University of British Columbia
Department of Computer Science
Technical Report TR-2008-16

Computer-Aided Design of High-Performance Algorithms

Holger H. Hoos
University of British Columbia
Department of Computer Science

24 December 2008

Abstract

High-performance algorithms play an important role in many areas of computer science and are core components of many software systems used in real-world applications. Traditionally, the creation of these algorithms requires considerable expertise and experience, often in combination with a substantial amount of trial and error. Here, we outline a new approach to the process of designing high-performance algorithms that is based on the use of automated procedures for exploring potentially very large spaces of candidate designs. We contrast this computer-aided design approach with the traditional approach and discuss why it can be expected to yield better performing, yet simpler algorithms. Finally, we sketch out the high-level design of a software environment that supports our new design approach. Existing work on algorithm portfolios, algorithm selection, algorithm configuration and parameter tuning, but also on general methods for discrete and continuous optimisation methods fits naturally into our design approach and can be integrated into the proposed software environment.

1 Introduction

High-performance algorithms can be found at the heart of many software systems; they often provide the key to effectively solving the computationally difficult problems encountered in the application areas in which these systems are deployed. Examples of such problems include scheduling, time-tabling, resource allocation, production planning and optimisation, computer-aided design and software verification. Many of these problems are $\mathcal{NP}$-hard and considered computationally intractable, meaning that in general, they cannot be solved by any polynomial-time algorithm. Many application-relevant $\mathcal{NP}$-hard optimisation problems are even inapproximable in the sense that not even good approximations to optimal solutions can be found in polynomial time. [1]

Nevertheless, these 'intractable' problems arise in practice, and finding good solutions to them in many cases tends to become more difficult as economic constraints tighten. For example, resource allocation problems are typically very easy to solve if there is an abundance of resources relative to the demands in a given situation. Conversely, as demands grossly exceed the resources available, there will not be any allocation that satisfies all demand and, slightly less obviously, this will typically be

[1] Here and in the remainder of this report, we make the common assumptions that $\mathcal{NP} \neq \mathcal{P}$ and, in the context of approximability results, $\mathcal{NP} \neq \mathcal{ZPP}$.

easy to demonstrate. It is between those extremes that the difficult cases arise, where the demands and available resources are balanced enough that finding a satisfying allocation or demonstrating that none exists becomes computationally very difficult. We note that in many real-world contexts, a natural tendency towards this critically constrained, computationally difficult case can be expected: The underconstrained case is typically economically wasteful, which provides an incentive for increasing the demands on resources (for example, by enlarging the customer base or by taking on more projects) or to reduce availability of resources (for example, by scaling back personnel or equipment allotment); the overconstrained case, on the other hand, typically corresponds to lost market opportunities and can cause substantial strain within an organisation, which provides an incentive to increase resource availability. Furthermore, growing awareness and concern about the limitations of natural resources, such as oil or natural gas, along with increased competition within larger, often global markets and an increasing trend towards just-in-time delivery of goods and services provide further incentives to find good solutions to large instances of computationally difficult problems as fast as possible.

In most (if not all) cases, the key to solving computationally challenging problems effectively lies in the use of heuristic algorithms, that is, algorithms that make use of heuristic mechanisms, whose effectiveness can be demonstrated empirically, yet remains inaccessible to the analytical techniques used for proving theoretical complexity results. (We note that our use of the term 'heuristic algorithm' includes methods without provable performance guarantees as well as methods that have provable performance guarantees, but nevertheless make use of heuristic mechanisms; in the latter case, the use of heuristic mechanisms often results in empirical performance far better than the bounds guaranteed by rigorous theoretical analysis.) The design of such effective heuristic algorithms is typically complicated by the fact that substantial benefits can arise from using multiple heuristic mechanisms, which may interact in complex, non-intuitive ways. For example, a DPLL-style complete solver for SAT (a prototypical $\mathcal{NP}$-complete problem with important applications in the design of reliable soft- and hardware) may use different heuristics for selecting variables to be instantiated and the values first explored for these variables, as well as heuristic mechanisms for managing and using logical constraints derived from failed solution attempts.

High-performance heuristic algorithms are not only crucial for solving $\mathcal{NP}$-hard problems, but also play an important role in dealing with large instances of easier problems, particularly when these need to be solved under tight time-constraints. For example, the problem of finding shortest paths in graphs, which arises prominently in the context of route planning in transportation networks (*e.g.*, using GPS navigation systems in cars), can be solved using Dijkstra's classical algorithm (Dijkstra, 1959) in time $O(n^2)$ in the worst case, where n is the number of vertices in the given graph, and in time $O(n \log n)$ for sparse graphs (see, *e.g.*, Sanders and Schultes, 2007). However, when dealing with real-world road networks, which can easily have more than one million nodes, the running time of Dijkstra's algorithm becomes prohibitive. Therefore, much effort has been invested in designing high-performance heuristic algorithms for solving the same problem, leading to procedures that solve realistic route planning problems up to one million times faster than Dijkstra's algorithm (Sanders and Schultes, 2007). Similarly, in the context of high-performance sorting algorithms, heuristic mechanisms (along with other techniques whose effectiveness is often primarily established empirically) are used to achieve substantial speedups over the methods found in textbooks on algorithms and in the research literature (see, *e.g.*, Bentley and McIlroy, 1993; Brodal et al., 2008).

For a number of reasons, which we will discuss in more detail in the following section, the design of effective heuristic algorithms, particularly algorithms for solving computationally hard problems, is a difficult task that requires considerable expertise and experience. By expertise, we refer to expert-level knowledge acquired, for example, in graduate-level university courses or from the research literature; this may include knowledge of generic, broadly applicable algorithmic techniques, such as stochastic local search or branch-and-cut methods, as well as of knowledge related to the specific problem for which an algorithm is to be developed or improved. By experience, on the other hand, we refer to insights and intuitions gained from actively working on effective heuristic algorithms, often under the guidance of a skilled and successful researcher or practitioner.

The knowledge gained from this experience is difficult to formalise and to communicate effectively; therefore, developing the ability to design effective heuristic algorithms typically not only takes years of dedicated work under the guidance of someone who is already an expert, but also involves a large amount of trial and error. In this respect, heuristic algorithm design resembles more a craft or an art than a scientific or engineering effort; due to the lack of formalised procedures, best practices and rigorous results, it is also often regarded with some suspicion in academia.

In the remainder of this report, we first discuss the traditional approach for designing high-performance algorithms and point out its shortcomings. Next, we outline a fundamentally different approach that is based on the idea of applying human expertise primarily for specifying possibly large combinatorial design spaces (*i.e.*, sets of algorithms for the given problem) that are subsequently searched for high-performance algorithms using powerful, automated procedures. We will explain the advantages of this computer-aided algorithm design approach over traditional design methods, followed by a high-level description of a (yet unrealised) software environment that supports computer-aided algorithm design procedures and their application. As will be argued in that section, there is a substantial amount of existing work that is applicable and likely to be useful in this context. Finally, we summarise the main differences between this novel approach and traditional methods for designing high-performance algorithms, and highlight its features and advantages.

The computer-aided algorithm design approach outlined in this report is conceptually related to existing work on automated algorithm selection, algorithm configuration, parameter tuning and algorithm portfolios (see, *e.g.*, Rice, 1976; Leyton-Brown et al., 2003; Birattari et al., 2002; Adenso-Diaz and Laguna, 2006; Hutter et al., 2007; Huberman et al., 1997; Gomes and Selman, 2001; Gagliolo and Schmidhuber, 2006), as well as to general-purpose discrete and continuous optimisation techniques (see, *e.g.*, Hansen and Ostermeier, 2001; Hansen and Kern, 2004; Bűrmen et al., 2006; Powell, 1998; Nelder and Mead, 1965). It has also connexions with work in algorithm synthesis (see, *e.g.*, Westfold and Smith, 2001; Van Hentenryck and Michel, 2007; Monette et al., 2009; Di Gaspero and Schaerf, 2007), algorithm engineering (see, *e.g.*, Sanders and Schultes, 2007; Maue and Sanders, 2007) and meta-learning (see, *e.g.*, Vilalta and Drissi, 2002), as well as work on so-called hyper-heuristics (see, *e.g.*, Cowling et al., 2002; Burke et al., 2003). Surveying these bodies of work in their entirety would is beyond the scope of this report, but we will point out the connexions and give prominent examples for techniques that can be directly applied in the context of our approach in Sections 3 and 4, while in Section 5, we will comment in more detail on the aforementioned research directions as they relate do (and differ from) our notion of computer-assisted algorithm design.

2 The traditional approach

High-performance heuristic algorithms are typically constructed in an iterative, manual process in which the designer gradually introduces or modifies components or mechanisms whose performance is then tested by empirical evaluation on one or more sets of benchmark problems. This process often starts with some generic or broadly applicable problem solving method (*e.g.*, an evolutionary algorithm), a new algorithmic idea or even an algorithm suggested by theoretical considerations. This initial algorithm is implemented and evaluated on some problem instances, after which heuristic mechanisms are added or modified. These additions and modifications typically happen in stages, with performance being assessed at each stage until a given performance target has been reached or exceeded (this target could be defined in terms of existing algorithms or be derived from real-world application requirements), or the algorithm designer has no more time to spend (*e.g.*, because of publication or project deadlines).

During this iterative design process, the algorithm designer has to make many decisions. These concern choices of the heuristic mechanisms being integrated and the details of these mechanisms, as well as implementation details, such as data structures. Some of these choices take the form of parameters, whose values are guessed or determined based on limited experimentation. Such parameters are often

not exposed to later users of the final version of the software, and in many cases are not clearly visible even at the source code level (*e.g.*, they may appear as numerical constants inside arithmetic expressions or logical conditions). Frequently, such hard-coded parameters are not mentioned in later descriptions or discussions of the algorithm (*e.g.*, in publications).

Various problems can and often do arise from this approach to algorithm design. The first of these stems from the fact that it is inherently labour-intensive: Even experienced designers often have to spend substantial amounts of time exploring and experimenting with various combinations of mechanisms and methods before obtaining an effective heuristic algorithm for a given computational problem. To a large extent, this work is being perceived as rather menial. Given this situation, human designers typically explore relatively few designs that are chosen in an *ad-hoc* manner rather than based on carefully planned experimental designs.

As a consequence, good designs, *i.e.*, high-performance algorithms that may be realisable given the components and mechanisms considered in the design process, may be easily missed. This applies especially to lengthy design processes during which many, possibly parameterised heuristic mechanisms are considered, which jointly define a vast combinatorial space of possible designs. Cognizant of this difficulty, algorithm designers tend to limit the size of the design space, *e.g.*, by choosing to neither expose nor vary certain parameters. While this indeed limits the effort involved in exploring the space of possible designs or configurations, and therefore increases the possibility of searching larger parts of that space, the arbitrary or *ad-hoc* nature of the choices made in this context (*e.g.*, hard-coded parameter values) can easily further reduce the performance potential realised at the end of the design process.

In many cases, the approach taken by human designers relies on implicit independence assumptions regarding the effectiveness of heuristic mechanisms or parameter settings. These assumptions essentially postulate that design choices can be made independently of each other; *e.g.*, the effects of a choice between two alternative mechanisms, say X and Y, in one part of a given algorithm, should be independent of those of choosing between three mechanisms, say A, B and C, in another part. Unfortunately, such independence assumptions are often incorrect, since the mechanisms interact, sometimes in unintuitive ways (consider, *e.g.*, the choice of variable- and value-ordering heuristics in a tree-search algorithm for SAT or CSP).

Similar problems arise from the tendency of human designers to over-generalise conclusions drawn from observations in the limited context of such experiments (*e.g.*, "using mechanism X rather than Y always results in better performance", where in reality this may only be true when also using mechanism A, but not when using B or C instead). Especially in the context of designing or optimising randomised algorithms, this tendency can easily lead to the perception of illusory patterns, and particularly in cases where there is an *a priori* expectation of a certain pattern or regularity, there is a risk of overestimating empirical evidence supporting that pattern while underestimating evidence to the contrary.

As a result, algorithm designers may choose (consciously or subconsciously) to focus on certain parts of a design space (*e.g.*, by committing to a heuristic mechanism or parameter setting that is subsequently never revisited) while leaving others unexplored. This tends to further limit the extent to which algorithms constructed in this manner realise the performance potential that is present in the underlying design space.

Finally, the iterative design process appears to be biased towards increasingly complicated designs: while sometimes, mechanisms or components whose addition has not led to performance improvements are removed (and then typically never considered again), often, unless they are seen to decrease performance, such mechanisms or components are kept in the design, while other mechanisms are added. This may be advantageous in cases where a mechanism A is relatively ineffective on its own, but leads to major performance gains in combination with other mechanisms $X_1, \ldots, X_k$ added at some later stage; however, it is problematic in cases where A remains ineffective throughout the design process or, worse, where in combination with $X_1, \ldots, X_k$, A's impact on overall performance actually decreases (*i.e.*, the performance of an algorithm using $X_1, \ldots, X_k$ and A is lower than that of the same algorithm without A — this may happen, *e.g.*, when mechanism A results in heuristic guidance that,

while ineffective on its own, partly counteracts the more effective guidance provided by X_i).

This bias in the design process leads to complicated, historically grown algorithms with many components, not all of which may be effective. Further development building on such algorithms rarely includes an assessment of the efficacy of components and mechanisms introduced in the original design, and therefore, over the course of multiple design cycles, the complexity tends to increase further. The resulting algorithms effectively are the result of very limited search in extremely large design spaces; and while in principle, larger design spaces can be expected to contain better solutions (in this case, better-performing algorithms), we would also expect the gap between the quality of the best solutions existing in a given space and those found by an ineffective search procedure to widen with increasing size of the design space.

In summary, while the traditional approach can and often does lead to satisfactory results, it tends to be tedious and labour-intensive; the resulting algorithms are often unnecessarily complicated while failing to realise the full performance potential present in the space of designs that can be built using the same underlying set of components and mechanisms.

3 The new approach

As an alternative to the traditional, manual algorithm design process outlined in the previous section, we propose an approach that uses fully formalised procedures, implemented in software, to permit a human designer to explore large design spaces more effectively, with the aim of realising algorithms with desirable performance characteristics. Since these procedures operate on algorithms, we characterise them as *meta-algorithms*, while we refer to the algorithms being constructed or modified as *target algorithms*.

In its simplest form, our approach, which may be called *computer-aided design of high-performance algorithms*, can be outlined as follows.

A human designer

- specifies a (possibly very large) design space, *i.e.*, a set of target algorithms, by means of a collection of components (which may be parameterised) and one or more generic ways of combining these components into an algorithm for the problem at hand (this may take the form of an instantiable schema or of a family of such schemata);
- supplies a set of problem instances to be used for performance evaluation (this set could comprise only one instance; a distribution of instances, *i.e.*, set of instances associated with probabilities; or an instance generator);
- specifies a performance metric which is used to assess candidate target algorithm designs when run on given problem instances (*e.g.*, median run-time or mean relative solution quality).

Based on this input, the meta-algorithmic system explores the design space in order to find target algorithms with highest possible performance on the given problem instances. For the final assessment of the target algorithm(s) thus obtained, in some cases a disjoint set of test problem instances may be used.

Note that this approach automates both, the construction of target algorithms as well as the assessment of their performance. By specifying the design space to be explored in an appropriate manner, computer-aided algorithm design can be used to construct algorithm portfolios, which consist of several target algorithms running concurrently in order to achieve increased performance robustness (see, *e.g.*, Huberman et al., 1997; Gomes and Selman, 2001); per-instance algorithm selection methods, which choose one of several target algorithms to be applied to a given problem instance based on properties of that instance determined just before attempting to solve it (see, *e.g.*, Rice, 1976; Leyton-Brown et al.,

2003); or reactive algorithms, which may switch between different heuristic mechanisms or different parameter configurations while running on a given problem instance (see, *e.g.*, Battiti and Protasi, 1997; Battiti et al., 2008).

More complex variants of computer-aided algorithm design may involve

- additional automated algorithm simplification phases, in which meta-algorithmic procedures are used to simplify a target algorithm as much as possible without incurring inacceptable performance losses (where what constitutes an 'acceptable loss' is specified by the algorithm designer) — such simplification mechanisms could also be integrated into the exploration procedure;
- iterative modifications of the design space (*e.g.*, by means of adding or changing algorithm components or through modifications of the instantiable schema), interspersed with automated exploration phases;
- iterative modifications of the set of problem instances used for performance evaluation (*e.g.*, in some cases, it may be useful to use relatively easy problem instances during an initial exploration stage, and increasingly more difficult instances for subsequent refinements);
- multiple performance criteria, in which case the meta-algorithmic procedure used for exploring the design space would aim to solve a multi-objective optimisation problem.

Computer-aided algorithm design allows human designers to focus on the creative task of specifying a design space in terms of potentially useful components. While some of these components will be specific to the problem to be solved by the target algorithm or even certain types of problem instances that are characteristic for a given application context, other components may be generic and reusable. Examples for such reusable components include generic local search procedures (*e.g.*, iterative first improvement), neighbourhood relations (*e.g.*, $k-$exchange) and generic look-ahead procedures that can be used in the context of construction or tree search methods. Such components could be collected in libraries, the use of which would help algorithm designers in specifying larger design spaces.

The use of powerful heuristic search and optimisation procedures in combination with significant amounts of computing power, which in many industry and research contexts is quite readily available, enables the exploration of larger design spaces. Larger design spaces have the potential to contain better-performing algorithms. However, in most cases, the problem of finding optimal or near-optimal designs within a large space can be expected to be computationally hard (for typical, combinatorial design spaces: $\mathcal{NP}$-hard); therefore, given limited computational resources, enlarging a given design space, even if this increases the quality of the optimal designs contained therein, may not result in practical gains, since these improved designs may never be found. On the other hand, an enlarged space may contain sufficiently many or easily reachable good designs that a better-performing algorithm can be found effectively.

Clearly, this tradeoff depends on the interactions between the meta-algorithmic procedures used for exploring the design space and characteristics of the space itself. As is the case in the context of solving other hard combinatorial problems effectively, we expect the experience and intuitions of the human algorithm designer to be key to achieving good results in this context. In this context, human skills are applied at the meta-level and leveraged by the automated stages of the computer-aided algorithm design process. We believe that this way of leveraging human skills yields far superior results compared to the direct application of the same abilities to the manual construction of target algorithms, analogous to the advantages realised by using human effort and ingenuity for making and effectively using versatile tools, rather than by applying human skills directly to the task of manufacturing a specific artifact.

Another useful feature of the computer-aided algorithm design approach is that it supports the construction of algorithms for specific types of instances of a given problem in a rather straight-forward way. For many problems, different types of solution strategies are known to work best for solving various kinds of instances. As the performance potential inherent to a class of algorithms, which may be represented by a given combinatorial design space, is increasingly exploited, it is reasonable to assume

that in addition to fairly instance-independent choices (*i.e.*, choices that would contribute to performance improvements on a broad class of problem instances), also components or configurations would be selected whose effectiveness depends on characteristics of the problem instances considered in the design process. Herein lies an additional pitfall of the traditional design process, which may encourage design decisions that really only work well in the limited context of the typically very small number of problem instances considered at a particular stage of the process.

In the computer-aided design process outlined here, the major part of the human effort involved is spent in the context of specifying design spaces, while the search for high-performance designs for a given set of problem instances is automated. Therefore, given a sufficiently rich design space and enough computing time, high-performance algorithms for specific classes of problem instances can be obtained relatively cheaply and in a straightforward way. Depending on the degree to which two given sets of problem instances, say I and J, differ from each other, the effort spent in finding a good algorithm $A(I)$ for solving I, may even be leveraged effectively by starting the search for a high-performance algorithm $A(J)$ for J from the design of $A(I)$. Related to this idea, in many real-world application scenarios, after an initial algorithm has been found and deployed, it may be useful to collect the problem instances solved by that algorithm and occasionally re-optimise the target algorithm using these instances (possibly in addition to ones used earlier in the algorithm design process). This way, the target algorithm could be automatically adapted to changing application requirements on an ongoing basis — a concept that is closely related to the idea of *life-long learning* (see, *e.g.*, Thrun, 1996).

Finally, computer-aided algorithm design can result in simpler target algorithms, because the use of meta-algorithmic procedures makes it possible and likely to explore larger parts of a given design space, while avoiding the previously discussed tendency of incremental, manual approaches to produce complex designs. Furthermore, the use of automated simplification procedures (which, in the simplest case, may perform simple abrasion analysis) makes it possible to explicitly search for simple designs with few components and heuristic mechanisms. Such designs are easier to understand and to explain; they are also more amenable to manual performance improvements by means of specially engineered data structures and other implementation enhancements.

Our approach shares much of its motivation with existing work on automated parameter tuning, algorithm configuration, algorithm portfolios and algorithm selection, all of which can be seen as special cases of computer-aided algorithm design.

In automated parameter tuning, the design space is defined by an algorithm whose behaviour is controlled by a set of parameters, and the task is to find performance-optimising settings of these parameters. Depending on the number of algorithm parameters and the nature of their domains, various methods can be used for exploring the resulting design spaces, ranging from well-known numerical optimisation procedures such as the Nelder-Nead Simplex algorithm (Nelder and Mead, 1965; Bűrmen et al., 2006) or the more recent, gradient-free CMA-ES algorithm (Hansen and Ostermeier, 2001; Hansen and Kern, 2004) to inherently discrete approaches based on experimental design methods (see, *e.g.*, Birattari et al., 2002; Balaprakash et al., 2007; Adenso-Diaz and Laguna, 2006), response-surface models (see, *e.g.*, Jones et al., 1998; Bartz-Beielstein, 2006) or stochastic local search procedures (see, *e.g.*, Hutter et al., 2007, 2008).

In automated algorithm configuration, the design space is defined by an algorithm schema that contains a number of instantiable components (typically, subprocedures or functions), along with a discrete set of concrete choices for each of these. This can be seen as a special case of parameter tuning, in which categorical parameters are used to select a set of components to instantiate the given schema. However, the nature of the respective optimisation problems is very different from those arising, *e.g.*, when tuning a small number of real-valued algorithm parameters, and in fact, several of the previously mentioned methods are not applicable. However, F-Race (Birattari et al., 2002; Balaprakash et al., 2007), Calibra (Adenso-Diaz and Laguna, 2006) and ParamILS (Hutter et al., 2007, 2008) have been used successfully in this context (although of these, so far only ParamILS has been demonstrated to be able to deal with the vast design spaces resulting from schemata with many, independently instantiable components), and promising results have been achieved by a genetic programming procedure applied to the configuration

of local search algorithms for SAT (Fukunaga, 2002, 2004).

In the case of algorithm portfolios (see, *e.g.*, Huberman et al., 1997; Gomes and Selman, 2001), the design space considered consists of sets of algorithms that are run concurrently on a given problem instance. In general, the components algorithms that make up the portfolio can be assigned different CPU shares in a multi-tasking environment; furthermore, in the case of randomised algorithms, it can be beneficial to include multiple copies of the same component algorithm. Extensions of this concept may allow communication between the component algorithms (as mentioned, but not explored by Huberman et al., 1997), or may dynamically re-allocate CPU shares between component algorithms (see, *e.g.*, Gagliolo and Schmidhuber, 2006). Given a set of component algorithms, the problem of determining performance-optimising (or risk-minimising) allocations of CPU shares can be seen as a parameter tuning problem; however, because of its special properties, it is likely that this problem is best solved using specific numerical optimisation methods. The problem of selecting component algorithms to be included in a portfolio can be seen as an algorithm configuration problem, but again, may be solvable most effectively using techniques that exploit special properties of the component selection task. [2]

Per-instance algorithm selection methods (see, *e.g.*, Rice, 1976; Leyton-Brown et al., 2003) exploit differences in the relative performance between various algorithms. Given a problem instance π to be solved, the idea is to select among several candidate algorithms the one that can be expected to give the best performance on π; this is done based on knowledge on the correlation of algorithm performance with instance features. This corresponds to an algorithm design problem in which the design space consists of selection functions, which map instance features to candidate algorithms. In recent work, performance-optimising selection functions are determined using statistical classification or regression procedures (see, *e.g.*, Guerri and Milano, 2004; Leyton-Brown et al., 2003). Advanced algorithm selection methods may run more than one algorithm on a given problem instance (see, *e.g.*, Xu et al., 2007, 2008), or dynamically switch between algorithms (see, *e.g.*, Carchrae and Beck, 2005); the corresponding design spaces and methods for determinining 'good designs' are conceptually related to those previously mentioned.

It is not hard to see how more general computer-aided algorithm design approaches can in principle combine aspects of and procedures for automated parameter tuning, algorithm configuration, algorithm portfolios and per-instance algorithm selection. In the case of stochastic local search (SLS), a fundamental and widely used approach for solving hard combinatorial problems, design spaces spanning such combinations can be defined based on Generalised Local Search Machines (GLSMs). GLSMs are a general formalism for modelling SLS algorithms that is based on a clean separation between lower-level search procedures and higher-level search control strategies (Hoos, 1999; Hoos and Stützle, 2004). As will be discussed in detail elsewhere, GLSMs can be used to structure large and complex spaces of SLS algorithms, and existing work on automated parameter tuning, algorithm configuration as well as on general discrete and continuous optimisation methods will likely provide a good basis for automatically exploring these spaces.

We also note that when using appropriately chosen design spaces, computer-aided algorithm design can yield algorithms with provable performance guarantees. As a trivial example, consider portfolio constructions that run two or more algorithms for a given optimisation problem Π concurrently and return the best solution found by any of them. Let A_1 be an algorithm for Π that is provably guaranteed to reach a certain approximation ratio after some runtime, say $t(\pi)$, where π is the given problem instance. Clearly, for k component algorithms, all with equal CPU share, any portfolio containing A_1 is guaranteed to achieve the same approximation ratio as A_1 in time $k \times t(\pi)$. Similarly, any sequential algorithm that first runs A_1 for $t(\pi)$, then executes a stochastic local search procedure initialised at the candidate solution produced by A_1, and finally returns the best solution encountered, is also guaranteed to achieve the same approximation ratio as A_1. While these simple examples illustrate the feasibility of the concept, there certainly exist more complex ways of specifying design spaces in which all algorithms have provable performance guarantees by construction.

[2] It may be noted that neither Huberman et al. (1997) nor Gomes and Selman (2001) present automated methods for portfolio construction, while Fukunaga (2000) uses a bootstrap sampling method for portfolio optimisation.

4 A software environment for computer-aided design of high-performance algorithms

In order to apply computer-aided algorithm design methods in practice, various software components need to be made available to a human designer. In the simplest case, these could be comprised of a parametric target solver along with an implementation of a meta-algorithmic optimisation procedure, where the former is an implementation of a family of target algorithms, whose members are obtained by means of setting exposed parameters to certain values, while the latter searches the configuration space defined by these parameters for performance-optimising settings. This search process involves selecting candidate configurations of the parametric target solver as well as measuring the performance observed when running these configuration of the target solver on the given problem instances.

As previously mentioned, many existing procedures could in principle be used for these parameter tuning or algorithm configuration tasks; depending on the nature and number of parameters of the given target algorithm, procedures such as Nelder-Mead Simplex (Bűrmen et al., 2006; Nelder and Mead, 1965), CMA-ES (Hansen and Kern, 2004; Hansen and Ostermeier, 2001), F-Race (Birattari et al., 2002; Balaprakash et al., 2007) or ParamILS (Hutter et al., 2007, 2008) might be utilised in this context. However, existing implementations of these procedures use different formats for specifying the parameter space, which complicates their use in the context of more general computer-aided algorithm design tasks that may benefit from or require the use of more than one meta-algorithmic optimisation procedure (*e.g.*, during sequential design stages).

However, we believe that computer-aided design of high-performance algorithms is best supported by a software environment that goes much beyond the application of a single parameter tuning and algorithm configuration procedure. We envision a system that integrates

- various approaches and formalisms for specifying algorithm design spaces, such as algorithm portfolios (see, *e.g.*, Huberman et al., 1997; Gomes and Selman, 2001), per-instance algorithm selectors (see, *e.g.*, Guerri and Milano, 2004; Leyton-Brown et al., 2003; Xu et al., 2008) or Generalised Local Search Machines (Hoos, 1999; Hoos and Stützle, 2004);
- a collection of methods for exploring design spaces, including algorithm configuration and paramater tuning procedures (see, *e.g.*, Hutter et al., 2008, 2007; Balaprakash et al., 2007; Bartz-Beielstein, 2006), continuous optimisation methods (see, *e.g.*, Bűrmen et al., 2006; Nelder and Mead, 1965; Hansen and Kern, 2004; Hansen and Ostermeier, 2001), as well as specialised procedures for constructing algorithm portfolios or selectors (see, *e.g.*, Gagliolo and Schmidhuber, 2006; Guerri and Milano, 2004; Leyton-Brown et al., 2003; Xu et al., 2008; Carchrae and Beck, 2005);
- support for a variety of performance metrics, including (descriptive statistics over) run-time as well as solution quality achieved after fixed run-time;
- support for managing sets of benchmark instances;
- support for managing and interpreting experimental results, which should also include mechanisms that avoid unnecessary duplication of algorithm runs;
- methods for automated algorithm simplification, including simple abrasion analysis as well as more complex approaches that support trading off performance against (description) complexity of algorithms.

Furthermore, we believe that the following features are especially important when designing such a system:

- component-based architecture with simple, clearly defined and well-documented interfaces between all components — this facilitates concurrent and independent development of components;

- clear separation between the user interface (which we envision to be web-based, possibly using Java applets) and other components of the system, in particular, the meta-algorithmic procedures used for algorithm construction, configuration, tuning and simplification — this makes it possible to separate the development and integration of new components to a large degree from work on the user interface;
- effective utilisation of parallel computation in the form of multi-core and multi-processor systems, compute clusters and possibly large-scale distributed computation in the form of CPU cycle-scavenging (*cf.* SETI@home) — given the compute-intensive nature of most computer-aided algorithm design tasks and the trend towards parallel computing environments, this is an indispensable requirement.

To be widely used, such a system would need to be easy to install and run on a range of commonly used compute platforms, in particular, on MS-Windows- and Unix-based systems (including Mac OS X). Furthemore, it should make it easy to perform conceptually simple tasks, such as parameter tuning or configuration of a given solver (provided as an executable) on a set of benchmark instances.

Finally, such a software environment for computer-aided algorithm design could and should make use of the functionality afforded by widely used utilities such as Sun Grid Engine (or similar distributed computing environments), mySQL (or similar database systems), CVS (or similar version control systems), R (or similar statistics packages) and Gnuplot (or similar graph plotting software). We see also potential for integration with software development environments, such as Eclipse, in the context of support for target algorithm component development and management, as well as for design space specification.

Based on these considerations, we are currently designing a system called *High-performance Algorithm Laboratory (HAL)*, which will support both, computer-aided design of high-performance algorithms, as well as empirical analysis of such algorithms (where the latter includes comparative performance analysis, performance scaling analysis, robustness analysis and parameter response analysis). A first version of HAL will likely support design space specifications in the form of black-box parametric target solvers, along with a number of meta-algorithmic optimisation procedures, including ParamILS (Hutter et al., 2007, 2008), F-Race (Birattari et al., 2002), Iterated F-Race (Balaprakash et al., 2007), Active Configurator and likely also SPO (Bartz-Beielstein, 2006), as well as one or two gradient-free continuous optimisation procedures, such as Nelder-Mead Simplex (Bűrmen et al., 2006; Nelder and Mead, 1965) or CMA-ES (Hansen and Kern, 2004; Hansen and Ostermeier, 2001). It will also provide some support for managing sets of benchmark instances and for archiving experimental results.

In subsequent versions, we plan to additionally support algorithm design patterns based on GLSMs as well as on per-instance algorithm selectors and algorithm portfolios; an automated simplification procedure (dubbed *Razor*); and finally, tools for target algorithm performance analysis based on widely used concepts and methods from empirical algorithmics, such as solution quality distributions, run-time distributions, solution cost distributions and parameter response curves (see, *e.g.*, Hoos and Stützle, 2004, Ch. 4).

5 Related work

Work on automated algorithm selection, algorithm configuration, parameter tuning and algorithm portfolios is motivated to a large extent by considerations similar to those presented here (see, *e.g.*, Rice, 1976; Leyton-Brown et al., 2003; Guerri and Milano, 2004; Carchrae and Beck, 2005; Hutter et al., 2007; Adenso-Diaz and Laguna, 2006; Huberman et al., 1997; Gomes and Selman, 2001; Gagliolo and Schmidhuber, 2006); focussed on particular design techniques, it is typically more limited in scope, but has very useful applications in the broader context of our more general approach. Work on general-purpose optimisation techniques (see, *e.g.*, Powell, 1998; Hansen and Kern, 2004; Hansen and Os-

termeier, 2001; Bűrmen et al., 2006; Nelder and Mead, 1965) is similarly directly applicable in the context of our approach, but is typically focussed on solving problems with characteristics that can differ considerably from those arising in the context of searching for high-performance algorithms in large, combinatorial design spaces.

Existing work on algorithm synthesis is mostly focussed on automatically generating algorithms that satisfy a given formal specification or that solve a specific problem from a large and diverse domain (see, *e.g.*, Westfold and Smith, 2001; Van Hentenryck and Michel, 2007; Monette et al., 2009; Di Gaspero and Schaerf, 2007), while the approach we consider is focussed on performance optimisation in a large space of candidate solvers for a given problem that are all guaranteed to be correct by construction. Clearly, there is complementarity between both approaches; at the same time, because of the significant difference in focus, the methods considered in algorithm synthesis and performance-oriented computer-assisted algorithm design, as considered here, are quite different.

Work on algorithm engineering is mostly focussed on the empirical performance of algorithms for polytime-solvable problems, and involves manual exploration of relatively modest numbers of algorithm variants, obtained by including various heuristic mechanisms and speed-up techniques (see, *e.g.*, Sanders and Schultes, 2007; Maue and Sanders, 2007). Our approach, on the other hand, deals mostly with performance-optimisation by means of search in much larger combinatorial design spaces; it is, in principle, applicable to problems of arbitrary computational complexity, but expected to be most effective on $\mathcal{NP}$-hard problems. Again, there is complementarity between the two research directions; *e.g.*, work in algorithm engineering could benefit from the meta-algorithmic procedures used in our approach for searching large design spaces, while the theoretical insights that often guide particular algorithm engineering efforts may be very useful in the context of defining design spaces in our approach.

In the machine learning community, meta-algorithmic techniques are used for improving the performance of machine learning procedures. This area is commonly referred to as *meta-learning*, and the meta-algorithmic procedures used in this context are typically based on machine learning approaches (see, *e.g.*, Vilalta and Drissi, 2002); examples include stacked generalisation and dynamic bias selection as well as, under a broad definition of meta-learning, model-combination techniques, such as bagging and boosting. Meta-learning can be seen as a special case of computer-assisted algorithm design in that, while being motivated very similarly, it is more restricted in scope. (At the same time, meta-learning addresses issues well outside of the context considered here.) Doubtlessly, commonly used machine learning procedures constitute a very important class of algorithms that can benefit enormously from computer-assisted algorithm design methods, and the specific methods used in meta-learning may well be useful in the context of designing other types of algorithms. At the same time, it is likely that meta-algorithmic procedures different from those already used in meta-learning may prove to be useful in the design of high-performance machine learning algorithms.

Another area in which approaches that fall under the umbrella of computer-aided algorithm design can be found is evolutionary computation. In a subarea known as *genetic programming (GP)* (see, *e.g.*, Poli et al., 2008), evolutionary algorithms are typically used to construct algorithms (or related formalisms, such as automata or functions) with specific properties. For example, Fukunaga (2002, 2004) used an evolutionary algorithm to search for high-performance SAT algorithms in an unbounded space of stochastic local search algorithms for SAT. Similarly, Bölte and Thonemann (1996) optimised the performance of a simulated annealing for the quadratic assignment problem by means of evolving annealing schedules. In genetic programming, the specification of design spaces and the evolutionary algorithms used for exploring these spaces are closely linked; furthermore, the design spaces are typically unbounded, and the target algorithms comprising these spaces are typically represented as trees. (However, there is now an increasing amount of work that uses other representations). Considering its focus on meta-algorithmic search procedures based on evolutionary algorithms and on unbounded design spaces, when applied to the design of high-performance algorithms, genetic programming can be seen as a special case of computer-aided algorithm design. Although there are contexts (such as the algorithm configuration and parameter tuning problems mentioned previously) in which other

computer-aided algorithm design methods may be considerably more effective than standard genetic programming techniques, we expect the latter to be useful for exploring potentially unbounded, highly structured design spaces, as encountered, for example, in the context of constructing GLSM-based designs of stochastic local search algorithms.

Work on so-called *hyper-heuristics* (see, *e.g.*, Cowling et al., 2002; Burke et al., 2003) and on *reactive search methods* (see, *e.g.*, Battiti and Protasi, 1997; Battiti et al., 2008) is also motivated by the goal of building high-performance algorithms using generic methods. However, this work is largely orthogonal to our approach, since it is typically focussed on using traditional design methods to design algorithms that use some higher-level control strategy in order to modify the behaviour of a given algorithm at run-time. Specific methods from both areas can, however, give rise to interesting design spaces; these spaces could then be searched for high-performance hyper-heuristics or reactive search methods using standard meta-algorithmic optimisation methods.

The feasibility and potential of using automated methods for constructing high-performance algorithms has been demonstrated in various application contexts. For example, Li et al. (2005) have used a genetic programming approach to create hybrid sorting algorithms that outperform those provided by several widely used libraries, including the C++ STL. Whaley et al. (2001) describe a general approach for optimising the performance of algorithms termed *Automated Empirical Optimization of Software (AEOS)* and its application to computing-platform-specific performance optimisation of linear algebra routines. The meta-algorithmic search procedure used in this work is very specific to the code-level tuning task studied by the authors; their more general AEOS approach, while focussed on performance-optimisation of libraries of fundamental computing routines for given computing platforms on the code-generation level, is very similar to the computer-aided algorithm design approach advocated here in terms of its nature and motivation. Conceptually similar work can also be found in compiler optimisation (see, *e.g.*, Pan and Eigenmann, 2006; Cavazos et al., 2007) and database query optimisation (see, *e.g.*, Stillger and Spiliopoulou, 1996; Dong and Liang, 2007).

6 Conclusions

The computer-aided approach for the design of high-performance algorithms discussed in this report has many advantages compared to the traditional, manual design processes commonly used today. These advantages are especially pronounced in the context of designing heuristic algorithms for hard computational problems that increasingly arise in many application areas as a consequence of resource limitations and competition in large, tightly coupled markets. However, the computer-aided algorithm design approach is also applicable to computationally easier problems that need to be solved in real-time or near-real-time scenarios. In contexts where several types of problem instances with different characteristics need to be solved, computer-aided algorithm design is particularly attractive, since it can be used to automatically adapt algorithms that perform well on one instance distribution to other distributions with little or no human intervention, as long as a sufficiently rich design space and sufficiently powerful meta-algorithmic search procedures are available.

When moving from traditional algorithm design methods to computer-aided algorithm design, the role of human designers changes fundamentally: instead of sequentially testing a series of candidate designs, the emphasis is now on specifying design spaces that are then explored automatically. This approach is scalable and benefits from readily available computing power: Given a large design space, additional computational resources can be expected to help in finding better algorithms as a result of more thorough exploration within that space; alternatively, an increase in computational resources can prompt further extensions of the design space. We note that parallel computation can easily be harnessed in this context. In many cases, a given design space can be partitioned into disjoint subspaces in obvious ways, for example, by splitting on a number of binary or discrete design choices, or by partitioning the domains of continuous algorithm parameters. (This seems particularly useful in cases where some top-level design choices precede and determine later, lower-level choices, and where there

is no *a priori* knowledge regarding which top-level choice might be best.) Furthermore, stochastic local search procedures, which are among the most attractive and best suited methods for exploring complex combinatorial spaces, tend to be parallelisable easily and efficiently by means of performing multiple independent runs (see, *e.g.*, Hoos and Stützle, 2004; Hutter et al., 2007).

Overall, computer-aided algorithm design can lead to higher-performance and simpler algorithms. As a design approach, it is also more principled than the *ad-hoc* methods currently used, which makes it easier to disseminate and support, in the form of software systems for computer-aided algorithm design. Because of their more formalised nature, computer-aided algorithm design methods are also easier to evaluate and to improve. Their development and application therefore constitutes a key step in transforming the design of high-performance algorithm from a craft that is based primarily on experience and intuition to an engineering effort involving formalised procedures and best practices.

Existing work in various areas has followed similar goals and, in some cases, has led to methods that will doubtlessly be useful in the context of the approach advocated here. However, by framing the computer-aided design approach in a general way, by pointing out how it naturally encompasses and builds on existing concepts, such as parameter tuning, algorithm configuration, algorithm portfolios and algorithm selection, and by outlining how it can be supported by a general-purpose system that includes existing meta-algorithmic procedures, we believe that the vast potential this approach brings to the notoriously difficult task of designing high-performance algorithms can be realised more effectively and in a principled way. Our work aims to bring together related research efforts from a number of areas — including artificial intelligence, empirical algorithmics, algorithm engineering, operations research, numerical optimisation, machine learning, statistics, databases, parallel computing and software engineering — to leverage commonalities and exploit complementarities, to articulate and realise a vision that will change the way in which we design high-performance algorithms. As a result, human experts will be able to more easily design effective algorithms for solving computational problems encountered in application domains ranging from bioinformatics to industrial scheduling, from compiler optimisation to robotics, from databases to production planning.

Acknowledgement: Some of the ideas discussed in this document have their roots in joint work and discussions with Frank Hutter, Thomas Stützle, Kevin Leyton-Brown and Chris Fawcett. The author gratefully acknowlegdes helpful comments by Chris Fawcett, Frank Hutter, Morten Irgens, Alan Mackworth and Lin Xu on earlier versions of this document.

References

Adenso-Diaz, B. and Laguna, M. (2006). Fine-tuning of algorithms using fractional experimental design and local search. *Operations Research*, 54(1):99–114.

Balaprakash, P., Birattari, M., and Stützle, T. (2007). Improvement strategies for the F-race algorithm: Sampling design and iterative refinement. In Bartz-Beielstein, T., Aguilera, M. J. B., Blum, C., Naujoks, B., Roli, A., Rudolph, G., and Sampels, M., editors, *4th International Workshop on Hybrid Metaheuristics (MH'07)*, pages 108–122.

Bartz-Beielstein, T. (2006). *Experimental Research in Evolutionary Computation: The New Experimentalism*. Natural Computing Series. Springer Verlag, Berlin, Germany.

Battiti, R., Brunato, M., and Mascia, F. (2008). *Reactive Search and Intelligent Optimization*, volume 45 of *Operations Research/Computer Science Interfaces*. Springer Verlag.

Battiti, R. and Protasi, M. (1997). Reactive search, a history-based heuristic for MAX-SAT. *ACM Journal of Experimental Algorithmics*, 2.

Bentley, J. L. and McIlroy, M. D. (1993). Engineering a sort function. *Software–Practice and Experience*, 23:1249–1265.

Birattari, M., Stützle, T., Paquete, L., and Varrentrapp, K. (2002). A racing algorithm for configuring metaheuristics. In *GECCO '02: Proc. of the Genetic and Evolutionary Computation Conference*, pages 11–18.

Bölte, A. and Thonemann, U. W. (1996). Optimizing simulated annealing schedules with genetic programming. *European Journal of Operational Research*, 92(2):402–416.

Brodal, G. S., Fagerberg, R., and Vinther, K. (2008). Engineering a cache-oblivious sorting algorithm. *J. Exp. Algorithmics*, 12:1–23.

Burke, E., Kendall, G., Newall, J., Hart, E., Ross, P., and Schulenburg, S. (2003). Hyper-heuristics: an emerging direction in modern search technology.

Bűrmen, Á., Puhan, J., and Tuma, T. (2006). Grid restrained nelder-mead algorithm. *Computational Optimization and Applications*, 34(3):359–375.

Carchrae, T. and Beck, J. (2005). Applying machine learning to low knowledge control of optimization algorithms. *Computational Intelligence*, 21(4):373–387.

Cavazos, J., Fursin, G., Agakov, F., Bonilla, E., O'Boyle, M. F. P., and Temam, O. (2007). Rapidly selecting good compiler optimizations using performance counters. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 185–197, Washington, DC, USA. IEEE Computer Society.

Cowling, P., Kendall, G., and Soubeiga, E. (2002). Hyperheuristics: A tool for rapid prototyping in scheduling and optimisation. In *Applications of Evolutionary Computing*, volume 2279 of *Lecture Notes in Computer Science*, pages 1–10. Springer.

Di Gaspero, L. and Schaerf, A. (2007). Easysyn++: A tool for automatic synthesis of stochastic local search algorithms. In Stützle, T., Birattari, M., and Hoos, H. H., editors, *Proc. International Workshop on Engineering Stochastic Local Search Algorithms (SLS 2007)*, volume 4638 of *Lecture Notes in Computer Science*, pages 177–181. Springer.

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271.

Dong, H. and Liang, Y. (2007). Genetic algorithms for large join query optimization. In *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1211–1218, New York, NY, USA. ACM.

Fukunaga, A. S. (2000). Genetic algorithm portfolios. In *Proc. of the 2000 Congress on Evolutionary Computation*, pages 1304–1311, Piscataway, NJ. IEEE Service Center.

Fukunaga, A. S. (2002). Automated discovery of composite sat variable-selection heuristics. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence*, pages 641–648.

Fukunaga, A. S. (2004). Evolving local search heuristics for SAT using genetic programming. In Deb, K., Poli, R., Banzhaf, W., Beyer, H.-G., Burke, E., Darwen, P., Dasgupta, D., Floreano, D., Foster, J., Harman, M., Holland, O., Lanzi, P. L., Spector, L., Tettamanzi, A., Thierens, D., and Tyrrell, A., editors, *Genetic and Evolutionary Computation – GECCO-2004, Part II*, volume 3103 of *Lecture Notes in Computer Science*, pages 483–494, Seattle, WA, USA. Springer-Verlag.

Gagliolo, M. and Schmidhuber, J. (2006). Dynamic algorithm portfolios. In Amato, C., Bernstein, D., and Zilberstein, S., editors, *Ninth International Symposium on Artificial Intelligence and Mathematics (AI-MATH-06)*.

Gomes, C. P. and Selman, B. (2001). Algorithm portfolios. *Artificial Intelligence*, 126(1-2):43–62.

Guerri, A. and Milano, M. (2004). Learning techniques for automatic algorithm portfolio selection. In *Proceedings of the 16th Eureopean Conference on Artificial Intelligence (ECAI 2004)*, pages 475–479. IOS Press.

Hansen, N. and Kern, S. (2004). Evaluating the CMA evolution strategy on multimodal test functions. In Yao, X. et al., editors, *Parallel Problem Solving from Nature PPSN VIII*, volume 3242 of *LNCS*, pages 282–291. Springer.

Hansen, N. and Ostermeier, A. (2001). Completely derandomized self-adaptation in evolution strategies. *Evolutionary Computation*, 9(2):159–195.

Hoos, H. (1999). *Stochastic Local Search – Methods, Models, Applications*. infix-Verlag, Sankt Augustin, Germany.

Hoos, H. H. and Stützle, T. (2004). *Stochastic Local Search—Foundations and Applications*. Morgan Kaufmann Publishers, USA.

Huberman, B., Lukose, R., and Hogg, T. (1997). An economics approach to hard computational problems. *Science*, 265:51–54.

Hutter, F., Hoos, H. H., Leyton-Brown, K., and Stützle, T. (2008). ParamILS: An automatic algorithm configuration framework. (Submitted).

Hutter, F., Hoos, H. H., and Stützle, T. (2007). Automatic algorithm configuration based on local search. In *Proceedings of the Twenty-Second National Conference on Artificial Intelligence*, pages 1152–1157.

Jones, D. R., Schonlau, M., and Welch, W. J. (1998). Efficient global optimization of expensive black box functions. *Journal of Global Optimization*, 13:455–492.

Leyton-Brown, K., Nudelman, E., Andrew, G., McFadden, J., and Shoham, Y. (2003). A portfolio approach to algorithm selection. In Rossi, F., editor, *Principles and Practice of Constraint Programming – CP 2003*, volume 2833 of *Lecture Notes in Computer Science*, pages 899–903. Springer Verlag, Berlin, Germany.

Li, X., Garzarán, M. J., and Padua, D. (2005). Optimizing sorting with genetic algorithms. In *Proc. International Symposium on Code Generation and Optimization*, pages 99–110. IEEE Computer Society.

Maue, J. and Sanders, P. (2007). Engineering algorithms for approximate weighted matching. In *Experimental Algorithms, 6th International Workshop (WEA 2007)*, volume 4525 of *Lecture Notes in Computer Science*, pages 242–255. Springer.

Monette, J.-N., Deville, Y., and Hentenryck, P. V. (2009). Aeon: Synthesizing scheduling algorithms from high-level models. In *Proc. 11th INFORMS Computing Society Conference (to appear)*.

Nelder, J. A. and Mead, R. (1965). A simplex method for function minimization. *The Computer Journal*, 7(4):308–313.

Pan, Z. and Eigenmann, R. (2006). Fast and effective orchestration of compiler optimizations for automatic performance tuning. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 319–332, Washington, DC, USA. IEEE Computer Society.

Poli, R., Langdon, W. B., and McPhee, N. F. (2008). *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`. (With contributions by J. R. Koza).

Powell, M. (1998). Direct search algorithms for optimization calculations. *ACTA NUMERICA*, pages 287–336.

Rice, J. R. (1976). The algorithm selection problem. *Advances in Computers*, 15:65–118.

Sanders, P. and Schultes, D. (2007). Engineering fast route planning algorithms. In *Proc. 6th International Workshop on Experimental Algorithms (WEA-2007)*, volume 4525 of *Lecture Notes in Computer Science*, pages 23–36. Springer.

Stillger, M. and Spiliopoulou, M. (1996). Genetic programming in database query optimization. In *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 388–393. MIT Press.

Thrun, S. (1996). *Explanation-Based Neural Network Learning: A Lifelong Learning Approach*. Kluwer Academic Publishers, Boston, MA.

Van Hentenryck, P. and Michel, L. D. (2007). Synthesis of constraint-based local search algorithms from high-level models. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence (AAAI-07)*, pages 273–278.

Vilalta, R. and Drissi, Y. (2002). A perspective view and survey of meta-learning. *Artificial Intelligence Review*, 18:77–95.

Westfold, S. J. and Smith, D. R. (2001). Synthesis of efficient constraint-satisfaction programs. *Knowl. Eng. Rev.*, 16(1):69–84.

Whaley, R. C., Petitet, A., and Dongarra, J. J. (2001). Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35.

Xu, L., Hutter, F., Hoos, H., and Leyton-Brown, K. (2007). Satzilla-07: The design and analysis of an algorithm portfolio for SAT. In *Principles and Practice of Constraint Programming – CP 2007*, pages 712–727.

Xu, L., Hutter, F., Hoos, H. H., and Leyton-Brown, K. (2008). SATzilla: portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research*, 32:565–606.