コンピュータサイエンス英語講義 I

XXXXXXXX: cocuh

2017 年 8 月 22 日

1　実装アルゴリズム

実装アルゴリズムについて説明する前に，私が数独ソルバーを実装する上で設定した要件について説明する．

実装は github 上に公開している．

`https://github.com/cocuh/sudoku-solver-2017`

1.1　要件

本ソルバーの要件を以下のように設定した．

- deterministic approach
  - すべての解が列挙可能 (enumerate all solution)
  - オプションで 1 つの充足解を見つけるモードの追加
  - 充足解が存在しない場合そのように出力する (prove unsatisfiablity)
- 任意次元に対応可能にする (available for sudoku whose size is 4x4, 9x9, 16x16 ...)
- 並列化可能なアルゴリズム (multi-processing)
- python で実装

すべての解の列挙可能な機能や充足解が存在しないことを示す機能のため，基本戦略として探索木をすべて探索する必要がある．この探索を如何に効率化させるかが重要となる．なるべく探索木の大きさの小さくなるように注意し，推論や変数選択を行うなどが必要となる．本アルゴリズムでは constraint propagation を用いた効率化を目指す．

また，近似アルゴリズム (approximation algorithm) ではこの要件を確率的にしかみたすことができないため用いることができず，決定論的アプローチを取る必要がある．

このため，本アルゴリズムでは constraint propagation base による，brute-force アルゴリズムを提案する．

次節で準備として数独についての定式化を行う．

1.2　数独の次数，notation

まず数独の次数を定義する．正方形の数独の大きさはは，4x4, 9x9, 16x16 など存在し，一辺は常に平方数となっている．このため，正の整数の d について $d^2 \times d^2$ のサイズの数独が存在すると解釈し，d を数独の次

数と呼ぶことにする．

次に，数独の一つのマスを cell と呼ぶことにする．次数 d の数独において，セルは d^4 個存在し，$C = \{c_{x,y}\}_{x,y\in 0,\cdots,d^2-1}$ と表記する．

数独のほぼすべての制約は cell の集合に対する all-different 制約であり，この all-different 制約を block と呼ぶことにする．たとえば，$y = 0$ の行についての block はセル集合 $\{c_{0,0}, c_{1,0}, c_{2,0}, \cdots, c_{d^2-1,0}\}$ に対する all-different 制約である．次数が 3 である場合，$\{c_{0,0}, c_{1,0}, c_{2,0}, c_{0,1}, c_{1,1}, c_{2,1}, c_{0,2}, c_{1,2}, c_{2,2}\}$ も block となるが，grid block と呼ぶことにする．$x = n$ の行の block は $b_{x=n}$，$y = n$ の行の block は $b_{y=n}$，grid block は $n, m \in 0, \cdots, d$ について $b_{c=(n,m)}$ と表記する．数独内でのすべての block を B とする．

以上のような定式化から，数独は cell と block の集合からなると考えられ，$S = (C, B)$ と表記する．

数独の目標は，すべての cell に 1 から d^2 の値を制約を満たすよう割り当てる (assign) ことである．

各 cell について，その cell が取りうる値の集合 possible values を考えることができる．

1.3 constraint propagation と探索木

ここでの探索木では，数独の状態をノードとし，ある cell について値を仮定 (assume) をすることで子ノードに移動すると考える．前提として，探索木のサイズはなるべく小さい方が探索時間が少ない．constraint propagation を行うことで，子ノードの数と深さを減らすことが期待される．

solve アルゴリズムは，再帰的アルゴリズムであり引数 S を取る．具体的には以下のような処理を行なっている．

1. 数独 S が与えられる
2. S について constraint propagation を行う
3. もし，constraint propagation 中に矛盾が発生したら空集合 {}(解なし) を返す
4. もし，すべての cell に値が割当済みならば $\{S\}$ を返す (充足解)
5. S から $c \in C$ を選択する．(変数選択)
6. すべての c の possible value v について以下を行う
7. results $\leftarrow$ {}
   (a) S をコピーする $S' = \mathrm{copy}(S)$
   (b) S' の c に v を代入する．(assume)
   (c) solve アルゴリズムを呼び出し結果を results に追加する
   results $\leftarrow$ results $\cup$ solve(S')
8. results を返す

1.4 二部グラフによる constraint propagation

all-different 制約は二部グラフによる解釈が可能であることが講義中で示された．違う形ではあるが，数独も二部グラフにより解釈ができる．このとき，二部グラフの片方の頂点集合は block の集合であり，もう片方は cell の集合である．

1.5 constraint propagation algorithm

本アルゴリズムでは，block を選択し，cell の possible values を更新するようなアルゴリズムとなっている．

block の選択ヒューリスティックとして，block に含まれる cell が前の propagation から何度更新されたかを利用する．[*1] block 内の cell の更新回数がおおければ多いほど，ほかの block との関係が強いと考えられ，効率的な更新が可能であると考えられるためだ．

cell ではなく block を選択している理由はメモリー容量と並列化した際の効率性である．cell の数は d^4，block の数は $3d^2$ である．上記のような選択ヒューリスティックを用いているため，HashMap などの辞書によるカウンターが必要である．このとき，辞書のレコードの個数は cell や block の数に依存する．このため，block を選択したほうがよいと考えた．

今回の実装では constraint propagation の並列化を行なっていないが，視野に入れていた．1 回の constraint propagation について，block に注目した場合は $O(d^2)$ の計算時間が掛かるが，cell に注目した場合は $O(1)$ である．このため，cell に注目した場合のほうが処理時間が短いことが期待され排他処理などのオーバーヘッドが相対的に大きくなり，並列化のメリットが失われるのではと考え cell ではなく block を選択した．

本アルゴリズムでの具体的な constraint propagation のアルゴリズムは以下のようになっている．

1. block b を選択する[*2]
2. block 内の cell で割当済みの値の集合を計算する $A = \{\text{getAssignedValue}(c) | c \in \text{getUnassignedCells}(b)\}$ [*3]
3. すべての値未割り当て cell c について，possible values を $\text{getPossibleValues}(c) \setminus A$ に更新する[*4]
4. すべての値未割り当て cell c について，possibles values の出現回数が 1 の値を数える．[*5]

$$P = \left\{ v | v \in [d^2], \exists!_{c \in \text{getUnassignedCells}(b)}\ v \in \text{getPossibleValues}(c) \right\}$$

5. block 内の値未割り当て cell c について，以下の操作を行う
   （a）もし c の possibe values と P に共通の値があるならば，c にその値を代入する．[*6]
   （b）もし c の possibe values が singleton であるならば，c にその値を代入する (assign)．[*7]
6. もし，前ループで cell の値割当が行われた場合，2 に進む．
7. 1 に進む

[*1] https://github.com/cocuh/sudoku-solver-2017/blob/0e1ebaf83f21f7424637e9655b313d16393a2f67/solve.py#L30-L37

[*2] https://github.com/cocuh/sudoku-solver-2017/blob/0e1ebaf83f21f7424637e9655b313d16393a2f67/solve.py#L264-L266

[*3] https://github.com/cocuh/sudoku-solver-2017/blob/0e1ebaf83f21f7424637e9655b313d16393a2f67/solve.py#L113-L117

[*4] https://github.com/cocuh/sudoku-solver-2017/blob/0e1ebaf83f21f7424637e9655b313d16393a2f67/solve.py#L131

[*5] https://github.com/cocuh/sudoku-solver-2017/blob/0e1ebaf83f21f7424637e9655b313d16393a2f67/solve.py#L136-L140

[*6] https://github.com/cocuh/sudoku-solver-2017/blob/0e1ebaf83f21f7424637e9655b313d16393a2f67/solve.py#L143-L151

[*7] https://github.com/cocuh/sudoku-solver-2017/blob/0e1ebaf83f21f7424637e9655b313d16393a2f67/solve.py#L152-L156

1.6 並列化に関して

本実装では，propagation においては並列化していないが，探索木を探索する際に並列に探索可能に実装している．1.3 で述べたように，ある cell c についてすべての possible values を試行する必要がある．このため，この possible values について並列に探索が可能である．

本実装では，並列化に際していくつかのヒューリスティックを活用しているが，詳しい説明は省略し紹介に留める．

- 変数選択ヒューリスティック (探索木での変数選択について)
- 並列化における worker に対する job 割当ヒューリスティック
- 並列化における job 生成ヒューリスティック

2 実験

16x16($d = 4$) の数独が課題であったため，インターネット上から問題[*8]をダウンロードし読み込めるように変換を行なった．

2.1 well-posed problem: 16x16

未並列において実行結果は以下のように成り，実行時間は 0.041645 秒であった．

```
$ python solve.py test_data/well-posed/16x16.csv
  ,  , 5,  | 7,  ,  ,  | 6,  ,  ,  |11,  ,  , 1
  ,  ,11, 3|14,  ,  , 4|16, 9,  ,  |  ,  , 8, 6
15,  ,  ,  |  ,  , 5, 1|10,  ,  ,  |  ,14,  ,
 9, 4,  , 2|  ,13,  , 6| 1,14,  ,15|  ,  ,12,
-------------------------------------------------
  ,  , 7,  | 4,  , 6, 9|  , 2,  ,  |  ,15,  ,
  ,  , 8,  |  ,  ,13, 2|  , 3,10,  |  ,  ,  ,
  ,  , 6,12|  ,  ,15, 7|  ,  ,  ,  | 1,  ,  ,
  ,  ,  ,  |  ,  ,  ,  |12,  ,  ,  |  , 6,  , 9
-------------------------------------------------
  ,  ,  ,  |  ,10,  ,  |  , 5,  ,  |  ,  ,  ,
  ,  , 9,  | 5,  ,  , 3|  ,  ,  ,  |16,  , 2,
  , 2,  ,15|  ,  , 9,  |11,  ,12, 4|  ,  , 5,14
11,  ,12,  |16,  ,  ,  |  , 8,  ,  | 3, 7,  , 4
-------------------------------------------------
10, 3,  , 6|  ,  ,  ,  | 2,  ,14,  | 7, 8, 4,
  ,14,  ,  | 2,  ,  ,  |  ,  ,  ,  |  ,10,  ,15
```

[*8] https://github.com/zonk1024/python-sudoku/blob/master/16x16.csv

```
  , 7,15,  |  , 8,  ,  |  ,  , 9,  | 5,  ,16,
  , 5,  ,  |  ,  ,11,  |  ,12,16, 8|  ,  ,  ,

SATISFIABLE: well-posed problem
#solutions = 1
spend time(sec) = 0.04164481163024902

solution 1/1
14,12, 5,10| 7, 9, 2,16| 6,13, 8, 3|11, 4,15, 1
 7, 1,11, 3|14,15,10, 4|16, 9, 2,12|13, 5, 8, 6
15, 6,13, 8| 3,12, 5, 1|10, 4,11, 7| 2,14, 9,16
 9, 4,16, 2|11,13, 8, 6| 1,14, 5,15|10, 3,12, 7
-----------------------------------------------
 5,11, 7,13| 4,16, 6, 9| 8, 2, 1,14|12,15, 3,10
16, 9, 8, 1|12,14,13, 2|15, 3,10, 6| 4,11, 7, 5
 4,10, 6,12| 8, 3,15, 7| 5,11,13, 9| 1,16,14, 2
 2,15, 3,14|10,11, 1, 5|12, 7, 4,16| 8, 6,13, 9
-----------------------------------------------
 6,16,14, 4|13,10,12,11| 3, 5, 7, 2|15, 9, 1, 8
 1, 8, 9, 7| 5, 6, 4, 3|14,10,15,13|16,12, 2,11
 3, 2,10,15| 1, 7, 9, 8|11,16,12, 4| 6,13, 5,14
11,13,12, 5|16, 2,14,15| 9, 8, 6, 1| 3, 7,10, 4
-----------------------------------------------
10, 3, 1, 6| 9, 5,16,13| 2,15,14,11| 7, 8, 4,12
 8,14, 4,16| 2, 1, 7,12|13, 6, 3, 5| 9,10,11,15
12, 7,15,11| 6, 8, 3,14| 4, 1, 9,10| 5, 2,16,13
13, 5, 2, 9|15, 4,11,10| 7,12,16, 8|14, 1, 6, 3

#solutions = 1
spend time(sec) = 0.04164481163024902
```

2.2 ill-posed problem: 16x16

問題中のいくつかの値を削除し ill-posed problem として，解を列挙可能か調べる．未並列において実行結果の一部[*9]は以下のように成り，実行時間は 59.7278 秒であった．

```
$ python solve.py test_data/ill-posed/16x16_2.csv
SATISFIABLE
```

[*9] 膨大であったため一部省略

```
#solutions = 1054
spend time(sec) = 59.7277717590332
```

並列化 (8cpu) した場合，実行結果の一部は以下のように成り，実行時間は 17.5305 秒であった．

```
$ python solve.py test_data/ill-posed/16x16_2.csv --parallel
SATISFIABLE
#solutions = 1054
spend time(sec) = 17.530484676361084
```

2.3 well-posed problem: 25x25

25x25 についても求解可能か試してみた．
未並列の場合は，実行時間は以下のようになり，実行時間は 228.2104 秒であった．

```
SATISFIABLE: well-posed problem
#solutions = 1
spend time(sec) = 228.21038031578064
```

並列化 (8cpu) した場合，実行結果は以下のように成り，実行時間は 72.0799 秒であった．

```
SATISFIABLE: well-posed problem
#solutions = 1
spend time(sec) = 72.07988691329956
```

3 さらなる速度向上のための指針

さらなる速度向上のために以下のような改善が考えられる．

- constraint propagation の集合演算は python の set を利用しているが，固定サイズの集合についての演算であるため bit array などを利用すればより高速に constraint propagation が可能であると思われる
- 並列化によるメリットがあまり享受できていない．ad-hoc な方法で worker に対し job を生成しているため，signal のようなものを利用し効率化できると推察される．
- python を利用しているため multi-processing などの並列化におけるオーバーヘッドが非常に大きい，rust などの言語を利用し更に Copy on Write を利用するとさらなる速度向上につながると思われる

4 結論

結論として，16x16 のみならず 25x25 の数独も一部の問題については実用的な時間で求解可能なソルバーができた．