# 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, 20th September 2018

**Abstract**

Stochastic local search (SLS) is an elementary technique for solving combinational problems. Probsat is an algorithm paradigm of the simplest SLS solvers for Boolean Satisfiability Problem (SAT), in which the decisions only based on the probability distribution. In the first section of this paper, we introduce an efficient Probsat heuristic. We experimentally evaluate and analyze the performance of our solver in a combination of different techniques, including simulated annealing and WalkSAT. With the approach of formula partition, we introduce a parallel version of our solver in the second section. The parallelism improves the efficiency of the solver. Using different random generator and other parameter settings 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. Probsat ist einer der einfachsten SLS-Solver für das Erfüllbarkeitsproblem der Aussagenlogik (SAT), bei dem die Entscheidungen nur auf der Wahrscheinlichkeitsverteilung basieren. Im ersten Teil dieser Arbeit stellen wir eine effiziente Probsat-basierte Heuristik vor. Die Leistung unseres Algorithmus in einer Kombination verschiedener Techniken, einschließlich simulierter Abkühlung und WalkSAT wurde auch experimentell bewertet und analysiert. Mit dem Ansatz der Formelpartition wird im zweiten Teil eine parallele Version unseres Algorithmus eingeführt, die die Effizienz des Lösers verbessert. Die flexible Parametereinstellungen bei der Lösung der Teil-formeln kann eine weitere Verbesserung unseres Algorithmus bringen.

# Contents

# 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 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 partition is one of the promising approaches in DPLL-like solvers [5]. By giving the order to the variables according to a good formula partition, the search gets a relatively balanced decision tree. But formula partition is rarely used in a local search for the SAT problem. How to combine the formula partition with local search, will the local search benefit from the partitioning, if the formula partitioning can guide a parallel local search, are still open questions.

## 1.2 Content

The SAT problem, as a well-known NP-complete problem, has received a great deal of attention and different local search heuristics have been developed. This paper is a survey on the stochastic local search on SAT problem with a guide of formula partition.
In section 1, we summarize the formal concept and introduces techniques used in this paper. One class of the most straightforward but efficient stochastic local search algorithms *Probsat* is the algorithm basic in our paper. *Probsat* was proposed in 2012 by Adrian Balint and Uwe Schoening [6]. Section 2 describes our *Probsat* algorithm and discusses our attempts to improve the original algorithm. By experimentally evaluation and comparison, some techniques turned out to be more efficient than the simple *Probsat* search. With the partition of variables and its corresponding formulas, the problem can be separated into two subproblems of similar size. In section **??**, we search the potential benefit of formula partition in a parallel search. Section 3 describes the details in experiments and several empiric results mentioned in section 2 and section 3. Section 4 concludes the paper with further works.

## 1.3 Definitions and Notations

***Propositional Satisfiability Problem***
A variable with two possible logical values $TRUE$ or *False* is a ***propositional variable***, which will be referred to as ***variable*** in this paper. 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 paper. An ***assignment*** a as a function $a$: $V \to \{True, False\}$ assigns the truth value to each variable $v$ in the formula. We say the assignment satisfies a formula if the truth value of the formula with this assignment turns out to be true. Specifically, an assignment satisfies a clause, if one literal in the clause with value *True* in this assignment. A formula is a satisfying formula if one assignment exists satisfies all its clauses. We say an assignment a satisfying assignment if it satisfies the formula. Otherwise, we say there are conflicts in some clauses

with this assignment, or some clauses are unsatisfying clauses with this assignment. The SAT problem is to determine whether a satisfying assignment exists for the given formula. If so, we denote the formula a ***satisfiable formula***.

Here is an example of SAT problem:

$$F = (v_1 \vee \bar{v_3}) \wedge (v_2 \vee v_1 \vee \bar{v_1})$$
$$Vars(F) = \{v_1, v_2, v_3\}$$
$$numVs(F) = |Vars(F)| = 3$$
$$Lits(F) = \{v_1, \bar{v_1}, v_2, v_3, \bar{v_3}\}$$
$$Cls(F) = \{C_1, C_2\}$$
$$numCs(F) = |Cls| = 2$$
$$C_1 = \{v_1, \bar{v_3}\}$$
$$C_2 = \{v_2, v_3, \bar{v_1}\}$$

$A(v_1) = True$, $A(v_2) = False$, $A(v_3) = True$,
$A$ is an assignment satisfies $F$.

$\hat{A}(v_1) = True$, $\hat{A}(v_2) = False$, $\hat{A}(v_3) = False$,
$\hat{A}$ 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 instance $I$ of a hard combinational Problem $P$, there is a set of solutions $S(I)$. According to the constraints of the problem, an object function (score or cost) $\Gamma$ is used to evaluate the candidate solutions. The Goal of the local search is to find the solution of minimum cost ( or the solution with the 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***. Starts from an initial solution, the search will evaluate the solutions which can be 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 [13] and the graph coloring problem [14].

***Local Search in SAT Problem***
In the Boolean satisfiability problem, a local search operates primarily as follows: The search start from a randomly generated assignment as the initial solution. If this current assignment satisfied the formula, the search stops with success. Otherwise, a variable is chosen depends on some criterion. This selection is called ***pickVar***. By change the assignment of the selected variable $v$, a neighbor assignment of our current solution $A$ is reached in next step, which is also called ***flipp***$(A, v)$. A local search will move in the space of the assignments by making the variable flipping until a satisfying assignment is reached by the search.
The heuristic used for the flipping variable selection $pickVar$ is based on some 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 ***breakcout*** of the local move from $A$ to $\hat{A}$. Accordingly, the number of clauses, which become satisfying because of the flipping, is the ***makecount***. The number of newly satisfying clauses (*makecount*) minus the number of newly unsatisfying clauses (*breakcount*), which is denoted as ***diffscore***, represents the local improvement of the corresponding flipping. Apart from this, other aspects like the repetition number of one flip or the number of occurrences of the variables can be considered in a selection heuristic. An example is the unit propagation embedded local solver *EagleUp*, which prefers flipping of variables with the highest number of occurrences in the formula to create new unit clauses sooner. To get local improvement effectively, man can only consider variables in unsat clauses for the flipping selection. This process is called a ***focused local search*** and commonly used.

**Algorithm 1:** Focused Local Search

`input` : A CNF Formula F
`parameter`: *Timeout*
`output` : a satisfying assignment $A$

$A \leftarrow$ random generated assignment $A$;
`while` ($\exists$ *unsatisfied clause* $\wedge$ *Timeout does not occur*) `do`
    $c \leftarrow$ random selected unsatisfied clause ;
    $x \leftarrow pickVar(A, c)$
    $A \leftarrow flip(A, x)$;

By choosing the variable with best score in $pickVal$, the seach will get greedy local improvemnet. The initial hope of the local search is that through iterative greedy local improvement the optimal global solution can be found. The typical problem of the local search is that the greedy local searches be trapped in local unattractive local optimal solution. To avoid this , some random flips are picked or even a worse solution will be chosen for the next step (***uphill moves***). There are some techniques following used in local search to avoid getting stuck in local optimum.

***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 advantage a move is, the probability of choosing it as the next step is higher. This randomization will avoid the stuck of the search in a local minimum and decrease the misguiding of the heuristic in specific situations.

***Statistical Local Search***
Tabu search is created by Fred W. Glover in 1986 [15] and formalized in 1989. For recognize the loop in a suboptimal region, the search trace is recorded in the process by mark the recently reached neighboring assignments as tabu. The 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 trach, in which the times each variable is chosen for flipping are counted. By using the stochastic information, the search will prefer the variables with fewer flippings before. By Experiments, a local statistic search can recognize besides short-term cycling like tabu search and permit long-term cyclings [7].

***Simulated Annealing***
Simulated Annealing is an approach of local search solver to difficult combinational optimization problems proposed by Kirkpatrick, Gelatt, and Vecchi [8]. This approach is inspired by the metallic process annealing of shaping the material by heating and then slowly cooling the material. This approach works as a local optimization algorithm guided by a controlling parameter ***temperature***. By high temperature, an uphill move is allowed with high probability while only small steps are allowed in low temperature. The temperature is varying according to the score of the current situation. For a current solution with a nearly optimal score, the temperature is near zero. For an unattractive local extreme with a poor score, the active search is tending to make uphill moves in high temperature.

***WalkSAT***
*WalkSAT* is a focused random local search strategy to solve SAT problem, which is originally introduces in 1994 [9]. *WalkSAT* may ignore the greedy flipping and flip a random variable in chosen unsatisfied clause wit 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 for flipping
for v in c do
    Evaluate v with function Γ(A, v);
with probability p: x ← v with maximum Γ(A, v);
with probability 1 − p: x ← randomly selected v in c.
```

***The Probsat***
Probsat is a class of SLS sat solver, which was introduced in 2012 by Adrian Balint and Uwe Schoening [6]. In a probsat solver, the score of a candidate flip is solely based on the make and break score. The paradigm is as follows: At first, a completely random assignment is set as the initial assignment. The algorithm performs local moves by flip a variable in a random chosen unsatisfying clause and stops as soon as there are no unsatisfied clauses exists, which means a satisfying assignment is found. The probability $p(v)$ of flipping the variable $v$ in the chosen clause proportina to the score of $v$, which is calculated in a function $\Gamma(v, A)$ based on break score of $v$ in the current assignment $A$ [1] The idea behind this selection heuristic is to give the advantageous flipping relative high score, but the other flipping with small score has chance to be chosen. There are two kinds of score functions are considered in the paper of Adrian Balint:

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

[1] As mentioned in the probsat paper, it turns out in experiments that the influence of make is rather weak in selection functions, so the one-parameter functions depends on *breakScore* can also lead to an efficient algorithm.

The pseudo code of a typical Probsat is shown below:

**Algorithm 3:** pickVar in probSAT

`input` : current assignment $A$, unsatisfied clause $c$
`output` : a variable $x$ in $c$ for flipping

`for` $v$ *in* $c$ `do`
  Evaluate $v$ with function $\Gamma(A, v)$;
$x \leftarrow$ randomly selected variable $v$ in $c$ with probability $p(v) = \frac{\Gamma(A,v)}{\sum_{u \in c} \Gamma(A,u)}$;

***Formula partition***

To introduce our parallel SAT solver with formula partitioning, we use a hypergraph representation of SAT problem.

A hypergraph $G = (V, H)$ is a generalized graph, in which an 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 $numVs$ vertices and $numCs$ 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:

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

$G(F)$:

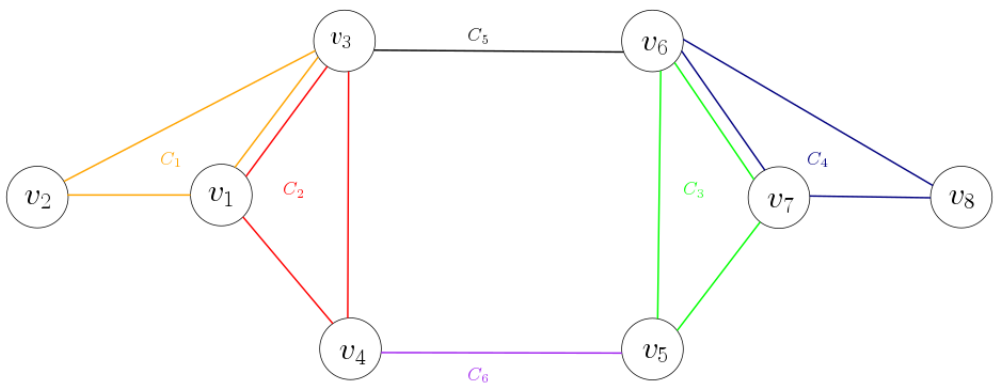



Figure 1: In a hypergraph, a hyperedge is a set of vertices. In our example, the vertices $v_3$ and $v_4$ are both in hyperedge $\{v_1, v_3, v_4\}$ and $\{v_1, v_2, v_3\}$, so they are connected twice. In our hypergraph representation of SAT problem, a hyperedge contains the vertices of the corresponding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

Formula partition is a promising way to improve the SAT problems solving. Two partitioning are investigated in some works. One is to split the variables, which is used in this paper, and another one is to separate the clauses. For the algorithms with the technique of decision tree like *DPLL*, the formula partition can guide the order of decisions. For the local search, there is nearly research of the use of formula partition in local search. Based on this hypergraph represenation, we describe in this paper the formula partition with the notations in graph partition. For a hypergraph $G = (V, H)$, a (2-way) partitioning is to seperate the verices set in two disjoint subsets. A good partition is to seperate $V$ in $P_0$ and $P_1$, which are of relative same size and only few hyperedges containing both verticse in $P_0$ and vertices in $P_1$. Based on a vertices partitioning, the hyperedges are seperated in three disjoint subsets $H_0$, $H_1$ and $I$. $H1$ 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 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 random SAT solver used for a comparison with our algorithm is *yalSAT*, which is the champion in random track category in SAT competition 2017 [10].

***probSAT*** [2]
The authors of the original Paper implement the ProbSAT. We compare our Solver with this original code [3].
In this original code, there are two implementation variants available. In the incremental approach, the *breakScores* of variables are calculated in the initialization phase and only updated in the further search. The other straightforward approach is to compute breakScores of the variables in consideration of flipping. This method is called non-incremental approach in original paper. As suggested in Experiments, we take the non-incremental approach for the 3SAT problems and incremental method for 5SAT and 7SAT to get optimal results of the probSAT solver.
The parameters of ProbSAT in our Experiments have been set as suggested in the original paper:

| $k$SAT[a] | score $\Gamma$ | $c_b$ | $\epsilon$ | 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 setting for competitor probSAT

[a] k is the maximum length of the clause

***yalSAT*** [4]
We use the version 03 submitted to the 2017 SAT competition of the yalSAT solver in our experiments. Armin Biere implements it as a reimplementation with extensions of probSAT. With the implementation of different variants of probSAT, the yalSAT uses a different variant of probSAT randomly in the restart of a round of search. In our comparison, we use the default settings of the yalSAT with specific seeds (See 3.4).

[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 shema:

**Algorithm 4:** Our Local Search

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

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

## 2.1 initAssign(F)

In our algorithm, we have three variants to make assignment initialization. One is the *RandomInit* which is the random initiation like in the original *probSAT* suggests. Two alternatives to this random assignment are with the consideration of number of literal occurrences. with the method *BiasInit* we assign *True* to a variable if the number of occurrences of its positive literal is more than its negative literal. Otherwise, a variable is assigned initially with *False*. *Bias-RandomInit* combines the two initializations above, in which the assignment is generated bias randomly based on the occurrences of literals. In Experiment 1 in Section 4 (see 3.7.1) we compare these three alternatives based on the probsat algorithm. Our local search uses *RandomInit* for 3SAT problems and *BiasInit* for 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. The unsatisfying clauses will be cached in a set *UNSAT*. During the local flipping, these numbers will be updated when the flipping variable is in the clauses (See 2.5). Comparing to the *numT*, the *UNSAT* is updated “lazily”. After Flipping, if the *numT* of one clause is decreased to zero, it will be added in the *UNSAT*. To select an unsat clause in *pickCla*$(A)$, man needs to select a clause from the *UNSAT* and check if it is still unsatisfied with its *numT* is zero. Otherwise, if the chosen clause $c$ with $numT(c)$ as zero, it will be removed from the *UNSAT* set. This step pickCla(A) will be repeated until one unsatisfied clause is found or the *UNSAT* set is empty, which means the current Assignment $A$ is a satisfying assignment.

## 2.3 pickVar(A,c)

Inspired by *probSAT* and *walkSAT*, Our *pickVar* combines the random walk and stochastic selection. In observation of the experiments of the *probSAT*, this stochastic search bases its selection on a random heuristic. Even the search is very close to a satisfying assignment, and the probability of the critical flipping is exceptionally high, it is possible that the stochastic search make uphill moves and leave then the region of the global minimum. To prevent this besides the stochastic way, we pick greedy flip with zero *breakScore* with a certain probability $p$. With probability $1-p$, we choose the variable for flipping using the *probSAT* heristic.

**Algorithm 5:** Our pickVar

`input` : current assignment $A$, unsatisfied clause $c$
`parameter`: probability $p$
`output` : a variable $x$ in $c$ for flipping

greedyVs $\leftarrow \emptyset$;
**for** *all v in c* **do**
    **if** (*break(A,v)= 0* ) **then**
        greedyVs = greedVs + $\{v\}$
with probability $p$: $x \leftarrow$ randomly selected variable $v \in$ greedyVs;
with probability $1-p$: $x \leftarrow$ randomly selected variable $v$ in $c$ with probability $\frac{\Gamma(A,v)}{\sum_{u \in c} \Gamma(A,u)}$;

We analyze experimentally the following variants for *pickVar*.

***1. Varinat: WALK***

Instead of using a constant probability $p$ to choose between a greedy Literal without clause break and the random Literal using *probSAT* flipping directly, we see a list, called statistic list $S$ to record how many times each variable is chosen for flipping. To avoid cycling, we see the variable $v_i$ with a high value of $S[i]$ to be disadvantages for flipping. After selecting 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

greedyVs $\leftarrow \emptyset$;
**for** *all v in c* **do**
    **if** (*break(A,v)= 0* $\wedge$ $Permit(v)$) **then**
        greedyVs = greedVs + $\{v\}$
*greedyV* $\leftarrow$ randomly selected variable $v \in$ greedyVs ;
*randomV* $\leftarrow$ randomly selected variable $v$ in $c$ with probability $\frac{\Gamma(A,v)}{\sum_{u \in c} \Gamma(A,u)}$;
with probability $p = \alpha \times \frac{s(greedyV)}{s(greedyV)+s(randomV)}$: $x \leftarrow$ randomV;
with probability $1-p$: $x \leftarrow$ greedyV;

***3. Varinat: GreedyBreak***

Compared to find the greedy literals with zero breakScores, the calculation of the decay function $\Gamma$ values and get a random literal according to its distribution takes the most part of runtime in the whole search. In this variant *greedyBreak*, we search greedy Literal with small statistic value. Here, we define a literal is a *permitted greedy literal* if its break value is zero and its statistic value is under some limit. If *permitted greedy literals* exist, we choose one randomly for flipping. Otherweise, we pick random Literal using probSAT heuristic.
To set the limit based on the search history, we compare two functions in our experiment. In the first approach *Average*, the limit is set statistic to $\alpha \times \frac{numFs}{numVs}$. Here the numFs is for the number of flips in search. In another approach *Random-Flip*, we select randomly a value $r$ in [0, *numFs*]. For each greedy Literal, we check if its statistic value is smaller than $\alpha \times r$.

**Algorithm 7:** TieBreak

`input` : current assignment $A$, unsatisfied clause $c$
`parameter`: probability $p$
`output` : a variable $x$ in $c$ for flipping

greedyVs $\leftarrow \emptyset$;
`for` *all* $v$ *in* $c$ `do`
    `if` (*break(A,v)= 0* $\wedge$ $Permit(v)$) `then`
        greedyVs = greedVs + $\{v\}$
`if` (*greedyVs is not empty*) `then`
    x $\leftarrow$ selected variable $v$ in $greedyVs$ at random
`else`
    x $\leftarrow$ randomly selected variable $v$ in $c$ with probability $\frac{\Gamma(A,v)}{\sum_{u \in c} \Gamma(A,u)}$;

## 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 $\alpha$ as a function depending on two parameters tolerence $\tau$, $c_b$ and the quality of current assignment $q$ (the *temperature*) instead of using a static parameter in the whole process:

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

To define the $q$, we have two variants: *Global* and *Local*.

**Global**:
In the process of search, the number of unsatisfied clauses is shown in $unsatN$. In this traditional variant *Global*, we use this number to define the quality of the current solution:

$$q_{Global}(A) = unsatN(A).$$

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

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

## 2.5 Data structures

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

***Literals***
Local search is a search where only small changes are made in each step. In our situation, only the clauses include the flipping variables are involved in the flipping step. To find the involving clauses of one variable, two $2D$ array $posL$ and $negL$ is made to record the clauses of positive nad negative literals. For variable $v_i$, the posL[i] record the indexes of clauses containing the positive literal $v_i$. The ones with negative literal $\bar{v_i}$ are in $negL[i]$. To implement the flipping of the variable $v_i$, we update the $numTs$ of clauses with indexes in $posL[i]$ and $negL[i]$.

***LookUp***
The most time used in the search is the repeated calculation of the polynomial or exponential decay function $\Gamma$. With this observation in our experiments, we calculate the $\Gamma(x)$ with $x$ from 0 to $0.5 \times numCs$ and keep the values in a list *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 includes the boolean assignment and three other structures to record information about the current assignment. The *solution* is computed after assignment initialization and is updated during each flipping :

| Name | Structure | Size | Meaning |
|---|---|---|---|
| $assignment$ | list | numVs | boolean assignment to variables |
| $numTs$ | list | numCs | number of $True$ values in each clause |
| $numUnsat$ | natural number | - | the number of unsatisfied clauses |
| $UNSAT$ | set | - | indexes of unsatisfied clauses [a] |

Table 2: Four structures in a solution

[a]This UNSAT is updated in flipping phase lazily by only adding new unsatisfied clauses and remove the clause chosen in $pickCla$.

# 3 Evaluation

## 3.1 DIMACS standard format

All the benchmark formula used in experiments are encoded in the DIMACS standard format [11]. 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 programs:

**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 $nV$ variables and $nC$ clauses ,the problem line in its DIMACS file is:

**p** cnf $nV$ $nC$

3. Clause descriptor: An clause $\{v_1, v_2, , v_n\}$ in the formula is described in an clause Descriptor:

**e** $v_1$ $v_2$ $v_n$

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

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

## 3.2 Benchmarks

The benchmark instances used in experiments are the 180 uniform instances ( *UNIF*) in random benchmark categories in SAT competition 2017 [11]. In an *UNIF* problem file, all the clause have the same length.
In the name of an *UNIF* file, the suffix $k$ denotes the length of clauses. The $r$ indicates the clause-to-variable ratio. The $c$ and $v$ are for the number of clauses and variables, while $s$ is for the seed used in the generation process. Without flitering, there are at least 60 ( 33% ) problems form our 180 benchmark collections are unsatisfiable.

## 3.3 Used plots and tables

the results of the following experiments are all shown in comparision tables and illusatrated in cactus plots.

***Comparison Table***
See Table 9 for example.
A comparison table shows the different results of algorithms. The first column contains the clause length $k$. The *UNIF* benchmark has 3 different sizes: 3SAT, 5SAT, and 7SAT. For

a $k$SAT instance, each clause contains $k$ variables. The fields of a comparison table in the following columns corresponds to a penalized runtime for the whole $k$SAT set, which assigns a runtime of two times the time limit for an unsolved instance. Because the *UNIF* benchmark problems without filtering 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. There are 61 3SAT instances, 89 5SAT instances and 66 7SAT instances found satisfied in our experiments. Besides the runtime score, the number of solved problems are in parentheses. The best results in comparison are in bold.

***Cactus Plot***
See Figure 18 for an example.
A cactus plot shows the performance of different algorithms. The y-axis shows the time in second used to solve the benchmark graphs. The x-axis is for the number of solved problems by a certain time. Each algorithm corresponds to a curve in different colors. The point $(u, v)$ on a curve means by $v$ seconds the corresponding algorithm have solved $u$ problems.

## 3.4 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 in experiments 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 the seeds used later in our experiments.

| solver | name | 1.seed | 2.seed | 3.seed |
|---|---|---|---|---|
| probSAT | probsat | 1988822874 | 338954226 | 858910419 |
| yalSAT | yalsat | 1851831967 | 280788293 | 1956345180 |
| our local solver | local | 1962042455 | 1112841915 | 566263966 |
| our parallel solver | parallel | 1749729997 | 68910537 | 473644167 |

Table 3: Parameter setting for our solvers and competitors

### 3.4.1 Soft- and Hardware

The single-threaded experiments were run on computers that had 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-threaded experiments were run on a computer that had Two Intel Xeon E5-2650 v2 processors (16 cores + 16 HT cores) with 128GB DDR3 RAM. The machine ran the 64-bit version of Ubuntu Devel.

## 3.5 Parameter Settings in Experiment

The $TimeOut$ is set to 5 Minutes in the experiments of local searches. For the probSAT selection heuristic, our local search uses the values for $c_b$ and $\epsilon$ suggested in the $probSAT$ paper. The parameter $\alpha$ and the $tolerence$ $\tau$ in variances with simulated annealing are generated with the help of the algorithm parameter optimization tool SMAC [29] (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 $\in [0, 10]$ with step 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* | $\alpha/\tau$ | *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* |

## 3.6 Benchmark Generation

To combine the formula partitioning and SAT local solver, we try to get a relatively balanced partition with small intersection for the hypergraphgraph representation of benchmark SAT problems. On *UNIF* benchmark, we try to get graph partitioning using some partitioning algorithms (KaHypar and hmetis). Since the uniform random generation of this benchmark, these problems are without 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 our *UNIF* benchmarks. To investigate the local search on graphs with proper partitioning, we generate our benchmark COMBINE using the *UNIF* benchmark instances. As the name *COMBINE* suggested, we combine two *UNIF* benchmark instances in one SAT problem and make for the new generated problem with two disjoint subproblems an intersection. To create a satisfiable formula, we build the intersection based on a pair of random chosen satisfying assignments of the two *UNIF* problems in combination. 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 solve the two partitioning sets individually using our local solver, the current assignment is the same or a similar 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 7SAT problems. Besides that, we consider five combinations between 3SAT and 5 SAT instances and three combination of 5SAT and 7SAT instances. We combine problem in similar vertices size, which corresponds to balanced partitioning in the structure. We also combine one massive problem with a small instance of an imbalanced partitioning. For the intersection generation part, we generate clauses with three vertices in different partitioning sets.
We first choose vertices of two partition sets for intersection generation randomly. Here we

[5] Because of high time consume in parameter optimation, we solely compare $\tau$ as a natural number form One to Ten.

consider also 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 the same. To control the size of the intersection, we also count the number of the clauses in an intersection, if the proposition of the intersection clauses in the clauses number in whole combined problems is upper a specified limit, the generation of intersection is stopped.

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

Table 4: COMBINE problems with small intersection ($\frac{numCIs}{numCs} > 1\%$)[6]

[6]numCIs: number of clauses in intersetcion

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

Table 5: COMBINE problems with small intersection ($\frac{numCIs}{numCs} < 1\%$)

## 3.7 Experiments

### 3.7.1 Experiment 1: initAssign(F)

Experiment 1 compares three strategies of initialization in our solver. In the original *pobSAT* algorithm, the first step *randomInit* builds a complete assignment randomly in the initialization phase.

The *biasInit* suggestion is assign 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 $\frac{numCs}{2^k}$. 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 *randomBiasInit*, the boolean value is assigned to variables based on bias randomly on literals occurrences. The probability to assign True to variable $v_i$ is $\frac{posOccurences[i]}{posOccurences[i]+negOccurences[i]}$. As a stochastic bias initialization, it can get an assignment with expectedly few unsatisfied clauses. With the robustness of a random process, the *randomBiasInit* permit local stucks.

In a local search algorithm, it is normal that a replacement of the current solution by a new initial solution after a limit of the number of tries. However, in our implementation, the current assignment will be changed locally with flippings of variables after the initialization. In the following table, we compare three variants in the original probSAT algorithms. Based on the results of this experiments, we will determine the way of initialization in our heuristic solver for the problems with different length.

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

Table 6: For 3SAT problems, the initializations get similar performances. According to the PAR2-score of the benchmark sets, the two bias initiliazatios show improvment with about 60% percentage reduction in penalized runtime for 5SAT problems. the *BiasInit* shows its Efficiency in 7SAT problems. The search is about 14% faster than the original *probSAT* algorithm with a (bias) randomly generated initilization.

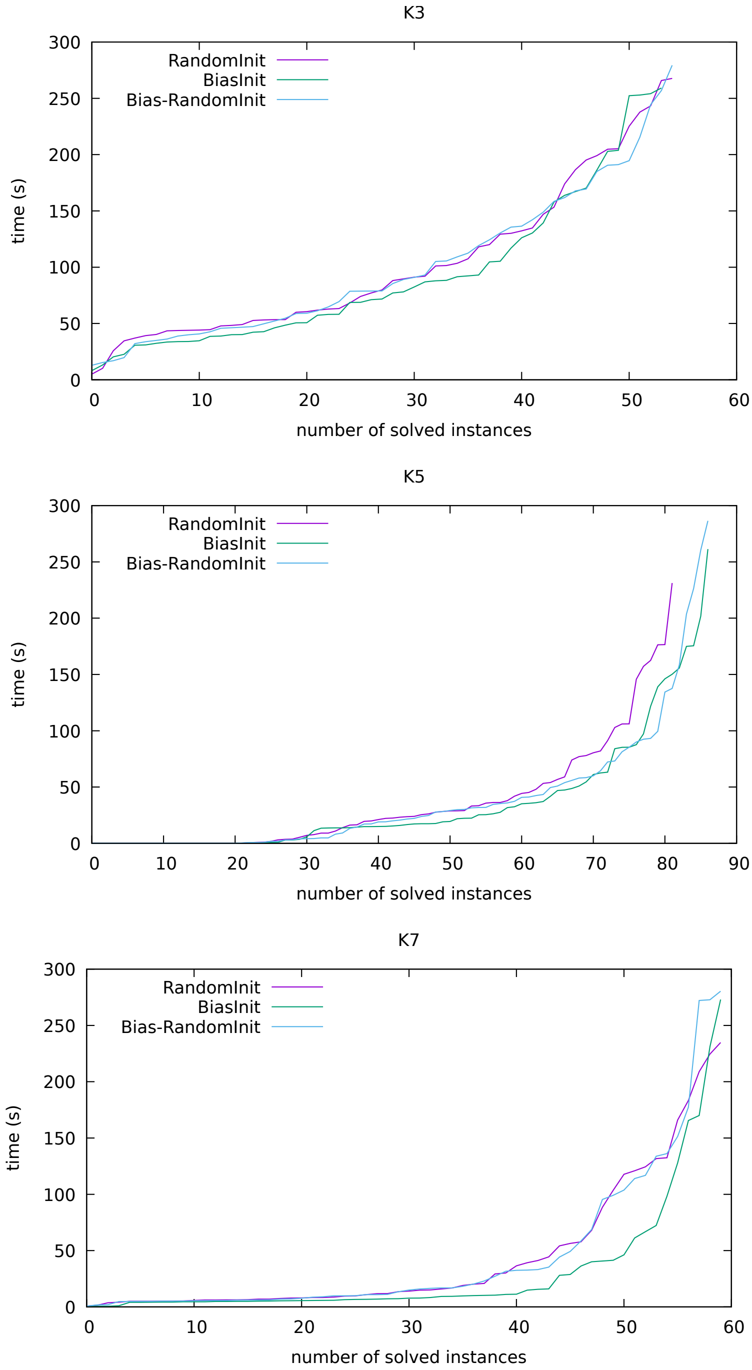



Figure 2: Three suggestions have very similar performance for 3SAT problems. In our solver, we use *RanomInit* for 3SAT as initiliazation method because of its simplify. Two bias suggestions show advantages especially for huge 5SAT instances. For 7SAT problems, the two random Initialization are similar in performance, while the bias initialization shows its efficiency.

### 3.7.2 Experiment 2: pickVar(F)

The probSAT algorithm paradigm introduced in section make pickVar as a randomized process To pick a variable for flipping in a chosen clause, we need to measure the score of all the literals in the classes and then model the probability distribution of the scores. In the observation of the implementation, this stochastic process is very time-consuming. The probSAT takes a lot of time in the first phase of the search and also go around with few conflicts in the last steps. However, in the first phase of the search, picking flippings greedy can lead the search quickly to the regions of interest.In our examples in the table 7. The probability to ignore the greedy literal is 30% for the 3SAT example and more than 50% for 5SAT and 7SAT. Then in the last steps of the search, where the search can reach the final satisfying assignment by a few deterministic steps, this probSAT algorithm may make "unintelligent", decisions with some probability and miss the critical steps.
To make the selection more greedily, we use a random walk to choose policy between greedy one and the probsat heuristic. In our algorithm, we refer the literals with no clause breaking as a greedy literal where greedy literal means the one with least break scores in original *WalkSAT*. The intention of not involving the literals with zero breaks in stochastic probSAT selection is the observation that the time-consuming probSAT will choose the non-breaking literal with high probability (see Table 7). To speed up this selection process and increase the randomness of the process, we just picked one greedy literal randomly as the greedy candidate for flipping. To avoid cycling in one region, we record the status information in the process. If the flips of chosen greedy literal have not been repeated many times, the selection will prefer another variable with a little breakscore, which has not been flipped so many times.

| k | Breaks | 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 7: In this table, we list three examples in 3SAT , 5SAT and 7SAT problem with probabilitys of the probSAT. Here we have in the chosen clause a greedy literal and other literals with breakscore 1. With the $\Gamma$ function with the recommended parameters in the original probSAT paper, we show in the 3.column the probabilities of each literal for flipping.

| 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 8: With the comparison of three Variant of *pickVar* with the staochastic selection in probSAT, our suggestions are faster and solve more instances in K3 and K5. For 3SAT, our suggestions have better performances. The $WALK$ and *Average* can solve more problems. The best one is the *Average* (with $\alpha = 1$), which solves 10% more problems. The *Average* shows advantages also for 5SAT problems, which can solve 8% more problems than the probSAT with only 46% time (in PAR-2 schema). For K7, there are no noticeable differences in results.

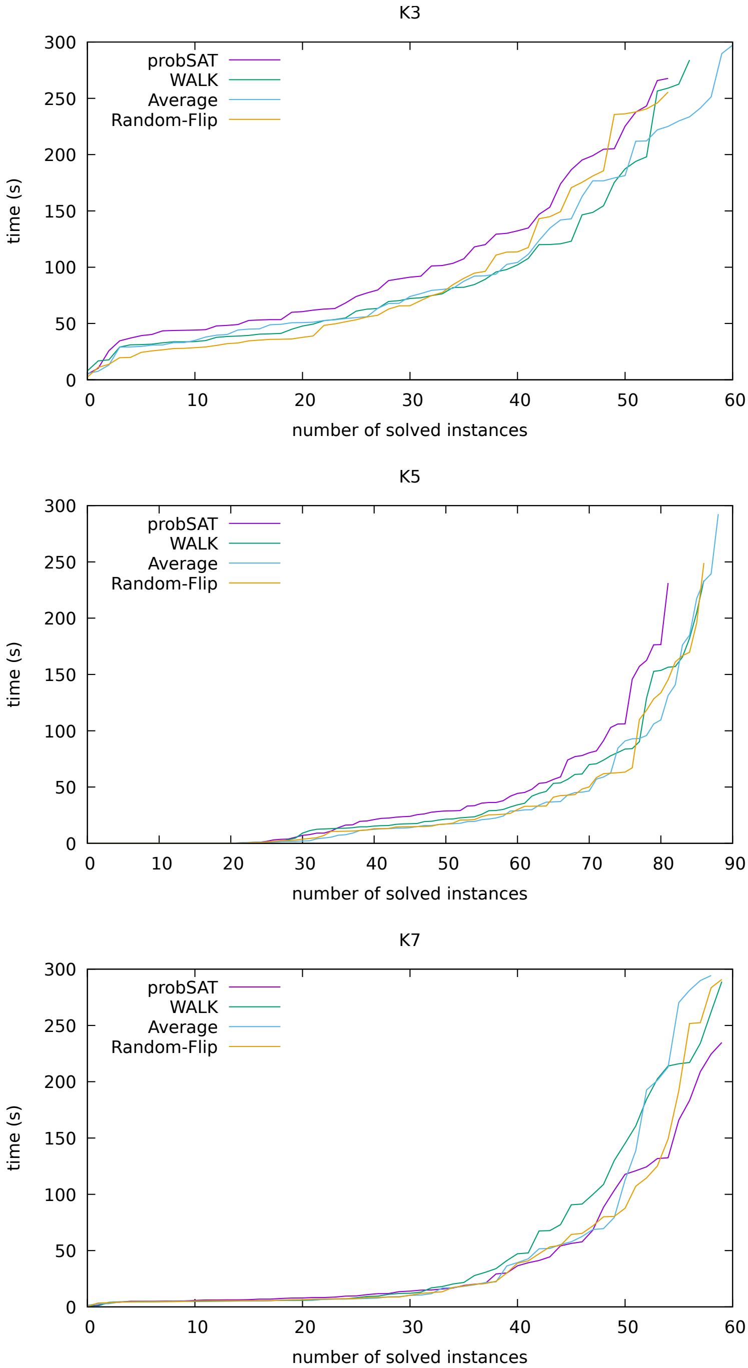



Figure 3: Our suggestions are faster than the probSAT algorithm in 3SAT problems. The *Averrage* shows an absolute improvement except for a few trivial small instances. For 5SAT problems,the improvement through our suggestions are big and stable. They solve more problems efficiently. For 7SAT problems,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

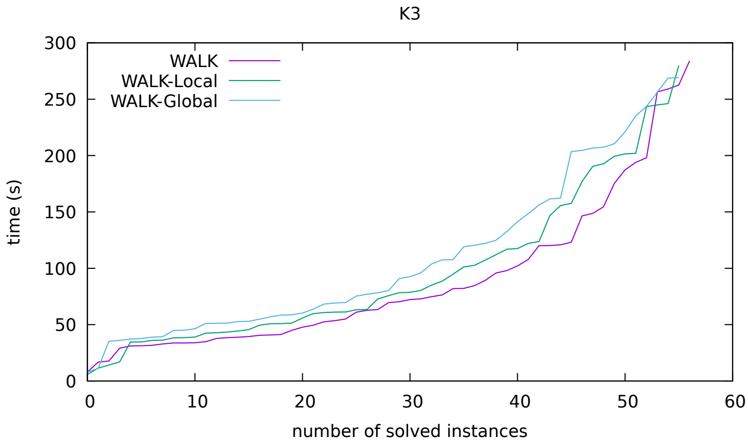
### 3.7.3 Experiment 3: Simulated Annealing

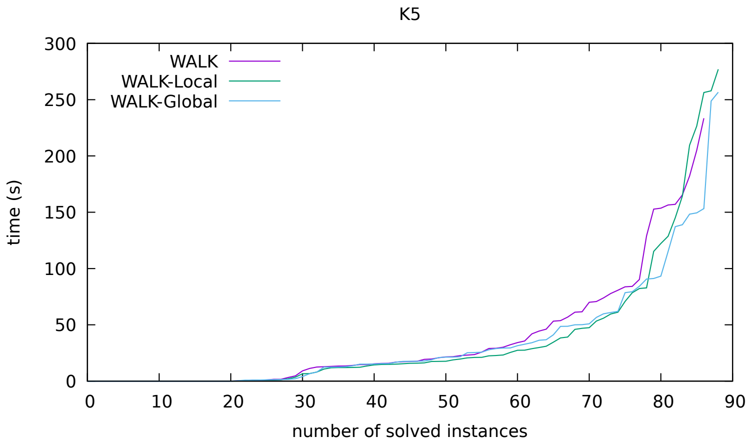
To use the technique *simulated annealing*, we need some quality to measure the quality of the current solution. One traditional way is to use the number of conflicts. Besides it, we propose the number of greedy literals in the chosen clause. Our hyperthesis behind this idea is that if the current solution is near a satisfying assignment without conflicts, it is likely to break a clause by a flipping. And in such case, there should not be only a few greedy literals exist in the chosen clause. Furthermore, this local quality is more specific about the chosen clause not generally about the whole search progress. In the following three experiments, we try to improve our three variants *WALK*, *Average* and *Random-Flip* with simulated annealing and compare these two definitions of qualities.
The combination with simulated Annealing show worse performance than the original *WALK* in 3SAT problems. they solve one less problems. The original *Average* can solve 15% more problems in 3SAT. For 5SAT, there are not noticabel differences in aspect of number of solved problems or the runtime. For 7SAT problems, the *Local* version can solve 6 problems more than the original *Average*. It has also shown its efficiency in solving huge 7SAT problems. The *Local* (with $\alpha = 2$) can solve 10% more problems than the *probSAT* within less than 60% time (in PAR-2 schema). Besides solving one less problem, the combination with simulated Annealing show also worse efficiency than the original *WALK* for 3SAT insatnces..
The performances of these three algorithms are guiet semiliar for small 3SAT problems within 50 seconds. The *Average* shows more adventages with the increasement of the problems size in 3SAT set. There are not big differences in *Average* with different combination of simulated annealing for 5SAT and 7SAT set while the variant *Local* has shown advantages in solving huge problems.

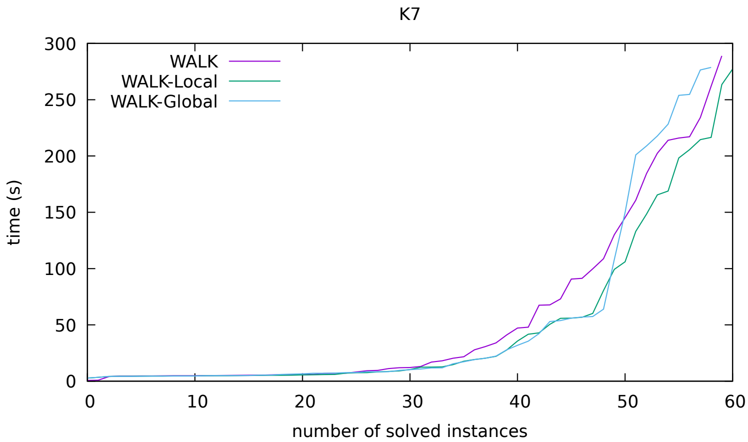
| k | *Average* | *AverageLocal* | *AverageGlobal* | *RF* | *RFLocal* | *RFGlobal* |
|---|---|---|---|---|---|---|
| 3 | 5.43% | 2.43% | 5.43% | 9.98% | 10.01% | 3.13% |
| 5 | 4.07% | 0.35% | 0.00% | 9.18% | 9.17% | 0.33% |
| 7 | 2.28% | 0.03% | 0.00% | 4.30% | 4.14% | 0.05% |

### 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) |



### 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) |

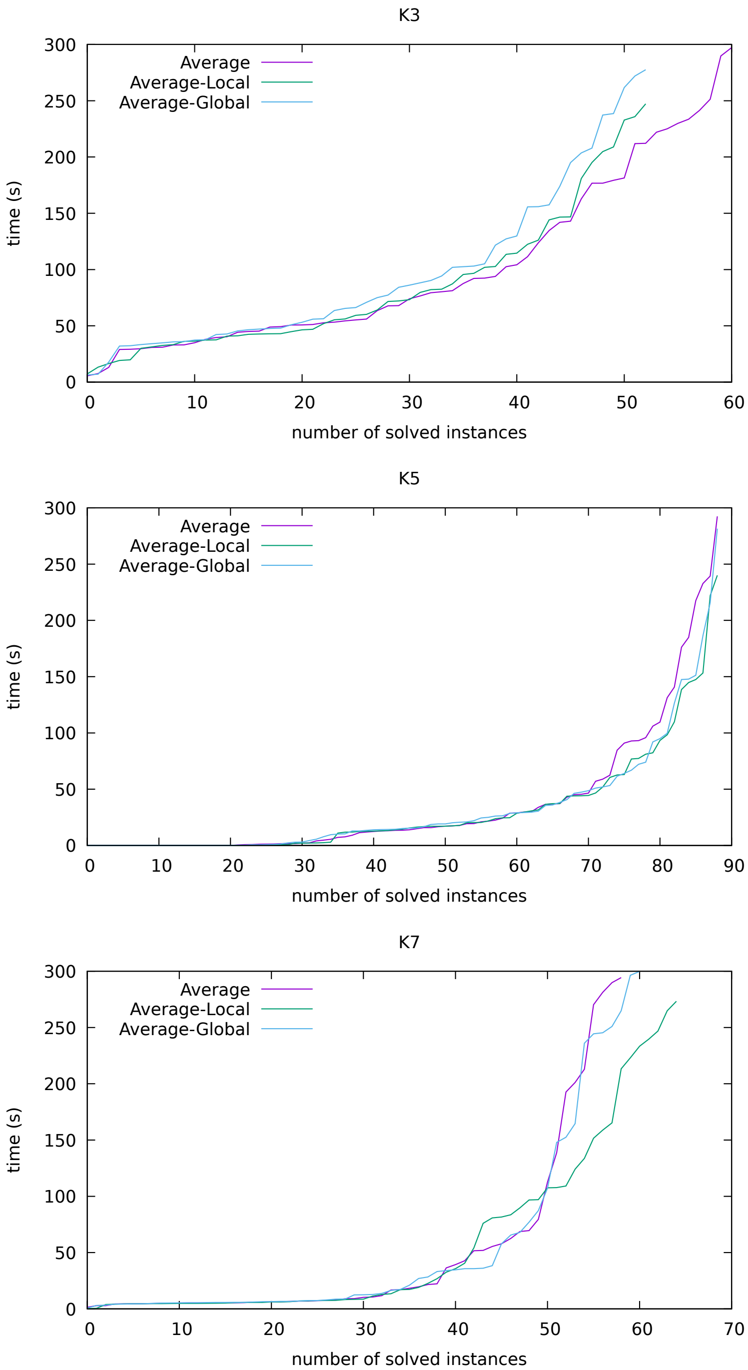
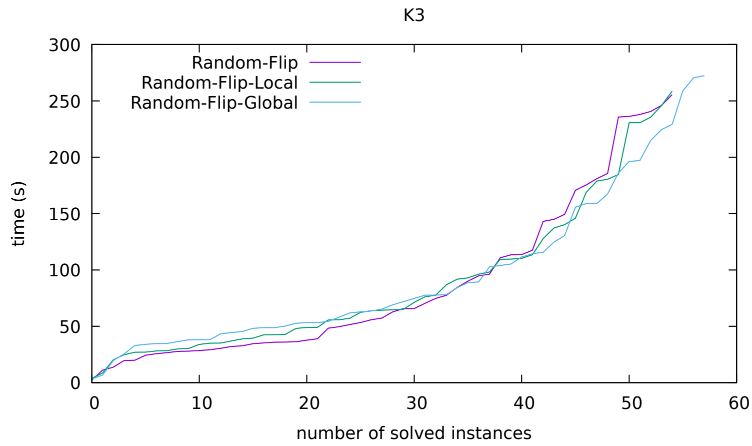
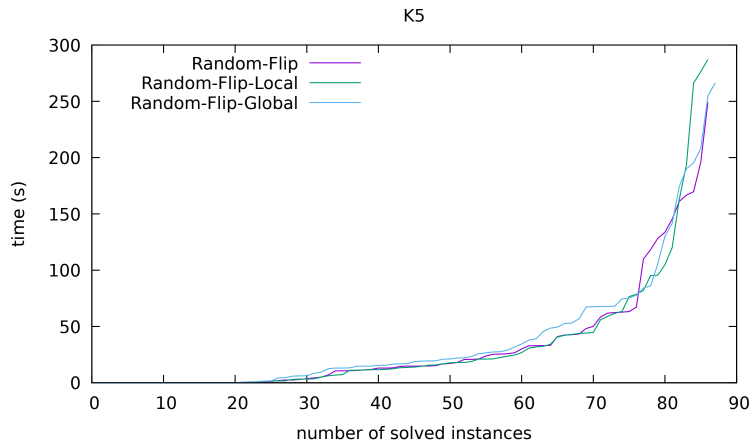


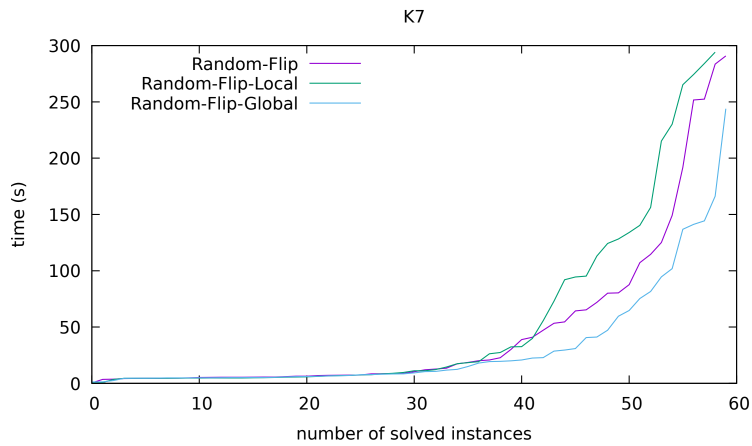

Figure 4:

### 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**) |



### 3.7.4 Experiment 6: probSAT vs WALK.

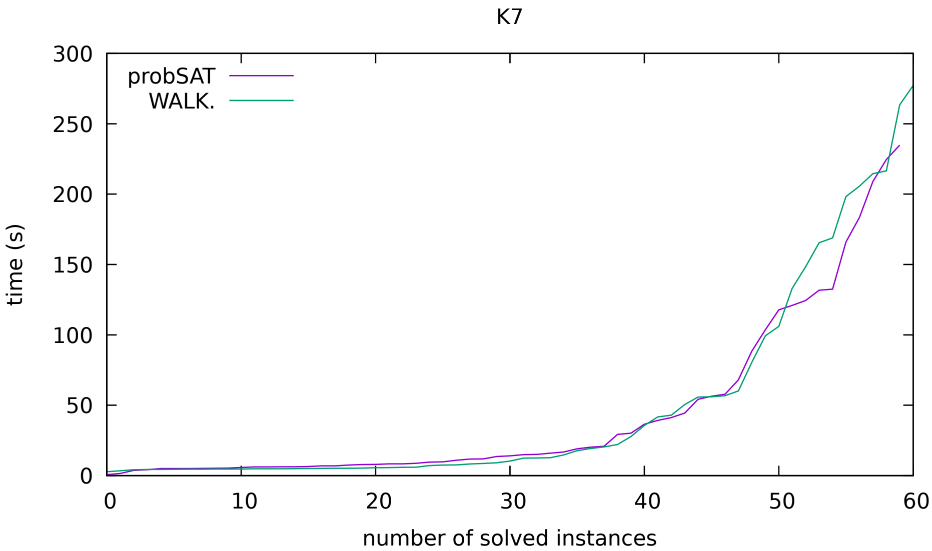



Figure 5: Three suggestions have very similar performance.

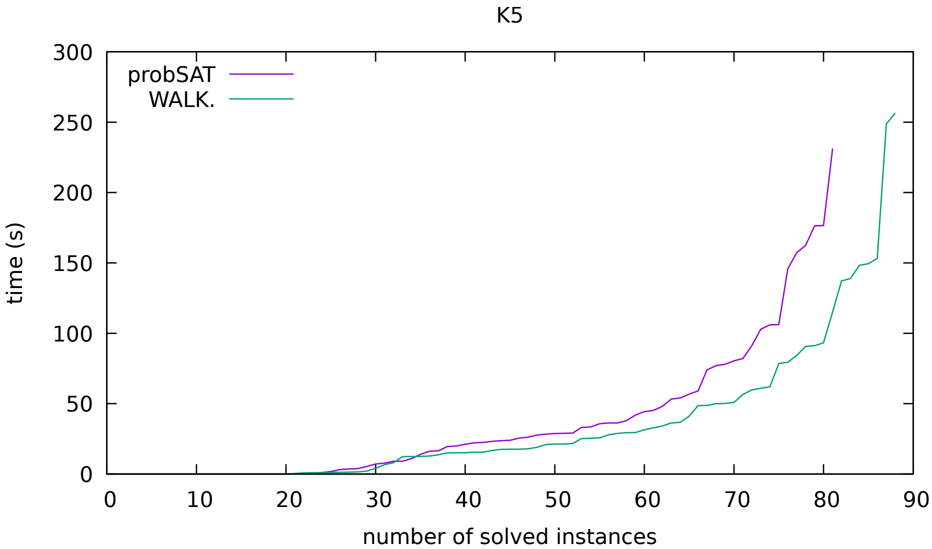



Figure 6: Two bias suggestions show advantages especially for huge instances.

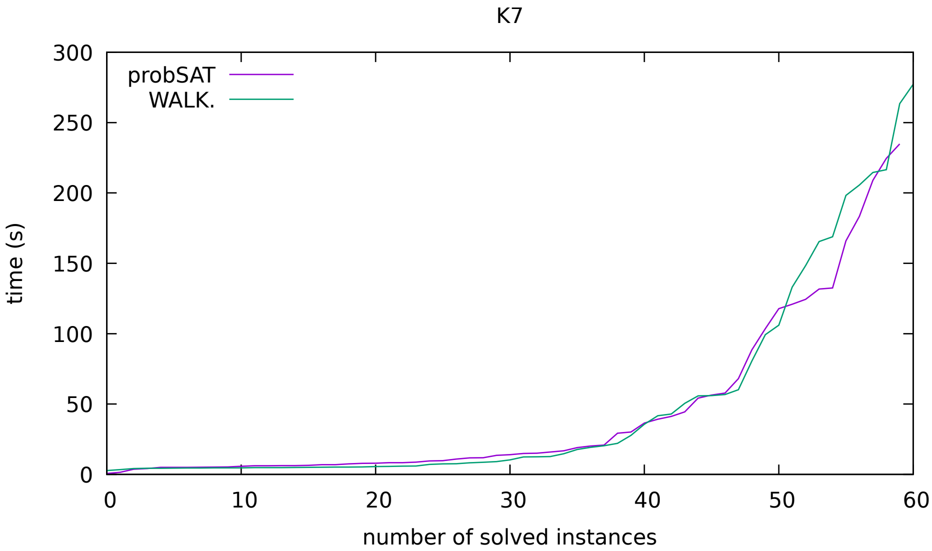



Figure 7: For 7SAT problems, the two random Initialization are similar in performance, while the bias initialization shows its efficiency.

### 3.7.5 Experiment 7: probSAT vs Average.

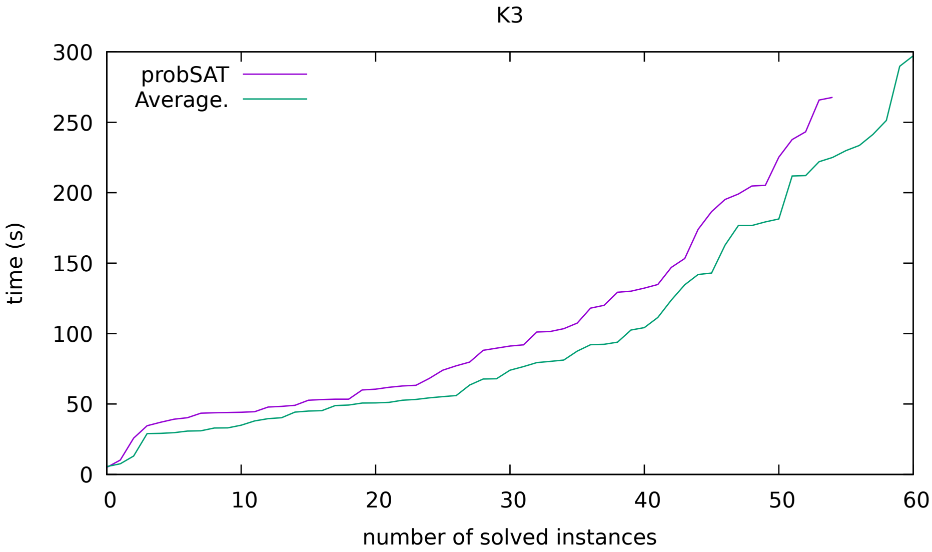



Figure 8: Three suggestions have very similar performance.

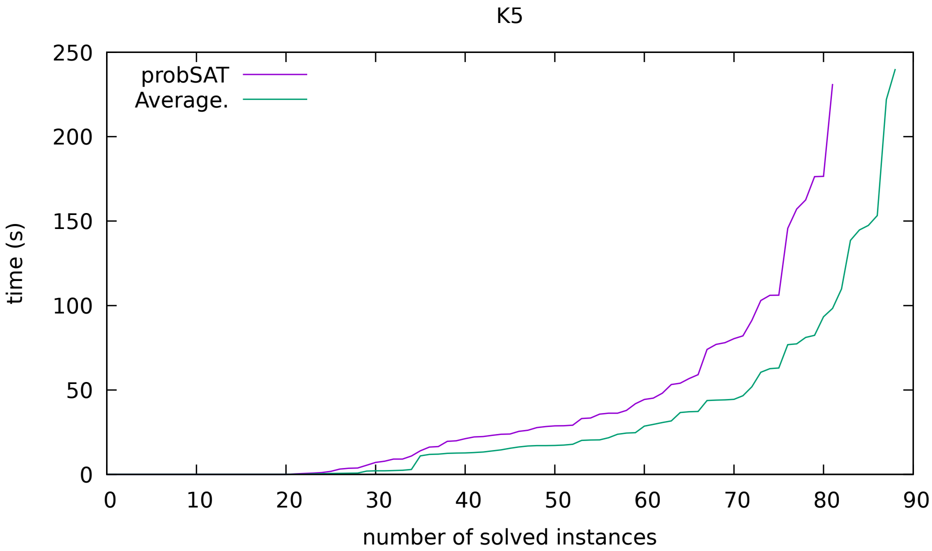



Figure 9: Two bias suggestions show advantages especially for huge instances.

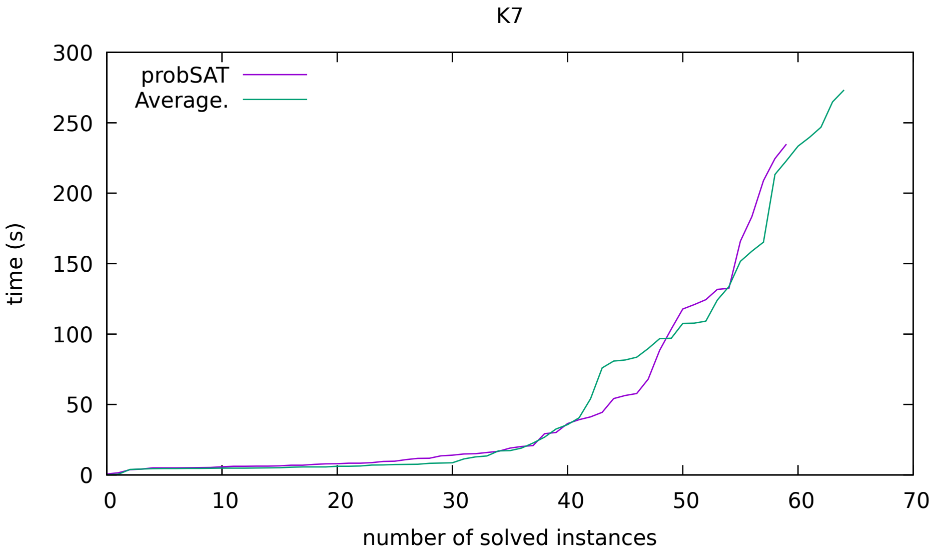



Figure 10: For 7SAT problems, the two random Initialization are similar in performance, while the bias initialization shows its efficiency.

### 3.7.6 Experiment 8:probSAT vs Random-Flip.

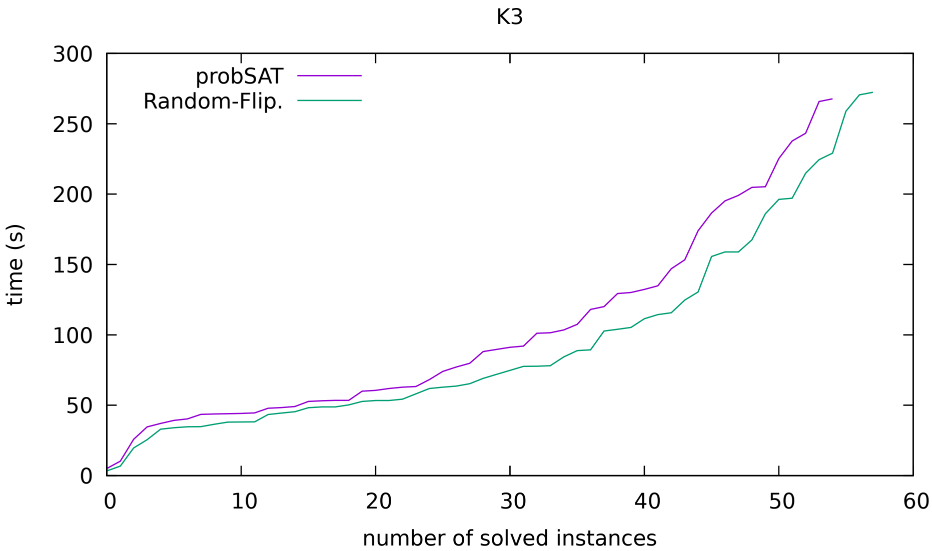



Figure 11: Three suggestions have very similar performance.

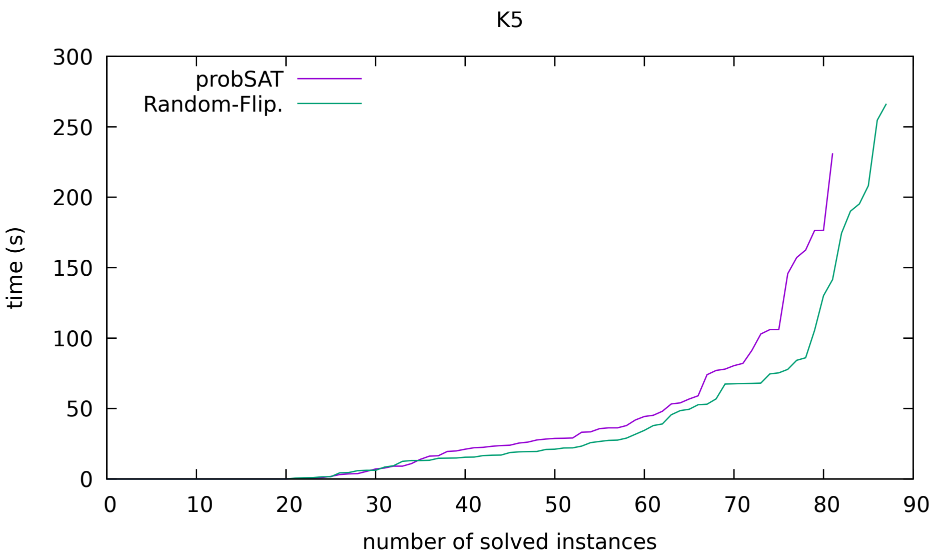



Figure 12: Two bias suggestions show advantages especially for huge instances.

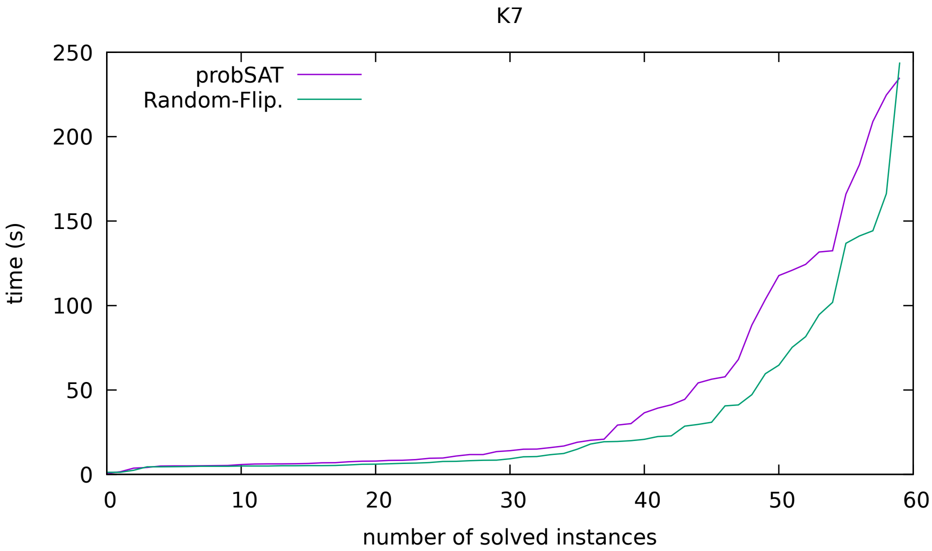



Figure 13: For 7SAT problems, the two random Initialization are similar in performance, while the bias initialization shows its efficiency.

### 3.7.7 Experiment 9: 2017-UNIF Comparision

| 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) |

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

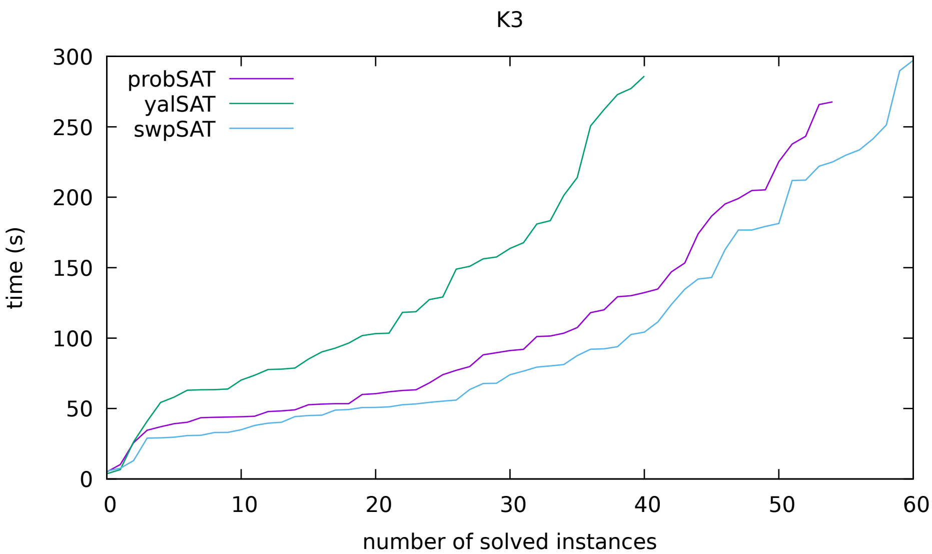



Figure 14: Three suggestions have very similar performance.

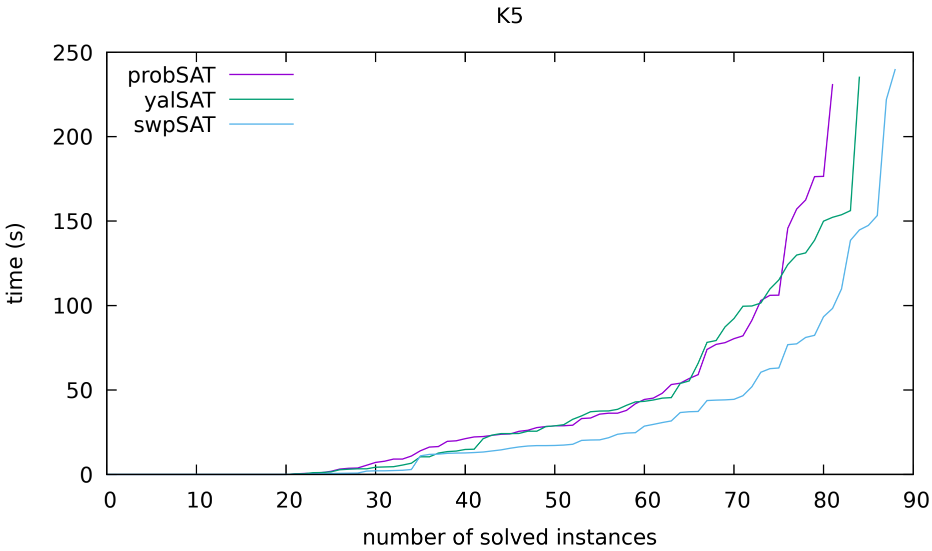



Figure 15: Two bias suggestions show advantages especially for huge instances.

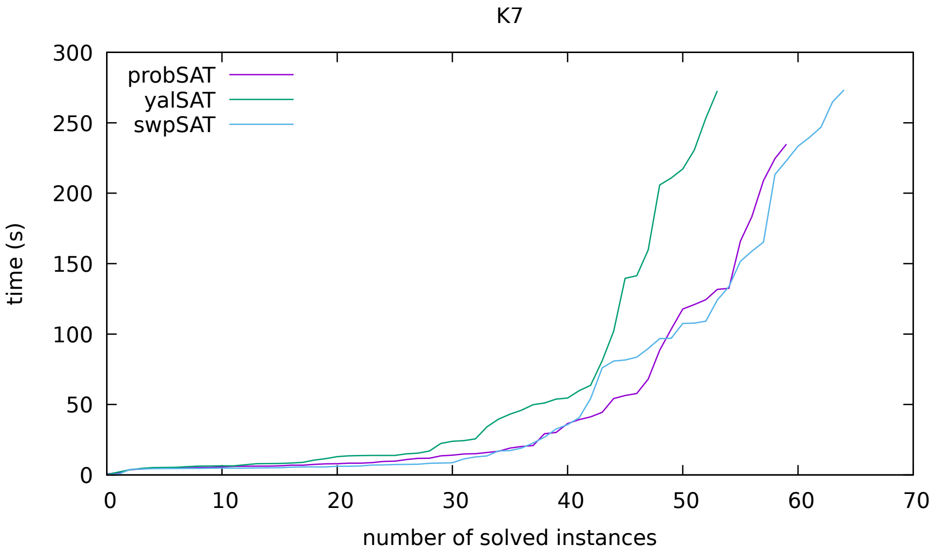



Figure 16: For 7SAT problems, the two random Initialization are similar in performance, while the bias initialization shows its efficiency.

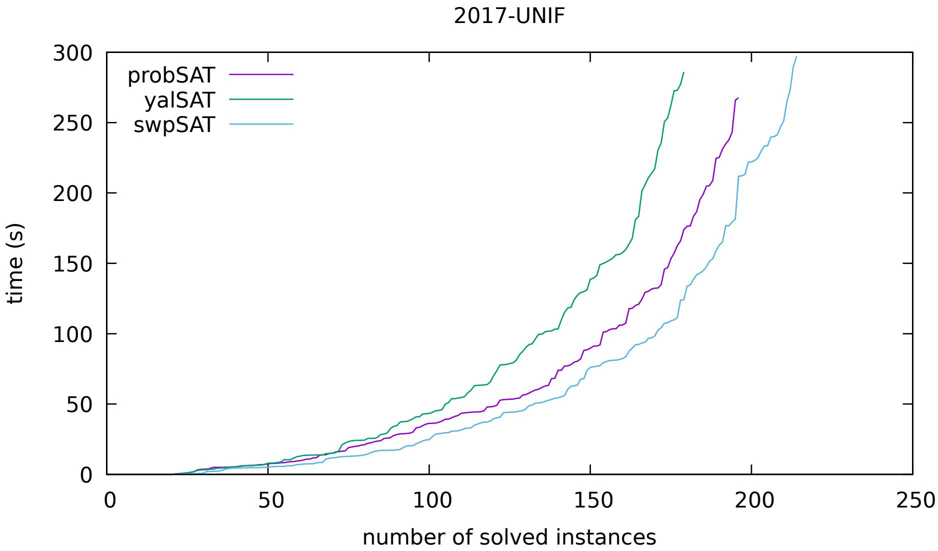



Figure 17: For 7SAT problems, the two random Initialization are similar in performance, while the bias initialization shows its efficiency.

### 3.7.8 Experiment 10: The pure portfolio approach

Our parallel implementation uses OpenMP to support shared memory multiprocessing. In our local *C++* Implementation, we use the function *rand*() in the standard library to generate pseudo-random integer. 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 for other threads. Based on the observation in our Experiments, the sequential implementation with the simple *rand*() function has the best performance for the whole set. For most problems, some random value generators can search following a valid search path to a satisfying assignment. But there is no one random generator that are advantages for all the problems. This approach takes advantage of the performances differences of random generator. The threads execute the swpSAT with different random engine in parallel. If one thread find a satisfying assignment,the whole parallel search can stop. With our experiment, the parallel search gets a performance like the minimum runtime of one-thread local search with different random value engines.

| k | *local swpSAT* | *pure portfolio* |
|---|---|---|
| 3 | 11171.3<br>59 | **5626.9**<br>**69** |
| 5 | 5339.98<br>89 | **2185.75**<br>**94** |
| 7 | 13406.26<br>66 | **2853.81**<br>**83** |

Table 9:

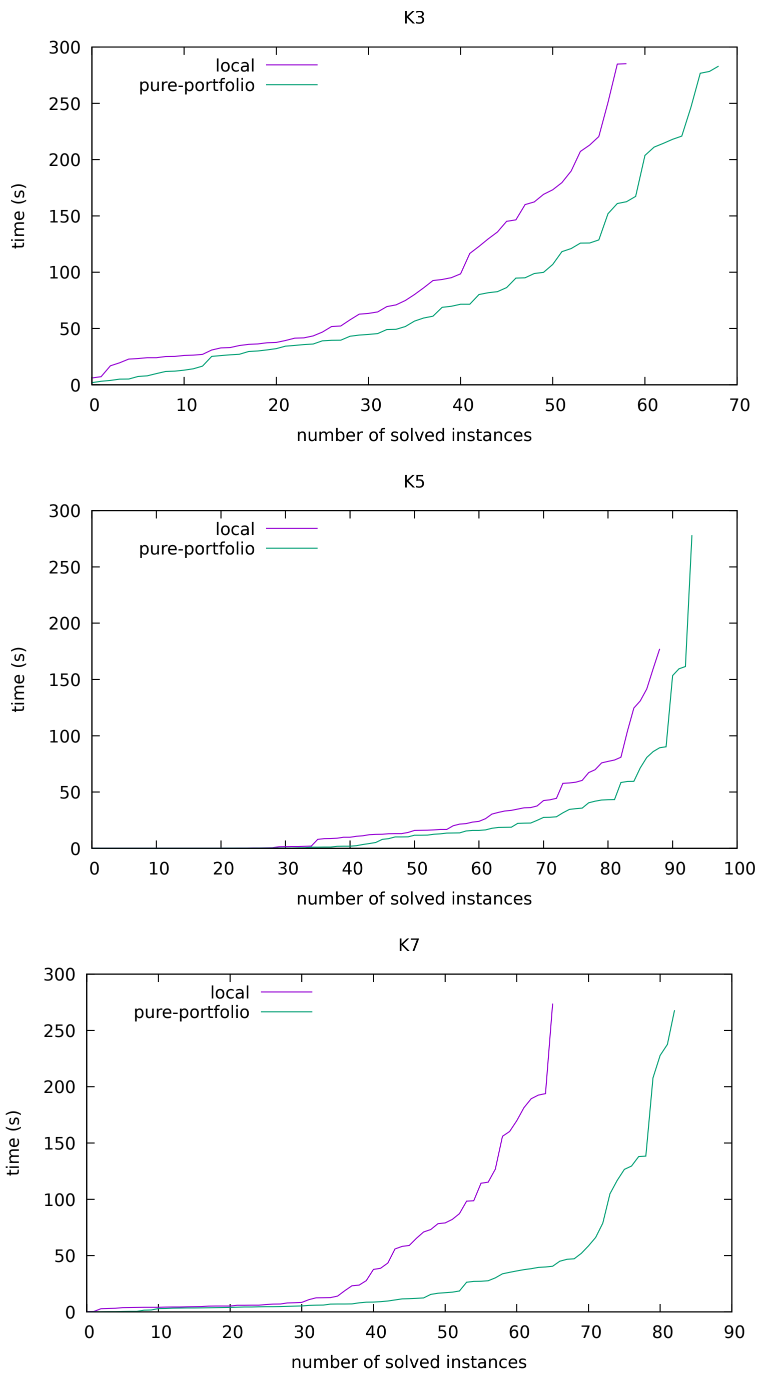

K3
local
pure-portfolio
time (s)
number of solved instances
K5
local
pure-portfolio
time (s)
number of solved instances
K7
local
pure-portfolio
time (s)
number of solved instances

Figure 18:

### 3.7.9 Experiment 11: Initialization with a guide of formula partitioning

| Solver | $3-3(big)$ | $3-3(small)$ | $3-5$ | $5-5$ | $5-7$ |
|---|---|---|---|---|---|
| swpSAT | 1861.73 | 7955.19 | - | 2748.13 | - |
| FineInit | 501.78 (26.95%) | 1734.79 (21.81%) | - | 621.47(22.61%) | - |
| Solver | $7-7$ | BIG | SMALL | COMBINE | |
| swpSAT | 3276.39 | 2794.15 | 15579.71 | 18373.86 | |
| FineInit | 376.06 (11.48% ) | 902.69(32.31%) | 3099.02(19.89%) | 4001.71(21.78%) | |

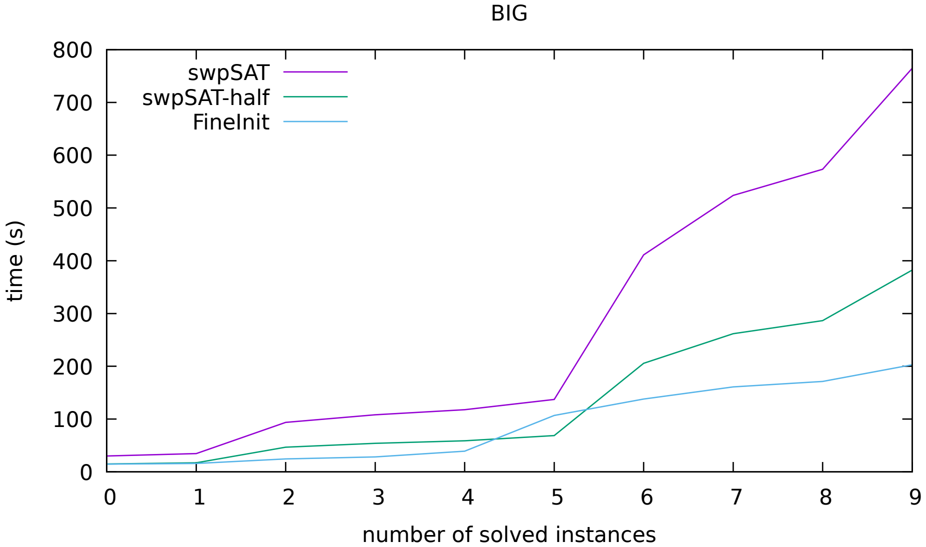



Figure 19:

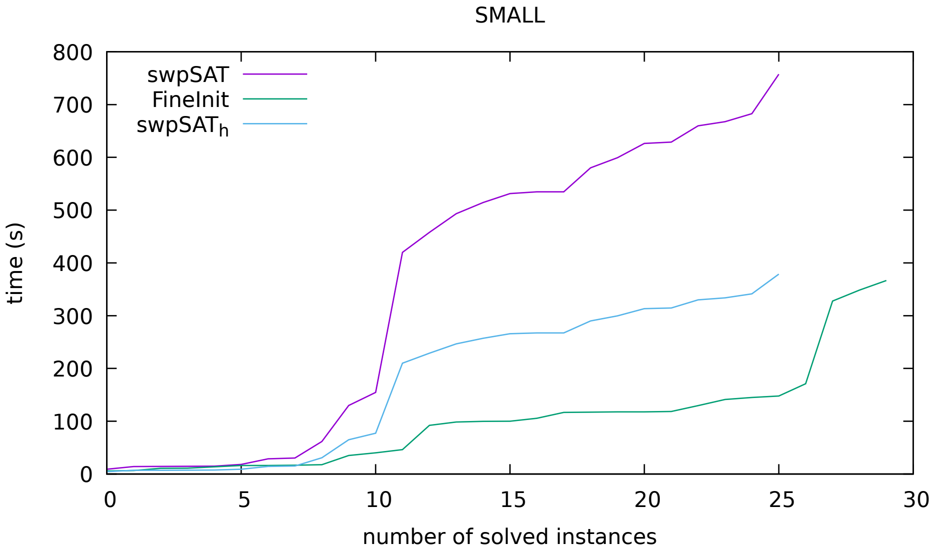



Figure 20:

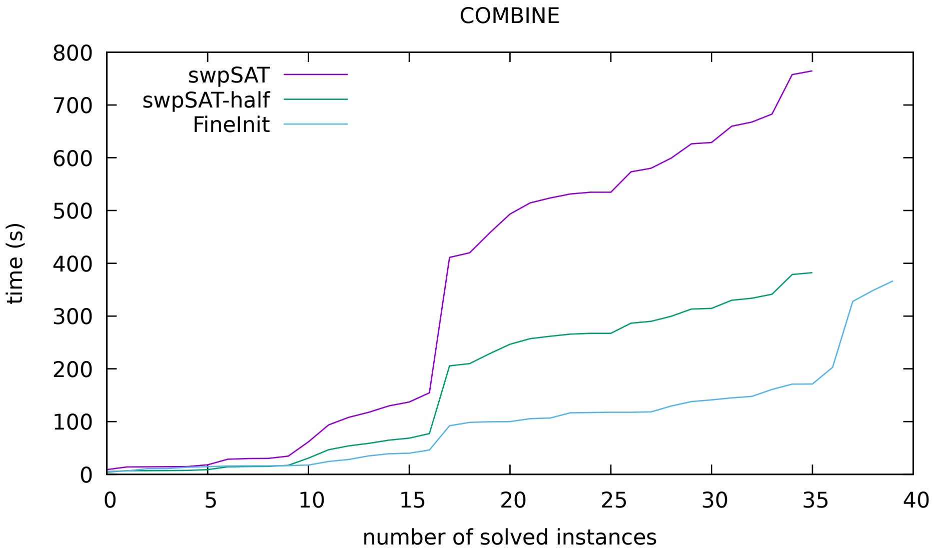

COMBINE
swpSAT
swpSAT-half
FineInit
time (s)
number of solved instances
0
100
200
300
400
500
600
700
800
0
5
10
15
20
25
30
35
40

Figure 21:

# 4 Conclusion

Local search is a universally applicable approach to solve random SAT problems. Our paper presents a stochastic local search algorithm with the cooperation of WALKSAT and probSAT.

In section 3, we discussed the basic scheme of our sequential algorithm. In, we compare the original probSAT with a randomly generated initial solution and our version based on the occurrence of os literals in formel. Based on our experiment, our initialization is advantages n the number of solved problems and also in the execution time of the search. To get the advantage of the greedy algorithm and use the robustness of the stochastic process, we introduce the data structure statistic list to guide the decision between these two processes in step $pickVar()$. Here we propose some variants to make the distinction between greedy choice and a random choice. Generally, our local searches get better performance than the probSAT algorithm.
Based on the performance of these searches in kSAT problems, we get our swpSAT solver, which combines the advantages of the local searches.

In section 4, we make our swpSAT solver parallel with different approaches. Most problems get similar results with the different seed of one random generator. However, the formulas get different results with different random generation. With this fact, we make the parallel version of our local search, in which the agents run the search with different random generators and then take advantages of the suitable one.
In the following part of this section, we discussed the combination of formula partitioning and our parallel solver. After trying several approaches with failure, we found the way of using formula partitioning to make a fine initial solution save search time and furthermore solve more problems. Our experiments evaluate the hyperthesis that the formula partitioning information can guide the local search and improve the efficiency in the parallel search.

## 4.1 Further work

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

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

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

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

[10] 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).

[11] T. Balyo, M. J. Heule, and M. Jaervisalo, "Proceedings of sat competition 2017: Solver and benchmark descriptions," 2017. (Page 11).