mathcal{NP}$-hard problem [Dec96]. Because of this, BNGenerator [IC03] uses the min-degree heuristic to generate an ordering, the induced width along which is used as an upper bound on the actual induced width. The min-degree heuristic greedily constructs an ordering by iteratively removing the graph's node with minimal degree, connecting all neighbours of a node when it is removed. Although our implementation employs the min-degree heuristic to generate an ordering as well, it is not guaranteed to yield identical results. This is because frequently during the construction, there are several nodes with equal minimal degree, and the algorithms may differ in their tie-breaking mechanism.

Instance	N	Dom size	VV	CPT size	Pot. entries	Degree	MB size	Min-degree	
								Width	Size
alarm	27	2.84(0.73) [2, 4]	79	20.32(27.46) [2, 108]	549	2.49(1.35) [1, 6]	4.51(2.08) [2, 9]	4	8.10e+01
barley	38	8.77(9.05) [2, 67]	343	2712.08(7498.15) [2, 40320]	1.03e+05	3.50(1.76) [1, 8]	6.25(2.81) [3, 14]	6	3.01e+05
diabetes	403	11.34(5.88) [3, 21]	4543	1116.39(1683.48) [5, 7056]	4.5e+05	2.92(1.71) [1, 24]	4.97(2.65) [3, 49]	6	8.58e+07
hailfinder	46	3.98(1.72) [2, 11]	185	66.80(164.69) [3, 1188]	3073	2.36(2.40) [1, 17]	4.54(2.70) [2, 18]	5	1.33e+03
insurance	17	3.30(0.99) [2, 5]	55	52.56(53.31) [3, 200]	894	3.85(2.03) [1, 9]	6.19(2.54) [2, 11]	3	1.25e+02
link	714	2.53(0.83) [2, 4]	1806	28.32(44.63) [2, 128]	20219	3.11(2.49) [0, 17]	5.80(4.40) [1, 32]	20	2.75e+11
mildew	25	17.60(27.01) [3, 100]	262	15633.09(57395.13) [3, 280000]	3.91e+05	2.63(1.54) [1, 5]	5.57(2.05) [2, 10]	4	1.60e+05
munin1	179	5.26(3.59) [1, 21]	960	102.99(122.70) [2, 600]	18437	2.98(2.26) [1, 15]	4.87(2.75) [2, 19]	11	6.05e+07
munin2	993	5.36(3.67) [2, 21]	5335	83.67(112.24) [2, 600]	83084	2.48(2.00) [1, 30]	4.31(2.29) [2, 31]	8	5.76e+06
munin3	1034	5.37(3.67) [1, 21]	5554	82.24(106.18) [2, 600]	85033	2.52(3.17) [1, 69]	4.34(3.44) [2, 74]	8	5.76e+06
munin4	1031	5.43(3.64) [1, 21]	5567	94.32(115.81) [2, 600]	97240	2.68(3.12) [1, 69]	4.54(3.45) [2, 74]	9	4.04e+07
pigs	431	3.00(0.00) [3, 3]	1293	19.11(11.29) [3, 27]	8236	2.68(2.57) [1, 41]	4.66(4.10) [3, 69]	12	5.31e+05
water	22	3.62(0.49) [3, 4]	79	421.38(894.70) [3, 3072]	9271	4.12(1.90) [1, 8]	8.69(3.62) [2, 14]	9	6.55e+04

Table 2.1: Characteristics of instances in problem set `bnrep`. “Dom size”, “CPT size”, “Degree”, and “MB size” summarize characteristics of all variables or potentials. Their format is average(standard deviation) in the first row and [minimum, maximum] in the second row. “N” denotes the number of non-evidence variables (we always use 10 variables as evidence); “Dom size” the variables’ domain sizes; “VV” the total number of variable-value pairs $\sum_{V \in \mathbf{V}} |D_V|$; “CPT size” the sizes of the CPTs; “Pot. entries” the total number of potential entries; “Degree” the variables’ degree in the independence graph; “MB size” the size of the variables’ Markov blanket; and “Min-degree” “Width” and “Size” the induced width and size along the min-degree ordering.

Instance	N	Dom size	VV	CPT size	Pot. entries	Degree	MB size	Min-degree	
								Width	Size
z100v3d5iw10	90	2.57(0.50) [2, 3]	229	42.70(73.91) [2, 486]	3843	4.34(0.81) [2, 5]	8.86(3.21) [3, 17]	10	5.90e+04
z100v3d5iw20	90	2.57(0.50) [2, 3]	233	44.84(68.27) [2, 486]	4036	4.46(0.67) [3, 5]	9.32(3.14) [4, 16]	18	1.29e+08
z100v6d5iw10	90	3.86(1.32) [2, 6]	344	488.25(1487.03) [2, 8640]	43943	4.38(0.83) [2, 5]	9.22(3.06) [3, 17]	13	1.31e+10
z100v6d5iw20	90	4.12(1.42) [2, 6]	372	524.59(948.13) [2, 4500]	47214	4.66(0.55) [3, 5]	10.16(3.48) [5, 18]	20	6.09e+14
z200v3d5iw10	190	2.54(0.50) [2, 3]	483	44.66(77.11) [2, 486]	8485	4.41(0.83) [2, 5]	9.08(3.24) [3, 18]	12	5.31e+05
z200v3d5iw20	190	2.53(0.50) [2, 3]	479	40.27(55.96) [2, 486]	7653	4.47(0.74) [2, 5]	9.26(2.97) [3, 17]	20	3.49e+09
z200v6d5iw10	190	4.08(1.48) [2, 6]	781	533.34(1723.32) [2, 18000]	1.01e+05	4.41(0.83) [2, 5]	9.08(3.24) [3, 18]	12	2.18e+09
z200v6d5iw20	190	4.08(1.35) [2, 6]	777	344.08(1229.81) [3, 12960]	65376	4.25(0.97) [1, 5]	8.45(2.99) [3, 17]	22	3.66e+15
z400v3d5iw10	390	2.48(0.50) [2, 3]	965	22.12(33.98) [2, 324]	8627	3.66(1.21) [1, 5]	6.82(2.87) [2, 16]	15	1.59e+06
z400v3d5iw20	390	2.49(0.50) [2, 3]	969	23.36(33.89) [2, 324]	9112	3.80(1.21) [1, 5]	7.20(3.09) [2, 17]	20	3.49e+09
z400v6d5iw10	390	4.11(1.39) [2, 6]	1604	155.89(340.11) [2, 3240]	60798	3.64(1.21) [1, 5]	6.78(2.90) [2, 15]	14	2.18e+09
z400v6d5iw20	390	3.89(1.44) [2, 6]	1509	145.13(506.13) [2, 7776]	56602	3.57(1.27) [1, 5]	6.71(3.06) [2, 17]	24	7.90e+17

Table 2.2: Characteristics of instances in problem set `gen`. "Dom size", "CPT size", "Degree", and "MB size" summarize characteristics of all variables or potentials. Their format is average(standard deviation) in the first row and [minimum, maximum] in the second row. "N" denotes the number of non-evidence variables (we always use 10 variables as evidence); "Dom size" the variables' domain sizes; "VV" the total number of variable-value pairs $\sum_{V \in \mathbf{V}} |D_V|$; "CPT size" the sizes of the CPTs; "Pot. entries" the total number of potential entries; "Degree" the variables' degree in the independence graph; "MB size" the size of the variables' Markov blanket; and "Min-degree" "Width" and "Size" the induced width and size along the min-degree ordering.

Chapter 3

Stochastic Local Search

In this chapter, we define the basic concepts of Stochastic Local Search (SLS) in general and for the Most Probable Explanation Problem in particular. The general part is based on the recent book [HS04], where much more details, applications and valuable insights for the development and analysis of SLS algorithms can be found.

3.1 Combinatorial Problems

Combinatorial problems, such as finding assignments of discrete values to a finite set of objects, or finding groupings or orderings of objects, arise in a variety of fields, amongst others artificial intelligence, operations research, and bioinformatics [HS04, LMS02, AFH$^+$04]. In this chapter, next to MPE we use two very prominent examples of combinatorial problems to illustrate the concepts we introduce. In the *propositional satisfiability (SAT)* problem, one is asked to assign truth values to each variable in a finite set of Boolean variables, such that a given propositional formula involving these variables evaluates to true; in the *traveling salesperson problem (TSP)*, the task is to find a shortest round-trip visiting each of a finite set of cities exactly once. The TSP is an example for a combinatorial *optimization* problem as – next to the hard constraint of visiting each city exactly once – one strives to *minimize* the total length of the round-trip.

Many combinatorial problems including MPE, SAT, and TSP are computationally hard, meaning that no algorithms are known to date which scale subexponentially with problem size in the worst case. Like MPE, SAT and TSP are $\mathcal{NP}$-hard, i.e., they are at least as hard as any problem in the complexity class $\mathcal{NP}$. This class consists of all problems which are solvable on a *nondeterministic* Turing machine in a time polynomial in the problem size. However, a nondeterministic Turing machine is a fictitious construct that can so far only be simulated with exponential overhead on current computer architectures. For $\mathcal{NP}$-hard problems, no exact deterministic algorithm is known to date which solves all possible problem instances in time less than exponential in the instance size. It is also assumed by many researchers that such an algorithm can not exist in the first place since this would imply the equivalence of complexity classes $\mathcal{NP}$ and $\mathcal{P}$, where $\mathcal{P}$ consists of all problems that are solvable on a *deterministic* Turing machine in time polynomial in the problem size. The deterministic Turing machine is a construct equivalent to current programming languages in power, and it is assumed that the nondeterministic Turing machine is essentially more powerful.

One can distinguish two variants of combinatorial problems: the *decision variant*, in which one is only asked to decide whether a solution exists or not; and the *search variant*, which is to actually find a solution. For combinatorial optimization problems like MPE and TSP, the decision variant is defined as deciding whether a solution of a given *quality* (probability or tour length, respectively) exists. The *decision versions* of SAT, TSP, and MPE are $\mathcal{NP}$-complete, which means that these problems are members of $\mathcal{NP}$ and they are $\mathcal{NP}$-hard.

The $\mathcal{NP}$-hardness of problems like SAT, TSP, and MPE does not mean that these problems are not solvable in practice. On the contrary, many successful algorithms have been developed for these problems [SLM92, ZS00, HTH02, LMS02, Dec96, KD99a, RN03]. This means that the general worst case result for the whole class of possible instances of a problem does not necessarily apply for certain tractable subclasses. For MPE, this tractable subclass for example comprises Bayesian networks with low *induced width*, a measure introduced later in Definition 4.2.1 on page 27. If a problem instance is at hand which can not efficiently be solved by current algorithms, one might settle for an approximate suboptimal solution. Alternatively, Stochastic Local Search (SLS) algorithms can often be applied with great success. They have been shown to be very efficient on a great variety of combinatorial problems in such heterogeneous domains as SAT [SLM92, HTH02], weighted Max-SAT [MT00, SHS03], TSP [LK73, ACR03, JM02], graph colouring [PS02], scheduling [dBSD01], RNA secondary structure design [AFH$^+$04] and protein folding [SAHH02].

3.2 Basic Concepts of Stochastic Local Search

During the search for the optimal solution to a combinatorial problem, we may evaluate a lot of potential solutions, also called *candidate solutions*. These consist of combinations of *solution components* like instantiations of certain variables in

SAT and MPE or adjacency relationsships of particular cities in the sequence of cities describing a TSP round-trip. The definition of candidate solutions depends on the particular approach to search. In a local search that only evaluates complete assignments, we would define the candidate solutions to be all *complete* variable instantiations, whereas in a backtracking search for SAT these are rarely encountered and *partial* variable assignments would be used as candidate solutions instead.

The *search space* consists of all possible candidate solutions. It usually grows exponentially with problem size, and the task of SLS algorithms is to move through this huge search space using only local information for the decisions about their trajectories. More specifically, SLS algorithms work by starting the search from some initial candidate solution and iteratively moving through the search space by going from a candidate solution to a neighbouring candidate solution. This neighbourhood of a candidate solution can be defined arbitrarily, but in most applications, a *k-exchange neighbourhood* is used which simply means that two neighbouring solutions may differ in at most k solution components. For SAT and MPE, the 1-exchange neighbourhood is prominently used, whereas in the TSP 2- and 3-exchange neighbourhoods yield better performance. From the SAT domain stems the expression *variable flip* for a 1-exchange move and we will adopt the term for the MPE domain.

SLS algorithms do not tumble through the search space completely blindfolded. Instead, they employ an evaluation function offering some guidance. In most SLS algorithms, the evaluation function is simply taken to be the objective function.[1] In the TSP, the evaluation function to be minimized is usually just the tour length, given that the sequence of cities described by the adjacency relationships is indeed a roundtrip visiting each city exactly once.

In SAT, where the propositional formula F usually is a conjunction of disjunctions (i.e. $F = \bigwedge_i \bigvee_{j=1}^{n_i} l_{ij}$), the most prominent evaluation function is the number of unsatisfied disjunctions, or *clauses*, which needs to be minimized. Note that this is equivalent to *maximizing* the number of *satisfied* clauses. In MPE, the objective function to be maximized over all assignments $\mathbf{v}$ consistent with some evidence $\mathbf{e}$ is the probability $\prod_{\phi \in \Phi} \phi[\mathbf{V} = \mathbf{v}]$. As an evaluation function either this function is applied directly, or, for various reasons detailed in Section 3.6 on page 24, its logarithm is used.

The most basic local search method, called *best improvement* or *greedy descent*, simply determines the evaluation function value for all its neighbours and always

[1]However, there is a very promising subclass of SLS algorithms which dynamically changes the evaluation function, taking into account the search trajectory. We call algorithms in this subclass Dynamic Local Search algorithms and formulate the general approach in Algorithm scheme 3.1 on page 21.

moves to the neighbour with the best evaluation function value. This results in a usually rapid improvement that ends in a *local optimum*, a candidate solution which does not have a neighbour with better evaluation function value. If no special action is taken to drive the search away from a local optimum, best improvement will be stuck in it forever, although its objective function value might be much inferior to that of the global optimum. This is a form of *search stagnation*, and research in SLS algorithms has come up with many approaches to prevent it; we will cover a few of these in the next Section.

3.3 Escaping from Local Minima

While it is usually not a problem in SLS algorithms to reach local optima quickly, the key question is how to make the search explore the whole search space but still keep a strong bias towards regions with good objective function value. In most applications, this can be achieved by a good balance between *intensification* phases, in which the search is guided by an evaluation function strongly resembling the objective function, and *diversification* phases which help explore the search space.

Remarkably good results can already be obtained by restarting the search once a local optimum is reached, or – even much better – by introducing occasional *random moves*, also called *noise*, into an otherwise greedy search. Simple yet effective these techniques have made the first generation of SLS algorithms for SAT improve the state-of-the-art in SAT solving at the time of their development [SLM92, SKC94].

Another more recent development is based on augmenting the objective function used for guiding the local search. This type of algorithm has been called *Dynamic Local Search (DLS)* [HTH02, HS04], as the function used for evaluating candidate solutions changes dynamically during the search; the basic outline of DLS algorithms is given in Algorithm scheme 3.1 on the next page. For SAT solving, the tradition of DLS algorithms dates back to the Breakout method developed in 1993 [Mor93] and since then DLS algorithms have reached and became the state-of-the-art [MT00, HTH02].

Iterated Local Search (ILS) [SH01] is a general framework for achieving high coverage of the search space as well as intensified search in local optima regions. ILS algorithms are especially often used in the TSP domain and other domains prominent in Operations Research [dBSD01, LMS02], but one ILS algorithm is also amongst the best-performing algorithms for Max-SAT [SHS03]. Algorithm scheme 3.2 on the facing page describes the general ILS framework, consisting of four adaptable components. The search is initialized either randomly or by some

Algorithm scheme 3.1: Dynamic Local Search

```
1  $s \leftarrow$ GenerateInitialSolution
2  $eval \leftarrow$ Initial evaluation function
3  while Not TerminationCriterion() do
4      if $s$ is local optimum of $eval$ then
5          $eval \leftarrow$ Modify$(eval)$
6      else
7          $s \leftarrow$ Best improvement$(s, eval)$
```

Algorithm scheme 3.2: Iterated Local Search

```
8  $s_0 \leftarrow$ GenerateInitialSolution
9  $s^* \leftarrow$ LocalSearch$(s_0)$
10 while Not TerminationCriterion() do
11     $s' \leftarrow$ Pertubation$(s^*, history)$
12     $s^{*\prime} \leftarrow$ LocalSearch$(s')$
13     $s^* \leftarrow$ AcceptanceCriterion$(s^*, s^{*\prime}, history)$
```

heuristic, followed by a basic local search. Following this, a number of *iterations* are performed, where an iteration consists of changing some solution components of the current candidate solution (*Pertubation*), followed by a local search and the decision to keep or reject the newly obtained candidate solution (*AcceptanceCriterion*).

For *LocalSearch*, in principle any local search can be used, such that one can turn any local search algorithm into an ILS algorithm by plugging it into the framework and defining the other ILS operations. *Pertubation* is best designed to achieve a diversification of the search, ending up in a different local optima region after applying a local search to the perturbed candidate solution. *AcceptanceCriterion* decides upon the usefulness of the newly obtained candidate solution, where candidate solutions with better objective function value than the last one are considered useful; but also worse new candidates can be useful in order to prevent search stagnation. An extreme yet possible way to make the algorithm explore completely different regions of the search space is to perform a random restart in the acceptance criterion.

3.4 Caching: Exploiting Local Computations to Improve Efficiency

One of the most important reasons for the success of local search is that it is simply really fast. In many applications, hundreds of thousands of search steps can be performed per second, making a good coverage of huge search spaces possible in the first place. This speed is to a large part due to only using local information to decide about the next search step. Very often, the evaluation function computes as a sum or product only few terms of which are altered by changing a small number of solution components. Thus, often the evaluation of a neighbour of a candidate solution s can be done much faster if one knows the evaluation function value of s.[2]

Take as an example a 1-exchange neighbourhood for SAT. There, one changes the truth value of a single variable v_i at a time. This can only effect the satisfaction status of the clauses containing v_i, such that the evaluation of flipping v_i can be done locally by only inspecting all clauses that contain v_i. Going one step further, one can memorize the number of clauses satisfied and unsatisfied by the flip of variable v_i since this information only changes when a variable v_j is flipped that shares a clause with v_i.[3] This way, the evaluation of a neighbouring candidate solution can be done in constant time by just subtracting the memorized number of clauses a variable flip unsatisfies from the number it satisfies. Finally, one can then cache the variables yielding an improvement in the number of satisfied clauses, bringing the time for evaluation of the whole neighbourhood down to a constant in practice since after an initial search phase typically only few variables yield an improvement when flipped. In Chapter 6, we will develop novel caching schemes for MPE which closely resemble the ones described here for SAT.

For some domains, no efficient caching of the evaluation function is possible. One example for this is the problem of RNA Secondary Structure Design [AFH$^+$04], where the objective function to be minimized is the secondary structure's free energy, computed by a dynamic programming algorithm of complexity $\Theta(n^3)$. Due to the non-applicability of caching in this domain, other means of complexity reduction must be found. In [AFH$^+$04], a hierarchical decomposition of the RNA strand of length n into smaller strands of length m is applied,

[2]In fact, for many SLS algorithms, one does not even need to keep track of the current evaluation function value. This is because all decisions about the search step to take from candidate solution s are based only on the differences in evaluation function value of s and its neighbours.

[3]These quantities have also been called the make-count and break-count of a variable. For more details on the efficient implementation of SAT algorithms, see [Hoo98].

decreasing the complexity of single search steps from n^3 to m^3.[4]

3.5 Systematic versus Local Search

SLS algorithms are *incomplete* algorithms, meaning they are not guaranteed to find the optimal solution to a given problem in finite time, and if they find it, they typically cannot proove its optimality.[5] *Complete* algorithms, such as Branch-and-Bound (BnB), are more powerful in principle since they can achieve the desirable situation of finding a solution which they can proof to be optimal. More specifically, we call an algorithm complete if and only if it is guaranteed to terminate with the correct solution in finite time when given enough space. For randomized algorithms, there exists a weaker notion of completeness: we call an algorithm $\mathcal{A}$ *probabilistically approximately complete (PAC)* [Hoo99] if and only if the probability that $\mathcal{A}$ finds the optimal solution approaches 1 as time goes to infinity.

Research in SLS algorithms is sometimes obstructed by the fact that one does not know when the optimal solution has been found unless there exist tight upper bounds on solution quality. However, the drawback of incompleteness is less pronounced in practice, where time and space constraints often render systematic algorithms incomplete as well. Moreover, systematic approaches like BnB extensively cover the parts of the search space they explore, while incomplete SLS algorithms usually explore larger parts of the search space much earlier. The small parts of the search space explored by systematic algorithms may not contain very good solutions, which can lead to very long runtimes to find solutions with a given not even optimal quality; SLS typically find such solutions much faster [HS04]. This desirable feature of providing good solutions throughout the search clearly speaks for the usage of SLS algorithms in anytime scenarios, in which algorithms are only alloted a previously undefined runtime.

Another drawback which is often criticized with SLS algorithms is that the random components and sometimes complicated heuristics make a theoretical analysis hard. However, on the positive side, the heavy randomization allows a straightforward parallelization of SLS algorithms; the analysis of so-called run-time distributions [HS99] suggests that performing N independent runs of a strong SLS algorithm $\mathcal{A}$ on a hard problem instance $\mathcal{I}$ results in an expected solution time of the N-th part of $\mathcal{A}$'s original expected solution time (unless N is very large, and initial search cost starts to skew the picture).

[4]Of course, this hierarchical decomposition does not come for free. The catch is that there is no guarantee for two good substrands to yield a good longer strand when merged together.

[5]An exception to this are SLS algorithms for decision problems like SAT, where if one finds a variable instantiation satisfying all clauses, this is proovably optimal.

3.6 MPE-Specific Issues of SLS

SLS algorithms can be applied to the Most Probable Explanation problem in a very straight-forward way. In this domain, the search space of candidate solutions consists of all complete variable instantiations $\mathbf{V} = \mathbf{v}$ that are consistent with the evidence $\mathbf{E} = \mathbf{e}$, and we want to maximize the probability $\prod_{\phi \in \Phi} \phi[\mathbf{V} = \mathbf{v}]$ over the assignments $\mathbf{v}$ consistent with $\mathbf{e}$. Since this product may become very small (10^{-500} for extreme problems), algorithms usually employ the log-probability of an assignment as an evaluation function to guide the local search. The log-probability computes as the sum of single log-probabilities $\sum_{\phi \in \Phi} log(\phi[\mathbf{V} = \mathbf{v}])$, and if at least one of the probabilities $\phi[\mathbf{V} = \mathbf{v}]$ is zero, it is $-\infty$. Using this measure as an evaluation function would create unsolvable problems for SLS algorithms since in the initial variable assignment typically quite a large number of the single probabilities are zero; all candidate solutions in the neighbourhood then had equal evaluation function value $-\infty$, such that the local search effectively became blindfolded and at best performed a simple random walk. We prevent this undesirable behaviour by a simple means: instead of using $-\infty$ for a single log-probability of probability zero, we use -10000, which dominates the summed log-probabilities for all instances we consider in this thesis. Effectively, this makes the local search remove any zero probabilities first, still using the other log-probabilities to break ties.

In terms of notation, we use $\mathbf{V} = \mathbf{v}|V_i = v_i$ to denote the variable instantiation $\mathbf{V} = \mathbf{v}$ with the single variable V_i flipped to v_i. When $\mathbf{V} = \mathbf{v}$ is the current variable instantiation, $\prod_{\phi \in \Phi} \phi[\mathbf{V} = \mathbf{v}|V_i = v_i]$ denotes the probability of the neighbouring instantiation reached by flipping V_i to v_i.

Chapter 4

Existing Algorithms for the MPE Problem

This chapter summarizes previous algorithms for solving the MPE problem. We start with previous SLS algorithms and Mini-Bucket elimination that are both very important for our work. Then, we introduce the complete algorithms based on Mini-Buckets we compare against and finally shortly cover other algorithms we do not compare against.

4.1 Stochastic Local Search

Probably the most prominent Stochastic Local Search algorithm for inference in Bayesian networks is a method called *Stochastic Simulation* [Pea88, KD99b]. This Markov Chain Monte Carlo (MCMC) algorithm starts by randomly initializing the Bayesian network's variables $\mathbf{V}$ and then continues to sample variables V_i according to the probability distribution $P(V_i|\mathbf{V} \setminus \{V_i\})$, i.e. given the current instantiation of all the other variables. After every variable flip, the current assignment is recorded. This approach has the property of being *unbiased*, meaning that the distribution of recorded assignments converges to the Bayesian network's joint probability distribution in the limit of infinitely many variable flips. However, the performance of this approach for MPE is much inferior to that of other SLS approaches; this is because for optimization problems like MPE, one does not wish for algorithms to be unbiased, but instead requires a strong bias towards good solutions.

A related approach is *Simulated Annealing* [KGV83, KD99b], another MCMC algorithm which accepts the flip of a particular variable if the resulting instantiation is *better* in the sense of having higher probability in the Bayesian network; otherwise, the new instantiation is accepted with a probability that depends on how much worse it is and on a parameter called the *temperature*. Inspired by annealing processes in nature as they occur in the formation of crystals, this temperature starts off high and decreases over time such that in the beginning of a local search almost any variable flip is accepted and later in the process almost no worsening variable flip is accepted anymore. Thus, the search settles in a local optimum which is sometimes praised in the Machine Learning community as the aspired property of convergence. In optimization problems, however, this property is far from wanted. It is a form of *search stagnation* and prevents an escape from the local optimum to seek for new better local optima. Unlike in nature, computational processes have the ability to save a promising state and move on to find better states without the risk of loosing the current one. Concerning the performance of Simulated Annealing, it has been found that with alternating temperature levels, quite good local search algorithms can be constructed. These start off with a high temperature, cool down slowly, settle in a local optima region and then increase the temperature again to escape the local optima region [HKT95, HS04].

The main difference we see between the MCMC algorithms introduced above and current Stochastic Local Search algorithms is the problem they focus on. While MCMC algorithms have traditionally been developed for sampling or estimation of probabilities, SLS algorithms focus on optimization. In many areas of study, in which researchers frequently use MCMC algorithms, optimization problems are also of interest; thus, MCMC algorithms have also been applied for solving them. SLS algorithms can in turn also be applied to sampling, but for classical optimization problems such as SAT and TSP, only very seldomly there is interest in unbiased samples of the search space. Biased sampling, on the other hand, is applied in search space analysis for optimization problems in order to characterize, e.g., the average distance between local optima in the search space.[1]

For optimization problems, more greedy SLS algorithms are generally suited much better than simple MCMC algorithms. For the MPE problem, this was shown only in 1999 when Kalev Kask and Rina Dechter [KD99b] introduced the first explicit SLS algorithm for MPE. Named *Greedy plus Stochastic Simulation (G+StS),* their approach is a simple hybrid of best improvement variable flips in a 1-exchange neighbourhood and Stochastic Simulation steps; in their empirical evaluation, G+StS clearly outperformed Stochastic Simulation, Simulated Annealing, and a purely greedy algorithm. They also found that good initial solutions, in

[1]Since both MCMC and SLS algorithms can be applied to sampling and optimization, the question of their generality arises. Since SLS algorithms allow for a more powerful set of techniques such as various initializations and dynamic changes of the evaluation function, we tend to see them as a superclass of MCMC algorithms. All the MCMC algorithms for MPE covered here are very simple SLS algorithms.

this case gathered with the approximate Mini-Buckets algorithm (cf. Section 4.2), can boost G+StS considerably, an effect found for many SLS algorithms in a variety of problem domains [HS04].

In 2002, James Park described a reduction from MPE to Max-SAT [Par02]. This opens the possibility to simply encode MPE problems into Max-SAT and use available optimized Max-SAT solvers to find the best solution. However, instead of using this approach, Park presented MPE versions of two well-performing Dynamic Local Search algorithms for MAX-SAT, namely *Discrete Lagrangian Method (DLM)* [SW97] and *Guided Local Search (GLS)* [MT00]. In an empirical evaluation, these algorithms outperformed G+StS, with GLS yielding mostly better results than DLM. To our best knowledge, the direct approach of using Max-SAT engines has thus far not been implemented. In preliminary experiments we conducted together with Kevin Smyth at the University of British Columbia, an Iterated Robust Tabu Search [SHS03] did not show very strong performance on Max-SAT encodings of MPE problems. This may be overcome by a thorrow parameter tuning, but due to time constraints, we did not pursue these studies furhter.

4.2 Bucket Elimination and Mini Buckets

Bucket Elimination [Dec96] is a general algorithmic framework generalizing dynamic programming to accommodate algorithms for SAT, constraint satisfaction problems, linear equalities and inequalities, combinatorial optimization and last but not least probabilistic reasoning. Bucket Elimination algorithms are exact, and their performance on a given problem can be predicted by a measure for graph complexity called the *induced width* or a new measure we call *induced size*[2]:

Definition 4.2.1 (Additional Graph Concepts). An *ordering* o of a graph $\mathcal{G} = (\mathbf{V}, \mathcal{E})$ is an ordering $V_{o_1}, \dots, V_{o_n}$ of $\mathcal{G}$'s nodes $\mathbf{V}$. The *width of a node* $V \in \mathbf{V}$ is the number of its neighbours preceding it in o and its *size* is the product of the domain sizes of these neighbours. The *induced width* $w^*(\mathcal{G}, o)$ and the *induced size* $s^*(\mathcal{G}, o)$ of a graph $\mathcal{G}$ along an ordering o are defined in a constructive way: process the nodes $V \in \mathbf{V}$ from V_{o_n} to V_{o_1}, connecting all of V's neighbours in $\mathcal{G}$ that precede V in o; $w^*(\mathcal{G}, o)$ is then defined as the maximal width among $\mathcal{G}$'s variables, and $s^*(\mathcal{G}, o)$ as the maximal size. The *induced width* $w^*(\mathcal{G})$ *of a graph* $\mathcal{G}$ is its minimal induced width along any ordering of its nodes $\mathbf{V}$, and the *induced size* $s^*(\mathcal{G})$ is the minimal induced size.

[2]We actually do not expect that this measure is new; probably, at least a similar measure already exists somewhere in the literature. However, we did not find any mention of it.

Bucket Elimination has time- and space-complexity exponential in the graph's induced width and can thus only be applied to relatively sparse graphs with low induced width. Of more interest to us is an approximative variant of Bucket Elimination that can be applied to networks with arbitrary induced width; *Mini-Buckets* [DR03] can approximate probabilistic inference tasks such as MPE, MAP and belief updating (cf. Section 2.3 on page 11) in arbitrary Bayesian networks. In the following, we introduce the Bucket Elimination (BE) algorithm for MPE; a small modification will then yield the Mini Buckets (MB) algorithm.

The MPE problem, as introduced in Section 2.2 on page 10, is to find an instantiation $\mathbf{V} = \mathbf{v}$ with maximal probability $\prod_{\phi \in \Phi} \phi[\mathbf{V} = \mathbf{v}]$ over all variable instantiations $\mathbf{v}$ consistent with the evidence $\mathbf{e}$. BE (see Algorithm 4.1 on the facing page) maximizes this product by re-arrangement of the terms. Given an ordering $V_{o_1}, \ldots, V_{o_n}$, it first partitions the potentials $\phi \in \Phi$ into so-called *buckets,* putting each potential ϕ into the bucket B_V of the variable $V \in \mathcal{V}_\phi$ which appears last in the ordering. We call this bucket ϕ's *highest possible* bucket.

Next, BE processes the buckets from last to first in the ordering. If variable V is an evidence variable, V is removed from all potentials in B_V only keeping the entries with V's correct instantiation; then, each potential $\phi \in B_V$ is put into its highest possible bucket (see lines 4-6 of Algorithm 4.1 on the next page). If V is not an evidence variable, a max-product-operation is performed for the potentials Φ_{B_V} contained in B_V. This max-product-operation yields a potential ϕ_m with scope $\mathcal{V}_{\phi_m} = \bigcup_{\phi \in \Phi_{B_V}} \backslash \{V\}$ and $\phi_m[\mathcal{V}_{\phi_m} = v_{\phi_m}] = max_V \prod_{\phi \in \Phi_{B_V}} \phi[\mathcal{V}_{\phi_m} = v_{\phi_m}]$; ϕ_m is then put into its highest possible bucket (lines 8-9). After this phase, the optimal MPE value can be retrieved by computing *max-product*$(V_{o_1}, \Phi_{B_{V_1}})$.

The last step in BE processes the buckets from first to last. If variable V in bucket B_V is an evidence variable, its observed value v is assigned to V. Otherwise, the MPE value v is assigned to V that maximizes the product of all potentials in B_V given the instantiations of variables preceding V in the ordering. It can be shown that this procedure results in the correct MPE assignment $\mathbf{V} = \mathbf{v}$ [Dec96].

The resource-critical phase in BE is its second step, processing the buckets from last to first, building larger and larger potentials with consecutive max-product operations. Bucket Elimination on graph $\mathcal{G}$ along ordering o builds at least one potential with $w^*(\mathcal{G}, o)$ variables and size $s^*(\mathcal{G}, o)$. This is responsible for BE's exponential time- and space-complexity, since potentials are exponentially large in their number of variables (cf. Definition 2.1.1 on page 7). Mini Bucket Elimination (MB) [DR03] solves this problem by bounding the maximal number of variables *ib* per potential. Higher values for this *i-bound* yield better approximations but higher computational complexity, and if *ib* exceeds the network's induced width, MB is equivalent to BE.

Algorithm 4.1: Bucket Elimination for MPE

A potential ϕ's "highest possible bucket" is the bucket B_V of variable $V \in \mathbf{V}$ with highest index in ordering o.

Operation *max-product*(V, Φ) yields a potential that is the product of the potentials in Φ, maximized over V; it is formally defined in the text.

```
   Input: Bayesian network $\mathcal{B} = \langle \mathbf{V}, \mathbf{D}, \mathcal{G}, \Phi \rangle$, ordering $V_{o_1}, \ldots, V_{o_n}$, i-bound $i$, evidence
          $\mathbf{E} = \mathbf{e}$
   Output: Optimal MPE assignment $\mathbf{V} = \mathbf{v}$
   // ===== Initialize buckets.
1  Partition potentials $\Phi$ into their highest possible buckets among $B_{V_1}, \ldots, B_{V_n}$.
   // ===== Process backwards.
2  foreach $V \in V_{o_n}, \ldots, V_{o_2}$ do
3      if $(V = v) \in (\mathbf{E} = \mathbf{e})$ then
4          foreach $\phi \in \Phi_{B_V}$ do
5              Remove $V$ from $\phi$, keeping only entries consistent with $V = v$.
6              Put $\phi$ in its highest possible bucket.
7      else
8          $\phi_m \leftarrow$ max-product$(V, \Phi_{B_V})$
9          Put $\phi_m$ in its highest possible bucket.
   // ===== At this point, max-product$(V_{o_1}, \Phi_{B_{V_1}})$ yields the optimal MPE value.
   // ===== Process forwards.
10 foreach $V_{o_i} \in V_{o_1}, \ldots, V_{o_n}$ do
11     if $(V_{o_i} = v) \in (\mathbf{E} = \mathbf{e})$ then
12         $V_{o_i} \leftarrow v$
13     else
14         $V_{o_i} \leftarrow$ argmax$_v \prod_{\phi \in \Phi_{B_V}} \phi[(V_{o_1}, \ldots, V_{o_i}) = (v_{o_1}, \ldots, v)]$
```

In Algorithm 4.2 on the next page, we present the MB algorithm that only differs from BE in the second step, where it does not perform the max-product-operation for all potentials in a bucket B_V, but approximates it by a number of smaller max-products. Mini-Buckets with i-bound *ib*, MB(*ib*), uses a partitioning $\mathcal{P} = \{\Phi_{MB_{V,1}}, \ldots, \Phi_{MB_{V,k}}\}$ of the potentials $\phi \in \Phi_{B_V}$ into so-called *Mini-Buckets* such that two potentials $\phi_1, \phi_2 \in \Phi_{B_V}$ are guaranteed to be in the same Mini-Bucket if $\mathcal{V}_{\phi_1} \subseteq \mathcal{V}_{\phi_2}$ and such that the potentials in any Mini-Bucket together span no more than *ib* different variables. Using partitioning $\mathcal{P}$, MB approximates BE's max-product operation $\max_V \prod_{\phi \in \Phi_{B_V}} \phi$ for the whole bucket by the product $\prod_{\Phi_{MB_{V,i}} \in \mathcal{P}} \max_V \prod_{\phi \in \Phi_{MB_{V,i}}} \phi$ of max-products of the Mini-Buckets. Since this yields an upper bound on the whole max-product, Mini-Buckets yields an upper bound on the optimal MPE value next to the variable instantiation generated in the final forwards pass.

Mini-Buckets has been shown to perform well on a variety of networks, both as a stand-alone technique [DR03] and in combination with SLS algorithms [KD99b]. Since it yields both an upper bound and a variable assignment $\mathbf{V} = \mathbf{v}$ whose probability bounds the optimal MPE value from below, MB's performance on an MPE instance can be evaluated quite easily by comparing upper and lower bound. If they agree, MB solved the problem instance optimally.

Since the G+StS algorithm was shown to yield better performance when using a Mini-Buckets initialization in [KD99b], we also implemented MB as an optional initialization procedure. The i-bound used for MB in [KD99b] was 10, but we can not use this bound for our purposes. This is because in our problem sets, there are Bayesian networks with considerably different domain sizes. There are networks like `link` with domain sizes between 2 and 4, for which i-bound 10 is feasible, yielding potentials of maximal size $4^{10} \approx 10^6$. However, there are also networks like `mildew` having domain sizes of up to 100, with an average of 17.6. With an i-bound of 10, this would yield potentials of size 2.8×10^{12} on average and considerably larger in the extreme; potentials of these dimensions would consume terrabytes of memory to store and thus MB(10) is not feasible.

In order to prevent this kind of problems stemming from different domain sizes, we suggest a new variant of MB which we call *MB**. MB* relates to MB much like induced size to induced width. In this variant, not the maximal number of variables in a Mini-Bucket is controlled, but the maximal size of any potential encountered during the execution of Mini-Buckets. When processing a single Mini-Bucket, the first step is to multiply all its potentials, resulting in a product potential whose size is the product of the domain sizes of variables which occur in the Mini-Bucket's potentials; this is the only step where large potentials are produced, and if we can bound the size of the product potentials of all Mini-Buckets we achieve our objec-

Algorithm 4.2: Mini-Bucket Elimination (MB/MB*) for MPE

Very similar to Algorithm 4.1 on page 29, but in the second step, the max-product operation is approximated as the product of a number of smaller max-products; and an upper bound on the optimal MPE value is computed.

GeneratePartitioning(Φ_B) generates a partitioning $\mathcal{P} = \{\Phi_{MB_{V,1}}, \ldots, \Phi_{MB_{V,k}}\}$ of the potentials $\phi \in \Phi_{B_V}$ such that two potentials $\phi_1, \phi_2 \in \Phi_{B_V}$ are guaranteed to be in the same Mini-Bucket if $\mathcal{V}_{\phi_1} \subseteq \mathcal{V}_{\phi_2}$ and such that the potentials in any Mini-Bucket together span no more than i-bound different variables. In our modified variant MB*, this latter constraint is changed to assert that not the number of such variables is bound by i-bound, but that the product over their domain sizes is bound by $size$-bound.

```
Input: Bayesian network $\mathcal{B} = \langle \mathbf{V}, \mathbf{D}, \mathcal{G}, \Phi \rangle$, ordering $V_{o_1}, \ldots, V_{o_n}$, i-bound $i$, evidence
       $\mathbf{E} = \mathbf{e}$.
Output: Variable assignment $\mathbf{V} = \mathbf{v}$ with approximately optimal probability
        $\prod_{\phi \in \Phi} \phi[\mathbf{V} = \mathbf{v}]$, upper bound on optimal MPE value.
   // ===== Initialize buckets.
1  Partition potentials $\Phi$ into their highest possible buckets among $B_{V_1}, \ldots, B_{V_n}$.
   // ===== Process backwards.
2  foreach $V \in V_{o_n}, \ldots, V_{o_2}$ do
3      if $(V = v) \in (\mathbf{E} = \mathbf{e})$ then
4          foreach $\phi \in \Phi_{B_V}$ do
5              Remove $V$ from $\phi$, keeping only entries consistent with $V = v$.
6              Put $\phi$ in its highest possible bucket.
7      else
8          $\mathcal{P} \leftarrow$ GeneratePartitioning$(\Phi_B)$
9          foreach $\Phi_{MB} \in \mathcal{P}$ do
10             $\phi_{MB} \leftarrow$ max-product$(V, \Phi_{MB})$
11             Put $\phi_{MB}$ in highest possible bucket.
12 upper $\leftarrow$ max-product$(V_1, \Phi_{B_{V_1}})$
   // ===== Process forwards.
13 foreach $V_{o_i} \in V_{o_1}, \ldots, V_{o_n}$ do
14     if $(V_{o_i} = v) \in (\mathbf{E} = \mathbf{e})$ then
15         $V_{o_i} \leftarrow v$
16     else
17         $V_{o_i} \leftarrow$ argmax$_v \prod_{\phi \in \Phi_{B_V}} \phi[(V_{o_1}, \ldots, V_{o_i}) = (v_{o_1}, \ldots, v)]$
```

Algorithm 4.3: Anytime Mini-Bucket Elimination for MPE (Anytime MB)

```
Input: Bayesian network $\mathcal{B} = \langle \mathbf{V}, \mathbf{D}, \mathcal{G}, \Phi \rangle$, ordering $V_{o_1}, \dots, V_{o_n}$, maximal i-bound
       $\max_{\text{ib}}$, time bound $t$, evidence $\mathbf{E} = \mathbf{e}$
Output: Best variable assignment $\mathbf{V} = \mathbf{v}$ with approximately optimal probability
        $\prod_{\phi \in \Phi} \phi[\mathbf{V} = \mathbf{v}]$ found with maximal i-bound $\max_{\text{ib}}$ and time $t$; upper bound on
        optimal MPE value.
  // ===== Initialize upper bound, i-bound, and variable assignment.
1 upper $\leftarrow \infty$
2 ib $\leftarrow 0$
3 Randomly initialize $\mathbf{v}$
  // ===== Run with increasing i-bound until optimality proofed or out of resources.
4 while runtime $< t$ and ib $< \max_{ib}$ do
5     $[\tilde{\mathbf{v}}, upper[ib]] \leftarrow MB(ib)$
6     if $\prod_{\phi \in \Phi} \phi[\mathbf{V} = \tilde{\mathbf{v}}] > \prod_{\phi \in \Phi} \phi[\mathbf{V} = \mathbf{v}]$ then $\mathbf{v} \leftarrow \tilde{\mathbf{v}}$
7     if $upper[ib] < upper$ then $upper \leftarrow upper[ib]$
8     if $upper = \prod_{\phi \in \Phi} \phi[\mathbf{V} = \mathbf{v}]$ then break // optimality proven.
9     ib $\leftarrow$ ib $+ 1$
```

tive of bounding all potential sizes. We ensure that this size is always smaller than a parameter $size$-*bound* by partitioning the potentials in a Bucket such that for each Mini-Bucket, the product of its variables' domain sizes remains smaller than the $size$-bound. This replaces the constraint of having no more than i-bound variables in a Mini-Bucket.[3] When using MB* as an initialization procedure, we use $size$-bound 10^5, meaning that for the largest Mini-Bucket potential we will need to save at most 100000 double precision entries. With this parameter setting, MB* yielded overall good results and had low complexity in our experiments, always finishing within a second of CPU time.

4.3 Exact Algorithms based on Mini-Buckets

The Mini-Buckets scheme is quite efficient with low i-bounds, but higher i-bounds yield better results. This immediately suggests an anytime variant of MB that starts off with low i-bound and iteratively increases it until the resources do not suffice for higher i-bounds. Algorithm 4.3 shows this Anytime MB algorithm, suggested in [DR03].

[3]Of course it is possible to choose a $size$-bound that is lower than the size of some potential ϕ in the original problem. In this case, MB* only guarantess not to produce a larger potential than ϕ. The same restriction applies to the original MB algorithm, where one can set the i-bound lower than the number of variables in some initial potential ϕ.

While the Anytime MB is very efficient for problems with low induced width, for larger induced widths it often does not find good approximations with feasible maximal i-bounds. However, since MB yields both upper and lower bounds on the optimal MPE value, it can be easily used as a subcomponent in systematic search algorithms like Branch and Bound or $\mathcal{A}^*$ [KD99a, RN03]. The most recent Branch and Bound algorithms based on Mini-Buckets are *BBMB* and *BBBT*. BBMB stands for Branch and Bound with Mini-Buckets heuristic and comes in two variants. The static variant, *s-BBMB*, introduced by Kask and Dechter in 1999 [KD99a] starts with a pass of Mini-Buckets. The functions computed in this pre-processing step are then combined to form an upper bound on the probability achievable by extending any partial variable assignment. Since MPE is a maximization problem, this approach yields an admissible heuristic that can be applied in $\mathcal{A}^*$ or Branch-and-Bound. The dynamic variant of BBMB, *d-BBMB*, has only just been developed [MD04], but outperforms s-BBMB considerably in the experiments performed thus far. As opposed to s-BBMB, d-BBMB computes the Mini-Buckets heuristic at every node in the search tree, conditional on the node's partial assignment. This takes more time, but pays off especially when using small i-bounds [MD04].

The dynamic BBMB variant also outperforms the quite recent BBBT algorithm [DKL01], which combines Branch-and-Bound search with Mini Cluster Tree Elimination (MCTE), an approximate variant of Cluster Tree Elimination. MCTE is applied at every node in the search tree to compute lower bounds for every possible extension of the node's partial assignment with a single variable instantiation. The lower bounds for all possible extensions are computed in parallel by MCTE, which is much faster than multiple calls to the Mini-Buckets algorithm [DKL01].

Some research has been done in comparing the described Branch-and-Bound algorithms with SLS approaches. As mentioned in Section 4.1 on page 25, Kask and Dechter suggested the SLS algorithm G+StS [KD99b] which was able to improve on many of the best solutions found with Mini-Buckets. However, there were also many instances on which G+StS was vastly inferior to Mini-Buckets. In 2003, Marinescu, Kask and Dechter published a paper claiming that s-BBMB and BBBT outperform SLS algorithms [MKD03]. More specifically, they implemented the Dynamic Local Search algorithms DLM [SW97] and GLS [MT00] like Park described them for MPE [Par02] as well as the G+StS algorithm [KD99b]. Their experiments showed superiority of s-BBMB and BBBT with the best-performing i-bound (between 2 and 10) over their parameterless versions of the SLS algorithms on most problem instances. However, GLS managed to considerably outperform s-BBMB/BBBT with any i-bound for some networks, especially for those with low domain sizes. On all reported instances except the `mildew` network, GLS outperformed the other SLS algorithms, and for `mildew`, G+StS performed considerably better.

Since the optimal i-bound for the s-BBMB/BBBT algorithms cannot be guessed a-priori[4], a practitioner would need to run experiments for multiple i-bounds in parallel until the fastest one terminates. Considering this and the fact that GLS was implemented with fixed parameters, the superiority of s-BBMB/BBBT over GLS remains at least questionable.

4.4 Other Algorithms for MPE

Approximate Decomposition (AD) [Lar03] is a scheme very similar to Bucket Elimination, or, more generally, Variable Elimination [ZP94]. The variables $\mathbf{V}$ in a Bayesian network $\mathcal{B} = \langle \mathbf{V}, \mathbf{D}, \mathcal{G}, \Phi \rangle$ are eliminated iteratively along the min-degree ordering, connecting the neighbours *nb*(V) of an eliminated variable $V \in \mathbf{V}$ in $\mathcal{G}$ by creating a new potential ϕ as the max-product over all potentials ϕ_i spanning V. This standard procedure from Variable Elimination is applied until connecting *nb*(V) in $\mathcal{G}$ increases $\mathcal{G}$'s width above a predefined bound *ib*. Once this bound is exceeded, the new edge with maximal sum of endpoint degrees is deleted until the width is less than or equal to *ib* again. Let $C_1, \ldots, C_m$ be the maximal cliques of the subgraph induced by *nb*(V) after the deletion of edges; ϕ is then approximated by a product $\prod_{i=1}^{m} \phi_i$ where ϕ_i has scope C_i. The potentials ϕ_i are computed with a linear program yielding either an upper or a lower bound of $\prod_{i=1}^{m} \phi_i$ on ϕ. In [Lar03], AD significantly outperformed Mini-Buckets in bounding solution quality for the MPE task from above and below, but both algorithms found only very poor bounds (e.g. upper bound 1) for random networks. Although AD seems to be superior to MB, we do not compare against it since its source code is not available; also, Mini-Buckets is by far the better-known algorithm, easier to implement and demonstrated good performance on a variety of Bayesian networks; for AD, there are only published results for two structured and some random networks. Nevertheless, for structured networks, AD is a very promising approach and future SLS algorithms might exploit it for initialization or to compute upper bounds in order to proof optimality of solutions found with SLS.

Most commercial tools for reasoning in Bayesian networks use a framework called *Junction Trees* [LS88, JLO90, CDLS99]. In this secondary structure for Bayesian networks, a network's potentials are organized into a tree structure $(\mathcal{C}, \mathcal{S})$ of so-called *clusters* $\mathcal{C}$ connected by a set of *separators* $\mathcal{S}$. Each cluster $C \in \mathcal{C}$

[4]Radu Marinescu advised me in email communication that ”it is hard to predict an optimal i-bound beforehand” and that thus always a range for the i-bound should be reported.

holds a cluster potential ϕ_C that is the product of some of the Bayesian network's potentials Φ, such that each potential $\phi \in \Phi$ is placed into exactly one cluster. We denote by $\mathcal{V}_{C_i}$ the scope of cluster C_i's potential ϕ_{C_i}. The tree structure must satisfy the so-called *running intersection property*, which for arbitrary C_1 and C_n requires every cluster C_i on the path $C_1, \ldots, C_n$ in $(\mathcal{C}, \mathcal{S})$ to span at least the variables $\mathcal{V}_{C_1} \cap \mathcal{V}_{C_n}$. In a *consistent* junction tree, each separator $S \in \mathcal{S}$ connecting two clusters C_i and C_j holds a separator potential ϕ_S that equals the potentials ϕ_{C_i} and ϕ_{C_j}, marginalized to their shared variables $\mathcal{V}_{C_i} \cap \mathcal{V}_{C_j}$. The joint probability distribution ϕ encoded by a consistent junction tree is then $\phi = (\prod_{C \in \mathcal{C}} \phi_C)/(\prod_{S \in \mathcal{S}} \phi_S)$. Evidence is incorporated into a junction tree by modifying single potentials, after which consistency is restored by local message passing between the clusters via the separators. By means of this simple message passing scheme, all the Bayesian network problems MPE, M-MPE, MAP, belief updating, and parallel computation of all single marginals can be solved exactly in time and space linear in the *size of the junction tree*. However, the catch is that this size, which is the sum of the sizes of the junction tree's potentials, grows exponentially in the network's induced width. Still junction trees have an important advantage over Bucket Elimination in that the computation of N marginals of single variables is parallelized making them N times faster than Bucket Elimination.

The drawback of prohibitively large cluster potentials for large networks in the exact junction tree framework is addressed by several approximation techniques for junction trees [JA90, Kjæ94, HND04]. The first approach for this is a method called *compression* which exploits the possibly large number of zero entries in the potentials by a new representation that does not explicitly store zeros. Then, for approximation, one can treat small entries as zero [JA90] subsequently performing compression, which can lead to considerable savings in complexity for the price of only approximative inference. Another way to approximate large clusters is to split them into two or more smaller clusters, connected by a separator [Kjæ94, HND04]. These smaller clusters might be subsumed by neighbouring clusters in the junction tree and can then be dropped, leading to considerable savings in junction tree size.

Another approach that employs local message passing is Judea Pearl's classic *belief propagation (BP)* algorithm [Pea88]. BP has been developed as an exact algorithm for singly-connected Bayesian networks, i.e. networks for which the independence graph is a tree, but it can also be used as an approximation scheme for multiply connected networks, then called *loopy belief propagation (LBP)*. Quite recently, it has been shown that an extraordinarily well performing error correcting code scheme called *Turbo Codes* employs an algorithm equivalent to LBP [MMC98], which renewed great interest in the algorithm among many researchers. Subsequently [MWJ99], LBP has been shown to perform very well for more general types of Bayesian networks if it converges. However, for some networks LBP oscillates, yielding very bad results.

Finally, there exists a subclass of Bayesian networks for which exact inference is cheaper than for the general case. For networks with a two-layer structure, noisy-OR CPTs and Boolean evidence variables in the second layer of the network, the *Quickscore* algorithm described in [Hec90] has complexity only exponential in the number of positive evidence nodes. This particular type of networks can be found in Bayesian networks used in medical diagnosis [JJ99] or computer network diagnosis [RBM02a].

Chapter 5

SLS Algorithms for MPE

In this chapter, we detail the SLS algorithms for MPE we implemented. These include the two best performing SLS algorithms for MPE developed to date, G+StS[KD99b] and GLS [Par02] (cf. Section 4.1 on page 25); a new improved version of GLS, which we call GLS$^+$; the first Iterated Local Search (ILS) algorithm for MPE; and a hybrid algorithm employing ILS, GLS$^+$, and our adapted Mini-Buckets variant MB* (cf. Algorithm 4.2 on page 31).

5.1 Greedy plus Stochastic Simulation (G+StS)

In Algorithm 5.1 on the next page, we present our implementation of the G+StS algorithm [KD99b]. G+StS employs a Mini-Buckets initialization, which was shown to clearly outperform a random initialization in [KD99b] when an i-bound of 10 is used. As detailed in Section 4.2 on page 27, we cannot use this i-bound for our experiments due to the large variance in domain sizes across our problem sets. Instead, we use our variant MB* with *size*-bound 10^5; for problems with Boolean variables, MB* with this *size*-bound is close to MB(17)[1], whereas for problems with constant domain size 10, it is equivalent to using MB(5).

The G+StS algorithm performs multiple tries, where a new try is started when the current one did not find an improvement over its best solution $\mathbf{V} = \mathbf{v}$ for time $cf \times t_{opt}$, where t_{opt} is the time in which it found $\mathbf{V} = \mathbf{v}$. Within each try, the MB* initialization is followed by a local search phase. In this phase, G+StS iteratively flips variables, at each time probabilistically deciding between stochastic simulation steps (lines 7-8 in Algorithm 5.1 on the next page) and greedy steps in a one-exchange-neighbourhood (line 10). Note that in the algorithm description

[1]MB$^*(131072)$ = MB$^*(2^{17})$ is equivalent to MB(17) for Boolean variables.

Algorithm 5.1: Greedy plus Stochastic Simulation (G+StS) for MPE
The algorithm starts a new "try" when it did not improve the best solution $\mathbf{v}$ found in the current try for a time longer than a cutoff factor *cf* multiplied by the time the try needed to find $\mathbf{v}$. The best solution found across all tries is returned.
$g(\mathbf{v} \mid V_i = v_i)$ abbreviates $\sum_{\phi \in \Phi} \log(\phi[\mathbf{V} = \mathbf{v} | V_i = v_i])$; and $g(\mathbf{v})$ abbreviates $\sum_{\phi \in \Phi} \log(\phi[\mathbf{V} = \mathbf{v}])$.

```
Input: Bayesian network $\mathcal{B} = \langle \mathbf{V}, \mathbf{D}, \mathcal{G}, \Phi \rangle$, evidence $\mathbf{E} = \mathbf{e}$, time bound $t$, noise
       probability $np$, cutoff factor cf
Output: Variable assignment $\mathbf{V} = \mathbf{v}$ with highest probability $\prod_{\phi \in \Phi} \phi[\mathbf{V} = \mathbf{v}]$ found in
        time $t$
1  while runtime $< t$ do
       // ===== Initialize variable assignment.
2      Initialize $\mathbf{V} = \mathbf{v}$ with MB$^*(10^5)$.
3      opt $\leftarrow -\infty$
       // ===== Flip single variables until restart or time-out.
4      repeat
5          Draw $x$ from uniform distribution $u(0,1)$
6          if $x < np$ then
7              Randomly pick $V_i \in \mathbf{V}$.
8              Sample $v_i$ from $P(V_i = v_i | [\mathbf{V} \setminus \{V_i\} = \mathbf{v} \setminus \{v_i\}])$
9          else
10             Randomly pick $V_i \in \mathbf{V} \setminus \mathbf{E}$ and $v_i \in D_{V_i}$ maximizing $g(\mathbf{v}|V_i = v_i)$.
11         Flip variable $V_i$ to value $v_i$. // this is even done if $V_i$ is already $v_i$
12         if $g(\mathbf{v}) >$ opt then
13             opt $\leftarrow g(\mathbf{v})$
14             $t_{opt} \leftarrow runtime$
15     until runtime $>$ cf $\times$ $t_{opt}$ or runtime $\geq t$
```

of G+StS and the other algorithms in this chapter, we use the high-level notations "Randomly pick $V_i \in \mathbf{V}$ and $v_i \in D_{V_i}$ maximizing $g(\mathbf{v}|V_i = v_i)$" (line 10) and "Flip variable V_i to value v_i." (line 11). The actual implementation of these two operations will be covered in great detail in our discussion of various caching schemes in Chapter 6.

5.2 Guided Local Search (GLS and GLS$^+$)

Guided Local Search [Vou97] is a rather general Dynamic Local Search algorithm that has been applied successfully to many combinatorial problems, including TSP [VT99], SAT, and weighted Max-SAT [MT00]. In its general form [Vou97], GLS associates penalties with domain-dependent so-called *solution features*, properties true for a subset of candidate solutions. GLS iterates a two-phase process, in which first a local search w.r.t. a particular evaluation function is performed and when a local minimum of this function is reached, the evaluation function is modified to make the next local search move on in the search space. This modification is implemented by increasing the penalties of solution features present in the current solution.

For minimization problems, GLS's evaluation function value $g(s)$ of a candidate solution s is the weighted sum of the objective function value of s, $f(s)$, and the penalties for solution features present in s. The evaluation function is used to guide the local search and once one of its local minima s' is reached, the overal penalty of s' is modified as follows. For each solution feature i present in s', a *utility* $u_i = c_i/(1 + \lambda_i)$ is computed, where λ_i is the penalty associated with solution feature i and c_i is its *cost*. Like the solution features, the costs are domain-dependent and represent the direct or indirect impact the solution features have on the objective function value (cf. [Vou97]). In local minima s' w.r.t. the current evaluation function g, for each solution feature i with maximal utility u_i present in s', the penalty λ_i is then increased by the constant 1. In order to prevent the penalties from growing indefinitely, all penalties are regularly smoothed by multiplying them with a factor $\rho \leq 1$ every N_ρ local minima.

As an example, consider GLS for the weighted Max-SAT domain, described in [MT00]. There, the solution features present in a variable assignment $\mathbf{V} = \mathbf{v}$ are the clauses unsatisfied by the assignment. The evaluation function in this domain is the unweighted sum $\sum_{C \in \mathcal{C}_{unsat}} w_C + \lambda_C$ of the weights w_C and the penalties λ_C associated with the currently unsatisfied clauses $\mathcal{C}_{unsat}$. A clause's weight w_C is also used as the cost for computing its utility.

Park [Par02] introduced a reduction from MPE to weighted Max-SAT, in which for every entry $\phi[V_\phi = v_\phi]$ in a potential one clause is created. Using the intuition from this reduction, he also adapted GLS to the MPE problem. The solution features in MPE are partial instantiations of the variables. More specifically, for each potential $\phi \in \Phi$, every instantiation $V_\phi = v_\phi$ of its variables is a solution feature with cost $-\log(\phi[V_\phi = v_\phi])$.[2] We will refer to the penalty value associated with this solution feature as $\lambda_\phi[V_\phi = v_\phi]$. Note that the number of solution features present in any candidate solution for an MPE instance is constant; this is because every variable instantiation $\mathbf{V} = \mathbf{v}$ is consistent with exactly one (partial) instantiation $V_\phi = v_\phi$ of each potential ϕ, and thus the number of solution features present

[2]The cost of solution feature $V_\phi = v_\phi$ is taken to be the negative logarithmic potential entry $-\log(\phi[V_\phi = v_\phi])$ following the intuitions gained from the reduction of MPE to Max-SAT. In Max-SAT, the weight associated with a clause for a potential entry $\phi[V_\phi = v_\phi]$ is also its negative log-probability $-\log(\phi[V_\phi = v_\phi])$, and the cost of a clause is defined as its weight.

in each instantiation is the total number of potentials $|\Phi|$.

For maximization problems, GLS's evaluation function for a candidate solution s is its objective function value, $f(s)$, minus the weighted sum of the penalties associated with the solution features of s. For the MPE domain, this results in the evaluation function $g(v) = \prod_{\phi \in \Phi} \phi[\mathbf{V} = \mathbf{v}] - w \times \sum_{\phi \in \Phi} \lambda_\phi[\mathbf{V} = \mathbf{v}]$, where w is a weighting factor which is, for example, set to 1 in the Max-SAT domain. However, Park [Par02] does not use exactly this evaluation function; he notes that the evaluation function to be minimized essentially becomes $\sum_{\phi \in \Phi} \lambda_\phi[\mathbf{V} = \mathbf{v}]$ and exploits this simplification in his implementation.[3] Indeed, since probabilities are bound by 1 and each penalty is incremented in intervals of 1, the penalties clearly dominate the evaluation function. Nevertheless, one might expect the small objective function values still to have a potentially great impact since they might beneficially break ties in the local search phases when two neighbouring solutions have equal penalties. Another consequence of omitting the objective function could be larger plateaus in the search space, which could affect the effectiveness of the subsidiary local search process.

For interesting larger problem instances, the actual values of the objective function we are dealing with are smaller than 10^{-50}, sometimes 10^{-500}, values which are indistinguishable from numerical instabilities using double precision representation. This may be overcome by multiplying the objective function by some constant. However, unless this constant be adapted during search, it will in most cases imply complete domination by the evaluation function either by the objective function or the penalties. Since domination of the objective function would make the search stagnate in the first local optimum encountered, Park's approach of only using the penalties as an evaluation function is very reasonable. We implemented GLS the same way and give pseudocode for it in Algorithm 5.2 on the facing page.

Although we argued above that when using the original GLS framework there is no straightforward way to incorporate both objective function value and penalties in the MPE domain, our intuition is that some more greedyness would be highly beneficial for GLS. This intuition is partly based on the observation that GLS needs considerable time to get started since initially all penalties are set to zero and only represent useful information after many local minima have been visited. Another reason for this intuition is that for example in the weighted Max-SAT domain (in which GLS performs very well) there is a stronger connection between objective function and penalties than for the MPE problem. In weighted Max-SAT, the evaluation function is the sum of penalties for all *unsatisfied* clauses, and since less clauses are unsatisfied in better variable assignments, this constitutes an indirect connection between objective function and applicable penalties. As discussed

[3]Many thanks to James Park for providing his implementation.

Algorithm 5.2: Guided Local Search (GLS) for MPE

$\lambda_\phi[V_\phi = v_\phi]$ denotes the penalty associated with the partial variable instantiation $V_\phi = v_\phi$ of potential ϕ; and $\lambda_\phi[\mathbf{V} = \mathbf{v}]$ denotes the unique penalty $\lambda_\phi[V_\phi = v_\phi]$ for which v_ϕ is consistent with $\mathbf{v}$.

$g(\mathbf{v}|V_i = v_i)$ abbreviates $\sum_{\phi\in\Phi} \lambda_\phi[\mathbf{V} = \mathbf{v}|V_i = v_i]$; and $g(\mathbf{v})$ abbreviates $\sum_{\phi\in\Phi} \lambda_\phi[\mathbf{V} = \mathbf{v}]$. The default parameters from [Par02] are $\langle\rho, N_\rho\rangle = \langle 0.8, 200\rangle$.

```
Input: Bayesian network $\mathcal{B} = \langle \mathbf{V}, \mathbf{D}, \mathcal{G}, \Phi \rangle$, evidence $\mathbf{E} = \mathbf{e}$, time bound $t$, smoothing
       factor $\rho$, smoothing interval $N_\rho$.
Output: Variable assignment $\mathbf{V} = \mathbf{v}$ with highest probability $\prod_{\phi\in\Phi} \phi[\mathbf{V} = \mathbf{v}]$ found in
        time $t$

// ===== Initialize variable assignment, penalties, and local minima counter.
foreach $V_i \in \mathbf{V} \setminus \mathbf{E}$ do Randomly initialize $V_i$ to $v_i \in D_{V_i}$
foreach $V_i \in \mathbf{E}$ do $V_i \leftarrow v_i$ consistent with $\mathbf{e}$
foreach $\phi \in \Phi$ and all instantiations $V_\phi = v_\phi$ do $\lambda_\phi[V_\phi = v_\phi] \leftarrow 0$
$\#(LM) \leftarrow 0$
// ===== Flip single variables or update evaluation function until termination.
while runtime $< t$ do
    Randomly pick $V_i \in \mathbf{V} \setminus \mathbf{E}$ and $v_i \in D_{V_i}$ minimizing $g(\mathbf{v}|V_i = v_i)$.
    if $g(\mathbf{v}|V_i = v_i) > g(\mathbf{v})$ then
        Flip variable $V_i$ to value $v_i$.
    else
        // ===== Local minimum, update evaluation function.
        foreach $\phi \in \Phi$ do
            if $\lambda_\phi[\mathbf{V} = \mathbf{v}] = \max_{\phi\in\Phi} [-\phi[\mathbf{V} = \mathbf{v}]/(1 + \lambda_\phi[\mathbf{V} = \mathbf{v}])]$ then
                $\lambda_\phi[\mathbf{V} = \mathbf{v}] \leftarrow \lambda_\phi[\mathbf{V} = \mathbf{v}] + 1$
        // ===== Regularly smooth penalties.
        $\#(LM) \leftarrow \#(LM) + 1$
        if $\#(LM)$ modulo $N_\rho = 0$ then
            for $\phi \in \Phi$ and all instantiations $V_\phi = v_\phi$ do
                $\lambda_\phi[V_\phi = v_\phi] \leftarrow \lambda_\phi[V_\phi = v_\phi] * \rho$
```

above, in the MPE domain, there are $|\Phi|$ solution features present in each variable instantiation, such that this additional connection does not apply. Hence, the sole interaction of objective function and penalties in this domain is via the utilities: an entry $\phi[V_\phi = v_\phi]$ with high probability is assigned low utility[4]; thus, its associated penalty $\lambda_\phi[V_\phi = v_\phi]$ will be increased less often, leading to the desirable partial variable instantiation $V_\phi = v_\phi$ to be aspired eventually, but possibly only after a considerable delay.

Due to the described initial and persistent lack of greediness we expect the performance of GLS for MPE to be boosted significantly when the objective function can be integrated into the search heuristic in some meaningful way. We achieve this by a simple change in GLS's evaluation function. Our improved version of GLS, which we call *GLS*$^+$, takes the logarithm of the objective function and adds this to the appropriate penalties; since the objective function is a product of probabilities, this is equivalent to using the summed log-probabilities, making the new evaluation function to be maximized $\sum_{\phi\in\Phi} \log(\phi[\mathbf{V} = \mathbf{v}]) - w \times \lambda_\phi[\mathbf{V} = \mathbf{v}]$, where w again is a weighting factor. The second difference between GLS and GLS$^+$ lies in a possibly different initialization. Since MB$^*(10^5)$ performed very well for G+StS, we also consider it for GLS$^+$ next to a random initialization. In Section 7.4 on page 77 on tuning GLS$^+$, we will demonstrate that this indeed considerably improves the performance of GLS$^+$ for some instances. We detail GLS$^+$ in Algorithm 5.3 on the facing page.

The initial performance of GLS is weak due to its temporary disorientation until the penalties represent meaningful information, and we expect GLS$^+$ to clearly outperform GLS for short runtimes. Our experimenal evaluation in Section 8.4 on page 94 shows that this is indeed the case and that for many problem instances the initial advantage of GLS$^+$ persists for longer runtimes, while for some other instances the algorithms behave virtually identical for longer runs.

5.3 Iterated Local Search (ILS)

In this section, we introduce a novel Iterated Local Search (ILS) algorithm for MPE. To our best knowledge, this algorithm is the first of its kind as the ILS framework (cf. Algorithm scheme 3.2 on page 21) has not been previously applied to MPE.. At the heart of ILS algorithms are the four components *GenerateInitialSolution*, *LocalSearch*, *Pertubation*, and *AcceptanceCriterion*. For the MPE problem, we fixed *LocalSearch* to be greedy hill-climbing, and leave the other components

[4]Recall that in MPE, the utility for potential entry $\phi[V_\phi = v_\phi]$ is defined as $-\log(\phi[V_\phi = v_\phi])/(\lambda_\phi[V_\phi = v_\phi] + 1)$.

Algorithm 5.3: Improved Guided Local Search (GLS$^+$) for MPE

This algorithm differs from Algorithm 5.2 on page 41 only in the initialization and in the evaluation function, where it additionally incorporates the log-probability of a neighbouring variable instantiation.
$g(\mathbf{v}|V_i = v_i)$ abbreviates $\sum_{\phi \in \Phi} \log(\phi[\mathbf{V} = \mathbf{v}|V_i = v_i]) - w \times \lambda_\phi[\mathbf{V} = \mathbf{v}|V_i = v_i]$; and $g(\mathbf{v})$ abbreviates $\sum_{\phi \in \Phi} \log(\phi[\mathbf{V} = \mathbf{v}]) - w \times \lambda_\phi[\mathbf{V} = \mathbf{v}]$. Our default parameters are $\langle \rho, N_\rho \rangle = \langle 0.999, 200 \rangle$ and initialization MB$^*(10^5)$.

Input: Bayesian network $\mathcal{B} = \langle \mathbf{V}, \mathbf{D}, \mathcal{G}, \Phi \rangle$, evidence $\mathbf{E} = \mathbf{e}$, time bound t, smoothing factor ρ, smoothing interval N_ρ.
Output: Variable assignment $\mathbf{V} = \mathbf{v}$ with highest probability $\prod_{\phi \in \Phi} \phi[\mathbf{V} = \mathbf{v}]$ found in time t

```
   // ===== Initialize variable assignment, penalties, and local minima counter.
1  $\mathbf{v} \leftarrow$ GenerateInitialSolution($\mathcal{B}$)
2  foreach $\phi \in \Phi$ and all instantiations $V_\phi = v_\phi$ do  $\lambda_\phi[V_\phi = v_\phi] \leftarrow 0$
3  $\#(LM) \leftarrow 0$
   // ===== Flip single variables or update evaluation function until termination.
4  while runtime $< t$ do
5      Randomly pick $V_i \in \mathbf{V} \setminus \mathbf{E}$ and $v_i \in D_{V_i}$ maximizing $g(\mathbf{v}|V_i = v_i)$.
6      if $g(\mathbf{v}|V_i = v_i) > g(\mathbf{v})$ then
7          Flip variable $V_i$ to value $v_i$.
8      else
           // ===== Local minimum, update evaluation function.
9          foreach $\phi \in \Phi$ do
10             if $\lambda_\phi[\mathbf{V} = \mathbf{v}] = \max_{\phi \in \Phi} [-\phi[\mathbf{V} = \mathbf{v}]/(1 + \lambda_\phi[\mathbf{V} = \mathbf{v}])]$ then
11                 $\lambda_\phi[\mathbf{V} = \mathbf{v}] \leftarrow \lambda_\phi[\mathbf{V} = \mathbf{v}] + 1$
           // ===== Regularly smooth penalties.
12         $\#(LM) \leftarrow \#(LM) + 1$
13         if $\#(LM)$ modulo $N_\rho = 0$ then
14             for $\phi \in \Phi$ and all instantiations $V_\phi = v_\phi$ do
15                 $\lambda_\phi[V_\phi = v_\phi] \leftarrow \lambda_\phi[V_\phi = v_\phi] * \rho$
```

to be determined later. Algorithm outline 5.4 on the next page shows the resulting basic ILS algorithm for MPE. We also include a restart mechanism in this basic ILS which is very similar to the one employed by G+StS[5]: whenever ILS does not improve its best solution for too many iterations of the current try, the search is continued from a new initial solution obtained using *GenerateInitialSolution*.[6]

Algorithm outline 5.4 on the facing page still leaves a lot of room for design choices since fundamentally different algorithms result from choosing a different pertubation or acceptance criterion. Equipped with an automated parameter optimization scheme described in Appendix A, we allowed for a rather large number of variants. For *GenerateInitialSolution*, we tried a random initialization and MB$^*(10^5)$, just as we used it for G+StS in Section 5.1 on page 37. For the acceptance criterion, we considered four alternatives in our experiments:

BETTER accepts a new solution $\mathbf{v}$ if and only if it is better than or equal to the last iterations's solution $\mathbf{v}^*$; otherwise, $\mathbf{v}^*$ is returned.

RW (for random walk) always accepts the new solution $\mathbf{v}^*$;

BE/RW is a simple hybrid of BETTER and RW that always accepts improving new solutions $\mathbf{v}$, but also accepts new solutions $\mathbf{v}$ that are worse than $\mathbf{v}^*$ with a certain acceptance probability *ap*; if $\mathbf{v}$ is not accepted, $\mathbf{v}^*$ is returned.

LSMC is a simulated annealing type acceptance criterion that always accepts improving new solutions. For worse solutions $\mathbf{v}$, the relative difference d in objective function value between the new solution and the last one is computed and the new solution is then accepted with probability $\exp(-d/T)$, where T is a parameter called the temperature. If $\mathbf{v}$ is rejected, $\mathbf{v}^*$ is returned.

Acceptance criterion BETTER usually yields good results for short runs, but unfortunately shows search stagnation for some problem domains. However, due to the restart mechanism we employ, this problem is much less pronounced in our application. As opposed to BETTER, acceptance criterion RW never exhibits search stagnation but quite often simply performs very badly due to its lack of greediness. For BE/RW, the greediness can be controlled by the acceptance probability *ap*; for *ap* $= 0$ and *ap* $= 1$, this is equivalent to BETTER and RW, respectively. LSMC finally is also configurable in its greediness by the temperature T; as T grows, the

[5]The only difference to the restart mechanism in G+StS is that in ILS, we use the number of iterations instead of the runtime to decide when to restart the algorithm. This yields reproducable results on different architectures and machines.

[6]This restart mechanism became necessary since we were not able to remove search stagnation from the algorithm by any other means.

Algorithm outline 5.4: Basic ILS for MPE

The algorithm starts a new try when it did not improve the best solution $\mathbf{v}$ found in the current try for more iterations than a cutoff factor *cf* multiplied by the number of iterations the try needed to find $\mathbf{v}$. The best solution found across all tries is returned. $g(\mathbf{v}|V_i = v_i)$ abbreviates $\sum_{\phi \in \Phi} \log(\phi[\mathbf{v}|V_i = v_i])$; and $g(\mathbf{v})$ abbreviates $\sum_{\phi \in \Phi} \log(\phi[\mathbf{V} = \mathbf{v}])$.

```
Input: Bayesian network $\mathcal{B} = \langle \mathbf{V}, \mathbf{D}, \mathcal{G}, \Phi \rangle$, evidence $\mathbf{E} = \mathbf{e}$, time bound $t$, cutoff factor cf
Output: Variable assignment $\mathbf{V} = \mathbf{v}$ with highest probability $\prod_{\phi \in \Phi} \phi[\mathbf{V} = \mathbf{v}]$ found in
        time $t$
1  while runtime $< t$ do
       // ===== Initialization.
2      $\mathbf{v_0} \leftarrow$ GenerateInitialSolution($\mathcal{B}$)
3      $\mathbf{v}^* \leftarrow$ LocalSearch($\mathbf{v_0}$)
4      opt $\leftarrow -\infty$
5      iteration $\leftarrow 0$
       // ===== Iterate until restart or time-out.
6      repeat
7          iteration $\leftarrow$ iteration $+ 1$
8          $\mathbf{v} \leftarrow$ Pertubation($\mathbf{v}^*$, history)
9          $\mathbf{v} \leftarrow$ LocalSearch($\mathbf{v}$)
10         $\mathbf{v}^* \leftarrow$ AcceptanceCriterion($\mathbf{v}^*, \mathbf{v}$, history)
11         if $g(\mathbf{v}^*) >$ opt then
12             opt $\leftarrow g(\mathbf{v}^*)$
13             $it_{opt} \leftarrow$ iteration
14     until iteration $>$ cf $\times$ $it_{opt}$ or runtime $\geq t$

15 Function LocalSearch ($\mathbf{v}$)
16 begin
       // ===== Flip best variables until in local minimum.
17     while true do
18         Randomly pick $V_i \in \mathbf{V} \setminus \mathbf{E}$ and $v_i \in D_{V_i}$ maximizing $g(\mathbf{v}|V_i = v_i)$.
19         if $g(\mathbf{v}|V_i = v_i) > g(\mathbf{v})$ then
20             Flip $V_i$ to $v_i$.
21         else
22             return $\mathbf{V} = \mathbf{v}$ // No improving step possible.
23 end
```

probability of accepting a new solution goes to $exp(0) = 1$ like in RW, and as it decreases, the probability goes to $exp(-\infty) = 0$ like in BETTER. The actual values we considered for *ap* and *T* are given in Section 7.5 on page 79 which deals with tuning the parameters of ILS.

For the pertubation, we considered two basic variants. The simpler variant VARS randomly chooses p variables and changes their instantiation to some other random value from their respective domain. A more localized pertubation POTS is achieved by randomly picking a potential $\phi \in \Phi$ and randomly changing the values of all variables $V \in \mathcal{V}_\phi$ to new values. This process is iterated, never changing variables more than once per pertubation, until the value of at least p variables has been changed. The number p of perturbed variables is called the *pertubation strength* and is another parameter of the algorithm.

A priori, it is also not clear that the optimal pertubation strength should be a fixed number; to obtain best performance, it may be better to let it grow, for example, linearly with some measure of problem size, such as the number of variables or the total number of potential entries. Here, we chose the total number of possible instantiations of single variables, *VV* $= \sum_{V \in \mathbf{V}} |D_V|$. This is similar to using the number of variables in the instance, but also considers the average domain size. The main reason for choosing *VV* was that it is the number of variables in a straightforward Max-SAT encoding of the instance. For a possibly varying pertubation strength, we use the Boolean parameter (*prel* for relative pertubation) which when true means that parameter p is multiplied by $0.01 \times$ *VV* and rounded up.

A means of making the pertubation more target-driven is to fix the perturbed variables after changing them, and to perform a local search during the pertubation phase in order to adjust the rest of the variables. Only after this additional local search, the perturbed variables are released again. If the Boolean parameter *pfix* is true, such a constrained local search is executed at the end of the pertubation, using an aspiration criterion to also flip fixed variables if this improves the best solution found so far in the current try.

5.4 Hybrid Algorithm

Although our new SLS algorithms GLS$^+$ and ILS outperform the state-of-the-art SLS algorithms GLS and G+StS by several orders of magnitude on all problem instances we tried, there still remains a number of instances too hard for SLS algorithms. In particular, the `diabetes` network and randomized versions of the `munin4` networks have not been solved by any SLS algorithm to date. However, these networks have very low induced width and, more importantly, manageable induced size. Thus, they can be solved exactly by the MB* algorithm.

Another very interesting observation we made is that Anytime MB sometimes finds the optimal upper bound on solution quality very quickly while it fails to find a matching assignment. E.g., for the `link` network without evidence variables, Anytime MB finds the optimal upper bound within a few milliseconds. This is particularly noteworthy since it is not able to find the correct lower bound even with a memory constraint of 4GB and unlimited time. GLS$^+$ on the other hand consistently solves this problem instance within 200 milliseconds, such that GLS$^+$ and Anytime MB together can find the optimal solution and proof its optimality in time well below a second.

As the `link` example shows, it is possible to fruitfully combine Anytime MB and SLS algorithms to proof optimality of solutions found with SLS. Clearly, any method that computes upper bounds can be plugged in for Anytime MB and any method computing lower bounds can be substituted for SLS. In an algorithm portfolio, it is also possible to use more than one algorithm to compute upper and lower bounds. Indeed, in our hybrid algorithm for MPE (see Algorithm 5.5 on the next page), MB*, ILS, and GLS$^+$ are used for lower bound computation, while only MB* computes an upper bound. Our hybrid basically loops through the algorithms, allocating more resources each iteration, computing upper and lower bounds on solution quality until the bounds coincide or it runs out of resources. As we demonstrate in Chapter 9, it shows very stable performance across all types of instances we considered. Overall, it is the best-performing algorithm.

Algorithm 5.5: Hybrid of MB*, ILS, and GLS$^+$ for MPE (to be continued)
This hybrid algorithm loops through MB*, ILS, and GLS$^+$, allocating more resources each iteration and computing an upper bound on solution quality (MB*), as well as a lower bound (all algorithms). Once lower and upper bound match, the algorithm terminates. ILS and GLS$^+$ are called with their default parameters, and are restarted from scratch everytime they are called.
$g(\mathbf{v})$ abbreviates $\sum_{\phi \in \Phi} \log(\phi[\mathbf{V} = \mathbf{v}])$.

```
Input: Bayesian network $\mathcal{B} = \langle \mathbf{V}, \mathbf{D}, \mathcal{G}, \Phi \rangle$, evidence $\mathbf{E} = \mathbf{e}$, time bound $t$, maximal size
       bound $size_{max}$ for Mini-Buckets
Output: Variable assignment $\mathbf{V} = \mathbf{v}$ with highest probability $\prod_{\phi \in \Phi} \phi[\mathbf{V} = \mathbf{v}]$ found in
        time $t$, upper bound $ub$ on solution quality

// ===== Initialize variable assignment, upper bound, and size for MB$^*$.
1  Initialize $\mathbf{v}$ randomly.
2  $ub \leftarrow \infty$
3  $size \leftarrow 10000$

// ===== Run with increasing size until optimality proofed, maximal size reached or
         time-out.
4  while $runtime < t$ and size $<$ size$_{max}$ do
5      $[\tilde{\mathbf{v}}, ub[size]] \leftarrow$ MB$^*(size)$
6      if $ub[size] \leq ub$ then $ub \leftarrow ub[size]$
7      if $g(\tilde{\mathbf{v}}) > g(\mathbf{v})$ then
8          $\mathbf{v} \leftarrow \tilde{\mathbf{v}}$
9          if $g(\mathbf{v}) = ub$ then return $[\mathbf{v}, ub]$ // optimality proven.
10     $t_{sls} \leftarrow$ time taken by MB$^*(size)$
11     $\tilde{\mathbf{v}} \leftarrow ILS(t_{sls})$
12     if $g(\tilde{\mathbf{v}}) > g(\mathbf{v})$ then
13         $\mathbf{v} \leftarrow \tilde{\mathbf{v}}$
14         if $g(\mathbf{v}) = ub$ then return $[\mathbf{v}, ub]$ // optimality proven.
15     $\tilde{\mathbf{v}} \leftarrow GLS^+(t_{sls})$
16     if $g(\tilde{\mathbf{v}}) > g(\mathbf{v})$ then
17         $\mathbf{v} \leftarrow \tilde{\mathbf{v}}$
18         if $g(\mathbf{v}) = ub$ then return $[\mathbf{v}, ub]$ // optimality proven.
19     $size \leftarrow 2 \times size$
```

Algorithm 5.5: Hybrid of MB*, ILS, and GLS$^+$ for MPE (continued)
$g(\mathbf{v})$ abbreviates $\sum_{\phi \in \Phi} \log(\phi[\mathbf{V} = \mathbf{v}])$.

```
   // ===== Run ILS and GLS$^+$ until optimality proofed or time-out.
20 while $runtime < t$ do
21     $\tilde{\mathbf{v}} \leftarrow ILS(t_{sls})$
22     if $g(\tilde{\mathbf{v}}) > g(\mathbf{v})$ then
23         $\mathbf{v} \leftarrow \tilde{\mathbf{v}}$
24         if $g(\mathbf{v}) = ub$ then return $[\mathbf{v}, ub]$ // optimality proven.
25     $\tilde{\mathbf{v}} \leftarrow GLS^+(t_{sls})$
26     if $g(\tilde{\mathbf{v}}) > g(\mathbf{v})$ then
27         $\mathbf{v} \leftarrow \tilde{\mathbf{v}}$
28         if $g(\mathbf{v}) = ub$ then return $[\mathbf{v}, ub]$ // optimality proven.
29     $t_{sls} \leftarrow 2 \times t_{sls}$
30 return $[\mathbf{v}, ub]$ // Time-out, no optimality proven.
```

Chapter 6

Efficient Implementation

In this chapter, we show how the algorithms introduced in Chapter 5 can be implemented efficiently. We introduce two novel caching schemes that apply to general SLS algorithms for MPE and yield a speed-up of up to two orders of magnitude over the caching scheme used so far in state-of-the-art SLS algorithms for MPE.

In all algorithms presented in Chapter 5, we use two high-level operations: *picking* a variable and its new value to maximize or minimize the evaluation function; and *flipping* the picked variable to its new value. This chapter is entirely devoted to explaining how these operations can be implemented efficiently. The caching schemes we introduce closely resemble the ones used in the SAT domain sketched out in Section 3.4 on page 22.

Since the caching schemes we introduce here are not trivial, we need to provide descriptions at a technically detailed level. Note that each potential ϕ is generally stored in a 1-dimensional array each entry $\phi[\mathcal{V}_\phi = v_\phi]$ of which represents the probability for a particular instantiation of its variables. Each variable $V_i \in \mathcal{V}_\phi$ is assigned a blocksize B_{ϕ,V_i}; ϕ's current index I_ϕ then computes as $\sum_{V_i \in \mathcal{V}_\phi} B_{\phi,V_i} \times \hat{v}_i$, where $\hat{v}_i$ denotes a unique value in $\{0, \ldots, |D_{V_i}| - 1\}$ V_i's current value v_i is mapped to.[1] When a variable is flipped from value $\tilde{v}_i$ to v_i, the index changes by $B_{\phi,V_i} \times (\hat{v}_i - \hat{\tilde{v}}_i)$, which we abbreviate by $B_{\phi,\tilde{v}_i \rightarrow v_i}$.

6.1 Caching Scheme *Naïve*

Straightforward implementations of SLS algorithms usually do not employ much caching, mostly spending considerable time to determine which variable to flip while the actual flip is done quickly. Our most basic implementation variant

[1]Any bijective function from D_{V_i} to $\{0, \ldots, |D_{V_i}| - 1\}$ can be used for this mapping.

Naïve (see Caching scheme 6.1 on page 54) picks the best variable-value pair by simply trying all possible flips and evaluating the resulting instantiations. Since absolutely no caching has to be done in this variant, procedure *Flip-naïve* (which flips the variable in the naïve caching scheme) has time complexity $\Theta(1)$. However, a lot of work remains for picking the variable: for each possible new variable-value combination, the log probability of the resulting assignment is computed from scratch, which requires computing the current index of each potential. Denoting the maximal domain size by $|D_V|$ and the maximal number of variables in a potential by $|\mathcal{V}_\phi|$, *Pick-naïve*'s time complexity is $O(|\mathbf{V}| \times |D_V| \times |\Phi| \times |\mathcal{V}_\phi|)$, because for each possible new variable-value combination, each potential ϕ_i must be evaluated and computing ϕ_i's index I_{ϕ_i} from scratch is linear in the number of variables $|\mathcal{V}_{\phi_i}|$ in its scope. As we will see in our experimental comparison of different caching variants at the end of this section, this is far too complex to yield an efficient local search method; we only include this naïve caching scheme here in order to establish a baseline for the assessment of the more efficient implementations discussed in the following.

For the penalty-based approaches GLS and GLS$^+$, some additional work is required when the penalties are updated in local minima. In these algorithms, the penalties associated with the current entries of potentials with maximal utility are increased in every local minimum. Denoting the number of such potentials with equal maximal utility by $|\Phi_{\text{maxutil}}|$, the complexity of this update is $\Theta(|\Phi_{\text{maxutil}}|)$ if the current potential indices I_{ϕ_i} are cached and $O(|\Phi_{\text{maxutil}}| \times |\mathcal{V}_\phi|)$ otherwise. When smoothing the penalties after a certain number of local minima, all penalties need to be multiplied by the smoothing parameter ρ, yielding time complexity $O(|\Phi| \times S_\phi)$, where S_ϕ denotes the maximal size of any potential in the network.

6.2 Caching Scheme *Simple*

Our second caching variant *Simple* (see Caching scheme 6.2 on page 55) is the one used in all previous implementations of SLS algorithms for MPE we are aware of. In particular, this includes the best performing SLS algorithms for MPE to date [KD99b, Par02, MKD03]. There are two differences between this caching variant and the first one, namely caching of the potential indices I_ϕ in the flip procedure; and the local evaluation of a new variable-value pair (V_i, v_i) by inspecting only the potentials $\Phi_{V_i} = \{\phi \mid V_i \in \mathcal{V}_\phi\}$, that span V_i and whose current index is thus affected by changing it.[2] In Bayesian networks, these potentials are the CPTs

[2]The equivalent to this method in SAT algorithms is to only check for changes in the satisfaction status of clauses that contain the flipped variable.

of variable V_i and its children *ch*(V_i).

We denote by $|\Phi_V|$ the maximal number of potentials that span a particular variable. When compared to the previous caching scheme, the complexity of *Pick-simple* has decreased to $O(|\mathbf{V}| \times |D_V| \times |\Phi_V|)$, whereas the complexity of *Flip-simple* has increased to $\Theta(|\Phi_{V_i}|)$ when variable V_i is flipped. Since each of these two actions is performed once per search step, the total complexity of this approach is $O(|\mathbf{V}| \times |D_V| \times |\Phi_V|)$ per search step. This is much lower than $O(|\mathbf{V}| \times |D_V| \times |\Phi| \times |\mathcal{V}_\phi|)$, the search step complexity of caching scheme *Naïve*, but is still subject to substantial further improvement.

6.3 Caching Scheme *Scores*

Our first novel caching scheme *Scores* (see Caching scheme 6.3 on page 58) improves on the standard variant by caching the change in evaluation function (the *score*) each possible variable flip yields.[3] This reduces the time complexity for picking a variable to $O(|\mathbf{V}| \times |D_V|)$. Denoting the maximal number of variables in any potential by $|\mathcal{V}_\phi|$, the complexity of flipping a variable V_i increases to $O(|\Phi_{V_i}| \times |\mathcal{V}_\phi| \times |D_V|)$ since for each potential ϕ containing V_i, the scores for all values of all other variables in $\mathcal{V}_\phi$ change when V_i is flipped and need to be updated correspondingly. Neither the time complexity of picking a new variable-value combination nor the one of flipping the variable now strictly dominates the other. However, since graphical models emphasize modularity, the number of potentials a single variable occurs in as well as the number of variables per potential is usually small as compared to the total number of variables. Thus, the picking still dominates time complexity in practice such that there is room for even further improvement.

If the underlying SLS algorithm is penalty-based, this caching variant requires some additional work. In this case, we cache the change in the constituents of the evaluation function, i.e. the change $L[V_i][v_i]$ in log-probability and the change $P[V_i][v_i]$ in the summed applicable penalties. For GLS, the score $S[V_i][v_i]$ to be *minimized* by every variable flip is simply $P[V_i][v_i]$, and for GLS$^+$, the score to be *maximized* computes as $L[V_i][v_i] - w \times P[V_i][v_i]$.

Whenever the penalty $\lambda_\phi[\mathbf{V} = \mathbf{v}]$ associated with a potential entry $\phi[\mathcal{V}_\phi = v_\phi]$ is increased by 1 in a local optimum, for all variables $V_j \in \mathcal{V}_\phi$ and all values $\hat{v}_j \in D_{V_j} \setminus \{v_j\}$, we need to decrease $P[V_j][\hat{v}_j]$ by the same amount, where v_j

[3] In the SAT domain, this compares to caching the number of clauses the flip of a variable would satisfy minus the number of clauses it would unsatisfy. These quantities have also been called the make-count and break-count of a variable. For more details on the efficient implementation of SAT algorithms, see [Hoo98].

Caching scheme 6.1: *Naïve*

Naïve variant of picking the best variable-value pair by computing the log probability for each possible new variable-value pair from scratch. The potential indices I_ϕ are recomputed in every variable flip.
$\epsilon = 10^{-6}$ is a small constant necessary for a stable comparison of real numbers. It will be employed by all our caching schemes.

```
1  Function Pick-naïve
   Input: Bayesian network $\mathcal{B} = \langle \mathbf{V}, \mathbf{D}, \mathcal{G}, \Phi \rangle$, evidence $\mathbf{E} = \mathbf{e}$, variable assignment $\mathbf{V} = \tilde{\mathbf{v}}$
   Output: Variable-value pair $\langle V_i, v_i \rangle$ maximizing $\sum_{\phi \in \Phi} \phi[\mathbf{V} = \tilde{\mathbf{v}} | V_i = v_i]$
2  begin
3      opt $\leftarrow -\infty$
4      Best $\leftarrow \emptyset$
5      foreach $V_i \in \mathbf{V} \setminus \mathbf{E}$ and $v_i \in D_{V_i} \setminus \{\tilde{v}_i\}$ do
           // ===== Compute score for $\langle V_i, v_i \rangle$.
6          score $\leftarrow 0$
7          foreach $\phi \in \Phi$ do
8              $I_\phi \leftarrow 0$
9              foreach $V_j \in \mathcal{V}_\phi$ do $I_\phi \leftarrow I_\phi + B_{\phi, v_j}$
10             score $\leftarrow$ score $+ \phi[I_\phi]$
           // ===== Compare score of $\langle V_i, v_i \rangle$ with other scores.
11         if score $>$ opt $- \epsilon$ then
12             if score $>$ opt $+ \epsilon$ then
13                 opt $\leftarrow$ score
14                 Best $\leftarrow \emptyset$
15             Best $\leftarrow$ Best $\cup \{\langle V_i, v_i \rangle\}$
16     return randomly sampled element from Best
17 end
18 Procedure Flip-naïve
   Input: Bayesian network $\mathcal{B} = \langle \mathbf{V}, \mathbf{D}, \mathcal{G}, \Phi \rangle$, previous variable assignment $\mathbf{V} = \tilde{\mathbf{v}}$, new
          variable-value pair $\langle V_i, v_i \rangle$.
   Effect: $V_i$ is flipped from $\tilde{v}_i$ to $v_i$.
19 begin
20     $V_i \leftarrow v_i$
21 end
```

Caching scheme 6.2: *Simple*
This is like Caching scheme 6.1 on the facing page, but the potential indices I_ϕ are cached and the score of each variable flip $\tilde{v}_i \to v_i$ is computed by its local effects on the potentials Φ_{V_i} that contain variable V_i. $\phi[x]$ denotes the xth entry of potential ϕ in its array representation.

```
Function Pick-simple
Input: Bayesian network $\mathcal{B} = \langle \mathbf{V}, \mathbf{D}, \mathcal{G}, \Phi \rangle$, evidence $\mathbf{E} = \mathbf{e}$, variable assignment $\mathbf{V} = \tilde{\mathbf{v}}$,
       potential index $I_\phi$ for each $\phi \in \Phi$
Output: Variable-value pair $\langle V_i, v_i \rangle$ maximizing $\sum_{\phi \in \Phi} \phi[\mathbf{V} = \tilde{\mathbf{v}} | V_i = v_i]$
begin
    opt $\leftarrow -\infty$
    Best $\leftarrow \emptyset$
    foreach $V_i \in \mathbf{V} \setminus \mathbf{E}$ and $v_i \in D_{V_i} \setminus \{\tilde{v}_i\}$ do
        score $\leftarrow 0$
        foreach $\phi \in \Phi_{V_i}$ do score $\leftarrow$ score $+ (\phi[I_\phi + B_{\phi,\tilde{v}_i \to v_i}] - \phi[I_\phi])$
        if score $>$ opt $- \epsilon$ then
            if score $>$ opt $+ \epsilon$ then
                opt $\leftarrow$ score
                Best $\leftarrow \emptyset$
            Best $\leftarrow$ Best $\cup \{\langle V_i, v_i \rangle\}$
    return randomly sampled element from Best
end
Procedure Flip-simple
Input: Bayesian network $\mathcal{B} = \langle \mathbf{V}, \mathbf{D}, \mathcal{G}, \Phi \rangle$, previous variable assignment $\mathbf{V} = \tilde{\mathbf{v}}$, new
       variable-value pair $\langle V_i, v_i \rangle$, potential index $I_\phi$ for each $\phi \in \Phi$.
Effect: $V_i$ is flipped from $\tilde{v}_i$ to $v_i$, potential indices are updated.
begin
    foreach $\phi \in \Phi_{V_i}$ do $I_\phi \leftarrow I_\phi + B_{\phi,\tilde{v}_i \to v_i}$
    $V_i \leftarrow v_i$
end
```

is V_j's current value. Thus, the time complexity required for this update is now $O(|\Phi_{\text{maxutil}}| \times |\mathcal{V}_\phi| \times |D_V|)$. Since there are mostly only few or just one potential with maximal utility, this additional complexity remains manageable in practice. Also, it only applies in local minima of the evaluation function, albeit empirical analysis shows that local minima are encountered as often as every third to fifth search step. A similar behaviour has been observed for current state-of-the-art Dynamic Local Search algorithms for SAT, such as SAPS [HTH02].

The smoothing operation in GLS-type algorithms also requires some additional work for this caching type. When smoothing, all penalties are multiplied by ρ, such that the difference in summed applicable penalties in the current instantiation and neigbouring instantiations is also multiplied by ρ. In order to reflect this in the cached differences of penalties, for all variables $V_i \in \mathbf{V}$ and all values $v_i \in D_{V_i}$, $P[V_i][v_i]$ is multiplied by ρ. The additional time complexity due to this caching of the penalties in the smoothing is only $O(|\mathbf{V}| \times |D_V|)$, much smaller than the complexity $O(|\Phi| \times S_\phi)$ caused by the smoothing anyways. The complexity of single search steps decreases as we introduce more and more sophisticated caching schemes. In practice, with caching scheme *Scores*, its complexity is already much smaller than the one of smoothing all the penalties. Therefore, for complexity reasons it is central not to perform the smoothing in each local minimum but only every N_ρ local minima.[4]

6.4 Caching Scheme *Improving*

Our second novel caching scheme *Improving* is detailed in Caching scheme 6.4 on page 59. It further reduces the average time complexity of picking a new variable-value pair considerably, while only marginally increasing the time complexity for performing a flip. This is achieved by caching the variables $\mathbf{V}_{imp}$ which when flipped to some value actually lead to an improvement. Because after a short initial search phase $|\mathbf{V}_{imp}|$ remains very low, considerable performance improvements can be achieved since the time complexity for picking a new variable-value pair decreases to $O(|\mathbf{V}_{imp}| \times |D_V|)$. Denoting the maximal cardinality of any variable's Markov Blanket by $|mb(V)|$, the time complexity for flipping the variables increases to $O(\, |\Phi_V| \times |\mathcal{V}_\phi| \times |D_V| + |mb(V)| \times |D_V| \times \log |\mathbf{V}_{imp}|\,)$. This is because

[4]The exact same observation in the SAT domain led to the development of SAPS [HTH02], which is amongst the state-of-the-art algorithms for SAT. The main difference between this algorithm and its predecessor ESG is that the smoothing is only executed in local minima with a low probability of around 5%; this simple change leads to SAPS consistently outperforming ESG by a large margin. Recent subsequent work [TH04] showed that the same effect persists if the smoothing is carried out deterministically every N_ρ steps.

after flipping a variable V_i like in caching scheme *Scores*, we now additionally need to check for each variable V_j in V_i's Markov Blanket whether the updated score $S[V_j][v_j]$ for any of its values v_j yields an improvement. If this is the case, V_j has to be incorporated into the initially empty set of improving variables $\mathbf{V}_{imp}$ unless it is already contained in it; vice versa, if V_j was contained in $\mathbf{V}_{imp}$ but flipping it to another value cannot lead to an improvement anymore, V_j has to be removed from $\mathbf{V}_{imp}$.

For the GLS variants, the same additional work as in the last caching scheme *Scores* has to be performed again when changing the penalties. For completeness, we mention that in GLS, the set of improving variables $\mathbf{V}_{imp}$ holds all variables which when flipped to some value *reduce* the overall penalty and for GLS$^+$ the ones which *increase* the log-probability minus the weighted overall penalty. Table 6.1 on page 60 gives an overview of the complexities for picking the best new variable-value combination and flipping a variable to a new value for all our caching schemes. For the caching of penalties in penalty-based algorithms, the additional complexities in Table 6.2 on page 60 apply.

In our discussion of the complexities arising from each caching scheme, we focussed on the complexity per search step and completely ignored the complexity of initializing the employed data structures after an initial solution has been determined. In caching scheme *Naïve*, no initialization is needed, and in caching scheme *Simple*, only the indices of each potential need to be computed, causing complexity $O(|\Phi| \times |\mathcal{V}_\phi|)$. For caching scheme *Scores*, we also need to compute the score for every possible variable flip, causing an additional one-time cost of $O(|\mathbf{V}| \times |D_V| \times |\Phi| \times |\mathcal{V}_\phi|)$. The initialization of caching scheme *Improving* causes another small one-time cost of $O(|\mathbf{V}| \times |D_V| + |\mathbf{V}| \times \log |\mathbf{V}_{imp}|)$ since for each value v_i of each variable V_i, we need to check whether the score $S[V_i][v_i]$ is greater than zero, and, if this is the case for any of the values, add V_i to the set of improving variables $\mathbf{V}_{imp}$.

6.5 Experimental Evaluation of Caching Schemes

Having compared the various caching schemes with respect to their theoretical time complexity, we now give evidence from practice underlining the significance of these results. Table 6.3 on page 63 and Table 6.4 on page 64 show the average number of steps per second performed by the algorithms G+StS, GLS, and ILS for problem sets `bnrep` and `gen`, respectively. For GLS$^+$, the number of steps for each of the caching schemes is virtually identical to GLS, and we omit it in the tables. We observe an enormous effect of efficient caching schemes. On problem

Caching scheme 6.3: *Scores*

This is like Caching scheme 6.2 on page 55, but additionally caches the score $S[V_i][v_i]$ of each variable-value combination (V_i, v_i), i.e. the increase in log-probability when V_i is flipped to v_i. The scores S are updated when flipping a variable and the pick operation simply picks the best one.

```
1  Function Pick-scores
   Input: Bayesian network $\mathcal{B} = \langle \mathbf{V}, \mathbf{D}, \mathcal{G}, \Phi \rangle$, evidence $\mathbf{E} = \mathbf{e}$, variable assignment $\mathbf{V} = \tilde{\mathbf{v}}$,
          2-dim. score array $S$
   Output: Variable-value pair $\langle V_i, v_i \rangle$ maximizing $\sum_{\phi \in \Phi} \phi[\mathbf{V} = \tilde{\mathbf{v}} | V_i = v_i]$
2  begin
3      opt $\leftarrow -\infty$
4      Best $\leftarrow \emptyset$
5      foreach $V_i \in \mathbf{V} \setminus \mathbf{E}$ and $v_i \in D_{V_i} \setminus \{\tilde{v}_i\}$ do
6          if $S[V_i][v_i] >$ opt $- \epsilon$ then
7              if $S[V_i][v_i] >$ opt $+ \epsilon$ then
8                  opt $\leftarrow S[V_i][v_i]$
9                  Best $\leftarrow \emptyset$
10             Best $\leftarrow$ Best $\cup \{\langle V_i, v_i \rangle\}$
11     return randomly sampled element from Best
12 end
13 Procedure Flip-scores
   Input: Bayesian network $\mathcal{B} = \langle \mathbf{V}, \mathbf{D}, \mathcal{G}, \Phi \rangle$, previous variable assignment $\mathbf{V} = \tilde{\mathbf{v}}$, new
          variable assignment $V_i = v_i$, potential index $I_\phi$ for each $\phi \in \Phi$, 2-dim. score array
          $S$.
   Effect: $V_i$ is flipped from $\tilde{v_i}$ to $v_i$, indices and scores are updated.
14 begin
15     foreach $\phi \in \Phi_{V_i}$ do
16         $\tilde{I}_\phi \leftarrow I_\phi$ // $\tilde{I}_\phi$ holds the index before flipping $V_i$ from $\tilde{v}_i$ to $v_i$.
17         $I_\phi \leftarrow I_\phi + B_{\phi,\tilde{v}_i \to v_i}$
           // ===== The local change in log-probability for this potential and the current flip
                  $\tilde{v}_i \to v_i$ is LC $= \phi[I_\phi] - \phi[\tilde{I}_\phi]$.
18         foreach $v_i \in D_{V_i}$ do  $S[V_i][v_i] \leftarrow S[V_i][v_i] + \phi[\tilde{I}_\phi] - \phi[I_\phi]$ // deal with LC
           // ===== Before the flip $\tilde{v}_i \to v_i$, the local change for other variable flips $\tilde{v}_j \to v_j$
                  is $\phi[\tilde{I}_\phi + B_{\phi,\tilde{v}_j \to v_j}] - \phi[\tilde{I}_\phi]$; afterwards, it is $\phi[I_\phi + B_{\phi,\tilde{v}_j \to v_j}] - \phi[I_\phi]$.
                  LC cancels out with two of these terms.
19         foreach $V_j \in \mathcal{V}_\phi \setminus \{V_i\}$ and $v_j \in D_{V_j}$ do
20             $S[V_j][v_j] \leftarrow S[V_j][v_j] - \phi[\tilde{I}_\phi + B_{\phi,\tilde{v}_j \to v_j}] + \phi[I_\phi + B_{\phi,\tilde{v}_j \to v_j}]$
21 end
```

Caching scheme 6.4: *Improving*

This is like Caching scheme 6.3 on the preceding page, but additionally caches the set of variables $\mathcal{V}_{imp}$ which yield an improvement in log probability when flipped to some value.

```
1  Procedure Pick-improving
   Input: Bayesian network $\mathcal{B} = \langle \mathbf{V}, \mathbf{D}, \mathcal{G}, \Phi \rangle$, variable assignment $\mathbf{V} = \tilde{\mathbf{v}}$, 2-dim. score
          array $S$, set $\mathcal{V}_{imp}$ of vars yielding an improvement when flipped to some value.
   Output: Variable-value pair $\langle V_i, v_i \rangle$ maximizing $\sum_{\phi \in \Phi} \phi[\mathbf{V} = \tilde{\mathbf{v}} | V_i = v_i]$
2  begin
3      $opt \leftarrow -\infty$
4      $Best \leftarrow \emptyset$
5      foreach $V_i \in \mathcal{V}_{imp}$ and $v_i \in D_{V_i} \setminus \{\tilde{v}_i\}$ do
6          if $S[V_i][v_i] > opt - \epsilon$ then
7              if $S[V_i][v_i] > opt + \epsilon$ then
8                  $opt \leftarrow S[V_i][v_i]$
9                  $Best \leftarrow \emptyset$
10             $Best \leftarrow Best \cup \{\langle V_i, v_i \rangle\}$
11     return randomly sampled element from $Best$
12 end
13 Procedure Flip-improving
   Input: Bayesian network $\mathcal{B} = \langle \mathbf{V}, \mathbf{D}, \mathcal{G}, \Phi \rangle$, evidence $\mathbf{E} = \mathbf{e}$, previous variable assignment
          $\mathbf{V} = \tilde{\mathbf{v}}$, new variable-value pair $\langle V_i, v_i \rangle$, potential index $I_\phi$ for each $\phi \in \Phi$, 2-dim.
          score array $S$, set $\mathcal{V}_{imp}$ of vars yielding an improvement when flipped to some value.
   Effect: $V_i$ is flipped from $\tilde{v}_i$ to $v_i$, indices, scores, and improving variables are updated.
14 begin
       // ===== The first loop is exactly the same as in procedure Flip-scores in Caching
                 scheme 6.3 on the facing page
15     foreach $\phi \in \Phi_{V_i}$ do
16         $\tilde{I}_\phi \leftarrow I_\phi$
17         $I_\phi \leftarrow I_\phi + B_{\phi, \tilde{v}_i \rightarrow v_i}$
18         foreach $v_i \in D_{V_i}$ do  $S[V_i][v_i] \leftarrow S[V_i][v_i] + \phi[\tilde{I}_\phi] - \phi[I_\phi]$
19         foreach $V_j \in \mathcal{V}_\phi \setminus \{V_i\}$ and $v_j \in D_{V_j}$ do
20             $S[V_j][v_j] \leftarrow S[V_j][v_j] - \phi[\tilde{I}_\phi + B_{\phi, \tilde{v}_j \rightarrow v_j}] + \phi[I_\phi + B_{\phi, \tilde{v}_j \rightarrow v_j}]$
21     foreach $V_j \in \mathrm{mb}(V_i) \setminus \mathbf{E}$ do
22         if $\exists v_j \in D_{V_j}$ such that $S[V_j][v_j] > 0$ then
23             $\mathcal{V}_{imp} \leftarrow \mathcal{V}_{imp} \cup \{V_j\}$
24         else
25             $\mathcal{V}_{imp} \leftarrow \mathcal{V}_{imp} \setminus \{V_j\}$
26     $V_i \leftarrow v_i$.
27 end
```

Caching scheme	Picking best variable-value combination	Flipping variable V_i to new value
Naïve	$O(\lvert\mathbf{V}\rvert \times \lvert D_V\rvert \times \lvert\Phi\rvert \times \lvert\mathcal{V}_\phi\rvert)$	$\Theta(1)$
Simple	$O(\lvert\mathbf{V}\rvert \times \lvert D_V\rvert \times \lvert\Phi_V\rvert)$	$\Theta(\lvert\Phi_{V_i}\rvert)$
Scores	$O(\lvert\mathbf{V}\rvert \times \lvert D_V\rvert)$	$O(\lvert\Phi_{V_i}\rvert \times \lvert\mathcal{V}_\phi\rvert \times \lvert D_V\rvert)$
Improving	$O(\lvert\mathbf{V}_{imp}\rvert \times \lvert D_V\rvert)$	$O(\lvert\Phi_V\rvert \times \lvert\mathcal{V}_\phi\rvert \times \lvert D_V\rvert$ $+ \lvert mb(V)\rvert \times \lvert D_V\rvert \times \log \lvert\mathbf{V}_{imp}\rvert)$

Table 6.1: Overview of computational complexity of *picking* the best neighbouring variable-value combination and *flipping* variable V_i to a new value. For GLS-type algorithms, the additional complexities in Table 6.2 apply.

Caching scheme	Penalty increase in local minima	Smoothing of all penalties
Naïve	$O(\lvert\Phi_{\text{maxutil}}\rvert \times \lvert\mathcal{V}_\phi\rvert)$	$O(\lvert\Phi\rvert \times S_\phi)$
Simple	$\Theta(\lvert\Phi_{\text{maxutil}}\rvert)$	$O(\lvert\Phi\rvert \times S_\phi)$
Scores	$O(\lvert\Phi_{\text{maxutil}}\rvert \times \lvert\mathcal{V}_\phi\rvert \times \lvert D_V\rvert)$	$O(\lvert\Phi\rvert \times S_\phi + \lvert\mathbf{V}\rvert \times \lvert D_V\rvert)$
Improving	$O(\lvert\Phi_{\text{maxutil}}\rvert \times \lvert\mathcal{V}_\phi\rvert \times \lvert D_V\rvert)$	$O(\lvert\Phi\rvert \times S_\phi + \lvert\mathbf{V}\rvert \times \lvert D_V\rvert)$

Table 6.2: Overview of computational complexity for *increasing* the penalties associated with the current entries of potentials with maximal utility and *smoothing* all penalties in GLS-type algorithms.

set `bnrep`, the speedup factor achieved by our caching scheme *Improving* over the previous state-of-the-art caching scheme *Simple* reaches from 1.9 to 116 for G+StS, from 2.5 to 39 for GLS, and from 1.16 to 110 for ILS. On problem set `gen`, this speedup factor ranges from 6.26 to 46 for G+StS, from 5.9 to 17 for GLS, and from 3.3 to 34 for ILS.

There are small differences in the number of search steps G+StS and ILS execute per second, with G+StS being slightly faster on average. We attribute these differences to two factors. Firstly, G+StS performs 40% of Stochastic Simulation steps, the performance of which is not bound as tightly to the caching schemes as picking the best new variable-value combination.[5] And secondly, a special situation occurs in ILS when the acceptance criterion decides to go back to reuse the local optimum $\mathbf{v}^*$ of the previous step. In this very frequent case[6], ILS performs a series

[5]Stochastic Simulation steps sample a variable V_i and then sample a value for this variable from the distribution $P(V_i = v_i \mid [\mathbf{V} \setminus \{V_i\} = \mathbf{v} \setminus \{v_i\}])$. Since this equals the change in overall probability when V_i is flipped to v_i, the new value for V_i can be sampled from the exponentiated scores $S[V_i][.]$. For caching scheme *Naïve*, these scores are computed from scratch, for caching scheme *Simple*, the cached indices are employed for the computation, and for the other two caching schemes, the scores are readily available.

[6]The new locally optimal variable instantiation $\mathbf{v}$ very often has smaller probability than the

of additional variable flips which simply redo the flips performed since leaving $\mathbf{v}^*$. Effectively, this leads to ILS performing almost twice as many flip-operations as pick-operations.

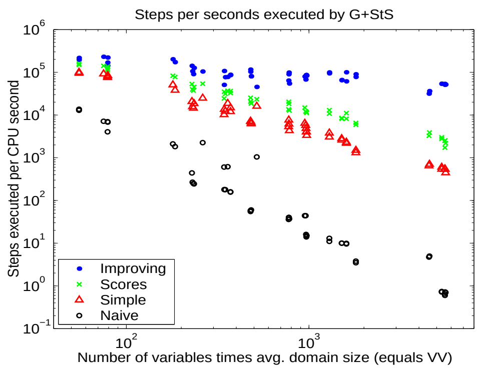
When comparing the CPU time per search step of GLS to G+StS and ILS, more significant differences can be found. For the large networks `munin2` to `munin4`, GLS performs only approximately 15% of the steps G+StS and ILS perform per second. We attribute these differences to the additional overhead in GLS stemming from incrementing and smoothing its penalties. Interestingly, smoothing seems to contribute only a small part to this additional complexity, since in preliminary experiments (not reported here), we observed a very similar performance of GLS without smoothing.[7]

Note that for the small networks `alarm`, `insurance`, and `water`, the effects of improved caching schemes are minimal, but that the speedup factor grows with the number of variables and the domain sizes, being highest for the large instances `diabetes`, `link`, and the `munin` networks. This effect is highlighted by the results for problem set `gen`, where we observe that the speedup factor consistently grows with the number of variables and the domain sizes. We visualize these increasing speedups for larger instances for each of the algorithms separately in Figures 6.1(a) on the next page, 6.1(b) on the following page, and 6.2(a) on the next page, where every data point represents one instance.

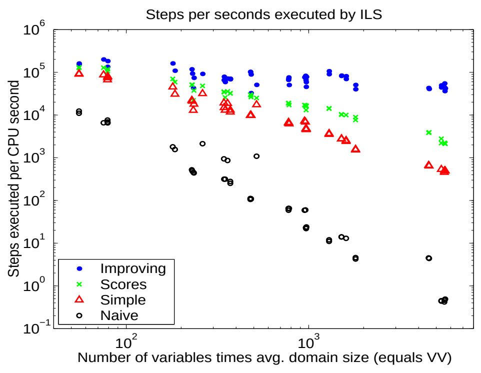
Having substantially improved the time complexity SLS algorithms for MPE exhibit per search step, in the next chapter we move on to another important component of efficient SLS algorithms for MPE, namely a thorough parameter tuning.

previous one $\mathbf{v}^*$. In this case, $\mathbf{v}^*$ is used with probability $1 - ap$, which per default is 99.7%.

[7]Although with an infinite smoothing interval N_ρ, GLS's number of performed search steps per second was up to 30% faster for some instances, for other instances, it was up to 20% slower. The most likely reason for it being faster for some instances is clearly that it saves the complexity of smoothing. We conjecture that the reason for it being slower for other instances is that without the smoothing more potential entries might share the same maximal utility, which leads to a greater number of penalty updates being performed.

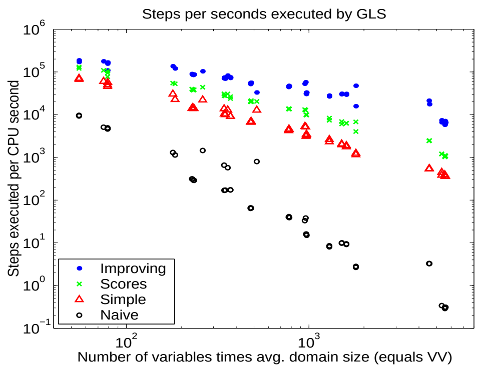


(a) Effect of caching for G+StS

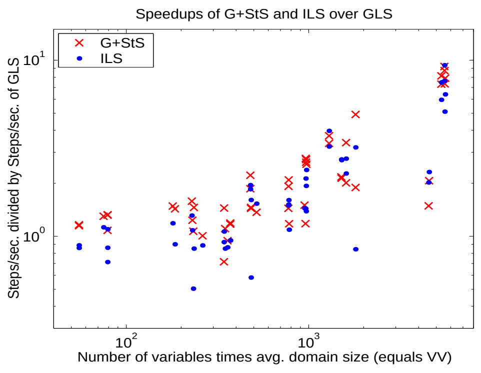


(b) Effect of caching for ILS

Figure 6.1: The effects of improved caching schemes for algorithms G+StS (a) and ILS (b). Note the increasing effect of strong caching schemes with an increasing number of variables and domain size (here only the product of the two is plotted).



(a) Effect of caching for GLS



(b) Speedup of G+StS and ILS over GLS

Figure 6.2: The effects of improved caching schemes for algorithm GLS (a), and the advantage in steps executed by algorithms G+StS and ILS when compared to GLS when all algorithms are using our improved caching scheme *Improving*.

	Stats		G+StS					GLS					ILS				
Instance	*N*	*Dom*	c=0 st./s.	c=1 st./s.	c=2 st./s.	c=3 st./s.	*sf*	c=0 st./s.	c=1 st./s.	c=2 st./s.	c=3 st./s.	*sf*	c=0 st./s.	c=1 st./s.	c=2 st./s.	c=3 st./s.	*sf*
alarm	27	2.78	7142	91904	141429	230507	2.51	5081	60322	108826	177453	2.94	6553	87189	128043	199495	2.29
alarm-rand	27	2.93	6831	84573	136225	220847	2.61	4723	56792	104816	166824	2.94	6530	80038	118695	182842	2.28
barley	38	9.45	616	18741	36912	77774	4.15	574	12904	30499	82396	6.39	861	18774	35621	71245	3.79
barley-rand	38	9.03	606	10210	24364	50678	4.96	658	13750	30805	70759	5.15	941	19212	35144	65532	3.41
diabetes	403	11.33	4.99	704	3897	36327	51.60	3.26	546	2486	17579	32.20	4.39	644	3920	40708	63.21
diabetes-rand	403	11.27	4.72	648	3264	31602	48.77	3.28	539	2432	21234	39.40	4.52	664	3848	42888	64.59
hailfinder	46	3.91	2106	50843	82890	201929	3.97	1304	30422	54829	136065	4.47	1799	45468	69994	161259	3.55
hailfinder-rand	46	4.02	1820	38035	77417	172865	4.54	1141	22645	52938	120753	5.33	1558	30725	58823	108685	3.54
insurance	17	3.24	12914	94800	149009	196759	2.08	9316	67159	121247	171379	2.55	12369	92942	126519	152300	1.64
insurance-rand	17	3.24	13640	100415	159976	217262	2.16	9635	71098	130910	187092	2.63	10969	90674	131232	160499	1.77
link	714	2.53	3.80	1310	6522	89915	68.64	2.81	1253	6788	47495	37.91	4.59	1520	7654	40080	26.37
link-rand	714	2.53	3.48	1494	5908	77652	51.98	2.66	1159	4021	15790	13.62	4.26	1579	8892	50498	31.98
mildew	25	20.72	1044	16073	23396	45343	2.82	800	12801	20356	33148	2.59	1084	17406	25078	50753	2.92
mildew-rand	25	10.48	2257	24854	54213	104124	4.19	1451	22124	43800	103526	4.68	2128	31753	48110	91768	2.89
munin1	179	5.30	44	6404	14910	79571	12.43	33	5275	13158	52968	10.04	59	7221	17078	76351	10.57
munin1-rand	179	5.36	44	5722	12125	67727	11.84	38	5175	12745	57449	11.10	60	6854	16658	82531	12.04
munin2	993	5.37	0.72	539	2998	53790	99.80	0.34	448	1204	6602	14.74	0.44	533	2785	49388	92.66
munin2-rand	993	5.37	0.74	600	2766	53998	90.00	0.34	384	1213	7391	19.25	0.45	533	2205	44028	82.60
munin3	1034	5.36	0.65	546	2492	53458	97.91	0.31	393	1058	5801	14.76	0.42	456	2228	44250	97.04
munin3-rand	1034	5.37	0.60	539	1730	51009	94.64	0.29	364	1029	5842	16.05	0.46	494	2172	54719	110.77
munin4	1031	5.43	0.70	445	2185	51776	116.35	0.31	358	1068	6549	18.29	0.49	494	2126	41879	84.78
munin4-rand	1031	5.40	0.72	539	2541	52134	96.72	0.31	362	1109	7097	19.60	0.48	494	2186	36215	73.31
pigs	431	3.00	13	3061	10761	94288	30.80	8.54	2322	8326	27965	12.04	11	3702	14120	90363	24.41
pigs-rand	431	3.00	11	3791	13002	99247	26.18	8.07	2631	7387	26676	10.14	12	3524	14369	105773	30.02
water	22	3.59	6986	79240	123185	168865	2.13	4907	50213	94355	156294	3.11	7614	74867	106093	134426	1.80
water-rand	22	3.59	4043	75334	112485	143196	1.90	4627	46053	77390	108539	2.36	6880	66769	78393	77440	1.16

Table 6.3: Steps performed per second for the algorithms G+StS, GLS, and ILS on problem set `bnrep` with caching schemes *Naïve* (c=0), *Simple* (c=1), *Scores* (c=2), and *Improving* (c=3). For each algorithm, *sf* denotes the speedup factor gained by our new superior caching scheme *Improving* over the previously best simple caching scheme *Simple*. This speedup is higher for larger instances with many free variables (column N) and/or high domain sizes (column *Dom*). GLS$^+$ performs almost exactly as many steps per seconds as GLS, and we omit it in the table. G+StS and ILS used initialization MB$^*(10^5)$, GLS random initialization. The other default parameters used for the algorithms are $\langle \mathit{cf}, \mathit{np} \rangle = \langle 2, 40 \rangle$ for G+StS, $\langle N_\rho, \rho \rangle = \langle 200, 0.999 \rangle$ for GLS, and $\langle \mathit{acc}, \mathit{an}, \mathit{cf}, p, \mathit{pert}, \mathit{pfix}, \mathit{prel} \rangle = \langle \mathit{HYBRID}, 0.003, 5, 2, \mathit{POTS}, \mathit{true}, \mathit{false} \rangle$ for ILS. GLS with an infitinite smoothing interval N_ρ performed very similar to GLS with default parameters.

	Stats		G+StS					GLS					ILS				
			c=0	c=1	c=2	c=3		c=0	c=1	c=2	c=3		c=0	c=1	c=2	c=3	
Instance	N	Dom	st./s.	st./s.	st./s.	st./s.	*sf*	st./s.	st./s.	st./s.	st./s.	*sf*	st./s.	st./s.	st./s.	st./s.	*sf*
z100v3d5iw10-rand	90	2.54	440	20698	53038	141758	6.85	314	13985	39771	89620	6.41	521	22067	50989	117445	5.32
z100v3d5iw10-struc	90	2.56	267	15656	39678	105943	6.77	312	14575	38462	85948	5.90	489	21092	50966	92900	4.40
z100v3d5iw20-rand	90	2.59	249	14461	37311	90523	6.26	293	14156	37423	84823	5.99	454	12915	21885	42728	3.31
z100v3d5iw20-struc	90	2.61	244	18534	45344	126959	6.85	290	13941	38352	87051	6.24	441	17999	37524	74066	4.12
z100v6d5iw10-rand	90	3.82	180	13615	34970	107184	7.87	169	10421	27913	74115	7.11	315	14945	34053	78783	5.27
z100v6d5iw10-struc	90	3.87	179	12301	31200	76811	6.24	171	10007	27577	69550	6.95	315	13013	25707	59174	4.55
z100v6d5iw20-rand	90	4.13	159	12031	32766	87635	7.28	173	9124	23841	74758	8.19	279	11750	32369	70697	6.02
z100v6d5iw20-struc	90	4.13	157	14797	36370	86008	5.81	174	9159	26400	72395	7.90	252	13079	32041	68592	5.24
z200v3d5iw10-rand	190	2.54	60	6249	18879	80728	12.92	65	6693	20094	55385	8.28	108	9871	28903	89028	9.02
z200v3d5iw10-struc	190	2.54	60	6580	18580	80510	12.24	65	6970	21327	55890	8.02	110	9793	26300	32545	3.32
z200v3d5iw20-rand	190	2.52	55	7165	25307	115693	16.15	64	6725	20050	52133	7.75	106	9836	29992	101759	10.35
z200v3d5iw20-struc	190	2.52	58	7092	20705	103104	14.54	66	6759	20481	55381	8.19	111	9749	30013	102419	10.51
z200v6d5iw10-rand	190	4.11	37	4317	12756	54433	12.61	39	4477	13534	46237	10.33	65	6221	17154	50358	8.09
z200v6d5iw10-struc	190	4.07	36	5336	13530	63970	11.99	40	4227	13578	44402	10.50	65	6665	18867	66418	9.97
z200v6d5iw20-rand	190	4.09	40	6176	18306	98406	15.93	41	4624	13792	47131	10.19	65	6682	19004	75570	11.31
z200v6d5iw20-struc	190	4.08	40	7651	20374	90057	11.77	40	4476	13717	46893	10.48	59	6473	18902	70918	10.96
z400v3d5iw10-rand	390	2.47	16	4839	11902	85923	17.76	16	3114	9992	31047	9.97	23	4762	15851	66065	13.87
z400v3d5iw10-struc	390	2.48	15	4056	11754	86576	21.35	16	3471	9856	32861	9.47	22	4553	13050	45580	10.01
z400v3d5iw20-rand	390	2.48	14	4027	11717	82920	20.59	16	3273	9790	30412	9.29	22	4787	15901	58763	12.28
z400v3d5iw20-struc	390	2.49	15	3347	11180	84085	25.12	15	3443	9919	32798	9.53	24	4736	16437	78071	16.48
z400v6d5iw10-rand	390	4.11	9.91	2314	8199	61500	26.58	9.16	1772	6330	30602	17.27	13	2534	9987	69514	27.43
z400v6d5iw10-struc	390	4.11	9.61	2181	11059	100137	45.91	9.46	1892	6263	29451	15.57	13	2404	9978	81229	33.79
z400v6d5iw20-rand	390	3.87	10	2805	8521	65196	23.24	9.81	2066	6893	30029	14.53	14	2746	10406	81992	29.86
z400v6d5iw20-struc	390	3.88	10	2601	8167	66112	25.42	10	1981	6039	30947	15.62	14	2780	10159	83747	30.12

Table 6.4: Steps performed per second for the algorithms G+StS, GLS, and ILS on problem set `gen` with caching schemes *Naïve* (c=0), *Simple* (c=1), *Scores* (c=2), and *Improving* (c=3). For each algorithm, *sf* denotes the speedup factor gained by our new superior caching scheme *Improving* over the previously best simple caching scheme *Simple*. This speedup is higher for larger instances with many free variables (column N) and/or high domain sizes (column *Dom*). GLS$^+$ performs almost exactly as many steps per seconds as GLS, and we omit it in the table. G+StS and ILS used initialization MB$^*(10^5)$, GLS random initialization. The other default parameters used for the algorithms are $\langle \mathit{cf}, \mathit{np} \rangle = \langle 2, 40 \rangle$ for G+StS, $\langle N_\rho, \rho \rangle = \langle 200, 0.999 \rangle$ for GLS, and $\langle \mathit{acc}, \mathit{an}, \mathit{cf}, p, \mathit{pert}, \mathit{pfix}, \mathit{prel} \rangle = \langle \mathit{HYBRID}, 0.003, 5, 2, \mathit{POTS}, \mathit{true}, \mathit{false} \rangle$ for ILS. GLS with an infitinite smoothing interval N_ρ performed very similar to GLS with default parameters.

Chapter 7

Tuning SLS Algorithms for MPE

In this chapter, we introduce our experimental methodology and show how we tuned the parameters of the algorithms introduced in Chapter 5 in order to achieve high performance. We also demonstrate that tuning the parameters of the previously best-performing SLS algorithm GLS improves its performance by several orders of magnitude.

7.1 Experimental Methodology

In the remainder of this thesis, we will repeatedly face the problem to evaluate the performance of an algorithm A on a set of problem instances $\mathcal{S}$. For this purpose, we let A run on all problem instances in $\mathcal{S}$ for a given time[1] and keep track of the solutions it finds. In order to evaluate how good the solutions it finds are, we compare them to provably optimal solutions we obtained with the exact algorithms s-BBMB and Anytime MB (cf. Chapter 4) for most of the instances we study. However, for some instances, especially for randomly generated networks with high induced width, these algorithms were not able to find optimal solutions and proof their optimality.[2] We thus employ *quasi-optimal* solutions, which we define to be the best solutions we ever encountered in any run of any algorithm we tried. We use the term quasi-optimal solution in its most general meaning, that is, we refer with this term for each instance to the best solution we found with any algorithm.

[1]Unless explicitly stated otherwise, all algorithms are run for $t = 100$ seconds on compute servers each equipped with dual 2GHz Intel Xeon CPUs with 512KB cache and 4GB RAM running Linux version 2.4.20, build 28.9.

[2]For all instances in `bnrep`, we could find optimal solutions and proof their optimality. For 5 of the networks in `gen` (all of them with maximal induced width 20 and maximal domain size 6), this was not the case. Detailed results for the exact algorithms can be found in Chapter 9.

In particular, this includes provably optimal solutions.

After we ran algorithm A on problem set $\mathcal{S}$, we compute a number of measures, such as A's ratio of solved problem instances and its average approximation quality. Moreover, we employ qualified run-time distributions as well as distributions of solution quality. These concepts are defined in the following (see also [HS04]).

Since SLS algorithms are randomized, the runtime $RT_{A,q}$ an algorithm A needs to find a solution with quality q for a given problem instance is a random variable. We perform multiple runs of A in order to estimate $RT_{A,q}$ for all qualities q of interest, especially for q^*, the quasi-optimal solution quality.

In each of R runs of A, we record for each quality q of interest and run $r = 1, \ldots, R$, the time $T_{q,r}$ when the run first reached a quality greater or equal to q. An empirical *qualified run-time distribution (empirical QRTD) for quality* q gives for each time t the percentage of runs r with $T_{q,r} \leq t$; it yields an estimate of the probability that algorithm A finds a quality greater or equal to q in time less than or equal to t.[3] Application areas of empirical QRTDs include visualization of search stagnation and general characterization of algorithm behaviour. When plotting empirical QRTDs, the x-axis gives algorithm runtime and the y-axis the percentage of runs in which the quality of interest was achieved. In this thesis, we only employ empirical QRTDs for quasi-optimal solution qualities q^*. QRTDs for optimal solution quality are simply called run-time distributions (RTDs); for example, Figure 7.3 on page 74 shows empirical RTDs for different parameter settings of GLS.

Like the runtime an algorithm A needs to reach a given solution quality, the quality it achieves within a given time t is a random variable, and its probability density function can be approximated by the sample solution qualities reached in runs $1, \ldots, R$ of A within time t. The *mean solution quality* at time t, a standard tool in the analysis of anytime algorithms, is then just the mean of this sample. It is especially useful when comparing algorithms with very different performance and can also be applied for deterministic algorithms.[4] For hard problems, in practice, one might primarily be interested in which algorithm promises to find the best solution for a particular instance or class of instances given a certain time t. Mean solution quality at time t gives the solution quality we expect an algorithm to reach on average, and plotting it over time also shows whether an algorithm shows

[3]This estimate naturally improves with the number of runs performed, but doing many runs is expensive in terms of CPU time. Thus, with a given limit on CPU time for an experiment, we always face a trade-off between the number of runs per algorithm, the number of instances we run the algorithm on, and the time each run is allowed to last.

[4]For deterministic algorithms, of course only one run is performed and the mean solution quality at any time t is simply the quality this run achieved in time t.

strong performance in the beginning but improves more slowly over time than others.[5] In plots of mean solution quality, the x-axis represents runtime and the y-axis solution quality; Figure 7.1 on page 71 shows an example for this. Recall that we use -10000 as the log-probability for probability zero; when plotting average log-probabilities, these values are suppressed to prevent cluttered figures, and the plot for a particular algorithm A only starts at the time t for which all runs of A found an assignment with strictly positive probability. Whenever possible without introducing clutter into the plots, we provide the minimal and maximal solution qualities achieved in all the runs of an algorithm as well. These are plotted at every time step, at which the underlying sample distribution changes, i.e., whenever one of the algorithm's runs improves its best solution found so far. In order to prevent clutter, we plot minimal and maximal solution qualities only for the best- and worst-performing algorithms if at all.

While RTDs and plots of mean solution quality facilitate a detailed analysis of algorithm performance on single problem instances, in order to evaluate algorithms on sets of problem instances we employ a number of statistics per instance. These statistics include the ratio of successful runs, average approximation quality and average runtime. In order to ease reading, we generally provide summary tables of our experiments. For each experiment, the full results for each problem instance are given in Appendix B and are referenced in the respective summary table (see e.g. Table 7.1 on page 69 for an example). Each summary table contains the following performance measures for each algorithm.

Avg. quality is short for an algorithm's *approximation quality* averaged over all its runs on all instances. We define the approximation quality of an algorithm run as the ratio of the nonlogarithmic solution quality it reached and the quasi-optimal one. We always state approximation qualities in percent of the quasi-optimal quality, i.e., a value of 100 is optimal. (One can think of this measure as the percentage of solved instances, plus some additional score for instances for which solution qualities at least close to the quasi-optimal one are found.)

Avg. runtime gives the total time the algorithm needed to complete all the runs in an experiment divided by the total number of runs in which the quasi-optimal solution quality was found. A run terminates when it finds a quasi-optimal solution or after $t = 100$ CPU seconds.[6]

[5]Obviously, in practice, additional measures, such as the variance of achieved solution qualities at time t, might play an important role as well.

[6]We employ this measure instead of averaged or median runtime quantiles since the performance of the algorithms we compare varies heavily and for many instances not even the 5% quantiles exist

Successful runs gives the ratio of the number of runs which found a quasi-optimal solution and the total number of runs performed.

Instances solved gives the ratio of instances for which at least one of the algorithm's runs was successful, i.e. the ratio of instances for which the algorithm found a quasi-optimal solution in at least one of its runs.

# amongst best gives for each algorithm the number of instances for which it performed best among the algorithms compared. Performance of algorithms for a particular instance is compared by first evaluating the ratio of successful runs for this instance, in case of ties using the approximation quality averaged over the runs for this instance, and again in case of ties, by the average runtime on this instance. If all these characteristics are equal for several algorithms, they are all among the best performing algorithms for this instance.

7.2 Tuning G+StS

The G+StS algorithm as introduced in [KD99b] and detailed in Section 5.1 on page 37 has two free parameters, the cutoff factor *cf*, and the noise probability *np*. No default values for these parameters were mentioned in [KD99b], so we tuned them manually. Note that here and in the following, we already employ our new caching scheme *Improving* for tuning the parameters of previous algorithms in order to make the best use out of a restricted CPU time. After preliminary experiments, we discretized the possible parameter values to $cf \in \{1.5, 2, 5, 10, 100\}$ and $np \in \{5, 10, 20, 30, 40, 50\}$ (we measure the noise probability in percent), and for each combination of these parameter values we ran G+StS 25 times on all instances in the problem sets `bnrep` and `gen`. The overall best-performing parameter combination when judging by the percentage of successful runs and the average approximation quality was $\langle cf, np\rangle = \langle 2, 40\rangle$.

With this setting of the noise probability, 40% of the steps in G+StS are Stochastic Simulation steps. In this case, G+StS's sensibility to its cutoff factor is not very high, which we demonstrate for problem sets `bnrep` and `gen` in Tables 7.1 on the facing page and 7.2 on the next page, respectively. However, G+StS is still quite sensitive to its noise parameter even with a low cutoff factor of $cf = 2$. This can

for an algorithm because it cannot solve the instance. However, it is important to note that the measure as we currently employ it strongly depends on the chosen time bound $t = 100$ since we average runtimes over problems with vast differences in difficulty, some of which are solved in milliseconds and some of which are never solved. However, in combination with the percentage of successful runs, average runtime can give very valuable information.

Statistics	G+StS				
	cf = 1.5	cf = 2	cf = 5	cf = 10	cf = 100
Avg. quality	77.18	77.04	76.58	76.31	75.43
Avg. runtime	55.46	55.89	58.43	60.27	62.90
Successful runs	424/650	423/650	415/650	410/650	400/650
Instances solved	17/26	17/26	17/26	17/26	16/26
# amongst best	14	12	10	8	9

Table 7.1: Summary statistics for G+StS with initialization MB$^*(10^5)$, noise probability *np* $= 40$, and varying cutoff factor *cf* on problem set `bnrep`. All algorithms are run 25 times for 100 CPU seconds each. Summary of Table B.1 on page 141.

Statistics	G+StS				
	cf = 1.5	cf = 2	cf = 5	cf = 10	cf = 100
Avg. quality	36.49	37.74	37.27	35.92	34.75
Avg. runtime	326.34	303.97	295.09	324.82	358.41
Successful runs	144/600	152/600	155/600	144/600	133/600
Instances solved	8/24	9/24	8/24	8/24	8/24
# amongst best	13	4	3	2	4

Table 7.2: Summary statistics for G+StS with initialization MB$^*(10^5)$, noise probability *np* $= 40$, and varying cutoff factor *cf* on problem set `gen`. All algorithms are run 25 times for 100 CPU seconds each. Summary of Table B.2 on page 142.

Statistics	G+StS					
	np = 5	np = 10	np = 20	np = 30	np = 40	np = 50
Avg. quality	76.33	76.83	77.31	77.11	77.00	76.72
Avg. runtime	60.08	58.57	55.50	55.73	56.08	56.04
Successful runs	408/650	413/650	423/650	422/650	423/650	422/650
Instances solved	17/26	17/26	17/26	17/26	17/26	17/26
# amongst best	9	8	14	11	9	7

Table 7.3: Summary statistics for G+StS with initialization MB$^*(10^5)$, cutoff factor *cf* $= 2$, and varying noise probability *np* on problem set `bnrep`. All algorithms are run 25 times for 100 CPU seconds each. Summary of Table B.3 on page 143.

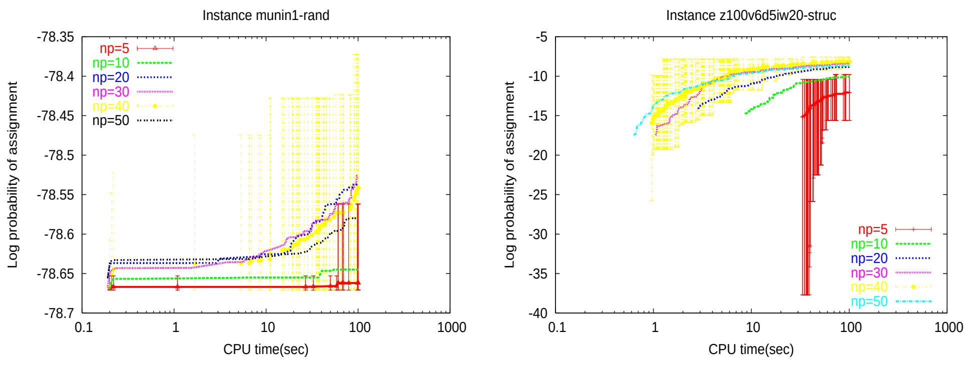
	G+StS					
Statistics	np = 5	np = 10	np = 20	np = 30	np = 40	np = 50
Avg. quality	27.75	32.26	34.45	37.20	37.99	35.85
Avg. runtime	449.56	370.42	343.87	313.65	289.55	317.94
Successful runs	111/600	131/600	138/600	148/600	159/600	147/600
Instances solved	7/24	7/24	9/24	7/24	9/24	6/24
# amongst best	1	2	6	4	8	4

Table 7.4: Summary statistics for G+StS with initialization MB$^*(10^5)$, cutoff factor *cf* $= 2$, and varying noise probability *np* on problem set `gen`. All algorithms are run 25 times for 100 CPU seconds each. Summary of Table B.4 on page 144.

only be guessed from the performance on problem set `bnrep` (see Table 7.3 on the preceding page), but can be seen clearly for problem set `gen` (see Table 7.4).[7] The picture becomes much clearer when considering plots of mean solution quality, such as the ones in Figure 7.1 on the facing page. From these plots, the stronger performance of relatively high noise values is very obvious for both structured and randomly generated networks. For the structured instance `munin1-rand`, initialization MB$^*(10^5)$ already finds a very high-quality initial solution. For too low noise probabilities, G+StS cannot improve much on this initial solution, which causes a very small variance in solution quality for too low noise values, such as *np* $= 5$. For higher noise probabilities, improvements on the initial solution are found quickly in this example, and the variance in solution quality grows quickly as well. For the randomly generated instance `z100v6d5iw20-struc`, initialization MB$^*(10^5)$ does not yield an initial solution with positive probability. Thus, for too low settings of the noise value, up to 30 seconds elapse before G+StS even finds instantiations with positive probability in all its runs. The variance in solution quality decreases both for low noise probabilities, such as *np* $= 5$, and higher ones, such as *np* $= 40$. For *np* $= 40$, after 100 CPU seconds, G+StS has found the optimal solution quality in 2 of its 25 runs.

As can be seen in the complete results in Table B.3 on page 143 and Table B.4 on page 144, the optimal setting for the noise parameter is not always 40%; however, for only one instance, a value smaller than 20% yields clearly better results than higher noise probabilities. The optimal cutoff factor also varies from instance to instance. In the case of randomly generated instances, we observe a

[7]We suppose the unclear picture for problem set `bnrep` is due to the fact that G+StS's Mini-Buckets initialization already yields solutions of high quality for structured instances, and that for many of these instances the relatively weak local search in G+StS cannot improve much on these initial solutions even with an optimal parameter setting.



(a) `munin1-rand` (b) `z100v6d5iw20-struc`

Figure 7.1: Plots of mean solution quality for 25 runs of G+StS with initialization MB$^*(10^5)$, cutoff factor *cf* $= 2$, and varying noise probability *np* on the structured instance `munin1-rand` (a) and the randomly generated instance `z100v6d5iw20-struc` (b). The estimation of mean solution quality is based on 25 runs of the algorithms per parameter value. In both plots, for the worst-performing parameter value *np* $= 5$ and the best-performing parameter value *np* $= 40$, we also provide the minimal and maximal solution qualities achieved in any of their runs.

pattern in the full results in Table B.2 on page 142: for small instances, large cutoff factors tend to work well, whereas for large instances the smallest considered cutoff factor *cf* $= 1.5$ performs best. For the structured instances in Table B.1 on page 141, the strong initialization skews the picture and we cannot infer any regularities other than that in the few cases for which there are significant differences between the parameter settings, the smallest cutoff factor *cf* $= 1.5$ always works best.

7.3 Tuning GLS

The GLS algorithm, as originally introduced for MPE by Park [Par02] and detailed in Section 5.2 on page 38, has two parameters, the smoothing factor ρ and the smoothing interval N_ρ. Recall from our discussion in Section 5.2 that in this original version for MPE, no weighting parameter for the penalties is necessary, since the evaluation function $\prod_{\phi\in\Phi}\phi[\mathbf{V}=\mathbf{v}] - w \times \sum_{\phi\in\Phi}\lambda_\phi[\mathbf{V}=\mathbf{v}]$ is clearly dominated by the penalties; also recall that, following Park's [Par02] implementation, we use only the summed penalties as an evaluation function, implicitly setting w to

Statistics	GLS					
	ρ = 0.7	ρ = 0.8	ρ = 0.9	ρ = 0.99	ρ = 0.999	ρ = 1.00
Avg. quality	40.70	44.09	48.63	63.96	75.52	75.45
Avg. runtime	223.71	200.13	137.50	85.46	59.32	52.06
Successful runs	201/650	219/650	277/650	353/650	429/650	451/650
Instances solved	9/26	10/26	12/26	15/26	19/26	20/26
# amongst best	1	5	1	0	3	19

Table 7.5: Summary statistics for GLS with smoothing interval $N_\rho = 200$ and varying smoothing factor ρ on problem set `bnrep`. All algorithms are run 25 times for 100 CPU seconds each. Summary of Table B.5 on page 145.

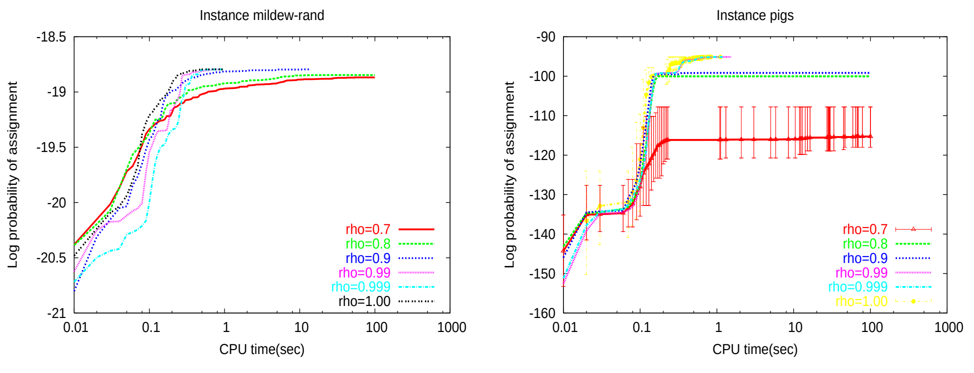
Statistics	GLS					
	ρ = 0.7	ρ = 0.8	ρ = 0.9	ρ = 0.99	ρ = 0.999	ρ = 1.00
Avg. quality	36.90	42.20	49.81	78.74	87.20	83.53
Avg. runtime	355.65	273.80	197.71	66.93	60.42	69.41
Successful runs	134/600	164/600	203/600	383/600	400/600	372/600
Instances solved	7/24	8/24	10/24	17/24	20/24	21/24
# amongst best	0	0	1	6	12	5

Table 7.6: Summary statistics for GLS with smoothing interval $N_\rho = 200$ and varying smoothing factor ρ on problem set `gen`. All algorithms are run 25 times for 100 CPU seconds each. Summary of Table B.6 on page 146.

∞.

Based on experience with other Dynamic Local Search algorithms such as SAPS [HTH02], we expected the optimal parameter setting of the smoothing parameter ρ and the smoothing interval N_ρ to be tightly coupled. Modifying the smoothing interval N_ρ would lead to two separate effects, namely computational savings due to the less frequent smoothing (cf. Chapter 6) and changes in the trajectory due to the penalties being smoothed less. On the other hand, modifying the smoothing parameter ρ would only lead to the latter effect of the trajectory being changed. In order to study only one effect at a time, we decided to focus on the smoothing parameter ρ first, assuming a fixed smoothing interval of $N_\rho = 200$.

In Table 7.5 and Table 7.6, we give the results of tuning GLS's smoothing parameter ρ with $N_\rho = 200$ for problem sets `bnrep` and `gen`, respectively. For the structured instances in `bnrep`, parameter value $\rho = 1.00$ outperforms all other settings, which means that performing any smoothing at all is detrimental. Figure 7.2 on the facing page presents two plots of mean solution quality that are representative for the performance of GLS with different values of ρ for the in-

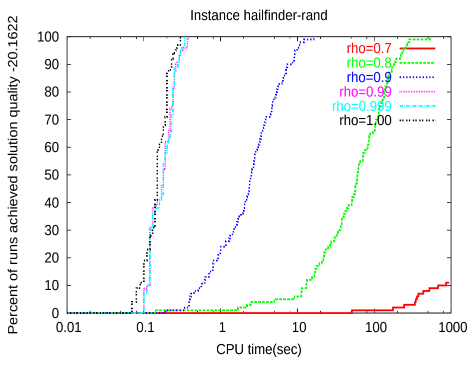


(a) `mildew-rand` (b) `pigs`

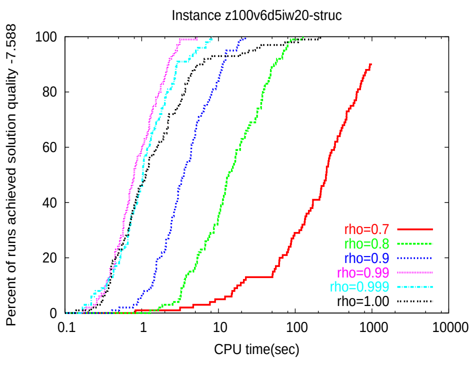
Figure 7.2: Plots of mean solution quality for GLS with smoothing interval $N_\rho = 200$ and varying values of the smoothing parameter ρ for the structured instances `mildew-rand` (a) and `pigs` (b). For instance `mildew-rand`, the plots for parameter values $\rho \in \{0.9, 0.99, 0.999, 1.00\}$ end early when the optimal solution quality (-18.795) has already been reached in all 25 runs. For instance `pigs` and parameter values $\rho = 0.7$ and $\rho = 1.00$, we also provide the minimal and maximal solution qualities achieved in any of their 25 runs. For $\rho = 0.7$, we see a clear indication of search stagnation since with this parameter setting, after 0.2 CPU seconds, the best overall solution found in the 25 runs of GLS does not improve anymore until the algorithm is terminated after 100 CPU seconds. For this instance, we also observe a rather odd development of mean solution quality over time with some plateaus that need to be overcome; the plots for $\rho = 0.999$ and $\rho = 1.00$ again end early when all 25 runs have already found the optimal solution quality (-95.125).

stances in `bnrep`. In these plots, the dominance of values for ρ close to 1.00 is very obvious; lower values of ρ, such as the default setting $\rho = 0.8$ from [Par02], lead to a performance that is orders of magnitude weaker than with $\rho = 1.00$ at best or to a complete inability to solve the instances at worst.

The empirical RTD for network `hailfinder-rand` in Figure 7.3(a) on the next page is representative for many instances in problem set `bnrep` and once more demonstrates the superiority of values of ρ close to 1.00 for structured instances. The empirical RTD for instance `z100v6d5iw20-struc` in Figure 7.3(b) on the following page is similarly representative for problem set `gen`: it shows generally much superior behaviour of values for ρ close to 1.00, but also suggests that completely omitting the smoothing may lead to search stagnation. As opposed to



(a) `hailfinder-rand`



(b) `z100v6d5iw20-struc`

Figure 7.3: Empirical RTDs for GLS with smoothing interval $N_\rho = 200$ and varying values of the smoothing parameter ρ for instances `hailfinder-rand` (a) and `z100v6d5iw20-struc` (b); for both instances, the optimality of the found solutions can be proven. The empirical RTDs for each parameter value are based on 100 runs of 1000 CPU seconds.

what we experienced with problem set `bnrep`, for a few instances from `gen`, we indeed observe search stagnation of the otherwise best-performing parameter value $\rho = 1.00$; this can, for example, be seen in Figure 7.4 on the next page. For parameter value $\rho = 0.999$, we never found evidence for search stagnation in our experiments. For this reason, we use this value as a default despite its slightly inferior performance for the structured instances in problem set `bnrep`.

As can be seen in the full results in Table B.5 on page 145 and Table B.6 on page 146, there are remarkably low differences in the best-performing parameter value across the instances within each of the problem sets. For problem set `bnrep`, GLS with $\rho = 1.00$ always finds the quasi-optimal solution if any of the other parameterizations does, and for all other instances achieves the best average approximation quality. For problem set `gen`, almost the same is true for parameter value $\rho = 0.999$, which is only outperformed by $\rho = 0.99$ on two instances, and by $\rho = 1.00$ on four other instances.

So far, we only tuned the smoothing parameter ρ assuming a fixed smoothing interval of $N_\rho = 200$. Having determined value $\rho = 0.999$ to be optimal for this smoothing interval, we now tune parameter N_ρ given the fixed smoothing parameter $\rho = 0.999$. For problem set `bnrep`, we have already seen that it is detrimental to perform any smoothing at all. High smoothing intervals correspond to little smoothing to be carried out, and Table 7.7 on page 76 demonstrates that, matching our intuition, an infinite smoothing interval performs best on problem set

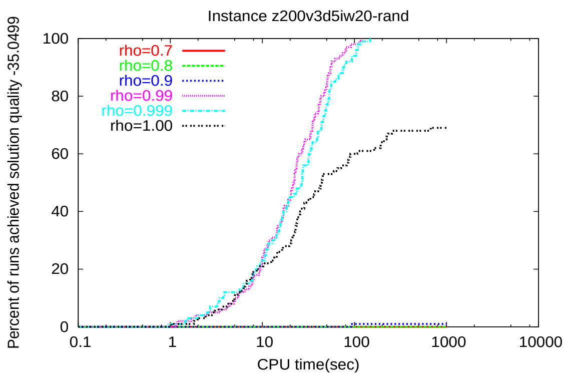
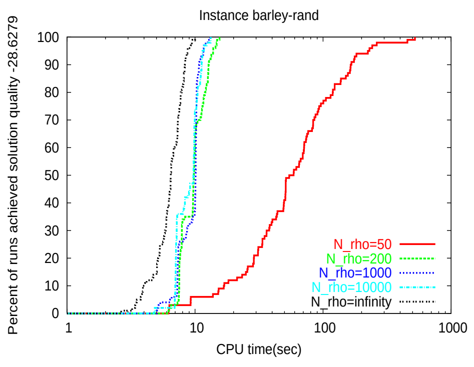


Figure 7.4: Empirical RTDs for GLS with smoothing interval $N_\rho = 200$ and varying values of the smoothing parameter ρ for instance `z200v3d5iw20-rand` and optimal solution quality. Clear search stagnation for parameter value $\rho = 1.00$ can be observed. For parameter values $\rho \in \{0.7, 0.8\}$, GLS did not find a solution in any run, and for $\rho = 0.9$, one run succeeded after 90 seconds. The empirical RTDs for each parameter value are based on 100 runs of 1000 CPU seconds.

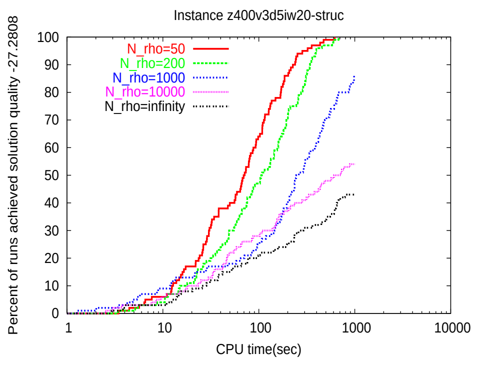
`bnrep` (for all measures except average approximation quality). However, for the instances in problem set `gen`, smoothing plays an important role. As we demonstrate in Table 7.7 on the next page, lower smoothing intervals yield significantly better performance in this case. Judging by average approximation quality and average runtime, $N_\rho = 200$ performs best, whereas a greater number of runs are successful for $N_\rho = 50$, and $N_\rho = 1000$ solves more instances and is amongst the best performing algorithms most often. In Figure 7.5 on the following page, we show an example of empirical QRTDs of GLS with a varying smoothing interval for a structured instance and a randomly generated one. Similarly to what we observed when varying the smoothing parameter, GLS performs poorly for structured instances when too much smoothing is performed whereas it stagnates without smoothing on randomly generated instances. Since none of N_ρ's possible values clearly performs better than its default $N_\rho = 200$ from [Par02], we employ this value here as well, making our default parameter configuration $\langle N_\rho, \rho\rangle = \langle 200, 0.999\rangle$.

We strongly expect that there exist other combinations of N_ρ and ρ which yield comparable or even better performance. This is because by our simple tuning of one parameter at a time, we have merely found a local optimum in the space of possible parameter configurations.[8] Finding an equally strong parameter setting

[8]In Appendix A, we introduce a parameter tuning scheme which automates the process we have carried out in tuning GLS and explores other configurations after a local optimum in configuration



(a) `barley-rand`



(b) `z400v3d5iw20-struc`

Figure 7.5: Empirical RTDs for GLS with smoothing parameter $\rho = 0.999$ and varying values of the smoothing interval N_ρ for instances `barley-rand` (a) and `z400v3d5iw20-struc` (b); for both instances, the optimality of the found solutions can be proven. The empirical RTDs for each parameter value are based on 100 runs of 1000 CPU seconds.

Statistics	GLS with $\rho = 0.999$				
	N_ρ = 50	N_ρ = 200	N_ρ = 1000	N_ρ = 10000	N_ρ = ∞
Avg. quality	70.31	75.28	76.02	76.47	75.43
Avg. runtime	79.35	59.95	55.59	53.99	50.95
Successful runs	373/650	425/650	433/650	444/650	459/650
Instances solved	17/26	18/26	19/26	20/26	20/26
# amongst best	1	2	8	8	15
# Instances > 0	26/26	26/26	26/26	26/26	26/26

Table 7.7: Summary statistics for GLS with smoothing parameter $\rho = 0.999$ and varying smoothing interval N_ρ on problem set `bnrep`. All algorithms are run 25 times for 100 CPU seconds each. Summary of Table B.7 on page 147.

Statistics	GLS with $\rho = 0.999$				
	N_ρ = 50	N_ρ = 200	N_ρ = 1000	N_ρ = 10000	N_ρ = ∞
Avg. quality	82.35	87.44	87.35	85.71	84.11
Avg. runtime	55.97	58.97	63.14	69.13	72.68
Successful runs	412/600	406/600	395/600	375/600	366/600
Instances solved	17/24	20/24	22/24	22/24	18/24
# amongst best	6	5	8	4	1

Table 7.8: Summary statistics for GLS with smoothing parameter $\rho = 0.999$ and varying smoothing interval N_ρ on problem set `gen`. All algorithms are run 25 times for 100 CPU seconds each. Summary of Table B.8 on page 148.

with a higher smoothing interval would be very important if the complexity of smoothing clearly dominated the total search cost and we could significantly reduce the complexity of GLS without impairing its potential.[9] However, preliminary experiments (not reported here) suggest that even with an infinite smoothing interval the number of search steps executed per time unit does not significantly grow. Thus, future work may be able to improve on our default parameter setting $\langle N_\rho, \rho\rangle = \langle 200, 0.999\rangle$, but we do not expect this to yield major savings due to a reduced computational complexity.

7.4 Tuning GLS$^+$

Our new variant of Guided Local Search, GLS$^+$ (detailed in Section 5.2 on page 38), has two additional parameters on top of the smoothing factor ρ and the smoothing interval N_ρ, namely the penalty weighting parameter w which was implicitly set to ∞ in GLS, and the initialization. As detailed in Section 5.2 on page 38, GLS$^+$ employs the evaluation function $\sum_{\phi\in\Phi}\log(\phi[\mathbf{V} = \mathbf{v}]) - w \times \lambda_\phi[\mathbf{V} = \mathbf{v}]$. In GLS for weighted Max-SAT with integer weights, w is, for example, set to 1 by default [MT00], but in our case, the choice is not that simple. Recall that frequently some potentials ϕ have zero-probability entries $\phi[\mathbf{V} = \mathbf{v}] = 0$ for a variable instantiation $\mathbf{v}$, and that our evaluation function (the logarithmic objective function) uses the value -10000 for each such zero-probability.

space has been reached. Applied to GLS, this parameter tuning scheme found a configuration that performs better than $\langle N_\rho, \rho\rangle = \langle 200, 0.999\rangle$ if only one run per instance is used for the evaluation of each parameter configuration. See Appendix A for details.

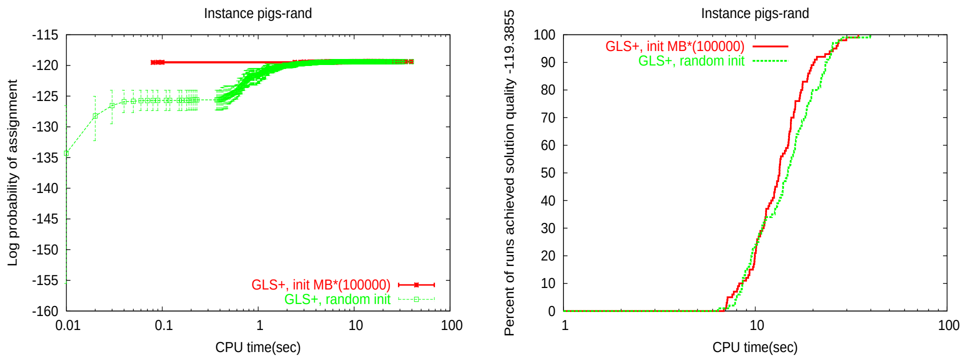
[9]A similar intuition has led to the development of SAPS [HTH02], a state-of-the-art Dynamic Local Search algorithm for SAT.

Especially in the beginning of a search, many zero-probabilities are encountered. Due to the large differences between log-probabilities for various assignments resulting from these zero-probabilities, we need to use a rather large value for the penalty weighting factor w. Otherwise, the log-probabilities would completely dominate the evaluation function, especially in the beginning of the search. This would lead to the search spending thousands of search steps in each encountered local optimum, incrementing the respective penalties only by 1 each step.

In the limited scope of this thesis, we simply deal with this problem by using a very high weighting factor $w = 10000$ which makes even small penalties comparable to the log-probabilities for zero-probability entries. Effectively, this enables a mixture of penalties and log-probabilities guiding the search in the beginning when there are still plenty zero-probabilities. Later, when not as many zero-probabilities are encountered anymore, the penalties dominate the search and the log-probabilities are merely used for tie-breaking. As we will see in the experimental evaluation in Chapter 8, this simple integration of the log-probabilities into the evaluation function already yields considerable performance improvements over GLS. However, we will also see that in many cases, only the initial performance increases, while in later stages of the search GLS and GLS$^+$ behave almost identically.

We do not expect our fixed setting of $w = 10000$ to be optimal. On the contrary, we assume that especially for later stages of the search a considerably stronger performance can be gained using an adaptive setting of w that yields a balanced mixture of log-probabilities and penalties in the evaluation function during the whole course of the search. Due to the limited scope of this thesis, we defer work on such an adaptive version of GLS$^+$ to future work.

For the initialization, we considered the two possibilities RANDOM and MB$^*(10^5)$. Our experiments for these variants suggest that with initialization MB$^*(10^5)$, GLS$^+$ gets a significant head start over GLS$^+$ with random initialization. However, as illustrated in Figure 7.6 on the next page, this initial advantage does not necessarily lead to superior performance for longer runs. Nevertheless, there are a few structured instances, such as the `munin2`, `munin3`, and `mildew` networks, for which the MB$^*(10^5)$ initialization quickly finds optimal solution quality whereas GLS$^+$ with a random initialization shows rather poor performance. For this reason, we use initialization MB$^*(10^5)$ as a default for GLS$^+$. In our experimental evaluation in Chapter 8, we will revisit the issue of different initializations and demonstrate that initialization MB$^*(10^5)$ indeed yields much better results for most instances.



(a) Mean solution qualities (b) Empirical RTDs

Figure 7.6: Performance of GLS$^+$ with smoothing interval $N_\rho = 200$, smoothing factor $\rho = 0.999$, penalty weighting factor $w = 10000$, and varying initialization for instance `pigs-rand`. The plot of mean solution quality in (a) suggests vastly superior behaviour of GLS$^+$ with initialization MB$^*(10^5)$, but the RTD in (b) shows that this superiority diminishes for longer runtimes and that optimal solutions are found equally fast when using a random initialization. The estimation is based on 100 runs for each initialization.

7.5 Tuning ILS

In Section 5.3 on page 42, we introduced an ILS algorithm for MPE with a few degrees of freedom left to fill in. The procedural parameters were as follows:

- *GenerateInitialSolution* $\in \{\text{RANDOM}, \text{MB}^*(10^5)\}$;
- *AcceptanceCriterion* $\in \{\text{RW}, \text{BETTER}, \text{BE/RW}, \textit{LSMC}\}$;
- Pertubation type: $pt \in \{\text{VARS}, \text{POTS}\}$;
- Additional local search at the end of the pertubation phase, keeping perturbed variables fix: *pfix* $\in \{\textit{true}, \textit{false}\}$; and
- Relative pertubation size: *prel* $\in \{\textit{true}, \textit{false}\}$.

The algorithm's numerical parameters are the cutoff factor *cf*, the pertubation strength p, as well as two conditional parameters that depend on the used acceptance criterion. In the case of BE/RW, the acceptance probability *ap* needs to be set, and in the case of LSMC, the temperature T is an additional parameter. In order to determine sets of possible discrete values for these numerical parameters, we carried out initial experiments. The discrete values we chose were:

- Cutoff factor *cf* $\in \{1.5, 2, 5, 10, 100\}$;
- Pertubation strength $p \in \{1, 2, 3, 4\}$;
- Acceptance probability *ap* $\in \{0.003, 0.01, 0.03\}$; and
- LSMC temperature $T \in \{0.01, 0.03, 0.1\}$.

Overall, there are 2 different initializations, $2 + 2 \times 3 = 8$ parameterized acceptance criteria, 5 possible values for the cutoff factor, and 2×2 different types of pertubations with 4×2 possible pertubation strengths. Thus, there are $2 \times 8 \times 5 \times 4 \times 8 = 2560$ possible parameter configurations. Evaluating a single parameter configuration on problem sets `bnrep` and `gen` for a single run of $t = 100$ CPU seconds takes 50×100 CPU seconds, and thus evaluating all possible parameter configurations just for a single run on each instance would take almost 5 CPU months.

In order to speed up this process, we developed an automated parameter tuning procedure that performs an Iterated Local Search in the space of possible parameter configurations. We call this procedure ParamILS and detail it in Appendix A. ParamILS evaluates each parameter configuration for ILS by performing a single run of ILS with that parameter configuration on all instances of `bnrep` and `gen`;[10] it uses a combination of the percentage of solved instances and the average approximation quality as an objective function to estimate the quality of a parameter configuration. ParamILS's major advantage over the brute-force approach to try out all possible parameter configurations is that the search is quickly driven towards promising parameter configurations such that only a small fraction of parameter configurations needs to be evaluated. This fraction hopefully contains the overall best-performing parameter configuration, but at the very least is biased towards well-performing parameter configurations.

We chose a very simple initial configuration for ParamILS with acceptance criterion BETTER, initialization RANDOM, pertubation strength $p = 4$, pertubation VARS, and *pfix* as well as *prel* set to *false*. What we expected from preliminary experiments and previous results from SLS algorithms for MPE [KD99b] was that very quickly the initialization would be changed to MB$^*(10^5)$, and that

[10]ParamILS can also be applied to the problem sets separately, or even on an instance-by-instance base. The smaller the problem sets are for which we optimize parameters, the closer we get to peak performance on the individual instances, which is considerably higher than the performance achieved with the overall best-performing set of parameters.

the acceptance criterion would be changed to either HYBRID or LSMC. Based on intuition from preliminary experiments, we also expected pertubation POTS and $\textit{pfix} = \textit{true}$ to be chosen. What we still could not anticipate from our initial experiments was whether acceptance criterion HYBRID or LSMC would turn out the best, and which values should be used for the numerical parameters.

Table 7.9 on the next page provides a trace documenting the automated parameter tuning process for the first 12 iterations. After 7 iterations, ParamILS found a parameter configuration that remained unbeaten for the first 25 iterations after which we terminated the procedure.[11] After iteration 12, no locally optimal parameter configuration was found anymore that had not been encountered before. This suggests that either the search in configuration space stagnates or that the total number of local optima in the configuration space is rather small.

ParamILS already finds a very good local optimum in the first iteration, and after 6 more iterations escapes to the best local optimum found. This locally optimal parameter configuration matches our expectations, employing initialization MB$^*(10^5)$, acceptance criterion HYBRID, pertubation POTS, and $\textit{pfix} = \textit{true}$. The best-performing combination $\langle \textit{cf}, p, \textit{prel}, \textit{ap}\rangle = \langle 5, 2, \textit{false}, 0.003\rangle$ of the other parameters did not greatly surprise us, but we had never looked at this configuration before, which strengthens our intuition that local search in the configuration space can greatly ease and improve the parameter tuning process. For completeness, we present the entire final ILS algorithm with optimized procedural parameters in Algorithm 7.1 on page 83 and page 84.

In order to evaluate ParamILS, we also tested its performance for tuning the parameters of G+StS and GLS. It performed very well for these tasks and we report the results in Appendix A.

[11]The 25 iterations we executed ParamILS for are not much for an ILS algorithm. However, we could not afford a longer run due to limited computational resources. For the 25 runs we performed, ParamILS needed approximately 1 CPU week. In contrast to the 5 CPU months the brute force approach would have taken, this was feasible and we conjecture to actually have found the globally optimal configuration in parameter space. We hope to considerably improve the performance of ParamILS in the future by a faster way of ruling out inferior parameter configurations.

Step	Iteration	Parameters									Solved	Avg. Quality
		T	acc	an	b	cf	p	pert	pfix	prel		
1	**1**	**-**	**BETTER**	**-**	**RANDOM**	**10**	**4**	**VARS**	**FALSE**	**FALSE**	**32.00**	**39.88**
2	**1**	**-**	**BETTER**	**-**	**MB**	**10**	**4**	**VARS**	**FALSE**	**FALSE**	**48.00**	**58.50**
3	**1**	**-**	**BETTER**	**-**	**MB**	**10**	**4**	**VARS**	**TRUE**	**FALSE**	**52.00**	**68.91**
4	**1**	**-**	**BETTER**	**-**	**MB**	**10**	**3**	**VARS**	**TRUE**	**FALSE**	**54.00**	**70.69**
5	**1**	**-**	**HYBRID**	**0.01**	**MB**	**10**	**3**	**VARS**	**TRUE**	**FALSE**	**56.00**	**71.52**
6	**1**	**-**	**HYBRID**	**0.01**	**MB**	**10**	**1**	**VARS**	**TRUE**	**FALSE**	**66.00**	**75.26**
7	2	-	HYBRID	0.01	MB	100	1	VARS	TRUE	FALSE	60.00	71.73
8	2 - same	-	HYBRID	0.01	MB	10	1	VARS	TRUE	FALSE	66.00	75.26
9	3	-	HYBRID	0.01	MB	10	4	VARS	TRUE	FALSE	50.00	68.03
10	3 - same	-	HYBRID	0.01	MB	10	1	VARS	TRUE	FALSE	66.00	75.26
11	4	-	RW	-	MB	2	4	VARS	TRUE	FALSE	38.00	48.39
12	4	0.01	LSMC	-	MB	2	4	VARS	TRUE	FALSE	52.00	68.95
13	4	0.1	LSMC	-	MB	2	4	VARS	TRUE	FALSE	56.00	71.32
14	4 - rej	0.1	LSMC	-	MB	2	3	VARS	TRUE	FALSE	58.00	73.66
15	5	-	HYBRID	0.01	MB	5	2	VARS	TRUE	TRUE	52.00	66.00
16	5	-	HYBRID	0.01	MB	5	2	VARS	TRUE	FALSE	62.00	72.37
17	5	-	HYBRID	0.01	MB	5	1	VARS	TRUE	FALSE	66.00	73.77
18	5 - same	-	HYBRID	0.01	MB	10	1	VARS	TRUE	FALSE	66.00	75.26
19	6	-	RW	-	MB	10	4	VARS	TRUE	FALSE	38.00	50.15
20	6	0.01	LSMC	-	MB	10	4	VARS	TRUE	FALSE	52.00	68.94
21	6 - rej	0.1	LSMC	-	MB	10	4	VARS	TRUE	FALSE	56.00	74.71
22	7	-	HYBRID	0.03	MB	10	2	VARS	TRUE	TRUE	50.00	62.53
23	7	-	HYBRID	0.03	MB	10	2	VARS	TRUE	FALSE	58.00	69.85
24	7	-	HYBRID	0.003	MB	10	2	VARS	TRUE	FALSE	64.00	75.10
25	**7**	**-**	**HYBRID**	**0.003**	**MB**	**10**	**2**	**POTS**	**TRUE**	**FALSE**	**64.00**	**77.72**
26	**7 - acc**	**-**	**HYBRID**	**0.003**	**MB**	**5**	**2**	**POTS**	**TRUE**	**FALSE**	**68.00**	**78.93**
27	8	-	BETTER	-	MB	1.5	2	POTS	TRUE	TRUE	44.00	60.46
28	8	-	BETTER	-	MB	1.5	1	POTS	TRUE	TRUE	50.00	64.59
29	8	-	BETTER	-	MB	5	1	POTS	TRUE	TRUE	50.00	65.62
30	8	-	HYBRID	0.003	MB	5	1	POTS	TRUE	TRUE	66.00	75.20
31	8	-	HYBRID	0.003	MB	5	1	POTS	TRUE	FALSE	68.00	78.20
32	8 - same	-	HYBRID	0.003	MB	5	2	POTS	TRUE	FALSE	68.00	78.93
33	9 - same	-	HYBRID	0.003	MB	5	2	POTS	TRUE	FALSE	68.00	78.93
34	10	-	BETTER	-	MB	5	1	POTS	FALSE	FALSE	46.00	58.45
35	10	-	HYBRID	0.003	MB	5	1	POTS	FALSE	FALSE	60.00	74.71
36	10	-	HYBRID	0.003	MB	5	1	POTS	TRUE	FALSE	68.00	78.20
37	10 - same	-	HYBRID	0.003	MB	5	2	POTS	TRUE	FALSE	68.00	78.93
38	11	-	HYBRID	0.01	MB	1.5	1	POTS	TRUE	FALSE	50.00	67.15
39	11	-	HYBRID	0.01	MB	5	1	POTS	TRUE	FALSE	60.00	72.17
40	11	-	HYBRID	0.003	MB	5	1	POTS	TRUE	FALSE	68.00	78.20
41	11 - same	-	HYBRID	0.003	MB	5	2	POTS	TRUE	FALSE	68.00	78.93
42	12	-	RW	-	MB	5	2	VARS	FALSE	FALSE	44.00	59.87
43	12 - rej	-	RW	-	MB	5	2	VARS	FALSE	TRUE	48.00	60.92

Table 7.9: Trace of ParamILS for tuning the parameters of ILS. Column “Step” gives the running number of search steps, i.e. flips of parameter values performed so far; “Iteration” gives the running number of iterations, and in the case of local optima whether the optimum is accepted (acc), rejected (rej) or is the same as the previous one (same); “Parameters” gives the current instantiation of ILS’s parameters; “Solved” gives the percentage of instances of problem sets `bnrep` and `gen` that are solved by ILS with the current parameter instantiation in single runs of 100 CPU seconds each; and “Avg. Quality” gives the average approximation quality achieved on all instances. The rows of search steps, in which the best performance so far is achieved, are printed in bold face.

Algorithm 7.1: Iterated Local Search (ILS) for MPE (to be continued)
Our default parameters are: $\langle cf, p, ap\rangle = \langle 5, 2, 0.003\rangle$. $g(\mathbf{v}|V_i = v_i)$ abbreviates $\sum_{\phi\in\Phi}\log(\phi[\mathbf{V} = \mathbf{v}|V_i = v_i])$; and $g(\mathbf{v})$ abbreviates $\sum_{\phi\in\Phi}\log(\phi[\mathbf{V} = \mathbf{v})$.

```
Input: Bayesian network $\mathcal{B} = \langle \mathbf{V}, \mathbf{D}, \mathcal{G}, \Phi\rangle$, evidence $\mathbf{E} = \mathbf{e}$, time bound $t$, cutoff factor
       $cf$, pertubation strength $p$, acceptance probability $ap$
Output: Variable assignment $\mathbf{v}$ with highest probability $\prod_{\phi\in\Phi}\phi[\mathbf{V} = \mathbf{v}]$ found in time $t$
while $runtime < t$ do
    $opt \leftarrow -\infty$
    $\mathbf{v_0} \leftarrow \mathrm{MB}^*(10^5)$
    $\mathbf{v}^* \leftarrow LocalSearch(\mathbf{v_0}, \emptyset)$
    $iteration \leftarrow 0$
    repeat
        $iteration \leftarrow iteration + 1$
        $\mathbf{v} \leftarrow Pertubation(\mathbf{v}^*)$
        $\mathbf{v} \leftarrow LocalSearch(\mathbf{v}, \emptyset)$
        $\mathbf{v}^* \leftarrow AcceptanceCriterion(\mathbf{v}^*, \mathbf{v})$
        if $g(\mathbf{v}^*) > opt$ then
            $opt \leftarrow g(\mathbf{v}^*)$
            $it_{opt} \leftarrow iteration$
    until $iteration > cf \times it_{opt}$ or $runtime \geq t$

Function $AcceptanceCriterion(\mathbf{v}^*, \mathbf{v})$
begin
    if $g(\mathbf{v}^*) > g(\mathbf{v})$ then return $\mathbf{v}^*$
    Draw $x$ from uniform distribution $u(0, 1)$
    if $x <$ ap then
        return $\mathbf{v}^*$
    else
        return $\mathbf{v}$
end
```

Algorithm 7.1: Iterated Local Search (ILS) for MPE (continued)
Our default parameters are: $\langle cf, p, ap \rangle = \langle 5, 2, 0.003 \rangle$. $g(\mathbf{v}|V_i = v_i)$ abbreviates $\sum_{\phi \in \Phi} \log(\phi[\mathbf{V} = \mathbf{v}|V_i = v_i])$; and $g(\mathbf{v})$ abbreviates $\sum_{\phi \in \Phi} \log(\phi[\mathbf{V} = \mathbf{v})$.

```
**Function** *Pertubation*$(\mathbf{v})$
**begin**
    $\mathcal{V}_{pert} \leftarrow \emptyset$
    **while** $|\mathcal{V}_{pert}| < p$ **do**
        Pick random potential $\phi \in \Phi$
        **foreach** $V_i \in \mathcal{V}_\phi \setminus (\mathbf{E} \cup \mathcal{V}_{pert})$ **do**
            Pick random value $\tilde{v}_i \in D_{V_i} \setminus \{v_i\}$
            Flip $V_i$ to $\tilde{v}_i$
            $\mathcal{V}_{pert} \leftarrow \mathcal{V}_{pert} \cup \{V_i\}$
    $\mathbf{v} \leftarrow$ *LocalSearch*$(\mathbf{v}, \mathcal{V}_{fix})$
    **return** $\mathbf{v}$
**end**
**Function** *LocalSearch* $(\mathbf{v}$, $\mathcal{V}_{fix})$
**begin**
    **while** ***true*** **do**
        Randomly pick $V_i \in \mathbf{V}$ and $v_i \in D_{V_i}$ maximizing $g(\mathbf{v}|V_i = v_i)$.
        **if** $g(\mathbf{v}|V_i = v_i) >$ best **then**
            Flip $V_i$ to $v_i$.
        **else**
            Randomly pick $V_i \in \mathbf{V} \setminus (\mathbf{E} \cup \mathcal{V}_{\text{fix}})$ and $v_i \in D_{V_i}$ maximizing $g(\mathbf{v}|V_i = v_i)$.
            **if** $g(\mathbf{v}|V_i = v_i) > g(\mathbf{v})$ **then**
                Flip $V_i$ to $v_i$.
            **else**
                **return** $\mathbf{v}$ *// No improving step possible anymore.*
**end**
```

Chapter 8

Experimenal Evaluation of SLS Algorithms

In this chapter, we evaluate our new algorithms, ILS and GLS$^+$, against the previous state-of-the-art SLS algorithms for MPE solving, G+StS [KD99b] and GLS [Par02]. We demonstrate that our new algorithms find solutions that are orders of magnitude better than those obtained from previous algorithms and are up to six orders of magnitude faster in solving instances to quasi-optimality. We also provide an analysis of the contributions every new component of our algorithms has to their strong performance.

8.1 Reproduction of Previous Results

For the experimental evaluation in this chapter, we employ our own implementations of the SLS algorithms G+StS [KD99b] and GLS [Par02]. We would have much preferred to use the original implementations of these algorithms, but these are tied into the larger reasoning systems SamIam[1] and REES[2] which obstructs a direct comparison.[3] The original version of GLS is part of the SamIam system, and the original version of G+StS is integrated into REES. Both reasoning sytems

[1]`http://reasoning.cs.ucla.edu/samiam/`

[2]`http://www.ics.uci.edu/~radum/rees`

[3]Both SamIam and REES generate MPE instances for an experimental evaluation on the fly, and can neither be easily used with automatic scripts nor parse external instances in the standard Bayesian interchange format (BIF). Further, SamIam is written in Java, obstructing a fair comparison with our C++ implementation, and REES is bound to the Windows operating system, which is incompatible with our infrastructure for larger experiments. Moreover, version 3.0.8 of REES as of April 2004 had a (known but unresolved) bug yielding wrong MPE values on instances which were not created by the system itself.

also provide independent implementations of the respective other algorithm, and there is one article by each of the groups that lists computational experiments comparing G+StS and GLS [Par02, MKD03]. Both of these papers provide results for real-world networks from the Bayesian networks repository as well as for a class of randomly generated instances (and some more in the case of [MKD03]). In order to demonstrate that the performance of our implementations of G+StS and GLS closely resembles their original performance, we first reproduce these results as closely as possible with our own implementations. We also reproduce further results in [MKD03] comparing G+StS and GLS to the systematic search algorithm s-BBMB [KD99a].

Unfortunately, we cannot perform experiments on exactly the same MPE instances as [Par02] and [MKD03]. This is because the networks in the Bayesian network repository do not include any evidence. In all research in the MPE domain we are aware of, evidence variables are sampled from the network on-the-fly, i.e., for each experiment, the evidence is newly sampled, the experiment is carried out, and then the evidence is discarded. This has the advantage that research does not concentrate on solving only an established set of benchmark instances, but always tries to optimize performance for all MPE instances that can result from a network. However, it also means that although we can compare the performance of our algorithms w.r.t. the previous algorithms' performance on a network in the Bayesian network repository, the actual MPE instances to be solved will not be identical. This limits the conclusions we can draw from our comparison. For random instances, the situation is only more extreme. There, not only the evidence is sampled on-the-fly, but also the networks themselves are generated on-the-fly. [Par02] and [MKD03] use simple but different generators for random networks that are integrated into the respective reasoning systems. We employ BNGenerator [IC02, IC03], which allows us to control important parameters of the networks, especially their induced width. Despite the different generation procedures, the relative performance of the algorithms we study seems to be consistent across the sets of randomly generated networks used in [Par02], [MKD03], and in this thesis.

For our "original" versions of G+StS and GLS, we use caching scheme *Simple*, the state-of-the-art caching scheme for SLS algorithms for MPE prior to our work. For GLS, we use the original parameter setting $\langle \rho, N_\rho \rangle = \langle 0.8, 200 \rangle$, and for G+StS, since no default parameters are specified in any of [KD99b, Par02, MKD03], we use our tuned parameter configuration $\langle cf, np \rangle = \langle 2, 40 \rangle$.

The first article reporting results for both G+StS and GLS is the one that introduces GLS for MPE [Par02]. This article compares the performance of GLS to the Discrete Lagrangian Method (DLM) and G+StS. Note that the implementations used in this evaluation are all versions by Park which are part of the SamIam system. Our "original" version of GLS seems to outperform his implementation of GLS. It manages to find solutions with strictly positive probability for each single instance in problem set `bnrep`, whereas in [Par02] none of 20 runs finds a solution with probability greater than zero for the `diabetes` network within 30 seconds; in our experiments, all 25 runs of GLS find a solution with positive evidence within 22.1 seconds. Also, for network `munin4`, we always find solutions with positive evidence within 11.5 seconds whereas only 18 out of the 20 runs reported in [Par02] do so within 30 seconds. We largely attribute these differences to the different programming languages (Java vs. C++) and computational architectures used.[4]

Our "original" version of G+StS also seems to clearly outperform the one used in [Par02]; it finds instantiations with strictly positive probability for more instances and much faster. For the `mildew` network, [Par02] reports G+StS to find instantiations with positive probability in 19 of 20 second runs of 30 seconds each, whereas our implementation finds instantiations of positive probability in all runs within just 0.08 seconds. For instance `munin1`, [Par02] reports G+StS to find no solution with positive probability in any of the runs, while all of our 25 runs find solutions of positive probability, and 22 of them do so within 30 seconds. Finally, for the `pigs` instance, [Par02] reports G+StS to find an instantiation of positive probability in only 1 of 20 runs, while our imlementation finds instantiations of positive probability within 0.06 seconds in every run. We attribute the fact that G+StS performs better in our experimental evaluation than in the one carried out in [Par02] to the fact that we employ our optimized parameter setting for G+StS which is likely to be better than the one used in [Par02].[5]

The fact that our G+StS implementation performs much faster than the one in [Par02] also has consequences for the comparison of GLS and G+StS. For the structured instances from `bnrep`, our implementation of G+StS solves more problems than our implementation of GLS and has a higher average approximation quality (see Table 8.1 on the next page). Nevertheless, as can be seen in Table 8.2 on the following page, our "original" version of GLS still outperforms our "original" version of G+StS by a large margin for the randomly generated instances in problem set `gen`. This is consistent with the results reported in [Par02].

The second article [MKD03] on which our reproduction of previous results is based compares a class of systematic search algorithms to the local search algo-

[4]As mentioned in Chapter 7, all our experiments are carried out on dual 2GHz Intel Xeon CPUs with 512KB cache and 4GB RAM running Linux version 2.4.20, build 28.9. We cannot relate this to the machines used in [Par02] since this article does not report on the computational architecture.

[5]We do not expect G+StS's parameter setting in [Par02] to be optimal since the parameter setting they employ for the GLS algorithm is very suboptimal although this algorithm is one of the main contributions of their paper. In their experiments, G+StS's noise probability was set to 20%, and the cutoff parameter is not reported.

	G+StS	GLS	s-BBMB				
Statistics	“original”	“original”	ib=2	ib=6	ib=10	ib=14	ib=18
Avg. quality	46.32	41.81	39.25	76.96	82.09	88.46	88.46
Avg. runtime	121.29	222.77	166.43	32.57	29.07	37.76	39.28
Successful runs	300/650	202/650	10/26	20/26	21/26	23/26	23/26
Instances solved	12/26	9/26	10/26	20/26	21/26	23/26	23/26
# amongst best	3	2	4	11	9	13	7

Table 8.1: Summary statistics for s-BBMB with different i-bounds and our “original” versions of G+StS and GLS on problem set `bnrep`. The SLS algorithms were run 25 times for 100 CPU seconds, the deterministic s-BBMB algorithm once for 100 CPU seconds for every i-bound. The SLS algorithms used a random initialization, simple caching, and parameter values $\langle np, cf \rangle = \langle 40, 2 \rangle$ (G+StS), and $\langle N_\rho, \rho \rangle = \langle 200, 0.8 \rangle$ (GLS). Summary of Table B.9 on page 149.

	G+StS	GLS	s-BBMB				
Statistics	“original”	“original”	ib=2	ib=6	ib=10	ib=14	ib=18
Avg. quality	18.30	36.35	0.03	12.70	51.82	75.84	67.51
Avg. runtime	760.01	339.95	∞	713.77	115.08	52.79	95.05
Successful runs	72/600	140/600	0/24	3/24	12/24	18/24	16/24
Instances solved	4/24	7/24	0/24	3/24	12/24	18/24	16/24
# amongst best	1	3	0	0	11	7	3

Table 8.2: Summary statistics for s-BBMB with different i-bounds and our “original” versions of G+StS and GLS on problem set `gen`. The SLS algorithms were run 25 times for 100 CPU seconds, the deterministic s-BBMB algorithm once for 100 CPU seconds for every i-bound. The SLS algorithms used a random initialization, simple caching, and parameter values $\langle np, cf \rangle = \langle 40, 2 \rangle$ (G+StS), and $\langle N_\rho, \rho \rangle = \langle 200, 0.8 \rangle$ (GLS). Summary of Table B.10 on page 150.

rithms G+StS, GLS, and DLM. It was published by the same group who suggested G+StS [KD99b], and, as far as we can tell, uses the original G+StS code. All algorithms are implemented as part of the REES system.

Our results for G+StS also seem to be somewhat better than the ones reported in [MKD03]. However, unfortunately, [MKD03] only reports the percentage of runs in which an algorithm found the optimal solution for an instance. For instances for which the optimal solution is not found, information about the suboptimal solution quality reached would be helpful; a special case of this would be whether or not an instantiation with positive evidence was found. For most of the instances from the Bayesian network repository, neither the original nor our implementation found the optimal solution and only two instances remain for a meaningful comparison, namely `barley` and `mildew`. While in [MKD03], G+StS never found the optimal solution for network `barley` within 30 seconds, our implementation always finds it within 100 seconds and needs 29.22 seconds on average. After 30 seconds, 60% of the runs had found the optimal solution.[6] For network `mildew`, [MKD03] reports 90% completed runs within 30 seconds. For this instance, our implementation needs 10.52 seconds on average; after 30 seconds, 92% of its runs had found optimal solutions, approximately matching the previous result. For randomly generated networks, according to [MKD03], GLS outperforms G+StS by a large margin which is consistent with our results.

In Table 8.1 on the preceding page and Table 8.2 on the facing page, we also reproduce the result from [MKD03] that systematic search algorithms are superior to both G+StS and GLS on most but not all MPE instances. For this comparison, we employ the original version of the Branch-and-Bound (BnB) algorithm s-BBMB [KD99a, MKD03]. This algorithm is described in Section 4.3 on page 32 and we will revisit it in Chapter 9 which compares our SLS algorithms to exact algorithms. The superiority of s-BBMB to both SLS algorithms is obvious for the structured instances in `bnrep` (see Table 8.1 on the facing page), for which s-BBMB with i-bounds 14 and 18 yields the best results. For the randomly generated instances in `gen` (see Table 8.2 on the preceding page), GLS outperforms s-BBMB for low settings of the i-bound of 2 and 6, but with i-bound 14, s-BBMB consistently performs better than GLS. This dominance of s-BBMB with optimal settings of the i-bound over GLS (and even more so G+StS) is consistent with the experimental results from [MKD03].

In summary, our “original” versions of G+StS and GLS closely resemble the original implementations introduced in [KD99b] and [Par02] and there is some evidence that our implementations may be slightly faster. This may be an artifact

[6]We cannot compare our computational architecture to the one used in [MKD03] since this article does report on it. As our algorithms, the algorithms in [MKD03] are implemented in C++.

due to the different computational architecture we employ.[7] Our implementation of G+StS seems to be much faster than the (non-original) implementation of G+StS that [Par02] uses. From here on, we will omit the quotation mark when we talk about our "original" versions.

8.2 Experimental Methodology: Correlation Plots

In our experimental analysis in this and the following chapter, we visualize pairwise comparisons of algorithms by employing correlation plots of (a) the average approximation quality they achieve for each instance in problem sets `bnrep` and `gen`, and (b) the average runtime they need to find quasi-optimal solutions. Each correlation plot holds one data point for each instance in problem sets `bnrep` and `gen`. We visualize instances in `bnrep` by red colour and instances in `gen` by blue colour. Instances with random CPTs are represented by circles, instances with structured CPTs by crosses; see Figure 8.1 on page 92 for an example. For all correlation plots in the remainder of this thesis, 25 runs of 100 CPU seconds each are performed.

Two special cases can occur in the correlation plots. Firstly, when plotting average approximation quality, for some instances one or both algorithms may have quality zero when they never found a solution with positive probability. Since we employ a logarithmic scale, in this case, instead of zero, we plot a small positive value q_0 which is significantly smaller than any positive quality achieved by one of the two algorithms on any instance; if applicable, we specify q_0 in the caption of the figure.[8] Secondly, in correlation plots of average runtime it happens frequently that one or both algorithms do not find quasi-optimal solutions in any of the 25 runs of 100 CPU seconds each. In this case, we plot the value $10,000 = 10^4$ which is greater than the maximal value $2,500$ that can be obtained if a solution is found in 1 of the 25 runs (recall that the average runtime is defined as the runtime summed over all runs, divided by the number of successful runs).

Recall, that the approximation quality for a run of an algorithm is defined as the ratio of the nonlogarithmic solution quality it achieves and the quasi-optimal

[7] However, we do not expect a major difference in the used architectures since the articles we compare our results with have been published very recently. Unfortunately, neither of the articles reports the computational architecture used in their experiments.

[8] In different correlation plots for approximation quality, we employ different values for q_0 since in the extreme it needs to be as low as 10^{-220}, whereas for some plots values of 10^{-2} suffice. Using $q_0 = 10^{-220}$ in these cases would compress the interesting parts of the figure to a hundreth of the space which would reduce clarity. For some plots, both algorithms find positive probability for all instances, in which case q_0 does not apply.

	G+StS			ILS	
	old caching	new caching		new caching	
Statistics	random	random	MB$^*(10^5)$	random	MB$^*(10^5)$
Avg. quality	46.32	46.91	77.08	57.96	85.59
Avg. runtime	121.29	117.62	56.63	94.11	37.22
Successful runs	300/650	300/650	421/650	338/650	479/650
Instances solved	12/26	12/26	17/26	17/26	20/26
# amongst best	0	1	10	4	18

Table 8.3: Summary statistics for non-penalty based algorithms on problem set `bnrep`. All algorithms were run 25 times for 100 CPU seconds each with their default parameters. Summary of Table B.11 on page 151.

one. We present average approximation qualities in percent, i.e., if an algorithm has approximation quality 100 for an instance, it finds quasi-optimal solutions for the instance in every run. Note that from the correlation plots of average approximation quality, we can infer the ratios of the average probability of the variable instantiations the algorithms find. If, for example, the two algorithms compared in a correlation plot have average approximation qualities 2×10^{-15} and 19.7, respectively, this means that the latter algorithm on average finds variable instantiations with a probability roughly 16 orders of magnitude higher than the average probability of the solutions found by the first algorithm.

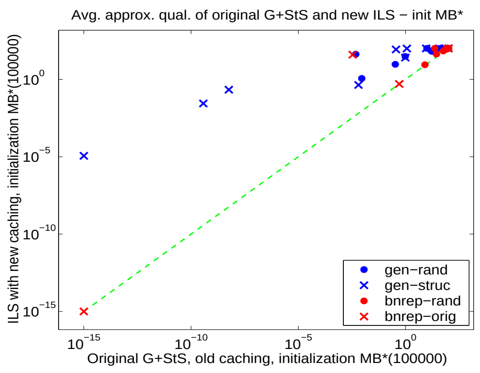
8.3 G+StS vs. ILS

In this section, we compare the performance of ILS and our original version of G+StS. We demonstrate that ILS yields immensely better results and independently study the effect of each component contributing to its high performance. Table 8.3 and Table 8.4 on the following page summarize the experiments the performance comparison in this section is based on. For the full results, see Table B.11 on page 151 and Table B.12 on page 152.

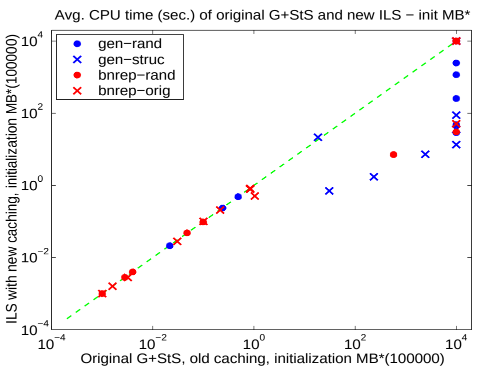
In Figure 8.1 on the next page, we compare the performance of ILS and our original version of G+StS for the case when both algorithms employ the same initialization MB$^*(10^5)$. ILS clearly outperforms G+StS, finding instantiations of more than 7 orders of magnitude higher probability, solving larger instances to quasi-optimality up to two orders of magnitude faster, and solving many instances which are unsolvable for G+StS. This already shows the clear superiority of ILS, but for small and easy instances the picture is incomplete since the common MB$^*(10^5)$ initialization already yields very high-quality solutions. This initializa-

	G+StS			ILS	
	old caching	new caching		new caching	
Statistics	random	random	MB$^*(10^5)$	random	MB$^*(10^5)$
Avg. quality	18.30	25.58	37.43	64.76	68.42
Avg. runtime	760.01	488.73	309.69	118.06	95.57
Successful runs	72/600	104/600	150/600	296/600	332/600
Instances solved	4/24	6/24	7/24	15/24	16/24
# amongst best	0	0	1	13	11

Table 8.4: Summary statistics for non-penalty based algorithms on problem set `gen`. All algorithms were run 25 times for 100 CPU seconds each with their default parameters. All algos ran for $t = 100$ seconds. Summary of Table B.12 on page 152.

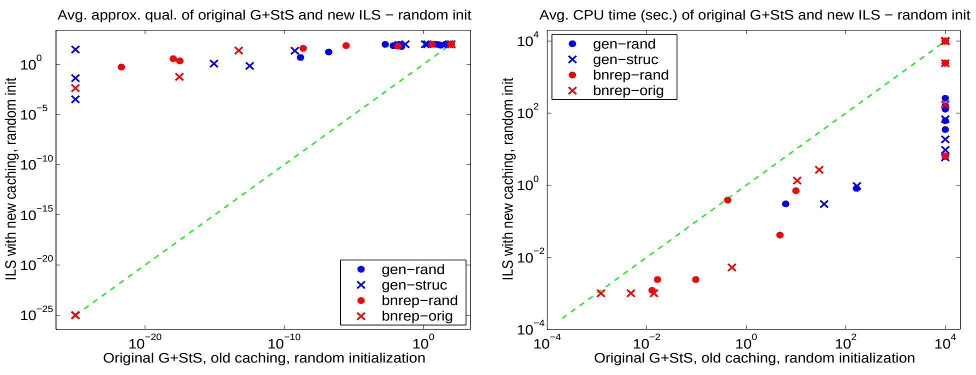


(a) Approximation quality, $q_0 = 10^{-15}$



(b) Runtime to find quasi-optimal solution

Figure 8.1: Performance differences between ILS and our original version of G+StS, both initialized with MB$^*(10^5)$. Average approximation quality (a) and average runtime to find a quasi-optimal solution (b). The algorithms were run 25 times for 100 CPU seconds each.



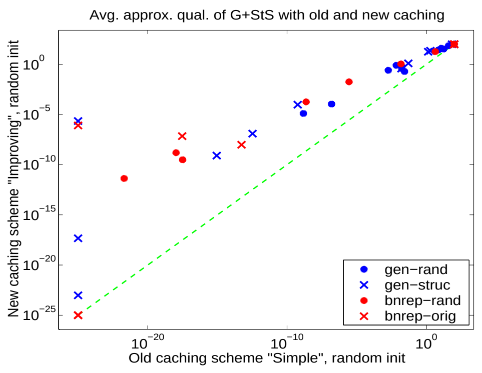
(a) Approximation quality, $q_0 = 10^{-25}$ (b) Runtime to find quasi-optimal solution

Figure 8.2: Performance differences between ILS and our original version of G+StS, both initialized at random. Average approximation quality (a) and average runtime to find a quasi-optimal solution (b). The algorithms were run 25 times for 100 CPU seconds each.

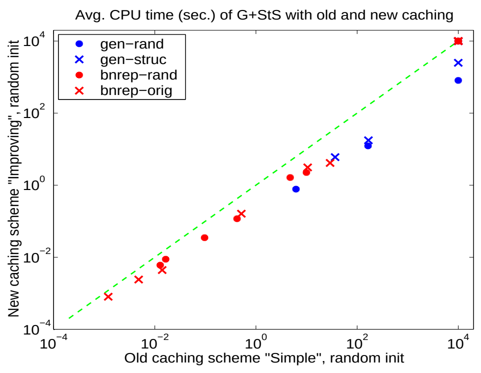
tion partly obstructs the performance comparison of the two algorithms since for some instances it yields high initial solution qualities which the local search part of G+StS alone would never find. In Figure 8.2, where both algorithms employ a random initialization, ILS performs much better for small instances as well, outperforming G+StS by up to 2 orders of magnitude. Also, when initialized at random, G+StS fails to find instantiations with positive probability for many instances for which ILS succeeds in this task. For other instances, ILS finds instantiations with up to 22 times higher probability than G+StS. Furthermore, the number of instances solved to quasi-optimality by ILS but unsolved by G+StS increases considerably.

ILS and our original version of G+StS differ in two components, namely the caching scheme used and the inner workings of the local search that follows the initialization in every try of the algorithms. Both algorithms employ a very similar restart mechanism. We now demonstrate how each of the two differing components affects algorithm performance.

Figure 8.3 on the next page demonstrates the large performance gains our new caching scheme *Improving* yields for G+StS when compared to the previous state-of-the-art caching scheme *Simple*. Up to 20 orders of magnitude more likely instantiations are found with our improved caching scheme. The speedup over simple caching for finding quasi-optimal solutions is up to a factor of 10 and increases for harder instances. Also, there are several instances for which simple caching only yields solutions with probability zero but the our improved caching yields positive probabilities.



(a) Approximation quality, $q_0 = 10^{-25}$



(b) Runtime to find quasi-optimal solution

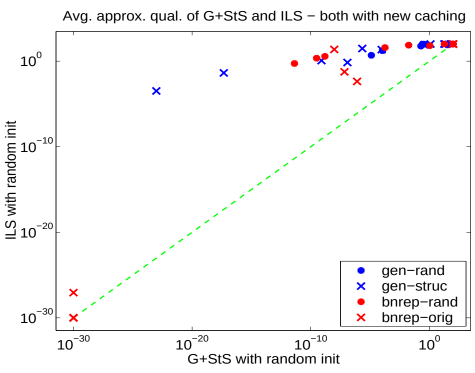
Figure 8.3: The performance differences due to our improved caching scheme for G+StS with random initialization. Average approximation quality (a) and average runtime to find a quasi-optimal solution (b). The algorithms were run 25 times for 100 CPU seconds each.

Although with our new caching, G+StS performs much better than in its original version, this algorithm is still vastly inferior to ILS; we demonstrate this in Figure 8.4 on the facing page, in which the only differing component between the algorithms is the local search strategy. ILS finds instantiations of up to 20 orders higher probability than G+StS, is about 10 times faster in finding quasi-optimal solutions for easy instances, and for harder instances where G+StS fails in finding quasi-optimal solutions, ILS often succeeds in seconds.

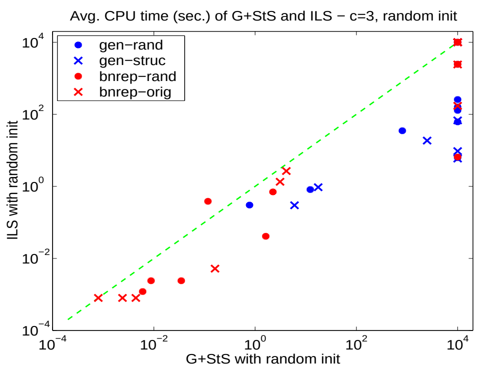
Since ILS clearly employs a much stronger local search than G+StS, it might not require a strong (and possibly time-consuming) initialization of the search anymore. However, in Figure 8.5 on the next page, we show that this is not the case and that ILS still gains much from a strong initialization heuristic. As opposed to ILS with random initialization, ILS with MB$^*(10^5)$ always finds solutions with strictly positive probability; it generally finds higher quality solutions, often finds quasi-optimal solutions several orders of magnitude faster, and sometimes quickly solves instances to quasi-optimality which remain unsolved employing a random initialization.

8.4 GLS vs. GLS$^+$

In this section, we compare the performance of our implementation of the original GLS algorithm and our improved GLS$^+$ algorithm. Figure 8.6 on page 96 demon-

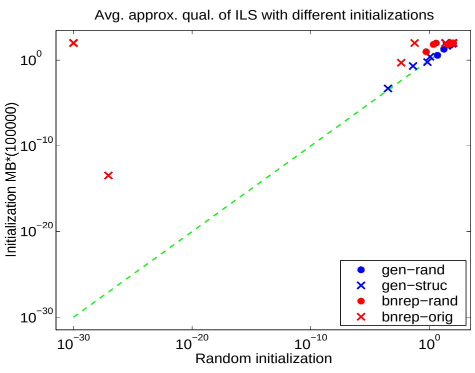


(a) Approximation quality, $q_0 = 10^{-30}$

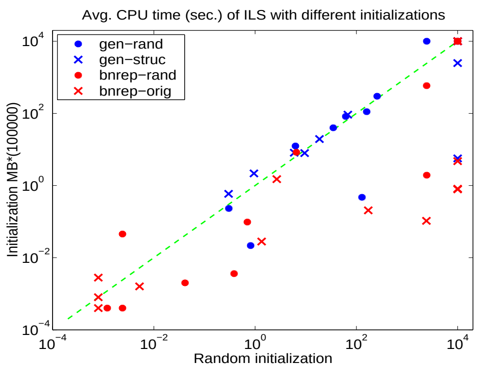


(b) Runtime to find quasi-optimal solution

Figure 8.4: The performance differences due to the local search strategy used after the initialization. G+StS and ILS employ the same caching scheme *Improving* and are initialized at random. Average approximation quality (a) and average runtime to find a quasi-optimal solution (b). The algorithms were run 25 times for 100 CPU seconds each.

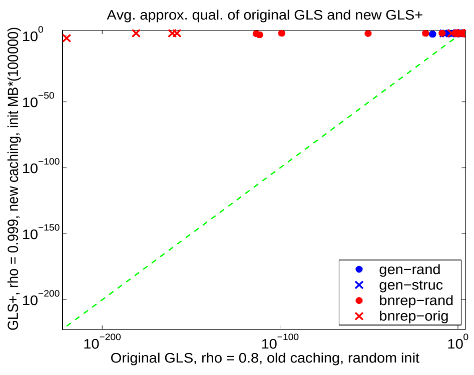


(a) Approximation quality, $q_0 = 10^{-30}$

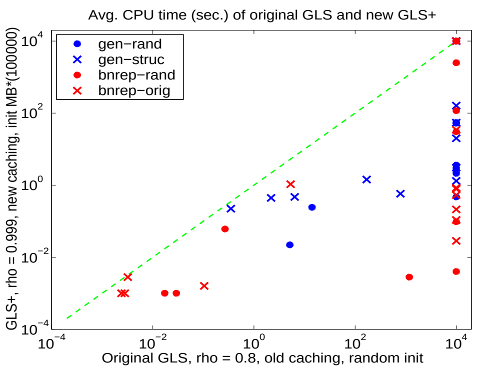


(b) Runtime to find quasi-optimal solution

Figure 8.5: The performance differences due to different initializations of ILS: random initialization and MB$^*(10^5)$. Average approximation quality (a) and average runtime to find a quasi-optimal solution (b). The algorithms were run 25 times for 100 CPU seconds each.



(a) Approximation quality



(b) Runtime to find quasi-optimal solution

Figure 8.6: Performance differences between the original GLS algorithm and our new algorithm GLS$^+$. Average approximation quality (a) and average runtime to find a quasi-optimal solution (b). The algorithms were run 25 times for 100 CPU seconds each.

strates the enormous performance differences between GLS$^+$ and our original version of GLS. GLS$^+$ finds solutions with probability up to 268 orders of magnitude higher than those found by GLS, is up to 6 orders of magnitude faster in finding quasi-optimal solutions and for a great number of instances finds quasi-optimal solutions that GLS cannot find, some of them in milliseconds.

Our implementation of the original GLS algorithm and our improved algorithm GLS$^+$ differ in a number of components, namely the parameter setting, the caching scheme, the evaluation function and the initialization. In the following, we demonstrate how each of these components contributes to the improved performance of GLS$^+$. Table 8.5 on the next page and Table 8.6 on the facing page summarize the performance resulting from introducing the novel components of GLS$^+$ into the original version of GLS one at a time. It shows the performance of the original version of GLS with the original parameter setting from [Par02] and the algorithm's steadily increasing excellence when incrementally introducing the tuned smoothing parameter $\rho = 0.999$, our improved caching scheme *Improving*, the evaluation function of GLS$^+$, and initialization MB$^*(10^5)$. For the full results, see Table B.13 on page 153 and Table B.14 on page 154.

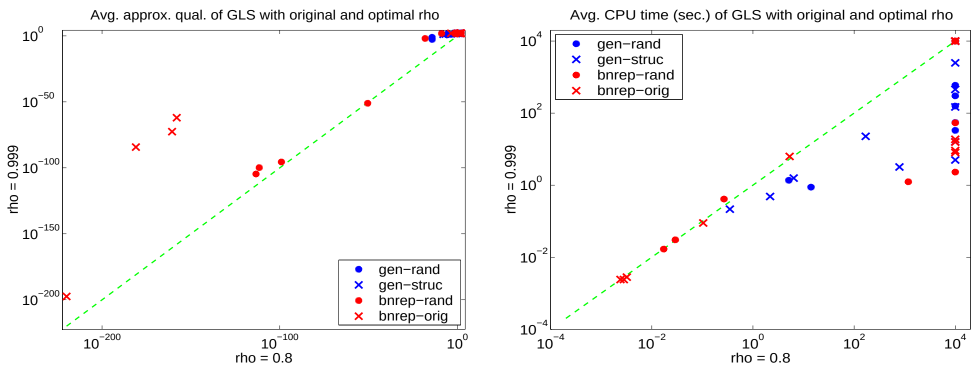
As in the previous section, we visualize the impact of each component in correlation plots of algorithm versions that only differ in one component. Figure 8.7 on page 98 shows the great differences in performance achievable by simply tuning the parameters of GLS better than in its original version [Par02], where the default parameter setting from the Max-SAT domain [MT00] was employed for N_ρ

	GLS, random intitialization			GLS$^+$	
Statistics	“original”	$\rho = 0.999$	$\rho = 0.999$,new caching	random	MB$^*(10^5)$
Avg. quality	41.81	63.77	75.52	77.16	89.05
Avg. runtime	222.77	81.05	59.32	43.22	37.61
Successful runs	202/650	375/650	429/650	476/650	491/650
Instances solved	9/26	15/26	19/26	21/26	21/26
# amongst best	0	0	1	11	16

Table 8.5: Summary statistics for penalty based algorithms on problem set `bnrep`. All algorithms were run 25 times for 100 CPU seconds each. Summary of Table B.13 on page 153.

	GLS, random intitialization			GLS$^+$	
Statistics	“original”	$\rho = 0.999$	$\rho = 0.999$,new caching	random	MB$^*(10^5)$
Avg. quality	36.35	70.17	87.20	88.18	89.35
Avg. runtime	339.95	107.96	60.42	57.06	49.96
Successful runs	140/600	314/600	400/600	409/600	430/600
Instances solved	7/24	17/24	20/24	22/24	18/24
# amongst best	0	0	2	12	10

Table 8.6: Summary statistics for penalty based algorithms on problem set `gen`. All algorithms were run 25 times for 100 CPU seconds each. Summary of Table B.14 on page 154.



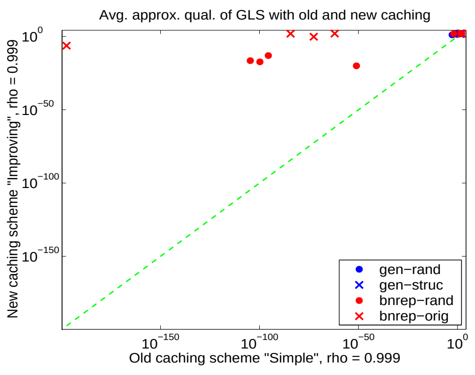
(a) Approximation quality (b) Runtime to find quasi-optimal solution

Figure 8.7: The performance differences only due to tuning ρ for GLS with simple caching and $N_\rho = 200$. Average approximation quality (a) and average runtime to find a quasi-optimal solution (b). The algorithms were run 25 times for 100 CPU seconds each.

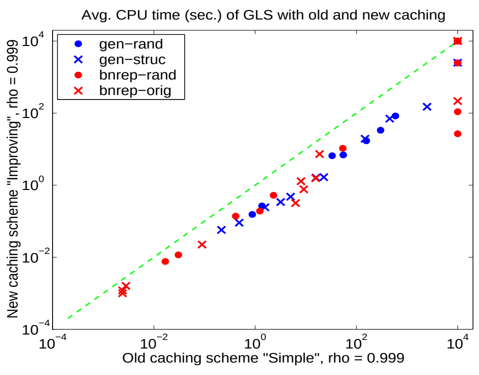
and ρ. In his formulation of GLS, Park does not even treat these crucial elements of the algorithm as parameters, and indeed states GLS to "have no parameters to tune" [Par02]. Figure 8.7 contradicts this statement, showing that by simply choosing a higher smoothing parameter $\rho = 0.999$, vastly stronger performance can be achieved: for 3 instances from `bnrep`, the improved version finds instantiations that are up to 96 orders of magnitude more likely; for harder instances, it finds quasi-optimal solutions much faster than with the original parameter setting for which GLS frequently fails to find them at all.

In Figure 8.8 on the next page, we demonstrate the comparably impressive impact of our new caching scheme *Improving*. With this caching scheme, GLS sometimes finds instantiations with a dramatically higher likelihood than with the previous caching scheme *Simple*. For some structured instances, this difference is up to 80 orders of magnitude, and for instance `diabetes` it is even 191 orders of magnitude. In terms of runtime to reach quasi-optimal solutions, our new caching scheme yields a speedup reaching from 2 for easy instances to approximately 10 for harder instances. There are also some instances which can be solved to quasi-optimality with our improved caching scheme but not with the previous one.

Figure 8.9 on page 100 shows the considerable performance gains of our new version GLS$^+$ over GLS with identical parameter setting, caching scheme, and initialization. For six instances, GLS$^+$ finds instantiations up to 7 orders of magnitude more likely than the ones found by GLS. It finds quasi-optimal solutions of all instances faster than GLS, sometimes outperforming it by a factor of up to 10. It also



(a) Approximation quality



(b) Runtime to find quasi-optimal solution

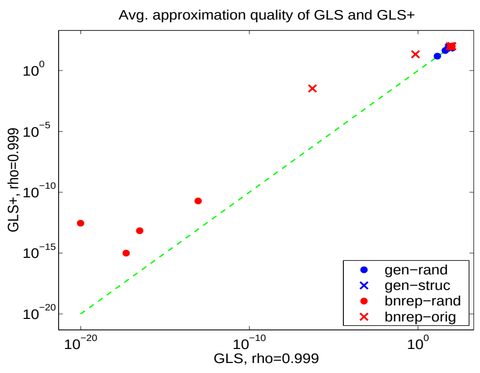
Figure 8.8: The performance differences only due to our improved caching scheme for GLS with $\langle \rho, N_\rho \rangle = \langle 0.999, 200 \rangle$. Average approximation quality (a) and average runtime to find a quasi-optimal solution (b). The algorithms were run 25 times for 100 CPU seconds each.

finds quasi-optimal solutions for four instances on which GLS fails.

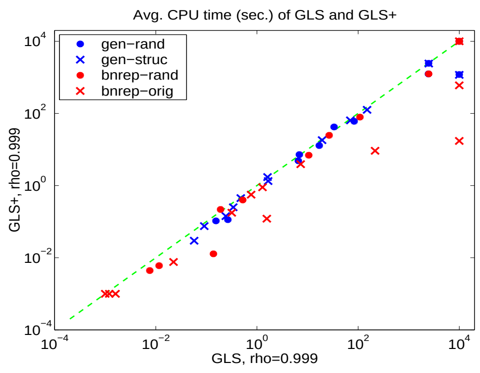
The final new component in GLS$^+$ is the different initialization MB$^*(10^5)$. We demonstrate the impact of this in Figure 8.10 on the next page, which compares the performance of GLS$^+$ with different initializations. On the one hand, we observe that for four instances, GLS$^+$ with initialization MB$^*(10^5)$ finds much more likely instantiations than GLS$^+$ with a random initialization, and that it is much faster on average; for some instances, the stronger initialization yields speedups of up to a factor of 80. However, on the other hand, we also observe that for some hard randomly generated instances, GLS$^+$ finds quasi-optimal solutions when initialized at random, but not with initialization MB$^*(10^5)$.[9] In total, however, initialization MB$^*(10^5)$ performs much better for GLS$^+$.

In summary, all of the four differing components between GLS$^+$ and our original version of GLS lead to great improvements. For us, the most interesting component is the new evaluation function of GLS$^+$ as we see possibilities for improving it even further in future work. We now provide some more evidence for the strong performance of GLS$^+$, but we also demonstrate that although it consistently shows stronger initial performance than GLS, for longer runs the solution qualities reached by both algorithms become virtually identical.

[9]However, for these instances, GLS$^+$ with random initialization is the only algorithm that finds the quasi-optimal solutions we employ, and only does so in 1 or 2 out of 25 runs. We are much in doubt of the actual optimality of these quasi-optimal solutions.

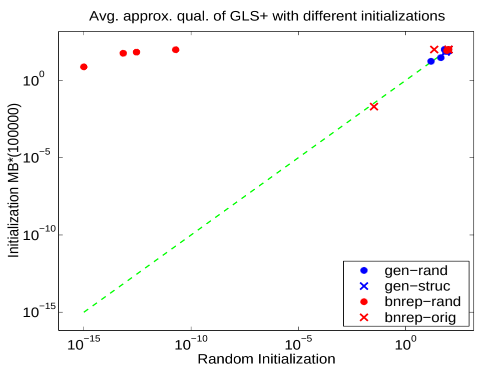


(a) Approximation quality

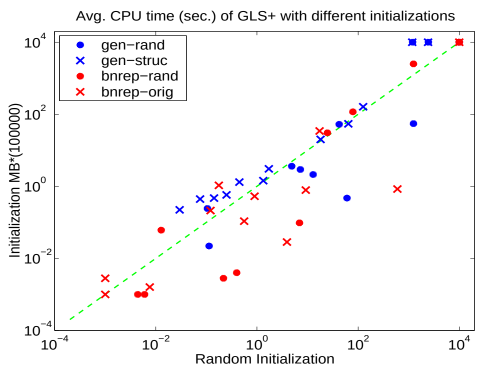


(b) Runtime to find quasi-optimal solution

Figure 8.9: $\text{GLS}(0.999, 200)$ vs. $\text{GLS}^+(0.999, 200, 10000)$, both with random initialization and caching scheme *Improving*. Average approximation quality (a) and average runtime to find a quasi-optimal solution (b). The algorithms were run 25 times for 100 CPU seconds each.

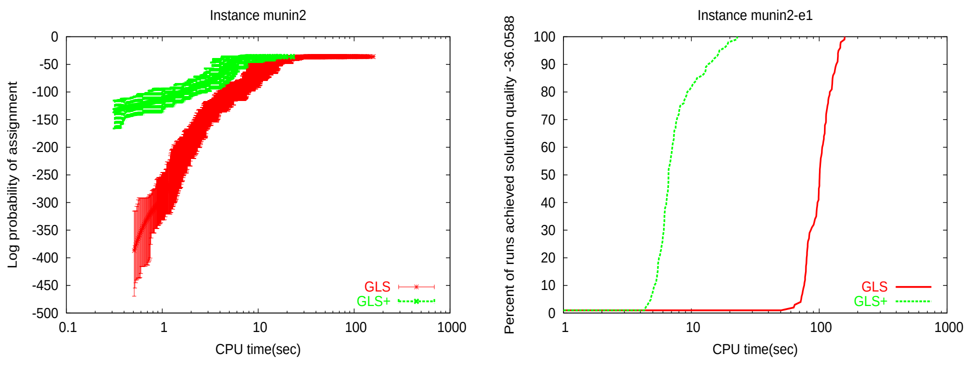


(a) Approximation quality



(b) Runtime to find quasi-optimal solution

Figure 8.10: The performance differences only due to different initializations of GLS^+: random initialization and $\text{MB}^*(10^5)$. Average approximation quality (a) and average runtime to find a quasi-optimal solution (b). The algorithms were run 25 times for 100 CPU seconds each.

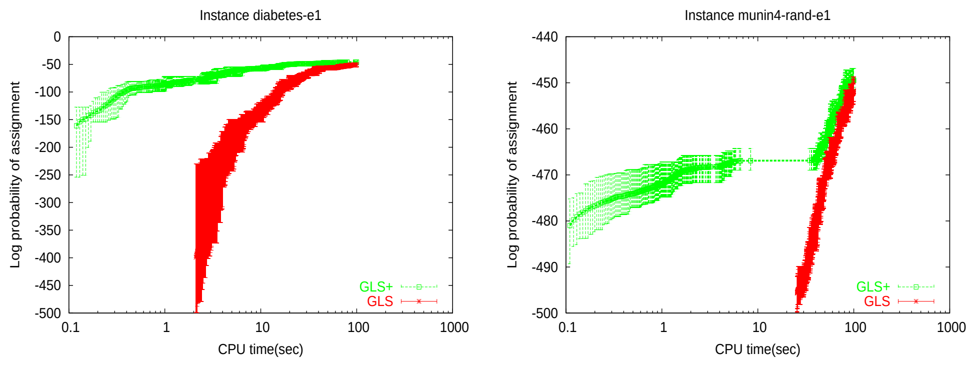


(a) Mean solution quality (b) Empirical runtime distribution

Figure 8.11: Plots of mean solution qualities (a) and empirical RTDs (b) of GLS and GLS$^+$ on instance `munin2`. Both algorithms employ a random initialization, caching scheme *Improving*, and default parameter setting $\langle N_\rho, \rho\rangle = \langle 200, 0.999\rangle$. The estimation is based on 100 runs of 1000 CPU seconds each. In (a), the mean solution quality plot for each algorithm ends when an optimal solution was found in all of its 100 runs; in (b), the target quality -36.0588 is the optimal solution quality.

Figure 8.11 shows a plot of mean solution qualities and empirical RTDs for GLS and GLS$^+$ with identical parameter settings for instance `munin2`. For this instance, the initially stronger performance of GLS$^+$ suffices to find the optimal solution quality more than ten times faster. The empirical RTD for GLS$^+$ suggests that it may potentially suffer from search stagnation, but we found no further evidence for this.

Figure 8.12 on the next page shows plots of mean solution quality for two other real-world instances, `diabetes` and `munin4-rand`. For both networks, GLS$^+$ shows very strong initial performance, finding instantiations of positive probability virtually instantaneously whereas GLS needs considerable time for this. It is also very interesting to observe that once GLS starts to perform very well for network `munin4-rand`, GLS$^+$ matches its performance closely. For network `diabetes` with structured CPTs, the initial performance of GLS$^+$ is stronger than for network `munin4-rand` with random CPTs. This different behaviour for structured and randomly generated CPTs can be observed for many other instances from `bnrep`. It is also present for the instances in `gen` which employ a randomly generated graph structure. Figure 8.13 on page 103 shows that for such a network with structured CPTs, GLS$^+$ finds very good solutions quickly, whereas in the case of random CPTs, GLS catches up more quickly and GLS$^+$ again closely matches its perfor-



(a) `diabetes` (b) `munin4-rand`

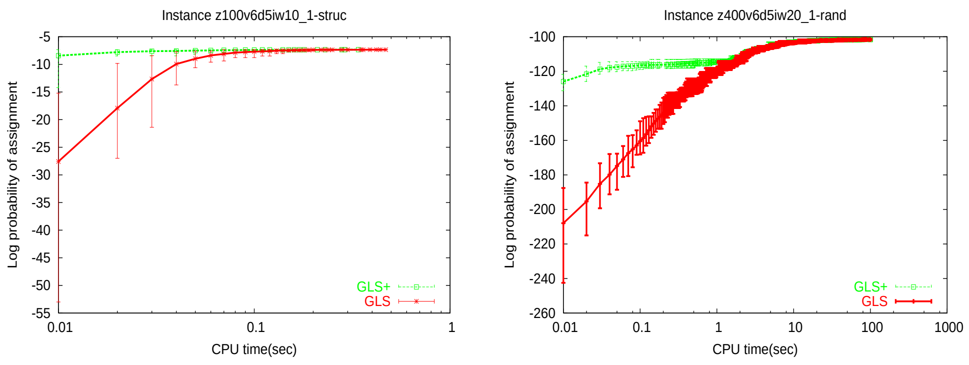
Figure 8.12: Plots of mean solution quality of GLS and GLS$^+$ on instances from problem set `bnrep`. Both algorithms employ a random initialization, caching scheme *Improving*, and default parameter setting $\langle N_\rho, \rho\rangle = \langle 200, 0.999\rangle$. The estimation is based on 25 runs of 100 CPU seconds each.

mance after its initial head start.

8.5 ILS vs. GLS$^+$

In the last two sections, we have demonstrated the much higher performance of our new algorithms ILS and GLS$^+$ when compared to the previous state-of-the-art in SLS algorithms for MPE.

To conclude our experimental analysis of SLS algorithms for MPE, we now compare our new algorithms against each other. Like in the comparison of G+StS and ILS, strong initializations skew the picture and we thus compare ILS and GLS$^+$ two times, with a random initialization and with initialization MB$^*(10^5)$. In Figure 8.15 on page 105, the fundamental differences between ILS and GLS$^+$ (both employing a random initialization) are quite obvious. On the one hand, GLS$^+$ finds instantiations with positive probability for all instances whereas ILS fails to do so for 2 structured instances with structured CPTs. For instances with structured CPTs, GLS$^+$ generally performs much better than ILS; the differences in approximation quality reach 26 orders of magnitude in this case. However, for four structured instances with random CPTs, ILS finds instantiations that are considerably more likely than the ones found by GLS$^+$; for these four instances, the differences are as big as 15 orders of magnitude. In terms of runtime to find quasi-optimal solutions, the picture is quite similar but more in favour for GLS$^+$. Instances with



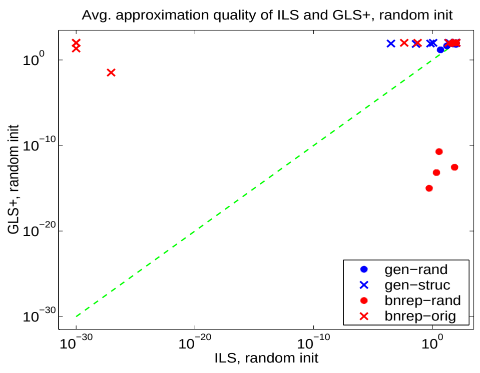
(a) `z100v6d5iw10-struc` (b) `z400v6d5iw20-rand`

Figure 8.13: Plots of mean solution quality of GLS and GLS$^+$ on instances from problem set `gen`. Both algorithms employ a random initialization, caching scheme *Improving*, and default parameter setting $\langle N_\rho, \rho\rangle = \langle 200, 0.999\rangle$. The estimation is based on 25 runs of 100 CPU seconds each.

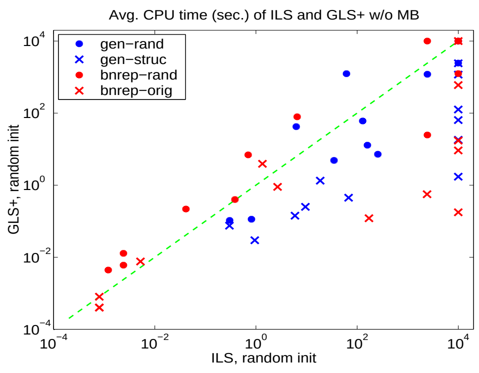
structured CPTs are solved to quasi-optimality up to 3 orders of magnitude faster by GLS$^+$, and GLS$^+$ also quickly solves many instances ILS cannot solve. For instances with random CPTs, ILS and GLS$^+$ both outperform the respective other algorithm on some instances; for a few instances, ILS is up to an order of magnitude faster than GLS$^+$, whereas for other instances, GLS$^+$ is almost up to 2 orders of magnitude faster.

When ILS and GLS$^+$ are initialized with MB$^*(10^5)$ (see Figure 8.15 on page 105), the overall picture becomes less clear but the main conclusions we can draw from it remain unchanged. MB$^*(10^5)$ already generates high quality solutions for all the instances for which ILS found much more probable solutions than GLS$^+$ when both algorithms were initialized at random. Thus, with the MB$^*(10^5)$ initialization, GLS$^+$ now clearly performs much better in terms of approximation quality than ILS. In terms of time needed to find quasi-optimal solutions, ILS still sometimes outperforms GLS$^+$ (for real world instances with randomized CPTs like `pigs-rand` and `munin1-rand`), but GLS$^+$ continues to outperform ILS by up to two orders of magnitude for other instances, especially for randomly generated instances with structured CPTs. GLS$^+$ also still solves some instances to quasi-optimality which ILS cannot solve, namely `link` and some large randomly generated instances.[10] Their low correlation of runtimes to find quasi-optimal solutions

[10]For a per instance comparison of ILS and GLS$^+$, see Tables B.17 and B.18 on pages 157 and 158.



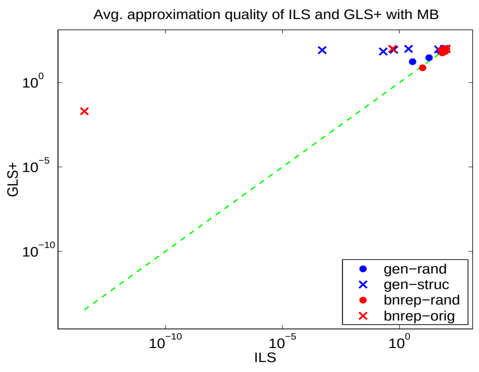
(a) Approximation quality, $q_0 = 10^{-30}$



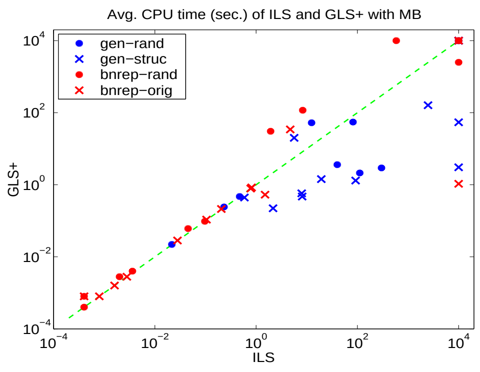
(b) Runtime to find quasi-optimal solution

Figure 8.14: Comparison of our new algorithms ILS and GLS$^+$, both with random initialization. Average approximation quality (a) and average runtime to find a quasi-optimal solution (b). The algorithms were run 25 times for 100 CPU seconds each.

suggests the suitability of ILS and GLS$^+$ to be combined in an algorithm portfolio. In Section 5.4 on page 46, we combined them with our Mini-Buckets variant MB* to form a hybrid algorithm that alternates phases of these algorithms. In Chapter 9, we will demonstrate this algorithm to be the new state-of-the art in MPE solving.



(a) Approximation quality



(b) Runtime to find quasi-optimal solution

Figure 8.15: Comparison of our new algorithms ILS and GLS$^+$, both with initialization MB$^*(10^5)$. Average approximation quality (a) and average runtime to find a quasi-optimal solution (b). The algorithms were run 25 times for 100 CPU seconds each.

Chapter 9

Comparison with Exact Algorithms

In this chapter, we evaluate a number of exact algorithms for MPE and compare our new algorithms against the previously best-performing ones. We demonstrate that our hybrid algorithm of ILS, GLS$^+$, and MB* shows better overall performance than any of the other algorithms. We also show that our SLS algorithms scale much better in terms of a number of important instance characteristics, namely the number of variables, the domain size, and the degree, and induced width of the independence graph. When compared to our new algorithms, s-BBMB with a small i-bound of 6 scales poorly with number of variables, degree, and induced width; and for higher i-bounds, it scales poorly with domain size and induced width. Anytime MB scales poorly with domain size and especially poorly with induced width.

9.1 Performance of Systematic Algorithms

In this section, we compare the performance of the systematic MPE algorithms Anytime MB, s-BBMB, and d-BBMB. All these algorithms are described in Section 4.3 on page 32. For Anytime MB, we employ our own C++ implementation since we are not aware of any available implementation. For s-BBMB and d-BBMB, we employ a UNIX-based C++ implementation provided by Radu Marinescu.[1] In compliance with his advice, we report results for a range of i-bounds, namely *ib* $\in \{2, 6, 10, 14, 18\}$. We do not compare these algorithms to BBBT, the other systematic search algorithm used in the experimental study of [MKD03], because BBBT is not included in the implementation provided by Radu Marinescu. It is available as part of the REES reasoning system, but as we mentioned in Section 8.1 on page 85, we experienced problems with this system, the most important

[1]Many thanks to Radu Marinescu for providing this implementation.

of which was a bug disallowing correct computations on instances that are not generated by the system itself.

The very recently developed d-BBMB algorithm [MD04] has not been listed in the experimental study of [MKD03]. It has subsequently been claimed to outperform s-BBMB[2], but in our experiments we show it to perform considerably worse on the instances we study.

Tables 9.1 and 9.2 on the facing page summarize our experiments with deterministic algorithms for problem sets `bnrep` and `gen`, respectively. In these tables, the row "Successful runs" is omitted since for these deterministic algorithms, it is identical to "Instances solved". The new row "Instances proved" states how many instances could be solved to optimality and their solutions been proven to be optimal within the time bound and the row "Instances > 0" represents for how many instances solutions with positive probability were found. As in the previous chapters, all our experiments were carried out on dual 2GHz Intel Xeon CPUs with 512KB cache and 4GB RAM running Linux version 2.4.20, build 28.9.

For the instances in problem set `bnrep`, very good performing i-bounds for s-BBMB are 10 and 14; i-bound 18 yields exactly the same results as i-bound 14, but causes a longer runtime on average. For d-BBMB, i-bound 6 yields the best results, closely followed by i-bounds 10 and 14. As can be seen in the full results for problem set `bnrep` in Table B.15 on page 155, there are significant differences in the optimal i-bounds for each instance. For example, instance `link-rand` can only be solved with i-bounds greater or equal to 14, whereas instance `munin1` causes the algorithm to break with such i-bounds but can be solved in seconds employing an i-bound as low as 2. Algorithm Anytime MB shows very strong performance on the structured instances in problem set `bnrep`, leaving only instance `link-rand` unsolved.

The situation is similar for problem set `gen`. In this case, the optimal i-bound for d-BBMB is 14, as high as the optimal one for s-BBMB. We again observe a large variability in the optimal i-bound across the instances. For large instances, high i-bounds tend to perform much better, but for networks with high induced size (networks with both high induced width and large domain size), high i-bounds cause both algorithms to break down even for small instances with 100 variables. On the randomly generated networks in problem set `gen`, Anytime MB does not perform as well as for problem set `bnrep`. It only solves 9 of the 24 instances and cannot even find solutions with positive probability for three of the others.

Like in Chapter 8, in the following we visualize the relative performance of two algorithms in correlation plots of their approximation quality and their run-

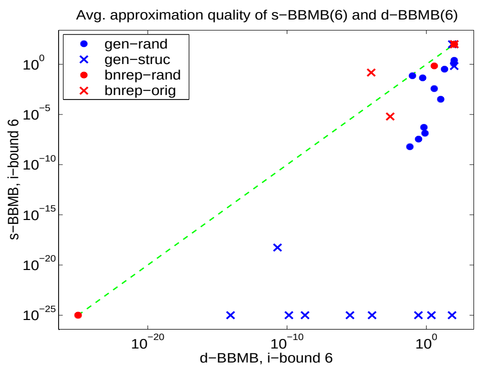
[2]This claim was expressed in email communication by Radu Marinescu and also follows from the experimental results presented in [MD04].

	d-BBMB					s-BBMB					Anytime
Statistics	ib=2	ib=6	ib=10	ib=14	ib=18	ib=2	ib=6	ib=10	ib=14	ib=18	MB
Avg. quality	46.20	69.37	67.86	65.38	65.38	39.25	76.96	82.09	80.77	80.77	97.69
Avg. runtime	122.94	48.07	63.05	63.38	63.39	166.43	32.57	29.07	28.36	29.55	5.96
Instances solved	12/26	18/26	17/26	17/26	17/26	10/26	20/26	21/26	21/26	21/26	25/26
Instances proved	10/26	18/26	17/26	16/26	16/26	8/26	20/26	21/26	21/26	20/26	25/26
Instances > 0	19/26	21/26	18/26	17/26	17/26	19/26	23/26	23/26	21/26	21/26	26/26
# amongst best	1	3	3	3	3	4	10	6	9	6	15

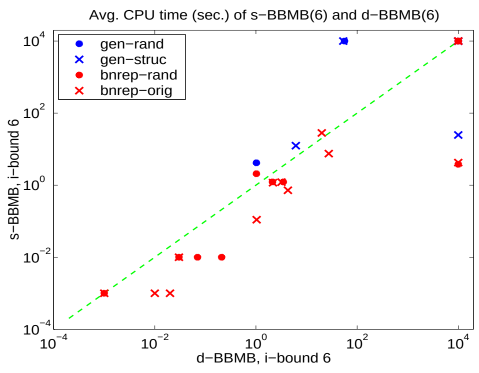
Table 9.1: Summary statistics for exact algorithms on problem set `bnrep`. All algorithms were run for 100 CPU seconds. Summary of Table B.15 on page 155.

	d-BBMB					s-BBMB					Anytime
Statistics	ib=2	ib=6	ib=10	ib=14	ib=18	ib=2	ib=6	ib=10	ib=14	ib=18	MB
Avg. quality	0.47	28.10	35.26	41.49	29.17	0.03	12.70	51.82	75.84	55.01	53.90
Avg. runtime	∞	529.19	252.11	196.79	264.97	∞	713.77	115.08	52.79	107.42	80.93
Instances solved	0/24	4/24	7/24	9/24	7/24	0/24	3/24	12/24	18/24	13/24	9/24
Instances proved	0/24	2/24	5/24	7/24	7/24	0/24	3/24	12/24	17/24	13/24	9/24
Instances > 0	16/24	24/24	12/24	11/24	7/24	13/24	16/24	23/24	18/24	13/24	21/24
# amongst best	0	1	0	0	0	0	0	14	7	3	1

Table 9.2: Summary statistics for exact algorithms on problem set `gen`. All algorithms were run for 100 CPU seconds. Summary of Table B.16 on page 156.



(a) Approximation quality, $q_0 = 10^{-25}$



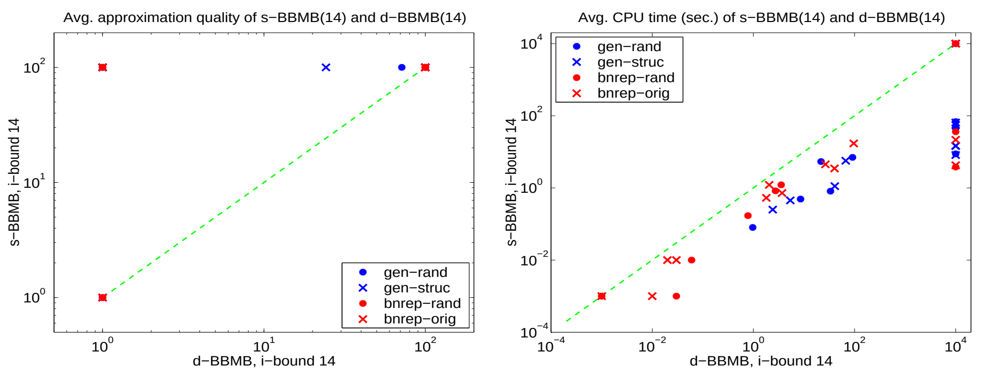
(b) Runtime to find quasi-optimal solution

Figure 9.1: Performance comparison of BnB algorithms s-BBMB and d-BBMB, both with i-bound 6. Average approximation quality (a) and average runtime to find a quasi-optimal solution (b). The algorithms were run 25 times for 100 CPU seconds each.

time to reach quasi-optimal solutions. As can be seen in the complete results in Tables B.15 on page 155 and B.16 on page 156, these algorithms often can proof optimality of the solutions they find in virtually the same time they need to find them. For few instances, such as the `link` network, some of the algorithms find provably optimal solution qualities, but are not able to proof their optimality within the time bound; and for one randomly generated instance, `z400v3iw20-rand`, s-BBMB(14) finds the quasi-optimal solution quality, but none of the algorithms can proof its optimality.

We start by comparing the two related algorithms s-BBMB and d-BBMB. In Figure 9.1, we demonstrate that with low i-bounds, such as 6, d-BBMB outperforms s-BBMB in terms of achieved solution quality and also finds solutions with positive probability for many more instances. Nevertheless, when both algorithms find quasi-optimal solutions, s-BBMB is faster in almost all cases. When both algorithms employ a higher and better-performing i-bound, such as 14, the picture changes and s-BBMB becomes clearly superior. It solves some instances to quasi-optimality for which d-BBMB does not even find solutions with positive probability, and it is faster on all instances both of the algorithms solve. For 9 instances, for which d-BBMB does not find solutions with positive probability, it finds optimal solutions and proofs their optimality.

We now move on to determine the optimal i-bound for s-BBMB on the instances we study. In Figure 9.3 on page 112, we demonstrate that s-BBMB(10) clearly outperforms s-BBMB(6); s-BBMB(10) generally finds solutions of much

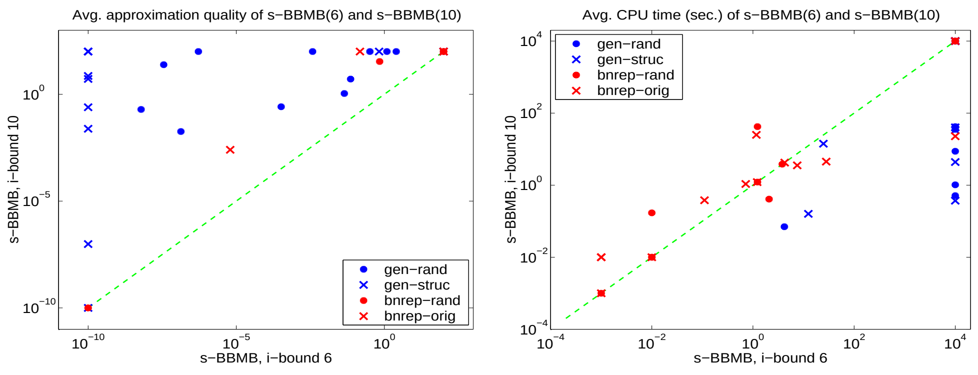


(a) Approximation quality, $q_0 = 10^0$ (b) Runtime to find quasi-optimal solution

Figure 9.2: Performance comparison of BnB algorithms s-BBMB and d-BBMB, both with i-bound 14. For most instances, both algorithms either find optimal solutions or fail completely (yielding zero-probability solutions). Average approximation quality (a) and average runtime to find a quasi-optimal solution (b). The algorithms were run 25 times for 100 CPU seconds each.

higher quality and for a variety of randomly generated instances with structured CPTs for which s-BBMB(6) finds only solutions of probability zero, it finds solutions with positive probability or even solutions it can proof to be optimal. Furthermore, s-BBMB(10) solves many instances to optimality s-BBMB(6) does not solve. In terms of runtime for instances which are solved by both algorithms, s-BBMB(6) and s-BBMB(10) perform comparably.

In Figure 9.4 on page 113, we compare i-bounds 10 and 14 for the s-BBMB algorithm. Recall that algorithm s-BBMB with i-bound *ib* first executes the approximate Mini-Buckets algorithm with i-bound *ib*, MB(*ib*), in order to compute a static heuristic function to guide the subsequent Branch and Bound search. For i-bounds as high as 14, this is often not feasible. For example, for instance `munin1`, the initial call of MB(14) does not terminate within 100 CPU seconds. For all randomly generated instances with maximal induced width 20 and maximal domain size 6, our limited memory of 4 GB renders MB(14) infeasible. However, for instances, for which its initial call to MB(14) succeeds, s-BBMB(14) generally performs better than s-BBMB(10), solving many instances s-BBMB(10) cannot solve. Interestingly, whenever s-BBMB(14) finds a non-zero probability for an instance, it solves the instance to quasi-optimality in our experiments. In terms of runtime, the algorithms perform comparably. We conclude from this comparison that there is no clear winner between s-BBMB(10) and s-BBMB(14): when MB(14) is feasi-



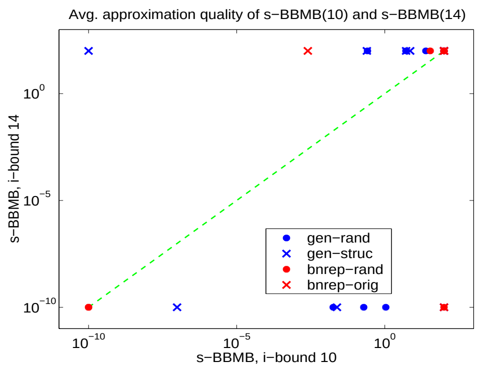
(a) Approximation quality, $q_0 = 10^{-10}$ (b) Runtime to find quasi-optimal solution

Figure 9.3: Performance comparison of i-bounds 6 and 10 for s-BBMB. Average approximation quality (a) and average runtime to find a quasi-optimal solution (b). The algorithms were run 25 times for 100 CPU seconds each.

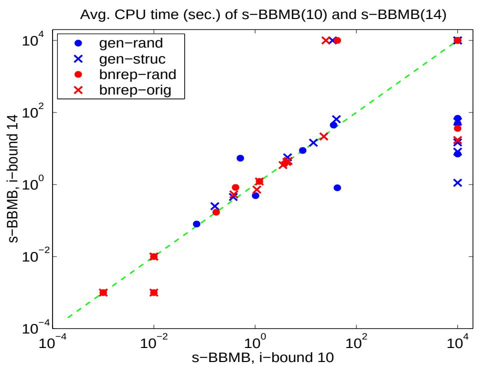
ble, s-BBMB(14) performs better; otherwise, s-BBMB(10) obviously is the better algorithm. The fact that the runtime of Mini-Buckets with a fixed ordering and i-bound can be estimated fairly accurately before the algorithm is executed suggests a combination of these approaches.

In Figure 9.5 on the next page, we show that s-BBMB(14) dominates s-BBMB(18). The only difference between the performance of these algorithms is that s-BBMB(18) breaks due to memory constraints for even more instances, namely on all but one randomly generated instance with maximal induced width 20, and that it is slower than s-BBMB(14) on the one remaining generated instance with induced width 20, as well as for network `link-rand` which also has a relatively high induced width.

We now compare algorithm Anytime MB against s-BBMB with the two best-performing i-bounds 10 and 14. In Figure 9.6 on page 114, we demonstrate that, despite the fact that both of the algorithms are based on the Mini-Buckets algorithm, the performance of Anytime MB and s-BBMB(10) is not highly correlated. The same holds true for the comparison of Anytime MB and s-BBMB(14) in Figure 9.7 on page 115. Either way, there are some instances which are solved and proved optimal by one of the algorithms whereas the other one does not even find a solution with positive probability. In terms of solution quality, Anytime MB performs much better on the structured instances from `bnrep`, whereas s-BBMB with i-bound 10 or 14 is faster for instances from `gen`. Since the different ways of employing the Mini-Buckets heuristic in Anytime MB and s-BBMB (as well as in

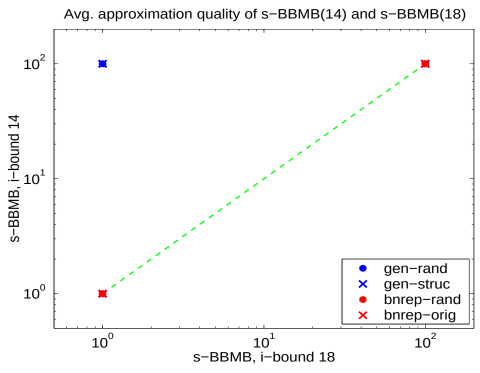


(a) Approximation quality, $q_0 = 10^{-10}$

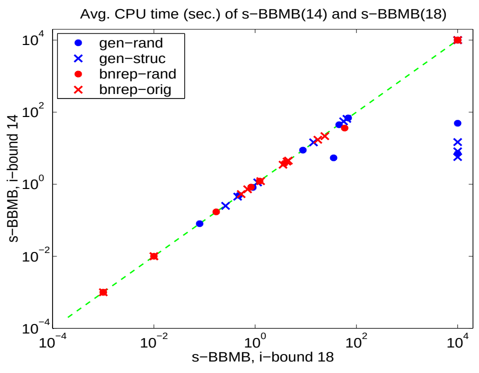


(b) Runtime to find quasi-optimal solution

Figure 9.4: Performance comparison of i-bounds 10 and 14 for s-BBMB. Average approximation quality (a) and average runtime to find a quasi-optimal solution (b). The algorithms were run 25 times for 100 CPU seconds each.

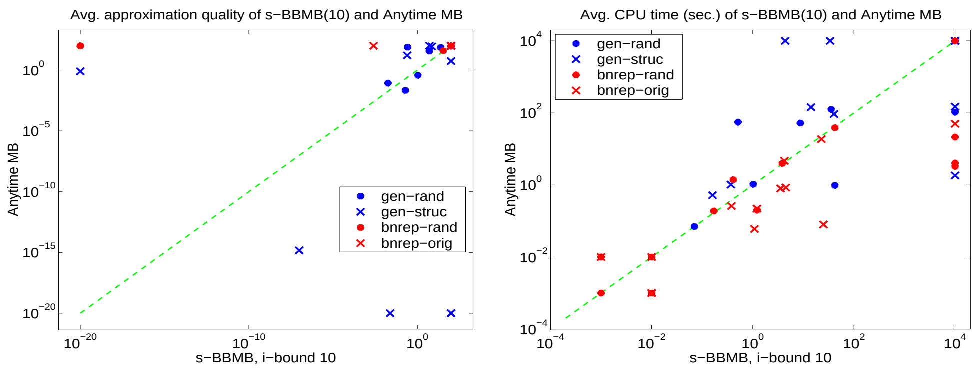


(a) Approximation quality, $q_0 = 10^0$



(b) Runtime to find quasi-optimal solution

Figure 9.5: Performance comparison of i-bounds 14 and 18 for s-BBMB. Average approximation quality (a) and average runtime to find a quasi-optimal solution (b). The algorithms were run 25 times for 100 CPU seconds each.



(a) Approximation quality, $q_0 = 10^{-20}$ (b) Runtime to find quasi-optimal solution

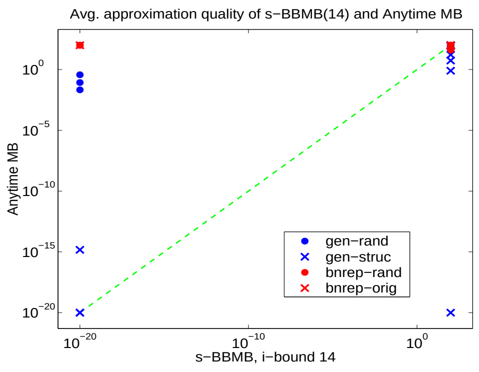
Figure 9.6: Performance comparison of algorithms Anytime MB and s-BBMB(10). Average approximation quality (a) and average runtime to find a quasi-optimal solution (b). The algorithms were run 25 times for 100 CPU seconds each.

d-BBMB) yield this very different behaviour, a combination of the two approaches might be worthwhile. One could, for example, employ a BnB search phase in each iteration of Anytime MB, or start BBMB with a small i-bound and increase it during the search if some online criterion suggests that the currently used bounds are too weak to efficiently support the search.

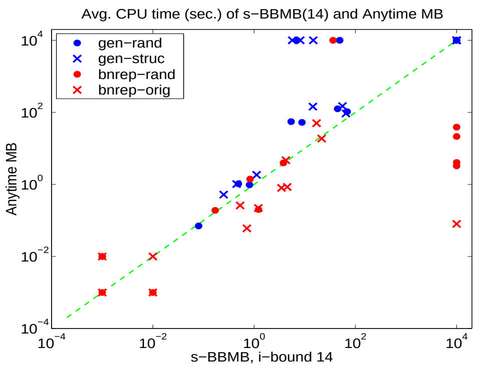
9.2 Comparison of Best-Performing Algorithms

In this section, we compare the best-performing systematic search algorithms s-BBMB(10), s-BBMB(14), and Anytime MB with our novel SLS algorithms ILS and GLS$^+$, and with our hybrid of ILS, GLS$^+$, and MB$^*(10^5)$. We simply refer to this latter hybrid algorithm as HYBRID. Tables 9.3 on the facing page and 9.4 on page 116 summarize the performance of the various algorithms for problem sets `bnrep` and `gen`, respectively. We observe a consistently very strong performance of HYBRID which we mainly attribute to its MB* component for instances from problem set `bnrep` and to its GLS$^+$ component for instances from problem set `gen`. HYBRID's calls of MB* with increasing *size*-bound closely resemble Anytime MB, the only difference being that not the number of variables in a Mini-Bucket is bounded but the size of the largest Mini-Bucket potential.

In the following, we compare HYBRID to each of the other algorithms, showing that it outperforms all of them for most instances. We start by evaluating the performance of HYBRID against each of its constituents. Figure 9.8 demonstrates



(a) Approximation quality, $q_0 = 10^{-20}$



(b) Runtime to find quasi-optimal solution

Figure 9.7: Performance comparison of algorithms Anytime MB and s-BBMB(14). Average approximation quality (a) and average runtime to find a quasi-optimal solution (b). The algorithms were run 25 times for 100 CPU seconds each.

Statistics	GLS$^+$ default	ILS default	s-BBMB ib=2	ib=6	ib=10	ib=14	ib=18	MB anytime	HYBRID default
Avg. quality	89.05	82.41	39.25	76.96	82.09	80.77	80.77	97.69	98.38
Avg. runtime	37.61	41.64	166.43	32.57	29.07	28.36	29.55	5.96	13.04
Successful runs	491/650	471/650	10/26	20/26	21/26	21/26	21/26	25/26	625/650
Instances solved	21/26	19/26	10/26	20/26	21/26	21/26	21/26	25/26	25/26
Instances > 0	26/26	26/26	19/26	23/26	23/26	21/26	21/26	26/26	26/26
# amongst best	3	4	3	8	5	7	5	10	2

Table 9.3: Summary statistics for best-performing algorithms on problem set `bnrep`. All algorithms were run 25 times for 100 CPU seconds each. Summary of Table B.17 on page 157.

Statistics	GLS$^+$ default	ILS default	s-BBMB ib=2	ib=6	ib=10	ib=14	ib=18	MB anytime	HYBRID default
Avg. quality	89.35	63.22	0.03	12.70	51.82	75.84	55.01	53.90	80.75
Avg. runtime	49.96	115.85	∞	713.77	115.08	52.79	107.42	80.93	72.52
Successful runs	430/600	299/600	0/24	3/24	12/24	18/24	13/24	9/24	384/600
Instances solved	18/24	15/24	0/24	3/24	12/24	18/24	13/24	9/24	18/24
Instances > 0	24/24	24/24	13/24	16/24	23/24	18/24	13/24	21/24	24/24
# amongst best	11	1	0	0	3	5	3	0	2

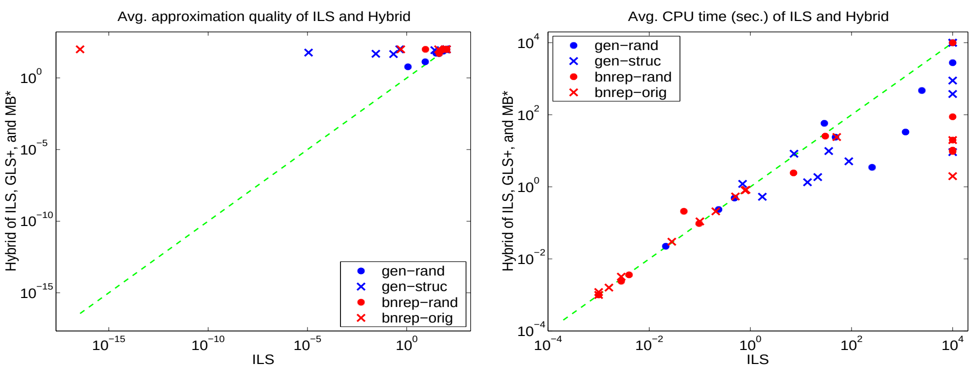
Table 9.4: Summary statistics for best-performing algorithms on problem set `gen`. All algorithms were run 25 times for 100 CPU seconds each. Summary of Table B.18 on page 158.

that HYBRID clearly outperforms ILS. It finds instantiations of higher or equal probability for every instance, is considerably faster in finding quasi-optimal solutions and solves many instances to quasi-optimality ILS cannot solve.

Figure 9.9 on the facing page shows that in the case of GLS$^+$, the case is not that simple. For few structured instances HYBRID finds solutions of higher quality, and for many randomly generated ones, GLS$^+$ finds slightly better solutions. In terms of runtime to find quasi-optimal solutions, GLS$^+$ outperforms HYBRID in most cases, usually by a factor between two and ten; only for few instances, HYBRID is faster. The only advantage of HYBRID over GLS$^+$ is that it finds optimal solutions for a number of large structured networks with low induced width which GLS$^+$ cannot solve. For example, the `diabetes` network with 403 free variables of average domain size 11.34 poses an impossible challenge to GLS$^+$, whereas due to its low induced width of 6 and the resulting low induced size of 8.58×10^7, HYBRID can quickly solve it to optimality with its MB* component.

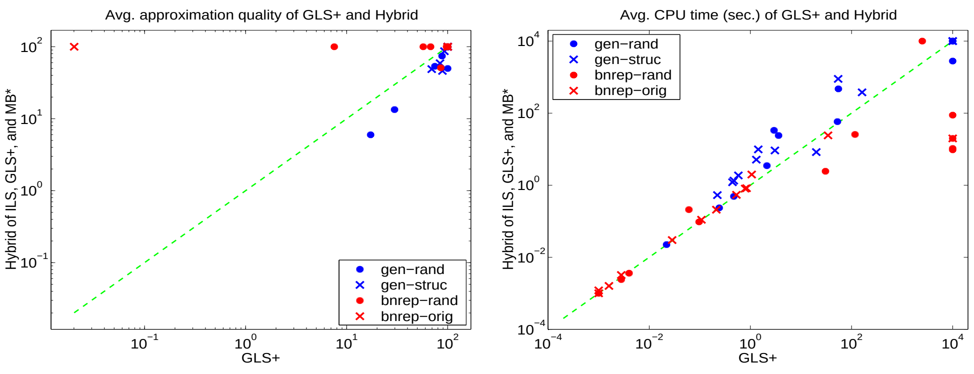
When compared to Anytime MB (see Figure 9.10 on page 118), HYBRID clearly performs better on the randomly generated instances from `gen`. For most instances from `bnrep`, Anytime MB is about three times faster than HYBRID, but for some other instances, HYBRID is also faster. We primarily atrribute is better behaviour on these instances to its GLS$^+$ component.

In Figures 9.11 on page 119 and 9.12 on page 119, we demonstrate that HYBRID outperforms s-BBMB with i-bounds 10 and 14, respectively. It especially solves more instances to quasi-optimality and is faster on average. The improvements in achieved solution quality HYBRID yields over s-BBMB(14) (see Figure 9.12(a) on page 119) appear to be somewhat smaller than in the case of s-BBMB(10) (see Figure 9.11(a) on page 119). This is an artifcact stemming from the fact that s-BBMB(14) breaks for most of the instances for which s-BBMB(10)



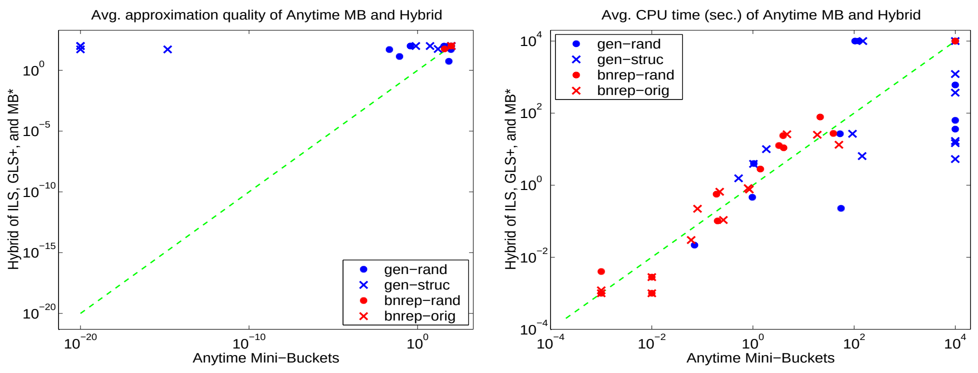
(a) Approximation quality (b) Runtime to find quasi-optimal solution

Figure 9.8: Performance comparison of pure ILS and HYBRID. Average approximation quality (a) and average runtime to find a quasi-optimal solution (b). The algorithms were run 25 times for 100 CPU seconds each.



(a) Approximation quality (b) Runtime to find quasi-optimal solution

Figure 9.9: Performance comparison of pure GLS$^+$ and HYBRID. Average approximation quality (a) and average runtime to find a quasi-optimal solution (b). The algorithms were run 25 times for 100 CPU seconds each.



(a) Approximation quality, $q_0 = 10^{-20}$ (b) Runtime to find quasi-optimal solution

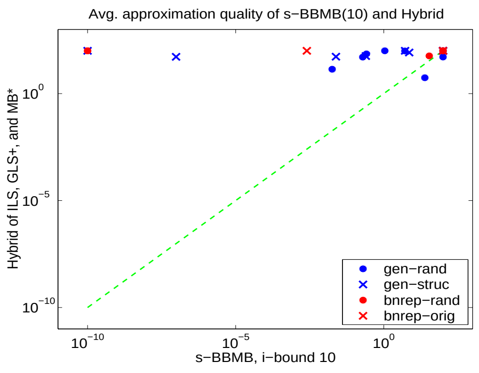
Figure 9.10: Performance comparison of Anytime MB and HYBRID. Average approximation quality (a) and average runtime to find a quasi-optimal solution (b). The algorithms were run 25 times for 100 CPU seconds each.

finds suboptimal qualities, yielding probability zero in these cases. Thus, the top left data point in Figure 9.12(a) on the facing page subsumes 7 instances for which s-BBMB(14) breaks and HYBRID finds quasi-optimal solutions.

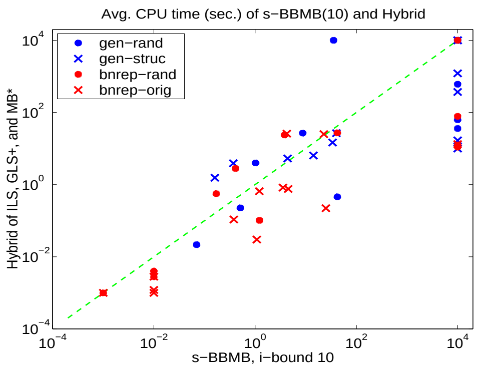
9.3 Scaling Studies

In this section, we study how s-BBMB, Anytime MB, and our novel SLS algorithms ILS and GLS$^+$ scale with important instance characteristics, such as the number of variables, the maximal domain size of the variables, the maximal degree of any node in the independence graph, and the maximal induced width of the independence graph. For each of these instance characteristics, a separate experiment is carried out in which all other characteristics are kept fix, thereby isolating the effects of the characteristic of interest. Since algorithm HYBRID alternates independent phases of ILS, GLS$^+$ and Anytime MB, its scaling behaviour is completely determined by the scaling of these components. We concentrate on studying the behaviour of each component and omit the hybrid algorithm from the scaling studies.

Using BNGenerator [IC02, IC03], we created a new set of problem instances for each experiment. We defined a number of possible values for the characteristic and generated 10 networks for each such value, leading to problem sets with 50 to 100 instances. When generating the problem set for an instance characteristic, we chose the rest of the characteristics such that we can find an optimal solution

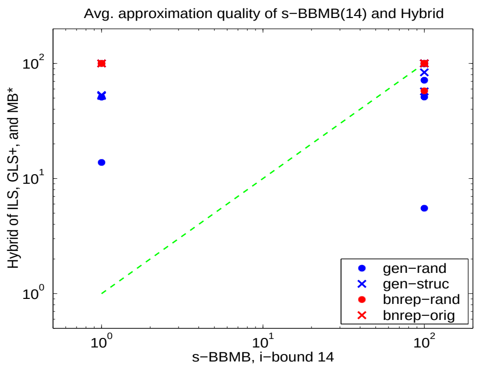


(a) Approximation quality, $q_0 = 10^{-20}$

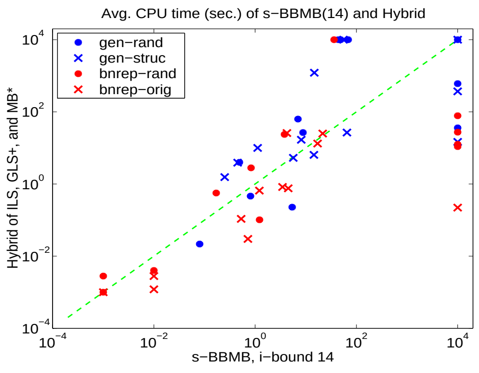


(b) Runtime to find quasi-optimal solution

Figure 9.11: Performance comparison of s-BBMB(10) and HYBRID. Average approximation quality (a) and average runtime to find a quasi-optimal solution (b). The algorithms were run 25 times for 100 CPU seconds each.



(a) Approximation quality, $q_0 = 10^0$



(b) Runtime to find quasi-optimal solution

Figure 9.12: Performance comparison of s-BBMB(14) and HYBRID. The top left data point in (a) subsumes 7 instances for which s-BBMB(14) breaks and HYBRID finds quasi-optimal solutions. Average approximation quality (a) and average runtime to find a quasi-optimal solution (b). The algorithms were run 25 times for 100 CPU seconds each.

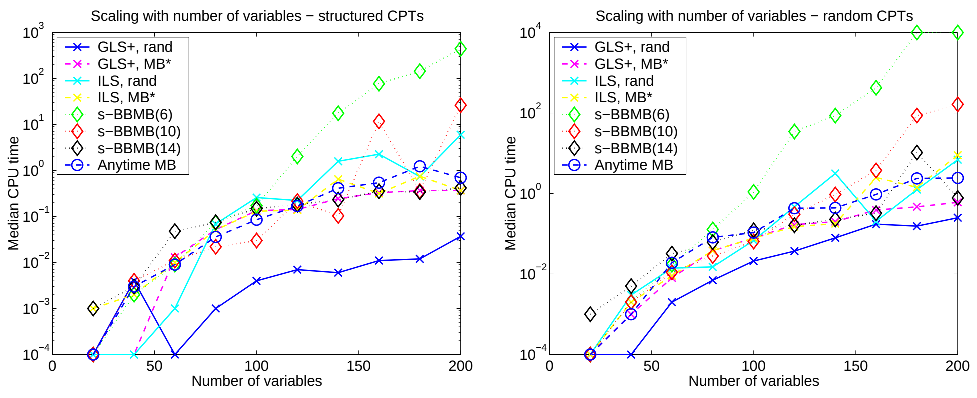
and proof its optimality for all instances. In all experiments except the one varying the induced width, we kept the maximal induced width at a moderate value of 15.[3] For each characteristic and each possible value of the characteristic, we ran all algorithms once for 100 CPU seconds on the applicable 10 instances and report the average runtime. If none of the 10 instances was solved, we plot the value 10000 instead.

In our scaling experiments, we observe that instances with random CPTs are much harder to solve for GLS than instances with structured CPTs – the difference in average solution time for otherwise equal characteristics is as large as up to an order of magnitude. For algorithm s-BBMB, this effect can also be observed in some of our scaling experiments. We therefore split every scaling experiment in one experiment for random CPTs and one for structured CPTs; the results are always reported side-by-side.

Before we describe the individual scaling experiments, we report a common characteristic of all of them. In order to be able to compute optimal solutions for all the instances in the scaling experiments, we had to keep them rather easy. For the very fast GLS$^+$ algorithm, this means that it seldomly takes longer than a few hundred milliseconds to solve any of the problems. The MB$^*(10^5)$ initialization usually takes some longer only for the initialization, such that for the small and easy instances studied in our scaling experiments, GLS$^+$ is consistently much faster when simply employing a random initialization.

Figure 9.13 on the next page shows that instances become considerably harder to solve as the number of variables increases. In this experiment, GLS$^+$ is the best performing algorithm for instances with structured and random CPTs, but for structured ones the margin by which it outperforms the second-best algorithm, s-BBMB(14), is much larger. For s-BBMB, i-bound 6 performs clearly inferior to higher i-bounds and also shows a very poor scaling with an increasing number of variables: for small instances with 20 and 40 variables, s-BBMB(6) is only between zero and one orders of magnitude slower than GLS$^+$, but for larger instances with 200 variables it is four orders of magnitude slower than GLS$^+$ for structured CPTs and cannot solve any instance with random CPTs. For the also poorly scaling s-BBMB(10), this effect is not as dramatic but still considerable; it leads to GLS$^+$ outperforming s-BBMB(10) by over two orders of magnitude for larger instances. In previous experiments (not reported here) we carried out to compare our SLS algorithms against d-BBMB, we found that the factor by which GLS$^+$ outperforms

[3]We also carried out some preliminary experiments with a maximal induced width of 30, but the performance of the systematic algorithms in these experiments was too poor for meaningful scaling studies (often, they did not solve any instance even for small values of the instance characteristic studied in an experiment).

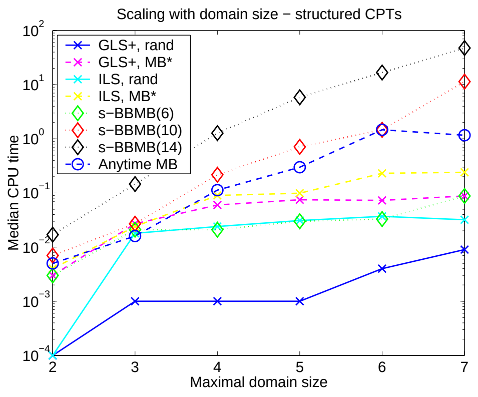


(a) Structured CPTs (b) Random CPTs

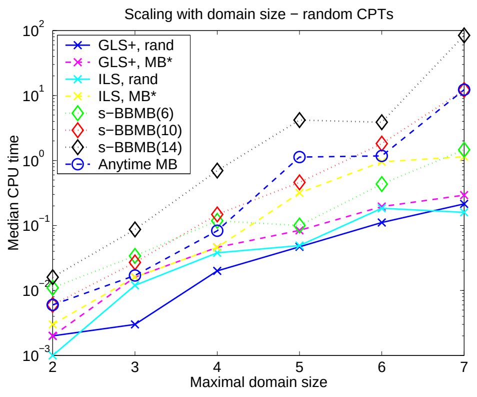
Figure 9.13: Scaling of solution time with varying number of variables for instances with domain size 2, degree 4, and induced width 15. Results for the previous systematic algorithms Anytime MB, d-BBMB(2), d-BBMB(6), and d-BBMB(10); and for our algorithms ILS and GLS$^+$, both with random and MB$^*(10^5)$ initialization. Instances with structured CPTs (a) and randomly generated CPTs (b).

all variants of d-BBMB grows considerably with an increasing number of variables. While for small problems GLS$^+$ was only about 1.5 orders of magnitude faster than d-BBMB(6) and d-BBMB(10), for larger problems the difference was up to three orders of magnitude. Since s-BBMB(10) scales roughly comparably to our SLS algorithms, we conclude that d-BBMB also scales worse than s-BBMB(10).

For increasing maximal domain sizes, Figure 9.14 on the following page shows that our SLS algorithms scale much better than Anytime MB, s-BBMB(10), and s-BBMB(14), whereas s-BBMB(6) shows a scaling behaviour comparable to our SLS algorithms and outperforms the other s-BBMB variants. Algorithm s-BBMB(2) is not reported in the Figure, but it yielded the worst performance in this as well as in all other experiments. GLS$^+$ continues to be the best-performing algorithm in this scaling experiment. In the case of structured CPTs, the factor by which it outperforms s-BBMB(14) increases from two orders of magnitude for domain size 2 to four orders of magnitude for maximal domain size 7; for random CPTs, the factor reaches from one to three orders of magnitude. The margin by which GLS$^+$ outperforms Anytime MB grows by approximately one order of magnitude from domain size 2 to maximal domain size 7. In previous experiments, we found d-BBMB with low i-bounds of 2 or 6 to scale comparably to our SLS algorithms, which suggests that it scales better with increasing domain sizes than s-BBMB does with high i-bounds. However, with higher i-bounds, d-BBMB did not scale



(a) Structured CPTs



(b) Random CPTs

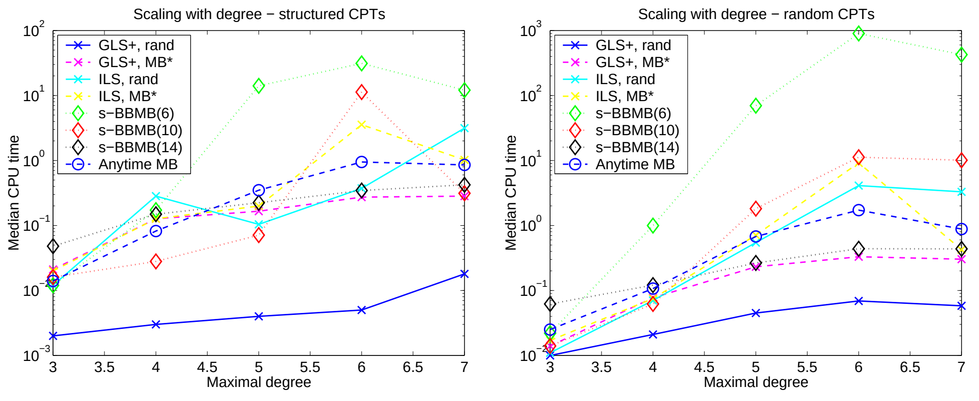
Figure 9.14: Scaling of solution time with varying maximal domain size for instances with 50 variables, degree 4, and induced width 15. Results for the previous systematic algorithms Anytime MB, d-BBMB(2), d-BBMB(6), and d-BBMB(10); and for our algorithms ILS and GLS$^+$, both with random and MB$^*(10^5)$ initialization. Instances with structured CPTs (a) and randomly generated CPTs (b).

well either. We attribute this to the fact that the computation of the Mini-Buckets heuristic (which both s-BBMB and d-BBMB employ) grows more complex with an increasing domain size and fixed i-bound.[4]

For an increasing degree of the independence graph, in Figure 9.15 on the next page we observe a very poor scaling behaviour of s-BBMB with low i-bounds. The performance of s-BBMB(14) scales comparably to the one of our SLS algorithms, whereas s-BBMB(10) and especially s-BBMB(6) exhibit much inferior scaling behaviour. Compared to s-BBMB(14), s-BBMB(6) is approximately three times faster in solving problems with maximal degree 3, whereas for maximal degree 7, it is over an order of magnitude slower for structured CPTs and almost three orders of magnitude slower for random CPTs. Once more, GLS$^+$ is the best-performing algorithm in this experiment. It is consistently about one order of magnitude faster than s-BBMB(14), slightly more for structured CPTs and slighty less for random CPTs. Previous experiments we carried out for d-BBMB showed it to degrade rather rapidly with increasing degree for all i-bounds.

Finally, for an increasing induced width of the independence graph, we observe that algorithm Anytime MB degrades rapidly. We expected exactly this behaviour

[4]Recall that MB(*ib*) computes potentials of size up to $|D|^{ib}$, where by $|D|$ we denote the domain size of the variables. With high i-bounds, this number quickly becomes very large yielding long runtimes or even infeasibility for large domain sizes.

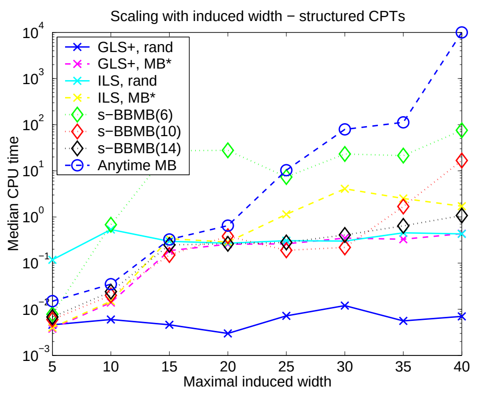


(a) Structured CPTs (b) Random CPTs

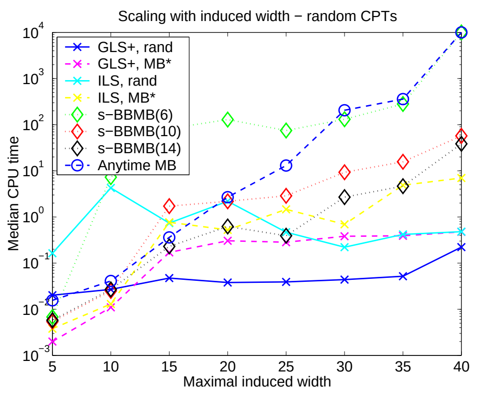
Figure 9.15: Scaling of solution time with varying maximal node degree for instances with 100 variables, domain size 2, and induced width 15. Results for the previous systematic algorithms Anytime MB, d-BBMB(2), d-BBMB(6), and d-BBMB(10); and for our algorithms ILS and GLS$^+$, both with random and MB$^*(10^5)$ initialization. Instances with structured CPTs (a) and randomly generated CPTs (b).

since in the worst case Anytime MB builds potentials that are exponential in the networks's induced width. While for small induced widths of 5, Anytime MB and the (again) best-performing algorithm GLS$^+$ have very similar runtimes, for induced widths of 40, the differences in runtime are between 3.5 and five orders of magnitude for random and structured CPTs, respectively. While s-BBMB is not affected by the increasing induced width as badly as Anytime MB, its runtime still scales much worse than the one of GLS$^+$ and ILS with random initialization. This effect is stronger for small i-bounds. For instances with maximal induced width 5, s-BBMB(6) and GLS$^+$ perform almost identically, whereas for induced width 40, s-BBMB(6) is about three orders of magnitude slower than GLS$^+$. For s-BBMB(10), this speed difference is two and three orders of magnitude for structured and random CPTs, respectively; and for s-BBMB(14), the difference is approximately two orders of magnitude.

For our SLS algorithms with initialization MB$^*(10^5)$, the MB$^*(10^5)$ initialization quickly finds optimal solutions for the low induced widths 5 and 10. For larger induced widths, due to its bounded size, its runtime remains almost constant which can be seen by the fact that for induced widths higher than 15, the runtime of ILS and GLS$^+$ with MB$^*(10^5)$ initialization only increases by a small margin in the case of ILS and not at all for GLS$^+$.



(a) Structured CPTs



(b) Random CPTs

Figure 9.16: Scaling of solution time with varying induced width for instances with 100 variables, domain size 2, and maximal degree 5. Results for the previous systematic algorithms Anytime MB, d-BBMB(2), d-BBMB(6), and d-BBMB(10); and for our algorithms ILS and GLS$^+$, both with random and MB$^*(10^5)$ initialization. Instances with structured CPTs (a) and randomly generated CPTs (b).

Previous experiments with d-BBMB showed that, like s-BBMB, this algorithm also scales poorly with increasing induced width. For structured CPTs, GLS$^+$ outperforms d-BBMB(6) and d-BBMB(10) by approximately one order of magnitude for networks with induced width 5 and by three orders of magnitude for networks with induced width 40. For random CPTs, these factors are approximately 1.5 and three, respectively.

In summary, our scaling experiments indicate that

- for the type of instances considered here, GLS$^+$ is always the best-performing algorithm and shows the best scaling behaviour for all instance characteristics;
- s-BBMB(10) and especially s-BBMB(6) scale poorly with an increasing number of variables;
- Anytime MB, s-BBMB(10), and especially s-BBMB(14) scale poorly with an increasing domain size;
- s-BBMB(6) scales poorly with an increasing degree;
- s-BBMB scales poorly with an increasing induced width, especially for small i-bounds; and

- Anytime MB scales very poorly with an increasing induced width.

The scaling experiments in this section especially underline the strong performance of our SLS algorithms for instances with high induced width. Furthermore, they show that for every i-bound, the previous exact algorithms s-BBMB and d-BBMB exhibit poor scaling behaviour with some instance characteristic. For example, while s-BBMB(14) seems to scale well with number of variables for otherwise unchanged instance characteristics, it scales poorly with domain size. On the other extreme, s-BBMB(6) scales poorly with degree of the graph and number of variables. A compromise is s-BBMB(10), which, however, scales worse than our new SLS algorithms with every instance characteristic we studied.

In conclusion of this chapter, we note that for structured instances with many variables and low induced width, systematic algorithms like Branch and Bound Algorithms or Anytime Mini-Buckets show very strong performance, but that their performance degrades rapidly with increasing induced width that causes the Mini-Buckets heuristic with feasible i-bounds to become exceedingly inaccurate.

Chapter 10

Conclusions and Future Directions

In this thesis, we have developed and studied novel Stochastic Local Search (SLS) algorithms for solving the Most Probable Explanation (MPE) problem in graphical models. We use Bayesian networks as a motivating example and also for our experimental evaluation, but the novel algorithms we introduce are applicable to general graphical models, including Markov Random Fields and factor graphs.

SLS algorithms have been applied to the MPE problem before, but the best-performing SLS algorithms G+StS [KD99b] and GLS [Par02] have been shown to be outperformed by systematic search algorithms that combine Branch-and-Bound with the Mini-Buckets heuristic [MKD03]. However, none of these previous SLS algorithms pays sufficient attention to such important concerns as algorithmic complexity per search step, search stagnation, and thorough parameter tuning. In this thesis, we removed these shortcomings of previous SLS algorithms for MPE, improving their efficiency by up to 4 orders of magnitude for non penalty-based algorithms (cf. Figure 8.2 on page 93) and 6 orders of magnitude for penalty-based algorithms (cf. Figure 8.6 on page 96). As demonstrated in the experimental evaluation of Chapter 8, this enormous speedup is due to a number of separate improvements. These include our novel caching schemes detailed in Chapter 6, a thorough parameter tuning described in Chapter 7, and the initialization with our new Mini-Buckets variant MB$^*(10^5)$ (cf. Section 4.2 on page 27).

For penalty-based algorithms, we demonstrated how the objective function can be integrated into the evaluation function that is guiding the previously best-performing SLS algorithm GLS. This improvement led to a new GLS variant we call GLS$^+$. All other components being equal, GLS$^+$ substantially outperforms the previously best-performing SLS algorithm for MPE, GLS (cf. Figure 8.9 on page 100); it is the the new state-of-the-art SLS algorithm for MPE. By introducing the first Iterated Local Search (ILS) algorithm we also significantly improved the state-of-the-art in non penalty-based SLS algorithms. Again, all other components being equal, ILS outperforms the previously best-performing non penalty-based algorithm G+StS [KD99b] by several orders of magnitude (cf. Figure 8.4 on page 95). The performance of the penalty- and non penalty-based algorithms GLS$^+$ and ILS is not tightly coupled and we combined both of these algorithms and our Mini-Buckets variant MB* in a new hybrid algorithm. This hybrid simply loops through independent executions of MB*, ILS, and GLS$^+$. MB* is called with increasing bounds on maximal cluster size, and in each iteration, ILS and GLS$^+$ are executed for the time MB* required to run to completion.

In Chapter 9, we demonstrated improved overall performance of this hybrid algorithm when compared to our SLS algorithms ILS and GLS$^+$, the systematic search algorithm s-BBMB with optimal i-bound, and Anytime MB, an anytime version of Mini-Buckets. For all these algorithms, we also studied the scaling of solution time with a number of important instance characteristics, namely the number of variables, domain size, degree, and induced width of the network's independence graph. When compared to our new algorithms, s-BBMB with a small i-bound of 6 scales poorly with an increasing number of variables, degree, and induced width, whereas for higher i-bounds, such as 10 and 14, it scales poorly with an increasing domain size and induced width. Anytime MB scales poorly with an increasing domain size and especially poorly with an increasing induced width.

Based on our experiments, we expect our novel SLS algorithms to outperform the current state-of-the-art in MPE solving for a large number of problem domains, especially for problem instances which exhibit large induced widths and domain sizes. However, we have not yet compared our algorithms to loopy belief propagation [Pea88], an algorithm that has recently gained much popularity [MMC98, MWJ99]. Another interesting algorithm we plan to study is Graph cuts [BVZ01]. We intend to carry out a comparison with both of these algorithms in the near future and to broaden our experimental analysis to include MPE instances from fault diagnosis in computer networks [RBM02a], computer vision [TF03], medical diagnosis [Hec90], and probabilistic decoding [DR03].

Other interesting directions we see for future work in the MPE domain can be grouped into four categories:

A better characterization of the behaviour of different MPE algorithms
In this thesis, we have studied the scaling behaviour of various MPE algorithms with a number of important instance characteristics, but many more interesting studies remain to be done. A systematic study of the search landscape for MPE instances from different domains may aid our understanding of which features make instances hard for SLS algorithms. Furthermore, a study of empirical hardness distributions both for SLS and exact algorithms

may shed some light on the relation between the two approaches for solving MPE.

It would also be worthwhile to further study the approach of encoding MPE instances into weighted Max-SAT problems with real-valued weights, especially in order to evaluate how the results GLS for Max-SAT obtains on these encodings compare to the results we achieve with GLS and GLS$^+$ on the original MPE instances.

Further improvements of the currently best-performing SLS algorithms

Since GLS$^+$ performs best in most of our experiments, a worthwhile direction for future research is to improve its performance further. We expect that by an adaptive version of GLS$^+$ that controls the weighting of the penalties in the evaluation function significant speedups can be gained over the current version of GLS$^+$. More advanced parameter tuning (for example by employing ParamILS) for these GLS variants may also lead to substantial improvements. We further expect that significant improvements of ILS are possible based on a more detailed study of its search behaviour.

A combination of various approaches for MPE solving

One straight-forward combination of the various approaches we studied in this thesis is to employ the solution qualities found in local search algorithms as lower bounds in systematic search algorithms based on Branch-and-Bound.

There is also a variety of possibilities to combine approaches in order to improve the local search algorithms themselves. Since our novel SLS algorithms ILS and GLS$^+$ move through the search space in a very different fashion, it may be possible to fruitfully combine them in a single algorithm that alternates phases of ILS and GLS$^+$ on a single search trajectory. While such an approach holds great promise, so far we have not implemented it due to the limited scope of this thesis. One particular possibility of combining the two approaches is to employ GLS or GLS$^+$ in the pertubation phase of ILS. This idea is based on the intuition that ILS algorithms usually benefit the most from pertubations which transform a very high-quality solution into a very good starting point for a local search, and that executing GLS/GLS$^+$ for a number of steps could yield such a pertubation. GLS/GLS$^+$ in turn shows a lack of greediness which might be complemented nicely by the additional local search phases such a hybrid of ILS and GLS/GLS$^+$ would perform. We would like to implement this is future work, but we expect many subtleties to need consideration when constructing such a hybrid algorithm.

Another promising idea we had was to use the strong MB* algorithm in the pertubation phase of ILS. We fixed a subset of the variables as evidence and let MB* determine the value of the rest of the variables. In order to guarantee not to end up in the same local optimum again, before fixing the evidence, we flipped a few of these evidence variables. In preliminary experiments, this approach performed well in terms of the number of iterations needed to find the optimal solution, but due to the considerable complexity of MB*, executing it in every local minimum lead to very slow runtimes. Fixing central variables in the network, or even cutsets of variables, may substantially improve the performance of this approach.

Similarly, one could perform a local search on a restricted subset $\mathbf{W}$ of the variables, which could be a cutset or simply a set of central variables in the network that, when conditioned on, yield low induced width of the remaining network. The extension of $\mathbf{W}$'s instantiation $\mathbf{w}$ to a complete variable instantiation $\mathbf{v}$ could in this case be done by a subsidiary exact or approximate MPE algorithm, such as variable elimination or loopy belief propagation. This approach would much resemble Kask's and Dechter's GSAT+CC algorithm for SAT [KD96] which executes a local search on a cutset of SAT variables and optimizes the rest of the variables with a subsidiary specialized local search algorithm for trees. We see much promise for such an approach in the domain we studied. Due to the modularity of graphical models, their variables are typically much sparser connected than in traditional SAT problems. It may thus suffice to condition on a small number of variables in order to render exact inference for all remaining variables feasible.

An extension of our algorithms to more general problems

As mentioned in Section 2.3 on page 11, our SLS algorithms are trivially extensible to the more general problem of finding the M most likely instantiations (M-MPE). To our best knowledge, this has not been done so far, and with adapted versions of our new algorithms we expect to significantly enhance the state-of-the-art for this important problem.

Another promising generalization of our new algorithms is to solve the MAP problem (cf. Section 2.3 on page 11) by applying local search on the MAP variables only and compute the probabiliy of their instantiations with a subsidiary inference algorithm. Such a generalization could, for example, follow the approach taken in [PD01], where loopy belief propagation is used to approximate the inference task.

A final very promising line of future research is not directly related to solving the MPE problem, but deals with the problem of automated parameter tuning.

Since we needed to tune many parameters for our new ILS algorithm, we developed ParamILS, a novel approach that performs an Iterated Local Search in parameter configuration space, searching for the best-performing parameter configuration. Future work in this research area may involve the application of statistical tests in ParamILS in order to prevent an unnecessarily detailed study of the algorithm's performance with inferior parameter settings. Once this has been implemented, we can afford to perform a substantial number of runs per well-performing parameter configuration and instance. With this improved evaluation function and its current performance, we expect ParamILS to become a general automated procedure for parameter tuning, applicable in any research area where the need to tune or fit many discrete or discretized parameters arises.

To conclude this thesis, let us take a bird's eye view on what we have done. Our motivation for studying SLS algorithms for solving MPE were two-fold. Firstly, due to the close similarity between MPE and Max-SAT [Par02] and the state-of-the-art status of SLS algorithms for Max-SAT, we expected a much greater potential for SLS algorithms than reported in [MKD03], and we wanted to study the reasons for this discrepancy; by speeding up the state-of-the-art in SLS algorithms for MPE by many orders of magnitude, we resolved the disagreement. Our second motivation was to construct an efficient anytime algorithm for MPE solving in Bayesian networks with induced widths prohibitive for exact algorithms. As demonstrated in our experiments, our new SLS algorithms achieve this goal: they are inherently anytime, yield high-quality solutions very quickly, and scale smoothly with induced width.

In the course of our research, we found that there are many other interesting and promising application areas for algorithms with these characteristics, most interesting of which we find early computer vision. In this research area, the development of efficient algorithms is very important since hard problems have to be solved in an online fashion within fractions of seconds. A variety of problems in this domain can be cast as solving the MPE problem in a grid-structured pairwise Markov Random Field, a graphical network with high induced width and often comparably large domain sizes [FH04]. The next step in our research will be to apply our algorithms to these problems, and to compare and possibly combine them with current state-of-the-art algorithms in the field.

Appendix A

Parameter Tuning by Iterated Local Search in Configuration Space

The problem of tuning some algorithm's parameters for maximal performance on a class of problem instances is ubiquitious in the design and empirical analysis of algorithms. Especially in the development of high-performing SLS algorithms, sometimes considerable effort is required to find a default parameter configuration that yields high and stable performance across all or at least most instances of a problem set. If there are only few parameters, it is often the easiest way to allow a certain number of values for each parameter and then try each combination, or *configuration*, of these parameter values; this approach is called *full factorial design*. However, if an algorithm has too many parameters, full factorial design is not feasible since the number of possible configurations grows exponentially in the number of parameters. One method researchers often use in this case is to start with the configuration that intuitively makes most sense to them or that is the "easiest" in some sense. From this initial configuration on, they often change one parameter at a time and keep the resulting configuration if performance improves, ending their optimization if no change of a single parameter yields an improvement anymore.[1]

Having read Chapter 3 about the basic principles of Stochastic Local Search, this should sound familiar. Although most researchers practicing the approach we just described are probably not familiar with the principles of Local Search algorithms, they actually perform a manual Local Search themselves. The search space for this Local Search is the space of possible configurations, the objective function is a configuration's performance (however it be defined), the initial configuration is the one the researcher starts with, the neighbourhood is a 1-exchange neigh-

[1] In this thesis, we have applied this approach ourselves in order to tune the parameters of GLS. We have also applied it in previous work [HTH02, AFH$^+$04].

bourhood (changing one parameter at a time), and the search strategy is simple first-improvement. In order not to spend our valuable time in the lab basically executing a manual Local Search in configuration space, we developed an automated general procedure that does the job.

Given an algorithm A, its parameter set and possible discrete values for each parameter, a set of problem instances $\mathcal{S}$, and the runtime t for which optimal performance shall be reached, it searches for the configuration with optimal overall performance on the instances of $\mathcal{S}$ when run for time t. We will refer to algorithm A with parameter configuration C and runtime t as $A(C, t)$.

Probably the hardest part to automate is the automatic evaluation of different parameter configurations. In the tool we developed, methods to compute and compare objective function values are accepted as parameters; these functions are called for the evaluation of configurations during the optimization process such that the tool be generally applicable. In our particular parameter optimization for MPE algorithms, we compute the objective function value for a configuration C by executing a predefined number of runs R of $A(C, t)$ on each problem instance $S \in \mathcal{S}$. For each run, the approximation quality is recorded and whether it found an optimal solution or not. The objective function value is then a tuple $(r, \overline{qual})$, representing the ratio of successful runs, and the average approximation quality, respectively.[2] We define a configuration C_1 with objective function value $(r_1, \overline{qual}_1)$ to be better than a configuration C_2 with objective function value $(r_2, \overline{qual}_2)$ if and only if $(r_1/r_2 + \overline{qual}_1/\overline{qual}_2)/2 > 1$.[3]

Explicating the aforementioned approach to parameter tuning as a Local Search in configuration space and automating it does not only help our laziness but also suggests the use of more powerful SLS schemes, such as Iterated Local Search (ILS). When employing only one pass of greedy hill-climbing in configuration space, we end up in a local optimum, from where no single parameter change yields an improvement. However, due to the interaction between various parameters, it may well be that changing two or more parameters at a time results in an improvement. For our implementation, we chose the general framework of ILS (cf. Algorithm scheme 3.2 on page 21). The initialization procedure simply uses the algorithm's default parameters if available and otherwise initializes the parameters at random. The basic local search is greedy hill-climbing, the acceptance criterion accepts a new parameter configuration if and only if its objective function value is

[2]If most of the problem instances in a problem class can be solved efficiently by an algorithm, the runtime it needs to find the solution is another very useful measure. If most instances are solved, one can also for example employ 0.95 quantiles to detect and prevent search stagnation.

[3]Due to statistical variance in the results, we experienced that combining r and $\overline{qual}$ yielded a more stable measure than first comparing r and only in case of ties judging by $\overline{qual}$.

at least as good as the previous one, and the pertubation changes three parameter values at random. We call this automated ILS in configuration space *ParamILS* and detail it in Algorithm A.1 on page 137.

We employed ParamILS in order to tune the parameters of our ILS algorithm and describe its results in Section 7.5 on page 79. In short, ParamILS found a very well-performing parameter configuration of ILS's 9 parameters already in its first iteration although 5 of the 9 parameters needed to be flipped to reach it from the initial simple parameter configuration. This parameter configuration remained the best for six iterations after which ParamILS found a better-performing one which differs in the setting of 4 parameter values from the previous one. We terminated the procedure after an additional 18 iterations in which it did not improve its best parameter configuration anymore. In total, only three local optima different from the two very well-performing ones were encountered. Due to the costly evaluation of search states, the 25 iterations of ParamILS already took one CPU week.[4] In contrast to the five CPU months a full factorial design would have taken, this was still feasible.

In order to evaluate ParamILS further, we also tested its performance for tuning the parameters of G+StS and GLS. These algorithms employ only two parameters each, such that a full factorial approach to determine the global optimum is feasible. In the case of G+StS, starting with a randomly chosen initial configuration $\langle cf, np\rangle = \langle 10, 20\rangle$, ParamILS found the optimal configuration $\langle cf, np\rangle = \langle 2, 40\rangle$ in the first iteration. For GLS, starting with the original configuration $\langle \rho, N_\rho\rangle = \langle 200, 0.8\rangle$ from [Par02], it found the parameter configuration $\langle \rho, N_\rho\rangle = \langle 10000, 0.99\rangle$ within the first iteration which yielded much better results than the original configuration $\langle \rho, N_\rho\rangle = \langle 200, 0.8\rangle$. When the evaluation of a parameter configuration is done by executing exactly one run on every instance with a fixed seed, $\langle \rho, N_\rho\rangle = \langle 10000, 0.99\rangle$ actually yields slightly better results than $\langle \rho, N_\rho\rangle = \langle 10000, 0.99\rangle$; it is indeed the globally optimal parameter configuration in this case. However, these results are subject to high statistical variance since only one run is executed per instance. When a parameter configuration is evaluated by running the algorithm for 25 runs on each instance, the globally optimal parameter configuration $\langle \rho, N_\rho\rangle = \langle 200, 0.999\rangle$ performs slightly better than $\langle \rho, N_\rho\rangle = \langle 10000, 0.99\rangle$.

While these results demonstrate that ParamILS quickly finds globally optimal parameter configurations w.r.t. its objective function, it also highlights the currently biggest weakness of ParamILS. In order to quickly find optimal parameter configurations with as little computational overhead as possible, ParamILS needs to imple-

[4]Although one CPU week is rather long, we could parallelize all runs necessary for evaluating each of the search states, such that with eight idle CPUs it took less than one day.

ment a method to focus on high-performing parameter configurations and carry out a greater number of runs for these to reduce the statistical variance inherent when dealing with randomized algorithms.

One alternative to employing ParamILS would be to use a racing algorithm to remove configurations that are significantly worse than others based on statistical tests [BSPV02]. However, racing algorithms require every single parameter configuration to be run at least on a few instances, and this quickly becomes infeasible for larger numbers of algorithm parameters as they frequently occur in SLS algorithms. Moreover, during the development of a new algorithm, one often allows substantially more parameters than in the final version. This is a clear drawback for racing algorithms for which an additional parameter with d possible values results in a d-times slower optimization.[5]

Finally, another advantage of ParamILS is its similarity to the human approach of parameter tuning. The results of a few iterations already yield very good intuitions about well-performing parameter combinations and when introducing new algorithm parameters we routinely ran a quick experiment employing shorter runs of $t = 5$ CPU seconds. We also discretized the continuous parameters of ILS using the intuition gained from shorter runs of ParamILS that allowed for a much larger number of discrete values than our final experiments.

A very promising line of future research is the combination of Local Search and statistical methods. ParamILS could, for example, stop the evaluation of a parameter configuration once it has gained sufficient evidence for the fact that it performs significantly better or worse than the one it is being compared to. This may solve the problem that ParamILS currently uses equivalent CPU time for the evaluation of each parameter configuration's performance, regardless of how well the configuration performs. In optimization, however, the exact performance of an inferior parameter configuration is of much less interest than the exact performance of a new promising parameter configuration. The quality of the latter estimate may decide whether we actually identify the best parameter configuration, whereas the quality of the former estimate has barely any consequences at all. Hence, the allocation of resources should mirror our interest. Achieving this goal would yield a great speedup of ParamILS since there exist many clearly suboptimal parameter configurations for whose exact evaluation ParamILS currently spends most of its time.

[5]Although for our final algorithm, an optimization using racing algorithms would have been of comparable CPU cost as our optimization with ParamILS, we could not employ this approach during the development of ILS. We considered using racing algorithms, but our version of ILS at that time had five additional non-boolean parameters, rendering racing algorithms infeasible but still allowing for an efficient optimization with ParamILS.

ParamILS

This algorithm performs an Iterated Local Search in the space of possible parameter configurations. For initialization, default parameters are used (or, if no default parameters are available, the parameters are initialized at random), the basic local search is greedy hill-climbing, the pertubation randomly changes 3 parameters, and the acceptance criterion only accepts better or equal parameter configurations.

Function *cmp*(C_1, C_2) compares two configurations C_1 and C_2 by executing $A(C_1, t)$ and $A(C_2, t)$ with a fixed seed for all instances in problem set $\mathcal{S}$ and comparing the results as described in the text. Previous results for the same configuration, time, and instance are reused in order to prevent multiple executions of identical experiments.

```
Input: Algorithm $A$, parameters $\mathcal{P}$, parameter domain $D_P$ for each $P \in \mathcal{P}$, time per run $t$,
       set $\mathcal{S}$ of problem instances, function cmp comparing two parameter configurations,
       time $t_{opt}$ available for parameter optimization.
Output: Parameter configuration $C$ with best overall performance of $A(C,t)$ found for
        problem set $\mathcal{S}$.
1  if default parameter setting available then
2      Init $C$ with default parameters
3  else
4      Init $C$ with random parameters
5  $C_{ils} \leftarrow LocalSearch(C)$
6  while runtime $< t_{opt}$ do
7      $C \leftarrow Pertubation(C_{ils}, 3)$
8      $C \leftarrow LocalSearch(C)$
9      if $cmp(C, C_{ils}) \geq 0$ then $C_{ils} \leftarrow C$
10 Function LocalSearch ($C$)
11 begin
12     $C_{best} \leftarrow C$
13     repeat
14         $C_{last} \leftarrow C_{best}$
15         for $P \in \mathcal{P}$ and $p \in D_P$ do
16             $C_{tmp} \leftarrow C_{last}$ with $P$ changed to $p$.
17             if $cmp(C_{tmp}, C_{best}) > 0$ then $C_{best} \leftarrow C_{tmp}$
18     until $C_{best} = C_{last}$
19     return $C_{best}$
20 end
21 Function Pertubation ($C$, strength)
22 begin
23     for $i = 1..strength$ do
24         $P \leftarrow$ Draw random parameter from $\mathcal{P}$.
25         $p \leftarrow$ Draw random value from $D_P \setminus \{\tilde{p}\}$, where $\tilde{p}$ is $P$'s current value.
26         $C \leftarrow C$ with $P$ changed to $p$.
27     return $C$
28 end
```

Appendix B

Detailed experimental results

In this appendix, we present the complete results for all experiments for which summary tables were presented in this thesis. For SLS algorithms, we provide three values per instance:

Solved gives the number of runs which found quasi-optimal solution quality for this instance, as well as the total number of runs.

Quality avg gives the average approximation quality at the time the algorithm was terminated. If no solution with positive probability was found, we print the symbol "(-)".

Time gives the average runtime to solve the instance, that is, the total runtime divided by the number of successful runs. If none of the runs was successful, we print the symbol "∞".

For deterministic algorithms only one run is carried out. In order to be able to compare results for many algorithms in one table, we compress all information about this run into one column. Whenever a run finds the quasi-optimal solution, we provide information in the format "*find*/*proof*", where *find* is the time the algorithm took to find the quasi-optimal solution and *proof* is the time it needed to proof optimality.[1] If the algorithm does not proof optimality, we print the information as "*find*/-".

For instances, for which a deterministic algorithm does not reach quasi-optimal solution quality, we provide information in the format "(*qual*)", where *qual* is the approximation quality the run reached. If no solution with positive probability is found, we print "(-)".

[1]Note that these times coincide very often. This happens, for example, if a tight upper bound on solution quality has already been found before the solution is found.

In each table, for each instance we highlight the entry of the best-performing algorithm. Like in the summary tables used throughout the thesis, “best-performing” is defined in terms of the percentage of successful runs, in the case of ties by the average approximation quality, and again in the case of ties by the average runtime. If all these measures are identical for several algorithms, they are all amongst the best-performing algorithms for this instance.

For the summary tables provided throughout the thesis, for exact algorithms the average runtime on a set of instances is computed using the time they needed to find quasi-optimal solutions per instance, not the time they needed to proof optimality.

	G+StS														
	cf = 1.5			cf = 2			cf = 5			cf = 10			cf = 100		
Instance	Solved	Quality avg	Time	Solved	Quality avg	Time	Solved	Quality avg	Time	Solved	Quality avg	Time	Solved	Quality avg	Time
alarm	25/25	100.00	0.0008	25/25	100.00	0.0008	25/25	100.00	0.0008	**25/25**	**100.00**	**0.0004**	25/25	100.00	0.0008
alarm-rand	**25/25**	**100.00**	**0.0008**	**25/25**	**100.00**	**0.0008**	**25/25**	**100.00**	**0.0008**	**25/25**	**100.00**	**0.0008**	**25/25**	**100.00**	**0.0008**
barley	**25/25**	**100.00**	**0.65**	25/25	100.00	0.87	25/25	100.00	4.25	25/25	100.00	4.25	25/25	100.00	4.24
barley-rand	25/25	100.00	0.10	25/25	100.00	0.10	**25/25**	**100.00**	**0.10**	25/25	100.00	0.10	25/25	100.00	0.10
diabetes	**0/25**	**6e-30**	∞	0/25	6e-32	∞	0/25	3e-39	∞	0/25	5e-45	∞	0/25	(-)	∞
diabetes-rand	**0/25**	**76.28**	∞	0/25	74.49	∞	0/25	70.29	∞	0/25	69.62	∞	0/25	67.88	∞
hailfinder	25/25	100.00	0.001	25/25	100.00	0.002	25/25	100.00	0.001	25/25	100.00	0.002	**25/25**	**100.00**	**0.0008**
hailfinder-rand	25/25	100.00	0.003	**25/25**	**100.00**	**0.002**	**25/25**	**100.00**	**0.002**	**25/25**	**100.00**	**0.002**	**25/25**	**100.00**	**0.002**
insurance	**25/25**	**100.00**	**0.0008**	**25/25**	**100.00**	**0.0008**	**25/25**	**100.00**	**0.0008**	**25/25**	**100.00**	**0.0008**	**25/25**	**100.00**	**0.0008**
insurance-rand	**25/25**	**100.00**	**0.0004**	**25/25**	**100.00**	**0.0004**	**25/25**	**100.00**	**0.0004**	**25/25**	**100.00**	**0.0004**	**25/25**	**100.00**	**0.0004**
link	**0/25**	**0.50**	∞	**0/25**	**0.50**	∞	**0/25**	**0.50**	∞	**0/25**	**0.50**	∞	**0/25**	**0.50**	∞
link-rand	**0/25**	**34.77**	∞	0/25	33.34	∞	0/25	30.37	∞	0/25	28.00	∞	0/25	21.43	∞
mildew	**25/25**	**100.00**	**0.03**	25/25	100.00	0.03	25/25	100.00	0.03	25/25	100.00	0.03	25/25	100.00	0.03
mildew-rand	25/25	100.00	0.004	**25/25**	**100.00**	**0.003**	25/25	100.00	0.004	25/25	100.00	0.004	25/25	100.00	0.004
munin1	25/25	100.00	0.21	**25/25**	**100.00**	**0.21**	25/25	100.00	0.21	25/25	100.00	0.21	25/25	100.00	0.21
munin1-rand	0/25	28.58	∞	**0/25**	**29.73**	∞	0/25	28.17	∞	0/25	26.90	∞	0/25	28.52	∞
munin2	25/25	100.00	0.74	**25/25**	**100.00**	**0.74**	25/25	100.00	0.74	25/25	100.00	0.74	25/25	100.00	0.81
munin2-rand	**0/25**	**97.43**	∞	**0/25**	**97.43**	∞	**0/25**	**97.43**	∞	**0/25**	**97.43**	∞	**0/25**	**97.43**	∞
munin3	25/25	100.00	0.82	25/25	100.00	1.03	25/25	100.00	0.82	25/25	100.00	0.82	**25/25**	**100.00**	**0.81**
munin3-rand	**0/25**	**61.26**	∞	0/25	59.87	∞	0/25	58.79	∞	0/25	57.56	∞	0/25	57.31	∞
munin4	0/25	0.003	∞	0/25	0.005	∞	**0/25**	**0.005**	∞	0/25	0.005	∞	0/25	0.004	∞
munin4-rand	0/25	8.15	∞	**0/25**	**8.16**	∞	0/25	8.02	∞	0/25	8.07	∞	0/25	7.91	∞
pigs	**25/25**	**100.00**	**0.10**	25/25	100.00	0.10	25/25	100.00	0.10	25/25	100.00	0.11	25/25	100.00	0.10
pigs-rand	**24/25**	**99.82**	**37.37**	23/25	99.63	44.03	15/25	97.42	105.07	10/25	96.06	204.25	0/25	80.29	∞
water	**25/25**	**100.00**	**0.003**	**25/25**	**100.00**	**0.003**	**25/25**	**100.00**	**0.003**	**25/25**	**100.00**	**0.003**	**25/25**	**100.00**	**0.003**
water-rand	25/25	100.00	0.05	25/25	100.00	0.05	**25/25**	**100.00**	**0.05**	25/25	100.00	0.05	25/25	100.00	0.05

Table B.1: Results of G+StS with initialization MB$^*(10^5)$, noise probability $np = 40$, and varying cutoff factor cf on problem set `bnrep`. All algorithms are run 25 times for 100 CPU seconds each. Summarized in Table 7.1 on page 69.

	G+StS														
	cf = 1.5			cf = 2			cf = 5			cf = 10			cf = 100		
Instance	Solved	Quality avg	Time	Solved	Quality avg	Time	Solved	Quality avg	Time	Solved	Quality avg	Time	Solved	Quality avg	Time
z100v3d5iw10-rand	25/25	100.00	0.02	25/25	100.00	0.02	**25/25**	**100.00**	**0.02**	**25/25**	**100.00**	**0.02**	**25/25**	**100.00**	**0.02**
z100v3d5iw10-struc	16/25	82.62	96.88	22/25	93.03	42.47	25/25	100.00	13.91	25/25	100.00	13.62	**25/25**	**100.00**	**13.49**
z100v3d5iw20-rand	25/25	100.00	0.23	**25/25**	**100.00**	**0.23**	25/25	100.00	0.24	25/25	100.00	0.24	25/25	100.00	0.24
z100v3d5iw20-struc	25/25	100.00	7.38	**25/25**	**100.00**	**4.70**	25/25	100.00	5.03	25/25	100.00	5.04	25/25	100.00	5.01
z100v6d5iw10-rand	1/25	62.69	2490.57	2/25	67.69	1203.71	4/25	75.92	552.76	2/25	68.85	1170.21	**4/25**	**78.79**	**581.28**
z100v6d5iw10-struc	0/25	26.74	∞	0/25	29.60	∞	0/25	21.81	∞	0/25	31.18	∞	**1/25**	**26.83**	**2469.79**
z100v6d5iw20-rand	0/25	32.27	∞	0/25	32.14	∞	0/25	31.97	∞	**0/25**	**34.21**	∞	0/25	32.33	∞
z100v6d5iw20-struc	0/25	9.30	∞	1/25	21.64	2478.36	**1/25**	**24.34**	**2468.81**	1/25	14.60	2442.19	0/25	22.70	∞
z200v3d5iw10-rand	**0/25**	**74.26**	∞	0/25	71.62	∞	0/25	66.04	∞	0/25	63.72	∞	0/25	60.49	∞
z200v3d5iw10-struc	**25/25**	**100.00**	**9.75**	25/25	100.00	11.50	25/25	100.00	22.22	16/25	88.11	92.92	3/25	65.89	797.01
z200v3d5iw20-rand	0/25	19.55	∞	0/25	24.95	∞	**0/25**	**31.44**	∞	0/25	27.95	∞	0/25	28.92	∞
z200v3d5iw20-struc	0/25	0.77	∞	**0/25**	**0.89**	∞	0/25	0.60	∞	0/25	0.55	∞	0/25	0.44	∞
z200v6d5iw10-rand	**0/25**	**1.36**	∞	0/25	1.12	∞	0/25	0.96	∞	0/25	0.79	∞	0/25	0.45	∞
z200v6d5iw10-struc	**0/25**	**0.008**	∞	0/25	0.007	∞	0/25	0.004	∞	0/25	0.001	∞	0/25	2e-06	∞
z200v6d5iw20-rand	0/25	0.13	∞	**0/25**	**0.48**	∞	0/25	0.47	∞	0/25	0.36	∞	0/25	0.44	∞
z200v6d5iw20-struc	**0/25**	**3e-06**	∞	0/25	2e-06	∞	0/25	4e-07	∞	0/25	3e-07	∞	0/25	8e-07	∞
z400v3d5iw10-rand	**25/25**	**100.00**	**0.46**	25/25	100.00	0.49	25/25	100.00	0.48	25/25	100.00	0.48	25/25	100.00	0.48
z400v3d5iw10-struc	**2/25**	**44.56**	**1205.84**	2/25	43.82	1205.45	0/25	25.57	∞	0/25	18.02	∞	0/25	6.82	∞
z400v3d5iw20-rand	**0/25**	**19.52**	∞	0/25	17.02	∞	0/25	14.30	∞	0/25	12.77	∞	0/25	9.38	∞
z400v3d5iw20-struc	**0/25**	**1.40**	∞	0/25	1.18	∞	0/25	0.73	∞	0/25	0.69	∞	0/25	0.45	∞
z400v6d5iw10-rand	**0/25**	**0.02**	∞	0/25	0.01	∞	0/25	0.01	∞	0/25	0.008	∞	0/25	0.001	∞
z400v6d5iw10-struc	**0/25**	**4e-22**	∞	0/25	3e-22	∞	0/25	2e-22	∞	0/25	2e-23	∞	0/25	3e-23	∞
z400v6d5iw20-rand	**0/25**	**0.56**	∞	0/25	0.47	∞	0/25	0.33	∞	0/25	0.26	∞	0/25	0.05	∞
z400v6d5iw20-struc	**0/25**	**7e-09**	∞	0/25	5e-09	∞	0/25	8e-10	∞	0/25	3e-10	∞	0/25	3e-13	∞

Table B.2: Results of G+StS with initialization MB$^*(10^5)$, noise probability *np* $= 40$, and varying cutoff factor *cf* on problem set `gen`. All algorithms are run 25 times for 100 CPU seconds each. Summarized in Table 7.2 on page 69.

Instance	G+StS																	
	np = 5			np = 10			np = 20			np = 30			np = 40			np = 50		
	Solved	Quality avg	Time	Solved	Quality avg	Time	Solved	Quality avg	Time	Solved	Quality avg	Time	Solved	Quality avg	Time	Solved	Quality avg	Time
alarm	25/25	100.00	0.0008	25/25	100.00	0.0008	25/25	100.00	0.0008	25/25	100.00	0.0008	**25/25**	**100.00**	**0.0004**	25/25	100.00	0.0008
alarm-rand	**25/25**	**100.00**	**0.0008**	**25/25**	**100.00**	**0.0008**	**25/25**	**100.00**	**0.0008**	**25/25**	**100.00**	**0.0008**	**25/25**	**100.00**	**0.0008**	**25/25**	**100.00**	**0.0008**
barley	25/25	100.00	2.39	25/25	100.00	1.10	**25/25**	**100.00**	**0.71**	25/25	100.00	0.77	25/25	100.00	1.13	25/25	100.00	1.45
barley-rand	25/25	100.00	0.10	25/25	100.00	0.10	**25/25**	**100.00**	**0.10**	25/25	100.00	0.10	25/25	100.00	0.10	25/25	100.00	0.10
diabetes	0/25	(-)	∞	0/25	4e-26	∞	**0/25**	**1e-25**	∞	0/25	3e-27	∞	0/25	8e-32	∞	0/25	4e-44	∞
diabetes-rand	0/25	75.01	∞	0/25	78.34	∞	**0/25**	**78.35**	∞	0/25	75.62	∞	0/25	72.99	∞	0/25	71.33	∞
hailfinder	25/25	100.00	0.002	25/25	100.00	0.002	25/25	100.00	0.002	**25/25**	**100.00**	**0.001**	**25/25**	**100.00**	**0.001**	25/25	100.00	0.002
hailfinder-rand	25/25	100.00	0.003	**25/25**	**100.00**	**0.002**	**25/25**	**100.00**	**0.002**	**25/25**	**100.00**	**0.002**	25/25	100.00	0.003	**25/25**	**100.00**	**0.002**
insurance	**25/25**	**100.00**	**0.0008**	**25/25**	**100.00**	**0.0008**	**25/25**	**100.00**	**0.0008**	**25/25**	**100.00**	**0.0008**	**25/25**	**100.00**	**0.0008**	**25/25**	**100.00**	**0.0008**
insurance-rand	**25/25**	**100.00**	**0.0004**	**25/25**	**100.00**	**0.0004**	**25/25**	**100.00**	**0.0004**	**25/25**	**100.00**	**0.0004**	**25/25**	**100.00**	**0.0004**	**25/25**	**100.00**	**0.0004**
link	**0/25**	**0.50**	∞	**0/25**	**0.50**	∞	**0/25**	**0.50**	∞	**0/25**	**0.50**	∞	**0/25**	**0.50**	∞	**0/25**	**0.50**	∞
link-rand	0/25	28.16	∞	0/25	31.41	∞	0/25	32.23	∞	0/25	32.45	∞	0/25	32.49	∞	**0/25**	**33.37**	∞
mildew	25/25	100.00	0.03	25/25	100.00	0.03	25/25	100.00	0.03	25/25	100.00	0.03	**25/25**	**100.00**	**0.03**	25/25	100.00	0.03
mildew-rand	**25/25**	**100.00**	**0.004**	25/25	100.00	0.004	25/25	100.00	0.004	**25/25**	**100.00**	**0.004**	25/25	100.00	0.004	25/25	100.00	0.004
munin1	**25/25**	**100.00**	**0.20**	25/25	100.00	0.20	25/25	100.00	0.21	25/25	100.00	0.20	25/25	100.00	0.21	25/25	100.00	0.20
munin1-rand	0/25	22.54	∞	0/25	23.44	∞	0/25	30.02	∞	**0/25**	**30.79**	∞	0/25	29.74	∞	0/25	27.23	∞
munin2	25/25	100.00	0.81	**25/25**	**100.00**	**0.74**	25/25	100.00	0.75	25/25	100.00	0.75	25/25	100.00	0.81	25/25	100.00	0.81
munin2-rand	**0/25**	**97.43**	∞	**0/25**	**97.43**	∞	**0/25**	**97.43**	∞	**0/25**	**97.43**	∞	**0/25**	**97.43**	∞	**0/25**	**97.43**	∞
munin3	25/25	100.00	0.82	**25/25**	**100.00**	**0.76**	25/25	100.00	0.82	25/25	100.00	0.81	25/25	100.00	1.02	25/25	100.00	0.81
munin3-rand	0/25	59.82	∞	0/25	61.99	∞	**0/25**	**64.01**	∞	0/25	60.33	∞	0/25	61.16	∞	0/25	57.31	∞
munin4	0/25	0.003	∞	0/25	0.003	∞	**0/25**	**0.005**	∞	0/25	0.004	∞	0/25	0.004	∞	0/25	0.003	∞
munin4-rand	0/25	8.00	∞	0/25	8.07	∞	0/25	8.15	∞	**0/25**	**8.22**	∞	0/25	8.13	∞	0/25	8.01	∞
pigs	**25/25**	**100.00**	**0.10**	25/25	100.00	0.10	25/25	100.00	0.10	**25/25**	**100.00**	**0.10**	25/25	100.00	0.10	25/25	100.00	0.10
pigs-rand	8/25	93.16	235.08	13/25	96.47	122.11	23/25	99.39	38.12	22/25	99.50	41.27	**23/25**	**99.66**	**47.61**	22/25	99.55	46.07
water	**25/25**	**100.00**	**0.003**	25/25	100.00	0.003	**25/25**	**100.00**	**0.003**	25/25	100.00	0.003	25/25	100.00	0.003	25/25	100.00	0.003
water-rand	25/25	100.00	0.05	25/25	100.00	0.05	**25/25**	**100.00**	**0.04**	25/25	100.00	0.05	25/25	100.00	0.05	25/25	100.00	0.05

Table B.3: Results of G+StS with initialization MB$^*(10^5)$, cutoff factor $cf = 2$, and varying noise probability *np* on problem set `bnrep`. All algorithms are run 25 times for 100 CPU seconds each. Summarized in Table 7.3 on page 69.

	G+StS																	
	np = 5			np = 10			np = 20			np = 30			np = 40			np = 50		
Instance	Solved	Quality avg	Time	Solved	Quality avg	Time	Solved	Quality avg	Time	Solved	Quality avg	Time	Solved	Quality avg	Time	Solved	Quality avg	Time
z100v3d5iw10-rand	**25/25**	**100.00**	**0.02**	25/25	100.00	0.02	25/25	100.00	0.02	**25/25**	**100.00**	**0.02**	25/25	100.00	0.02	25/25	100.00	0.02
z100v3d5iw10-struc	2/25	42.03	1200.68	7/25	55.56	310.84	9/25	65.11	214.93	22/25	93.40	38.79	**25/25**	**100.00**	**29.76**	22/25	93.57	47.60
z100v3d5iw20-rand	25/25	100.00	0.23	**25/25**	**100.00**	**0.23**	25/25	100.00	0.24	25/25	100.00	0.24	25/25	100.00	0.24	25/25	100.00	0.24
z100v3d5iw20-struc	8/25	81.46	253.81	22/25	97.53	48.49	25/25	100.00	13.99	25/25	100.00	7.41	25/25	100.00	5.86	**25/25**	**100.00**	**4.63**
z100v6d5iw10-rand	0/25	14.90	∞	0/25	31.90	∞	1/25	49.86	2417.84	0/25	61.07	∞	**3/25**	**64.93**	**787.52**	0/25	56.41	∞
z100v6d5iw10-struc	0/25	5.56	∞	0/25	26.90	∞	**1/25**	**22.85**	**2426.35**	0/25	32.28	∞	0/25	20.40	∞	0/25	14.17	∞
z100v6d5iw20-rand	0/25	8.58	∞	0/25	16.83	∞	0/25	25.39	∞	0/25	29.10	∞	**0/25**	**32.08**	∞	0/25	27.39	∞
z100v6d5iw20-struc	0/25	0.003	∞	0/25	0.32	∞	0/25	5.49	∞	0/25	16.46	∞	**2/25**	**24.59**	**1208.47**	0/25	8.75	∞
z200v3d5iw10-rand	0/25	63.28	∞	0/25	70.23	∞	**0/25**	**73.54**	∞	0/25	68.71	∞	0/25	73.29	∞	0/25	68.48	∞
z200v3d5iw10-struc	25/25	100.00	19.99	**25/25**	**100.00**	**13.85**	25/25	100.00	16.80	25/25	100.00	14.83	25/25	100.00	17.94	25/25	100.00	19.58
z200v3d5iw20-rand	0/25	7.07	∞	0/25	11.39	∞	0/25	14.57	∞	0/25	20.92	∞	0/25	26.63	∞	**0/25**	**31.50**	∞
z200v3d5iw20-struc	0/25	0.27	∞	0/25	0.38	∞	0/25	0.76	∞	0/25	0.63	∞	**0/25**	**0.85**	∞	0/25	0.61	∞
z200v6d5iw10-rand	0/25	1.18	∞	0/25	1.45	∞	**0/25**	**1.65**	∞	0/25	1.53	∞	0/25	1.22	∞	0/25	0.89	∞
z200v6d5iw10-struc	0/25	0.008	∞	0/25	0.01	∞	**0/25**	**0.01**	∞	0/25	0.009	∞	0/25	0.008	∞	0/25	0.003	∞
z200v6d5iw20-rand	0/25	0.002	∞	0/25	0.008	∞	0/25	0.05	∞	0/25	0.17	∞	0/25	0.31	∞	**0/25**	**0.32**	∞
z200v6d5iw20-struc	0/25	6e-13	∞	0/25	4e-08	∞	0/25	6e-07	∞	0/25	9e-07	∞	**0/25**	**1e-06**	∞	0/25	5e-07	∞
z400v3d5iw10-rand	25/25	100.00	0.49	25/25	100.00	0.49	25/25	100.00	0.49	25/25	100.00	0.48	**25/25**	**100.00**	**0.46**	25/25	100.00	0.49
z400v3d5iw10-struc	1/25	25.00	2407.07	2/25	42.00	1177.24	2/25	47.17	1163.41	1/25	47.46	2434.46	**4/25**	**46.76**	**586.75**	0/25	39.31	∞
z400v3d5iw20-rand	0/25	15.17	∞	0/25	17.83	∞	0/25	18.14	∞	**0/25**	**19.12**	∞	0/25	18.95	∞	0/25	17.44	∞
z400v3d5iw20-struc	0/25	0.99	∞	0/25	1.34	∞	**0/25**	**1.59**	∞	0/25	1.41	∞	0/25	1.28	∞	0/25	1.30	∞
z400v6d5iw10-rand	0/25	0.008	∞	0/25	0.01	∞	0/25	0.01	∞	**0/25**	**0.01**	∞	0/25	0.01	∞	0/25	0.009	∞
z400v6d5iw10-struc	0/25	(-)	∞	0/25	(-)	∞	0/25	2e-23	∞	0/25	2e-24	∞	0/25	2e-23	∞	**0/25**	**3e-21**	∞
z400v6d5iw20-rand	0/25	0.47	∞	0/25	0.64	∞	**0/25**	**0.73**	∞	0/25	0.51	∞	0/25	0.43	∞	0/25	0.28	∞
z400v6d5iw20-struc	0/25	3e-09	∞	0/25	3e-09	∞	0/25	3e-09	∞	**0/25**	**2e-08**	∞	0/25	5e-09	∞	0/25	1e-09	∞

Table B.4: Results of G+StS with initialization MB$^*(10^5)$, cutoff factor *cf* $= 2$, and varying noise probability *np* on problem set `gen`. All algorithms are run 25 times for 100 CPU seconds each. Summarized in Table 7.4 on page 70.

	GLS																	
	ρ = 0.7			ρ = 0.8			ρ = 0.9			ρ = 0.99			ρ = 0.999			ρ = 1.00		
Instance	Solved	Quality avg	Time	Solved	Quality avg	Time	Solved	Quality avg	Time	Solved	Quality avg	Time	Solved	Quality avg	Time	Solved	Quality avg	Time
alarm	**25/25**	**100.00**	**0.001**	**25/25**	**100.00**	**0.001**	25/25	100.00	0.002	25/25	100.00	0.002	**25/25**	**100.00**	**0.001**	25/25	100.00	0.002
alarm-rand	25/25	100.00	0.01	**25/25**	**100.00**	**0.009**	25/25	100.00	0.01	25/25	100.00	0.01	25/25	100.00	0.01	25/25	100.00	0.009
barley	0/25	1.03	∞	0/25	12.50	∞	0/25	48.00	∞	25/25	100.00	1.36	25/25	100.00	1.29	**25/25**	**100.00**	**1.08**
barley-rand	0/25	16.07	∞	0/25	27.19	∞	0/25	33.43	∞	3/25	78.83	751.41	25/25	100.00	10.56	**25/25**	**100.00**	**7.98**
diabetes	0/25	4e-162	∞	0/25	2e-143	∞	0/25	5e-112	∞	0/25	1e-38	∞	0/25	6e-07	∞	**0/25**	**0.04**	∞
diabetes-rand	0/25	4e-48	∞	0/25	4e-45	∞	0/25	3e-39	∞	0/25	8e-25	∞	0/25	1e-20	∞	**0/25**	**3e-11**	∞
hailfinder	25/25	100.00	0.03	25/25	100.00	0.02	25/25	100.00	0.02	25/25	100.00	0.02	25/25	100.00	0.02	**25/25**	**100.00**	**0.02**
hailfinder-rand	1/25	94.95	2452.63	15/25	99.09	103.67	25/25	100.00	3.94	25/25	100.00	0.22	25/25	100.00	0.19	**25/25**	**100.00**	**0.16**
insurance	25/25	100.00	0.001	**25/25**	**100.00**	**0.0008**	25/25	100.00	0.001	25/25	100.00	0.001	**25/25**	**100.00**	**0.0008**	25/25	100.00	0.001
insurance-rand	25/25	100.00	0.007	**25/25**	**100.00**	**0.006**	25/25	100.00	0.008	25/25	100.00	0.007	25/25	100.00	0.008	25/25	100.00	0.007
link	25/25	100.00	0.32	25/25	100.00	0.31	**25/25**	**100.00**	**0.26**	25/25	100.00	0.32	25/25	100.00	0.32	25/25	100.00	0.34
link-rand	0/25	1e-20	∞	0/25	1e-18	∞	0/25	9e-13	∞	0/25	0.74	∞	1/25	84.47	2484.21	**3/25**	**88.08**	**816.32**
mildew	0/25	34.22	∞	0/25	47.06	∞	2/25	82.32	1173.79	25/25	100.00	9.94	**25/25**	**100.00**	**7.32**	25/25	100.00	7.39
mildew-rand	0/25	84.61	∞	4/25	88.83	565.53	25/25	100.00	3.74	25/25	100.00	0.70	25/25	100.00	0.52	**25/25**	**100.00**	**0.48**
munin1	0/25	27.44	∞	0/25	71.67	∞	25/25	100.00	21.47	25/25	100.00	2.94	25/25	100.00	1.56	**25/25**	**100.00**	**0.88**
munin1-rand	0/25	2e-10	∞	0/25	3e-09	∞	0/25	2e-06	∞	0/25	2.61	∞	17/25	99.65	108.58	**25/25**	**100.00**	**27.09**
munin2	0/25	3e-123	∞	0/25	2e-94	∞	0/25	9e-32	∞	0/25	86.13	∞	11/25	98.76	216.65	**21/25**	**99.86**	**87.89**
munin2-rand	0/25	9e-101	∞	0/25	4e-94	∞	0/25	2e-88	∞	0/25	1e-43	∞	0/25	9e-14	∞	**0/25**	**4e-08**	∞
munin3	0/25	1e-117	∞	0/25	1e-88	∞	0/25	2e-60	∞	0/25	3e-05	∞	0/25	0.71	∞	**0/25**	**0.89**	∞
munin3-rand	0/25	2e-112	∞	0/25	2e-107	∞	0/25	1e-98	∞	0/25	2e-47	∞	0/25	3e-17	∞	**0/25**	**4e-12**	∞
munin4	0/25	2e-118	∞	0/25	5e-102	∞	0/25	1e-84	∞	0/25	12.64	∞	0/25	79.97	∞	**2/25**	**72.83**	**1228.32**
munin4-rand	0/25	2e-110	∞	0/25	4e-106	∞	0/25	2e-95	∞	0/25	2e-48	∞	0/25	5e-18	∞	**0/25**	**1e-10**	∞
pigs	0/25	7e-19	∞	0/25	0.001	∞	0/25	0.01	∞	25/25	100.00	0.88	25/25	100.00	0.77	**25/25**	**100.00**	**0.62**
pigs-rand	0/25	0.009	∞	0/25	0.06	∞	0/25	0.50	∞	0/25	81.97	∞	25/25	100.00	26.70	**25/25**	**100.00**	**23.04**
water	25/25	100.00	0.002	**25/25**	**100.00**	**0.001**	25/25	100.00	0.002	25/25	100.00	0.002	25/25	100.00	0.002	25/25	100.00	0.002
water-rand	25/25	100.00	0.10	25/25	100.00	0.10	25/25	100.00	0.16	25/25	100.00	0.16	25/25	100.00	0.14	**25/25**	**100.00**	**0.09**

Table B.5: Results of GLS with smoothing interval $N_\rho = 200$ and varying smoothing factor ρ on problem set `bnrep`. All algorithms are run 25 times for 100 CPU seconds each. Summarized in Table 7.5 on page 72.

	GLS																	
	ρ = 0.7			ρ = 0.8			ρ = 0.9			ρ = 0.99			ρ = 0.999			ρ = 1.00		
Instance	Solved	Quality avg	Time	Solved	Quality avg	Time	Solved	Quality avg	Time	Solved	Quality avg	Time	Solved	Quality avg	Time	Solved	Quality avg	Time
z100v3d5iw10-rand	25/25	100.00	12.88	25/25	100.00	0.71	25/25	100.00	0.19	25/25	100.00	0.23	25/25	100.00	0.27	**25/25**	**100.00**	**0.19**
z100v3d5iw10-struc	25/25	100.00	0.10	25/25	100.00	0.06	**25/25**	**100.00**	**0.05**	25/25	100.00	0.05	25/25	100.00	0.06	25/25	100.00	0.07
z100v3d5iw20-rand	25/25	100.00	13.92	25/25	100.00	1.70	25/25	100.00	0.38	**25/25**	**100.00**	**0.14**	25/25	100.00	0.15	25/25	100.00	0.15
z100v3d5iw20-struc	25/25	100.00	0.69	25/25	100.00	0.30	25/25	100.00	0.10	25/25	100.00	0.11	**25/25**	**100.00**	**0.09**	25/25	100.00	0.11
z100v6d5iw10-rand	0/25	44.79	∞	0/25	52.15	∞	0/25	55.28	∞	25/25	100.00	7.40	**25/25**	**100.00**	**6.57**	24/25	99.72	12.58
z100v6d5iw10-struc	25/25	100.00	1.93	25/25	100.00	0.80	25/25	100.00	0.30	**25/25**	**100.00**	**0.24**	25/25	100.00	0.24	25/25	100.00	0.29
z100v6d5iw20-rand	0/25	4.09	∞	0/25	7.30	∞	0/25	17.07	∞	23/25	98.78	34.82	**25/25**	**100.00**	**6.89**	25/25	100.00	7.69
z100v6d5iw20-struc	8/25	81.99	251.82	25/25	100.00	27.69	25/25	100.00	5.09	**25/25**	**100.00**	**0.93**	25/25	100.00	1.66	24/25	99.73	7.24
z200v3d5iw10-rand	0/25	46.86	∞	0/25	74.89	∞	2/25	94.79	1216.84	25/25	100.00	19.58	**25/25**	**100.00**	**17.05**	22/25	99.69	40.18
z200v3d5iw10-struc	0/25	72.39	∞	12/25	95.33	147.62	25/25	100.00	3.31	25/25	100.00	0.40	**25/25**	**100.00**	**0.34**	25/25	100.00	0.44
z200v3d5iw20-rand	0/25	21.97	∞	0/25	32.99	∞	0/25	63.44	∞	**24/25**	**99.86**	**37.29**	24/25	99.84	33.44	16/25	97.44	74.01
z200v3d5iw20-struc	1/25	69.61	2404.11	2/25	87.07	1175.07	25/25	100.00	2.02	25/25	100.00	0.48	**25/25**	**100.00**	**0.48**	25/25	100.00	0.63
z200v6d5iw10-rand	0/25	0.0003	∞	0/25	0.0009	∞	0/25	0.008	∞	0/25	17.75	∞	0/25	70.47	∞	**1/25**	**64.59**	**2439.31**
z200v6d5iw10-struc	0/25	41.55	∞	0/25	53.98	∞	1/25	99.42	2415.36	25/25	100.00	1.97	**25/25**	**100.00**	**1.62**	25/25	100.00	2.06
z200v6d5iw20-rand	0/25	2e-06	∞	0/25	1e-05	∞	0/25	0.0003	∞	0/25	2.43	∞	**1/25**	**56.83**	**2452.50**	0/25	44.55	∞
z200v6d5iw20-struc	0/25	5e-05	∞	0/25	0.007	∞	0/25	1.26	∞	**23/25**	**96.96**	**40.21**	18/25	82.38	70.14	13/25	67.25	109.79
z400v3d5iw10-rand	0/25	0.02	∞	0/25	0.09	∞	0/25	1.44	∞	4/25	94.04	568.37	**18/25**	**98.48**	**83.38**	14/25	97.23	109.80
z400v3d5iw10-struc	0/25	1.80	∞	0/25	8.16	∞	0/25	55.81	∞	**25/25**	**100.00**	**14.11**	25/25	100.00	19.41	21/25	99.20	41.87
z400v3d5iw20-rand	0/25	0.15	∞	0/25	0.15	∞	0/25	1.04	∞	0/25	68.31	∞	0/25	83.11	∞	**1/25**	**79.74**	**2445.11**
z400v3d5iw20-struc	0/25	0.29	∞	0/25	0.57	∞	0/25	5.81	∞	9/25	94.01	233.12	**12/25**	**94.07**	**150.35**	8/25	83.32	247.34
z400v6d5iw10-rand	0/25	8e-14	∞	0/25	5e-13	∞	0/25	7e-12	∞	0/25	0.002	∞	0/25	14.01	∞	**0/25**	**14.44**	∞
z400v6d5iw10-struc	0/25	4e-09	∞	0/25	1e-06	∞	0/25	0.0002	∞	0/25	53.80	∞	**1/25**	**82.87**	**2496.85**	1/25	62.76	2456.27
z400v6d5iw20-rand	0/25	3e-15	∞	0/25	2e-13	∞	0/25	2e-11	∞	0/25	0.01	∞	1/25	40.89	2477.87	**2/25**	**38.45**	**1161.35**
z400v6d5iw20-struc	0/25	5e-07	∞	0/25	3e-05	∞	0/25	0.003	∞	0/25	63.73	∞	**0/25**	**69.75**	∞	0/25	56.55	∞

Table B.6: Results for GLS with smoothing interval $N_\rho = 200$ and varying smoothing factor ρ on problem set gen. All algorithms are run 25 times for 100 CPU seconds each. Summarized in Table 7.6 on page 72.

	GLS with $\rho = 0.999$														
	N_ρ = 50			N_ρ = 200			N_ρ = 1000			N_ρ = 10000			$N_\rho = \infty$		
Instance	Solved	Quality avg	Time	Solved	Quality avg	Time	Solved	Quality avg	Time	Solved	Quality avg	Time	Solved	Quality avg	Time
alarm	25/25	100.00	0.002	25/25	100.00	0.001	**25/25**	**100.00**	**0.0004**	25/25	100.00	0.001	25/25	100.00	0.002
alarm-rand	25/25	100.00	0.01	25/25	100.00	0.01	**25/25**	**100.00**	**0.009**	**25/25**	**100.00**	**0.009**	**25/25**	**100.00**	**0.009**
barley	25/25	100.00	1.12	25/25	100.00	1.38	25/25	100.00	1.32	25/25	100.00	1.07	**25/25**	**100.00**	**1.00**
barley-rand	11/25	96.77	176.20	25/25	100.00	10.46	25/25	100.00	9.98	25/25	100.00	11.66	**25/25**	**100.00**	**6.81**
diabetes	0/25	3e-23	∞	0/25	2e-07	∞	0/25	0.22	∞	**0/25**	**1.18**	∞	0/25	0.22	∞
diabetes-rand	0/25	1e-22	∞	0/25	5e-21	∞	0/25	2e-17	∞	0/25	9e-16	∞	**0/25**	**1e-09**	∞
hailfinder	25/25	100.00	0.02	25/25	100.00	0.02	**25/25**	**100.00**	**0.02**	25/25	100.00	0.02	25/25	100.00	0.02
hailfinder-rand	25/25	100.00	0.22	25/25	100.00	0.18	25/25	100.00	0.20	25/25	100.00	0.16	**25/25**	**100.00**	**0.15**
insurance	**25/25**	**100.00**	**0.0008**	25/25	100.00	0.001	**25/25**	**100.00**	**0.0008**	25/25	100.00	0.001	**25/25**	**100.00**	**0.0008**
insurance-rand	25/25	100.00	0.008	25/25	100.00	0.007	**25/25**	**100.00**	**0.006**	25/25	100.00	0.006	25/25	100.00	0.006
link	25/25	100.00	0.30	**25/25**	**100.00**	**0.28**	25/25	100.00	0.28	25/25	100.00	0.30	25/25	100.00	0.32
link-rand	0/25	37.89	∞	0/25	71.63	∞	5/25	88.14	482.81	**11/25**	**94.40**	**199.93**	8/25	88.13	284.37
mildew	25/25	100.00	8.32	25/25	100.00	7.44	**25/25**	**100.00**	**6.59**	25/25	100.00	11.60	25/25	100.00	7.84
mildew-rand	25/25	100.00	0.76	25/25	100.00	0.50	25/25	100.00	0.51	**25/25**	**100.00**	**0.43**	25/25	100.00	0.51
munin1	25/25	100.00	2.02	25/25	100.00	1.52	25/25	100.00	1.17	**25/25**	**100.00**	**0.85**	25/25	100.00	0.85
munin1-rand	0/25	46.86	∞	13/25	99.51	142.15	25/25	100.00	24.95	**25/25**	**100.00**	**24.65**	25/25	100.00	26.01
munin2	1/25	91.55	2499.65	12/25	98.48	202.85	3/25	98.52	815.11	7/25	98.88	342.14	**19/25**	**99.63**	**106.56**
munin2-rand	0/25	2e-24	∞	0/25	3e-13	∞	0/25	7e-13	∞	0/25	6e-10	∞	**0/25**	**6e-08**	∞
munin3	0/25	0.52	∞	0/25	1.06	∞	0/25	1.14	∞	0/25	1.15	∞	**0/25**	**2.04**	∞
munin3-rand	0/25	2e-27	∞	0/25	1e-17	∞	0/25	2e-14	∞	0/25	8e-17	∞	**0/25**	**3e-11**	∞
munin4	0/25	55.33	∞	0/25	86.61	∞	0/25	88.61	∞	1/25	92.69	2486.76	**7/25**	**71.08**	**343.85**
munin4-rand	0/25	7e-30	∞	0/25	1e-17	∞	0/25	1e-13	∞	0/25	2e-17	∞	**0/25**	**4e-11**	∞
pigs	25/25	100.00	0.96	25/25	100.00	0.75	25/25	100.00	0.61	25/25	100.00	0.62	**25/25**	**100.00**	**0.59**
pigs-rand	11/25	99.19	210.00	25/25	100.00	25.19	**25/25**	**100.00**	**22.72**	25/25	100.00	24.11	25/25	100.00	22.97
water	25/25	100.00	0.002	**25/25**	**100.00**	**0.002**	**25/25**	**100.00**	**0.002**	**25/25**	**100.00**	**0.002**	**25/25**	**100.00**	**0.002**
water-rand	25/25	100.00	0.18	25/25	100.00	0.13	25/25	100.00	0.12	**25/25**	**100.00**	**0.08**	**25/25**	**100.00**	**0.08**

Table B.7: Full results for GLS with smoothing parameter $\rho = 0.999$ and varying smoothing interval N_ρ on problem set `bnrep`. All algorithms are run 25 times for 100 CPU seconds each. Summarized in Table 7.7 on page 76.

	GLS with $\rho = 0.999$)														
	N_ρ = 50			N_ρ = 200			N_ρ = 1000			N_ρ = 10000			N_ρ = ∞		
Instance	Solved	Quality avg	Time	Solved	Quality avg	Time	Solved	Quality avg	Time	Solved	Quality avg	Time	Solved	Quality avg	Time
z100v3d5iw10-rand	25/25	100.00	0.20	25/25	100.00	0.25	25/25	100.00	0.22	25/25	100.00	0.24	**25/25**	**100.00**	**0.19**
z100v3d5iw10-struc	25/25	100.00	0.06	**25/25**	**100.00**	**0.05**	25/25	100.00	0.06	25/25	100.00	0.07	25/25	100.00	0.07
z100v3d5iw20-rand	25/25	100.00	0.16	25/25	100.00	0.15	**25/25**	**100.00**	**0.14**	25/25	100.00	0.16	25/25	100.00	0.15
z100v3d5iw20-struc	25/25	100.00	0.09	**25/25**	**100.00**	**0.08**	25/25	100.00	0.09	25/25	100.00	0.11	25/25	100.00	0.10
z100v6d5iw10-rand	25/25	100.00	7.29	**25/25**	**100.00**	**6.11**	25/25	100.00	7.39	25/25	100.00	6.97	25/25	100.00	11.69
z100v6d5iw10-struc	25/25	100.00	0.22	25/25	100.00	0.24	**25/25**	**100.00**	**0.18**	25/25	100.00	0.24	25/25	100.00	0.26
z100v6d5iw20-rand	25/25	100.00	8.28	25/25	100.00	6.62	25/25	100.00	11.08	**25/25**	**100.00**	**5.86**	25/25	100.00	6.96
z100v6d5iw20-struc	25/25	100.00	1.86	**25/25**	**100.00**	**1.62**	25/25	100.00	1.98	25/25	100.00	2.81	24/25	99.73	7.54
z200v3d5iw10-rand	**25/25**	**100.00**	**13.63**	25/25	100.00	16.49	23/25	99.67	28.11	22/25	99.77	37.69	24/25	99.75	32.44
z200v3d5iw10-struc	25/25	100.00	0.33	25/25	100.00	0.32	**25/25**	**100.00**	**0.32**	25/25	100.00	0.42	25/25	100.00	0.42
z200v3d5iw20-rand	**25/25**	**100.00**	**22.17**	24/25	99.84	33.57	16/25	98.04	79.60	13/25	97.17	114.54	17/25	97.33	70.29
z200v3d5iw20-struc	25/25	100.00	0.51	**25/25**	**100.00**	**0.46**	25/25	100.00	0.51	25/25	100.00	0.60	25/25	100.00	0.61
z200v6d5iw10-rand	0/25	34.52	∞	0/25	67.63	∞	**1/25**	**76.92**	**2488.73**	0/25	72.58	∞	0/25	72.10	∞
z200v6d5iw10-struc	25/25	100.00	2.45	25/25	100.00	1.55	**25/25**	**100.00**	**1.50**	25/25	100.00	1.59	25/25	100.00	1.65
z200v6d5iw20-rand	0/25	14.59	∞	1/25	56.74	2436.20	1/25	58.69	2426.06	**5/25**	**61.84**	**473.53**	0/25	48.33	∞
z200v6d5iw20-struc	**21/25**	**89.57**	**42.00**	20/25	87.51	71.85	11/25	64.30	167.61	10/25	60.04	162.22	9/25	58.07	210.39
z400v3d5iw10-rand	**21/25**	**99.46**	**73.66**	21/25	99.11	56.52	20/25	98.50	57.75	14/25	98.27	108.14	14/25	96.45	114.73
z400v3d5iw10-struc	**25/25**	**100.00**	**12.65**	25/25	100.00	18.84	25/25	100.00	27.44	23/25	99.67	41.38	22/25	99.51	38.40
z400v3d5iw20-rand	0/25	78.82	∞	0/25	83.50	∞	**2/25**	**80.77**	**1192.93**	1/25	75.60	2447.04	0/25	77.49	∞
z400v3d5iw20-struc	**20/25**	**98.33**	**69.12**	12/25	93.88	155.81	15/25	90.72	110.10	7/25	85.38	280.16	5/25	76.33	428.36
z400v6d5iw10-rand	0/25	0.74	∞	0/25	15.42	∞	**0/25**	**22.38**	∞	0/25	20.60	∞	0/25	18.09	∞
z400v6d5iw10-struc	0/25	81.53	∞	1/25	82.45	2441.40	1/25	79.60	2461.42	**2/25**	**74.93**	**1187.44**	1/25	69.03	2406.19
z400v6d5iw20-rand	0/25	1.94	∞	2/25	41.65	1223.30	**5/25**	**59.03**	**467.16**	1/25	45.30	2448.01	0/25	45.80	∞
z400v6d5iw20-struc	0/25	77.01	∞	0/25	70.91	∞	0/25	67.71	∞	**2/25**	**65.90**	**1221.78**	0/25	60.62	∞

Table B.8: Full results for GLS with smoothing parameter $\rho = 0.999$ and varying smoothing interval N_ρ on problem set `gen`. All algorithms are run 25 times for 100 CPU seconds each. Summarized in Table 7.8 on page 77.

	G+StS “original”			GLS “original”			BBMB				
Instance	Solved	Quality avg	Time	Solved	Quality avg	Time	ib=2	ib=6	ib=10	ib=14	ib=18
alarm	25/25	100.00	0.01	25/25	100.00	0.003	**0.00/0.00**	**0.00/0.00**	0.01/0.01	**0.00/0.00**	**0.00/0.00**
alarm-rand	25/25	100.00	0.10	25/25	100.00	0.03	0.02/0.02	**0.00/0.00**	**0.00/0.00**	**0.00/0.00**	**0.00/0.00**
barley	25/25	100.00	29.22	0/25	8.39	∞	(0.07)	**1.21/2.91**	**1.21/2.89**	**1.21/2.94**	1.28/5.38
barley-rand	25/25	100.00	9.93	0/25	21.10	∞	28.69/-	1.23/1.23	**1.22/1.22**	**1.22/1.22**	1.23/1.23
diabetes	0/25	(-)	∞	0/25	3e-268	∞	(-)	4.23/4.23	**4.22/4.22**	4.24/4.24	4.23/4.23
diabetes-rand	0/25	3e-06	∞	0/25	3e-51	∞	(5e-17)	**3.80/3.80**	3.82/3.82	**3.80/3.80**	3.81/3.81
hailfinder	25/25	100.00	0.51	25/25	100.00	0.10	4.33/4.33	**0.00/0.00**	0.01/0.01	0.01/0.01	0.01/0.01
hailfinder-rand	25/25	100.00	4.74	2/25	97.22	1177.87	(18.92)	0.01/0.01	0.01/0.01	**0.00/0.00**	**0.00/0.00**
insurance	25/25	100.00	0.001	25/25	100.00	0.002	**0.00/0.00**	**0.00/0.00**	**0.00/0.00**	**0.00/0.00**	**0.00/0.00**
insurance-rand	25/25	100.00	0.01	25/25	100.00	0.02	**0.00/0.00**	**0.00/0.00**	**0.00/0.00**	**0.00/0.00**	**0.00/0.00**
link	0/25	(-)	∞	**25/25**	**100.00**	**5.30**	(9e-37)	(6e-06)	(0.003)	17.10/-	17.29/-
link-rand	0/25	2e-09	∞	0/25	6e-19	∞	(3e-19)	(0.69)	(34.26)	**36.14/36.14**	58.57/58.57
mildew	25/25	100.00	10.52	0/25	34.03	∞	(1.39)	0.72/0.72	1.08/1.08	0.72/0.72	**0.71/0.71**
mildew-rand	25/25	100.00	0.42	0/25	85.61	∞	28.67/31.74	**0.01/0.01**	**0.01/0.01**	**0.01/0.01**	**0.01/0.01**
munin1	0/25	3e-18	∞	0/25	40.68	∞	**1.03/-**	1.17/2.20	24.93/24.93	368.52/-	377.83/-
munin1-rand	0/25	4.40	∞	0/25	1e-09	∞	(0.03)	**1.23/13.96**	42.08/42.08	104.39/-	104.96/-
munin2	0/25	(-)	∞	0/25	1e-158	∞	(-)	7.51/8.25	3.55/3.55	**3.47/3.47**	3.54/4.93
munin2-rand	**0/25**	**1e-18**	∞	0/25	8e-100	∞	(-)	(-)	(-)	(-)	(-)
munin3	0/25	(-)	∞	0/25	2e-161	∞	(-)	28.07/28.72	**4.51/5.88**	**4.51/4.51**	4.54/5.77
munin3-rand	**0/25**	**3e-18**	∞	0/25	4e-114	∞	(-)	(-)	(-)	(-)	(-)
munin4	0/25	(-)	∞	0/25	1e-181	∞	(-)	(0.15)	22.76/22.76	**21.54/21.71**	23.78/23.79
munin4-rand	**0/25**	**2e-22**	∞	0/25	3e-112	∞	(-)	(-)	(-)	(-)	(-)
pigs	0/25	5e-14	∞	0/25	0.0003	∞	(2e-05)	**0.11/0.11**	0.38/0.38	0.53/0.53	0.53/0.53
pigs-rand	0/25	0.01	∞	0/25	0.02	∞	(1e-08)	2.08/2.08	**0.41/0.41**	0.83/0.83	0.83/0.83
water	25/25	100.00	0.005	**25/25**	**100.00**	**0.003**	0.01/0.01	0.01/0.01	0.01/0.01	0.01/0.01	0.01/0.01
water-rand	25/25	100.00	0.02	25/25	100.00	0.27	1.54/1.54	**0.01/0.01**	0.17/0.17	0.17/0.17	0.17/0.17

Table B.9: Comparison of s-BBMB with different i-bounds and our “original” versions of G+StS and GLS on problem set `bnrep`. The SLS algorithms were run 25 times for 100 CPU seconds, the deterministic s-BBMB algorithm once for 100 CPU seconds for every i-bound. The SLS algorithms used a random initialization, simple caching, and parameter values $\langle np, cf \rangle = \langle 40, 2 \rangle$ (G+StS), and $\langle N_\rho, \rho \rangle = \langle 200, 0.8 \rangle$ (GLS). Summarized in Table 8.1 on page 88.

	G+StS “original”			GLS “original”			BBMB				
Instance	Solved	Quality avg	Time	Solved	Quality avg	Time	ib=2	ib=6	ib=10	ib=14	ib=18
z100v3d5iw10-rand	11/25	91.24	164.58	25/25	100.00	5.11	(0.05)	4.17/73.11	**0.07/0.07**	0.08/0.08	0.08/0.08
z100v3d5iw10-struc	11/25	68.77	167.15	25/25	100.00	0.35	(-)	12.46/37.46	**0.16/0.16**	0.25/0.25	0.26/0.26
z100v3d5iw20-rand	25/25	100.00	6.19	25/25	100.00	14.06	(0.57)	(2.51)	**0.51/0.51**	5.38/5.38	35.46/35.46
z100v3d5iw20-struc	25/25	100.00	36.65	**25/25**	**100.00**	**2.18**	(-)	(0.67)	4.38/5.38	5.66/5.66	(-)
z100v6d5iw10-rand	0/25	38.93	∞	0/25	32.76	∞	(0.002)	(1.22)	**8.71/8.71**	8.81/8.81	8.77/8.78
z100v6d5iw10-struc	0/25	1.93	∞	**25/25**	**100.00**	**6.40**	(-)	24.67/25.03	14.15/14.15	14.45/14.45	14.22/14.22
z100v6d5iw20-rand	**0/25**	**17.55**	∞	0/25	3.30	∞	(0.0001)	(0.04)	(1.08)	(-)	(-)
z100v6d5iw20-struc	0/25	1.34	∞	12/25	89.70	168.95	(1e-14)	(-)	**33.82/42.56**	(-)	(-)
z200v3d5iw10-rand	0/25	11.93	∞	0/25	62.04	∞	(2e-06)	(0.33)	1.02/1.02	0.49/0.49	**0.48/0.48**
z200v3d5iw10-struc	0/25	0.05	∞	3/25	78.25	787.80	(-)	(-)	**0.37/0.37**	0.45/0.45	0.45/0.45
z200v3d5iw20-rand	0/25	7.29	∞	0/25	18.86	∞	(6e-07)	(0.07)	(5.15)	**7.01/7.01**	113.93/-
z200v3d5iw20-struc	0/25	0.02	∞	0/25	61.36	∞	(-)	(-)	(-)	**8.13/8.13**	157.06/-
z200v6d5iw10-rand	0/25	0.03	∞	0/25	0.0002	∞	(9e-12)	(5e-07)	**35.48/36.08**	44.53/44.53	45.48/45.48
z200v6d5iw10-struc	0/25	9e-16	∞	0/25	23.75	∞	(-)	(-)	**40.22/40.23**	64.87/64.87	64.78/64.78
z200v6d5iw20-rand	0/25	0.02	∞	0/25	1e-06	∞	(1e-16)	(6e-09)	**(0.19)**	(-)	(-)
z200v6d5iw20-struc	0/25	3e-13	∞	**0/25**	**0.0003**	∞	(-)	(-)	(1e-07)	(-)	(-)
z400v3d5iw10-rand	0/25	0.002	∞	0/25	0.02	∞	(8e-15)	(0.004)	42.01/70.83	**0.81/0.81**	0.90/0.90
z400v3d5iw10-struc	0/25	6e-10	∞	0/25	2.03	∞	(-)	(-)	(5.24)	**1.12/1.12**	**1.12/1.12**
z400v3d5iw20-rand	0/25	0.007	∞	0/25	0.03	∞	(5e-18)	(0.0003)	(0.26)	**49.06/-**	(-)
z400v3d5iw20-struc	0/25	(-)	∞	0/25	0.18	∞	(-)	(-)	(7.21)	**14.73/15.58**	153.33/-
z400v6d5iw10-rand	0/25	1e-09	∞	0/25	6e-15	∞	(1e-24)	(4e-08)	(24.33)	68.88/68.88	**68.81/68.82**
z400v6d5iw10-struc	0/25	(-)	∞	0/25	1e-08	∞	(-)	(6e-19)	(0.25)	**55.48/55.48**	55.68/55.68
z400v6d5iw20-rand	0/25	2e-07	∞	0/25	5e-15	∞	(3e-25)	(1e-07)	**(0.02)**	(-)	(-)
z400v6d5iw20-struc	0/25	(-)	∞	0/25	2e-07	∞	(-)	(-)	**(0.02)**	(-)	(-)

Table B.10: Comparison of s-BBMB with different i-bounds and our “original” versions of G+StS and GLS on problem set `gen`. The SLS algorithms were run 25 times for 100 CPU seconds, the deterministic s-BBMB algorithm once for 100 CPU seconds for every i-bound. The SLS algorithms used a random initialization, simple caching, and parameter values $\langle np, cf \rangle = \langle 40, 2 \rangle$ (G+StS), and $\langle N_\rho, \rho \rangle = \langle 200, 0.8 \rangle$ (GLS). Summarized in Table 8.2 on page 88.

	G+StS									ILS					
	random init, old caching			random init, new caching			MB$^*(10^5)$ init, new caching			random initialization			MB$^*(10^5)$ initialization		
Instance	Solved	Quality avg	Time	Solved	Quality avg	Time	Solved	Quality avg	Time	Solved	Quality avg	Time	Solved	Quality avg	Time
alarm	25/25	100.00	0.01	25/25	100.00	0.004	25/25	100.00	0.0008	25/25	100.00	0.0008	**25/25**	**100.00**	**0.0004**
alarm-rand	25/25	100.00	0.10	25/25	100.00	0.03	25/25	100.00	0.0008	25/25	100.00	0.002	**25/25**	**100.00**	**0.0004**
barley	25/25	100.00	29.22	25/25	100.00	4.13	**25/25**	**100.00**	**1.26**	25/25	100.00	2.67	25/25	100.00	1.50
barley-rand	25/25	100.00	9.93	25/25	100.00	2.25	**25/25**	**100.00**	**0.10**	25/25	100.00	0.70	25/25	100.00	0.10
diabetes	0/25	(-)	∞	0/25	(-)	∞	0/25	8e-32	∞	0/25	9e-28	∞	**0/25**	**3e-14**	∞
diabetes-rand	0/25	3e-06	∞	0/25	0.02	∞	0/25	76.32	∞	1/25	73.32	2440.48	**4/25**	**89.10**	**585.05**
hailfinder	25/25	100.00	0.51	25/25	100.00	0.16	**25/25**	**100.00**	**0.002**	25/25	100.00	0.005	**25/25**	**100.00**	**0.002**
hailfinder-rand	25/25	100.00	4.74	25/25	100.00	1.62	25/25	100.00	0.003	25/25	100.00	0.04	**25/25**	**100.00**	**0.002**
insurance	25/25	100.00	0.001	**25/25**	**100.00**	**0.0008**	**25/25**	**100.00**	**0.0008**	**25/25**	**100.00**	**0.0008**	**25/25**	**100.00**	**0.0008**
insurance-rand	25/25	100.00	0.01	25/25	100.00	0.006	**25/25**	**100.00**	**0.0004**	25/25	100.00	0.001	**25/25**	**100.00**	**0.0004**
link	0/25	(-)	∞	0/25	8e-07	∞	**0/25**	**0.50**	∞	0/25	0.004	∞	**0/25**	**0.50**	∞
link-rand	0/25	2e-09	∞	0/25	0.0002	∞	0/25	32.85	∞	0/25	38.55	∞	**0/25**	**60.59**	∞
mildew	25/25	100.00	10.52	25/25	100.00	3.11	25/25	100.00	0.03	25/25	100.00	1.34	**25/25**	**100.00**	**0.03**
mildew-rand	25/25	100.00	0.42	25/25	100.00	0.12	25/25	100.00	0.004	25/25	100.00	0.38	**25/25**	**100.00**	**0.004**
munin1	0/25	3e-18	∞	0/25	7e-08	∞	25/25	100.00	0.21	10/25	0.06	171.48	**25/25**	**100.00**	**0.21**
munin1-rand	0/25	4.40	∞	0/25	18.48	∞	0/25	30.05	∞	**25/25**	**100.00**	**6.54**	25/25	100.00	8.33
munin2	0/25	(-)	∞	0/25	(-)	∞	**25/25**	**100.00**	**0.75**	0/25	(-)	∞	25/25	100.00	0.78
munin2-rand	0/25	1e-18	∞	0/25	2e-09	∞	**0/25**	**97.43**	∞	0/25	3.63	∞	**0/25**	**97.43**	∞
munin3	0/25	(-)	∞	0/25	(-)	∞	**25/25**	**100.00**	**0.80**	0/25	(-)	∞	25/25	100.00	0.81
munin3-rand	0/25	3e-18	∞	0/25	3e-10	∞	0/25	59.91	∞	0/25	2.20	∞	**0/25**	**68.01**	∞
munin4	0/25	(-)	∞	0/25	(-)	∞	0/25	0.003	∞	0/25	(-)	∞	**25/25**	**100.00**	**4.72**
munin4-rand	0/25	2e-22	∞	0/25	4e-12	∞	0/25	8.10	∞	0/25	0.53	∞	**0/25**	**9.77**	∞
pigs	0/25	5e-14	∞	0/25	1e-08	∞	**25/25**	**100.00**	**0.10**	1/25	23.65	2411.99	25/25	100.00	0.10
pigs-rand	0/25	0.01	∞	0/25	1.09	∞	21/25	98.89	57.91	1/25	65.13	2448.12	**25/25**	**100.00**	**1.93**
water	25/25	100.00	0.005	25/25	100.00	0.002	25/25	100.00	0.003	**25/25**	**100.00**	**0.0008**	25/25	100.00	0.003
water-rand	25/25	100.00	0.02	25/25	100.00	0.009	25/25	100.00	0.05	**25/25**	**100.00**	**0.002**	25/25	100.00	0.05

Table B.11: Results for non-penalty based algorithms on problem set `bnrep`. All algorithms were run 25 times for 100 CPU seconds each with their default parameters.

	G+StS									ILS					
	random init, old caching			random init, new caching			MB$^*(10^5)$ init, new caching			random initialization			MB$^*(10^5)$ initialization		
Instance	Solved	Quality avg	Time	Solved	Quality avg	Time	Solved	Quality avg	Time	Solved	Quality avg	Time	Solved	Quality avg	Time
z100v3d5iw10-rand	11/25	91.24	164.58	25/25	100.00	12.29	**25/25**	**100.00**	**0.02**	25/25	100.00	0.82	**25/25**	**100.00**	**0.02**
z100v3d5iw10-struc	11/25	68.77	167.15	25/25	100.00	17.59	23/25	95.61	40.33	**25/25**	**100.00**	**0.95**	25/25	100.00	2.16
z100v3d5iw20-rand	25/25	100.00	6.19	25/25	100.00	0.77	25/25	100.00	0.23	25/25	100.00	0.30	**25/25**	**100.00**	**0.23**
z100v3d5iw20-struc	25/25	100.00	36.65	25/25	100.00	6.00	25/25	100.00	6.11	**25/25**	**100.00**	**0.30**	25/25	100.00	0.59
z100v6d5iw10-rand	0/25	38.93	∞	3/25	70.38	806.79	0/25	68.93	∞	**23/25**	**99.43**	**34.96**	23/25	99.43	39.94
z100v6d5iw10-struc	0/25	1.93	∞	0/25	22.34	∞	0/25	28.05	∞	**25/25**	**100.00**	**5.90**	25/25	100.00	8.12
z100v6d5iw20-rand	0/25	17.55	∞	0/25	33.22	∞	0/25	29.89	∞	**8/25**	**78.92**	**256.97**	7/25	76.96	298.94
z100v6d5iw20-struc	0/25	1.34	∞	1/25	17.92	2490.86	0/25	14.64	∞	**25/25**	**100.00**	**18.63**	25/25	100.00	19.37
z200v3d5iw10-rand	0/25	11.93	∞	0/25	40.14	∞	0/25	74.73	∞	11/25	93.21	159.56	**15/25**	**95.94**	**111.39**
z200v3d5iw10-struc	0/25	0.05	∞	0/25	1.24	∞	25/25	100.00	14.66	25/25	100.00	9.50	**25/25**	**100.00**	**7.94**
z200v3d5iw20-rand	0/25	7.29	∞	0/25	26.81	∞	0/25	24.19	∞	**25/25**	**100.00**	**6.25**	25/25	100.00	12.46
z200v3d5iw20-struc	0/25	0.02	∞	0/25	0.38	∞	0/25	0.82	∞	**18/25**	**84.04**	**67.78**	17/25	83.02	92.04
z200v6d5iw10-rand	0/25	0.03	∞	0/25	0.19	∞	0/25	1.32	∞	**0/25**	**58.13**	∞	0/25	54.23	∞
z200v6d5iw10-struc	0/25	9e-16	∞	0/25	8e-10	∞	0/25	0.007	∞	0/25	1.16	∞	**0/25**	**2.46**	∞
z200v6d5iw20-rand	0/25	0.02	∞	0/25	0.37	∞	0/25	0.36	∞	**21/25**	**96.50**	**61.58**	19/25	86.07	82.03
z200v6d5iw20-struc	0/25	3e-13	∞	0/25	1e-07	∞	0/25	1e-06	∞	**0/25**	**0.69**	∞	0/25	0.58	∞
z400v3d5iw10-rand	0/25	0.002	∞	0/25	0.25	∞	25/25	100.00	0.49	14/25	95.85	128.87	**25/25**	**100.00**	**0.47**
z400v3d5iw10-struc	0/25	6e-10	∞	0/25	9e-05	∞	2/25	39.76	1217.07	0/25	22.31	∞	**25/25**	**100.00**	**5.64**
z400v3d5iw20-rand	0/25	0.007	∞	0/25	0.78	∞	0/25	18.20	∞	**1/25**	**72.41**	**2448.06**	0/25	74.32	∞
z400v3d5iw20-struc	0/25	(-)	∞	0/25	2e-06	∞	0/25	1.22	∞	0/25	29.86	∞	**1/25**	**46.49**	**2483.89**
z400v6d5iw10-rand	0/25	1e-09	∞	0/25	1e-05	∞	0/25	0.02	∞	**0/25**	**4.80**	∞	0/25	3.66	∞
z400v6d5iw10-struc	0/25	(-)	∞	0/25	1e-23	∞	0/25	6e-23	∞	0/25	0.0003	∞	**0/25**	**0.0005**	∞
z400v6d5iw20-rand	0/25	2e-07	∞	0/25	0.0001	∞	0/25	0.55	∞	0/25	16.78	∞	**0/25**	**18.58**	∞
z400v6d5iw20-struc	0/25	(-)	∞	0/25	5e-18	∞	0/25	6e-09	∞	0/25	0.04	∞	**0/25**	**0.20**	∞

Table B.12: Results for non-penalty based algorithms on problem set `gen`. All algorithms were run 25 times for 100 CPU seconds each with their default parameters.

	GLS, random initialization									GLS$^+$					
	“original”			“original”,$\rho = 0.999$			$\rho = 0.999$, new caching			random initialization			initialization MB$^*(10^5)$		
Instance	Solved	Quality avg	Time	Solved	Quality avg	Time	Solved	Quality avg	Time	Solved	Quality avg	Time	Solved	Quality avg	Time
alarm	25/25	100.00	0.003	25/25	100.00	0.002	25/25	100.00	0.001	**25/25**	**100.00**	**0.0004**	25/25	100.00	0.0008
alarm-rand	25/25	100.00	0.03	25/25	100.00	0.03	25/25	100.00	0.01	25/25	100.00	0.006	**25/25**	**100.00**	**0.0008**
barley	0/25	8.39	∞	25/25	100.00	8.02	25/25	100.00	1.29	25/25	100.00	0.90	**25/25**	**100.00**	**0.53**
barley-rand	0/25	21.10	∞	25/25	100.00	53.79	25/25	100.00	10.56	25/25	100.00	6.93	**25/25**	**100.00**	**0.10**
diabetes	0/25	3e-268	∞	0/25	4e-198	∞	0/25	6e-07	∞	**0/25**	**0.03**	∞	0/25	0.02	∞
diabetes-rand	0/25	3e-51	∞	0/25	9e-52	∞	0/25	1e-20	∞	0/25	3e-13	∞	**0/25**	**67.78**	∞
hailfinder	25/25	100.00	0.10	25/25	100.00	0.09	25/25	100.00	0.02	25/25	100.00	0.008	**25/25**	**100.00**	**0.002**
hailfinder-rand	2/25	97.22	1177.87	25/25	100.00	1.24	25/25	100.00	0.19	25/25	100.00	0.22	**25/25**	**100.00**	**0.003**
insurance	25/25	100.00	0.002	25/25	100.00	0.002	**25/25**	**100.00**	**0.0008**	**25/25**	**100.00**	**0.0008**	**25/25**	**100.00**	**0.0008**
insurance-rand	25/25	100.00	0.02	25/25	100.00	0.02	25/25	100.00	0.008	25/25	100.00	0.004	**25/25**	**100.00**	**0.0004**
link	25/25	100.00	5.30	25/25	100.00	6.26	25/25	100.00	0.32	**25/25**	**100.00**	**0.18**	25/25	100.00	1.06
link-rand	0/25	6e-19	∞	0/25	0.01	∞	1/25	84.47	2484.21	**2/25**	**84.61**	**1245.07**	1/25	85.48	2497.44
mildew	0/25	34.03	∞	25/25	100.00	18.70	25/25	100.00	7.32	25/25	100.00	3.92	**25/25**	**100.00**	**0.03**
mildew-rand	0/25	85.61	∞	25/25	100.00	2.30	25/25	100.00	0.52	25/25	100.00	0.40	**25/25**	**100.00**	**0.004**
munin1	0/25	40.68	∞	25/25	100.00	15.72	25/25	100.00	1.56	**25/25**	**100.00**	**0.12**	25/25	100.00	0.21
munin1-rand	0/25	1e-09	∞	0/25	68.23	∞	17/25	99.65	108.58	**20/25**	**99.79**	**78.91**	15/25	99.84	117.22
munin2	0/25	1e-158	∞	0/25	1e-62	∞	11/25	98.76	216.65	25/25	100.00	9.21	**25/25**	**100.00**	**0.79**
munin2-rand	0/25	8e-100	∞	0/25	3e-96	∞	0/25	9e-14	∞	0/25	2e-11	∞	**0/25**	**97.43**	∞
munin3	0/25	2e-161	∞	0/25	3e-73	∞	0/25	0.71	∞	4/25	21.64	598.17	**25/25**	**100.00**	**0.84**
munin3-rand	0/25	4e-114	∞	0/25	2e-105	∞	0/25	3e-17	∞	0/25	7e-14	∞	**0/25**	**57.31**	∞
munin4	0/25	1e-181	∞	0/25	5e-85	∞	0/25	79.97	∞	**25/25**	**100.00**	**17.30**	25/25	100.00	34.30
munin4-rand	0/25	3e-112	∞	0/25	2e-100	∞	0/25	5e-18	∞	0/25	1e-15	∞	**0/25**	**7.54**	∞
pigs	0/25	0.0003	∞	25/25	100.00	9.14	25/25	100.00	0.77	25/25	100.00	0.56	**25/25**	**100.00**	**0.11**
pigs-rand	0/25	0.02	∞	0/25	89.84	∞	25/25	100.00	26.70	**25/25**	**100.00**	**24.69**	25/25	100.00	30.46
water	25/25	100.00	0.003	25/25	100.00	0.003	25/25	100.00	0.002	**25/25**	**100.00**	**0.0004**	25/25	100.00	0.003
water-rand	25/25	100.00	0.27	25/25	100.00	0.41	25/25	100.00	0.14	**25/25**	**100.00**	**0.01**	25/25	100.00	0.06

Table B.13: Results for penalty based algorithms on problem set `bnrep`. All algorithms were run 25 times for 100 CPU seconds each. Summarized in Table 8.5 on page 97.

	GLS, random initialization									GLS$^+$					
	“original”			“original”,$\rho = 0.999$			$\rho = 0.999$, new caching			random initialization			initialization MB$^*(10^5)$		
Instance	Solved	Quality avg	Time	Solved	Quality avg	Time	Solved	Quality avg	Time	Solved	Quality avg	Time	Solved	Quality avg	Time
z100v3d5iw10-rand	25/25	100.00	5.11	25/25	100.00	1.36	25/25	100.00	0.27	25/25	100.00	0.11	**25/25**	**100.00**	**0.02**
z100v3d5iw10-struc	25/25	100.00	0.35	25/25	100.00	0.22	25/25	100.00	0.06	**25/25**	**100.00**	**0.03**	25/25	100.00	0.22
z100v3d5iw20-rand	25/25	100.00	14.06	25/25	100.00	0.88	25/25	100.00	0.15	**25/25**	**100.00**	**0.10**	25/25	100.00	0.24
z100v3d5iw20-struc	25/25	100.00	2.18	25/25	100.00	0.48	25/25	100.00	0.09	**25/25**	**100.00**	**0.08**	25/25	100.00	0.44
z100v6d5iw10-rand	0/25	32.76	∞	24/25	99.72	33.14	25/25	100.00	6.57	25/25	100.00	4.87	**25/25**	**100.00**	**3.60**
z100v6d5iw10-struc	25/25	100.00	6.40	25/25	100.00	1.57	25/25	100.00	0.24	**25/25**	**100.00**	**0.14**	25/25	100.00	0.47
z100v6d5iw20-rand	0/25	3.30	∞	24/25	98.30	55.04	25/25	100.00	6.89	25/25	100.00	7.20	**25/25**	**100.00**	**2.94**
z100v6d5iw20-struc	12/25	89.70	168.95	24/25	99.43	22.69	25/25	100.00	1.66	**25/25**	**100.00**	**1.33**	25/25	100.00	1.43
z200v3d5iw10-rand	0/25	62.04	∞	12/25	96.27	157.10	25/25	100.00	17.05	25/25	100.00	12.89	**25/25**	**100.00**	**2.12**
z200v3d5iw10-struc	3/25	78.25	787.80	25/25	100.00	3.19	25/25	100.00	0.34	**25/25**	**100.00**	**0.25**	25/25	100.00	0.58
z200v3d5iw20-rand	0/25	18.86	∞	7/25	94.22	300.68	24/25	99.84	33.44	24/25	99.95	42.14	**25/25**	**100.00**	**52.55**
z200v3d5iw20-struc	0/25	61.36	∞	25/25	100.00	5.00	25/25	100.00	0.48	**25/25**	**100.00**	**0.45**	25/25	100.00	1.31
z200v6d5iw10-rand	0/25	0.0002	∞	0/25	13.44	∞	0/25	70.47	∞	0/25	74.23	∞	**0/25**	**74.60**	∞
z200v6d5iw10-struc	0/25	23.75	∞	25/25	100.00	15.88	**25/25**	**100.00**	**1.62**	25/25	100.00	1.72	25/25	100.00	3.06
z200v6d5iw20-rand	0/25	1e-06	∞	0/25	5.62	∞	1/25	56.83	2452.50	2/25	65.24	1243.03	**25/25**	**100.00**	**54.85**
z200v6d5iw20-struc	0/25	0.0003	∞	5/25	38.17	453.85	18/25	82.38	70.14	18/25	83.32	64.13	**21/25**	**88.74**	**54.32**
z400v3d5iw10-rand	0/25	0.02	∞	4/25	77.51	590.08	18/25	98.48	83.38	21/25	99.32	60.35	**25/25**	**100.00**	**0.47**
z400v3d5iw10-struc	0/25	2.03	∞	13/25	94.07	147.72	25/25	100.00	19.41	**25/25**	**100.00**	**18.14**	23/25	99.68	20.07
z400v3d5iw20-rand	0/25	0.03	∞	0/25	59.88	∞	0/25	83.11	∞	**2/25**	**83.56**	**1194.24**	0/25	87.97	∞
z400v3d5iw20-struc	0/25	0.18	∞	1/25	62.43	2484.46	12/25	94.07	150.35	**13/25**	**94.41**	**125.80**	11/25	92.84	161.18
z400v6d5iw10-rand	0/25	6e-15	∞	0/25	0.002	∞	0/25	14.01	∞	0/25	15.32	∞	**0/25**	**17.30**	∞
z400v6d5iw10-struc	0/25	1e-08	∞	0/25	18.16	∞	**1/25**	**82.87**	**2496.85**	1/25	82.53	2417.76	0/25	83.98	∞
z400v6d5iw20-rand	0/25	5e-15	∞	0/25	0.09	∞	1/25	40.89	2477.87	**1/25**	**44.29**	**2451.07**	0/25	29.79	∞
z400v6d5iw20-struc	0/25	2e-07	∞	0/25	26.66	∞	0/25	69.75	∞	**2/25**	**74.20**	**1172.01**	0/25	69.53	∞

Table B.14: Results for penalty based algorithms on problem set `gen`. All algorithms were run 25 times for 100 CPU seconds each. Summarized in Table 8.6 on page 97.

	d-BBMB					s-BBMB					Anytime
Instance	ib=2	ib=6	ib=10	ib=14	ib=18	ib=2	ib=6	ib=10	ib=14	ib=18	MB
alarm	**0.00/0.00**	**0.00/0.00**	**0.00/0.00**	0.01/0.01	**0.00/0.00**	**0.00/0.00**	**0.00/0.00**	0.01/0.01	**0.00/0.00**	**0.00/0.00**	**0.00/0.00**
alarm-rand	0.03/0.03	**0.00/0.00**	**0.00/0.00**	**0.00/0.00**	0.01/0.01	0.02/0.02	**0.00/0.00**	**0.00/0.00**	**0.00/0.00**	**0.00/0.00**	**0.00/0.00**
barley	(0.07)	3.20/14.64	3.20/13.63	2.05/7.19	2.05/7.24	(0.07)	1.21/2.91	1.21/2.89	1.21/2.94	1.28/5.38	**0.22/0.22**
barley-rand	7.17/-	3.47/6.83	3.46/6.79	3.54/3.54	3.45/3.45	28.69/-	1.23/1.23	1.22/1.22	1.22/1.22	1.23/1.23	**0.20/0.20**
diabetes	(-)	(-)	(-)	(-)	(-)	(-)	4.23/4.23	**4.22/4.22**	4.24/4.24	4.23/4.23	4.70/4.70
diabetes-rand	(9e-09)	(-)	(-)	(-)	(-)	(5e-17)	**3.80/3.80**	3.82/3.82	**3.80/3.80**	3.81/3.81	3.95/7.82
hailfinder	0.40/0.40	0.02/0.02	0.03/0.03	0.02/0.02	0.02/0.02	4.33/4.33	**0.00/0.00**	0.01/0.01	0.01/0.01	0.01/0.01	**0.00/0.00**
hailfinder-rand	51.20/97.28	0.03/0.03	0.02/0.02	0.03/0.03	0.03/0.03	(18.92)	0.01/0.01	0.01/0.01	**0.00/0.00**	**0.00/0.00**	0.01/0.01
insurance	0.01/0.01	0.01/0.01	0.01/0.01	**0.00/0.00**	**0.00/0.00**	**0.00/0.00**	**0.00/0.00**	**0.00/0.00**	**0.00/0.00**	**0.00/0.00**	0.01/0.01
insurance-rand	0.01/0.01	**0.00/0.00**	**0.00/0.00**	**0.00/0.00**	**0.00/0.00**	**0.00/0.00**	**0.00/0.00**	**0.00/0.00**	**0.00/0.00**	**0.00/0.00**	0.01/0.01
link	(3e-31)	(0.003)	(0.50)	96.13/-	96.11/-	(9e-37)	(6e-06)	(0.003)	**17.10/-**	17.29/-	50.05/50.05
link-rand	(4e-11)	(3.71)	(63.80)	(-)	(-)	(3e-19)	(0.69)	(34.26)	**36.14/36.14**	58.57/58.57	(39.81)
mildew	3.07/30.00	4.29/4.50	3.65/3.65	3.69/3.69	3.70/3.70	(1.39)	0.72/0.72	1.08/1.08	0.72/0.72	0.71/0.71	**0.06/0.06**
mildew-rand	7.16/8.24	0.07/0.07	0.07/0.07	0.06/0.06	0.07/0.07	28.67/31.74	0.01/0.01	0.01/0.01	0.01/0.01	0.01/0.01	**0.00/0.01**
munin1	1.02/-	2.17/8.56	(0.15)	(-)	(-)	1.03/-	1.17/2.20	24.93/24.93	(-)	(-)	**0.08/7.39**
munin1-rand	(1.04)	2.12/49.95	88.91/94.50	(-)	(-)	(0.03)	**1.23/13.96**	42.08/42.08	(-)	(-)	38.84/38.84
munin2	(-)	27.51/30.06	40.77/40.77	40.11/40.11	40.34/40.34	(-)	7.51/8.25	3.55/3.55	3.47/3.47	3.54/4.93	**0.80/3.17**
munin2-rand	(-)	(-)	(-)	(-)	(-)	(-)	(-)	(-)	(-)	(-)	**3.27/3.27**
munin3	(-)	20.02/23.14	27.31/27.31	26.55/26.55	26.60/26.60	(-)	28.07/28.72	4.51/5.88	4.51/4.51	4.54/5.77	**0.85/3.63**
munin3-rand	(-)	(-)	(-)	(-)	(-)	(-)	(-)	(-)	(-)	(-)	**4.05/4.05**
munin4	(-)	(0.0001)	(-)	(-)	(-)	(-)	(0.15)	22.76/22.76	21.54/21.71	23.78/23.79	**18.70/18.70**
munin4-rand	(-)	(-)	(-)	(-)	(-)	(-)	(-)	(-)	(-)	(-)	**21.36/21.36**
pigs	(0.02)	1.03/1.05	1.66/1.66	1.80/1.80	1.79/1.79	(2e-05)	**0.11/0.11**	0.38/0.38	0.53/0.53	0.53/0.53	0.26/0.32
pigs-rand	(0.008)	1.02/3.68	2.00/2.00	2.73/2.73	2.72/2.72	(1e-08)	2.08/2.08	**0.41/0.41**	0.83/0.83	0.83/0.83	1.40/1.40
water	0.07/0.07	0.03/0.03	0.04/0.04	0.03/0.03	0.04/0.04	**0.01/0.01**	**0.01/0.01**	**0.01/0.01**	**0.01/0.01**	**0.01/0.01**	**0.01/0.01**
water-rand	5.12/5.62	0.21/0.21	0.77/0.77	0.78/0.78	0.76/0.76	1.54/1.54	**0.01/0.01**	0.17/0.17	0.17/0.17	0.17/0.17	0.19/0.19

Table B.15: Full results for exact algorithms on problem set `bnrep`. All algorithms were run for 100 CPU seconds. Summarized in Table 9.1 on page 109.

	d-BBMB					s-BBMB					Anytime
Instance	ib=2	ib=6	ib=10	ib=14	ib=18	ib=2	ib=6	ib=10	ib=14	ib=18	MB
z100v3d5iw10-rand	(5.27)	1.02/48.53	1.05/1.26	0.97/0.97	0.98/0.98	(0.05)	4.17/73.11	**0.07/0.07**	0.08/0.08	0.08/0.08	**0.07/0.07**
z100v3d5iw10-struc	(5.38)	6.14/30.05	2.12/2.75	2.41/2.41	2.41/2.41	(-)	12.46/37.46	**0.16/0.16**	0.25/0.25	0.26/0.26	0.52/0.52
z100v3d5iw20-rand	(0.22)	56.35/-	4.25/13.11	21.75/21.75	64.01/64.01	(0.57)	(2.51)	**0.51/0.51**	5.38/5.38	35.46/35.46	55.16/55.16
z100v3d5iw20-struc	(0.29)	53.25/-	19.67/-	66.79/-	(-)	(-)	(0.67)	**4.38/5.38**	5.66/5.66	(-)	(5.49)
z100v6d5iw10-rand	(0.06)	(92.97)	(-)	(-)	(-)	(0.002)	(1.22)	**8.71/8.71**	8.81/8.81	8.77/8.78	52.58/92.28
z100v6d5iw10-struc	(0.01)	(72.53)	(-)	(-)	(-)	(-)	24.67/25.03	**14.15/14.15**	14.45/14.45	14.22/14.22	(-)
z100v6d5iw20-rand	(0.06)	(0.54)	(-)	(-)	(-)	(0.0001)	(0.04)	**(1.08)**	(-)	(-)	(0.37)
z100v6d5iw20-struc	(2e-08)	(2e-09)	(-)	(-)	(-)	(1e-14)	(-)	**33.82/42.56**	(-)	(-)	(-)
z200v3d5iw10-rand	(0.008)	(20.03)	7.58/43.88	8.56/8.56	8.59/8.59	(2e-06)	(0.33)	1.02/1.02	0.49/0.49	**0.48/0.48**	1.04/1.04
z200v3d5iw10-struc	(-)	(2.32)	4.83/4.83	5.36/5.36	5.35/5.35	(-)	(-)	**0.37/0.37**	0.45/0.45	0.45/0.45	1.02/1.02
z200v3d5iw20-rand	(0.0001)	(0.10)	(41.54)	91.14/-	(-)	(6e-07)	(0.07)	(5.15)	**7.01/7.01**	(-)	(37.15)
z200v3d5iw20-struc	(8e-07)	(0.0001)	(0.002)	(-)	(-)	(-)	(-)	(-)	**8.13/8.13**	(-)	(0.80)
z200v6d5iw10-rand	(9e-07)	(0.67)	(-)	(-)	(-)	(9e-12)	(5e-07)	**35.48/36.08**	44.53/44.53	45.48/45.48	(95.99)
z200v6d5iw10-struc	(4e-19)	(1e-10)	(-)	(-)	(-)	(-)	(-)	**40.22/40.23**	64.87/64.87	64.78/64.78	92.90/92.90
z200v6d5iw20-rand	(1e-10)	(0.07)	(-)	(-)	(-)	(1e-16)	(6e-09)	**(0.19)**	(-)	(-)	(0.02)
z200v6d5iw20-struc	(-)	(9e-15)	(-)	(-)	(-)	(-)	(-)	**(1e-07)**	(-)	(-)	(1e-15)
z400v3d5iw10-rand	(4e-08)	(3.64)	(69.51)	33.44/33.44	33.15/33.15	(8e-15)	(0.004)	42.01/70.83	**0.81/0.81**	0.90/0.90	0.97/3.14
z400v3d5iw10-struc	(-)	(69.37)	25.28/-	40.70/40.70	40.28/40.28	(-)	(-)	(5.24)	**1.12/1.12**	**1.12/1.12**	1.84/7.05
z400v3d5iw20-rand	(1e-08)	(10.79)	(11.50)	(71.52)	(-)	(5e-18)	(0.0003)	(0.26)	**49.06/-**	(-)	(76.25)
z400v3d5iw20-struc	(-)	(0.27)	(23.80)	(24.23)	(-)	(-)	(-)	(7.21)	**14.73/15.58**	(-)	(89.09)
z400v6d5iw10-rand	(1e-14)	(0.27)	(-)	(-)	(-)	(1e-24)	(4e-08)	(24.33)	68.88/68.88	**68.81/68.82**	(72.04)
z400v6d5iw10-struc	(-)	(2e-11)	(-)	(-)	(-)	(-)	(6e-19)	(0.25)	**55.48/55.48**	55.68/55.68	(16.29)
z400v6d5iw20-rand	(3e-12)	**(0.80)**	(-)	(-)	(-)	(3e-25)	(1e-07)	(0.02)	(-)	(-)	(0.08)
z400v6d5iw20-struc	(-)	(3e-06)	(-)	(-)	(-)	(-)	(-)	**(0.02)**	(-)	(-)	(-)

Table B.16: Full results for exact algorithms on problem set `gen`. All algorithms were run for 100 CPU seconds. Summarized in Table 9.2 on page 109.

Instance	GLS$^+$ default			ILS default			s-BBMB					MB	HYBRID default		
	Solved	Quality avg	Time	Solved	Quality avg	Time	ib=2	ib=6	ib=10	ib=14	ib=18	MB	Solved	Quality avg	Time
alarm	25/25	100.00	0.0008	25/25	100.00	0.0008	**0.00/0.00**	**0.00/0.00**	0.01/0.01	**0.00/0.00**	**0.00/0.00**	**0.00/0.00**	25/25	100.00	0.0004
alarm-rand	25/25	100.00	0.0008	25/25	100.00	0.0008	0.02/0.02	**0.00/0.00**	**0.00/0.00**	**0.00/0.00**	**0.00/0.00**	**0.00/0.00**	25/25	100.00	0.0004
barley	25/25	100.00	0.53	25/25	100.00	0.51	(0.07)	1.21/2.91	1.21/2.89	1.21/2.94	1.28/5.38	**0.22/0.22**	25/25	100.00	0.66
barley-rand	**25/25**	**100.00**	**0.10**	25/25	100.00	0.10	28.69/-	1.23/1.23	1.22/1.22	1.22/1.22	1.23/1.23	0.20/0.20	25/25	100.00	0.10
diabetes	0/25	0.02	∞	0/25	4e-17	∞	(-)	4.23/4.23	**4.22/4.22**	4.24/4.24	4.23/4.23	4.70/4.70	25/25	100.00	25.87
diabetes-rand	0/25	67.78	∞	0/25	85.54	∞	(5e-17)	**3.80/3.80**	3.82/3.82	**3.80/3.80**	3.81/3.81	3.95/7.82	25/25	100.00	23.66
hailfinder	25/25	100.00	0.002	25/25	100.00	0.002	4.33/4.33	**0.00/0.00**	0.01/0.01	0.01/0.01	0.01/0.01	**0.00/0.00**	25/25	100.00	0.001
hailfinder-rand	25/25	100.00	0.003	25/25	100.00	0.003	(18.92)	0.01/0.01	0.01/0.01	**0.00/0.00**	**0.00/0.00**	0.01/0.01	25/25	100.00	0.003
insurance	25/25	100.00	0.0008	25/25	100.00	0.0008	**0.00/0.00**	**0.00/0.00**	**0.00/0.00**	**0.00/0.00**	**0.00/0.00**	0.01/0.01	25/25	100.00	0.0008
insurance-rand	25/25	100.00	0.0004	25/25	100.00	0.0004	**0.00/0.00**	**0.00/0.00**	**0.00/0.00**	**0.00/0.00**	**0.00/0.00**	0.01/0.01	25/25	100.00	0.0004
link	**25/25**	**100.00**	**1.06**	0/25	0.50	∞	(9e-37)	(6e-06)	(0.003)	17.10/-	17.29/-	50.05/50.05	25/25	100.00	13.23
link-rand	1/25	85.48	2497.44	0/25	42.88	∞	(3e-19)	(0.69)	(34.26)	**36.14/36.14**	58.57/58.57	(39.81)	0/25	57.75	∞
mildew	25/25	100.00	0.03	**25/25**	**100.00**	**0.03**	(1.39)	0.72/0.72	1.08/1.08	0.72/0.72	0.71/0.71	0.06/0.06	25/25	100.00	0.03
mildew-rand	25/25	100.00	0.004	25/25	100.00	0.004	28.67/31.74	0.01/0.01	0.01/0.01	0.01/0.01	0.01/0.01	**0.00/0.01**	25/25	100.00	0.004
munin1	25/25	100.00	0.21	25/25	100.00	0.21	1.03/-	1.17/2.20	24.93/24.93	(-)	(-)	**0.08/7.39**	25/25	100.00	0.22
munin1-rand	15/25	99.84	117.22	24/25	100.00	30.32	(0.03)	**1.23/13.96**	42.08/42.08	(-)	(-)	38.84/38.84	25/25	100.00	27.22
munin2	25/25	100.00	0.79	**25/25**	**100.00**	**0.78**	(-)	7.51/8.25	3.55/3.55	3.47/3.47	3.54/4.93	0.80/3.17	25/25	100.00	0.82
munin2-rand	0/25	97.43	∞	0/25	97.43	∞	(-)	(-)	(-)	(-)	(-)	**3.27/3.27**	25/25	100.00	12.59
munin3	25/25	100.00	0.84	25/25	100.00	0.82	(-)	28.07/28.72	4.51/5.88	4.51/4.51	4.54/5.77	0.85/3.63	**25/25**	**100.00**	**0.76**
munin3-rand	0/25	57.31	∞	0/25	68.20	∞	(-)	(-)	(-)	(-)	(-)	**4.05/4.05**	25/25	100.00	10.87
munin4	25/25	100.00	34.30	22/25	39.40	50.93	(-)	(0.15)	22.76/22.76	21.54/21.71	23.78/23.79	**18.70/18.70**	25/25	100.00	24.96
munin4-rand	0/25	7.54	∞	0/25	8.76	∞	(-)	(-)	(-)	(-)	(-)	**21.36/21.36**	25/25	100.00	77.69
pigs	25/25	100.00	0.11	**25/25**	**100.00**	**0.10**	(2e-05)	0.11/0.11	0.38/0.38	0.53/0.53	0.53/0.53	0.26/0.32	25/25	100.00	0.11
pigs-rand	25/25	100.00	30.46	25/25	100.00	7.13	(1e-08)	2.08/2.08	**0.41/0.41**	0.83/0.83	0.83/0.83	1.40/1.40	25/25	100.00	2.80
water	**25/25**	**100.00**	**0.003**	**25/25**	**100.00**	**0.003**	0.01/0.01	0.01/0.01	0.01/0.01	0.01/0.01	0.01/0.01	0.01/0.01	**25/25**	**100.00**	**0.003**
water-rand	25/25	100.00	0.06	25/25	100.00	0.05	1.54/1.54	**0.01/0.01**	0.17/0.17	0.17/0.17	0.17/0.17	0.19/0.19	25/25	100.00	0.56

Table B.17: Full results for best-performing algorithms on problem set `bnrep`. All algorithms were run 25 times for 100 CPU seconds each. Summarized in Table 9.3 on page 115.

	GLS$^+$			ILS			s-BBMB					Anytime	HYBRID		
	default			default									default		
Instance	Solved	Quality avg	Time	Solved	Quality avg	Time	ib=2	ib=6	ib=10	ib=14	ib=18	MB	Solved	Quality avg	Time
z100v3d5iw10-rand	25/25	100.00	0.02	**25/25**	**100.00**	**0.02**	(0.05)	4.17/73.11	0.07/0.07	0.08/0.08	0.08/0.08	0.07/0.07	25/25	100.00	0.02
z100v3d5iw10-struc	25/25	100.00	0.22	25/25	100.00	1.72	(-)	12.46/37.46	**0.16/0.16**	0.25/0.25	0.26/0.26	0.52/0.52	25/25	100.00	1.55
z100v3d5iw20-rand	25/25	100.00	0.24	25/25	100.00	0.24	(0.57)	(2.51)	0.51/0.51	5.38/5.38	35.46/35.46	55.16/55.16	**25/25**	**100.00**	**0.23**
z100v3d5iw20-struc	**25/25**	**100.00**	**0.44**	25/25	100.00	0.70	(-)	(0.67)	4.38/5.38	5.66/5.66	(-)	(5.49)	25/25	100.00	5.29
z100v6d5iw10-rand	**25/25**	**100.00**	**3.60**	23/25	99.43	48.17	(0.002)	(1.22)	8.71/8.71	8.81/8.81	8.77/8.78	52.58/92.28	25/25	100.00	26.63
z100v6d5iw10-struc	**25/25**	**100.00**	**0.47**	25/25	100.00	13.46	(-)	24.67/25.03	14.15/14.15	14.45/14.45	14.22/14.22	(-)	25/25	100.00	6.40
z100v6d5iw20-rand	**25/25**	**100.00**	**2.94**	2/25	63.71	1169.41	(0.0001)	(0.04)	(1.08)	(-)	(-)	(0.37)	25/25	100.00	35.91
z100v6d5iw20-struc	**25/25**	**100.00**	**1.43**	24/25	98.56	35.45	(1e-14)	(-)	33.82/42.56	(-)	(-)	(-)	25/25	100.00	14.67
z200v3d5iw10-rand	25/25	100.00	2.12	8/25	93.00	255.30	(2e-06)	(0.33)	1.02/1.02	0.49/0.49	**0.48/0.48**	1.04/1.04	25/25	100.00	3.99
z200v3d5iw10-struc	25/25	100.00	0.58	25/25	100.00	21.51	(-)	(-)	**0.37/0.37**	0.45/0.45	0.45/0.45	1.02/1.02	25/25	100.00	3.90
z200v3d5iw20-rand	25/25	100.00	52.55	24/25	99.84	29.00	(6e-07)	(0.07)	(5.15)	**7.01/7.01**	(-)	(37.15)	21/25	99.06	62.87
z200v3d5iw20-struc	**25/25**	**100.00**	**1.31**	17/25	85.42	88.18	(-)	(-)	(-)	8.13/8.13	(-)	(0.80)	25/25	100.00	16.77
z200v6d5iw10-rand	0/25	74.60	∞	0/25	30.01	∞	(9e-12)	(5e-07)	**35.48/36.08**	44.53/44.53	45.48/45.48	(95.99)	0/25	51.05	∞
z200v6d5iw10-struc	**25/25**	**100.00**	**3.06**	0/25	0.44	∞	(-)	(-)	40.22/40.23	64.87/64.87	64.78/64.78	92.90/92.90	25/25	100.00	26.63
z200v6d5iw20-rand	**25/25**	**100.00**	**54.85**	1/25	42.45	2458.55	(1e-16)	(6e-09)	(0.19)	(-)	(-)	(0.02)	4/25	50.96	605.54
z200v6d5iw20-struc	**21/25**	**88.74**	**54.32**	0/25	0.21	∞	(-)	(-)	(1e-07)	(-)	(-)	(1e-15)	7/25	52.29	368.62
z400v3d5iw10-rand	25/25	100.00	0.47	25/25	100.00	0.49	(8e-15)	(0.004)	42.01/70.83	0.81/0.81	0.90/0.90	0.97/3.14	**25/25**	**100.00**	**0.46**
z400v3d5iw10-struc	23/25	99.68	20.07	25/25	100.00	7.28	(-)	(-)	(5.24)	**1.12/1.12**	**1.12/1.12**	1.84/7.05	25/25	100.00	9.96
z400v3d5iw20-rand	0/25	87.97	∞	0/25	69.17	∞	(5e-18)	(0.0003)	(0.26)	**49.06/-**	(-)	(76.25)	0/25	71.46	∞
z400v3d5iw20-struc	11/25	92.84	161.18	0/25	25.42	∞	(-)	(-)	(7.21)	**14.73/15.58**	(-)	(89.09)	2/25	83.71	1210.31
z400v6d5iw10-rand	0/25	17.30	∞	0/25	1.15	∞	(1e-24)	(4e-08)	(24.33)	68.88/68.88	**68.81/68.82**	(72.04)	0/25	5.52	∞
z400v6d5iw10-struc	0/25	83.98	∞	0/25	1e-05	∞	(-)	(6e-19)	(0.25)	**55.48/55.48**	55.68/55.68	(16.29)	0/25	57.12	∞
z400v6d5iw20-rand	**0/25**	**29.79**	∞	0/25	8.49	∞	(3e-25)	(1e-07)	(0.02)	(-)	(-)	(0.08)	0/25	13.80	∞
z400v6d5iw20-struc	**0/25**	**69.53**	∞	0/25	0.03	∞	(-)	(-)	(0.02)	(-)	(-)	(-)	0/25	53.13	∞

Table B.18: Full results for best-performing algorithms on problem set `gen`. All algorithms were run 25 times for 100 CPU seconds each. Summarized in Table 9.4 on page 116.

Bibliography

[ACR03] David Applegate, William J. Cook, and Andre Rohe. Chained Lin-Kernighan for large traveling salesman problems. *INFORMS Journal on Computing*, 15(1):82–92, 2003.

[AFH$^+$04] Mirela Andronescu, Anthony P. Fejes, Frank Hutter, Holger H. Hoos, and Anne Condon. A new algorithm for RNA secondary structure design. *Journal of Molecular Biology*, 336(3):607–624, February 2004.

[Bib93] Wolfgang Bibel. *Wissensrepräsentation und Inferenz*. Vieweg, Wiesbaden, Germany, 1993. In German.

[BJ02] Francis R. Bach and Michael I. Jordan. Thin junction trees. In T. G. Dietterich, S. Becker, and Z. Ghahramani, editors, *Advances in Neural Information Processing Systems 13 (NIPS-01)*, pages 569–576. MIT Press, Cambridge, MA, USA, 2002.

[BSPV02] Mauro Birattari, Thomas Stützle, Luis Paquete, and Klaus Varrentrapp. A racing algorithm for configuring metaheuristics. In W. B. Langdon et al., editor, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2002)*, pages 11–18. Morgan Kaufmann Publishers, San Francisco, CA, USA, 2002.

[BVZ01] Yuri Boykov, Olga Veksler, and Ramin Zabih. Fast approximate energy minimization via graph cuts. *Pattern Analysis and Machine Intelligence*, 23(11), 2001.

[CDLS99] Robert G. Cowell, A. Philip Dawid, Steffen L. Lauritzen, and David J. Spiegelhalter. *Probabilistic Networks and Expert Systems*. Statistics for Engineering and Information Science. Springer, 1999.

[dBSD01] Matthijs den Besten, Thomas Stützle, and Marco Dorigo. Configuration of iterated local search: An example application to the single machine total weighted tardiness problem. In W. Egbert Boers,

J. Gottlieb, P.L. Lanzi, R.E. Smith, S. Cagnoni, E. Hart, G.R. Raidl, and H. Tijink, editors, *Applications of Evolutionary Computing*, pages 441–451. Springer Verlag, 2001.

[Dec96] Rina Dechter. Bucket elimination: A unifying framework for probabilistic inference. In *Proceedings of the Twelfth Conference on Uncertainty in Artificial Intelligence (UAI'96)*, pages 211–219. Morgan Kaufmann Publishers, San Francisco, CA, USA, 1996.

[DEKM98] Richard Durbin, Sean R. Eddy, Anders Krogh, and Graeme Mitchison. *Biological Sequence Analysis : Probabilistic Models of Proteins and Nucleic Acids*. Cambride University Press, Cambride, UK, 1998.

[DKL01] Rina Dechter, Kalev Kask, and Javier Larrosa. A general scheme for multiple lower bound computation in constraint optimization. In *Principles and Practice of Constraint Programming (CP'01)*, pages 346–360, 2001.

[DR03] Rina Dechter and Irina Rish. Mini-buckets: A general scheme for bounded inference. *Journal of the ACM*, 50(2):107–153, 2003.

[FH04] Pedro F. Felzenszwalb and Daniel P. Huttenlocher. Efficient belief propagation for early vision. In *Conference on Computer Vision and Pattern Recognition (CVPR-04)*, pages 261–268. IEEE Computer Society, Washington, DC, USA, 2004.

[Hec90] David Heckerman. A tractable inference algorithm for diagnosing multiple diseases. In *Proceedings of the Sixth Conference on Uncertainty in Artificial Intelligence (UAI'90)*. Morgan Kaufmann Publishers, San Francisco, CA, USA, 1990.

[HKT95] Te C. Hu, Andrew B. Kahng, and Chung-Wen A. Tsao Tsao. Old bachelor acceptance: A new class of non-monotone threshold accepting methods. *ORSA Journal on Computing*, 7(4):417–425, 1995.

[HND04] Frank Hutter, Brenda Ng, and Richard Dearden. Incremental thin junction trees for dynamic Bayesian networks. Technical report, Intellectics Group, Darmstadt University of Technology, Germany, 2004.

[Hoo98] Holger H. Hoos. *Stochastic Local Search — Methods, Models, Applications*. PhD thesis, TU Darmstadt, FB Informatik, Darmstadt, Germany, 1998.

[Hoo99] Holger H. Hoos. On the run-time behaviour of stochastic local search algorithms for SAT. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI'99)*, pages 661–666. AAAI Press / The MIT Press, Menlo Park, CA, USA, 1999.

[HS99] Holger H. Hoos and Thomas Stützle. Towards a characterisation of the behaviour of stochastic local search algorithms for SAT. *Artificial Intelligence*, 112:213–232, 1999.

[HS04] Holger H. Hoos and Thomas Stützle. *Stochastic Local Search - Foundations & Applications*. Morgan Kaufmann Publishers, San Francisco, CA, USA, 2004.

[HTH02] Frank Hutter, Dave A.D. Tompkins, and Holger H. Hoos. Scaling and probabilistic smoothing: Efficient dynamic local search for SAT. In P. Van Hentenryck, editor, *Principles and Practice of Constraint Programming (CP'02)*, volume 2470 of *Lecture Notes in Computer Science*, pages 233–248. Springer Verlag, Berlin, Germany, 2002.

[IC02] Jaime S. Ide and Fabio G. Cozman. Random generation of Bayesian networks. In *Proceedings on 16th Brazilian Symposium on Artificial Intelligence (SBIA-02), Advances in Artificial Intelligence*, pages 366–375. Springer Verlag, Berlin, 2002.

[IC03] Jaime S. Ide and Fabio G. Cozman. Generation of random Bayesian networks with constraints on induced width, with applications to the average analysis od d-connectivity, quasi-random sampling, and loopy propagation. Technical report, University of So Paulo, Brazil, 2003.

[JA90] Frank Jensen and Stig Andersen. Approximations in Bayesian belief universes for knowledge based systems. In *Proceedings of the Sixth Conference on Uncertainty in Artificial Intelligence (UAI'90)*, pages 162–169. Morgan Kaufmann Publishers, San Francisco, CA, USA, 1990.

[JJ99] Tommi S. Jaakkola and Michael I. Jordan. Variational probabilistic inference and the QMR-DT network. *Journal of Artificial Intelligence Research*, 10:291–322, 1999.

[JLO90] Finn V. Jensen, Steffen L. Lauritzen, and Kristian G. Olesen. Bayesian updating in causal probabilistic networks by local computations. *Computational Statistics Quaterly*, 4:269–282, 1990.

[JM02] David S. Johnson and Lyle A. McGeoch. Experimental analysis of heuristics for the STSP. In G. Gutin and A. Punnen, editors, *The Traveling Salesman Problem and its Variations*, pages 369–443. Kluwer Academic Publishers, Dordrecht, The Netherlands, 2002.

[KD96] Kalev Kask and Rina Dechter. A graph-based method for improving gsat. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI'96)*, pages 350–355. AAAI Press / The MIT Press, Menlo Park, CA, USA, 1996.

[KD99a] Kalev Kask and Rina Dechter. Branch and bound with mini-bucket heuristics. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI'99)*, pages 426–435. Morgan Kaufmann Publishers, San Francisco, CA, USA, 1999.

[KD99b] Kalev Kask and Rina Dechter. Stochastic local search for Bayesian networks. In *Proceedings of the 7th International Workshop on Artificial Intelligence and Statistics (AISTATS-99)*, January 1999.

[KGV83] Scott Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science, Number 4598, 13 May 1983*, 220, 4598:671–680, 1983.

[Kjæ94] Uffe Kjærulff. Reduction of computational complexity in Bayesian networks through removal of weak dependences. In *Proceedings of the Tenth Conference on Uncertainty in Artificial Intelligence (UAI'94)*, pages 374–382. Morgan Kaufmann Publishers, San Francisco, CA, USA, 1994.

[Lar03] David Larkin. Approximate decomposition: A method for bounding and estimating probabilistic and deterministic queries. In *Proceedings of the Nineteenth Conference on Uncertainty in Artificial Intelligence (UAI'03)*, pages 346–353. Morgan Kaufmann Publishers, San Francisco, CA, USA, 2003.

[LK73] S. Lin and B.W. Kernighan. An effective heuristic algorithm for the travelling salesman problem. *Operations Research*, 21(2):498–516, 1973.

[LMS02] Helena Ramalhino Lourenco, Olivier Martin, and Thomas Stützle. Iterated local search. In F. Glover and G. Kochenberger, editors, *Handbook of Metaheuristics*, pages 321–353. Kluwer Academic Publishers, 2002.

[LS88] Steffen L. Lauritzen and David J. Spiegelhalter. Local computations with probabilities on graphical structures and their application to expert systems. *Journal of the Royal Statistical Society, Series B*, 50(2):157–224, 1988.

[LW99] Jürgen Lehn and Helmut Wegmann. *Einführung in die Statistik*. B.G.Teubner Stuttgart - Leipzig, 3rd edition, 1999. In German.

[MD04] Radu Marinescu and Rina Dechter. AND/OR tree search for optimization in graphical models. Submitted to CP-04, 2004.

[MKD03] Radu Marinescu, Kalev Kask, and Rina Dechter. Systematic vs. non-systematic algorithms for solving the MPE task. In *Proceedings of the Nineteenth Conference on Uncertainty in Artificial Intelligence (UAI'03)*, pages 394–402. Morgan Kaufmann Publishers, San Francisco, CA, USA, 2003.

[MMC98] Robert J. McEliece, David J. C. MacKay, and Jung-Fu Cheng. Turbo decoding as an instance of Pearl's "belief propagation" algorithm. *Journal on Selected Areas in Communications*, 16(2):140–151, February 1998.

[Mor93] Paul Morris. The breakout method for escaping from local minima. In *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI'93)*, pages 40–45. AAAI Press / The MIT Press, Menlo Park, CA, USA, 1993.

[MT00] Patrick Mills and Edward P. K. Tsang. Guided local search for solving SAT and weighted Max-SAT problems. In I. P. Gent, H. van Maaren, and T. Walsh, editors, *SAT2000 — Highlights of Satisfiability Research in the Year 2000*, pages 89–106. IOS Press, Amsterdam, The Netherlands, 2000.

[MWJ99] Kevin P. Murphy, Yair Weiss, and Michael I. Jordan. Loopy-belief propagation for approximate inference: An empirical study. In *Proceedings of the Fifteenth Conference on Uncertainty in Artificial Intelligence (UAI'99)*, pages 467–475. Morgan Kaufmann Publishers, San Francisco, CA, USA, 1999.

[Par02] James D. Park. Using weighted Max-SAT engines to solve MPE. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI'02)*, pages 682–687. AAAI Press / The MIT Press, Menlo Park, CA, USA, 2002.

[PD01] James D. Park and Adnan Darwiche. Approximating map using local search. In *Proceedings of the 17th Conference in Uncertainty in Artificial Intelligence*, pages 403–410. Morgan Kaufmann Publishers, San Francisco, CA, USA, 2001.

[Pea88] Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan-Kaufmann Series In Representation And Reasoning. Morgan Kaufmann Publishers, San Francisco, CA, USA, 1988.

[PS02] Luis Paquete and Thomas Stützle. An experimental investigation of iterated local search for coloring graphs. In S. Cagnoni, J. Gottlieb, E. Hart, M. Middendorf, and G. Raidl, editors, *Applications of Evolutionary Computing*, volume 2279 of *Lecture Notes in Computer Science*, pages 122–131. Springer Verlag, Berlin, Germany, 2002.

[RBM02a] Irina Rish, Mark Brodie, and Sheng Ma. Accuracy vs. efficiency trade-offs in probabilistic diagnosis. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI'02)*, pages 560–566. AAAI Press / The MIT Press, Menlo Park, CA, USA, 2002.

[RBM02b] Irina Rish, Mark Brodie, and Sheng Ma. Efficient fault diagnosis using probing. In *Proceedings of 2002 AAAI Spring Symposium on "Information Refinement and Revision for Decision Making: Modeling for Diagnostics, Prognostics, and Prediction"*, pages 16–23, 2002.

[RN03] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, USA, 2nd edition, 2003.

[SAHH02] Alena Shmygelska, Rosalia Aguirre-Hernandez, and Holger H. Hoos. An ant colony optimisation algorithm for the 2d HP protein folding problem. In *Proceedings of the 3rd International Workshop on Ant Algorithms (ANTS-02)*, pages 40–52. Springer Verlag, 2002.

[SD03] Solomon E. Shimony and Carmel Domshlak. Complexity of probabilistic reasoning in directed-path singly connected Bayes networks. *Artificial Intelligence*, 151:213 – 225, December 2003.

[SDA$^+$75] Edward H. Shortliffe, Randall Davis, Stanton G. Axline, Brice G. Buchanan, C. Cordell Green, and Stanley N. Cohen. Computer-based consultations in clinical therapeutics: Explanation and rule acquisition capabilities of the MYCIN system. *Computers and Biomedical Research*, 8:303–320, 1975.

[SH01] Thomas Stützle and Holger H. Hoos. Analysing the run-time behaviour of iterated local search for the travelling salesman problem. In P. Hansen and C. C. Ribeiro, editors, *Essays and Surveys on Metaheuristics*, pages 589–611. Kluwer Academic Publishers, Boston, MA, USA, 2001.

[SHS03] Kevin Smyth, Holger H. Hoos, and Thomas Stützle. Iterated robust tabu search for Max-SAT. In Y. Xiang and B. Chaib-draa, editors, *Advances in Artificial Intelligence, 16th Conference of the Canadian Society for Computational Studies of Intelligence (AI'03)*, volume 2671 of *Lecture Notes in Computer Science*, pages 129–144. Springer Verlag, Berlin, Germany, 2003.

[SKC94] Bart Selman, Henry A. Kautz, and Bram Cohen. Noise strategies for improving local search. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI'94)*, pages 337–343. AAAI Press / The MIT Press, Menlo Park, CA, USA, 1994.

[SLM92] Bart Selman, Hector J. Levesque, and David G. Mitchell. A new method for solving hard satisfiability problems. In Paul Rosenbloom and Peter Szolovits, editors, *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI'92)*, pages 440–446. AAAI Press / The MIT Press, Menlo Park, CA, USA, 1992.

[SW97] Yi Shang and Benjamin W. Wah. Discrete Lagrangian-based search for solving Max-SAT problems. In M. E. Pollack, editor, *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI'97)*, volume 1, pages 378–383. Morgan Kaufmann Publishers, San Francisco, CA, USA, 1997.

[TF03] Marshall F. Tappen and William T. Freeman. Comparison of graph cuts with belief propagation for stereo, using identical mrf parameters. In *Proceedings of the Ninth IEEE International Conference on Computer Vision (ICCV'03)*, volume 2, pages 900 – 906. IEEE Computer Society Press, 2003.

[TH04] Dave A. D. Tompkins and Holger H. Hoos. Warped landscapes and random acts of sat solving. In *Proceedings of the Eighth Interna-*

tional Symposium on Artificial Intelligence and Mathematics (AIMA-04), 2004. To appear.

[Vou97] Cristos Voudouris. *Guided Local Search for Combinatorial Optimization Problems*. PhD thesis, University of Essex, Department of Computer Science, Colchester, UK, 1997.

[VT99] Cristos Voudouris and Edward P. K. Tsang. Guided Local Search and its application to the travelling salesman problem. *European Journal of Operational Research*, 113(2):469–499, 1999.

[YW03] Chen Yanover and Yair Weiss. Approximate inference and protein-folding. In S. Thrun S. Becker and K. Obermayer, editors, *Advances in Neural Information Processing Systems 14 (NIPS-02)*, pages 1457–1464. MIT Press, Cambridge, MA, USA, 2003.

[YW04] Chen Yanover and Yair Weiss. Finding the m most probable configurations in arbitrary graphical models. In Sebastian Thrun, Lawrence Saul, and Bernhard Schölkopf, editors, *Advances in Neural Information Processing Systems 15 (NIPS-03)*. MIT Press, Cambridge, MA, USA, 2004.

[Zad83] Lotfi A. Zadeh. The role of fuzzy logic in the management of uncertainty in expert systems. *Fuzzy Sets and Systems*, 11:199–228, 1983.

[ZP94] Nevin L. Zhang and David Poole. A simple approach to Bayesian network computations. In *Advances in Artificial Intelligence, 10th Conference of the Canadian Society for Computational Studies of Intelligence (AI-94)*, Lecture Notes in Computer Science, pages 171–178. Springer Verlag, Berlin, Germany, 1994.

[ZS00] Hantao Zhang and Mark E. Stickel. Implementing the Davis-Putnam method. *Journal of Automated Reasoning*, 24(1/2):277–296, 2000.