Overview of search-based optimization algorithms used in software engineering

G. Mauša [1], T. Galinac Grbac [1], B. Dalbelo Bašić [2]

[1] Goran Mauša, University of Rijeka, Faculty of Engineering, Vukovarska 58, HR-51000 Rijeka, Croatia

[1] Tihana Galinac Grbac, University of Rijeka, Faculty of Engineering, Vukovarska 58, HR-51000 Rijeka, Croatia

[2] Bojana Dalbelo Bašić, University of Zagreb, Faculty of Electrical Engineering and Computing, Unska 3, HR-10000 Zagreb, Croatia

Keywords: software engineering, optimization, algorithms

Abstract Search-based software engineering is a novel and an emerging area of software engineering. It consists of search-based optimization algorithms used in various problem solving applications. Although search-based software engineering is becoming used in almost every area of software engineering, there are still areas that exploit its advantages less often than others. This paper presents how to use the search-based software engineering algorithms and which problem domains can they be used upon. It also explores a potential scope of further research within the area itself.

Introduction

Usage of search-based optimization algorithms in software engineering is an emerging and separate area of software engineering, called the search-based software engineering (SBSE). It is present in all sorts of optimization or multi-objective problems and every study indicates the advantages of solving a problem with such algorithms [1]. Many software engineering problems have too many possible solutions to a problem in their search space for the exhaustive search to finish in reasonable time. Sometimes it is sufficient to find one optimal solution or a “near optimal” solution to a problem [2]. In both cases, search-based optimization algorithms, often referred to as heuristic algorithms, may find its use in efficiently solving a problem or offering an insight into the range of possible solutions.

This paper presents an overview of current research in search-based software engineering and explores the potential scope of further research in the area. The next section describes the heuristic algorithms, along with requirements for their usage, general steps that are common to every algorithm, division of algorithms and the most often used ones. The following section presents a scope of some of the current research in the search-based software engineering area and the final section gives the conclusion.

Heuristic algorithms

Heuristic algorithms are a large group of search-based optimization algorithms. They originate from the desire to speed up the process of exhaustive search, which are too slow when dealing with problems with wide range of possible solutions. The simplest definition which presents the essence of heuristic algorithms is “proceeding by trial and error”, found in Oxford Reference Dictionary. Every heuristic algorithm employs some degree of randomness into the optimization procedure and skips a great deal of possible solutions in its search for an optimal one [3].

Requirements

There are two key requirements for using search-based optimization algorithms: search space and fitness function. The search space can be defined as *n*-dimensional space where an object of optimization can be found, where *n* represents the number of variable parameters that define the object. The larger the search space is, the greater is the need for search-based optimization algorithm usage. The fitness function is a quantitative measure of the object’s quality. It is the basis for comparison of different variations of the object, i.e. the candidate solutions we examine during optimization and it leads the search to an optimal or near optimal solution. The fitness function shows great flexibility, because it is defined according to the problem one is trying to analyze. This is the reason why search-based algorithms are applicable in various problem domains, covering all parts of software product life cycle.

General procedure

There are four general steps common to any heuristic algorithm, presented in figure 1. The initialization is the first phase of each heuristic algorithm and it is the only one that occurs without repeating. Most often, it involves random choice of initial candidate solution. Depending on the algorithm, it may be a single solution or a set of solutions. After initializing the iterative search for an optimal solution, the iterative search procedure may begin. First, the quality of a candidate solution needs to be checked and the fitness function, a key factor of any heuristic algorithm, comes into place. The modification of the candidate solution with the goal to find an optimal one is the second step of iterative search. Usually, the modification phase slightly changes the previously selected candidate solution and looks for neighboring solutions in the search space. Slightly changing the candidate solution means incrementing or decrementing the values of parameters that describe that candidate solution. The decision is the fourth step of the whole process, but the third step of iterative search process and it depends on the algorithm we choose to implement.

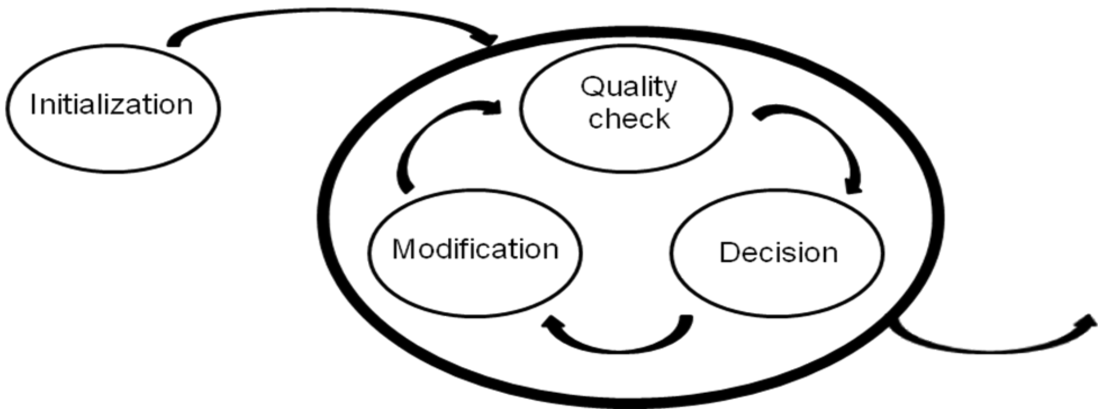

Fig. 1 Heuristic algorithm general steps

There are several reasons for an algorithm to stop the search. If the fitness function presents a well known parameter, the domain experts may be able to define a fitness function threshold that presents a solution that is good enough for their specific task. If the algorithm does not have such a stopping criteria, some algorithms stop after determining they are unable to find a better solution that the one they had already found. Either way, every algorithm must have a maximum number of iterations that prevents the algorithm from turning into an endless loop since the search is mostly random and does not keep track of previously examined solutions.

Most common algorithms

According to some differences in the previously explained general steps, the heuristic algorithms may be divided into following groups:

1. Single-State Methods and Population Methods
2. Local Search Methods and Global Search Methods

The single-state methods examine only one candidate solution at a time, while population methods use a larger set of candidate solution at each step of the search procedure. Using a set of solutions gives additional ways to modify the candidate solutions, often inspired from nature. This can be noticed in the mere names of algorithms like genetic algorithms, ant colony or swarm optimization. The second division refers to the performance of the algorithm. Local search methods look for an optimal solution on a local level, which can trap them in a local optimum. Global search methods, on the other hand, avoid local optimum traps and perform the search on a global level. An obvious question why would anyone use a single state local search method, with such obvious limitations and flaws may arise from the division we presented. The answer to that question is in the ever present trade-off between efficiency and effectiveness.

Heuristic algorithms, as a part of search-based software engineering, found its usage across all the stages of software product life cycle, from requirements to testing. The most common algorithms are genetic algorithms, genetic programming, hill climbing and simulated annealing [4]. Genetic algorithms are a group of population based, global search methods, inspired by nature. The modification phase of these algorithms involves gene mutation and recombination from the fittest parents with the goal to find even fitter offspring. Hill climbing and simulated annealing, on the other hand, are representatives of single state, local search methods. Hill climbing is inspired by climbers will to always go up the hill each step of the way, meaning it will choose a better solution each time there is one in the neighborhood. Simulated annealing found inspiration in metallurgy and unlike hill climbing, it can sometimes choose an inferior solution before next modification phase.

Search-based software engineering usage

SBSE is a widespread group of optimization techniques that found usage in almost every stage of software product lifecycle. However, there are still software engineering areas that exploit its benefits much more often than others, like software testing [5]. Here are some applications of SBSE in most recent studies:

- Solving the next release problem [5]–[7]
- Automatic software repair problem [8]–[13]
- Test data optimization problem [14]–[18]
- Generating higher order mutants [19], [20]
- Other multi-objective problems in software engineering [21], [22]

Next Release Problem

The next release problem is a multi-objective problem which requirements should appear in the next release of a software product. The task of SBSE is to choose a small subset among all possible requirements in order to satisfy as many stake-holders or customers as possible and to minimize the cost at the same time. Solving the next release problem is difficult and often has so many combinations of solutions that it becomes practically impossible to solve manually. Using SBSE does not give a specific answer to that problem but instead gives a range of equally good, near optimal solutions. That provides a valuable and necessary support for the decision maker. Zhang et al. [6] used search-based optimization techniques to automate the search for optimal or near optimal allocations of requirements that balance competing stake holder objectives in next release problem. NSGAII and two-archive algorithm were compared in performance and the two-archive algorithm proved to the better one. Durillo et al. [5] analyzed which features to include in the next release of product in order to satisfy as many customers as possible with minimal cost using 3 multi-objective metaheuristics. NSGA-II was used as a reference algorithm in the field of multi objective optimization, MOCell because it outperformed NSGA-II in several studies and PAES as one of the simplest techniques. The results showed that NSGA-II finds the highest number of optimal solutions, MOCell finds the widest range of different solutions and PAES was fastest but worst, as expected. Finkelstein et al. [7] made a proposition that each notion of fairness in next release problem should form an objective in multi-objective, pareto optimal SBSE setting. Comparing the NSGA-II and the two-archive algorithm, the results showed that they performed equally well with random data sets, while NSGA-II performed better with real data set obtained from Motorola.

Automatic Software Repair

Fixing bugs is a difficult and time-consuming manual process and some reports say software maintenance usually consumes about 90% of all costs after delivery of a product. An efficient, fully automated technique for repairing program defects could, therefore, alleviate that heavy burden. The idea of using SBSE in automatic software repair is actually rather simple but efficient. First task is to locate the region of the program relevant to an error. An SBSE algorithm is then used to produce simple changes along the path where the fault lies, trying to eliminate the fault and maintain the functionality of original program at the same time. Weimer et al. [8] used genetic programming for software repair phase in fully automated method for locating and repairing bugs in software. Fast et al. [9] used genetic programming in automated program repairing as well, but their focus was in improving fitness-function which resulted in efficiency improvement of 81%. Forrest et al. [10] combined genetic programming with program analysis methods to repair bugs in the off-the-shelf legacy C programs and managed to repair all of 11 programs they used. Schulte et al. [11] explored the advantages of assembly-level repair using evolutionary computing in automated program repair problem. Having all the advantages of assembly level approach over source code level approach, they obtained nearly as efficient results as at source code level. Nguyen et al. [12] successfully used genetic programming approach to automate repairing program bugs in existing software with average running time from half a second to ten minutes. Weimer et al. [13] also successfully used genetic programming in automatic bug repair in off-the-shelf legacy C programs combining program analysis methods with evolutionary computing. Their approach requires 1428 seconds and 3903 fitness evaluations per constructing a repair on average. All of these promising results indicate SBSE being the proper choice for automated software repair.

Test Data Optimization

Test data generation, regeneration and minimization are three different approaches to test data optimization problem. They tend to identify near optimal test sets in reasonable time using Evolutionary testing, a sub-field of SBSE techniques, often called Search-based Testing. Test data generation begins from scratch, regeneration uses the pre-existing test data as a starting point and test suite minimization tends to identify and remove redundant test cases.

The problem how to test something that reacts in different manner to the same test input over time can be found when dealing with autonomous agents. Nguyen et al. [14] proposed a solution to that problem and used evolutionary optimization to generate demanding test cases for such autonomous agents that produce different output to the same input, due to increasing knowledge for example. NSGA-II algorithm as a fast multi-objective genetic algorithm that in other studies proved to be better in finding widely spread solutions and with better convergence to the optimum compared to other algorithms was used in their study. Harman and McMinn [15] analyzed which type of search is best for structural test data generation problem. They used evolutionary testing as a global search algorithm, hill climbing as a local and a hybrid memetic algorithm for that purpose. Their suspicion proved to be correct and the results showed that hybrid approach, in terms of coverage, is capable of best overall performance. McMinn et al. [16] also compared evolutionary testing as a global search algorithm, hill climbing as a local search and a hybrid memetic algorithm in search-based structural test data generation problem, but with the improvement of irrelevant input variable removal (INVR). The results expectedly showed that all search algorithms are more successful and cover more branches with INVR and that memetic algorithm is the most prolific technique at successfully finding significance with and without INVR.

There are many testing scenarios in which a tester may already have some pre-existing test cases. They could have been created by tester, based on his experience, expertise and domain knowledge, by developer or they may be present from regression testing of previous version of product. In order to exploit the effort and knowledge put together to form this test cases, Yoo and Harman [17] proposed using pre-existing test data as a starting point in search-based test data generation, making it a regeneration process. They used hill climbing without random initialization, but with random-first-ascent and pre-existing test data. The results were promising, indicating the proposed approach can be up to two orders of magnitude more efficient, achieve higher structural coverage and equal component level of mutation score with much lower cost compared to a state-of-the-art search-based testing technique.

Test suite minimization can prove to be very important in strict time limits, often given in regression testing. Regression testing has to guarantee that the recent changes in a program do not interfere with its functionality. Due to ever growing test suite, it is prohibitively expensive to execute the entire test suite. Yoo and Harman [18] analyzed the hybrid algorithm that combines the efficient approximation of the greedy algorithm with the capability of population based genetic algorithm for pareto efficient multi-objective test suite minimization. They found greedy algorithm may provide a good approximation of pareto front in smaller software products, but for larger products, the usage of HNSGA-II is suggested due to more precise test suite minimization.

Higher Order Mutation Generation

It is said that 90% of faults which survived the testing procedure have to be complex ones. In order to reduce their number, we need to learn more about them. Higher order mutants (HOMs) are deliberately faulty programs used in software testing process. The order of mutant reflects the number of injected faults into the original program. Finding higher order mutants that create subtle and complex faults and sometimes practically mask one another helps us locating this hazardous combination of otherwise harmless faults when they exist separately. Such HOMs are more difficult to find with simple test cases and there lies a possible usage of HOMs - to find better test cases.

Jia and Harman [19] compared the performance of 3 algorithms for finding optimal HOMs: greedy algorithm, genetic algorithm and hill climbing algorithm. The results showed that genetic algorithm performed best because it finds most subsuming HOMs, hill climbing always finds the highest fitness HOMs and greedy algorithm finds the highest order HOMs. What makes their findings questionable is the fact that random search found more HOMs that greedy algorithm and hill climbing. Langdon et al. [20] explored the usage of multi-objective pareto optimal approach with Monte Carlo sampling, NSGA-II genetic algorithm and genetic programming to search for higher order mutants which are both harder to kill and more realistic. They found their higher order genetic programming mutation testing approach able to find even such simple faults that in combination mask each other and therefore form complex faults very difficult to detect.

Other Multi-Objective Problems in Software Engineering

Besides next release problem, there are many more problems which involve incomparable and often opposite multiple objectives in software engineering. A well known and highly important and challenging problem in software engineering is the one that requires high degree of cohesion and low degree of coupling for a good module structure. This problem is even more intensified as software evolves and its modular structure tends to degrade. Praditwong et al. [21] compared automated techniques for suggesting software clustering, delimiting boundaries between modules that maximize cohesion and minimize coupling. They used hill climbing as a single-objective algorithm and two-archive algorithm as a multi-objective with 2 different approaches: the maximizing cluster approach (MCA) and equal-size cluster approach (ECA). The results indicated that ECA is superior to MCA and hill climbing in this task.

Many other multi-objective, but completely different problems in software engineering can be found in software engineering management. Project managers often do not understand the complex optimization techniques and they do not need to, in order to benefit from them. It is important to provide them with tool they can easily give input to and to obtain visually acceptable output that can help them in making important decisions. One such problem was analyzed by Di Penta et al. [22] where they used 3 metaheuristics in staff and task allocation with the objective to minimize the completion time and reduce schedule fragmentation. They compared the performance of NSGA-II, stochastic hill climbing and simulated annealing as most widely used SBSE techniques that appear in 80% publications. The results were compared in single objective optimization approach where simulated annealing was the best algorithm.

Conclusion

Unlike other engineering disciplines where search-based algorithms found application in, software engineering is the only discipline whose artifacts are solely virtual. The mere lack of possible simulations or models which can represent and optimize its artifacts makes SBSE and used fitness function the closest thing to an artifact. This property makes SBSE very attractive and potentially beneficial field. Search-based algorithms are attractive in software engineering also due to the fact that the data in software engineering are often inaccurate, over dispersed and incomplete, making some traditional optimization techniques inappropriate.

Software testing exploited the search-based algorithms more than any other software engineering field. We showed the basic requirements and potential application for these algorithms in other optimization or multi-objective problems from various software engineering areas. Besides making use of obviously beneficial algorithms in other unexplored areas, there is also work to be done with the algorithms themselves. The emerging hybrid algorithms show very promising results and offer a potential scope for future research. To sum up, the potential of using SBSE is vast and still needs to be explored more thoroughly.

References

[1] M. Harman and A. Mansouri: Search based software engineering: Introduction to the special issue of the IEEE transactions on software engineering, Volume 36(6), p.p. 737–741, 2010.
[2] S. Luke: Essentials of Metaheuristics (Lulu, 2009), Available for free at http://cs.gmu.edu/_sean/book/metaheuristics/
[3] D. L. Kreher, D. R. Stinson: Combinatorial Algorithms: Generation, Enumeration and Search. (CRC Press LLC, 1999)
[4] G. Lu, R. Bahsoon, and X. Yao: Applying Elementary Landscape Analysis to Search-Based Software Engineering, SSBSE '10, p.p. 3-8, 2010
[5] J. J. Durillo, Y. Zhang, E. Alba, M. Harman and A. J. Nebro: A study of the bi-objective next release problem. Empirical Software Engineering, Vol. 16(1), p.p. 29–60, February 2011
[6] Y. Zhang, M. Harman, A. Finkelstein, and S. A. Mansouri: Comparing the performance of metaheuristics for the analysis of multi-stakeholder tradeoffs in requirements optimization, Information and Software Technology, Vol. 53(7), p.p. 761–773, July 2011
[7] A. Finkelstein, M. Harman, S. A. Mansouri, J. Ren, and Y. Zhang: A search based approach to fairness analysis in requirement assignments to aid negotiation, mediation and decision making, Requir. Eng., Vol. 14, p.p. 231–245, October 2009
[8] W. Weimer, T. Vu Nguyen, C. Le Goues, and S. Forrest: Automatically finding patches using genetic programming, IEEE Computer Society, In Proc. of the ICSE '09, p.p. 364–374, 2009.
[9] E. Fast, C. Le Goues, S. Forrest, and W. Weimer: Designing better fitness functions for automated program repair. In Proc. of 12th annual conference GECCO '10, p.p. 965–972, 2010
[10] S. Forrest, T. Vu Nguyen, W. Weimer, and C. Le Goues: A genetic programming approach to automated software repair, in Proc. of the 11th Annual conference GECCO '09, p.p. 947–954, 2009
[11] E. Schulte, S. Forrest, and W. Weimer: Automated program repair through the evolution of assembly code, in Proc. of the IEEE/ACM international conference ASE '10, p.p. 313–316, 2010
[12] T. Vu Nguyen, W. Weimer, C. Le Goues, and S. Forrest: Using execution paths to evolve software patches, in Proc. of the IEEE International Conference on ICSTW '09, p.p. 152–153, 2009
[13] W. Weimer, S. Forrest, C. Le Goues, and T. Vu Nguyen: Automatic program repair with evolutionary computation, Commun. ACM, Vol. 53, p.p. 109–116, May 2010
[14] Cu D. Nguyen, A. Perini, P. Tonella, S. Miles, M. Harman and M. Luck: Evolutionary testing of autonomous software agents, in Proc. of The 8th International Conference AAMAS '09, Vol. 1, , p.p. 521–528, 2009
[15] M. Harman and P. McMinn: A theoretical and empirical study of search-based testing: Local, global, and hybrid search, IEEE Transactions on Software Engineering, Vol. 36, p.p. 226–247, March 2010
[16] P. McMinn, M. Harman, K. Lakhotia, Y. Hassoun and Joachim Wegener: Input domain reduction through irrelevant variable removal and its effect on local, global and hybrid searchbased structural test data generation, IEEE Transactions on Software Engineering, 2011
[17] S. Yoo and M. Harman: Test data regeneration: Generating new test data from existing test data. Journal of Software Testing, Verification and Reliability, To appear
[18] S. Yoo and M. Harman: Using hybrid algorithm for pareto efficient multi-objective test suite minimization, Journal of Systems and Software, Vol. 83, p.p. 689–701, April 2010
[19] Y. Jia and M. Harman: Higher order mutation testing, Information and Software Technology, Vol. 51, p.p. 1379–1393, October 2009
[20] W. B. Langdon, M. Harman, and Y Jia: Efficient multiobjective higher order mutation testing with genetic programming, Journal of Systems and Software, Vol. 83, p.p. 2416–2430, December 2010
[21] K. Praditwong, M. Harman, and X. Yao: Software module clustering as a multi-objective search problem. IEEE transactions on software engineering, Vol. 37, p.p. 264–282, March 2011
[22] M. Di Penta, M. Harman, and G. Antoniol: The use of search-based optimization techniques to schedule and staff software projects: an approach and an empirical study, Software: Practice and Experience, Vol. 41, p.p. 495–519, April 2011