



## A distributed GSAT/Random-Walk For The Satisfiability Problem

By

*Ali Chelli*

*Mohamed Altamimi*

*Farouk Dhahbi*

Supervisor: Bouhmala Nouredine

**Project report for <Distributed System> in <Spring 2006>**

Agder University College  
Faculty of Engineering and Science

Grimstad, 1 June 2006

Status: <Final>

**Keywords:** <Satisfiability, Random walk, Algorithm, Greedy, Optimization>

### **Abstract:**

*Many problems in the area of computer science can be solved by transforming them into a SAT problem. Thus finding an efficient manner to solve SAT problem is of crucial importance. In fact SAT problem can be used to verify whether a product or even an algorithm fulfils its requirements or if two different components fulfil the same task. This last feature can allow us to produce a product with an optimal use of resources.*

*In our project we try to study the performance of the algorithm GSAT/Random Walk used for solving SAT problems. First we try to evaluate the execution time for the centralized version of this algorithm by using it with different instance of SAT problems. Then we tried to design a distributed GSAT/Random Walk algorithm and evaluate its performance based on the two parameters speed up and efficiency.*

<Altamimi, Chelli, Dhahbi >

<GSAT/Random Walk>

## Table of Contents

1	Introduction.....	4
2	Background .....	4
2.1	Definition of the satisfiability problem.....	4
2.2	Motivation.....	5
3	Problem area .....	7
4	Solution.....	7
4.1	Requirements Specification for GSAT/Random Walker.....	7
4.2	Design of GSAT/Random Walker.....	7
4.3	Implementation.....	11
4.4	Validation and testing.....	15
5	Results and Discussion.....	15
6	Conclusion.....	20

## Table of figures

Figure 2-1:	equivalent logical circuit.....	5
Figure 2-2:	regrouping circuit for equivalence testing .....	6
Figure 4-3:	Architecture of centralized system.....	8
Figure 4-4:	Architecture for distributed system.....	8
Figure 4-5:	GSAT/Random walk algorithm.....	11
Figure 4-6:	Random walk approach.....	12
Figure 4-7:	GSAT algorithm.....	12
Figure 4-8:	sketching the scenario of sending start solution from the commander.....	14
Figure 4-9:	sketching the scenario of sending response from each participant.....	14
Figure 4-10:	The commander sends the correspondent variable that must be updated.....	15
Figure 5-11:	Representation for a SAT problem.....	16
Figure 5-12:	Execution time for a centralized algorithm.....	16
Figure 5-13:	Time execution for a distributed algorithm.....	17
Figure 5-14:	influence of complexity on execution time.....	17
Figure 5-15:	Speed up.....	19
Figure 5-16:	Efficiency.....	19

## 1 Introduction

Nowadays the area of computer science is of a big interest since many activities are related directly to this field. The domain of computer science can be considered as the area having the fastest development ever seen. The microprocessor and other kind of hardware produced are becoming more and more sophisticated. This makes the testing phase a bottleneck for the company working in hardware design. Today the formal verification is arising as a solution to this problem. Many problems in formal verification can be transformed into a satisfiability problem, which makes the interest of finding an efficient manner to solve the satisfiability problem.

In this project, we try to find an efficient solution for the satisfiability problem based on a distributed Greedy Sat/Random Walk approach. Then, we implement this solution under Java. Java is a great language to use that can provide many opportunities to realize the distribution. With java, it is easy to make client/server program based on RPC procedure.

Our report is organized as following:

- Firstly, we have described the satisfiability problem
- Secondly, we give some backgrounds founded in the literature related to the topic
- Thirdly, we did explain our proposed solution that involves design, implementation of a centralized and a distributed algorithm for the satisfiability problem.
- Finally, we discussed the efficiency of our solution.

## 2 Background

### 2.1 Definition of the satisfiability problem

The Boolean satisfiability problem is an optimization problem. In this problem we have a conjunctive formula containing Boolean variables and the logic operators: AND, OR and NOT.

$$F = (!x_1 \vee x_2 \vee !x_3) \wedge (!x_4 \vee x_2 \vee x_5) \wedge \dots \wedge (x_{17} \vee !x_{29} \vee x_{33})$$

F is an instance of the SAT problem. We can see that this formula is formed by clauses that contain variables related with the “OR” operator for example (! X1 or X2 or! X3), but between the block themselves we find the “AND” operator. This form is called a conjunctive form. The formula contains n variables and k clauses. We remark also that clause can contain a value of a variable X<sub>2</sub> for instance or its negation! X1 for instance, we say that a clause is formed by literals. Each variable has two literals, then for n variables we have 2n literals.

The clauses are not accepted by construction if they contain several copies of the same variable for instance (X<sub>1</sub> or X<sub>1</sub> or! X3) or if they are tautological (i.e., contains a literal and its negation) for example (! X1 or X<sub>1</sub> or! X3).

The formula F can be satisfied or not. The formula is satisfied when a set of n literals {True, False... True} assigned to the variables {X<sub>1</sub>... X<sub>n</sub>} makes the value of F true. But there also instance of the SAT problem that have no solution and they are then unsatisfied problems.

In our project, we work only with satisfied instance of the SAT problem. We also focus on 3-SAT problem. In this kind of problems we have 3 literals in each clause, but in general SAT problem we can find a variable number of literal form a clause to another.

## 2.2 Motivation

The propositional satisfiability problem (SAT), which decides whether a given propositional formula is satisfied or not, is of central importance in various areas of computer science, including theoretical computer science, algorithmic, artificial intelligence, hardware design and verification.

Many decision problems that are encountered in the listed domain can be easily transformed into a SAT problem. Thus, finding an efficient solution to solve SAT problems is crucial since the solving of SAT leads directly to the solving of many problems in the domain of computer science.

For instance, the SAT problem is well related to the area of formal verification. In this domain, the main objective is to prove that a system fulfils the specification it was designed for, or to prove that two systems are functionally equivalent. In the contrary of the traditional verification based on testing, the formal verification is based on logic or mathematical demonstration.

Verification is an important phase for any new product. In fact if we want to verify that the system fulfill or not its specification we can use testing. The problem with testing is that the system may succeed for one thousand performed tests but we have no guarantee that the system will succeed if we perform the 1001<sup>st</sup> test then we need infinity of tests to confirm that a designed system is working well. In real world, infinity of tests can't be performed; the solution to this problem is to replace testing by formal verification. In fact, any system can be represented by a logic formula, having as parameters the input and the output of the system. Also any specification can also be represented as a formula. We can then transform this problem to a satisfiability problem. If we find that it doesn't exist any assignment of the variables that make the formula false then the property is verified.

An important issue when designing a system is cost minimizing. This goal can be achieved by limiting the amount of resources without influencing the required task that have to be fulfilled by the system. In electronics, we can find two circuits that achieve the same task but having different scheme. The objective here is to find an optimized circuit that uses the minimum number of logical gates. For a little circuit, it is easy to perform the optimization but when the circuit becomes huge ( $10^{20}$  gates), like it is the case today for processors, then the optimization task can lead to fault. Thus, it would be very interesting to find a method to verify whether two circuits are equivalent or not. By modeling this problem as a SAT problem we can easily solve the circuit equivalence problem. The figure below represents tow circuits that are equivalent. We are going to convert this problem into a SAT problem.

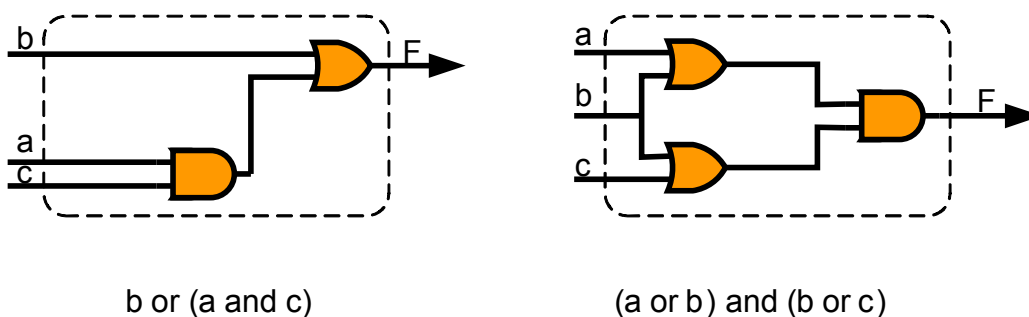


Figure 2-1: equivalent logical circuit

The question is: are those two circuits equivalent? We can easily see that the first scheme use only two logic gates but the second one use three gates, then if the two circuits are equivalent it would be better to use the first scheme. This appears a little gain (i.e. we are just making a gain of one gate) but if the circuit becomes bigger then the optimization task will lead to an important gain. We give this example only to illustrate the usefulness of SAT problem and how we can present circuit equivalence problem using SAT. Lets then consider the following scheme:

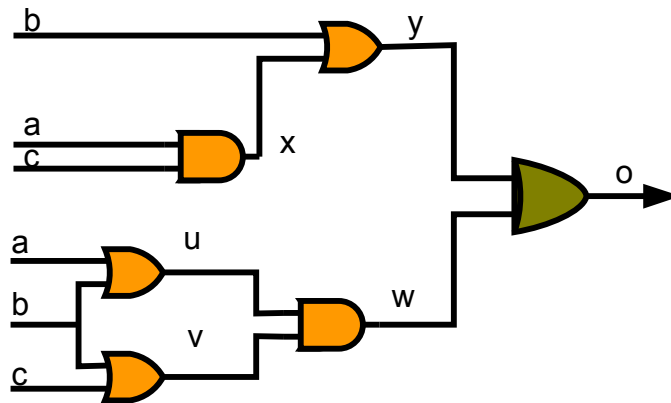
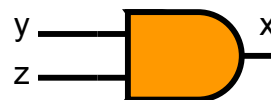


Figure 2-2: regrouping circuit for equivalence testing

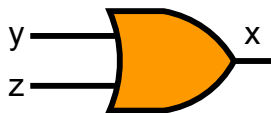
In order to formulate this problem into a SAT problem we have first to give for each signal different of a, b and c a name like it appears in the scheme x, y, u, v and w. And then for each gate we produce a complete input/output constrain as a clause.

$$F = o \wedge (x \leftrightarrow a \wedge c) \wedge (y \leftrightarrow b \vee x) \wedge (u \leftrightarrow a \vee b) \wedge (v \leftrightarrow b \vee c) \wedge (w \leftrightarrow u \wedge v) \wedge (o \leftrightarrow y \vee w)$$

This transformation from circuit to the CNF formula is called Tseitin transformation. F is not yet in a CNF form we have to make still some transformation to reach CNF form. For instance:



$$\begin{aligned} x \leftrightarrow (y \wedge z) &\Leftrightarrow (x \rightarrow y) \wedge (x \rightarrow z) \wedge ((y \wedge z) \rightarrow x) \\ &\Leftrightarrow (\bar{x} \vee y) \wedge (\bar{x} \vee z) \wedge ((\bar{y} \wedge \bar{z}) \vee x) \\ &\Leftrightarrow (\bar{x} \vee y) \wedge (\bar{x} \vee z) \wedge (\bar{y} \vee \bar{z} \vee x) \end{aligned}$$



$$x \leftrightarrow (y \vee z) \Leftrightarrow (\bar{y} \vee x) \wedge (\bar{z} \vee x) \wedge (\bar{x} \vee y \vee z)$$

We can check then that the two circuits are equivalent if the formula is satisfied for all possible values that could be assigned to the variables. Therefore, we arrive at the transformation of a simple problem to satisfiability problem

### 3 Problem area

Many algorithms were developed in order to solve the SAT problem. There are two classes of high-performance algorithms for solving instances of SAT in practice: modern variants of the DPLL algorithm, such as Chaff; and local search algorithms, such as Random Walk and Greedy algorithm. In our project, we adopt the Greedy algorithm for SAT problem. First, we are going to study the performance of the centralized GSAT/Random Walk based on the execution time. Afterwards, we study the same criteria for a distributed algorithm GSAT/Random Walk. We measure also the speed up, and the efficiency of the distributed GSAT/Random Walk.

### 4 Solution

As mentioned before, our solution holds on the GSAT/Random walk that means that we combined the Greedy approach and a simple random walker in order to converge faster to a solution for the SAT problem. In this section, we defined the different requirements specification for satisfiability problem than the design of our SAT solver that can be centralized or distributed. Finally, we test our implemented solution which was developed under java platform.

#### 4.1 Requirements Specification for GSAT/Random Walker

In this section, we present the requirements as mentioned above. Even though the satisfiability problem deals specially with mathematics and stochastic processes, it is often used in electronics and many other fields. We need some requirements that are mandatory to implement our solution. Therefore we have to:

- Make some tests
- Evaluate the performance of our solution

#### 4.2 Design of GSAT/Random Walker

The main goal of this project is to handle the satisfiability problem using a distributed system in order to decrease the workload held on communications and calculation between processors and get better results. Thus, we implement both centralized and distributed algorithm, then we compared the performances of each one.

To realize the distribution we applied many strategies; first we tried to share the calculation fairly under the different processors; afterwards we did distribute the data based on the number of variables; finally we equally separate the time calculation and the data according to the number of clauses that indicates what time is needed to terminate running process.

As consequence, the solution could be distributed or centralized. Likewise, the distribution was achieved in different manner. The centralized system works on only one processor. The following schema hints an overview about it.

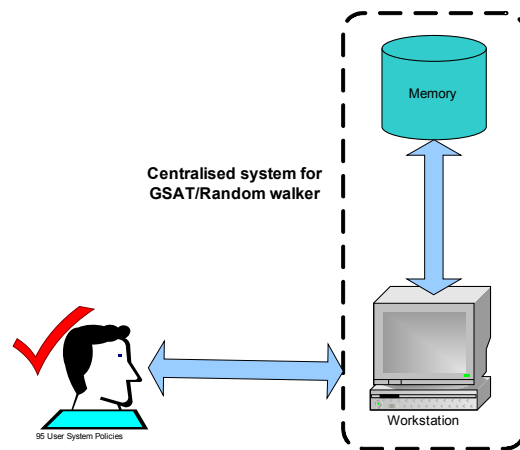


Figure 4-3: Architecture of centralized system

For the distributed, the first solution consists of distributing the time calculation between many processors.

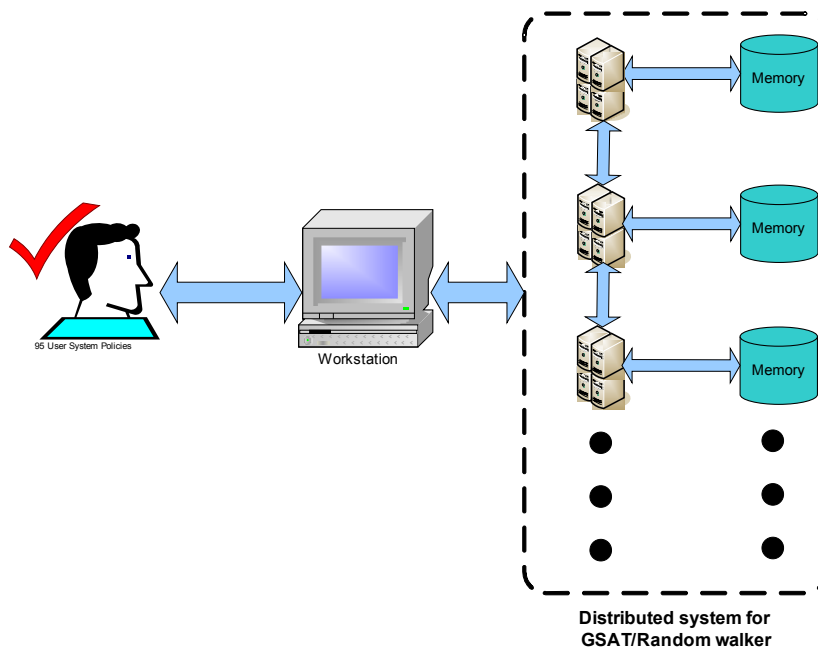


Figure 4-4: Architecture for distributed system

To implement both the centralized and distributed algorithm, we used java platform (eclipse). The reason for that choice is because of many opportunities and tools that can make our implementation easier. Java could handle a RPC procedure for a client/server that is easier than sockets.

Java platform present many features:

“Here we list the basic features that make Java a powerful and popular programming language:

- **Platform Independence**

- The *Write-Once-Run-Anywhere* ideal has not been achieved (tuning for different platforms usually required), but closer than with other languages.

- **Object Oriented**
  - Object oriented throughout - no coding outside of class definitions, including main().
  - An extensive class library available in the core language packages.
- **Compiler/Interpreter Combo**
  - Code is compiled to bytecodes that are interpreted by a Java virtual machines (JVM).
  - This provides portability to any machine for which a virtual machine has been written.
  - The two steps of compilation and interpretation allow for extensive code checking and improved security.
- **Robust**
  - Exception handling built-in, strong type checking (that is, all data must be declared an explicit type), local variables must be initialized.
- **Several dangerous features of C & C++ eliminated:**
  - No memory pointers
  - No pre-processor
  - Array index limit checking
- **Automatic Memory Management**
  - Automatic garbage collection - memory management handled by JVM.
- **Security**
  - No memory pointers
  - Programs run inside the virtual machine sandbox.
  - Array index limit checking
  - Code pathologies reduced by
    - *bytecode verifier* - checks classes after loading
    - *Class loader* - confines objects to unique namespaces. Prevents loading a hacked "java.lang.SecurityManager" class, for example.
    - *Security manager* - determines what resources a class can access such as reading and writing to the local disk.
- **Dynamic Binding**

- The linking of data and methods to where they are located, is done at run-time.
- New classes can be loaded while a program is running. Linking is done *on the fly*.
- Even if libraries are recompiled, there is no need to recompile code that uses classes in those libraries.

This differs from C++, which uses static binding. This can result in *fragile* classes for cases where linked code is changed and memory pointers then point to the wrong addresses.

- **Good Performance**

- Interpretation of bytecodes slowed performance in early versions, but advanced virtual machines with adaptive and just-in-time compilation and other techniques now typically provide performance up to 50% to 100% the speed of C++ programs.

- **Threading**

- *Lightweight* processes, called threads, can easily be spun off to perform multiprocessing.
- Can take advantage of multiprocessors where available
- Great for multimedia displays.

- **Built-in Networking**

- Java was designed with networking in mind and comes with many classes to develop sophisticated Internet communications.

Features such as eliminating memory pointers and by checking array limits greatly help to remove program bugs. The garbage collector relieves programmers of the big job of memory management. These and the other features can lead to a big speedup in program development compared to C/C++ programming.

### 4.3 Implementation

#### 4.3.1 Centralized algorithm

The centralized algorithm consists of having the data and calculation in the same machine. The calculation will be handled by one processor. The figure below shows the functioning of the algorithm:

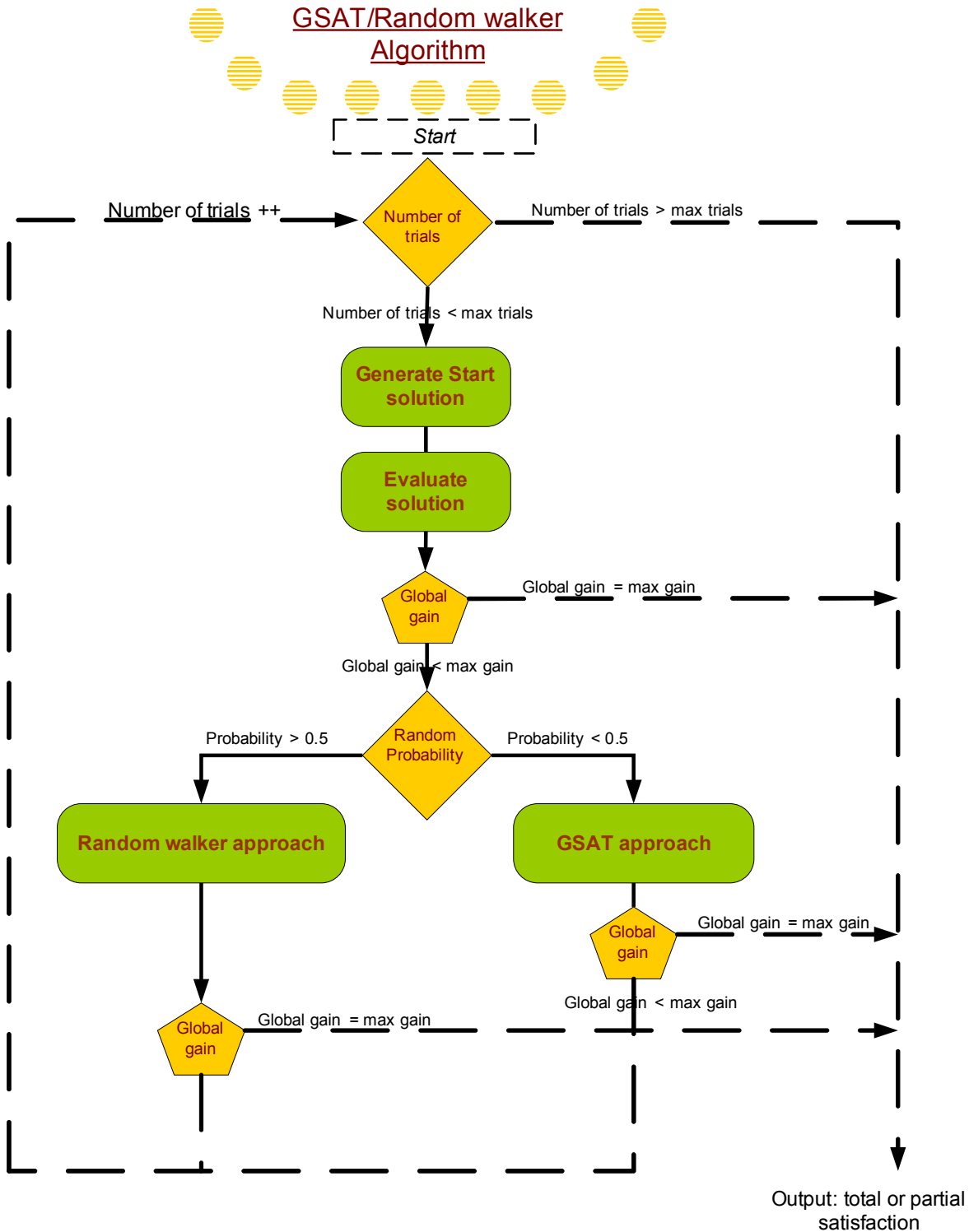


Figure 4-5: GSAT/Random walk algorithm

The random walk approach consists of flipping a random variable and evaluating the global satisfaction gain until finding the fitted combination for global satisfaction. The figure below describes the random walker processing.

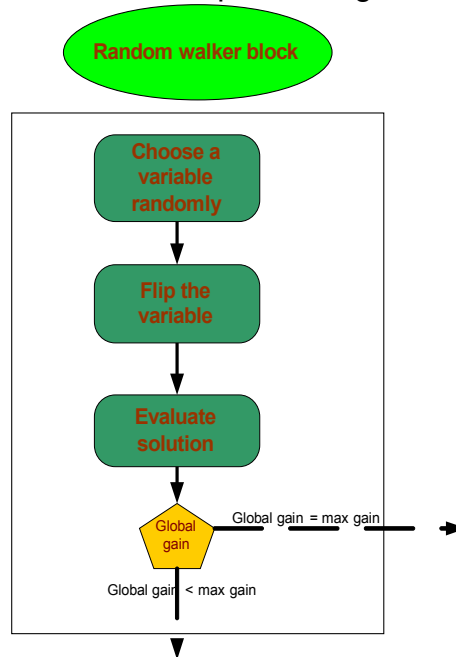


Figure 4-6: Random walk approach

The second figure illustrates the different steps of GSAT algorithm.

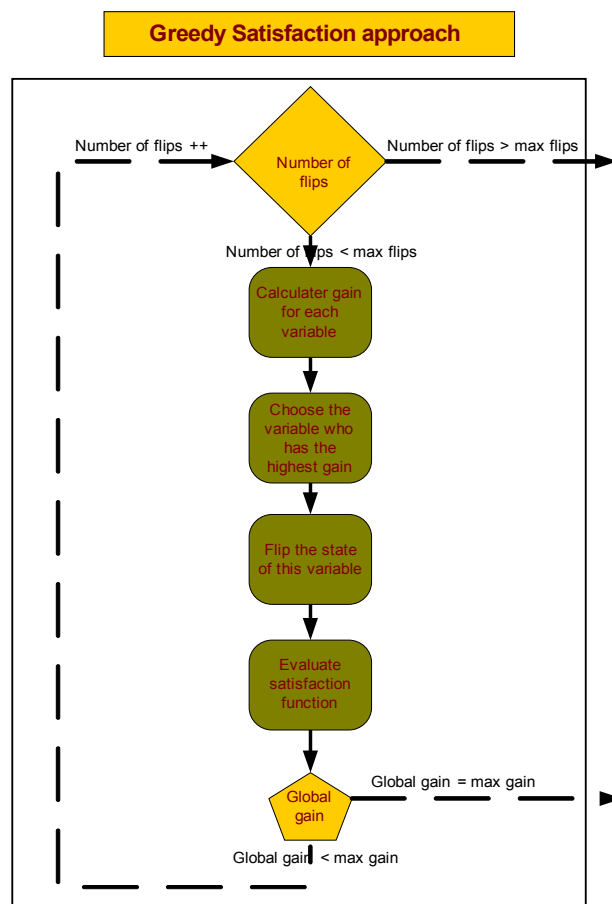


Figure 4-7: GSAT algorithm

We have introduced some functions into the algorithm to calculate for instance the global gain. To calculate the initial for a given function we did use the method below:

```
public void Initialgain() {
    for (int j = 0; j < n; j++) {
        x[j].oldgain = 0;
        if (x[j].track.size() > 0) {
            for (int i = 0; i < x[j].track.size(); i += 3) {
                Variable xtemp1 = (Variable) x[j].track.get(i);
                Variable xtemp2 = (Variable) x[j].track.get(i + 1);
                Variable xtemp3 = (Variable) x[j].track.get(i + 2);
                if (xtemp1.state || xtemp2.state || xtemp3.state)
                    x[j].oldgain = +1;
            }
        }
    }
}
```

The next method is used to calculate the global gain.

```
public int globalgain() {
    int g = 0;
    for (int i = 0; i < k; i++) {
        if (clauses[i][0].state || clauses[i][1].state
            || clauses[i][2].state)
            g += 1;
    }
    return g;
}
```

We created a class called variable shown in the following code. Each variable is characterized by a set of attributes and methods.

```
package sat;
import java.util.ArrayList;
public class Variable {
    public boolean state=false;
    public int gain=0;
    public int oldgain=0;
    public int currentgain=0;
    public double t;
    public ArrayList track = new ArrayList();
    public Variable(){
    }
    public boolean State(){
        t = Math.floor(Math.random()+0.5);
        if(t==0)
            this.state=false;
        else
            this.state=true;
        return this.state;
    }
    public void flip(){
        this.state=!state;
    }
    public void Gain(){
        gain=currentgain-oldgain;
    }
    public void update(){
        this.state= ! state;
    }
}
```

### 4.3.2 Distributed algorithm

In the following figures, we explain the functioning of a distributed algorithm by illustrating one loop of the distributed GSAT/Random Walk. We created a commander to supervise the messages exchanged between processors and choose the variable that must be updated.

At the beginning, the commander generates a start solution and sends it to each participant. The scheme below shows that the distribution is based on variables: To each participant we assign a set of variables on which he is responsible.

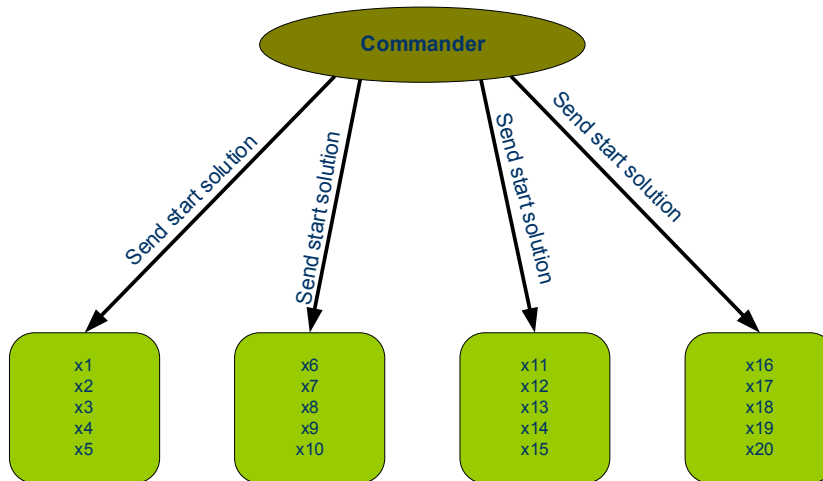


Figure 4-8: sketching the scenario of sending start solution from the commander

The second step consists of that each participant executes GSAT locally and determines the variable that has the maximum local gain and sends the identity and the gain of that variable to the commander.

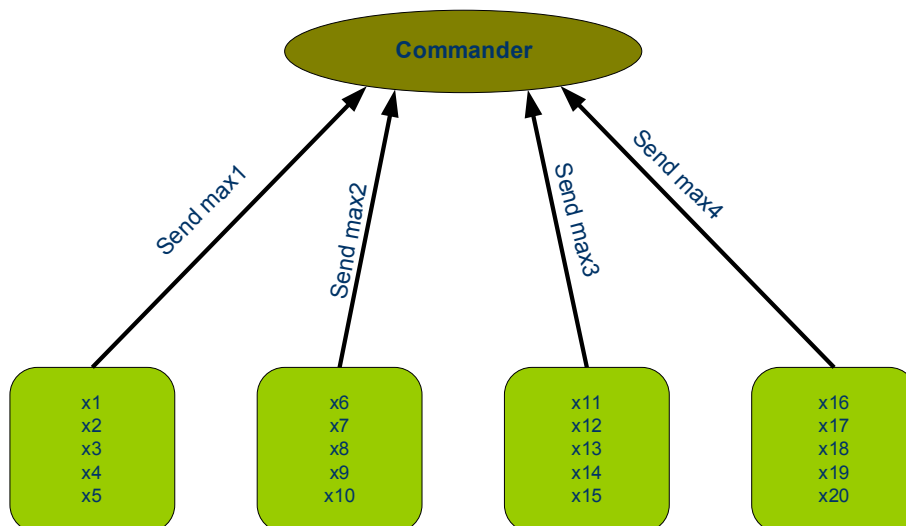


Figure 4-9: sketching the scenario of sending response from each participant

After receiving the maximum gain from each participant, the commander decides which one has the highest value. This latter must be updated by all participants.

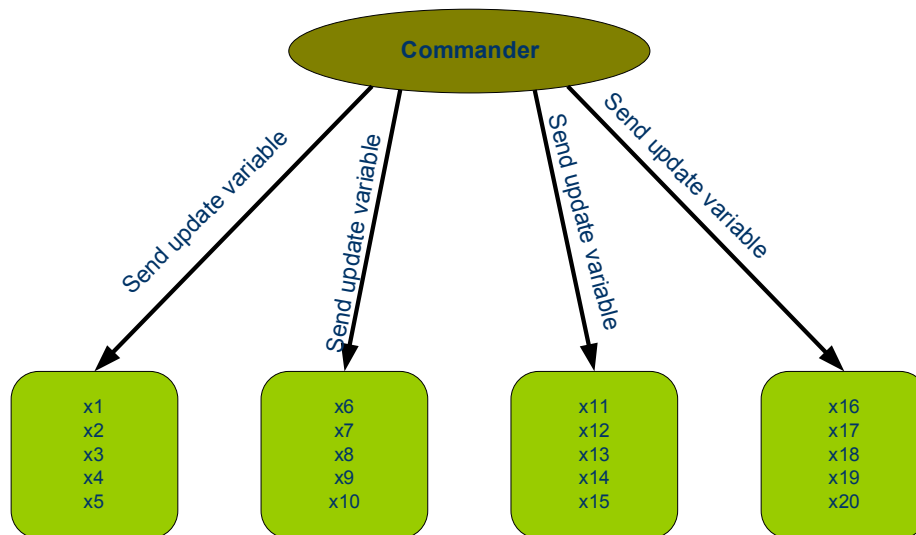


Figure 4-10: The commander sends the correspondent variable that must be updated

The next section presents the different tests that we have done to validate our solution and to evaluate the performance of the distributed GSAT/Random Walk versus its centralized version.

### 4.4 Validation and testing

In the validation phase, we measure the execution time for many instance of SAT problem in order to evaluate the performance of the distributed and the non-distributed GSAT/Random Walk approach. For a defined instance the execution time can change from an attempt to another that's why we need to calculate the average for many trials in order to estimate the execution time.

We measure the execution time for many instance of SAT problem in order to show how the execution time increases when the number of clauses and variables increase.

In order to show the influence of the distribution on the parallel execution time we choose an instance of SAT problem and we use first two processes to solve the SAT problem, and then we increase the number of processes that participate in SAT solving.

Each time we change the number of processes that participate in solving the SAT problem we measure two important parameters: the speed up and the efficiency. The speed up is a ratio given by the execution time for the centralized algorithm over the execution time for the distributed algorithm. The efficiency is also a ratio given by the speed up over the number of process used in the distributed algorithm.

## 5 Results and Discussion

The execution time for a given instance of SAT problem changes from an attempt to another. That's why we need to average over many trials and to estimate the execution time. In fact the SAT problem is a maximisation problem. This mean that we try to find an assignment for the variables that makes the formula true. But we have a very low probability that the start solution corresponds to the optimal one. That's why we use the greedy algorithm in order to enhance the start solution and to converge to the optimal solution. The problem with the greedy algorithm consists in the local maxima. In fact if we reach a local maximum we will be blocked in that point that's why we need to choose

again anew starting point in order to avoid this kind of problems. The figure 5.1 represents a SAT problem, this figures contains many local maxima. The starting point could be any point of this function. If this point is near to the optimal point we have a good chance to converge rapidly, then the execution time is short. But if the chosen point is near to a local maximum we will probably be blocked there, then we need to choose another starting point. All of this makes the execution time change from an attempt to another. Thus we need to calculate an average over a number of trials.

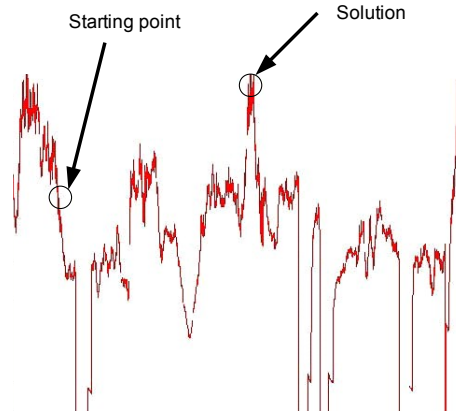


Figure 5-11: Representation for a SAT problem

First we make the test for an instance of SAT problem that contains 20 variables ( $n=20$ ) and 91 clauses ( $k=91$ ). We choose a value of maxtrials equal to 10 and a value of maxflips equal to  $10*n$ , then we calculate the execution time for many attempts and try to find out the average.

The figure 5.2 shows the mean execution time for a centralized algorithm which is approximately 130 milliseconds. It is clear that the execution time varies from one test to another, thus we need to calculate a mean value to estimate the execution time.

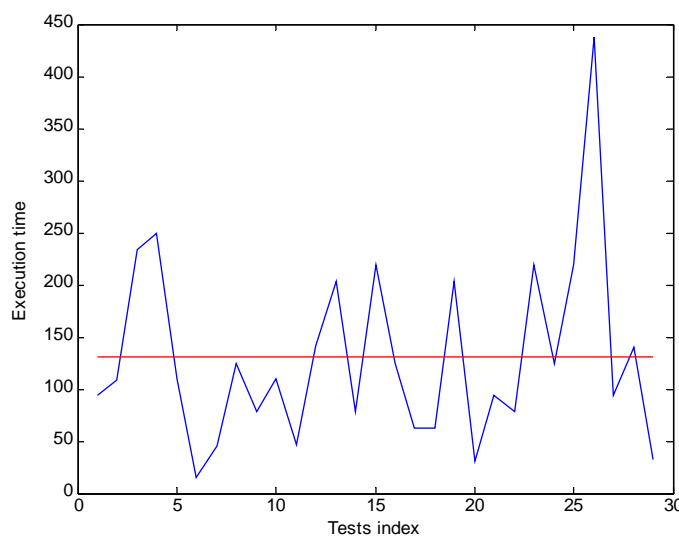


Figure 5-12: Execution time for a centralized algorithm

The figure 5.3 shows the execution time average for a distributed algorithm which is approximately 75 milliseconds. Here we have distributed the workload between two processes.

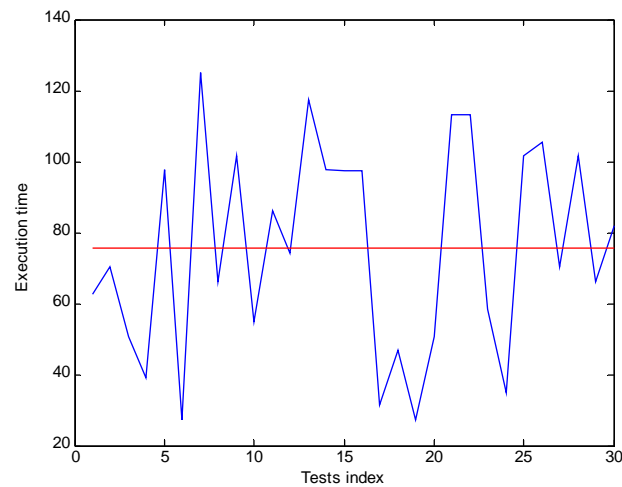


Figure 5-13: Time execution for a distributed algorithm

In order to show the need of distributing the GSAT/Random walk algorithm, we show in the figure 5.4 how the execution time increases when we choose a SAT problem with higher complexity (i.e. when we increase the number of variables and clauses). In order to be able to see the influence on an increasing complexity on execution time we have fixed the parameters maxflip (100\*n) and maxtrials (10) for all the tests but we change only the considered problems.

The table below illustrate the different instances of SAT problems for which we calculate the execution time in the case of a centralized algorithm. Execution time is given in millisecond.

SAT instance index	Number of variables (n)	Number of clauses (k)
instance 1	20	91
instance 2	50	218
instance 3	75	325
instance 4	150	645
instance 5	250	1065

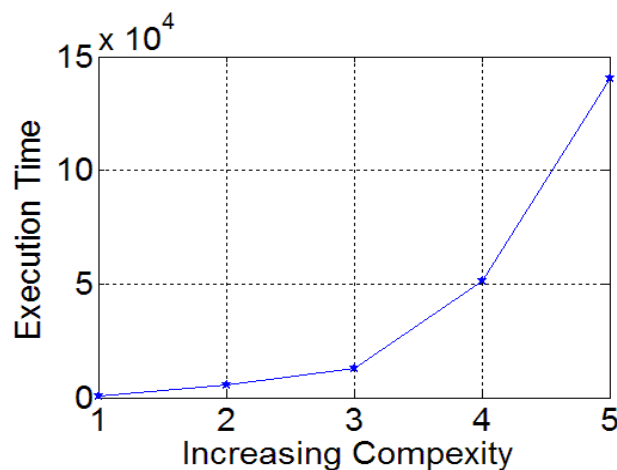


Figure 5-14: influence of complexity on execution time

The figure 5.4 shows that the execution time increases exponentially when the complexity of the problem increases. We have to mention also that for an instance of SAT problem that has a big number of variables and clauses, choosing the value of maxtrials equal to 10 and maxflip  $100 \times n$  is not enough to reach the solution. Thus we can conclude that when we have a SAT problem with a big number of variables and clauses the execution time is very big. Then it will be interesting to find a solution that permit to reduce the execution time. For these reason we distribute the GSAT/Random Walk.

In the following we show the influence of the distribution on the execution time by studying the two parameters speed up and efficiency.

We have chosen an instance of SAT problem containing 75 variables and 325 clauses. Then we make our testing for the distributed algorithm. We divide first the workload between two processes, then 3, 6, and 8. We calculate then the execution time for each process. The process that takes the longest time to finish his task is called laziest process. The parallel processing time is defined as the time that takes the laziest process to finish his task. We calculate also for each case the speed up and the efficiency. In the tables below we give the obtained results.

Process name	P1	P2
Time per process	6118	6710
Parallel processing time	6710	
Total time for centralized	12700	
Speed up	$12700/6710=1.89$	
Efficiency	$1.89/2=94.63\%$	

Results obtained for 2 processes

Process name	P1	P2	P3
Time per process	3973	4500	4355
Parallel processing time	4500		
Total time for centralized	12700		
Speed up	$12700/4500=2.8$		
Efficiency	$2.8/3=93.33\%$		

Results obtained for 3 processes

Process name	P1	P2	P3	P4
Time per process	2907	3210	3578	3133
Parallel processing time	3578			
Total time for centralized	12700			
Speed up	$12700/3578=3.5$			
Efficiency	$3.5/4=87.5\%$			

Results obtained for 4 processes

Process name	P1	P2	P3	P4	P5	P6
Time per process	1815	2158	2144	2355	2131	2225
Parallel processing time	2355					
Total time for centralized	12700					
Speed up	$12700/2355=5.4$					
Efficiency	$5.4/6=90\%$					

Results obtained for 6 processes

Process name	P1	P2	P3	P4
Time per process	1500	1658	1592	1500
Process name	P5	P6	P7	P8
Time per process	1908	1539	1394	1737
Parallel processing time	1737			
Total time for centralized	12700			
Speed up	$12700/1737=7.3$			
efficiency	$7.3/8=91.25\%$			

Results obtained for 8 processes

The influence of the number of process on the speed up and the efficiency is shown in the figure 5.5 and 5.6.

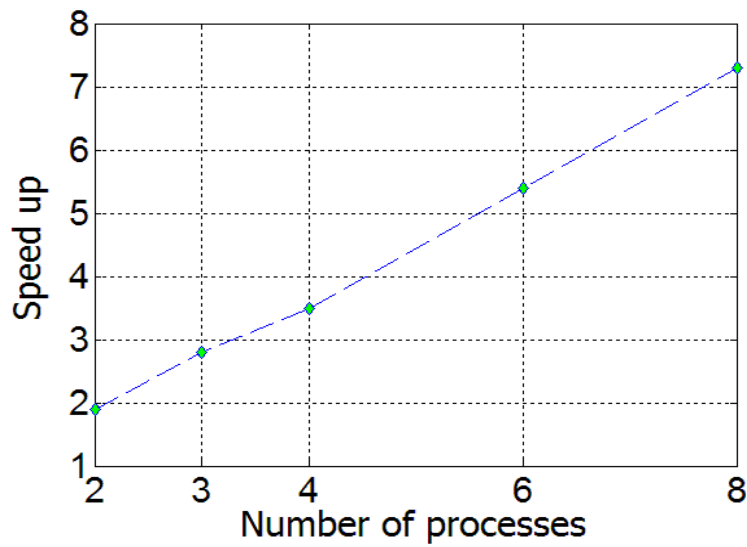


Figure 5-15: Speed up

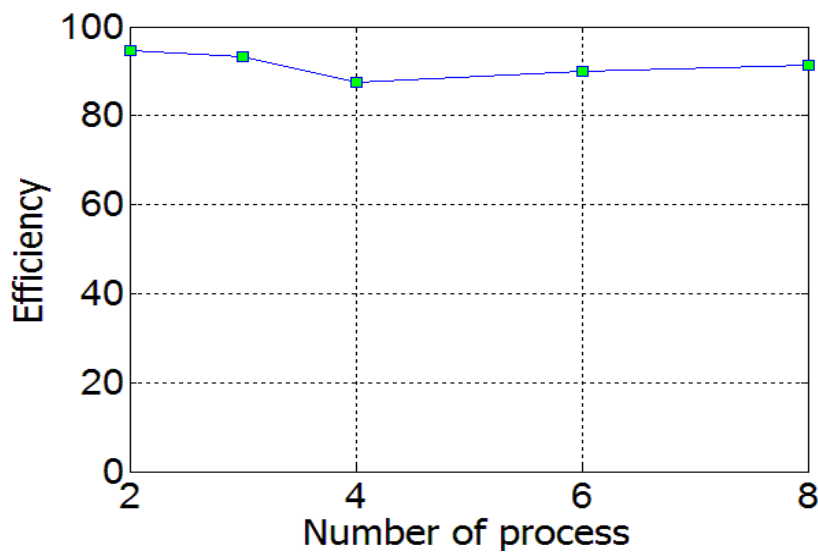


Figure 5-16: Efficiency

The figure 5.5 illustrates the variation of the speed up as a function of the process used for the distributed algorithm. The speed up expresses how many times the distributed algorithm is faster than the non-distributed one. We can easily see that the speed up is almost a linear function of the number of process. Then when we increase the number of process, the execution time will decrease linearly.

The figure 5.6 shows the efficiency as a function of the number of process. It can be seen that efficiency is almost constant. This is due to the fact that we don't have a lot of communication in our distributed version of GSAT. In fact if a distributed algorithm includes a lot of communication between different participants, then the increasing of process number will decrease the efficiency and this makes the algorithm not scalable. In our distributed version of GSAT we make the partitioning of workload based on the variables and this lead to lower communication, and this makes this algorithm scalable because we can distribute the workload between many participants without influencing the efficiency parameter.

A major limitation for the GSAT algorithm is the fact that we need to assign very big values for the parameters maxflip and maxtrials in order to converge to the optimal solution, if the number of variables and clauses become high. We can overcome this problem by using a distributed GSAT in order to reduce the execution time.

## 6 Conclusion

In the conclusion, the problem of satisfiability that consists of finding the combination of a set of clauses that reach an entire satisfiability can be solved by implementing GSAT/Random walk. This latter is an efficient solution for satisfiability problem whereas it has some limitations when we increase the number of variables. In fact, the execution time increases exponentially with the complexity of the treated problem. A distributed GSAT/Random Walk can be used in order to decrease the execution time. But sometimes this solution is not enough that's why many other approaches have been developed in order to solve the SAT. Those approaches can be the subject of further study.