Karlsruhe Institute of Technology

SAT Solving with distributed local search

Master Thesis of

Guangping Li

At the Department of Informatics
Institute of Theoretical informatics, Algorithmics II

Advisors: Dr. Tomáš Balyo
Prof. Dr. Peter Sanders

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen, als die angegebenen Quellen und Hilfsmittel benutzt, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe.

Karlsruhe, 26th Oktober 2018

Abstract

Stochastic local search (SLS) is an elementary technique for solving combinational problems. In the first section of this paper, we introduce an efficient SLS heuristic solver for Boolean Satisfiability Problem (SAT), in which the decisions only based on the probability distribution. We experimentally evaluate and analyze the performance of our solver in a combination of different techniques, including simulated annealing and walkSAT. With formula partitioning, we introduce a parallel version of our solver in the second section. The parallelism improves the efficiency of the solver. Using different random generator in solving the sub-formula can bring further improvement in performance to our parallel solver.

Zusammenfassung

Stochastische lokale Suche (SLS) stellt eine elementare Technik zur Lösung von komplizierten kombinatorischen Problemen dar. Im ersten Teil dieser Arbeit stellen wir eine effiziente SLS Heuristik für das Erfüllbarkeitsproblem der Aussagenlogik (SAT) vor, bei dem die Entscheidungen nur auf der Wahrscheinlichkeitsverteilung basieren. Die Leistung unseres Algorithmus in einer Kombination verschiedener Techniken, einschließlich simulierter Abkühlung und walkSAT, wurde auch experimentell bewertet und analysiert. Mit Formelpartition wird im zweiten Teil eine parallele Version unseres Algorithmus eingeführt, die die Effizienz des Solvers verbessert. Flexible Parametereinstellungen kann eine weitere Verbesserung unseres Algorithmus bringen.

Contents

1 Introduction **1**
- 1.1 Problem/Motivation . . . 1
- 1.2 Content . . . 1
- 1.3 Definitions and Notations . . . 1
- 1.4 The Competitors . . . 6

2 Our local Solver **7**
- 2.1 initAssign(F) . . . 7
- 2.2 pickCla(A) . . . 7
- 2.3 pickVar(A,c) . . . 8
- 2.4 pickVar(A,c) with simulated annealing . . . 9
- 2.5 Data structures . . . 10
- 2.6 Our swpSAT solver . . . 10

3 Our Parallel Algorithm **11**
- 3.1 1st Approach: The pure portfolio approach . . . 11
- 3.2 2nd Approach: Solver with formula partitioning . . . 11
- 3.3 3nd Approach: Solver with combination of sub-assignments . . . 12
- 3.4 4th Approach: Initialization with a guide of formula partitioning . . . 13

4 Evaluation **14**
- 4.1 DIMACS standard format . . . 14
- 4.2 Benchmarks . . . 14
- 4.3 Used plots and tables . . . 15
- 4.4 Random Seeds used in Experiments . . . 15
- 4.5 Soft- and Hardware . . . 15
- 4.6 Parameter Settings in Experiment . . . 16
- 4.7 COMBINE Benchmark Generation . . . 16
- 4.8 Experiments . . . 19
  - 4.8.1 Experiment 1: initAssign(F) . . . 19
  - 4.8.2 Experiment 2: pickVar(F) . . . 21
  - 4.8.3 Experiment 3: Simulated Annealing . . . 23
  - 4.8.4 Experiment 4: 2017-UNIF Comparision (local) . . . 27
  - 4.8.5 Experiment 5: The pure portfolio approach . . . 29
  - 4.8.6 Experiment 6: Initialization with a guide of formula partitioning . . . 31
  - 4.8.7 Experiment 7:2017-UNIF Comparison (parallel) . . . 33

5 Conclusion **35**
- 5.1 Further work . . . 35

6 Bibliography **36**

1 Introduction

1.1 Problem/Motivation

The ***propositional satisfiability problem*** (***SAT***) is the first proven NP-complete problem [1]. The problem is to determine whether an assignment of boolean values to variables in a boolean formula exsits such that the expression evaluates to true. Hard combinational problems can be resolved with appropriate encoding as a SAT problem. The SAT problem has many applications in computer science like chip model checking [2], software verification [3] or in automated planning and scheduling in artificial intelligence [4].
Formula partitioning is one of the promising approaches in DPLL-like solvers [5]. By prioritizing the variables according to a good formula partitioning, the search gets a relatively balanced decision tree. But formula partitioning is rarely used in a local search for the SAT problem. Questions such as how to combine the formula partitioning with local search, whether the local search can benefit from the partition, and whether the formula partitioning can guide a parallel local search still remain open.

1.2 Content

The SAT problem, as a well-known NP-complete problem, has drawn remarkable attention and different local search heuristics have been developed to tackle this problem. In this thesis, we introduced a stochastic local search on SAT problem using formula partitioning as guidance.
In section 1, we summarized the formal concept and techniques used in this thesis. Section 2 described our method which is a combination of *probSAT* and *walkSAT*. Then we discussed some attempts at improving the algorithm. By experimental evaluation and comparison, some techniques turned out to be more efficient than the simple *probSAT* search. In section 3, we inspected the potential benefit of formula partitioning in a parallel search. Section 4 described the experiments mentioned in section 2 and section 3 with details and empiric results. Section 5 concluded our work based on the experiments, alongside a discussion of some limitations of our solver and further work.

1.3 Definitions and Notations

Propositional Satisfiability Problem
A variable with only two possible logical values *true* or *false* is a ***propositional variable***, which will be referred to as ***variable*** in this thesis.
A ***literal*** is an atomic formula in propositional logic. A literal can either be a ***positive literal*** v as the variable v or a ***negative literal*** $\bar{v}$ as negation of v.
A ***clause*** is a disjunction of literals. A formula in conjunctive normal form (CNF) is a conjunction of clauses. We refer it as ***CNF-formula*** or simply as ***formula*** in this thesis.
An ***assignment*** a: $V \rightarrow \{true, false\}$ assigns a truth value to each variable v in the formula. An assignment satisfies a formula if the truth value of the formula with this assignment evaluates to true. Specifically, an assignment satisfies a clause, if at least one literal in the clause is assigned with value *true*.
An assignment a is a satisfying assignment if it satisfies all clauses in the formula. Otherwise, there are conflicts in some clauses with this assignment, or some clauses are unsatisfied clauses with this assignment. A ***satisfiable formula*** is a formula which can be satisfied by some

assignment. The SAT problem is to determine whether a given formula is satisfiable or not.

Here is an example of SAT problem:

$$
\begin{aligned}
&F = (v_1 \vee \bar{v_3}) \wedge (v_2 \vee v_1 \vee \bar{v_1}) \\
&Vars(F) = \{v_1, v_2, v_3\} \\
&numV(F) = |Vars(F)| = 3 \\
&Lits(F) = \{v_1, \bar{v_1}, v_2, v_3, \bar{v_3}\} \\
&Cls(F) = \{C_1, C_2\} \\
&numC(F) = |Cls| = 2 \\
&C_1 = \{v_1, \bar{v_3}\} \\
&C_2 = \{v_2, v_3, \bar{v_1}\}
\end{aligned}
$$

$A(v_1) = true$, $A(v_2) = false$, $A(v_3) = true$,
A is an assignment satisfying F.

$\hat{A}(v_1) = true$, $\hat{A}(v_2) = false$, $\hat{A}(v_3) = false$,
$\hat{A}$ is an assignment with conflict in C_2.

Set
A set is a container of unique elements. A set of three objects a, b, c is written as $\{a, b, c\}$. The size of a set is the number of elements in the set.

Local Search
For an instance I of a hard combinational problem P, there is a set of solutions $S(I)$. An object function (score or cost) Γ is derived from the constraints of the problem to evaluate the candidate solutions. The goal of the local search is to find the solution with minimum cost (or the solution with maximal score).
A local search starts with an initial complete solution. According to some heuristic, the local search makes local changes to its current solution iteratively, hence the name ***local search***. Starting from an initial solution, the search will evaluate the neighbor solutions which are reached by applying a local change to the current solution and choose one of the neighbor solutions with local optimization. The search applies local moves until the optimal solution is reached, or in some cases, a generally good solution is reached. Local search is widely used in hard combinational problems such as the traveling salesman problem [6] and the graph coloring problem [7].

Local Search in SAT Problem
In the boolean satisfiability problem, a local search operates primarily as follows: The search starts from a randomly generated assignment as the initial solution. If this current assignment satisfies the formula, the search stops with success. Otherwise, a variable is chosen depending on some criterion. This selection is called ***pickVar***. By changing the assignment of the selected variable v, a neighbor assignment $\hat{A}$ of our current solution A is reached in next step. The move is also called ***flip***(A, v). A local search will move in the space of the assignments by flipping the variables until a satisfying assignment is reached by the search.
The heuristic used for *pickVar* is based on scores of the variables in the current assignment. Consider the assignment $\hat{A}$ reached by taking a flip of the variable in the current assignment

A. The number of clauses satisfied in A, but not in $\hat{A}$ is called the ***breakcount*** of the local move from A to $\hat{A}$. Accordingly, the number of clauses which become satisfied because of the flipping, is the ***makecount***. The number of newly satisfied clauses (*makecount*) minus the number of newly unsatisfied clauses (*breakcount*), which is denoted as ***diffscore***, represents the local improvement of the corresponding flipping. Apart from this, other aspects such as the repetition number of each flip or the number of occurrences of the variables can be included in a selection heuristic. An example is the local solver *EagleUp*, which prefers flipping of variables with the highest number of occurrences in the formula to create new unit clauses for unit propagation. To get local improvement effectively, it is sufficient to only consider variables in unsatisfied clauses for the flipping selection. This commonly used process is called a ***focused local search***.

Algorithm 1: Focused Local Search

```
input    : A CNF Formula F
parameter: Timeout
output   : a satisfying assignment A
1 A ← random generated assignment A
2 while (∃ unsatisfied clause ∧ Timeout does not occur) do
3     c ← random selected unsatisfied clause
4     x ← pickVar(A, c)
5     A ← flip(A, x)
```

By choosing the variable with best score in *pickVar*, the search will get greedy local improvement. The initial intention of the local search is that the optimal global solution can be found through iterative greedy local improvement. The typical problem of the local search is that the greedy moves can be trapped in local optimal solutions, which are however not global optimums. To avoid this, some random flips are picked or even a worse solution will be chosen for the next step (***uphill moves***). The following techniques are commonly used to prevent local search from getting stuck in local regions.

Stochastic Local Search (SLS)

The stochastic local search will use the probability distribution of the scores of candidate solutions instead of the static decision. For the candidate moves, the probability of being chosen $p(\Gamma(s))$ corresponds to the score $\Gamma(s)$ of the solution s. In this way, the more advantageous a move is, the higher is the probability of choosing that move as the next step. This randomization can help the search get rid of cycling and decrease the chance of getting misled by specific heuristics.

Statistical Local Search

Tabu search was proposed by Fred W. Glover in 1986 and formalized in 1989 [8]. To recognize the loops in suboptimal region, the recent search moves are marked as tabu moves. These tabu moves will not be touched in the further search to discourage getting stuck in a region. Inspired by the tabu search, a statistical search will record the whole search trace, in which the number of times each variable is chosen for flipping are counted. Through the use of the statistical information, the search will prefer the variables with fewer flippings before. We have experimentally verified that a statistical search can recognize short-term cyclings like tabu search and permit long-term cyclings [9].

Simulated Annealing
Simulated Annealing is an approach of local search solver to difficult combinational optimization problems proposed by Kirkpatrick, Gelatt, and Vecchi [10]. This approach is inspired by the metallic process annealing, in which material is shaped by heating and then slowly cooling. This approach works as a local optimization algorithm guided by a controlling parameter ***temperature***. Higher temperature allows uphill moves with higher probability. The temperature varies according to the score of the current situation. For a current solution with a nearly optimal score, the temperature is nearly zero. For an unattractive local extreme with a poor score, the active search tends to make uphill moves due to high temperature.

walkSAT
The focused random local search strategy *walkSAT* to solve SAT problem, which is originally introduced in 1994 [11]. The *walkSAT* may ignore the greedy flipping and flip a random variable in a chosen unsatisfied clause with probability p . By introducing these "uphill noises", the walkSAT combines greedy local search and random walk to get an effective and robust random solver.

Algorithm 2: PickVar in walkSAT

```
input    : current assignment A, unsatisfied clause c
parameter: probability p
output   : a variable x in c to be flipped
1 for v in c do
2 |  Evaluate v with function Γ(A, v)
3 with probability p: x ← v with maximum Γ(A, v)
4 with probability 1 − p: x ← randomly selected v in c.
```

The probSAT
The SLS solver *probSAT* was introduced in 2012 by Adrian Balint and Uwe Schoening [12]. In a *probSAT* solver, the score of a candidate flip is solely based on the make- and breakcount. The paradigm is as follows: Firstly, a completely random assignment is set as the initial assignment. Then the algorithm performs local moves by flipping a variable in a random chosen unsatisfied clause and stops as soon as there exists no unsatisfied clause, which means a satisfying assignment is found. The probability $p(v)$ of flipping the variable v in the chosen clause is proportional to the score of v, which is calculated by the function $\Gamma(v, A)$ based on breakcount of v in the current assignment A.[1]

There are two kinds of score functions in the paper by Adrian Balint:

$$\Gamma(v, A) = (c_b)^{-break(v,A)} \text{ (break-only-exp-function)}$$
$$\Gamma(v, A) = (\epsilon + break(v, A)^{-c_b}) \text{ (break-only-poly-function)}$$

[1]As mentioned in the *probSAT* paper, the influence of makecount is rather weak in selection functions in experiments.

The pseudocode of *probSAT* is shown below:

Algorithm 3: PickVar in *probSAT*

```
input  : current assignment A, unsatisfied clause c
output : a variable x in c to be flipped
1 for v in c do
2   Evaluate v with function Γ(A, v)
3 x ← randomly selected variable v in c with probability p(v) = Γ(A,v) / Σ_{u∈c} Γ(A,u)
```

Formula partitioning

To introduce our parallel SAT solver with formula partitioning, we use a hypergraph representation of the SAT problem.
A hypergraph $G = (V, H)$ is a generalized graph, in which a hyperedge $h \in H$ is a non-empty subset of the vertices set V. For a SAT formula F, its hypergraph representation $G(F) = (Vars(F), Cls(F))$ consists of $numV$ vertices and $numC$ hyperedges. Each vertex corresponds to a variable in F, and a hyperedge refers to a clause, which connects the variables in this clause.

Here is an example:

$$F = \underbrace{(v_1 \vee v_2 \vee \bar{v_3})}_{C_1} \wedge \underbrace{(v_1 \vee v_3 \vee \bar{v_4})}_{C_2} \wedge \underbrace{(v_5 \vee v_6 \vee \bar{v_8})}_{C_3} \wedge \underbrace{(v_6 \vee v_7 \vee \bar{v_8})}_{C_4} \wedge \underbrace{(v_3 \vee \bar{v_6})}_{C_5} \wedge \underbrace{(v_4 \vee v_5)}_{C_6}$$

$G(F)$:

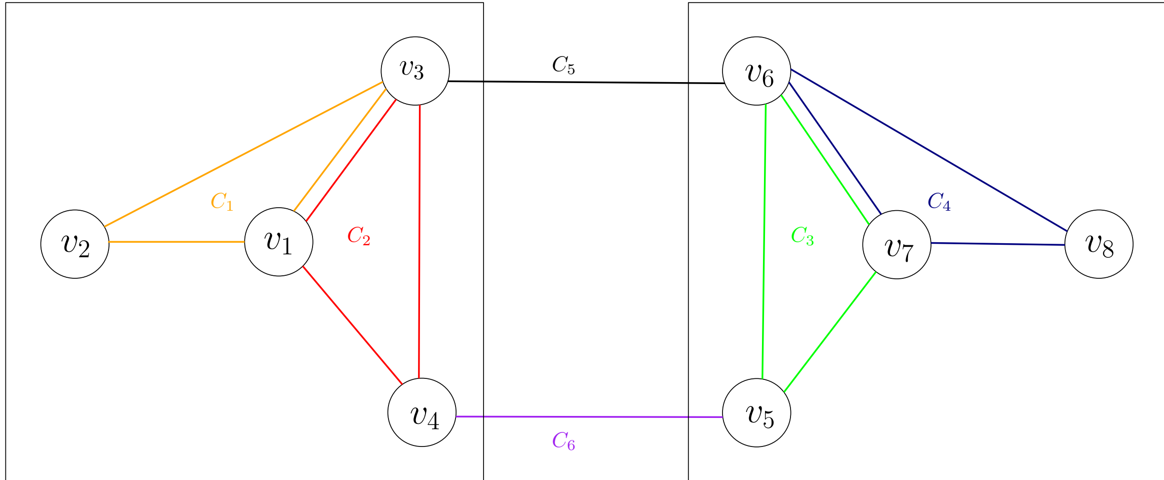


Figure 1: In a hypergraph, a hyperedge is a set of vertices. In this example, the vertices v_3 and v_4 are in both hyperedge $\{v_1, v_3, v_4\}$ and $\{v_1, v_2, v_3\}$, so they are connected twice. In our hypergraph representation, a hyperedge contains the vertices of the corresponding clauses. In another variant, each literal refers to one vertex and a hyperedge contains all the literals of the corresponding clause.

Formula partitioning is a promising way to improve the SAT problem solving. Two partitioning have already been well investigated. One is to divide the variables, which is used in this thesis, and another one is to separate the clauses. For algorithms with the technique of decision tree like *DPLL* [13], the formula partitioning can be used to prioritize decisions. For the local search, there is scarce research using formula partitioning in local search. Based on hypergraph represenation, we describe the formula partitioning with the notations of graph partition in this thesis. For a hypergraph $G = (V, H)$, a two-way partitioning is to seperate the vertices in two disjoint subsets P_0 and P_1. Based on the partitioning of vertices, the hyperedges are seperated

in three disjoint subsets H_0, H_1 and I. H_0 contains the edeges connecting verticse in P_0, H_1 are edges in P_1. The **intersection** I are the edges containing vertices in P_0 and vertices in P_1. In a good balanced partition, P_0 and P_1 are relatively of the same size and only few edges are in the intersection. In our example formula above, a minimum cost balanced partition is $P_0 = \{v_1, v_2, v_3, v_4\}$ and $P_1 = \{v_5, v_6, v_7, v_8\}$. In this partition, $H_0 = \{C_1, C_2\}$, $H_1 = \{C_4, C_5\}$ and the intersection $I = \{C_5, C_6\}$

1.4 The Competitors

Our heuristic is based on the *probSAT* paradigm. To evaluate the performance of our algorithm, we compare our heustic with the original *probSAT*. Another competitor is *yalSAT*, which is the champion in random track category of the SAT competition 2017 [14].

probSAT

The authors of the original paper implemented the *probSAT*.[2] We compare our solver with the original implemen.[3]

Two implementation variants are available. In the incremental approach, the breakcounts of variables are calculated in the initialization phase and only updated in the further search. The other straightforward approach is to compute scores of the variables in chosen clauses on the fly. This method is called non-incremental approach in original paper. To get optimal results of the *probSAT* solver, we take the non-incremental approach to the 3SAT problems and incremental method for 5SAT and 7SAT.

The parameters of *probSAT* in our experiments have been set as suggested in the original paper:

kSAT	score Γ	c_b	ϵ	variants
3SAT	break-only-poly	2.06	0.9	non-incrementel
5SAT	break-only-exp	3.7	-	incremental
7SAT	break-only-exp	5.4	-	incremental

Table 1: Parameter settings for competitor *probSAT*

yalSAT

We use the third version of yalSAT submitted to the 2017 SAT competition in our experiments. Armin Biere implements it as a reimplementation of *probSAT* with extensions.[4] The yalSAT uses a variant of *probSAT* randomly in the restart of a searchround. In our comparison, we use the default settings of the yalSAT with specific seeds (see 4.4).

[2]https://github.com/adrianopolus/probSAT

[3]Using same parameter settings our implementation gets similar performance to the original code

[4]https://baldur.iti.kit.edu/sat-competition-2017/solvers/random/

2 Our local Solver

Our algorithm is a typical focused SLS algorithm, which solves the SAT problem with the basic schema:

Algorithm 4: Our Local Search

```
input    : A CNF Formula F
parameter: Timeout
output   : a satisfying assignment A
1 A ← initAssign(F)
2 while (∃ unsatisfied clause ∧ Timeout does not occur) do
3   | c ← pickCla(A)
4   | x ← pickVar(A, c)
5   | A ← flip(A, x)
```

In the following, we will describe the steps used in our local search.

2.1 initAssign(F)

In our algorithm, there are three variants to initialize assignment. *RandomInit* is the random initiation also used in the original *probSAT*. The other two alternatives are *BiasInit* and *Bias-RandomInit*, which take the number of literal occurrences into consideration. With the method *BiasInit* we assign *true* to a variable if the number of occurrences of its positive literal is larger than that of its negative literal. Otherwise, a variable is initialized with *false*. *Bias-RandomInit* combines the two methods above by generating the assignment bias randomly based on the occurrences of literals. In experiment 1 (see 4.8.1) we compare these three alternatives based on the *probSAT* algorithm. Our local search uses *RandomInit* for 3SAT problems and *BiasInit* for the other problems.

2.2 pickCla(A)

The number of *true* values in each clause c, $numT(c)$, are counted in Initilization phase and maintained in further search. During the local flipping, these numbers will be updated for clauses containing the flipping variable (see 2.5). Unsatisfied clauses will be stored in a set *UNSAT*. Compared to *numT*, *UNSAT* is updated "lazily". After flipping, if *numT* of one clause is decreased to zero, the clause will be added to *UNSAT*. To select an unsatisfied clause in *pickCla*(A), one needs to select a clause from *UNSAT* and check if it is still unsatisfied with its *numT* being zero. Otherwise, if the chosen clause *c* with $numT$(c) as zero, it will be removed from *UNSAT* set. This step *pickCla*(A) will be repeated until one unsatisfied clause is found.

2.3 pickVar(A,c)

Inspired by *probSAT* and *walkSAT*, our *pickVar* combines the random walk and stochastic selection. The selection procedure in *probSAT* is a random heuristic. Judging from the experiments, even if the search is very close to a satisfying assignment, and the probability of the critical flipping is exceptionally high, it is still possible that the stochastic search make uphill moves and leave the region of the global minimum. To prevent this, we pick greedy flips with zero breakcounts with a certain probability p. With probability $(1-p)$, we choose the variable to be flipped using the *probSAT* heuristic.

Algorithm 5: Our pickVar

```
input     : current assignment A, unsatisfied clause c
parameter: probability p
output    : a variable x in c to be flipped
1 greedyVs ← ∅
2 for all v in c do
3   if (break(A,v)= 0 ) then
4     greedyVs = greedyVs + {v}
5 with probability p: x ← randomly selected variable v ∈ greedyVs
6 with probability (1 − p): x ← randomly selected variable v in c with probability
  Γ(A,v) / Σ_{u∈c} Γ(A,u)
```

We analyze the following variants for *pickVar* experimentally (see experiment in 4.8.2).

Variant 1: Walk

Instead of using a constant probability p to choose between a greedy literal without clause break and the random literal using *probSAT* selection directly, we use a statistic list S to record how many times each variable is chosen for flipping. To avoid cycling, the variable v_i with a high value of $S[i]$ is used for flipping with low probability. After selecting a greedy literal and a variable using the *probSAT* stochastic distribution, we make the choice randomly according to the statistic values of these two variables.

Algorithm 6: Walk

```
input     : current assignment A, unsatisfied clause c
parameter: probability p
output    : a variable x in c for flipping
1 greedyVs ← ∅
2 for all v in c do
3   if (break(A,v)= 0) then
4     greedyVs = greedVs + {v}
5 greedyV ← randomly selected variable v ∈ greedyVs
6 randomV ← randomly selected variable v in c with probability Γ(A,v) / Σ_{u∈c} Γ(A,u)
7 with probability p = α × s(greedyV) / (s(greedyV)+s(randomV)): x ← randomV
8 with probability 1 − p: x ← greedyV
```

Varinat 2: GreedyBreak

Getting a random literal using stochastic process consumes the most runtime in the search. In this variant *GreedyBreak*, we search for greedy literals with small statistic values. A literal is treated as a *permitted greedy literal* if its breakcount is zero and its statistic value is under a certain threshold. If one permitted greedy literal exists, we choose it for flipping. If multiple permitted greedy literals exist, we choose one randomly for flipping. Otherweise, we pick a literal using *probSAT* heuristic.
To set the threshold based on the search history, we compare two functions in our experiment. In the first approach *Average*, the threshold is set to $\alpha \times \frac{numF}{numV}$. Here the *numF* represents the number of flips. In another approach *Random-Flip*, we select a value r randomly in [0, *numF*]. For each greedy literal, we check if its statistic value is smaller than $\alpha \times r$.

Algorithm 7: GreedyBreak

```
input    : current assignment A, unsatisfied clause c
parameter: probability p
output   : a variable x in c for flipping
1 greedyVs ← ∅
2 for all v in c do
3     if (break(A,v)= 0 ∧ Permit(v)) then
4         greedyVs = greedyVs + {v}
5 if (greedyVs is not empty) then
6     x ← selected variable v in greedyVs randomly
7 else
8     x ← randomly selected variable v in c with probability Γ(A,v) / Σ_{u∈c} Γ(A,u)
9
```

2.4 pickVar(A,c) with simulated annealing

Simulated annealing is a technique, whose combination with *walkSAT* is extensively studied. Besides the dynamic noises introduced above, we use *simulated annealing* to improve our three suggestions of *pickVar*. That is, we define the parameter α as a function of two parameters tolerence τ, c_b and the quality function of current assignment $q(A)$ instead of using a static parameter in the whole process:

$$\alpha = \tau \times (c_b)^{-q(A)}$$

To define the quality function $q(A)$, we have two variants: *Global* and *Local*.
Global:
In the process of search, the number of unsatisfied clauses is stored in $unsatN$. In the variant *Global*, we use this number to define the quality of the current solution:

$$q_{global}(A) = unsatN(A)$$

Local: As suggested by the name, in the *Local* variant, we measure the quality of the current assignment focused on the chosen clause c. The quality $q(A, c)$ is equal to the number of greedy literals in current clause:

$$q_{local}(A) = |\{v | v \in c \wedge break(v) = 0\}|$$

2.5 Data structures

In this section, the data structure of our SAT solver is introduced.
Occurrences
In the process of initialization, the numbers of positive and negative occurrences of one variable will be compared. In our implementation, we use a list to count and record these numbers of occurrences. This list of size $2 \times numC$ is denoted as *occurrences list OL*. For the variable with index i, $OL[2i]$ is the number of occurrences of literal v_i; $OL[2i{+}1]$ is for its negative occurrences.

Literals
Only small changes are made in each step of local search. In our situation, only the clauses including the flipping variables are involved in the flipping step. To find the involved clauses of one variable, two *2D* array *posL* and *negL* are created to record the clauses of positive nad negative literals. For variable v_i, $posL[i]$ records the indices of clauses containing the positive literal v_i. The ones with negative literal $\bar{v_i}$ are in $negL[i]$. We flip variable v_i by updating the $numT$ of clauses with indices in $posL[i]$ and $negL[i]$.

LookUp
The most time used in the search is the repeated evaluation of the polynomial or exponential decay function Γ. We calculate the $\Gamma(x)$ with x from 0 to $0.5 \times numC$ and keep the values in a table $lookUp$. In our implementation, we use this table to get the values instead of the reputation of time-consuming (exponential) operation.

Solution
A *solution* in our implementation consists of a boolean assignment and three other structures to record information about the current assignment. The *solution* is set up after assignment initialization and updated during each flipping:

Name	Structure	Size	Meaning
$assignment$	list	$numV$	boolean assignment to variables
$numT$	list	$numC$	number of *true* values in each clause
$numUnsat$	natural number	-	number of unsatisfied clauses
$UNSAT$	set	-	indices of unsatisfied clauses

Table 2: Structures in a solution

2.6 Our swpSAT solver

According to the comparison of performances of different variants, we konfigure our final SAT solver as follows: For 3SAT problems, we choose *randINIT* for initilizataion and the *Average* without simulated annealing as *pickVar* heuristic. For 5SAT and 7SAT problems, we use the *biasINIT* and *Average-Local* (see experiment in 4.8.4). Our local solver is a *statistic* search with a combination of *walkSAT* and *probSAT*, hence the name $swpSAT$.

3 Our Parallel Algorithm

3.1 1st Approach: The pure portfolio approach

Section 2 introduces our local solver *swpSAT*. In experiments, which pseudo-random generator in use to make random values affects the performance. In the pure portfolio version of our algorithm, the agents run the *swpSAT* with different random generation policies. After a satisfying assignment is found by an agent, the search will stop. This approach improved the performance compared to our sequential local search (see experiment 5 in 4.8.5).

Algorithm 8: Pure portfolio approach

```
input     : A CNF Formula F
parameter: Timeout, number of Processors n_p
output    : a satisfying assignment A
1 sat ← false
2 A ← initAssign(F)
3 for each Processor_i do
4 |   A_i ← A
5 |   swpSAT(F)          // This search will be interrupted if sat is set to true.
6 |   sat ← true
7 |   A ← A_i
```

This pure portfolio approach can try different search pathes in parallel. But the threads do not split the whole task. In the following part, we will introduce some partitioning-based techniques. With a good formula partitioning, the search can assign independent jobs to threads.

3.2 2nd Approach: Solver with formula partitioning

With the information of the formula partitioning, the first idea to speed up the search is to parallelize flippings in both partitioning sets. It is possible because the clauses in different partitions do not share variables. The schema of this approach is as follows: The slave thread t_0 execute the *swpSAT* with clauses in H_0. The slave thread t_1 deals with H_1 in parallel with t_0. Then the assignments of the corresponding partitioning sets are written in the shared memory. Then the master thread uses the assignment in shared memory as the initial assignment and makes flips in the intersection I. Handling with conflicts in the intersection will lead to some flips in both partitioning set. As a result, it may introduce conflicts in H_0 and H_1. Then the both processes will start again to handle conflicts in partitioning sets individually. This process will repeat itself until no conflicts exists.

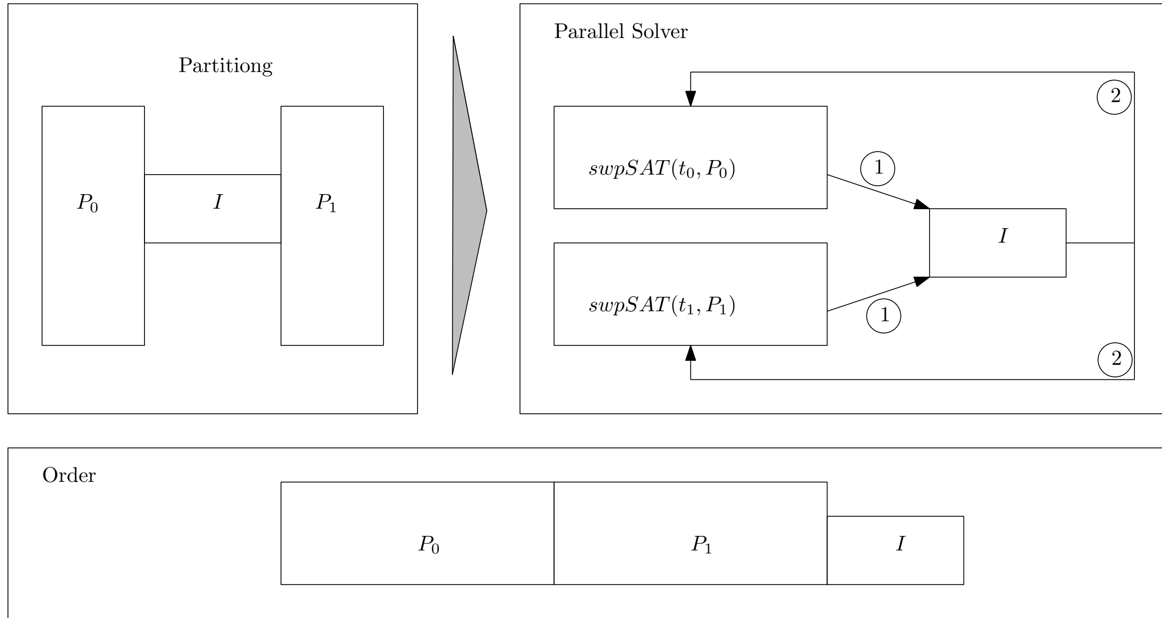
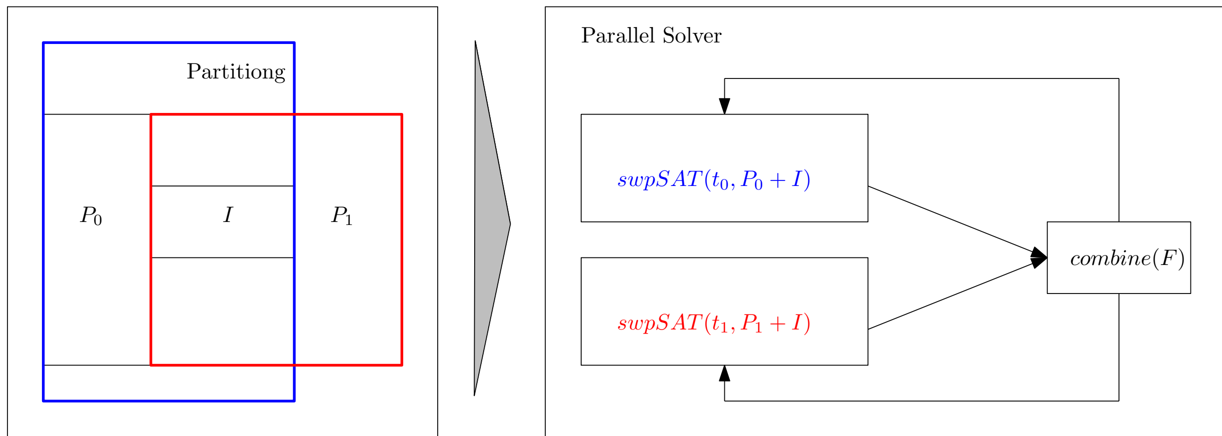


Figure 2: Solver with formula partitioning

We did experiments with benchmark *COMBINE* to test the performance of this approach. It shows a worse performance compared to the original *swpSAT* especially in terms of the number of solved problems. It can only solve trivial small problems. According to our observation in the implementation, one possible explanation is that its sequential version is precisely our *swpSAT* solver with prioritization of unsatisfied clauses. Instead of choosing an unsatisfied clause in the whole formula, this approach solves first P_0 and then P_1 (or inversely) and at last the Intersection I. Still Worse, because of the order of solving the clauses in search, the flippings done in slave threads are easily destroyed in solving the intersection part. These repeated flippings of some variables make the search stuck in one cycling.

3.3 3nd Approach: Solver with combination of sub-assignments

In this approach, the slave threads take intersection into consideration. First, each slave thread finds a satisfying sub-assignment for both the corresponding partitioning set and the intersection part. Here two slave threads may have some different assignments to variables in the boundary of the partitioning sets. Then the master thread deals with these differences and combine these two sub-assignment such that we can get a satisfying assignment or an assignment with only a few conflicts. Then the slave threads will adjust this assignment by make flips in their own part (including the intersection part).



If one variable is assigned with different values in different sub-solutions, there are several cases which need to be considered separately:

1. This variable is not critical for at least one sub-assignment, so we can assign it to the value in the other sub-assignment.

2. If the variable in consideration is critical in both sub-assignment, there are some different policies. The simple one is to assign the variable randomly. We call this policy *randomCombine*. Another one is to assign the variable to the value suggested by its charging thread. In other words, we combine the assignment according to the partitioning. This policy will not break any clauses in partitioning sets. The slave thread will deal with conflicts in intersection individually in the next round. This policy is referred as *partitionCombine*.

In experiments, we compare these two policies. Neither policy can bring a better performance compared with our *swpSAT* solver. The *partitionCombine* process suffers from repeated flips and may not terminate. The possible reason is that the slave threads tend to make the variables in other partitioning set the critical ones. Because flippings of these varibales will not break any clauses in the corresponding partitioning set of the threads.

3.4 4th Approach: Initialization with a guide of formula partitioning

After the analysis of the failures in the two approaches above, we came up with this approach, in which the formula partitioning information is only used to get an initial solution, and the further search in the whole problem is the same as our pure portfolio approach. A local search from this initial solution with only a few conflicts in the intersections can reduce the search space and prevent long-term cycling in the search. The statistic information shared among the agents encourages the further search to flip non-critical variables in clauses.
In experiments (see details in 4.8.6 and 4.8.7), this approach shows a good performance.

4 Evaluation

4.1 DIMACS standard format

All the benchmark formulas used in experiments are encoded in the DIMACS standard format [15]. This format is the standard format of benchmarks used in SAT competitions. A DIMACS file contains the description of an instance using three types of lines:

1. Comment line: Comment lines give information about the formula for human readers, like the author of the file or the seed used in problem generation. A comment line starts with a lower-case character c and will be ignored by DIMACS parser:

c *this is an example of the comment line*

2. Problem line: The problem line appears exactly once in each DIMACS format file. The problem line is signified by a lower-case character p. For a formula with $numV$ variables and $numC$ clauses ,the problem line in its DIMACS file is:

p cnf $numV$ $numC$

3. Clause descriptor: A clause $\{v_1, v_2, \cdots, v_n\}$ in the formula is described in a clause descriptor ended with 0:

$$v_1\ v_2\ \cdots\ v_n\ 0$$

Here is the DIMACS format of the formula $F = (v_1 \vee \bar{v_3}) \wedge (v_2 \vee v_3 \vee \bar{v_1})$:

```
c simple F.cnf
p cnf 3 2
1 -3 0
2 3 -1 0
```

4.2 Benchmarks

The benchmark instances used in our experiments are the 180 instances (*UNIF*) in random benchmark categories in SAT competition 2017 [15]. In a *UNIF* problem file, all the clause have the same length.
To generate an instance with n variables and m clauses, the uniform generation will work in the following way: to construct one clause, k literals are randomly chosen from the $2n$ possible literals. If the generated clause contains multiple copies of the same variables, it will be abandoned. The process generates clauses until m permitted clauses are gathered.
A *UNIF* file name contains some basic information of the problem. The suffix k denotes the length of clauses. The r indicates the clause-to-variable ratio. The c and v are the number of clauses and variables respectively, while s is the seed used in the generation process. Without flitering, at least 60 (33%) problems form our 180 benchmark collections are unsatisfiable.

4.3 Used plots and tables

The results of the following experiments are shown in comparison tables and illusatrated in cactus plots.

Comparison Table
A comparison table shows the results of different algorithms. The first column contains the clause length k. The *UNIF* benchmark has 3 different sizes: 3SAT, 5SAT, and 7SAT. For a kSAT instance, each clause contains k variables. The fields of a comparison table in the following columns correspond to a PAR-2 runtime for the whole kSAT set [15]. Because the *UNIF* benchmark problems contain a part of unsatisfied problems, we assign penalized time only to problems which can not be solved by any solvers in our whole experiments. The number of solved problems are shown below the runtime. The best results in comparison are in bold.
See Table 7 for an example.

Cactus Plot
A cactus plot shows the performance of different algorithms. The y-axis shows the time in second used to solve the benchmark problems. The x-axis represents the number of solved problems by a certain time. Each algorithm corresponds to a curve in a unique color. A point (u, v) on a curve means by v seconds the corresponding algorithm has solved u problems.
See Figure 3 for an example.

4.4 Random Seeds used in Experiments

To make our experiments results reproducible and robust, we repeat our tests with three specific seeds. We produce the seeds as follows: First, we use the sum of characters of the name of the solver to seed the pseudo-random generator in C++. Then we use this reinitialized generator to produce three random values, which are later used as seeds in our experiments.

solver	name	seed 1	seed 2	seed 3
probSAT	probsat	1988822874	338954226	858910419
yalSAT	yalsat	1851831967	280788293	1956345180
our local/parallel solver	local	1962042455	1112841915	566263966

Table 3: Seeds in our solvers and competitors

4.5 Soft- and Hardware

The single-thread experiments were run on computers that equipped with two Intel Xeon E5-2683 v4 processors (2.1 GHz 2x16-cores + 2x16-HT cores) and 512GB RAM. The machine ran the 64-bit version of Ubuntu 14.04.5 LTS. The multi-thread experiments were run on a computer that had two Intel Xeon E5-2650 v2 processors (16 cores + 16 HT cores) with 128GB RAM. The machine ran the 64-bit version of Ubuntu Devel.

4.6 Parameter Settings in Experiment

The *TimeOut* is set to 5 Minutes in our experiments. For the *probSAT* selection heuristic, our local search uses the values for c_b and ϵ suggested in the *probSAT* paper. The parameter α (or the *tolerence* τ in variants with simulated annealing) is configured with the help of the algorithm parameter optimization tool SMAC [16] (sequential model-based algorithm configuration). SMAC ran our algorithms on LARGE problems in the *UNIF* category in SAT 2012 (75% of the instances training instances, 25% as test instances) with values $\in [0, 10]$, step equal to 0.5 and randomly generated seeds[5].

k	*Walk*	*Walk-Local*	*Walk-Global*	*Average*	*Average-Local*	*Average-Global*
3	1	1	10	1	2	0.5
5	0.5	1	1	1	2	2
7	1	0.5	2	1	2	0.5

k	*RF*	*RF-Local*	*RF-Global*	*swpSAT*	α/τ	*SA*
3	1	0.5	0.5	*Average*	1	–
5	0.5	2	9.5	*Average*	2	*Local*
7	0.5	1	0.5	*Average*	2	*Local*

Table 4: Settings of α or τ in solvers

4.7 COMBINE Benchmark Generation

To combine the formula partitioning and SAT local solver, we tried to get a relatively balanced partitioning with small intersection for the hypergraphgraph representation of benchmark SAT problems. We tried to get formula partitioning in problems of *UNIF* benchmark using some partitioning algorithms (KaHypar and hmetis). Due to the uniform random generation of this benchmark, these problems do not possess a real-world-like structure. Even with a high imbalance like 0.3 and high tolerance of intersection size (50% of edge sizes), the partitioning algorithms take more time than our SAT solver for more than half of the *UNIF* benchmarks. To investigate the local search on graphs with proper partitioning, we generate our benchmark *COMBINE* using the *UNIF* benchmark instances. As suggested by the name *COMBINE*, we combine two *UNIF* benchmark instances in one SAT problem and make an intersection for the new generated problem. To create a satisfiable formula, we build the intersection based on a pair of randomly chosen satisfying assignments of the two *UNIF* problems. To get satisfying assignments of *UNIF* benchmarks, we run different solvers in SAT competitions including *CSCCSAT*, *DCCASAT*, *score2SAT*, *probSAT*, and *yalSAT*. We do not use our local search to collect satisfying assignments. Otherwise, it is possible that after solving the two partitioning sets individually using our local solver, the assignment is the same as or similar to the one used for the intersection generation.
We combine the *UNIF* problems in consideration of real-world uses. We combine 4 pairs of instances in 3SAT, 5SAT problems and 7SAT problems. Besides that, we consider five combinations between 3SAT and 5SAT instances and three combinations of 5SAT and 7SAT instances. We combine problems in similar vertex size, which corresponds to balanced partitioning in the structure. We also combine one massive problem with a small instance. For the intersection generation part, we generate clauses with three vertices in different partitioning sets.

[5] Because of high time consume in parameter optimation, we solely compare τ as a natural number form one to ten.

We first choose vertices of two partition sets for intersection generation randomly. Here we also consider the balanced intersection and imbalanced intersection. In a balanced intersection, same proportion of vertices in both partitioning sets are chosen to generate the intersection. In an imbalanced intersection, the portions are not of the same size. To control the size of the intersection, we also count the number of the clauses in an intersection, if the propotions of the intersection clauses reaches a specified limit, the generation of intersection is stopped.
In a *COMBINE* file name f1.cnf-f2.cnf-x-y-z.cnf, x and y are the proportions of vertices in the two partitioning set respectively, whereas z is the proportion of the clauses of the intersection in the whole problem.

Problem	Intersection
k3-r3.94.cnf-k3-r4.04.cnf-0.1-0.1-0.1.cnf	1.04%
k3-r3.96.cnf-k3-r4.06.cnf-0.1-0.1-0.2.cnf	1.00%
k3-r3.98.cnf-k3-r4.0.cnf-0.1-0.1-0.4.cnf	1.03%
k7-r55.0.cnf-k7-r56.0.cnf-0.1-0.2-0.2.cnf	15.53%
k7-r55.0.cnf-k7-r56.0.cnf-0.1-0.4-0.1.cnf	6.84%
k7-r55.0.cnf-k7-r56.0.cnf-0.2-0.2-0.2.cnf	12.04%
k7-r55.0.cnf-k7-r56.0.cnf-0.2-0.4-0.1.cnf	7.53%
k7-r55.0.cnf-k7-r56.0.cnf-0.4-0.1-0.4.cnf	6.38%
k7-r55.0.cnf-k7-r56.0.cnf-0.4-0.2-0.1.cnf	7.35%
k7-r55.0.cnf-k7-r56.0.cnf-0.4-0.4-0.2.cnf	6.11%
k7-r57.0.cnf-k7-r60.0.cnf-0.1-0.1-0.2.cnf	16.67%
k7-r57.0.cnf-k7-r60.0.cnf-0.1-0.1-0.4.cnf	25.21%
k7-r57.0.cnf-k7-r60.0.cnf-0.1-0.2-0.2.cnf	14.77%
k7-r57.0.cnf-k7-r60.0.cnf-0.1-0.4-0.1.cnf	6.46%
k7-r57.0.cnf-k7-r60.0.cnf-0.2-0.1-0.2.cnf	13.92%
k7-r57.0.cnf-k7-r60.0.cnf-0.2-0.2-0.4.cnf	11.40%
k7-r57.0.cnf-k7-r60.0.cnf-0.2-0.4-0.2.cnf	7.08%
k7-r57.0.cnf-k7-r60.0.cnf-0.4-0.1-0.4.cnf	5.88%
k7-r57.0.cnf-k7-r60.0.cnf-0.4-0.2-0.4.cnf	6.86%
k7-r57.0.cnf-k7-r60.0.cnf-0.4-0.4-0.4.cnf	5.81%
k7-r58.0.cnf-k7-r62.0.cnf-0.1-0.1-0.1.cnf	9.09%
k7-r58.0.cnf-k7-r62.0.cnf-0.1-0.1-0.2.cnf	16.67%
k7-r58.0.cnf-k7-r62.0.cnf-0.1-0.1-0.4.cnf	25.07%
k7-r58.0.cnf-k7-r62.0.cnf-0.1-0.2-0.1.cnf	9.09%
k7-r58.0.cnf-k7-r62.0.cnf-0.1-0.2-0.4.cnf	14.68%
k7-r58.0.cnf-k7-r62.0.cnf-0.1-0.4-0.4.cnf	6.36%
k7-r58.0.cnf-k7-r62.0.cnf-0.2-0.1-0.1.cnf	9.09%
k7-r58.0.cnf-k7-r62.0.cnf-0.2-0.1-0.2.cnf	13.96%
k7-r58.0.cnf-k7-r62.0.cnf-0.2-0.2-0.2.cnf	11.58%
k7-r58.0.cnf-k7-r62.0.cnf-0.2-0.4-0.2.cnf	6.97%
k7-r58.0.cnf-k7-r62.0.cnf-0.4-0.1-0.4.cnf	5.98%
k7-r58.0.cnf-k7-r62.0.cnf-0.4-0.4-0.4.cnf	5.60%
k7-r59.0.cnf-k7-r87.79-v90.cnf-0.1-0.1-0.4.cnf	1.30%
k7-r59.0.cnf-k7-r87.79-v90.cnf-0.1-0.2-0.2.cnf	2.52%
k7-r59.0.cnf-k7-r87.79-v90.cnf-0.1-0.4-0.2.cnf	4.88%

Table 5: *COMBINE* problems with big intersection ($\frac{numCI}{numC} > 1\%$)[6]

Problem	Intersection
k3-r3.92.cnf-k3-r3.88.cnf-0.2-0.1-0.4.cnfP	0.38%
k3-r3.92.cnf-k3-r3.88.cnf-0.2-0.4-0.1.cnf	0.13%
k3-r3.94.cnf-k3-r4.04.cnf-0.1-0.2-0.1.cnf	0.36%
k3-r3.94.cnf-k3-r4.04.cnf-0.1-0.4-0.4.cnf	0.11%
k3-r3.94.cnf-k3-r4.04.cnf-0.2-0.1-0.2.cnf	0.37%
k3-r3.94.cnf-k3-r4.04.cnf-0.2-0.2-0.1.cnf	0.27%
k3-r3.94.cnf-k3-r4.04.cnf-0.4-0.2-0.4.cnf	0.11%
k3-r3.94.cnf-k3-r4.04.cnf-0.4-0.4-0.4.cnf	0.09%
k3-r3.96.cnf-k3-r4.06.cnf-0.1-0.2-0.2.cnf	0.37%
k3-r3.96.cnf-k3-r4.06.cnf-0.1-0.4-0.1.cnf	0.11%
k3-r3.96.cnf-k3-r4.06.cnf-0.2-0.1-0.2.cnf	0.35%
k3-r3.96.cnf-k3-r4.06.cnf-0.2-0.2-0.2.cnf	0.27%
k3-r3.96.cnf-k3-r4.06.cnf-0.2-0.4-0.2.cnf	0.13%
k3-r3.96.cnf-k3-r4.06.cnf-0.4-0.1-0.2.cnf	0.10%
k3-r3.96.cnf-k3-r4.06.cnf-0.8-0.8-0.05.cnf	0.06%
k3-r3.98.cnf-k3-r4.0.cnf-0.1-0.2-0.1.cnf	0.38%
k3-r4.267-v11000.cnf-k5-r16.2.cnf-0.8-0.8-0.05.cnf	0.52%
k3-r4.267-v11200.cnf-k5-r16.8.cnf-0.8-0.8-0.05.cnf	0.49%
k3-r4.267-v11600.cnf-k5-r17.0.cnf-0.8-0.8-0.05.cnf	0.52%
k3-r4.267-v7400.cnf-k5-r17.2.cnf-0.8-0.8-0.05.cnf	0.34%
k3-r4.267-v9600.cnf-k5-r17.4.cnf-0.8-0.8-0.05.cnf	0.42%
k5-r21.117-v200.cnf-k5-r16.0.cnf-0.1-0.1-0.2.cnf	0.04%
k5-r21.117-v200.cnf-k5-r16.0.cnf-0.2-0.1-0.2.cnf	0.09%
k5-r21.117-v200.cnf-k5-r16.0.cnf-0.4-0.1-0.4.cnf	0.17%
k5-r21.117-v220.cnf-k5-r17.6.cnf-0.8-0.8-0.05.cnf	0.01%
k5-r21.117-v280.cnf-k5-r16.4.cnf-0.1-0.1-0.4.cnf	0.06%
k5-r21.117-v280.cnf-k5-r16.4.cnf-0.2-0.1-0.1.cnf	0.12%
k5-r21.117-v280.cnf-k5-r16.4.cnf-0.4-0.1-0.1.cnf	0.23%
k5-r21.117-v290.cnf-k5-r16.6.cnf-0.1-0.1-0.2.cnf	0.06%
k5-r21.117-v290.cnf-k5-r16.6.cnf-0.2-0.1-0.4.cnf	0.12%
k5-r21.117-v290.cnf-k5-r16.6.cnf-0.4-0.1-0.1.cnf	0.24%
k7-r59.0.cnf-k7-r87.79-v90.cnf-0.2-0.1-0.4.cnf	0.21%
k7-r59.0.cnf-k7-r87.79-v90.cnf-0.2-0.2-0.4.cnf	0.41%
k7-r59.0.cnf-k7-r87.79-v90.cnf-0.2-0.4-0.2.cnf	0.80%
k7-r59.0.cnf-k7-r87.79-v90.cnf-0.4-0.1-0.4.cnf	0.04%
k7-r59.0.cnf-k7-r87.79-v90.cnf-0.4-0.2-0.4.cnf	0.09%
k7-r59.0.cnf-k7-r87.79-v90.cnf-0.4-0.4-0.2.cnf	0.18%
k7-r87.79-v102.cnf-k5-r17.8.cnf-0.8-0.8-0.05.cnf	0.00%
k7-r87.79-v106.cnf-k5-r18.0.cnf-0.8-0.8-0.05.cnf	0.01%
k7-r87.79-v110.cnf-k5-r18.2.cnf-0.8-0.8-0.05.cnf	0.01%

Table 6: COMBINE problems with small intersection ($\frac{numCI}{numC} < 1\%$)

[6]numCI: number of clauses in intersection

4.8 Experiments

4.8.1 Experiment 1: initAssign(F)

Experiment 1 compares three strategies of initialization in our solver. The original *probSAT* algorithm builds a complete assignment randomly in the initialization phase.

The *BiasInit* assigns boolean values to variables based on occurrences of their literals. It assigns *true* to variables whose positive literal occurs more than its negative literal. The number of unsatisfied clauses in the bias initialized assignment of a kSAT problem is limited to $\frac{numC}{2^k}$. Our hyperthesis in the assignment initialization is that an initial solution with good quality can speed up the search generally.

In a combination of these two variants, the *Bias-RandomInit*, the boolean value is assigned to variables bias randomly based on literals occurrences. The probability of assigning *true* to variable v_i is $\frac{posOccurences[i]}{posOccurences[i]+negOccurences[i]}$.

In a local search algorithm, it is common practise to replace the current solution with a new initial solution after a certain number of tries. However, in our implementation, the current assignment will be changed by locally flipping of variables after the initialization. In the following table, we compare three variants with the original *probSAT* algorithms. Based on the results of this experiments, we determined the way of initialization in our heuristic solver.

k	*RandomInit*	*BiasInit*	*Bias-RandomInit*
3	9221.9 **55**	9157.76 54	**9078.27** **55**
5	7143.9 82	**4351.09** **87**	4582.54 **87**
7	6238.51 60	**5421.9** 60	6310.7 60

Table 7: For 3SAT problems, different initializations get similar performances. According to the PAR2-score of the benchmark sets, the two bias initiliazations show improvement of about 40% percentage reduction in runtime for 5SAT problems. the *BiasInit* shows its efficiency in 7SAT problems.

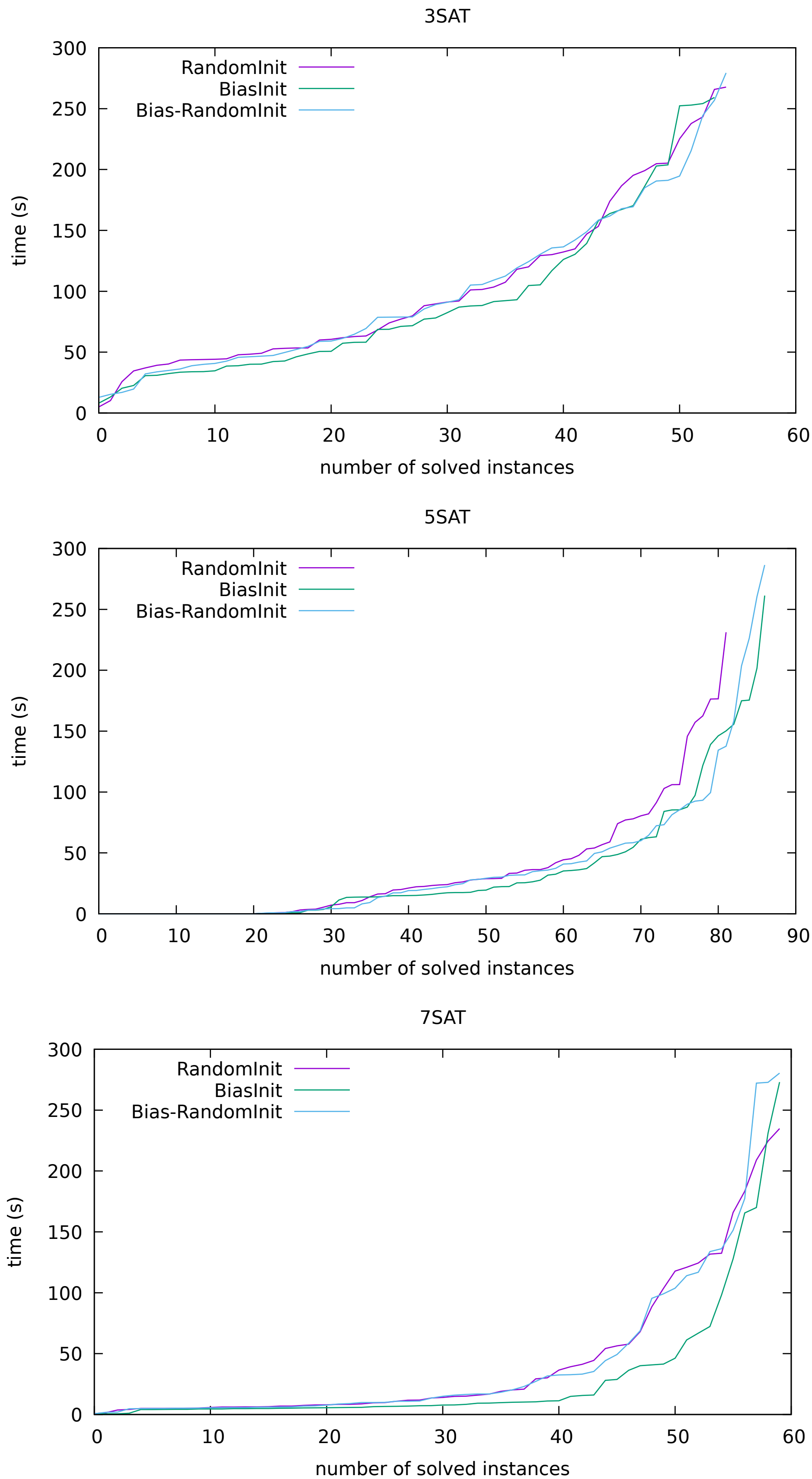


Figure 3: Three suggestions have very similar performance for 3SAT problems. In our solver, we use *RandomInit* for 3SAT because of its simplicity. Two bias suggestions show advantages especially for huge 5SAT instances. For 7SAT problems, the two random initialization are similar in performance, while the bias initialization shows its efficiency.

4.8.2 Experiment 2: pickVar(F)

To pick a variable for flipping, the *probSAT* needs to measure the scores of the literals in the chosen clause and then model the probability distribution of the scores. This stochastic process is very time-consuming. However, in the first phase of the search, picking greedy flippings can lead the search quickly to an optimal solution. In our examples in the Table 8, the probability to ignore the greedy literal is 30% for the 3SAT example and more than 50% for 5SAT and 7SAT. As a result, even if the search can reach the final satisfying assignment by making a few "right" moves, the probability of the *probSAT* algorithm making "wrong" moves is still rather high.

k	Breakcount	Probability
3	$\{0, 1, 1\}$	$\{70\%, 15\%, 15\%\}$
5	$\{0, 1, 1, 1, 1\}$	$\{48\%, 13\%, 13\%, 13\%, 13\%\}$
7	$\{0, 1, 1, 1, 1, 1, 1\}$	$\{47\%, 9\%, 9\%, 9\%, 9\%\}$

Table 8: Three clause examples in 3SAT , 5SAT and 7SAT problem with a greedy literal and other literals with breakcount 1. With the Γ function and the recommended parameters in the original *probSAT* paper, the probabilities of literals for flipping are schown in the 3rd column.

To make the selection more greedily, we use a random walk to choose policy between greedy flipping and the *probSAT* heuristic. In our algorithm, we refer the literals with no clause breaking as a greedy literal whereas in original *walkSAT* greedy literal means the one with smallest breakcount. To avoid cycling in one region, we record the statistic information in the process. If the flips of chosen greedy literal have been repeated many times, the selection will prefer another variable with a small breakcount, which has been flipped only a few times.

k	*probSAT*	*Walk*	*Average*	*Random-Flip*
3	9221.9 (55)	7430.12 (57)	**6161.11**(**61**)	8362.42 (55)
5	7143.9 (82)	4433.05 (87)	**3308.16**(**89**)	4052.47(87)
7	6238.51(**60**)	6358.76(**60**)	6525.597(59)	**5800.46**(**60**)

Table 9: In the comparison with *probSAT*, our three variants of *pickVar* are faster and solve more instances in 3SAT and 5SAT. For 3SAT, our suggestions have better performances. The *Walk* and *Average* can solve more problems. The best one is the *Average*, which solves 10% more problems. The *Average* also shows advantages for 5SAT problems, which can solve 8% more problems than the *probSAT* using only 46% time. For 7SAT, there are no noticeable differences in results.

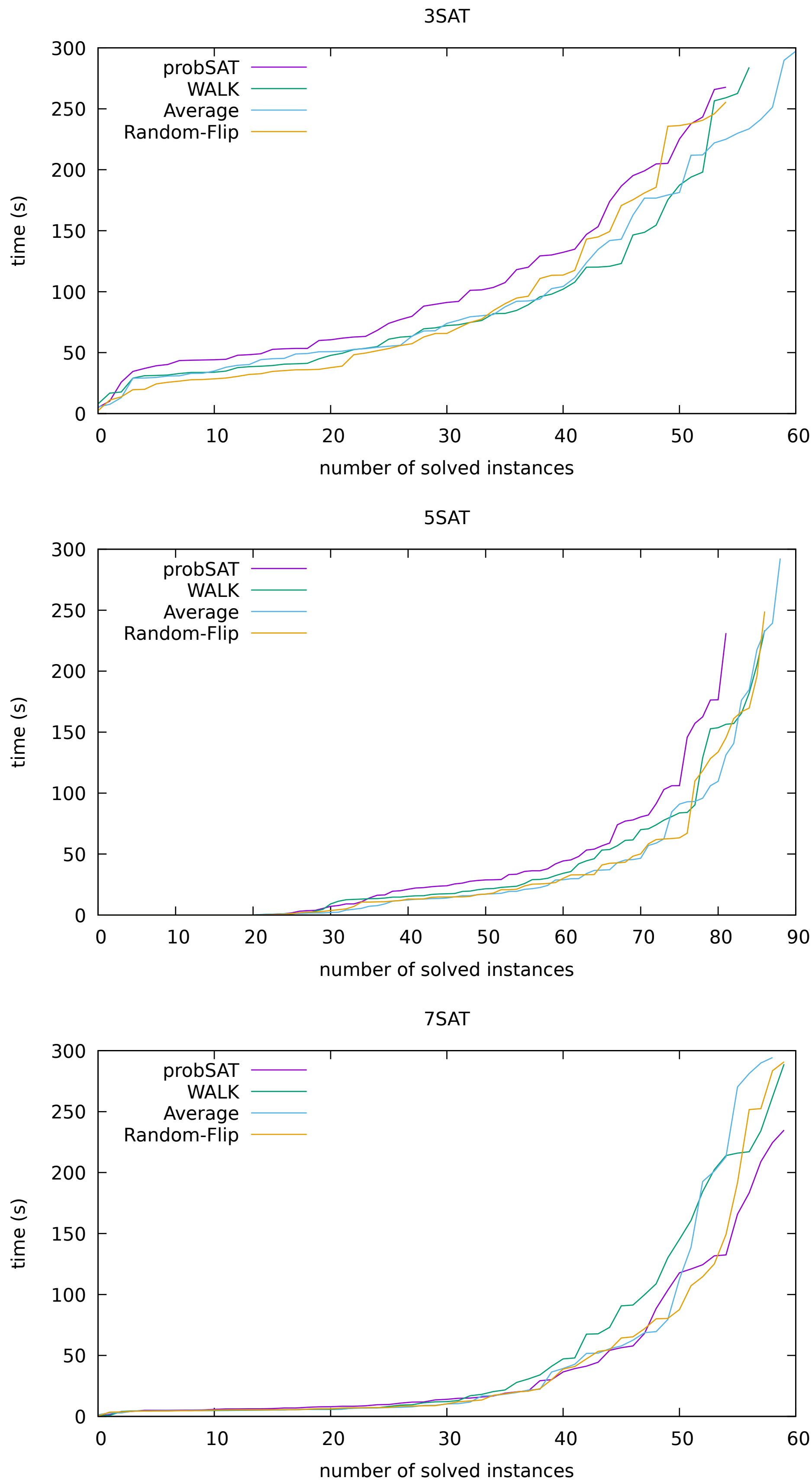


Figure 4: Our suggestions are faster than the *probSAT* algorithm in 3SAT problems. The *Averrage* shows an improvement except for a few trivial small instances. For 5SAT problems, the improvement through our suggestions are obvious. They can efficiently solve more problems. For 7SAT problems, the *Walk* and *Average* have generally worse inperformance. The *Random-Flip* can solve the same number of problems as the original *probSAT* and has shown littel improvement in performance within 125 seconds.

4.8.3 Experiment 3: Simulated Annealing

To use the technique *simulated annealing*, we need to define the quality of the current solution. One traditional way is to use the number of conflicts. In addition, we propose to use the number of greedy literals in the chosen clause. Our hyperthesis behind this idea is that if the current solution is close to a satisfying assignment without conflicts, it is likely to break a clause by a flipping. And in such cases, there should be only a few greedy literals in the chosen clause. We refer this quality as local quality because it is more specific about the chosen clause, rather than about the assignment.

The paramters α is defined as $\alpha = \tau \times (c)^{-q}$. Here the paramter c is in $[2, \cdots, 6]$. The noise value α is proportional to the probability of a random walk. If the quality of the current solution is bad, the search will have a big noise, and thus the search will prefer random literals for flippings. The search can take advantage of random algorithm and avoid getting stuck in one region. That will speed up the search in the first phase. When the search is close to a satisfying assignment, the search will make small improvements with greedy steps.

In our implementation, we set up a lookup table for the exponential values of the parameter c_b to avoid the repeated calculations of Γ function. To reuse these exponential values in variants with the simulated annealing, we set the paramter c to the value of c_b.

k	*Average*	*Average-Local*	*Average-Global*	*RF*	*RF-Local*	*RF-Global*
3	5.43%	2.43%	5.43%	9.98%	10.01%	3.13%
5	4.07%	0.35%	0.00%	9.18%	9.17%	0.33%
7	2.28%	0.03%	0.00%	4.30%	4.14%	0.05%

Table 10: To confirm that the simulated annealing with the parameters in our experiments changes the behavior of the search, we count the number of greedy flips and the number of random flips for the whole problem set. Through our observation, using the parameters we choose, the fraction of greedy flips in total flips are changed. This table shows the values for two variants *Average* and *Random-Flip*.

In the following, we show the results of our three variants *Walk*, *Average* and *Random-Flip* with simulated annealing and compare these two definitions of quality.

Based on the results, we modify our variants where performance improvement exists with simulated annealing. In experiment 4 in 4.8.4, we list the performances of these modified variants.

Walk with Simulated Annealing

k	*Walk*	*Walk-Local*	*Walk-Global*
3	**7430.12**(**57**)	8346.76(56)	9023.96(56)
5	4433.05(87)	3330.61(89)	**3117.45**(**89**)
7	6358.76(60)	**5409.67**(**61**)	6566.06(59)

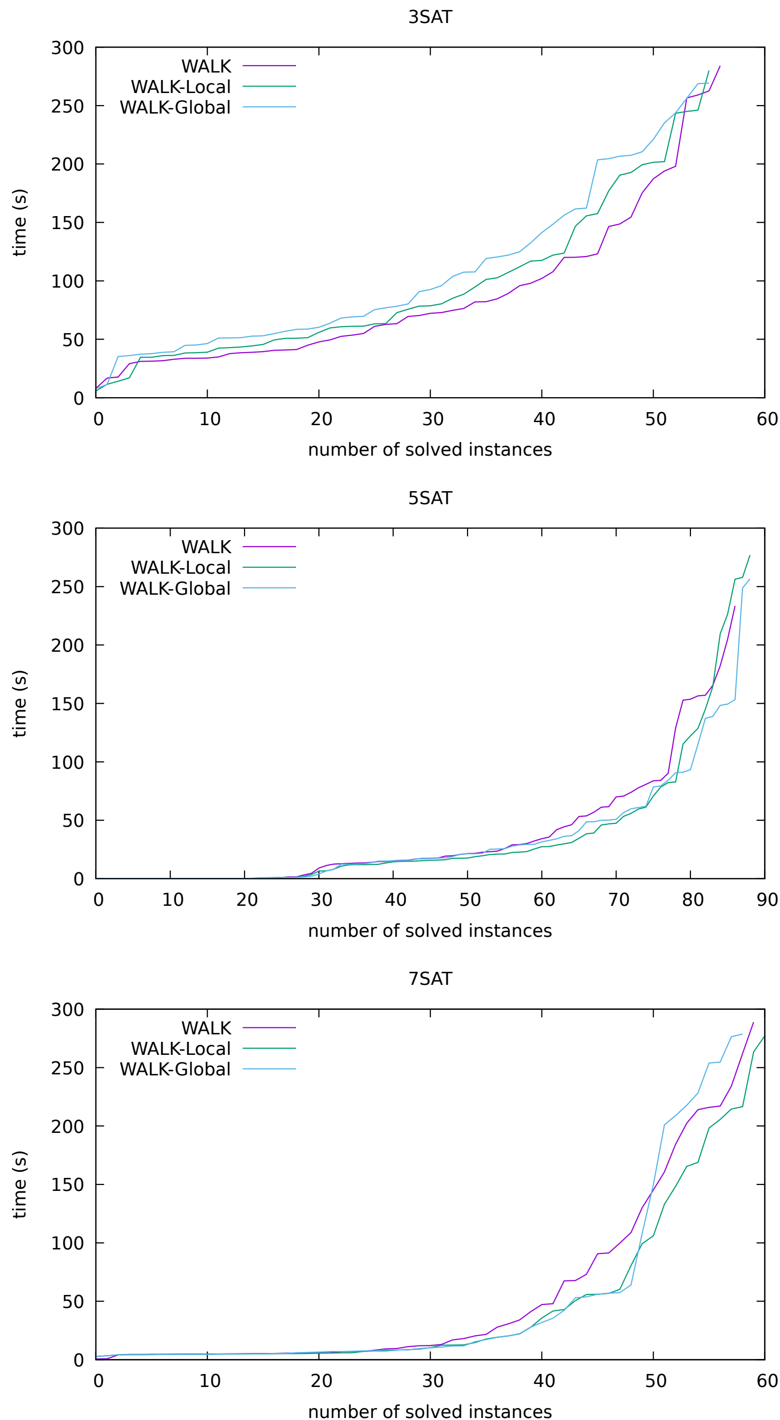


Figure 5: For 3SAT, the two combination with simulated annealing show worse performance than the original *Walk*. The combinations have shown their efficiency in solving huge 5SAT problems. The *Local* shows its advantages for 7SAT problems.

Average with Simulated Annealing

k	*Average*	*Average-Local*	*Average-Global*
3	**6161.11(61)**	9254.18(53)	9870.25(53)
5	3308.16(89)	**2793.32**(89)	2939.74(89)
7	6525.59(59)	**3829.95(65)**	5738.2(61)

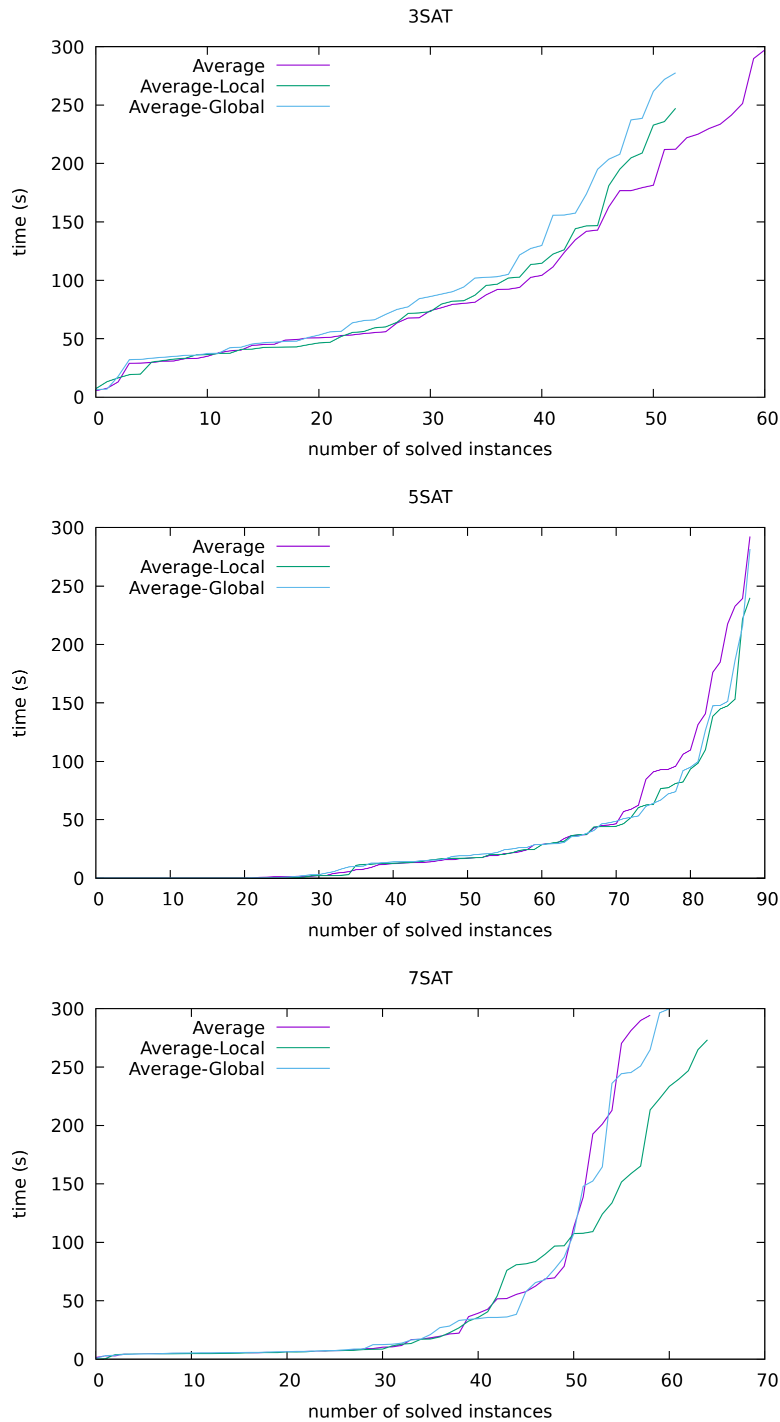


Figure 6: The original *Average* can solve 15% more problems in 3SAT. For 5SAT, there are not noticable differences in aspect of number of solved problems or the runtime. For 7SAT problems, the *Local* version can solve 6 more huge problems than the original *Average* within less runtime.

Random-Flip with Simulated Annealing

k	*Random-Flip*	*Random-Flip-Local*	*Random-Flip-Global*
3	8362.42(55)	8409.8(55)	**7308.01**(**58**)
5	4052.47(87)	4132.07(87)	**4003.06**(**88**)
7	5800.46(60)	6792.23(59)	**4903.61**(**60**)

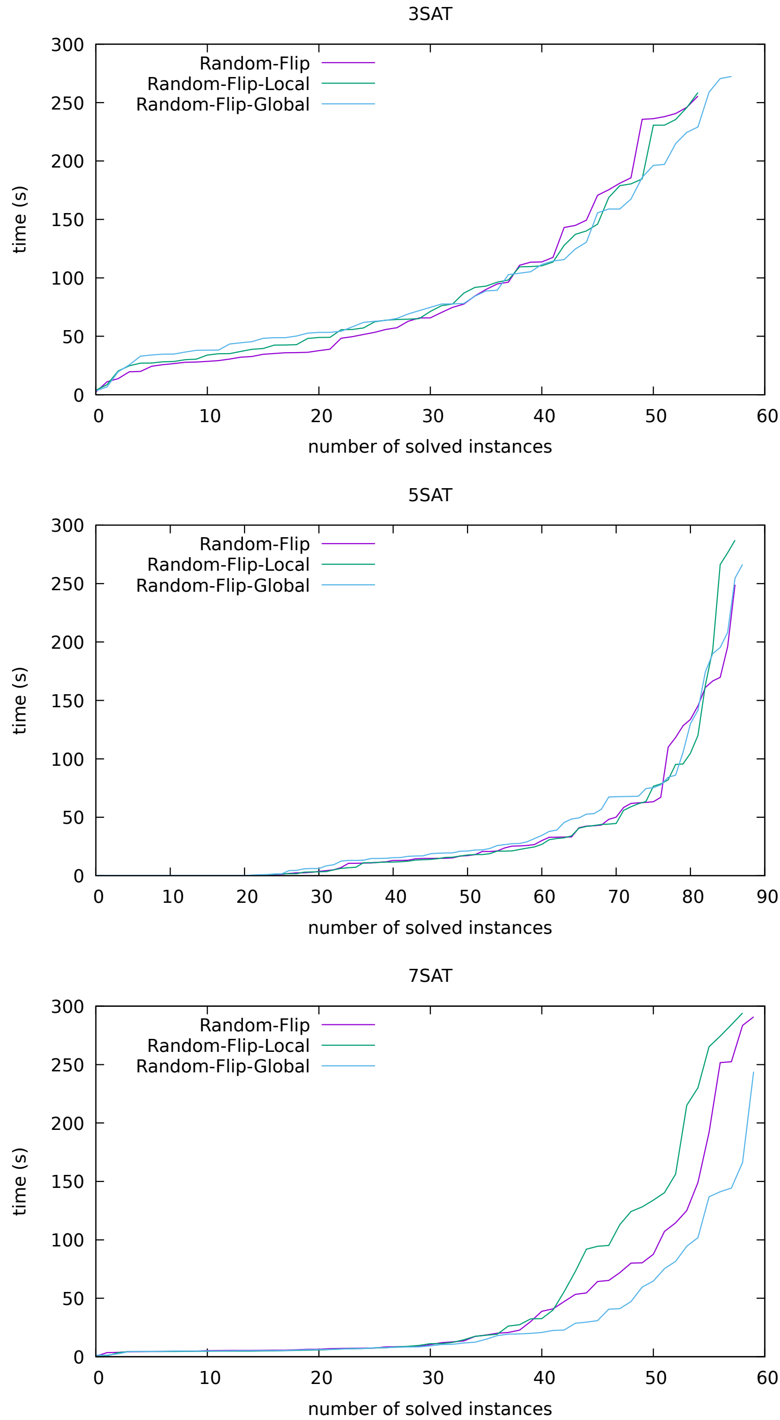


Figure 7: The performances of these three algorithms are quiet similar for 3SAT and 5SAT problems. For 7SAT set, the *Local* get worst performance while the variant *Global* has shown advantages in solving huge problems.

4.8.4 Experiment 4: 2017-UNIF Comparision (local)

In Table 14, we compare our variants with *probSAT* and *yalSAT*. Generally, our variants get advantages with respect to runtime and the number of solved problems. According to the comparison of performances of these three variants, we configure our final SAT solver *swpSAT* as follows: For 3SAT problems, we choose *randINIT* for initilizataion and the *Average* without simulated annealing as *pickVar* heuristic. For 5SAT and 7SAT problems, we use the *biasINIT* and *Average-Local*.
We count in the search the total number of flips for a problem set. There are no noticeable differences in the average number of flips per second[7]. It confirms that the improvement we get with our variants is mostly from the advantages of our algorithms instead of the implementation.

k	*probSAT*	*yalSAT*	*Walk*	*Average/swpSAT*	*Random-Flip*
3	9221.9 55	17062.35 41	7430.12 57	**6161.11** **61**	7308.01 58
5	7143.9 82	5676.63 85	3330.61 **89**	**2939.74** **89**	4003.06 88
7	6238.51 60	10063.4 54	5409.67 61	**3829.95** **65**	4903.61 60

Table 11: Comparison in UNIF

k	*probSAT*	*Walk*	*Average/swpSAT*	*Random-Flip*
3	954426	946662	**1012761**	988477
5	389453	**443172**	414765	423364
7	248025	237387	**248028**	221107

Table 12: Average flips per second

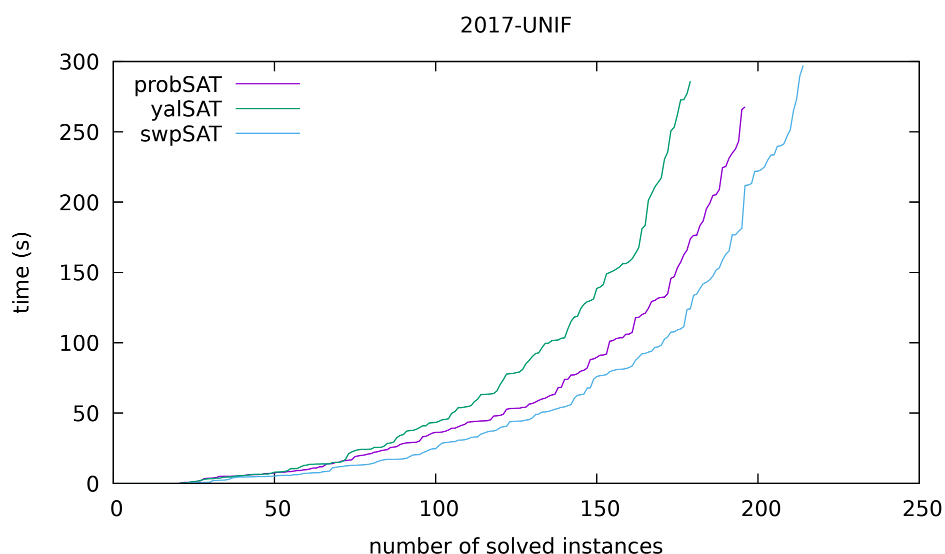


Figure 8: *swpSAT* gets best performance in *UNIF* compared with *probSAT* and *yalSAT*

[7] Here we consider the solved instances without runtime penalization.

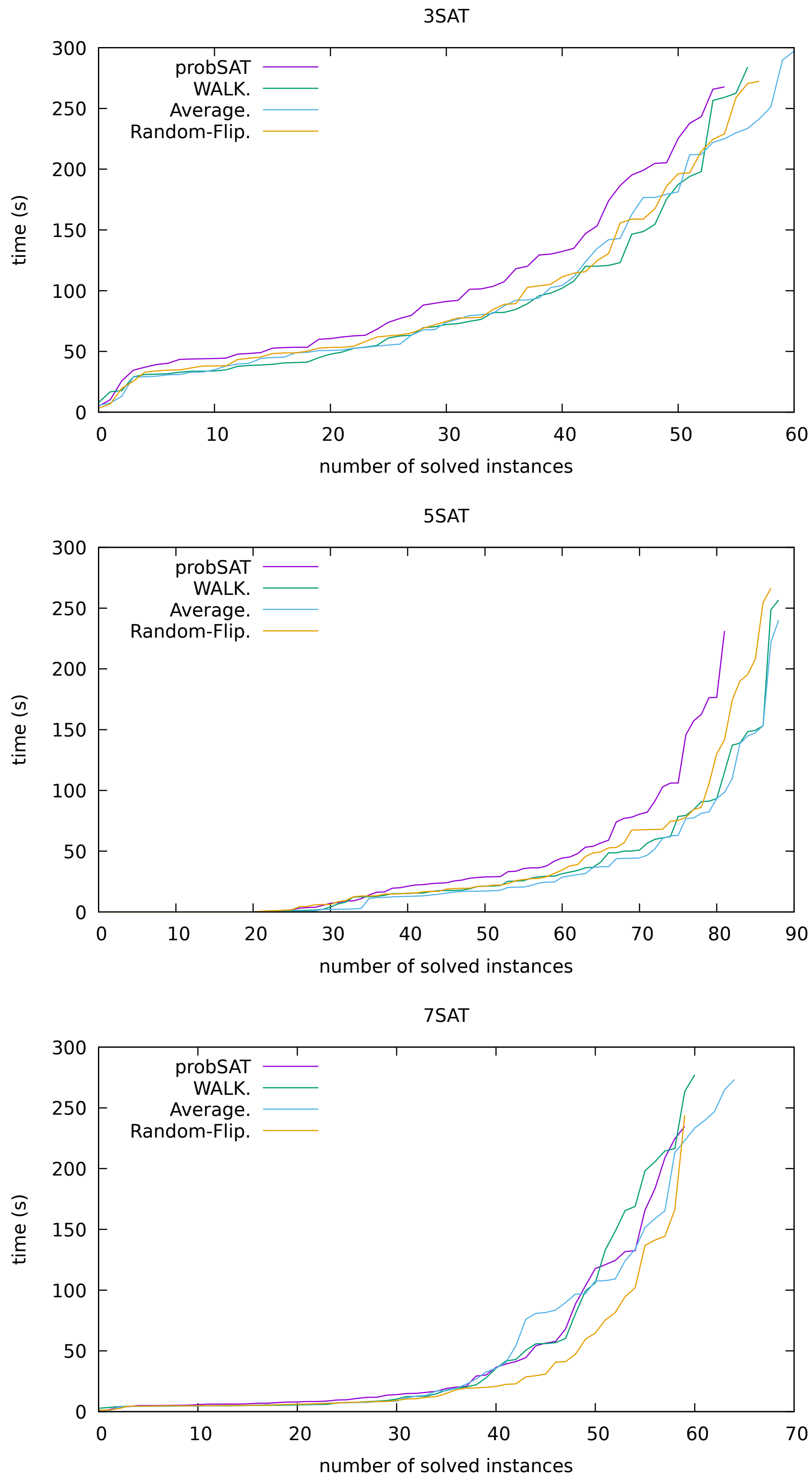


Figure 9: Our variants can get better performances for 3SAT and 5SAT problems in the whole process. For 7SAT poroblems, our variants can solve more problems than the original *probSAT*.

4.8.5 Experiment 5: The pure portfolio approach

Our parallel implementation uses *OpenMP* to support shared memory multiprocessing. In the *C++* implementation of our local solver, we use the function *rand*() in the standard library to generate pseudo-random integers. This function is not thread-safe. To implement a deterministic parallel implementation, we use *rand*() for in the first thread and use the thread-safe random engine in *C++* for random value generation in other threads. Based on our experiments, the sequential implementation with the simple *rand*() function has the best performance for the whole set. But there is no single random generator that is advantageous for all the problems.

This approach takes advantage of the differences of random generator in performance. The threads execute the *swpSAT* with different random engines in parallel. If one thread find a satisfying assignment, the whole parallel search will be stopped.

rand()	*minstd_rand*	*mt*19937
*mt*19937_64	*ranlux*24_*base*	*ranlux*48_*base*
*ranlux*24	*ranlux*48	*knuth_b*
default_random_engine	*minstd_rand*0	-

Table 13: Pseudo-random number generators in use. In our pure portfolio approach, we run 11 threads. Each thread uses a random generator for generation of all boolean and integer values in the whole search process.

k	*swpSAT*	*pure portfolio*	Speedup	Efficiency
3	12971.3 59	7426.9 69	1.75	0.16
5	8339.98 89	5185.75 94	1.61	0.14
7	13406.26 66	2853.81 83	4.70	0.43

Table 14: Our pure portfolio approach can solve more problems compared to our local *swpSolver*. The ratio of the sequential execution time to the parallel execution time is shown in the column Speedup. According to the speedup values, our pure portfolio parallel approach accelerates the process. Another metric to measure the performance of a parallel algorithm, namely efficiency, is defined as the ratio of speedup to the number of threads. The best speedup and efficiency values we got are the ones in the 7SAT problem set.

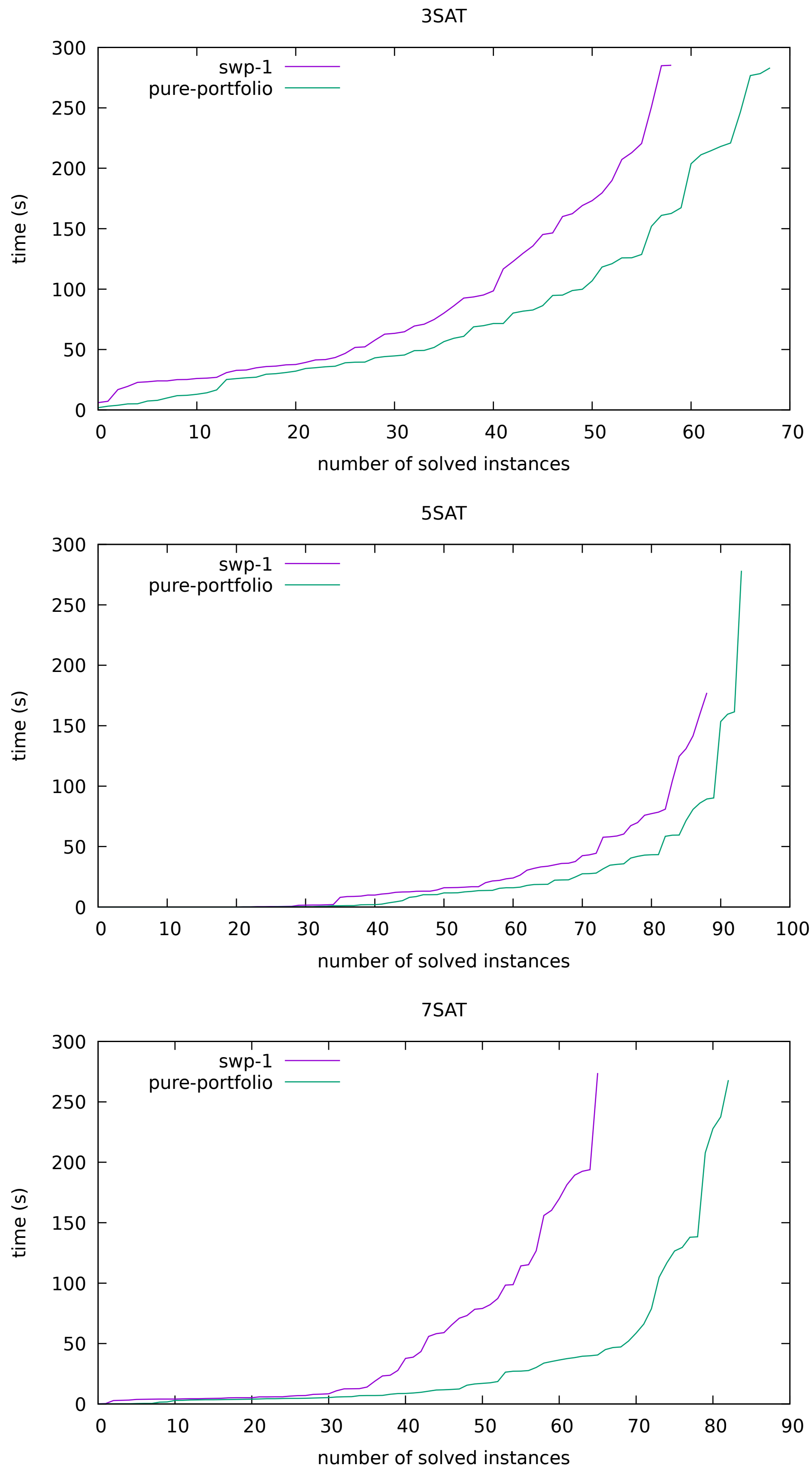


Figure 10: The pure portfolio approach can get better performances and efficiencies in whole problem set. Expecially for 7SAT problems, the parallel solver can solve 30% more problems in the same amount of execution time.

4.8.6 Experiment 6: Initialization with a guide of formula partitioning

A file of *COMBINE* benchmark contains a combination of two UNIF problems and a relatively small intersection between them. So according to the number of vertices in two partitioning sets, the solver can easily divide the clauses into H_0, H_1, and intersection I. This experiment simulated a situation, in which the SAT problem is from some real-world application and a proper partitioning is available.
In this experiment, we solve the *COMBINE* benchmark with the 4th parallel approach (see 3.4).
In the initialization phase, two slave threads solve the partitioning sets in parallel and then use the assignment without conflicts in both partitioning sets as the initial solution. This approach is called *FineInit* in the following. The pseudocode of the approach is shown below:

Algorithm 9: FineInit

```
input    : A CNF Formula F
parameter: Timeout
output   : a satisfying assignment A
1  sat ← false
2  A ← initAssign(F)
3  foreach (Processor_t for t ∈ {0,1}) do
4      A_t ← A
5      // If a satisfying sub-assignment is found, it will be written in the
6      // assignment A in shared memory.
7      swpSAT(P_i)
8      while (!sat ∧ !Timeout) do
9          A_t ← A
10         swpSAT(F)
11         sat ← true
```

problems	*swpSAT*	*FineInit*	Speedup	Efficiency
$3-3$(big)	1861.73	501.78	3.71	1.86
$3-3$(small)	7955.19	1734.79	4.59	2.30
$3-5$	-	-	-	-
$5-5$	2748.13	621.47	4.42	2.21
$5-7$	-	-	-	-
$7-7$	3276.39	376.06	8.71	4.36
BIG	2794.15	902.69	3.10	1.55
SMALL	15579.71	3099.02	5.03	2.52
COMBINE	18373.86	4001.71	4.59	2.30

Table 15: We compare our local swpSolver and our 4th approach of the parallel solver, which is referred as *FineInit* in this table. This approach gets a super-linear speedup thanks to the guide of formula partition information.

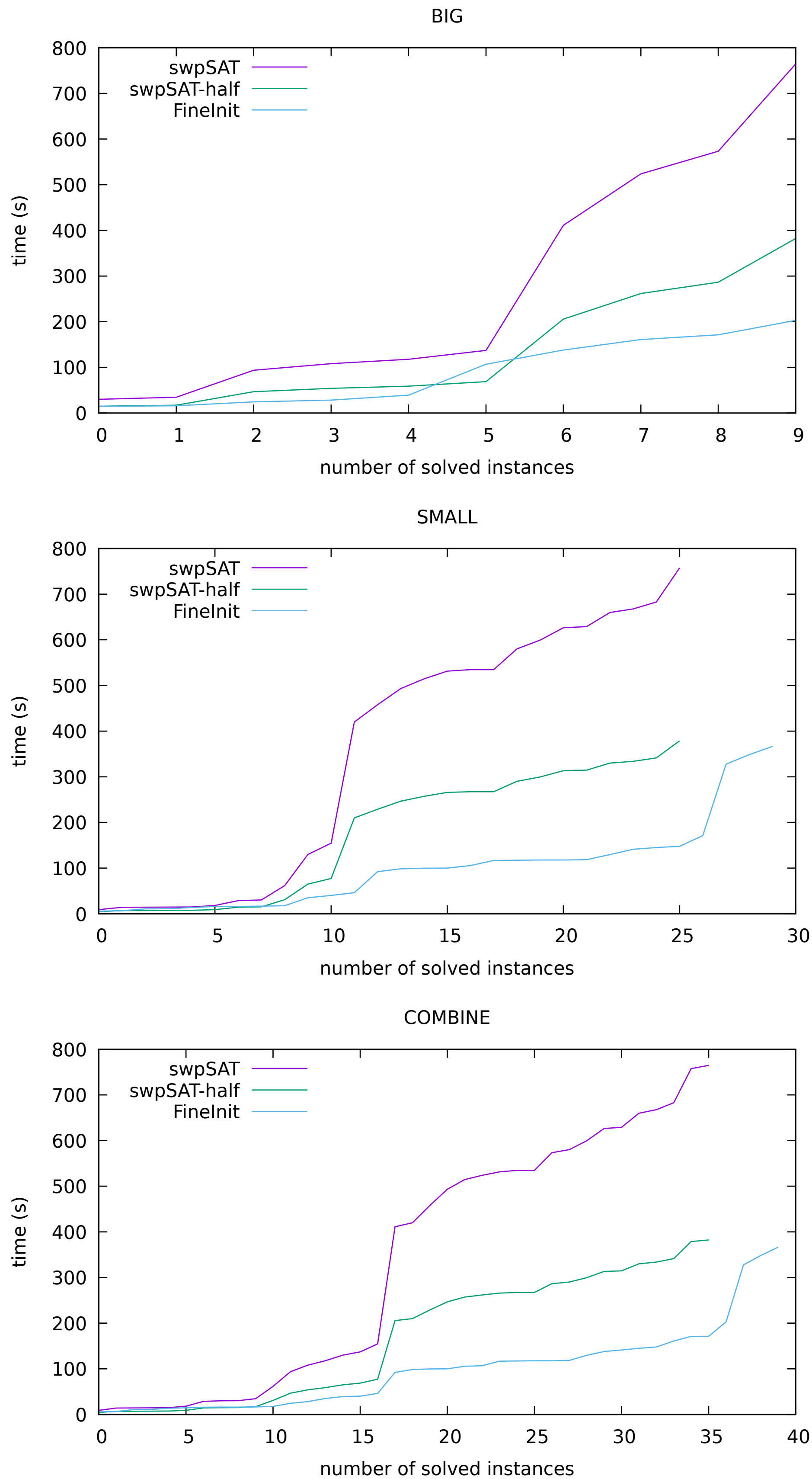


Figure 11: The green curve represents the half of execution times with swpSolver. With a comparison of the swpSAT-half curve and the FineInit curve, the efficiency of our *FineInit* solver with two threads are shown in figures. Our *FineInit* solver gets a super-linear efficiency for problems even with relative big intersections.

4.8.7 Experiment 7:2017-UNIF Comparison (parallel)

Our final parallel solver uses *FineInit* as initialization. Then the pure portfolio approach tries different search paths. The pseudocode is shown below.

Algorithm 10: Our parallel solver

```
input    : A CNF Formula F, number of Processors n_p
parameter: Timeout
output   : a satisfying assignment A
1  sat_0 ← false
2  sat_1 ← false
3  sat ← false
4  A ← initAssign(F)
5  foreach (Processor_t for t ∈ {1, .., n_p}) do
6  |  A_t ← A
7  |  i ← t%2
8  |  //The search swpSAT(P_i) will be interrupted if sat_i is set to true.
9  |  // If a satisfying sub-assignment is found, it will be written in the
10 |  // assignment A in shared memory.
11 |  swpSAT(P_i)
12 |  swpSAT(P_{1−i})
13 |  while (!sat ∧ !Timeout) do
14 |  |  A_t ← A
15 |  |  swpSAT(F)
16 |  |  sat ← true
```

A *UNIF* problem is generated uniform randomly. It is hard to get a relatively balanced partitioning with a small intersection. In this experiment, we test our final parallel solver on the *UNIF* benchmark with one certain partitioning. We separate the vertices according their indices in two partition sets. All vertices v_i with $i < \frac{numV}{2}$ belong to P_0. The rest vertices belong to P_1.

Random number generator used in solvers for comparison are seeded from system time.

k	*swpSolver*	*swp-4*	Speedup	Efficiency	*swp-11*	Speedup	Efficiency
3	4990.53 19	3350.36 21	1.49	0.37	2376.26 24	2.10	0.19
5	3109.75 29	2535.86 30	1.23	0.31	1115.99 33	2.79	0.25
7	4974.01 21	3874.51 22	1.28	0.32	1076.26 27	4.62	0.42

Table 16: With the increment of the number of threads, our solver can solve more problems.

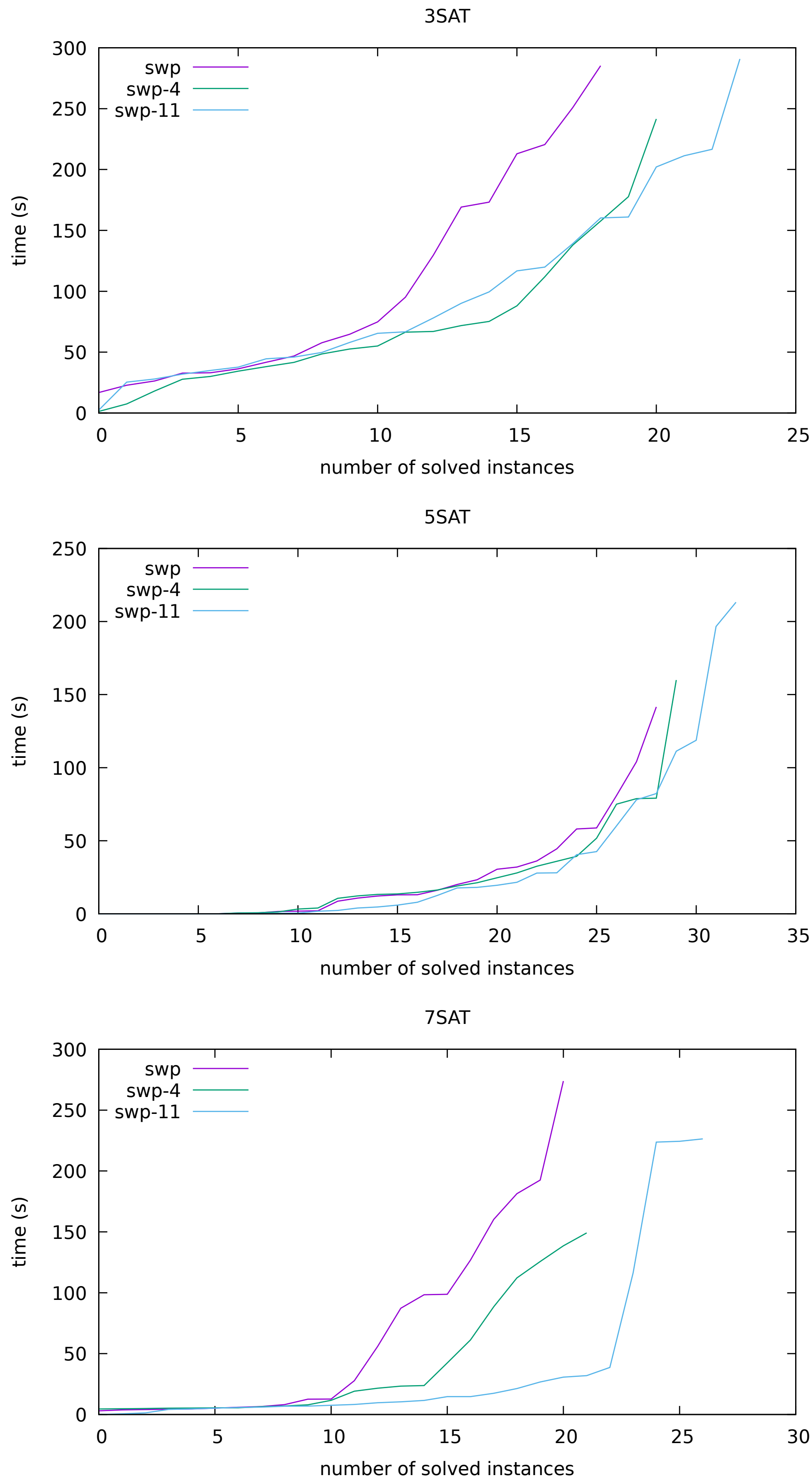


Figure 12: For 3SAT and 5SAT problems, the more threads the solver spawns, the more problems are solved. The time used for solving small and medium problems are quite similar. For 7SAT problems, our parallel solver shows advantages in terms of both the number of solved problems and the runtime.

5 Conclusion

Local search is a universally applicable approach to solve random SAT problems. We presented a stochastic local search algorithm with the incorporation of *walkSAT* and *probSAT*.

In section 2 we discussed the basic scheme of our sequential algorithm. We oprimized the assignment initialization. As shown in our experiments, our initialization, which is based on the occurrence of literals in formula, is advantageous with respect to both the number of solved problems and the execution time of the search. To get the advantage of the greedy algorithm and the stochastic process, we introduced a data structure called statistic list to guide the decision between these two processes in step *pickVar*. Here we proposed some variants to combine greedy choice and statistic heuristic. Generally, our local searches get better performance than the *probSAT* algorithm.

Based on the performance of these variants in differnt categories of *UNIF* problems, we got our *swpSAT* solver, which combined the advantages of the local searches.

In section 3, we parallelized our *swpSAT* solver with different approaches, in which the agents run the search with different random generators. Then we discussed different combinations of formula partitioning with our parallel solver. After trying several approaches with failures, we found that the approcah *FineInit* which used formula partitioning to guide the assignment initialization is able to reduce the time expense. Furthermore, it solved more problems. Our experiments verified the hyperthesis that the formula partitioning information can guide the local search and improve the efficiency in the parallel search.

5.1 Further work

While this thesis has demonstrated the potential of parallel searches for the SAT problem, many opportunities of other investigation directions remain. This section presents some of these directions.

Using different search strategies
In our parallel algorithm, we use the stochastic algorithm *swpSolver* as the subroutine. In further research, the agents can use different search strategy. It is meaningful to investigate the cooperation of different local searches.

Using different cooperation strategies
In this thesis, the cooperation is limited to sub-assignment combination. Some other directions like generic population-based metaheuristic are definitely worth further research.

Using different random generation in local search
Our parallel solver explored different search paths by using different random generation strategies in different agents. An interesting topic is the cooperation of these generation strategies in one sequential search.

6 Bibliography

References

[1] S. A. Cook, "The complexity of theorem-proving procedures," in *Proceedings of the third annual ACM symposium on Theory of computing*, pp. 151–158, ACM, 1971. (Page 1).

[2] E. Clarke, A. Biere, R. Raimi, and Y. Zhu, "Bounded model checking using satisfiability solving," *Formal methods in system design*, vol. 19, no. 1, pp. 7–34, 2001. (Page 1).

[3] F. Ivančić, Z. Yang, M. K. Ganai, A. Gupta, and P. Ashar, "Efficient sat-based bounded model checking for software verification," *Theoretical Computer Science*, vol. 404, no. 3, pp. 256–274, 2008. (Page 1).

[4] H. Kautz and B. Selman, "Unifying sat-based and graph-based planning," in *IJCAI*, vol. 99, pp. 318–325, 1999. (Page 1).

[5] Z. Á. Mann and P. A. Papp, "Guiding sat solving by formula partitioning," *International Journal on Artificial Intelligence Tools*, vol. 26, no. 04, p. 1750011, 2017. (Page 1).

[6] D. S. Johnson, "Local optimization and the traveling salesman problem," in *International colloquium on automata, languages, and programming*, pp. 446–461, Springer, 1990. (Page 2).

[7] P. Galinier and A. Hertz, "A survey of local search methods for graph coloring," *Computers & Operations Research*, vol. 33, no. 9, pp. 2547–2562, 2006. (Page 2).

[8] F. Glover, "Tabu search—part i," *ORSA Journal on computing*, vol. 1, no. 3, pp. 190–206, 1989. (Page 3).

[9] G. Li, "Solving the graph coloring problem with cooperative local search," (Page 3).

[10] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *science*, vol. 220, no. 4598, pp. 671–680, 1983. (Page 4).

[11] H. H. Hoos *et al.*, "An adaptive noise mechanism for walksat," in *AAAI/IAAI*, pp. 655–660, 2002. (Page 4).

[12] A. Balint and U. Schöning, "Engineering a lightweight and efficient local search sat solver," in *Algorithm Engineering*, pp. 1–18, Springer, 2016. (Page 4).

[13] R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "Abstract dpll and abstract dpll modulo theories," in *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pp. 36–50, Springer, 2005. (Page 5).

[14] A. Biere, "Yet another local search solver and lingeling and friends entering the sat competition 2014," *SAT Competition*, vol. 2014, no. 2, p. 65, 2014. (Page 6).

[15] T. Balyo, M. J. Heule, and M. Jaervisalo, "Proceedings of sat competition 2017: Solver and benchmark descriptions," 2017. (Pages 14, 15).

[16] "Smac: Sequential model-based algorithm configuration." `http://www.cs.ubc.ca/labs/beta/Projects/SMAC/`. Accessed: 2017-03-10. (Page 16).