Simulated Annealing and Artificial Bee Colony for MAX-3SAT

Bastian Rössl

September 19, 2021

Contents

1 Introduction and Motivation **2**

2 Theory **2**

2.1 Optimization Task for MAX-3SAT 2

2.2 Simulated Annealing 2

2.3 Artificial Bee Colony 3

2.4 Adjustments for Discrete ABC 5

3 Concept and Implementation **5**

3.1 Concept 5

3.2 Implementation 6

3.2.1 How to run 6

3.3 Technical Framework 6

4 Results **7**

4.1 JMH Benchmark 7

4.2 Applicability 7

4.3 JVM Profiling 12

5 Evaluation and Discussion **12**

6 Summary and Outlook **13**

A Appendix **14**

1 Introduction and Motivation

Optimisation algorithms are often copied from observed phenomena as they occur in the environment in general. Simulated Annealing (SA) is a technique which name comes from the annealing process in the domain of metallurgy. By treating materials with heat in a controlled way, physical or chemical properties can be optimized for an overall enhanced stability. Artificial Bee Colony (ABC) mirrors the swarm intelligence of a bee colony searching for the best food sources nearby. In contrast to SA, ABC is not based on physical procedures but is the outcome of a evolutionary adaptation.

In this report, the implementation of SA and ABC will be discussed and applied to MAX-3SAT problems. MAX-3SAT comes from the topic of computational complexity and is NP-complete, thus the existence of a polynomial-time solution is yet unknown and only achievable if P = NP. It is a combinatorical problem because one has to find a propositional variable assignment for a conjunctive normal form (CNF) so that it satisfies the largest number of clauses possible. Since SA is already often applied on discrete problems like Traveling Salesman Problem (TSP), an adoption to MAX-3SAT seems straightforward. ABC has been proposed in 2005 by Derviş Karaboğa and in its origin targets non-discrete, continuous functions. So the leading question is: How can we apply ABC for MAX-3SAT and how would it perform against SA?

2 Theory

2.1 Optimization Task for MAX-3SAT

The boolean satisfiability problem (SAT) is a well-known problem in computer science. While it is computable in polynomial time to evaluate whether a given variable assignment of a formula evaluates to true, it is hard to compute if such a variable assignment exists. 3SAT is a subset of SAT, where the given formula is formed solely out of conjunctions of clauses. Each clause contains three literals in disjunctive composition.

$$(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3) \tag{1}$$

An example of such a formula is given in equation 1 and is also called a "3" conjunctive normal form (3CNF). The clauses add up to four, while three variables are assignable to true or false. E.g. the assignment $x_1 = false, x_2 = false, x_3 = false$ evaluates to $false$, because the first clause is unsatisfied, thus the whole formula as well. Nevertheless, $x_1 = true, x_2 = false, x_3 = false$ proves that 1 is satisfiable. The amount of variable assignments (2^n) is growing exponentially with the number of variables n, which is the root cause of the hard complexity.

In contrast to 3SAT, MAX-3SAT does not target to know if the whole formula is satisfiable, but what is the largest number of satisfiable clauses. For equation 1, the answer is four since we have already shown an assignment which evaluates to true for all clauses.

In this report, we do not aim to solve MAX-3SAT exactly, but to find an approximated solution. The optimization problem is discrete because the solution space is countable - 2^n to be precise, since each variable assignment forms a possible solution. Furthermore, it is a maximization problem if we consider the amount of satisfied clauses as objective function, also called fitness. The input for such an optimization would be a representation of a 3CNF. Let us summarize:

- 3CNF is the problem P with v variables and c clauses.
- All possible variable assignment forms the solution space $S = \{S_1, S_2, ..., S_i\}$ where $i = 2^v$.
- The objective function is the amount of satisfied clauses $c_s \leq c$ for a given solution and problem.
- We target to maximize the objective function's outcome.

2.2 Simulated Annealing

Algorithm 1 depicts the execution plan of a SA optimization. The main idea is to continuously explore solutions near the current solution and to compare them. If the neighbour S_n performs better than the current position S_c, greedy selection will take place (Line 8 - 10). If S_c is a local maximum, the selection condition would always be false and we are stuck – a better, global maximum is not reachable. Therefore, SA also accepts worse solution, but with a certain probability. This depends on the virtual temperature T and the fitness of S_n, as it is defined in Line 12 with $e^{(f_n - f_c)/T} > random[0, 1]$, where $(f_n - f_c)/T$ is a

Algorithm 1: Simulated Annealing

```
Input: The problem $P$ to solve, the initial temperature $T_i > 1$ and the cooling rate $c$ so that
       $0 < c < 1$
Output: An approximated solution $S_g$ for $P$
1  $T \leftarrow T_i$
2  $S_c \leftarrow rndSolution()$
3  $f_g \leftarrow fitness(S_g)$
4  $S_g \leftarrow S_c$
5  while $T > 1$ do
6      $S_n \leftarrow createNeighbour(S_c)$
7      $f_n \leftarrow fitness(S_n)$
8      if $f_n > f_c$ then
9          $S_c \leftarrow S_n$
10         $f_c \leftarrow f_n$
11     else
12         if $e^{(f_n - f_c)/T} > random[0,1]$ then
13             $S_c \leftarrow S_n$
14             $f_c \leftarrow f_n$
15     if $f_c > f_g$ then
16         $S_g \leftarrow S_c$
17         $f_g \leftarrow f_c$
18     $T \leftarrow (1 - c) * T$
19 return $S_g$
```

negative number which approaches 0 for higher T or higher f_n. Thus, high temperatures and a relatively good fitness value enhances the chance to exceed a random number between 0 and 1 ($random[0, 1]$).

At the beginning, a high temperature T_i should be set so that the algorithm is more likely to explore many possible solutions. The virtual annealing (Line 18) with a given cooling rate c lowers the temperature until it hits a limit. Here, the limit is set to 1 (Line 5), but it could be a non-negative parameter as well. Please note that the number of iterations is predictable due to the deterministic calculations involved.

The whole process is initialized with a random start solution. At the end, the best ever traversed solution S_g gets returned.

2.3 Artificial Bee Colony

ABC falls into the category of swarm intelligence (SI), where the outcome is determined by a collective behaviour of a decentralised system consisting of many, rather simplistic agents. In this case, the agents are simulated bees, and the bee behaviour can be of different types. There are worker bees, onlooker bees and scout bees, which perform their actions in subsequent phases. Accordingly, an ABC instance has to be configured with the size n of the swarm or colony.

Worker Bee Phase Half of the colony is consider to be workers. Each worker bee remembers a food source and how valuable it is. A food source is a synonym for a solution and thus, each worker bee is initialized with a random solutions, see Line 2 in Algorithm 2. The Worker Bee Phase takes place in Line 7 to 15, where the worker bee explores a solution nearby the remembered solution. If the food source is better than the previous one, it gets replaced. This procedure helps to approach a local maximum, step by step, as long as the path to it is available via subsequent neighbouring solutions.

Onlooker Bee Phase The onlooker bee agents are used to perform a positive feedback within the swarm. Onlooker bees use the information shared by the worker bees and select one of them by using roulette wheel selection. Thus, better food sources are more likely to be explored in more depth. In general, this phase (Line 16 - 25) is very similar to the previous one with except that a roulette selection is preceded. The probability p_m for Solution S_m to be selected is calculated as in Equation 2.

Algorithm 2: Artificial Bee Colony

```
Input: The problem $P$ to solve, the bee colony size $n > 0$ and the abandonment number $a > 0$
Output: An approximated solution $S_g$ for $P$
1  $w = n/2$
2  $o = n/2$
3  $Solutions \leftarrow \langle S_1 = random(), S_2 = random(), \ldots, S_w = random() \rangle$
4  $AbandonmentCounters \leftarrow \langle a_1 = 0, a_2 = 0, \ldots, a_w = 0 \rangle$
5  $S_g \leftarrow bestOf(Solutions)$
6  while $optimize$ do
7      for $i \leftarrow 1$ to $w$ do
8          $S_n \leftarrow createNeighbour(S_i)$
9          $f_n \leftarrow fitness(S_n)$
10         $f_i \leftarrow fitness(S_i)$
11         if $f_n > f_i$ then
12             $S_i \leftarrow S_n$
13             $a_i \leftarrow 0$
14         else
15             $a_i \leftarrow a_i + 1$
16     for $i \leftarrow 1$ to $o$ do
17         $S_r \leftarrow roulette(Solutions)$
18         $S_n \leftarrow createNeighbour(S_r)$
19         $f_n \leftarrow fitness(S_n)$
20         $f_r \leftarrow fitness(S_r)$
21         if $f_n > f_r$ then
22             $S_r \leftarrow S_n$
23             $a_r \leftarrow 0$
24         else
25             $a_r \leftarrow a_r + 1$
26     for $i \leftarrow 1$ to $w$ do
27         if $a_i > a$ then
28             $S_i \leftarrow random()$
29             $a_i \leftarrow 0$
30     $S_g \leftarrow bestOf(Solutions \cup S_g)$
31 return $S_g$
```

$$p_m = \frac{fitness(S_m)}{\sum\limits_{n=1}^{w} fitness(S_n)} \tag{2}$$

Scout Bee Phase In the last step (Line 26 - 29), those solutions are discarded that could not be improved within a configurable limit a. That worker bee's process is stopped and the bad food source is abandoned to avoid a waste of energy. The worker bee becomes a scout bee which randomly selects a new solution. This adds a negative feedback for the overall SI due to the abandonment.

Stopping Condition Beside the higher complexity compared to SA, there is a noteworthy difference for the stopping condition. While SA's integrated cooldown triggers to halt the procedure after a deterministic amount of iterations, ABC can theoretically run forever (Line 6). For this implementation, we will use a stopwatch to determinate when to exit the loop.

2.4 Adjustments for Discrete ABC

ABC has been proposed for numerical optimization, e.g. for the Rastrigin function. This is due to the neighbourhood operator $createNeighbour(S)$

$$S_n = S + \phi(S - S_r) \tag{3}$$

where S_r is a randomly selected food source and ϕ is a random number between $[-1, 1]$. For discrete problems, this neighbourhood operator has to be redefined reasonable so that it works on variable assignments.

A simple and cost-efficient approach is to flip just one random variable from true to false or vice versa. We will use a slightly enhanced operator, which preselects variables which are listed in one or more unsatisfied clause. Thus, variables which are present in many unsatisfied clauses are more likely to be flipped, causing those clauses to become satisfied and getting a higher probability to create an optimized neighbour. Here, we use the knowledge about our optimization target, but we need to evaluate the variable assignment with a more sophisticated, cost-intense way. E.g.: Clauses $(x_1 \vee x_8 \vee x_5)$ and $(x_1 \vee \neg x_4 \vee x_5)$ are not satisfied. Then, the list to randomly choose from would be $L = \langle x_1, x_1, \neg x_4, x_5, x_5, x_8 \rangle$. Note, that duplicates (or triplicates, ...) are allowed to double (or triple, ...) their probability, because more clauses would benefit if selected.

The described operator will be used for the SA implementation as well.

3 Concept and Implementation

3.1 Concept

The algorithms are tested for various aspects. First, the benchmark tool JMH is used to measure the throughput of the respective implementations at method level. The concrete methods are the parts within the optimization loop, which means that initialization runs are not included in the measurements. The used 3CNF problem is the largest unsatisfiable problem provided with 250 variables and 1065 clauses. For a satisfiable problem, the best possible fitness value is known and if it is reached, algorithms would stop further executions.

Before the measurements are recorded by JMH, warm-up phases are provided. This gives the JVM the chance to perform internal optimizations on byte-code level. In addition, the algorithms are measured in different configurations, since, for example, a large bee colony certainly requires more CPU and RAM than a small one. For SA, a cooling rate of 0 is chosen so that the termination condition is artificially avoided. Further parameter for SA is the temperature, which is chosen high (1.7976931348623157E308) and low (4.9E-324) to deterministically accept respectively not accept the solution to execute appropriate code lines. For ABC, six variants are under test – combinations of small (25), medium (250, one bee per variable) and large (1065, one bee per clause) colonies with fast (1) or restrained (250, one trial per variable) abandonment limits.

The second test is to compare the applicability of various configuration. For a set of problems, a fixed time frame is given for each instance. The mean fitness value over time will be recorded and plotted. Mean, min, max and standard deviation after the completion of all instances are also written to a text file. Although the problem sets included provide unsatisfiable 3CNF, the focus is set on large, satisfiable 3CNF. By doing so, we know that the optimal fitness value is always 1.

Not in scope is the profiling of memory usage. The measurement is not that easy in a JVM, because the used memory fluctuates a lot due to the internal memory management. We can assume that SA has a low memory footprint due to a small fixed amount of attributes and that ABC's footprint grows with the colony size.

Last, we will inspect the CPU consumption with the tool JProfiler by attaching it to the JVM running our algorithm. This will help us to understand the call tree and possible time consuming operations.

3.2 Implementation

Details about the implementation will not be discussed, but can be derived from the source code provided. Some meta information about the implementation are listed below:

- The programming language is Java.
- The implementation is single-threaded for a more predictable outcome.
- The code itself does not aim to be extremely optimized, but the author tried to avoid unnecessary computations whenever possible.
- System.nano() is used to measure the time spent in the execution path. The actual algorithm implementations are wrapped so that the time is measurable and the execution stops after a defined measured time.
- The time measured is the time spend in all optimization steps combined.
- Each algorithm behaviour is encapsulated in one class, a running instance of an algorithm is an object instance of that class.
- Various data sets of problems are included as resource files in DIMACS CNF format [?]. A custom parser has been implemented, see classes *TestSet* and *CNF3Parser*. E.g. Set "uf100_430" contains 1000 satisfiable 3CNF with 100 variables and 430 clauses. "uuf200_860" contains 100 unsatisfiable 3CNF with 200 variables and 860 clauses.

There is one special attribute of the temporal measured SA algorithm: Because we compare both algorithms by time, SA has to be time sensitive. This means, that the cooling rate is derived from the time spend so far and compared with the overall runtime we want to spend. The calculation is as following:

$$Temp(t) = Temp_{init}(\frac{1}{Temp_{init}}^{t/t_{max}}) \tag{4}$$

The calculation is expected to be more compute intense than the code in Line 18 of Algorithm 1, but it enables us to control the runtime with an additional parameter t_{max}.

While $obj(S) = \#satisfiedClauses(S)$ is the objective funtion for MAX-3SAT, the fitness function is defined as $fitness(S) = \frac{\#satisfiedClauses(S)}{\#clauses(S)}$ so a solution's fitness does always approach to 1. This has been done for convenience reason so that all solutions are plottable between 0 and 1.

3.2.1 How to run

Please read the README.md file provided with the source code.

3.3 Technical Framework

The Implementation is meant to be portable and reusable on any device with a JVM runtime supporting Java 8. The comparison outcome is expected to behave different on other environments but the comparison itself is based on time spend on the same system. Thus, relative comparison of the results should still hold true. Nevertheless, the technical properties which were used for all test runs are listed below:

- OS: Windows 10 Pro
- CPU: Intel Core i7-8565U
- RAM: 16 GB
- Compiler and JVM: openjdk 11.0.2 2019-01-15
- JVM arguments: -Xms256m -Xmx2048m

4 Results

4.1 JMH Benchmark

Benchmark	Mode	Cnt	Score	Error	Units
ABC_1065_1	avgt	100	49989932,374	± 899073,052	ns/op
ABC_1065_250	avgt	100	37979940,767	± 900708,249	ns/op
ABC_250_1	avgt	100	12065738,816	± 263829,083	ns/op
ABC_250_250	avgt	100	8959700,473	± 167591,582	ns/op
ABC_25_1	avgt	100	1128228,072	± 22305,276	ns/op
ABC_25_250	avgt	100	828259,096	± 22774,925	ns/op
SA_HIGH	avgt	100	51380,866	± 1332,330	ns/op
SA_LOW	avgt	100	35030,381	± 550,726	ns/op

Table 1: JMH benchmark results for various configurations from slowest to fastest

Table 1 shows how long an average execution for given algorithm took in nanoseconds per operation. The name of the benchmark must be interpreted as following:
ABC_#*colonySize*_#*abandonmentCounter* or **SA**_*temperature*. See Section 3.1 for a more detailed explanation.

4.2 Applicability

In this sections, results about the fitness performances over time are listed. An interactive console application receives costumizable configuration settings to generate a comparison run for one or more algorithms. The naming convention for the series in the chart legend is like in Table 1, but for SA the temperature is the actual used parameter.

The auto-generated titles for all the graphs read as following: *Problem_Iterations_Runtime*. The naming of *Problem* is subdivided into **uuf**#*variables*_#*clauses* for unsatisfiable formulas, **uf**#*variables*_#*clauses* for satisfiable formulas.

100 satisfiable 3CNF, 250 variables, 1065 clauses, 10 seconds				
Series	Mean	Max	Min	Standard Deviation
abc_c2000_a10	0,981286	0,987793	0,976526	0,002203
abc_c200_a10	0,982329	0,987793	0,977465	0,002286
abc_c20_a10	0,981840	0,989671	0,976526	0,002369
abc_c2000_a100	0,985014	0,991549	0,980282	0,002153
abc_c200_a100	0,987690	0,991549	0,984038	0,001615
abc_c20_a100	0,987934	0,992488	0,984038	0,001697
sa_t1000000	0,981991	0,988732	0,977465	0,002644
sa_t1000	0,981681	0,992488	0,975587	0,003208

Table 2: Mixed Comparison
The final results after 8*100 executions, each lasting 10 seconds.

100 satisfiable 3CNF, 250 variables, 1065 clauses, 10 seconds				
Series	Mean	Max	Min	Standard Deviation
sa_t10	0,980789	0,988732	0,974648	0,002941
sa_t1000	0,981023	0,990610	0,974648	0,002955
sa_t100000	0,980676	0,990610	0,974648	0,002895
sa_t10000000	0,981192	0,987793	0,974648	0,003259
sa_t1000000000	0,981014	0,989671	0,974648	0,003350
sa_t100000000000	0,980995	0,987793	0,975587	0,002919

Table 3: Temperature Comparison
The final results after 6*100 executions, each lasting 10 seconds.

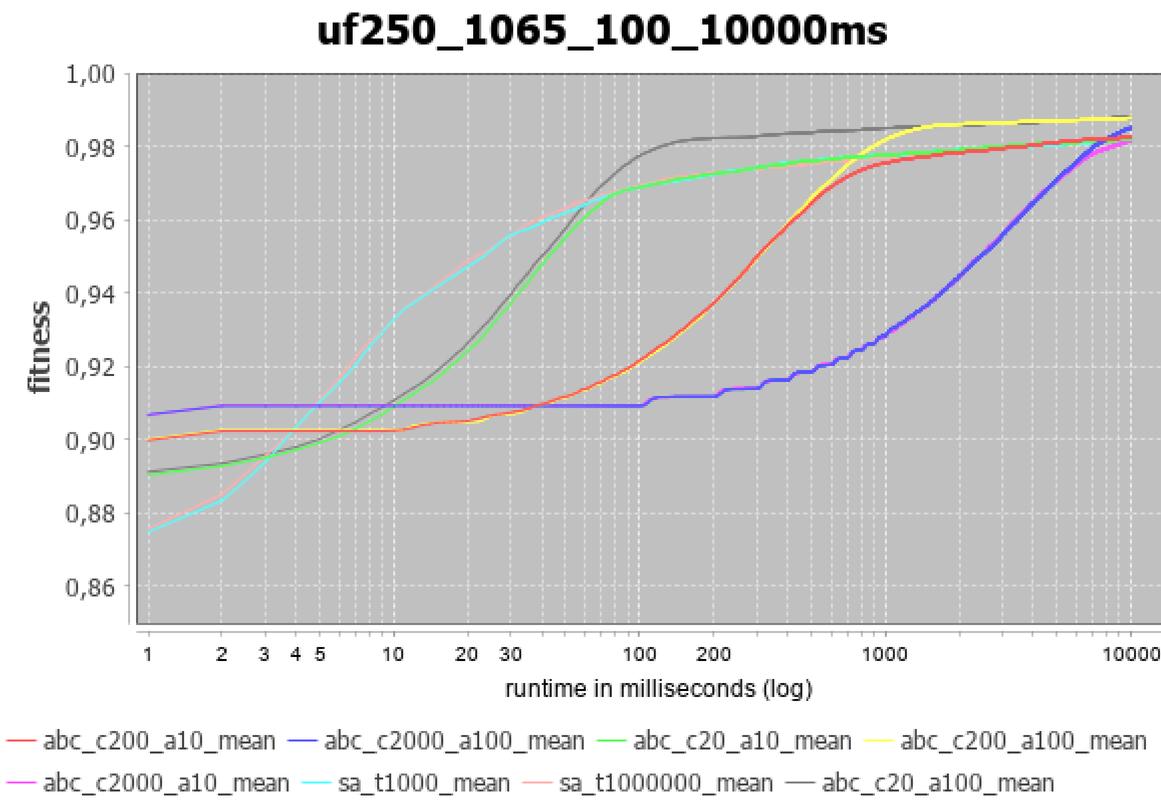


Figure 1: Mixed Comparison
A mixed comparison between eight configurations. The problem is satisfiable but very large (250 variables, 1065 clauses). Each optimization instance executed for 10 seconds, repeating it for 100 different problems.

100 satisfiable 3CNF, 250 variables, 1065 clauses, 10 seconds				
Series	Mean	Max	Min	Standard Deviation
abc_c8_a10	0,982366	0,988732	0,977465	0,002081
abc_c32_a10	0,981897	0,986854	0,977465	0,002198
abc_c128_a10	0,981962	0,987793	0,977465	0,002195
abc_c512_a10	0,981606	0,987793	0,976526	0,002226
abc_c2048_a10	0,980958	0,988732	0,975587	0,002427

Table 4: Colony Size Comparison
The final results after 5*100 executions, each lasting 10 seconds.

100 satisfiable 3CNF, 250 variables, 1065 clauses, 10 seconds				
Series	Mean	Max	Min	Standard Deviation
abc_c200_a8	0,979596	0,985915	0,974648	0,002423
abc_c200_a64	0,987634	0,992488	0,984038	0,001835
abc_c200_a512	0,987362	0,991549	0,984038	0,001457

Table 5: Abandonment Comparison
The final results after 3*100 executions, each lasting 10 seconds.

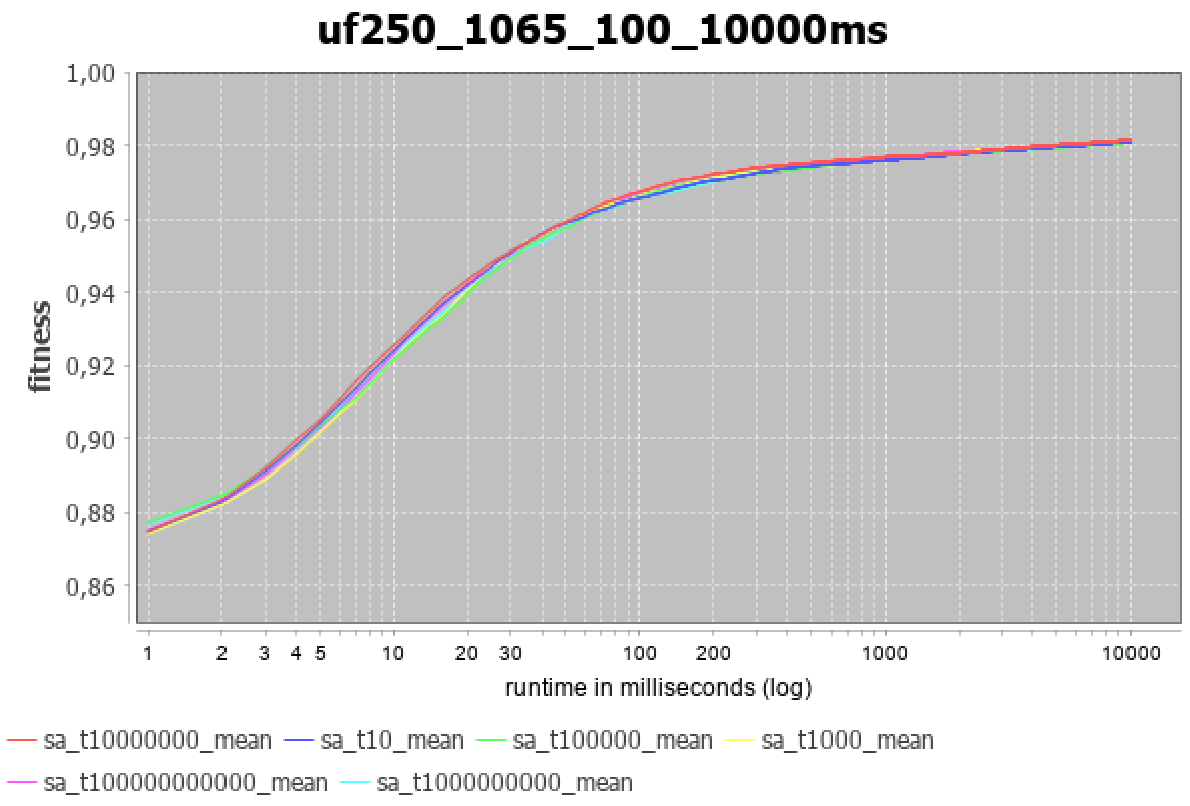


Figure 2: Temperature Comparison
A comparison between different SA temperature configurations. The problems are satisfiable with 250 variables and 1065 clauses. Each optimization instance executed for 10 seconds, repeating it for 100 different problems.

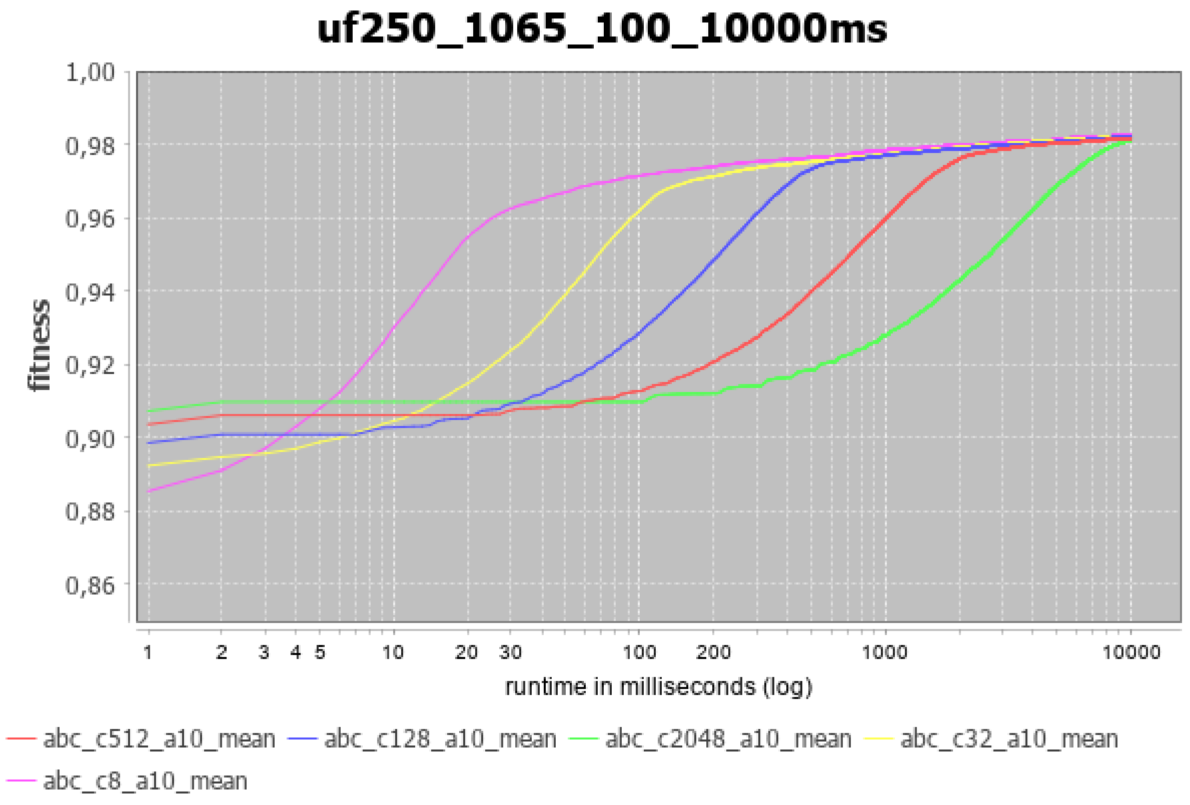


Figure 3: Colony Size Comparison
A comparison between different ABC colony size configurations. The problems are satisfiable with 250 variables and 1065 clauses. Each optimization instance executed for 10 seconds, repeating it for 100 different problems.

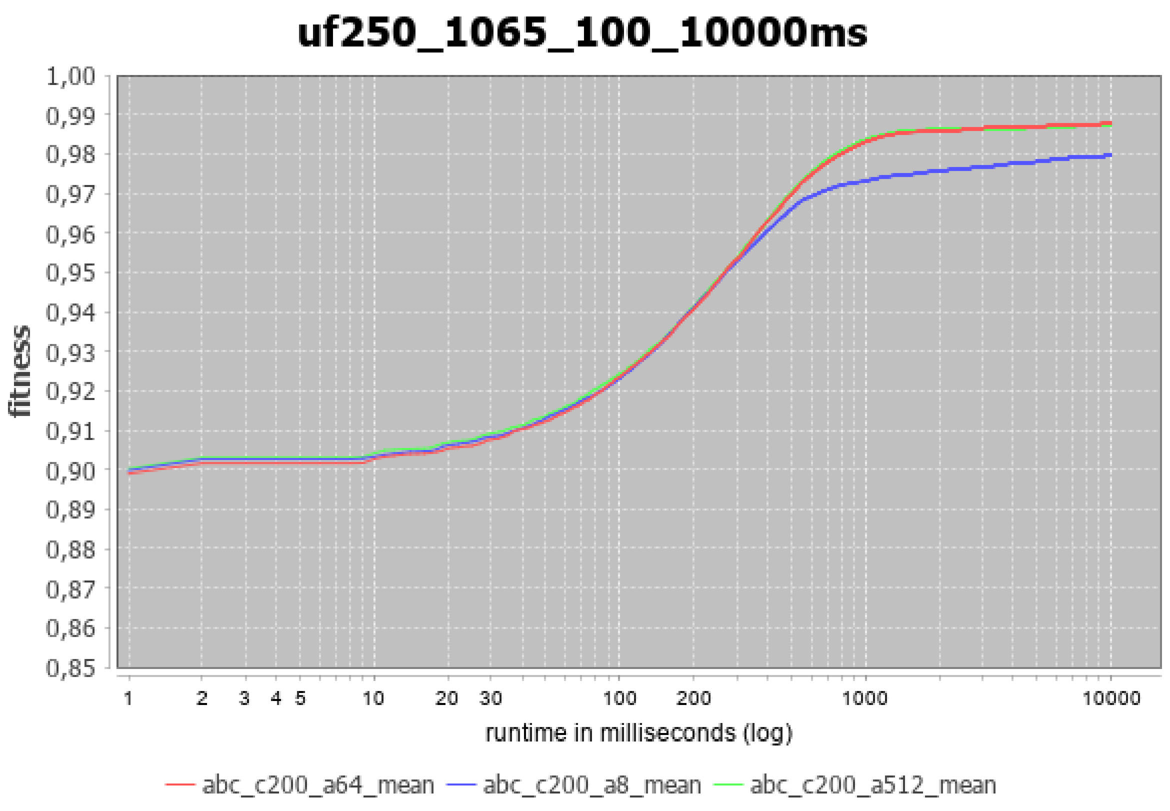


Figure 4: Abandonment Comparison
A comparison between different ABC abandonment configurations. The problems are satisfiable with 250 variables and 1065 clauses. Each optimization instance executed for 10 seconds, repeating it for 100 different problems.

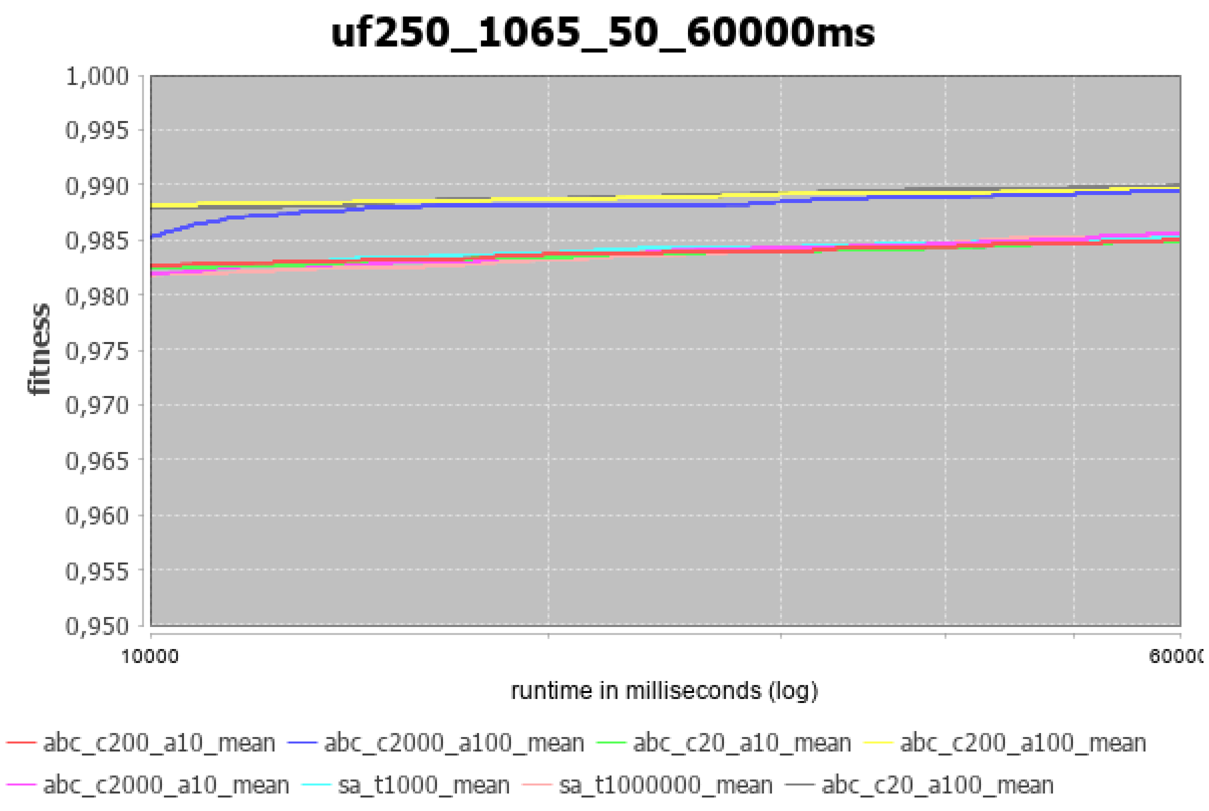


Figure 5: Long Run Comparison
A comparison between execution time up to 60 seconds. The problems are satisfiable with 250 variables and 1065 clauses. Each optimization instance executed for 60 seconds, repeating it for 50 different problems.

50 satisfiable 3CNF, 250 variables, 1065 clauses, 60 seconds				
Series	Mean	Max	Min	Standard Deviation
abc_c2000_a10	0,985634	0,990610	0,981221	0,002287
abc_c200_a10	0,985014	0,990610	0,982160	0,002057
abc_c20_a10	0,984883	0,991549	0,981221	0,002347
abc_c2000_a100	0,989371	0,994366	0,986854	0,001565
abc_c200_a100	0,989690	0,992488	0,986854	0,001424
abc_c20_a100	0,989897	0,993427	0,987793	0,001281
sa_t1000000	0,985484	0,992488	0,981221	0,002755
sa_t1000	0,985296	0,993427	0,978404	0,003106

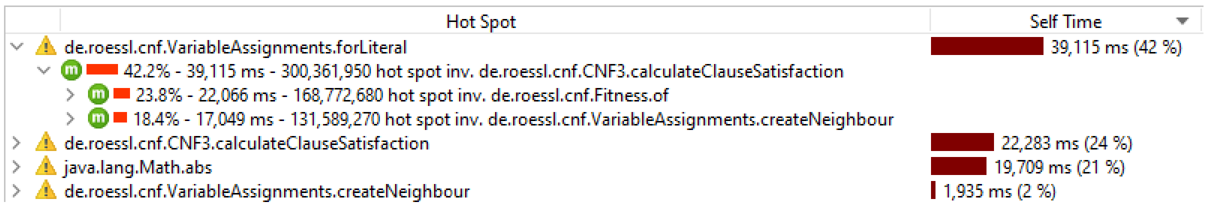
Table 6: Long Run Comparison
The final results after 8*100 executions, each lasting 60 seconds.

100 satisfiable 3CNF, 10 seconds						
Series	Variables	Clauses	Mean	Max	Min	Standard Deviation
abc_c2000_a10	20	91	1,000000	1,000000	1,000000	0,000000
	50	218	0,999954	1,000000	0,995413	0,000456
	100	430	0,994698	1,000000	0,988372	0,002485
	150	645	0,988868	0,996899	0,982946	0,002579
	200	860	0,984023	0,990698	0,979070	0,002369
abc_c200_a10	20	91	1,000000	1,000000	1,000000	0,000000
	50	218	1,000000	1,000000	1,000000	0,000000
	100	430	0,994395	1,000000	0,990698	0,002419
	150	645	0,989240	0,995349	0,984496	0,002611
	200	860	0,984570	0,993023	0,977907	0,002482
abc_c20_a10	20	91	1,000000	1,000000	1,000000	0,000000
	50	218	0,999954	1,000000	0,995413	0,000456
	100	430	0,994651	1,000000	0,988372	0,002406
	150	645	0,989535	0,995349	0,984496	0,002314
	200	860	0,984570	0,993023	0,979070	0,002348
abc_c2000_a100	20	91	1,000000	1,000000	1,000000	0,000000
	50	218	0,999954	1,000000	0,995413	0,000456
	100	430	0,995000	1,000000	0,990698	0,001689
	150	645	0,991612	0,995349	0,986047	0,001849
	200	860	0,987651	0,993023	0,981395	0,001944
abc_c200_a100	20	91	1,000000	1,000000	1,000000	0,000000
	50	218	0,999771	1,000000	0,995413	0,001000
	100	430	0,994744	1,000000	0,988372	0,002122
	150	645	0,991318	0,996899	0,987597	0,001684
	200	860	0,988837	0,993023	0,984884	0,001489
abc_c20_a100	20	91	1,000000	1,000000	1,000000	0,000000
	50	218	0,999817	1,000000	0,995413	0,000899
	100	430	0,994884	1,000000	0,990698	0,002080
	150	645	0,991752	0,995349	0,987597	0,001885
	200	860	0,985634	0,990610	0,981221	0,002287
sa_t1000000	20	91	1,000000	1,000000	1,000000	0,000000
	50	218	1,000000	1,000000	1,000000	0,000000
	100	430	0,995465	1,000000	0,986047	0,003212
	150	645	0,989814	0,998450	0,982946	0,003391
	200	860	0,984640	0,990698	0,973256	0,002980
sa_t1000	20	91	1,000000	1,000000	1,000000	0,000000
	50	218	1,000000	1,000000	1,000000	0,000000
	100	430	0,995163	1,000000	0,983721	0,003234
	150	645	0,989364	0,998450	0,979845	0,003569
	200	860	0,984279	0,990698	0,976744	0,002888

Table 7: Comparison between various problem sizes.
The plotted results are added to the Appendix.

4.3 JVM Profiling

The results of the profiling were exploratively tested for relevance. It was noticed that a very high proportion of the CPU falls in the clause satisfaction calculation. This is not surprising, since it is used for the calculation of fitness as well as for the calculation of a neighbour. In Figure 6 a screenshot of the JVM profiling session with JProfiler illustrates this aspect. The call hierarchy can be obtained in Figure 12, see Appendix.



Hot Spot	Self Time
de.roessl.cnf.VariableAssignments.forLiteral	39,115 ms (42 %)
42.2% - 39,115 ms - 300,361,950 hot spot inv. de.roessl.cnf.CNF3.calculateClauseSatisfaction	
23.8% - 22,066 ms - 168,772,680 hot spot inv. de.roessl.cnf.Fitness.of	
18.4% - 17,049 ms - 131,589,270 hot spot inv. de.roessl.cnf.VariableAssignments.createNeighbour	
de.roessl.cnf.CNF3.calculateClauseSatisfaction	22,283 ms (24 %)
java.lang.Math.abs	19,709 ms (21 %)
de.roessl.cnf.VariableAssignments.createNeighbour	1,935 ms (2 %)

Figure 6: Hot Spots
Method calculateClauseSatisfaction() is by far the most CPU consuming part of the application.

5 Evaluation and Discussion

We can derive the following conclusions from the results:

1. In Table 1 we see that SA iterations run faster than ABC iterations, which in turn take longer the larger the colony. However, if one divides the times by the colony size, one recognizes that per bee similarly much time is used up as in one SA execution.

2. Table 1 also shows that fast abandonments for ABC or high temperatures for SA slow down respective algorithm. This is due to the higher probability to execute additional code lines.

3. Figure 2 shows that the initial temperature has no effect on the performance of SA for MAX-3SAT. This is due to our adjustments as described in section 3.2. When using a higher temperature, the cooling rate is proportionally faster. This combination answers why the expected impact of a higher temperature dissipates. To fix this, but still be able to configure SA with a time limit rather than a cooling rate, one should consider to change the fixed target temperature $T_{target} = 1$ to a configurable parameter.

4. In Figure 1 it can be seen that SA improves the solution the fastest, but ABC catches up and overtakes it over time.

5. The mean time in Table 4 suggests that the bee size does not make up a lot of a difference, but in Figure 3 it is visible that larger colonies start to increase the fitness value later than smaller ones.

6. Table 5 and Figure 4 indicates that ABC instances with a low abandonment number struggle to perform well in the end. It seems like the bee has no chance to optimize its food source within a strict limit.

7. For long running optimizations, ABC instances with high abandonment configurations are best.

8. Max values of Table 6 prove that no single instance was able to find the best possible solution with fitness value 1, and the very low gradient obtained in Figure 5 does not give reason to expect to do so in an appropriate time frame.

9. All algorithms were able to find the global maximum as long as the variable size does not hit a certain limit between 100 and 150, as the max values of Table 7 imply. These are $2^{100} \approx 1.267 * 10^{30}$ respectively $2^{150} \approx 1.427 * 10^{45}$ possible assignments. If we would use a brute force approach and we could check each assignment within 10000 nanoseconds (educated guess from Table 1), the algorithm would take around $4.019 * 10^{17}$ years. For Comparison: The age of the universe is assumed to be 13,7 billion years.

10. If one targets to use a more predictable outcome after a long execution of one algorithm instance, a small ABC with a high abandonment limit seems to be the best choice (0,001281 standard deviation, see Table 6), and SA with low initial temperature seems worst.

11. Most computation time is lost to derive a neighbour solution or to calculate the fitness. This is not a surprise because there are invoked heavily (see Section 4.3). If one targets to optimize the code, dedicated data structures and methods for CNF evaluation should be in scope.

6 Summary and Outlook

In this comparison, one can see that ABC is adjustable to discrete problems and that it is superior to SA in optimizing MAX-3SAT. The choice of a small colonies with a long residence time on food sources is recommended if a small standard deviation is desired. Even if for larger problems the global maximum is never reached with very high probability, it is vastly preferable to a brute force variant. It might be interesting how ABC performs compared to a dedicated algorithm used to solve SAT, e.g. DPLL [**?**].

A little optimization potential for the implementation of SA has been identified, which may impact the overall result. The cooling function dictates how long SA accepts worse solution with higher probability and this is one of the key elements of SA.

Of particular interest is whether all algorithms could be greatly accelerated by adjusting the internal CNF representation so that calculations applied to it take less time. We have seen how important the fast execution of the neighbourhood operation is, and it might be that the assumption made in Section 2.4 was counterproductive. For future investigation, it is interesting to see if a simple neighbourhood operator is actually healthier for the overall performance than a heuristic approach.

A Appendix

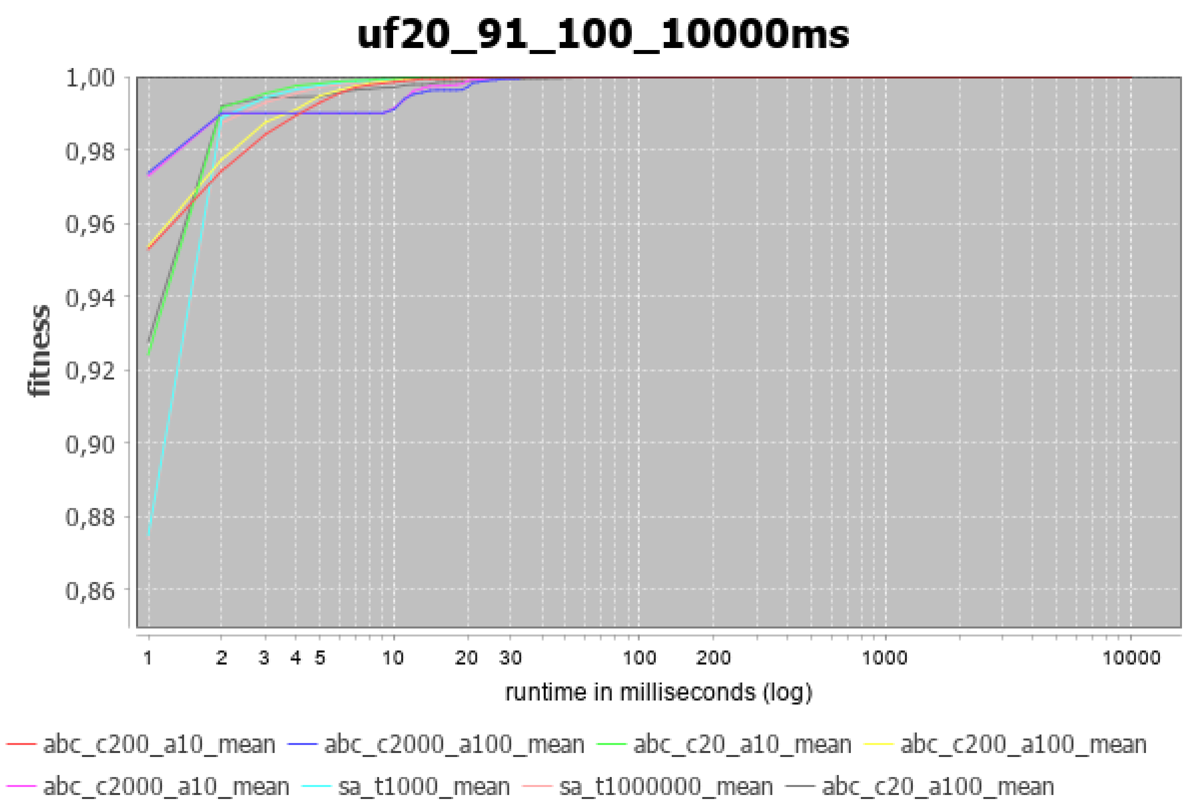


Figure 7: Comparison for 20 variables.

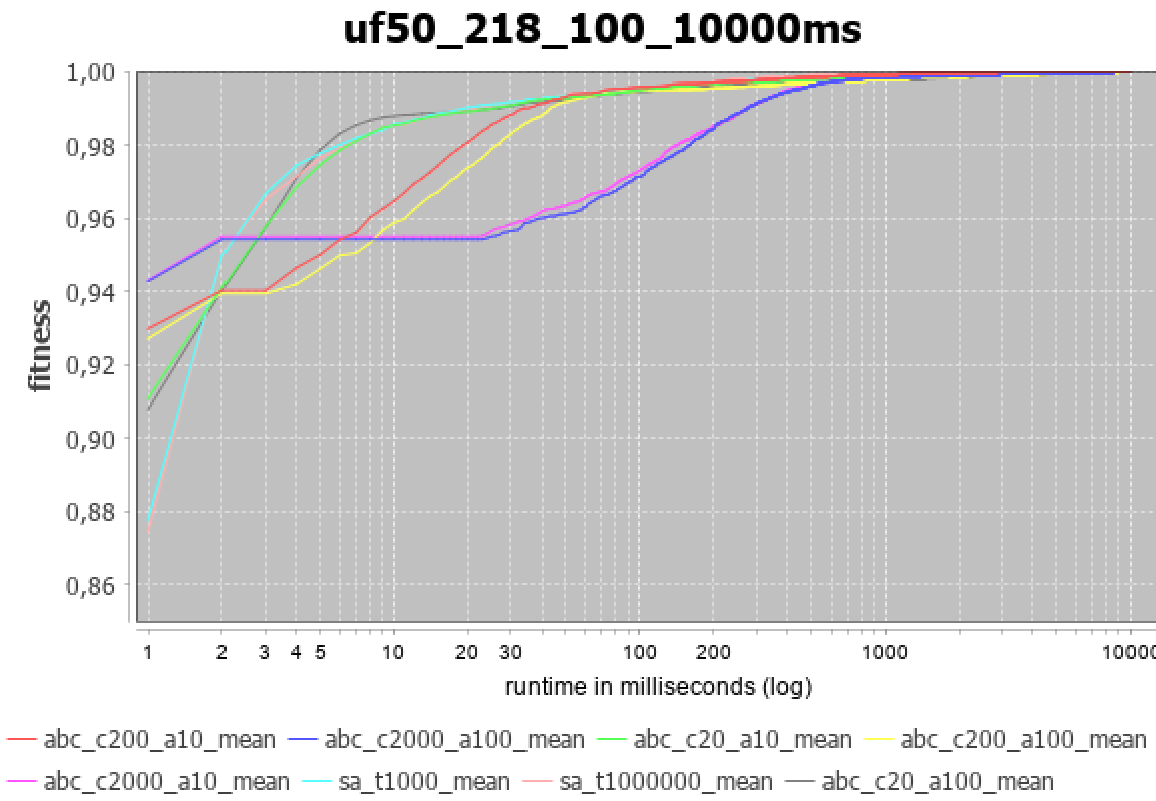


Figure 8: Comparison for 50 variables.

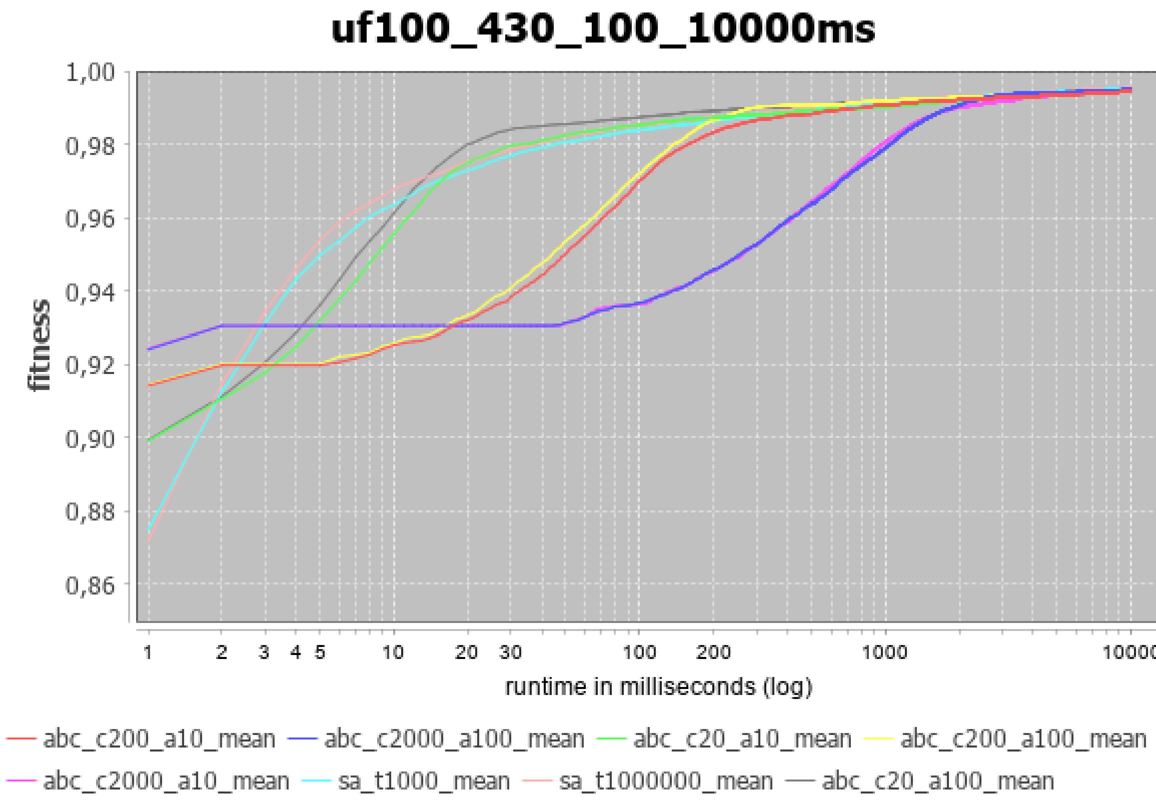


Figure 9: Comparison for 100 variables.

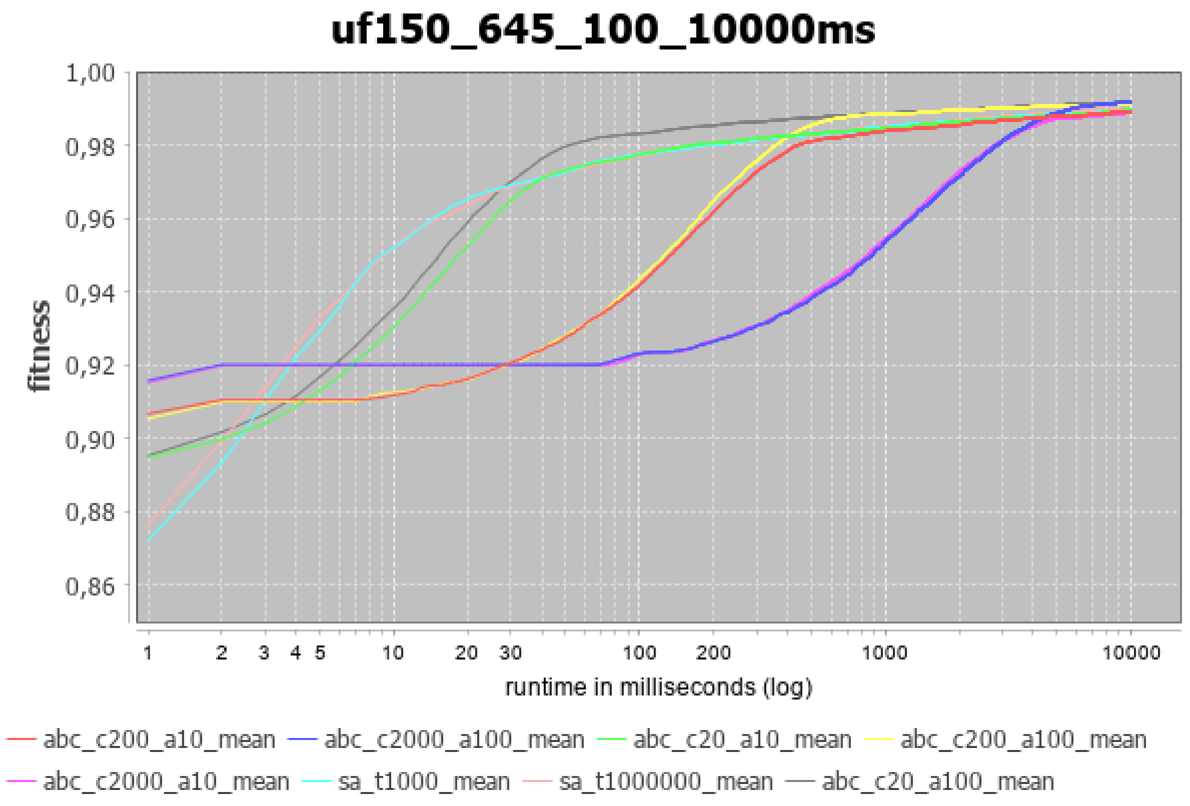


Figure 10: Comparison for 150 variables.

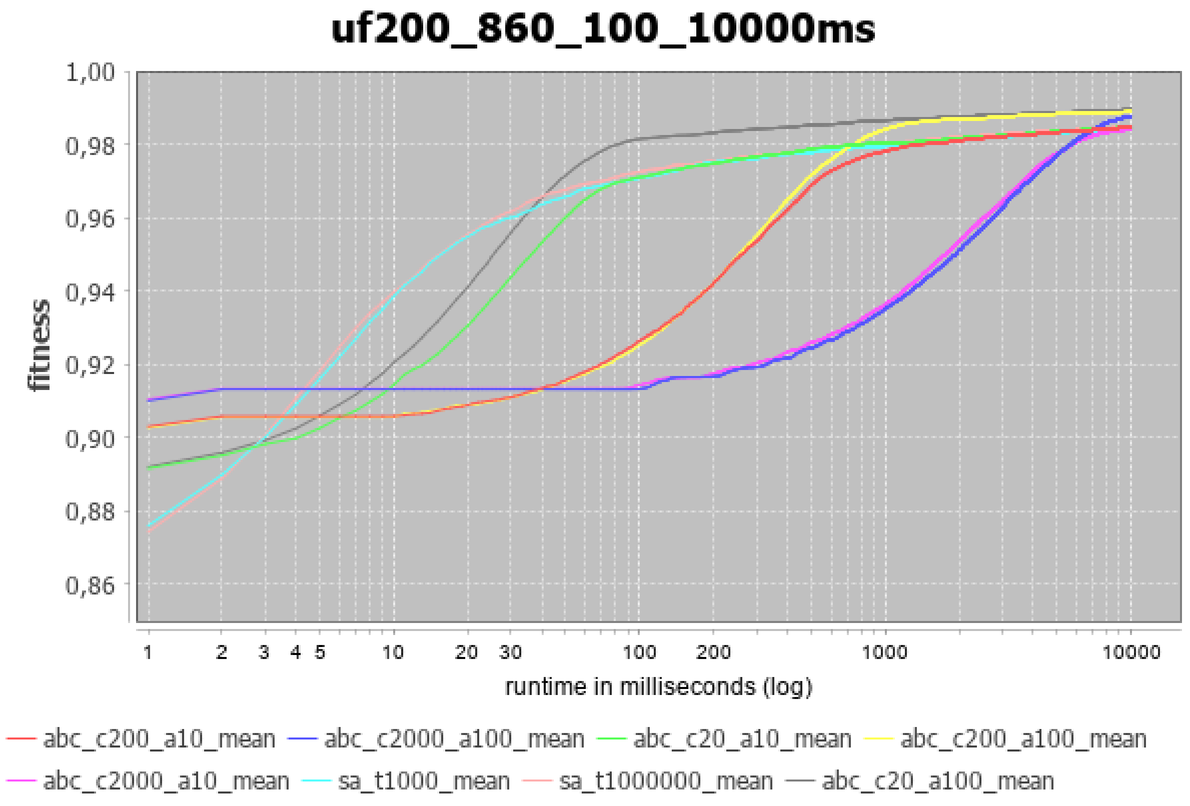


Figure 11: Comparison for 200 variables.

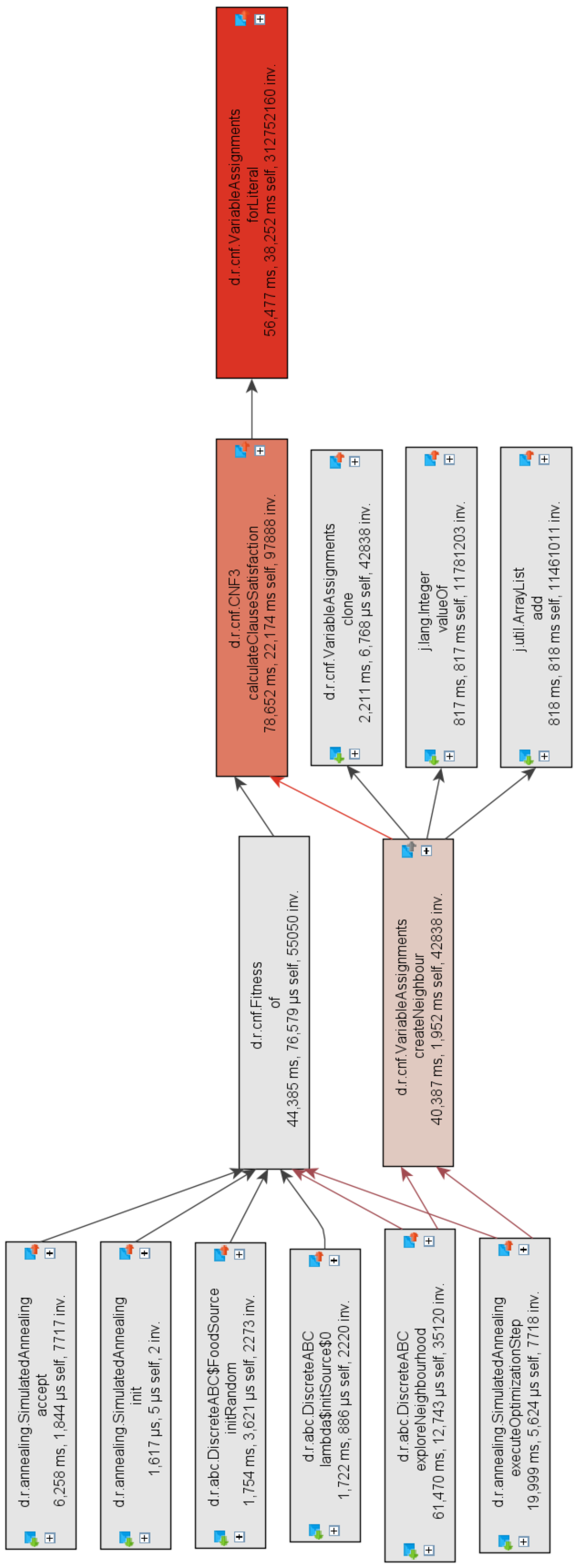

d.r.cnf.VariableAssignments
forLiteral
56,477 ms, 38,252 ms self, 31275160 inv.
d.r.cnf.CNF3
calculateClauseSatisfaction
78,652 ms, 22,174 ms self, 97888 inv.
d.r.cnf.VariableAssignments
clone
2,211 ms, 6,768 µs self, 42838 inv.
j.lang.Integer
valueOf
817 ms, 817 ms self, 11781203 inv.
j.util.ArrayList
add
818 ms, 818 ms self, 11461011 inv.
d.r.cnf.Fitness
of
44,385 ms, 76,579 µs self, 55050 inv.
d.r.cnf.VariableAssignments
createNeighbour
40,387 ms, 1,952 ms self, 42838 inv.
d.r.annealing.SimulatedAnnealing
accept
6,258 ms, 1,844 µs self, 7717 inv.
d.r.annealing.SimulatedAnnealing
init
1,617 µs, 5 µs self, 2 inv.
d.r.abc.DiscreteABC$FoodSource
initRandom
1,754 ms, 3,621 µs self, 2273 inv.
d.r.abc.DiscreteABC
lambda$initSource$0
1,722 ms, 886 µs self, 2220 inv.
d.r.abc.DiscreteABC
exploreNeighbourhood
61,470 ms, 12,743 µs self, 35120 inv.
d.r.annealing.SimulatedAnnealing
executeOptimizationStep
19,999 ms, 5,624 µs self, 7718 inv.

Figure 12: Call hierarchy for fitnessOf() and createNeighbour()
Most CPU time has been spend within the fitness calculation and while creating a neighbour solution. For this profiling session, 97k invocations for calculateClauseSatisfaction() were needed