卒業研究報告書

題目

遺伝アルゴリズムによる NQueen 解法

～GPU 実装～

指導教員

石水 隆　助教

報告者

07-1-037-0276

今村 光良

近畿大学理工学部情報学科

平成 24 年 1 月 31 日提出

概要

今日、様々な種類の最適化問題に対してその問題の性質に応じた解放が考案されている。しかしながら、制約条件が多い場合には問題の定式化が困難となる。そこで近年、これらの問題を解決する最適化手法として、生物進化プロセスを数理モデルとした遺伝的アルゴリズムが考案されている。遺伝的アルゴリズムは、最適化において評価関数のみに依存して解探索を行う数値計算技法である。したがって、これまで考案された最適化手法では困難であった複雑な制約条件を有する問題に対しても、厳密な問題の定式化を行うことなく有効な解探索が可能である。このような特徴を有する遺伝的アルゴリズムもいくつかの注意すべき点が存在する。例えば、最適解が複数存在する問題に遺伝的アルゴリズムを適用する場合には、1 つの解しか探索できない単純遺伝的アルゴリズムでは不十分であると思われる。一般に複数の最適解を有する問題に単純遺伝的アルゴリズムを適応するには限界があり、問題内容によって遺伝的アルゴリズムを改良しなくてはならない。今回、複数の最適解をもつ問題に対し遺伝アルゴリズムを用いてその解探索にあたる。複数の最適解をもつ問題としては NQueen 問題をとりあげる。本研究は遺伝的アルゴリズムを用いた NQueen 問題の解探索の高速化を図るため、GPU への実装を試みる。CPU のみと比べ GPU を用いるとどの程度高速化が図れるかを検証した。

目次

1　序論

1.1 本研究の背景

今日、様々な種類の最適化問題に対して数理計画法[4]やニューラルネットワーク[4]などその問題の性質に応じた解法が考案されている。しかしながら、制約条件が多い場合には問題の定式化が困難となる。そこで近年、これらの問題を解決する最適化手法として、生物進化プロセスを数理モデルとした遺伝的アルゴリズム[1][2]が考案されている。遺伝的アルゴリズムは、最適化において評価関数のみに依存して解探索を行う数値計算技法である。したがって、これまで考案された最適化手法では困難であった複雑な制約条件を有する問題に対しても、厳密な問題の定式化を行うことなく有効な解探索が可能である。このような特徴を有する遺伝的アルゴリズムもいくつかの注意すべき点が存在する。例えば、最適解が複数存在する問題に遺伝的アルゴリズムを適用する場合には、1 つの解しか探索できない単純遺伝的アルゴリズムでは不十分であると思われる。一般に複数の最適解を有する問題に単純遺伝的アルゴリズムを適応するには限界があり、問題内容によって遺伝的アルゴリズムを改良しなくてはならない。

1.2 本研究の目的

今回、複数の最適解をもつ問題に対し遺伝アルゴリズムを用いてその解探索にあたる。複数の最適解をもつ問題としては NQueen 問題をとりあげる。本研究は遺伝的アルゴリズムを用いた NQueen 問題の解探索の高速化を図るため、GPU への実装を試みる。CPU のみと比べ GPU を用いるとどの程度高速化が図れるかを検証した。

1.3 NQueen 問題

1.3.1 組み合わせ最適化問題とは

組み合わせ最適化問題[4]とは離散最適化問題のうち，解集合の定義が組合せ的条件によるものをいう。多くの組合せ的条件は、変数の整数性を含む形式で表現できるため、整数計画問題とほぼ同義的に用いられることも多い。一般に問題のサイズが大きくなるにつれ、対象とすべき解の数が爆発的に増加するため、有効な時間で最適解を得るのが困難な問題が多く含まれている。そのため、近似的な解を有効な時間や精度で求める研究も盛んである。

1.3.2 NQueen 問題とは

「8×8 のチェス盤上に、8 つのクイーンを互いに利き筋に当たらないように配置する」という古典的なパズル問題を 8 クイーン問題という。チェスのクイーンは、将棋の飛車と角を合わせた動きをする。つまり、図 3.2 に示す通り、上下、左右、それに斜めにどこまでも進むことができる。

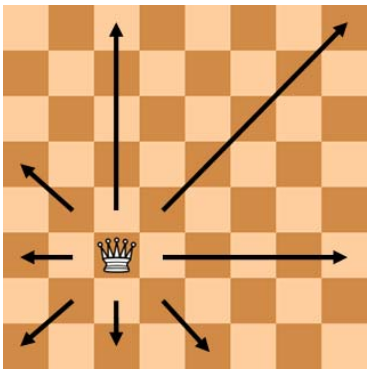

図 3.2　クイーンの利き筋

これを一般化した「N×N のマス目上に N 個のクイーンを配置する」という問題を NQueen 問題という。NQueen 問題は N が大きくなると解の量が爆発的に増え、解を求めるのに非常に多くの時間を費やすという特性がある。しかしながら現在のところこの問題を解析的な方法では解くことはできず、盤面に実際に駒を配置して確認しなければならない。

1.3.3 NQueen 問題の既知の結果

今現在 NQueen の探索方法としてよく使用されているのが Jeff Somers 氏のビット演算を用いた探索アルゴリズムである[5]。最新の研究ではこの探索アルゴリズムを改良したものが N＝26[2]までの解探索を行っている。表 1.3.3.1 に[5]に掲載されている解の探索結果を示す。また表 1.3.3.2 に今現在解明されている解の数を示す。

表 1.3.3.1 ビット演算を用いた NQueen 問題の解探索結果[5]

問題のサイズ	解の数	実行時間(時間 : 分 : 秒)
1	1	0
2	0	0
3	0	0
4	2	0
5	10	0
6	4	0
7	40	0
8	92	0
9	352	0
10	724	0
11	2680	0
12	14200	0
13	73712	0
14	365596	00:00:01
15	2279184	00:00:04
16	14772512	00:00:23
17	95815104	00:02:38
18	666090624	00:19:26
19	4968057848	02:31:24
20	39029188884	20:35:06
21	314666222712	174:53:45
22	2691008701644	?
23	24233937684440	?
24	?	?

表 1.3.3.2 最新の NQueen 問題における解探索状況[8]

N	公表日	公表機関名	基本プログラム	解の数	文献
24	2004.04.11	電気通信大学	qn24b	227,514,171,973,736	[9]
25	2005.06.11	ProActive	不明	2,207,893,435,808,350	[8]
26	2009.07.11	Tu-dresden	JSomer 版の改良版	22,317,699,616,364,000	[10]

また最近、NQueen 問題の新しいアプローチとして部分解合成法[8]というものが注目されている。これは問題における部分解を作成し、最終的にその部分解を合成して一つの全体解を作成するといものである。これは N＝21 において先に記述した[6]が約 33 時間かかるのに対して、この部分解合成法のプログラムでは約 3 時間とおよそ 10 倍程度の高速化がされている。

1.4 本報告書の構成

本研究の構成を以下に述べる。2 節では今回使用した遺伝的アルゴリズムや解探索にあたったNQueen 問題の説明、3 節では本研究の内容、4 節では結果・考察、5 節では結論・今後の課題を述べている。

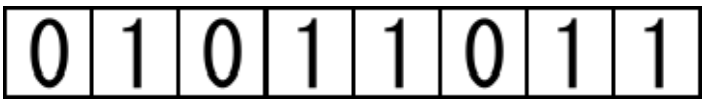

図 2.1 GTYPE の例

2 遺伝アルゴリズム

2.1 遺伝アルゴリズムとは

遺伝的アルゴリズム（Genetic Algorithm：以降は GA とする）[1][2]は 1960 年代の終り頃からミシガン大学[3]のホランド（John Holland）が基礎的な研究を重ね、提唱した考え方で、ダーウィン（Charles R. Darwin）の進化論をそのまま探索や最適解の求解に応用したものである。GA は評価関数のみに依存して解探索を行うため、問題の定式化を行うことなく有効な解探索が可能であることから、人工知能やその他の分野で注目をあつめている。進化には遺伝子（染色体）が大きく関与するが、GA においても遺伝子に相当する記号を決め、それの複数の並びを染色体とした配列が基本的な役割を担っている。この遺伝子の記号化を遺伝子コーディングと呼ぶ。GA の応用においては、解こうとする問題の何を遺伝子として表現するかがもっとも重要なポイントになる。

GA は、文字列で表現される個体集団に対し、遺伝的操作を繰り返して適用することで、近似な最適解を得ようとするアルゴリズムである。また、GA で扱かわれる情報は PTYPE と GTYPE の二つの構造から成り立っている。GTYPE は遺伝子型の集合であり、GA オペレータの操作対象となる。PTYPE は表現型であり、GTYPE の環境内での変化によって表現される大域的な行動や構造である。また、各個体が求めたい最適解とどれくらい離れているかを示す値をその個体の適合度と呼ぶ。以降は適合度の大きい数値を取るほど良い個体とする。したがって適合度が 1.0 と 0.3 の個体では前者のほうが環境により適合し生き残りやすいことを示す。本研究では、GA の GTYPE として一次元のビット列を考え、それをバイナリ表現で変換したものを PTYPE としている。また、生物学において、染色体上の遺伝子の場所を遺伝子座といい[1]、GA においては GTYPE の場所を指すのに遺伝子座という用語を転用する。例えば図 2.1 に示すような GTYPE においては 1 番目の遺伝子座の遺伝情報は 0、4 番目の遺伝子座の遺伝情報は 1、6 番目の遺伝子座の遺伝情報は 0 といったように表現されている。また 0 と 1 の 2 進数で表現される GA を特に単純 GA（以降 SGA とする）と呼ぶ[2]。以降では GA の手法の 1 つである SGA を例に記述していく。

まず、SGA のアルゴリズム[2]を次に示す。

[SGA のアルゴリズム]

① ランダムに初期個体集団を生成する
② 集団に対して選択を適用し、すぐれた個体を選ぶ
③ 集団内の個体ペアに対して交叉を適用する
④ 集団内の個体に対して突然変異を適用する
⑤ 停止条件が満たされれば終了し、満たされなければ②へ戻る

SGA のアルゴリズムにおける選択、交叉、突然変異の一回の繰返しを世代と呼ぶ。また、⑤で示されているアルゴリズムの停止条件としては

（1） あらかじめ定めた世代で終了する
（2） 一定世代間解が改善されない場合に終了する

などの条件が用いられる。これまでにさまざまな GA の手法が提案されている[2]が、それらのアルゴリズムは SGA と本質的には同じで、選択、交叉、突然変異などの遺伝的操作を繰り返して適用するものである。この SGA を用いて解くことのできる問題としては巡回セールスマン問題[1]やブール関数の充足問題[1]などがある。以降、選択、交叉、突然変異について説明する。

2.2 選択

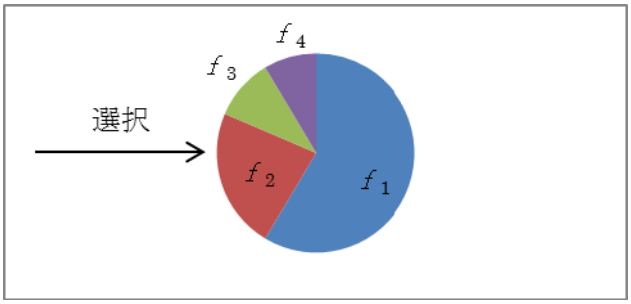

図 2.2　ルーレット法

選択とは、個体の評価に基づいて次世代の親となる個体を選ぶ操作である[1]。選択方法にはさまざまな戦略があるが、ここでは単純 GA で用いられるルーレット選択を例に選択を説明する。ルーレット法では、個体の適応度に応じた確率で次世代の個体を選択する。ある集団 P に存在する個体 i の適応度を f_i とすると、個体 i が次世代で選択される確率 p_i は、

$$p_i = \frac{f_i}{\sum_{j \notin P} f_j}$$

として計算される。したがって、次世代では適応度の大きい個体が選ばれる確率が高くなり、適応度が低い個体は選択されにくくなる。この手法を簡略的に図示したものが図 2.2 である。図 2.2 において、各 p_i の値は、各 f_i の面積で表わされる。

2.3 交叉

交叉は集団内から選ばれた 2 つの個体の間で遺伝子の部分列を交換、または組み替えて、新しい個体を生成する操作である[1]。交叉においては、集団内から選ばれた 2 つの個体に対してある一定の確率で遺伝子の部分列を交換する、または集団内から選ばれた 2 つの個体の遺伝子のうちのある割合の部分列を交換する。このとい、遺伝子の部分列が交換される確率、または遺伝子のうち交換される割合を交叉確率と言う。交叉は、GA において部分解を交換するという本質的な役割を担っているものと考えられる。交叉にもさまざまな種類があるが、ここでは単純 GA で用いられる一点交叉を例に交叉を説明する。

一点交叉は個体の遺伝子を構成する文字列のある一点を境に文字列を互いに交換する手法である。たとえば、次の 2 つの個体文字列（s_1,s_2）が与えられた場合を考える。

s_1＝01101011011
s_2＝00100101011

交叉を行う点（交叉点, crossing site）として、たとえば先頭から 4 文字目と 5 文字目の間が選ばれたとする。すると交叉後の個体は

s_1＝0110**0101011**
s_2＝0010**1011011**

というように、5 文字目以降の文字列が交換されたものとなる。このように、一点交叉は、長さ l の文字のうち、その l−1 箇所の文字間から 1 箇所をランダムに選び、それ以降の文字列を互いに交換する。

2.4 突然変異

突然変異は個体の遺伝子を構成する文字の一部を突然変異率に従って別の文字に変更する操作であり[1]、単純突然変異では個体の遺伝子を構成するそれぞれの文字について、ある一定の確率によりそれを別の文字へと変更する。この確率を突然変異確率と呼び、多くの場合 0.1～0.01 程度の小さい値が用いられる。遺伝子がビット列の場合は、突然変異が発生すると遺伝子座の 0 と 1 が入れ替えられる。遺伝子座が整数などの数値の場合であれば、10 進数を 2 進数に置き換えて、ビット列として突然変異を起こす。文字の場合であれば、文字をビット列のバイナリ表現としてあつかい、ビット列として突然変異を発生させる。

たとえば遺伝子座がビット列の場合、次に示す個体 i の遺伝子 S_i に単純突然変異を適用する場合を考える。

$$S_i = 01101011011$$

ここで突然変異を適用すると、それぞれの文字が、突然変異確率で別の文字に文字を変化する。ここでは 5 文字目でその変化がおきたものと仮定する。その場合、5 文字目が 1→0 と変化するため、突然変異後の個体は

$$S'_i = 0110\underline{0}011011$$

となり、突然変異により個体 i の遺伝子が S_i から S'_i に変化したことがわかる。

突然変異は、選択と組み合わせることで局所探索を実現している。多くの最適化問題は、適応度が最大となる最適解以外に局所的に極大となる局所解を持つ。GA において、多くの個体が局所解の周囲に集まると、より広い範囲の探索が困難になり、最適解が出にくくなる。突然変異を用いることにより、探索範囲を局所解周辺から離し、より広い範囲の探索を行えるようになる。突然変異は、GA において、評価型の個体が局所解に陥るのを防ぎ、より広い範囲での最適解の探索を可能にするために行われ、交叉を補佐する 2 次的な役割を担っているものと考えられる。

3　遺伝的アルゴリズムを用いた NQueen 問題の解探索方法

実際に N クイーン問題を解くにあたり、遺伝的アルゴリズムを問題に適応させなくてはならない。そこで本章では本研究で行った単純 GA を N クイーン問題に適応させたプログラムの仕様および説明を記述する。

3.1 遺伝子コーディング

前述したように、SGA は遺伝子を 0 と 1 の 2 進数で表現する。しかし、今回 N クイーン問題を取り扱う場合においては 2 進数ではなく、Y 座標 j ($0 \leqq j < N$)に配置されている駒の X 座標を j 番目の遺伝子座の遺伝情報として持つ遺伝子とする。つまり N が 8 であれば、遺伝子は長さ 8 の数列であり、各遺伝子座の遺伝情報は 0～7 の数値で表現される。図 4.1 に N=8 の場合の遺伝子コーディング例を示す。

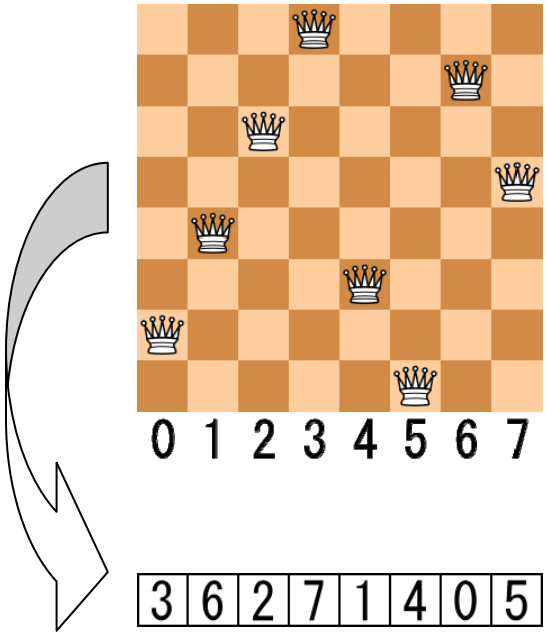

図 4.1 遺伝子コーディング例

3.2 遺伝子集団の決定

本研究では、遺伝子の集団は二次元配列 a[M][N]を用いて表現する。ここで、M は遺伝子の集団数であり、N はチェス盤のサイズである。配列の要素 a[i][j]には、個体 i ($0 \leqq i < M$)の j 番目の遺伝子座の遺伝情報の値、すなわち、Y 座標 j の駒の X 座標の値が設定されるとする。

例として、M=20、N=8 とし、図 4.2 のような初期集団を生成したとする。このとき、配列 a[0]={0,1,2,3,4,56,7}、配列 a[1]={2,7,4,1,5,3,0,6}として表わされる。

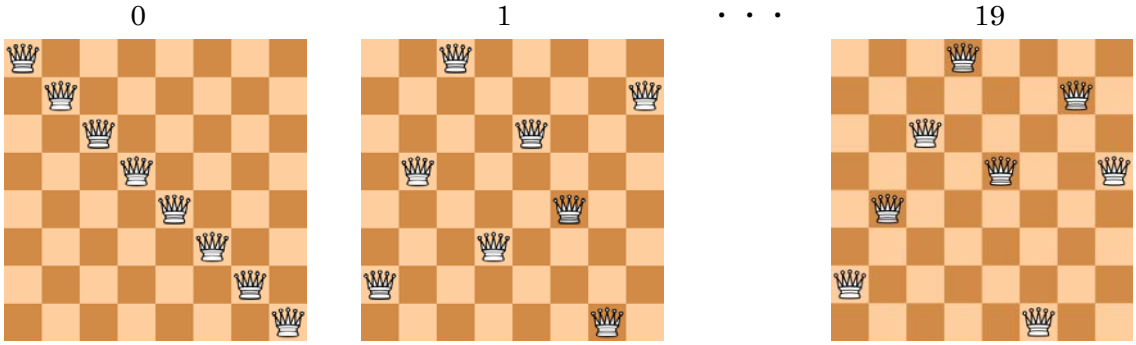

図 4.2　初期集団の状態

3.3 遺伝子の評価方法

本研究では、ある配置において複数個の駒が存在する縦横斜めのラインの数の和を競合数と呼ぶ。本研究における遺伝子の評価方法は、駒の配置情報からなる競合数の大きさによって決定する。つまり、競合数がおおくなると悪く、競合数が少なくなると良いと判断する。また解となった場合を最適解とし、その時の競合数は 0 となる。以降では個体 i ($0 \leqq i < M$)の競合数を c_i と表す。

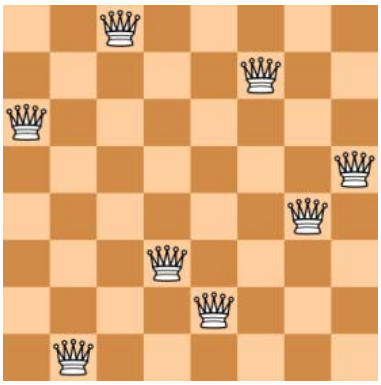

図 4.3　駒の配置状態の例

3.4 選択方法

本研究における選択方法は、SGA で紹介したルーレット選択を用いる。本研究では、個体 i ($0 \leqq i < M$)の選ばれる選択確率 p_i を以下の式で定義する。

$$p_i = \frac{1}{1 + c_i}$$

たとえば、個体 i の競合数 c_i が 4 であれば個体 i がルーレット選択により選択される確率は 0.2 となる。ルーレットの回転方法として擬似的なルーレットを作成するために p_i を個体 i の選択確率とし、累積確率 q_i を以下の式で定義する。

$$q_i = \begin{cases} \dfrac{p_0}{\sum_{j \epsilon p} p_i}, & if(i = 0) \\ \dfrac{p_i}{\sum_{j \epsilon p} p_i} + q_{i-1}, & if(i \geq 1) \end{cases}$$

個体選択する際は、乱数 r ($0 \leqq r < 1$)を発生させ、$q_{i-1} \leqq r < q_i$ を満たす個体 i を選択する。

3.5 交叉方法

本研究における交叉方法は単純 GA で使用される一点交叉を用いる。まず交叉の方法としては、親となる集団から M が偶数であれば M /2 組、奇数であれば(M -1)/2 組の交叉するペアを作成する。次に、ペアごとに交叉発生率による交叉の発生を判定する。

たとえば以下の図 4.5.1 に示す親 1 と親 2 で交叉が発生したとする。

親 1　　親 2

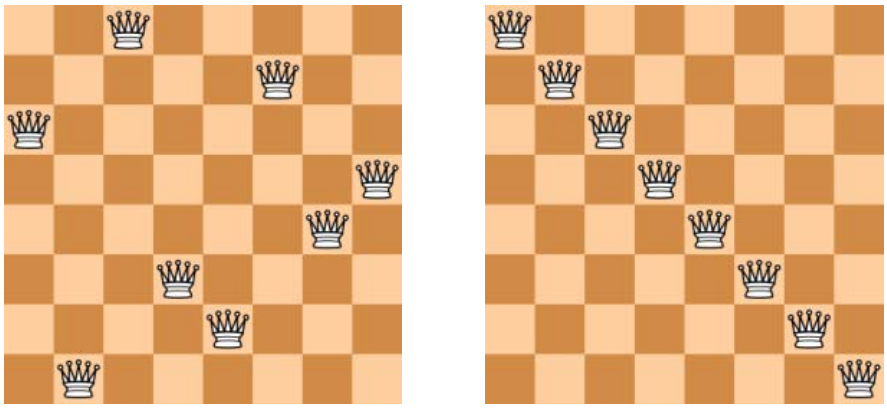

図 4.5.1　交叉前

交叉対象が決定すると次に交叉点が乱数により決まる。今回は交叉点として 4 が選ばれたとする。これにより、遺伝子座の 4 番目以降の遺伝情報が交換され、新たに 2 つの子遺伝子が誕生する。その誕生した子遺伝子を以下の図 4.5.2 に示す

子 1　　子 2

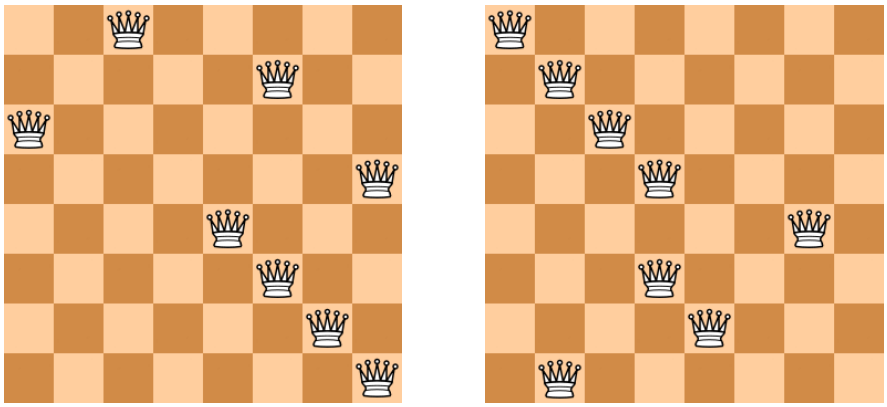

図 4.5.2　交叉後

図 4.5.2 より、4 番目の遺伝子座以降で遺伝情報が交換されているのがわかる。N クイーン問題の遺伝アルゴリズムはこのように交叉を発生させる。

3.6 突然変異方法

本研究における突然変異の発生方法は、遺伝子ごとに突然変異の発生が判断され、もし突然変異が発生した場合、乱数により遺伝子座をランダムに設定し、決定した遺伝子座の情報を 0～N -1 の間で状態変異を起こさせる。例えば今図 4.6.1 に示す遺伝子の 5 番目の遺伝子座で突然変異が起こるとする。

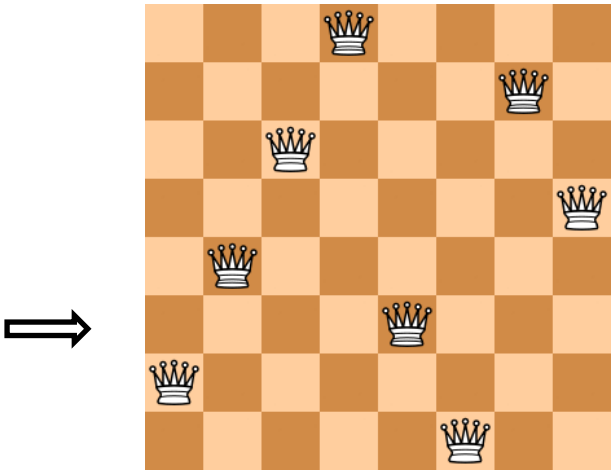

図 3.6.1 突然変異発生前

図 3.6.1 において、突然変異発生前の 5 番目の遺伝子座の位置情報は 4 である。ここで 5 番目の遺伝子座の遺伝情報をランダムに生成した値と交換する突然変異を発生させる。突然変異発生後の遺伝子を図 4.6.2 に示す。

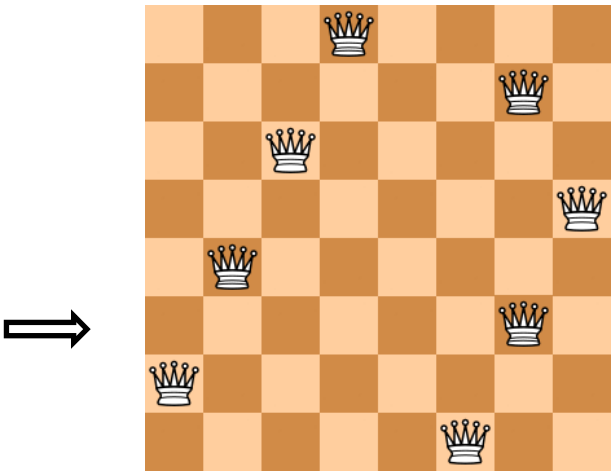

図 3.6.2 突然変異発生後

図 3.6.1 および図 3.6.2 より突然変異の発生で 5 番目の遺伝子座の位置情報が 6 となっていることがわかる。N クイーン問題の遺伝アルゴリズムはこのように突然変異を発生させていく。

4　研究内容

4.1 GPU 実装のための準備

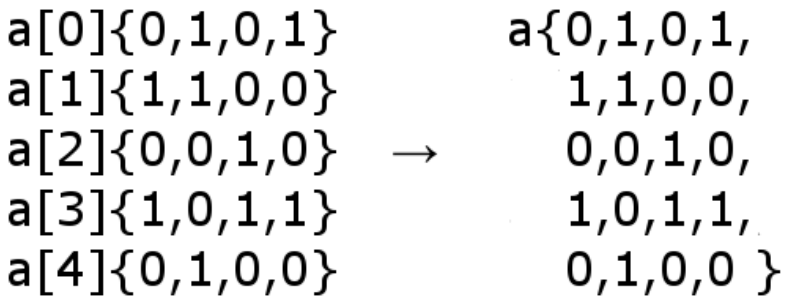

図 4.1 2 次元配列を 1 次元配列に変換する例

今回作成した遺伝的アルゴリズム用いた NQueen 問題の解探索プログラムは 2 次元配列である。GPU の開発環境として選択した CUDA では多次元配列も扱えるが、1 次元配列を扱う方が圧倒的に楽で効率がよい[7]ので、NQueen 問題の解探索プログラムを 2 次元配列から 1 次元配列に書き換え、2 次元配列でのデータアクセスと同じアクセスを 1 次元配列で実現する。本研究では、2 次元配列 a[M][N]を 1 次元配列 a'[M*N]に置き換え、要素 a[*i*][*j*]の値は a'[*i* *N +*j*]に格納する。図 4.1 に 2 次元配列を 1 次元配列に置き換えた例を示す。

4.2 CUDA を用いた高速化

次に 4.1 章で 1 次元配列に変換した c++のプログラムを GPU で扱えるよう CUDA を使用したプログラムに書き換える。付録に本研究で作成した CUDA を使用した c++言語プログラムを示す。また、今回使用した CUDA の変数などの意味を以下で述べる。

CUT_DEVICE_INIT();
デバイスの初期化を行う。また、実行されると、使用している GPU の名前が表示される

CUDA_SAFE_CALL()
この関数で囲むと通常の実行時は、引数として与えられた CUDA 関数の呼び出す。
_DEBUG が定義されていると、引数の関数を呼び出し、その戻り値のエラーを検査し、
エラーの場合はエラーメッセージを出力する。

cudaMalloc(void* dst, const void* src, size_t count, enum cudaMemcpyKind kind)
ホスト(CPU)からデバイス(GPU)にデータをコピーします。
dst：転送先のメモリアドレス
src：転送元のメモリアドレス
count：コピーするデータのサイズ
kind：転送の種類
転送の種類としては
cudaMemcpyHostToDevice：ホスト(CPU)からデバイス(GPU)に転送
cudaMemcpyDeviceToHost：デバイス(GPU)からホスト(CPU)に転送
などがある。

cudaMalloc(void** devPtr, size t size)
デバイス(GPU)のメモリ領域を確保します。
devPtr：デバイスメモリのポインタ
size：メモリのサイズ

関数名<<<ブロック数,スレッド数>>>(引数);
設定したブロック数、スレッド数を用いて関数に引数を渡し、実行する。

CUT_EXIT()
デバイスを終了する。

本研究では、付録に示すプログラムを用いて CUDA によって *n*=(*n* の範囲)に対する *n* クイーン個数問題の解を求めた。以下に本研究で用いた計算機のスペックを示す。

CPU：Intel(R)Celeron(R) CPU G530 @ 2.4GHz
GPU：Geforce 210
OS：windouws7Homepremium

表 6.1.1 プログラムを実行した条件

N	集団数 M	終了世代数	選択方法	交叉方法	突然変異率	試行回数
8	100	1000	エリート選択	一様交叉	0.1	1000

表 5.1 プログラムを実行した条件

N	終了世代数	選択方法	交叉方法	突然変異率	試行回数
8	1000	エリート選択	一様交叉	0.1	1000

表 5.2 各集団サイズにおける解発見数および探索時間

集団サイズ	解発見数	CPU のみ[ms]	CPU+GPU[ms]
10	8	362	200
100	14	3474	1131
1000	78	76655	12684
10000	92	1742139	390623

5 結果・考察

今回は同一のプログラムを用いているので、CPU のみを用いた場合と、GPU も使用した場合とで解探索の性能は同じである。表 5.1 にプログラムを実行した条件を示す。また、表 5.2 に発見した解の個数および解発見に要した時間を示す。まず CPU のみのプログラムと GPU を用いたプログラムを比較するため、遺伝的アルゴリズムは同一のものを使用し、N=8 のときにおける初期集団の数による速度の比較を行った結果を図 1 に示す。集団に含まれる個体の数を 100 で 1、2 秒程度しか速度に開きはないが、10000 になると、CPU では約 30 分程度かかるのに対して、GPU では約 6 分程度と 5 倍近くの速度さがでた。その一方で、問題サイズ 1000 の場合、約 7 倍と、当初問題サイズを大きくすれば大きくなるほど速度が向上予定であったが、小さいサイズの問題の方が、高速化できたという結果になった。原因として考えられたのが、本プログラムではすべての処理を GPU が行うのではなく、一部分のみを GPU が行うため、CPU と GPU 間でデータを通信しなくてはならない。そのため、問題サイズが大きくなるとその分データ通信も大きくなり、高速化した時間に対して通信コストが大きくなり、思った以上の速度がでていないのではないかと考えられる。

本研究では高速化を目標としてきたが、表 1.3.3.1 に示されている実行速度と比較すると高速化できたとは言い難い。これは既存の方法が再帰的なアルゴリズムをもちいているため、確率的なアルゴリズムを用いているため、解の生成数が一遺伝状況により一定ではないからではないだろうか。しかしながら、再帰的なアルゴリズムを用いているため、一定の時間で生成される解は毎回同じで、毎回違う解が生成される本アルゴリズムと単純比較することはできないと考えられる。そういった点でいえば、元のアルゴリズムを高速化できたのではないだろうか。

6　結論・今後の課題

本研究は遺伝的アルゴリズムを用いた NQueen 問題の解探索の高速化を図るため、GPU への実装を試みた。CPU のみと比べ GPU を用いると高速化することができた。しかしながら、目的としていたすべての処理を GPU で行うことができず、目標としていた高速化までとはいかなかった。今後の課題としては、全ての処理を GPU で行えるようプログラムの改良が必要となってくる。

謝辞

本研究において御指導してくださった石水隆先生にはご迷惑をかけましたお詫びと卒業研究完成に至れた感謝をこの場を借りて申し上げます。また共同研究者のみなさまの御助力への感謝をこの場を借りて申し上げます。

参考文献

[1] 伊庭斉志. 遺伝的アルゴリズムの基礎. オーム社, 1994.

[2] 棟朝雅晴. 遺伝的アルゴリズム―その理論と先端的手法. 森北出版, 2008.

[3] University of Michigan, 2011, http://www.umich.edu/

[4] 同志社大学 知的イシステムデザイン研究室 ゼミ資料, 1999,

[5] http://mikilab.doshisha.ac.jp/dia/seminar/1999/optim/optim01.pdf

[6] Jeff Somers's N Queens Solutions, http://www.jsomers.com/nqueen_demo/nqueens.html

[7] NQueen 問題(解の個数を求める), http://www.ic-net.or.jp/home/takaken/nt/queen/index.html

[8] 青木尊之. はじめての CUDA プログラミング―驚異の開発環境[GPU+CUDA]を使いこなす!, 工学社, 2009.

[9] 萩野谷一二, “NQueen 問題への新しいアプローチ(部分解合成法)について,” 情報処理学会報告書, Vol.2011-GI-26, No.11, 2011.

[10] 吉瀬謙二, N-Queens Homepage in Japanese, 電気通信大学, 2004,

[11] http://www.arch.cs.titech.ac.jp/~kise/nq/index.htm

[12] Queen@TUD, Technische Universitat Dresden, 2009, http://Queens.inf.tu-dresden.de/

付録

以下に本研究で作成したプログラムを示す。

main.cu

```
#include <iostream>
#include <ctime>
#include <cstdlib>
#include <cutil.h>
#include "set.h"

#include<vector>//リスト構造を持つ配列vectorを使用するため
#include<string>//stringを使用するため
#include<sstream>//int型の変数をstring型に変換するために使用

using namespace std;


/**
*@param gpua CPU側で遺伝子情報を保存しているaをGPU側での扱うための変数
*/
__global__ void inita(int *gpua){
    // スレッドID
          int k = blockDim.x*blockIdx.x+threadIdx.x;
          gpua[k] = threadIdx.x;

}


/**
*@param gpusum GPU側で計算した競合数を保存する変数
*/
__global__ void initsum(int *gpusum){
    // スレッドID
          gpusum[blockIdx.x]=0;
}

/**
*@param gpua CPU側で遺伝子情報を保存しているaをGPU側での扱うための変数
*@param gpup GPU側で右斜め上の競合数を判定する変数
*@param gpuq GPU側で右斜め下の競合数を判定する変数
*@param gpur GPU側で縦の競合数を判定する変数
*@param gpusum GPU側で計算した競合数を保存する変数
*/
__global__ void check(int *gpua,int *gpup,int *gpuq,int *gpur,int *gpusum){

          int k1 = blockDim.x*blockIdx.x+threadIdx.x;

          if(gpup[(threadIdx.x+gpua[k1])+(((NBIT*2)-1)*blockIdx.x)]==0){
                    gpup[(threadIdx.x+gpua[k1])+(((NBIT*2)-1)*blockIdx.x)]=1;
          }
          else{
                    gpusum[blockIdx.x] +=1;
          }

          if(gpuq[(threadIdx.x-gpua[k1]+(NBIT-1))+(((NBIT*2)-1)*blockIdx.x)]==0){
                    gpuq[(threadIdx.x-gpua[k1]+(NBIT-1))+(((NBIT*2)-1)*blockIdx.x)]=1;
```

```
		}
		else{
			gpusum[blockIdx.x]+=1;
		}

		if(gpur[gpua[k1]+(NBIT*blockIdx.x)] ==0){
			gpur[gpua[k1]+(NBIT*blockIdx.x)] =1;
		}
		else{
			gpusum[blockIdx.x]+=1;
		}

}

int main(int argc, char** argv){

		int a[N*NBIT]; //遺伝情報格納配列
		int match[N]; //競合数保存用
		int *gpua; //GPU用遺伝情報格納配列
		int *gpup; //GPU用右斜め上競合判定用配列
		int *gpuq; //GPU用右斜め下競合判定用配列
		int *gpur; //GPU用縦競合判定用配列
		int *gpusum; //競合数保存用

		srand((unsigned)time(NULL));

		vector<string> vector; //出現解保存用リスト型配列

		void func(int [],int [N],std::vector<std::string>&);
		void elite(int [],int [N]);
		void select1(int[],int [N]); //エリート選択用

		void select(int[],int [N]); //エリート用

		void cross(int []);//交叉点以降の交換
		void cross1(int []);

		void mutation(int [],int [N]);
		void answercheck(int [],int [],std::vector<std::string>&);

		//デバイスの初期化
	CUT_DEVICE_INIT(argc, argv);

		// デバイス(GPU)のメモリ領域確保
		CUDA_SAFE_CALL(cudaMalloc((void**)&gpua, sizeof(int)*(NBIT*2-1)*N));
		CUDA_SAFE_CALL(cudaMalloc((void**)&gpup, sizeof(int)*(NBIT*2-1)*N));
		CUDA_SAFE_CALL(cudaMalloc((void**)&gpuq, sizeof(int)*(NBIT*2-1)*N));
		CUDA_SAFE_CALL(cudaMalloc((void**)&gpur, sizeof(int)*(NBIT*N)));
		CUDA_SAFE_CALL(cudaMalloc((void**)&gpusum, sizeof(int)*N));


		//ブロックとスレッドの数を設定
	dim3 blocks(N,1);
	dim3 threads(NBIT,1);


		CUDA_SAFE_CALL(cudaMemcpy(gpua, a, sizeof(int)*(N*NBIT), cudaMemcpyHostToDevice));
		CUDA_SAFE_CALL(cudaMemcpy(gpusm, match, sizeof(int)*N, cudaMemcpyHostToDevice));
```

```
        inita<<< blocks, threads >>>(gpua); //集団の初期化
        check<<< blocks, threads >>>(gpua,gpup,gpuq,gpur,gpusum);//競合数判定
        // デバイス(GPU)からホスト(CPU)へ転送
    CUDA_SAFE_CALL(cudaMemcpy(a, gpua, sizeof(int)*(N*NBIT), cudaMemcpyDeviceToHost));
        CUDA_SAFE_CALL(cudaMemcpy(match, gpusum, sizeof(int)*N, cudaMemcpyDeviceToHost));

        for(int i =0;i<N;i++){
                printf("%d番目の競合度は%d ¥n",i,match[i]);
        }


        unsigned int timer = 0;
        CUT_SAFE_CALL( cutCreateTimer( &timer));//タイマーの生成
        CUT_SAFE_CALL( cutStartTimer( timer));//スタート


        for(int i=0;i<MAX;i++){
                elite(a,match);

                select(a,match);
                cross(a);//一様交叉
                mutation(a,match);

        }

        CUT_SAFE_CALL( cutStopTimer( timer));//ストップ

        int w = vector.size();
        printf("resolt is %d¥n",w);
        for(int i =0;i<w;i++){
                std::cout << "解" << i+1 << ":"<< vector[i] << std::endl;
        }


        for(int i =0;i<N;i++){
                printf("%d番目の競合度は%d ¥n",i,match[i]);
        }

        printf( "Processing time: %f (ms)¥n", cutGetTimerValue( timer));//結果表示
        // 終了処理
    CUT_EXIT(argc, argv);

}


/**
*@param x 乱数生成のための元となる値
*@return 0～1の乱数を返す
*/
float rnd(short int x){
        static short int ix=1,init_on=0;
        if((x%2) && (init_on==0)){
                ix=x;
                init_on=1;
        }
        ix=899*ix;
        if(ix<0)
                ix=ix+32767+1;
```

```
        return((float)ix/32768.0);
}
```

func.cpp

```
#include <iostream>
#include "set.h"

#include<vector>
#include<string>
#include<sstream>

using namespace std;
/**
*@param a[] 遺伝情報を格納している配列
*@param match[] 各遺伝子の競合数を格納している
*@param vector 発見した解を格納している
*/
void func(int a[],int match[N],std::vector<std::string>& vector){
        int i=0;
        int sum=0;
        int p[NBIT*2-1]; //右斜め上判定用配列
        int q[NBIT*2-1]; //右斜め下判定用配列
        int r[NBIT]; //縦列判定用配列

        std::ostringstream l;//string型を連結保存できる変数


        //集団に属する各盤上の駒の判定を盤の数(N個)だけ行う
        for(int x=0;x<N;x++){

                //右斜め判定用配列の初期化を行っている
                for(int x1=0;x1<NBIT*2-1;x1++){

                        p[x1]=0;
                        q[x1]=0;
                }
                //縦列判定用配列の初期化を行っている
                for(int x2=0;x2<NBIT;x2++){
                        r[x2]=0;
                }


                //駒の数(NBIT個)だけ各駒について競合数を算出する
                for(int y=0;y<NBIT;y++){
                        int k = NBIT*x + y;
                        i=a[k];// x集団のy列目コマの配置情報

                        //右斜め上判定
                        if(p[y+i]==0){
                                p[y+i]=1;
                        }
                        else{
                                sum+=1;
                        }

                        //右斜め下判定
                        if(q[y-i+(NBIT-1)]==0){
```

```
                                        q[y-i+(NBIT-1)]=1;
                                }
                                else{
                                        sum+=1;

                                }

                                //縦の判定
                                if(r[i]==0){
                                        r[i]=1;
                                }
                                else{
                                        sum+=1;
                                }

                        }

                        //集合x番目の競合数がわかった
                        match[x]=sum;//競合数をmatchに保存


                        //競合数が0の場合解の情報を保存する

                        if(sum==0){
                                std::ostringstream l;//string型を連結保存できる変数
                                for(int i=0;i<NBIT;i++){
                                        int g = NBIT*x + i;
                                        l<<a[g];
                                }
                                vector.push_back(l.str());

                                //重複解を見つけて取り出す
                                int u = vector.size();
                                for(int j=0;j<u-1;j++){
                                        if(vector[j]==vector[u-1]){
                                                vector.pop_back();
                                                break;
                                        }
                                }
                        }
                        sum = 0;//競合数の初期化
                }
        }
```

select.cpp

```
#include <iostream>
#include<iostream>
#include <vector>
#include<string>
#include<sstream>

#include"set.h"

using namespace std;
/**
*@param match[] 各遺伝子の競合数を格納している
*/
```

```
void select(int a[],int match[N]){

	int newa[N*NBIT];
	int cnum =0;
	int newm[N];

	//エリートを選択
	for(int i1=0;i1<N;i1++){
		for(int j1=0;j1<NBIT;j1++){
			int x1 = NBIT*cnum+j1;
			int x2 = NBIT*i1+j1;
			newa[x2] = a[x1];
			newm[i1] = match[cnum];
		}
		cnum +=1;
		if(cnum==elitenumber){
			cnum = 0;
		}

	}

	//選択されたエリートを保存する
	for(int s=0;s<N;s++){
		for(int t=0;t<NBIT;t++){
			int x3 = NBIT*s+t;
			a[x3] = newa[x3];
			match[s] = newm[s];
		}
	}
}
```

elite.cpp

```
#include <iostream>
#include "set.h"
/**
*@param a[] 遺伝情報を格納している配列
*@param match[] 各遺伝子の競合数を格納している
*/
void elite(int a[],int match[N]){

	int cha[NBIT];
	int chm=0;

	for(int i=0;i<N;i++){
		for(int j=0;j<N;j++){
			if(match[i]>match[j]){
				//交換用遺伝子の保存
				for(int p=0;p<NBIT;p++){
					int x1 = NBIT*j +p;
					cha[p]=a[x1];
				}
				chm = match[i];

				//遺伝子の交換
				for(int q=0;q<NBIT;q++){
					int x2 = NBIT*j+q;
					int x3 = NBIT*i+q;
```

```
                                        a[x3] =a[x2];
                                }
                                match[i]= match[j];

                                //遺伝子の交換
                                for(int r=0;r<NBIT;r++){
                                        int x4 = NBIT*j+r;
                                        a[x4]=cha[r];
                                }
                                match[j]= chm;
                        }
                }
        }
}
```

cross.cpp

```
#include<iostream>
#include"set.h"
/**
*@param a[] 遺伝情報を格納している配列
*/
void cross(int a[]){


        int w[NBIT];//ランダム保管用配列
        int crossa[NBIT];//交叉保管用配列
        float rnd(short int); //0～1までのランダム関数
        float rnd0; //ランダム関数保存用

        //a[][]をランダムに並び替え
        for(int i=0;i<N; i++){
                for(int j=0;j<NBIT;j++){
                        int x1 = NBIT*i+j;
                        w[j] = a[x1];
                }

                int r=rand()%(N);
                for(int s=0;s<NBIT;s++){
                        int x2 = NBIT*r+s;
                        int x3 = NBIT*i+s;
                        a[x3] = a[x2];
                }
                for(int t=0;t<NBIT;t++){
                        int x4 = NBIT*r+t;
                        a[x4] = w[t];
                }
        }

        //a[i][]とa[i+1][]が交叉するかランダムに決めていく

                for(int x=0;x<N;x++){

                        int r = rand()%(100);
                        rnd0 = rnd(r);//交叉が起こるかどうかの判定
                        //保存用配列の初期化
                        for(int e=0;e<NBIT;e++){
                                crossa[e]=0;
```

```
                                        }

                                        if(crate<rnd0){
                                                int cross_point = rand()%(NBIT);
                                                for(int x1=cross_point;x1<NBIT;x1++){
                                                        int k1 = NBIT*x+x1;
                                                        crossa[x1] = a[k1];
                                                }
                                                for(int x2=cross_point;x2<NBIT;x2++){
                                                        int k2 = NBIT*x+x2;
                                                        int k3 = NBIT*(x+1)+x2;
                                                        a[k2] = a[k3];
                                                }
                                                for(int x3=cross_point;x3<NBIT;x3++){
                                                        int k4 = NBIT*(x+1)+x3;
                                                        a[k4] = crossa[x3];
                                                }
                                                x +=1;
                                        }
                                        else{
                                                x +=1;
                                        }
                                }
}
```

mutation.cpp

```
#include<iostream>
#include"set.h"
#include<cstdlib>
#include<ctime>

using namespace std;
/**
*@param a[] 遺伝情報を格納している配列
*@param match[] 各遺伝子の競合数を格納している
*/
void mutation(int a[],int match [N]){

                float p = 0.0; //事前に決めていた突然変異の発生確率を格納する変数
                p = (float)mrate; //突然変異確率の格納

        float rnd(short int);//mainメソッドで作成したランダム関数を呼び出している
                float q = 0.0;//rnd()の変数を格納する変数を作っている
                q = rnd(9);//変数にrnd()で作成した変数を格納している




                for(int i=0; i<N; i++){

                        q = rnd(9);//変数にrnd()で作成した変数を格納している

                        //競合数が0で解になっている場合は必ず突然変異が起こる
                        if(match[i]==0){
                                int e =rand()%(NBIT); //突然変異の起こる位置
                                int k1 = NBIT*i+e;
                                int g = a[k1];
```

```
                                        int j =rand()%(NBIT); //コマの位置情報の置き換え
                                        int k2 = NBIT*i+j;
                                        a[k1] = a[k2];     //i番目のyの位置で突然変異が起こり、zの位置情
報に置き換え
                                        a[k2] = g;
                                }

                                //突然変異が起こった場合
                                else {
                                        if(p < q){
                                                int y =rand()%(NBIT); //突然変異の起こる位置
                                                int k3 = NBIT*i+y;
                                                int w = a[k3];
                                                int z =rand()%(NBIT); //コマの位置情報の置き換え
                                                int k4 = NBIT*i+z;
                                                a[k3] = a[k4];     //i番目のyの位置で突然変異が起こり、
zの位置情報に置き換え
                                                a[k4] = w;
                                        }
                                }
                        }
}
```

answercheck.cpp

```
#include <iostream>
#include "set.h"
#include<vector>
#include<string>
#include<sstream>

using namespace std;
/**
*@param a[] 遺伝情報を格納している配列
*@param match[] 各遺伝子の競合数を格納している
*@param vector 発見した解を格納している
*/
void answercheck(int a[],int match[N],std::vector<std::string>& vector){

        for(int i=0;i<N;i++){
                std::ostringstream l;//string型を連結保存できる変数
                if(match[i]==0){
                        //std::ostringstream l;//string型を連結保存できる変数
                        for(int j1=0;j1<NBIT;j1++){
                                int g = NBIT*i + j1;
                                l<<a[g];
                        }
                        vector.push_back(l.str());

                        //重複解を見つけて取り出す
                        int u = vector.size();
                        for(int j=0;j<u-1;j++){
                                if(vector[j]==vector[u-1]){
                                        vector.pop_back();
                                        break;
                                }
                        }
                }
```

```
        }
}
```

set.h

```
#define N 100 //初期集団の数
#define NBIT 4 //クイーンの位置情報

#define mrate 0.90 //突然変異の起こる確率
#define crate 0.40 //交叉の起こる確率

#define permatch 2 //許容競合数
#define elitenumber 5 //エリートとする数

#define MAX 1000 //何世代まで求めるか
```