

A NOVEL META-HEURISTIC FOR GRAPH COLORING PROBLEM : SIMULATED
ANNEALING WITH BACKTRACKING

by
Buse Yılmaz

Submitted to the Institute of Graduate Studies in
Science and Engineering in partial fulfillment of
the requirements for the degree of
Master of Science
in
Computer Engineering

Yeditepe University
2011

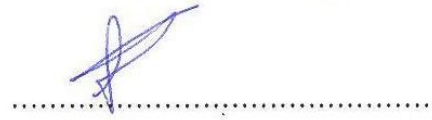
A NOVEL META-HEURISTIC FOR GRAPH COLORING PROBLEM : SIMULATED
ANNEALING WITH BACKTRACKING

APPROVED BY:

Assoc. Prof. Dr. Emin Erkan KORKMAZ
(Thesis Supervisor)



Prof. Dr. Linet ÖZDAMAR



Dr. Yaşar SAFKAN



DATE OF APPROVAL:/...../2010

ACKNOWLEDGEMENTS

I would like to thank Assoc. Prof. Dr. Emin Erkan Korkmaz for his support and mentorship during the preparation of my thesis. It was a great opportunity for me to work with him.

I would also like to thank Prof. Dr. Linet Özdamar and Dr. Yaşar Safkan for serving on my thesis committee and their valuable advises to make this thesis better. Also I deeply appreciate the help and advises of Asst. Prof. Dr. Dionysis Goularas and Dr. Mustafa B. Mutlu during the preperation of the presentation of the thesis.

Last but not least, I would like to express my sincere gratitude to my mother who stood by me whenever I needed her. Without her endless patience and support, I would not be able to achieve finishing my thesis.

To my beloved mom; my mentor and my best friend.

ABSTRACT

A NOVEL META-HEURISTIC FOR GRAPH COLORING

PROBLEM : SIMULATED ANNEALING WITH BACKTRACKING

Graph coloring problem (GCP) is one of the most extensively studied combinatorial optimization problems. Many algorithms have been proposed to solve GCP efficiently. It has been proved that hybrid algorithms are competitive with their pure counterparts as they yield promising results. This thesis presents a new meta-heuristic named as *Simulated Annealing with Backtracking (SABT)* for solving GCP. The algorithm proposed combines simulated annealing approach (SA) with a backtracking mechanism. SABT is a hybrid general purpose algorithm designed to solve any grouping problem. It does not exploit any domain-specific information. Several tests have been run on a collection of benchmarks from DIMACS challenge suite and promising results are obtained. A comparison of SABT with some other state-of-the-art algorithms is also presented along with a performance analysis of the algorithm.

ÖZET

ÇİZGE BOYAMA PROBLEMİ İÇİN YENİ BİR SEZGİ ÖTESİ ALGORİTMA: GERIYE DÖNÜŞ YÖNTEMİYLE BENZETİLMİŞ TAVLAMA (GDBT)

Çizge Boyama Problemi (ÇBP) üzerinde en yaygın çalışılan tümleşik optimizasyon problemlerinden biridir. ÇBP'yi etkin şekilde çözmek için birçok algoritma tasarlanmıştır. Hibrit algoritmaların elde ettiği umut verici sonuçlar, bu algoritmaların standard teknik ve yaklaşımlar kadar iyi olduğunu kanıtlamaktadır. Bu tezde, ÇBP'yi çözmek için, yeni bir üst-sezgisel algoritma olan *Geriye Dönüş Yöntemiyle Benzetilmiş Tavlama (GDBT)* yöntemi geliştirilmiştir. Tasarlanan algoritmada benzetilmiş tavlama tekniği (BT) ile geriye dönüş yöntemi birleştirilmiştir. GDBT, gruplama problemlerini çözmek için tasarlanmış hibrit ve genel amaçlı bir algoritmadır ve bu algoritmada tanım kümesine özgü bilgi kullanmaz. DIMACS challenge suit'ten birçok kıyaslama noktası örneği üzerinde yapılan testlerde umut verici sonuçlar elde edilmiştir. Diğer yandan, GDBT'nin en son gelişmeleri yansıtan birçok algoritmayla karşılaştırması ve algoritmanın performans analizi de tezde sunulmuştur.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS	iii
ABSTRACT	v
ÖZET	vi
LIST OF FIGURES	viii
LIST OF TABLES	xi
LIST OF SYMBOLS/ABBREVIATIONS	xii
1. INTRODUCTION	1
2. LITERATURE SURVEY	4
2.1. Graph Coloring Problem.....	4
2.2. Graph Coloring as a Grouping Problem	5
2.3. Overview of the Solution Methodologies for Graph Coloring.....	7
3. SIMULATED ANNEALING ALGORITHM	14
4. BACKTRACKING ALGORITHM.....	18
5. SIMULATED ANNEALING WITH BACKTRACKING	21
5.1. Main scheme and evaluation function	22
5.2. Individual Structure in SABT	26
5.3. Construction Mechanism	28
5.4. Backtracking Mechanism.....	30
5.5. Initial Version of SABT ($SABT_0$)	33
6. EXPERIMENTAL RESULTS	36
6.1. Problem instances and Experimental Details	36
7. CONCLUSION AND FUTURE WORK	68
REFERENCES	70

LIST OF FIGURES

Figure 2.1.	Example of coloring a graph.....	5
Figure 4.1.	Example of a tree representing the search space of all possible solutions ..	18
Figure 5.1.	Change of evaluation function based on <i>power</i>	25
Figure 5.2.	Structure of an individual	27
Figure 5.3.	Extraction from $V_{V_{set}}$	30
Figure 5.4.	Backtracking Amount of DSJC125.5	33
Figure 5.5.	Example of an individual in the old version of SABB	34
Figure 5.6.	Example of backtracking on an individual of $SABB_0$	35
Figure 6.1.	Utility value of DSJC125.5	46
Figure 6.2.	Backtracking Amount of DSJC125.5	47
Figure 6.3.	Utility value of DSJC125.9	48
Figure 6.4.	Backtracking Amount of DSJC125.9	49
Figure 6.5.	Utility value of DSJC250.1	49
Figure 6.6.	Backtracking Amount of DSJC250.1	50
Figure 6.7.	Utility value of DSJC250.9	50

Figure 6.8. Backtracking Amount of DSJC250.9	51
Figure 6.9. Utility value of DSJC500.5	51
Figure 6.10. Backtracking Amount of DSJC500.5	52
Figure 6.11. Utility value of DSJC1000.1	52
Figure 6.12. Backtracking Amount of DSJC1000.1	53
Figure 6.13. Utility value of DSJC1000.5	53
Figure 6.14. Backtracking Amount of DSJC1000.5	54
Figure 6.15. Utility value of DSJR500.1	54
Figure 6.16. Backtracking Amount of DSJR500.1	55
Figure 6.17. Utility value of R250.5.....	55
Figure 6.18. Backtracking Amount of R250.5	56
Figure 6.19. Utility value of le450.15b.....	56
Figure 6.20. Backtracking Amount of le450.15b	57
Figure 6.21. Utility value of le450.25c	57
Figure 6.22. Backtracking Amount of le450.25c	58
Figure 6.23. Utility value of flat300.20.....	58

Figure 6.24. Backtracking Amount of flat300.20	59
Figure 6.25. Utility value of school1_nsh	59
Figure 6.26. Backtracking Amount of school1_nsh	60
Figure 6.27. Utility value of fpsol2.i.2	60
Figure 6.28. Backtracking Amount of fpsol2.i.2	61
Figure 6.29. Utility value of inithx.i.2	61
Figure 6.30. Backtracking Amount of inithx.i.2	62
Figure 6.31. Utility value of mulsol.i.1	62
Figure 6.32. Backtracking Amount of mulsol.i.1	63
Figure 6.33. Utility value of mulsol.i.4	63
Figure 6.34. Backtracking Amount of mulsol.i.4	64
Figure 6.35. Utility value of zeroin.i.1	64
Figure 6.36. Backtracking Amount of zeroin.i.1	65

LIST OF TABLES

Table 6.1. Best colorings for SABT	38
Table 6.2. User-defined parameters	39
Table 6.3. Comparison of SABT with population based algorithms and other algorithms	41
Table 6.4. Comparison of SABT with local search algorithms.....	43
Table 6.5. Comparison of SABT with hybrid algorithms.....	44
Table 6.6. Comparison of initial version of SABT ($SABT_0$) and the current version ..	66
Table 6.7. Comparison of evaluation functions	67

LIST OF SYMBOLS/ABBREVIATIONS

$backtrackingAmount$	The number of vertices to be removed from the individual
$elementCount$	The number of vertices in an individual(utility value)
$elementCount_{current}$	The $elementCount$ of the current individual
$elementCount_{old}$	The $elementCount$ of the old individual
$Ind_{current}$	The current individual constructed by SABT
Ind_{old}	The previous individual constructed by SABT
$iterCnt$	Iteration count of SABT
$iterCnt_{max}$	Maximum iteration count of SABT
k_B	Boltzman constant
$lowerBound$	The lower bound of the range used when $backtrackAmount$ is <i>zero</i>
$power$	parameter used in the evaluation function
$randIdx$	Random value selected between 1 and $elementCount$
T	Temperature T used in SA
T_0	initial value that temperature T gets
T_{max}	maximum value that temperature T can get
$upperBound$	The upper bound of the range used when $backtrackAmount$ is <i>zero</i>
V_{Cset}	Set containing conflicting vertices
V_{Vset}	set containing separators and uncolored vertices
$Z(T)$	Normalization factor in Boltzman distribution
ΔE	The difference in the energies of two states in SA
$exp(-E/k_B T)$	Boltzman factor
ϑ	The set of uncolored vertices
CSP	Constraint satisfaction problem
CX	Crossover
DFS	Depth-First search
DSATUR	Largest saturation degree heuristic

EVA	Evolutionary Annealing Algorithm
FAP	Frequency assignment problem
GA	Genetic algorithm
GCP	Graph coloring problem
GLS	Genetic local search
GRASP	Greedy randomized adaptive search procedure
HEA	Hybrid evolutionary algorithm
ILS	Iterated Local Search
LS	Local search
MOGA	multi-objective genetic algorithm
PSA	Parallel Simulated Annealing
RLF	Recursive largest first heuristic
SA	Simulated annealing
SLS	Stochastic local search
<i>SABT</i>	Simulated annealing with backtracking
<i>SABT₀</i>	Old version of SABT
TS	Tabu search
VNS	Variable neighborhood search

1. INTRODUCTION

Solving grouping problems is accepted to be challenging as these problems belong to the set of NP-Complete problems [1]. The aim of these problems is to partition a given set of items into a number of different sets. Generally there is set of constraints which has to be satisfied while partitioning the items into these sets. There are many grouping problems such as graph coloring, bin packing and Knapsack problem.

Graph coloring problem (GCP) is one of the most extensively studied NP-complete problems [1]. Given an undirected graph $G = (V, E)$ where V is a set of vertices and E is a set of edges, GCP is a grouping problem in which the set V is partitioned into k independent sets and k is attempted to be minimized.

Many researchers have been studying graph coloring problem since 1980s. This combinatorial optimization problem is attractive for researchers for mainly two reasons. First of all, there are several real-world problems which could easily be reduced to the graph coloring problem. Besides, GCP is considered to be a difficult combinatorial optimization problem being NP-hard [2–4].

In this thesis, a new meta-heuristic algorithm named *Simulated Annealing with Backtracking* (SABT) is proposed to solve grouping problems. GCP is chosen as the testbed for the methodology proposed in this thesis. The algorithm utilizes simulated annealing (SA) algorithm as the local search mechanism while making use of a simple backtracking algorithm when the search is stuck. SABT is quite simple and efficient. It does not have a pre-processing step which is costly in terms of computational time. It is not a population based algorithm, thus at every iteration a single candidate solution (individual) is updated. The algorithm accepts candidate solutions only with a legal coloring (i.e. there are no conflicting vertices in a candidate solution). SABT simply constructs the groups of an individual by randomly selecting vertices from the vertex set. A randomly selected vertex can be placed in a group if it does not conflict with any of the elements of that group. When it is impossible to insert any more vertices into the groups of the individual, the backtracking

mechanism removes vertices of some groups by backtracking to a point in the individual and then the individual is reconstructed. Hence, the algorithm constructs individuals with a variable length at every iteration.

Simulated annealing approach is used to decide whether to accept or reject the reconstructed individual. The reconstructed individual is either a better or a worse candidate solution than the current individual. An individual with more vertices is a better individual. A better individual is always accepted by the algorithm, whereas, a worse individual is accepted with a probability determined by the SA approach. We also propose a new exponential function for the cooling down schedule in the SA process. It is a simple exponential function which avoids heavy computations, contributing to the performance of the algorithm. Backtracking mechanism also makes use of the function proposed for the cooling down schedule used in simulated annealing. This time, the function is utilized to determine the amount of backtracking. This approach provides a balance between diversification (exploration of the search space) and intensification (exploitation of the previous solutions). The rate of backtracking decreases in time, intensifying the search on immediate neighbors of the current solution.

The instances used in studies for GCP are chosen from the DIMACS Challenge Suite. Some of these instances are benchmarks including randomly created graphs, Leighton graphs [5] and register allocation problems in real codes. For graph coloring problem, some of the randomly created large graphs and some Leighton graphs are proved to be difficult to tackle. Some algorithms use domain-specific information which is extracted from the graph in order to deal with the difficult instances [6, 7]. This information is then exploited to enhance the algorithm [8–10]. An important attribute worth mentioning about SABB is that it is designed as a general-purpose algorithm for grouping problems as it does not exploit any domain-specific information. Hence, SABB could be utilized for any grouping problem.

In addition, many researchers utilize an initialization phase in their algorithms as in [11, 12]. SABB does not use any initialization phase for dealing with the large instances.

In this study, several tests have been run on a collection of benchmark graphs from

the DIMACS Challenge Suite. The results match many of the best solutions presented in the literature. Thus, it is proved that the algorithm is competitive with other state-of-the-art algorithms.

This thesis is organized as follows; Chapter 1 gives an overview of the thesis. In chapter 2, the problem that is solved is introduced in detail and an overview of the solution methodologies in the literature is provided. Chapter 3 gives a detailed explanation of the stochastic local search algorithm (SLS) that is utilized in the algorithm. In the following chapter, the search method which is combined with this SLS is explained in detail. In chapter 5, the algorithm proposed to solve the problem at hand is provided in detail. The general algorithm, several algorithmic and implementation details are given in this chapter. Chapter 6 provides the problem instances used in the experiments, experimental results and comparison of the results with results of some state-of-the-art algorithms. In addition, a brief comparison with the previous version of the algorithm proposed in this thesis is given in this chapter. Chapter 7 presents the conclusion, comments on the thesis and the future work.

2. LITERATURE SURVEY

2.1. GRAPH COLORING PROBLEM

Graph coloring problem (GCP) is one of the well-known combinatorial optimization problems. Its significance, apart from being an NP-complete problem [1] comes from its easiness to be utilized to model several real-world applications. Many applications such as timetabling [13, 14], frequency assignment problem (FAP) [15, 16], register allocation [17], air traffic flow management [18] and satellite range scheduling [19] are modeled using GCP.

Given an undirected graph $G = (V, E)$ where V is a set of vertices and E is a set of edges, the vertices of the given graph G are attempted to be colored with a minimum number of colors (k). While coloring the graph, no adjacent vertices can be colored with the same color. Two vertices are considered to be *adjacent* if there is an edge connecting them. When adjacent vertices have the same color they are considered to be in *conflict*. Thus, solving GCP is the attempt to color a given graph G with the minimum possible number of colors while avoiding any conflicts.

A more formal definition is as follows;

Given $G = (V, E)$ and k to be the minimum number of colors, a k - *coloring* of G is a function $c : V \rightarrow 1, \dots, k$ where $c(x)$ of a vertex x is called the color of x . A color class V_r is defined by the vertices with color r with the constraint $1 \leq r \leq k$. When vertices x and y which are adjacent have the same color r , they are in conflict. Hence, the edge between them $e_{x,y} \in E$ is a conflicting edge. If the k - *coloring* does not have any conflicting vertices then it is a valid coloring. Hence, GCP is to determine the minimum integer k (the chromatic number $\chi(G)$) such that there exists a legal k - *coloring* of G [20].

2.2. GRAPH COLORING AS A GROUPING PROBLEM

In the grouping problem, a set of items has to be partitioned into mutually disjoint subsets (stable sets). V_i of V such that $V = V_1 \cup V_2 \cup V_3 \dots \cup V_N$ and $V_i \cap V_j = \emptyset$ where $i \neq j$ [21].

In most of the grouping problems, not all possible groupings are permitted. A set of constraints has to be complied with, in order to have a valid k -coloring. The objective is to optimize a cost function defined over a set of valid groupings.

Graph Coloring Problem (GCP) (also known as *vertex coloring problem*) could be considered as a grouping problem. Given a graph $G = (V, E)$, being V the vertex set and E the edge set, the set of vertices V are partitioned into k number of groups and the objective is to minimize k . The constraint for GCP is that the groups are formed such that only vertices not sharing an edge (without a conflict) are placed into the same group. In other words, vertices of the graph G are colored with a minimum number of colors (the chromatic number $\chi(G)$) where each color represents a group and no two adjacent vertices are colored with the same color.

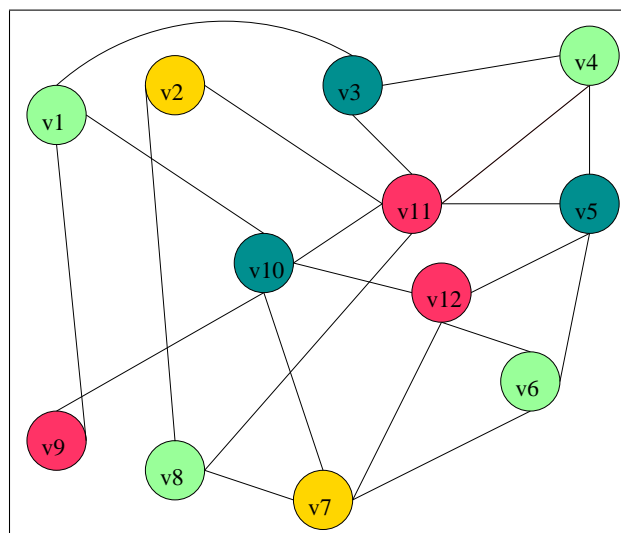


Figure 2.1. Example of coloring a graph

In the example given in Figure 2.1, a small graph with twelve vertices is colored using

four colors. As seen in the figure, no adjacent vertices have the same color. Hence, there are no conflicting vertices and this is a *valid* coloring. The minimum number of colors used to color this graph is *four* since it is a *dense* graph having many vertices connected to each other with an edge. Unlike *sparse* graphs, more colors are needed to color dense graphs to avoid having conflicting vertices. Hence, dense graphs are difficult to tackle.

The graph coloring problem could be solved as a constraint satisfaction problem. From this perspective, it is equal to solving a series of k -coloring problems [22]. The algorithms solving GCP with this approach, start with a number of colors $1 \leq k \leq |V|$ and solve the k -coloring problem. Once it is solved, algorithm tries to solve the k -coloring problem for $k - 1$ colors until no legal coloring could be found [9]. There are basically three approaches to vertex coloring based on this perspective. The third approach on which SABL is based, is rather rare among researchers. Before explaining these approaches, let us give some basic definitions that will be used in the thesis;

A *stable set* is defined to be a set of mutually non-adjacent vertices. Then *k-coloring* can be defined as the partition of the vertex set V into k stable sets S_1, \dots, S_k which are called color classes. Hence, a k -coloring is a potential solution to the problem at hand.

A solution containing no conflicting vertices is a *valid (proper)* solution. Any constructed solution, which is referred to as *(valid) candidate solution* is a *partial solution*. A candidate solution that colors the graph using k number of colors is called a valid k -coloring. Hence, any candidate solution that SABL constructs is a valid partial k -coloring which is simply referred to as k -coloring throughout the thesis. An improper k -coloring may contain conflicting vertices. Given a fixed integer k , the optimization problem *k-GCP* is the attempt to determine a k -coloring of G with a minimum number of conflicting edges. If the optimal value of the *k-GCP* is zero, then G has a valid (legal) k -coloring. Also *individual* and *partial solution* will be used interchangeably throughout the thesis.

The first and most common approach is to produce k -colorings while attempting to decrease k . In the second approach, the algorithm starts with an improper k -coloring. Then the number of conflicts are tried to be reduced to zero. If the algorithm succeeds, it is restarted

with an improper $(k - 1)$ coloring. If no k -coloring is found, the algorithm is restarted to search for a $(k + 1)$ coloring. The stopping criteria is to consider a k twice. Thus, the output solution is a k -coloring with the smallest k possible. The last approach uses partial k -colorings which consists of k mutually disjoint sets and a set of uncolored vertices. Thus, a partial k -coloring contains no conflicting vertices. The goal is to increase the size of the partial solution by coloring the uncolored vertices. Although this approach is rare, there are algorithms utilizing this methodology [12,23]. The algorithm presented in this study is based on this approach. For a detailed explanation about these approaches, one can refer to [12].

2.3. OVERVIEW OF THE SOLUTION METHODOLOGIES FOR GRAPH COLORING

Exact algorithms are able to color small graphs with at most 100 vertices [24]. For larger graphs, more sophisticated methods are needed. Many algorithms have been developed and many more have been utilized together (hybrid algorithms) to create more general, robust and efficient algorithms. Heuristics and meta-heuristics have been widely utilized to attack GCP. These methods could be classified under three categories; sequential construction, local search and population based search.

The first heuristics developed for GCP belong to the sequential construction category. These heuristics mainly have a greedy approach. Given a permutation of the vertices, these heuristics attack the problem by trying to color the vertices of the graph one by one using a greedy function. Unlike these heuristics, *largest saturation degree* heuristic (DSATUR) [25] and *the recursive largest first* heuristic (RLF) [5] generate permutations of vertices in a graph G dynamically. Among constructive heuristics with a greedy approach, both DSATUR and RLF algorithms choose the vertex with the highest number of differently colored adjacent vertices [12].

Another heuristic of the same approach is XRLF [5,26]. This method which is utilized to extract stable sets, combines exhaustive search with a variant of RLF algorithm. Although these are fast algorithms, their efficiency is not satisfactory in terms of solution quality.

For better solutions, local search based meta-heuristics such as tabu search (TS) [27, 28] and simulated annealing (SA) [26, 29, 30] have been utilized. These methods which belong to the second category, have extensively been utilized by many researchers especially in hybrid algorithms as a local search operator.

Tabucol is a well-known tabu search algorithm that was proposed in 1980s [27]. It is developed and improved by several researchers [28, 31]. *Tabucol* accepts solutions with conflicting edges (edges with both vertices having the same color) and uses penalties for these solutions. *Tabucol* has been utilized in many hybrid algorithms as the local search mechanism.

Simulated annealing is among the first meta-heuristic approaches utilized to solve GCP. It is first applied to GCP by Chams et al [30] and Johnson et al. [26] who intensively tested SA on random graphs. It is a famous local search method based on the physical process of heating a solid to a high temperature and cooling it down gradually. The algorithm is based on Monte Carlo Method. At each step, the current state is compared to the next state trying to bring the system to a state with minimum possible energy. The algorithm decides probabilistically whether to accept or reject the new state according to *Boltzman Distribution*. Hence, the system tends to move to states with lower energy [29].

The first local search algorithm for GCP was proposed in 1987 by Chams et al. [30]. The algorithm operates on k-colorings that are not necessarily legal and the number of conflicts in the individual is attempted to be minimized. The color of a single vertex is changed in each iteration. SA algorithm is utilized as the local search mechanism.

The following study on GCP was done by the same authors either. It is a two phase algorithm. The first phase is a preprocessing phase to extract stable sets in the graph. In the second phase, again, SA algorithm is utilized. The results obtained were better for large graphs [22].

Short after this study, Hertz and Werra implemented a new local search algorithm while keeping the solution space, neighborhood and objective function the same. Again, the

algorithm has two phases. But instead of a greedy approach, this algorithm utilizes a tabu search algorithm to extract the stable sets [22].

Other local search meta-heuristic methods include Iterated Local Search (ILS) [11], Reactive Partial Tabu Search [12], *Greedy Randomized Adaptive Search Procedure* (GRASP) [32], Variable Neighborhood Search [20], Variable Space Search [33] and Clustering-Guided Tabu Search [8].

Iterated Local Search algorithm is a simplified version of the Variable Neighborhood Search algorithm. In this algorithm, the perturbations (neighbors) of the current search point are exposed to the local search which is done iteratively leading to a random walk in the space of the local optima [34].

Reactive Partial Tabu Search method is designed to solve k-GCP problem. Tabu search algorithm is utilized in which the tabu list reacts to the oscillations of the objective function. The authors proposed two new algorithms namely PARTIALCOL FOO-PARTIALCOL. They claim could be an alternative to TABUCOL and its variations. FOO-PARTIALCOL is proposed to improve the performances of both TABUCOL and PARTIALCOL. The tabu tenure of the algorithm is able to adjust itself depending on both the graph and the state of the search. The algorithm proposed is a general purpose algorithm not relying on the problem definition. The results obtained from the experiments are promising [12].

GRASP is used to color sparse graphs. Although it is a fast algorithm, the results are far from being optimal. It has two phases, namely construction and improvement phase. In the construction phase, a randomized version of the RLF algorithm is utilized for the generation of the initial colorings. In the improvement phase, a local search method is utilized to find improved neighbor solutions [32].

In Variable Neighborhood Search (VNS) method, various neighborhoods are utilized instead of a single neighborhood. The method attempts to escape from local minima by utilizing more than one neighborhood [20]. The local search method utilized in the algorithm is either *Tabucol* or SA. Experiments show that the results are better when compared to a pure

implementation of the *Tabucol* algorithm.

Variable Space Search is another local search methodology which utilizes more than one search space. It is an extension of VNS method. In addition, more than one objective function is considered. The algorithm moves from one neighborhood to another when the search is stuck at a local optimum. The problem at hand is solved by combining different formulations of the problem which differ from each other in terms of constraints. While some constraints are possibly relaxed in one search space, they are always satisfied in another. The results are competitive with other hybrid evolutionary algorithms. The algorithm is applied on the k-coloring problem [33].

For Clustering-Guided Tabu Search, two different algorithms, namely TS-Div and TS-Int which are both based on *Tabucol* are implemented in [8]. TS-Int is a second stage algorithm exploiting the colorings that TS-Div found. TS-Div performs a search space analysis on the spatial distribution of the high quality configurations (local optima) obtained by *Tabucol*. The authors introduce a *clustering* hypothesis: The high quality solutions are not randomly scattered in the search space, but rather grouped in clusters within spheres of specific diameter. The results are competitive for difficult graphs.

Although these heuristics are favored, their inability of solving some instances efficiently has weakened their reputation. These methods have a low performance on some large random graphs [22]. Thus, several approaches have been proposed to deal with these difficult instances [22] also resulting in the emergence of a third group of methods.

The third category which is highly favored among researchers recently includes population based algorithms [21, 35–37] and evolutionary hybrid algorithms [31, 38–40]. A well-known example of population based algorithms is the genetic algorithm (GA). A population is created and genetic operators such as mutation and crossover are applied to evolve the candidate solutions. In a genetic algorithm, mutation resembles a random walk in the search space, whereas crossover passes the genetic material from the current population to the next generation. The genetic operators preserve the diversity of the individuals in the population while exploring the search space.

An implementation of genetic algorithm to solve GCP first appeared in the early 90s [22]. In this implementation, the solutions are encoded as permutations of vertices of the graph G . A greedy algorithm is utilized to color the vertices of a solution sequentially giving each vertex the first color still available. Unfortunately, the results of the experiments are of poor quality [22].

Another example is provided in [41]. In this study, a genetic algorithm is applied on constrained optimization problems. Penalty functions are utilized to handle constraints. Since GCP can be solved as a constrained optimization problem, this algorithm can be utilized to solve the problem.

Second approach in this group is to embed a local search algorithm within the framework of an evolutionary algorithm. Among first studies of this approach, there is one done by Costa et al. [42] and Fleurent and Ferland [31]. They have developed a genetic local search algorithm (GLS) to color graphs. Just like the classical GAs, GLS operates on a population of solutions and it has a crossover operator. But instead of a random mutation operator, GLS utilizes a local search (LS) operator [22]. The algorithm also extracts stable sets from the graph leaving a residual graph to be colored.

Other implementations of GLS [39, 40] obtained better results on large graphs than their predecessors. In addition, GLS algorithms proposed in [39] and [40] do not have a preprocessing step to extract stable sets.

The technique of utilizing algorithms together (hybrid algorithms) has proved to be promising especially when dealing with very large random graphs. In addition, many studies have been conducted on the structure of the search space that exists in GCP with the expectation of using the information to improve the algorithm. An example to this approach is extracting several stable sets and then coloring the residual graph [5, 22].

Most of the recent graph coloring heuristics belong to either second or third group explained above. Generally, local search techniques are utilized in the framework of a population based algorithm. Population based algorithms are considered to be inefficient

algorithms due to the fact that the process of creating new individuals undergoes a considerable fitness calculation. The fitness value of an individual is the measure of how close this candidate solution is to the best solution that can exist for the current problem. By means of a selecting mechanism, individual(s) are chosen according to their fitnesses to create new individuals (offspring). But this is a costly operation affecting the overall performance of the algorithm.

On the other hand, local search methods are fast and efficient algorithms. They provide a fast searching mechanism in the search space moving from one solution to a neighbor solution. They do not need to undergo heavy fitness calculations as genetic algorithms. In most of the case, these are constructive algorithms in which the fitness of the candidate solution can be determined directly during the construction process. Several studies show that hybridization of population based algorithms with local search methods seems to be promising [40].

There are several studies in which similar approaches to that of SABB are utilized. A new algorithm based on Parallel Simulated Annealing (PSA) algorithm is presented in [43] to solve GCP. The algorithm proposed combines two popular approaches of parallelization of the classical SA algorithm. The first approach is to evaluate the current solution by calculating possible moves from one state to another on multiple processing units. The second approach is to compute independent solutions by multiple threads and exchange the obtained results on a regular basis. The algorithm uses multiple processors. The coordination of the algorithm is based on the master-slave model in which a processing unit is responsible for collecting and distributing data among slave units. Unfortunately the results are poor in quality.

Another similar approach is observed in [44]. The algorithm which is called FCNS is a hybridization of DSATUR [25] backtracker and IMPASSE local search algorithm [45]. It is applied on bandwidth multicoloring problem. The backtracking mechanism that FCNS uses is a randomized form of *Dynamic Backtracking* [46] where a previously assigned variable can be unassigned without unassigning those assigned after this variable. In this approach, the backtrack variables are selected randomly. Whenever the search is stuck, B backtrack

variables are unassigned where $B \geq 1$.

The next study we will mention [12], presents the algorithm FOO-PARTIALCOL which considers feasible and partial solutions only and tries to increase the size of the current partial solution. A solution consists of k disjoint stable sets and a set of uncolored vertices. As mentioned in the previous section, SABT uses the same approach for vertex coloring.

The Evolutionary Annealing (EVA) Algorithm [47] maintains a population of individuals and SA approach is utilized as a local search method. By starting SA with low temperatures, the algorithm exploits quickly any good solutions that exist in the neighborhood (intensification phase). On the contrary, when SA is started on high temperatures the diversity of the population is maintained allowing random jumps in the search space (diversification phase). In this study, SA framework is utilized in a similar way with SABT.

3. SIMULATED ANNEALING ALGORITHM

Recent developments in local search algorithms have been inspired by nature. Physical annealing of metals is one of the naturally occurring phenomena which led to a strongly improved algorithmic approach known as simulated annealing. Simulated annealing is a general powerful searching scheme. It can be applied to several problems to improve local search performance [48].

Simulated annealing approach is based on the physical process of heating a solid up by increasing the temperature to a maximum value and then cooling it slowly. When maximum temperature provided is sufficiently high and cooling down process is carried out sufficiently slowly, all particles arrange themselves in the low energy ground state of a corresponding lattice which are perfect crystal structures. Cooling needs to be done very slowly in order to avoid defects (irregularities) in the crystal which correspond to meta-stable sets in the model [48].

The simulated annealing process is started at the maximum temperature T and T is slowly decreased providing the solid to reach *thermal equilibrium* at each temperature value T . *Thermal equilibrium* is characterized by the probability of being in a state with energy E given by the *Boltzman distribution*.

$$Pr\{E = E\} = \frac{1}{Z(T)} * \exp\left(-\frac{E}{k_B T}\right) \quad (3.1)$$

where $Z(T)$ is a normalization factor depending on T , k_B is the *Boltzman constant* and $\exp(-E/k_B T)$ is the *Boltzman factor*.

As T converges to zero, *Boltzman distribution* concentrates on the states with lowest energy. The cooling down process must be sufficiently slow to allow the solid to reach *thermal equilibrium* for each temperature value [29].

Simulated annealing algorithm starts from a randomly created initial solution. In each

step a neighbor s' of the current solution s is randomly chosen (*proposal mechanism*), then an acceptance criterion which is parametrized by the temperature parameter T is used to decide whether to accept s' or stay at s . One standard choice for this acceptance criterion is a probabilistic choice according to the *Metropolis condition*. It was proposed as early as 1953 in [49].

An *annealing schedule (cooling schedule)* is a function that determines a temperature value $T(t)$. It is commonly identified with an initial temperature T_0 , a temperature update scheme, a number of search steps performed at each temperature and a termination condition. The initial temperature T_0 is generally determined by the properties of the given problem instance. Among the temperature update schemes, a common one is a geometric cooling schedule. In this update scheme, the temperature is updated as $T_{t+1} = \alpha T_t$. This update scheme is proved to be efficient in many cases [29, 50]. For the number of steps performed at each temperature, often a multiple of the neighborhood size is used. Among a variety of termination conditions, a specific one is based on the *acceptance ratio* which is the ratio of the proposed steps to accepted steps. The search process is terminated if *acceptance ratio* falls below a certain threshold or when no improving candidate solution has been found for a given number of search steps [51].

As an example, to simulate the evolution to *thermal equilibrium* of a solid, the classical simulated annealing method based on *Monte Carlo method (Metropolis Algorithm)* [49] is provided. In the following paragraphs, the simulated annealing method is introduced in detail, followed by a general pseudo code.

The algorithm generates a sequence of states of the solid. The current state of the solid is characterized by the positions of its particles. A randomly chosen particle is exposed to a small, randomly generated perturbation caused by a small displacement of the particle. If in the new state the solid has a lower energy, the process is continued with the new state. i.e. the difference in energy between the new state and the current state being ΔE , if $\Delta E \leq 0$ the process accepts the new state. if $\Delta E > 0$ then the probability of acceptance is given by $\exp(-\Delta E/k_B T)$. This acceptance criterion is called *Metropolis criterion*. Hence, the system evolves into *thermal equilibrium*.

A given *annealing schedule* (*cooling schedule*) is used to adjust the temperature T . Initially a control parameter c is given a high value representing T . Given a configuration i (representing a state of the solid), configuration j could be obtained by randomly choosing an element which belongs to the neighborhood of i . This process corresponds to the small perturbation in the *Metropolis Algorithm*. The configurations are the states of the combinatorial optimization problem at hand. Given $\Delta C_{ij} = C(j) - C(i)$ where C_i denoting configuration i , the probability of $C(j)$ being the next configuration in the sequence which is denoted by $Prob(C_j)$ is given below;

$$Prob(C_j) = \begin{cases} \Delta C_{ij} \leq 0 & 1 \\ \Delta C_{ij} > 0 & \exp(-\frac{\Delta C_{ij}}{c}) \end{cases} \quad (3.2)$$

This process is continued until the *thermal equilibrium* is reached. The algorithm terminates for a small value of c for which no deteriorations are accepted any more. The final *frozen* configuration is the solution of the current problem.

Below the algorithm of simulated annealing is given. The algorithm starts with an initial state(configuration). The *cooling schedule* is a series of temperature values of an annealing process provided to the algorithm. Variable t is set to the maximum value of the temperature T and it takes the values that *cooling schedule* provides. The next state (s') is chosen among the neighbors of the current state (s). The difference in the energy (ΔE) of s' and s is used in the decision phase of the algorithm. s' becomes the new state with a probability given by the *Metropolis condition*.

Algorithm 3.1. Simulated Annealing Algorithm

```
s ← initialState;  
t ←  $T_{max}$ ;  
while  $t \leq T_{max}$  do  
  t ← coolingSchedule(t);  
  if  $t = 0$  then  
    | return s;  
  end  
  s' ← randomly selected neighbor of s;  
   $\Delta E \leftarrow s'_{VALUE} - s_{VALUE}$ ;  
  if  $\Delta E \leq 0$  then  
    | s ← s';  
  else  
    | s ← s' with probability  $\exp(-\frac{\Delta E}{T})$ ;  
  end  
end
```

4. BACKTRACKING ALGORITHM

Backtracking algorithm is a method based on the famous brute force approach. It systematically searches for a solution among all available options. Beginning with a partial solution, it searches for an exact solution by extending the partial solution at every iteration. When the search is stuck, in other words, when the extended partial solution is not valid any more, the algorithm backtracks and tries to extend the partial solution in an alternative way.

In this method, a solution is created sequentially. Validity of a given constraint is checked when all the elements in the solution relevant to this constraint are inserted in the individual. A partial solution violates a constraint if it contains a element violating this constraint. If a partial solution violates any of the constraints, backtracking is performed to the most recently inserted element that still has alternatives available. Hence, backtracking algorithm is able to eliminate a subspace from the search space [52].

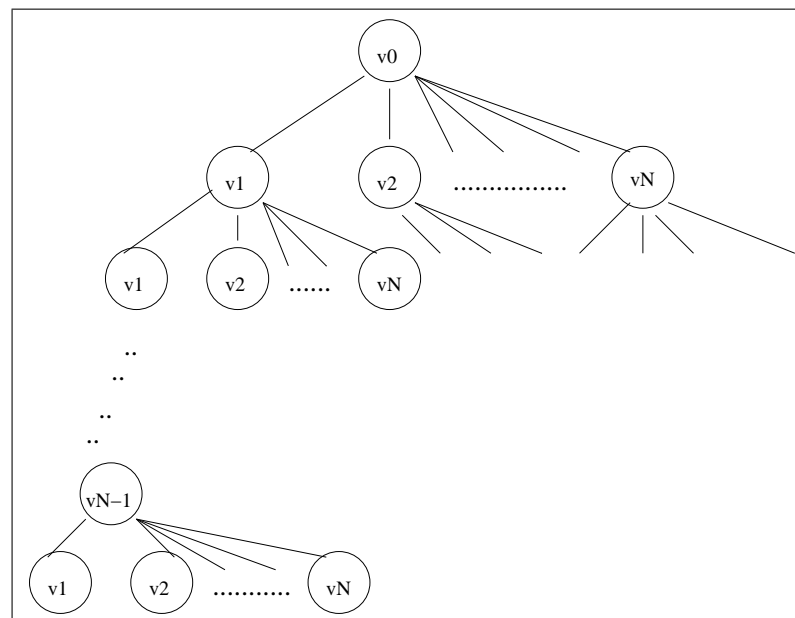


Figure 4.1. Example of a tree representing the search space of all possible solutions

Algorithm 4.1. Backtracking Algorithm

```

Func: BT( $v_i$ )
if  $v_i$  is a leaf node then
    if the leaf node is a goal node then
        | return true;
    else
        | return false;
    end
else
    foreach child  $v_{c_i}$  of  $v_i$  do
        if  $BT(v_{c_i}) == true$  then
            | return true;
        end
    end
    return false;
end

```

The algorithm for backtracking is provided in Algorithm 2. The search space of a problem consists of all possible solutions both valid and invalid. Thus, the search space can be constructed as a search tree which contains all the candidate solutions. The complete search space for n elements consists of n^n candidate solutions. Note that in Figure 4.1 the search tree has $\sum_{i=0}^n n^i$ nodes. Since backtracking is generally used to traverse a tree in a depth-first manner (DFS), it can be used to search in the search space for the exact solution to the problem at hand. DFS searches in the vertical direction on the tree rather than in the horizontal direction. Hence, the algorithm tends to search toward the leaf nodes on the tree.

The recursive algorithm provided constructs a partial solution by adding an element to the solution at every iteration and checks its validity. If the newly added element violates a constraint, the algorithm removes that element and tries to add another element to the solution. An element corresponds to a *node* in the search tree. *leaf node* is a node without any children. A child v_{c_i} of v_i is an element that is reachable from v_i where v_{c_i} is below v_i on the tree. *goal node* is the last element to be added to the solution in order to construct the complete solution to the problem.

Backtracking algorithm is extensively utilized in constraint satisfaction problem (CSP). Examples of CSP can be given as crosswords, verbal arithmetic and sudoku. Many modern algorithms designed to solve CSP [52–54] are simple backtracking-style algorithms with certain improvements [48]. Apart from CSP, backtracking method is also utilized to solve the famous N-queens puzzle.

5. SIMULATED ANNEALING WITH BACKTRACKING

The algorithm proposed in this thesis utilizes a backtracking algorithm within a framework based on simulated annealing (SA). It is called *Simulated Annealing with Backtracking (SABT)*. SABT combines simulated annealing algorithm with a backtracking mechanism and proposes a new hybrid local search algorithm to solve GCP efficiently. It has a simple design based on the classical simulated annealing algorithm. As noted in chapter 1, SABT can be utilized for any partitioning problem. In this thesis, it is only used to solve GCP.

Before introducing the algorithm briefly, let us remind a few terms used in the thesis; As discussed in the previous section, we refer to a valid partial (candidate) solution as a valid k -coloring of the graph. We should also note that, a set of uncolored vertices exists in the algorithm that contains vertices that the algorithm has not attempted to color yet and the conflicting vertices that cannot be inserted into the current individual.

The algorithm starts with a randomly created individual which is a valid k -coloring. Then, at each iteration, a new individual is constructed out of the previous one in the following way; A backtracking amount is calculated by a stochastic backtracking mechanism. Based on the backtracking amount, some of the groups are removed from the current individual and the vertices in these groups are put back into the set containing the uncolored vertices. Then the new individual is constructed by using the vertices in the uncolored vertices set. The simulated annealing framework is used to determine if the newly constructed individual will be accepted or not.

SABT operates on a single individual. The individual has a variable length, depending on how many of the vertices could be colored. The partial solution constructed is always a valid k -coloring without any conflicts. The number of vertices in the individual is used as the utility value. The more vertices in the individual, the higher its utility is, making it a better individual.

This chapter is divided into sections for an easy follow of the whole work. First, the general design of the algorithm and a simple mathematical definition of the approach are introduced. Then the specific details of the algorithm are presented.

5.1. MAIN SCHEME AND EVALUATION FUNCTION

SABT is initialized by the construction of a randomly created individual. It is created by inserting randomly selected vertices into groups until no more vertices can be inserted without causing conflicts. The conflicting vertices are put into the set of uncolored vertices. SABT algorithm carries out the search process by constructing a new individual exploiting the previous one.

As stated before, to construct a new individual, some of the vertices already placed in the current one are removed; The evaluation function of SA algorithm ($f(iterCnt)$) is also utilized to determine the number of vertices to be removed from the individual (backtracking amount). Hence, the backtracking operation is also carried out in a stochastic manner. The amount of vertices determined by the backtracking amount are removed from randomly selected groups of the individual and they are put back into the set of uncolored vertices. Then the new individual is constructed by extending the current individual with the uncolored vertices. At this point, SABT decides whether to accept the reconstructed individual or not using the SA approach. In this approach, if the reconstructed individual is better than the previous one, it is always accepted. If it is worse, it is accepted with the probability given by the evaluation function of SA. Current and next (reconstructed) individuals are compared using their utility values. Hence, the evaluation function of SA is utilized in both deciding whether to accept the newly constructed individual or not and in the calculation of the backtracking amount. That is why, it is more probable to have considerably large backtracking amounts at the beginning of the search while it gets smaller toward the end of the search. The algorithm terminates if one of the following termination criteria are met; Constructing an individual with $|V|$ number of vertices divided into k mutually disjoint sets (successfully coloring the graph with k colors) or maximum number of iterations has been reached. The general algorithm for SABT is given in Algorithm 5.1.1.

In Algorithm 5.1, V_{Vset} is the set containing the vertices to be colored and the separators that are used to indicate the group boundaries. $Ind_{current}$ refers to the individual constructed in the current iteration, whereas, Ind_{old} refers to the individual constructed in the previous iteration. $elementCount_{current}$ refers to the number of vertices in $Ind_{current}$ and $elementCount_{old}$ refers to the number of vertices in Ind_{old} . Hence, $elementCount$ of an individual gives its utility value. ΔE is the parameter used in the decision phase of the algorithm in which the utility values of the current and the next (reconstructed) individual are compared. ΔE is set to the difference between $elementCount_{old}$ and $elementCount_{current}$. If ΔE is a positive value, then $Ind_{current}$ is worse than Ind_{old} . Hence, it is accepted with a probability given by the evaluation function of SA. The probability of accepting a worse individual decreases with time. If it is rejected, Ind_{old} is replaced with $Ind_{current}$.

Algorithm 5.1. General algorithm for SABT

```

Initialize  $V_{Vset}$ ;
Construct the individual;
power  $\leftarrow$  0.25;
 $elementCount_{best} \leftarrow elementCount_{current}$ ;
 $iterCnt \leftarrow 0$ ;
while There are uncolored vertices in  $V_{Vset}$  do
    Select a backtrackAmount determined by  $\frac{f(iterCnt)}{iterCnt_{max}}$ ;
    Copy  $Ind_{current}$  to  $Ind_{old}$ ;
    Move all vertices till backtrackPoint from  $Ind_{current}$  to  $V_{Vset}$ ;
    Reconstruct  $Ind_{current}$ ;
     $\Delta E \leftarrow elementCount_{old} - elementCount_{current}$ ;
    if  $\Delta E > 0$  then
        Copy  $Ind_{old}$  to  $Ind_{current}$  only with probability
         $iterCnt_{max} - iterCnt^{power} * koeff$ ;
    end
     $iterCnt \leftarrow iterCnt + 1$ ;
    if  $iterCnt == iterCnt_{max}$  then
        break;
    end
end

```

The algorithm begins with an initialization part; An initial individual is created and the parameter $power$ is set to 0.25. This is a parameter used for the evaluation function of SA. It is a user defined parameter. We kept it at 0.25 in all our experiments.

The most well-known evaluation function used for the cooling down schedule of simulated annealing is $e^{-\Delta E/T}$. This function which makes use of the difference of the utility values of current and next individuals is based on temperature T . This function has a logarithmic behavior and it converges to zero as T goes to zero. However, in this thesis, we propose an evaluation function based on a simple exponential function. The exponential function is based on the iteration count only. Since the difference between the utility values of current and next individuals is considerably small, in this study it is neglected in the evaluation function. The behavior of the function depends on parameter $power$. It is possible to adjust the pace of the function by alternating this parameter. The evaluation function $f(iterCnt)$ is given below;

$$f(iterCnt) = iterCnt_{max} - iterCnt^{power} * \frac{iterCnt_{max}}{iterCnt_{max}^{power}} \quad (5.1)$$

where $power = 0.25$ and $iterCnt = 0, 1, \dots, iterCnt_{max}$

As one could notice easily, $iterCnt^{power}$ grows slowly as $iterCnt$ goes to $iterCnt_{max}$ and $iterCnt^{power} \in [0, iterCnt_{max}^{power}]$. The evaluation function $f(x)$ makes use of $iterCnt^{power}$ to obtain a function decreasing gradually. To obtain a function slowly dropping down from $iterCnt_{max}$ to 0, we must scale this function with a coefficient ($iterCnt_{max}/iterCnt_{max}^{power}$). Then this value is subtracted from $iterCnt_{max}$.

The evaluation function should decrease gradually so that at the beginning, the algorithm is more likely to accept bad moves. In other words, the algorithm has more tendency to construct individuals with relatively bad utility value. This is a simple diversification phase. The search must be carried out in the whole search space rather than focusing on a region of individuals with high utility values. The ratio of accepting the bad

moves decreases with time allowing the algorithm to intensificate the search on individuals with higher utility values.

Below, an example of how evaluation function changes according to the parameter *power* is given in Figure 5.1. In the figure, the evaluation function is given for *three* different values of *power* (0.05,0.25 and 0.40). As seen in the figure, if we increase *power*, algorithm tends to accept more worse individuals. In this case, the search becomes too diversified in a similar way to random walk. This might lead the algorithm to destroy high-quality individuals. If *power* is decreased, more worse individuals are rejected. Hence, the algorithm behaves like a hill-climber leading the destruction of the diversification phase.

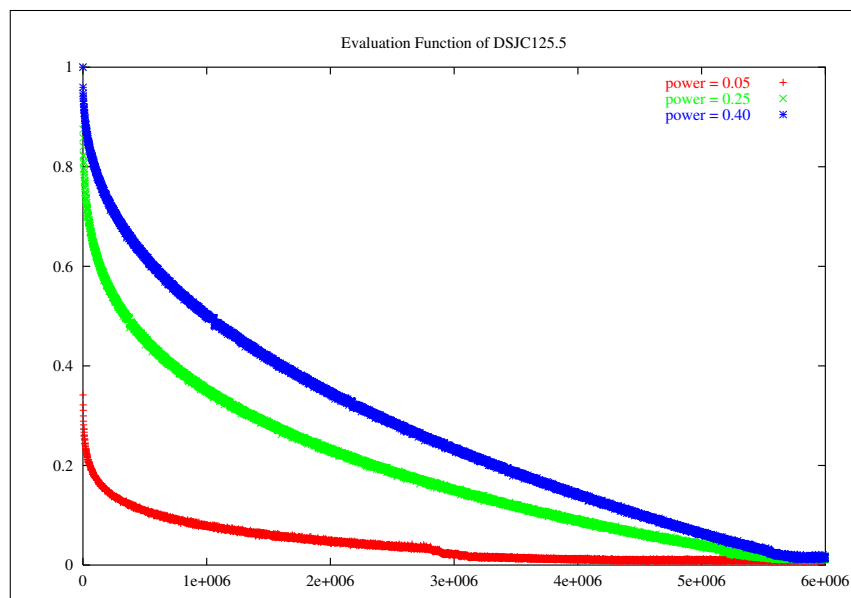


Figure 5.1. Change of evaluation function based on *power*

The parameter *power* has to be chosen carefully in order to balance the diversification and the intensification phase. On the other hand, one can alter *power* to change the behaviour of the algorithm to obtain a hill-climbing algorithm or a random walk on the graph. Since, *power* is user defined. it gives the user this flexibility.

5.2. INDIVIDUAL STRUCTURE IN SABT

The search space consists of partial k -colorings. As mentioned in section 2.2, SABT evaluates partial solutions in order to search for the exact solution. In this approach, the vertices are distributed into $k + 1$ sets. The first set contains the uncolored vertices and k mutually disjoint sets constitute the partial solution. The goal is to increase the size of the partial solution by coloring the uncolored vertices. Let us give the formal definition of a partial solution. Given S to denote the set of all possible valid partial solutions of the graph coloring problem, a partial solution $s_i \in S$ would be defined as; $s_i = \{\vartheta \cup [\bigcup_i^k V_i] \mid \text{where } \forall x, y, x \in V_i, y \in V_i \text{ then } e_{x,y} \notin E \text{ where } i = 1, \dots, k, \text{ such that } \forall z, z \notin V_i, z \in \vartheta \text{ where } i = 1, \dots, k, V = V_1 \cup V_2 \cup V_3 \cup \dots \cup V_k \cup \vartheta\}$. $e_{x,y}$ denotes an edge between vertices x and y .

As given in the above definition for a partial solution, apart from the sets containing the colored vertices, there is a set containing the uncolored vertices. The partial solution constructed by the algorithm is always a valid k -coloring. Thus, it contains no conflicting vertices. The conflicting vertices and the vertices that the algorithm has not attempted to color yet belong to the set ϑ .

Now let us introduce how a partial solution is designed in the algorithm. SABT uses separators to denote group boundaries. It basically operates on two sets, vertex set and conflict set. Vertex set contains all of the vertices that have not been colored yet and also the separators to denote the group boundaries. As one can guess, $k - 1$ separators must be used for a valid k -coloring. The cardinality of the vertex set is then defined as $0 \leq |vertex\ set| \leq |V| + (k - 1)$. The other set is the set of vertices that could not be placed in any group of the individual as they conflict. Note that this set, along with the vertices that the algorithm has not attempted to color yet corresponds to the set ϑ of uncolored vertices in the definition given above. We will name the set containing the vertices to be colored and separators as V_{set} and the set of vertices that conflicts with already colored vertices (thus could not be placed in the individual) as V_{Cset} .

To create an initial individual, SABT fills the array for V_{set} with the vertices and the

separators. Then, shuffles them to create a homogeneous distribution of both type of items in the set. Each group of the individual is a linked list. Note that, each group in an individual represents a different color set. The values of the vertices are kept in the nodes of a group. Initially, a fixed number of nodes are created for each group which is determined by $|V|/k$. This value gives a good estimation of the number of nodes to be created in each group. When all the nodes of a group are occupied with vertices, a new node is created for the new coming vertex. A node physically is deleted from a group if and only if the vertex it contains in that node is removed as a result of a backtrack operation. This is due to the fact that the algorithm starts removing vertices from the beginning of a group.

SABT operates on an individual which consists of k number of groups each one linked to a header. These headers are stored in a header array. A header consists of two pointers, one pointing to the first item of the group and the other pointing to the last. In addition, there is a mark field indicating whether the group is constituted or not and a field where the number of vertices in the group is stored. The mark field is set to *zero* if the group is filled otherwise it is set to *one*. The Figure 5.2 there is an example of an individual of 75 vertices. Here k is *seven* and *six* groups are constituted by the construction mechanism. The first group in the individual is constituted as its mark field is set to *zero* and it has *seven* nodes. Note that separators are not directly inserted into the individual. Instead, when the construction mechanism selects a separator from V_{Vset} , it is understood that a new group will be constituted.

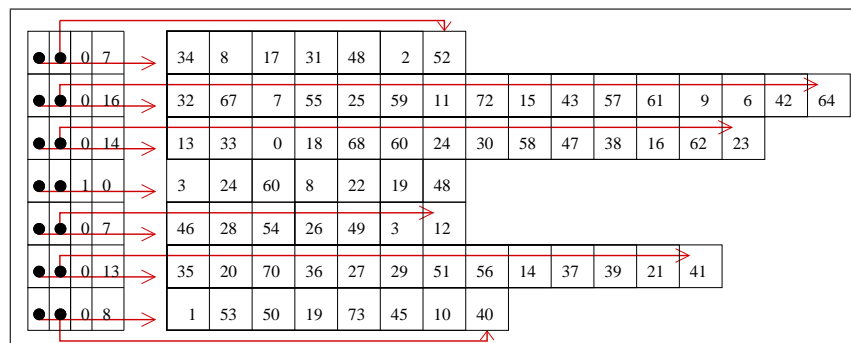


Figure 5.2. Structure of an individual

5.3. CONSTRUCTION MECHANISM

Construction of an individual is a straight-forward operation. As mentioned in section 5.1 selecting an item from V_{Vset} is a random operation. Both the vertices and separators in V_{Vset} have an equal probability to be selected by the algorithm. The algorithm selects the first group in the individual that is marked as *one*. We call this group as the *current group*. The algorithm selects an item from V_{Vset} randomly and tries to insert it into the *current group*. If this item is a vertex and if it conflicts with any of the vertices in the current group, the vertex is put into V_{Cset} . Otherwise, it is inserted into the *current group*. If the item randomly selected is a separator, an empty group (marked as *one*) is selected and all the vertices in V_{Cset} are put back into V_{Vset} . Then, another item is chosen from V_{Vset} randomly starting the process again. When both V_{Vset} and V_{Cset} are empty the algorithm terminates returning the valid k-coloring found. If V_{Vset} is empty and V_{Cset} is not, then there are still uncolored vertices but no separator is left to create new groups. In this case, it becomes impossible to proceed with the insertion operation. Hence, the backtracking mechanism has to be invoked at this point to deform the individual partially.

In Algorithm 5.2, the construction method for SABT is given. The variable *flag* is used to prevent the algorithm from choosing two separators one after the other. As the items in V_{Vset} are chosen randomly, it is possible for construction method to select two separators consecutively. This causes two groups to be selected and the first group to be left empty but marked as filled. By setting the flag to 1 whenever a separator is encountered, the algorithm guarantees that a vertex will be chosen next. The parameter *groupCount* indicates the number of groups that are filled with vertices so far. Whereas, *groupElements* indicates the number of vertices in a group. *elementCount* is used to indicate the number of all vertices in the individual and *VCount* indicates the number of items in the V_{Vset} that are not chosen to be colored yet.

After V_{Vset} is empty, another method called *addToIndividual* is invoked in the construction algorithm. This method simply searches for any group that a vertex in V_{Cset}

Algorithm 5.2. Construction algorithm for an individual

```

if there is no empty group left then
  | return;
else
  | Select an empty group;
  | Mark the group as filled;
  | flag  $\leftarrow$  1;
  | groupCount  $\leftarrow$  groupCount+1;
end
while  $V_{V_{set}}$  is not empty do
  | Select an item from  $V_{V_{set}}$  randomly;
  | if selected item is a vertex then
  | | flag  $\leftarrow$  0;
  | | if selected vertex conflicts with any of the vertices of the selected group then
  | | | Put the selected vertex into  $V_{C_{set}}$ ;
  | | else
  | | | Insert the selected vertex into the selected group;
  | | | groupElements  $\leftarrow$  groupElements+1;
  | | | elementCount  $\leftarrow$  elementCount+1;
  | | end
  | | else
  | | | if flag == 1 then
  | | | | continue;
  | | | end
  | | | Put all items in  $V_{C_{set}}$  into  $V_{V_{set}}$ ;
  | | | Select an empty group;
  | | | Mark the selected group as filled;
  | | | flag  $\leftarrow$  1;
  | | | groupCount  $\leftarrow$  groupCount+1;
  | | end
  | | Swap the selected vertex and the last item in  $V_{V_{set}}$ ;
  | | VCount  $\leftarrow$  VCount-1;
  | end
end
Call addToIndividual;

```

could be inserted into. Note that, at this point, all the vertices that could not be inserted into the ultimately selected group are in V_{Cset} . This is due to the fact that whenever a separator is selected all vertices in V_{Cset} are put back into V_{Vset} . As separators and vertices have equal probability to be chosen, some of the vertices that could be inserted into a group without a conflict are possibly left in V_{Vset} . Hence, at the end of the construction process, such elements are inserted into the previously created groups by the method *addToIndividual*.

In the algorithm, we used simple arrays for V_{Vset} and V_{Cset} . Whenever an item is removed from V_{Vset} , the construction algorithm simply swaps the item to be removed with the last item in the array. And a simple index keeper (*VCount*) that keeps the index of the last element is decremented by one. In this way, removing an item from the array costs only constant time and the algorithm always removes an item without any shift or search operations.

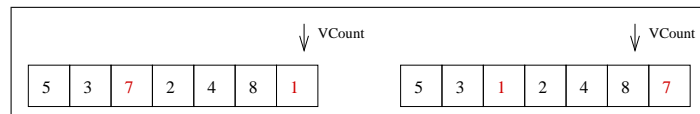


Figure 5.3. Extraction from V_{Vset}

5.4. BACKTRACKING MECHANISM

Now let us explain the backtracking mechanism in detail. Backtracking mechanism transfers some of the vertices inserted in the individual back into V_{Vset} , letting a partial reconstruction of the individual at hand. The amount of backtracking determines the nature of the search. We could jump to a very different point in the search space by backtracking a considerable amount or we could continue our search in the neighborhood of the current individual by tiny backtracks.

As mentioned before, when the algorithm is unable to extend the individual any more, backtracking algorithm is invoked. A value stochastically determined by the evaluation function is used to backtrack to a point in the individual. Hence, the number of vertices that will be removed from the individual is determined by the evaluation function. The groups

to be extracted are randomly chosen and the vertices in these groups are removed from the individual. All the groups of the individual are equally likely to be removed keeping the algorithm from concentrating the search on a specific part of the search space. The algorithm for backtracking mechanism is given below;

As seen in Algorithm 5.3, *backTrackAmount* is determined by the evaluation function $f(iterCnt)$ in a stochastic manner. Initially, a random value is chosen (*randIdx*). Then the value of $f(iterCnt)$ is calculated using the iteration count ($iterCnt^{power}$).

$$\begin{aligned} backTrackAmount &= randIdx * \frac{f(iterCnt)}{iterCnt_{max}} \\ randIdx &= rand(1, elementCount) \end{aligned} \quad (5.2)$$

$f(iterCnt)/iterCnt_{max}$ is utilized to let *backTrackAmount* take a value between $[1, randIdx]$. As *iterCnt* increases in time, $f(iterCnt)/iterCnt_{max}$ decreases causing *backTrackAmount* to take smaller values. *backTrackAmount* is high at the beginning of the algorithm, hence the backtracking mechanism removes a considerable amount of vertices from the individual. It becomes relatively smaller toward the end of the algorithm. In other words, the algorithm decreases the rate of deformation in the individual in time. In this way the algorithm is allowed to jump to different regions in the search space (*diversification phase*) at the beginning of the search process while the immediate neighbors of the individual are searched toward the end (*intensification phase*). Although the utility value of the individuals constructed at the beginning of the search are relatively low, it increases in time. And in a parallel manner, the algorithm drops the rate of backtracking to search for the exact solution.

There might be times that *backTrackAmount* becomes zero, either because *randIdx* or $f(iterCnt)/iterCnt_{max}$ becomes very close to zero. In this case, the algorithm is unable to continue the search. At this point, the algorithm recalculates *backTrackAmount*. This time, according to the number of vertices to be colored ($|V|$), a random value is chosen as the *backTrackAmount*. This approach guarantees that *backTrackAmount* does not fall below a certain threshold. However, this random *backTrackAmount* is kept very small to prevent the algorithm from jumping to a different part of the search space. The parameters *lowerBound*

Algorithm 5.3. Backtracking algorithm for an individual

```

randIdx ← rand(1, elementCount);
backTrackAmount ← randIdx *  $\frac{f(iterCnt)}{iterCnt_{max}}$ ;
if backTrackAmount == 0 then
  | if numberOfvertices * lowerBound ≥ 1 then
  | | backTrackAmount ← rand (numberOfvertices * lowerBound);
  | else
  | | backTrackAmount ← rand (numberOfvertices * upperBound);
  | end
end
Copy Indcurrent to Indold;
backtrackCount ← 0;
randomGroup ← randomly selected non-empty group;
while backtrackCount < backTrackAmount do
  | i ← 0;
  | while i < groupElements and backtrackCount < backTrackAmount do
  | | VVset ← randomGroupElementi;
  | | VCount ← VCount+1, i ← i+1, backtrackCount ← backtrackCount+1;
  | end
  | randomGroup ← randomly selected non-empty group;
end
if No group becomes empty then
  | Move all items in VVset to VCset and Call addToIndividual;
else
  | Move all items in VCset to VVset and Call construct;
end
if elementCountcurrent > elementCountbest then
  | Copy Indcurrent to Indbest;
end
ΔE ← elementCountold − elementCountcurrent;
if ΔE > 0 then
  | prob of acceptance ← rand (iterCntmax);
  | if prob of acceptance > (f(iterCnt)) then
  | | Copy Indold to Indcurrent;
  | end
end

```

and *upperBound* are used in this approach. They are user defined parameters. Below, in Figure 5.4 an example of how *backTrackAmount* changes along the algorithm is provided. The example provided is one of the instances which is utilized in the experiments.

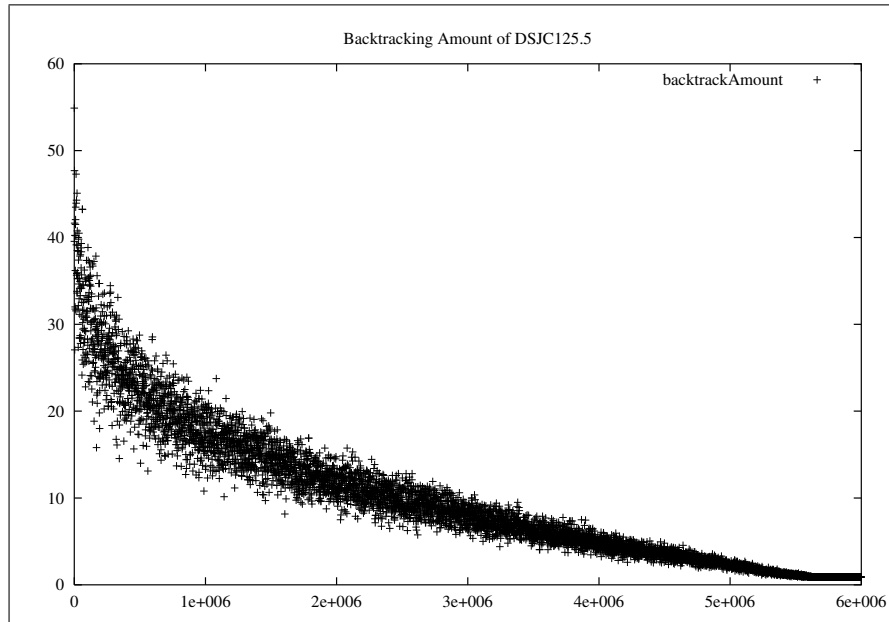


Figure 5.4. Backtracking Amount of DSJC125.5

5.5. INITIAL VERSION OF SABB ($SABB_0$)

In this section, the initial version of SABB ($SABB_0$) is explained briefly. It is the first framework proposed to solve GCP. This version lacks both in solution quality and performance due to some design issues. There are a few differences in the design of the individual, evaluation function of SA and the backtracking mechanism compared to SABB.

Instead of a group of linked lists, an array is used to represent an individual in $SABB_0$. In the Figure 5.5 below, an example individual is given. As seen in the figure, the vertices are inserted into an array of length $|V| + k - 1$. In addition, the separators are directly inserted into the array denoting the boundaries of the groups.

The evaluation function for $SABB_0$ is provided below. Instead of the exponential function proposed for SABB, a linear function is utilized. It is based on the iteration count.

5	2	4	-1	28	3	7	-1	25	10	8	6	29	-1
---	---	---	----	----	---	---	----	----	----	---	---	----	----	-------

Figure 5.5. Example of an individual in the old version of SABT

The parameter $\alpha \in [0, 1]$ is a user defined value. This is a simple but inefficient function in terms of calculating the backtracking amount. Hence, there is not a good balance between diversification and intensification phases and the solution quality is directly affected.

$$f(iterCnt) = \alpha * iterCnt \quad (5.3)$$

In this version, backtracking is done either starting from the left or right end of the individual. Initially, a backtracking direction is selected randomly. Then the backtracking amount is calculated. Again, the backtrack amount is determined stochastically.

$$backtrackIndex = fitnessutility - (randIdx * (f(iterCnt)/iterCnt_{max})) \quad (5.4)$$

According to the direction of the backtracking, some vertices, determined by the backtracking amount, are removed from the individual. Toward the end of the algorithm, as backtracking amount decreases, fewer vertices are removed from the individual. However, in this way, a bias is introduced to some groups. As backtracking is done starting from left or right end of the individual, the groups which are not in the middle of the array are more likely to be selected for backtracking. Hence, the search is carried out on a specific region of the search space affecting the diversity of the individual constructed. The experiment results obtained with this framework are not satisfactory in terms of solution quality. This is based on the unequal probability distribution for being selected in the backtracking mechanism.

In Figure 5.6, backtracking operation is illustrated. In this example, there are *four* groups and 12 vertices in total and backtracking is done starting from right end of the individual. The backtracking point is indicated by an arrow. The last two groups and *one* element of the second group (7) are removed from the individual. Hence, *eight* vertices are removed from the individual in total.

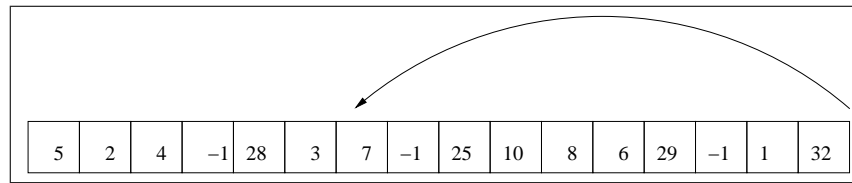


Figure 5.6. Example of backtracking on an individual of $SABT_0$

Another disadvantage is that whenever backtracking is done starting from the left of the individual, the remaining vertices have to be shifted to left. This is a costly operation. To eliminate these issues, the design of the individual is altered and linked lists are used. In addition, backtracking mechanism is redesigned such that the groups to be removed from the individual are selected randomly with equal probability. In the following chapter where the experiment results are presented, a comparison between SABT and $SABT_0$ is also presented in terms of solution quality.

6. EXPERIMENTAL RESULTS

In this chapter, we present the experimental results on several benchmark instances obtained from DIMACS challenge suite. In addition, SABB is compared with some other state-of-the-art algorithms in the literature. The comparison is carried out in terms of solution quality.

6.1. PROBLEM INSTANCES AND EXPERIMENTAL DETAILS

All problem instances that we use in our experiments are selected from DIMACS challenge suite. The collection of instances provided by DIMACS is recognized as a standard for testing the algorithms on graph coloring problem [55]. The instances of DIMACS challenge suite are classified into two categories according to their difficulty. The instances in the first category are solved easily by most of the modern graph coloring heuristics in the literature. However it is quite difficult to color the instances in the second category with the corresponding chromatic numbers or with the minimum (best) number of colors reported so far (for those whose chromatic numbers are unknown). In the experiments, a mixture of instances from both categories are utilized. DSJC500.5, DSJC1000.1, DSJC1000.5, R250.5 and le450_25c are the difficult instances selected for the experiments. The rest of the instances are among the easy instances. In the experiments, 18 different instances are utilized in total.

The instances used in the experiments have different structural characteristics including randomly generated graphs, leighton graphs and instances obtained from register allocation problems in real codes. The algorithm is run on each instance 20 times independently with different random seeds. The test runs terminate either when the desired solution is obtained or when the maximum number of iteration count is reached. The maximum number of iteration count used in the experiments range from $3 * 10^6$ to $90 * 10^6$. The parameter *power* utilized in the evaluation function of SA is set to 0.25 for all the experiments. The other parameter $Factor_0$, which is utilized when *backtrackAmount* goes down to zero, takes its value between 0.01 and 0.005. The upper limit of this range (0.01)

is selected such that the backtracking amount for the smallest instances (instances with 125 vertices) is guaranteed not to become zero. Lower limit (0.005) is used for all of the other instances. All these parameters are user defined providing a certain flexibility to the user in demonstrating different scenarios for the experiments. The average computational times are given in terms of CPU time (in seconds). The algorithm is coded with C programming language and compiled with GNU GCC compiler. The hardware of the platform used for the experiments is a 2.66 Core Duo PC with 2GB RAM.

In Table 6.1 the parameters used in the experiments and best colorings that SABL has found are presented. In the first column, the instances used in the experiments are given. They are grouped according to their types. There are *six* groups of instances. First group of instances are the random graphs which are generated by Johnson et al. [26]. They are denoted by the prefix *DSJC* followed by a fractional number. The integer part of the number denotes the number of vertices in the graph ($|V|$). The fractional part of the number denotes the density of the graph. For example, the instance *DSJC125.5* has 125 vertices and 50% of the vertex pairs are connected with an edge in this graph. Second group consist of random geometric graphs. These graphs are generated by picking points uniformly in a square and then by setting an edge between all pairs of vertices situated within a certain distance. Again the density of the graph is denoted by the fractional part of the number in the name of an instance. In the third group, there exist the Leighton graphs. They all have the same fixed number of vertices (450) and the second number in the instance name denotes the chromatic number of the graph [5]. In the following group, there is only one instance of a flat graph by J. Culberson [56]. Flat graphs are generated by partitioning the vertex set into K almost equal sized classes and then by selecting edges only between vertices of different classes. The next group also has only one instance which is an example of class scheduling graphs generated by Gary Lewandowski in the second DIMACS challenge. The last group is again generated by Gary Lewandowski and the instances are based on register allocation for variables in real codes.

The first column in Table 6.1 gives the group names of the instances and in the second column the names of the instances are given. Third and fourth columns denote the number of vertices for each instance and the density of the graph. In the fifth column the chromatic

Table 6.1. Best colorings for SABB

	Instances	n	dens.	χ/k^*	SABB	Diff.	avg CPU Time
random graphs	DSJC125.5	125	0.50	?/17	17	–	61.95 sec
	DSJC125.9	125	0.89	?/44	44	–	31.45 sec
	DSJC250.1	250	0.10	?/8	8	–	38.40 sec
	DSJC250.9	250	0.90	?/72	72	–	255.50 sec
	DSJC500.5	500	0.50	?/48	51	3	2895.30 sec
	DSJC1000.1	1000	0.10	?/20	21	1	1336.40 sec
	DSJC1000.5	1000	0.50	?/83	92	9	2916.31 sec
random geo. graphs	DSJR500.1	500	0.03	?/12	12	–	62.15 sec
	R250.5	250	0.48	65/65	68	3	246.3187 sec
leighton graphs	le450_15b	450	0.08	15/15	16	1	88.15 sec
	le450_25c	450	0.17	25/25	27	2	1299.60 sec
flat graphs	flat300.20	300	0.48	20/20	20	–	173.15 sec
sched. graphs	school1_nsh	352	0.24	14/14	14	–	147.55 sec
reg. alloc. graphs	fpsol2.i.2	451	0.08	30/30	30	–	43.75 sec
	inithx.i.2	645	0.07	31/31	31	–	71.6315 sec
	mulsol.i.1	197	0.20	49/49	49	–	< 1 sec
	mulsol.i.4	185	0.23	31/31	31	–	13.00 sec
	zeroin.i.1	211	0.18	49/49	49	–	8.25 sec

number of the instance (χ) and the minimum number of colors reported so far (k^*) given. If (χ) is unknown for an instance (denoted by ?), k^* is taken into consideration. The following column gives the best number of colors for each instance that SABB has found. The colors matching χ or k^* are indicated in bold face. The seventh column gives the difference between SABB and χ/k^* in terms of number of colors. The average CPU time (in seconds) of 20 runs is presented in the last column of Table 6.1. Since the software specifics and physical conditions of the test environments differ for different studies, it has not been possible to compare SABB with other state-of-the-art algorithms in terms of computational time.

The Table 6.2 gives the user-defined parameters utilized in the algorithm. In the first column, the group names of the instances are given. The second column denotes the names of the instances. Third column denotes the number of vertices for each instance. In fourth

column, the value of the parameter used when *backtrackAmount* becomes zero ($Factor_0$) is presented. The next column presents the maximum iteration count used for each instance ($IterCnt_{max}$).

Table 6.2. User-defined parameters

	Instances	n	$Factor_0$	$IterCnt_{max}$
random graphs	DSJC125.5	125	0.01	$6 * 10^6$
	DSJC125.9	125	0.01	$3 * 10^6$
	DSJC250.1	250	0.005	$3 * 10^6$
	DSJC250.9	250	0.005	$6 * 10^6$
	DSJC500.5	500	0.01	$80 * 10^6$
	DSJC1000.1	1000	0.005	$6 * 10^6$
	DSJC1000.5	1000	0.007	$90 * 10^6$
random geo. graphs	DSJR500.1	500	0.005	$6 * 10^6$
	R250.5	250	0.005	$80 * 10^6$
leighton graphs	le450_15b	450	0.005	$3 * 10^6$
	le450_25c	450	0.005	$80 * 10^6$
flat graphs	flat300.20	300	0.005	$3 * 10^6$
sched. graphs	school1_nsh	352	0.005	$3 * 10^6$
reg. alloc. graphs	fpsol2.i.2	451	0.005	$3 * 10^6$
	inithx.i.2	645	0.005	$3 * 10^6$
	mulsol.i.1	197	0.01	$3 * 10^6$
	mulsol.i.4	185	0.01	$3 * 10^6$
	zeroin.i.1	211	0.005	$3 * 10^6$

Tables 6.3, 6.4 and 6.5 give the comparison of SABT with some other state-of-the-art coloring algorithms in terms of solution quality. Most of these algorithms are among the most effective algorithms in the literature covering the best known results for the tested instances. In the first column, the names of the groups of instances are provided. Name of the instances are given in the following column. Third column in the table indicates the chromatic number (χ) of the instance and the best number of colors reported (k^*). In the fourth column, the best number of colors found by SABT are given. The following columns give the best number of colors found by the algorithms compared with SABT and the differences between the number of colors that are found by SABT and the algorithms compared with it. The best

colorings are indicated in bold face. *Four* local search algorithms, *four* hybrid algorithms, *three* population-based algorithms, and an algorithm based on branch-and-bound method are selected for comparison. Below, some explanation for initialization phase, preprocessing step and usage of domain-specific knowledge are provided. These three approaches exist in some of the algorithms compared to SABB. Then, these algorithms are explained briefly.

Initialization phase is the utilization of a fast algorithm to create a single initial coloring or a collection of initial colorings if the algorithm is population-based. The algorithm utilized is generally an exact algorithm or a heuristic. The aim is to start the main algorithm with a high-quality solution or solutions so that the convergence for the problem at hand could be improved.

Preprocessing can be applied to a graph G to reduce it to another graph G' . There are two reduction rules for preprocessing step in [57]. The first rule is extracting stable sets from the graph G and the remaining graph is called the *residual graph* which is mentioned in Section 2.3. This process involves removing all the vertices in G that have a degree less than k where k is the chromatic number. Degree of a vertex is the number of edges that vertex has. Since the degree of a vertex u is less than k , it is guaranteed that a color that is not used in the set of adjacent vertices can be directly assigned to u . Certainly this assignment would not break the feasibility of the coloring formed. Second rule is to remove any vertex $v \in V$, for which there is a $u \in V$, $v \neq u$, $e_{u,v} \notin E$, and u is connected to every vertex to which v is connected (subsumption). In this case, any color that can be assigned to u can also be assigned to v . In the preprocessing step, these two rules are applied iteratively until no other vertex can be removed from the graph [58].

Domain-specific knowledge which is processed by the algorithm is the information related to the problem instance itself. Examples of domain-specific knowledge are measuring diversity among the individuals constructed (for population-based algorithms) [9, 10] and utilizing a digraph [33]. Also, preprocessing step itself exploits domain-specific knowledge.

In [59], the authors present an ant-based algorithm named ABAC. Unlike the previous ant algorithms, in this algorithm, an ant colors just a portion of the graph using only local

information instead of coloring the whole graph. The algorithm has a initialization step. MXRLF, which is a modified version of the XRLF algorithm [5, 26] is utilized to obtain a proper k -coloring. Here k is an upper bound on the $\chi(G)$ of graph G . An initial coloring of G which may not be proper is derived from the coloring generated by MXRLF. Then the algorithm utilizes its ants.

Table 6.3. Comparison of SABT with population based algorithms and other algorithms

	Instances	χ/k^*	SABT	Population-based Algorithms						Other Alg.	
				[59]		[35]		[37] ¹		[37] ²	
random graphs	DSJC125.5	?/17	17	17	-	19	-2	20	-3	20	-3
	DSJC125.9	?/44	44	44	-	44	-	-	-	-	-
	DSJC250.1	?/8	8	8	-	9	-1	-	-	-	-
	DSJC250.9	?/72	72	72	-	74	-2	-	-	-	-
	DSJC500.5	?/48	51	50	1	56	-5	59	-8	65	-14
	DSJC1000.1	?/20	21	21	-	23	-2	-	-	-	-
	DSJC1000.5	?/83	92	91	1	-	-	-	-	-	-
random geo. graphs	DSJR500.1	?/12	12	12	-	-	-	12	-	12	-
	R250.5	65/65	68	-	-	-	-	65	3	66	2
leighton graphs	le450_15b	15/15	16	15	1	17	-1	17	-1	15	1
	le450_25c	25/25	27	26	1	29	-2	-	-	-	-
flat graphs	flat300.20	20/20	20	20	-	20	-	23	-3	39	-19
sched. graphs	school1_nsh	14/14	14	14	-	14	-	20	-6	26	-12
reg. alloc. graphs	fpsol2.i.2	30/30	30	30	-	-	-	-	-	-	-
	inithx.i.2	31/31	31	31	-	-	-	-	-	-	-
	mulsol.i.1	49/49	49	49	-	-	-	49	-	49	-
	mulsol.i.4	31/31	31	31	-	-	-	-	-	-	-
	zeroin.i.1	49/49	49	49	-	-	-	-	-	-	-

In [35], a multi-objective genetic algorithm (MOGA) named LLE&LLE-e MOGA is presented. The algorithm is based on the encoding scheme LLE which discards the redundancy of other traditional encoding schemes. And a supplementary encoding scheme named LLE-e is utilized whenever a genetic operator is costly in LLE, enhancing the performance of the algorithm.

In [37], an evolutionary algorithm named Evolve-P is introduced. It is based on genetic

programming whose aim is to produce a program that will provide any graph of n nodes with a minimal coloring while attempting to minimize k . The algorithm works on sequences which are arranged in non-increasing order of node degrees. Node degree is the number of edges a node has. The algorithm is denoted as [37]² in Table 6.3.

The algorithm T_B&B [37], is a branch-and-bound algorithm that employs the elements that are utilized in tabu search for control. The algorithm is good for either exact or heuristic graph coloring. T_B&B is denoted as [37]¹ in Table 6.3.

In [33] a local search algorithm called VSS-Col is proposed. Three different search spaces along with their own formulations of the problem are used. Third space is a *digraph* which is a directed graph with an orientation on each edge of the graph G . Hence, the algorithm exploits domain-specific knowledge. VSS moves from a search space to another when it is trapped in a local optimum. The algorithm is based on tabu search algorithm.

In [11], an application of iterated local search algorithm (ILS) is presented. The authors claim that in real applications, it is not possible to solve GCP by first guessing a very good coloring and then running the algorithm. Hence, a good initial feasible coloring has to be determined. The algorithm proposed has an initialization phase; An initial coloring is constructed using the exact coloring algorithm implemented by Trick [60]. This algorithm is based on DSATUR [25]. Also, the algorithm has a pre-processing step. As the local search algorithm, tabu search (TS) is utilized.

A local search algorithm (FOO-TABUCOL), again based on tabu search algorithm is proposed in [12]. The reactive tabu list size (tabu tenure) adjusts itself depending on both the graph and the state of the search. Hence, the algorithm exploits domain-specific knowledge. As mentioned in chapter 2 section 2.3, SABB is based on the third approach of vertex coloring. FOO-TABUCOL is also based on this approach. The algorithm has an initialization phase in which a greedy algorithm is used to generate an initial partial k -coloring.

Another local search based algorithm is presented in [61]. In this study, a multilevel

Table 6.4. Comparison of SABB with local search algorithms

	Instances	χ/k^*	SABB	Local Search Algorithms							
				[33]		[11]		[12]		[61]	
random graphs	DSJC125.5	?/17	17	-	-	-	-	-	-	18	-1
	DSJC125.9	?/44	44	-	-	-	-	-	-	44	-
	DSJC250.1	?/8	8	-	-	-	-	-	-	9	-
	DSJC250.9	?/72	72	-	-	-	-	-	-	74	-
	DSJC500.5	?/48	51	48	3	49	2	48	3	54	-3
	DSJC1000.1	?/20	21	20	1	20	1	20	1	23	-2
	DSJC1000.5	?/83	92	86	6	89	3	89	3	97	-5
random geo. graphs	DSJR500.1	?/12	12	-	-	-	-	-	-	12	-
	R250.5	65/65	68	-	-	-	-	66	2	68	-
leighton graphs	le450_15b	15/15	16	-	-	15	1	-	-	16	-
	le450_25c	25/25	27	25	2	26	1	25	2	27	-
flat graphs	flat300.20	20/20	20	-	-	-	-	-	-	20	-
sched. graphs	school1_nsh	14/14	14	-	-	-	-	-	-	14	-
reg. alloc. graphs	fpsol2.i.2	30/30	30	-	-	-	-	-	-	30	-
	inithx.i.2	31/31	31	-	-	-	-	-	-	31	-
	mulsol.i.1	49/49	49	-	-	-	-	-	-	49	-
	mulsol.i.4	31/31	31	-	-	-	-	-	-	31	-
	zeroin.i.1	49/49	49	-	-	-	-	-	-	49	-

approach to coloring is presented. The multilevel paradigm involves recursive coarsening to create a hierarchy of approximations to the original problem. An initial solution is found and then iteratively refined at each level. The algorithm utilizes either iterated greedy search or tabu search algorithm as the local search mechanism. The multilevel operations involve domain-specific knowledge heavily.

In [40] a hybrid evolutionary algorithm (HEA) is implemented. The mutation operator is replaced with a LS operator. A CX operator is used to produce an offspring from two parents s_1 and s_2 . DSATUR [25] is used to produce an initial population. Hence, the algorithm has a initialization phase. Also, the algorithm exploits domain-specific knowledge as it extracts stable sets from the graph. Tabu search algorithm is used as the local search (LS) operator.

Table 6.5. Comparison of SABT with hybrid algorithms

	Instances	χ/k^*	SABT	Hybrid Algorithms							
				[40]		[38]		[10]		[9]	
random graphs	DSJC125.5	?/17	17	-	-	17	-	-	-	17	-
	DSJC125.9	?/44	44	-	-	44	-	-	-	44	-
	DSJC250.1	?/8	8	-	-	8	-	-	-	8	-
	DSJC250.9	?/72	72	-	-	72	-	-	-	72	-
	DSJC500.5	?/48	51	48	3	48	3	48	3	48	3
	DSJC1000.1	?/20	21	20	1	20	1	20	1	20	1
	DSJC1000.5	?/83	92	83	9	84	8	83	9	83	9
random geo. graphs	DSJR500.1	?/12	12	-	-	12	-	-	-	12	-
	R250.5	65/65	68	-	-	-	-	-	-	65	3
leighton graphs	le450_15b	15/15	16	-	-	15	1	-	-	15	1
	le450_25c	25/25	27	26	1	26	1	25	2	25	2
flat graphs	flat300.20	20/20	20	-	-	20	-	-	-	-	-
sched. graphs	school1_nsh	14/14	14	-	-	14	-	-	-	14	-
reg. alloc. graphs	fpsol2.i.2	30/30	30	-	-	30	-	-	-	-	-
	inithx.i.2	31/31	31	-	-	31	-	-	-	-	-
	mulsol.i.1	49/49	49	-	-	49	-	-	-	-	-
	mulsol.i.4	31/31	31	-	-	31	-	-	-	-	-
	zeroin.i.1	49/49	49	-	-	49	-	-	-	-	-

AMACOL [38] is a hybrid evolutionary heuristic that uses a central memory M . This central memory contains pieces of solutions. A recombination operator is applied to create offspring (s). Then a LS operator is applied on the offspring. Solutions created by the LS operator (s') are not necessarily legal ones. Pieces of (s') are used to update M . The algorithm also utilizes domain-specific knowledge.

EVACOL, which is proposed in [10] is an evolutionary hybrid algorithm. An enhanced CX operator that extracts best color classes from more than two parents is used. In addition, a new method to assure population diversity is introduced; The distances between individuals of the population are kept as large as possible.

In [9], a hybrid meta heuristic algorithm named MACOL is proposed. MACOL,

which is based on tabu search algorithm is combined with an evolutionary algorithm. A new CX operator called *adaptive multi-parent CX operator* is proposed. Furthermore, a new replacement criterion is proposed for updating the population. This criterion takes into account both the quality and the diversity among the individuals. In both, EVACOL and MACOL, domain-specific knowledge is utilized for assuring the diversity among the individuals.

As seen in Table 6.1, SABT can color all of the instances in less than 1 CPU hour. For the relatively easier instances where the iteration count is set as $3 * 10^6$ or $6 * 10^6$, SABT can color all of these instances with the corresponding χ/k^* colors except for le450_15b. Even if the iteration count is increased to high values, SABT could not color le450_15b with less than 16 colors while k^* is 15 for this instance.

On the other hand, DSJC1000.1 is a difficult instance and again, better colorings could not be achieved on this instance despite large iteration counts. DSJC1000.1 could be colored with 21 colors while k^* is 20. For other difficult instances, again, higher iteration counts are utilized. DSJC500.5 is colored with 51 colors while k^* is 48, DSJC1000.5 with 92 colors while k^* is 83, R250.5 with 68 colors while k^* is 65 and le450_25c with 27 colors while k^* is 25.

As seen from Tables 6.3, 6.4 and 6.5, the performance of SABT is the same with the other state-of-the-art algorithms on easy instances. SABT performs better than the local search algorithm proposed in [61], T_B&B and all of the population-based algorithms except for ABAC [59]. As seen in Table 6.3, ABAC can color the difficult instances DSJC500.5, DSJC1000.5, le450_15b and le450_25c with one less color than SABT. The local search and hybrid algorithms in Table 6.4 and 6.5, are chosen among the coloring algorithms covering the best results in the literature except for [61].

For the difficult instances, some local search algorithms and hybrid algorithms are better than SABT. The reason behind this is that all these algorithms utilize an initialization phase, a pre-processing step or they exploit domain-specific knowledge. SABT performs slightly worse than these algorithms for difficult instances as it is a simple and

basic meta-heuristic without any initialization or pre-processing step. In addition, no domain-specific knowledge is exploited in the algorithm. SABB is designed to be utilized to solve any grouping problem. Hence, we have focused on obtaining the best results possible without utilizing any domain-specific knowledge to GCP.

Below, the graphs of utility values and backtracking amounts are provided for each instance. The graphs give the average utility values and backtracking amounts of 20 runs. The test runs are sampled at each 1000th iteration. The average values are obtained from these samples. As mentioned in Section 5.1, the utility value is determined by the number of elements in an individual. The graph for the utility value gives the change in the element count of the individual. In other words, it is the graph that presents how the algorithm converges to the solution of the problem. The graph of backtracking amount shows how the backtracking amount changes throughout the run. As mentioned in Section 5.4, backtracking amount decreases gradually toward the end of the algorithm as it is calculated in a stochastic manner based on the evaluation function of the simulated annealing (SA) approach.

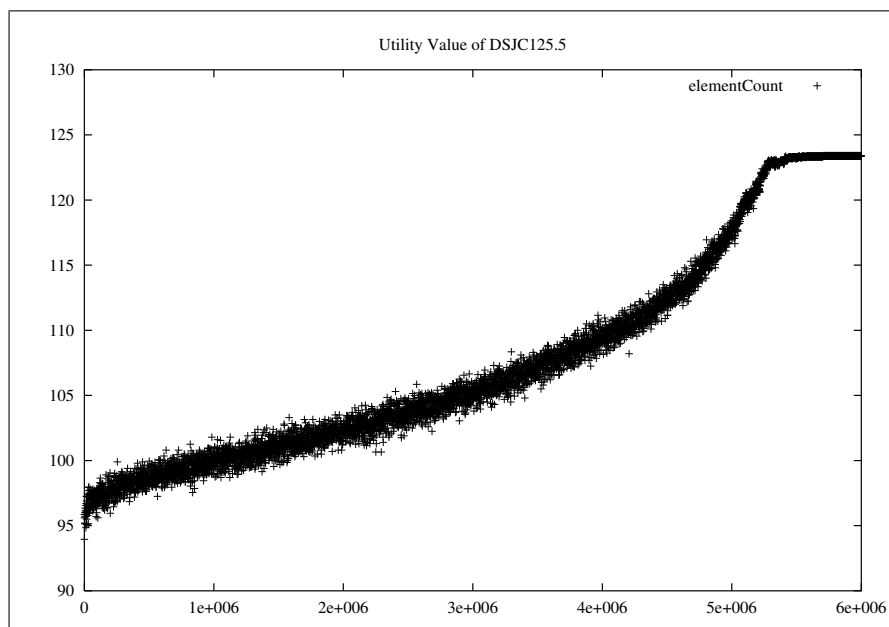


Figure 6.1. Utility value of DSJC125.5

As seen in the figures below, SABB can color small instances 125.5, 125.9, DSJC250.1 and DSJC250.9 quickly. The algorithm terminates before the maximum iteration count is

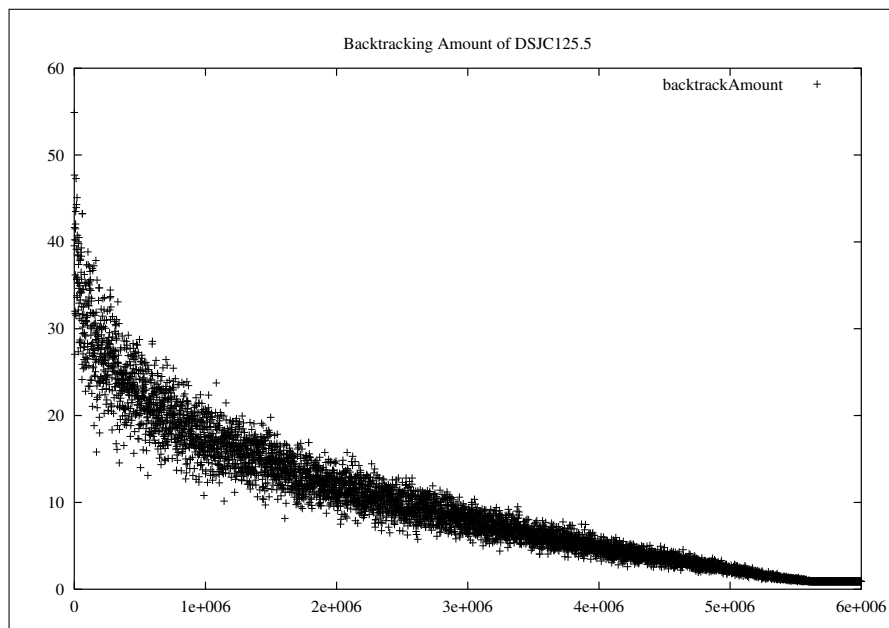


Figure 6.2. Backtracking Amount of DSJC125.5

reached. As mentioned before, DSJC500.5 is a difficult graph. The iteration count is set to 80×10^6 for this instance and it is observed that the algorithm has a faster convergence toward the end of the run. Same behavior is observed for other difficult instances DSJC1000.1 and DSJC1000.5.

DSJR500.1 can be colored much before the maximum iteration count has been reached. The iteration count is set to 6×10^6 for this instance. The point where backtracking amount drops down to zero is more obvious in this graph. After about 1.8×10^6 iterations, the backtracking amount starts to take very small random values. R250.5, which is another difficult graph could be colored with a maximum iteration count of 80×10^6 . Again, a faster convergence can be observed toward the end of the algorithm, although not as obvious as in the case of DSJC1000.1 and DSJC1000.5. This is due to the fact that the iteration count is much larger and the rate of decrease in the backtracking amount is less for this instance.

For leighton graph le450.15b, rather small iteration count (around 2.7×10^6) is enough to color it at a cost of one color. While χ is 15, SABT could color le450.15b with 16 colors. It took SABT approximately 75×10^6 iterations to color le450.25c with 27 colors. The maximum iteration count is set to 80×10^6 as this is a difficult instance. The convergence of

the algorithm again increases toward the end of the algorithm.

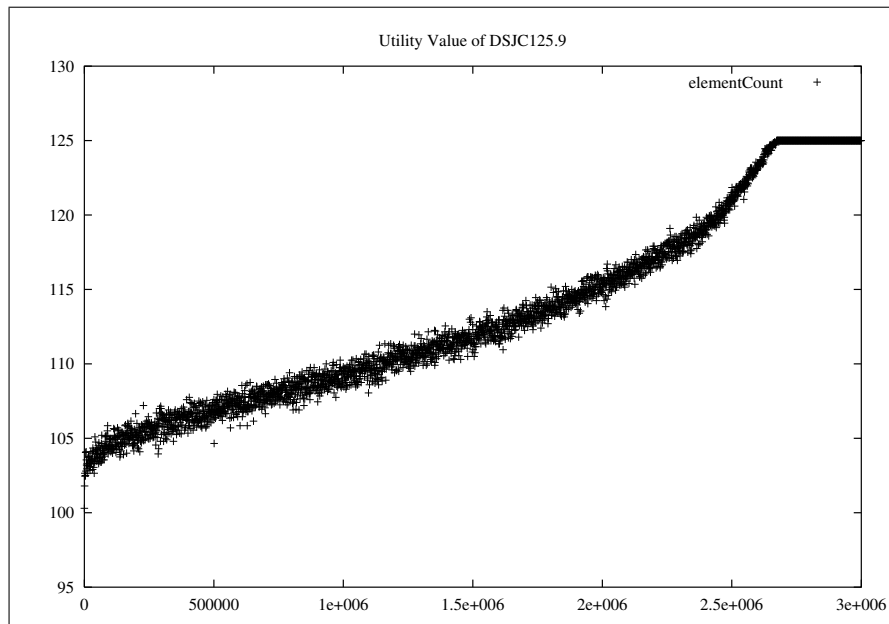


Figure 6.3. Utility value of DSJC125.9

flat300.20 is an instance with many local optima. Many algorithms get stuck in one of these local optima. However, SABT could color this instance very quickly and without getting stuck in a local optimum. The iteration count is slightly over $2 * 10^6$. school1_nsh was an easy instance for SABT which is again colored at around $2 * 10^6$ iterations. For both flat300.20 and school1_nsh, SABT almost hopped to the exact solution toward the end of the algorithm. It is observed that when backtracking amount is chosen randomly, the rate of convergence increases, the conflict count drops really fast.

For fpsol.2.i.2 and inithx.i.2, it is observed that the algorithm converges faster toward the end of the algorithm again due to the randomly chosen backtracking amount. mulsol.i.1 is the easiest instance to color. It takes only a few hundreds of iterations for SABT to color this graph within a period of time less than a second. mulsol.i.4 is colored very quickly at around $1 * 10^6$ iterations and it takes SABT again less than $1 * 10^6$ iterations to color zeroin.i.1.

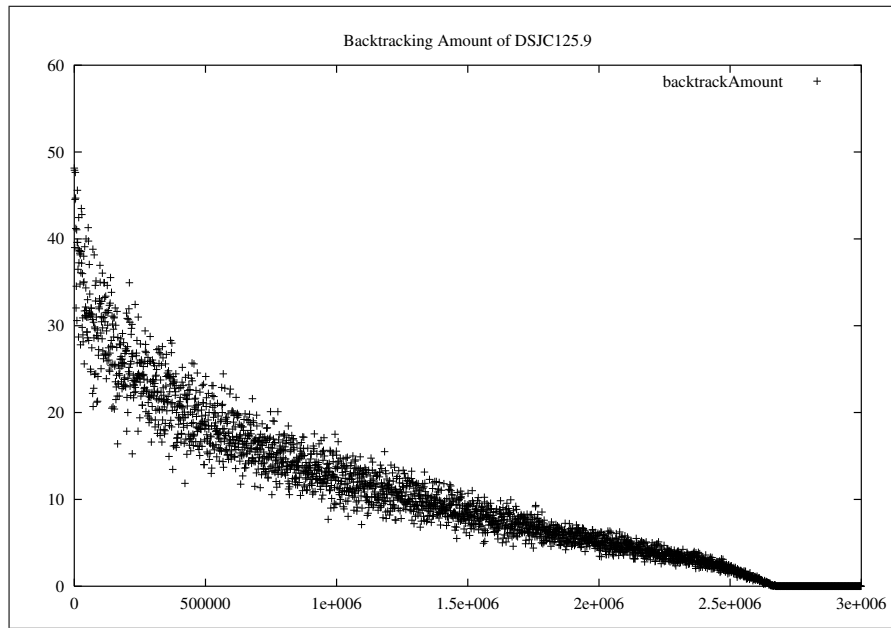


Figure 6.4. Backtracking Amount of DSJC125.9

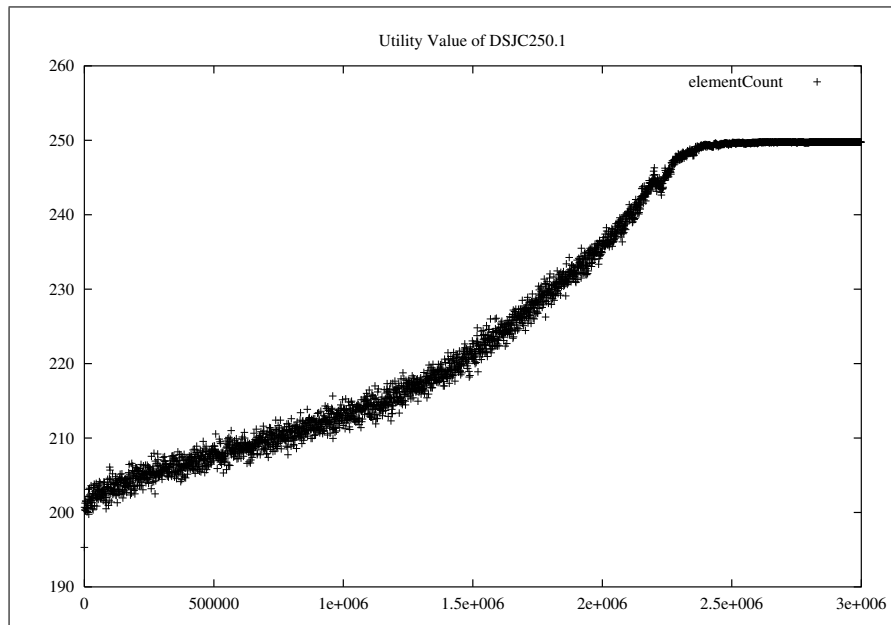


Figure 6.5. Utility value of DSJC250.1

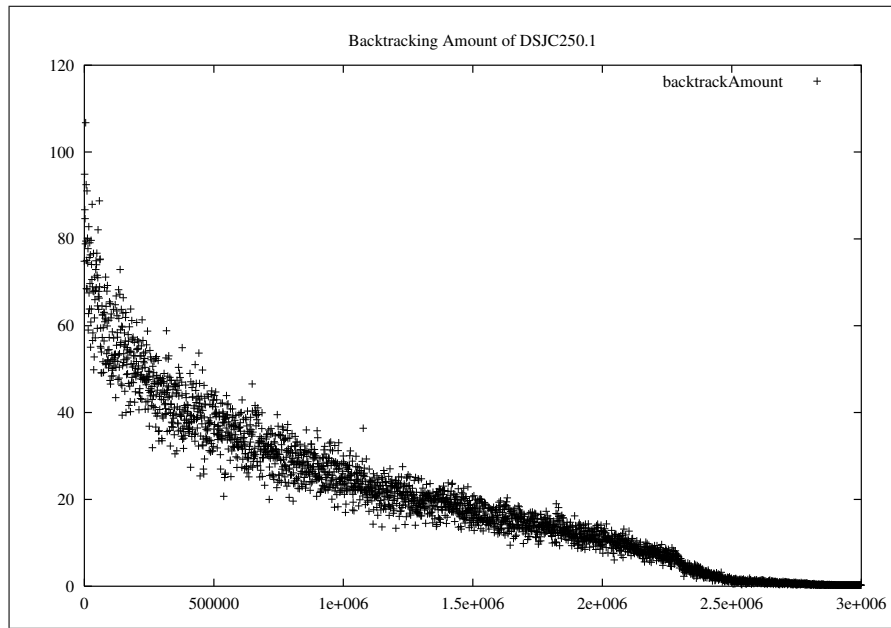


Figure 6.6. Backtracking Amount of DSJC250.1

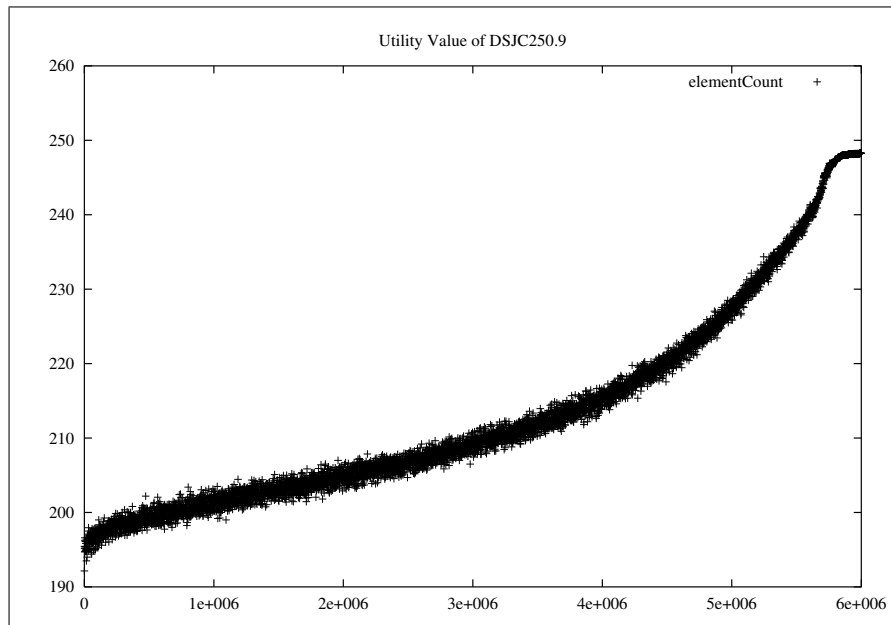


Figure 6.7. Utility value of DSJC250.9

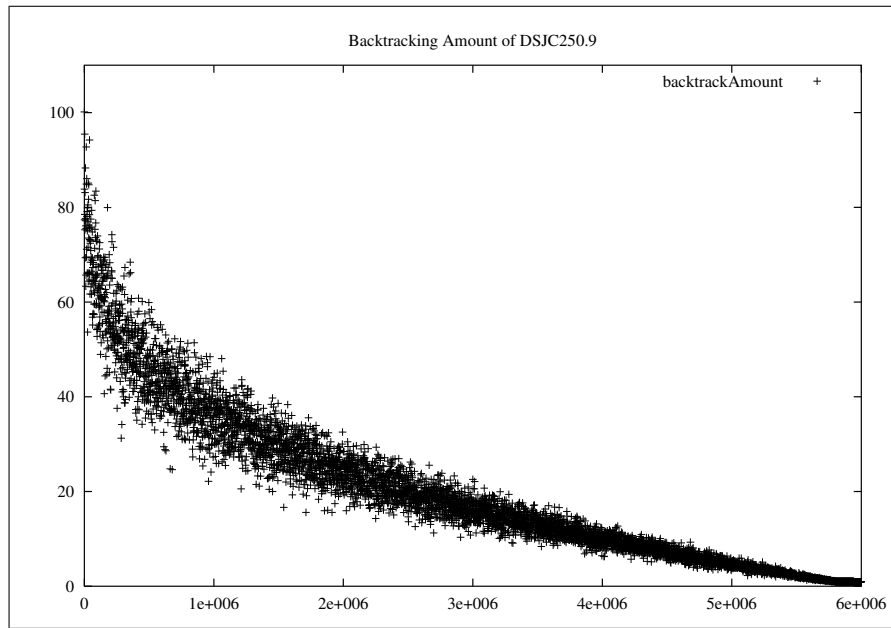


Figure 6.8. Backtracking Amount of DSJC250.9

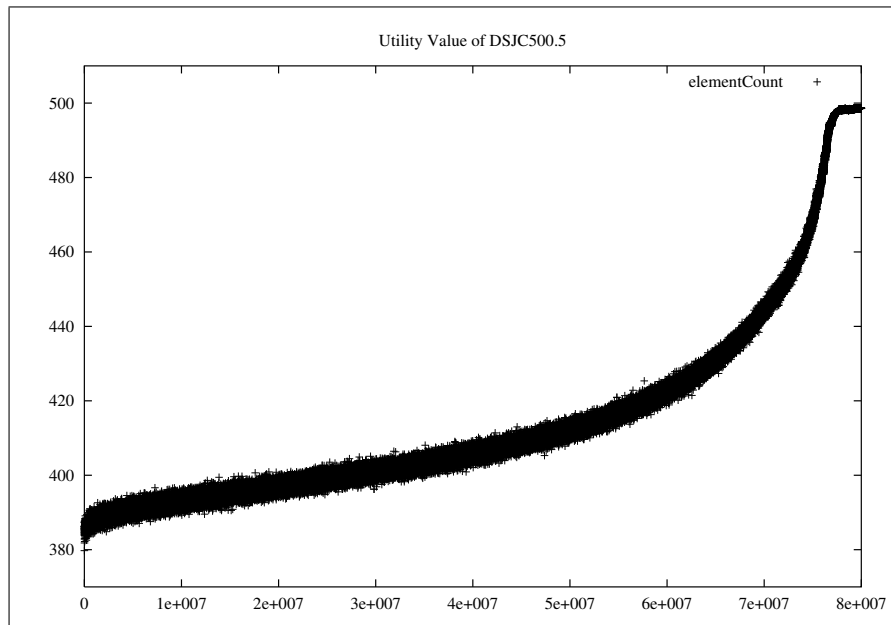


Figure 6.9. Utility value of DSJC500.5

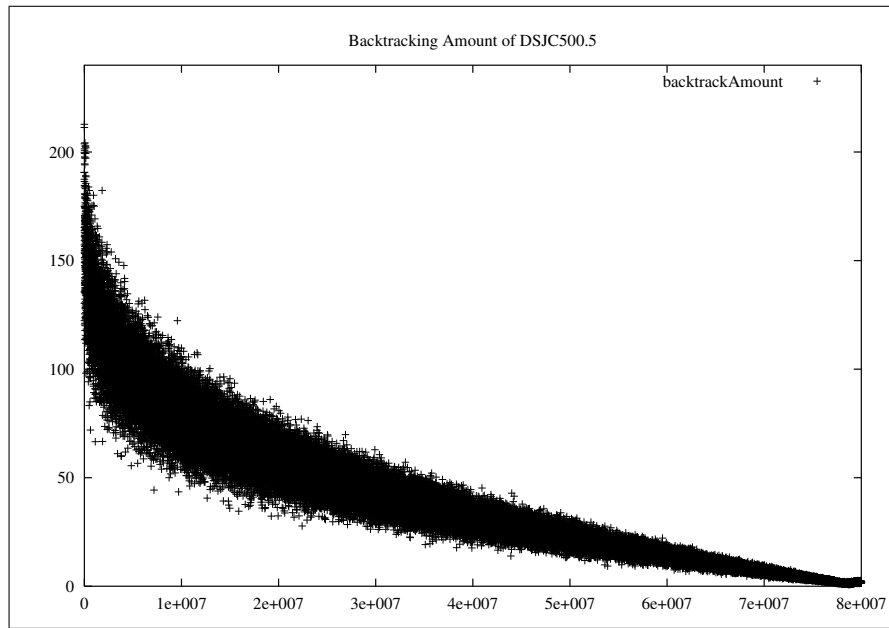


Figure 6.10. Backtracking Amount of DSJC500.5

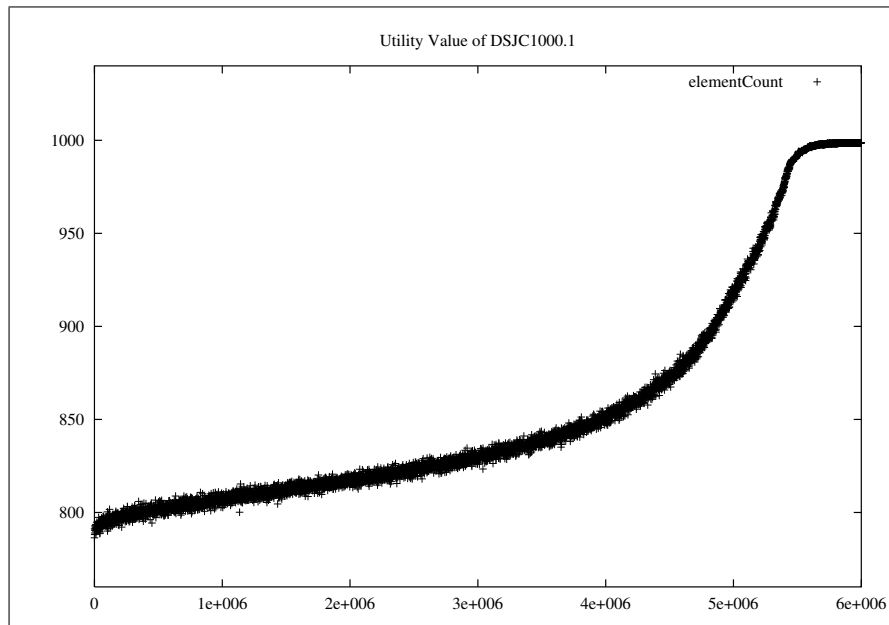


Figure 6.11. Utility value of DSJC1000.1

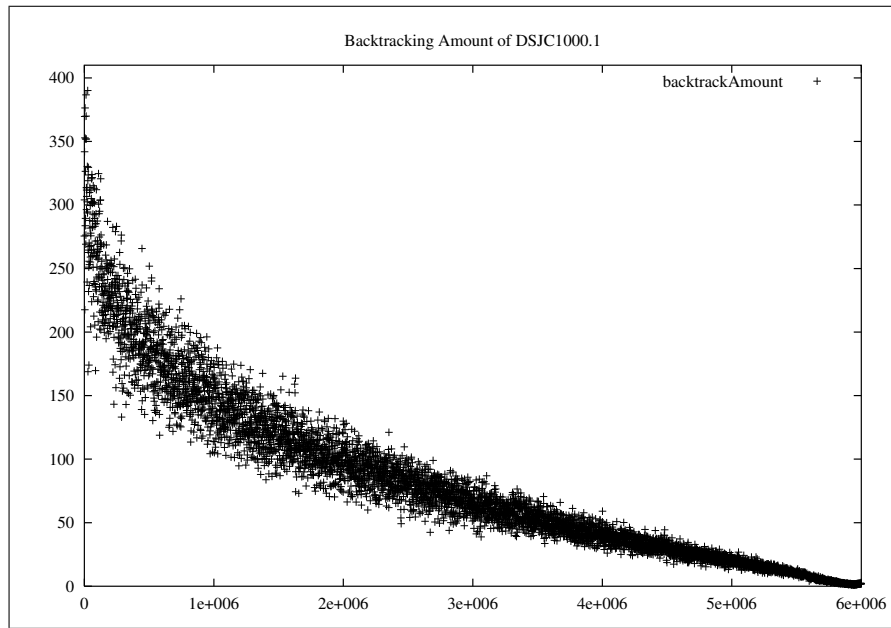


Figure 6.12. Backtracking Amount of DSJC1000.1

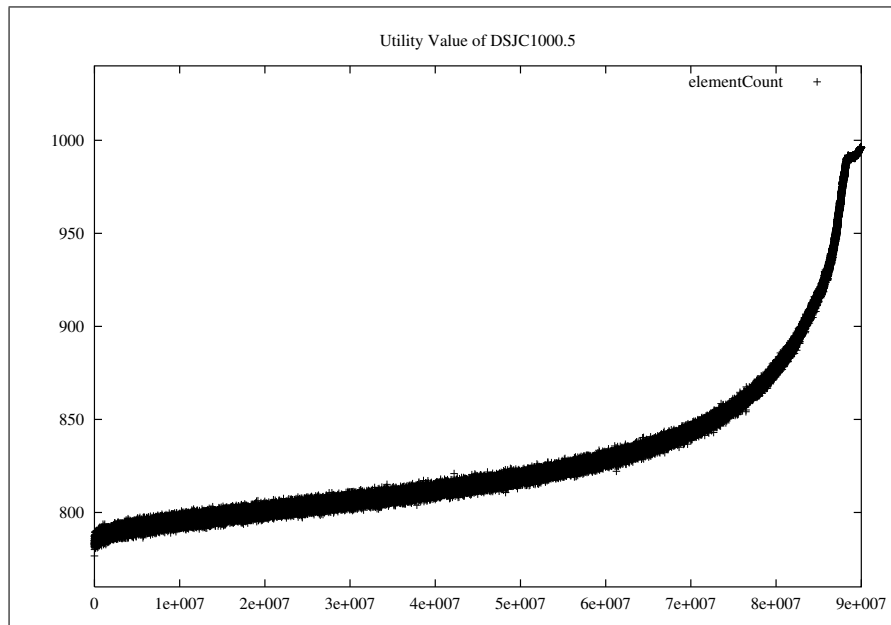


Figure 6.13. Utility value of DSJC1000.5

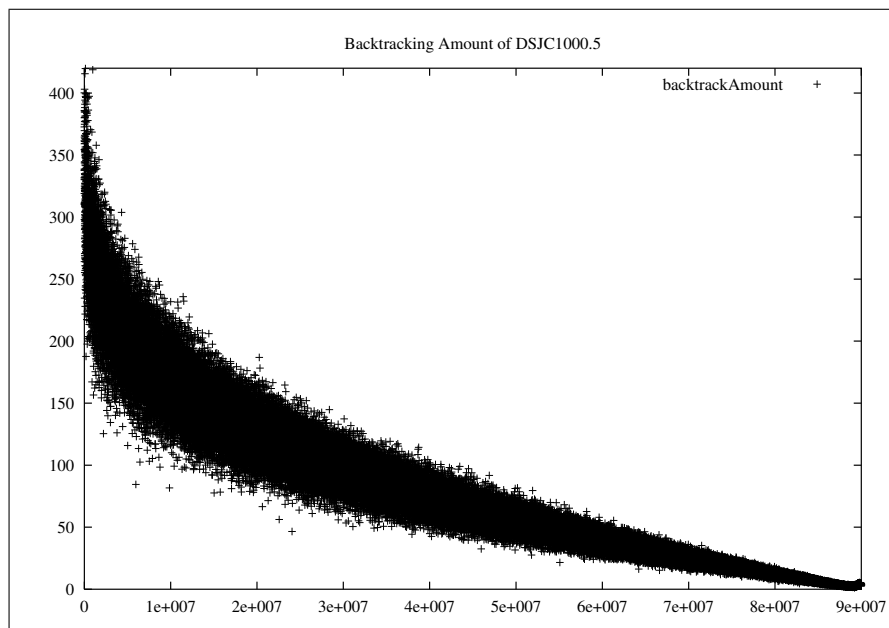


Figure 6.14. Backtracking Amount of DSJC1000.5

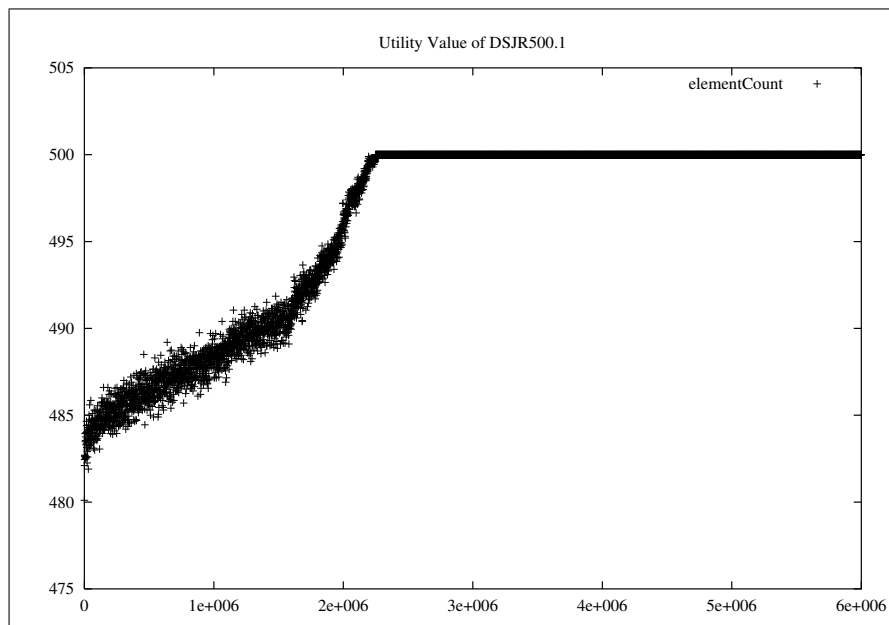


Figure 6.15. Utility value of DSJR500.1

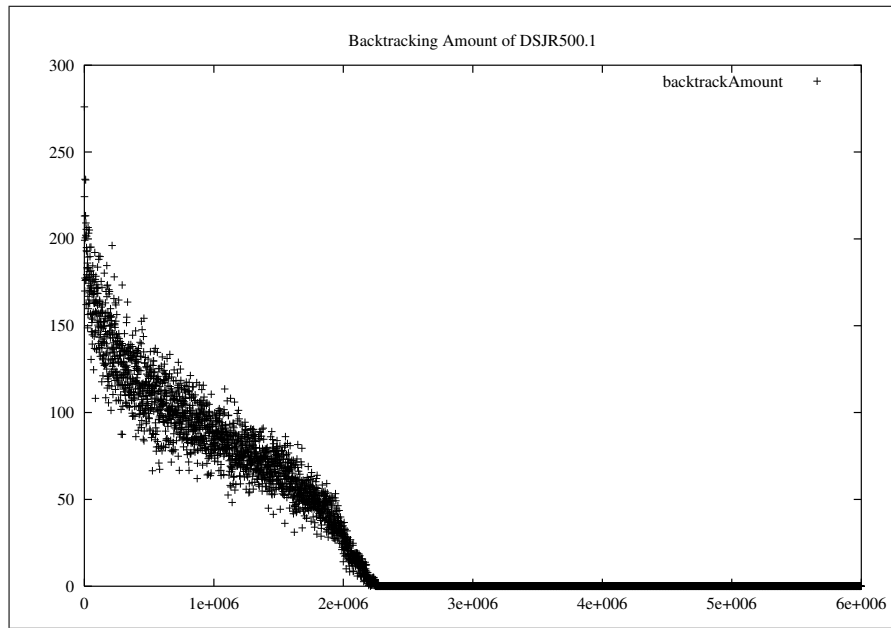


Figure 6.16. Backtracking Amount of DSJR500.1

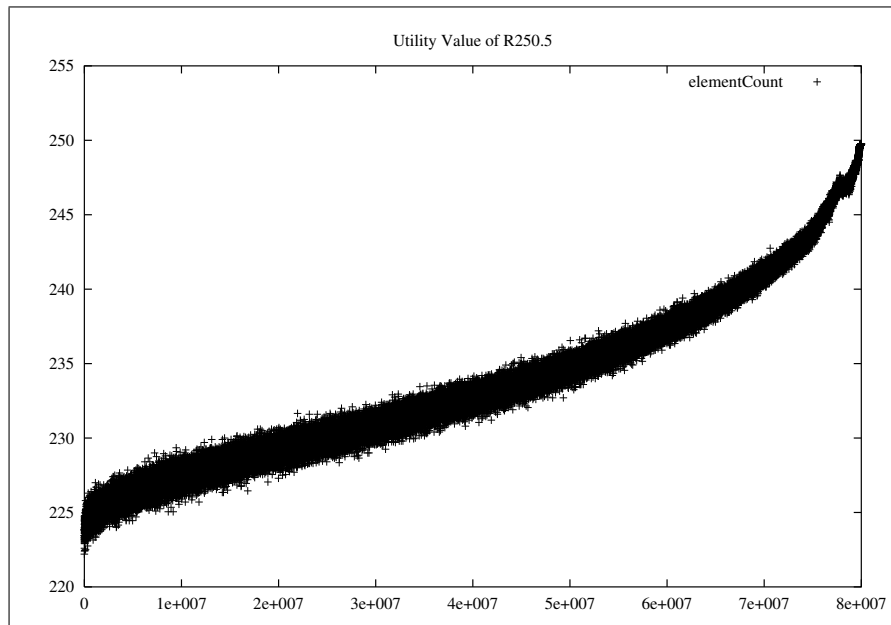


Figure 6.17. Utility value of R250.5

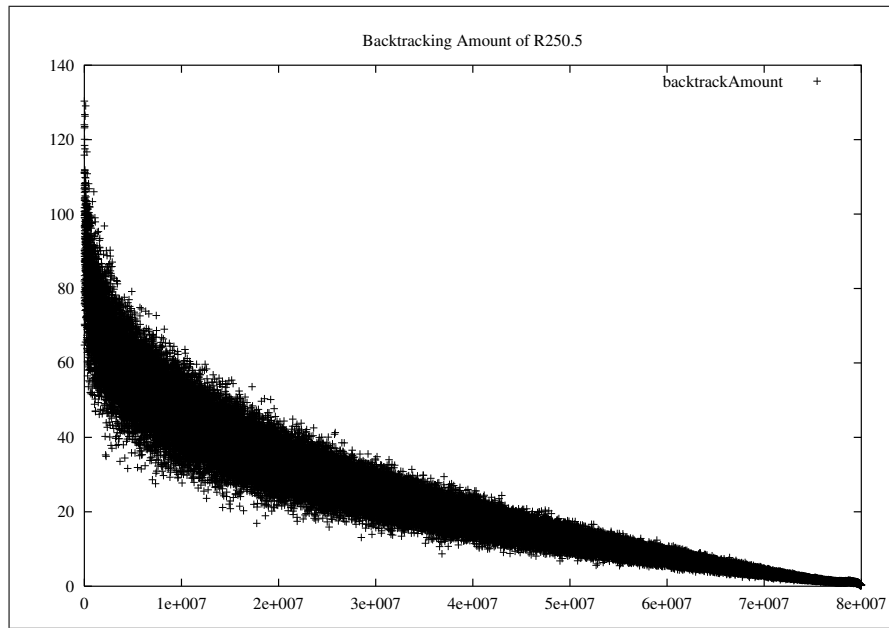


Figure 6.18. Backtracking Amount of R250.5

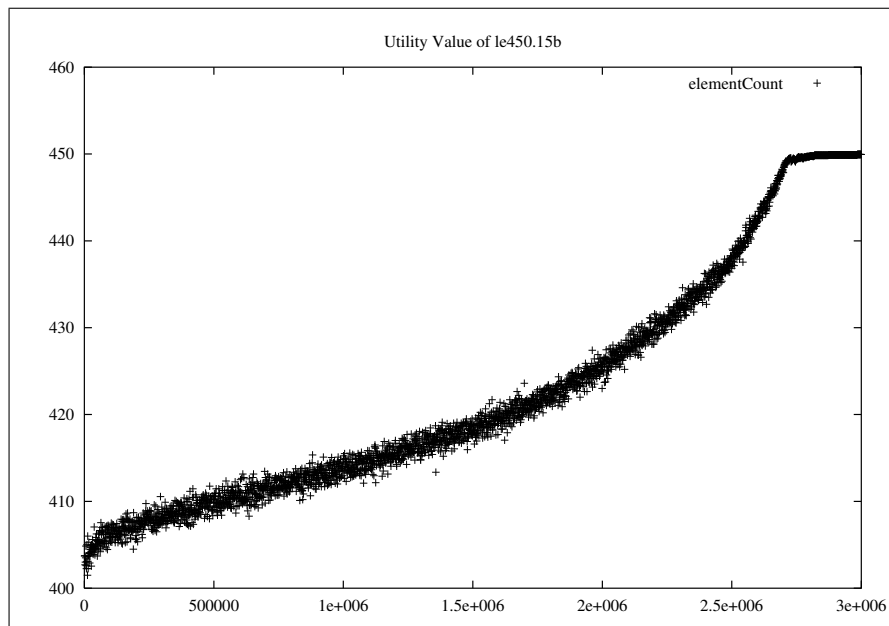


Figure 6.19. Utility value of le450.15b

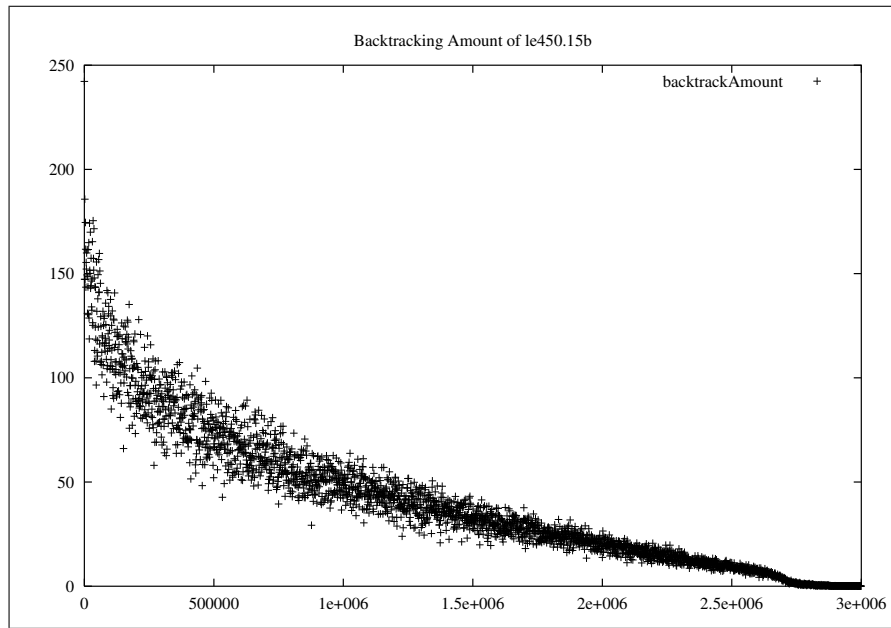


Figure 6.20. Backtracking Amount of le450.15b

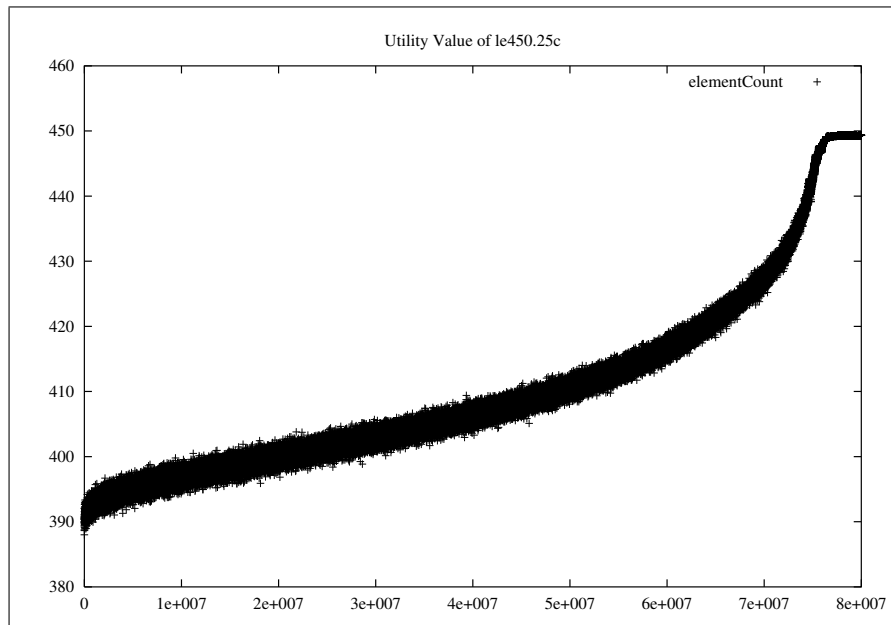


Figure 6.21. Utility value of le450.25c

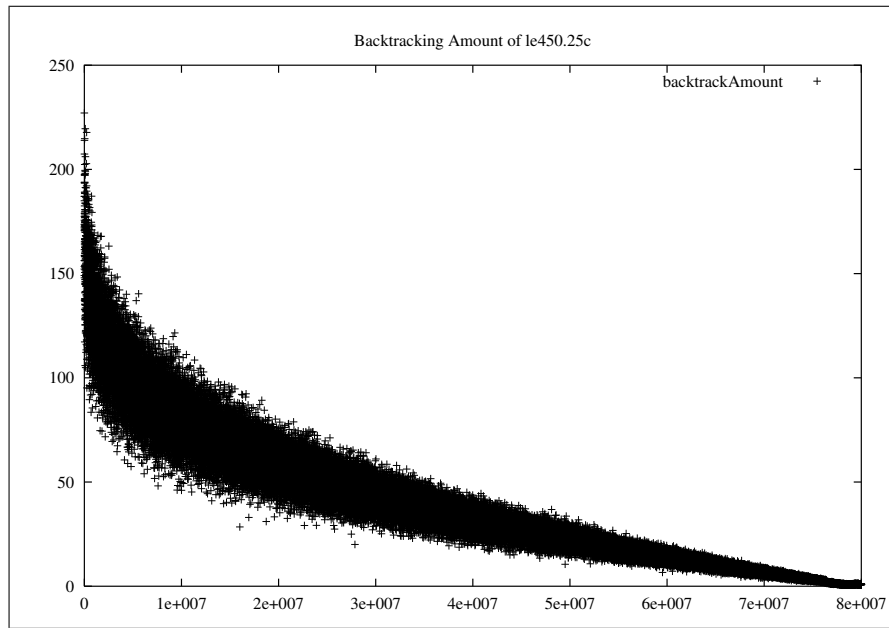


Figure 6.22. Backtracking Amount of le450.25c

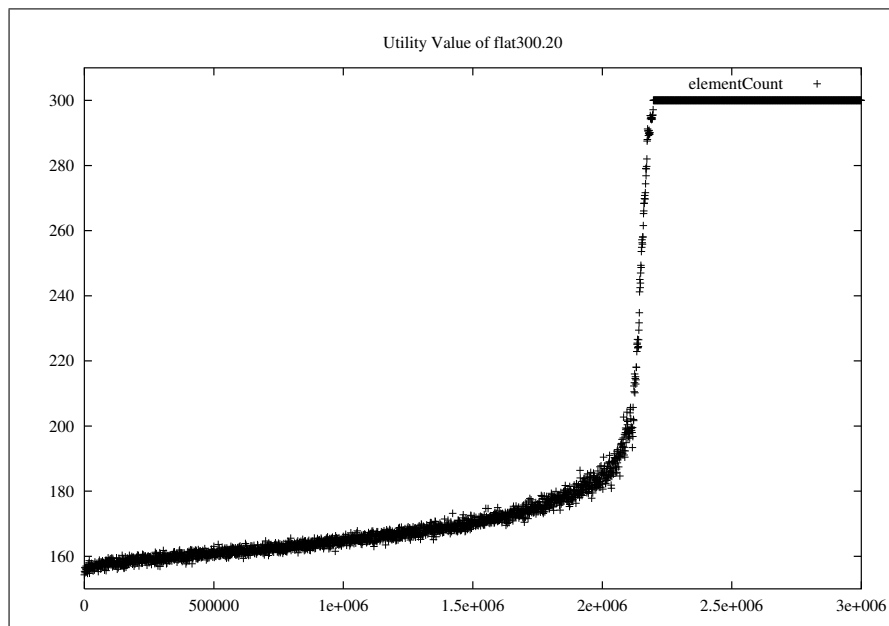


Figure 6.23. Utility value of flat300.20

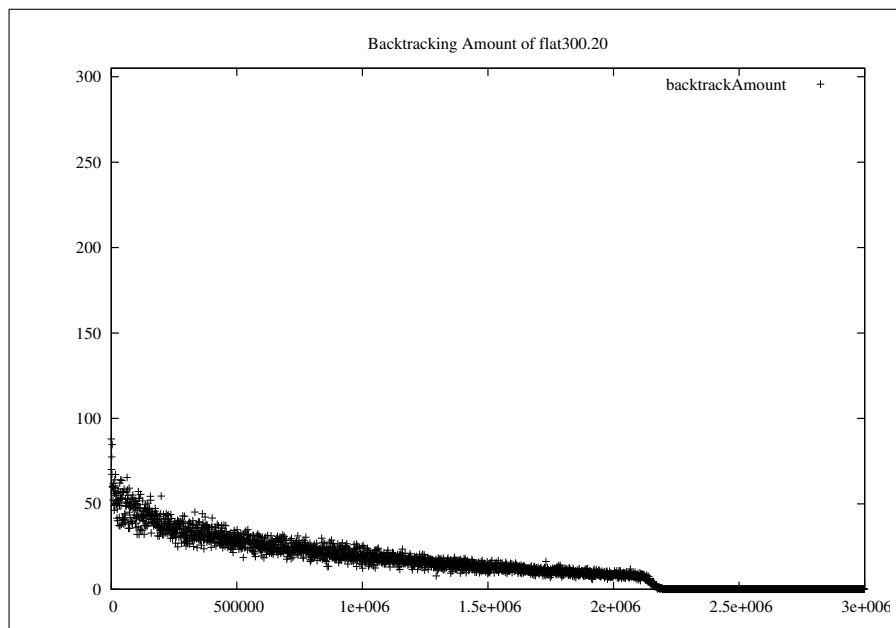


Figure 6.24. Backtracking Amount of flat300.20

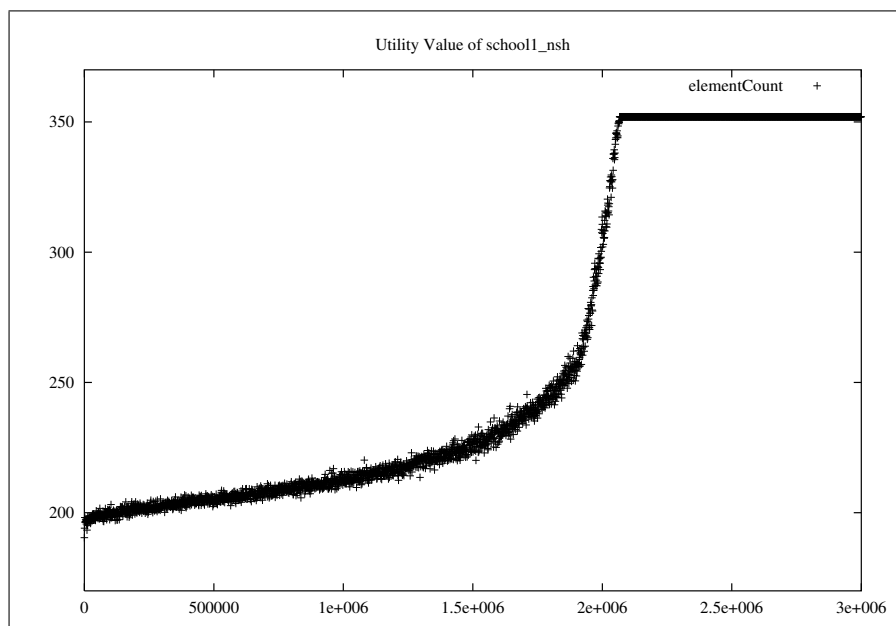


Figure 6.25. Utility value of school1_nsh

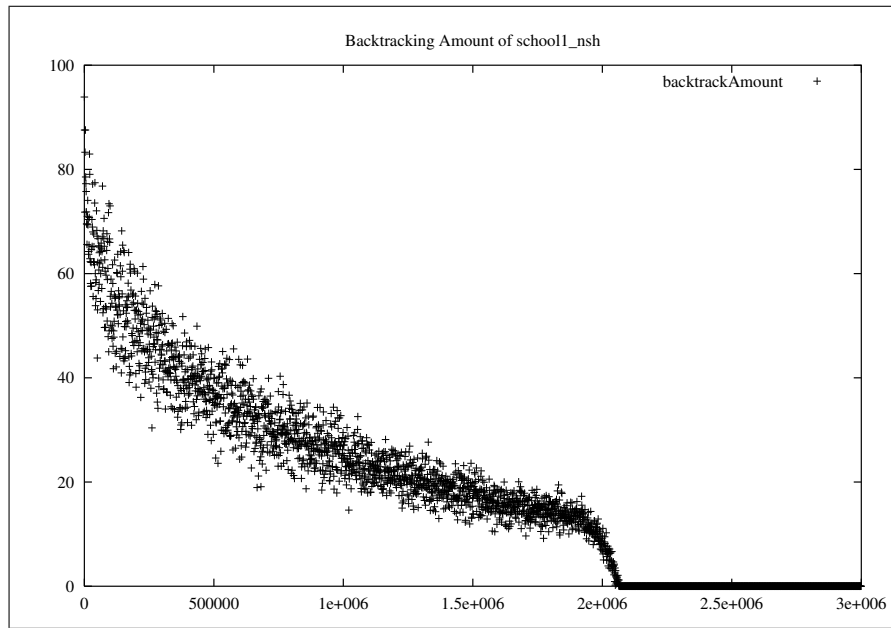


Figure 6.26. Backtracking Amount of school1_nsh

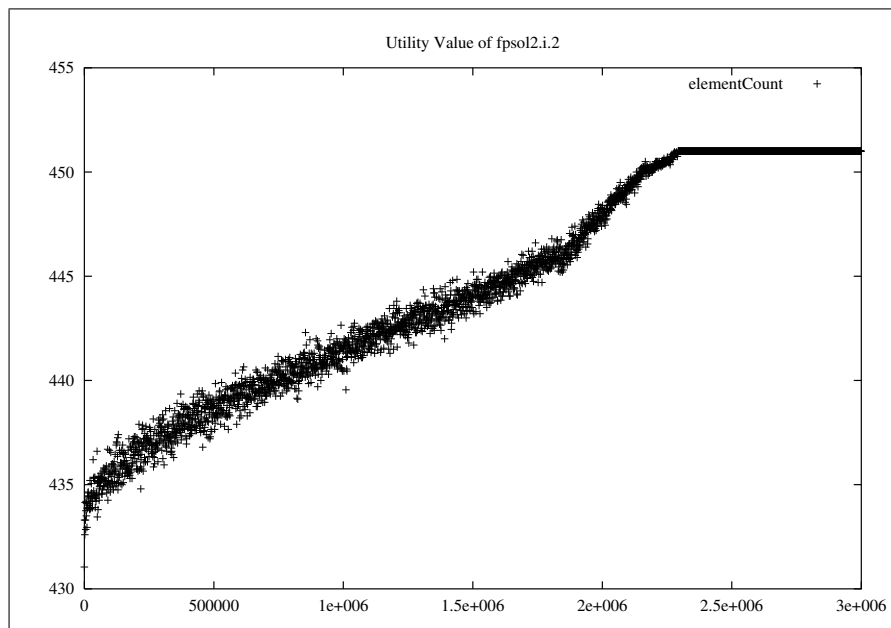


Figure 6.27. Utility value of fpsol2.i.2

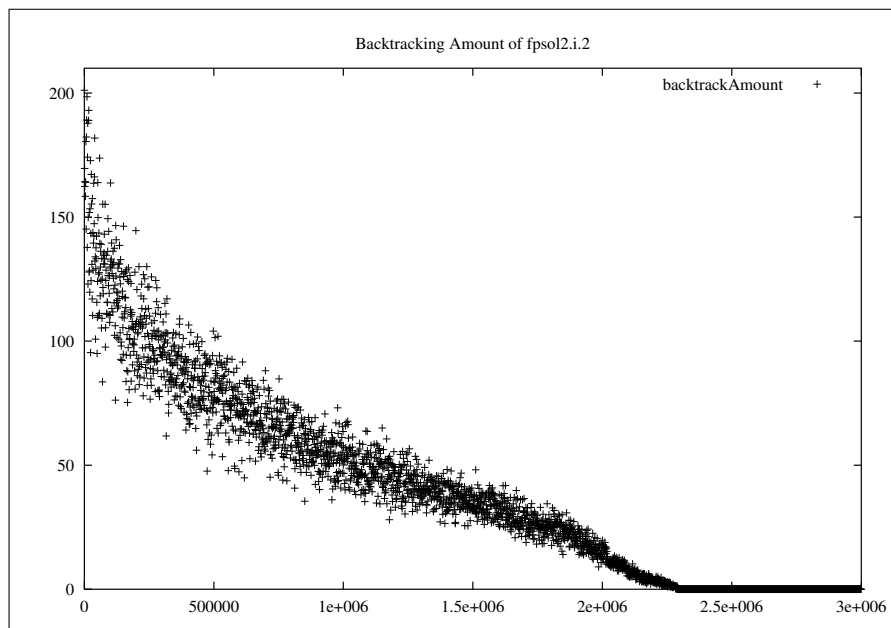


Figure 6.28. Backtracking Amount of fpsol2.i.2

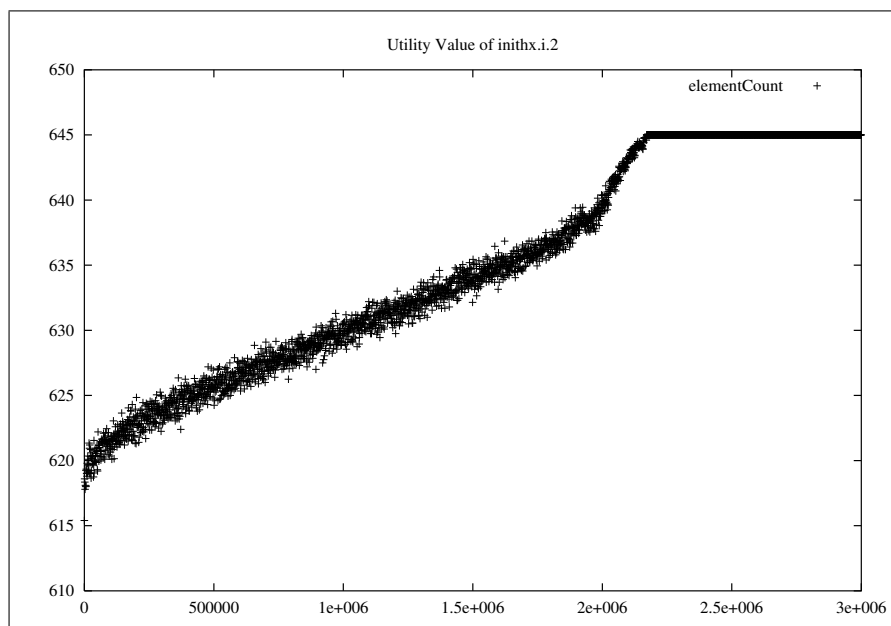


Figure 6.29. Utility value of inithx.i.2

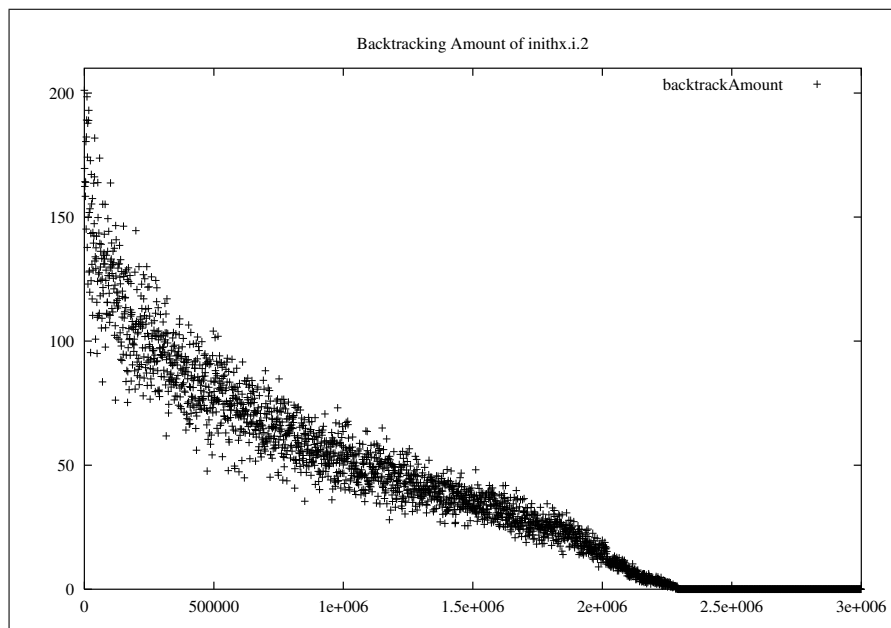


Figure 6.30. Backtracking Amount of inithx.i.2

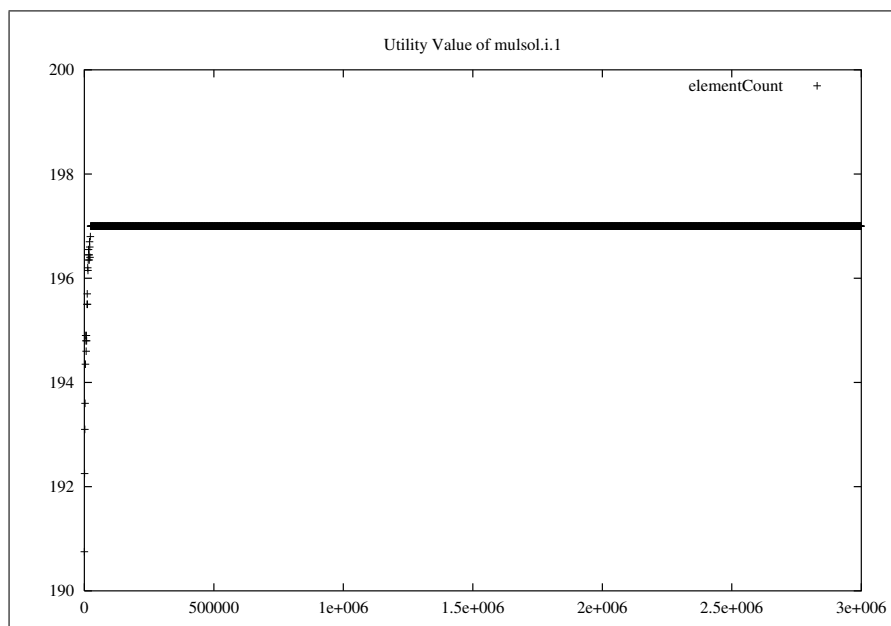


Figure 6.31. Utility value of mulsol.i.1

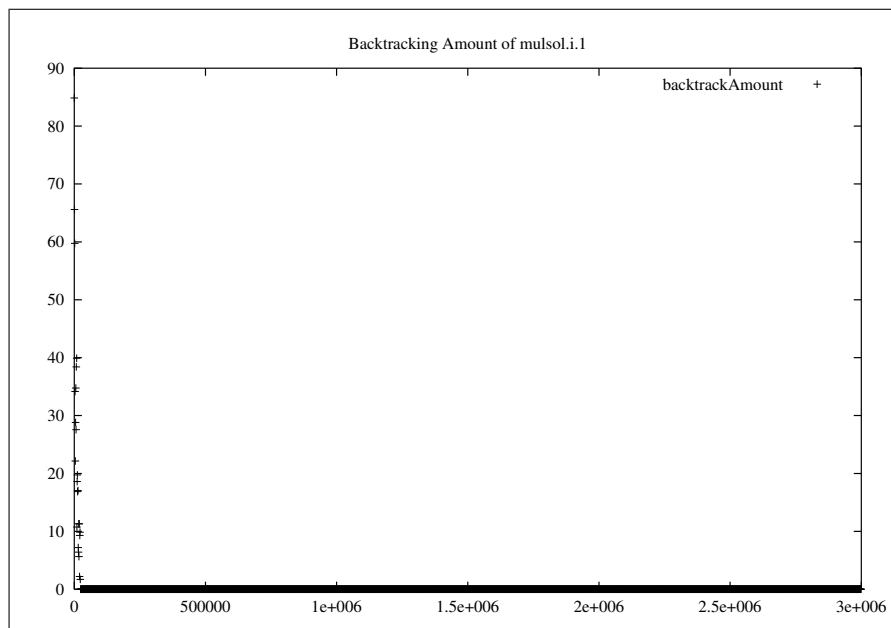


Figure 6.32. Backtracking Amount of mulsol.i.1

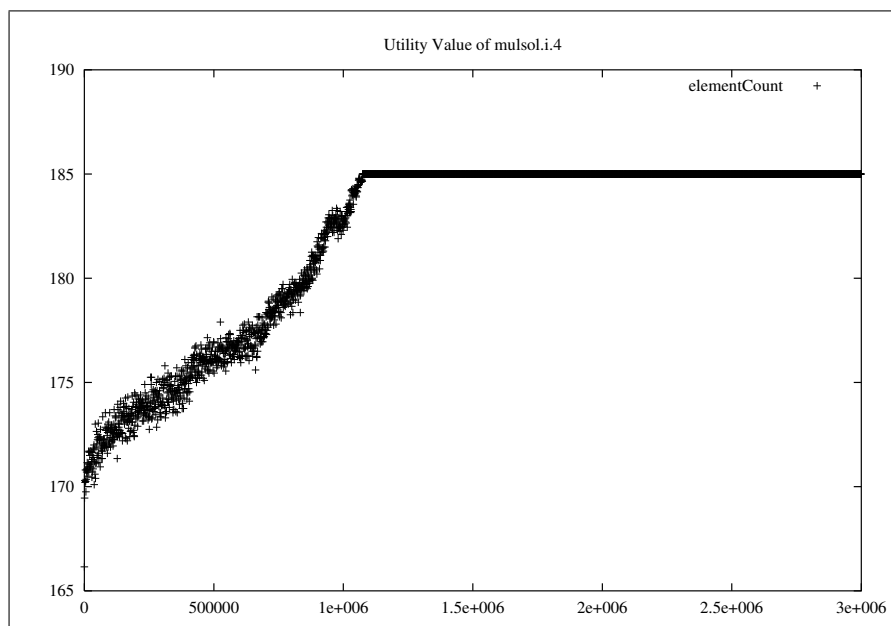


Figure 6.33. Utility value of mulsol.i.4

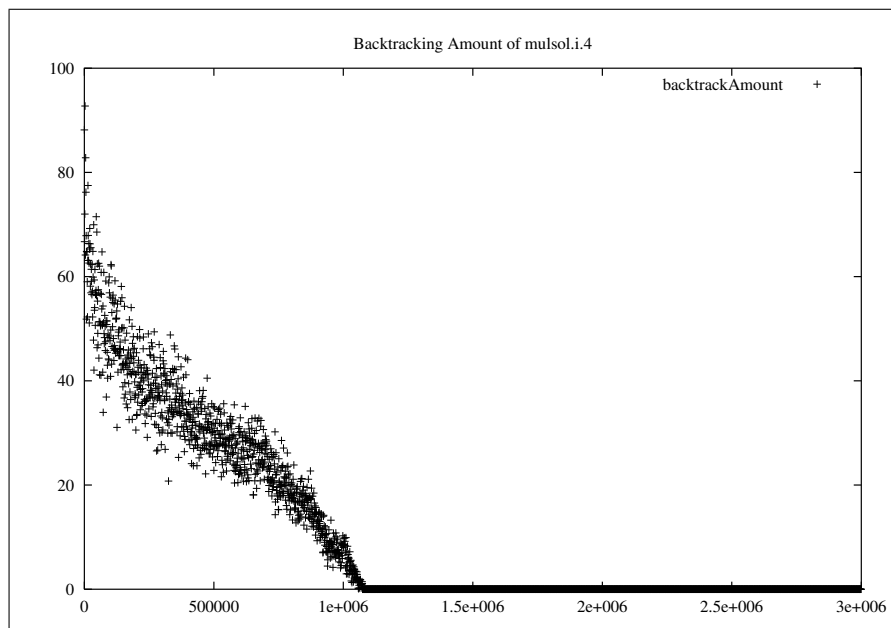


Figure 6.34. Backtracking Amount of mulsol.i.4

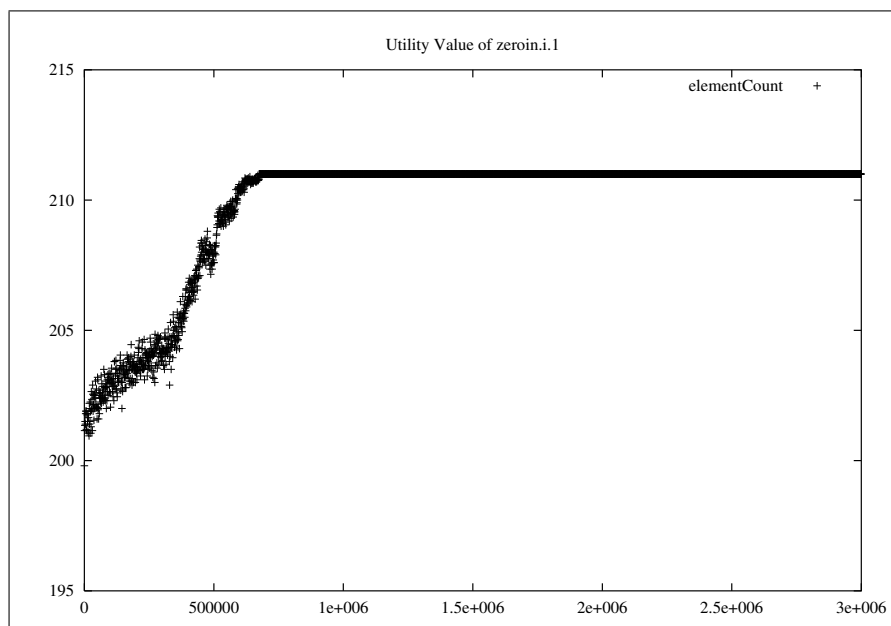


Figure 6.35. Utility value of zeroi.i.1

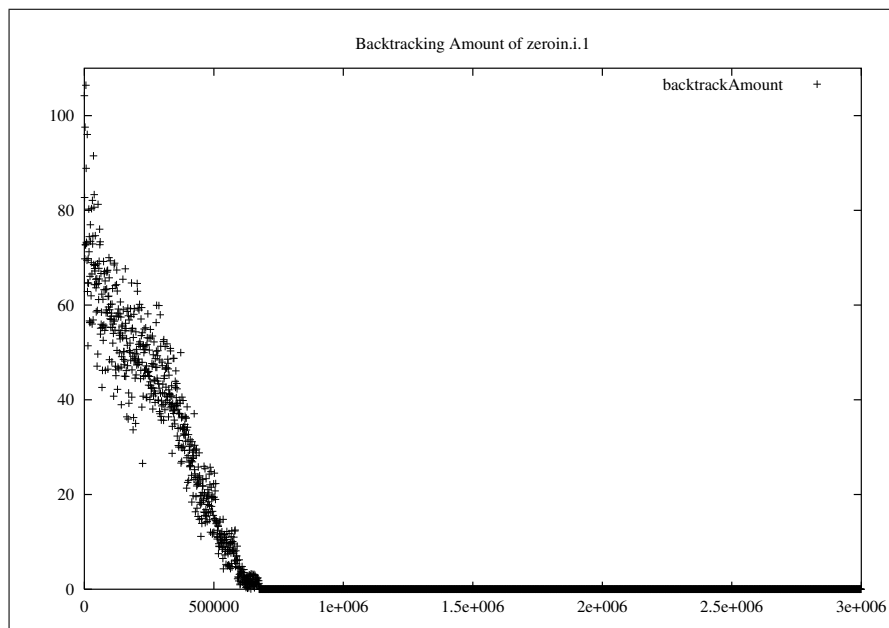


Figure 6.36. Backtracking Amount of zero.in.i.1

In Table 6.6, the initial version of SABT ($SABT_0$) and the current version are compared using paired T-test on conflict counts of 20 runs for each instance. Also, the mean values and standard deviations of both versions are provided. $SABT_0$ is run with the minimum number of colors that $SABT$ colored that instance successfully (provided in Table 6.1). Then, for each instance, mean values and standard deviations of the conflict counts are calculated and T-test is applied on these *two* sets of conflict counts. A comparison between the sets of conflict counts is done by setting the confidence interval to 99%. It is observed that except for DSJC500.1, school1 and all the instances from register allocation graphs group, the *two* sets of conflict counts are significantly different from each other. For school1, the performance of SABT in terms of solution quality is slightly better than $SABT_0$. Both algorithms can color DSJC500.1 and the instances from register allocation graphs group with the same number of colors. Hence, it is observed that SABT provides far more high quality solutions compared to $SABT_0$.

It is seen in the table that for some instances the mean values are negative. This is due to the fact that, in the algorithm, a k -coloring without any conflicts is denoted by -1 conflict count. Hence, when the algorithm terminates with a valid solution to GCP, as there is no

conflicting vertices left, it is denoted by -1 .

Table 6.6. Comparison of initial version of SABT ($SABT_0$) and the current version

		$SABT_0$		SABT		P-VALUE
		MEAN	STDEV	MEAN	STDEV	
random graphs	DSJC125.5	2.45	1.146	0.25	0.639	$3.06933E - 07$
	DSJC125.9	-0.1	0.641	-1	0	$4.96139E - 06$
	DSJC250.1	5.1	1.889	-0.75	0.444	$3.99214E - 11$
	DSJC250.9	15.05	1.638	0.6	0.821	$1.04336E - 18$
	DSJC500.5	52.8	4.34	-0.1	0.74	$2.21315E - 11$
	DSJC1000.1	127.15	2.815	0.3	1.081	$6.35646E - 32$
	DSJC1000.5	134.5	0.71	2.5	2.12	0.007261508
random geometric graphs	DSJC500.1	-1	0	-1	0	-
	R250.5	4.55	0.826	-0.8	0.41	$1.01605E - 16$
leighton graphs	le450.15b	10.05	1.82	-0.95	0.224	$1.09376E - 16$
	le450.25c	12.625	1.962	-0.438	0.512	$7.15027E - 14$
flat graphs	flat300.20	50.7	39.05	-1	0	$1.06285E - 05$
scheduling graphs	school1	5.05	17.67	-1	0	0.142154656
reg. alloc. graphs	fpsol2.i.2	-1	0	-1	0	-
	inithx.i.2	-1	0	-1	0	-
	mulsol.i.1	-1	0	-1	0	-
	mulsol.i.4	-1	0	-1	0	-
	zeroin.i.1	-1	0	-1	0	-

In Table 6.7, a comparison with evaluation function proposed in this thesis ($f(iterCnt)$) and the classical evaluation function $e^{-\Delta E/\alpha T}$ for simulated annealing algorithm is provided for some instances in terms of computational time. In the table, the names of the instances, maximum iteration count ($iterCnt_{max}$) and the average CPU times for $f(iterCnt)$ and $e^{-\Delta E/\alpha T}$ are given. α is set to 0.999995 for all the experiments done for each instance. 20 runs each with a different seed are made for each instance. The results prove that $f(iterCnt)$ is less costly in terms of computational time. Only for DSJC250.9, $e^{-\Delta E/\alpha T}$ gives a better result. Since the best colorings found are the same for both evaluation functions, they are not provided in Table 6.7.

Table 6.7. Comparison of evaluation functions

Instances	$iterCnt_{max}$	$f(iterCnt)$	$e^{\Delta E/\alpha T}$
		avg CPU time	avg CPU time
DSJC125.5	$6 * 10^6$	61.95 sec	66.75 sec
DSJC250.9	$6 * 10^6$	255.50 sec	219.37 sec
DSJC1000.1	$6 * 10^6$	1336.40 sec	1425.15 sec
DSJR500.1	$6 * 10^6$	62.15 sec	94.05 sec

7. CONCLUSION AND FUTURE WORK

In this thesis, a new hybrid meta-heuristic named SABT is proposed and applied on the famous GCP. The novel meta-heuristic proposed in this study is based on simulated annealing algorithm which is a well-known local search method. Simulated annealing algorithm is combined with another search method called backtracking. A new exponential function which avoids heavy calculations and which is based on the iteration count is proposed for the cooling down schedule of simulated annealing algorithm. SABT is an easy to implement, fast and efficient algorithm. It does not have an initialization phase or a pre-processing step. And no domain-specific knowledge is utilized in the algorithm. Hence SABT proposes a framework which can be applied to other grouping problems like bin-packing, scheduling and etc.

The experiment results are promising. SABT achieves the best results obtained in the literature for most of the instances used in the experiments. However, the performance of SABT in terms of solution quality is slightly worse than the state-of-the-art algorithms presented in Table 6.4 for large random graphs (difficult instances). As mentioned before, these are among the state-of-the-art algorithms covering the best results for the tested instances. Such algorithms either have a pre-processing step or they use domain-specific knowledge heavily. Hence, we have observed that for difficult instances, adding domain-specific knowledge will enhance the performance of SABT in terms of solution quality.

For the future work, to color difficult instances more efficiently, a hill-climber can be designed to search the immediate neighbors toward the end of the algorithm. Also, a mutational or recombination operator can be added to the algorithm. Probably adding a simple memory to the algorithm would be a better alternative. One should notice that simulated annealing algorithm does not keep previous states, however algorithms like tabu search has a finite memory to hold visited neighbors. This approach is obviously aiding in terms of searching for high quality candidate solutions. As seen in Tables 6.3, 6.4 and 6.5 the state-of-the-art algorithms compared to SABT generally utilize TS as the local search

method. Hence, a short term memory which is like a tabu list could be added to the algorithm. Another alternative for enhancing the algorithm would be utilizing SABB as a search method in a population based algorithm. Hence, diversity among individuals can be maintained by holding multiple candidate solutions at hand which could increase the performance of the algorithm further.

REFERENCES

1. Karp, R. M., “Reducibility among combinatorial problems”, in R. E. Miller and J. W. Thatcher (editors), *Complexity of Computer Computations*, pp. 85–103, Plenum Press, New York, USA, 1972.
2. Arora, S. and C. Lund, “Hardness of approximations”, pp. 399–446, 1997.
3. Bellare, M., O. Goldreich and M. Sudan, “Free Bits, PCPs, and Nonapproximability—Towards Tight Results”, *SIAM J. Comput.*, vol. 27, no. 3, pp. 804–915, 1998.
4. Garey, M. R. and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*, W. H. Freeman, January 1979.
5. Leighton, F. T., “A Graph Coloring Algorithm for Large Scheduling Problems”, *Journal of Research of the National Bureau of Standards*, vol. 84, no. 6, pp. 489–506, 1979.
6. Hamiez, J.-P. and J.-K. Hao, *An analysis of solution properties of the graph coloring problem*, pp. 325–345, Kluwer Academic Publishers, Norwell, MA, USA, 2004.
7. Hertz, A., B. Jaumard and M. P. de Aragao, “Local optima topology for the k-coloring problem”, *Discrete Applied Mathematics*, vol. 49, no. 1-3, pp. 257 – 280, 1994.
8. Porumbel, D. C., J.-K. Hao and P. Kuntz, “A search space “cartography ”for guiding graph coloring heuristics”, *Computers & Operations Research*, vol. 37, no. 4, pp. 769 – 778, 2010.
9. Lü, Z. and J.-K. Hao, “A memetic algorithm for graph coloring”, *European Journal of Operational Research*, vol. 203, no. 1, pp. 241 – 250, 2010.
10. Porumbel, D., J.-K. Hao and P. Kuntz, “Diversity Control and Multi-Parent

- Recombination for Evolutionary Graph Coloring Algorithms”, in C. Cotta and P. Cowling (editors), *Evolutionary Computation in Combinatorial Optimization*, vol. 5482 of *Lecture Notes in Computer Science*, pp. 121–132, Springer Berlin / Heidelberg, 2009.
11. Chiarandini, M., T. Stützle, F. Intelletik, F. Informatik and T. U. D. Darmstadt, “An application of Iterated Local Search to Graph Coloring Problem”, in *Proceedings of the Computational Symposium on Graph Coloring and its Generalizations*, pp. 112–125, 2002.
 12. B., I. and N. Zufferey, “A graph coloring heuristic using partial solutions and a reactive tabu scheme”, *Comput. Oper. Res.*, vol. 35, no. 3, pp. 960–975, 2008.
 13. Qu, R., E. K. Burke and B. McCollum, “Adaptive automated construction of hybrid heuristics for exam timetabling and graph colouring problems”, *European Journal of Operational Research*, vol. 198, no. 2, pp. 392 – 404, 2009.
 14. Burke, E., B. MacCloumn, A. Meisels, S. Petrovic and R. Qu, “A Graph-Based Hyper Heuristic for Timetabling Problems”, , 2007.
 15. Weicker, N., G. Szabo, K. Weicker and P. Widmayer, “Evolutionary multiobjective optimization for base station transmitter placement with frequency assignment”, *IEEE Transactions on Evolutionary Computation*, vol. 7, p. 2003, 2003.
 16. Smith, D. H., S. Hurley and S. U. Thiel, “Improving heuristics for the frequency assignment problem”, *European Journal of Operational Research*, vol. 107, no. 1, pp. 76 – 86, 1998.
 17. de Werra, D., C. Eisenbeis, S. Lelait and B. Marmol, “On a graph-theoretical model for cyclic register allocation”, *Discrete Applied Mathematics*, vol. 93, no. 2-3, pp. 191 – 203, 1999.
 18. Barnier, N. and P. Brisset, “Graph Coloring for Air Traffic Flow Management”, , 2002.

19. Zufferey, N., P. Amstutz and P. Giaccari, “Graph colouring approaches for a satellite range scheduling problem”, *Journal of Scheduling*, vol. 11, pp. 263–277, 2008, 10.1007/s10951-008-0066-8.
20. Avanthay, C., A. Hertz and N. Zufferey, “A variable neighborhood search for graph coloring”, *European Journal of Operational Research*, vol. 151, no. 2, pp. 379 – 388, 2003, meta-heuristics in combinatorial optimization.
21. Ülker, z., E. Özcan and E. Korkmaz, “Linear Linkage Encoding in Grouping Problems Applications on Graph Coloring and Timetabling”, in E. Burke and H. Rudová (editors), *Practice and Theory of Automated Timetabling VI*, vol. 3867 of *Lecture Notes in Computer Science*, pp. 347–363, Springer Berlin / Heidelberg, 2007.
22. Galinier, P. and A. Hertz, “A survey of local search methods for graph coloring”, *Computers & Operations Research*, vol. 33, pp. 2547–2562, 2006.
23. Morgenstern, C., “Distributed coloring neighborhood search”, pp. 335–358, *Discrete Mathematics and Theoretical Computer Science*, 1996.
24. Malaguti, E., M. Monaci and P. Toth, “An Exact Approach for the Vertex Coloring Problem”, *Discrete Optimization*, December 2010, in Press.
25. Brélaz, D., “New methods to color the vertices of a graph”, *Commun. ACM*, vol. 22, no. 4, pp. 251–256, 1979.
26. Johnson, D. S., C. R. Aragon, L. A. McGeoch and C. Schevon, “Optimization by simulated annealing: an experimental evaluation; part II, graph coloring and number partitioning”, *Oper. Res.*, vol. 39, no. 3, pp. 378–406, 1991.
27. Hertz, A. and D. de Werra, “Using Tabu Search Techniques for Graph Coloring”, *Computing*, vol. 39, no. 4, pp. 345–351, 1987.
28. Dorne, R., J.-K. Hao and L. Ema-eerie, “Tabu Search For Graph Coloring, T-Colorings And Set T-Colorings”, , 1998.

29. Kirkpatrick, S., Gelatt, C. D., Jr and Vecchi, M. P., “Optimization by Simulated Annealing”, *Science*, vol. 220, pp. 671–680, 1983.
30. Chams, M., A. Hertz and D. de Werra, “Some experiments with simulated annealing for coloring graphs”, *European Journal of Operational Research*, vol. 32, no. 2, pp. 260 – 266, 1987, third EURO Summer Institute Special Issue Decision Making.
31. Fleurent, C. and J. Ferland, “Genetic and hybrid algorithms for graph coloring”, *Annals of Operations Research*, vol. 63, pp. 437–461, 1996, 10.1007/BF02125407.
32. Laguna, M. and R. Martí, “A GRASP for Coloring Sparse Graphs”, *Comput. Optim. Appl.*, vol. 19, no. 2, pp. 165–178, 2001.
33. Hertz, A., M. Plumettaz and N. Zufferey, “Variable space search for graph coloring”, *Discrete Applied Mathematics*, vol. 156, no. 13, pp. 2551–2560, 2008.
34. Stützle, T., “Iterated local search for the quadratic assignment problem”, *European Journal of Operational Research*, vol. 174, no. 3, pp. 1519 – 1539, 2006.
35. Yilmaz, B. and E. Korkmaz, “Representation Issue in Graph Coloring”, in *The Tenth International Conference on Intelligent System Design and Applications (ISDA 2010)*, Cairo, Egypt, 11 2010.
36. Eiben, A., J. van der Hauw and J. van Hemert, “Graph Coloring with Adaptive Evolutionary Algorithms”, *Journal of Heuristics*, vol. 4, pp. 25–46, 1998, 10.1023/A:1009638304510.
37. Barbosa, V. C., C. A. Assis and J. O. Do Nascimento, “Two Novel Evolutionary Formulations of the Graph Coloring Problem”, *Journal of Combinatorial Optimization*, vol. 8, pp. 41–63, 2004, 10.1023/B:JOCO.0000021937.26468.b2.
38. Galinier, P., A. Hertz and N. Zufferey, “An adaptive memory algorithm for the k -coloring problem”, *Discrete Applied Mathematics*, vol. 156, no. 2, pp. 267–279, 2008.

39. Dorne, R. and J.-K. Hao, “A New Genetic Local Search Algorithm for Graph Coloring”, in A. Eiben, T. Bäck, M. Schoenauer and H.-P. Schwefel (editors), *Parallel Problem Solving from Nature V PPSN V*, vol. 1498 of *Lecture Notes in Computer Science*, pp. 745–, Springer Berlin / Heidelberg, 1998.
40. Galinier, P. and J. Hao, “Hybrid evolutionary algorithms for graph coloring”, *Journal of Combinatorial Optimization*, vol. 3, no. 4, pp. 379–397, 1999.
41. Özgür Yeniay, “Penalty function methods for constrained optimization with genetic algorithms”, *Mathematical and Computational Applications*, vol. 10, pp. 45–56, 2005.
42. Costa, D., A. Hertz and C. Dubuis, “Embedding a sequential procedure within an evolutionary algorithm for coloring problems in graphs”, *Journal of Heuristics*, vol. 1, pp. 105–128, 1995, 10.1007/BF02430368.
43. Lukasik, S., Z. Kokosinski and G. Swieton, “Parallel Simulated Annealing Algorithm for Graph Coloring Problem”, in R. Wyrzykowski, J. Dongarra, K. Karczewski and J. Wasniewski (editors), *Parallel Processing and Applied Mathematics*, vol. 4967 of *Lecture Notes in Computer Science*, pp. 229–238, Springer Berlin / Heidelberg, 2008.
44. Prestwich, S., “Constrained Bandwidth Multicoloration Neighbourhoods”, , 2002.
45. Lewandowski, G. and A. Condon, “Experiments with Parallel Graph Coloring Heuristics and Applications of Graph Coloring”, , 1994.
46. Ginsberg, M. L., “Dynamic backtracking”, *J. Artif. Int. Res.*, vol. 1, pp. 25–46, August 1993.
47. Fotakis, D., S. D. Likothanassis and S. K. Stefanakos, “An Evolutionary Annealing Approach to Graph Coloring”, in *Proceedings of the EvoWorkshops on Applications of Evolutionary Computing*, pp. 120–129, Springer-Verlag, London, UK, 2001.
48. Stützle, T. G., *Local Search Algorithms for Combinatorial Problems Analysis, Improvements, and New Applications*, Ph.D. thesis, Technischen Universität Darmstadt,

- 1998.
49. Metropolis, N., A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller and E. Teller, “Equation of State Calculations by Fast Computing Machines”, *J. Chem. Phys.*, vol. 21, pp. 1087–1092, 1953.
 50. Johnson, D. S., C. R. Aragon, L. A. McGeoch and C. Schevon, “Optimization by simulated annealing: an experimental evaluation. Part I, graph partitioning”, *Oper. Res.*, vol. 37, pp. 865–892, October 1989.
 51. Hoos, H. H. and T. Stützle, *Stochastic Local Search : Foundations & Applications (The Morgan Kaufmann Series in Artificial Intelligence)*, Morgan Kaufmann, 1 edn., September 2004.
 52. Kumar, V., “Algorithms for constraint-satisfaction problems: a survey”, *AI Mag.*, vol. 13, pp. 32–44, April 1992.
 53. Mackworth, A., *Constraint Satisfaction*, vol. 1, pp. 205–211, John Wiley & Sons, Inc., New York, NY, USA, 1987.
 54. Tsang, E., “Foundations of Constraint Satisfaction”, , 1993.
 55. Johnson, D. S. and M. Trick (editors), *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, 1993*, vol. 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, American Mathematical Society, Providence, RI, USA, 1996.
 56. Culberson, J. and F. Luo, “Exploring the k -colorable Landscape with Iterated Greedy”, pp. 245–284.
 57. Cheeseman, P., B. Kanefsky and W. M. Taylor, “Where the Really Hard Problems Are”, in J. Mylopoulos and R. Reiter (editors), *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, pp. 331–337, Morgan Kaufmann Publishers, San Francisco, CA, USA, 1991.

58. Chiarandini, M., I. Dumitrescu and T. Stützle, “Stochastic Local Search Algorithms for the Graph Colouring Problem”, in T. F. Gonzalez (editor), *Handbook of Approximation Algorithms and Metaheuristics*, Computer & Information Science Series, pp. 63.1–63.17, Chapman & Hall/CRC, Boca Raton, FL, USA, 2007.
59. Bui, T. N., T. H. Nguyen, C. M. Patel and K.-A. T. Phan, “An ant-based algorithm for coloring graphs”, *Discrete Applied Mathematics*, vol. 156, no. 2, pp. 190 – 200, 2008, computational Methods for Graph Coloring and it’s Generalizations.
60. Mehrotra, A. and M. Trick, “A Column Generation Approach for Graph Coloring”, *INFORMS Journal On Computing*, vol. 8, no. 4, pp. 344–354, 1996.
61. Walshaw, C., “A Multilevel Approach to the Graph Colouring Problem”, Tech. Rep. 01/IM/69, School of Computing and Mathematical Science, University of Greenwich, London, UK, May 2001.