命题逻辑

赵耀

命题逻辑

- 形如 ¬P，P∧Q，P∨Q，P → Q ，P↔Q的语句，值为True或者False
- 推理规则较简单，往往通过（1.真值表 2.为数不多的推理规则，例如Modus Ponens等几个）
- 缺点，不能或者很难表示复杂的语句，不能记录推理过程中的变化

逻辑表达的重点

- 合取范式的转换
- 合取范式的重大意义就是让定义一个逻辑问题的搜索空间成为了可能，很自然的，逻辑问题可以转换为通过回溯的方法求解

合取范式的转换（回顾）

$B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$

1. Eliminate $\Leftrightarrow$, replacing $\alpha \Leftrightarrow \beta$ with $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$.

   $(B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})) \wedge ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1})$

2. Eliminate $\Rightarrow$, replacing $\alpha \Rightarrow \beta$ with $\neg\alpha \vee \beta$.

   $(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg(P_{1,2} \vee P_{2,1}) \vee B_{1,1})$

3. Move $\neg$ inwards using de Morgan's rules and double-negation:

   $(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge ((\neg P_{1,2} \wedge \neg P_{2,1}) \vee B_{1,1})$

4. Apply distributivity law ($\vee$ over $\wedge$) and flatten:

   $(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg P_{1,2} \vee B_{1,1}) \wedge (\neg P_{2,1} \vee B_{1,1})$

回顾

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
false	false	true	false	false	true	true
false	true	true	false	true	true	false
true	false	false	false	true	false	false
true	true	false	true	true	true	true

$$\neg P \vee Q$$

$$(\neg P \wedge \neg Q) \vee (P \wedge Q)$$

$$(\neg P \vee Q) \wedge (P \vee \neg Q)$$

将连接词在Python中重载（一种处理方式，逻辑表达式在代码中的表达）

Operation	Book	Python Infix Input	Python Output	Python `Expr` Input
Negation	$\neg P$	`~P`	`~P`	`Expr('~', P)`
And	$P \wedge Q$	`P & Q`	`P & Q`	`Expr('&', P, Q)`
Or	$P \vee Q$	`P | Q`	`P | Q`	`Expr('|', P, Q)`
Inequality (Xor)	$P \neq Q$	`P ^ Q`	`P ^ Q`	`Expr('^', P, Q)`
Implication	$P \rightarrow Q$	`P |'==>'| Q`	`P ==> Q`	`Expr('==>', P, Q)`
Reverse Implication	$Q \leftarrow P$	`Q |'<=='| P`	`Q <== P`	`Expr('<==', Q, P)`
Equivalence	$P \leftrightarrow Q$	`P |'<=>'| Q`	`P <=> Q`	`Expr('<=>', P, Q)`

合取范式的转换

$B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$

1. Eliminate $\Leftrightarrow$, replacing $\alpha \Leftrightarrow \beta$ with $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$.

   $(B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})) \wedge ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1})$

2. Eliminate $\Rightarrow$, replacing $\alpha \Rightarrow \beta$ with $\neg\alpha \vee \beta$.

   $(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg(P_{1,2} \vee P_{2,1}) \vee B_{1,1})$

3. Move $\neg$ inwards using de Morgan's rules and double-negation:

   $(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge ((\neg P_{1,2} \wedge \neg P_{2,1}) \vee B_{1,1})$

4. Apply distributivity law ($\vee$ over $\wedge$) and flatten:

   $(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg P_{1,2} \vee B_{1,1}) \wedge (\neg P_{2,1} \vee B_{1,1})$

```python
def to_cnf(s):
    """Convert a propositional logical sentence to conjunctive normal form.
    That is, to the form ((A | ~B | ...) & (B | C | ...) & ...) [p. 253]
    >>> to_cnf('~(B | C)')
    (~B & ~C)
    """
    s = expr(s)
    if isinstance(s, str):
        s = expr(s)
    s = eliminate_implications(s)  # Steps 1, 2 from p. 253
    s = move_not_inwards(s)  # Step 3 p. 254
    return distribute_and_over_or(s)  # Step 4 p. 254
```

消除等价和蕴涵连接词

```python
def eliminate_implications(s):
    """Change implications into equivalent form with only &, |, and ~ as logical operators."""
    s = expr(s)
    if not s.args or is_symbol(s.op):
        return s  # Atoms are unchanged.
    args = list(map(eliminate_implications, s.args))
    a, b = args[0], args[-1]
    if s.op == '==>':
        return b | ~a
    elif s.op == '<==':
        return a | ~b
    elif s.op == '<=>':
        return (a | ~b) & (b | ~a)
    elif s.op == '^':
        assert len(args) == 2  # TODO: relax this restriction
        return (a & ~b) | (~a & b)
    else:
        assert s.op in ('&', '|', '~')
        return Expr(s.op, *args)
```

将否定词内移

```python
def move_not_inwards(s):
    """Rewrite sentence s by moving negation sign inward.
    >>> move_not_inwards(~(A | B))
    (~A & ~B)"""
    s = expr(s)
    if s.op == '~':
        def NOT(b):
            return move_not_inwards(~b)
        a = s.args[0]
        if a.op == '~':
            return move_not_inwards(a.args[0])  # ~~A ==> A
        if a.op == '&':
            return associate('|', list(map(NOT, a.args)))
        if a.op == '|':
            return associate('&', list(map(NOT, a.args)))
        return s
    elif is_symbol(s.op) or not s.args:
        return s
    else:
        return Expr(s.op, *list(map(move_not_inwards, s.args)))
```

根据分配率，变成CNF

```python
def distribute_and_over_or(s):
    """Given a sentence s consisting of conjunctions and disjunctions
    of literals, return an equivalent sentence in CNF.
    >>> distribute_and_over_or((A & B) | C)
    ((A | C) & (B | C))
    """
    s = expr(s)
    if s.op == '|':
        s = associate('|', s.args)
        if s.op != '|':
            return distribute_and_over_or(s)
        if len(s.args) == 0:
            return False
        if len(s.args) == 1:
            return distribute_and_over_or(s.args[0])
        conj = first(arg for arg in s.args if arg.op == '&')
        if not conj:
            return s
        others = [a for a in s.args if a is not conj]
        rest = associate('|', others)
        return associate('&', [distribute_and_over_or(c | rest)
                               for c in conj.args])
    elif s.op == '&':
        return associate('&', list(map(distribute_and_over_or, s.args)))
    else:
        return s
```

运行的例子

```python
from utils import *
from logic import *
s= expr("(B11 <=> (P12 | P21))  &  ~B11")
print(s)
s_cnf = to_cnf(s)
print(s_cnf)
clauses = conjuncts(s_cnf)
print(clauses)
```

```
((B11 <=> (P12 | P21)) & ~B11)
((~P12 | B11) & (~P21 | B11) & (P12 | P21 | ~B11) & ~B11)
[(~P12 | B11), (~P21 | B11), (P12 | P21 | ~B11), ~B11]
```

表达式

合取范式

合取范式中的各子句

SAT问题

SAT:(Boolean) Satisfiability Problem
检验命题语句的可满足性

比如：如下的表达式是否可能为true，为true时，model是怎样？

```
~P11 & (B11 <=> (P12 | P21)) & (B21 <=>(P11 | P22 | P31)) &  ~B11 & B21
```

所谓model，即是上述表达式所包含符号的一组取值

比如，要想让上述表达式为true， 一组可能的取值为
[P11: False, B11: False, P12: False, P21: False, B21: True, P31: True]
这就是一个model
回溯过程中给符号取值的过程，可以认为是建立model的过程

联想CSP中变量
以及对变量赋值的过程

DPLL

```python
def dpll_satisfiable(s):
    """Check satisfiability of a propositional sentence.
    This differs from the book code in two ways: (1) it returns a model
    rather than True when it succeeds; this is more useful. (2) The
    function find_pure_symbol is passed a list of unknown clauses, rather
    than a list of all clauses and the model; this is more efficient."""
    clauses = conjuncts(to_cnf(s))
    symbols = prop_symbols(s)
    return dpll(clauses, symbols, {})


def dpll(clauses, symbols, model):
    """See if the clauses are true in a partial model."""
    unknown_clauses = []  # clauses with an unknown truth value
    for c in clauses:
        val = pl_true(c, model)
        if val is False:
            return False
        if val is not True:
            unknown_clauses.append(c)
    if not unknown_clauses:
        return model
    P, value = find_pure_symbol(symbols, unknown_clauses)
    if P:
        return dpll(clauses, removeall(P, symbols), extend(model, P, value))
    P, value = find_unit_clause(clauses, model)
    if P:
        return dpll(clauses, removeall(P, symbols), extend(model, P, value))
    if not symbols:
        raise TypeError("Argument should be of the type Expr.")
    P, symbols = symbols[0], symbols[1:]
    return (dpll(clauses, symbols, extend(model, P, True)) or
            dpll(clauses, symbols, extend(model, P, False)))
```

DPLL回溯的过程

- 如果存在已经返回的某个子句为false，那么可以整体返回false，
- 如果所有子句在该模型均已确定为true，那么可以整体返回true，
- 优先寻找纯符号：比如所有子句中如果只有A，那么可以直接将A赋值为true；如果所有子句中只有~B,那么可以直接将B赋值为false。此处的所有子句将忽略模型建立以来可以判定为真的子句。
- 优先寻找单元子句：只有一个文字的子句。当然，如果一个子句除了一个文字其他文字都已经赋值为false了，那么这样的子句也可以认为是单元子句。比如A是一个单元子句，~A也是一个单元子句，当A已经被赋值为false的情况下，A|~B也是一个单元子句，因为此时B必须得是false了；已知A,B都是false的情况下，A|B|C也是一个单元子句，因为C必须得是true了。此处的过程有点像约束传播，被称为单元传播。
- 以上符号都没有找到，寻找下一个符号
- 尝试该符号所有可能的赋值，递归调用下一层（当然只可能有2种赋值，True或False）。

联想CSP优化：提前终止回溯

联想CSP的最小剩余值的选择思路

DPLL示例：转成CNF，提取所有合取的子句

```
~P11 & (B11 <=> (P12 | P21)) & (B21 <=>(P11 | P22 | P31)) &  ~B11 & B21
```

~P11

~P12 | B11

~P21 | B11

~P22 | B21

~P31 | B21

~P11 | B21

~B11

B21

clauses

P12 | P21 | ~B11

P11 | P22 | P31 | ~B21

symbols:

P31, P22, B21, P12, P21, P11, B11

将合取范式的每一个子句都提取出来
上述每个子句都为true，整个表达式才有可能为true

DPLL示例：优先选择纯符号，单元子句

有纯符号吗？

~P11

~P12 | B11

~P21 | B11

~P22 | B21

~P31 | B21

~P11 | B21

~B11

B21

unknown clauses

P12 | P21 | ~B11

P11 | P22 | P31 | ~B21

symbols:

P31, P22, B21, P12, P21, P11, B11

model:

{}

DPLL示例：优先选择纯符号，单元子句

单元子句

~P11

~P12 | B11

~P21 | B11

~P22 | B21

~P31 | B21

~~~P11 | B21~~

true

~B11

B21

unknown clauses

P12 | P21 | ~B11

P11 | P22 | P31 | ~B21

P22 | P31 | ~B21

symbols:

P31, P22, B21, P12, P21, ~~P11,~~ B11

model:

{P11:False}
~~~

DPLL示例：优先选择纯符号，单元子句

单元子句

~P11

~P12 | B11

~P21 | B11

~P22 | B21

~P31 | B21

~B11

B21

~P11 | B21

~P12

~P21

unknown clauses

P12 | P21 | ~B11

True

P11 | P22 | P31 | ~B21

P22 | P31 | ~B21

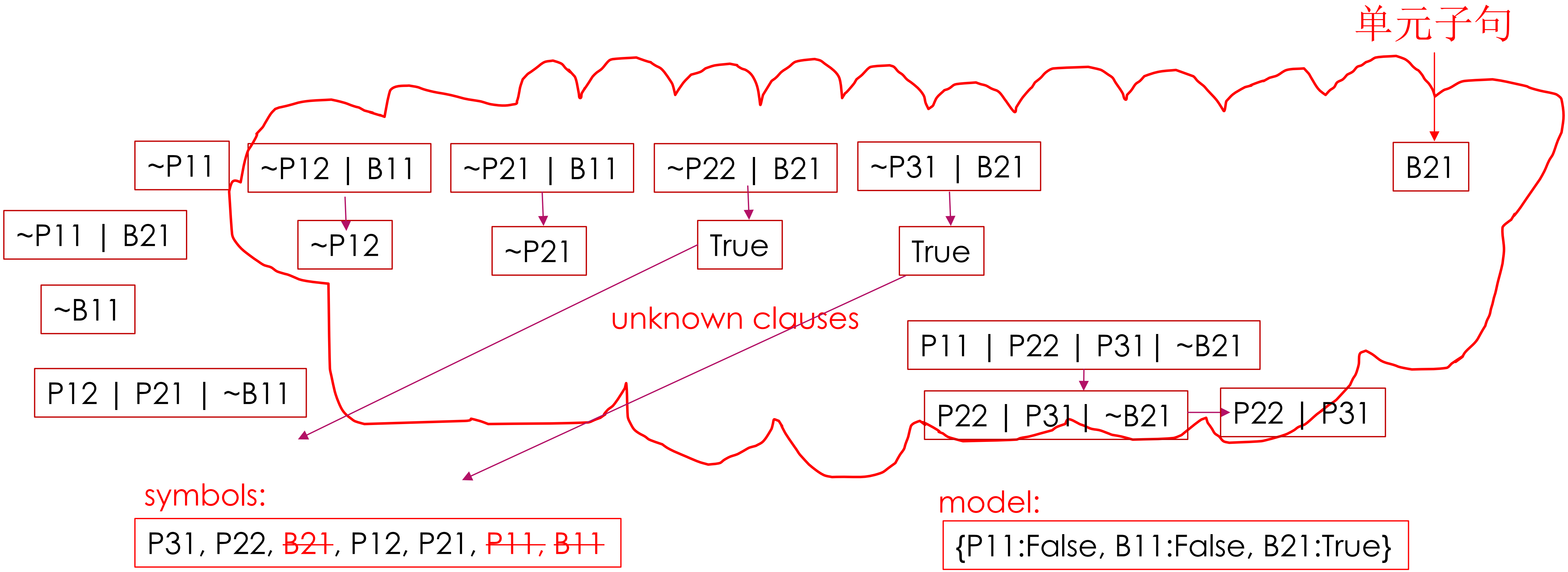
symbols:

P31, P22, B21, P12, P21, ~~P11, B11~~

model:

{P11:False, B11:False}

DPLL示例：优先选择纯符号，单元子句



DPLL示例：优先选择纯符号，单元子句

~P11

~P11 | B21

~B11

P12 | P21 | ~B11

~P22 | B21

~P31 | B21

B21

单元子句

~P12 | B11

~P12

~P21 | B11

~P21

unknown clauses

P11 | P22 | P31 | ~B21

P22 | P31 | ~B21

P22 | P31

symbols:

P31, P22, ~~B21~~, ~~P12~~, P21, ~~P11, B11~~

model:

{P11:False, B11:False, B21:True，P12:False}

DPLL示例：优先选择纯符号，单元子句

~P12 | B11

~P11

~P11 | B21

~B11

P12 | P21 | ~B11

~P22 | B21

~P31 | B21

B21

单元子句

~P21 | B11

~P21

unknown clauses

P11 | P22 | P31 | ~B21

P22 | P31 | ~B21

P22 | P31

symbols:

P31, P22, ~~B21~~, ~~P12~~, ~~P21~~, ~~P11, B11~~

model:

{P11:False, B11:False, B21:True,P12:False,P21:False}

DPLL示例：没有纯符号，单元变量，选择一个变量，先赋值使之为true

~P11

~P11 | B21

~P12 | B11

~P21 | B11

~B11

P12 | P21 | ~B11

~P22 | B21

~P31 | B21

B21

没有单元子句！

unknown clauses

P11 | P22 | P31| ~B21

P22 | P31| ~B21

P22 | P31

True

symbols:

选中，赋值为true

P31, P22, ~~B21~~, ~~P12~~, ~~P21~~, ~~P11~~, ~~B11~~

model:

{P11:False, B11:False, B21:True,P12:False,P21:False,P22:True}

DPLL示例：没有纯符号，单元变量，选择一个变量，先赋值使之为true

~P11

~P11 | B21

~P12 | B11

~B11

~P21 | B11

P12 | P21 | ~B11

~P22 | B21

~P31 | B21

B21

没有单元子句！

unknown clauses

P11 | P22 | P31| ~B21

P22 | P31| ~B21

P22 | P31

symbols:

选中，赋值为true

P31, P22, ~~B21~~, ~~P12~~, ~~P21~~, ~~P11~~, ~~B11~~

model:

{P11:False, B11:False, B21:True,P12:False,P21:False,P22:True}

DPLL示例：没有纯符号，单元变量，选择一个变量，先赋值使之为true

~P11

~P11 | B21

~P12 | B11

~P21 | B11

~B11

P12 | P21 | ~B11

~P22 | B21

~P31 | B21

B21

P11 | P22 | P31 | ~B21

剩余子句集合为空
返回model

unknown clauses

symbols:

P31, ~~P22~~, ~~B21~~, ~~P12~~, ~~P21~~, ~~P11, B11~~

model:

解（只是其中的一个）

{P11:False, B11:False, B21:True,P12:False,P21:False,P22:True}

总结

- 以上示例比较简单，但是可以体会出两点：1、提前终止回溯，比如最后剩余符号P31，但是其取值已经不影响最终结果；2、优先选择纯符号以及单元子句，对于剪枝的作用，体会单元传播的过程。
- 对于大型的问题，还可以有更多优化的方向。

DPLL优化

- 变量和值优先选择（当找不到纯符号或单元子句的时候）：比如变量可以优先选择剩余子句中出现得最频繁的变量；比如总是先赋值为true然后是false。（联想CSP的最少约束变量启发式）
- 成分分析：当某些符号已经被赋值，可能子句集会变成不相交的子集。比如原始子句为[(A | B | C), (~B | C),(C | D | E),(~D | C)]，当C作为纯符号被赋值为true，则子句集可以拆分为2个子成分，一个是[(A | B),(~B)],另一个是[(D | E),(~D)],分别对2个子成分独立求解，可以加快求解速度
- 智能回溯：直接回溯到导致冲突的相关点（记录冲突集）
- 智能索引：加快索引查找到“符号P以正文字（或负文字）出现在所有的剩余子句集”
- 随机重新开始：有时一轮运行看起来没有任何进展，此时可以选择从搜索树顶端重新开始，要好过从原路继续。重新开始后，会做出不同的随机选择（比如变量或值）。之前的记录仍然保留，可以帮助剪枝。

思考

- DPLL与CSP的异同

Thank You