

Project Report

Introduction

WalkSAT is a search algorithm for SAT problem, which is a problem of satisfiability. In this practical, the input of the WalkSAT satisfies the Dimac CNF Format.

Design

The practical is designed mainly according to the first character in the project requirement, which provided a pseudo code to implement.

```
1. Choose a random complete truth assignment T
2. while (T leaves at least one unsatisfied clause) {
3.     choose an unsatisfied clause C at random
4.     generate a random number r between 0 and 1
5.     if (r > p) {
6.         set v = pickvar(C) // select variable v in C to flip which
                             // maximises number of satisfied clauses
7.     } else {
8.         set v = randomly chosen variable in C
9.     } // end if
10.    set T = flip(v) // T with v set to the opposite value
11. } // end while
12. Return T
```

Pickvar(C) – line 6

Pickvar selects a variable which can maximise the number of satisfiable clauses if the variable is chosen. To choose the variable in C, it will search all clauses for all literals in the clause and select the variable that maximises changes. Changes can be calculated by counting the sub of how many clauses will be affected from unsatisfied to satisfied and how many clauses will be modified from satisfied to unsatisfied.

Flip(v) – line 9

Flip will search all clauses that contain the variable and not variable (x and not x). Then it will flip the truth assignment of the variable from true to false or from false to true.

Additional Structure:

Since Pickvar requires picking each literal in the clause, the structure of the truth assignment is designed as

{Clause index: {Variable: Assignment, ...}, ...}

This structure allows picking the variable and choose its assignment efficiently, and the clause is easy to be accessed by using the clause index.

Since both Pickvar and Flip must search all clauses contains such literal. Considering the condition that literals are not moved to new clauses, a literal static map can be implemented to make the process easier. The structure of the map is:

{Variable: {Clauses(list)} ...}

Such structure allows the quick search of the literal when flipping and picking var.

Since looping runs when there are still unsatisfiable clauses, a list which contains false clauses can be created to make the looping more efficient. By cooperating with an integer array which stores how many literals are satisfied by the clause, it is easy to monitor the change of the satisfiability to make the program more efficient.

[clauseInd1, clauseInd5...]

[0, 1, 1, 1, 2, 0]

Randomly chosen – line 1 & line 4 & line 8

All random variables are generated by using Random class.

Line 1 will generate a random truth assignment by using *random.nextBoolean()*, we set a Boolean variable to each literal and assigning the assignment to the truth assignment, a random T is generated.

Line 4 requires a random double between (0, 1); it can be generated by using *random.nextDouble()* and a random r is chosen to choose we should use Pickvar or randomly choose a variable.

Line 8 will randomly choose a variable in the clause. By accessing the keyset of the clause in truth assignment, we can get a random clause by converting the cause to an array list and randomly choose the variable by its index by using *random.nextInt(list.size)*.

Implementation

The implementation of the design is the implementation of reading contents from the file, initialising the data structure and implementing functions according to the pseudo code efficiently.

Initialising

As what is introduced in the design above, additional data structures are implemented to efficiently running the program. Since both numbers of variables and number of clauses are told, to initialise “clausesContainingLiteral”, we will put “+/- var” as the key element and initialising the array list to avoid null pointer exception. To initialise the initial truth assignment, we need only reversely initialising the truth assignment. By scanning the file, literals are filled into both clauses containing literals and truth

assignment, where truth assignment assigned false to all of the elements in the assignment table.

The screenshot of two data structures are as follows:

```
Clauses containing literal: {-1=[], 1=[], -2=[], 2=[], -3=[], 3=[], -4=[], 4=[], -5=[], 5=[], -6=[], 6=[]}  
Clauses :{0={1=false, 2=false, 3=false}, 1={1=false, 2=false, -6=false}, 2={1=false, 2=false, -5=false}, :
```

Then we need to initialise clauses after all literals are filled into both maps. As what has described in design, by scanning from clausesContainingLiteral, we will get the literal and the reverse of the literal and randomly assign contrast value to them. Then we check the satisfiability of clauses and add all clauses, which has 0 satisfiable value to false clauses.

```
[3, 6, 7]
```

```
{0={1=false, 2=true, 3=false}, 1={1=false, 2=true, -6=false}, 2={1=false, 2=true, -5=true}, 3={1=false, 3=false, 5=false},
```

File reading

Scanning a CNF Dimacs Format requires scanning keywords, skipping 0 and comments. The sample of the input file shows as follows:

```
c Example CNF in Dimacs format  
c A cnf encoding of the letter SAT problem used in Search lectures  
c Comment lines before declaration start with c  
c Declaration starts with p, gives format (cnf) and num vars/clauses  
c Variables are 1 ... num vars, Negated variables are negative integers.  
c Clauses are lists of positive or negated variables terminated by a 0  
c  
p cnf 3 7  
1 2 3 0  
1 2 -3 0  
1 -2 3 0  
-2 -3 1 0  
-1 2 3 0  
3 -2 -1 0  
-1 -2 -3 0
```

What should be done first is to skip all comments. Comments are starting with a single character c, skipping comments need only to skip this line when the scanner scans the character c. while scanner scans p, it will scan next as type. If the type is cnf, it will follow with two variables for the number of variables and number of clauses. After initialising clauses, the scanner will scan next clauses until the end of the file and fill all that has scanned to clauses and clausesContainingLiteral.

Pick and flip

Pickvar picks the variable that maximizes the change. Change is mathematically calculated by $\text{make}(x) - \text{break}(x)$ where $\text{make}(x)$ is the number of clauses that will turn unsatisfied to satisfied, where the $\text{break}(x)$ is the number of clauses that will turn satisfied to unsatisfied. So we can count the specific index of the int array “numSatisfiedPerClause.” If the result is 1, which means it will be turned to 0 as it turns false. If it is 0, it means it will turn to 1 as it becomes true.

While flipping the variable, it will scan where this variable located from “clausesContainingLiteral”. After flipping those variables, it will update numSatisfiedPerClause. If numSatisfiedPerClause with specific index is now 0, the clause will be added into the false clause and vice versa.

There is an issue happened when picking the var is that sometimes changes might be negative. This will cause a bad move to prevent the exception. Here the minimal changes is INT_MIN, which also gives an limitation to the input file. The limitation will be explained in evaluation.

Printing result

After WalkSAT is done, we have to print the result of the truth assignment. There are two possible result, first is that the WalkSAT might not be complete, this will print the time out alert. This is an example with maximum time 1 second

```
Timed out!
Process ends in 1005 ms.
Thread-0 is called: Thanks for using!
```

Another possibility is that input clauses are complete and time cost is less than the maximum time from command line argument, it will show the result, total loop the program has experienced and its time cost.

```
Total looping: 4 result:
0: [true false true ]
1: [true false true ]
2: [true false true ]
3: [true true false ]
4: [true false false ]
5: [true false true ]
6: [true true false ]
7: [true true true ]
8: [true true false ]
9: [false false true ]
10: [false true true ]
11: [false true false ]
Process ends in 109 ms.
Thread-0 is called: Thanks for using!
```

Testing

Sample testing:

Sample testing includes three testing: testing from sat format sample file, testing from requirement and testing more cnfs.

Input:

```

c Example CNF format file
c
p cnf 4 3
1 3 -4 0
4 0 2
-3

```

Output:

```

Total looping: 1 result:
0: [true true false ]
1: [true ]
2: [true false ]
SATISFIABLE
Process ends in 75 ms.
Thread-0 is called: Thanks for using!

```

Input:

```

c Example CNF in Dimacs format
c A cnf encoding of the letter SAT problem used in Search lectures
c Comment lines before declaration start with c
c Declaration starts with p, gives format (cnf) and num vars/clauses
c Variables are 1 ... num vars, Negated variables are negative integers.
c Clauses are lists of positive or negated variables terminated by a 0
c
p cnf 3 7
1 2 3 0
1 2 -3 0
1 -2 3 0
-2 -3 1 0
-1 2 3 0
3 -2 -1 0
-1 -2 -3 0

```

Output:

```

Total looping: 4
result:
0: [true false true ]
1: [true false false ]
2: [true true true ]
3: [true true false ]
4: [false false true ]
5: [false true true ]
6: [false true false ]
SATISFIABLE
Process ends in 76 ms.
Thread-0 is called: Thanks for using!

```

Input:

File: ./test/2/cnf250/uf250-0100.cnf

Output:

```

- SATISFIABLE -
Total looping: 215078
result:
0: [false false true ]
1: [false false true ]
2: [true true false ]
3: [false true false ]
4: [true false true ]
5: [true true true ]
6: [false false true ]
7: [true false true ]
8: [true true false ]

```

Process ends in 796 ms.

Thread-0 is called: Thanks for using!

Testing the above tests three conditions, which evaluate the success of the project:

1. Testing the unregular input file: the input file of the first input is unregular, the success of the first file confirms the readability of the program.
2. Sample file testing: testing a sample file is the necessary and fundamental testing which tests that this file can be minimally used. This could test the satisfiability as it is not too large and is simple to judge the result is true.
3. Complex file testing. The succeed of testing the multiple files evaluates that there are no apparent bugs in the program, which makes the numerous file runnable.

There should be a fourth test, which tests the timeout (result=unknown). Since timeout can be tested by using while(true), so the timeout test is passed too.

Evaluation

Performance

Performance for the program is nice, as flips is generally spend too much time and deals with large issues is not too slow. An 250 variables questions costs 900ms in average as an 50 variables questions costs 170 ms.

Process ends in 170 ms. ./test/2/cnf250/uf250-0100.cnf

Process ends in 917 ms. ./test/2/cnf50/uf50-0100.cnf

Good p:

While p is so large, it will always walk into stuck, as when p is so small, it will ever walk randomly.

This is the result when using different p in ./test/2/cnf250/uf250-0100.cnf:

P = 0.1 – 0.3 Time Out

P = 0.4: 8129 ms

P = 0.5: 917 ms

P = 0.6: 318 ms

P = 0.7: 1721 ms

P = 0.8 – 1.0 Time Out

P is probably better to set to 0.6, which makes hill climbing is a bit more than random moving, but not stuck into loops.

Questions:

1. How many flips per second can your program do on problems on different sizes?
2. What are the largest problems your program can solve in a reasonable time?
3. What are the smallest satisfiable problems that your program can't solve in a reasonable time?

Answer

1. By adding two lines of code before and after flipping, I got a graph as following:

<code>long current = System.nanoTime();</code>	6000
<code>for (int k : clausesContainingLiterals.get(var)) {</code>	8200
<code>boolean now = clauses.get(k).get(var);</code>	6100
<code>clauses.get(k).put(var, !now);</code>	5700
<code>update_clause(now, k);</code>	8300
<code>}</code>	5500
<code>for (int k : clausesContainingLiterals.get(-var)) {</code>	4700
<code>boolean now = clauses.get(k).get(-var);</code>	4200
<code>clauses.get(k).put(-var, !now);</code>	11100
<code>update_clause(now, k);</code>	5900
<code>}</code>	4500
<code>System.out.println(System.nanoTime() - current);</code>	4200

← Code / Time →

Average time cost for each flipping is 11287.598425197 nanoseconds for 250 variables, 1065 clauses input.

Average time cost for each flipping is 2997.7521929825 nanoseconds for 200 variables, 860 clauses input.

Average time cost for each flipping is 13041.234439834 nanoseconds for 150 variables, 645 clauses input.

Average time cost for each flipping is 12140.915688880 nanoseconds for 100 variables, 430 clauses input.

As a result, time cost for each flipping does not affected so much by number of variables and number of clauses.

2. There are several elements related to the solution of this problem.
 - a. Flipping: flipping could be affected by the null pointer exception if the maximum changes is less than INT_MIN or larger than INT_MAX.
 - b. File reading and memory: If the file is so large, since clauses are loaded twice, it will take 2N memory for storing the file into the memory. Loading clauses to memory will be affected and failed if the file size is larger than a half of the memory.
 - c. INT_MAX issue: variables and clauses should less than INT_MAX, or the program will failed.

3. This is the smallest unsatisfiable problem that the program cannot be solved.

```
p cnf 1 2
1 0
-1 0
```

- UNKNOWN -
 Timed out!
 Process ends in 10004 ms.
 Thread-0 is called: Thanks for using!


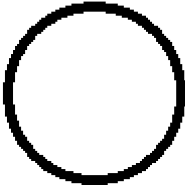
Since x is true, -x is always false and vice versa. So this will never satisfiable.

Conclusion

This practical is about implementing the WalkSAT search algorithm. In my perception, WalkSAT has its benefit either harm, which is the random walking. The “bad move” let the regress of searching, though it is not a kind of trap. It provides the chance to prevent the moving trapped into stuck. A random pick will offer more opportunity as simply do the hill-climbing will cause the program held in the loop. This is a brilliant algorithm to understanding AI as intelligence should not make everything correct, or it is just a kind of computer but not intelligence.

Check Table

Alternation implementation might be associated with clauses data structure. But I am not sure about that. As a result, a question mark is put in check table.

Phase	Task	DONE?	REPORTED?
Implementation	Key design decisions made		
	Data Structures Implemented		
	WalkSAT Algorithm Implemented		
	Correctness Testing		
	Code appropriately documented		
	JAR provided		
	<i>Alternative implementations done</i>	?	?
Evaluation	Supplied Instances Evaluated		
	Other Sized Instances Found/Evaluated		
	Flips per second		
	Size of problems that can be solved		
	<i>Good values of p evaluated</i>		
Report	(additional to the above items)		
	Problems Encountered / Overcome		
	<i>Description of additional algorithms</i>		
	<i>Comparison with results from literature</i>		

Reference:

Introduction to Logic

<http://intrologic.stanford.edu/extras/satisfiability.html>

Local Search Strategies for Satisfiability Testing. Selman, Kautz, and Cohen, 1993.

This is the original paper introducing WalkSAT.

<http://www.cs.cornell.edu/selman/papers/pdf/dimacs.pdf>