PA1 Report

软73 金昕祺 2016010524

1 作业基本情况介绍

1.1 提交文件组织方式

```
pa1/
    dpll/
        CMakeLists.txt
        DPLL.cpp
        DPLL.h
        DimacsParser.h
        common.h
        CHRONOLOGICAL_BACKTRACK_DPLL.cpp
        CHRONOLOGICAL_BACKTRACK_DPLL.h
        compare.cpp
        main.cpp
        main2.cpp
        report.pdf
        tests/
            queen5.dimacs
            queen10.dimacs
            queen15.dimacs
        test_code/
            generate_NQUEEN_cnf.py
            random_cnf_test.py
```

其中，DPLL.h和DPLL.cpp即为实现了Backjump和Conflict Clause Generation的DPLL，且对外提供的接口（即check_sat和get_model函数）并未修改。因而助教可以直接使用自己的main.cpp文件替代本人提交的main.cpp文件，无需修改CMakeLists.txt，编译后生成的可执行程序dpll即为和助教提供的main.cpp相对应的可执行程序。

1.2 如何运行程序

1.2.1 如何编译得到可执行程序

解压提交的压缩包后，在解压得到的文件夹内pa1/dpll目录下运行如下命令即可生成三个可执行程序（CHRONOLOGICAL_BACKTRACK_DPLL、dpll、compare）：

```
mkdir release
cd release
cmake -DCMAKE_BUILD_TYPE=Release ..
make
```

1.2.2 使用编译生成的可执行程序

在编译得到可执行程序后，若想使用支持Backjump和Conflict Clause Generation的DPLL来处理CNF，可以在pa1/dpll目录下执行如下命令(其中命令行参数数量即argc只要大于等于2即可)：

```
./release/dpll [path_to_dimacs_file1] [path_to_dimacs_file2] ...
```

若想使用原始的backtrack的DPLL来处理CNF，可以在pa1/dpll目录下执行如下命令(其中命令行参数数量即argc只要大于等于2即可)：

```
./release/CHRONOLOGICAL_BACKTRACK_DPLL [path_to_dimacs_file1] [path_to_dimacs_file2] ...
```

若想比较原始的backtrack的DPLL和支持Backjump/Conflict Clause Generation的DPLL的运行速度，可以在pa1/dpll目录下执行如下命令(其中命令行参数数量即argc只要大于等于2即可)：

```
./release/compare [path_to_dimacs_file1] [path_to_dimacs_file2] ...
```

1.2.3 如何使用test_code文件夹下的测试文件

test_code文件夹下有两个文件，其中random_cnf_test.py用于自动生成数据来测试本人的DPLL实现是否正确，generate_NQUEEN_cnf.py用于生成N皇后问题对应的CNF并保存在.dimacs文件里。本人使用python3.5可执行这两个python文件，在pa1/dpll/test_code目录下打开终端输入如下命令即可执行这两个py文件（**需要注意的是，若想要运行random_cnf_test.py，需要在系统里提前安装好minisat并能在命令行里直接输入minisat调用相应程序，若助教运行时出现问题可联系本人**）：

```
python3 generate_NQUEEN_cnf.py
python3 random_cnf_test.py -p ../release/dpll -e 100
```

想要了解命令行参数的作用，可以在pa1/dpll/test_code目录下打开终端输入命令：

```
python3 random_cnf_test.py —help
```

2 算法实现方式

2.1 数据结构介绍

DPLL.h定义了结构体node，对应implication graph中的节点，其assigned属性表明该节点是否被分配了真值，value属性记录分配的真值，decision_level属性为该节点的决策层次（decision_level范围为-1, 0, 1, … ；decision_level为-1的节点对应由只含有一个literal的clause按照unit propagation rule来决定value属性取值的命题变量），antecedent属性表明该节点的值通过对哪个子句进行unit propagation得到。

DPLL.h还定义了结构体graph，该结构体由两个数组成员nodes、edges，其中nodes为(num_variables +1)维node类型数组（nodes[0]对应conflict node，而对于大于0的i则是nodes[i]对应第i个命题变量），edges为$(num_variables+1)^2$维bool类型数组（$edges[i*(num_variables+1)+j]$为True当且仅当implication graph中存在从nodes[i]到nodes[j]的边）。

DPLL.h还声明了DPLL类，该类使用一个graph类型的成员变量_graph来存储implication graph。

2.2 具体实现

DPLL类的check_sat函数如下：

```
bool DPLL::check_sat() {
    while(1){
        while(exists_unit()){};
        decide();
        if(conflict()){
```

```
            if(! has_decision()){
                return false;
            }
        }else if(sat()){
            return true;
        }
    }
}
```

可以看到，在conflict函数检测到冲突的存在后，会调用has_decision方法，而has_decision方法在检测到决策变量后会执行backjump和conflict clause generation。实现backjump和conflict clause generation的步骤可以归纳如下：

步骤（1）：

首先从conflict node（记为κ）开始生成一个节点集合$causes_of(\kappa)$。

为了形式化地描述该集合，我们定义如下的一些函数和符号：

对于implication graph中的任意节点x，定义

$$A(x) = \{y | there\ is\ an\ edge\ from\ node\ y\ to\ node\ x\ in\ the\ implication\ graph\}$$
$$\Lambda(x) = \{y \in A(x) | y.decision\_level < x.decision\_level\}$$
$$\Sigma(x) = \{y | y.decision\_level = x.decision\_level\}$$

由于我在往implication graph中添加每一个由unit propagation rule决定value属性的节点时，都让该节点的decision level等于图中所有通过边指向它的节点（即应用unit propagation rule的clause的其他literal对应的节点）的decision level当中的最大值，所以$(A(x) = \Lambda(x) \cup \Sigma(x)) \wedge (\Lambda(x) \cap \Sigma(x) = \varnothing)$。对于implication graph中的任意节点x，我们还可以定义：

$$causes\_of(x) = \begin{cases} x & if\ A(x) = \varnothing \\ \Lambda(x) \cup [\cup_{y \in \Sigma(x)} causes\_of(y)] & otherwise \end{cases}$$

根据上述定义，我们即可得到想要的节点集合$causes_of(\kappa)$。

步骤（2）：

记$max_decision_level = max(\{y.decision_level | y \in causes_of(\kappa)\})$。若$max_decision_level == -1$，则按照本报告2.1节中提到的规定，可知产生该冲突的所有相关的命题变量的取值都是被unit clause（即只有一个literal的clause）决定的。此时冲突无法避免，直接判断该CNF是不可满足的，无需进行下一个步骤；否则继续执行下面的步骤(3)。

步骤（3）：

从集合$causes_of(\kappa)$中挑选出一个decision level最大的节点y（若有多个decesion level最大的节点时随机调出一个），让$causes_of(\kappa) - \{y\}$中的所有节点都产生一条指向节点y的边，增加一条新的子句$\vee_{n \in causes_of(\kappa)} \neg(n.value)$，然后再修改$y.value = \neg(y.value)$。最后，删除implication graph中节点y的所有子节点（y的子节点是指存在从y到该节点的路径的节点，这里的路径是指一系列首尾相接的边；删除子节点包括删除和这些子节点关联的任意边、修改子节点的assigned属性等）。

3 实验

3.1 实验环境

操作系统：基于VMware Fusion虚拟机的Ubuntu 16.04.6 LTS

硬件配置：基于VMware Fusion虚拟机，4096Mb内存，2个处理器内核。

第三方依赖：如果想进行报告3.3节中的正确性实验，需要使用apt安装minisat，并确保设置好环境变量，使得可以在任何工作目录下打开终端输入'minisat'都能直接启动minisat。

3.2 性能实验

性能实验的步骤如下：

（1）使用cmake编译程序，具体步骤见本报告1.2.1节

（2）在pa1/dpll目录下打开终端，执行如下命令：

```
./release/compare tests/queen5.dimacs tests/queen10.dimacs tests/queen15.dimacs
```

最终的实验结果如下表所示：

测试文件	原始DPLL耗时(ms)	基于Backjump和Conflict Clause Generation的DPLL耗时(ms)
queen5.dimacs	0.136524	0.029939
queen10.dimacs	12.0674	3.35687
queen15.dimacs	448.461	220.34

从实验结果可以发现，基于Backjump和Conflict Clause Generation的DPLL在运行速度上明显快于原始的DPLL。

3.3 正确性实验

本实验的目的是使用随机生成的CNF，检查本人实现的DPLL SAT Solver是否正确。

实验基于pa1/dpll/test_code文件夹下的random_cnf_test.py。执行该文件要求系统里提前安装好minisat并配置好环境变量，保证在任意工作目录下能打开终端输入minisat调用minsiat(**若助教运行时出现问题可联系本人**)。

random_cnf_test.py接受的命令行参数如下：

```
optional arguments:
  -h, --help            show this help message and exit
  -p PATH, --path PATH  path to executable SAT solver
  -e EPOCHS, --epochs EPOCHS
                        epochs to test
  --min_num_variables MIN_NUM_VARIABLES
```

```
                    minimum number of variables
--max_num_variables MAX_NUM_VARIABLES
                    maximum number of variables
--min_num_clauses MIN_NUM_CLAUSES
                    minimum number of clauses
--max_num_clauses MAX_NUM_CLAUSES
                    maximum number of clauses
```

本人在测试时的步骤如下：

（1）使用cmake编译程序，具体步骤见本报告1.2.1节

（2）在pa1/dpll/test_code目录下打开终端，执行如下命令：

```
python3 random_cnf_test.py -p ../release/dpll --epochs 100
```

经过本人测试，测试程序并未报错，说明对于100个随机生成的CNF，我实现的程序和通过apt安装的minisat程序在可满足性的判断上是一致的，并且在可满足的情况下本人实现的程序提供了能使CNF为True的assignment。