Accelerating SAT solving with best-first-search*

Dávid Bartók and Zoltán Ádám Mann
Department of Computer Science and Information Theory
Budapest University of Technology and Economics
Hungary

Abstract

Solvers for Boolean satisfiability (SAT), like other algorithms for NP-complete problems, tend to have a heavy-tailed runtime distribution. Successful SAT solvers make use of frequent restarts to mitigate this problem by abandoning unfruitful parts of the search space after some time. Although frequent restarting works fairly well, it is a quite simplistic technique that does not do anything explicitly to make the next try better than the previous one. In this paper, we suggest a more sophisticated method: using a best-first-search approach to quickly move between different parts of the search space. This way, the search can always focus on the most promising region. We investigate empirically how the performance of frequent restarts, best-first-search, and a combination of the two compare to each other. Our findings indicate that the combined method works best, improving 36-43% on the performance of frequent restarts on the used set of benchmark problems.

1 Introduction

The Boolean satisfiability problem (SAT) is one of the most studied algorithmic problems in computer science. It is of significant theoretical interest (e.g., it was the first problem shown to be NP-complete) and it has a number of important practical applications, such as in hardware and software verification [17]. Accordingly, substantial research effort has been invested into the development of highly efficient SAT solvers. As a result, modern SAT solvers can often solve even some huge problem instances quite quickly [9]. The development of ever more efficient SAT solvers is also fuelled by the yearly SAT competition (`http://satcompetition.org/`).

The tremendous success of SAT solvers can be attributed to – among other techniques – clever heuristics that guide the search to the most promising parts of the search space. When these heuristics work well, then the solver will finish rather quickly. However, in other, less lucky cases, the solver may spend orders of magnitude more time in unfruitful parts of the search space. As a result, SAT solvers – like other algorithms for NP-hard problems – tend to exhibit large variance in their runtime. This applies especially to solvable instances, where a lucky series of choices may lead the search directly to a solution, but in the worst case, an exponential number of steps are needed. In order to mitigate the effect of unlucky decisions, most successful SAT solvers use restarting: when a given number of steps elapses without being able to solve the problem, the algorithm is restarted in the hope that the next try will be more successful [9].

Apparently, frequent restarting is quite successful in interrupting long useless searches. Nevertheless, we felt a potential for improvement: our aim is that the search should – instead of simply restarting to *some* new direction – follow the direction that seems *best* based on the information that we have gathered so far about the search space.

Such an approach has been suggested for the constraint satisfaction problem (CSP) [19]. The idea is to split the search space into some parts, and create a solver instance for each of these parts.

*This paper was published in the *Proceedings of the 15th IEEE International Symposium on Computational Intelligence and Informatics*, pp. 43-48, IEEE, 2014

In every iteration, the most promising solver instance is selected, as determined by a heuristic evaluation function, based on some statistical features of the solver instances' past performance. The chosen solver instance is run for a predefined number of steps. Then, again the most promising solver instance is selected etc. As shown in [19], this best-first-search (BFS) approach can speed up a backtracking-style algorithm for CSP more than frequent restarts. Our aim is to adapt this approach to SAT.

This is by no means trivial. Although SAT can be regarded as a special case of CSP, and most exact SAT solvers are also based on backtracking, modern SAT solvers are much more sophisticated than the backtracking algorithm considered in [19]. In particular, the algorithm in [19] did not contain any learning, while clause learning is a fundamental ingredient of modern SAT solvers. In fact, clause learning – supposing that learned clauses are kept between restarts – in itself mitigates some of the shortcomings of frequent restarts: it makes sure that the restarted search will avoid the dead-ends that it has already encountered before the restart. Therefore, the combination of restarting and clause learning is a very powerful technique in SAT solving. Nonetheless, restarting is still not a very intelligent method; this is why the more sophisticated BFS approach may yield better results, especially for satisfiable instances.

In this work, we extended MiniSat, a typical example of a modern CDCL (conflict-driven clause learning) SAT solver [7], with the BFS technique. We tried two different approaches: (1) replacing restarting completely with BFS and (2) combining restarting and BFS. In contrast to [19], we found that replacing restarting with BFS does not improve the solver's performance, probably due to the above-mentioned effect of clause learning. However, combining restarting and BFS (which was not considered in [19]) decreases both the mean and the variance of the runtime considerably.

It should be noted that the resulting solver is a sequential and exact algorithm, just like the original MiniSat.

2 Previous work

Modern SAT solvers are mostly descendants of the Davis-Putnam-Logemann-Loveland (DPLL) algorithm from the early 1960s [6, 5], a kind of backtrack search. Since then, several extensions and refinements have been proposed, leading to today's prevalent CDCL solvers, with prominent examples like GRASP [22], Chaff [20], MiniSat [7], and Glucose [2].

The high variance of the runtime of backtrack search algorithms was observed and analyzed by several researchers [15, 10, 13]. In particular, Gomes et al. introduced the notion of heavy-tailed runtime distributions to describe this phenomenon [10]. Gomes and co-workers also suggested that frequent restarts could remedy the problem of heavy tails [8]. Since then, frequent restarts have become an integral part of most successful solvers. Also, several different schemes have been suggested concerning the frequency with which restarts should be carried out during the course of the algorithm [16, 23, 4, 14, 11]. Experience has shown that it is useful to start with short runs interrupted by frequent restarts, and then gradually increase the restart interval. (The restart interval is either the time or the number of steps between subsequent restarts, e.g., the number of propagations or number of conflicts.) Many solvers use an exponential strategy, in which the restart interval is multiplied with a constant greater than one after each restart.

Another related approach suggested in the SAT literature is cube-and-conquer [12]. Similarly to our approach, cube-and-conquer splits the search space into disjoint parts (called cubes) that can be solved independently. Although cube-and-conquer uses a quite sophisticated method – a so-called look-ahead solver – for the splitting step, the solving step, in which the resulting sub-problems are solved by a CDCL solver, is implemented in a straight-forward manner, either one cube after the other or in parallel. In contrast, the focus of our approach is to apply a sophisticated method to solve the sub-problems in an intelligently interleaved way.

3 Preliminaries

3.1 SAT solvers

In the *SAT problem*, we are given n Boolean variables and m clauses, where each clause is the disjunction of a set of literals, and a literal is a variable or its negation. The aim is to decide whether there is an interpretation of the variables that satisfies all clauses.

CDCL-type solvers attack this problem by searching the space of partial solutions according to the following scheme [9]. First, a variable that has no value yet is selected and is set to one of its two possible values. From this decision, potentially other information can be inferred, e.g. using unit propagation: if there is only one unassigned literal in a clause and all assigned literals are false, then the last literal must be true.

If all variables are assigned values and no conflict has arisen, then a solution has been found and we are done. If there is a conflict in the current partial solution – i.e., there is a clause in which all literals are false –, then the reasons for this conflict are analyzed: the algorithm determines which assignments are responsible, directly or indirectly, for this conflict. As a result of the analysis, a new clause is learned that makes sure that the same set of assignments will not be made in the future. Afterwards, the search jumps back one or more levels in the search tree, such that at least one variable in the learned clause becomes unassigned.

If there is neither a conflict nor has a solution been found, then the search continues by again assigning a value to one of the free variables, and so on. After a defined number of steps, if the algorithm has not succeeded in finding a solution nor in proving that there is no solution, then the search is restarted. Usually, the restart interval is increased with each restart.

Whenever the database of learned clauses becomes too big, it is reduced by removing some of the learned clauses that did not prove to be useful.

Beyond this basic scheme, CDCL solvers use many other techniques to improve efficiency, such as lazy data structures for storing the current status of the variables and clauses, as well as sophisticated heuristics for choosing the next variable to branch on etc., but these are less relevant for our purposes.

3.2 Best-first search (BFS)

BFS is a standard technique in artificial intelligence to decrease the time needed to traverse the search space of a problem in a tree-like manner, by making use of an intelligent evaluation of the possibilities at each decision point [21]. In contrast, backtrack search algorithms, like the DPLL algorithm which is at the heart of most exact SAT solvers, traverse the search space in a depth-first-search (DFS) manner [5, 6]. DFS is a rather "insistent" approach: if it starts visiting a branch of the tree, it does not leave that branch until it is completely traversed. BFS is more flexible: it allows leaving the branch temporarily if another branch seems more promising, and come back later if necessary. For this purpose, BFS employs a heuristic evaluation function to determine which of the open branches of the search tree is most likely to contain a solution.

In [19], it was suggested to use BFS to speed up backtrack search. Clearly, BFS causes some overhead: it requires computing, storing, and comparing the values of the evaluation function, as well as switching of search state. In order to minimize this overhead, the idea of [19] was to perform the BFS-style decision-making only occasionally, and otherwise carry on according to the standard DFS strategy. To implement this idea, it was suggested to split the search space into multiple segments, and run one solver instance per search space segment. Each solver instance carries out the standard backtrack search, but confined to the given search space segment. One solver instance is run with a limit on the number of steps it is allowed to make; when this number of steps is reached, control is transferred to the solver instance that receives the highest evaluation score according to the BFS heuristic, which is again run for a predefined number of steps, and so on.

In the approach of [19], all but one of the search instances are confined to a subtree of the search tree. Moreover, there is a special search instance – called the initial search instance – that

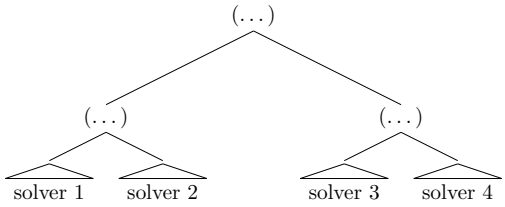

Figure 1: Dividing the search tree among solver instances

searches those parts of the search tree that are not contained in any of the subtrees, in order to guarantee the completeness of the search. The initial search instance starts in the leftmost branch of the search tree. Whenever the initial search instance reaches the starting node of one of the subtrees, it is merged with the search instance working in that subtree. For this approach to work, it is important that the search instances operate in the same search tree, i.e., the variables must be assigned values in the same order by all search instances.

As shown in [19], this approach works very well for constraint satisfaction problems.

4 Adapting best-first-search to SAT

BFS can be used for SAT as well by transferring the basic idea from [19]. However, several details need to be changed or worked out for SAT, as shown below. In particular, the following challenges must be addressed:

- The variable selection heuristic in typical CDCL solvers is dynamic, meaning that the next variable to branch on is decided on the fly, based on information gathered during the search so far. For the different search instances, this can be different. As a consequence, the approach of [19] regarding the initial search instance and merging of solver instances does not work with a CDCL solver.
- The evaluation function that estimates the usefulness of search instances must be defined, tailored to the specifics of SAT solving. This means choosing the right characteristics based on which the efficiency of a search instance can be well predicted, also taking into account how easily those characteristics can be measured in a CDCL solver.
- Clause learning or other learning mechanisms were not considered in [19], hence it must be investigated how clause learning affects the BFS mechanism. Furthermore, since clause learning also makes restarting more powerful, it is not clear whether restarting should really be completely replaced by BFS or the two techniques can be somehow combined.

In the following, we describe in detail how we addressed these challenges.

4.1 Splitting the search space

Since the mechanism of [19] with the special initial search instance cannot be used, we devised another approach, which is actually simpler than the one of [19]. The idea is to partition the search tree into a set of subtrees such that they are disjoint and cover all leaves of the search tree. This way, the subtrees can be searched independently, without losing the completeness of the search. No merging is necessary, and there is no restriction on the variable selection order within the subtrees.

As shown in Figure 1, the search tree is divided into 2^k subtrees with roots on level k of the search tree, and a solver instance is created for each subtree. That is, the values for k variables are assigned in all possible ways. These first k variables must be the same for all solver instances, but afterwards, the solver instances can choose the variables independently from each other, according

```
BFS(SolverList L)
{
    Solver currentSolver;
    int limit=startLimit;
    boolean satisfied=false;
    while(NOT(satisfied) AND L.existsActiveSolver())
    {
        currentSolver=L.findMaxHeuristic();
        limit=increaseLimit(limit);

        retCurr=currentSolver.solve(limit);

        if(retCurr == UNRESOLVED)
            recalculateHeuristic(currentSolver);

        else if(retCurr == SAT)
            satisfied=true;

        else if(retCurr == UNSAT)
            L.remove(currentSolver);
    }

    return satisfied;
}
```

Listing 1: The BFS control algorithm

to their normal variable selection heuristics. In practice, k will be a small number compared to the number of all variables, so that the impact on the effectiveness of the variable selection will be minimal.

For the choice of the first k variables, we implemented two strategies: (i) choosing them randomly and (ii) choosing the variables that participate in the highest number of clauses. The idea behind the second strategy is that this way the solver instances will work in significantly different search space segments.

4.2 Operating the search instances

Each solver instance gets as input a tuple of values for the k variables, and starts by assigning these values to them. This also includes the propagation of information from the assignments (mainly by unit propagation). In the following, the solver instance works according to the normal CDCL procedure. However, backjumping higher than level k of the search tree is not allowed. If a solver instance must undo one of the k initial assignments, this means that there is no solution in the subtree of the given solver instance; thus, this solver instance has finished its execution.

If a solver instance finds a solution, then the problem is solved and the algorithm terminates. Otherwise, the solver instance is stopped after a pre-defined limit L on the number of conflicts. Similarly to the method generally used with restarts, we multiply L with a number greater than 1 each time a solver was stopped.

When the currently running solver instance is stopped, we select – using the evaluation function – the solver instance to run next and transfer control to it (with the new limit on the number of conflicts). If the solver instance stopped because it finished exploring its search space segment, then, also, the solver instance to run next is selected. The only difference is that in this case we must mark the finished solver instance as not active anymore and hence exclude it from the set of solver instances from which we will select in the future. Moreover, if there is no more active solver instance, then it is established that the problem has no solution, and the algorithm can terminate.

The details of the algorithm are given in pseudo-code in Listing 1.

4.3 The evaluation function

The most crucial part is how to select the solver instance to run next. For this, we need a heuristic evaluation function that can estimate how quickly a solver instance is likely to find a solution, based on its history of previously explored parts of the search tree. We identified the following quantities that may help to predict this and are not hard to collect:

- Depth, i.e., the number of assigned variables. The intuition is that the deeper the solver instance is or has been, the nearer it is or has been to a solution, and hence the more likely it is to find a solution soon. Specifically, we store the current depth of each solver instance, the biggest depth it has reached so far, and its average depth.
- Decision level, i.e., number of variables that were assigned values by decision. This is similar to depth, with the difference that the decision level does not contain the number of variables that were assigned values by propagation. We store for each solver instance its current decision level and the maximum reached so far.
- Average length of learned clauses. Shorter clauses are more useful because they become unit clauses sooner and thus can be used sooner in unit propagation. Therefore, the lower the average length of learned clauses, the more efficient the solver instance is likely to be in the future.
- Number of steps already taken: if the solver instance has already made many steps without success, this suggests that it is unlikely to succeed in the near future.

The evaluation function is the weighted sum of the above quantities. The weights can be tuned to optimize algorithm performance. We will come back to this in Section 5.

It is important to note that the lazy data structures commonly used in modern SAT solvers, primarily the watched literals scheme [9], drastically limit the set of information that is readily available. For example, the average number of free variables per clause is not known explicitly during the search in modern SAT solvers; hence, we do not rely on such information. Indeed, all of the above metrics can be easily recorded in a modern SAT solver, as demonstrated by our implementation.

4.4 Clause learning and restarting

In our current implementation, clause learning is done locally. That is, each solver instance grows (and sometimes reduces) its own database of learned clauses. In principle, it would also be possible to share the learned clauses, or some of them; this is a possibility for further experimentation. Otherwise, clause learning is not affected by our modifications.

Concerning the role of restarts, we implemented two versions. In the first version, restarts are not used at all, their role is completely taken over by BFS. In the second version, we combine BFS with restarts: every time we transfer control to a solver instance, we restart it from the root of its subtree, but keep its database of learned clauses.

5 Empirical results

We implemented the described BFS scheme as an extension to MiniSat, one of the most successful and popular CDCL solvers. This was quite straight-forward, because the data structures and the algorithm of MiniSat are encapsulated in a class Solver, allowing the creation of multiple Solver objects. Moreover, MiniSat supports the notion of *assumptions* [7], which we leveraged for the initial k variable assignments.

Our implementation relies on MiniSat 2.2.0 with its default configuration. The implementation was carried out in C++.

For the subsequent experiments, we used BCAT [18], a program for the systematic experimentation with algorithms. All measurements were carried out on a desktop PC with 2.6 GHz Pentium E5300 Dual-Core CPU and 3 GB DDR2 800MHz RAM, running MS Windows 7.

5.1 Parameter tuning

In a first experiment, we wanted to find the best configuration of the algorithm's parameters: number of solver instances, weights in the evaluation function, L (limit on the number of steps before control is transferred to the newly selected solver instance), increase rate for L, method for choosing the k initial variables, use of restarts.

Since the parameter space is quite large, it was not feasible to test all parameter configurations thoroughly. Hence, we used the following methodology. After some experimentation, we fixed all parameters to a plausible value. Then, for each parameter, we tried 4 very different values, while leaving the other parameters unchanged. We fixed each parameter to the value that proved best. Then, we made one more iteration, again trying 4 different values for each parameter, but this time with smaller increments around the value that was chosen previously. Each configuration was evaluated on 50 test problem instances with roughly 4,500 variables and 50,000 clauses, encoding quasigroup completion problem instances [1]. The following configuration yielded the best results:

- number of solver instances: 64 (i.e., $k = 6$)
- weights in the evaluation function:
  - current depth: 200
  - maximum depth: -100
  - average depth: 200
  - average learnt clause length: -100
  - current decision level: 200
  - maximum decision level: 200
  - number of steps: -25
- L: 100
- increase rate for L: 1.1
- method for choosing the k initial variables: most frequent variables
- use of restarts: yes

Most of these results are in line with our expectations. For instance, we were expecting that choosing the most frequent variables for the initial assignments is better than using randomly chosen variables. Also, the usage of restarts is important; actually, our findings indicate that simply replacing restarts with BFS, as was done successfully in [19], does not improve the performance of the solver at all in our case. We attribute this difference to the effect of clause learning. On the other hand, the combination of BFS and restarts does improve performance, as we will show next.

It is also interesting to look at the sign of the weights in the evaluation function. Most of them are as expected, e.g., the weight of the current depth is positive: the bigger the current depth, the better the solver instance; the weight of the average learnt clause lengths is negative because shorter clauses are better. The only surprise is the negative weight (-100) of the maximum depth. A possible explanation can be given if we look at it together with the positive weight (200) of the maximum decision level. Since depth is the sum of the decision level and the number of inferred variables, this means that the contribution of decisions to depth is more important in the evaluation of a solver instance than the contribution of propagations.

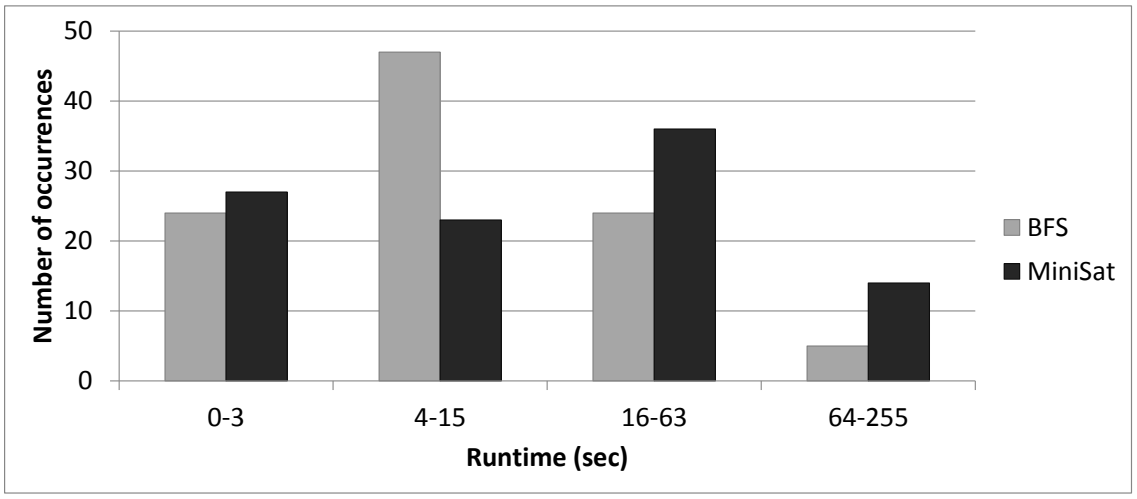

Figure 2: Runtime distribution on quasigroup completion problems

5.2 Comparison

In a second experiment, we used 100 problem instances (again quasigroup completion problems with roughly 4,500 variables and 50,000 clauses), to compare the runtime distribution of the proposed BFS algorithm with the base version of MiniSat. The results are shown in Figure 2.

As can be seen, the overall performance of BFS is significantly better than that of MiniSat. For example, the number of problem instances solved within 16 seconds is 71 for BFS versus 50 for MiniSat. The average running time was 17.43 seconds for BFS versus 27.07 seconds for MiniSat, a 36% improvement.

We used quasigroup completion problems for our evaluation because the hardness of such problem instances can be precisely tuned, thus we could easily generate a number of instances that are neither too easy nor too hard for MiniSat. As pointed out in [1], quasigroup completion problems have an interesting combination of a global structure and random elements, making them ideal for benchmarking algorithm performance. The quasigroup instances in our experiment were generated with lsencode 1.1 (`http://www.cs.cornell.edu/gomes/SOFT/lsencode-v1.1.tar.Z`), creating so-called "regular" filled squares of size 37*37 and poking 44% holes using the "balanced" setting.

However, we also wanted to test how BFS performs on other problems. Therefore, in a next experiment, we tested the algorithm on a completely different set of problem instances, without any additional tuning. These are 30 instances with roughly 1,500 variables and 7,500 clauses, encoding prime factorization problems, generated with ToughSAT (`http://toughsat.appspot.com/`). The results can be seen in Figure 3. It can be observed once again how BFS helps in decreasing the heavy tail. For example, the number of instances requiring more than 64 seconds is 8 for MiniSat, but only 3 for BFS. The average running time is 26.33 seconds for BFS versus 43.97 seconds for MiniSat, a 40% improvement.

The instances mentioned so far are all structured: they are crafted combinatorial problems representing specific application domains. We also wanted to test on problems without any kind of structure. Therefore, for a final experiment, we generated random instances with the same method that was used for the Random tracks of SAT Competition 2013 [3], with the same generator (`http://sourceforge.net/projects/ksatgenerator/`). We generated 30 instances; all of them are satisfiable and contain 102 variables and 6120 clauses, where every clause has exactly 7 variables.

The results on the random instances are displayed in Figure 4. Once more it can be observed how BFS reduces the heavy tail of the runtime distribution. The average running time again shows a significant improvement: 12.53 seconds for BFS versus 21.83 seconds for MiniSat, meaning a

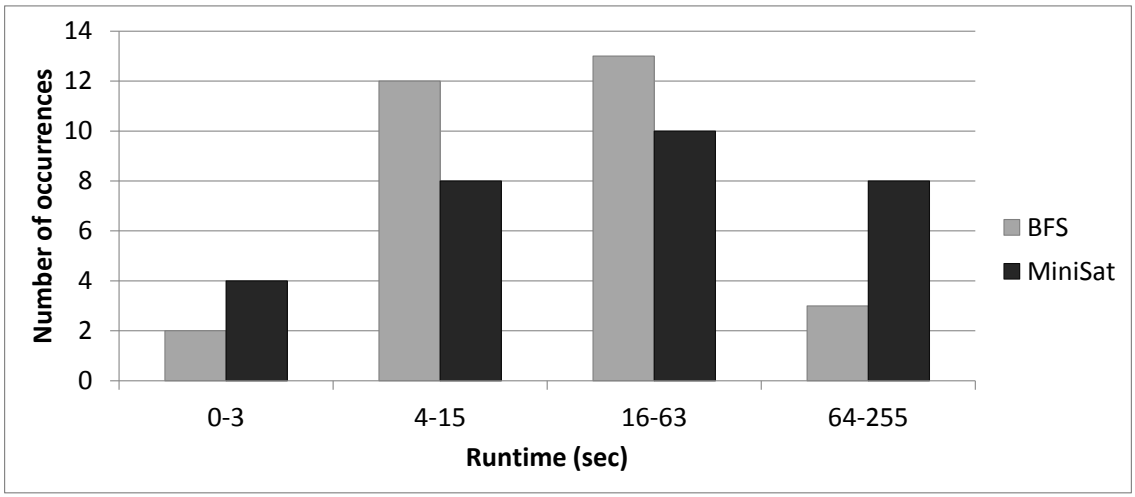

Figure 3: Runtime distribution on prime factorization problems

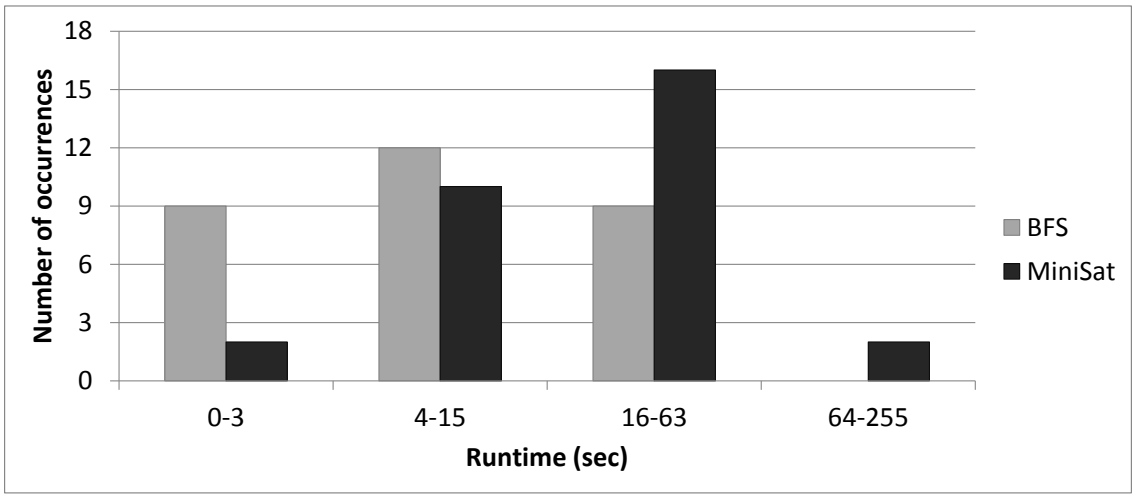

Figure 4: Runtime distribution on random problems

43% decrease.

The fact that the parameter configuration that we found best for quasigroup completion problems also yielded good results on prime factoring and random instances demonstrates the stability and robustness of the method, which is very important for its practical applicability, also in new problem domains.

6 Conclusions and future work

In this paper, we proposed an extension to current CDCL-type SAT solvers, based on a best-first-search approach. Our algorithm allows to regularly re-focus the search on the part of the search space that looks most promising based on the experience gained so far. We extended MiniSat with the proposed method and investigated its performance empirically. The test results show that the new method can help significantly in cutting off overly long runs and thus reducing the average running time of the solver.

Our future work will focus on the possibilities of the sharing of learned clauses between solver instances as well as the investigation of how the optimal parameter configuration depends on the characteristics of the problem instance. Moreover, we will work on the parallelization of our BFS algorithm. This is a very promising direction because our approach inherently splits the search space into segments that can be searched independently; thus we can expect an almost linear speedup from using more computational resources in parallel.

We are also currently experimenting with the implementation of our BFS method on top of Glucose, another CDCL SAT solver that has been shown to outperform MiniSat [2].

When dealing with large-scale problem instances, memory consumption becomes critical, since multiple solver instances share the limited memory of the computer. Therefore, we also need to devise mechanisms to handle memory constraints, such as adaptive heuristics to limit the number of solver instances.

Acknowledgements

This work was partially supported by the Hungarian Scientific Research Fund (Grant Nr. OTKA 108947) and the János Bolyai Research Scholarship of the Hungarian Academy of Sciences.

References

[1] Dimitris Achlioptas, Carla Gomes, Henry Kautz, and Bart Selman. Generating satisfiable problem instances. In *Proceedings of the 17th National Conference on Artificial Intelligence (AAAI-00)*, pages 256–261, 2000.

[2] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In *21st International Joint Conference on Artificial Intelligence (IJCAI'09)*, pages 399–404, 2009.

[3] Adrian Balint, Anton Belov, Marijn J.H. Heule, and Matti Järvisalo. Generating the uniform random benchmarks for SAT Competition 2013. In Adrian Balint, Anton Belov, Marijn J.H. Heule, and Matti Järvisalo, editors, *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of *Department of Computer Science Series of Publications B*, pages 97–98. University of Helsinki, 2013.

[4] Armin Biere. Adaptive restart strategies for conflict driven SAT solvers. In Hans Kleine Büning and Xishun Zhao, editors, *Theory and Applications of Satisfiability Testing – SAT 2008*, volume 4996 of *Lecture Notes in Computer Science*, pages 28–33. Springer Berlin / Heidelberg, 2008.

[5] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, 1962.

[6] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

[7] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing*, volume 2919 of *Lecture Notes in Computer Science*, pages 333–336. Springer Berlin / Heidelberg, 2004.

[8] Carla Gomes, Bart Selman, and Henry Kautz. Boosting combinatorial search through randomization. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, pages 431–437, 1998.

[9] Carla P. Gomes, Henry Kautz, Ashish Sabharwal, and Bart Selman. Satisfiability solvers. In Frank van Harmelen, Vladimir Lifschitz, and Bruce Porter, editors, *Handbook of Knowledge Representation*, pages 89–134. Elsevier, 2008.

[10] Carla P. Gomes, Bart Selman, Nuno Crato, and Henry Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasoning*, 24(1-2):67–100, 2000.

[11] Shai Haim and Marijn Heule. Towards ultra rapid restarts. Technical report, UNSW and TU Delft, 2010.

[12] Marijn J. H. Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In *Hardware and Software: Verification and Testing. 7th International Haifa Verification Conference*, pages 50–65, 2012.

[13] Haixia Jia and Cristopher Moore. How much backtracking does it take to color random graphs? Rigorous results on heavy tails. In *Principles and Practice of Constraint Programming (CP 2004)*, pages 742–746, 2004.

[14] Henry Kautz, Eric Horvitz, Yongshao Ruan, Carla Gomes, and Bart Selman. Dynamic restart policies. In *Eighteenth National Conference on Artificial Intelligence*, pages 674–681, Menlo Park, CA, USA, 2002. American Association for Artificial Intelligence.

[15] Donald E. Knuth. Estimating the efficiency of backtrack programs. *Mathematics of Computation*, 29:121–139, 1975.

[16] Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of Las Vegas algorithms. *Information Processing Letters*, 47(4):173–180, 1993.

[17] Zoltán Ádám Mann. *Optimization in computer engineering – Theory and applications.* Scientific Research Publishing, Irvine CA, USA, ISBN: 978-1-935068-58-7, 2011.

[18] Zoltán Ádám Mann and Tamás Szép. BCAT: A framework for analyzing the complexity of algorithms. In *8th IEEE International Symposium on Intelligent Systems and Informatics*, pages 297–302, 2010.

[19] Zoltán Ádám Mann and Tamás Szép. A best-first-search approach to constraint satisfaction problems. In *Proceedings of the 7th Hungarian-Japanese Symposium on Discrete Mathematics and Its Applications*, pages 409–418, 2011.

[20] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient SAT solver. In *Proceedings of the 38th annual Design Automation Conference*, DAC '01, pages 530–535, New York, NY, USA, 2001. ACM.

[21] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, ISBN: 978-0136042594, 3rd edition, 2010.

[22] Joao P. Marques Silva and Karem A. Sakallah. Grasp – a new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, pages 220–227, 1997.

[23] Toby Walsh. Search in a small world. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence*, pages 1172–1177, 1999.