CSE294 Term Project Report

Logic in Computer Science

Practical Satisfiability Solver Implementation

Subject: Logic in Computer Science

Department of Computer Science and Engineering

Student Number: 2000160184

Name: 이 진 무

CSE294 Term Project Report

1. Abstract

이 문서는 2003년 가을 학기 CSE294 전산논리 수업의 term project에 대한 보고서이다. 이 project는 NP-complete 문제의 대표적인 예인 Satisfiability(SAT)문제를 효율적으로 해결하는 프로그램을 작성하는 것이다. SAT 문제는 brute-force algorithm으로 풀면 평균수행시간이 $O(2^n)$의 time complexity를 가진다. 이 방법으로는 해결이 거의 불가능하다는 것을 의미한다. 그러므로 heuristic을 이용하여 search space를 줄일 수 있는 practical solution이 필요하다. SAT를 해결하는 가장 대표적인 방법으로는 Davis, Putnam, Logemann, and Loveland가 제안한 DPLL algorithm이 있다. 이 방법은 depth-first search에 기반을 두고 있다. 이번 프로젝트에서는 DPLL algorithm을 기본으로 프로그램을 작성하였다.

2. Development and Execution Environment

- Operating System: Linux Redhat 8.0 kernel 2.4.20
- Programming Language: C
- Compiler: GCC version 3.2
- CPU: Pentium III 800 MHz
- RAM: 128 MB

3. Overview of my Program

이번에 작성한 프로그램은 하나의 C source file, 하나의 header file과 compile을 돕는 makefile로 이루어져 있다. C source file은 main(), DPLL(), pick_next_var(), reduce(), un_reduce(), find_unit(), find_pure(), is_empty(), exist_empty_clause(), init(), new_count(), show(), show_stack(), stack_get(), stack_put() 등 함수들로 이루어져 있다. 그리고 이 함수들은 CLAUSE structure와 LITERAL structure들로 이루어진 linked list와 array로 이루어진 stack을 handle한다.

처음 프로그램이 실행되면, 사용자로부터 어떤 DIMACS file을 사용할지 선택하게 한다. 지정한 file을 열고, 자료를 읽으면서 data structure를 구성한다(init() function). 그 다음, 현재 clock time을 저장하고, DPLL algorithm을 수행한 후(DPLL() function), 다시 clock time을 받는다. DPLL()의 결과를 받아서, 해당 CNF가 satisfiable인지 unsatisfiable한지 출력한다. Satisfiable한 경우, 어떤 truth assignment가 formula를 satisfy하는가 보여준다. 마지막으로, 수행시간을 출력한다.

이제 세부적으로 어떤 알고리즘과 자료구조를 사용하였는지 알아보자.

4. About the Algorithm

앞에서 설명하였듯이 DPLL algorithm을 사용하여서 프로그램을 구현하였다. DPLL algorithm의 pseudo code를 보면 다음과 같다.

```
DPLL () {
        // termination condition
        if f is empty, return SAT;
        if empty clause ∈ f, return UNSAT;
        // unit preference
        if ∃c ∈ f with c = {l}, then reduce(l) and DPLL();
        // pure literal
        if ∃l ∈ f with l is a pure, then reduce(l) and DPLL();
        // now, branch!
        v = pick_next_var();
        reduce (v);
        if DPLL() == SAT, return SAT;
        else undo all the reductions after the branching
        reduce (!v);
        return DPLL();
}//end DPLL()
```

사용한 DPLL algorithm의 특징

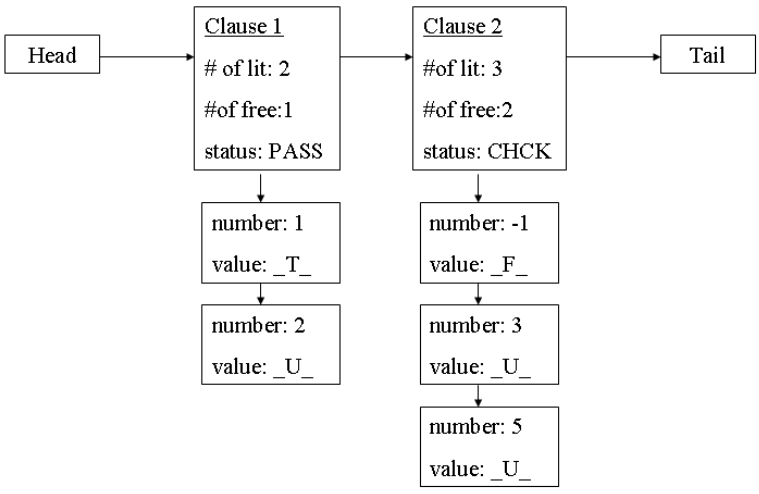
- Pure literal checking보다 unit clause checking을 먼저 수행한다. unit clause를 먼저 찾아는 것이 경험상으로 더 빠른 수행 속도를 보였기 때문이다.
- DPLL()함수에 parameter로 reduced CNF formula 전체를 넘기지 않고, reduce()를 수행한 후에 recursively DPLL()를 부르는 형식으로 하였다. 이때, branch하다가 UNSAT의 결과가 나오면, 해당 branch time이후의 모든 reduction의 과정을 undo해야 한다. 이 backtracking process를 위해서 truth assignment하는 모든 과정을 stack에 저장하도록 하였다. Reasoning 중간에 empty clause가 생성되면, 처음 branch를 시작했던 곳까지 stack을 pop하면서 역으로 추적해나가게 된다.
- 마지막으로, pick_next_var() 함수에 대해서 설명해보자. 처음 이 알고리즘을 구현하였을 때에는, CNF의 가장 처음 나오는 unassigned variable을 찾았다. 하지만, 보다 나은 성능을 위해서 CNF전체를 살핀 후에, 가장 많이 등장하는 variable을 찾는 방식으로 바꾸었다. 그 결과, report_3.cnf에서의 결과가 2초 정도 향상되었다.
- 방학 후에, iteratively 알고리즘을 바꾸고, zChaff에서 사용하는 방법론을 적용하려고 생각 중이다.

5. About the Data Structure

사용한 자료구조는 1) CNF에 대한 정보를 가지고 있는 2차원의 linked list (CLAUSE, LITERAL)과 2) truth assignment 과정을 기록하고 있는 stack이 있다. Stack은 간단히 integer형 array를 사용하였다. CNF에 대한 structure들을 보면, 다음과 같다.

```
//the data structure of clauses
typedef struct _clause {
        int     lineno;         //line number for debugging
        int     num_of_literals;//number of literals
        int     num_of_free;    //number of free variables
        int     status;         //status of the clause: PASS, EMPTY, CHCK
        struct _literal *first;//ptr to the first literal
        struct _literal *sat;  //ptr to the literal that makes this clause true
        struct _clause *next;  //ptr to the next clause
} CLAUSE;
//the data structure of literals
typedef struct _literal {
        int     number;         //variable number (ex)1, -1, 3, -2, etc.
        int     value;          //value: _F(false), _T(true), _U(unknown)
        struct _literal *next;//ptr to the next literal if end point itself
} LITERAL;
```

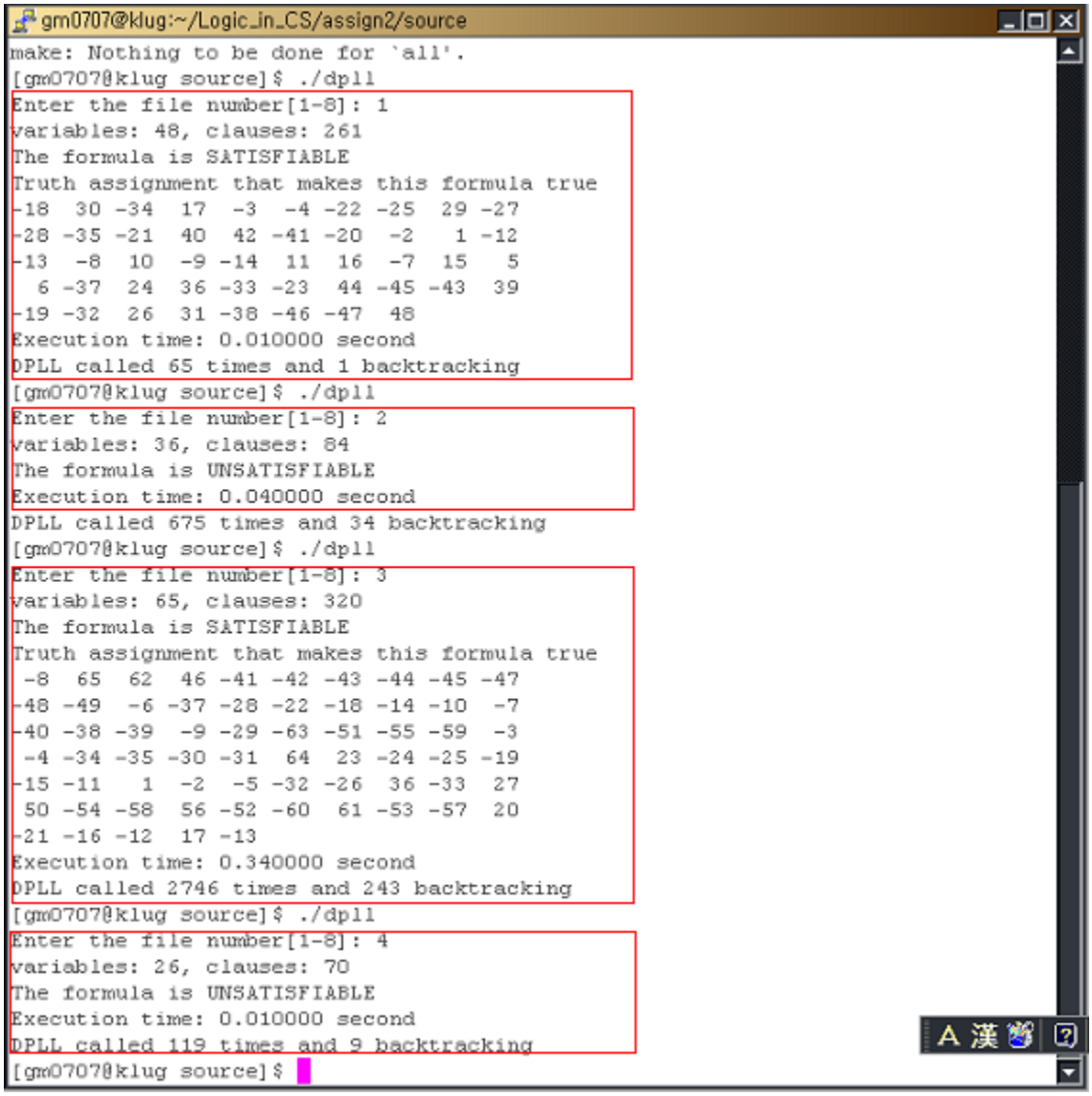
예를 들어, { {1, 2}, {-1, 3, 5}} 의 CNF를 위 structure들을 이용해 표현한 모습은 다음과 같다. 현재 1이 true가 되었다고 하자. 그러면 다음과 같은 자료구조가 만들어 진다.



[Two dimensional linked list]

6. Execution Results

- input: report_1.cnf (3KByte) - output: satisfiable/ 0.03 second
- input: report_2.cnf (1KByte) - output: unsatisfiable/ 0.04 second
- input: report_3.cnf (4KByte) - output: satisfiable/ 0.43 second
- input: report_4.cnf (1KByte) - output: unsatisfiable/ 0.02 second



Execution Environment는 Development Environment와 동일. (Linux RedHat 8.0 Kernel 2.4.20)

7. Conclusion and Future Work

위의 실행결과에서는 비교적 빠른 속도로 프로그램이 실행됨을 알 수 있다. 하지만, 위의 4가지 경우에서는 비교적 작은 용량의 파일들을 입력으로 받은 것이다. 하지만, 이전에 주어졌던 파일들(297KB, 603KB, 398KB, 693KB)을 입력으로 받아서 실행했을 경우에는 10시간 이상을 수행하고도 결과를 내지 못하였다. 메모리 공간을 너무 많이 사용하였기 때문에 process가 page에 대한 연산을 제대로 하지 못한 채, 계속 page-out이나 swap-out을 반복하는 thrashing현상을 그 원인으로 추측할 수 있다. 또는, DPLL()함수를 recursively 구현하여서 overhead가 많았다고 생각할 수도 있다.

일단 개선할 수 있는 부분은 1) recursive function을 iteratively 수정하고, 2) 각 clause가 truth value가 아직 지정되지 않은 literal들을 직접 pointing함으로써 access time을 줄일 수 있다. 이외에도 밑의 논문과 자료들을 분석함으로써 많은 개선점을 찾을 수 있을 것이다.

Reference

[1] Marc Herbstritt. zChaff: Modification and Extensions, Technical Report No. 188, July, 2003.

[2] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver, July 2001.

[3] Wonhong Nam. Boolean Satisfiability Solvers, November 2002.

[4] X. Y. Li, M. F. Stallman, and F. Brglez. QingTing: A Local Search SAT Solver using an effective switching strategy and an efficient unit propagation, July 2003.

[5] A. D. Mali and Y. Lipen. MFSAT: A SAT Solver using Multi-Flip Local Search.