

Community Branching for Parallel Portfolio SAT Solvers

Tomohiro Sonobe^{1,2}, Shuya Kondoh³, and Mary Inaba³

¹ National Institute of Informatics, Japan

² JST, ERATO, Kawarabayashi Large Graph Project, Japan

³ Graduate School of Information Science and Technology, University of Tokyo,
Japan
tominlab@gmail.com

Abstract. Portfolio approach for parallel SAT solvers is known as the standard parallelisation technique. In portfolio, diversification is one of the important factors in order to enable workers (solvers) to conduct a vast search. The diversification is implemented by setting different parameters for each worker in the state-of-the-art parallel portfolio SAT solvers. However, it is difficult to combine the search parameters properly in order to avoid overlaps of search spaces between the workers. For this issue, we propose a novel diversification technique, called community branching. In this method, we assign a different set (or sets) of variables (called a community) to each worker and force them to select these variables as decision variables in early decision levels. In this manner, we can avoid the overlaps of the search spaces between the workers more vigorously than the existing method. We create a graph, where a vertex corresponds to a variable and an edge stands for a relation between two variables in a same clause, and we apply a modularity-based community detection algorithm to it. The variables in a community have strong relationships, and a distributed search for different communities can benefit the whole search. Experimental results show that we could speedup an existing parallel SAT solver with our proposal.

Keywords: Parallel SAT solver, Portfolio, Diversification, Decision, Community detection

1 Introduction

The state-of-the-art SAT solvers are based on the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [5]. The so-called Conflict-Driven Clause Learning (CDCL) SAT solvers consist of various techniques such as clause learning, non-chronological backtracking, Variable State Independent Decaying Sum (VSIDS) [14] decision heuristic, and restart upon DPLL algorithm.

On the other hand, state-of-the-art parallel SAT solvers are also based on CDCL solvers. Solvers based on the divide-and-conquer approach prevailed first, such as PSATO [17], MiraXT [12], and PMSat [7]. The divide-and-conquer solvers often have a difficulty for selecting splitting variables of the search space,

and that causes a difficult load-balancing. In recent years, the portfolio approach [9] [11] has become the mainstream of parallel SAT solvers. In the portfolio, workers (solvers) conduct a competitive and cooperative search without the splitting of the search space between other workers, and the answer (SAT or UNSAT) of the fastest solver is adopted. The advantages of the portfolio are that no load-balancing is needed and no synchronisation of the end of the search for unsatisfiable instances (because all the workers conduct the search for the whole search space). In order to fully utilise the advantages of the portfolio, diversification [10] is needed. The diversification is a differentiation of workers' search to avoid overlaps between the workers, and is achieved by differentiating search parameters of the workers, such as decision heuristics, learning schemes, and frequency of restart.

However, it is difficult to combine the search parameters properly in order to avoid overlaps of search spaces between the workers. In addition, there is no guaranty that different parameter sets can prevent two or more solvers from searching for the same spaces. This is because the existing diversification only focuses on differentiating "search activity" of the SAT solver. For this issue, we propose a new diversification technique, *community branching*. Community branching forces each worker in the portfolio to branch on a specific set (or sets) of the variables in early decision levels, and prompt the workers to conduct searching in the different search spaces. This method focuses on differentiating "search space" of the SAT solver as compared to the existing diversification. In order to achieve effective branchings, we used a community detection algorithm to make the sets of variables.

Community detection algorithms [4, 3, 16] gathered attention in analysing real-world networks, such as Internet [6] and social networks [8]. These algorithms are used in uncovering structures of graphs by gathering vertices into a community, where a high density of edges within the community and low density of the edges between the communities. The recent algorithms are designed to handle over one million vertices within a reasonable computational time, and they are based on modularity [4] optimisation. The modularity value stands for a relative value compared with random partition of the vertices. The higher the modularity value is, the more structured the graph is. As for SAT problem, [1] points out that Variable Incidence Graph (VIG) made from a real-world SAT instance can have a high modularity value. In VIG, the vertices correspond to the Boolean variables and the edges correspond to relations between two variables in a same clause. The variables in the community have close relationships (strongly constrained) to each other. Hence an intensive search for them in parallel can benefit the whole search. A similar attempt was evaluated for MAXSAT problems in [13].

In community branching, a graph is made from the given CNF and learnt clauses, and a community detection algorithm is performed to the graph. Then the detected communities are assigned to each worker. Each worker conducts intensive searches for the given communities, which means that each worker selects the variables in the communities as decision variables in the early phase of

the search. We implemented community branching to PeneLoPe [2] with Graph Folding Algorithm [3] as community detection algorithm and conducted experiments.

In Section 2, we detail community branching. We show the experimental results in Section 3 and conclude the paper in Section 4.

2 Community branching

In this section, we detail the procedure of community branching.

First, we create a graph in the VIG format from the given CNF. In the VIG, the vertices correspond to the Boolean variables and the edges correspond to the relations between the variables in the same clause. A clause “ c ” generates $(|c| \times |c - 1|/2)$ edges between every pair of the variables. The weight of each edge is $(1/(|c| \times |c - 1|/2))$, therefore, the sum of the weights of the edges added by each clause is always 1. There are various modularity-based community detection algorithms such as fast greedy algorithm [4], label propagation algorithm [15], and Graph Folding Algorithm (GFA) [3]. Note that a vertex is not included in two or more communities in these algorithms. As stated in [1], the SAT instances from the real-world applications can show high modularity values. In this paper, we used the GFA because it exhibited a good performance for execution time in our preliminary experiments. After the community detection, the communities are assigned to the workers. Each worker conducts intensive searches for the variables in the given community. In particular, the VSIDS scores of the variables are increased periodically. In this manner, each worker searches in the different search spaces and the diversification of search space can be achieved.

Figure 1 shows the pseudo code of community branching. The function “assign_communities” conducts the community detection for the VIG made from the given CNF and the learnt clauses. The reason why the learnt clauses are included is that this function can be called multiple times during the search. We should reconstruct the communities along with the transformation of the graph caused by the learnt clauses, which is also mentioned in [1]. The function “convert_CNF_to_VIG” converts the CNF including the learnt clauses into the VIG and “community_detection” returns detected communities. The variable “coms” is a set of the communities (a two-dimensional array). Then the communities in “coms” are sorted by each size in descending order. We should conduct the search for larger communities preferentially because they can be core parts of the given instance, and these cores should be distributed to each worker. Finally, each community is assigned to each worker (the variable “worker_num” stands for the number of the workers). This procedure is conducted only by a master thread (thread ID 0), and the learnt clauses only in the master thread are used. Note that the number of the communities depends on the community detection algorithm. If the number of the communities is greater than the number of the workers, two or more communities are assigned to one worker. In the reverse case, some workers have no communities (for example in the case of 8 workers

and 6 detected communities, 2 workers have no communities and they conduct the search without community branching).

```

assign_communities() {
    coms = community_detection(convert_CNF_to_VIG(given_CNF + learnt_clauses));
    sort_by_size_in_descending_order(coms);
    for(i = 0; i < coms.size(); i++) {
        worker_id = i % worker_num;
        assign(coms[i], worker_id);
    }
}

run_count = 0;
community_branching() {
    if (run_count++ % INTERVAL > 0)
        return;
    [choose one of the assigned communities in rotation]
    for each var in the chosen community
        bump_VSIDS_score(var, BUMP_RATIO);
}

```

Fig. 1. Pseudo code of community branching

After the assignment, each worker calls the function “community_branching” in Figure 1 for every restart. This function chooses a community from the assigned communities and increases the VSIDS scores of the variables in the chosen community. The variable “run_count” counts the number of executions of this function, and the main part of this function is executed for every “INTERVAL” restarts. We need this constant number because we should change the speed of the switching of the branching communities appropriately. In the main part of this function, the community is chosen from the assigned communities in rotation and the VSIDS scores of the variables in the chosen community are increased by proportional to “BUMP_RATIO”. In general, “BUMP_RATIO” is set to 1 at clause learning (the VSIDS scores are bumped at clause learning). In order to force the variables in the community to be selected as decision variables right after the restart, “BUMP_RATIO” should be a large value, such as 100. In the following section, we set “INTERVAL” and “BUMP_RATIO” as certain values by considering preliminary experiments. In addition, we set an interval for reconstruction of the communities, an interval for the function “assign_communities”. We call it “Community Reconstruction Interval (CRI)” in this paper. In particular, the communities are reconstructed for every “CRI” restarts.

3 Experimental results

In order to confirm the effectiveness of our proposal, we conducted experiments by using a state-of-the-art parallel portfolio SAT solver PeneLoPe [2]. We utilised the default settings of PeneLoPe, but we turned on the deterministic mode for reproducibility. We used 300 instances from the application category of the SAT Competition 2011. The experiments were conducted on a Linux machine with two Intel Xeon six-core CPUs, running at 3.33 GHz and 144 GB of RAM. Time-out was set to 5000 seconds (wall-clock time) for each instance. The number of running threads was set to eight.

From our preliminary experiments, we found that good performances could be achieved when “INTERVAL” = 1 and “BUMP_RATIO” = 100 for PeneLoPe. On the basis of this result, we compared three types of parameter settings: no community branching (no_cb), community branching with CRI = 500, INTERVAL = 1, and BUMP_RATIO = 100 (cri500_int1_bump100), community branching with CRI = 3000, INTERVAL = 1, and BUMP_RATIO = 100 (cri3000_int1_bump100).

Figure 2 and Figure 3 show the cactus plots of the results. Figure 2 shows 105 satisfiable instances and Figure 3 shows 126 unsatisfiable instances. Instances that were not solved by the three solvers were not included. Table 1 details the results. From these results, we can see that community branching improved the performance of PeneLoPe, especially for the unsatisfiable instances. Eventually, community branching with CRI = 3000, INTERVAL = 1, and BUMP_RATIO = 100 could solve eleven more instances than the base solver within a shorter runtime. However, we could not find suitable CRIs in this experiment. Although longer CRI (cri3000) exhibited a better performance, we should reconstruct the communities appropriately, as stated in [1].

Figure 4 and Figure 5 show the scatter plots of the relation between the number of the detected communities (made only from the given CNF) and speedup (seconds) achieved by community branching (the runtime difference no_cb - cri3000_int1_bump1, instances that were not solved within the time limit are calculated as 5000) for each instance. Figure 4 shows 105 satisfiable instances and Figure 4 shows 126 unsatisfiable instances. There were only three satisfiable instances and four unsatisfiable instances where the number of the detected communities was less than the number of the workers (eight). From these results, we can find that instances in which the number of the detected communities was less than around 100 could make impacts on the performances, even though a few instances were exceptional in the satisfiable instances. For instances where the number of the detected communities is large compared to the number of the workers (e.g. 100 communities and 8 workers), each worker is forced to conduct the search for various communities. It may cause an excessively diversified search, and the performance can be deteriorated. As future work, we are going to merge the detected communities, for example adjacent communities, into one community, with consideration for the number of the workers. We are also planning to change the parameters of community branching dynamically according to the number of the detected communities and their size.

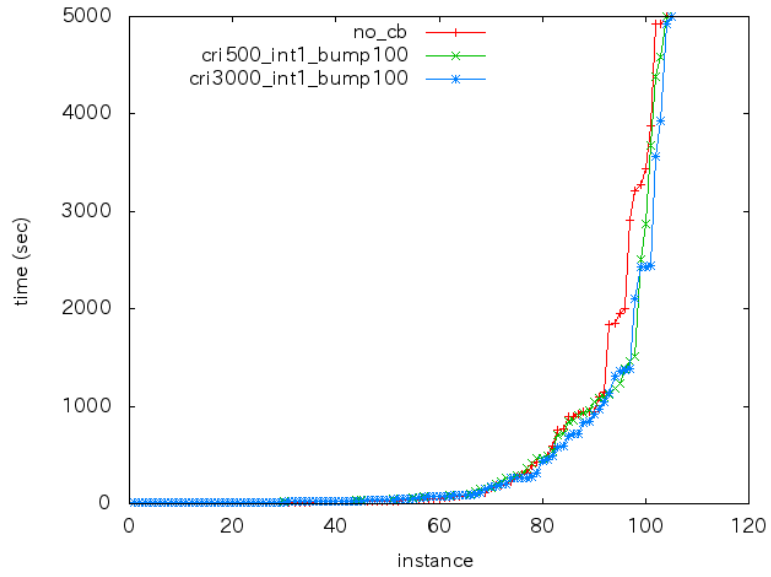


Fig. 2. The experimental results of 105 satisfiable instances.

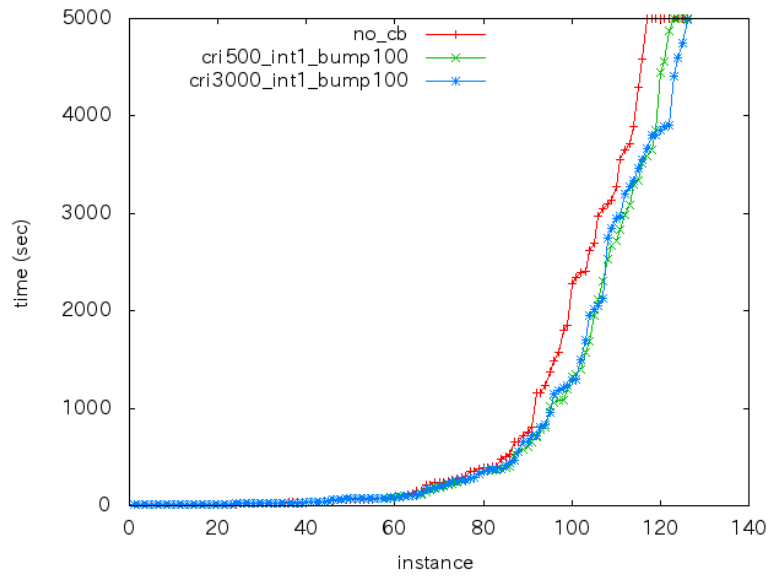


Fig. 3. The experimental results of 126 unsatisfiable instances.

Table 1. The details of the results: 231 (SAT: 105, UNSAT: 126) instances could be solved at least one solver. The instances that could not be solved within 5000 seconds are calculated as 5000.

	SAT (105)	UNSAT (126)	total	total time for 231 instances
no_cb	103	116	219	385220
cri500_int1_bump1	103	122	225	313450
cri3000_int1_bump1	104	126	230	310090

4 Conclusion

In this paper, we proposed community branching for making more diversification in the parallel portfolio SAT solvers. Compared with the existing diversification technique that focuses on differentiating the search activity, community branching can achieve diversification of the search spaces. Experimental results indicate that our proposal can improve the performance of the base solver. In addition, we expect that community branching is effective for higher degree of parallelisation, more than eight workers. As future work, we have to tune the parameters of community branching. We are also planning to change them dynamically according to the given instance.

5 Acknowledgment

We appreciate the insightful comments from the reviewers.

References

1. C. Ansótegui, J. Giráldez-Cru, and J. Levy. The Community Structure of SAT Formulas. In *Proceedings of the 15th international conference on Theory and Applications of Satisfiability Testing*, SAT’12, pp. 410–423, 2012.
2. A. Balint, A. Belov, M. J. H. Heule, and M. J. Jarvisalo. Proceedings of SAT Competition 2013; Solver and Benchmark Descriptions. 2013.
3. V. D. Blondel, J. Guillaume, R. Lambiotte, and E. Lefebvre. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, Vol. 2008, No. 10, p. 10008, 2008.
4. A. Clauset, M. E. J. Newman, and C. Moore. Finding community structure in very large networks. *Physical Review E*, Vol. 70, No. 6, p. 066111, 2004.
5. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, Vol. 5, No. 7, pp. 394–397, 1962.
6. K. A. Eriksen, I. Simonsen, S. Maslov, and K. Sneppen. Modularity and Extreme Edges of the Internet. *Physical Review*, Vol. 90, p. 148701, 2003.
7. L. Gil, P. F. Flores, and L. M. Silveira. PMSat: a parallel version of MiniSAT. *JSAT*, Vol. 6, No. 1-3, pp. 71–98, 2009.
8. M. Girvan and M. E. J. Newman. Community Structure in Social and Biological Networks. *Proceedings of the National Academy of Sciences*, Vol. 99, No. 12, pp. 7821–7826, 2002.

9. C. P. Gomes and B. Selman. Algorithm Portfolio Design: Theory vs. Practice. In *Proceedings of the 13th conference on Uncertainty in artificial intelligence*, UAI'97, pp. 190–197, 1997.
10. L. Guo, Y. Hamadi, S. Jabbour, and L. Sais. Diversification and Intensification in Parallel SAT Solving. In *Proceedings of the 16th international conference on Principles and practice of constraint programming*, CP'10, pp. 252–265, 2010.
11. Y. Hamadi, S. Jabbour, and L. Sais. ManySAT: a Parallel SAT Solver. *JSAT*, Vol. 6, No. 4, pp. 245–262, 2009.
12. M. Lewis, T. Schubert, and B. Becker. Multithreaded SAT Solving. In *Proceedings of the 2007 Asia and South Pacific Design Automation Conference*, ASP-DAC '07, pp. 926–931, 2007.
13. R. Martins, V. Manquinho, and I. Lynce. Community-Based Partitioning for MaxSAT Solving. In *Proceedings of the 16th international conference on Theory and Applications of Satisfiability Testing*, SAT'13, pp. 182–191, 2013.
14. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th annual Design Automation Conference*, DAC'01, pp. 530–535, 2001.
15. U. N. Raghavan, R. Albert, and S. Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical Review E*, Vol. 76, No. 3, p. 036106, 2007.
16. M. Rosvall and C. T. Bergstrom. Maps of random walks on complex networks reveal community structure. *Proceedings of the National Academy of Sciences*, Vol. 105, No. 4, pp. 1118–1123, 2008.
17. H. Zhang, M. P. Bonacina, and J. Hsiang. PSATO: a Distributed Propositional Prover and Its Application to Quasigroup Problems. *J. Symb. Comput.*, Vol. 21, pp. 543–560, 1996.

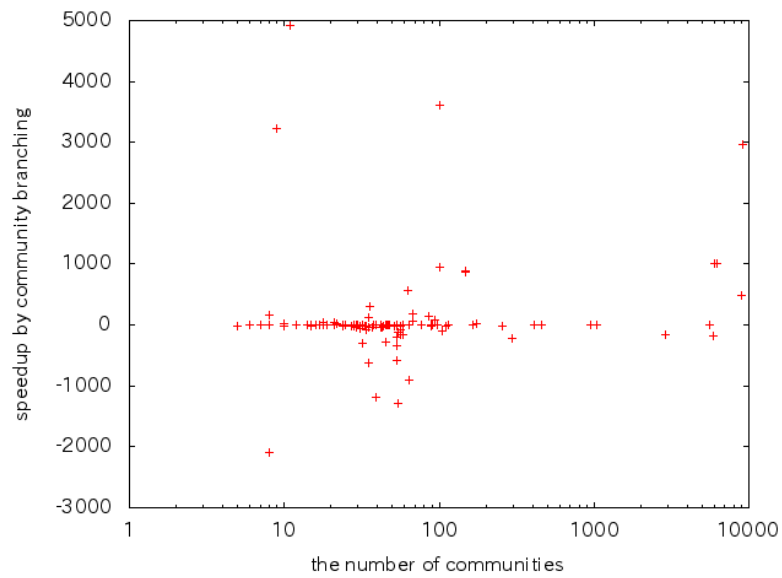


Fig. 4. The relation between the number of communities and speedup (seconds) by cri3000_int1.bump1 for 105 satisfiable instances.

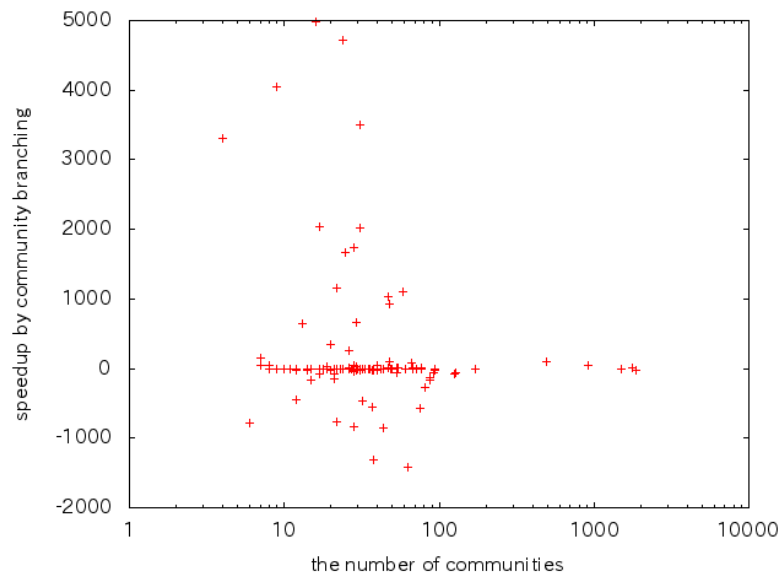


Fig. 5. The relation between the number of communities and speedup (seconds) by cri3000_int1.bump1 for 126 unsatisfiable instances.