A Complete Random Jump Strategy with Guiding Paths

Hantao Zhang[⋆]

Department of Computer Science
University of Iowa Iowa City, IA 52242, U.S.A.
hzhang@cs.uiowa.edu

Abstract. The restart strategy can improve the effectiveness of SAT solvers for satisfiable problems. In 2002, we proposed the so-called random jump strategy, which outperformed the restart strategy in most experiments. One weakness shared by both the restart strategy and the random jump strategy is the ineffectiveness for unsatisfiable problems: A job which can be finished by a SAT solver in one day cannot not be finished in a couple of days if either strategy is used by the same SAT solver. In this paper, we propose a simple and effective technique which makes the random jump strategy as effective as the original SAT solvers. The technique works as follows: When we jump from the current position to another position, we remember the skipped search space in a simple data structure called "guiding path". If the current search runs out of search space before running out of the allotted time, the search can be recharged with one of the saved guiding paths and continues. Because the overhead of saving and loading guiding paths is very small, the SAT solvers is as effective as before for unsatisfiable problems when using the proposed technique.

1 Introduction

Modern SAT solvers based on the DPLL method can handle instances with hundreds of thousands of variables and several million clauses. To improve the chance of solving these problems, in [16], we proposed another technique called the "random jump strategy" which solves the same problem as the restart strategy [11]: When the procedure stuck at a region of the search space along a search tree, the procedure jumps to another region of the search space. One major advantage of the random jump strategy over the restart strategy is that there is no danger of visiting any region of the search space more than once (except those nodes appearing in a guiding path).

However, the strategy proposed in [16] may (with a small chance) destroy the completeness of a SAT solver because of skipped search space: A job which can be finished by a SAT solver in one day might not be finished in one or two days if either this strategy or the restart strategy is used by the same SAT solver. In this paper, we propose a simple technique which makes the random jump strategy

⋆ Supported in part by NSF under grant CCR-0098093.

effective for both satisfiable and unsatisfiable problems. The technique works as follows: When we jump from the current search position to another position, we remember the skipped search space as a "guiding path" [18]. If we have no more search space left in the current run before running out of the allotted time, we may load one of the saved guiding paths and start another run of the search. The procedure stops when either a model is found or when no time left or when no more guiding paths left. Because the overhead of saving and loading guiding paths is very small, the SAT solvers are as effective as before for unsatisfiable problems when using this version of the random jump strategy. While the proposed technique can be used for any combinatorial backtrack search procedures, to keep the presentation simple, in this paper we will limit the discussion of our idea on the DPLL method.

We wish to point out that our random jump strategy is different from the general random jump strategy proposed in [10] in that when we backtrack to a previously selected literal, we require that the selected literal be set to its opposite value. This will prevent the selected literal from being assigned the same value. The strategy proposed in [10] cannot be complete with this requirement; they require that the literal be unassigned. Since the literal may take the same value in the next step, some portion of the search will be repeated. Another difference is that the completeness of their strategy is based on keeping the lemmas learned from the conflict analysis. The completeness of our strategy does not depend on the use of lemmas.

2 Random Jump in the DPLL Method

To use the random jump strategy, we divide the allotted search time into, say eight, equal time intervals and set up a checkpoint at the end of each time slot. At a checkpoint, we look at (the first few nodes of) the path from the root to the current node to estimate the percentage of the remaining space. If the remaining space is sufficiently large, we may jump up along the path, skipping some open branches along the way. After the jump, we wish that the remaining space is still sufficiently large.

2.1 Search Space of the DPLL Method

The space explored by a backtrack search procedure can be represented by a tree, where an internal node represents a backtrack point, and a leaf node represents either a solution or no solution. The search space explored by the DPLL method is a binary tree where each internal node is a decision point and the two outgoing branches are marked, respectively, by the two literals of the same variable with opposite sign. The case-splitting rule creates an internal node with two children in the search space. Without loss of any generality, if L is the literal picked at an internal node for splitting, we assume that the link pointing to the left child is marked with $\langle L, 1\rangle$ (called *open link*) while the link pointing to the right child is labeled with $\langle L, 0\rangle$ (called *closed link*). A leaf node in the search space is marked with either an empty clause (a conflict) or an empty set of clauses (a model).

We can record the path from the root of the tree to a given node, called *guiding path* in [18], by listing the links on the path. Thus, a *guiding path* is a list of literals each of which is associated with a Boolean flag [18]. The Boolean flag tells if the chosen literal has been assigned true only (1) or both true and false (0). According to [9], the concept of "guiding path" was first introduced in [6]. However, the name of "guiding path" does not appear in [6] and the literals used in [6] do not have an associated Boolean flag. For the implementation of the DPLL method, the current guiding path is stored in a stack.

In addition to the input formula, a DPLL algorithm can also take a guiding path as input. The DPLL algorithm will use the guiding path for decision literals until all the literals in the guiding path are used. If a literal is taken from a closed link, the DPLL algorithm will treat it as having only one (right) child. We can use guiding paths to avoid repeated search so that the search effort can be accumulated. For instance, the guiding path $(\langle x_1, 1\rangle\ \langle \neg x_5, 0\rangle\ \langle x_3, 0\rangle)$ tells us that the literals are selected in the order of $x_1, \neg x_5, x_3$, and the subtree starting with the path $(\langle x_1, 1\rangle\ \langle \neg x_5, 1\rangle)$ is already complete; so is the subtree starting with $(\langle x_1, 1\rangle\ \langle \neg x_5, 0\rangle\ \langle x_3, 1\rangle)$.

For the standard DPLL method, the backtrack is done by looking bottom-up for the first open link in the guiding path. If an open link is found, we replace it by its corresponding closed link and continue the search from there. To implement the jump, we may skip a number of open links in the guiding path as long as the remaining search space is sufficiently large.

2.2 Random Jump in the DPLL Method

The random jump strategy is proposed in [16] with the following goals in mind.

- Like the restart strategy, the new strategy should allow the search to jump out of a "trap" when it appears to explore a part of the space far from a solution.
- The new strategy will never cause any repetition of the search (except the few nodes appearing in guiding paths) performed by the original search algorithm.
- The new strategy will not demand any change to the branching heuristic, so that any powerful heuristic can be used without modification.
- If a search method can exhaust the space without finding a solution, thus showing unsatisfiability, the same search method with the new strategy should be able to do the same using the same amount of time.

Suppose the root node is at *level* 0 and the two children of a level i node are at *level* $i + 1$. To facilitate the presentation in this section, we assume that each path of the search tree is longer than three. Under this assumption, there are eight internal nodes at level 3 and we number the subtrees rooted by these nodes as T_1, T_2, ..., T_8. To check if the current search position in one of these subtrees, we just need to check the first three Boolean flags in the current guiding path: If they are $(1, 1, 1)$, then it is in T_1; if they are $(1, 1, 0)$, then it is in T_2; ...; if they are $(0, 0, 0)$, then it is in T_8.

The main idea of our new strategy can be described as follows: Suppose a search program is allotted t hours to run on a problem. We divide t into eight time intervals. At the end of time interval i, where $1 \leq i < 8$, suppose the current search position is in T_j, we say the search is "on time" if $j \geq i$ and is "late" if $j < i$. If the search is on time, we continue the search; if the search is late, then we say the remaining space is *sufficiently large* [16]. If this is the case, we may skip some unexplored space to avoid some traps, as long as we do not skip the entire T_j. To skip some unexplored space, we simply remove some open links (bottom up) on the guiding path.

At any checkpoint during the execution of the DPLL method, we can use the current guiding path to check in which subtree the current position is. Suppose we are at the end of time interval j and the current search position is still in T_1. The first three links (they must be open as we are in T_1) may or may be removed, depending on the value of j; all the other open links can be removed. If $j \geq 4$, then the first link can be skipped; if $j \geq 2$, then the second link can be skipped; if $j = 1$, then the third link can be skipped.

Example 1. Suppose the current guiding path is

$$(\langle l_1, 1\rangle\langle l_2, 1\rangle\langle l_3, 1\rangle\langle l_4, 0\rangle\langle l_5, 1\rangle\langle l_6, 1\rangle\langle l_7, 0\rangle\langle l_8, 1\rangle),$$

and we are at the end of time interval 5. The total number of open links in this case is 6 and every one can be skipped. In our implementation, a random number will be picked from $\{4, 5, 6\}$. If the chosen number is 4, then four open links will be removed and $\langle l_3, 1\rangle$ is called the *cutoff* link which is the last open link to be removed. The guiding path after skipping will be $(\langle l_1, 1\rangle\langle l_2, 1\rangle\langle \overline{l_3}, 0\rangle)$, where $\overline{l}$ is the negation of l. The search will use this path to continue the search. Note that the value of l_3 will be set to 0 when using this path.

2.3 Remembering the Skipped Search Space

Of course, it is true that each tree T_i will take different amounts of time to finish. If T_1 needs 99% of total time and the rest trees need only 1%, then we may skip too much after time interval 1 and become idle once T_2, ..., and T_8 are exhausted. To avoid the risk of being idle, we may memorize what has been skipped and this information can be stored as a guiding path.

For the previous example, the cutoff link is $\langle l_3, 1\rangle$. To remember all the skipped open links, we just need to replace all the open links before the cutoff link, i.e., $\langle l_3, 1\rangle$, by the corresponding closed links. That is, we need to save the following guiding path:

$$(\langle l_1, 0\rangle\langle l_2, 0\rangle\langle l_3, 1\rangle\langle l_4, 0\rangle\langle l_5, 1\rangle\langle l_6, 1\rangle\langle l_7, 0\rangle\langle l_8, 1\rangle).$$

This example illustrates how to skip a portion of the search space and how to save the skipped search space. For a more general description of the random jump strategy with guiding path, and related work, please refer to [17].

In [2], the concept of *search signature* is proposed to avoid some repeated search between restarts. At first, the search signature, which is a set of lemmas, takes more memory to store than a guiding path. Secondly, lemmas can avoid the repetition of leaf nodes but cannot eliminate the repeated visitation of internal nodes at the beginning of a search path before these lemmas become unit clauses. Because of this reason, their strategy creates bigger overhead than our strategy.

3 Conclusion

We have presented an improvement to a randomization strategy which takes the allotted run time as a parameter and checks at certain points if the remaining search space is sufficiently large comparing to the remaining run time; if yes, some space will be skipped and the skipped space is recorded as a guiding path. Like the restart strategy, it can prevent a backtrack search procedure from getting trapped in the long tails of many hard combinatorial problems and help it to find a solution quicker. Unlike the restart strategy, it never revisits any search space decided by the original search procedure. Unlike the restart strategy, it does not lose the effectiveness when working unsatisfiable problems as the overhead of the strategy is very small and is ignorable.

The motivation behind this research is to solve open quasigroup problems [14]. Without the random jump strategy, given a week of run time, SATO could not solve any of the possible exceptions in the theorems in [5,4,13,21]. Four cases were reported satisfiable in [4] and two cases were found satisfiable in [5]. In [13], four previously open cases were found satisfiabl; they are: QG3(4^9), QG3(5^9), QG3($5^1 2$), and QG3(7^9). With the strategy, SATO solved each of them in less than a week. This clearly demonstrated the power of the new strategy. Moreover, since the random jump strategy keeps the completeness of SATO, we are able to prove that several previously unknown problems have no solutions, including QG5(18) [14].

References

1. Baptista, L., and Margues-Silva, J.P., Using Randomization and Learning to Solve Hard Real-World Instances of Satisfiability, in Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming (CP), September 2000.
2. Baptista, L., Lynce I., and Marques-Silva, J.P., Complete Search Restart Strategies for Satisfiability, in the IJCAI'01 Workshop on Stochastic Search Algorithms (IJCAI-SSA), August 2001.
3. Bayardo, R., Schrag, R., Using CSP look-back techniques to solve exceptionally hard SAT instances. In Proceedings of CP-96, 1996.
4. Bennett, F.E., Du, B., and Zhang, H.: Conjugate orthogonal diagonal latin squares with missing subsquares, Wal Wallis (ed.) "Designs 2002: Further Computational and Constructive Design Theory", Ch 2, Kluwer Academic Publishers, Boston, 2003.
5. Bennett, F.E., Zhang, H.: Latin squares with self-orthogonal conjugates, *Discrete Mathematics*, 284 (2004) 45–55

6. Bohm, M., Speckenmeyer, E.: A fast parallel SAT-solver – Efficient Workload Balancing, URL:`http://citeseer.ist.psu.edu/51782.html`, 1994.
7. Davis, M., Putnam, H. (1960) A computing procedure for quantification theory. *Journal of the ACM*, **7**, 201–215.
8. Davis, M., Logemann, G., and Loveland, D.: A machine program for theorem-proving. *Communications of the Association for Computing Machinery 5,* 7 (July 1962), 394–397.
9. Feldman, Y., Dershowitz, N., Hanna, Z.: Parallel Multithreaded Satisfiability Solver: Design and Implementation. Electronic Notes in Theoretical Computer Science 128 (2005) 75-90
10. Lynce, I., Baptista, L., and Marques-Silva, J. P., Stochastic Systematic Search Algorithms for Satisfiability, in the LICS Workshop on Theory and Applications of Satisfiability Testing (LICS-SAT), June 2001.
11. Gomes, C.P., Selman, B., and Crato, C.: Heavy-tailed Distributions in Combinatorial Search. In Principles and Practices of Constraint Programming, (CP-97) Lecture Notes in Computer Science 1330, pp 121-135, Linz, Austria., 1997. Springer-Verlag.
12. Marques-Silva, J. P., and Sakallah, K. A., GRASP: A Search Algorithm for Propositional Satisfiability, in IEEE Transactions on Computers, vol. 48, no. 5, pp. 506-521, May 1999.
13. Xu, Y., Zhang, H.: Frame self-orthogonal Mendelsohn triple systems, *Acta Mathematica Sinica*, Vol.20, No.5 (2004) 913–924
14. Zhang, H.: (1997) Specifying Latin squares in propositional logic, in R. Veroff (ed.): Automated Reasoning and Its Applications, Essays in honor of Larry Wos, Chapter 6, MIT Press.
15. Zhang, H.: (1997) SATO: An efficient propositional prover, Proc. of International Conference on Automated Deduction (CADE-97). pp. 308–312, Lecture Notes in Artificial Intelligence 1104, Springer-Verlag.
16. Zhang, H.: (2002) A random jump strategy for combinatorial search. Proc. of Sixth International Symposium on Artificial Intelligence and Mathematics, Fort Lauderdale, FL, 2002.
17. Zhang, H.: (2006) A complete random jump strategy with guiding paths (full version). `http://www.cs.uiowa.edu/~hzhang/crandomjump.pdf`
18. Zhang, H., Bonacina, M.P., Hsiang, H.: PSATO: a distributed propositional prover and its application to quasigroup problems. *Journal of Symbolic Computation* (1996) 21, 543–560.
19. Zhang, H., Bennett, F.E.: Existence of some $(3, 2, 1)$–HCOLS and $(3, 2, 1)$–HCOLS. *J. of Combinatoric Mathematics and Combinatoric Computing*, 22 (1996) 13-22.
20. Zhang, H., Stickel, M.: Implementing the Davis-Putnam method, *J. of Automated Reasoning* 24: 277-296, 2000.
21. Zhu, L., Zhang, H.: Completing the spectrum of r-orthogonal latin squares. *Discrete Mathematics* 258 (2003)