名古屋大学情報学部
コンピュータ科学科
卒業論文

SMT ソルバーにおける *distinct* 制約の高速化とクイーングラフ彩色問題への応用

2020 年 2 月

101730135 小菅 脩司

概要

命題論理の充足可能性判定問題 (Boolean SATisfiability; SAT) は，与えられた命題論理式の充足可能性を判定する問題である．SAT は，人工知能および計算機工学における最も基本的な問題として，システム検証，プランニング，スケジューリングなど，さまざまな分野に応用されている．近年，SAT の解を求める SAT ソルバーの性能が大きく向上し，SAT を拡張・発展させた問題を中心に，SAT 技術が大きな広がりを見せている: MaxSAT，擬似ブール制約 (PB)，限量ブール式 (QBF)，モデル計数 (#SAT)，背景理論付き SAT.

なかでも，背景理論付き SAT (Satisfiability Modulo Theories; SMT) は，等号や算術，配列やリスト，ビットベクターなど様々な背景理論が扱えるように，SAT を拡張・発展させた技術である．SAT では，これらの背景理論を命題論理で記述する必要があるが，複雑な背景理論を表現することは困難な場合が多い．SMT は，背景理論をより表現能力の高い述語論理で記述できるため，問題を簡潔に記述することができる点が特長である．近年，SAT ソルバーを拡張した高速 SMT ソルバーが開発され，制約充足問題，プログラム検証などへの応用が活発に研究されている．

制約プログラミングの言語である制約充足問題は，与えられた制約を満たす解を探索する問題である．人工知能分野で生じる多くの組合せ問題は，制約充足問題として定式化できることが知られている．制約充足問題では，グローバル制約を用いて，複数の変数に対する複雑な制約を簡潔に表現できる点が特長の一つである．代表的なグローバル制約 $distinct(x_1, x_2, \ldots x_n)$ は，x_i が互いに異なる値をとることを表す．この *distinct* 制約は，記述性の向上を目的として SMT ソルバーにも取り入れられている．しかしながら，制約充足問題に対する SMT ソルバーの求解性能は，SAT 型制約ソルバーと比べて劣っているとの報告もあり，*distinct* 制約を含めその効率的な実装は重要な研究課題となっている．

本論文では，D. E. Knuth の教科書 The Art of Computer Programming でも取り上げられているクイーングラフ彩色問題を題材とし，その SMT

符号化と *distinct* 制約の高速化について述べる．クイーングラフ彩色問題の SMT 符号化として，クイーンの色を表す整数変数を用いた定式化 (色変数モデル)，クイーンの位置を表す整数変数を用いた定式化 (位置変数モデル)，0-1 変数を用いた定式化 (0-1 変数モデル)，チャネリング制約を用いて色変数モデルと 0-1 変数モデルをハイブリッドした定式化 (ハイブリッドモデル) を実装した．また，*distinct* 制約の高速化として，色変数モデルと位置変数モデルでは，SAT 型制約ソルバーで有効性が示されている鳩の巣原理等に基づくヒント制約を追加した．0-1 変数モデルとハイブリッドモデルでは，*distinct* 制約の PB 符号化 [大野,2019] を応用し，探索空間の枝刈りを実装した．

実装した SMT 符号化と *distinct* 制約の高速化手法の有効性を評価するために，クイーングラフ彩色問題 ($5 \leq N \leq 13$) を用いた実行実験を行なった．その結果，色変数モデル+ヒント制約と位置変数モデル+ヒント制約が，$N = 11$ まで解を求め，最も良い性能を示した．これにより，*distinct* 制約の高速化について，SAT でのヒント制約が SMT においても有効であることが確認できた．その一方で，チャネリング制約を用いたハイブリッドモデルは，SMT ソルバーの場合，求解性能が低下することが確認された．

目 次

図 目 次

表 目 次

コード目次

第1章　序論

命題論理の充足可能性判定問題 (Boolean SATisfiability; SAT) は，与えられた命題論理式の充足可能性を判定する問題である．SAT は，人工知能および計算機工学における最も基本的な問題として，システム検証，プランニング，スケジューリングなど，さまざまな分野に応用されている．近年，SAT の解を求める SAT ソルバーの性能が大きく向上し，SAT を拡張・発展させた問題を中心に，SAT 技術が大きな広がりを見せている: MaxSAT，擬似ブール制約 (PB)，限量ブール式 (QBF)，モデル計数 (#SAT)，背景理論付き SAT.

なかでも，背景理論付き SAT (Satisfiability Modulo Theories; SMT) は，等号や算術，配列やリスト，ビットベクターなど様々な背景理論が扱えるように，SAT を拡張・発展させた技術である．SAT では，これらの背景理論を命題論理で記述する必要があるが，複雑な背景理論を表現することは困難な場合が多い．SMT は，背景理論をより表現能力の高い述語論理で記述できるため，問題を簡潔に記述することができる点が特長である．近年，SAT ソルバーを拡張した高速 SMT ソルバーが開発され，制約充足問題，プログラム検証などへの応用が活発に研究されている．

制約プログラミングの言語である制約充足問題は，与えられた制約を満たす解を探索する問題である．人工知能分野で生じる多くの組合せ問題は，制約充足問題として定式化できることが知られている．制約充足問題では，グローバル制約を用いて，複数の変数に対する複雑な制約を簡潔に表現できる点が特長の一つである．代表的なグローバル制約 $distinct(x_1, x_2, \ldots x_n)$ は，x_i が互いに異なる値をとることを表す．この $distinct$ 制約は，記述性の向上を目的として SMT ソルバーにも取り入れられている．しかしながら，制約充足問題に対する SMT ソルバーの求解性能は，SAT 型制約ソルバーと比べて劣っているとの報告もあり，$distinct$ 制約を含めその効率的な実装は重要な研究課題となっている．

本論文では，D. E. Knuth の教科書 The Art of Computer Programming[8] でも取り上げられているクイーングラフ彩色問題を題材とし，その SMT

符号化と *distinct* 制約の高速化について述べる．クイーングラフ彩色問題の SMT 符号化として，クイーンの色を表す整数変数を用いた定式化 (色変数モデル)，クイーンの位置を表す整数変数を用いた定式化 (位置変数モデル)，0-1 変数を用いた定式化 (0-1 変数モデル)，チャネリング制約を用いて色変数モデルと 0-1 変数モデルをハイブリッドした定式化 (ハイブリッドモデル) を実装した．また，*distinct* 制約の高速化として，色変数モデルと位置変数モデルでは，

本論文の構成は以下の通りである．第 2 章では SMT について，第 3 章ではクイーングラフ彩色問題について SMT でのプログラム例を示しながらモデル化方法について説明を行う．第 4 章では用いた SMT ソルバにおける *distinct* 制約の高速化手法についてプログラムを示しながら説明し，第 5 章では作成したプログラムの比較実験とその考察について述べる．

第2章　背景理論付きSAT

2.1　SMTとSMTソルバー

背景理論付きSAT(Satisfiability Modulo Theories; SMT)は，命題論理よりも高い表現力を持つ論理体系で記述された背景理論をSAT技術で効果的に扱うことを目的とした技術である．SATが命題論理を扱うのに対して，SMTでは述語論理を扱う[7].

SMTソルバーは，プログラム検証, スケジューリング, プランニングなどの場面に使用される．

今回SMTソルバーを扱う際に使用する記述方法はsmt-lib2[5]である．SMT-LIBはSMTの研究開発の促進を目的とした国際的な取り組みであり，その目的としては，SMTシステムで使用される背景理論の標準的で厳密な記述の提供やSMTソルバーのための共通の入出力言語を開発し促進することなどが挙げられる．

SMTで扱われる背景理論には等号や算術，配列やリスト，ビットベクターなどが挙げられる．

既存のSMTソルバーとしては，OpenSMT[4], YICES[6], UCLID[2], CVC3[1], Z3[3]などが挙げられる．今回使用したソルバーは$z3$ソルバーである．Microsoft社によって開発されたもので，プログラム解析やテスト, 検証などを目的としている[10].

2.2　プログラム例

smt-lib2形式でのプログラムの例をグラフ彩色問題を用いて説明する．使用するSMTソルバーは$z3$ソルバー($ver.$4.8.9)とする．

グラフ彩色問題とは，辺で結ばれたノードが同じ色にならないように，各ノードを塗り分ける問題である．例として，図2.1のグラフを赤(1)，青(2)，緑(3)の3色で塗り分ける問題を考える．この問題を表すプログラムをコード2.1に示す.

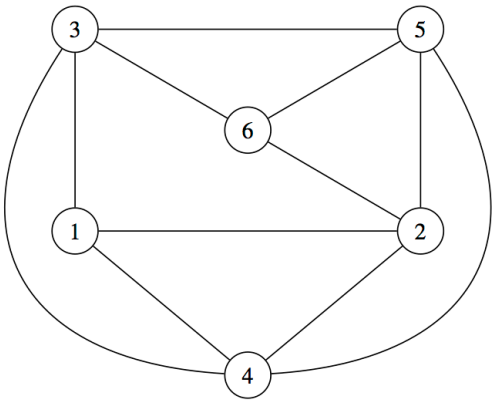


図 2.1: グラフ

2 から 7 行目は整数変数の宣言をしている．ここでは x_i の i がノード番号を表している．8 から 13 行目は x_i は 3 色に塗り分けられるため，1 以上 3 以下であるという制約を追加している．14 から 23 行目は辺で結ばれたノードの色が同じにならないことを等号を使って表している．24 行目はそのプログラムに解が存在するかどうかをチェックし出力することを表し，25 行目は求められた解を出力する．26 行目はプログラムの終わりを表す.

コード 2.1 の $z3$ ソルバー での実行例をコード 2.2 に示す．この出力から，ノード 2 と 3 は赤，ノード 1 と 5 は青，ノード 4 と 6 は緑に塗り分けられることがわかる.

2.3 *distinct* 制約

本研究で扱う *distinct* 制約とは，$(distinctx_1...x_n)$ で表され，その要素 x_i が互いに異なることを表す制約である．*distinct* 制約の実装方法として，今回使用した SMT ソルバーの $z3$ ソルバー では以下のように実装される..

$$\bigwedge_{1 \leq i < j \leq n} x_i \neq x_j$$

また，以下のように PB 符号化を用いて実装することができる.

$$distinct(x_1, x_2, \ldots, x_n)\ (x_i \in \{l, l+1, \ldots, u\}, n-1 \leq u-l)$$

```
; color.smt2
(declare-const x_1 Int)
(declare-const x_2 Int)
(declare-const x_3 Int)
(declare-const x_4 Int)
(declare-const x_5 Int)
(declare-const x_6 Int)
(assert (and (>= x_1 1) (<= x_1 3)))
(assert (and (>= x_2 1) (<= x_2 3)))
(assert (and (>= x_3 1) (<= x_3 3)))
(assert (and (>= x_4 1) (<= x_4 3)))
(assert (and (>= x_5 1) (<= x_5 3)))
(assert (and (>= x_6 1) (<= x_6 3)))
(assert (not (= x_1 x_2)))
(assert (not (= x_1 x_3)))
(assert (not (= x_1 x_4)))
(assert (not (= x_2 x_4)))
(assert (not (= x_2 x_5)))
(assert (not (= x_2 x_6)))
(assert (not (= x_3 x_5)))
(assert (not (= x_3 x_6)))
(assert (not (= x_4 x_5)))
(assert (not (= x_5 x_6)))
(check-sat)
(get-model)
(exit)
```

コード 2.1: グラフ彩色問題のプログラム (`color.smt2`)

は 0-1 変数 $p_{ij} \in 0, 1$ $(p_{ij} = 1 \Leftrightarrow x_i = j)$ を導入して,

$$\sum_{j=l}^{u} p_{ij} = 1 \ (i \in \{1, 2, \ldots, n\}) \quad (2.1)$$

$$\begin{cases} \sum_{i=1}^{n} p_{ij} = 1 \ (j \in \{l, l+1, \ldots, u\}) & (n-1 = u-l \text{ のとき}) \\ \sum_{i=1}^{n} p_{ij} \leq 1 \ (j \in \{l, l+1, \ldots, u\}) & (otherwise) \end{cases} \quad (2.2)$$

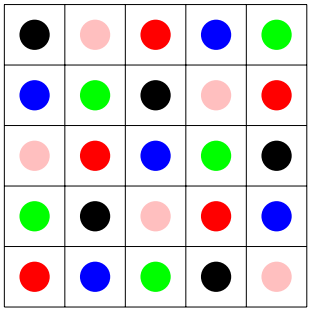
と表される.

```
sat
(model
  (define-fun x_2 () Int
    1)
  (define-fun x_4 () Int
    3)
  (define-fun x_1 () Int
    2)
  (define-fun x_3 () Int
    1)
  (define-fun x_5 () Int
    2)
  (define-fun x_6 () Int
    3)
)
(:added-eqs                       10
 :arith-eq-adapter                10
 :arith-bound-propagations-cheap  16
 :arith-bound-propagations-lp     6
 :arith-conflicts                 4
 :arith-diseq                     20
 :arith-lower                     38
 :arith-make-feasible             18
 :arith-max-columns               20
 :arith-max-rows                  10
 :arith-propagations              16
 :arith-upper                     16
 :binary-propagations             11
 :conflicts                       4
 :decisions                       11
 :final-checks                    1
 :max-memory                      3.06
 :memory                          2.55
 :mk-bool-var                     54
 :mk-clause                       36
 :num-allocs                      33298
 :num-checks                      1
 :propagations                    21
 :rlimit-count                    2188
 :time                            0.01
 :total-time                      0.02)
```

コード 2.2: `color.smt2`に対する $z3$ ソルバー の実行例

第3章　クイーングラフ彩色問題

クイーングラフ彩色問題は，N × N の大きさのチェス盤上に N 色のクイーンの駒を各色 N 個ずつ互いに取り合わないように配置する問題である．互いに取り合わないとは，各行，各列，各右上がり対角線，各右下がり対角線に配置されるクイーンの色が互いに異なるということである．



N=5 のとき

図 3 は N=5 の時のクイーングラフ彩色問題の解の一例を示している．

今回この問題の解を求めるにあたって使用したモデル化方法は色変数モデルと位置変数モデルである．

3.1　色変数モデル

色変数モデルは盤面に配置されるクイーンの駒の色を整数変数として解を求める制約モデルである．チェス盤上の行を i，列を j，クイーンの色を k とし，それらが取り得る値の集合を **N** とする．また，$i+j$ が取り得る値の集合を **U**，$i-j$ が取り得る値の集合を **D** とする．

$$\mathbf{N} = \{0, 1, ..., N-1\}$$
$$\mathbf{U} = \{0, 1, ..., 2N-2\}$$
$$\mathbf{D} = \{1-N, 2-N, ..., N-1\}$$

位置 (i, j) に配置されたクイーンの色を整数変数として

$$c_{ij} \in \mathbf{N}(i, j \in \mathbf{N}) \tag{3.1}$$

で表す.

各行について, 配置されるクイーンの色が互いに異なることから $distinct$ 制約を用いて

$$distinct\{c_{ij}|j \in \mathbf{N}\}(i \in \mathbf{N}) \tag{3.2}$$

という制約が得られる.

同様にして, 各列については

$$distinct\{c_{ij}|i \in \mathbf{N}\}(j \in \mathbf{N}) \tag{3.3}$$

という制約が得られる.

各右上がり対角線について $i + j$ が取り得る値の集合 $\mathbf{U}$ を用いて

$$distinct\{c_{ij}|i, j \in \mathbf{N}, i + j = u\}(u \in \mathbf{U}) \tag{3.4}$$

という制約が得られる.

各右下がり対角線について $i - j$ が取り得る値の集合 $\mathbf{D}$ を用いて

$$distinct\{c_{ij}|i, j \in \mathbf{N}, i - j = d\}(d \in \mathbf{D}) \tag{3.5}$$

という制約が得られる.

3.2 位置変数モデル

位置変数モデルは盤面上のある行に対して何列目にクイーンが配置されるのかを整数変数とした制約モデルである.

色 k のクイーンが行 i において配置される列を整数変数として,

$$y_{ik} \in \mathbf{N}(i, k \in \mathbf{N}) \tag{3.6}$$

で表す.

同一の行に同色のクイーンが配置されないことから

$$distinct\{y_{ik}|k \in \mathbf{N}\}(i \in \mathbf{N}) \tag{3.7}$$

という制約が得られる.

同様にして，同一の列に同色のクイーンが配置されないことから

$$distinct\{y_{ik}|i \in \mathbf{N}\}(k \in \mathbf{N}) \tag{3.8}$$

という制約が得られる.

また，各右上がり対角線については，行と列の和が 2 つ以上同じにならないことから,

$$distinct\{y_{ik} + i|i \in \mathbf{N}\}(k \in \mathbf{N}) \tag{3.9}$$

という制約が得られる.

同様に各右下がり対角線については，行と列の差が 2 つ以上同じにならないことから,

$$distinct\{y_{ik} - i|i \in \mathbf{N}\}(k \in \mathbf{N}) \tag{3.10}$$

という制約が得られる.

3.3 0-1 変数モデル

0-1 変数モデルは盤面に配置されるクイーンの駒の色を 0-1 変数として解を求める制約モデルである．つまり，3.1 に示した色変数モデルの整数変数を 0-1 変数で表したモデルである.

チェス盤上の行 i, 列 j に色 k のクイーンが配置されるかどうかを 0-1 変数として,

$$c_{ijk} \in \{0, 1\}(i, j, k \in \mathbf{N}) \tag{3.11}$$

で表す.

これは整数変数 3.1 を用いて

$$c_{ijk} = 1 \Leftrightarrow c_{ij} = k \tag{3.12}$$

のように表すことができる.

各行について，式 3.2 は式 2.1,2.2 を用いて以下のように表される.

$$\sum_{k=0}^{N-1} c_{ijk} = 1(i, j \in \mathbf{N}) \tag{3.13}$$

$$\sum_{j=0}^{N-1} c_{ijk} = 1(i, k \in \mathbf{N}) \tag{3.14}$$

という制約が得られる．

同様にして，各列については

$$\sum_{k=0}^{N-1} c_{ijk} = 1(i, j \in \mathbf{N}) \tag{3.15}$$
$$\sum_{i=0}^{N-1} c_{ijk} = 1(j, k \in \mathbf{N}) \tag{3.16}$$

という制約が得られる．

各右上がり対角線について $i + j$ が取り得る値の集合 $\mathbf{U}$ を用いて

$$\sum_{k=0}^{N-1} c_{ijk} = 1(i, j \in \mathbf{N}) \tag{3.17}$$
$$\sum_{i=0}^{u} c_{ijk} \leq 1(k \in \mathbf{N}, i + j = u, u \in \mathbf{U}) \tag{3.18}$$

という制約が得られる．$u = N - 1$ のとき，式 3.18 は以下のように変更される．

$$\sum_{i=0}^{u} c_{ijk} = 1(k \in \mathbf{N}, i + j = u, u \in \mathbf{U}) \tag{3.19}$$

各右下がり対角線について $i - j$ が取り得る値の集合 $\mathbf{D}$ を用いて

$$\sum_{k=0}^{N-1} c_{ijk} = 1(i, j \in \mathbf{N}) \tag{3.20}$$
$$\sum_{i=0}^{d} c_{ijk} \leq 1(k \in \mathbf{N}, i - j = d, d \in \mathbf{D}) \tag{3.21}$$

という制約が得られる．$d = N - 1$ のとき，式 3.21 は以下のように変更される．

$$\sum_{i=0}^{d} c_{ijk} = 1(k \in \mathbf{N}, i - j = d, d \in \mathbf{D}) \tag{3.22}$$

第4章　クイーングラフ彩色問題のSMT符号化と*distinct*制約の高速化

本章ではSMTソルバーにおける*distinct*制約の高速化のために用いた手法についてそれぞれ述べる．まず最初に，クイーングラフ彩色問題のSMT符号化について示し，次に高速化手法についてクイーングラフ彩色問題においてどのように表されるかを交えて説明を行う．

4.1　クイーングラフ彩色問題のSMT符号化

ここではN=5の場合のクイーングラフ彩色問題について使用したSMT符号化について説明する．本研究ではチェス盤上の一番上の行を0行目，一番左の列を0列目とし，クイーンの色は整数として0から数えるものとする．

4.1.1　色変数モデルのSMT符号化

まず，整数変数c_0_0は以下のように宣言される．c_i_jはi行j列目のクイーンの色を示している．

```
(declare-const c_0_0 Int)
```

また，c_0_0は0以上4以下であるので以下の制約を追加する．

```
(assert (and (>= c_0_0 0) (<= c_0_0 4)))
```

他の変数についても同様に宣言される．

次に，0行目に配置されるクイーンの色が互いに異なるという制約は以下のように宣言される．

```
(assert (distinct c_0_0 c_0_1 c_0_2 c_0_3 c_0_4)))
```

他の行や列，右上がり対角線や右下がり対角線についても同様に宣言される．

作成したN=5の場合の色変数モデルのコードをコード4.1に示す．3から52行目が各整数変数の宣言とそのドメインを指定しており，53から57行目が各行についての制約であり，58から62行目が各列について，63から69行目が各右上がり対角線について，70から76行目が角右下がり対角線についてである．また，77から81行目は対称性除去のために0行目の値を指定している．

4.1.2　位置変数モデルのSMT符号化

まず，整数変数y_0_0は以下のように宣言される．y_i_kはi行目のk色のクイーンが配置される場所を示している．

```
(declare-const y_0_0 Int)
```

また，y_0_0は0以上4以下であるので以下の制約を追加する．

```
(assert (and (>= y_0_0 0) (<= y_0_0 4)))
```

他の変数についても同様に宣言される．

次に，0行目に配置される5色のクイーンがそれぞれ異なる列に配置されるという制約は以下のように宣言される．

```
(assert (distinct y_0_0 y_0_1 y_0_2 y_0_3 y_0_4)))
```

他の行や列についても同様に宣言される．

右上がり対角線について，色0が配置される行と列の和が2つ以上同じにならないという制約は以下のように宣言される．

```
(assert (distinct (+ y_0_0 0) (+ y_1_0 1) (+ y_2_0 2) (+ y_3_0 3) (+ y_
```

他の色についても同様に宣言される．

右下がり対角線について，色0が配置される行と列の差が2つ以上同じにならないという制約は以下のように宣言される．

```
(assert (distinct (- y_0_0 0) (- y_1_0 1) (- y_2_0 2) (- y_3_0 3) (- y_
```

```
; QGCP model=0 n=5 opts={'-m': '0', '-x': ''}
; color-variable model (alldiff)
(declare-const c_0_0 Int)
(assert (and (>= c_0_0 0) (<= c_0_0 4)))
(declare-const c_0_1 Int)
(assert (and (>= c_0_1 0) (<= c_0_1 4)))
(declare-const c_0_2 Int)
(assert (and (>= c_0_2 0) (<= c_0_2 4)))
(declare-const c_0_3 Int)
(assert (and (>= c_0_3 0) (<= c_0_3 4)))
(declare-const c_0_4 Int)
(assert (and (>= c_0_4 0) (<= c_0_4 4)))
(declare-const c_1_0 Int)
(assert (and (>= c_1_0 0) (<= c_1_0 4)))
(declare-const c_1_1 Int)
(assert (and (>= c_1_1 0) (<= c_1_1 4)))
(declare-const c_1_2 Int)
(assert (and (>= c_1_2 0) (<= c_1_2 4)))
(declare-const c_1_3 Int)
(assert (and (>= c_1_3 0) (<= c_1_3 4)))
(declare-const c_1_4 Int)
(assert (and (>= c_1_4 0) (<= c_1_4 4)))
(declare-const c_2_0 Int)
(assert (and (>= c_2_0 0) (<= c_2_0 4)))
(declare-const c_2_1 Int)
(assert (and (>= c_2_1 0) (<= c_2_1 4)))
(declare-const c_2_2 Int)
(assert (and (>= c_2_2 0) (<= c_2_2 4)))
(declare-const c_2_3 Int)
(assert (and (>= c_2_3 0) (<= c_2_3 4)))
(declare-const c_2_4 Int)
(assert (and (>= c_2_4 0) (<= c_2_4 4)))
(declare-const c_3_0 Int)
(assert (and (>= c_3_0 0) (<= c_3_0 4)))
(declare-const c_3_1 Int)
(assert (and (>= c_3_1 0) (<= c_3_1 4)))
(declare-const c_3_2 Int)
(assert (and (>= c_3_2 0) (<= c_3_2 4)))
(declare-const c_3_3 Int)
(assert (and (>= c_3_3 0) (<= c_3_3 4)))
(declare-const c_3_4 Int)
(assert (and (>= c_3_4 0) (<= c_3_4 4)))
(declare-const c_4_0 Int)
(assert (and (>= c_4_0 0) (<= c_4_0 4)))
(declare-const c_4_1 Int)
(assert (and (>= c_4_1 0) (<= c_4_1 4)))
(declare-const c_4_2 Int)
(assert (and (>= c_4_2 0) (<= c_4_2 4)))
(declare-const c_4_3 Int)
(assert (and (>= c_4_3 0) (<= c_4_3 4)))
(declare-const c_4_4 Int)
(assert (and (>= c_4_4 0) (<= c_4_4 4)))
(assert (distinct c_0_0 c_0_1 c_0_2 c_0_3 c_0_4))
(assert (distinct c_1_0 c_1_1 c_1_2 c_1_3 c_1_4))
(assert (distinct c_2_0 c_2_1 c_2_2 c_2_3 c_2_4))
(assert (distinct c_3_0 c_3_1 c_3_2 c_3_3 c_3_4))
(assert (distinct c_4_0 c_4_1 c_4_2 c_4_3 c_4_4))
(assert (distinct c_0_0 c_1_0 c_2_0 c_3_0 c_4_0))
(assert (distinct c_0_1 c_1_1 c_2_1 c_3_1 c_4_1))
(assert (distinct c_0_2 c_1_2 c_2_2 c_3_2 c_4_2))
(assert (distinct c_0_3 c_1_3 c_2_3 c_3_3 c_4_3))
(assert (distinct c_0_4 c_1_4 c_2_4 c_3_4 c_4_4))
(assert (distinct c_0_1 c_1_0))
(assert (distinct c_0_2 c_1_1 c_2_0))
(assert (distinct c_0_3 c_1_2 c_2_1 c_3_0))
(assert (distinct c_0_4 c_1_3 c_2_2 c_3_1 c_4_0))
(assert (distinct c_1_4 c_2_3 c_3_2 c_4_1))
(assert (distinct c_2_4 c_3_3 c_4_2))
(assert (distinct c_3_4 c_4_3))
(assert (distinct c_0_3 c_1_4))
(assert (distinct c_0_2 c_1_3 c_2_4))
(assert (distinct c_0_1 c_1_2 c_2_3 c_3_4))
(assert (distinct c_0_0 c_1_1 c_2_2 c_3_3 c_4_4))
(assert (distinct c_1_0 c_2_1 c_3_2 c_4_3))
(assert (distinct c_2_0 c_3_1 c_4_2))
(assert (distinct c_3_0 c_4_1))
(assert (= c_0_0 0))
(assert (= c_0_1 1))
(assert (= c_0_2 2))
(assert (= c_0_3 3))
(assert (= c_0_4 4))
(check-sat)
(get-model)
(exit)
```

コード 4.1: N=5 の色変数モデルのクイーングラフ彩色問題

他の色についても同様に宣言される．

作成したN=5の場合の位置変数モデルのコードを4.2に示す．色変数モデルの時と同様にして，3から52行目が各整数変数の宣言とそのドメインを指定しており，53から72行目が各行，各列，各右上がり対角線，各右下がり対角線についてそれぞれ互いに異なるという制約を追加している．また，73から77行目は対称性除去のために0行目に配置されるクイーンの列を指定して宣言している．

4.1.3　0-1変数モデルのSMT符号化

高速化のための2つ目の手法は*distinct*制約をPB符号化して解くというものである．この手法では，求める解を整数変数としてではなく0-1変数として求める．

例としては，整数変数$x \in \{1,2,3\}$は0-1変数$x_1, x_2, x_3 \in 0,1$として，$x_i = 1$なら$x = i$というようにして解を求める．

N=5の場合について本研究で使用したSMT符号化を説明するまず，0-1変数c_0_0_0は以下のように宣言される．c_i_j_kはi行j列目がk色に塗られるかどうかを示している．

```
(declare-const c_0_0_0 Int)
```

また，c_0_0_0は0-1変数であるので以下の制約を追加する．

```
(assert (and (>= c_0_0_0 0) (<= c_0_0_0 1)))
```

他の変数についても同様に宣言される．盤面上の一つのマスに塗られる色は一色だけなので，0行0列目が一つの色で塗られる制約は以下のようになる．

```
(assert (= (+ c_0_0_0 c_0_0_1 c_0_0_2 c_0_0_3 c_0_0_4) 1))
```

他のマスについても同様に宣言される．この制約は式??に対応しており，重複を避けるためにまとめて宣言する．

次に，0行目に配置されるクイーンの色が互いに異なるという制約は以下のように宣言される．

```
(assert (= (+ c_0_0_0 c_0_1_0 c_0_2_0 c_0_3_0 c_0_4_0) 1))
(assert (= (+ c_0_0_1 c_0_1_1 c_0_2_1 c_0_3_1 c_0_4_1) 1))
```

```
; QGCP model=7 n=5 opts={'-m': '7', '-x': ''}
; position-variable model (alldiff)
(declare-const y_0_0 Int)
(assert (and (>= y_0_0 0) (<= y_0_0 4)))
(declare-const y_0_1 Int)
(assert (and (>= y_0_1 0) (<= y_0_1 4)))
(declare-const y_0_2 Int)
(assert (and (>= y_0_2 0) (<= y_0_2 4)))
(declare-const y_0_3 Int)
(assert (and (>= y_0_3 0) (<= y_0_3 4)))
(declare-const y_0_4 Int)
(assert (and (>= y_0_4 0) (<= y_0_4 4)))
(declare-const y_1_0 Int)
(assert (and (>= y_1_0 0) (<= y_1_0 4)))
(declare-const y_1_1 Int)
(assert (and (>= y_1_1 0) (<= y_1_1 4)))
(declare-const y_1_2 Int)
(assert (and (>= y_1_2 0) (<= y_1_2 4)))
(declare-const y_1_3 Int)
(assert (and (>= y_1_3 0) (<= y_1_3 4)))
(declare-const y_1_4 Int)
(assert (and (>= y_1_4 0) (<= y_1_4 4)))
(declare-const y_2_0 Int)
(assert (and (>= y_2_0 0) (<= y_2_0 4)))
(declare-const y_2_1 Int)
(assert (and (>= y_2_1 0) (<= y_2_1 4)))
(declare-const y_2_2 Int)
(assert (and (>= y_2_2 0) (<= y_2_2 4)))
(declare-const y_2_3 Int)
(assert (and (>= y_2_3 0) (<= y_2_3 4)))
(declare-const y_2_4 Int)
(assert (and (>= y_2_4 0) (<= y_2_4 4)))
(declare-const y_3_0 Int)
(assert (and (>= y_3_0 0) (<= y_3_0 4)))
(declare-const y_3_1 Int)
(assert (and (>= y_3_1 0) (<= y_3_1 4)))
(declare-const y_3_2 Int)
(assert (and (>= y_3_2 0) (<= y_3_2 4)))
(declare-const y_3_3 Int)
(assert (and (>= y_3_3 0) (<= y_3_3 4)))
(declare-const y_3_4 Int)
(assert (and (>= y_3_4 0) (<= y_3_4 4)))
(declare-const y_4_0 Int)
(assert (and (>= y_4_0 0) (<= y_4_0 4)))
(declare-const y_4_1 Int)
(assert (and (>= y_4_1 0) (<= y_4_1 4)))
(declare-const y_4_2 Int)
(assert (and (>= y_4_2 0) (<= y_4_2 4)))
(declare-const y_4_3 Int)
(assert (and (>= y_4_3 0) (<= y_4_3 4)))
(declare-const y_4_4 Int)
(assert (and (>= y_4_4 0) (<= y_4_4 4)))
(assert (distinct y_0_0 y_0_1 y_0_2 y_0_3 y_0_4))
(assert (distinct y_1_0 y_1_1 y_1_2 y_1_3 y_1_4))
(assert (distinct y_2_0 y_2_1 y_2_2 y_2_3 y_2_4))
(assert (distinct y_3_0 y_3_1 y_3_2 y_3_3 y_3_4))
(assert (distinct y_4_0 y_4_1 y_4_2 y_4_3 y_4_4))
(assert (distinct y_0_0 y_1_0 y_2_0 y_3_0 y_4_0))
(assert (distinct y_0_1 y_1_1 y_2_1 y_3_1 y_4_1))
(assert (distinct y_0_2 y_1_2 y_2_2 y_3_2 y_4_2))
(assert (distinct y_0_3 y_1_3 y_2_3 y_3_3 y_4_3))
(assert (distinct y_0_4 y_1_4 y_2_4 y_3_4 y_4_4))
(assert (distinct (+ y_0_0 0) (+ y_1_0 1) (+ y_2_0 2) (+ y_3_0 3) (+ y_4_0 4)))
(assert (distinct (+ y_0_1 0) (+ y_1_1 1) (+ y_2_1 2) (+ y_3_1 3) (+ y_4_1 4)))
(assert (distinct (+ y_0_2 0) (+ y_1_2 1) (+ y_2_2 2) (+ y_3_2 3) (+ y_4_2 4)))
(assert (distinct (+ y_0_3 0) (+ y_1_3 1) (+ y_2_3 2) (+ y_3_3 3) (+ y_4_3 4)))
(assert (distinct (+ y_0_4 0) (+ y_1_4 1) (+ y_2_4 2) (+ y_3_4 3) (+ y_4_4 4)))
(assert (distinct (- y_0_0 0) (- y_1_0 1) (- y_2_0 2) (- y_3_0 3) (- y_4_0 4)))
(assert (distinct (- y_0_1 0) (- y_1_1 1) (- y_2_1 2) (- y_3_1 3) (- y_4_1 4)))
(assert (distinct (- y_0_2 0) (- y_1_2 1) (- y_2_2 2) (- y_3_2 3) (- y_4_2 4)))
(assert (distinct (- y_0_3 0) (- y_1_3 1) (- y_2_3 2) (- y_3_3 3) (- y_4_3 4)))
(assert (distinct (- y_0_4 0) (- y_1_4 1) (- y_2_4 2) (- y_3_4 3) (- y_4_4 4)))
(assert (= y_0_0 0))
(assert (= y_0_1 1))
(assert (= y_0_2 2))
(assert (= y_0_3 3))
(assert (= y_0_4 4))
(check-sat)
(get-model)
(exit)
```

コード 4.2: N=5 の位置変数モデルのクイーングラフ彩色問題

```
(assert (= (+ c_0_0_2 c_0_1_2 c_0_2_2 c_0_3_2 c_0_4_2) 1))
(assert (= (+ c_0_0_3 c_0_1_3 c_0_2_3 c_0_3_3 c_0_4_3) 1))
(assert (= (+ c_0_0_4 c_0_1_4 c_0_2_4 c_0_3_4 c_0_4_4) 1))
```

他の行や列についても同様に宣言される．

右上がり対角線について，c_0_3，c_1_2，c_2_1，c_3_0が互いに異なるという制約は以下のように宣言される．

```
(assert (<= (+ c_0_3_0 c_1_2_0 c_2_1_0 c_3_0_0) 1))
(assert (<= (+ c_0_3_1 c_1_2_1 c_2_1_1 c_3_0_1) 1))
(assert (<= (+ c_0_3_2 c_1_2_2 c_2_1_2 c_3_0_2) 1))
(assert (<= (+ c_0_3_3 c_1_2_3 c_2_1_3 c_3_0_3) 1))
(assert (<= (+ c_0_3_4 c_1_2_4 c_2_1_4 c_3_0_4) 1))
```

他の右上がり対角線についても同様に宣言されるが，$i+j=N-1$の場合は上記の<=は=に変更して以下のように宣言される．

```
(assert (= (+ c_0_4_0 c_1_3_0 c_2_2_0 c_3_1_0 c_4_0_0) 1))
(assert (= (+ c_0_4_1 c_1_3_1 c_2_2_1 c_3_1_1 c_4_0_1) 1))
(assert (= (+ c_0_4_2 c_1_3_2 c_2_2_2 c_3_1_2 c_4_0_2) 1))
(assert (= (+ c_0_4_3 c_1_3_3 c_2_2_3 c_3_1_3 c_4_0_3) 1))
(assert (= (+ c_0_4_4 c_1_3_4 c_2_2_4 c_3_1_4 c_4_0_4) 1))
```

右下がり対角線については$i-j=N-1$の場合に<=を=に変更し同様にして宣言される．

作成したN=5の場合の色変数モデルのコードをコード4.3に示す．3から277行目が各整数変数の宣言とそのドメインと色の制約を追加しており，278から425行目が各行，各列，各右上がり対角線，各右下がり対角線についてそれぞれ互いに異なるという制約を追加している．また，426から430行目は対称性除去のために0行目の値を指定している．

```
1  ; QGCP model=18 n=5 opts={'-m': '18', '-x': ''}
2  ; color-variable model (pb)
3  (declare-const c_0_0_0 Int)
4  (assert (and (>= c_0_0_0 0) (<= c_0_0_0 1)))
5  (declare-const c_0_0_1 Int)
6  (assert (and (>= c_0_0_1 0) (<= c_0_0_1 1)))
7  (declare-const c_0_0_2 Int)
8  (assert (and (>= c_0_0_2 0) (<= c_0_0_2 1)))
9  (declare-const c_0_0_3 Int)
10 (assert (and (>= c_0_0_3 0) (<= c_0_0_3 1)))
11 (declare-const c_0_0_4 Int)
12 (assert (and (>= c_0_0_4 0) (<= c_0_0_4 1)))
13 (declare-const c_0_1_0 Int)
```

```
(assert (and (>= c_0_1_0 0) (<= c_0_1_0 1)))
(declare-const c_0_1_1 Int)
(assert (and (>= c_0_1_1 0) (<= c_0_1_1 1)))
(declare-const c_0_1_2 Int)
(assert (and (>= c_0_1_2 0) (<= c_0_1_2 1)))
(declare-const c_0_1_3 Int)
(assert (and (>= c_0_1_3 0) (<= c_0_1_3 1)))
(declare-const c_0_1_4 Int)
(assert (and (>= c_0_1_4 0) (<= c_0_1_4 1)))
(declare-const c_0_2_0 Int)
(assert (and (>= c_0_2_0 0) (<= c_0_2_0 1)))
(declare-const c_0_2_1 Int)
(assert (and (>= c_0_2_1 0) (<= c_0_2_1 1)))
(declare-const c_0_2_2 Int)
(assert (and (>= c_0_2_2 0) (<= c_0_2_2 1)))
(declare-const c_0_2_3 Int)
(assert (and (>= c_0_2_3 0) (<= c_0_2_3 1)))
(declare-const c_0_2_4 Int)
(assert (and (>= c_0_2_4 0) (<= c_0_2_4 1)))
(declare-const c_0_3_0 Int)
(assert (and (>= c_0_3_0 0) (<= c_0_3_0 1)))
(declare-const c_0_3_1 Int)
(assert (and (>= c_0_3_1 0) (<= c_0_3_1 1)))
(declare-const c_0_3_2 Int)
(assert (and (>= c_0_3_2 0) (<= c_0_3_2 1)))
(declare-const c_0_3_3 Int)
(assert (and (>= c_0_3_3 0) (<= c_0_3_3 1)))
(declare-const c_0_3_4 Int)
(assert (and (>= c_0_3_4 0) (<= c_0_3_4 1)))
(declare-const c_0_4_0 Int)
(assert (and (>= c_0_4_0 0) (<= c_0_4_0 1)))
(declare-const c_0_4_1 Int)
(assert (and (>= c_0_4_1 0) (<= c_0_4_1 1)))
(declare-const c_0_4_2 Int)
(assert (and (>= c_0_4_2 0) (<= c_0_4_2 1)))
(declare-const c_0_4_3 Int)
(assert (and (>= c_0_4_3 0) (<= c_0_4_3 1)))
(declare-const c_0_4_4 Int)
(assert (and (>= c_0_4_4 0) (<= c_0_4_4 1)))
```

⋮

```
(assert (= (+ c_0_0_0 c_0_0_1 c_0_0_2 c_0_0_3 c_0_0_4) 1))
(assert (= (+ c_0_1_0 c_0_1_1 c_0_1_2 c_0_1_3 c_0_1_4) 1))
(assert (= (+ c_0_2_0 c_0_2_1 c_0_2_2 c_0_2_3 c_0_2_4) 1))
(assert (= (+ c_0_3_0 c_0_3_1 c_0_3_2 c_0_3_3 c_0_3_4) 1))
(assert (= (+ c_0_4_0 c_0_4_1 c_0_4_2 c_0_4_3 c_0_4_4) 1))
```

⋮

```
; alldiff_pb c_0_0 c_0_1 c_0_2 c_0_3 c_0_4
(assert (= (+ c_0_0_0 c_0_1_0 c_0_2_0 c_0_3_0 c_0_4_0) 1))
(assert (= (+ c_0_0_1 c_0_1_1 c_0_2_1 c_0_3_1 c_0_4_1) 1))
(assert (= (+ c_0_0_2 c_0_1_2 c_0_2_2 c_0_3_2 c_0_4_2) 1))
(assert (= (+ c_0_0_3 c_0_1_3 c_0_2_3 c_0_3_3 c_0_4_3) 1))
(assert (= (+ c_0_0_4 c_0_1_4 c_0_2_4 c_0_3_4 c_0_4_4) 1))
```

⋮

```
(assert (= c_0_0_0 1))
(assert (= c_0_1_1 1))
```

```
(assert (= c_0_2_2 1))
(assert (= c_0_3_3 1))
(assert (= c_0_4_4 1))
(check-sat)
(get-model)
(exit)
```

コード 4.3: N=5 の 0-1 変数のみを用いた色変数モデルのクイーングラフ彩色問題

4.2　*distinct* 制約の高速化

高速化のための手法として 3 つの手法を用いた．

4.2.1　鳩の巣原理を用いたヒント制約

1 つ目の手法は鳩の巣原理を用いたヒント制約の追加である．このヒント制約は参考文献 [9] から引用したものであり，追加した制約は以下の通りである．

$distinct(x_1...x_n)$ について，$x_i \in \{l, l+1, ..., u\}$ であるとき

$$\bigvee_{i=1}^{n} x_i \geq l + n - 1 \tag{4.1}$$

$$\bigvee_{i=1}^{n} \neg(x_i \geq u - n + 1) \tag{4.2}$$

である．

(4.2) は，全ての x_i が $l + n - 2$ 以下になることを禁止しており，(4.2) は，全ての x_i が $u - n + 1$ 以上になることを禁止している．

この制約について SMT 符号化を行うと．例として，N=5 での時に `(distinct c_0_3 c_1_2 c_2_1 c_3_0)` は以下のように宣言される．

```
(assert (distinct c_0_3 c_1_2 c_2_1 c_3_0))
(assert (or (>= c_0_3 3) (>= c_1_2 3) (>= c_2_1 3) (>= c_3_0 3)))
(assert (or (<= c_0_3 1) (<= c_1_2 1) (<= c_2_1 1) (<= c_3_0 1)))
```

この制約を追加する利点としては，上記の例においては，c_0_3,c_1_2,c_2_1 がそれぞれ 0,1,2 の値を取るとしたときに，c_3_0 の取り得る範囲が 3 以上 4 以下に制限される．このように，制限を加わえることができるので，探索範囲が狭まり求解速度を上げることができる．

4.2.2 at-least-one 制約を用いたヒント制約

2つ目の手法は，*distinct* 制約の要素のドメインのサイズと要素数が等しい場合，つまり $distinct(x_1...x_n)$ について，$x_i \in \{l, l+1, ..., u\}$ かつ $u - l = n - 1$ であるときに以下の制約を追加するものである．

$$\bigvee_{i=1}^{n} x_i = a \ (a \in \{l, l+1, ..., u\}) \tag{4.3}$$

この制約はドメインサイズと要素数が等しい場合には，値 a に対してその値をとる変数 x_i が存在するというヒントを追加している．

この制約について SMT 符号化を行うと．例として，N=5 の時に `(distinct c_0_0 c_0_1 c_0_2 c_0_3 c_0_4)` は以下のように宣言される．

```
(assert (distinct c_0_0 c_0_1 c_0_2 c_0_3 c_0_4))
(assert (or (= c_0_0 0) (= c_0_1 0) (= c_0_2 0) (= c_0_3 0) (= c_0_4 0)))
(assert (or (= c_0_0 1) (= c_0_1 1) (= c_0_2 1) (= c_0_3 1) (= c_0_4 1)))
(assert (or (= c_0_0 2) (= c_0_1 2) (= c_0_2 2) (= c_0_3 2) (= c_0_4 2)))
(assert (or (= c_0_0 3) (= c_0_1 3) (= c_0_2 3) (= c_0_3 3) (= c_0_4 3)))
(assert (or (= c_0_0 4) (= c_0_1 4) (= c_0_2 4) (= c_0_3 4) (= c_0_4 4)))
```

この制約を追加する利点としては，上記の例においては，探索を進めていくにつれて c_0_0,c_0_1,c_0_2,c_0_3,c_0_4 が 0 にならないと分かった時点でその解が間違っているとわかるので枝刈りを行うことができ，求解速度を上げることができる．

4.2.3 *distinct* 制約の PB 符号化を用いた改良

3つ目の手法は *distinct* 制約の PB 符号化に改良を加えるものである．これは参考文献 [9] から引用したものであり，2つの改良手法を用いた．

改良手法 1

1つ目の改良手法は *distinct* 制約の PB 符号化を以下のように表すものである．

$distinct(x_1...x_n)$について，$n < d$の時，各値jごとのx_{ij}の和を表す変数y_jを導入し，以下のように表す．

$$x_{i1} + ... + x_{id} = 1 \ (i = 1...n) \tag{4.4}$$
$$x_{1j} + ... + x_{nj} \leq 1 \ (j = 1...d) \tag{4.5}$$
$$y_j = x_{1j} + ... + x_{nj} \ (j = 1...d) \tag{4.6}$$
$$y_1 + ... + y_d = n \tag{4.7}$$

(4.6)(4.7)についてSMT符号化を行うと，例として，N=5での時に(distinct c_0_1 c_1_0)は以下のように宣言される．

```
(declare-const _T1_0 Int)
(assert (and (>= _T1_0 0) (<= _T1_0 1)))
(declare-const _T1_1 Int)
(assert (and (>= _T1_1 0) (<= _T1_1 1)))
(declare-const _T1_2 Int)
(assert (and (>= _T1_2 0) (<= _T1_2 1)))
(declare-const _T1_3 Int)
(assert (and (>= _T1_3 0) (<= _T1_3 1)))
(declare-const _T1_4 Int)
(assert (and (>= _T1_4 0) (<= _T1_4 1)))
(assert (= _T1_0 (+ c_0_1_0 c_1_0_0)))
(assert (= _T1_1 (+ c_0_1_1 c_1_0_1)))
(assert (= _T1_2 (+ c_0_1_2 c_1_0_2)))
(assert (= _T1_3 (+ c_0_1_3 c_1_0_3)))
(assert (= _T1_4 (+ c_0_1_4 c_1_0_4)))
(assert (= (+ _T1_0 _T1_1 _T1_2 _T1_3 _T1_4) 2))
```

この制約を追加する利点としては，上記の例においては，探索を進めていくにつれてc_0_1,c_1_0が1,2,3,4にならないと分かった時点では，4.4と4.5の制約には違反しないが，`_T1_0 + _T1_1 + _T1_2 + _T1_3 + _T1_4 = 2`が成り立たないことがわかるため枝刈りを行うことができ，求解速度を上げることができる．

改良手法2

1つ目の改良手法は*distinct*制約のPB符号化を以下のように表すものである．2つ目の改良手法は*distinct*制約を以下のように表すものである．

$distinct(x_1...x_n)$ について，$n < d$ の時，各値 j ごとに新たな変数 $x_{(n+1)j}$ を導入し，以下のように表す.

$$x_{i1} + ... + x_{id} = 1 \ (i = 1...n) \tag{4.8}$$
$$x_{(n+1)1} + ... + x_{(n+1)d} = d - n \tag{4.9}$$
$$x_{1j} + ... + x_{(n+1)j} = 1 \ (j = 1...d) \tag{4.10}$$
$$\tag{4.11}$$

(4.9)(4.10) について SMT 符号化を行うと，例として，N=5 での時に (distinct c_0_1 c_1_0) は以下のように宣言される.

```
(declare-const _T1_0 Int)
(assert (and (>= _T1_0 0) (<= _T1_0 1)))
(declare-const _T1_1 Int)
(assert (and (>= _T1_1 0) (<= _T1_1 1)))
(declare-const _T1_2 Int)
(assert (and (>= _T1_2 0) (<= _T1_2 1)))
(declare-const _T1_3 Int)
(assert (and (>= _T1_3 0) (<= _T1_3 1)))
(declare-const _T1_4 Int)
(assert (and (>= _T1_4 0) (<= _T1_4 1)))
(assert (= (+ _T1_0 _T1_1 _T1_2 _T1_3 _T1_4) 3))
(assert (= (+ c_0_1_0 c_1_0_0 _T1_0) 1))
(assert (= (+ c_0_1_1 c_1_0_1 _T1_1) 1))
(assert (= (+ c_0_1_2 c_1_0_2 _T1_2) 1))
(assert (= (+ c_0_1_3 c_1_0_3 _T1_3) 1))
(assert (= (+ c_0_1_4 c_1_0_4 _T1_4) 1))
```

この制約を追加する利点としては，上記の例においては，探索を進めていくにつれて c_0_1,c_1_0 が 1,2,3,4 にならないと分かった時点では，**??**の制約には違反しないが，_T1_0 + _T1_1 + _T1_2 + _T1_3 + _T1_4 = 3 が成り立たないことがわかるため枝刈りを行うことができ，求解速度を上げることができる.

4.3　チャネリング制約を用いたクイーングラフ彩色問題のSMT符号化

4.2.3に示した高速化手法は3.3の0-1変数モデルに対してのみ用いることができる．整数変数を用いている色変数モデルと位置変数モデルに対してもこの高速化手法を実装するためにチャネリング制約を用いた．

この手法では求める解は整数変数として求めるが，*distinct*制約を解く際にチャネリング制約を追加し求める整数変数を0-1変数に変換してからPB符号化で解く．

N=5の時，c_0_0について追加したチャネリング制約は以下のものである．

```
(assert (=> (= c_0_0_0 1) (= c_0_0 0)))
(assert (=> (= c_0_0 0) (= c_0_0_0 1)))
(assert (=> (= c_0_0_1 1) (= c_0_0 1)))
(assert (=> (= c_0_0 1) (= c_0_0_1 1)))
(assert (=> (= c_0_0_2 1) (= c_0_0 2)))
(assert (=> (= c_0_0 2) (= c_0_0_2 1)))
(assert (=> (= c_0_0_3 1) (= c_0_0 3)))
(assert (=> (= c_0_0 3) (= c_0_0_3 1)))
(assert (=> (= c_0_0_4 1) (= c_0_0 4)))
(assert (=> (= c_0_0 4) (= c_0_0_4 1)))
```

これは

$$c_{ijk} = 1 \Leftrightarrow c_{ij} = k \tag{4.12}$$

を表す．他の変数に対しても同様に宣言される．また，*distinct*制約を扱う際には`c_i_j_k`を用いて制約が生成される．

第5章　実行実験

導入した高速化手法が有効であるかを評価するために，先に示したクイーングラフ彩色問題を用いて実験をし，その結果について比較評価を行う．

実験は，SMT ソルバーの z3(ver. 4.8.9) を用いてクイーングラフ彩色問題の大きさ N=5〜13 について，1 問あたりの制限時間を 2 時間として実験し，各実行 CPU 時間を計測した．

実験環境は，Mac mini，3.2GHz，64GB メモリである．

実行実験の結果をまとめたものを表 5.1 と 5.2 に表す．

表 5.1 は??のヒント制約を加える手法と??の PB 符号化を行う手法の実験結果である．

上から 1 行目が問題の大きさ N と，その問題に解があるかを示している．(sat が解が存在していることを表し，unsat が解が存在しないことを表している)

左から 1 列目が用いたモデル化方法と高速化手法を示している．

COL と POS はそれぞれ色変数モデルと位置変数モデルを用いることを表している．

H1 と H2 と H3 はそれぞれ *distinct* 制約に対するヒント制約 1 とヒント制約 2 とヒント制約 1 と 2 を合わせて使うことを表している．

PB1 と PB2 と PB3 はそれぞれ *distinct* 制約の PB 符号化と 4.2.3 に示した PB 符号化の改良手法 1 と改良手法 2 を用いることを表している．

表中の赤字は同じ大きさの問題の中で最も速く解けたことを示している．(N=5,6 については差が無かったため省略している)

実験結果より，単に *distinct* 制約のみを用いたものは SMT では色変数モデルよりも位置変数モデルの方が性能が良いということがわかる．また，N=11 まで解けていることから加えた 4.2.1,4.2.2 のヒント制約は SMT においても有効性が確認できた．

distinct 制約を PB 符号化して解く手法は,4.2.3 は有効性を確認できたが，4.2.3 は有効性は確認できなかった．また，実行した際の出力からこ

表 5.1: 実験結果 1

	N=5 SAT	N=6 UNSAT	N=7 SAT	N=8 UNSAT	N=9 UNSAT	N=10 UNSAT	N=11 SAT	N=12 SAT	N=13 SAT
COL	0.01	0.09	1.09	91.88	TO	TO	TO	TO	TO
COL+H1	0.01	0.02	0.11	7.41	4044.89	TO	TO	TO	TO
COL+H2	0.01	0.03	0.08	0.35	30.96	123.69	TO	TO	TO
COL+H3	0.02	0.03	0.06	0.23	32.54	138.47	2338.89	TO	TO
COL+PB1	0.17	0.51	2.41	6.88	401.54	3413.20	TO	TO	TO
COL+PB2	0.21	0.72	3.49	9.26	530.82	6736.73	TO	TO	TO
COL+PB3	0.47	1.39	193.88	TO	TO	TO	TO	TO	TO
POS	0.01	0.02	0.05	1.00	58.37	1484.24	TO	TO	TO
POS+H1	0.01	0.02	0.06	0.96	62.46	2093.45	TO	TO	TO
POS+H2	0.01	0.02	0.06	0.28	46.77	167.53	4677.32	TO	TO
POS+H3	0.01	0.02	0.05	0.43	47.38	231.74	1257.02	TO	TO

の方法では SMT ソルバーは SAT ソルバーのみを使用していることがわかった．このことからこの手法は背景理論ソルバーを扱えるという利点を活かせていないと言える．また，符号化方法 1 が PB 符号化した中で一番速く解けていることから，SMT ソルバーにおいて PB 符号化手法ではヒント制約を追加しないほうが良いと言える．これは，今回扱った符号化方法ではヒント制約追加による性能向上よりも制約数増加による性能の低下の方が影響が大きいと考えられる．

表 5.2 はチャネリング制約を用いた PB 符号化の実験結果である．

C はチャネリング制約を用いることを表す．

表中の青文字については間違った解を導き出したことを表している．

表 5.2: 実験結果 2

	N=5 SAT	N=6 UNSAT	N=7 SAT	N=8 UNSAT	N=9 UNSAT	N=10 UNSAT	N=11 SAT	N=12 SAT	N=13 SAT
COL+PB1+C	0.03	0.04	0.10	2.68	141.00	2796.16	3967.33	TO	TO
COL+PB2+C	0.14	905.22	TO	TO	TO	TO	TO	TO	TO
COL+PB3+C	1.03	1481.66	TO	TO	TO	TO	TO	TO	TO
POS+PB1+C	0.05	0.08	0.16	0.87	45.18	3770.98	TO	TO	TO
POS+PB2+C	0.06	0.10	0.18	1.15	58.12	3155.96	TO	TO	TO
POS+PB3+C	0.06	0.10	0.16	1.49	56.63	3254.16	TO	TO	TO

4.3 に示したチャネリング制約を用いた手法は，COL+PB2+C, COL+PB3+C では単に *distinct* 制約を用いた場合よりも性能が低下しており，COL+PB1+C では，N=11 において間違った解を出力している．このことから z3 ソルバーにおいてチャネリング制約を用いることは不適であると言える．

第6章　結論

本論文では，SMT の *distinct* 制約に高速化手法を用い，その評価を行うためにクイーングラフ彩色問題の解を求めた．実装した高速化手法は 3 つあり，鳩の巣原理を用いたヒント制約の追加と at-most-one 制約を用いたヒント制約の追加と *distinct* 制約の PB 符号化の改良手法であるこれらの手法は探索空間の枝刈りを行うことで求解速度を向上させる．また，チャネリング制約を用いることで色変数モデルと位置変数モデルにおいても *distinct* 制約の PB 符号化を用いることができるハイブリッドモデルを実装した．実行実験では，クイーングラフ彩色問題の N=11 をヒント制約を追加する手法を用いたものが解くことができ，単に *distinct* 制約を用いたものよりも高速に解くことに成功した．これにより，SAT ソルバーにおいて有効な手法が SMT ソルバーにおいても有効であることを確認した．また，z3 ソルバーにおいて，チャネリング制約を用いると性能低下の原因になりうることを発見した．今後の課題は，新たな *distinct* 制約の符号化の提案とチャネリング制約を用いた際の性能低下の原因の解消である．

謝辞

XXX

参考文献

[1] Cvc3 page. https://cs.nyu.edu/acsys/cvc3/. (Accessed on 02/02/2021).

[2] Github - uclid-org/uclid: Uclid5: formal modeling, verification, and synthesis of computational systems. https://github.com/uclid-org/uclid. (Accessed on 02/02/2021).

[3] Github - z3prover/z3: The z3 theorem prover. https://github.com/Z3Prover/z3. (Accessed on 02/02/2021).

[4] Opensmt — formal verification and security lab. http://verify.inf.usi.ch/opensmt. (Accessed on 02/02/2021).

[5] Smt-lib the satisfiability modulo theories library. http://smtlib.cs.uiowa.edu/. (Accessed on 01/28/2021).

[6] The yices smt solver. https://yices.csl.sri.com/. (Accessed on 02/02/2021).

[7] 岩沼宏治, 鍋島英知. SMT: 個別理論を取り扱う SAT 技術. 人工知能学会誌, Vol. 25, No. 1, pp. 86–95, 2010.

[8] Donald E. Knuth. *Satisfiability*, Vol. 4, Fascicle 6 of *The Art of Computer Programming*. Addison-Wesley Professional, 2015.

[9] 田村直之, 宋剛秀, 番原睦則. SAT ソルバーの使い方 —問題を SAT に符号化する方法—. 第 58 回プログラミング・シンポジウム予稿集, Vol. 109, pp. 165–172, 2017.

[10] 梅村晃広. SMT ソルバ・SMT ソルバの技術と応用. コンピュータソフトウェア, Vol. 27, No. 3, pp. 24–35, 2010.