SOLUTION OF THE MAXIMUM WEIGHTED SATISFIABILITY PROBLEM USING SIMULATED ANNEALING

Andrei Shchapaniak

January 7, 2024

Contents

1 Introduction

1.1 Assignment

The goal of this task was to implement a chosen algorithm for solving the Maximum Weighted Satisfiability Problem (MWSAT) of a Boolean formula:

- Given: a vector X of variables $(x_1 \dots x_n)$, where $x_i \in \{0, 1\}$, and a Boolean formula of these variables in conjunctive normal form with m clauses (sum terms), and then for each variable a weight $w(c)$.
- Construct: an evaluation Y of variables X such that $F(Y) = 1$ and $\sum_{i=1}^{n} y_i w(i)$ is maximized.

1.2 Simulated Annealing

Simulated Annealing (SA) is a probabilistic technique for approximating the global optimum of a complex optimization problem. This method draws inspiration from the physical process of annealing in metallurgy, where materials are heated to a high temperature and then gradually cooled to remove defects and achieve a more stable structure. Similarly, SA navigates the solution space of an optimization problem by initially allowing a higher degree of randomness in its search and gradually becoming more focused as the 'temperature' of the algorithm lowers.

The main advantage of SA lies in its ability to avoid being stuck in local optima - suboptimal solutions that are better than neighboring solutions but worse than the global optimum. In the early stages of the algorithm, when the 'temperature' is high, SA is more likely to accept both improvements and worsened solutions. This behavior enables the algorithm to explore a broad range of the solution space. As the temperature decreases, SA becomes more selective, accepting fewer worsened solutions, and converges toward a stable solution.

In this study, SA is applied to solve the **Maximum Weighted Satisfiability Problem** (MWSAT). The heuristic nature of SA, combined with its ability to navigate large and complex search spaces, makes it an apt choice for seeking near-optimal solutions to MWSAT. The following sections delve into the specific implementation details of the algorithm, highlighting how it has been adapted and optimized to address the unique challenges presented by MWSAT.

2 Implementation

The heuristic was implemented in C++, a choice driven by the language's ability to expedite problem-solving processes, especially when compared to certain dynamic languages. This choice significantly enhanced and sped up the evaluation of all runs. The following subsections will delve deeper into certain functions, highlighting how their design and what role they play in the overall effectiveness of the simulated annealing algorithm.

2.1 Initial temperature

To compute the initial temperature T_0 for the simulated annealing algorithm, the following formula was used:

$$T_0 = -\frac{\Sigma\delta_r}{|S|\ln(x_0)} \tag{1}$$

Here, $\Sigma\delta_r$ is the sum of the cost differences between solutions if the generated neighbor was worse, $|S|$ is the number of solutions sampled, and $\ln(x_0)$ is the natural logarithm of the initial acceptance probability x_0, which was set to fifty percent in this implementation. This formula has been adopted from the findings presented here [1].

2.2 Fitness function

The function **calculate_total_weight()** calculates the weighted sum of a given array configuration using corresponding weights, implemented in a straightforward manner, returning the total as a single numerical value.

```
func calculate_total_weight(configuration)
    total_weight_sum = 0.0
    weights = get_weights()

    for i from 0 to size(configuration) - 1 do
        total_weight_sum = total_weight_sum + configuration[i] *
            weights[i]

    return total_weight_sum
```

2.3 Neighbour generation

The function **generate_neighbour()** generates a new solution for a problem, by probabilistically altering its variables to find a neighboring solution in the solution space. With a ten percent chance, it generates a completely random solution. Otherwise, it seeks to improve the current solution by flipping a variable that increases the number of satisfied clauses. If no such improvement is found, it randomly flips one variable. The function returns this new, potentially improved, configuration.

```
func generate_neighbour(current_configuration):
    new_configuration = copy(current_configuration)
    probability_of_flip = get_random_int(0, 99)

    with 10% probability do
        return random_configuration()

    if count_satisfied_clauses(new_configuration) !=
       total_number_of_clauses then
        for each clause in get_clauses() do
            if clause is not satisfied in new_configuration then
                for each pair in clause do
                    gain = gain_of_flip(new_configuration, pair.variable
                       - 1)
                    if gain > 0 then
                        flip the value at pair.variable - 1 in
                           new_configuration
                        return new_configuration

    random_index = get_random_int(0, number_of_variables - 1)
    flip the value at random_index in new_configuration
    return new_configuration
```

2.4 Main function

The **main()** function for a simulated annealing algorithm begins by setting up essential parameters, such as the initial and final temperatures, that guide the simulation. The function then enters a main loop, which runs until the current temperature is higher than the final.

Within this loop lies an inner cycle, dedicated to achieving equilibrium at the current temperature level. This stage is vital for thorough exploration and evaluation of potential solutions. Here, the **generate_neighbour()** function, detailed in Subsection 2.3, is used to generate new potential solutions. The effectiveness of each new solution is assessed using the **calc_total_weight()** function, described in Subsection 2.2. These two functions form the backbone of our algorithm, driving its ability to explore the solution space effectively.

At the core of this loop is an inner cycle, tasked with achieving equilibrium at the current temperature. Within this cycle, the **generate_neighbour()** function is employed to create a new solution, and its effectiveness is evaluated using **calc_total_weight()**. These functions, elaborated in Subsections 2.3 and 2.2 respectively, are fundamental to the algorithm's operation. The algorithm accepts a new solution if it is superior to the current one, updating both the current and potentially the best solution found thus far. If the new solution is less effective, the algorithm uses a probabilistic approach, based on the current temperature and the quality difference, to decide on its acceptance.

If this count exceeds a set threshold, here **1000** iterations, the algorithm concludes that it is unlikely to find better solutions and terminates the search. This threshold is a strategic choice, balancing the need for thorough exploration against the risk of excessive computation on unproductive paths. It ensures that the algorithm remains efficient and focused, terminating the search when improvements become improbable.

3 White box

The objective of this Section is to delve into the internal mechanics of the application through white box testing. This analytical phase involves rigorously testing the heuristic against small sets of instances, with a focus on adjusting key parameters to effectively address various levels of instance complexity. Should the heuristic exhibit suboptimal performance under these conditions, a revision of the program's code will be considered.

To accurately gauge the heuristic's performance, two critical metrics will be closely observed: the trend of satisfied clauses and the solution's weight. By analyzing how the number of satisfied clauses evolves during each run, we can assess the effectiveness of the heuristic in finding optimal solutions. Meanwhile, keeping an eye on the weight of the solution will inform us about the heuristic's efficiency in balancing quality with computational cost. This dual-metric approach will provide a clearer picture of the heuristic's overall performance, where an increasing number of satisfied clauses and a reasonable solution weight would indicate a successful optimization process.

For this purpose, the instance **M/wuf50-0472.mwcnf** was utilized as a test case, and the heuristic was run 10 times for each combination of the three parameters being examined. The initial values for these parameters were sourced from university lessons, providing a baseline for our tests. As the testing progresses, these parameters will be adjusted to explore their impact on the heuristic's performance more thoroughly. This approach is designed to provide a comprehensive understanding of how each parameter influences the performance of the heuristic and to identify the optimal configuration for effectively solving the problem.

The parameters under examination are:

- `Length of equilibrium` - determines the number of iterations at a given temperature before cooling proceeds, impacting the algorithm's balance between exploration and exploitation.
- `Final temperature` - sets the stopping condition for the algorithm, where a higher final temperature may end the search prematurely, and a lower one extends the search, possibly yielding a more refined solution.
- `Cooling coefficient` - affects how quickly the temperature decreases; a higher coefficient can lead to rapid cooling, potentially missing the global optimum, while a lower one allows for a more exhaustive search of the space.

This structured approach, beginning with established parameter values and then iteratively refining them, ensures a methodical and informed exploration of the solution space, essential for the heuristic's effective optimization.

3.1 Tests

The first run: T = 10, e = 100, a = 0.9

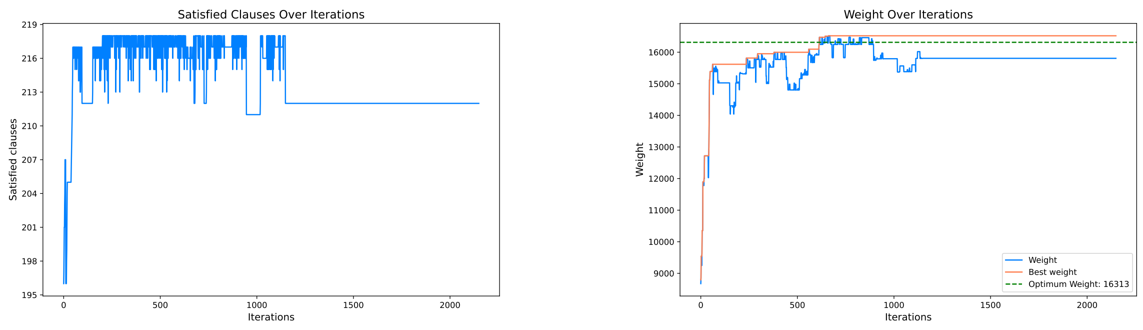


Figure 1: First run results.

During the series of tests conducted, success was not achieved by any run. It has been observed from the analysis of the graphs that the algorithm consistently failed to achieve equilibrium, with the number of clauses resolved remaining in a state of flux between 210 and 217. To guide the algorithm toward a more stable convergence on an extremum, a change will be made to the final temperature parameter of the simulated annealing process, adjusting it from 10 to 100. This alteration is aimed at allowing a more tempered and progressive reduction in temperature, thereby enhancing the algorithm's ability to methodically seek out and approach an optimal solution.

The second run: T = 100, e = 100, a = 0.9

In this set of runs conducted, success was again not achieved. From the data observed from the plots, it is clear that the algorithm is making progress toward a local optimum, yet there is still potential for further refinement. To enhance the algorithm's performance, a strategy will be implemented to extend its exploration phase at each temperature setting. This approach involves increasing the length of equilibrium from 100 to

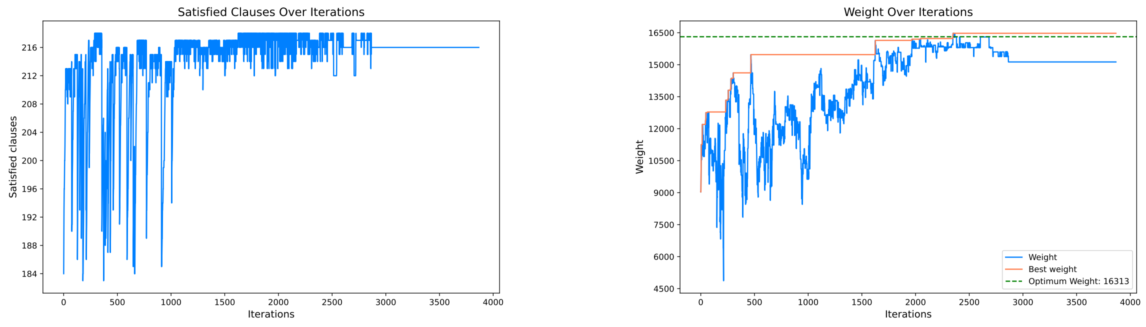


Figure 2: Second run results.

300, allowing for a more in-depth and prolonged search for optimal solutions at each temperature stage. This adjustment is expected to provide the algorithm with a greater opportunity to thoroughly explore and potentially escape local optima, thereby improving its chances of finding more effective solutions.

The third run: T = 100, e = 300, a = 0.9

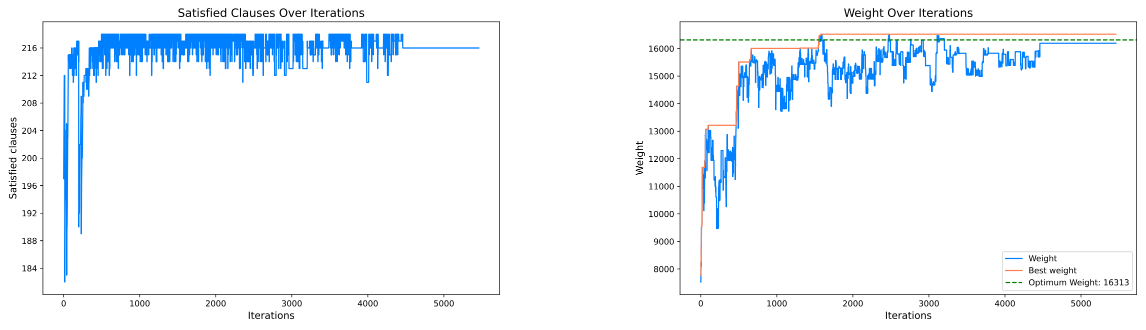


Figure 3: Third run results.

From the plots of the third run, it was observed that successful outcomes were still not achieved. This situation necessitated a reevaluation of the algorithm's implementation, especially considering that the local optimum attained bore a higher cost than a real solution. This pattern suggested a tendency of the algorithm to prioritize finding the highest weight over actually solving the problem. It is necessary to modify the implementation of the algorithm to achieve better results.

3.2 Algorithm adjustments

Fitness function

Further analysis revealed that the cost calculation function lacked penalization, leading to configurations that didn't solve all clauses but still had a high cost, making them more favorable to the algorithm. To address this, a penalization for solutions that failed to satisfy all clauses was added.

```
satisfied = get_satisfied_clauses_number(config)
unsatisfied = get_total_clauses() - satisfied

if unsatisfied is 0 then
    return total_weight_sum

scale_factor = get_total_clauses() / 8.0
penalty = get_avg_weight() * unsatisfied * unsatisfied
return total_weight_sum / scale_factor - penalty
```

Neighbour generation

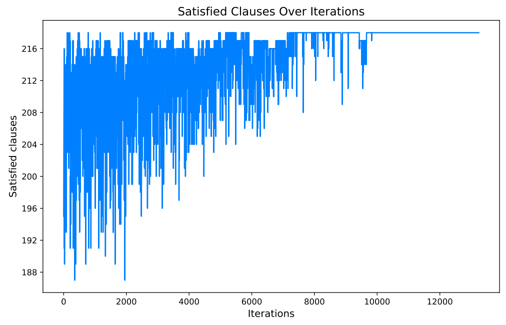
Additionally, to avoid getting stuck in local minima, if a problem remains unsolved and the number of iterations without improvement reaches a maximum, the counter resets. A significant value is added to the solution sum, providing the algorithm with a renewed direction and essentially a 'second life'. This approach helped to enhance the algorithm's ability to navigate through complex problem spaces.

```
num_vars = get_random_int(1, get_num_vars() / 2)

for i from 0 to num_vars - 1 do
    random_idx = get_random_int(0, get_num_vars() - 1)
    flip the value at random_idx in new_conf

return new_conf
```

The forth run: T = 100, e = 300, a = 0.9



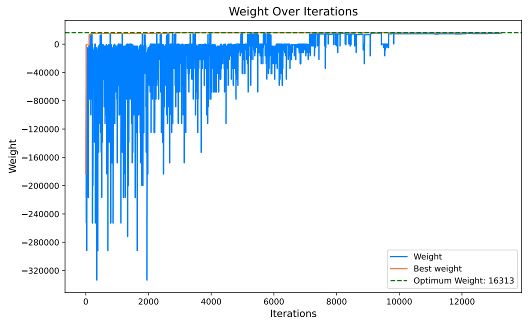


Figure 4: Forth run results.

Following the adjustments made to the functions, the results of the fourth run showed that the problem was solved in all ten attempts. However, only one run resulted in the optimal weight, indicating that there is still potential to improve the algorithm's performance. The cooling coefficient appears to be a key factor for further refinement. By tweaking this parameter, the aim is to improve the algorithm's ability to not just solve the problem but to do so in a way that consistently achieves the optimal weight across all runs. This fine-tuning of the cooling coefficient is the forthcoming step to enhance the efficiency and effectiveness of the algorithm's search process.

The temperature factor will be decreased based on plots because it wastes a lot of time in the end.

3.3 Factorial design for cooling coefficient

This section is dedicated to the determination of the optimal cooling coefficient, a crucial parameter influencing the rate of temperature decline and, consequently, the search strategy of the simulated annealing algorithm. A greater cooling coefficient results in a slower cooling process, affording the algorithm a comprehensive exploration capacity that may prevent premature convergence on local optima. Conversely, a lower cooling coefficient hastens the cooling, potentially leading to faster convergence but at the increased risk of missing the global optimum.

As the final step in fine-tuning the algorithm's parameters, the cooling coefficient's adjustment is pivotal. A factorial design approach has been employed for its systematic nature, allowing an array of values to be rigorously examined. This strategic testing is essential to identify a cooling coefficient that achieves an ideal equilibrium between in-depth exploration and computational expediency, complementing the fine-tuning completed on the other parameters.

The rationale for employing a factorial design lies in its ability to methodically evaluate the impact of varying cooling coefficients on the algorithm's efficiency. Implementing this approach provides a clear picture of performance across different settings, guiding the selection of the most suitable coefficient.

Moving forward, the discussion will extend to the input data and the results of the algorithm's runs.

Input data

The 3-SAT problem instances for measurement were obtained from the study materials available on the page[1]. The following sets from the SATLIB library were used in the experiment:

- `uf20-91R-{M,Q}` - 20 variables, 91 clauses, 10 instances.
- `uf20-218R-{N,R}` - 20 variables, 91 clauses, 10 instances.

The {M,N} sets are categorized as straightforward, with variable weights aiding the problem-solving process. Conversely, the {Q,R} sets are labeled as deceptive, potentially leading to entrapment in local optima.

3.4 Results

Based on careful examination of the data from Tables **??** and **??**, the choice of a 0.975 cooling coefficient for black box testing is well-founded, supported by its superior performance metrics. This coefficient not only demonstrated a higher frequency of solving the problem across various runs but also yielded the most optimal weight outcomes. Furthermore, it is characterized by a reduced relative error and a greater proportion of solved problems when averaged over multiple runs, underscoring its effectiveness.

[1]`https://courses.fit.cvut.cz/NI-KOP/download/index.html`

	wuf20-91							
	N-set				Q-set			
	solved	**optimal solved**	**average iterations**	**relative error**	**solved**	**optimal solved**	**average iterations**	**relative error**
0.9	1	0.99	13266	0.0004	1	0.92	13244	0.03
0.95	1	1	26574	0	1	1	26934	0
0.975	1	1	53721	0	1	1	54390	0
0.99	1	1	131981	0	1	1	134875	0

Table 1: Caption for the first table

	wuf50-218							
	M-set				R-set			
	solved	**optimal solved**	**average iterations**	**relative error**	**solved**	**optimal solved**	**average iterations**	**relative error**
0.9	0.86	0.24	12727	0.022	0.813	0.227	13200	0.165
0.95	0.953	0.46	24920	0.01	0.91	0.33	26689	0.11
0.975	1	0.68	46496	0.006	0.954	0.52	52495	0.07
0.99	1	0.82	105566	0.002	0.99	0.72	132191	0.03

Table 2: Caption for the second table

The final run: T = 50, e = 300, a = 0.975

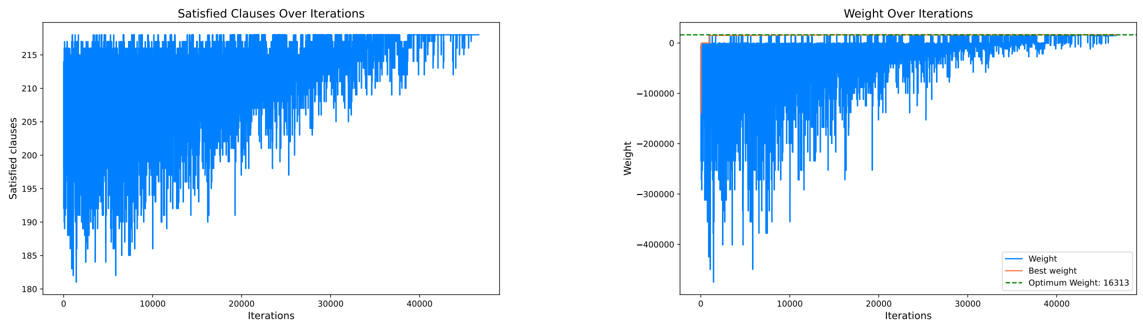


Figure 5: Final run results for **M/wuf50-0472.wcnf** instance.

The algorithm's extended diversification period is a strategic choice, justified by the highly random nature of the neighbor generation process. Such an approach is advantageous when tackling complex and deceptive problems. It provides the algorithm with ample opportunity to thoroughly explore the solution space, thereby enhancing its ability to escape local minima. This is crucial for stochastic algorithms like simulated annealing, which rely on a certain degree of randomness to bypass suboptimal solutions that could ensnare more deterministic approaches.

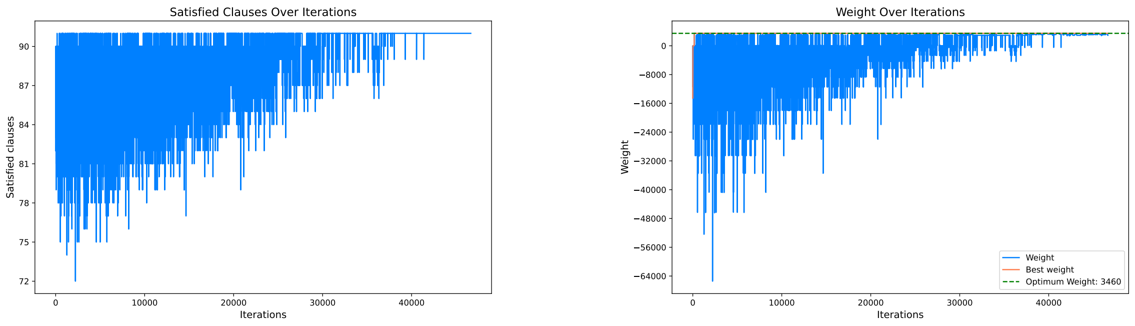


Figure 6: Final run results for **M/wuf20-0766.wcnf** instance.

3.5 Discussion

White box testing has provided valuable insights into the heuristic's functionality, with a focused examination of its behavior under varied parameter settings. The primary outcome of this phase was the effective calibration of key parameters — the length of equilibrium, the final temperature, and the cooling coefficient. These adjustments were pivotal in enhancing the search strategy, steering clear of local optima, and ensuring computational efficiency. The testing confirmed the significance of these parameters in the algorithm's ability to consistently find solutions.

The transition to black box testing will test the heuristic's adaptability to a wider array of problems, validating its robustness and the effectiveness of the improvements implemented during the white box testing stage.

4 Black box

This Section aims to assess the application's performance through black box testing. Unlike white box testing which delves into the internal workings of the algorithm, black box testing treats the algorithm as an opaque entity and focuses solely on its outputs. The following Subsections will describe the setup of the testing environment in detail.

4.1 Methods

Following the comprehensive white box testing detailed in Section 3, the optimal parameters for analysis have been determined as follows:

- `Cooling coefficient`: 0.975.
- `Equilibrium length`: 300.
- `Temperature factor`: 50.

The objective of black box testing is to evaluate the software's functionality based on its inputs and outputs. This approach intentionally disregards the internal mechanisms of the application, focusing instead on its behavior and compliance with the specified requirements.

Input data

The problem instances used for black box testing were sourced from the same repository as those for white box testing, as outlined in Subsection 3.3. However, for black box testing, the sets were not reduced and comprised various combinations of clauses and variables. This diversity in problem size and complexity was intended to rigorously evaluate the algorithm's performance across a range of different scenarios

- `uf20-71-{M,N,Q,R}` - 20 variables, 71 clauses, 1000 instances.
- `uf20-91-{M,N,Q,R}` - 20 variables, 91 clauses, 1000 instances.
- `uf50-218-{M,N,Q,R}` - 50 variables, 218 clauses, 1000 instances.

4.2 Metrics

In the realm of algorithm analysis, the selection of appropriate metrics is crucial. These metrics offer a quantitative measure of an algorithm's performance.

The following key metrics have been employed to compare the algorithms:

- `Number of solved problems.`
- `Number of problems with optimal solution.`
- `Average number of iterations.`
- `Average relative error.`

Experiments

The algorithm was run 10 * 1000 * 12 = 120000, 10 times for each problem.

The entire process, from executing the two algorithms to calculating statistics and plotting graphs, was automated using Python. This approach offers flexibility in making adjustments to the programs. All output data was written into files in the appropriate format.

4.3 Results

		total	**total solved**	**optimal solved**	**average iterations**	**relative error**
wuf20-71	M set	10000	1.0	1.0	59595	0.0
	N set	10000	1.0	1.0	59851	0.0
	R set	10000	1.0	1.0	59533	0.0
	Q set	10000	0.995	0.995	59491	0.0
wuf20-91	M set	10000	1.0	1.0	58038	0.0
	N set	10000	1.0	1.0	58212	0.0
	R set	10000	0.99	0.99	58579	0.0
	Q set	10000	0.99	0.99	58591	0.0
wuf50-218	M set	10000	0.971	0.681	49138	0.008
	N set	10000	0.969	0.63	48586	0.007
	R set	10000	0.966	0.56	56891	0.073
	Q set	10000	0.966	0.582	56864	0.063

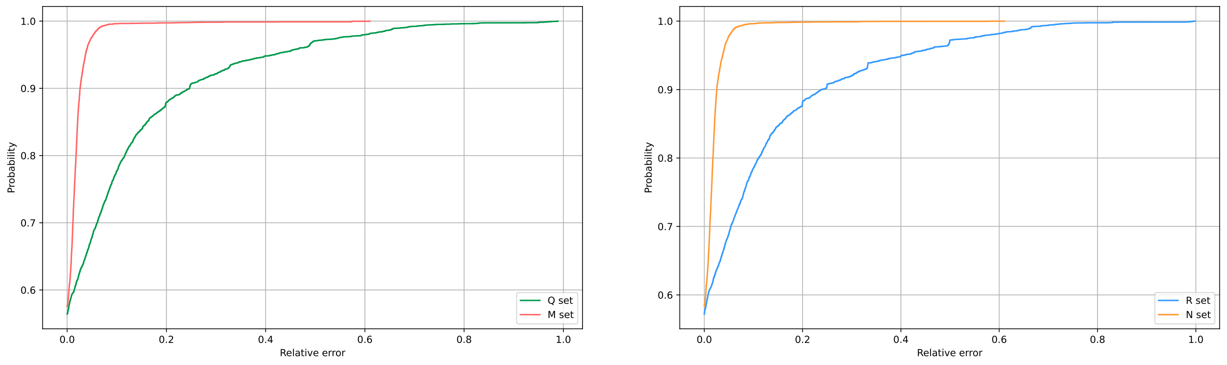


Figure 7: Emperical cumulative distribution function for the relative error from the wuf50-218 set.

4.4 Discussion

Black box testing, in contrast to white box testing, evaluates the algorithm's effectiveness solely on its output, without consideration of internal processing. The tests were performed on a wide variety of instances, encompassing different complexities and sizes.

The results indicated a notable variance in the algorithm's performance based on the instance complexity. Smaller instances with 20 variables showed high accuracy and efficiency, while larger instances with 50 variables presented challenges, with success rates dipping below 50% mark. Such outcomes suggest that while the algorithm performs reliably on less complex problems, its efficacy diminishes as complexity increases.

Figure ?? displays two graphs depicting the relative error distribution across varied sets of problems. In the case of direct, non-deceptive problems, the results are promising, with most problems demonstrating a minimal relative error, implying that the program was generally close to finding the optimal solution. On the other hand, when dealing with deceptive problems, the outcome is less favorable; a substantial portion of these problems yielded a relatively high error, suggesting that the program faced challenges in approximating the optimal solution accurately.

5 Summary

This work encompassed the design, implementation, and rigorous testing of a simulated annealing algorithm adapted for the MWSAT optimization problem. Throughout the white box testing phase, the algorithm was fine-tuned, leading to critical adjustments in key parameters such as the cooling coefficient, equilibrium length, and temperature factor. These modifications were grounded in a systematic approach to optimize performance and were carried forward into the black box testing phase.

The subsequent black box testing provided a broader perspective on the algorithm's generalizability across various problem sets. The automated testing offered a clear view of the algorithm's capabilities.

Looking ahead, there are several avenues for future work:

- Further refinement of the algorithm to enhance its robustness on more complex instances.
- Exploration of additional heuristics and metaheuristics for comparison and potential hybridization.

In summary, the developed algorithm exhibits promising results, particularly for simpler instances. Its current performance on more intricate problems provides a valuable foundation for further research and development. With targeted improvements, there is potential to elevate the overall success rate, which currently stands at approximately 69%.

Personal Reflection

Reflecting on the results obtained, the algorithm's performance aligns with expectations for simpler problems but leaves room for enhancement in handling larger, more intricate instances. The experience gained from this iterative testing and evaluation process is invaluable, offering a clear direction for ongoing algorithmic refinement. The potential improvements identified through this study instill confidence that with persistent effort, the success rate can be improved, making the algorithm a robust tool for MAXSAT and similar optimization challenges.

References

[1] W. Ben-Ameur, “Computing the initial temperature of simulated annealing,” *Computational Optimization and Applications*, vol. 29, pp. 369–385, 12 2004.