Automated Design of Algorithms and Genetic Improvement: Contrast and Commonalities

Saemundur O. Haraldsson
University of Stirling
Stirling, Scotland, UK
soh@cs.stir.ac.uk

John R. Woodward
University of Stirling
Stirling, Scotland, UK
jrw@cs.stir.ac.uk

Categories and Subject Descriptors

I.2.2 [**Automatic Programming**]: Program modification

ABSTRACT

Automated Design of Algorithms (ADA) and Genetic Improvement (GI) are two relatively young fields of research that have been receiving more attention in recent years. Both methodologies can improve programs using evolutionary search methods and successfully produce human competitive programs. ADA and GI are used for improving functional properties such as quality of solution and non-functional properties, e.g. speed, memory and, energy consumption. Only GI of the two has been used to fix bugs, probably because it is applied globally on the whole source code while ADA typically replaces a function or a method locally. While GI is applied directly to the source code ADA works ex-situ, i.e. as a separate process from the program it is improving.

Although the methodologies overlap in many ways they differ on some fundamentals and for further progress to be made researchers from both disciplines should be aware of each other's work.

Keywords

Automated Design of Algorithms (ADA), Genetic Improvement (GI), Genetic Programming (GP), Abstract Syntax Tree (AST), Genetic Algorithm (GA), Search Based Software Engineering (SBSE), Genetic Algorithm (GA)

1. INTRODUCTION

Recent decades has seen surge in the development of Metaheuristics [10] and Hyper-heuristics [7] where the latter could be defined as a special case of Automated Design of Algorithms (ADA). ADA is a general methodology employing search methods to improve or discover algorithms for computational problems. It generates algorithms by sampling the large, discontinuous and complex space of programs.

GECCO'14, July 12–16, 2014, Vancouver, BC, Canada.

ACM 978-1-4503-2881-4/14/07 ...$15.00.
http://dx.doi.org/10.1145/2598394.2609874.

GI has seen some progress due to the well-established Search Based Software Engineering (SBSE) [27, 26] with initiatives in automating many key aspects of software engineering [22]. SBSE applies computational search to software engineering problems and GI is the special case of automatic software adjustment methods working directly on the source code. The idea is that the source code acts as the genetic material and can be rearranged to improve the program for a given objective. The need to decrease the cost of maintenance in the development of software is very important since it is usually the largest part of the software life cycle. Automating the maintenance process would more than likely reduce the cost considerably and give developers more time and money to focus on more qualitative features of programs.

Recent years have seen research that can be defined as ADA and GI gaining momentum and an increase in number of publications. The two fields have much in common but they can be divided by their application and definition. They apply same and similar methods, typically Genetic Programming (GP) or other metaheuristics. They are often used for identical objectives, e.g. speed or accuracy, albeit those objectives sometimes have different evaluation criteria. For example to measure speed both CPU time and number of iterations until a sufficient solution has been reached are used. The time is ripe to synchronize their terminologies and thus recognize the opportunities that lie in the inevitable crossing of the two methodologies.

The reminder of this paper will firstly give an overview of ADA and GI in sections 2 and 3 respectively. In section 4 the two methodologies will be compared by highlighting their similarities and differences. In section 5 the paper will be summarized and it will conclude in section 6.

2. AUTOMATED DESIGN OF ALGORITHMS

In this section we will define ADA and give examples of its applications and opportunities. Designing algorithms is a complex and sometimes unintuitive search and the automation of that process is therefore necessary and often inevitable. Algorithms are defined as a set of stepwise instructions for general problem solving and each algorithm can be viewed as a recipe for tackling a specific type of problem. When designing an algorithm one can either do it bottom-up by defining a general outline and filling in with appropriate components or start with an already functioning algorithm and adjust it for ones needs. Given a certain abstraction each algorithm can be altered and adjusted in three ways.

- By *replacing* components with an alternative [55]. In its simplest form this is done by selecting from a set of suitable replacement components which confines the search for a better algorithm to a discrete and finite space. An example of this might be to have a evolutionary algorithm and a set of predefined selection methods to choose from, such as elitist selection and tournament selection. Another approach is to evolve or generate replacement components and depending on multiple aspects such as, component representation, generation method and the original algorithm this can easily become very complex and impossible for exhaustive search.

- By *reordering* the application of the components [12]. Given certain restrictions, where the algorithm needs to be initiated and what it returns must be of specific type, this way is a special case of a permutation problem.

- *Parameter tuning*, i.e. changing the amount of any or each component [49]. E.g. altering the likelihood of mutation in a Genetic Algorithm (GA). Even if this is not in the strictest sense a design problem, it should be combined with the other two ways whenever possible. When comparing the newly designed algorithm with its predecessor, both subjects should perform at optimal capacity given the evaluation criteria.

The combined search space for those three ways can be vast, discontinuous, non-linear and/or multidimensional. Not to mention the difficulties with selecting and presenting the fitness which guides the search. In machine learning literature an objective of finding or making new algorithms has typically been to generalize them as much as possible, given a certain domain [41, 48]. Recent research and in particular ADA research has however suggested that automating the process reveals opportunities to specialise the algorithms to a certain problem class [8, 55, 56]. A problem class is a probability distribution over all instances of a given domain, e.g. Travelling Salesman Problem (TSP), timetabling and bin packing. The current access to computational resources can give us millions of variations of each algorithm in seconds and thus we should not be limiting the search for the one algorithm that solves all or a "Jack-of-all-Trades" [8]. Then we can create general frameworks that can be adjusted to produce algorithms for a wide variety of computational problems. There could be a general framework that generates algorithms to solve problems where the solution representation and another for problems where the solution representation is bit-strings. When the algorithm being designed is a search or an optimization algorithm the main objective of the fitness function is typically to improve a functional property that is evaluated as the fitness of the best solution found by the evolved algorithm [4, 13, 18, 19, 29]. Speed has also been of interest for others [21] as well as combinations of both where GP was used to evolve search heuristics [21], and novel matrix multiplication methods where discovered with GA [31].

2.1 Definition of Automated Design of Algorithms

We define ADA as a general methodology for using search methods to improve or discover algorithms for any given computational problem. When generating complete algorithms the amount of information provided can be as little as only a set of problems, their required output data type and a selection of primitive instructions or functions to assemble the algorithm from. Generating algorithms can also be done by using an existing algorithm and adjust it to serve either a different purpose or just to improve it for its current purpose. The algorithm in question must be suitable for a partition into blocks or components that can be replaced by a function with the same type signature, such that the original algorithm can serve as a template or scaffold.

Given this definition, research from as early as the 1990's can be categorized as ADA e.g. where [4] use GP to evolve annealing schedule for a simulated annealing algorithm. Hyper-heuristics to generate heuristics is a sub-field of ADA, as they are defined to be a heuristic search in the space of heuristics [11]. The definition of ADA is broad and its application is included in many well established fields of research, such as machine learning [30], data mining [45] and operation research [6]. Many of those fields are overlapping and sometimes the line between them is not distinct. Therefore it is necessary to give a more general platform for discussion and debate so that progress can be made.

The last decade has seen rapid increase in the frequency of research that falls under the definition of ADA and because of that, effort has been made to gather relevant publications in this paper. There is shortage in the literature of solid categorization of research that is ADA which makes it difficult to guarantee that a published algorithm or methodology is indeed novel. Completely different names and terminology for similar things are somewhat scattered, so extensive comparison is difficult. The Workshop on the Automated Design of Algorithms in the annual Genetic and Evolutionary Computation Conference (GECCO) has been held since 2011 and serves as an international platform for exchanging ideas[1].

2.2 Applications of Automated Design of Algorithms

ADA has been used in many different and creative ways. We now examine component replacement and reordering. We do not however examine automatic parameter tuning as there is already a vast literature on that [3, 15].

2.2.1 Component replacement

The most common example of ADA is where an algorithm is improved by selecting one of its components and replacing it with an automatically designed alternative. The component is usually something that can be easily replaced by a function due to its place in the program where it has known input and output data type. Some examples of such components are:

- Annealing schedule for the simulated annealing algorithm [4].

- Selection heuristic for bin packing problems [9] and vehicle routing algorithm [12].

- Scheduling rules for job shop scheduling [43].

- Flowtime estimators for job shop scheduling [42].

[1] `http://www.sigevo.org/gecco-2011/workshops.html#ecdga`

- Variable-selection heuristic for Boolean Satisfiability (SAT) solvers [19, 20].

For each of these applications note that the components being designated for replacements can be easily isolated in the algorithm structure. Then they can be replaced by any arbitrary function so long as that function has the same type signature as defined by the original component. Note also that each of these examples are relatively small algorithms and components that can be expressed in a few lines of code but still the search space for ADA is huge and complex. The annealing schedule can for example be replaced with any function that returns a double precision variable on the closed interval between zero and one [4]. GP has proved to be very effective in evolving the replacement [55] and its ability to generate small but complex programs makes it the primary method for these types of ADA. Also, the abstraction and a careful selection of the GP function set allows an easy way of guaranteeing error free compilation and termination, e.g. if the function set only contains Turing complete operators [44]. The evolution of the replacement always takes place ex situ i.e. it is a search process that runs separate from the original algorithm. It typically takes place in parallel and with a representation that does not need to be identical to the original algorithm's representation. The outcome of that process is evaluate by either converting the component into the original source language before substituting the evolved code into the original program or building a wrapper function to interpret.

2.2.2 *Component reordering*

ADA has not to the authors' best knowledge been applied as a permutation problem but only in combination with the replacement idea and with a given framework for scaffolding [14]. The scaffolding method is another ADA application that uses an existing algorithm framework as a template and finds the best combination of a finite number of predefined alternatives for each place in the scaffold. This method was used to evolve multiple meta-heuristics for a vehicle routing problem. A general outline of the simplex variant called a push/pull algorithm was used as a template and the components were extracted from three established and widely used algorithms to serve as the blocks that could be put into the scaffold [12].

The search space of ADA has also been widened by extracting components from many different algorithms to use them as building blocks [39, 45]. Allowing the structure to evolve with minimal restrictions as well as the what the structure contains, i.e. the blocks can be stacked in any order with the only limitation being that the output is meaningful. While one describes a general automation process for various data mining algorithm [45], the other evolves a structure of a black-box algorithm that tackles the Deceptive Trap problem [39]. Both papers rely on grammatical evolution which describes the restrictive condition each block has, such as what blocks can precede and follow other blocks. These conditions are primarily based on the blocks data signatures.

3. GENETIC IMPROVEMENT

Human programmers working in industry cannot be expected to optimize their software with respect to non-functional properties within a given budget or available project time frame [5]. The software development life cycle is a iterative process where the program is constantly undergoing improvements. Whether it is bug fixing, speed up, memory optimization, platform specific adaptation or many more examples. This process is carried out by human programmers that are usually only experts on a few platforms, languages or environments. End users are vaguely aware of this process when their device prompts them to update but for developers this is a cycle of often tiny tweaks and minor improvements. Moreover because programmers, without automated tools, are limited by trial and error methods and they can only view a small portion of the space of all available adjustments those updates could have to be reverted in later cycles.

Many methods have been proposed to counteract software's inevitable high maintenance cost, substandard releases with annoyingly frequent updates and other maintenance problems. One promising approach developed in the recent decade suggests automation in one or all aspects, employing SBSE [28] methods. More specifically evolutionary search methods such as GP have been used to explore program variants. GI is one of these SBSE methods that has in recent years been surfacing as the conquering automatic improvement candidate in the software engineering literature.

Since GI involves adjusting an already functioning software it is often highly susceptible to producing compiler errors non-terminating programs. That is anticipated and methods to handle it usually involve some kind of penalty such that it has little to no possibility to survive to the next iteration [54]. Other methods employ direct elimination, by discarding them entirely from the population of programs being evaluated [36, 50]. Although solutions have been suggested to this using reflective properties of certain programming languages and an *actor model* [50].

In this paper variants of GI will be divided into 4 categories based on their application domain.

- Bug fixing, probably its most common application.
- Increasing speed, which has always been of much interest maybe because of easy evaluation.
- Migration and transplantation. Migrating software from one platform to another, transplanting a code segment or a functionality from on program to another.
- Dynamic adaptive software improvement where the program is improved while it is being used.

3.1 Definition of Genetic Improvement

GI is the name given to automatic software adjustment methods that operate directly on the source code, treating it as the genetic material. These operations may for example consist of copying, deleting and swapping lines of code. An existing software's source code is used as the building material for the improvements, meaning that it is assumed the software already possesses sufficient expressions [24, 33]. The search is therefore limited to all variants of said software code, albeit those variants can be of variable length because expressions can be duplicated or deleted. The pool of genetic material can however be expanded with transplantation from other programs, either multiple variants of the same program or other programs that have desirable features. Code transplantation has been used to expand the pool of genetic material with already existing variants

of the software that was being improved. Two award winning and manually improved variants of the MiniSAT solver introduced enough new genetic material to improve the performance of the original beyond the donor programs [46].

The automation involves applying some search method in the program space driven by the objectives set by the developer. While these objectives are relatively easy to define, finding an effective fitness function can pose a real problem [16] because the software must maintain its correct functionality. The correctness of an improvement is found either by comparing the output with the output of the original software as an oracle [34, 2] or by evaluating its performance on a suite of test cases that has already been run on the original code. The fitness is then monitored during compilation and runtime by instrumentation, simple timing mechanisms or whatever the objective defines. Given that the search space is highly noisy [32] evolutionary methods have been preferable over others. Particularly GP with grammatical representation, again for its ability to represent and evolve programs.

As with ADA, some focus of GI has recently been turned towards specialisation of software rather than generalization [23, 24, 46]. With increasing computational resources and constantly better methods for improvement, making multiple variants of a software and maintaining them is more viable than before. A program that is fast but energy inefficient might be acceptable for a desktop computer but the user might be willing to consider slower but energy saving variant for his laptop in order to extend the usage of the battery before needing to be recharged.

Earlier in GI the typical approach to evolve improvements was to maintain multiple copies of the program in a population but that was shortly discarded in favour of maintaining a population of edits. Each edit describes changes that should be made to the original program before evaluation. This approach is much more memory efficient and makes explaining the changes to the software developer easier, although the extra effort of applying the changes must be made before each evaluation.

3.2 Applications of Genetic Improvement

The appeal of GI is that the process of optimization does not need a model because the software is its own substrate. Therefore the evaluation process does not give an approximation of how the software will perform but potentially an accurate vision. Unlike other engineering practices where e.g. structural loads are approximated in civil engineering and productivity is estimated in industrial engineering with models. GI in software engineering makes it possible to monitor the effects of every small change directly which makes it well suited for fixing faulty programs.

3.2.1 Bug Fixing

GI has been extensively applied to automatic bug fixing where test cases were used to identify nodes in the AST that might be faulty and successfully focus an evolution of patches [17, 52]. They were able to repair multiple different bugs in 20 legacy C programs. Further progress was then made where test suite optimization was used to improve the fitness evaluation of the fixes [16, 51]. Their success has spurred further research into other domains and programming languages such as Python [1] where a novel patch representation was introduced. The work of Weimer et al. has since then culminated in a framework called GenProg [38]. It also built on the Python novel patch representation [1] by changing the representation to edits from ASTs to be able to scale properly to larger programs. GenProg was reported to fix 55 of 105 known bugs for about US $8 each in 8 open-source programs [37]. Therefore giving plenty of evidence for the merit of GI as an automatic software repair method.

3.2.2 Increasing Speed

GI is appropriate for improving non-functional properties of an existing software when there is sufficiently good test suit available. The original version of the software can act as an oracle to maintain proper functionality [34, 2] alongside evaluation. The non-functional property that is probably most popular to improve has to be speed since its evaluation is quite uncomplicated and comparison is easy.

Strongly typed GP was successfully applied on small programs with seeding [2, 54]. That work also introduced multi-objective approach to the improvement process by allowing some error while searching for faster alternatives to the original seed. Later the speed improvements were generalised to other non-functional properties specifically for low-resource systems [53].

Larger systems have been optimized for speed in [36] where Bowtie2, a DNA sequencing system that is made up of approximately 50k lines of C++ code. A 70 fold speed up on specialized DNA sequences was reported without decreasing functionality by evolving edits with grammatical evolution. The same framework and methodology was used to improve the computational time of the boolean satisfiability (SAT) solver, MiniSAT, with moderate success [47]. Later a significant speed gain was observed when the genetic pool was widened with multiple versions of the solver software [46].

3.2.3 Migration and transplantation

Code transplantation has been suggested in [25] for reverse engineering purposes such as inserting a feature from one system into another. It seems that GI has the tools and right approach to accomplish this task automatically by using GP and appropriate representation. Transplantation has been reported with success, although it was not between systems or software but between different versions of the same software [46]. Proper automatic code transplantation between systems written in the same language or even systems written in different languages has yet to be made but migrating software from one environment to another has seen success.

The migration and complete a repurposing was accomplished in [34] where CUDA C++ code was adapted for the gzip functionality to run on parallel processes, accelerating the function considerably in the process. The CUDA C++ code, StereoCamera system, was also migrated to parallel graphics hardware and gained from 5% to a factor of 6.8 speed up [35].

3.2.4 Dynamic adaptive approach

Attaining dynamic adaptivity means to be able to improve the software while it is in use and therefore potentially being able to reduce maintenance costs to absolute minimum. If the software can improve itself that also means that downloading updates could be eliminated thus using bandwidth and data charges for other more productive or entertaining means. Some methods have been suggested such as the semi

online approach in [23] where it is proposed that optimization of non-functional properties could happen on mobile devices while charging. A more dynamically online stance is suggested and implemented in [50] with a web service running on the Java Virtual Machine (JVM) and utilizing the reflection abilities of the Scala programming language.

4. DISCUSSION AND CONTRAST

Although the broadest definition of an algorithm includes the simplest instructions that can be executed with a pen and paper, the automation is only possible for when those instructions have been implemented into a software. As such we can view ADA as a specific software improvement methodology and compare it with the field of GI. This section will first compare similarities and then compare differences. So that one might be better equipped to make the decision which one to use for a given problem. This is not a trivial decision.

4.1 Similarities

The apparent similarities are firstly that both methodologies are driven by the need to automatically improve or adapt software and therefore they partly share a domain. ADA and GI both work on either existing software or a software framework where the functionality is known and at least the minimal number of components that are needed for that functionality are known and available. ADA has mainly been used to evolve small functions from scratch as a replacement component for a bigger algorithm and although GI sometimes operates on large scale systems, its output is usually small set of edits to that system. So typically the output from both methodologies are relatively small such as a few edits or a function that can be expressed in a few lines of code.

GI has been successfully applied to accelerate the MiniSAT solver and so has ADA, although the ADA only required access to the heuristic selection operator.

Last but not least the part of the shared domain of computer programs is highly complex and large and the task of searching it is consequently complex. GI and ADA both employ evolution for that search which make them comparatively resource consuming both in terms of CPU time and memory.

4.2 Difference

The most notable difference is that GI is applied in-situ or directly to the source code while ADA works ex-situ, i.e. evolves a function that is injected into the original code. GI makes small changes, sometimes many small changes that do not have to be constrained to a small region of the source code. ADA's improvement of an existing program on the other hand is a replacement of a certain call or statement in the source and is thus limited to the places where that call is made.

As observed GI and ADA evolve small edits and functions respectively but the main difference is that the GI's edits are human readable while ADA's functions are often quite unintuitive and cannot be explained simply. This can possibly be the cause for their difference in sub-domain application, where ADA is commonly used to improve heuristics [4, 9, 12, 19, 20, 43], data mining, and machine learning algorithms [30, 45]. GI is applied to software in general, although it improves MiniSAT [46] and a classification algorithm [2] it is not limited to algorithms and has fixed bugs in legacy programs [17, 52] and 20 other open-source programs [37].

ADA has typically been used to evolve small functions but those can be relatively large compared to the whole software when the program is a heuristic. GI's successes are typically reported on a larger systems and therefore suggesting that it fairs better when it has access to a larger gene pool of potential changes. The difference here lies in how the methodologies use the ASTs, GI alters the whole AST while ADA takes one node or a cluster of nodes and replaces them. For that reason GI has been used more often for automatically repairing software since it can make small changes in the software that are separated by many lines or even files in the code. Changes that potentially fix a fault that a human could not find because the bug's location might not be co-located with the fault it causes or the fault is caused by a series of bugs that are dispersed in different sections of the code. ADA would probably not be able to fix the latter kind of fault with its limited scope into the original software. Similarly for non-functional properties such as speed or memory usage, GI might find solutions that are not confined to a single node or a cluster of nodes in the AST. Sometimes those improvements are just to turn off non-vital functionalities of the original software [36]. Also, although the two methodologies have the objective to speed up a software in common, they usually approach the task from different angles. While GI tries to speed up programs by finding variants of the original software, ADA's approach is often to make the algorithm find a better solution sooner.

A side effect of the GI process is that it produces lots of faulty individuals out of scope, semantically incorrect or non terminating, while ADA can more easily be applied so that all individuals compile and terminate. For some, a few faulty program variants while iterating over many is considered to be an annoyance but nothing to be concerned about and it has been argued that it does not affect the evolution to much [36]. Others suggest using better approaches with reflection and agent based coding to prevent these side effects [50].

5. SUMMARY

This paper has given a brief overview of the fields of ADA and GI. Where some examples of usage and application were talked about as well as their merits and shortcomings, limitations and some opportunities. At last their similarities and differences were highlighted while trying to give some idea of when one should be chosen over the other.

The following bullet points highlight the main findings of this paper and overall what is observed is that the two fields are very similar and share many characteristics but they still differ in a fundamental way.

- ADA and GI both produce human compatible outputs, i.e. they improve upon existing software and even successfully improve a program [2] that has been manually improved multiple times.
- ADA and GI both typically apply GP or other evolutionary methods to generate changes. Although ADA has been using it in the more traditional sense [32] while GI converted to evolving edit lists as they are less memory consuming and more easily explainable to humans.

[2]For example MiniSAT [46, 19, 20]

- Current application of GI can produce faulty and non-terminating programs in the process of finding improved variants while ADA generates syntactically correct terminating code.

- ADA is used to improve functional and non-functional properties while GI's focus has typically been non-functional properties although with different approach than ADA. ADA's approach to speed up has been to improve the convergence of an algorithm to a good solution and GI has been used to optimize the code structure, i.e. in what sequence the statements are executed. Also, only GI has been used to fix bugs.

- ADA and GI share a general domain where they are both used to improve software but ADA has only been used on heuristics and machine learning and data mining algorithms.

6. CONCLUSIONS

ADA and GI are two related fields which are similar and are both at an early stage of their development but an increase in use by researchers and industry alike is to be expected. It would probably be beneficial for further progress for researchers in either discipline to be aware of each other's work. The former is mainly manned by computer scientists while the latter consists of mainly software engineers. They share a domain but the approach and applicability are quite different.

The search method of choice for both fields is GP and recently it has been accused of tackling toy problems [40] so ADA and GI offer a rich domain of real world problems which will necessitate improvements in GP, the search technology. The usage of GP is also different between the fields where representation of improvements and objectives are often not the same.

When fixing bugs, GI is preferable for its ability to make small but significant changes to a software and ADA when the task at hand is to find novel components for a computational method. For accelerating software the choice of method depends on the software, if the software is an easily compartmentalized algorithm it seems that ADA could be better but for a general software GI might be more suitable. Because GI can work on a more fine grained level of the software and does not have to be focused on one particular AST node or a cluster of nodes.

7. REFERENCES

[1] T. Ackling, B. Alexander, and I. Grunert. Evolving Patches for Software Repair. In *GECCO'11, 13th annual conference on Genetic and evolutionary computation*, pages 1427–1434, Dublin, Ireland, July 2011. ACM.

[2] A. Arcuri, D. R. White, J. Clark, and X. Yao. Multi-objective improvement of software using co-evolution and smart seeding. In X. Li, M. Kirley, M. Zhang, D. Green, V. Ciesielski, H. Abbass, Z. Michalewicz, T. Hendtlass, K. Deb, K. Tan, J. Branke, and Y. Shi, editors, *7th International Conference, SEAL 2008*, Lecture Notes in Computer Science, pages 61–70, Melbourne, Australia, Dec. 2008. Springer Berlin Heidelberg.

[3] M. Birattari, Z. Yuan, P. Balaprakash, and T. Stützle. F-Race and iterated F-Race: An overview. In T. Bartz-Beielstein, M. Chiarandini, L. Paquete, and M. Preuss, editors, *Experimental Methods for the Analysis of Optimization Algorithms*, chapter 13, pages 311–336. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

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

[5] F. P. Brooks. *The Mythical Man-Month, Essays on Software Engineering, Anniversary Edition*. Addison Wesley, 1995.

[6] E. Burke, M. Hyde, G. Kendall, and J. Woodward. Genetic Programming Hyper-Heuristic Approach for Evolving Two Dimensional Strip Packing Heuristics. *IEEE Transactions on Evolutionary Computation*, 14(6):942–958, 2010.

[7] E. K. Burke, M. Hyde, G. Kendall, G. Ochoa, E. Özcan, and J. R. Woodward. A classification of hyper-heuristic approaches. In M. Gendreau and J.-Y. Potvin, editors, *Handbook of Metaheuristics*, volume 146 of *International Series in Operations Research & Management Science*, pages 449–468. Springer US, 2010.

[8] E. K. Burke, M. Hyde, G. Kendall, and J. Woodward. Automatic Heuristic Generation with Genetic Programming : Evolving a Jack-of-all-Trades or a Master of One. In *GECCO'07, 9th annual conference on Genetic and evolutionary computation*, pages 1559–1565, New York, New York, USA, 2007. ACM.

[9] E. K. Burke, M. R. Hyde, and G. Kendall. Grammatical Evolution of Local Search Heuristics. *IEEE Transactions on Evolutionary Computation*, 16(3):406–417, June 2012.

[10] E. K. Burke and G. Kendall. *Search Methodologies*. Springer US, Boston, MA, 2005.

[11] E. K. Burke, G. Kendall, J. Newall, E. Hart, P. Ross, and S. Schulenburg. Hyper-heuristics: An emerging direction in modern search technology. In *Handbook of Metaheuristics*, International Series in Operations Research & Management Science, pages 457–474. Springer US, 2003.

[12] Y. Caseau, F. Laburthe, and G. Silverstein. A meta-heuristic factory for vehicle routing problems. In *5th International Conference, Principles and Practice of Constraint Programming, CP'99*, Lecture Notes in Computer Science, pages 144–159, Alexandria, VA, USA, 1999. Springer Berlin Heidelberg.

[13] L. Diosan, M. Oltean, and L. Dio. Evolving crossover operators for function optimization. In *9th European Conference on Genetic Programming, EuroGP 2006*, Lecture Notes in Computer Science, pages 97–108, Budapest, Hungary, 2006. Springer Berlin Heidelberg.

[14] J. Drake, N. Kililis, and E. Özcan. Generation of VNS components with grammatical evolution for vehicle routing. In K. Krawiec, A. Moraglio, T. Hu, A. ÃĔ. Etaner-Uyar, and B. Hu, editors, *16th European Conference, EuroGP 2013, April 3-5*, Lecture Notes in Computer Science, pages 25–36, Vienna, Austria, 2013. Springer Berlin Heidelberg.

[15] A. E. Eiben and S. K. Smit. Evolutionary Algorithm Parameters and Methods to Tune Them. In Y. Hamadi, E. Monfroy, and F. Saubion, editors, *Autonomous Search*, chapter 2, pages 15–36. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[16] E. Fast, C. L. Goues, S. Forrest, and W. Weimer. Designing better fitness functions for automated program repair. In *GECCO '10 Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pages 965–972, Portland, Oregon, USA, 2010. ACM.

[17] S. Forrest, T. Nguyen, W. Weimer, and C. Le Goues. A genetic programming approach to automated software repair. In *11th Annual conference on Genetic and evolutionary computation, GECCO'09*, pages 947–954, New York, New York, USA, 2009. ACM.

[18] W. Fu, M. Johnston, and M. Zhang. Automatic construction of invariant features using genetic programming for edge detection. In *25th Australasian Joint Conference, AI 2012: Advances in Artificial Intelligence*, number 2 in Lecture Notes in Computer Science, pages 144–155, Sydney, Australia, 2012. Springer Berlin Heidelberg.

[19] A. Fukunaga. Automated Discovery of Composite SAT Variable-Selection Heuristics. In *The 18th National Conference on Artificial Intelligence*, pages 641–648, Edmonton, Canada, 2002. The AAAI Press.

[20] A. S. Fukunaga. Evolving local search heuristics for SAT using genetic programming. In K. Deb, editor, *Genetic and Evolutionary Computation - GECCO 2004*, volume 3103 of *Lecture Notes in Computer Science*, pages 483–494, Seattle, WA, USA, June 2004. Springer Berlin Heidelberg.

[21] A. S. Fukunaga. Automated discovery of local search heuristics for satisfiability testing. *Evolutionary Computation*, 16(1):31–61, Jan. 2008.

[22] M. Harman, E. Burke, J. Clark, and X. Yao. Dynamic adaptive search based software engineering. In *International Symposium on Empirical Software Engineering and Measurement*, 2012.

[23] M. Harman, Y. Jia, W. B. Langdon, J. Petke, I. H. Moghadam, S. Yoo, and F. Wu. Genetic Improvement for Adaptive Software Engineering (keynote paper). In G. Engels, editor, *SEAMS '14*, Hyderabad, India, 2014. ACM.

[24] M. Harman, W. B. Langdon, Y. Jia, D. R. White, A. Arcuri, and J. A. Clark. The GISMOE challenge: Constructing the pareto program surface using genetic programming to find better programs (keynote paper). In *27th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–14, Essen, Germany, Sept. 2012. ACM.

[25] M. Harman, W. B. Langdon, and W. Weimer. Genetic programming for Reverse Engineering. In *WCRE'13 20th Working Conference on Reverse Engineering*, pages 1–10, Koblenz, Germany, Oct. 2013. IEEE.

[26] M. Harman, S. Mansouri, and Y. Zhang. Search based software engineering: A comprehensive analysis and review of trends techniques and applications. *University College London, Tech. Rep. TR-09-03*, pages 1–78, 2009.

[27] M. Harman, S. A. Mansouri, and Y. Zhang. Search-based Software Engineering: Trends, Techniques and Applications. *ACM Computing Surveys*, 45(1):1–61, Nov. 2012.

[28] M. Harman, P. McMinn, J. T. de Souza, and S. Yoo. Search based software engineering: Techniques, taxonomy, tutorial. *Empirical Software Engineering and Verification*, 7007:1–59, 2012.

[29] L. Hong, J. Woodward, J. Li, and E. Özcan. Automated design of probability distributions as mutation operators for evolutionary programming using genetic programming. In *16th European Conference, EuroGP 2013*, Lecture Notes in Computer Science, pages 85–96, Vienna, Austria, 2013. Springer Berlin Heidelberg.

[30] U. Johansson, R. König, and L. Niklasson. Genetically evolved kNN ensembles. *Data mining*, 8:299–313, 2010.

[31] A. Joó, A. Ekart, and J. Neirotti. Genetic algorithms for discovery of matrix multiplication methods. *IEEE transactions on evolutionary computation*, 16(5):749–751, 2012.

[32] J. R. Koza. *Genetic Programming, on the programming of computers by means of natural selection*. The MIT Press, 1992.

[33] W. B. Langdon. Genetic improvement of programs (Keynote). In *18th International Conference on Soft Computing, MENDEL'12*, pages 7–12, 2012.

[34] W. B. Langdon and M. Harman. Evolving a CUDA kernel from an nVidia template. In P. Sobrevilla, editor, *IEEE Congress on Evolutionary Computation*, pages 1–8, Barcelona, Spain, July 2010. IEEE.

[35] W. B. Langdon and M. Harman. Genetically improved CUDA C++ software. In M. Nicolau, K. Krawiec, and M. Heywood, editors, *17th European Conference on Genetic Programming, EuroGP 2014*, Lecture Notes in Computer Science, pages 1–12, Granada, Spain, 2014. Springer Berlin Heidelberg.

[36] W. B. Langdon and M. Harman. Optimising Existing Software with Genetic Programming. *IEEE Transactions on Evolutionary Computation*, PP(99):1–18, 2014.

[37] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 3–13, Zurich, Swiss, June 2012. IEEE.

[38] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, 2012.

[39] M. A. Martin and D. R. Tauritz. Evolving black-box search algorithms employing genetic programming. In *GECCO'13, 15th annual conference on Genetic and evolutionary computation*, pages 1497–1504, Amsterdam, The Netherlands, July 2013. ACM Press.

[40] J. Mcdermott, D. R. White, S. Luke, L. Manzoni, M. Castelli, L. Vanneschi, W. Jaśkowski, K. Krawiec, R. Harper, K. D. Jong, and U.-M. O'Reilly. Genetic Programming Needs Better Benchmarks. In *GECCO'12, 14th annual conference on Genetic and evolutionary computation*, pages 791–798, Philadelphia, Pennsylvania, USA, July 2012. ACM.

[41] T. M. Mitchell. *Machine Learning*. McGraw Hill, 1997.

[42] S. Nguyen, M. Zhang, M. Johnston, and K. C. Tan. Evolving reusable operation-based due-date assignment models for job shop scheduling with genetic programming. In *15th European Conference, EuroGP 2012*, Lecture Notes in Computer Science, pages 121–133, Málaga, Spain, 2012. Springer Berlin Heidelberg.

[43] S. Nguyen, M. Zhang, M. Johnston, and K. C. Tan. Learning iterative dispatching rules for job shop scheduling with genetic programming. *The International Journal of Advanced Manufacturing Technology*, 67(1-4):85–100, Feb. 2013.

[44] P. Nordin and W. Banzhaf. Evolving Turing-Complete Programs for a Register Machine with Self-modifying Code. In *Genetic algorithms: proceedings of the sixth international conference (ICGA95)*, pages 318–325. Morgan Kaufmann, 1995.

[45] G. L. Pappa and A. A. Freitas. *Automating the Design of Data Mining Algorithms*. Natural Computing Series. Springer Publishing Company, Heidelberg, 1st edition, 2009.

[46] J. Petke, M. Harman, W. B. Langdon, and W. Weimer. Using Genetic Improvement & Code Transplants to Specialise a C++ Program to a Problem Class. In M. Nicolau, K. Krawiec, and M. Heywood, editors, *17th European Conference on Genetic Programming, EuroGP 2014*, Lecture Notes in Computer Science, Granada, Spain, 2014. Springer Berlin Heidelberg.

[47] J. Petke, W. B. Langdon, and M. Harman. Applying Genetic Improvement to MiniSAT. In G. Ruhe and Y. Zhang, editors, *5th International Symposium, SSBSE 2013*, Lecture Notes in Computer Science, pages 257–262, St. Petersburg, Russia, Aug. 2013. Springer Berlin Heidelberg.

[48] R. Sedgewick and K. Wayne. *Algorithms*. Addison Wesley, Westford, Massachusetts, USA, 4th edition, 2011.

[49] S. Smit and A. Eiben. Comparing parameter tuning methods for evolutionary algorithms. In *2009 IEEE Congress on Evolutionary Computation*, pages 399–406, Trondheim, May 2009. IEEE.

[50] J. Swan, M. G. Epitropakis, and J. R. Woodward. Gen-O-Fix: An embeddable framework for Dynamic Adaptive Genetic Improvement Programming. Technical Report January, Department of Computing Science and Mathematics, University of Stirling, Stirling, UK, 2014.

[51] W. Weimer, S. Forrest, C. Le Goues, and T. Nguyen. Automatic Program Repair with Evolutionary Computation. *Communications of the ACM*, 53(5):109–116, 2009.

[52] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *2009 IEEE 31st International Conference on Software Engineering*, pages 364–374, Vancouver, Canada, 2009. IEEE.

[53] D. R. White. *Genetic Programming for Low-Resource Systems*. Phd, University of York, 2009.

[54] D. R. White, A. Arcuri, and J. a. Clark. Evolutionary Improvement of Programs. *IEEE Transactions on Evolutionary Computation*, 15(4):515–538, Aug. 2011.

[55] J. Woodward and J. Swan. The automatic generation of mutation operators for genetic algorithms. In G. L. Pappa, J. Woodward, M. R. Hyde, and J. Swan, editors, *GECCO'12, 14th annual conference on Genetic and evolutionary computation*, pages 67–74, Philadelphia, Pennsylvania, USA, 2012.

[56] J. R. Woodward and J. Swan. Automatically designing selection heuristics. In *GECCO'11, 13th annual conference on Genetic and evolutionary computation*, page 583, New York, New York, USA, 2011. ACM Press.