**布林可滿足性問題（Boolean satisfiability problem；SAT ）**

**屬於哪種?(NP,NP-complete,NP-hard)**

這個問題屬於NP-complete。[1]

**定義/解釋**

布林滿足性問題，就是回答「問題是否可以用布林描述，並可以被滿足」的問題，以兩人猜拳為例，問題就會變成:兩人猜拳是否可以用布林描述並可被滿足。

|  |  |  |  |
| --- | --- | --- | --- |
| P1/P2 | 剪刀 | 石頭 | 布 |
| 剪刀 |  | P1 | P2 |
| 石頭 | P2 |  | P1 |
| 布 | P1 | P2 |  |

這是以直觀的方式，用表格呈現出所有可能出現的輸贏的結果，但若要以布林表示的話，就只會有對或錯，變成以下這樣:

|  |  |  |  |
| --- | --- | --- | --- |
| 可能的情況 | 剪刀 | 石頭 | 布 |
| 剪刀 | True | False | False |
| 石頭 | False | True | False |
| 布 | False | False | True |

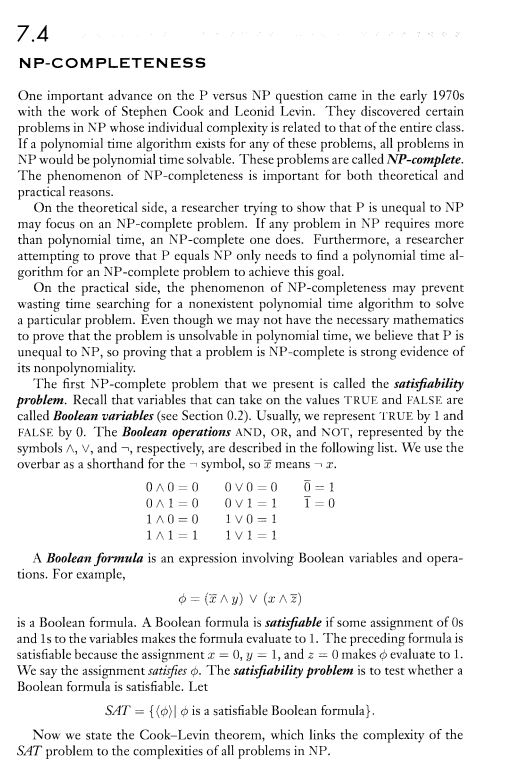
出剪刀就變成了 : 剪刀成立，石頭不成立，布不成立

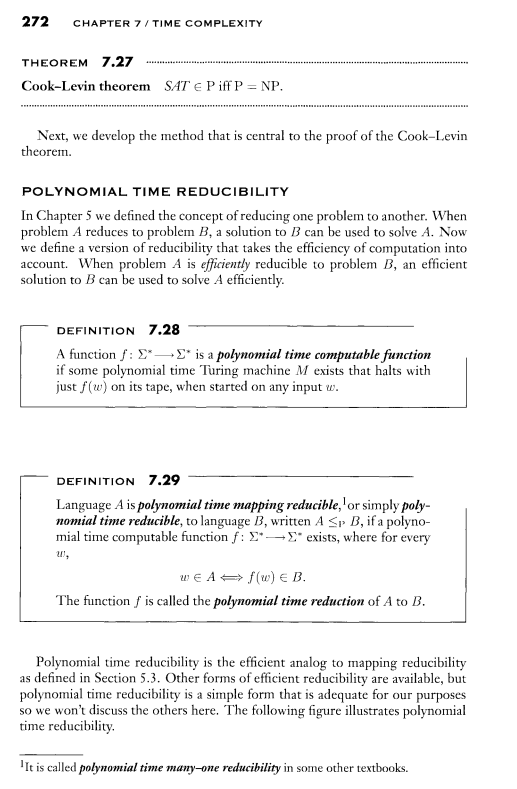
|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 剪刀  (P1) | 石頭  (P1) | 布  (P1) | 剪刀  (P2) | 石頭  (P2) | 布  (P2) | P1勝 | P2  勝 | 平手 |
| T | F | F | F | T | F | F | T | F |

那其中一個可能就可以表示成這樣，**玩家1出剪刀**& 玩家1沒出石頭 & 玩家1 沒出布 & 玩家 2 沒出剪刀 & **玩家 2 出石頭**& 玩家 2 沒出布 & **玩家 1沒贏** & 玩家 2贏 & 沒有平手 => 結果成立(滿足布林滿足性問題) [2]

**假設是NP-C=>如何證明?**

是第一個用Cook-Levin理論證明的NP完全問題，使用DTM將NP=P? 歸約成「一個布林方程式是否存在解」的問題。[3]如果想要知道如何證明是不太容易的，在某些計算理論的書籍當中有清楚的解釋，以下我就簡單貼個兩頁來自於Introduction to the Theory Of Computation, 2ed第七章[4]



****

**目前已有的解法=>透過什麼演算法?=>程式**

網路上目前有很多種SAT Solver 可以協助判斷一個問題有沒有滿足布林可滿足性問題，以下就附上一個例子: [5]

'Utility functions to humanize interaction with pycosat'

\_\_author\_\_ = 'Raymond Hettinger'

**import** **pycosat** *# https://pypi.python.org/pypi/pycosat*

**from** **itertools** **import** combinations

**from** **functools** **import** lru\_cache

**from** **sys** **import** intern

**def** make\_translate(cnf):

*"""Make translator from symbolic CNF to PycoSat's numbered clauses.*

*Return a literal to number dictionary and reverse lookup dict*

*>>> make\_translate([['~a', 'b', '~c'], ['a', '~c']])*

*({'a': 1, 'c': 3, 'b': 2, '~a': -1, '~b': -2, '~c': -3},*

*{1: 'a', 2: 'b', 3: 'c', -1: '~a', -3: '~c', -2: '~b'})*

*"""*

lit2num = {}

**for** clause **in** cnf:

**for** literal **in** clause:

**if** literal **not** **in** lit2num:

var = literal[1:] **if** literal[0] == '~' **else** literal

num = len(lit2num) // 2 + 1

lit2num[intern(var)] = num

lit2num[intern('~' + var)] = -num

num2var = {num:lit **for** lit, num **in** lit2num.items()}

**return** lit2num, num2var

**def** translate(cnf, uniquify=**False**):

'Translate a symbolic cnf to a numbered cnf and return a reverse mapping'

*# DIMACS CNF file format:*

*# http://people.sc.fsu.edu/~jburkardt/data/cnf/cnf.html*

**if** uniquify:

cnf = list(dict.fromkeys(cnf))

lit2num, num2var = make\_translate(cnf)

numbered\_cnf = [tuple([lit2num[lit] **for** lit **in** clause]) **for** clause **in** cnf]

**return** numbered\_cnf, num2var

**def** itersolve(symbolic\_cnf, include\_neg=**False**):

numbered\_cnf, num2var = translate(symbolic\_cnf)

**for** solution **in** pycosat.itersolve(numbered\_cnf):

**yield** [num2var[n] **for** n **in** solution **if** include\_neg **or** n > 0]

**def** solve\_all(symcnf, include\_neg=**False**):

**return** list(itersolve(symcnf, include\_neg))

**def** solve\_one(symcnf, include\_neg=**False**):

**return** next(itersolve(symcnf, include\_neg))

*############### Support for Building CNFs ##########################*

**@lru\_cache**(maxsize=**None**)

**def** neg(element) -> 'element':

'Negate a single element'

**return** intern(element[1:] **if** element.startswith('~') **else** '~' + element)

**def** from\_dnf(groups) -> 'cnf':

'Convert from or-of-ands to and-of-ors'

cnf = {frozenset()}

**for** group **in** groups:

nl = {frozenset([literal]) : neg(literal) **for** literal **in** group}

*# The "clause | literal" prevents dup lits: {x, x, y} -> {x, y}*

*# The nl check skips over identities: {x, ~x, y} -> True*

cnf = {clause | literal **for** literal **in** nl **for** clause **in** cnf

**if** nl[literal] **not** **in** clause}

*# The sc check removes clauses with superfluous terms:*

*# {{x}, {x, z}, {y, z}} -> {{x}, {y, z}}*

*# Should this be left until the end?*

sc = min(cnf, key=len) *# XXX not deterministic*

cnf -= {clause **for** clause **in** cnf **if** clause > sc}

**return** list(map(tuple, cnf))

**class** **Q**:

'Quantifier for the number of elements that are true'

**def** \_\_init\_\_(self, elements):

self.elements = tuple(elements)

**def** \_\_lt\_\_(self, n: int) -> 'cnf':

**return** list(combinations(map(neg, self.elements), n))

**def** \_\_le\_\_(self, n: int) -> 'cnf':

**return** self < n + 1

**def** \_\_gt\_\_(self, n: int) -> 'cnf':

**return** list(combinations(self.elements, len(self.elements)-n))

**def** \_\_ge\_\_(self, n: int) -> 'cnf':

**return** self > n - 1

**def** \_\_eq\_\_(self, n: int) -> 'cnf':

**return** (self <= n) + (self >= n)

**def** \_\_ne\_\_(self, n) -> 'cnf':

**raise** NotImplementedError

**def** \_\_repr\_\_(self) -> str:

**return** f'*{self.\_\_class\_\_.\_\_name\_\_}*(elements=*{self.elements!r}*)'

**def** all\_of(elements) -> 'cnf':

'Forces inclusion of matching rows on a truth table'

**return** Q(elements) == len(elements)

**def** some\_of(elements) -> 'cnf':

'At least one of the elements must be true'

**return** Q(elements) >= 1

**def** one\_of(elements) -> 'cnf':

'Exactly one of the elements is true'

**return** Q(elements) == 1

**def** basic\_fact(element) -> 'cnf':

'Assert that this one element always matches'

**return** Q([element]) == 1

**def** none\_of(elements) -> 'cnf':

'Forces exclusion of matching rows on a truth table'

**return** Q(elements) == 0

**參考資料:**

[1] <https://www.itread01.com/content/1548235989.html>

[2] <https://medium.com/leep3/%E5%BE%9E-sat-solver-%E7%8E%A9-leetcode-37809cb61df>

[3] <https://en.wikipedia.org/wiki/Cook%E2%80%93Levin_theorem>

[4]<http://fuuu.be/polytech/INFOF408/Introduction-To-The-Theory-Of-Computation-Michael-Sipser.pdf>

[5] <https://rhettinger.github.io/einstein.html#essential-utilities-for-humanization>