CS402 Introduction to Logic in Computer Science

Coursework 2: Normal Forms, Validity, and Satisfiability

20165501 박준영

1. Conjunctive Normal Form

1.1. 구현 환경 및 실행 방법

OS : Ubuntu 16.04 LTS

구현언어 : C++

사용 컴파일러 : g++

실행방법

```
> cd cnf
> make
> ./cnf "{INPUT FORMULA}"
```

1.2. 구현 방법

Lecture node 의 알고리즘을 충실히 따랐다.

The translation algorithm consists of three parts:

1) Transform φ into the implication-free form, φ1.

2) Transform the implication-free φ1 into Negation Normal Form (NNF), φ2.

3) Transform the implicatio-free and NNF φ2 into CNF ψ.

단, 1) 단계에서 implication 말고도 reverse implication, equivalence 연산자가 있는데 implication 과 비슷하게 처리하면 된다.

1.3 최적화 가능성

Formula 는 parsing 하여 추상적으로 생각하면 tree structure 를 갖는다.

이 때 반복되서 나타나는 subtree 를 중복된 다른 formula 로 갖지않고 C 언어의 포인터 개념을 활용하여 동일한 포인터를 갖는 formula 로 연결할 수 있다.

즉, 실제 구현에서는 tree 가 아닌 graph 형태로 formula 를 생성할 수 있어, 중복되는 계산을 줄일 수 있다.

이는 1.2 의 알고리즘 3) 단계의 DISTR 함수에서 중복되는 계산이 많은 것을 방지해

줄 수 있다.

2. Nonogram

2.1. 구현 환경 및 실행 방법

OS : Ubuntu 16.04 LTS

구현언어 : C++

사용 컴파일러 : g++ -std=c++11

2.1.1 MiniSAT 설치

```
> cd nonogram
> cd minisat_dir
> export MROOT={MINISAT_DIR} //(README 참고)
> cd core
> make rs
> cd ../..
> cp minisat_dir/core/minisat_static minisat
```

2.1.2 nonogram 실행

```
> make
> vi input.cwd //cwd 포맷의 입력 파일
> ./nonogram
```

2.2. SAT encoding

2.2.1 Variable

SAT encoding 의 variable 은 각각의 cell 이 박스가 채워지는지, 비워져있는지로 나타난다. N*M grid 인 경우 총 N*M 개의 variable 이 생긴다.

입력으로 받은 연속 box 개수의 list 들에 의해 constraint 가 생긴다.

2.2.2 Constraints Memoization

남아있는 cell 의 개수, 채워야하는 box 의 list 를 state 로 보고,

state 가 생성하는 constraint 를 저장한다.

Constraint 를 저장하는 형태는 CNF 이다.

예를들어 남은 칸이 3 개, 채워야하는 box 의 list 가 empty 라면,

State: (3, [])에 대해서 3 개의 and clause 가 생성된다.

남아있는 칸의 1 부터시작하는 index 가 atom 이 된다.

박스가 채워지는 곳은 true, 비어있어야 하는 곳은 false 가 되어 아래와 같은 CNF formula 가 나온다.
CNF: -1 & -2 & -3
어떤 State 에서 어떤 CNF 가 생성됐는지 hash map 을 이용하여 기록해둔다.

Box list 의 front (현재 채워야하는 연속된 box 수)를 어떤 index 에 둘지 brute force 하게 탐색한다. 중복되는 계산을 방지하기 위해 memoization 기법을 사용하였다.
Memoization 기법을 사용하면, 기록된 CNF 와 현재 front box 를 둔 constraint 를 병합해야하는데, CNF form 끼리의 병합이기 때문에 1.2. 3)의 DISTR 함수를 사용하게 된다. 또한 CNF 에 기록된 atom 들도 새로 추가한 박스 뒤에 오기때문에 shift 시켜주어야한다.
이렇게 각각의 row, column 에 대해서 CNF 를 누적시키면 nonogram puzzle 를 푸는 SAT encoding 을 완성할 수 있다.

2.3 최적화

2.3.1 Memoization

2.2.2 에서 설명한대로 brute force 한 탐색에서 발생하는 중복되는 계산을 memoization 기법을 통해 줄일 수 있다. Row/column 에 상관없이 상태를 기록하기 때문에 N*M grid 에서 N=M 일 때 특히 효과적이다.

2.3.2 Bitwise-operation

CNF formula 를 표현하는 data structure 를 설계할 때 boolean array 에 대해 연산하는 것 보다 bitwise operation 이 빠르기 때문에 C++의 64bit 자료형을 사용하여 boolean array 를 대신하였다.
CNF 의 clause 들의 OR, IMPLICATION, REVERSE IMPLICATION 연산을 빠르게 할 수 있어 체감상 약 2 배정도 빠른 것 같다.

2.4. 성능

Puzzle Size	# Variable	# Clause	Time (s)
6*6 (숙제 예제)	36	152	0.109
10*9	90	1044	11.089
13*10	130	3062	132.614
15*15	-	-	-

13*10 은 2 분정도 걸리는 것을 확인하였고,
15*15 부터는 너무 오래걸려서 시간을 측정하지 못했다.
Clause 의 개수는 중복된 clause 를 걸러내지 않아 약간 크지만 전체 성능에는 큰 영향을 주지않는다.

2.4. 한계점과 향상 방향

2.4.1 한계점

현재 구현은 memoization 을 사용하기 때문에 정형화된 formula 가 있어야 구현하기에 용이하였다. 하지만 constraint 생성이 성능에 병목지점인데, 특히 CNF 와 CNF 를 병합하는 과정에서 DISTR 연산이 계속 쓰이고 이 부분에서 clause 가 계속 늘어나는 것이 문제다. n 이 row 개수, C 가 clause 의 개수일 때, C 이 보통 200~300 정도 되고 DISTR 연산을 300 개의 clause 를 갖는 n 개의 CNF 에 대해서 하기 때문에 이 부분에서 시간/공간이 너무 많이 소모된다.

2.4.2 향상 방향

memoization 을 하는 것은 상관없지만 CNF formula 로 상태를 저장하는 것은 옳지 않다. NNF form 으로 저장을하고 DISTR 연산은 row/column 의 constraint 을 모두 모은 후에 해야한다. 이 때 어떤 variable 은 true/false 둘 중 하나로 결정되는 경우가 있는데, 이 정보를 활용하여 clause 에서 그 variable 과 연관된 clause 들을 수정하여 최적화 할 수 있을 것이다.
Bitwise-operator 를 사용하는 것 보다, 중복된 clause 를 제거하는 것이 더 효과적일 것으로 예상된다. 따라서 알파벳 순서로 clause 를 정렬하여 중복을 빠르게 제거하는 데이터 구조를 사용하면 더 좋을 것 같다.