软件分析与验证 PA1 报告

李晨昊 2017011466

2020 年 3 月 4 日

目录

1 简介

我删除了`DPLL.cpp`文件 (同时也在`CMakeList.txt`中去掉了它),而把逻辑都是现在了`DPLL.h`文件中，这主要是个人习惯因素。

`DPLL.h.basic`文件中包含了无 backjump 版本的 DPLL 求解器，把现在的`DPLL.h`替换成它即可运行无 backjump 版本的求解器。

`bitset.hpp`文件中实现了一个简单的`bitset`，backjump 版本的求解器中用到了它。

2 基础部分实现思路

2.1 数据结构

我定义了如下基础数据结构:

1.
```
struct Var {
  u32 id : 31;
  u32 flag : 1;
};
```

Var逻辑上表示一个整数和布尔值的二元组，只是用 bitfield 来节约了一点空间。它在不同的环境中有不同的含义，例如它可以表示子句中的一个文字，用id表示变量编号，flag表示是否有否定 (为了方便，我没有使用 dimacs 中的带符号整数来表示字面值)。

2.
```
struct VarInfo {
  enum {
    Undef = 0, Pos = 1, Neg = -1
  } state;
  std::vector<u32> pos_clauses;
  std::vector<u32> neg_clauses;
};
```

VarInfo记录一个变量的状态，state表示当前的部分解释中该变量的值，pos_clauses和neg_clauses存储这个变量在哪些子句中出现了，分别表示以正文字和负文字形式出现。它们用来协助进行 unit propagation。

2.2 预处理

预处理部分主要做以下几个操作：

1. 将带符号整数表示的文字转化成Var表示的文字。
2. 做一个简单的优化：处理所有由单个文字组成的子句，为了让整个式子成立，子句的值必须为真，所以可以立即确定这个文字的变量的值。对于其它包含这个变量的子句，如果对应文字为真则删除这个子句，如果为假则删除这个文字。这个操作需要反复进行，直到不存在单个文字组成的子句。

   这个操作的主要目的其实不是优化，而是为了方便后续处理，后面会提到。
3. 维护每个变量的pos_clauses和neg_clauses。

2.3 核心算法

大致的伪代码如下：

```
while exist undefined variable x
  seq = []
  stk = [(x, true)]
  is_decision = true
  while stk is not empty
    (var, pos) = stk.pop()
    if var is defined
      if the old definition is conflict with pos
        if exist last definition d
          remove seq[index of d : end of seq]
          stk = [(var(d), ~pos(d))]
          continue
        else
          return false
    else
      seq += (x, is_decision)
      is_decision = false
      vars[x] = pos
      unit prop on x, push newly inferred var to stk
return true
```

在进行 unit prop 的过程中维护一个栈，确保能够一次性找到所有传播的变量。

算法不会主动检测当前解释是否满足模型，判断冲突的唯一方法是：一个用 unit prop 得到的变量值与记录在`vars`中的值冲突。这样之所以成立，是因为预处理阶段保证了不存在只包含一个文字的子句，假设当前解释不满足模型，那么必然存在一个子句中所有变量都有定义，但当倒数第二个变量被定义的时候，必然可以用 unit prop 确定最后一个变量的值，从而发现冲突。

3 backjump 部分实现思路

对于一个冲突 P 和 $\neg P$ 我选择的冲突节点是：推导出 P 和 $\neg P$ 的所有决策变量的并集。为了得到这个信息，(逻辑上) 为每个变量维护一个集合 $decisions$，在上面的伪代码中，在`vars`中记录值的同时维护集合，对于决策变量 x，值为 $\{x\}$；对于由子句 cls 进行 unit

prop 得到的变量 x，值为 $\bigcup_{lit \in cls, var(lit) \neq x} decisions(var(lit))$。

实际实现的时候，用一个长度为 $|Vars|$ 的`bitset`来表示集合。为了更进一步优化性能，实现中不是为每个变量申请`bitset`的空间，而是为所有变量申请一块空间，用变量的偏移量在其中访问。

backjump 与基础版本中的 backtrack 在同一实际被调用，大致的伪代码为：

```
if exist last definition d
  assert d is in conflict node set
  last = index of the first decision variable in seq, that is
    - before d (inclusive)
    - after the first variable before d & in conflict node set (exlusive)
  construct new_cls from conflict node set and old state
  update pos_clauses & neg_clauses according to new_cls
  add new_cls to clauses
  reset the states of vars in seq[last : end of seq]
  remove seq[last : end of seq]
  stk = [(var(d), ~pos(d))]
  continue
else
  return false
```

4 性能比较

我在网络上寻找了一些测例，从中选出了无 backjump 版本和包含 backjump 版本的性能表现差距比较大的几个：

	backtrack	backjump
uf100-0250	745.393ms	1.34635ms
uf100-0842	128.788ms	0.329696ms
uf100-0469	298.535ms	0.797293ms

不过仍然需要指出的是，对于许多测例 backjump 版本没有特别明显的性能优势，甚至在常数上稍劣一些，这可能与我选择冲突节点的策略有关，如果采用更复杂的策略也许会有更好的效果。

5 参考资料

https://www.cs.ubc.ca/ hoos/SATLIB/benchm.html