DPLL 实验报告

隋唯一2017011430

2020 年 3 月 12 日

1 实验目标

实现一个简单的DPLL算法，对输入的cnf范式判定其可满足性，并给出一个使其为真的解释。

2 实验思路

DPLL算法基于深度优先搜索，所以总的时间复杂度是$O(2^n)$，关键是如何在细节上优化。

一个最直观的想法就是对所有的变量，逐一实验其所有的解释。比如，假定有三个变量V_1, V_2, V_3，先令其全部为true （当前指针指向V_3)，检查cnf是否为真。若为真，返回true 以及当前的状态；若为假，指针暂不回退，指针指向的变量为false，再进行对cnf的检查。此时由于指针指向的变量true与false两个值都已经试验过了，所以当前变量（或说节点）置为“脏”，指针回退一步，指向V_2，并令V_2为false，V_3重新置为true。如此循环，直到找到令cnf为真的解释，或所有可能都已经尝试。

然而这么做有一个问题，指针的回退不够智能，这种每次只回退一个节点，且直到所有节点都已经赋值才开始检查cnf的算法实际上等价于暴力遍历。比如，对于三个变量的情况，相当于遍历从000 到111的所有二进制数。

对于一个cnf，如果当前能够判断有一个子句是假，那么整个cnf都为假，指针就可以回退。对于cnf的一个子句，如果能判断一个literature为真，那这个子句就是真，可以跳出这个子句，进行对下一个子句的判断。也就是说，某些情况下，没有确定所有变量的取值，就可以判定cnf为假（此时指针回退），或cnf 为真（返回解）。

指针回退的目的地，应当是当前节点上溯，遇到的第一个还可以尝试的祖先节点（就是说，这个节点并没“脏”）。

3 算法实现

3.1 存储结构

```
std::unordered_map<int,int> atomStatus;//0,1,2
int clause_num = phi.clauses.size();//number of clauses
int atomNum = phi.num_variable;//number of atoms
for(int i=1;i<=atomNum;i++)
    result.insert(std::make_pair(i,true));
int* nodeStatus = new int[atomNum];
for(int i=0;i<atomNum;i++)
    nodeStatus[i]=0;
int ptr = 0;//point to current node
```

这里，每一个atom就是一个节点，每个节点有三个状态0，1，2，0代表这个节点被刚刚扩展，还没有尝试true或false；1代表这个节点已经尝试过false，如果在这个节点上再扩展应该把它置为true；2代表这个节点的true与false都已经试过了，都不能令cnf为真，应该回溯。使用atomStatus存储节点的状态。

3.2 整体框架

```
while(true){
        if(nodeStatus[ptr]==2)
            break;
        else if(nodeStatus[ptr]==0) {
            nodeStatus[ptr]++;
            result[ptr + 1] = false;
        }
        else {
            nodeStatus[ptr]++;
            result[ptr + 1] = true;
        }
        //check if the current interpret satisfies the CNF
                ...
        if(wholeValue)
            solveStatus=1;
        //now the interpret has been checked, solveStatus has been set
```

```
        if(solveStatus==0)//not satisfy, back up
        {
            while(ptr>0&&nodeStatus[ptr]==2)
                ptr--;
            for(int i=ptr+1;i<atomNum;i++)
                nodeStatus[i]=0;
        }
        else if(solveStatus==1)//satisfy!
            return true;
        else ptr++;//not sure yet, check the next node
    }
    return false;//no satisfying interpret, return unsat
```

这是一个典型的dfs框架，但是并没有维护一个二叉链表，而是用atomStatus维护搜索树。注意，指针回退到某一个变量时，这个变量之后的所有变量的atomStatus都置为0，因为此时它们的祖先发生了变化。

3.3 对特定解释的判断

```
        //check the interpret
        bool wholeValue = true;//the value of the whole cnf
        for(int i=0;i<clause_num;i++) //for each clause
        {
            bool currValue=false;//is the current clause false?
            bool any_notsure=false;//any literature not sured?
            int len = phi.clauses[i].size();
            for(int j=0;j<len;j++)
            {
                int currvar = phi.clauses[i][j];
                if(VAR(currvar)<=ptr+1)
                {
                    if((POSITIVE(currvar)&&result[currvar])||(NEGATIVE(currvar)&
                        currValue=true;
                        break;
                    }
                }
                else{
```

```
                any_notsure=true;
            }
        }
        wholeValue=wholeValue&&currValue;
        if(currValue||any_notsure){
            continue;
        }
        else{
            solveStatus=0;
            break;
        }
    }
    if(wholeValue)
        solveStatus=1;
    //check over
```

这里的判断思路就是二层循环，外循环是每一个子句，内循环是子句中的每一个literature。每一个内循环中，有一个literature为真，这个子句为真，循环结束。为循环中，有一个子句为假，cnf为假，外循环结束。最后的结果分为真、假、不确定。若为真，返回结果；若为假，指针回退；若不确定，指针加一，向下扩展。

4 改进空间

首先，针对存在不出现的atom的情况可以优化
其次，对于指针的回退，可以更加智能。

5 实验心得

通过本次实验，我对于dfs的理解更加深入，同时对搜索的使用更加灵活，不局限于二叉链表。