University of Alberta

Search Space Structures of Local Search Algorithms for SAT

by

Guang Li ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Fall 2004

Your file Votre référence
ISBN: 0-612-95794-2
Our file Notre référence
ISBN: 0-612-95794-2

Acknowledgements

I would like to thank my supervisor, Dr. Joseph Culberson, for leading me to the research area of stochastic algorithms, and for his many helpful and crucial comments and advices on my research during my preparation for this thesis. Without his invaluable suggestions, this thesis would have been impossible.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Local search algorithms randomly select a starting point and then make moves according to the heuristics they use, in which randomness is typically employed. These algorithms are surprisingly effective for various classes of constraint satisfaction problems(CSP), such as graph coloring, scheduling and satisfaction problem(SAT). SAT problems have the binary domain, {True, false} or {0, 1}, which lets them be in the simplest form of CSP problems. In spite of their simple form, SAT problems do play a prominent role in many computer science areas. CSP problems including SAT are NP-complete. NP-complete is one of the computational complexity classes. In contrast to the class P, which is another one of computational complexity classes, NP-complete problems are not expected to be solved efficiently by any solver. In the worst-case, local search algorithms still need exponential time to solve CSP problems. However, local search algorithms are much more efficient under some classes of problems than classic search algorithms that are complete search methods, such as backtracking, backmarking and backjumping. Inevitably, local search algorithms also have their disadvantages, such as being sensitive to parameters and being unable to handle instances with no solution.

Local search became popular in early 90's and many SAT solvers based on local search procedures have been proposed since then, such as GSAT, CSAT, Novelty and R-Novelty. These local search solvers are simple but powerful. Some of them can solve hard SAT instances with even several thousands of variables, including encoded SAT instances of other CSP problems. These instances sometimes cannot be solved in reasonable time by classic SAT solvers based on popular Davis-Putnam procedures. That is why people paid attention to local search algorithms even though they are only semi-decision procedures that only work on satisfiable SAT instances. However, the good performance on large instances does not mean that local search procedures are better than systematic procedures. Actually, no one can really answer whether local search algorithms out-perform systematic search algorithms on satisfiable SAT instances or not so far. To date, researchers still do not have proper general theoretically tools for the analysis of local search algorithms because of their inherent randomness and those existing theoretical analyses are usually limited in the practical use. Hence, it is difficult to analyze local search procedures theoretically. Therefore, many researchers still analyze local search methods using empirical methods.

In this thesis, we analyze GSAT and WalkSAT local search procedure families using empirical methods too. These two families contain two major local search architectures in the SAT research area and they can be easily extended and have been confirmed to be efficient in experiments. The structure of search spaces and the convergence speed of these local search algorithms are two main issues of this thesis. A search space basically represents all the possible states and moves of a local search procedure. Our research focuses on the coverage of traps and the convergence speed on search spaces. Traps consist of assignments that local search procedures should avoid. A local search procedure associated with search space graphs containing fewer traps has less chance to get stuck in traps. The convergence speed measures how fast an algorithm converges to the sinks. The solutions are a part of those sinks. Therefore, a faster convergence speed is desired. On the other hand, the convergence speed is directly affected by the average out-degree of states in search spaces. We empirically confirmed the correlations among the coverage of traps, the average out-degree and algorithm performance through our study.

In Chapter 2, we present the basic concepts of SAT and CSP as well as some systematic and local search methods for SAT in detail. The most well-known systematic search procedure, such as the Davis-Putnam procedure(DP) and its variant Davis-Putnam Logemann Loveland procedure(DPLL), are introduced, and then GSAT and WalkSAT local search procedure families. After that, we introduce the new algorithm from Schuurmans et. al [35], SDF, which performed well under their three measurements for the evaluation of local search algorithms.

Some related work on the search spaces of local search algorithms are introduced in Chapter 3. The early work on escaping plateaus and traps by Selman et. al. and some theoretical work on the hardness of instances are introduced first. Then, we explain Clark et. al and Schuurmans and Southey's work in state spaces in detail. After that, we discussed the work of Frank et. al. on the topology of local search spaces.

In Chapter 4, to give an outline of the performance of local search algorithms, we designed experiments for the comparison of local search algorithms' performance. We compared the performance between local search algorithms and a systematic search algorithm, where the performance was measured by actual running time. We also provide the performance comparisons among the local search algorithms as well, where the performance was measured by the number of flips. By the comparison, we hope readers gain a concrete image on how good or bad those algorithms are.

Finally, we presented our research work on the structure of the search space graphs and the convergence speed under some local search algorithms in Chapter 5. In the first half of this chapter, we carefully studied the coverage of traps of the search spaces of GSAT and WalkSAT families. In the second half, we simulated the searching process on the search space graph using stochastic matrices. By comparing the difference of the convergence speeds of these algorithms, we can see how the differences among search space graphs affects the search performance. We empirically confirm that small values on the coverage of traps and the average out-degree lead to a good performance.

Chapter 2

Background

2.1 Satisfiability (SAT) Problem

In the computational complexity area, problems are categorized into many classes. P, NP and NP-Complete are three of them. The class P consists of problems that can be solved in polynomial time under the input size n, which means that there exists a k such that the problems can be solved in time $O(n^k)$. The class NP consists of those problems that can be verified in polynomial time which means that given a certificate of a solution for a problem we can verify this certificate in polynomial time under the size of the input. Clearly, the class P is a subset of the class NP. If a problem can be proven to be as "hard" as any problem in the class NP, this problem is in the class NP-Complete. The NP-Complete problems are usually considered as the barrier separating the computational tasks that can be solved in realistic time and resources from those that cannot.

The Satisfiability(SAT) problem is the first problem proved to be in the class NP-Complete. It holds a central position in the study of computational complexity. A SAT instance consists of[5]:

1. A set of variables $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$, whose domains are $\{0, 1\}$;
2. A set of literals $\mathcal{L} = \{l_1, l_2, \ldots, l_{2n}\}$, each of which is equal to x_i or $\neg x_i$, where $i = 1, 2, \ldots, n$;
3. A set of clauses $\mathcal{C} = \{C_1, C_2, \ldots C_m\}$, each of which contains a subset of the literal set $\mathcal{L}$.

The form of $C_1 \wedge C_2 \wedge \cdots \wedge C_m$ is called the Conjunctive Normal Form(CNF). We will call a SAT instance $f = C_1 \wedge C_2 \wedge \cdots \wedge C_m$ a CNF *formula*. All of first order logic formulas can be translated into CNF since all basic logic connectives have their own equivalent CNF. For example, $x_1 \rightarrow x_2 \Leftrightarrow \neg x_1 \vee x_2$.

A solution of a SAT instance is an assignment to all variables in the instance such that each clause contains at least one literal whose value is 1. The value of a literal l_j is 1 when $x_i = 1$ if $l_j = x_i$ or $x_i = 0$ if $l_j = \neg x_i$. For example: $f = \{C_1, C_2, C_3\}$ is a SAT instance where $C_1 = \{x_1, x_2, \neg x_4\}$, $C_2 = \{\neg x_1, x_3, \neg x_4\}$, $C_3 = \{x_2, \neg x_3, x_4\}$. Then, both $x_1 = 0, x_2 = 1, x_3 = 0, x_4 = 0$ and $x_1 = 1, x_2 = 1, x_3 = 1, x_4 = 1$ are solutions of the formula $f = (x_1 \vee x_2 \vee \neg x_4) \wedge (\neg x_1 \vee x_3 \vee \neg x_4) \wedge (x_2 \vee \neg x_3 \vee x_4)$. Solutions are frequently denoted as tuples. For instance, the two solutions mentioned above can also be represented as $s_1 = (0, 1, 0, 0)$ and $s_2 = (1, 1, 1, 1)$. The *solution space* $\mathcal{S}$ of a SAT instance is the set of all of the solutions to this SAT instance. The assignment space of a formula f contains all possible 2^n assignments for $x_i (i = 1, 2, \ldots, n)$. If an instance contains n variables, the size of the assignment space is 2^n. If a formula's solution space is not empty, we say that the formula is *satisfiable*; if it is empty, we say it is *unsatisfiable*.

If each clause in a CNF formula contains exactly k literals, this formula is called a k-SAT formula. For example, $f = (x_1 \vee x_2) \wedge (\neg x_2 \vee x_3) \wedge (x_1 \vee x_3)$ is a 2-SAT CNF formula. k-SAT problems are a subclass of SAT. When $k = 2$, this subclass(2-SAT) is in class P; when $k \geqslant 3$, k-SAT is in class NP-Complete. If $P \neq NP$, then we do not expect any efficient algorithm to solve k-SAT($k \geqslant 3$) formulas. Therefore, how to solve SAT problems with less effort and why SAT problems are hard become important issues.

2.2 Solving SAT problems

SAT is a special case of Constraint Satisfaction Problems(CSP) which is another well-known NP-Complete problem. A CSP problem consists of[12] :

1. A set of variables $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$;
2. A set of domains $\mathcal{D} = \{D_1, D_2, \ldots, D_n\}$, each of which is the finite domain of the corresponding variable x_i.
3. A set of constraints $\mathcal{C} = \{C_1, C_2, \ldots, C_m\}$ which restrict the values of variables in $\mathcal{X}$;

A solution of a CSP is an assignment $V = (v_1, v_2, \ldots, v_n)$ to all variables in $\mathcal{X}$, which satisfies all constraints in $\mathcal{C}$, where v_i is a value in D_i for the variable x_i. SAT is just a

CSP with all domains $D_1, D_2, \ldots, D_n$ being binary, $\{0, 1\}$ or $\{false, true\}$. Therefore, CSP solvers can solve SAT problems directly. These solvers may be roughly categorized into two classes: complete search algorithms and incomplete or local search algorithms.

Complete search algorithms can search the assignment space completely, by verifying every assignment in the assignment space. "Complete" here means to check all possible assignments without missing or rechecking. Of course, these algorithms are able to skip some redundant assignments which can be proved to be irrelevant in advance. Considering that the size of $\mathcal{A}$ increases exponentially with the number of variables, we do not expect that complete search algorithms are able to solve instances containing many variables.

Local search algorithms, sometimes called incomplete search algorithms, search the assignment space $\mathcal{A}$ heuristically using local information. Usually, this local information causes these algorithms to search $\mathcal{A}$ partially or incompletely. Because they cannot recognize the assignments they have checked such local search algorithms may check some assignments repeatedly. Therefore, those local search algorithms can only solve satisfiable instances because they can never tell whether there is no solution for a given SAT instance or not when they do not find any.

Under some conditions, a local search algorithm may keep searching in some closed subsets of the assignment space $\mathcal{A}$ and will never get out of these closed subsets. We say that these subsets are *traps* and the algorithm gets stuck in traps. Besides this, local search algorithms may also be sensitive to parameters. A proper parameter may lead to significantly better performance. In spite of these defects, however, local search algorithms have their advantages, such as, their storage space is usually small and they can solve some large size SAT problems efficiently. For example, using some effective heuristics, the most efficient complete search procedures can solve up to about 350 variable formulas in about one hour in 1992(Buro and Büning [3]). GSAT can solve problems of the same size 10 times faster(Selman and Kautz [36]). This does not always hold, though. For example, highly structured instances, such as blocks-world planning formulas, can be solved with less effort by specialized complete search procedures using the unit propagation technique. The unit propagation can recursively remove values in a domain that conflict with the values in other domains. Because of the structures contained in those problems, the search space can be dramatically reduced by large numbers of unit propagations. However, GSAT does not have any mechanism for handling the unit propagation so it can not solve those problems as efficiently as the specialized complete search algorithms. Although they improved its performance with some mechanisms[36], Selman et al. note that they do not claim that "GSAT will be able to outperform backtracking search methods on all possible problems". Actually they believe some certain highly structured problems can be solved more efficiently by exhaustive search approaches and domain specific heuristics.

2.3 Resolution and Davis-Putnam Method

Because variable domains are binary domains in SAT, some search procedures are designed specially for SAT problems. Many of these procedures are based on the Davis-Putnam method. We will introduce Davis-Putnam(DP) and Davis-Putnam Logemann Loveland(DPLL) procedures in Section 2.3.2 and Section 2.3.3 respectively.

2.3.1 Resolution

Resolution is the technique used by many SAT solvers. DP and DPLL are two of the most well-known ones. They use general resolution and unit-resolution respectively, where unit-resolution is a special case of general resolution.

The general resolution rule[2]: If $C_i \in f$, $C_j \in f$, where f is a formula, and $\neg x_k \in C_i$, $x_k \in C_j$, the resolvent is $C_i \cup C_j - \{x_k, \neg x_k\}$.

For instance:

$$\left\{ \begin{array}{l} \neg x \vee y_1 \vee z_1 \\ \\ x \vee y_2 \vee z_2 \end{array} \right. \quad \Longrightarrow \quad y_1 \vee z_1 \vee y_2 \vee z_2$$

It is easy to prove that:

1. t is an assignment s.t. $t(C_i) = 1$ and $t(C_j) = 1 \Rightarrow t(C_i \cup C_j - \{x_k, \neg x_k\}) = 1$, where $t(C) = 1$ or 0 means that the value of clause C is 1 or 0 under the assignment t;

2. t is an assignment such that $t(C_i \cup C_j - \{x_k, \neg x_k\}) = 1 \Rightarrow \exists t'$ s.t. $t'(C_i) = 1$ and $t'(C_j) = 1$.

So if a formula f is able to be transformed into f' by adding a resolvent, then f is satisfiable if and only if f' is satisfiable.

If the clause C_j mentioned above is a unit clause $\{x_k\}$ and C_i contains $\neg x_k$. The resolvent is $C_i - \{\neg x_k\}$. We call this process *unit-resolution*[2].

2.3.2 Davis-Putnam Procedure(DP)

Algorithm 1 Davis-Putnam Procedure

```
while true do
  if There exist empty clauses then
    RETURN Unsatisfiable
  end if
  while ∃ a variable $x$ which only occurs in one phase(positive($x$) or negative($\neg x$)) do
    Delete all clauses containing the variable $x$;
  end while
  if There exists no clause then
    RETURN Satisfiable
  end if
  if There exists a variable $x$ occurring in both phases (positive and negative) then
    Add all possible resolvents based on the variable $x$ into the formula
    Delete all clauses containing the variable $x$
  end if
end while
```

Algorithm 1 is the pseudocode of the Davis-Putnam procedure, based on the description of the Davis-Putnam procedure in Cook and Mitchell's paper[5]. It generates a sub-problem with one less variable at each step. The termination condition of this procedure is the existence of an empty clause or an empty formula(a formula containing no clause) and the procedure ends in finite iterations. First, the appearance of an empty clause implies there is no solution for the original formula. An empty clause may be produced by a sequence of resolution operations. For example: Clauses $C_1 = \{x_1, x_2\}$, $C_2 = \{\neg x_1, x_2\}$, $C_3 = \{x_1, \neg x_2\}$ and $C_4 = \{x_1, \neg x_2\}$ can lead to clauses $C_{13} = \{x_1\}$ and $C_{24} = \{\neg x_1\}$, and then C_{13} and C_{24} lead to an empty clause $\{\}$. Second, The appearance of an empty formula means that the formula is satisfiable and a solution for the formula has been found

However, the DP procedure has some disadvantages. The most serious problem is that it may generate quadratically more clauses at each step when adding resolvents into the clause set. The search process possibly produces too many resolvents to manage sometimes. For example, if both x and $\neg x$ occur in half of the clauses in a formula, the number of clauses will increase quadratically. Another problem is that the clauses may become longer and longer. ~~These mean DP may consume a huge storage space in a short time.~~

2.3.3 Davis-Putnam Logemann Loveland (DPLL)

Davis-Putnam Logemann Loveland(DPLL) is a variation of the basic Davis-Putnam procedure mentioned in Section 2.3.2. DPLL is a back-tracking depth first search procedure, using the unit resolution operation. First, the term of "subsume"[2] is defined, which is related to "subset". C_1 subsumes C_2 where C_1 and C_2 are both clauses, and $C_1 \subset C_2$. Any assignment that satisfies C_1 must also satisfies C_2. In other words, the solution space $\mathcal{S}_{C_1} \subset \mathcal{S}_{C_2}$, where $\subset$ means that the set of the solution for C_1 is included in the set of the solutions for C_2. The pseudocode of DPLL is shown in Algorithm 2[5]

Algorithm 2 DPLL(f)

```
Apply all possible unit resolutions;
Remove all subsumed clauses;
if $f$ is an empty formula then
  RETURN satisfiable
end if
if $f$ contains empty clauses then
  RETURN unsatisfiable
end if
while $\exists$ a variable $x$ which occurs in one phase(only positive or negative) do
  Remove all clauses containing the variable $x$;
end while
Select a variable $x$ in $f$;
if DPLL($f \cup \{x\}$) = satisfiable then
  RETURN satisfiable
end if
RETURN DPLL($f \cup \{\neg x\}$)
```

Using the unit-resolution, the DPLL requires smaller space than DP since at each step unit-resolvents generated are shorter than the original clauses and only one unit clause is added into the formula. We may notice that the DPLL(f) is a recursive procedure by calling DPLL($f \cup \{x\}$) and DPLL($f \cup \{\neg x\}$). Selecting a variable leads to a smaller subproblem: when the unit clause $\{x\}$ is added into the formula f, the assignments with $x = 1$ are verified; when the unit clause $\{\neg x\}$ is added into the formula f, the assignments with $x = 0$ are verified. So we say that the DP procedure uses an "elimination rule" while the DPLL procedure uses a "splitting rule". The methods for selecting the variable x can be as simple as selecting the first variable in a randomly ordered variable list but can also be sophisticated. Selection heursitics have been widely studied in the past thirty years. Using various heuristics on selecting variables we have various DPLL variations. These variations of DPLL work fairly well in practice and are " probably the most widely known and used SAT testing method[5]".

In recent years, many new efficient SAT solvers have been built, such as SATZ, EQSATZ and SATO. Most recent progress on the SAT problem can be found at http://www.satlive.org and http://www.satlib.org.

2.4 Phase transition and Hardness

Many computational complexity analyses are based on the worst-case analysis. In practice, average-case analysis may be more practical. In the process of studying random formulas, researchers found threshold phenomena, that is, the probability that instances in some subclasses of SAT are satisfiable drops from 1 to 0 very quickly in a narrow range under the value of the ratio of clauses-to-variables. For the convenience of description, we will represent the ratio of clauses-to-variables as the C/V ratio in the following chapters. The phase transition is the transition from the phase in which the probability of the instances being satisfiable is 1 to the phase in which the probability of the instances being satisfiable is 0. The study of phase transitions focuses on the following conjecture,

$$\lim_{n\to\infty} Pr\{\text{Random } k - SAT \text{ with } n \text{ variables and } m \text{ clauses is satisfiable}\} = \begin{cases} 1, & if\ m/n < C_k \\ 0, & if\ m/n > C_k \end{cases} \tag{2.1}$$

[16] and on determining the threshold C_k. Random 3-SAT was investigated in Selman et. al paper, 1992[29]. The empirical threshold of Random 3-SAT is approximately 4.3

It was noticed that the random 3-SAT instances are easy to solve when the C/V ratio is small, say less than 3, and the instances become relatively easy again when the C/V ratio is large, say more than 6.5. The hardness peak is at the C/V ratio near 4.3 which is also the threshold of the random 3-SAT instances. This pattern is called the easy-hard-easy pattern. Although we call it "easy-hard-easy", the two "easy" regions are not equally easy. Mitchell and Levesque's paper[28] shows that the hardness of the second "easy" region increases when the number of variables increases. Intuitively, the first "easy" region exists because it is easy to find solutions for instances in this region while the reason for the existence of the second "easy" region is that it is relatively easy to prove the unsatisfiability of the instances in this region. Beame et. al.[1] give a lower bound of $2^{\Omega(n/\Delta^{4+\epsilon})}$ for the DP procedure proving unsatisfiability for random 3-SAT instances, where Δ is the C/V ratio, n is the number of variables and $0 < \varepsilon < 1$ is a constant. So if the hardness of the second "easy" region mainly comes from the hardness of proving unsatisfiability of instances, the hardness of this region will increase exponentially when the number of variables increases. On the other hand, with a fixed number of variables the hardness becomes easier when the C/V ratio increases.

Larrabee and Tsuji [26] found the same pattern using two substantially different algorithms. So people conjectured that this easy-hard-easy pattern will hold for all complete search algorithms. The easy-hard-easy pattern and phase transition were not only found in random 3-SAT problems, but also in k-SAT$(k > 3)$ problems, which is presented in Mitchell and Levesque's paper[28]. Their experiments indicated that the larger the k the larger Cf_k is.

2.5 Local Search Algorithms

A typical local search method starts with an arbitrary complete assignment for a SAT instance and tries to improve this complete assignment according to the algorithm's evaluation function. Every iteration, it re-assigns values to one or more variables selected by some heuristic methods. We usually call the process of re-assigning value to a variable a "flip" because the domain of a variable is $\{0, 1\}$ and re-assigning the value of a variable is a "flip" from 1 to 0 or 0 to 1. We set a limit on the number of flips. When a local search reaches this limit, it will stop searching or restart another search. See Algorithm 3, which is a general framework of local search algorithms. Different algorithms may have different optimal values for this limit on the number of flips. Some algorithms converge slowly. These algorithms need more flips to converge to solutions. For many local search algorithms, there exist subsets of assignments in the assignment space such that once these algorithms visit an assignment in these subsets they will never be able to visit any assignment outside the subsets. If none of the assignments in the subsets is a solution, these subsets are called *traps*. If a local search algorithm is stuck in a trap, the algorithm will never find a solution without a restart. Some algorithms have more traps than others. These algorithms will be more affected by restarts. They need to restart more frequently.

In Algorithm 3, there is a function *SelectHeuristic(C, T)*. It is a heuristic function for the variable selections. Various local search algorithms have various implementations of

Algorithm 3 General Framework of the Local Search SAT Solver

```
$\mathcal{C}$ = The Set of Clauses;
for ($i = 0$; $i < MaxNumTries$; $i{+}{+}$) do
  $T$ = Randomly Generate A Complete Assignment;
  for ($j = 0$; $j < MaxNumFlips$; $j{+}{+}$) do
    if $T$ satisfies $\mathcal{C}$ then
      RETURN $T$;
    else
      $v$ = SelectHeuristic($\mathcal{C}$, $T$);
      T = T with the value of $v$ flipped;
    end if
  end for
end for
RETURN  false;
```

SelectHeuristic($\mathcal{C}$, T). In the following section, we will introduce some well-know heuristics for *SelectHeuristic($\mathcal{C}$, T).*

2.5.1 GSAT

Selman et. al. proposed a new local search algorithm named GSAT [39] to solve SAT problems. It selects a variable for flipping such that the number of unsatisfied clauses will be minimized in each iteration. See Algorithm 4. GSAT is efficient for some relatively large instances which is hard to be solved by complete search methods in experiments. For example, in 1992, on a PC GSAT could solve randomly generated satisfiable 500-variable 3-SAT instances that DP cannot solve; GSAT could solve some problems with 140 variables within 14 seconds while DP needs 4.7 hours under the same instances[39].

Algorithm 4 GSAT–SelectHeuristic($\mathcal{C}$, T)

```
$L = \phi$   /*$L$ is a variable list*/
Add variables in $T$ that minimize in the number of unsatisfied clauses into $L$;
$v$ = Randomly pick up one from $L$;
RETURN $v$;
```

Later, Gent et. al.[17, 18] used "buckets" to select variables to flip in their new version of a GSAT solver, which is published at www.satlib.org[32]. In this new version, GSAT does not select the variables that minimize the number of unsatisfied clauses. Instead, it categorizes all variables into three categories(buckets): 1) the category with value 1 containing all variables that will reduce the number of unsatisfied variables after flipping;

2) the category with value 0 containing all variables that will not change the number of unsatisfied variables after flipping; 3) the category with value -1 containing all variables that will increase the number of unsatisfied clauses. Each iteration, GSAT selects one variable from the bucket with highest value to flip. See Algorithm 5. In the user manual of the new GSAT version Selman et. al. mentioned that this improvement "leads to about a 20 fold speedup for some instances with very large numbers of variables (10,000 or up)" when using other auxiliary techniques such as "random walk" and "tabu window". "Tabu window" is a variable list recording the recently visited variables which will usually be avoided by the search process.

We will name the former local search procedure *basic GSAT* and name the latter *CSAT*.

Algorithm 5 CSAT–SelectHeuristic($\mathcal{C}$, T)

```
$L_1 = \{x_i|$ flipping $x_i$ will decrease the number of unsatisfied clauses in $f\}$
$L_0 = \{x_i|$ flipping $x_i$ will not change the number of unsatisfied clauses in $f\}$
$L_{-1} = \{x_i|$ flipping $x_i$ will increase the number of unsatisfied clauses in $f\}$
if $L_1 \neq \phi$ then
  $v$ = Randomly pick up one from $L_1$;
  RETURN $v$;
end if
if $L_0 \neq \phi$ then
  $v$ = Randomly pick up one from $L_0$;
  RETURN $v$;
end if
if $L_{-1} \neq \phi$ then
  $v$ = Randomly pick up one from $L_{-1}$;
  RETURN $v$;
end if
```

2.5.2 WalkSAT

WalkSAT selects a variable by a two-step procedure. In the first step, it randomly selects an unsatisfied clause; in the second step, it selects a variable in this selected clause using a heuristic method. Different heuristics provide variations of WalkSAT. Algorithm 6 is a framework for the WalkSAT family. If a variable is randomly selected in the second step, the algorithm is called Basic WalkSAT[27].

We will introduce three well-know variations–Novelty, R-Novelty and Tabu in this section. They are based on various heuristics on how to select a variable from the selected clause. These algorithms were proposed in McAllester et. al[27].

Algorithm 6 Framework for WalkSAT–SelectHeuristic Procedure

```
Framework for WalkSAT–SelectHeuristic($\mathcal{C}$, $T$)
$C$ = Randomly pick up a clause from $\mathcal{C}$;{The First Step}
$v$ = Select a variable from $C$;{The Second Step}
RETURN $v$;
```

Novelty

Novelty selects a variable in the selected clause that will minimize the number of unsatisfiable clauses. However, if the variable selected is the variable that was flipped most recently, Novelty selects the second best variable with probability p, which is a parameter; with probability $1 - p$ we still choose the best variable. The probability of falling into local minima is reduced in this way because the probability of flipping the same variable twice has been reduced. The intuition behind Novelty is to avoid flipping the same variable back and forth.

R-Novelty

R-Novelty is a variation of Novelty. It is different from Novelty only when the best variable in the selected clause is the the most recently flipped variable. R-Novelty uses an objective function to assign a value to the best and second-best variables. This objective function should influence the choice between the best and the second best variables. In the source code of R-Novelty solver from Selman et. al., this objective function returns the change in the number of satisfied clauses when flipping a variable. This value is denoted as $w(w \geq 1)$. A variable is selected in the selected clauses according to the following four cases, where p is a preset parameter for the solver[27]:

1. If $p < 0.5$ and $w > 1$, select the best variable.
2. If $p < 0.5$ and $w = 1$, with a probability of $2p$, select the second-best variable; otherwise, select the best one.
3. If $p \geq 0.5$ and $w = 1$, select the second-best variable.
4. If $p \geq 0.5$ and $w > 1$, with a probability $2(p - 0.5)$ pick the second-best variable, otherwise select the best one.

We can see that a deterministic loop may happen. So R-Novelty will randomly select a variable in the clause to flip in every 100 iterations to break deterministic loops.

Tabu

Tabu means the basic WalkSAT + tabu technique in this paper. A Tabu records the variables flipped in the last m iterations. It chooses the best of variables in unsatisfied clauses not in the last tabu record. If all of the variables in all of the unsatisfied clauses are in the recorded iterations, Tabu simply ignores the recorder.

2.5.3 Smoothed Descent and Flood(SDF)

Dale Schuurmans and Finnegan Southey[34] proposed a new algorithm named SDF(Smoothed Descent and Flood) in 2000 after studying the facts that affect the performance of local search algorithms. As we mentioned before, local search algorithms select variables according to their evaluation function. The WalkSAT and GSAT families' evaluation functions count the number of unsatisfied clauses. SDF instead attempts to maximize the number of satisfied clauses and tries to increase the number of satisfied clauses that are satisfied by more literals. For this purpose it uses an objective function to count the number of variables satisfying each clause. Under an assignment t for a SAT instance containing m clauses it gives the weight

$$W(t) = \sum_{c} f(number\ of\ x_i\ that\ satisfies\ the\ clause\ c)$$

where

$$f(l) = \sum_{i=1}^{l} m^{k-i}$$

for $l > 0$, in which k is the number of the variables in the clause, and

$$f(0) = 0$$

for $l = 0$. This function f favors an assignment that makes clauses containing more literals being true. Intuitively, SDF is more deterministic than the GSAT family and the WalkSAT family. If a SAT instance has m clauses, the GSAT family and the Novelty family have at most m different values to weight each variable in the search process. Under the same instance, SDF usually has many more values to weight each variable. So fewer ties will happen in the search process. Therefore, SDF selects from a smaller set of variables, which means that it has a stronger bias on the selection of variables.

The second strategy introduced is their re-weighting strategy to escape local maxima. SDF assigns a weight $w(c)$ to each clause c.

$$W(t) = \sum_c w(c) f(\textit{number of } x_i \textit{ in the clause } c)$$

is used as the new objective function considering the weights of clauses. Once the assignment is a local maximum SDF will use its re-weighting strategy to reassign $w(c)$ to escape the maxima. First, the re-weighting strategy multiplicatively re-weights the *unsatisfied clauses* and normalizes the weights such that the largest difference in $W(t)$ value will be less than or equal to an assigned value δ. By increasing the weight of the unsatisfied clauses, it creates a new greedy search direction. Re-weighting is not a new technique to escape maxima or minima, however, researchers used additive updates for re-weighting[36] before. By multiplicative re-weight, SDF can more swiftly change the weight of clauses such that it can escape the maxima more rapidly. For the details of the multiplicative re-weight procedure, please check Schuurmans and Southey's paper [35]. Secondly, use the re-weight function

$$w(c) = (1 - \rho)\, w_{satisfied} + \rho\, w(c)$$

to re-weight the satisfied clauses, where $w_{satisfied}$ is the mean of the weight of satisfied clauses and $0 < \rho < 1$ is a parameter. The re-weight procedure flattens the weights of the *satisfied clauses* by shrinking towards their common mean $w_{satisfied}$. In this way, it prevents the weights of the satisfied clauses being so small that the clauses are falsified gratuitously.

SDF performs much better than the WalkSAT and GSAT families according to the measurement of the number of flips, although the total running time may be larger. SDF provides a good new approach to avoid local minima and plateaus. Algorithm 7 is the pseudo-code of SDF[34].

Algorithm 7 SDF(f, ρ)

```
1: Flip variable $v_i$ which maximized the objective function
       $W(t) = \sum_c w(c) f(\textit{number of } x_i \textit{ in the clause } c;)$
2: If the current $t$ is a local maximum, call re-weight procedure $\textit{re-weight}(\delta, \rho)$;
```

2.5.4 Automated Discovery of Variable Selection Heuristics

The local search algorithms presented above have a uniform framework(Algorithm 3). The differences among them are the strategies on how to select a variable to flip. GSAT was developed in 1991, while Novelty was not proposed until 1997.

Some researchers began to work on automated discovery of variable selection heuristics. Alex Fukunaga proposed a possible method in 2002 [15]. He considers randomly selecting variables, random walk, tracing ages of variables(the number of flips since a variable was last flipped), and so on, as atomic operations for variable selections. An automated heuristic generator generates the probabilities of choosing each atomic operation. Each step, only one atomic operation will be used to select variables. In other words, the generator deploys several atomic operations but only one of them is used at each step. Which atomic operation will be used depends on its probability. This is equivalent to combining those well-know heuristic methods with different weights. By changing the weight or probability of each atomic operation, the automated heuristic generator can produce many combinations. After experiments on more than 1000 combinations, Fukunaga found that some combinations perform as well as some existing heuristics, such as GSAT and Novelty.

In this chapter, we presented some basic concepts and well-know algorithms in SAT research. Two major approaches for solving SAT problems are introduced in Section 2.3 and Section 2.5 respectively. We have also introduced their advantages and disadvantages briefly. In the following chapters, we will concentrate on these local search algorithms and their properties.

Chapter 3

Previous Work On Search Space And Traps in Search Spaces

Researchers have proposed many local search techniques for SAT to escape local minima. However, most of them concentrate on the performance of the algorithms and did not spend much effort on the structures of SAT problem and the search space of the algorithms. The structure of the problem and the structure of the algorithms' search space can tell us more about which mechanisms really make the algorithms work. With a clear understanding of the structure in problems and the search space, we can make better improvements for the algorithms. We will introduce some techniques to escape the local minima in Section 3.1, which were proposed in early 90's and are still widely used today. We will introduce the research on the structure of the local search algorithms in the other sections.

3.1 Strategies for Escaping Traps in Local Search Procedures

In 1991 Selman et. al. proposed a new algorithm named GSAT, which shares some important features of Simulated Annealing[23], another well-known local search algorithm based on simulating physical annealing process. Instead of using the heuristics with bias on a descending variable called "temperature", GSAT is based on the heuristics with bias on the number of satisfied clauses. However, Selman et. al find that it cannot perform well on highly structured problems, for example the encoding of the blocks-world planning problem[24]. Actually, the same problem has been found with many other randomized local search type procedures, such as simulated annealing(Johnson et. al. 1991)[23]. Selman and Kautz confirmed this problem in their paper(Selman and Kautz 1993)[36]. They proposed extensions to solve this in the same paper[36] as well.

At that time, people concentrated on the hardness of instances themselves. Through analyzing the hardness of the formulas, researchers provided lower bounds and upper bounds for CSAT and GSAT[21]. However, the theoretical results of the bounds usually come with some restrictions. For example, some bounds are related to SAT instances in k-CNF-d form, where d means that any variable in the k-CNF formula appears at most d times[21]. If there is no restriction, Hirsch[20] has proved that CSAT and GSAT need the expected time at least 2^n, where n is the number of variables in a formula. In other words, they might be no better than a trivial algorithm that checks all states(2^n states) in the assignment space. From the discussion in this paper and in Selman et. al. paper[39], we can see how some structures in the formula "mislead" the local search processes of CSAT and GSAT. A structure can be some clauses. These clauses may lead the search process to a trap sometimes, or to a longer path to solutions. Hence, for local search algorithms, structures contained in some problems might be considered as the reason why they are hard for local search procedures. These structures are not categorized or defined in a formal way. Because the SAT problem is an NP-Complete problem, nobody expects that these structures can be categorized into a finite number of categories such that efficient heuristic methods can be designed specially for each category. So people pursue general heuristics to escape from the local minima and large plateaus caused by those unknown structures. A plateau for a procedure consists of the assignments that are considered as good as one another according to the evaluation function of this procedure and each of these assignments has at least one neighbor that is also in this plateau. Two assignments are neighbors if one assignment can be changed to the other in one iteration. For hill-climbing greedy algorithms, a plateau consists of the assignments on the same level and each of them has at least one neighbor in this plateau. A huge plateau makes the local search process got stuck inside since all vertices inside are as "good" as each other. Selman and Kautz proposed three general strategies, called *Adding Weight*, *Averaging in Previous Near solutions* and *Random Walk* for GSAT, to escape traps and plateaus[36].

Adding Weight increases the weight of each unsatisfied clause by K at the end of each try, where K is a parameter. A try refers to the iterations between two restarts. In search processes, the weight for each variable is the sum of the weights of the unsatisfied clauses under the assignment after flipping this variable. The second strategy considers the assignment at the end of the previous try by using the bitwise average of the last assignment

T_i and the last assignment T_{i-1} in the try before the last try, where the bitwise average of two assignments is an assignment which keeps the values of the variables that are identical in both assignments and randomly assigns value to the remaining variables. T_0 and T_1 come from the first two tries without bitwise operations. These two strategies can be considered as restarting strategies. The third strategy, *Random Walk*, randomly picks up a variable in an unsatisfied clause and flips its value with probability p and follows the standard GSAT procedure with probability $1 - p$, where the probability p is a preset parameter.

In fact, these three strategies provide general techniques to escape local minima. But are they necessary for many other local search algorithms besides GSAT? Why does the random walk work? To answer these questions, we should have a clear understanding on the search mechanisms. We believe that the properties of the search space of local algorithms could help us understand these algorithms more deeply. After all, performance of algorithms is only the result of these mechanisms. Researchers have done some valuable work on the search spaces. We will introduce some of them in following sections.

3.2 Search Space Structures of Local Search Algorithms and SAT problem

To further understand the mechanisms that make local searches work, people may need to study the local search algorithms' search space directly as well as the structure of SAT problems. Search spaces are the space that a local search procedure really "walks" in. The properties of the search space can tell us more about the local search algorithms.

3.2.1 State Space

States(assignments) together with some relations construct a search space. Clark and Frank et. al.[4] proposed the correlation between the number of solutions and the hardness of SAT instances. This correlation is robust across problem classes as well as types of local search procedures, such as GSAT and GCSP. GCSP is a similar procedure to GSAT for CSP[4]. For local search algorithms, the hardness was found to reach the peak at thresholds. Researchers also found that although the number of solutions decreases beyond phase transition thresholds, the cost of local search procedures still decrease. Parkes[30] tried to explain the former phenomenon — why they are hard at the threshold through studying backbones. For satisfiable SAT instances, the backbone consists of variables forced to take a particular truth value in all solutions. A large backbone that appears at the threshold might affect the performance of local search algorithms because, on the threshold, it is difficult for local search algorithms to identify this backbone, which means that the algorithms spend too much time to flip back and forth. Also, Singer et. al.[40] tried to reveal why the hardness decays beyond the threshold by studying the backbone again. They found the large backbone in the region beyond the threshold is easier to find than that on the threshold, which means a hill-climbing algorithm might more easily identify the backbone, such that the values of the variables are more quickly determined.

The research mentioned in this section focuses on the states but does not consider the movements among the states. It does not provide the understanding of the dynamic information when the local search algorithms move in the search spaces. In the remaining sections, we will introduce how people study the movement of local search algorithms in state spaces. The movements and the states together can be considered as the search space of local algorithms. With further understanding of the dynamic information, researchers can know more details about local search mechanisms.

3.2.2 Measurements of Local Search Process Performance

Schuurmans and Southey proposed three measurements of local search algorithms' performance[34]. The first one is the *depth* of local search, which measures the number of unsatisfied clauses in the search process. The "depth" here is different from the depth of the complete search algorithms. Usually, the depth of complete search trees means how many variables are not instantiated. The depth of local search algorithms usually indicates how many unsatisfied clauses remain. If the depth is zero, the problem has been solved. So for the measurement of the depth, a small value is desired.

The second measurement is the *mobility*, which measures how rapidly a local search moves in the search space. They measure the mobility by counting the number of variables that are assigned different values in two assignments being k iterations apart in the search sequence. The number of the variables assigned different values is called the *Hamming distance*. We will give a formal definition for "Hamming distance" in Chapter 5. For complete search algorithms, we hope the search algorithms can verify more assignments in a period of time. For the same consideration, the mobility intuitively indicates how fast local search algorithms search through the space. So a large value of mobility is desired.

The last measurement is *coverage*, which measures how systematically a local search explores the entire search space. It is a measurement that considers the completeness of local searches. A local search algorithm that is more "complete" is expected to have a better performance so a high coverage rate is desired. The coverage is measured by estimating the size of the largest unexplored area in the search space and the speed of the reduction of this size. This unexplored size is estimated using the maximal Hamming distance between any checked assignment and its nearest unchecked assignment. The coverage rate is ($n -$ max size of uneplored area)/search steps.

Schuurmans and Southey empirically investigated the necessity of these three measurements. They show that poor performance under any one of these three measurement leads to poor problem solving performance, using a large number of experiments. Although whether good depth, mobility and coverage rate will ensure a good problem solving performance still needs more evidence, these three measurements provide a good understanding of local search performance. They may guide us to make better searching heuristics. In fact, Schuurmans and Southey built a new solver–SDF based on their theory. We have introduced SDF in Section 2.5.3. Schuurmans and Southey showed that SDF has the best performance measured by the number of flips and the best performance under any of those three measurements as well. It empirically confirms the necessity of the measurements by measuring the number of flips, although the performance of SDF measured by the actual running time sometimes is even worse than the other well-known local search procedures, such as Novelty, Novelty+ and DLM. Novelty+ is Novelty with the random walk technique. The SDF's running time problem reminds us that if we pursue good performance under all those three measurements, the program may spend too much time for the selection of "better" variables for flipping and "better" assignments for restarting. Searching smarter may cost more time than simpler searching heuristics do. We noticed that Schuurmans and Southey empirically studied the search space of local search algorithms by simulating the search process and sampling. By tracking how the local search processes visit assignments in the whole assignment space, they investigated the properties of those local search algorithms' search spaces as well.

3.2.3 Local Search Topology

Frank et. al proposed an assignment sampling method for studying the assignment spaces[14]. They used GSAT to locate an assignment at the level that they need at first. Two assignments are at the same level if they satisfy the same number of clauses. Then, using this assignment as the starting point, they explore the assignment space with Breath-First search and collect those assignments at the same level where the starting point was located.

Frank et. al. categorized an assignment space into several categories–plateau, bench, minima, contour. In the assignment space, two assignments are connected if their hamming distance is 1. A *plateau* is a maximal connected region of an assignment space, in which all assignments are at the same level. The *level* of this plateau is the number of unsatisfied clauses under the assignments in this plateau. If a plateau P_1 has assignments adjacent to another plateau P_2 at lower level, plateau P_1 is a *bench* and, in P_1, those assignments adjacent to P_2 are *exits* of plateau P_1. If a plateau contains no exit, this plateau is called a *minimum*. If all assignments in a plateau are exits, this plateau is a *contour*. Frank et. al. investigated the proportion of minima, the size of the minima, the proportion of benches, the proportion of exits to the bench size and the distribution of the size of benches and minima in both satisfiable and unsatisfiable problems. They also investigated these structures under various subclasses of 3-SAT problems, such as Random 3-SAT problems and Cluster 3-SAT problems. They found "conclusive evidence of the existence of local minima in assignment spaces, and show that they become more prevalent as the number of unsatisfied clauses becomes close to 0."[14] Their work provides a good basis for how to schedule restarting and random walk in local search procedure or the determination of the size of tabu list. On the other hand, their work also gives a way to identify the parts that are worth investigating.

The sampling method Frank et. al. used is good for even larger size instances, since the minima and benches themselves are relatively small compared with the entire search space. In this paper they used instances with 100 variables whose assignment spaces contains 2^{100} assignments. By GSAT and Breath-First search, they can investigate the plateaus at any levels. However, the algorithms like WalkSAT, Novely and R-Novelty may not benefit much from this method because these algorithms do not follow the gradient strictly. Also, Frank et. al. may need to study the assignment space with various classes of SAT problems besides 3-SAT, such as encoded Graph Coloring instances, encoded Latin Square instances and planning problems.

All of these works indicate that studying the structure of search spaces might be a prospective direction to further understand how algorithms work. Currently, little work has been done on the search space of a particular local search algorithm. The search space of each of Novelty/Novelty+, R-Novelty/R-Novelty+, Basic WalkSAT, GSAT and CSAT should have its own special properties. We will compare the search spaces of these local search algorithms in Chapter 5. To study the search space of local search algorithms, we also used brute-force search on small size search spaces in this thesis. We hope that we can reveal some basic properties of local search algorithms' search spaces such that we can know their search mechanisms better. We hope our work could help to make a clearer understanding of the local search mechanisms.

Chapter 4

Performance Comparisons

We have introduced some local search algorithms in previous sections. To show a more concrete image of these algorithms we will discuss their performances on various problems in this chapter. In Section 4.1 we will show some comparisons between local search algorithms and a systematic search algorithm, SATO[41], which is based on unit-resolution. Since what we are concentrating on is the local search algorithms and what we intend to show is an outline on the differences of the performance between local search algorithms and systematic search algorithms, we use only one complete search algorithm, SATO, in this thesis. The performance comparisons between local search algorithms and systematic search algorithms are measured by actual running time. Although the performance measured by running time will vary in various computation environments, it can still give us a rough view of the differences among these algorithms. In Section 4.2, we will present the comparisons among local search algorithms. The performances in this section are measured by the number of flips. A flip is to change the value of a variable from 0 to 1 or from 1 to 0. If a local search algorithm can not successfully solve an instance in limited number of restarts, we will consider this instance is unsolvable for the local search algorithm, although theoretically all satisfiable instances can be solved eventually if we allow an unlimited number of restarts. Given the fact that NP-Complete problems cannot be solved efficiently or,say, not all instances in NP-Complete class can be solved in polynomial time if $NP \neq P$, we do not expect that all instances can be solved in reasonable time by systematic search algorithms or local search algorithms. Also, how many flips should be executed in each try and how to set randomness parameters in search procedures are still big issues. For example, in Novelty, the optimal probability of flipping the second best variable in a selected clause can only be determined by experiments and experience. With different classes of instances the optimal value for the parameters may be different. But this is not what we are concerned with. Instead, we just want to give an outline on how good they are. So we used only the default setting or the recommended setting. All solvers come from SATLIB.ORG[32].

4.1 Comparisons of Local Search and Systematic Search Algorithms

In this section, we present the comparisons among local search algorithms and a systematic search algorithm, SATO. SATO is an efficient implementation of the Davis-Putnam method developed by Hantao Zhang[41], University of Iowa. Note that local search algorithms may not solve all instances in a limited number of ties. Considering that, in this scenario, the performance on unsolved instances completely depends on the number of tries and number of flips in each try, we only measure the performance with medians of the total numbers of flips used. Therefore, if the number of solved instances is less than 50% of the number of all instances, the corresponding point will be absent in our figures. Hence, we provided the success rate of solving instances as another measurement of performance as well.

4.1.1 Local Search and Systematic Search with Randomly Generated Instances

In this section, we test Novelty, Novelty+, R-Novelty, R-Novelty+ and SATO's performances measured by actual running time on the instances of Uniform Random-3-SAT, which are downloaded from SATLIB.ORG[32]. Uniform Random-3-SAT is a subclass of SAT problems, which are 3-CNF formulas randomly generated in the following way[32]: given the requirement for n variables and m clauses, 1) produce the m clauses, each of which draws 3 literals randomly from the all $2n$ possible literals(n variables and their n negations) and each literal is selected with the same probability of $1/2n$; 2)reject clauses containing multiple copies of the same literal or those being tautological. By adjusting n and m, we may generate instances with various ratios and number of variables. The instances tested in this section are instances with the C/V ratio being around 4.30 which is the threshold of Uniform Random-3-SAT. Since local search algorithms can only solve satisfiable instances, only the "uf" series at SATLIB.ORG[32] are used, which are proved to be satisfiable instances by some complete search solvers("uuf" series at this site are proved to be unsatisfiable instances).

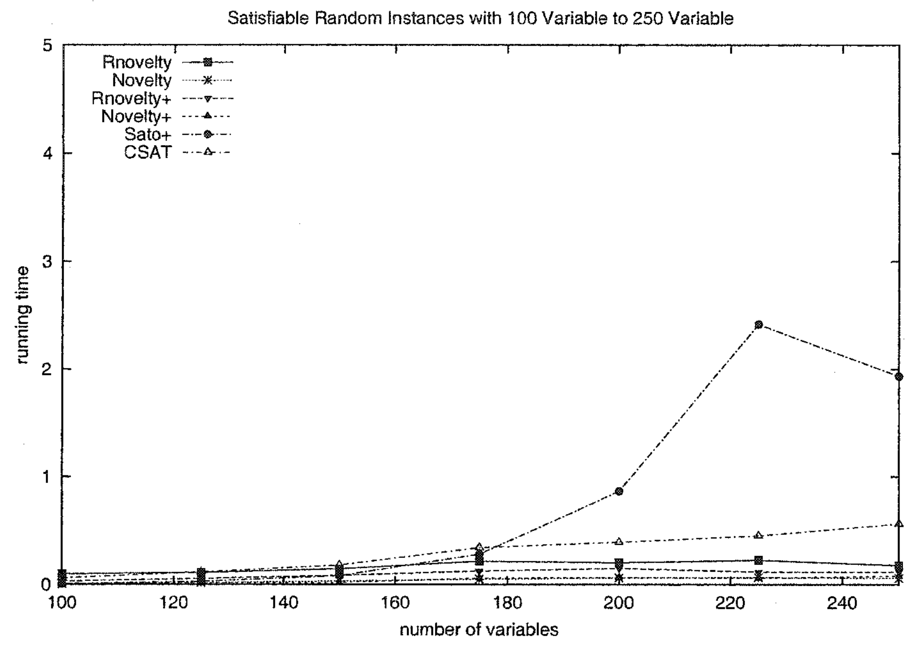

Figure 4.1: Performance of Systematic Search and Local Search On Uniform Random 3-SAT Satisfiable Instances

In the figure 4.1, the Y axis represents the median of running time under these SAT instances. The X axis represents the number of variables in formulas. The ratio of these instances are fixed at 4.30, except that the ratio of the instances with 250 variables is 4.26 that is less that the others. Let's have a look at their performances in Figure 4.1

It is obvious that local search algorithms out-perform SATO in this test. We may notice that the local search algorithms' running times increase quite slowly, compared with SATO. SATO's performance curve has a relatively sharper uphill after the point 175 on the X axis. Their running times drop at 250 on the X axis. This might be caused by the C/V ratio at this point(4.26) being slightly smaller than the C/V ratio of the others(4.3).

4.1.2 Local Search and Systematic Search with Instances Encoded from Other Problems

The Graph Coloring problem is a well-known NP complete problem: Given a graph $G = (V, E)$ and a color set C, in which $V = \{v_1, v_2, \ldots, v_n\}$ is the set of vertices, E is the set of edges and $C = \{c_1, c_2, \ldots, c_k\}$ is a set of k colors, find a coloring $f : V \rightarrow C$ such that a pair of vertices that are connected by the same edge cannot share the same color. The decision variant of the coloring problem is to decide whether there exists a coloring for a particular number of colors. The answer is "Yes" or "No". The optimization variant of the coloring problem is to find a coloring with a minimum number of colors. An optimization problem can be solved using series of decision problem inquires, so people usually focus on the decision problem[32].

Given a graph $G = (V, E)$, we encode the k-colorable decision problem in this way[32]: 1) $\forall v_i \in V$, v_i is represented by the k variables $x_{i_1}, x_{i_2}, \ldots, x_{i_k}$, where x_{i_l} being *true* represents that v_i is colored with the l-th color; $\forall e_h = (v_i, v_j) \in E$, these k clauses are added – $\{\neg x_{i,h}, \neg x_{j,h}\}$, where $h \in C$, and these clauses guarantee that no two adjacent vertices will be colored with the same color; 2) to guarantee that every vertex will be colored, for $v_i \in V, i = 1, 2, \ldots, n$, the clause, $\{x_{i,c_1}, x_{i,c_2}, \cdots, x_{i,c_n}\}$ is added into the formula; 3) to guarantee that each vertex will be colored only once, for $\forall v_i \in V$, these $\binom{k}{2} = k \times (k-1)/2$ clauses are added – $\{\{\neg x_{i,c_p}, \neg x_{i,c_q}\} : 1 \leqslant p < q \leqslant k\}$. Therefore, if an encoded SAT instance is satisfiable, the original graph is k-colorable as the encoding process guarantees that each vertex is colored once and only once and no adjacent vertices are colored with the same color.

For example, given a graph $G = (V, E)$ and $C = \{c_1, c_2, c_3\}$, where $V = \{v_1, v_2, v_3\}$ and $E = \{(v_1, v_2), (v_1, v_3), (v_2, v_3)\}$, we will have the set of variables $\{v_{11}, v_{12}, v_{13}, v_{21}, v_{22}, v_{23}, v_{31}, v_{32}, v_{33}\}$, where v_{ij} represents the color c_j for the vertex v_i, the set of clauses $\{\neg v_{11}, \neg v_{21}\}, \{\neg v_{12}, \neg v_{22}\}, \{\neg v_{13}, \neg v_{23}\}, \{\neg v_{11}, \neg v_{31}\}, \{\neg v_{12}, \neg v_{32}\}, \{\neg v_{13}, \neg v_{33}\}, \{\neg v_{21}, \neg v_{31}\}, \{\neg v_{22}, \neg v_{32}\}, \{\neg v_{23}, \neg v_{33}\}$, ensuring that a pair of vertices at the ends of the same edge can not share the same color, the set of clauses, $\{v_{11}, v_{12}, v_{13}\}, \{v_{21}, v_{22}, v_{23}\}, \{v_{31}, v_{32}, v_{33}\}$, ensuring that each vertex will be colored, and the set of clauses, $\{\neg v_{11}, \neg v_{12}\}, \{\neg v_{11}, \neg v_{13}\}, \{\neg v_{12}, \neg v_{13}\}, \{\neg v_{21}, \neg v_{22}\}, \{\neg v_{21}, \neg v_{23}\}, \{\neg v_{22}, \neg v_{23}\}, \{\neg v_{31}, \neg v_{32}\}, \{\neg v_{31}, \neg v_{32}\}, \{\neg v_{32}, \neg v_{33}\}$.

Let's have a look at Figure 4.2. We used 3-colorable flat[13] graphs in this comparison, which are downloaded from SATLIB.ORG[32]. Since 3-colorable flat graph encoded SAT instances usually are hard, we increase the number of flips in one try quadratically with the increasing of the number of variables. That is, we flip n^2 times in each try, where n is the number of variables. Unlike the performances on Uniform Random-3-SAT instances, the best local search algorithms, Novelty and Novelty+, and SATO's performance are very close to one another when the number of vertices is smaller than 400. After that, the runtime of SATO has a big jump. It took about a month for SATO to solve all of the instances

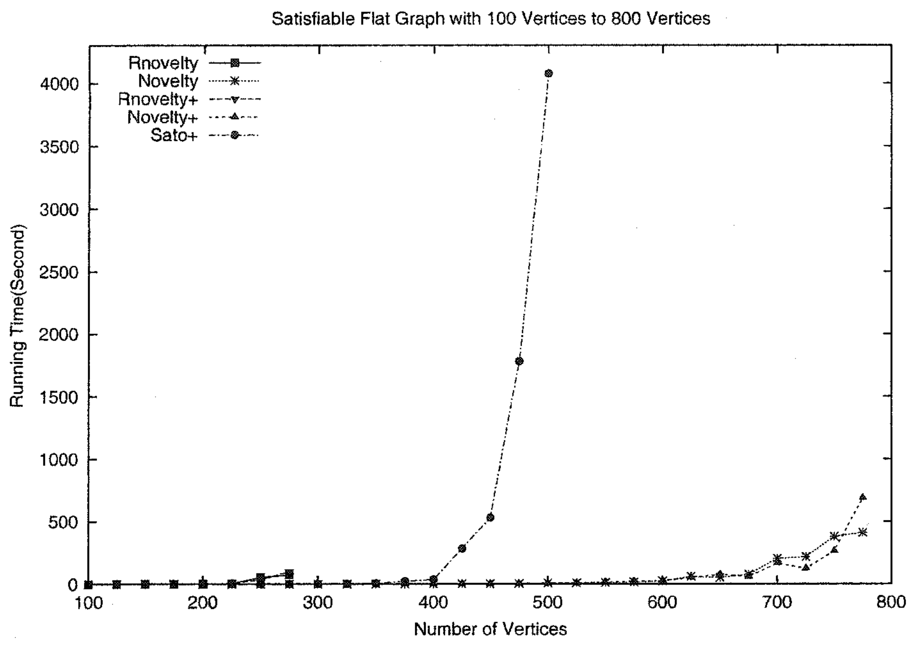

Figure 4.2: Performance of Systematic Search and Local Search On Flat Graphs

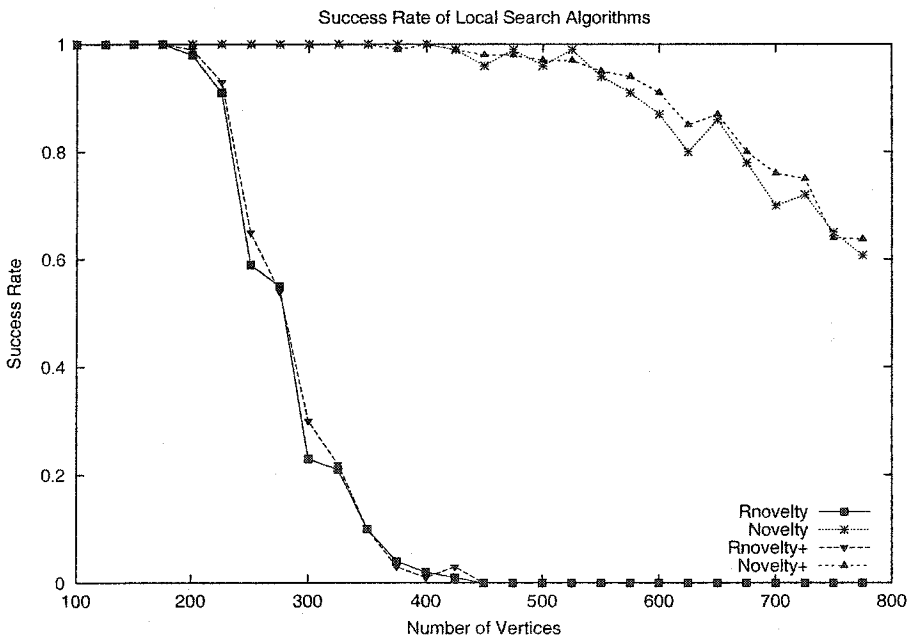

Figure 4.3: Success Rate of Local Searches On Flat Graphs

with 425 to 500 vertices. R-Novelty and R-Novelty+ stop working when the number of vertices is larger than 275. Figure 4.3 shows the success rate of R-Novelty/R-Novelty+ and Novelty/Novelty+. We do not show CSAT and Basic WalkSAT because they both stop working on very small instances.

When the number of vertices is not large, say less than 400 vertices, on these instances we cannot tell whether local search algorithms definitely out-perform SATO. SATO performs well under these instances because a complete search may take advantage of some structures inside, such as symmetry, to do further pruning irrelevant assignments. When the size of instances(the number of vertices) is large, say, bigger than 425, the unit propagation does not work either. Given the fact that local search algorithms cannot prove a formula being satisfiable or not, a complete search solver might be a better choice when the instances are relatively small.

4.2 Comparisons of Local Search Algorithms

In this section, we compare the performances of Novelty, Novelty+, R-Novelty, R-Novelty+, CSAT and Basic WalkSAT. The performances are measured by the number of flips to solve an instance. Again, we use the medians of the numbers of flips as the points in the figures. Besides the performance graphs, we plot the success rate graphs as well. If the success rate is less than 50%, the corresponding point will not be drawn on the figures. CSAT and Basic WalkSAT stop working on small instances again. In this case the figures in this experiment will not include them.

4.2.1 Local Search Algorithms on Uniform Random 3-SAT Instances

Figure 4.4 compares the selected local search algorithms' performances on Uniform Random 3-SAT satisfiable instances used in last section. We may notice that the performances of Novelty and Novelty+ are very close to each other. CSAT's performance is worse than that of Novelty, Novelty+, R-Novelty and R-Novelty+. Basic WalkSAT can only solve 26% of the instances with 100 variables and then the success rate drops to 0%. So we do not include Basic WalkSAT in the performance figure.

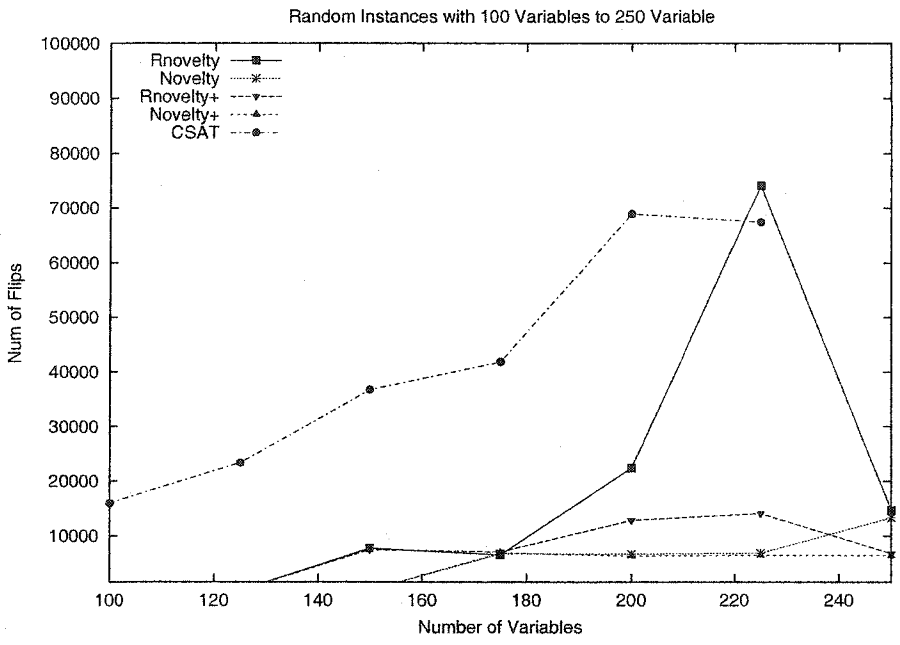

Figure 4.4: Performance of Local Search Algorithms On Uniform Random 3-SAT Satisfiable Instances Measured by Number of Flips

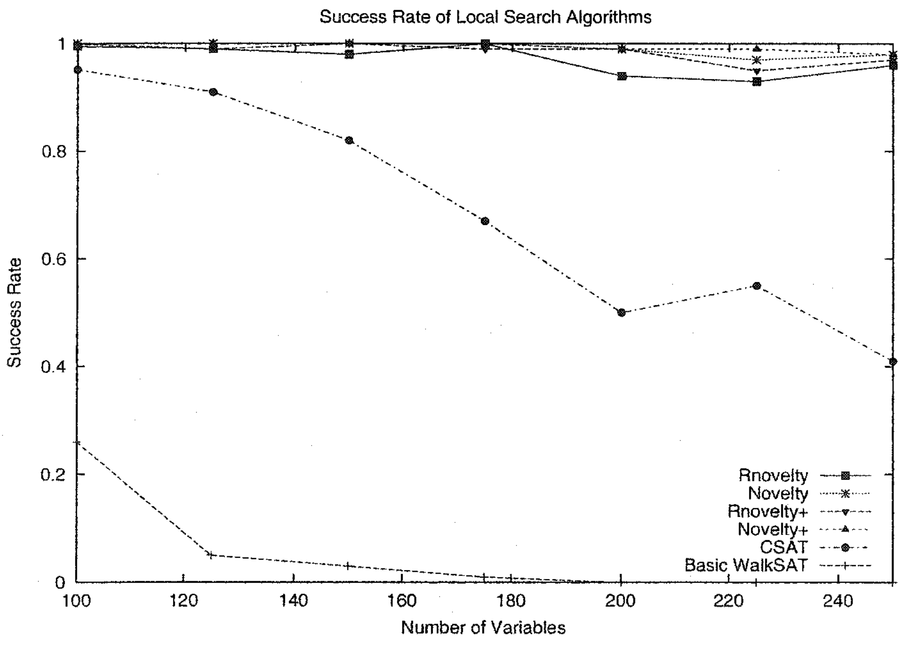

Figure 4.5: Success Rate On Uniform Random 3-SAT Satisfiable Instances

4.2.2 Local Search with FLAT Graph Coloring Encoded Instances

We use exactly the same instances used in Section 4.1. CSAT and Basic WalkSAT can hardly do anything with this group of instances. In Figure 4.7, R-Novelty's and R-Novelty+ success rate drops to less than 40% soon after 275 on X axis, while Novelty/Novelty+ can still solve about 60% of the instances even when there are 775 vertices in instances.

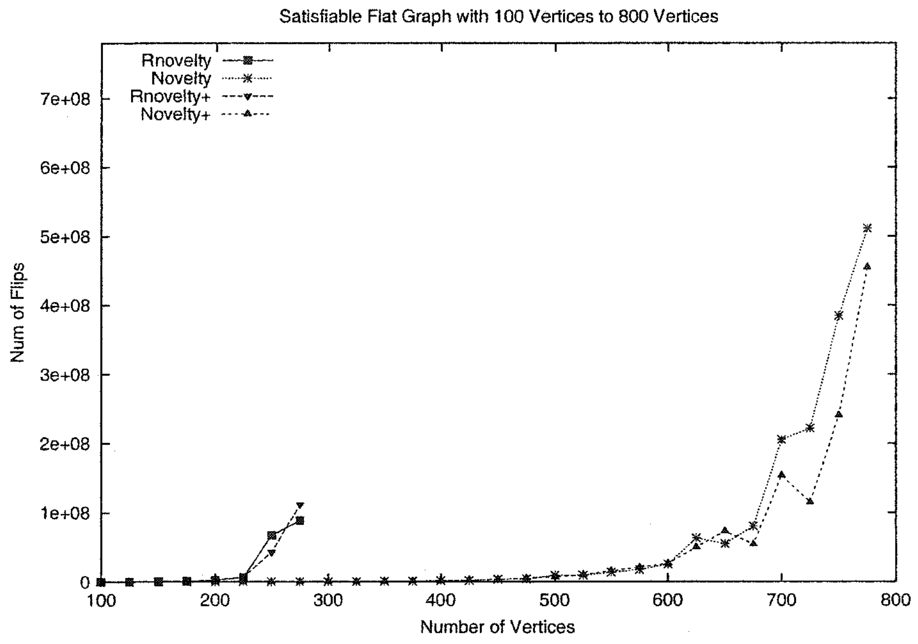

Figure 4.6: Performance of Local Search Algorithms On Flat Graph with 100 to 800 vertices

In Section 4.1 and Section 4.2 we discussed the performances of local search algorithms and a systematic search algorithm–SATO. The sizes of instance vary. Basic WalkSAT and CSAT do not work well in our experiments. In the experiments with FLAT graph instances, Novelty and Novelty+ out-perform R-Novelty and R-Novelty+. R-Novelty and R-Novelty+ are Novelty and Novelty+ respectively with a different selection bias on the best and second best variable in a selected clause. We can tell the heuristics for local search is sensitive to such variation.

4.3 The Correlation of Hardness with the C/V Ratio

In previous sections, we used instances with various number of variables. In this section, we will show the effect of the C/V ratio on the performances of local search algorithms. That is, when the number of variables is fixed, how does the C/V ratio affect the performance?

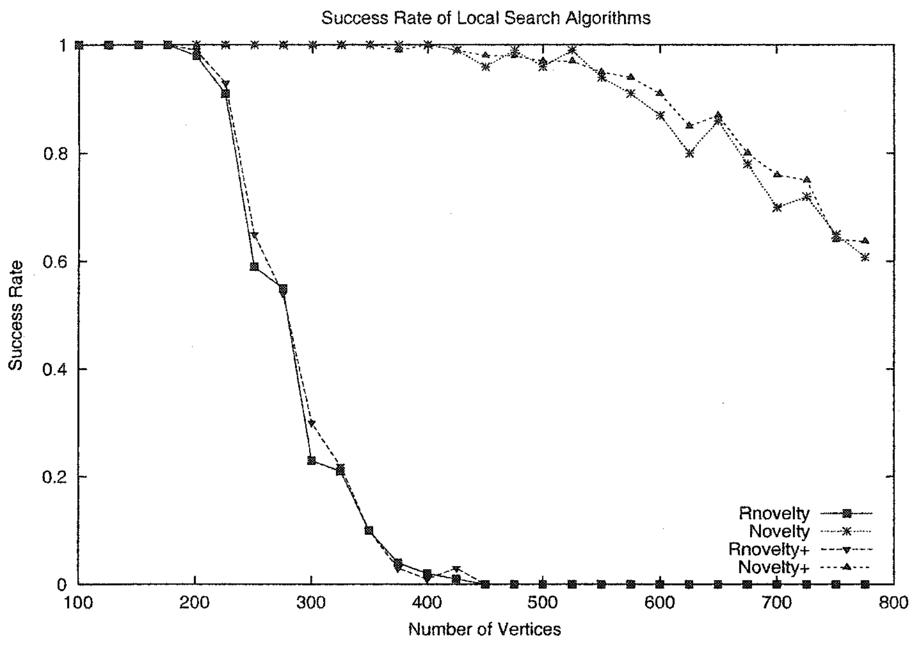

Figure 4.7: Success Rate On Flat Graphs with 100 to 800 vertices (Same as Figure 4.3)

4.3.1 Random Hidden Solution 3-SAT Instances

Random hidden solution 3-SAT instances are used in this section. We generate a random hidden solution instance in this way: 1) Randomly generate truth assignments to each variable first. 2) Generate clauses by randomly picking up 3 literals in all possible $2n$ literals(n variables and their negations) but only keep the clauses that are satisfied by the truth assignment generated in 1). 3) Repeat 2) until we have the number of clauses desired. In this way, it is guaranteed that there exists at least one solution for the generated instances.

Figure 4.8 shows us the experimental results. The X axis represents the C/V ratio value and the Y axis represents the number of flips. It indicates that CSAT, Novelty, Novelty+, R-Novelty and R-Novelty+ show a weak easy-hard-easy pattern on these satisfiable instances. Basic WalkSAT cannot solve instances with a high C/V ratio. Let's have a look at the success rate of Basic WalkSAT. In Figure 4.9, in which the Y axis represents the success rate, we find that the success rate of Basic WalkSAT has a sharp drop starting at 3.5 on the X axis. After that, the success rate of Basic WalkSAT is 0. On the other hand, all of the other four local search algorithms–Novelty, Novelty+, R-Novelty and R-Novelty+–can solve all of the hidden solution 3-SAT instances so their success rates are always 100%.

CSAT shows a much better performance than Basic WalkSAT's. Actually, on random hidden solution 3-SAT instances, CSAT's performance is quite close to Novelty/Novely+'s and R-Novelty/R-Novelty+'s performances.

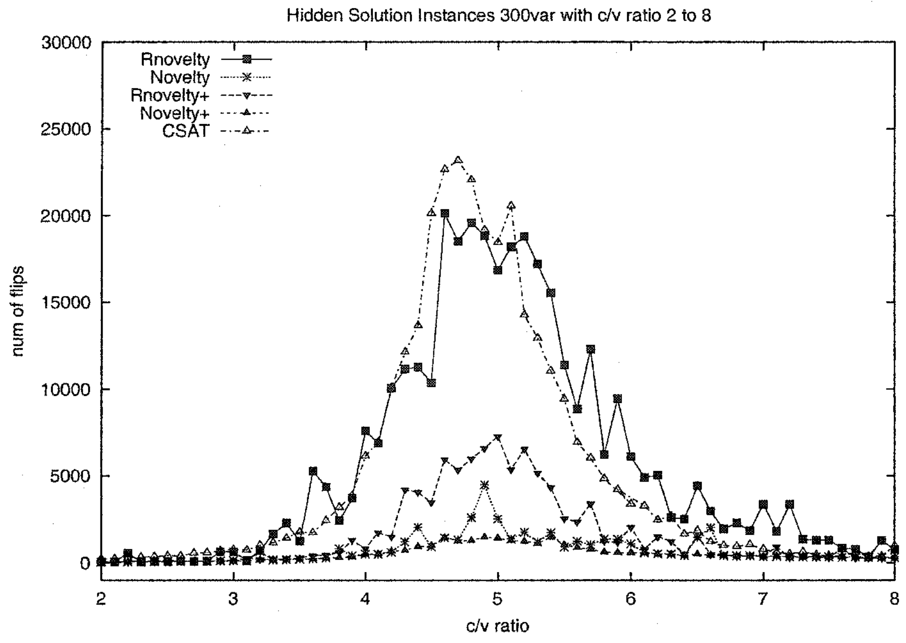

Figure 4.8: Hidden Solution 3-SAT Instances

4.3.2 Encoded Hidden Solution 3-Colorable Graph Instances

In this section, we use SAT instances encoded from random hidden solution 3-colorable graph instances instead since Flat can only be solved well by Novelty/Novelty+ in previous experiments. The hidden solution 3-colorable graph instances are generated in this way: 1) Use hidden solution graph generater provided by Dr. Culberson[8] to generate the graphs, whose probability of existence of an edge vary from 0.01 to 0.08. The hidden solution here means that we generate a coloring and a set of vertices first, then generate edges with a probability p(the probability of existence of this edge) and only keep the edges that do not connect two vertices colored the same color. 2) Use an encoding tool[8] to encode the hidden solution graphs into SAT instances in the way introduced in the previous section.

Figure 4.10 shows the local search algorithms performances. It is measured by the number of flips. The X axis represents the probability of existence of edges and the Y axis represents the number of flips. Again, Basic WalkSAT does not work well on these

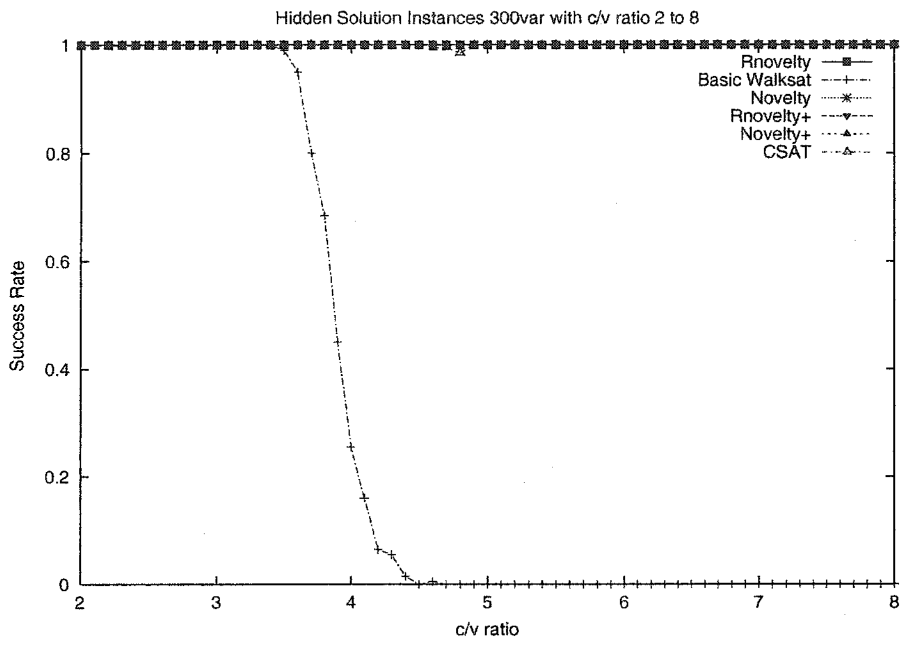

Figure 4.9: Successful Solving Rate on Hidden Solution 3-SAT Instances

instances, and Novelty/Novelty+ and R-Novelty/R-Novelty+ follow the easy-hard-easy pattern. Basically, CSAT still follows the easy-hard-easy pattern. But its performance is worse than Novelty/Novelty+ and R-Novelty/R-Novelty+. In the Figure 4.12, where the Y axis represents the success rate of the algorithms, we find that CSAT's success rate shows a "U" shape curve. In the interval, which is the hard region for other algorithms, CSAT's success rate is close to zero. That is, this region is too hard for CSAT. Basic WalkSAT's success rate drops suddenly at the point close to the point where CSAT drops. But Basic WalkSAT's success rate never rises again. It seems that it might not really tell where are easy regions.

To give a picture of how the systematic search algorithms perform on the hidden solution 3-colorable graph instances, we compared the local search algorithms' and SATO's performances in Figure 4.11. As mentioned in previous sections, SATO's performance is close to or better than the local search algorithms' when the instances are relatively small. As there are only 200 vertices in these instances, we can see that the performance of SATO is close to some of the local search algorithms.

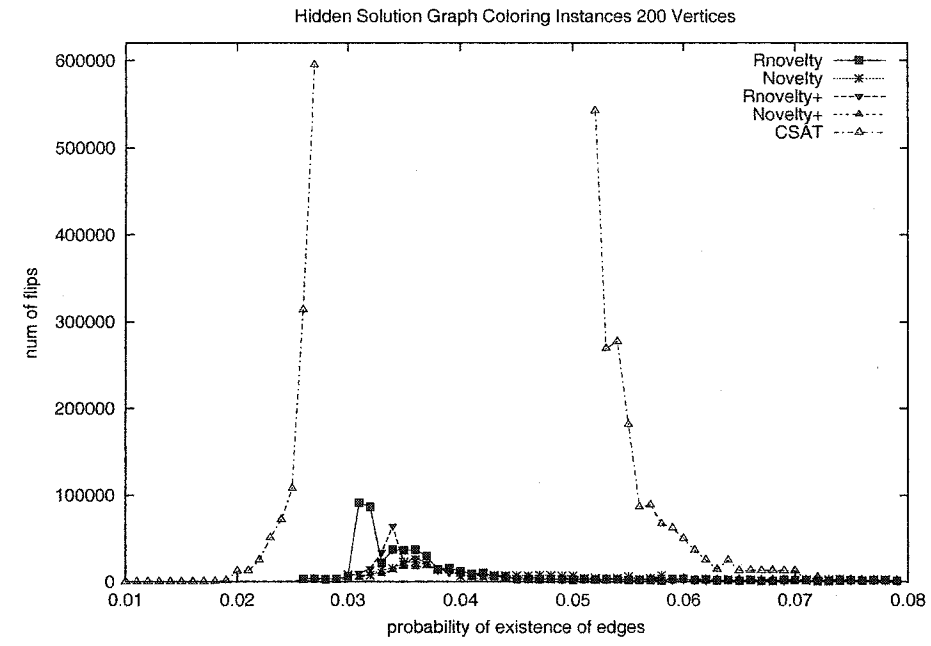

Figure 4.10: Performance for Hidden Solution 3SAT Instances Measured by Number of Flips

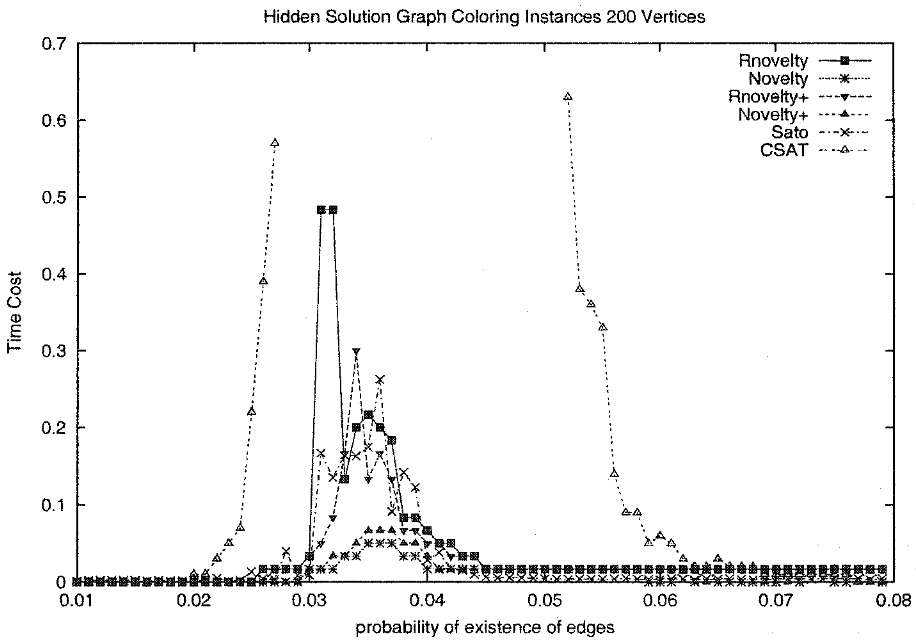

Figure 4.11: Performance for Hidden Solution 3-SAT Instances Measured by Running Time

4.4 Conclusion

In this chapter, we compared the performance of some local search algorithms and SATO, a complete search algorithm. We have seen that local search procedures perform quite well, especially Novelty/Novelty+. Under the relatively small instances, SATO can perform as well as local search algorithms. But under large instances, SATO performs much worse than Novelty/Novelty+ in our experiments. In Section 4.3, we find that local search algorithms also follow the easy-hard-easy pattern when the C/V ratio increases. Basic WalkSAT is weak under all types of instances used in this chapter. It stops working even under small size instances with the high C/V ratio which are easy for the other local search algorithms.

The performance evaluation of local search algorithms is a complicated issue. We still have a lot to do on this. But we have provided an outline of the differences of these algorithms' performance. We note that, generally, Novelty/Novelty+ is better than the others, RNovelty is better than CSAT, CSAT is better than GSAT and Basic WalkSAT and WalkSAT is the worst one, under the default and recommended settings in our experiments.

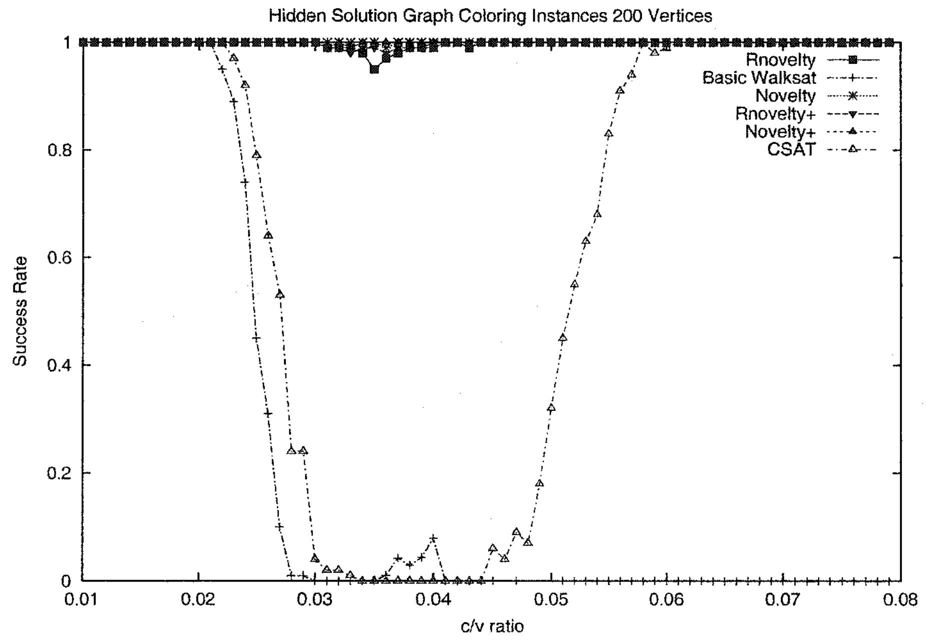

Figure 4.12: Performance for Hidden Solution Graph instances Measured by Number of Flips

Chapter 5

Search Space Graph

5.1 Definitions

For the convenience of the future discussion, we are going to give some definitions in this section:

Definition 5.1.1 *Hamming Distance* is $H(A,B) = \sum_{k=1}^{n} h(a_k, b_k)$, where $A = (a_1, a_2, \ldots, a_n)$ and $B = (b_1, b_2, \ldots, b_n)$ are two vectors. The function $h(a_k, b_k) = 1$, when $a_k \neq b_k$; otherwise, $h(a_k, b_k) = 0$. It measures the differences between two vectors.

Definition 5.1.2 ***Directed Hyper Cube*(DHC)** with n variables is a directed graph $DHC_n = (V, E)$, where $V = \{v : v = (x_1, x_2, \ldots, x_n), x_i \in \{0,1\}, i = 1, 2, \ldots, n\}$, i.e. all binary vectors of length n, is the set of all the vertices in G, and $E = \{(v_i, v_j)$ where $v_i, v_j \in V$ and $H(v_i, v_j) = 1\}$ is the set of all the edges in G.

Suppose the algorithm A is a local search algorithm for SAT. Given a formula f and an assignment t, if there is a possibility of t being transferred to another assignment t' after one or multiple variables are flipped within one step by A, we denote it as $A(f,t) \models t'$. Usually, given an assignment t, a certain local search algorithm A and a formula f, there may exist different t'. Hence, $A(f,t) \models$ is a one-to-multiple relation.

Definition 5.1.3 $SP_n(A, f) = (V, E)$ is a directed graph, where A is a local search algorithm and f is a formula on n variables, in which $V = \{v : v = (x_1, x_2, \ldots, x_n), x_i \in \{0,1\}, i = 1, 2, \ldots, n\}$ is the vertices set and $E = \{e : e = (v_i, v_j), v_i, v_j \in V$ s.t. $A(f, v_i) \models v_j\}$ is the edge set. $SP_n(A, f)$ is called the *Search Space Graph*(SSG) for the local search algorithm A under the formula f on n variables.

Note that the real search space of a local search algorithm A is not always a unit weighted directed graph as it defined in the Definition 5.1.3. For example, given a local search algorithm A, a formula f and an assignment t, if both $A(f,t) \models t'_1$ and $A(f,t) \models t'_2$ exist, the probability of either occurring may not be the same. If we say $A(f,t) \models t'$ with the probability p which will be denoted as $A(f,t) \models_p t'$, the edge $e = (t, t')$ should be given weight p that represents the probability of the existence of e is p. We may denote the *weighted search space graph* by $WSP_n(A,f)$. $SP_n(A,f)$ defined above is actually a simplified version of $WSP_n(A,f)$. Besides the search space graph and weighted search space graph, there exist another type of search space. The algorithms such as Novelty, R-Novelty and Tabu search the state space according to their search histories. Hence, their real search spaces are dynamic. We say their search spaces are *dynamic search space graphs*. The latter two types of graphs are much more complicated than $SP_n(A,f)$ and it is not possible to calculate the precise probability of each edge in the dynamic search space graph. Fortunately, since $SP_n(A,f)$ keeps the topology of the other two types of graph, it provides plenty of information for the study of the structure of the search spaces and the local search mechanism behind them. The search space graphs' structures and the local search mechanism are what this thesis concerns, so we will limit our research to $SP_n(A,f)$ in the thesis. The SSG discussed in this section will be $SP_n(A,f)$ if no special declaration. Note that the percentage of vertices in traps in $SP_n(A,f)$ is actually a lower bound for the percentage of vertices in real traps considering that there are some extra edges which may not exist in a history dependent local search algorithm. For example, in many cases the best variable in a selected clause is not a recently flipped one so Novelty does not even consider the second best variable in search such that the edge that represents the second best one does not exist in this search process at all.

Definition 5.1.4 $\mathcal{SA}_m$ denotes the set of local search algorithms that flip m and only m variables each time. We denote $\mathcal{SA}_1$ as $\mathcal{SA}$. We use $\mathcal{W}$ to represent the family of WalkSAT and $\mathcal{G}$ to represent the family of GSAT. Note that $\mathcal{W}, \mathcal{G} \in \mathcal{SA}$.

Definition 5.1.5 $SP_n(A)=\{SP_n(A,f) : f$ is any CNF formula with n variables$\}$ is the set of all search space graphs of A under CNF formulas on n variables, where $A \in \mathcal{SA}_m$. $SP_n(\mathcal{A})=\{SP_n(A) : A \in \mathcal{A}$ and $\mathcal{A} \subset \mathcal{SA}_m\}$ is a set of $SP_n(A)$.

If graph G' is a subgraph of graph G, it will be denoted as $G' \trianglelefteq G$. If G' cannot be equal to G, it will be denoted as $G' \lhd G$.

5.2 Basic Properties of Search Space Graphs

From the definitions above, we easily get to know two basic properties of search space graphs: $\forall A \in \mathcal{SA}_m, SP_n(A, f) \lhd DHC_n{}^m$ and $\forall A \in \mathcal{SA}, SP_n(A, f) \lhd DHC_n$. Therefore, the search space graphs of WalkSAT and GSAT with n variables are subgraphs of DHC_n. From now on, we use W_{Basic}, $W_{Novelty}$, $W_{Novelty+}$, $W_{R-Novelty}$, $W_{R-Novelty+}$, W_{tabu}, G_{GSAT}, G_{RWalk} and G_{CSAT} to represent the stochastic algorithms: WalkSAT, Novelty, Novelty+, R-Novelty, R-Novelty+, Tabu/WalkSAT, GSAT, GSAT with the random walk strategy and CSAT respectively. Novelty+ and R-Novelty+ are Novelty and R-Novelty with random walk techniques. Furthermore, it is easy to conclude following properties:

$\forall f$ on n variables,

1. $\mathcal{W}, \mathcal{G} \subset \mathcal{SA}$;
2. $\forall G \in SP_n(\mathcal{W}, f), G \lhd DHC_n$;
3. $\forall G \in SP_n(\mathcal{G}, f), G \lhd DHC_n$;
4. $SP_n(W_{Novelty}, f) = SP_n(W_{R-Novelty}, f) \lhd SP_n(W_{Basic}, f)$
5. $SP_n(G_{GSAT}, f) \trianglelefteq SP_n(G_{CSAT}, f) \lhd SP_n(G_{RWalk}, f)$

5.3 Structures in SSG

In this section we will concentrate on structures of SSGs of G_{GSAT}, G_{CSAT}, W_{Basic} and $W_{Novelty}$. The real search space of G_{GSAT} and G_{CSAT} are SSG, the real search space of W_{Basic} is weighted search space graph(WSSG) and the real search space of $W_{Novelty}$ is dynamic weighted search space graph(DWSSG) depending on search histories, therefore, we have included all the three types of graphs in our study. Since the SSGs of Novelty and R-Novelty are the same under the same first order formula, we will only discuss the SSG of Novelty in this chapter. To understand the effects of considering the second best variable in Novelty and RNovelty, we studied the SSG of Novelty without the option to select the second best variable as well. We called this modified Novelty *Only Best*. The study concentrates on the coverage of the traps in SSGs, the average out-degree of SSGs and the coverage of the parts which can reach both traps and solutions in SSG. The coverage of a part means the percentage of vertices in this part. So the coverage of traps in an SSG should be measured by the percentage of vertices in traps. Hence, a smaller value for the coverage of traps indicates that more vertices can reach solutions in SSG through some paths on SSGs and the corresponding algorithm has less chances to get stuck in traps. A trap in an SSG is a subgraph from which there is no path to a solution.

Intuitively, the smaller the average out-degree of an SSG, the faster the convergence speed to solutions is because, in most cases, the solutions are only a tiny portion compared to the entire SSG and more options for each assignment will "confuse" a search process such that it will have more chances to be led away from a correct "direction" to solutions . We will empirically confirm this later. However, a small average out-degree will yield more vertices that are not able to reach solutions. So there might be an optimal value for the out-degree to make less traps with less edges.

By the description above, the entire search space can be divided into four parts: the solutions, the traps, the parts that can reach both traps and solutions through some paths and the parts that can reach solutions but no trap. We will call the parts that can reach both traps and solutions the *intermediate parts* and the last type of parts the *promising parts*. See Figure 5.1, in which the black area represents the traps, the gray part represents the intermediate part, the light gray grided square represents the solutions and the white part represents the promising part. The arrows among the parts represent the directions of the edges between two parts.

To locate these parts in SSGs, we explore reversed SSGs using Depth First Search(DFS), where a reversed SSG keeps all vertices in the corresponding original SSG but reversed all edges in it. We start DFS from the solutions in reversed SSGs. The vertices that are not visited must be in traps. To find the intermediate parts, we use a second DFS starting from the vertices in traps that have been found in the first turn. The visited vertices in this turn are in the intermediate parts. The remaining non-solution vertices are in the promising parts.

Although DFS is a polynomial time algorithm, the number of vertices in graphs increases exponentially. Therefore, its computational complexity is $\Omega(2^n)$, where n is the number of variables. In our experiments we only considered small size instances with 10 to 18 variables because these instances are the ones that we can handle in reasonable time. All the instances

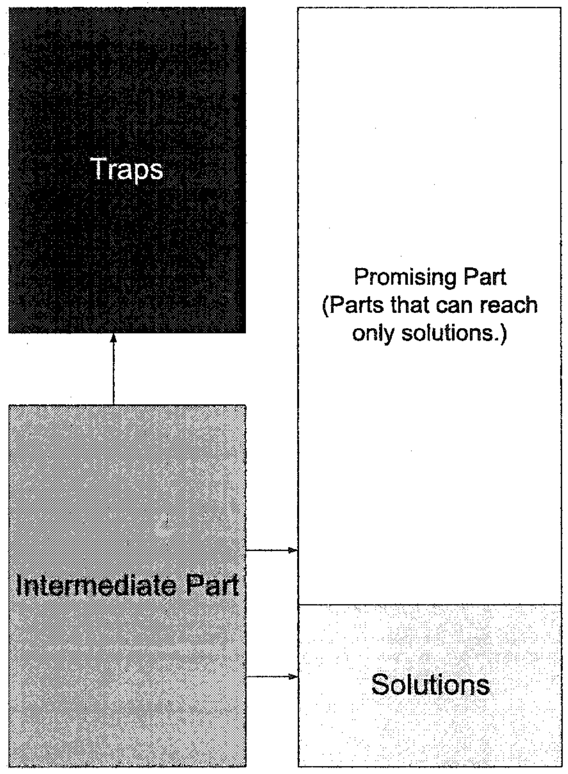

Figure 5.1: Four Parts of an Search Space Graph

tested in the following experiment are random hidden solution 3-SAT instances. Some SSGs under 6 variables are provided in Appendix A.

GSAT

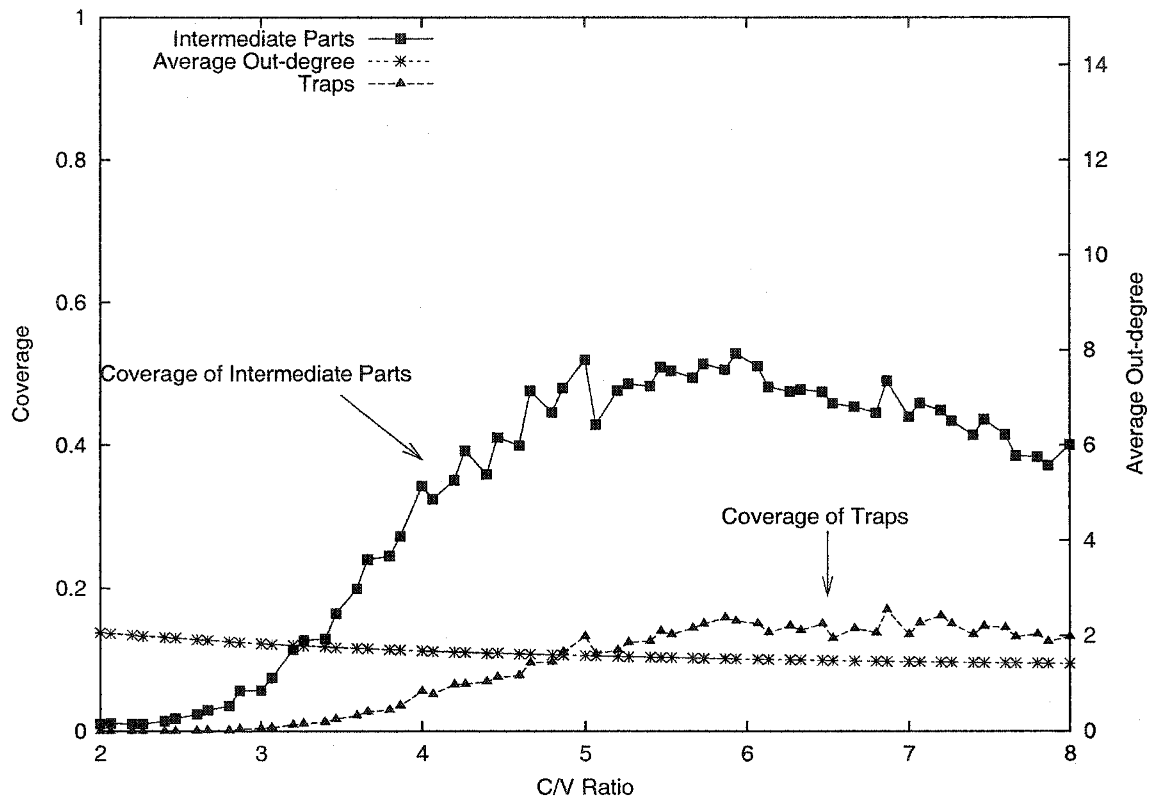

Figure 5.2: The coverage of Traps and Intermediate Parts and The Average Degree in G_{GSAT}'s SSGs with 15 variables

In Figure 5.2, the left Y axis measures the percentage of the vertices in two parts, traps and intermediate parts, the right Y axis measures the average out-degree of SSGs and the X axis represents the C/V ratio. The three axes measure the same three measurements respectively in the following figures. In this figure, the coverage of traps begins to increase from a certain point in interval 3.2-3.5 on X axis(C/V ratio) and it stops increasing when this C/V ratio is approximately 6. The coverage of the intermediate part shows a small peak. It decreases after the value of the C/V ratio is larger than 6, which means more vertices are in the promising part.

We notice that, the average out-degree of $SP_{15}(G_{GSAT}, f)$ decreases slowly, while the

C/V ratio increases. The reason why the out-degree decreases is that G_{GSAT} probably has less options with a larger C/V ratio. G_{GSAT} always selects a variable from the variables that can maximize the number of satisfied clauses. We denote the number of variables that can maximize the number of satisfied clauses as k, the number of variables as n and the number of clauses as m. Given any assignment t, flipping variable v_i, $G_{GSAT}(f,t) \vDash t'_i$. The assignment t'_i may satisfy 0 to m clauses. We denote the number of satisfied clauses by t'_i as l_i. Then $0 \leqslant l_i \leqslant m$. For variables $v_{i_1}, v_{i_2}, \ldots v_{i_j}$, if $l_{i_1} = l_{i_2} = \cdots = l_{i_j}$ are the maximum in $l_1, l_2, \ldots, l_n$, G_{GSAT} has to randomly pick up one from these variables to break the ties. Clearly, the out-degree of the vertex representing assignment t is j in this case. With a larger m, there are less variables that have the same l value because l has more possible values now. So G_{GSAT} will have less ties with a larger m. When n is fixed, a larger C/V ratio means a greater m. This is why the average out-degree decreases when the C/V ratio increases.

We can see that at least 40% of vertices are in the promising part. Considering that there may be around 20% of the vertices in traps, the promising part in G_{GSAT} is not small at all. This partially supports the idea that the heuristic of always choosing the best variable provides a good guidance to solutions. Furthermore, the coverage of traps is up to almost 20%. Thus, G_{GSAT} has up to 20% chance of selecting a vertex as a starting point in a trap directly on $SP_{15}(G_{GSAT}, f)$. It indicates that how to select a starting point for G_{GSAT} is important.

Figure 5.3 shows the changes of the coverage of traps in $SP_n(G_{GSAT}, f)(n = 10, 11, \ldots, 18)$ while the number of variables increases and the C/V ratio is fixed at 6 where 6 is a stable point for the coverage of traps. We see that the average out-degree is stable, which can also be explained by the reason why the average out-degree decreases when the C/V ratio increases. Considering that m increases 6 times faster than n, the effects brought by the increase of n can be counteracted by the increase of m. We also see that the coverage of traps decreases. The decrease of the coverage of traps indicates that G_{GSAT} may need less number of restarting under larger size instances. Of course, we need further experiments under large size instances to support this point.

CSAT

Compared with G_{GSAT}, CSAT(G_{CSAT}) has more freedom to select variables. We expect that $SP_{15}(G_{CSAT}, f)$ has a smaller coverage of traps and larger average out-degree. Figure

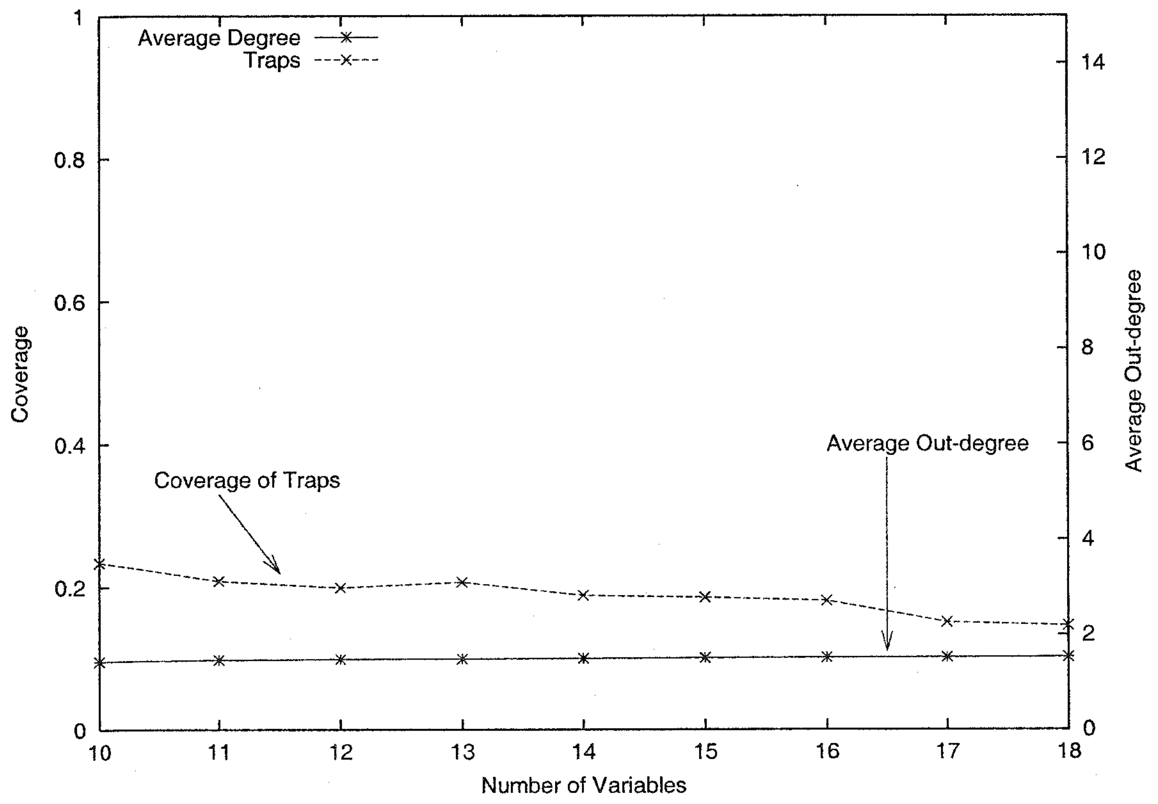

Figure 5.3: The Coverage of Traps and The Average Degree with the Number of Variables Changing in G_{GSAT}'s SSGs

5.4 is a graph showing the structures of $SP_{15}(G_{CSAT}, f)$.

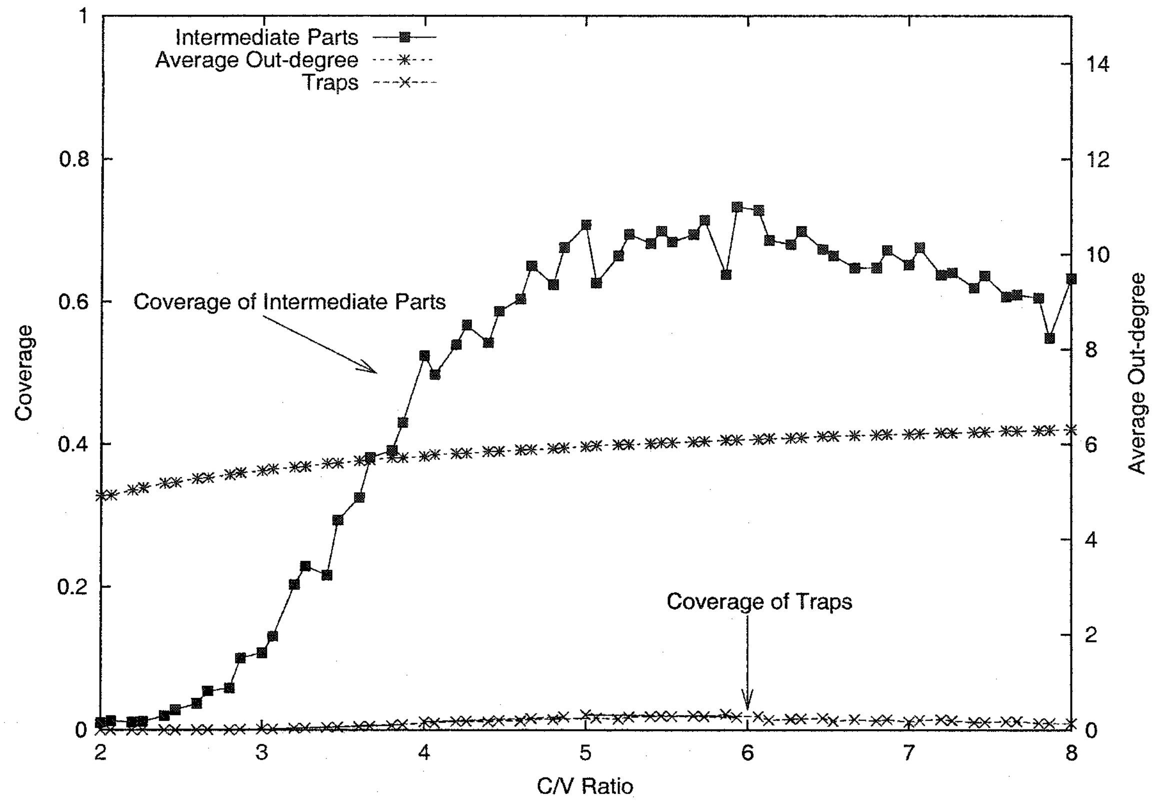

Figure 5.4: The Coverage of Traps and Intermediate Parts and The Average Degree in CSAT's SSGs

Compared with G_{GSAT}, the coverage of G_{CSAT} traps is much smaller, say, 2.2% at most. At the same time, its intermediate part is larger than G_{GSAT}'s. The differences are caused by more edges being added into G_{CSAT}. Because $SP_n(G_{GSAT}, f) \trianglelefteq SP_n(G_{CSAT}, f)$, so $SP_n(G_{CSAT}, f)$ is $SP_n(G_{GSAT}, f)$ with extra edges. In other words, $E_{GSAT} \subseteq E_{CSAT}$, where E_{GSAT} and E_{CSAT} are the sets of edges of $SP_n(G_{GSAT}, f)$ and $SP_n(G_{CSAT}, f)$, respectively. On $SP_n(G_{GSAT}, f)$, 1) if an edge from a vertex v in a trap to a vertex u in another part is added, all vertices that can reach v will belong to the intermediate parts or promising parts; 2) if an edge from a vertex v in the promising parts to a vertex u in the intermediate parts or traps is added , all vertices in the promising parts that can reach v will belong to the intermediate parts. In both cases, the coverage of intermediate parts increases and in the first case, the coverage of traps reduces. Compared with G_{GSAT}, the average out-degree of G_{CSAT} is almost doubled, which indicates that the number of edges are doubled. From the analysis above, we know that the coverage of traps will be smaller and the coverage of intermediate parts of $SP_n(G_{CSAT}, f)$ will be larger than $SP_n(G_{GSAT}, f)$ in most cases.

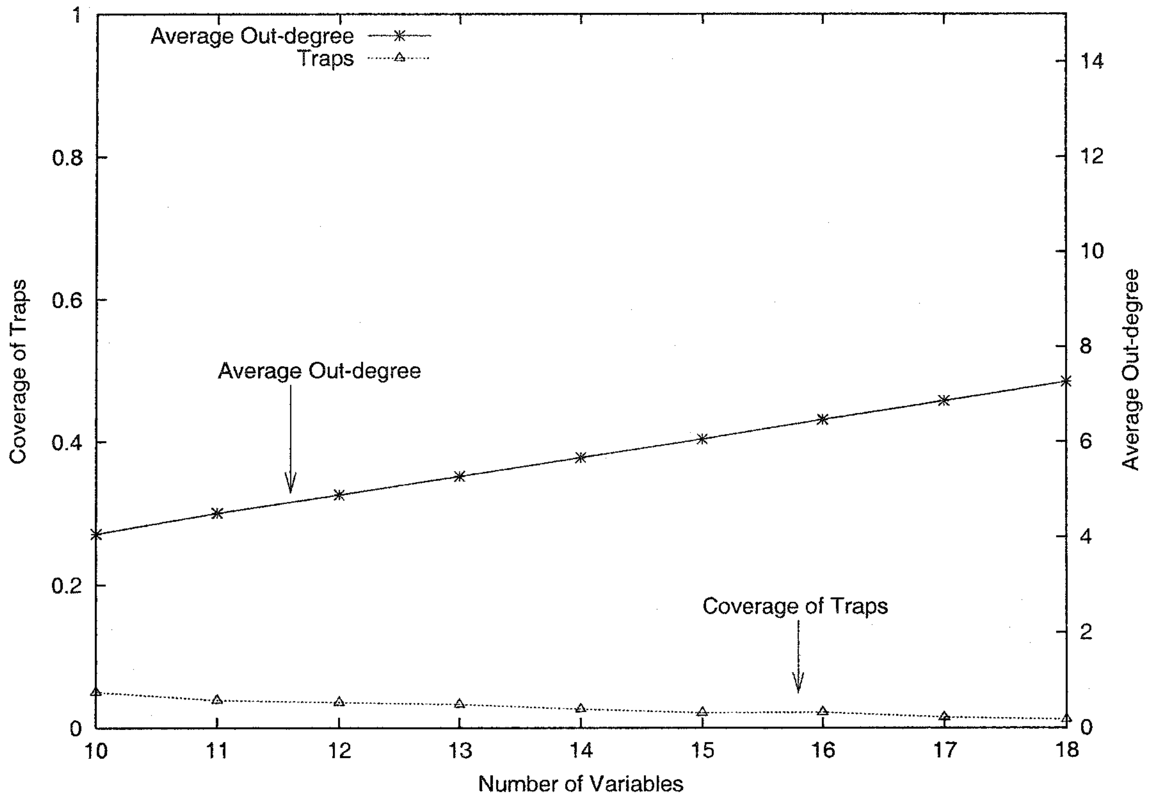

Figure 5.5: The Coverage of Traps and The Average Degree with the Number of Variables Changing in CSAT's SSGs

Figure 5.5 is the graph for the coverage of traps when the number of variables increases and the C/V ratio is fixed at 5. The average out-degree increases in a linear scale and the rate of traps decreases also in a linear scale. We guess that the coverage of the traps converges to a constant eventually when the number of variables is large enough.

Basic WalkSAT

Although researchers guess that WalkSAT(W_{Basic}) is probabilistic approximate complete(PAC), there is no formal proof for that. That an algorithms is PAC indicates that the probability of one vertex being checked is larger than zero. Culberson et. al.[7] proved that W_{Basic} for 2-SAT is PAC, however, whether W_{Basic} for k-SAT($k \geq 3$) is PAC or not is unknown.

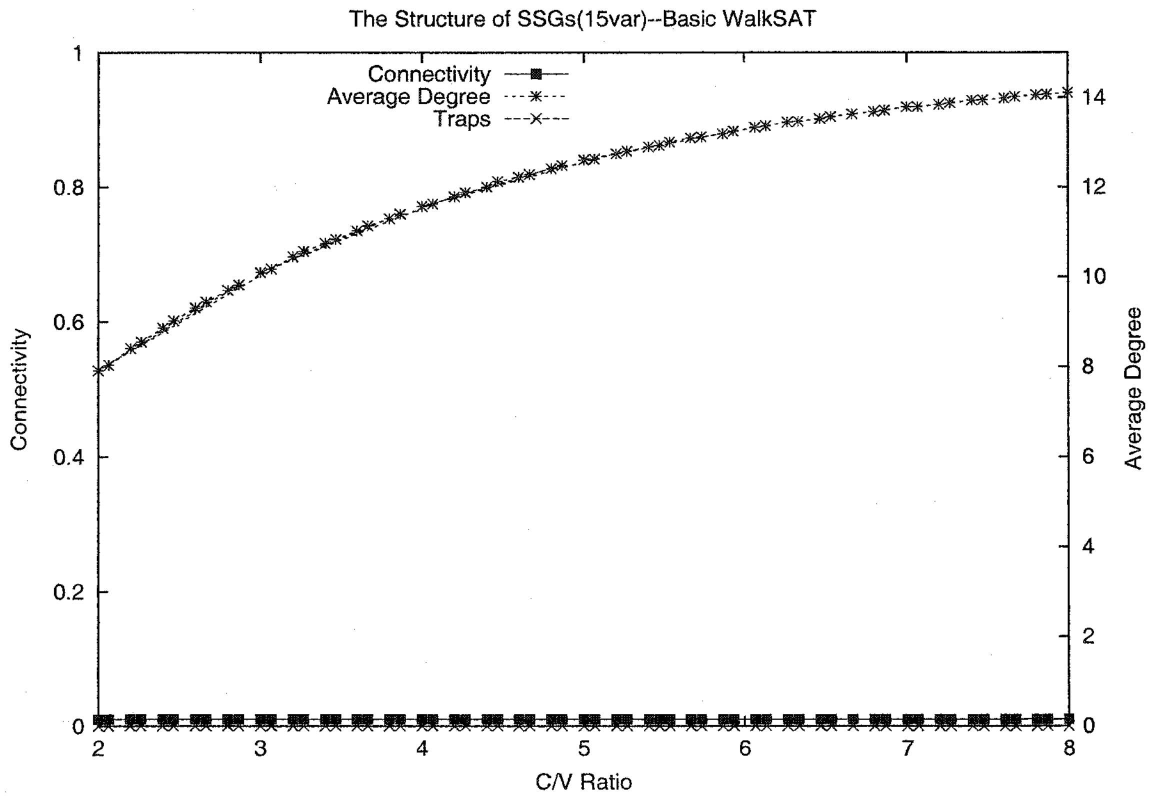

Figure 5.6: The Coverage of Traps and Intermediate Parts and The Average Degree with Various C/V Ratios in Basic WalkSAT's SSGs(15 Variables)

In our experiments(Figure 5.6), unlike $SP_{15}(G_{GSAT}, f)$ and $SP_{15}(G_{CSAT}, f)$, no trap was found in $SP_{15}(W_{Basic}, f)$. In other words, every vertex is on a path to at least one solution in $SP_{15}(W_{Basic}, f)$ in our experiments. This result further supports that W_{Basic} is PAC. We see that the average out-degree is about 14 when the value of the C/V ratio is 8, which indicates almost all variables are selectable on average. In this case, W_{Basic} behaves more like random selections. The bias only comes from the frequency of the appearance of each variable. This bias is too weak to work well.

As the coverage of traps for W_{Basic} is always 0% in our experiments, we do not discuss the tendency of the change of the coverage of traps further.

Novelty

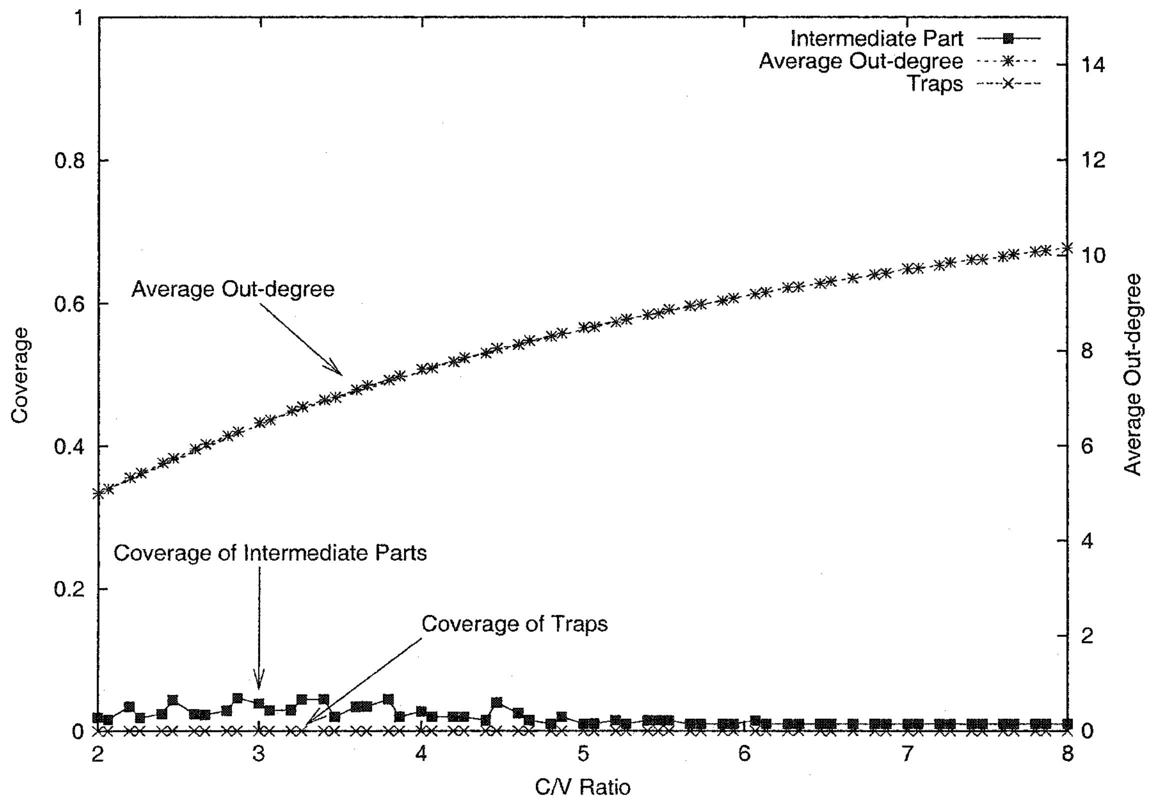

Figure 5.7: The Coverage of Traps and Intermediate Parts and The Average Degree with Various C/V ratios in Novelty's SSGs(15 Variables)

The experimental results of $W_{Novelty}$(Figure 5.7) looks similar to that of W_{Basic} except that the coverage of traps for $W_{Novelty}$ is slightly larger than zero and its average out-degree is smaller. We show the rescaled graph below(Figure 5.8). Compared with $SP_{15}(G_{CSAT}, f)$, $SP_{15}(W_{Novelty}, f)$ has much smaller coverage of traps and intermediate parts. From the rescaled graph, we can tell that the coverage of traps trends to decrease when the C/V ratio increases. However, unlike W_{Basic}, a few of $SP_{15}(W_{Novelty}, f)$ contain some very small traps, although in most cases the coverage of traps in $SP_{15}(W_{Novelty}, f)$ is 0. We say that the coverage of traps for $SP_{15}(W_{Novelty}, f)$ is as small as that of $SP_{15}(W_{Basic}, f)$ but $SP_{15}(W_{Novelty}, f)$ has smaller average out-degree, on the other hand. The smaller coverage of traps means that it is not easy for $W_{Novelty}$ to get stuck in traps compared with G_{GSAT} and G_{CSAT}; the smaller average out-degree indicates that $W_{Novelty}$ has a stronger bias than W_{Basic} and this stronger bias leads to a faster convergence speed to solutions. We will empirically confirm the latter point later. This is partially the reason why $W_{Novelty}$ performs better than W_{Basic}, G_{GSAT} and G_{CSAT}.

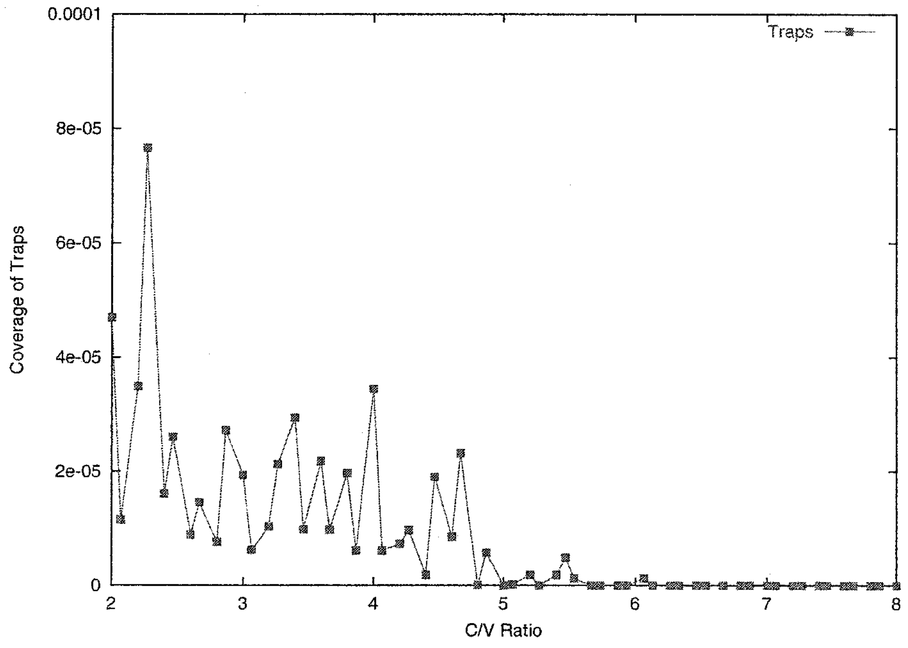

Figure 5.8: The Coverage of Traps in Novelty's SSGs(Rescaled) Under 15 Variables

Because Novelty's traps are very small and the percentage of instances containing traps is very small too, the information based on average case is not really helpful. We further analyzed Novelty. This time we generate 5000 instances for each C/V ratio.

In Figure 5.9, X axis represents the ratio of instances and Y axis represents the number of instances containing traps. Clearly, there is an obvious peak at 3.4 on X axis. However,

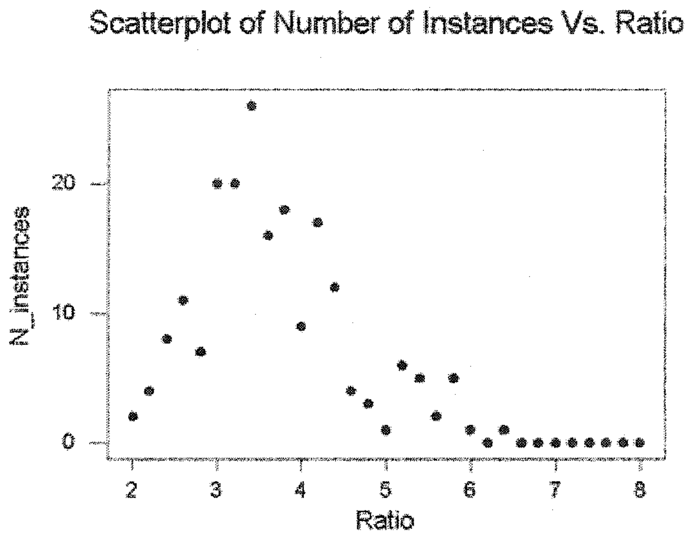

Figure 5.9: Scatter Plot of the number of Novelty SSGs Containing Traps. (It counts the number of instances containing traps at each C/V ratio)

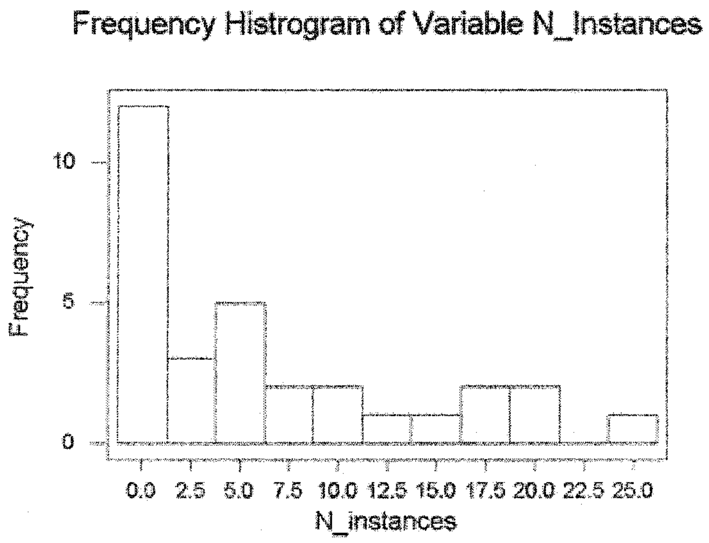

Figure 5.10: Frequency Histrogram of Number of Instances Containing Traps for Novelty

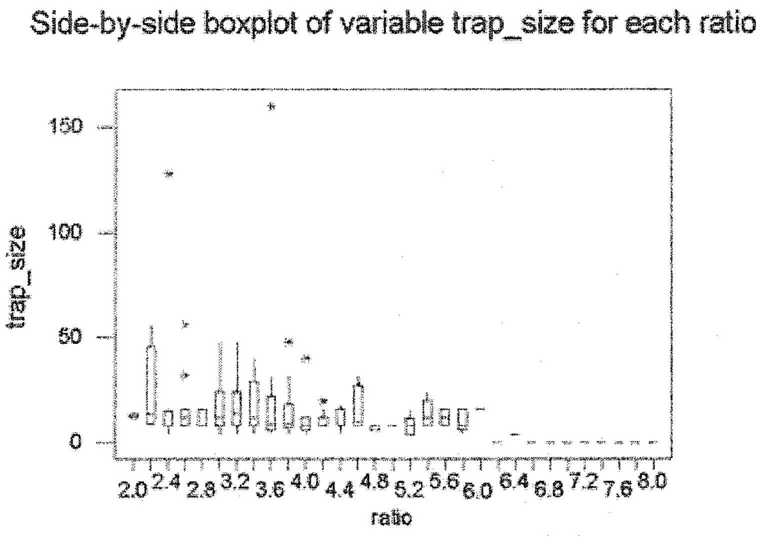

Figure 5.11: Side-by-Side Boxplot of the number of Novelty SSGs Containing Traps. (The trap size is measured by the number of vertices. The top and bottom of a box represent 75% and 25% of the number of vertices in traps respectively. The bar inside a box is the median.)

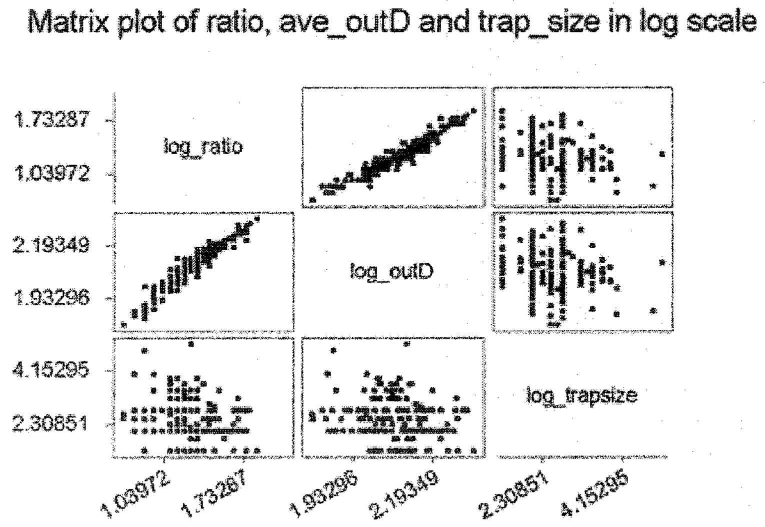

Figure 5.12: Relations between the Size of Traps, the Average Out-degree and the C/V ratio for Novelty

none of the instances at the C/V ratio being greater than 6.6 contains any traps. We plot a frequency histrogram of the number of the instances containing traps(Figure 5.10). It is skewed to the right. Most ratios have less than 5 instances with traps in 5000 instances. We used the boxplot to analyze the size of Novelty's traps(Figure 5.11). The line in middle of each box is the median value of the size of traps at each C/V ration. The top and the bottom of each box represent the 75% and 25% of the number of vertices in traps at each C/V ratio. The star points out of those boxes are outliers according to the statistical analysis. Since their sizes are too large compared with others, it has been considered as occasional incidents. However, these exceptions may be the interesting parts that need further investigation. Moreover, we can see that the values of the medians are stable.

To study the correlations among the size of traps, the average out-degree and the C/V ratio, we ploted Figure 5.12, which is a matrix plot of those three measurements with a logarithmic scale. In the plot of "ratio" $\times$ "log_outD", which use C/V ratio and average out-degree as Y and X axes respectively, we see that there is a linear relationship between these two measurements on the logarithmic scale, but the relationships of the other two pairs are almost random. So we conjecture that the average out-degree does not affect the size of traps in Novelty.

Only Best

We may notice that the probability of $W_{Novelty}$ selecting the best variable and the probability of it selecting the second best variable are quite different. Although the exact probability of selecting either variable is unknown since it depends on search histories, we can roughly estimate that the probability of selecting the second best variable is much smaller. So we repeat experiments on the graph with only the edges that represent the best variables in each unsatisfied clause. We call the corresponding algorithm *Only Best*($W_{OnlyBest}$), which only selects the best variable in the selected unsatisfied clause. Let's have a look at Figure 5.13. There is an obvious peak for both the coverage of traps and the coverage of intermediate parts at around 4 on the C/V ratio. Considering that the average out-degrees increase monotonically, the phenomenon is interesting, because it confirms that the size of traps is not or not only related to the average out-degrees. We guess that formulas might also affect the structure of SSGs through affecting the distribution of edges in SSG. Figure 5.14 shows the change tendency of the average out-degree and the coverage of traps when the number of variables increases. It is similar to CSAT.

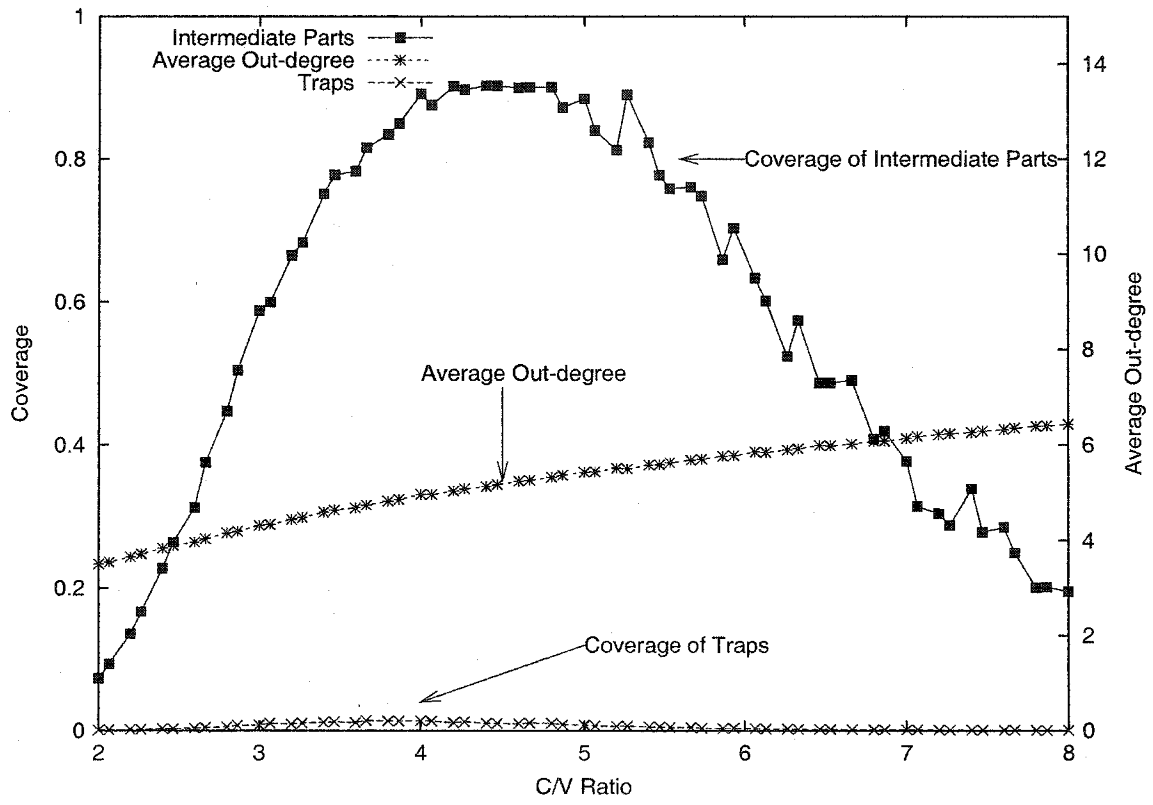

Figure 5.13: The Coverage of Traps and Intermediate Parts and The Average Degree in Only Best's SSGs

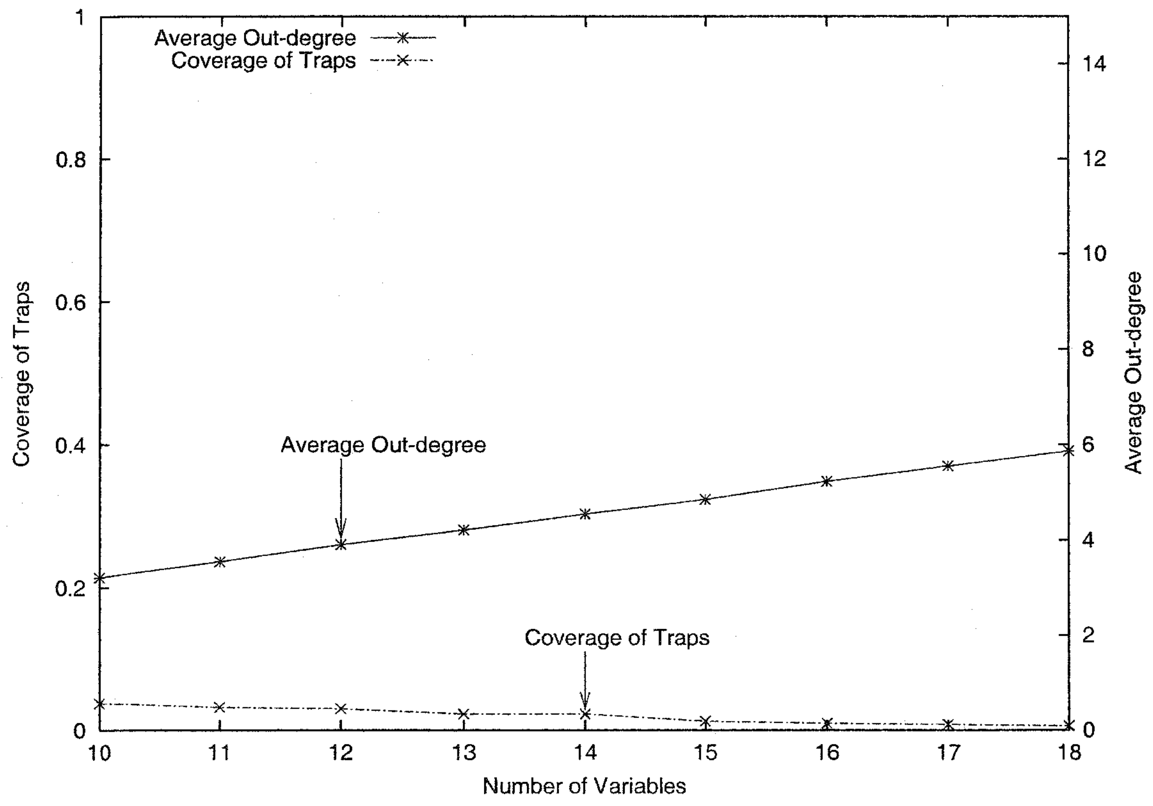

Figure 5.14: The Coverage of Traps and The Average Degree with the Number of Variables Changing in Only Best's SSGs(The C/V ratio=4)

5.4 Convergence Speed

Each local search algorithm's SSG has its own properties which make it different from the others. The coverage of traps and average out-degree are two of the features affecting performance. If an SSG contains a relatively larger coverage of traps, these traps will affect the corresponding algorithm's performance. In this scenario, restarting and random walk will be essential for the algorithm because it needs these two strategies to escape traps. G_{GSAT} is this type of algorithm. If an SSG contains tiny trap areas, restarting and random walk may not be the best strategies because there is no guarantee that the new start point will be better than the current assignment if the current assignment is not in any trap. Note that no traps does not indicate a good performance. For example, W_{Basic} does not work very well. A lot of edges avoid constructing traps but too many edges may lead to a low convergence speed to solutions. In this scenario, the average out-degree may be the main factor affecting performance through affecting the convergence speed to solutions. $W_{Novelty}$ and $W_{R-Novelty}$ are this type of algorithm. In this section, we discuss some experiments on convergence speed of G_{GSAT}, CSAT, Novelty(R-Novelty) and Basic WalkSAT.

We use a Markov chain model to simulate transfers between vertices, where "transfer" means that if there is an arc from vertex v_1 to vertex v_2, the search process can walk from v_1 to v_2 by flipping a selected variable, which is represented $A(f, t_{v_1}) \models t_{v_2}$ where A is a local search algorithm. If a vertex v has k out-arcs to $v_1, v_2, \ldots v_k$ respectively, the probability of a transfer from v to v_i is $1/k$ which is represented by $A(f, t_v) \models_{1/k} t_{v_i}$. In other words, we assume A transfers with a uniform distribution on each edge. We can use a stochastic matrix P to represent the transfers among vertices.

We give weights to each vertex and denote them as $a = (w_1, w_2, \ldots, w_n)$, where the weights represent the probability of the search process checking this vertex. Since we assume that there is no bias at the selection of starting points, we initially set $a = (1, 1, \ldots, 1)$. Therefore, $a' = a \times p$ can be considered as a vector of the probability of the search process stopping at a vertex at first iteration, where the probabilities have been represented as weights. The sum of the i-th column in $a_1 = a \times p$ is the weight of the i-th vertex after one iteration. So the second iteration can be simulated by $a_2 = a_1 \times p = a \times p^2$. By induction $a_n = a \times p^n$. This is a simple infinite markov chain model. So there exists a q such that $a(P^t - P^{t+1}) = E(t \geq q)$, where E is a matrix in which any element e satisfies $-\delta < e < \delta$(δ is a small positive constant). In our SSG model, the weights should converge to both solution vertices and traps. Therefore, using the number of iterations as time tags, we can measure the convergence speed by the number of vertices whose weights are larger than a small positive constant ϵ. We use ϵ here because the weights on non-solution and non-trap vertices that is on any cycle will not be zero when we find the q such that $a(P^t - P^{t+1}) = E(t \geq q)$. But their weights should be very small such that we can ignore them. In this experiment, ϵ is set to 0.00001. If the weight of a vertex is less than ϵ, we consider the probability of stopping at this node being zero after q iterations. We must clarify that an algorithm's convergence speed here is the convergence speed on its SSG as we defined previously.

Figure 5.15 shows the simulation of convergence process under formulas on 6 variables with the C/V ratio being 3.5. The X axis represents the number of iterations and the Y axis represents the number of vertices whose value is larger than ϵ. G_{GSAT} and G_{CSAT}'s convergence speeds are the fastest. The next fastest one is $W_{OnlyBest}$. Its convergence speed is very close to G_{GSAT} and G_{CSAT}. $W_{Novelty}$'s convergence speed is second slowest one. They all follow the same pattern—starting with a steep drop and then followed by a long flat curve. W_{Basic}'s is the worst one. Its curve does not have a steep drop. The number of vertices with value being larger than ϵ drops smoothly. We notice that W_{Basic} does not really converge to the solutions even after 1024 iterations, while the others converge to almost the same number of vertices.

Let's have a look at Figure 5.16, which shows the simulation of the convergence processes on instances on 10 variables with the C/V ratio being 6. Their convergence speeds still follow the same order as in Figure 5.15. But $W_{Novelty}$'s curves are smoother than that in Figure 5.15, which indicates that $W_{Novelty}$ converges slower under these instances. As for W_{Basic}, it does not converge. G_{GSAT}, G_{CSAT} and $W_{OnlyBest}$ still show a deep drop on the number of vertices whose weights are greater than ϵ.

To further understand how their convergence speeds are effected, Figure 5.19 through Figure 5.21 compare these local search algorithms' convergence speeds on instances with various C/V ratios. In Figure 5.19, we find that the change of $W_{Novelty}$'s convergence speed becomes slower, or say, the curve becomes less steep when the C/V ratio increases. On the other hand, W_{Basic}'s convergence speed is greatly affected by the C/V ratio(see Figure 5.20). There is an obvious plateau before W_{Basic} begins to converge. This plateau extends when the C/V ratio increases. When the ratio is equal to 8, it does not converge at all in the

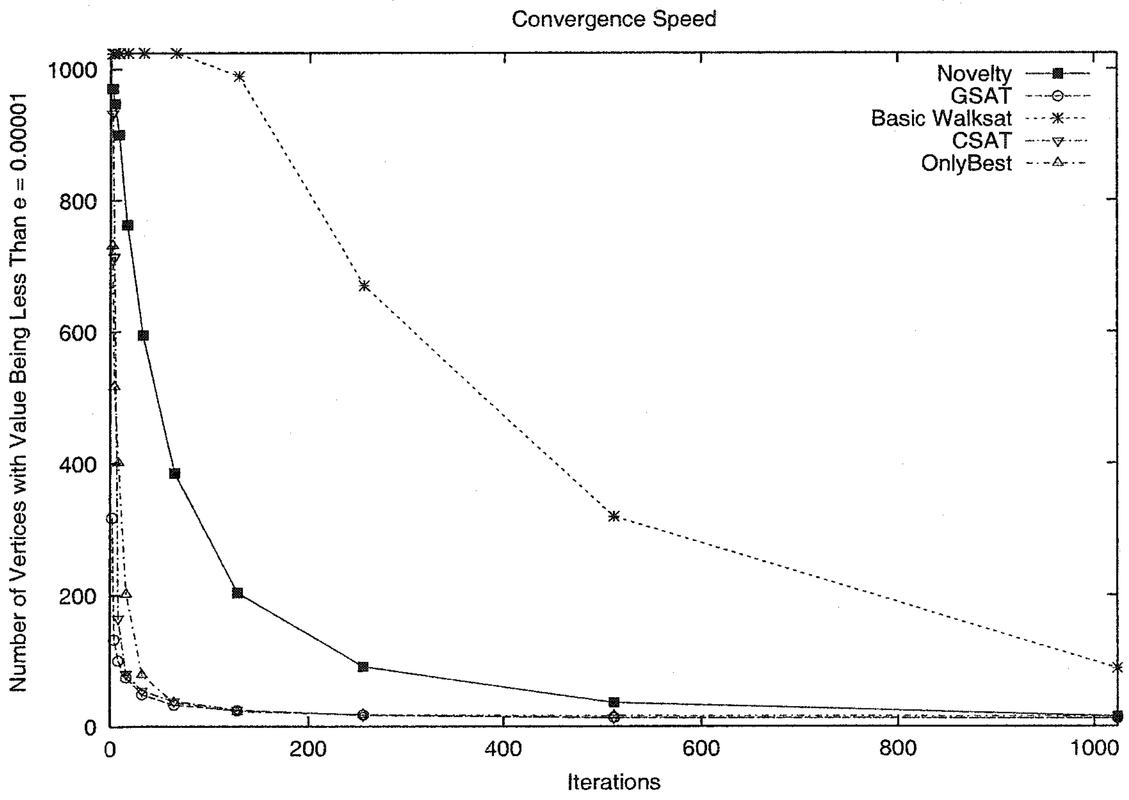

Figure 5.15: Simulation of Convergence Speed on 3-SAT instances with 10 Variables and C/V ratio value being 3.5

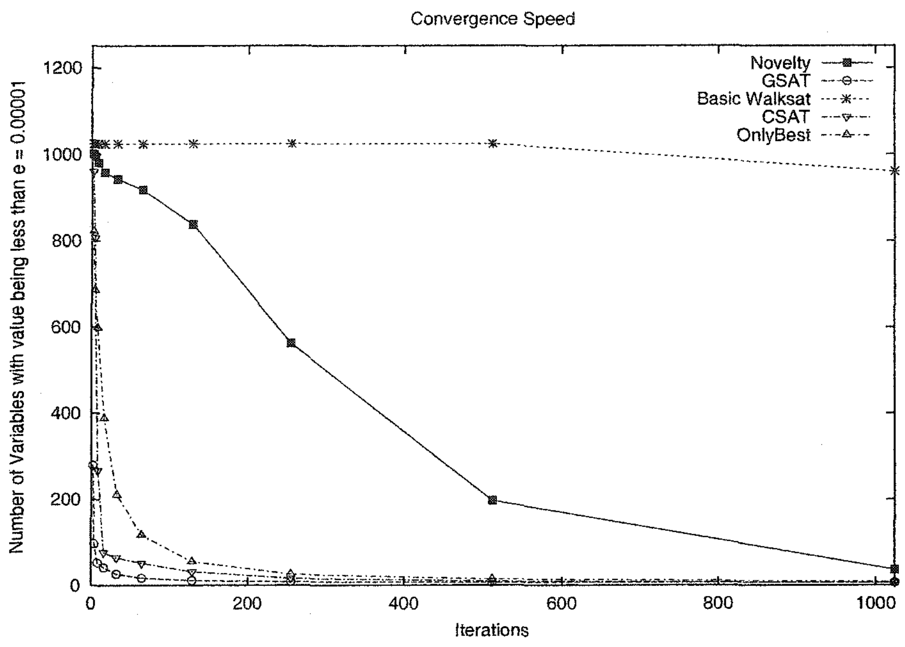

Figure 5.16: Simulation of Convergence Speed on 3-SAT instances with 10 Variables and C/V ratio value being 6

first 1024 iterations. Figure 5.17, Figure 5.18 and Figure 5.21 show that the convergence speeds of G_{GSAT}, G_{CSAT} and $W_{OnlyBest}$ are not greatly affected by the C/V ratio. All of their convergence speeds are quite stable, compared with $W_{Novelty}$ and W_{Basic}. We also notice that G_{GSAT} converges very fast such that the number of the vertices whose weight is larger than ϵ has reduced to less than 400 at the first 2 iterations. It is less than half of G_{CSAT}'s at the same point.

In contrast to the other three algorithms, G_{GSAT}'s convergence speed increases when the C/V ratio increases. The larger the C/V ratio, the faster the convergence speed. Recall that only G_{GSAT}'s average out-degree decrease when the C/V ratio increases(Figure 5.2). So for G_{GSAT}, the change of the convergence speed has the same tendency of the change of the average out-degree. On the other hand, for the other three algorithms, the change on the convergence speeds also have the same tendency as their changes on the average out-degree. W_{Basic} has such a large average out-degree at the high C/V ratio that it does not converge in first 1024 iteration. $W_{Novelty}$ has a larger average out-degree than G_{GSAT} and G_{CSAT} and $W_{Novelty}$ has slower convergence speed than them but it has a faster convergence speed than W_{Basic} because of a smaller average out-degree than W_{Basic}. All of these empirically

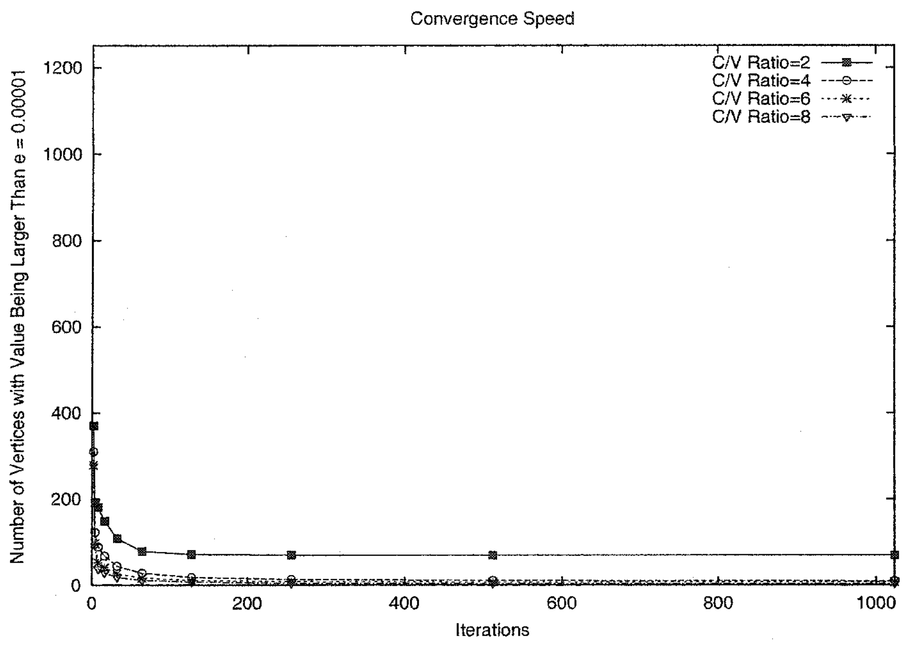

Figure 5.17: Simulation of G_{GSAT}'s Convergence Speed on 3-SAT instances with 10 Variables and various C/V ratios

supports that the average out-degree greatly affect the convergence speed and the smaller the average out-degree, the faster the convergence speed.

5.5 Conclusion

In this chapter, we empirically studied SSGs of G_{GSAT}, G_{CSAT}, W_{Basic}, $W_{Novelty}$ and $W_{OnlyBest}$ under hidden solution 3-SAT instances. Through the analysis of the coverage of traps and intermediate parts, we find that each algorithm's SSG has its own features. $W_{Novelty}$ shows that it has tiny traps and a good convergence speed. Since the probability of flipping the second best variable is small, the real convergence speed of $W_{Novelty}$ is between the speed of $W_{Novelty}$ and the speed of $W_{OnlyBest}$ on SSG. In Chapter 4, we have shown that W_{Novely} usually out-performs the others. It partially supports that both a small coverage of traps and a faster convergence speed are necessary for good performance. Generally, G_{CSAT} also out-performs G_{GSAT}. In this chapter, we see that G_{CSAT} has an obvious smaller coverage of traps and a comparable convergence speed to G_{GSAT}. On the other hand, W_{Basic} has no trap, but its convergence speed is too slow such that it has the worst performance in Chapter 4. We empirically confirmed the necessity of both the small

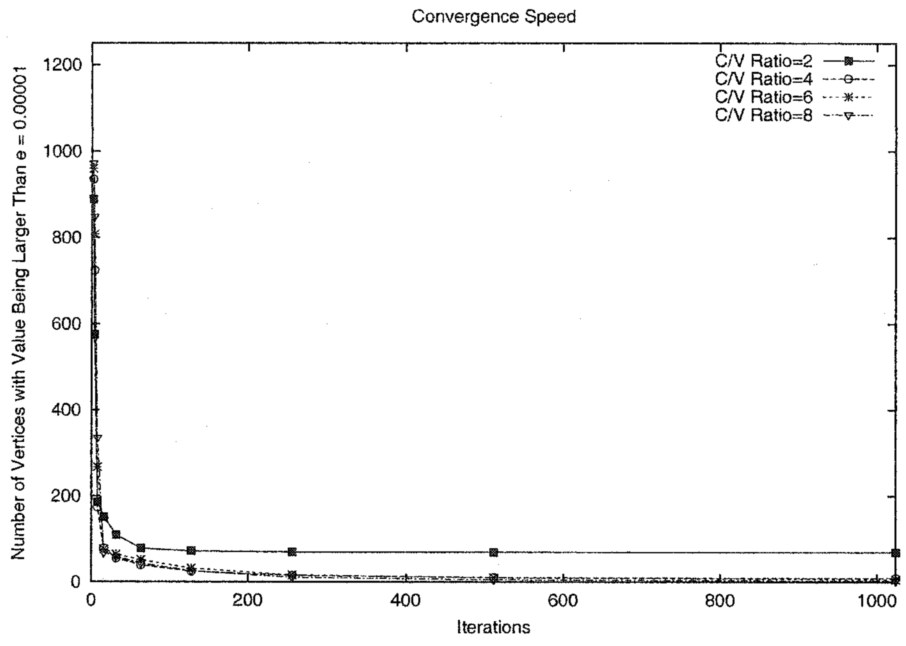

Figure 5.18: Simulation of CSAT's Convergence Speed on 3-SAT instances with 10 Variables and various C/V ratios

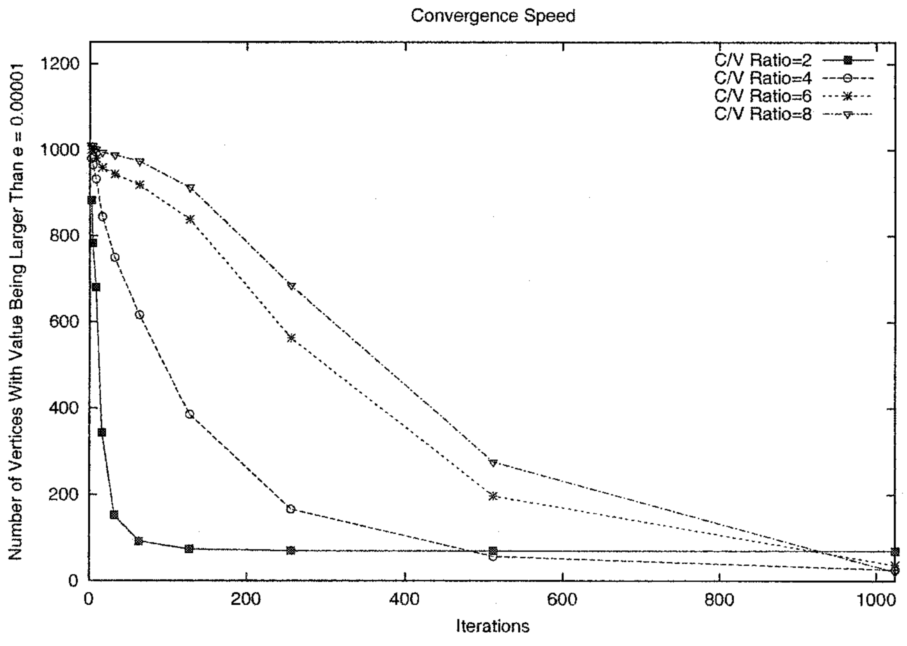

Figure 5.19: Simulation of Novelty's Convergence Speed on 3-SAT instances with 10 Variables and various C/V ratios

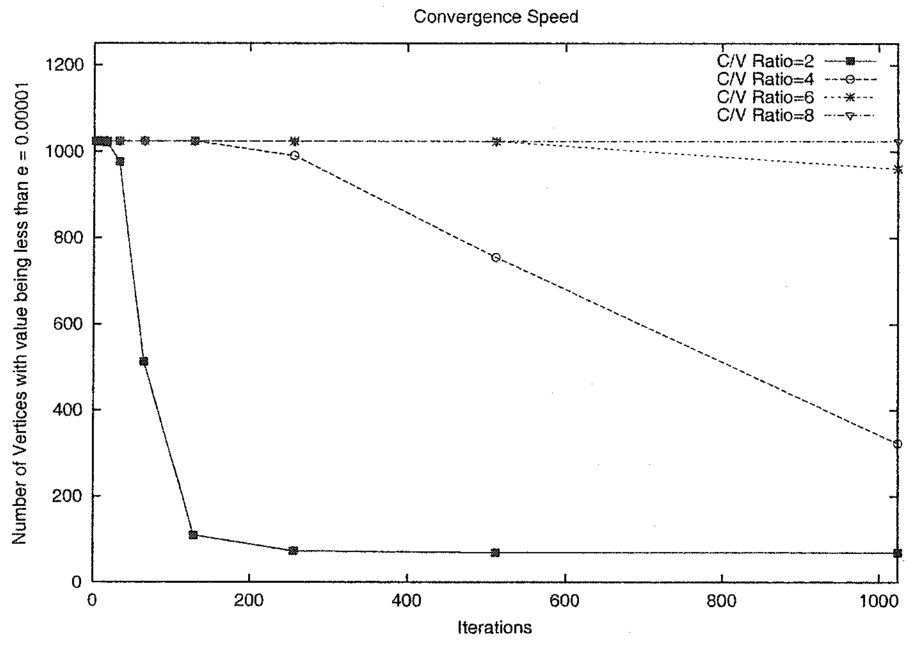

Figure 5.20: Simulation of Basic WalkSAT's Convergence Speed on 3-SAT instances with 10 Variables and various C/V ratios

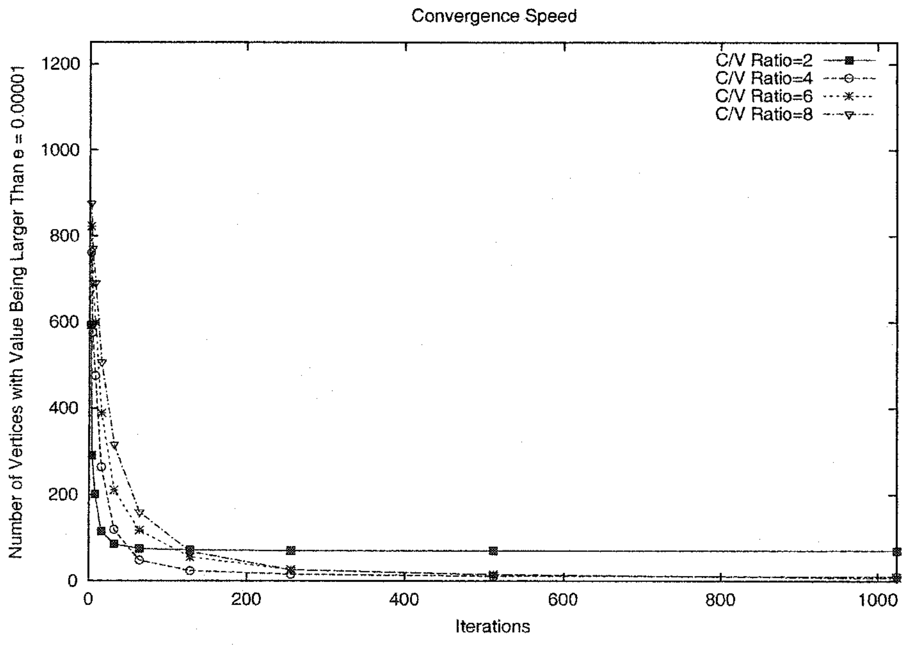

Figure 5.21: Simulation of Only Best's Convergence Speed on 3-SAT instances with 10 Variables and various C/V ratios

coverage of traps and the fast convergence speed for good performance.

However, what determines the coverage of traps and the convergence speed? Assume we have a graph G with traps $T = \{t_1, t_2, \ldots, t_p\}$ where $t_i, i = 1, 2, \ldots, p$ are traps. If extra edges from t_i to any other part are added, some vertices in traps will be in intermediate parts or the promising parts. However, if an edge whose two ends are in the same part is added, it will not change the coverage of any part. If an edge from a promising part to a trap or an intermediate part is added, it only increases the size of the intermediate part. We do not think this kind of edge will help for a better performance since it only increases chances to traps. Any edge from an intermediate part to any other part will not change the coverage of any part since it has edges to all the other parts already. By the simple analysis above, we know that, theoretically speaking, a greater average out-degree does not necessarily mean a smaller coverage of traps. However, note that any edge added will not cause a larger coverage of traps, so when the average out-degree increases much, there is a greater probability of constructing an SSG with smaller coverage of traps. On the other hand, we have also noticed that the tendency of the change on the coverage of traps is not the same as the tendency of the change on the average out-degree when the C/V ratio increases because the average out-degree usually increases or decreases monotonicly but the coverage of traps does not show a monotonic increase or decrease. In the analysis in the previous sections for $W_{Novelty}$ and $W_{OnlyBest}$, we have found these two factors are not strongly related. So the average out-degree is not the only factor that affects the the coverage of traps. We conjecture that the hardness of SAT instances affects the structure of SSG as well through affecting the distribution of edges. We still need further experiments to confirm this, though. On the other hand, our experiments also show that the average out-degree is strongly related to the convergence speed. The smaller the average out-degree, the faster the convergence speed.

So we guess that a key point for designing a good algorithm is to pursue a proper average out-degree that can lead to a relatively small coverage traps and a good convergence speed. This might be able to explain why the automated heuristics generator by Alex Fukunaga [15] works well, as we mentioned in Chapter 2. He succeed in finding new heuristics by deploying multiple heuristic methods with a probability for each of them. In this way, he mixed the search space graphs of various local search procedures. We guess that his method adjusted the distribution of edges and the average out-degree of an SSG such that it leads to good values for both factors.

All these experiments are based on small size instances. However, we guess that the tendency of the change of the coverage of traps and the average out-degree might hold but they might converge to some constants under large size random hidden solution 3-SAT instances. Since the convergence speed are most likely affected by the average out-degree, we conjecture that the tendency of the convergence speed would keep under large size random hidden solution 3-SAT instances as well.

Chapter 6

Conclusion

Some local search procedures work efficiently on SAT problems under some sub-classes of SAT problems, as we have seen in Chapter 4. Novelty can even solve 3-colorable FLAT graph instances with up to 775 vertices in graphs that are 2325 variables in SAT formulas. The search mechanism behind this efficiency is still not clearly understood. In this thesis, we tried to analyze them more deeply by exploring the search spaces of those local search algorithms under small size instances, say, instances on 10 to 18 variables. We have presented how the coverage of traps and the coverage of intermediate parts change when the number of variables or the C/V ratio increases. The different coverage of the traps among various algorithms shows that an algorithm having more possible moves or bigger average out-degree in each step has a smaller coverage of traps, although the change tendencies of the coverage of traps for each algorithm are not the same. An algorithm like GSAT that has few choices at each step has a larger coverage of traps. It easily gets stuck in traps, although it converges pretty fast. However, this does not mean that more choices the better performance. Basic WalkSAT does not have any traps in our experiments. But we can see that it converges quite slowly when the C/V ratio of 3-SAT instances is big, say larger than 6. Novelty/R-Novelty are ones that have the best balance between the coverage of traps and the convergence speed. Here, we assume that a perfect local search algorithm lets its search space graphs contain no traps. Reducing any edge in this perfect search space will cause traps and $\sum_{i,j}^{2^n}(a_i - a_j)^2$ should be minimized, in which a_i is the out-degree of the i-th vertex. We have empirically confirmed that an algorithm being good at both the coverage of traps and the convergence speed will lead to good performance. We also guess that the SAT instances affect the coverage of traps as well through affecting the distribution of edges in graph since not all edges added into a graph can lead to a smaller coverage traps..

So far, all our analyses on the coverage of traps and the convergence speed are done under small size SAT instances. We need further experiments to analyze the tendency of the change of the coverage of each part in search spaces under larger size instances. We may explore large instances search spaces traps by the sampling method. First, track the search process of a local search algorithm without random-walk and restarting. Once we find an assignment that has been visited many times, this assignment is probably in a trap. Second, to find the possible trap, explore the search space using breath first search or depth first search on the search space graph starting from this assignment. If there is a trap containing this assignment, this trap has no edge out. But if the trap is huge, it is difficult to identify it since we do not know when to stop. If the assignment is not in a trap, the search will take a long time. As a trade-off, we can set an upper limit on the number of assignments for this search since a complete search is not realistic in this scenario. However, we will not be able to find any trap that contains more nodes than the limit. We need statistical theory tools for the analysis of the result because the analysis is not easy when a local search algorithm searches all states with some biases such that the traps found are not on a uniform distribution in the whole state space.

We had mentioned that SAT instances might affect the distribution of edges in SSG as well. We can analyze the distribution of edges in SSG under various classes of SAT instances using statistical methods.

In the process of exploring the search space graphs, we have accumulated some experience on constructing small size traps. We can estimate possible traps by gathering and analyzing the pattern of traps in small size instances. Avoiding visiting assignments in traps, a local search algorithm with smaller average out-degree can be improved. This kind of algorithm may benefit from faster convergence speed but suffer less from traps. This might be another interesting future research direction.

Appendix A

Trap Examples

Figure A.2 and Figure A.5 show a example for each of Novelty and GSAT. Table A.1 and Table A.2 are the formulas of these two examples respectively.

$\{x_1 \vee \neg x_3 \vee x_6\}$	$\{\neg x_2 \vee \neg x_4 \vee \neg x_6\}$	$\{x_2 \vee \neg x_3 \vee \neg x_6\}$	$\{\neg x_2 \vee \neg x_3 \vee x_6\}$
$\{x_2 \vee \neg x_3 \vee \neg x_5\}$	$\{\neg x_3 \vee \neg x_5 \vee x_6\}$	$\{\neg x_4 \vee x_5 \vee x_6\}$	$\{x_2 \vee x_5 \vee \neg x_6\}$
$\{\neg x_1 \vee \neg x_5 \vee x_6\}$	$\{x_2 \vee \neg x_5 \vee x_6\}$	$\{x_3 \vee x_4 \vee x_5\}$	$\{\neg x_1 \vee x_3 \vee x_4\}$
$\{\neg x_2 \vee \neg x_4 \vee x_5\}$	$\{x_1 \vee x_3 \vee \neg x_4\}$	$\{\neg x_1 \vee \neg x_2 \vee \neg x_4\}$	$\{\neg x_2 \vee \neg x_3 \vee \neg x_5\}$
$\{\neg x_3 \vee \neg x_4 \vee x_5\}$	$\{\neg x_1 \vee x_3 \vee \neg x_4\}$	$\{x_1 \vee \neg x_5 \vee x_6\}$	$\{\neg x_1 \vee \neg x_2 \vee \neg x_3\}$
$\{x_2 \vee \neg x_4 \vee \neg x_5\}$	$\{\neg x_1 \vee x_5 \neg \vee x_6\}$	$\{x_2 \vee \neg x_3 \vee \neg x_4\}$	$\{\neg x_1 \vee \neg x_3 \vee \neg x_4\}$
$\{\neg x_2 \vee \neg x_5 \vee \neg x_6\}$	$\{x_1 \vee \neg x_2 \vee x_4\}$	$\{\neg x_3 \vee \neg x_4 \vee x_6\}$	$\{\neg x_1 \vee \neg x_2 \vee x_5\}$
$\{\neg x_1 \vee x_2 \vee \neg x_4\}$	$\{\neg x_2 \vee x_3 \vee \neg x_4\}$	$\{\neg x_1 \vee x_2 \vee x_3\}$	$\{x_2 \vee x_4 \vee \neg x_6\}$
$\{x_1 \vee x_2 \vee \neg x_3\}$	$\{\neg x_2 \vee x_3 \vee x_5\}$	$\{x_1 \vee x_4 \vee x_5\}$	$\{x_1 \vee \neg x_2 \vee \neg x_4\}$

Table A.1: An Instance Containing A Trap For GSAT

$\{\neg x_1 \vee x_2 \vee \neg x_6\}$	$\{x_2 \vee \neg x_5 \vee x_6\}$	$\{\neg x_2 \vee x_5 \vee \neg x_6\}$	$\{x_1 \vee \neg x_5 \vee \neg x_6\}$
$\{\neg x_2 \vee \neg x_3 \vee x_4\}$	$\{\neg x_2 \vee \neg x_4 \vee \neg x_6\}$	$\{\neg x_1 \vee x_2 \vee x_6\}$	$\{x_3 \vee \neg x_4 \vee \neg x_5\}$
$\{\neg x_2 \vee \neg x_4 \vee \neg x_5\}$	$\{\neg x_1 \vee \neg x_2 \vee x_5\}$	$\{\neg x_1 \vee \neg x_2 \vee \neg x_4\}$	$\{\neg x_2 \vee \neg x_3 \vee x_5\}$
$\{\neg x_1 \vee \neg x_2 \vee \neg x_3\}$	$\{\neg x_2 \vee \neg x_3 \vee \neg x_6\}$	$\{\neg x_2 \vee x_4 \vee x_5\}$	$\{x_1 \vee x_3 \vee x_4\}$
$\{x_1 \vee x_2 \vee x_5\}$	$\{\neg x_1 \vee x_2 \vee x_4\}$	$\{x_1 \vee \neg x_2 \vee x_3\}$	

Table A.2: An Instance Containing A Trap For Novelty

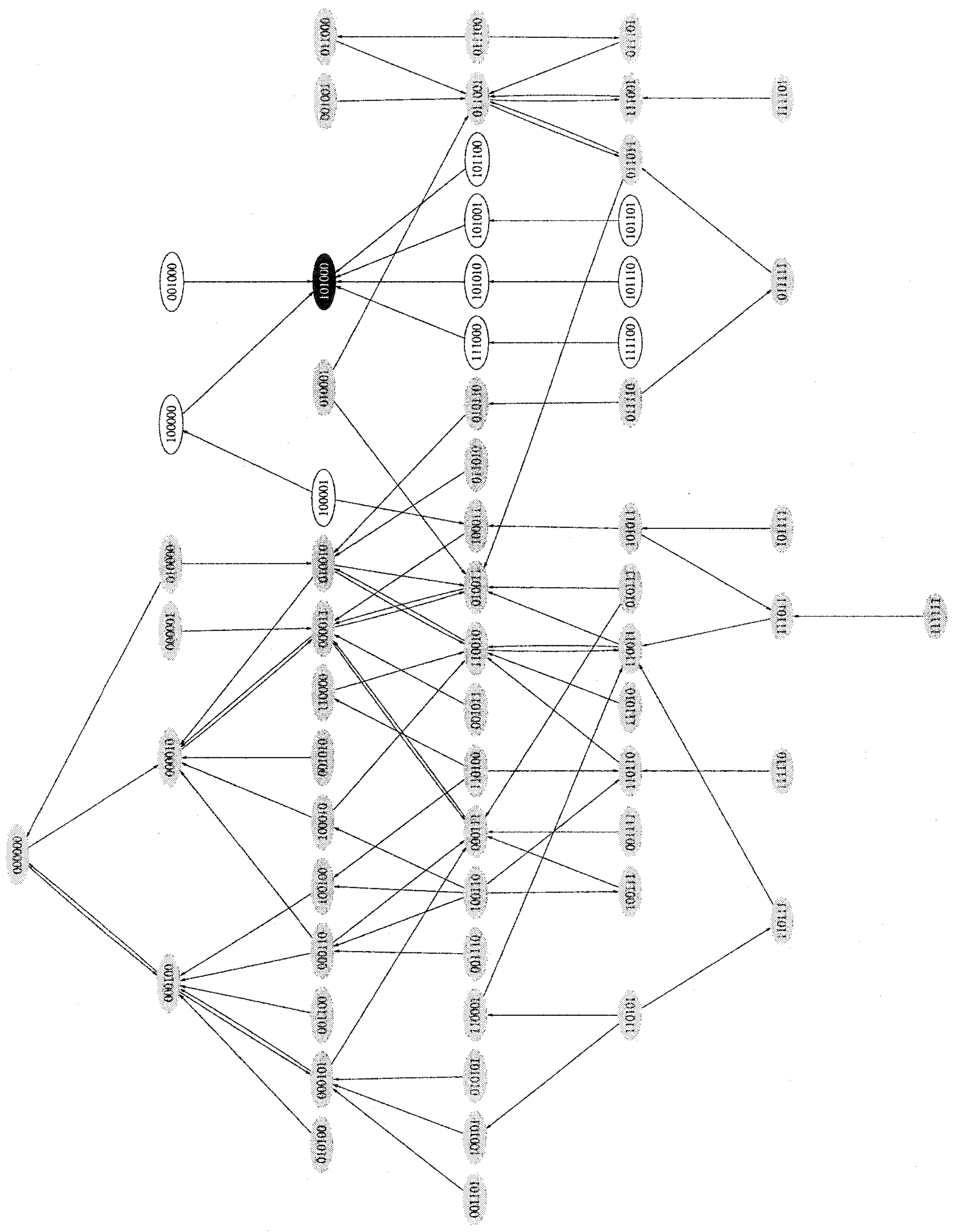

Figure A.1: An example of GSAT's Search Space Graph

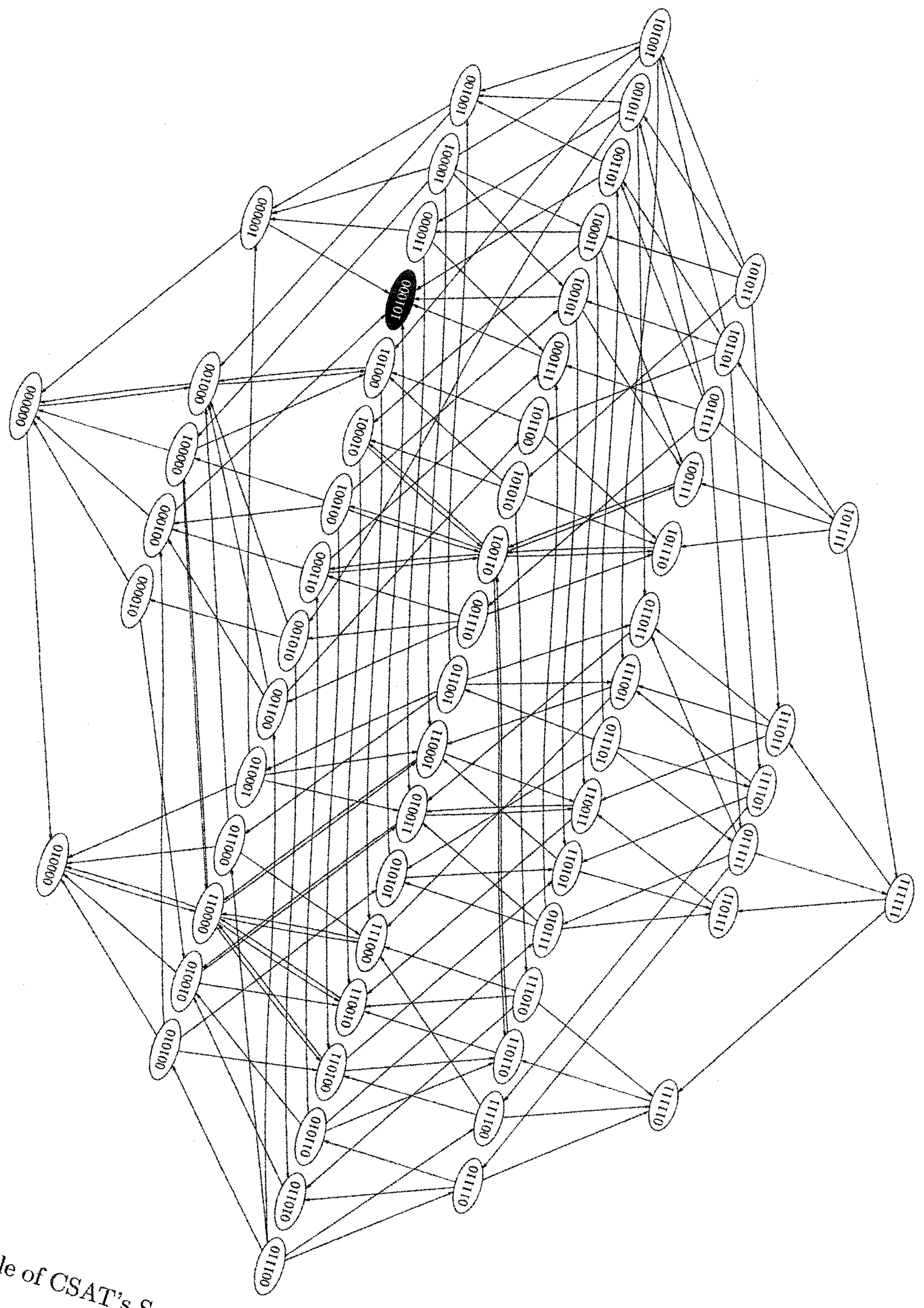

Figure A.2: An example of CSAT's Search Space Graph Under the Same Instance in Figure A.2

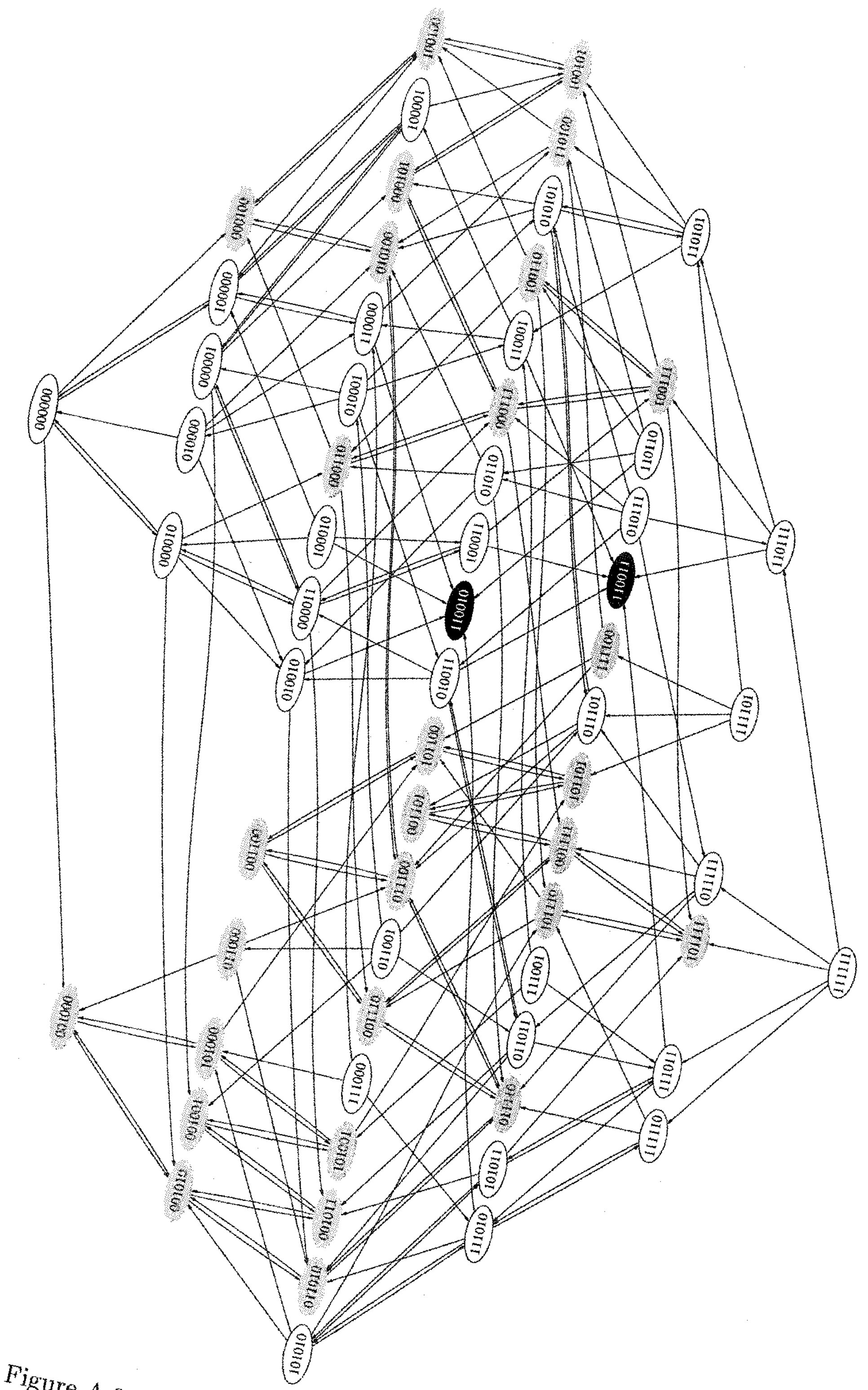

Figure A.3: An example of Novelty's Search Space Graph

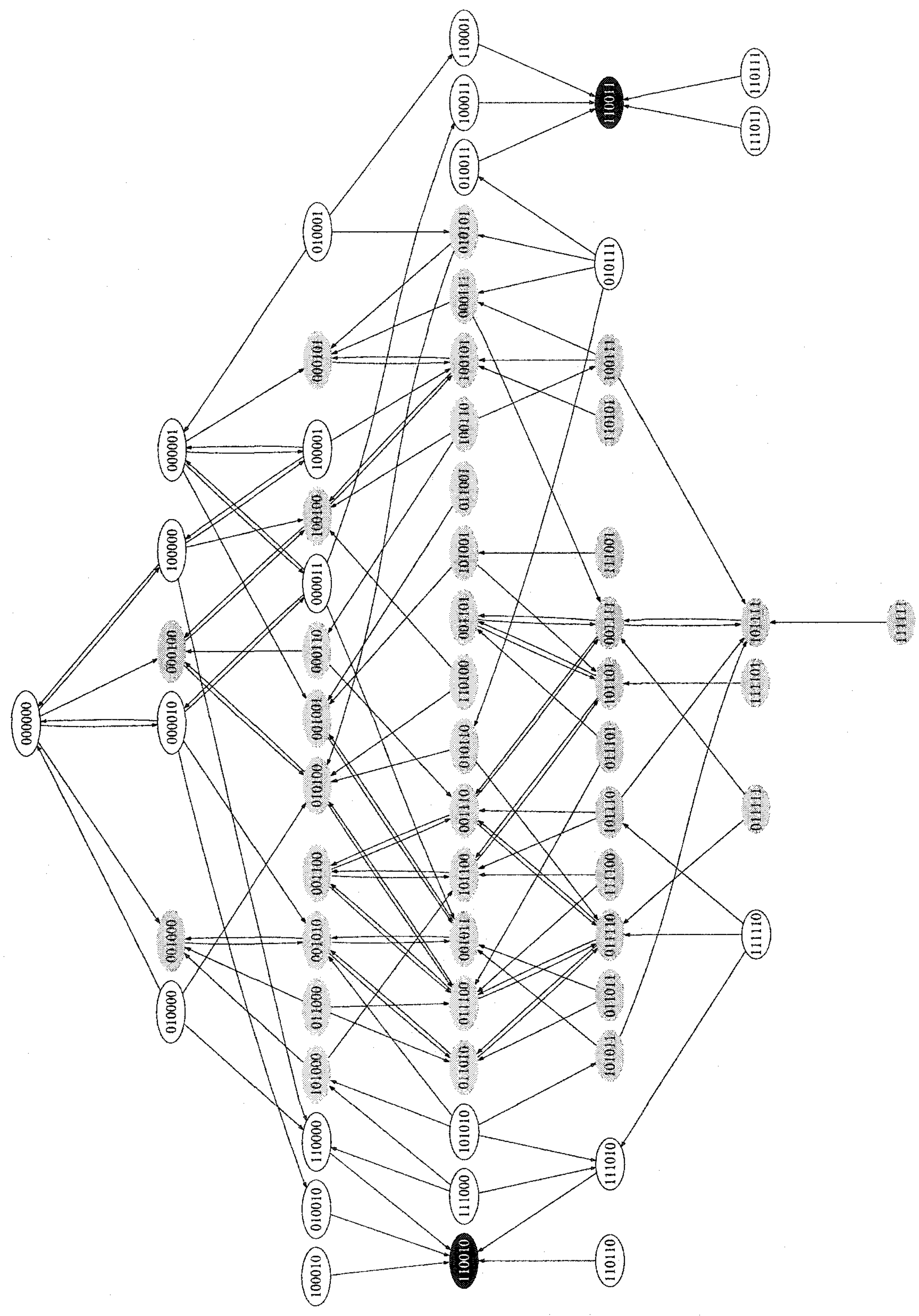

Figure A.4: An example of Only Best's Search Space Graph Under the Same Instance in Figure A.5

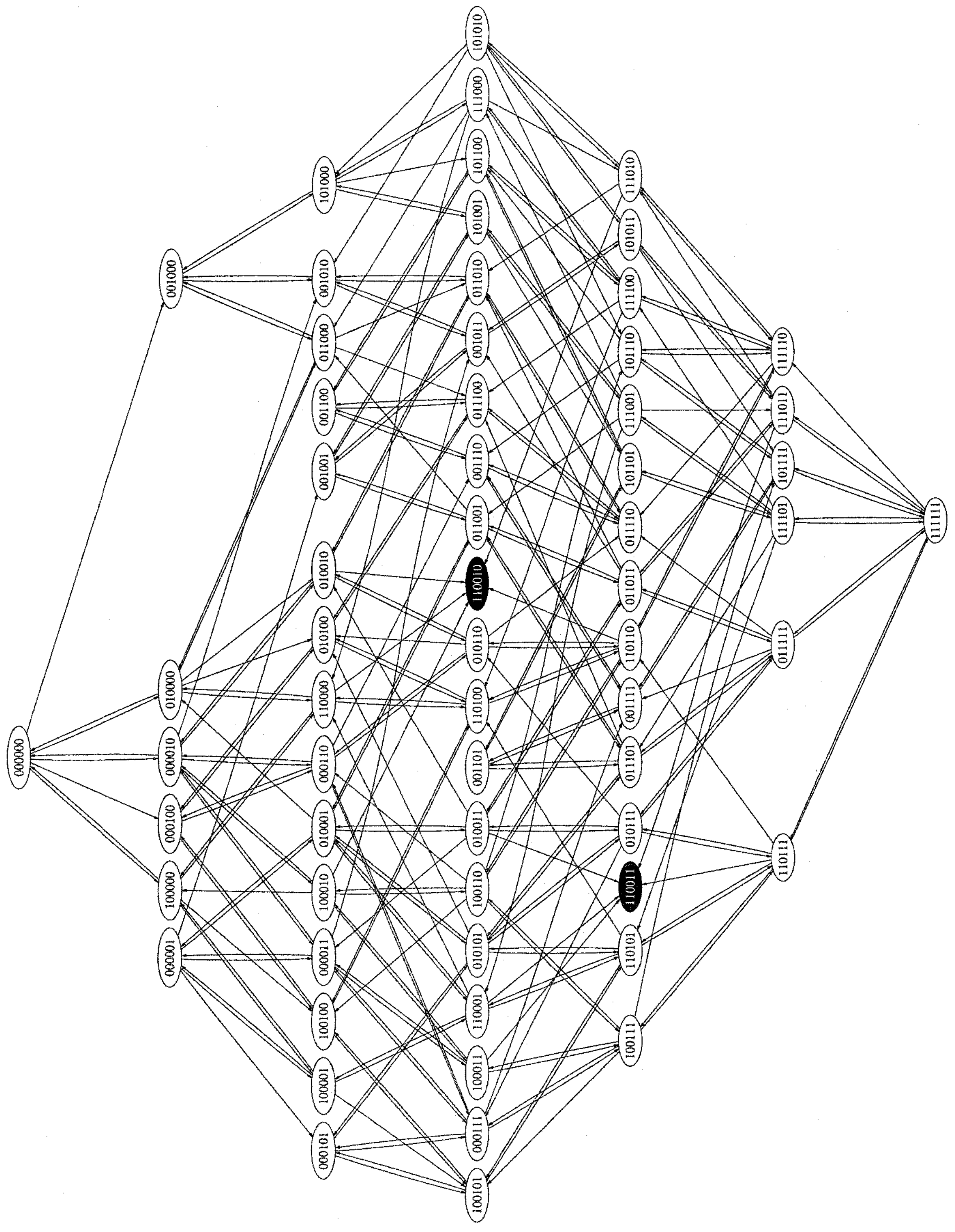

Figure A.5: An example of Basic WalkSAT's Search Space Graph Under the Same Instance in Figure A.5

Bibliography

[1] Paul Beame, Richard M. Karp, Toniann Pitassi, and Michael E. Saks. On the complexity of unsatisfiability proofs for random k -CNF formulas. In *ACM Symposium on Theory of Computing*, pages 561–571, 1998.

[2] Armin Biere. Special course on formal verification.

[3] M. Buro and H. Kleine Büning. Report on a SAT competition. *Bulletin of the European Association for Theoretical Computer Science*, 49:143–151, 1993.

[4] David A. Clark, Jeremy Frank, Ian P. Gent, Ewan MacIntyre, Neven Tomov, and Toby Walsh. Local search and the number of solutions. In *Principles and Practice of Constraint Programming*, pages 119–133, 1996.

[5] Stephen A. Cook and David G Mitchell. Finding hard instances of the satisfiability problem: A survey. In Du, Gu, and Pardalos, editors, *Satisfiability Problem: Theory and Applications*, volume 35, pages 1–17. American Mathematical Society, 1997.

[6] James M. Crawford and L. D. Anton. Experimental results on the crossover point in satisfiability problems. In Richard Fikes and Wendy Lehnert, editors, *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 21–27, Menlo Park, California, 1993. AAAI Press.

[7] J. Culberson, I. Gent, and H. Hoos. The probabilistic approximate completeness of WalkSAT for 2-SAT.

[8] Joseph Culberson. Graph coloring page.

[9] Joseph Culberson and Jonathan Lichtner. On searching a-ary hypercubes and related graphs. *On Searching A-ary Hypercubes and Related Graphs*, pages 263–290, 1996.

[10] Joseph C. Culberson. Mutation-crossover isomorphisms and the construction of discriminating functions. *Evolutionary Computation*, 2(3):279–311, 1994.

[11] D. A. Dawson. *Introduction to Markov chains.* Montral, Canadian Mathematical Congress, 1970.

[12] R. Dechter and D. Frost. Backtracking algorithms for constraint satisfaction problems; a survey, 1998.

[13] Joseph Culberson Dept. Hidden solutions, tell-tales, heuristics and anti-heuristics.

[14] Jeremy Frank, Peter Cheeseman, and John Stutz. When gravity fails: Local search topology. *Journal of Artificial Intelligence Research*, 7:249–281, 1997.

[15] Alex Fukunaga. Automated discovery of composite SAT variable-selection heuristics. 2002.

[16] Yong Gao. Threshode phenomena in nk landscapes. Master's thesis, University of Alberta, Edmonton, Alberta, Canada, 2001.

[17] Ian P. Gent and Toby Walsh. The enigma of SAT hill-climbing procedures. Technical Report TR 605, 1992.

[18] Ian P. Gent and Toby Walsh. Towards an understanding of hill-climbing procedures for SAT. In *National Conference on Artificial Intelligence*, pages 28–33, 1993.

[19] Ian P. Gent and Toby Walsh. The SAT phase transition. In *Proceedings of the Eleventh European Conference on Artificial Intelligence (ECAI'94)*, pages 105–109, 1994.

[20] Edward A. Hirsch. Hard formulas for SAT local search algorithms, 1998.

[21] Edward A. Hirsch. SAT local search algorithms: Worst-case study. *Journal of Automated Reasonning*, 24(1/2):127–143, 2000.

[22] Holger H. Hoos. *Stochastic Local Search - Methods, Models, Applications.* PhD thesis, TU Darmstadt, 1998.

[23] David S. Johnson, Cecilia R. Aragon, Lyle A. McGeoch, and Catherine Schevon. Optimization by simulated annealing: An experimental evaluation. part i, graph partitioning. *Oper. Res.*, 37(6):865–892, 1989.

[24] Henry A. Kautz and Bart Selman. Planning as satisfiability. In *Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI'92)*, pages 359–363, 1992.

[25] Grzegorz Kondrak and Peter van Beek. A theoretical evaluation of selected backtracking algorithms. In Chris Mellish, editor, *IJCAI'95: Proceedings International Joint Conference on Artificial Intelligence*, Montreal, 1995.

[26] Tracy Larrabee and Yumi Tsuji. Evidence for a satisfiability threshold for random 3CNF formulas. Technical Report UCSC-CRL-92-42, 1992.

[27] David McAllester, Bart Selman, and Henry Kautz. Evidence for invariants in local search. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97)*, pages 321–326, Providence, Rhode Island, 1997.

[28] David G. Mitchell and Hector J. Levesque. Some pitfalls for experimenters with random SAT. *Artificial Intelligence*, 81(1-2):111–125, 1996.

[29] David G. Mitchell, Bart Selman, and Hector J. Levesque. Hard and easy distributions for SAT problems. In Paul Rosenbloom and Peter Szolovits, editors, *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 459–465, Menlo Park, California, 1992. AAAI Press.

[30] Andrew J. Parkes. Clustering at the phase transition. In *AAAI/IAAI*, pages 340–345, 1997.

[31] V.S. Ananda Rangan. Backbone guided local search techniques for solving satisfiability problems. Master's thesis, Washington University, 2002.

[32] http://WWW.SATLIB.ORG.

[33] http://WWW.SATLIVE.ORG.

[34] Dale Schuurmans and Finnegan Southey. Local search characteristics of incomplete SAT procedures. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence*, 2000.

[35] Dale Schuurmans and Finnegan Southey. Local search characteristics of incomplete SAT procedures. *Artificial Intelligence*, 132(2):121–150, 2001.

[36] Bart Selman and Henry A. Kautz. Domain-independent extensions to GSAT: solving large structured satisfiability problems. In *Proceedings of theInternational Joint Conference on Artificial Intelligence(IJCAI-93)*, Chambry, France, 1993.

[37] Bart Selman, Henry A. Kautz, and Bram Cohen. Local search strategies for satisfiability testing. In Michael Trick and David Stifler Johnson, editors, *Proceedings of the Second DIMACS Challange on Cliques, Coloring, and Satisfiability*, Providence RI, 1993.

[38] Bart Selman, Henry A. Kautz, and Bram Cohen. Noise strategies for improving local search. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI'94)*, pages 337–343, Seattle, 1994.

[39] Bart Selman, Hector J. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In Paul Rosenbloom and Peter Szolovits, editors, *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 440–446, Menlo Park, California, 1992. AAAI Press.

[40] Josh Singer, Ian Gent, and Alan Smaill. Backbone fragility and the local search cost peak. *Journal of Artificial Intelligence Research*, 12:235–270, 2000.

[41] Hantao Zhang. Finite model generation page.