

Chapter 8

Satisfiability and Randomization

The world consists of the tension between order and chaos. When simulating physical phenomena, order is supplied by the laws of physics and chaos is supplied by random numbers.

Gregory Chaitin

With increasing temperature we see a succession of phase transitions for water in which its properties change dramatically: the solid phase – ice – melts to the liquid phase – water – and then eventually boils to the gaseous phase – steam.

Cambridge Cosmology home page

Contents

8.1 Randomly Generated SAT Problems	104
8.2 Sharp Phase Transition	105
8.3 Randomized Algorithms for SAT	107
8.3.1 Local Search	110
8.3.2 Random Walk	111
8.4 Randomly Generated non-Clausal Problems	114
8.5 Literature	114
Exercises	114

In this chapter we consider two topics related to satisfiability-checking and randomization. Section 8.1 discusses how one can generate random satisfiability problems. In Section 8.2 we show that randomly generated satisfiability problems expose a phenomenon of *sharp phase transition*: a change from satisfiable to unsatisfiable problems happens in a very narrow region, eventually degenerating into a point. In addition to having sharp phase transition, randomly generated problems turn out to be extremely hard for existing satisfiability checkers.

In Section 8.3 we show that randomization not only creates hard satisfiability problems but also helps in solving them. We present several randomized algorithms for propositional satisfiability based on the idea of random generation of interpretations.

8.1 Randomly Generated SAT Problems

To define a model for random generation of SAT problems, we have to understand what is a random clause. A clause is a disjunction of literals $L_1 \vee \dots \vee L_k$. We cannot generate all clauses randomly with an equal probability since the length k of clauses is unbounded. Let us fix $k \geq 1$ and define what is a random k -clause. Again, we cannot generate k -clauses randomly with an equal probability, since the number of variables is unbounded too. Let us also fix the number of variables n and use a fixed set of variables $P = \{p_1, \dots, p_n\}$.

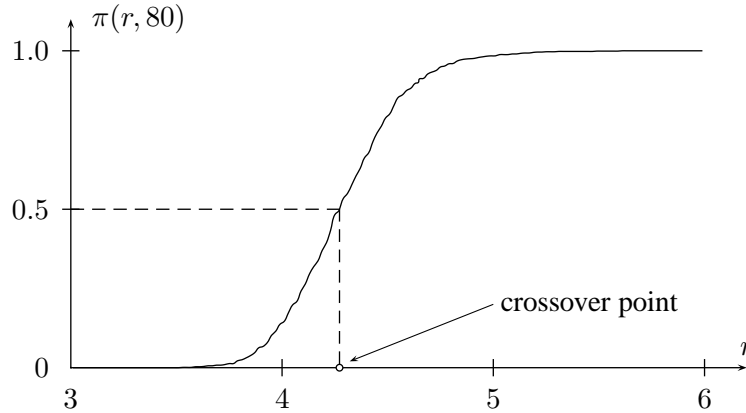
To generate such a k -clause with variables in P randomly, we have to generate k literals randomly, with an equal probability and independently. We can choose among $2n$ literals $p_1, \dots, p_n, \neg p_1, \dots, \neg p_n$. This suggests the following model for generating a random k -clause with variables in P : randomly generate k literals (not necessarily different) among $p_1, \dots, p_n, \neg p_1, \dots, \neg p_n$ so that each literal is chosen with equal probability $\frac{1}{2n}$.

Suppose that using our model of random clause we generate m random clauses C_1, \dots, C_m . What is the probability that the set $\{C_1, \dots, C_m\}$ is unsatisfiable?

Evidently, when $m = 1$, the set is satisfiable, so the probability is 0. When $m = 2$, then we can obtain an unsatisfiable set, but every such set consists of two clauses $p_i \vee \dots \vee p_i$ and $\neg p_i \vee \dots \vee \neg p_i$. It is not hard to argue that the probability of generating a set of clauses of this form is $\frac{1}{(2n)^{2k-1}}$, which is a very small number even for small values of k and n . However, when we generate a sufficiently large number of clauses, the probability can become as close to 1 as we want.

We can say that for small m the set of clauses is *under-constrained*: due to a small number of constraints on the possible values of p_i the set is easy to satisfy. When we generate a larger number of clauses, the probability of obtaining an unsatisfiable set grows. Finally, when we generate a very large number of clauses, the problems become *over-constrained* and hence difficult to satisfy, so the probability becomes close to 1. For each n there exists a *crossover point*, i.e., the number m where the probability changes from a number less than $\frac{1}{2}$ to a number greater than or equal to $\frac{1}{2}$. What is the distribution of probability for different values of m , and in particular what is the behavior of this probability near the crossover point? Experiments conducted in [Mitchell, Selman and Levesque 1992] have shown an interesting behavior of the probability: for large n the change from a nearly-zero probability to a nearly-one probability happens in a very narrow region around the crossover point.

From now on we will fix k to be 3, that is, we will consider randomly generated 3-SAT problems. Let us define several notions. First of all, we explain the model for the random generation of clauses. We will randomly generate 3-clauses in a signature with n boolean variables, where n can vary, but the number of clauses per variable will be fixed. This real number will be called a *ratio* and denoted by r . Denote by $\pi(r, n)$ the probability that

Figure 8.1: Function $\pi(r, 80)$ and the crossover point

a randomly generated set of $[rn]$ 3-clauses¹ of variables p_1, \dots, p_n is unsatisfiable. It is easy to see that $\pi(r_1, n) \geq \pi(r_2, n)$ if $r_1 \geq r_2$, since the more clauses we generate, the more likely we are to obtain an unsatisfiable set. The experimentally obtained value of the probability function for $n = 80$ is given in Figure 8.1.²

For every fixed n , we call the *crossover point* the number r such that for all $r' < r$ we have $\pi(r', n) < 0.5$ and for all $r' > r$ we have $\pi(r', n) \geq 0.5$. Since the probability function π is monotone in its first argument, there exists a unique crossover point. The crossover point is illustrated in Figure 8.1. Denote the crossover point by $crossover(n)$, that is,

$$crossover(n) = r \Leftrightarrow \pi(r, n) \geq 0.5 \text{ and} \\ \text{for all } r' < r \text{ we have } \pi(r', n) < 0.5.$$

8.2 Sharp Phase Transition

Figure 8.1 shows that for values of r smaller than (say) 3.5 the value $\pi(r, 80)$ is “almost 0”. Likewise, for values of r greater than 6 the value $\pi(r, 80)$ is “almost 1”. We are interested in the region where the probability changes from “almost 0” to “almost 1”. To formalize the notions of “almost 0” and “almost 1” we choose a (small) value ϵ and consider values smaller than ϵ as “almost 0” and values greater than $1 - \epsilon$ as “almost 1”.

Let ϵ be a real number such that $0 < \epsilon < 0.5$. We call the ϵ -window the set of all numbers r' such that $\epsilon \leq \pi(r', n) \leq 1 - \epsilon$. Essentially, the ϵ -window is the region where the probability changes from “almost 0” to “almost 1”. It follows from the monotonicity

¹Here $[rn]$ denotes the integer closest to rn .

²All graphs in this chapter are obtained experimentally by running the system SATO [Zhang 1997]. To obtain trustworthy results, we ran SATO on problems obtained by incrementing the values of the ratios r by 0.02 from 3 to 6. For each value of the ratio we used 1000 randomly generated problems.

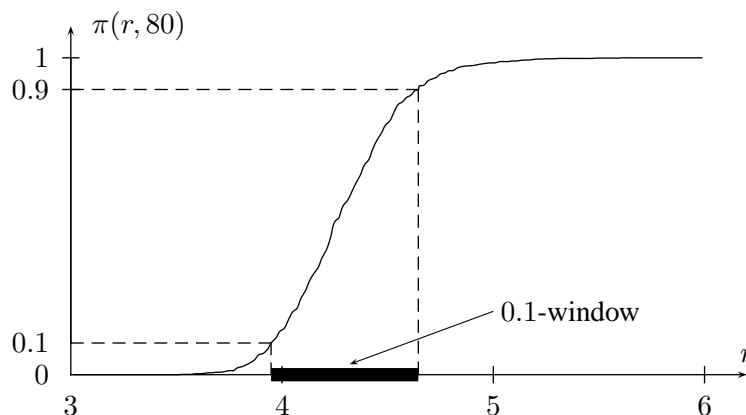


Figure 8.2: ϵ -window for $\epsilon = 0.1$ and $n = 80$

of π that the ϵ -window is an interval, and moreover if this interval is non-empty, then it contains the crossover point. The ϵ -window is illustrated in Figure 8.2.

Denote by $window(\epsilon, n)$ the width of the window. Experiments with the randomly generated 3-SAT problems lead to a conjecture that for every ϵ we have

$$\lim_{n \rightarrow \infty} \left(\frac{window(\epsilon, n)}{crossover(n)} \right) = 0. \quad (8.1)$$

This means that the ϵ -window shrinks for larger values of n and eventually degenerates into a point. This effect is called the *scaling window effect* and is illustrated in Figure 8.3.

If the ϵ -window degenerates into a point, the curve $\pi(r, n)$ becomes very steep in the region of the crossover point, and in the limit becomes a step function having value 0 to the left of the crossover point and value 1 to the right of the crossover point. This effect is called *sharp phase transition* or *sharp threshold*.

It has been recently proved [Friedgut 1999] that (8.1) holds, and therefore the sharp phase transition takes place. Experiments suggest that the crossover point is close to 4.25, but its exact value is unknown. Figure 8.4 shows the behavior of the probability function $\pi(r, n)$ for different values of n . One can see from it that the curve depicting this function is becoming steeper around the crossover point for larger values of n .

It is interesting to observe the behavior of the state-of-the-art satisfiability checkers on randomly generated problems. It turns out that the problems generated with a ratio near the crossover point are hard for all existing systems. For $r = 4.25$ the fastest systems can solve within one hour only problems with $n \approx 500$ on a single processor computer. The problems generated using the ratio r much smaller than the crossover point are relatively easy to solve. The same holds for the problems generated using the ratio r much greater than the crossover point. Figure 8.5 illustrates the number of branches exploited by the DLL algorithm as implemented in the system SATO for various values of r .

The distribution of the difficulty of problems depending on r forms the so-called *easy-*

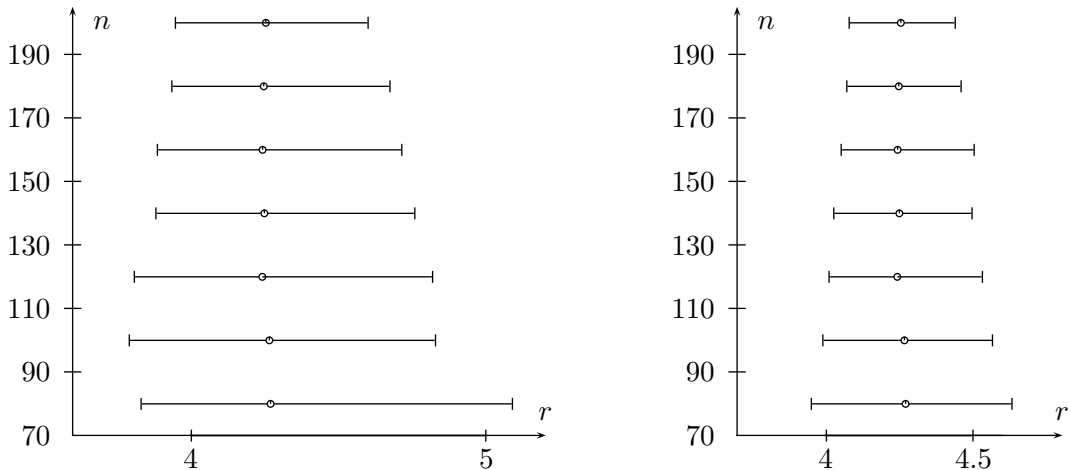


Figure 8.3: Scaling window: 0.01-window and 0.1-window for $n = 80 \dots 200$. The crossover point is marked by a small circle.

hard-easy pattern: under-constrained problems are easy, problems near the crossover point are hard, and the over-constrained problems are again easy. The randomly generated k -SAT problems near the crossover point are sometimes called *hard random k -SAT problems*.

Randomly generated k -SAT problems for $k > 3$ expose the same pattern of behavior as randomly generated 3-SAT problems. Namely, both the sharp phase transition and the easy-hard-easy pattern have been experimentally observed. The crossover point is different, for example for 4-SAT the crossover point is believed to be around 9.8.

8.3 Randomized Algorithms for SAT

Although the best DLL-based systems can only establish unsatisfiability of hard random 3-SAT problems with $n \approx 500$ within one hour, there exist methods which allow one to find models of such problems with over 2000 variables. Interestingly, these methods cannot be used for establishing unsatisfiability, because they are *incomplete*. *Complete algorithms* always give a *yes-no* answer. *Incomplete algorithms* sometimes give the “don’t know” answer.

There is a subtle difference between establishing satisfiability and unsatisfiability. To establish that a formula A is satisfiable, it is enough to show *one* model of this formula. The model is a mapping from variables of A to boolean values, so this model has a very compact representation: if A has n different variables p_1, \dots, p_n , then one can use n bits to represent this mapping. Given an interpretation I , one can easily check whether $I \models A$ by evaluating A in I . So if we have a clever heuristics which “guesses” interpretations I which

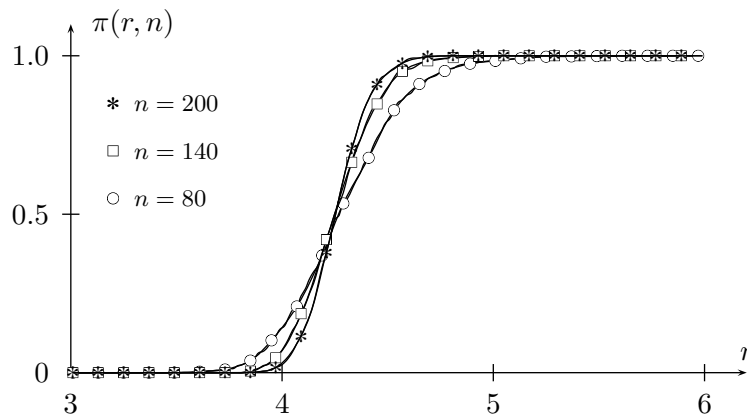


Figure 8.4: Sharp phase transition: the values of $\pi(r, n)$ for $n = 80, 140, 200$

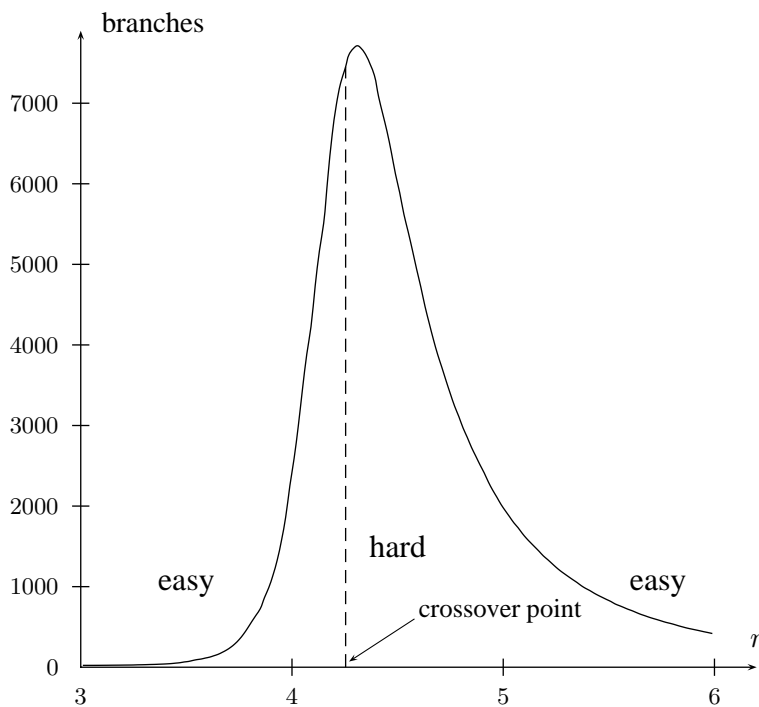


Figure 8.5: The easy-hard-easy pattern: the number of branches for $n = 200$

```

procedure CHAOS(S)
input: set of clauses S
output: interpretation I such that  $I \models S$  or don't know
parameters: positive integer MAX-TRIES
begin
  repeat MAX-TRIES times
    I := random interpretation
    if  $I \models S$  then return I
  return don't know
end

```

Figure 8.6: The CHAOS algorithm

can potentially satisfy A , and we are lucky, we can guess a model of I without applying the standard satisfiability-checking algorithms such as DLL.

Let us illustrate the difference between satisfiability and unsatisfiability by an example. First, note that we can generate a random interpretation I for variables p_1, \dots, p_n by choosing a random n -bit string $b_1 \dots b_n$ and defining I to be $\{p_1 \mapsto b_1, \dots, p_n \mapsto b_n\}$. The algorithm *CHAOS* is given in Figure 8.6. Essentially, this algorithm generates a completely random interpretation a fixed number of times. If one of the generated interpretation is a model, it terminates and returns “satisfiable”. Otherwise, it returns “don’t know”.

This algorithm can be used for checking satisfiability but not for checking unsatisfiability. Indeed, even we generate a very large of interpretations there is still a non-zero probability that we have not generated all possible interpretations.

On the contrary, there is no straightforward way to establish unsatisfiability by guessing models. Indeed, to show that a formula A is unsatisfiable, we have to show that A is false in *every* interpretation. If A has n different variables, then there are 2^n possible interpretations of the signature of A .

This phenomenon is common for all NP-complete problems. Problems in NP have *polynomially short witnesses*. Intuitively, a witness for an instance i of a decision problem is a string s such that, given the pair (s, i) , one can check in polynomial time that i has the yes-answer. For propositional satisfiability we can take as a witness the bit-string representing a model of this formula. Unsatisfiability is coNP-complete, and coNP-complete problems do not have polynomially short witnesses, unless $\text{NP}=\text{coNP}$, the equation generally believed to be false.

If a problem has a short witness, we can try to solve instances of this problem by guessing a possible witness, and then checking that the witness really witnesses the instance. We terminate if a witness was found or if some number of guesses was done. In this way we obtain incomplete algorithms, since termination without finding a witness does not mean that the instance has no witness at all. So if a witness is found, then the answer is “yes”, but

```

procedure GSAT(S)
input: set of clauses S
output: interpretation I such that  $I \models S$  or don't know
parameters: positive integers MAX-TRIES, MAX-FLIPS
begin
  repeat MAX-TRIES times
    I := random interpretation
    if  $I \models S$  then return I
    repeat MAX-FLIPS times
      p := a variable such that flip(I, p) satisfies
                                     the maximal number of clauses in S
      I = flip(I, p)
      if  $I \models S$  then return I
  return don't know
end

```

Figure 8.7: The GSAT algorithm

if it is not found, then we can only answer “don’t know”.

There are several incomplete algorithms for checking propositional satisfiability. All these algorithms use random numbers and are based on the following general idea. First, choose a random interpretation. If this interpretation is not a model, repeatedly choose a variable and change its value in the interpretation (*flip* the variable). The variables whose values are changed are chosen using heuristics or randomly, or both. To formalize this idea, let us introduce an operator *flip* on interpretations as follows:

$$\text{flip}(I, p)(q) = \begin{cases} I(q), & \text{if } p \neq q; \\ 1, & \text{if } p = q \text{ and } I(p) = 0; \\ 0, & \text{if } p = q \text{ and } I(p) = 1. \end{cases}$$

In other words, the interpretation *flip*(*I*, *p*) is obtained from *I* by changing its value on *p*.

8.3.1 Local Search

A *local search* procedure is based on the idea of minimizing the number of unsatisfied clauses. This procedure was described in [Selman, Levesque and Mitchell 1992] and is known as *GSAT*. Given an interpretation *I* which does not satisfy a set *S* of clauses, we flip a variable in *I* which results in a maximal number of satisfied clauses. If there are several variables giving the same number of satisfied clauses, we choose one of them at random. The *GSAT algorithm* is given in Figure 8.7.

EXAMPLE 8.1 (GSAT) Consider the following set of clauses:

$$p_1 \vee \neg p_2 \vee p_3, \quad \neg p_2 \vee \neg p_3, \quad \neg p_1 \vee \neg p_3, \quad \neg p_1 \vee p_2, \quad p_1 \vee p_2.$$

This set of clauses is satisfiable. Suppose that the initial random interpretation I is $\{p_1 \mapsto 0, p_2 \mapsto 0, p_3 \mapsto 1\}$. A possible GSAT run with this initial interpretation is illustrated in the following table.

flip no.	interpretation			satisfied clauses			candidates for flipping	flipped variable	
	p_1	p_2	p_3	p_1	p_2	p_3			
1	0	0	1	4	3	4	4	p_2, p_3	p_2
2	0	1	1	4	3	4	4	p_2, p_3	p_3
3	0	1	0	4	5	4	4	p_1	p_1
	1	1	0	5					

The first column contains the flip number. Columns 2–4 show the interpretation before the flip. The next four columns show the number of satisfied clauses: column 5 gives the number before the flip and the following three columns the number if we flipped one of the variables. The last column gives the variable chosen for flipping.

In this example, initially 4 clauses are satisfied. If we flip p_1 , then the number of satisfied clauses will become 3. If we flip p_2 or p_3 , then the number of satisfied clauses will become 4. So GSAT randomly flips one of the variables p_2, p_3 . Assuming that p_2 is chosen, we obtain the interpretation shown in line 2. For the second flip the procedure could have chosen for flipping either p_2 or p_3 because both flips result in a maximal number of satisfied clauses (namely, 4). For the third flip the choice is unique: only p_1 can be flipped. After three flips the procedure terminates since all clauses are satisfied. \square

For local search procedures, there is a possibility that the procedure goes to a wrong direction and gets stuck in a local optimum point: a point where further flips do not increase the number of satisfied clauses. Even worse, local optima often occur in a *plateau*: a set of points where the number of satisfied clauses does not change. GSAT uses a radical way to escape from local optima: when the limit *MAX-FLIPS* on the number of flips is reached, the current interpretation is discarded and a random interpretation is chosen again.

8.3.2 Random Walk

The problem of escaping local optima or plateaus can be attacked by further randomizing the GSAT algorithm. Instead of always choosing the best possible move (resulting in the largest number of satisfied clauses), with some probability π we flip a random variable p . Such flips are called *sideways moves*. It was experimentally discovered that choosing p randomly among *all* variables is not a very good idea. The best strategy known so far is to flip a variable which occurs in at least one unsatisfied clause.

This gives a modification of GSAT, called *GSAT with random walks*. This algorithm is shown in Figure 8.8. Random walks are reported to boost the performance of GSAT drastically. For example, on the hard 3-SAT problems GSAT with random walks is able to solve

```

procedure GSAT(S)
input: set of clauses S
output: interpretation I such that  $I \models S$  or don't know
parameters: positive integers MAX-TRIES, MAX-FLIPS
                real number  $0 \leq \mu \leq 1$  (probability of a sideways move),
begin
  repeat MAX-TRIES times
    I := random interpretation
    if  $I \models S$  then return I
    repeat MAX-FLIPS times
      with probability  $\mu$ 
        p := a variable such that flip(I, p) satisfies
                the maximal number of clauses in S
      with probability  $1 - \mu$ 
        randomly choose p among variables occurring in clauses false in I
        I = flip(I, p)
      if  $I \models S$  then return I
    return don't know
end

```

Figure 8.8: GSAT with random walks

in a few minutes problems with 2000 variables, which is far beyond the capabilities of the complete methods, such as DLL. GSAT with random walks also shows good performance on some structured, non-random, problems.

Note that when $\mu = 1$, GSAT with random walks behaves exactly like GSAT. When μ is small almost all the moves are sideways moves. It is also possible to design an efficient algorithm which is entirely based on sideways moves. This an algorithm is called *WSAT* and given in Figure 8.9.

EXAMPLE 8.2 (WSAT) Consider the set of clauses of Example 8.1. Suppose that the initial random interpretation *I* is again $\{p_1 \mapsto 0, p_2 \mapsto 0, p_3 \mapsto 1\}$. A possible *WSAT* run with this initial interpretation is illustrated in the following table.

```

procedure WSAT(S)
input: set of clauses S
output: interpretation I such that  $I \models S$  or don't know
parameters: positive integers MAX-TRIES, MAX-FLIPS
begin
  repeat MAX-TRIES times
    I := random interpretation
    if  $I \models S$  then return I
    repeat MAX-FLIPS times
      randomly choose a clause  $C \in S$  such that  $I \not\models C$ 
      randomly choose a variable p in C
       $I = flip(I, p)$ 
      if  $I \models S$  then return I
  return don't know
end

```

Figure 8.9: The WSAT algorithm

flip no.	interpretation			unsatisfied clauses	candidates for flipping	flipped variable
	p_1	p_2	p_3			
1	0	0	1	$p_1 \vee p_2$	p_1, p_2	p_1
2	1	0	1	$\neg p_1 \vee \neg p_3$ $\neg p_1 \vee p_2$	p_1, p_2, p_3	p_2
3	1	1	1	$\neg p_2 \vee \neg p_3$ $\neg p_1 \vee \neg p_3$	p_1, p_2, p_3	p_3
	1	1	0			

Note that there is an essential difference between the behaviors of GSAT and WSAT on the clauses of this example. For flip 1 GSAT had p_2, p_3 as the candidates for flipping, while WSAT had p_1, p_2 . The variable p_3 was not a candidate for WSAT because it does not occur in any unsatisfied clause. For flips 2 and 3 WSAT could have chosen any variable. For flip 2, the probability of choosing p_1 is $\frac{1}{2}$, while the probabilities of choosing either p_2 or p_3 are only $\frac{1}{4}$. The variable p_1 had a higher probability for flipping since it occurs in both unsatisfied clauses. Likewise, for flip 3, p_3 is the most likely candidate because it also occurs in both unsatisfied clauses. \square

WSAT shows a very good performance on many problems, but it is surprisingly not so good on hard randomly generated 3-SAT problems.

Note that there is an essential difference between GSAT with random walks and $\mu = 0$ and WSAT. GSAT with random walks chooses a random variable among all variables occurring in unsatisfiable clauses. WSAT first chooses a random unsatisfied clause and

then chooses a variable in it. If a variable occurs in many unsatisfiable clauses, then WSAT is more likely to choose it compared to GSAT with random walks.

WSAT behaves very well on some classes of problems, and may even clearly outperform GSAT with random walks. However, on hard random 3-SAT problems WSAT does not perform well.

If an incomplete algorithm for propositional satisfiability terminates with a “don’t know” answer, it signals that either the problem is too hard or there is an evidence that the problem is unsatisfiable. There are theoretical results showing that a don’t know answer can be used to establish unsatisfiability with a high probability [Schöning 1999], but the bounds on the number of tries proved in these papers are not very useful in practice.

8.4 Randomly Generated non-Clausal Problems

8.5 Literature

The random clause generation model used here was described in Mitchell et al. [1992]. This paper also observes the sharp phase transition effect and the easy-hard-easy pattern.

The GSAT local search procedure was described in [Selman et al. 1992]. Independently, a similar procedure was discovered by [Gu 1992]. GSAT with random walks is described in [Selman, Kautz and Cohen 1994].

Exercises

EXERCISE 8.1 Explain the following terms:

- (1) under- and over-constrained problems;
- (2) crossover point;
- (3) ϵ -window;
- (4) scaling window effect;
- (5) sharp phase transition;
- (6) easy-hard-easy pattern;
- (7) hard random k -SAT problem;
- (8) sideways move. □

EXERCISE 8.2 ★★ A student decides to change the model of random generation of clauses as follows. Choose n real numbers r_1, \dots, r_n such that $r_i \neq 0.5$ for all i . Then generate 3-clauses by choosing independently 3 literals as follows. First, randomly choose a variable p_i among p_1, \dots, p_n with equal probability $1/n$. Then, choose the literal p_i with probability r_i and the literal $\neg p_i$ with probability $1 - r_i$. What is your opinion about the following questions:

- (1) Will the crossover point be greater than, equal to, or less than the crossover point for the hard random 3-SAT problems?
- (2) Does this model of random generation result in harder, same complexity, or easier problems around the crossover point as compared to the hard random 3-SAT problems
 - (a) for GSAT and WSAT?
 - (b) for DLL?
- (3) Does this model of random generation have a sharp phase transition?

Note that mathematically these questions may be very hard, so simply try to use your intuition to answer them. Of course, you can make extensive experiments to confirm your answers. \square

EXERCISE 8.3 Implement a program for random generation of k -clauses. Modify it so that it generates k -clauses with some given probability π and $k + 1$ -clauses with the probability $1 - \pi$. \square

EXERCISE 8.4 Implement a computer program able to solve the random 3-SAT problems with 430 clauses and 100 variables in 1 hour. \square

EXERCISE 8.5 Implement a computer program able to solve the random 3-SAT problems with 430 clauses and 100 variables in 1 minute. \square

EXERCISE 8.6 (★) Implement a computer program able to solve the random 3-SAT problems with 430 clauses and 100 variables in 1 second. \square

EXERCISE 8.7 (★★) Implement a computer program able to solve the random 3-SAT problems with 430 clauses and 100 variables in 0.1 second. \square

EXERCISE 8.8 (See Examples 8.1 and 8.2.) Consider the set consisting of the following clauses:

$$\neg p_0 \vee \neg p_1 \vee \neg p_2, \quad p_0 \vee \neg p_2, \quad \neg p_0 \vee p_1, \quad p_1 \vee p_2, \quad \neg p_0 \vee \neg p_1 \vee p_2.$$

- (1) Show how GSAT can find a model of this set starting with the initial random interpretation $\{p_0 \mapsto 1, p_1 \mapsto 0, p_2 \mapsto 1\}$.
- (2) Answer the same question, but for WSAT instead of GSAT. \square

EXERCISE 8.9 Answer the same question as in Exercise 8.8 but for the following sets of clauses and interpretations:

- (1) $\neg p_3 \vee p_4, \neg p_4 \vee \neg p_3, p_3 \vee \neg p_4, \neg p_2 \vee \neg p_1, p_2 \vee p_3, p_1 \vee p_2$ and $\{p_1 \mapsto 1, p_2 \mapsto 0, p_3 \mapsto 1, p_4 \mapsto 1\}$.
- (2) $p_1 \vee p_3, p_3 \vee \neg p_1, p_4 \vee \neg p_1, \neg p_4 \vee \neg p_1, p_4 \vee p_1, \neg p_2 \vee p_4, \neg p_2 \vee \neg p_4, p_3 \vee p_4, \neg p_4 \vee p_3$ and $\{p_1 \mapsto 1, p_2 \mapsto 1, p_3 \mapsto 0, p_4 \mapsto 0\}$.
- (3) $\neg p_1 \vee p_2, p_1 \vee p_3, \neg p_3 \vee p_1, p_1 \vee p_4, \neg p_3 \vee p_4 \vee \neg p_1, \neg p_3 \vee \neg p_2, p_2 \vee p_4, p_3 \vee \neg p_4, p_3 \vee p_4$ and $\{p_1 \mapsto 0, p_2 \mapsto 0, p_3 \mapsto 0, p_4 \mapsto 0\}$. \square

EXERCISE 8.10 (See Example 8.2.) Consider the set consisting of the following clauses:

$$\begin{array}{cccc}
 p_0 \vee \neg p_1 \vee p_2 & p_0 \vee \neg p_1 \vee p_2 \vee p_4 & \neg p_0 \vee p_1 \vee \neg p_2 & \neg p_0 \vee \neg p_1 \vee \neg p_2 \vee \neg p_4 \\
 p_0 \vee \neg p_1 \vee p_4 & p_3 \vee p_2 \vee p_4 \vee \neg p_0 & \neg p_2 \vee \neg p_2 \vee p_4 \vee p_3 & \neg p_2 \vee \neg p_0 \vee p_4 \vee p_4 \\
 p_0 \vee p_3 \vee \neg p_4 & p_0 \vee \neg p_1 \vee \neg p_2 \vee \neg p_3 & \neg p_1 \vee \neg p_2 \vee \neg p_3 & p_1 \vee \neg p_2 \vee \neg p_3 \vee \neg p_4 \\
 p_1 \vee p_2 & p_2 \vee p_3 \vee \neg p_4 \vee p_3 & \neg p_0 \vee \neg p_2 \vee \neg p_3 \vee \neg p_4 & p_0 \vee p_2 \vee p_4
 \end{array}$$

For each of the variables p_0, p_1, p_2, p_3, p_4 find the probability that WSAT will choose this variable for flipping

- (1) when the current interpretation is $\{p_0 \mapsto 0, p_1 \mapsto 0, p_2 \mapsto 0, p_3 \mapsto 0, p_4 \mapsto 0\}$.
- (2) when the current interpretation is $\{p_0 \mapsto 1, p_1 \mapsto 1, p_2 \mapsto 1, p_3 \mapsto 1, p_4 \mapsto 1\}$. □

EXERCISE 8.11 Answer the same question as in Exercise 8.10 but using GSAT instead of WSAT. □

EXERCISE 8.12 Consider the set consisting of two following two clauses

$$p \quad \neg p \vee q$$

Suppose that the initial random interpretation is $\{p \mapsto 0, q \mapsto 0\}$ and k is a positive integer.

- (1) What is the probability that GSAT will find a model of this set after $2k$ flips?
- (2) What is the probability that WSAT will find a model of this set after $2k$ flips? □

EXERCISE 8.13 Do the same as in Exercise 8.12, but for the following set of two clauses:

$$p \vee q \quad \neg p \vee \neg q \quad \square$$