卒業研究報告書

題目

遺伝アルゴリズムによる NQueen 解法

問題特性に着目した突然変異方法の改善

指導教員

石水 隆　助教

報告者

08-1-037-0092

瀬渡 昭良

近畿大学理工学部情報学科

平成 24 年 1 月 31 日提出

概要

近年、問題の定式化を必要とせず、最適解を導く手法の一つとして遺伝的アルゴリズムが提案された。この遺伝的アルゴリズムはダーウィンの進化論、正確にはネオダーウィニズムの考え方を、ほとんどそのままアルゴリズム化したものである。すなわち与えられた自然環境の中で、個体集団の各個体同士が交配と突然変異を繰り返しながら、その自然環境によく適応する個体ほど生き残り子孫を増やすことができるようにするアルゴリズムである。しかしながら、この遺伝的アルゴリズムは最適解が複数になると、全ての解を求めることが困難になるという問題がある。したがって遺伝的アルゴリズムを用いて、最適解が多峰性を持つ問題を解く場合、アルゴリズムを改良する必要がある。本研究では突然変異において改良を行い、全ての解を探索できる遺伝アルゴリズムを提案する。また、本研究では、最適解が多峰性を持つ問題として NQueen 問題を取り上げ、遺伝アルゴリズムによる解探索を行う。

目次

1　序論

1.1 本研究の背景

近年、問題の定式化を必要とせず、最適解を導く手法の一つとして遺伝的アルゴリズムが提案された。この遺伝的アルゴリズムはダーウィンの進化論、正確にはネオダーウィニズムの考え方を、ほとんどそのままアルゴリズム化したものである。すなわち与えられた自然環境の中で、個体集団の各個体同士が交配と突然変異を繰り返しながら、その自然環境によく適応する個体ほど生き残り子孫を増やすことができるようにするアルゴリズムである。しかしながら、この遺伝的アルゴリズムは最適解が複数になると、全ての解を求めることが困難になるという問題がある。したがって遺伝的アルゴリズムを用いて、最適解が多峰性を持つ問題を解く場合、アルゴリズムを改良する必要がある。

1.2 本研究の目的

単純な遺伝的アルゴリズム用いて、最適解が多峰性を持つ問題を解くには限界がある。そこで今回、最適解が多峰性を持つ問題として NQueen 問題を取り上げ、最適解が多峰性を持つ問題でも解探索が行えるように遺伝的アルゴリズムの改良にあたった。最適解が多峰性をもつことから、通常であれば局所解からの脱出をはかる突然変異を、最適解からも脱出させ、新たな解を探索させることが必要となってくる。1 つの解に収束するだけでなく、収束したのちにまた別の解に収束するよう、多様性を持たせなくてはならい。そこで本研究では突然変異において改良を行い、NQueen 問題の解探索を試みた。

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[6]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。

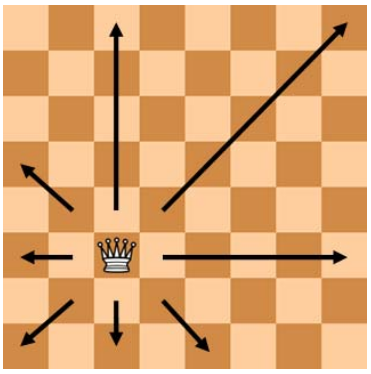

図 3.2　クイーンの利き筋

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[3]。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[5]までの解探索を行っている。表 1.3.3.1 に[3]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[3]

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[5]

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	[7]
25	2005.06.11	ProActive	不明	2,207,893,435,808,350	[5]
26	2009.07.11	Tu-dresden	JSomer 版の改良版	22,317,699,616,364,000	[9]

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[5]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 において図 1.3.3.2 の N＝26 のプログラムを用いると約 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

本報告書では 2 章においては本研究で使用した遺伝的アルゴリズムについて、3 章では遺伝的アルゴリズムを用いた NQueen 問題の解探索方法について、4 章においては研究内容について、5 章においては結果と考察について、6 章においては結論と今後の課題についてを記述する。

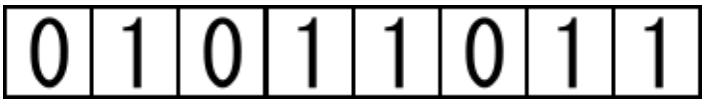

図 2.1 GTYPE の例

2　遺伝的アルゴリズム

2.1 遺伝的アルゴリズムとは

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[8]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

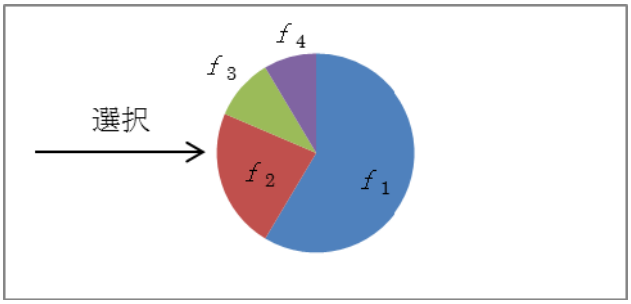

図 2.2　ルーレット法

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、

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

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う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**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

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 遺伝子集団の設定

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。

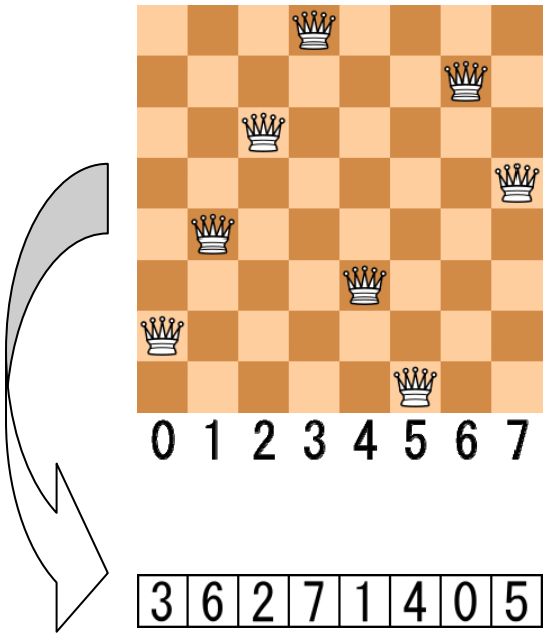

図 4.1 遺伝子コーディング例

3.2 遺伝子コーディング

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。

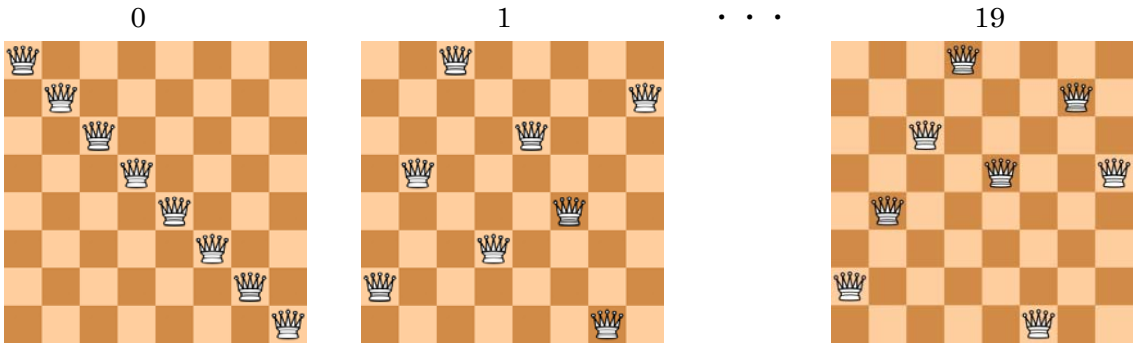

図 4.2　初期集団の状態

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。

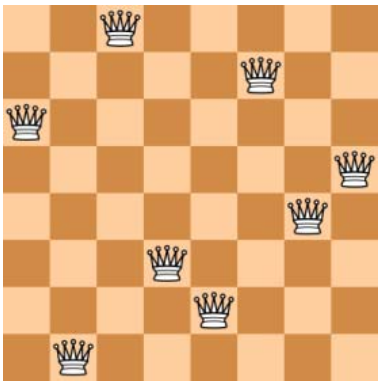

図 4.3　駒の配置状態の例

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。

$$q_i = \begin{cases} \dfrac{p_0}{\sum_{j \in P} p_j} & (if \quad i = 0) \\ \dfrac{p_i}{\sum_{j \in P} p_j} + q_{i-1} & (if \quad i \geq 1) \end{cases}$$

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

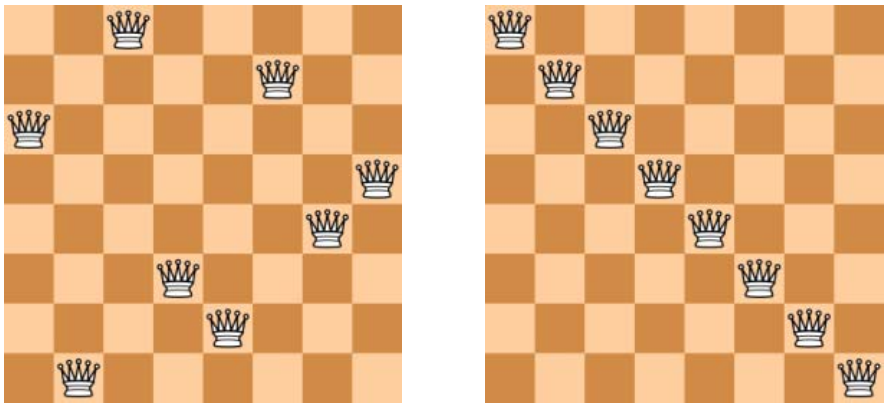

図 4.5.1　交叉前

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

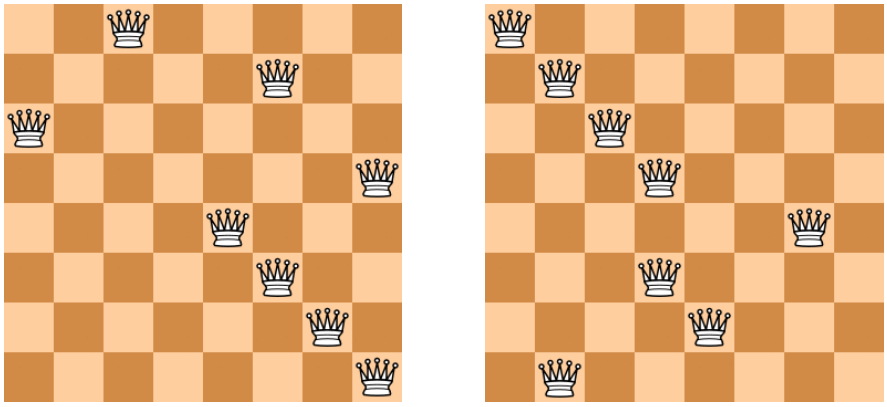

図 4.5.2　交叉後

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。

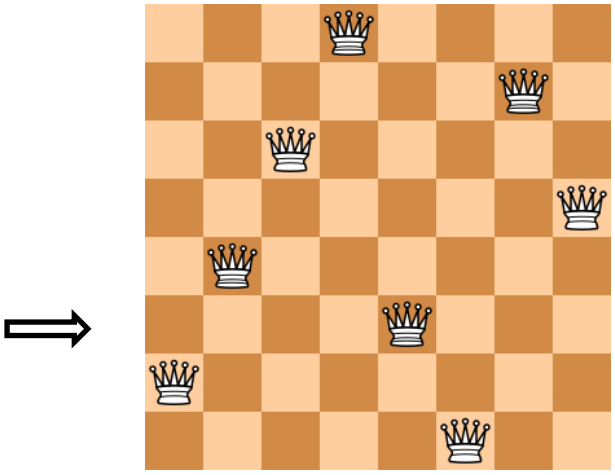

図 3.6.1 突然変異発生前

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。

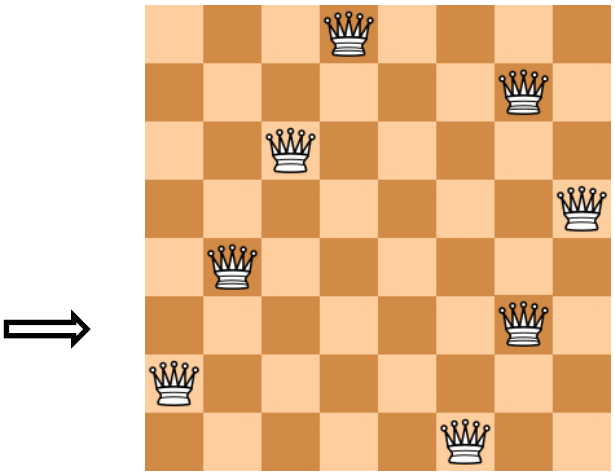

図 3.6.2 突然変異発生後

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

1 章で述べた通り、遺伝アルゴリズムでは、既存の個体間で交叉することにより新たな個体を生成する。しかし交叉だけでは、個体の親に依存する限られた範囲の子しか生成することができないため局所解に陥る危険がある。そこで多くの場合個体の生成時に突然変異を起こすことにより、個体群の全ての解が局所解に陥ってしまうことを防ぐ手法が用いられる。しかし単にランダムな突然変異を発生させるだけでは、進化阻害が多発し、解の探索効率が落ちてしまう。そのため、問題に応じて突然変異の方法を工夫する必要がある。

本研究では、NQueen 問題に対して、突然変異を用いることにより最適解を得る遺伝アルゴリズムを提案する。本研究で提案する遺伝アルゴリズムは複数の最適解を生成するため以下の 4 項目について改良を施した突然変異を生成する。

① 突然変異時における突然変異の内容変更
② 最適解発生時における突然変異
③ 競合度数における突然変異率
④ 同一遺伝子の重複発生時における突然変異

以降にその提案理由、および実装方法の具体的な説明を記述する。

4.1 突然変異時における突然変異の内容変更

NQueen 問題を解く GA では多くの場合において、ある駒の X 座標をランダムに変化させるという突然変異、すなわち突然変異の発生した遺伝子の遺伝子座情報が 0～N-1 の間で変化するという単純突然変異が用いられる。

この仕様では図 1 に示す例ように、斜めでのみ競合が発生している遺伝子の場合、突然変異の発生により遺伝座情報が元の値以外になると、必ず縦での競合が発生してしまう。このことが、突然変異発生のたびに競合数が元の数値より上昇してしまう(以降これを進化阻害と呼ぶ)可能性を高めるという結果につながる。つまり、突然変異が発生することにより進化阻害が起こり、解探索の性能が下がることが考えられる。そこで突然変異の内容を遺伝子座の状態変異から遺伝子内での状態交換に変更する。こうすることで、たとえ斜めのみで競合が発生している場合に突然変異が発生したとしても縦の競合が発生しなくなり、進化阻害を誘発しにくくなる。

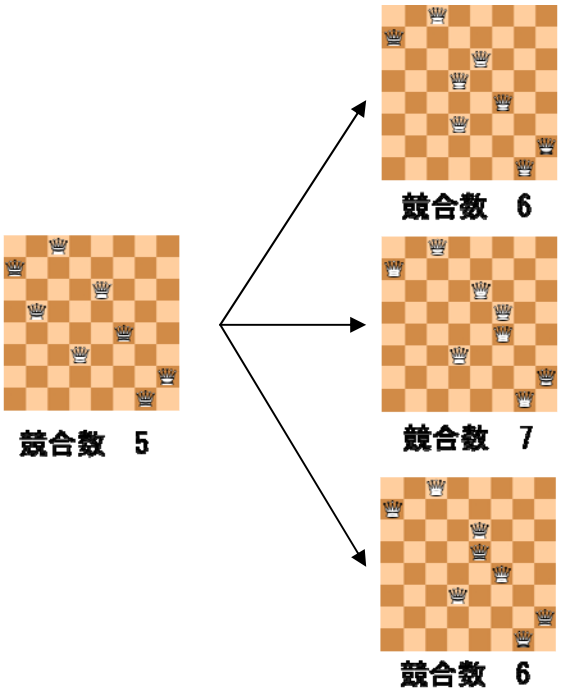

図 4.1.1　単純突然変異による突然変異の例

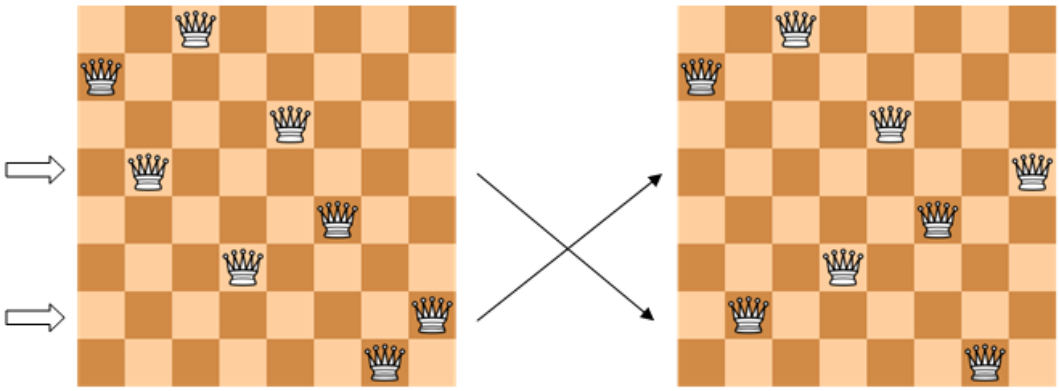

図 4.1.2 状態交換突然変異による突然変異の例

本研究で提案する突然変異は、状態交換用に遺伝子内の 2 つの遺伝情報を乱数により選択し、選ばれた遺伝情報同士を交換するというものである。図 2 に状態交換突然変異による突然変異の例を示す。

本研究で提案する状態交換突然変異アルゴリズムを以下に示す。

[状態交換突然変異アルゴリズム]

① 乱数 $r \in \mathbf{R}$ $(0 \leqq r < 1)$を発生させる。
② 突然変異率 p に対して $r \geqq p$ ならば停止、$r < p$ ならば③へ進む。
③ 乱数 $m, n \in \mathbf{N}$ $(1 \leqq m, n \leqq \mathrm{N},\ m \neq n)$を発生させる。
④ m 番目の遺伝座と n 番目の遺伝座の遺伝情報を交換する。

4.2 最適解発生時における突然変異

GA の対象とする問題に多峰性がある場合、すなわち対象とする問題が複数の最適解を持つ場合、いかに多くの最適解を発見できるかが問題である。NQueen 問題に対して、単純な GA を用いたプログラムを用いて検証してみたところ、ごくわずかな最適解の出現しか確認できなかった。そこで出現した最適解の数を確認してみると、同一の最適解が重複して生成されていることがわかった。つまり、最適解が生成され、突然変異が行われずに次の世代に移行すると、最適解の競合数は 0 であるからプログラムの仕様から必然的に生き残る可能性が増え、その結果集団中の全ての個体が一度発生した最適解に収束してしまい、新たな解が発生しなくなってしまう。このことから、本研究においては、交叉によって解となった遺伝子は強制的に突然変異を起こすようにした。

本研究で提案する最適解発生時における突然変異のアルゴリズムを以下に示す。

[最適解発生時における突然変異のアルゴリズム]

① 遺伝子 G の競合数 $C(G)$に対して $C(G)>0$ ならば停止、$C(G)=0$ ならば②へ進む。
② [状態交換突然変異アルゴリズム]を用いて遺伝子 G を突然変異させ、①へ戻る。

4.3 競合数における突然変異

単純突然変異では、全ての遺伝子において突然変異は等しい確率で発生する。この仕様にしたがうと、競合数 1 の遺伝子であっても、競合数 10 の遺伝子であっても等しく突然変異が発生し、最悪の場合、突然変異発生により競合数が増加してしまい、最適解への収束を阻害しかねない。このことから、競合数により、突然変異の発生コストは変わると考えられる。そこで、ある遺伝子に対する突然変異発生率を(基本突然変異発生率)＋(補正値)とし、補正値を遺伝子の競合数に応じて変化させる。競合数には許容競合数という閾値を設け、閾値を超えた競合数を持つ遺伝子に対してはより大きな補正をかける。ある個体 i に対する突然変異発生率 p_i は、基本突然発生率を p_{base}、個体 i の競合数 c_i 、許容競合数 c_{tol} を用いて以下の式(1)で表される。

$$p_i = \begin{cases} p_{base} - \left(\dfrac{c_i}{1+c_i} \right) \times \dfrac{1}{10} \ (if \quad c_i \le c_{tol}) \\ p_{base} + \left(\dfrac{c_i}{1+c_i} \right) \quad (if \quad c > c_{tol}) \end{cases} \quad \cdots(1)$$

式(1)より、突然変異発生率 p_i は最初緩やかに減少し、許容競合数 c_{tol} を超えた瞬間急激に浄化し、その後緩やかに上昇する。これにより、競合数の大きい遺伝子ほど突然変異し易くなる。また、p_i の増加は、競合数 c_i が許容競合数 c_{tol} 以下の場合は緩やかであるが、c_{tol} を越えると急激に増加する。これにより、許容競合数以下の競合数を持つ遺伝子に対する突然変異を抑制する一方、許容競合数を超える競合数を持つ遺伝子に対しては突然変異を誘発できる。例えば、基本突然変異発生率 p_{base} を 0.1、許容競合数 c_{tol} を 2 とし、競合数 c_i が 1 と 4 の遺伝子について考える。競合数 1 の場合は許容競合数以内であり上の式が適用され、p_i は 0.05 となり、競合数 4 の場合は許容競合数以外であり下の式が適用され、p_i は 0.9 となる。

4.4 同一遺伝子の重複発生時における突然変異

2.3 節で提案した許容競合数を設定することにより、競合数増加による進化阻害の発生率を抑えることができる。しかしながら、一度でも生き残りやすい競合数の小さい遺伝子が集団内に発生すると、選択や交叉で淘汰されず、また突然変異も発生しにくくなり、何世代にも渡り集団内に存在し続け、解探索の多様性が失われるという結果につながる。このことから、同一遺伝子が重複して発生すると、同一の部分解しか生成されず、多くの解探索につながらないことがわかる。つまり、解探索において、同一遺伝子は発生回数が増えるとその価値が低下していくと考えられる。そこで同一遺伝子が生存できる回数を決め、もしその発生回数を超えて同一遺伝子が発生した場合、たとえ競合数が許容競合数以内であっても、その許容範囲から除外され、突然変異が発生するようにした。

本研究で提案する同一遺伝子の重複発生時における突然変異のアルゴリズムを以下に示す。

[同一遺伝子の重複発生時における突然変異のアルゴリズム]

① 遺伝情報を確認する

② 発生したことのない遺伝子なら、遺伝情報を保存し、発生回数をカウントしはじめ、突然変異の判定へ移行する。もし、すでに発生したことのある遺伝子であれば、発生回数を確認する。

③ 発生回数が設定した発生回数よりも多ければ許容範囲から外して、突然変異の判定へ移行する。設定した発生回数未満であれば、カウントを増やして、突然変異へ移行する。

4.5 改良を施した NQueen 問題の解探索プログラム

付録に本研究で改良を行った突然変異のプログラムを記載する。以下に c++言語で作成したプログラムについて説明する。

mutation1.cpp

mutational.cpp は遺伝的アルゴリズムを用いて NQueen 問題を解くために最初に提案した、状態変異型の突然変異を発生させるプログラムであり、3.1 章の要綱にそった仕様となっている。mutaion1 メソッドは、遺伝子に状態変異の突然変異を起こすメソッドである。変数 q は main.cpp にある rnd メソッドを用いて 0～1 の生成した乱数を格納するためのものである。この値を突然変異発生確率と比較し、突然変異を起こす。以降開発した突然変異も発生方法はこの変数 q と突然変異発生確率の比較を行う。

mutation2.cpp

mutation2 は 4.1 章の要綱にそった仕様となっている。

mutation3.cpp

mutaition3 は 4.2 章の要綱にそった仕様となっている。ここでは競合数の保存されている変数 match を呼び出し、競合数 0 の場合、突然変異を強制的に発生させている。

mutation4.cpp

mutation4 は mutation3 で強制的に突然変異を発生させた場合、出現回数に応じて、交換する状態を増やすという仕様になっている。ここでは新たな重複解判定のため、vectorcout という vector 型の変数を用いている。この vector 型の変数に出現回数を保存している。

mutation5.cpp

mutation5 は 4.3 章の要綱にそった仕様となっている。ここでは競合数によって突然変異に補正をかけるため、新たな突然変異発生率を格納する変数 matpoint と、許容競合数かどうか判定するため、許容競合数が格納されている変数 pernumber を用いている。

mutation6.cpp

mutation6 は 4.4 章の要綱にそった仕様となっている。ここでは許容競合数以内における遺伝子の発生回数を保存するために、新たに mucount という変数を用いている。

本研究では、上記のプログラムを用いて N=(N の範囲)に対する NQueen 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。

表 5.1 プログラムを実行した条件

N	集団数 M	終了世代	選択方法	交叉方法	基本突然変異率	許容競合度	試行回数
8	100	1000	※エリート選択	一様交叉	0.1	2	1000

※エリート選択は、共同研究者作成のものを使用する。

5　結果・考察

5.1 結果

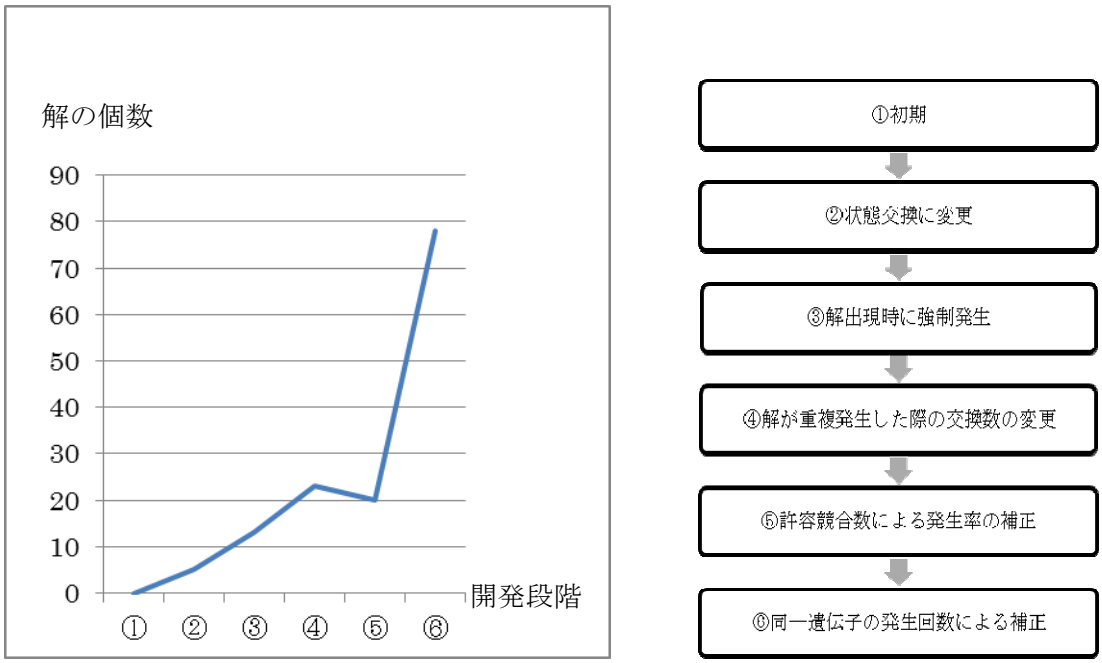

図 3　開発過程における解探索能力の推移

表 5.1 に本研究でプログラムを実行した条件を示す。表 5.1 に示す条件で、突然変異以外の条件を全て揃えてプログラム mutation1.cpp～mutation6.cpp を実行したときの解探索能力の推移を図 3 に示す。開発が進むにつれておおよそ解探索能力が向上していることがわかる。しかしながら④から⑤へと開発が進んだ際に、解の探索能力が低下している。その原因については後述する。次に各世代における解の平均探索数を図 4 に示す。Y 軸は解の個数、X 軸は世代数を表している。ここでは最終的な改良を施した GA と初期に作成した単純 GA の比較を行っている。図 4 より改良を施した GA は各世代においても安定して解を生成できていることがわかる

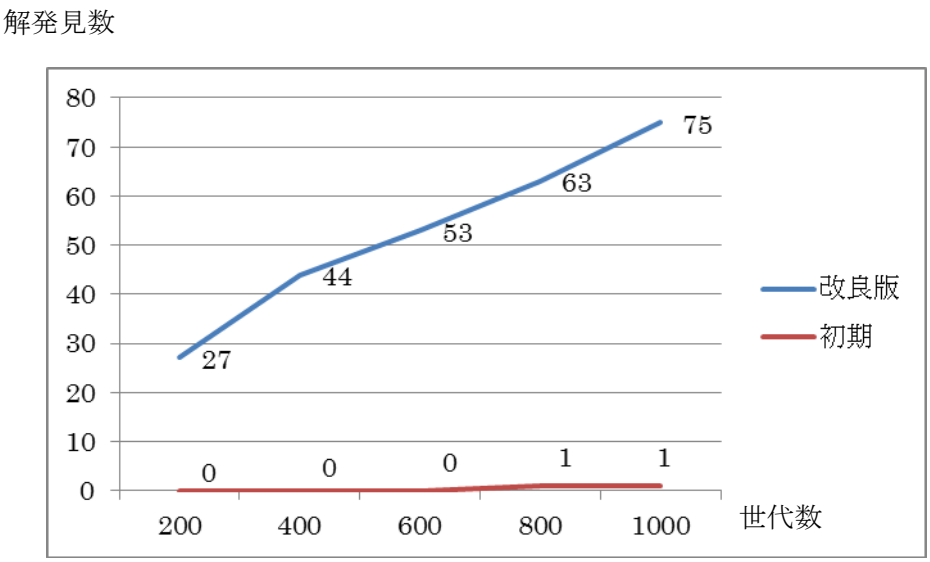

図 4　各世代における解の平均出現数

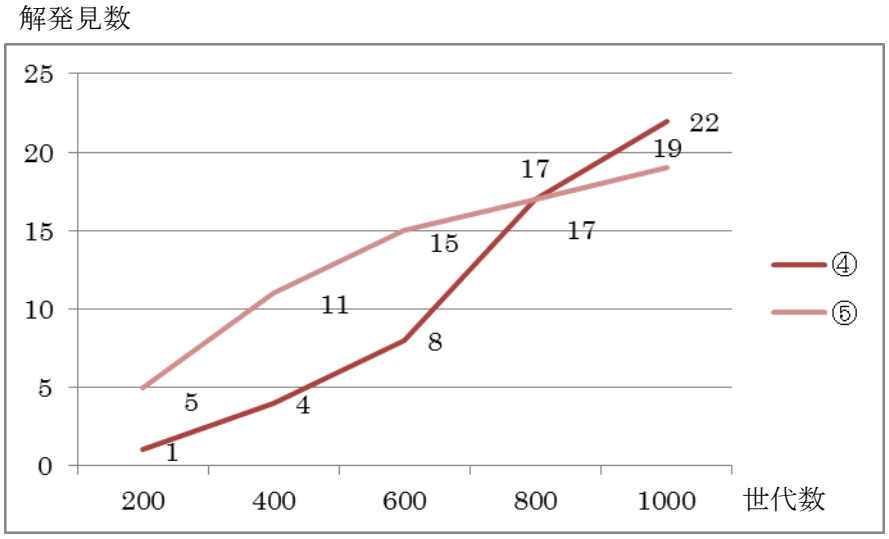

図 5 ④と⑤における解の平均出現

また、図 3 をみると、④から⑤へ開発が推移すると解の探索能力が減少している。そこで④と⑤の解の出現状況を確認するため図 5 に④と⑤の世代別による解の出現状況を示す。ここでは④は許容競合数の仕様を含まず、⑤は重複解による交換数の変更の仕様を含まない。またそれ以外は N を 8 とし、選択や交叉を含め同一条件としている。図 5 から④では早い段階で解が探索され、⑤では終盤になると解が探索されていることがわかる。④は重複解の発生数が多くなる中盤から終盤にかけて、④の仕様である重複解に応じて遺伝情報の交換数が増加することが作用し、新たな遺伝子が多数生まれたことによるものではないかと考えられる。⑤は許容競合数によって競合数の少ない遺伝子が生き残りやすくなり、収束がはやまったためではないかと考えられる。なお開発においては⑤は④の仕様を含んでいる。しかしながら図 3 と同じように④の方が⑤より解探索能力がすぐれているという結果となった。その理由としては次のことが考えられる。⑤では序盤で大量に解が生成され、中盤から終盤にかけて新たな遺伝子が多数うまれているはずだが、新たな遺伝子の競合数が設定した許容競合数より大きく、選択で淘汰されてしまったためではないかと考えられる。また⑤の仕様で競合数が許容範囲内であれば、突然変異が起こりにくくなってしまうことから、競合数が許容範囲内である重複解の部分解が、最適解にならず生き残り続けてしまい、新しい解の発生を抑制し、結果として終盤での解探索が向上しなかったのではないだろうか。

また集団数を 100、世代数 1000、選択や交叉はもっとも性能がよかったものを使用し、1000 回試行するとしたうえで、本研究で作成したプログラムを用いて問題サイズによる解探索をした実行結果を表 5.1 に示す。

表 5.1 各問題のサイズにおける解発見数および探索時間

N	解発見数	全解数	解発見数/全解数	探索時間
4	2±0	2	100%	3882±11ms
5	10±0	10	100%	4163±12ms
6	2±1	4	50%	3054±12ms
7	37±3	40	92.5%	3753±201ms
8	75±10	92	81.5%	3588±205ms
9	104±14	352	29.5%	3855±209ms
10	78±10	724	10.7%	3503±206ms
11	89±12	2680	3.3%	3504±181ms
12	117±20	14200	0.8%	3652±177ms
13	102±24	73712	0.1%	3745±144ms

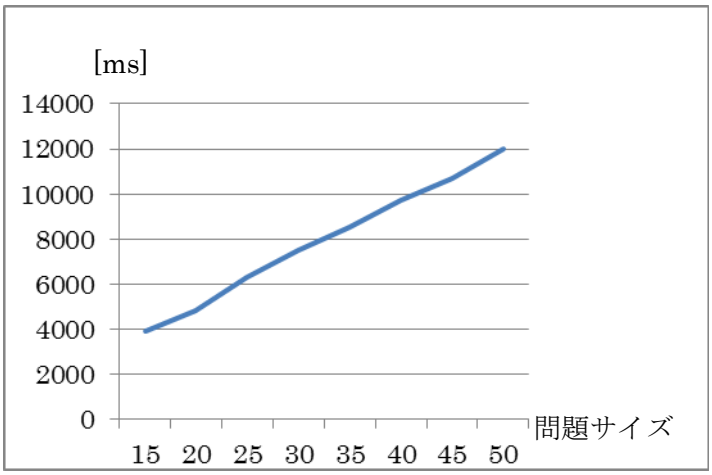

図 7 問題サイズにおける実行時間

次に、解が 1 つでもいいので生成される条件ことを条件に、問題サイズにおける実行時間を図 7 に示す。NQueen 問題の既知の結果と図 6 照らし合わせると問題サイズ 7 までであれば、最初に提示した条件で全解探索が可能である。また、図 7 の結果より、問題サイズが増えると扱うデータの量が膨大になり、実行時間が上昇しているのがわかる。このことから問題サイズと時間は比例関係にあることがわかる。同一条件で解が 1 でもいいので生成されることを条件に、どこまで大きなサイズの問題が扱えるか検証したところ実行時間 200000ms で問題サイズ 100 での解生成まで確認できた。

5.2 考察

上記 3.1 節の結果から、最適解が多峰性を持つ問題を遺伝的アルゴリズムにより解探索する際、同一遺伝子の価値に注目すべきではないかと考えた。本研究結果では解の遺伝子である場合、そうでない場合、どちらにおいても発生回数が増加することにより解探索を抑制することがわかった。しかしながら、同一遺伝子を発生させないために突然変異を乱発してしまってはランダムに解を探索するのと変わりなくなってしまう。本研究においては、突然変異の発生を 1 割未満に抑えながら同一遺伝子の発生を抑えることができたと考えられる。また問題サイズ 100 でも解の生成を確認できた。 また 1.3 章で紹介したように、既存の結果に比べると、実行時間に対して求められる解の数が非常にすくない。しかしながら、既存の研究は解の数は保存しているがその解がどういった配置をしているかはわからない。つまり、解の配置情報を保存している本プログラムとは単純に速度のみでは比較できないと考える。既存の研究で用いられているプログラムに解の配置情報を保存する機構を実装してはじめて実行時間の比較を行えるのではないだろうか。

6　結論・今後の課題

6.1 結論

今回の研究では単純遺伝アルゴリズムを改良することによりＮクイーン問題の解探索が行えることがわかり、また具体的な遺伝アルゴリズムの改良方法を提示できた。しかしながら、当初の目標としていた初期集団 100、世代数 1000 においての N＝8 の全解探索にわずかながら届かなかった。

6.2 今後の課題

今回作成したプログラムでは同一解の判定のため、発生した遺伝子をすべて保存している。そのため問題範囲が大きくなると保存すべき遺伝子の量も爆発てきに増えると考えられる。今後はいかに遺伝情報を保存せずに同一遺伝子かを判定すべきかが問題となってくる。また、結果として、まだ同一解の生成が多数行われていた。同一解の生成を回避するために、似たような進化過程をたどっているかどうかを判定すし、似たような進化過程をたどりそうなら、ある世代の遺伝子集団にもどり再度進化させていくなどを考えるべきである。

謝辞

卒業論文の作成にあたり、石水隆助教には数多くの御指導や御助言を頂きましたお礼をこの場をかりて申し上げます。また、共同研究者一同にはいろいろとお世話になり、感謝を申し上げます。

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

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

[5] 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.

[6] 同志社大学 知的イシステムデザイン研究室 ゼミ資料, 1999, http://mikilab.doshisha.ac.jp/dia/seminar/1999/optim/optim01.pdf

[7] 吉瀬謙二, N-Queens Homepage in Japanese, 電気通信大学, 2004, http://www.arch.cs.titech.ac.jp/~kise/nq/index.htm

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

[9] 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.cpp

```
#include <iostream>
#include <ctime>
#include <cstdlib>
#include <windows.h>
#include<vector>
#include<string>
#include<sstream>
#include "set.h"
using namespace std;
void main(){

	int a[N][NBIT]; // 初期集団
	int	match[N]; // 競合数を保存ための配列

	vector<int> vectorcount; // 解の出現回数保管用
	vector<int> mucount; //突然変異において同一のコマ配置が何回発生したか調べるためのリスト
	vector<string> muvector; //突然変異において重複するコマ配置が発生したか調べるためのリスト
	vector<string> vector; //出現解保存用リスト型配列

	LARGE_INTEGER freq,time_start,time_end;//周波数、開始時間、終了時間

	void init(int [][NBIT]);
	void print0(std::vector<std::string>);
	void func(int [][NBIT],int [N],std::vector<std::string>&,std::vector<int>&);
	void mutation1(int [][NBIT]);//初期作成の突然変異
	void mutation2(int [][NBIT],int [N]);//状態交換の突然変異
	void mutation3(int [][NBIT],int [N]);//解発生時に強制的に突然変異発生
	void mutation4(int [][NBIT],int [N],std::vector<std::string>&,std::vector<int>&);//重複解発生時に突然変異による交換数の補正
	void mutation5(int [][NBIT],int [N],std::vector<std::string>&,std::vector<int>&);//許容競合数による突然変異補正
	void mutation6(int [][NBIT],int [N],std::vector<std::string>&,std::vector<int>&,std::vector<std::string>&,std::vector<int>&);//重複発生回数による突然変異の変更
	void elite(int [][NBIT],int [N]);
	void select(int [][NBIT],int [N]);
	void cross(int [][NBIT]);//一様交叉
	srand((unsigned)time(NULL));
	/*時間計測用*/
	QueryPerformanceFrequency(&freq);
	QueryPerformanceCounter(&time_start);//時間計測開始
	init(a);
	for(int i=0;i<MAX;i++){
		elite(a,match);
		select(a,match);
		cross(a);
		func(a,match,vector,vectorcount);
		//mutation1(a);
		//mutation2(a,match);
		//mutation3(a,match);
		//mutation4(a,match,vector,vectorcount);
		//mutation5(a,match,vector,vectorcount);
```

```
                    mutation6(a,match,vector,vectorcount,muvector,mucount);
                    func(a,match,vector,vectorcount);
          }
          QueryPerformanceCounter(&time_end);//計測時間停止
          print0(vector);
          printf("処理時間:%d[ms]¥n",(time_end.QuadPart-time_start.QuadPart)*1000 / freq.QuadPart);
}
/**
*@param x 乱数発生の元となる変数
*@return  0～1の間の値を返す
*/
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);
}
```

init.cpp

```
#include <iostream>
#include "set.h"
/**
*@param a[][NBIT] 外側の配列が集団数を格納し、内側の配列が駒の配置情報を格納している
*/
void init(int a[][NBIT]){
          //ここで初期集団を生成している
          for(int x=0;x<N;x++){
                    for(int y=0;y<NBIT;y++){
                              a[x][y]=y;
                    }
          }
}
```

func.cpp

```
#include <iostream>
#include<vector>
#include<string>
#include<sstream>
#include "set.h"
using namespace std;
/**
*@param a[][NBIT] 外側の配列が集団数を格納し、内側の配列が駒の配置情報を格納している
*@param match[N] 遺伝子の競合数が格納されている
*@param vector 解となった配置を格納している
*@param vectorcount 解の出現回数が格納されている
*/
void func(int a[][NBIT],int match[N],std::vector<std::string>& vector,std::vector<int>& vectorcount){
          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++){
			i=a[x][y];// 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+=fitness;
			}
		}
		//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++){
				l<<a[x][i];
			}
			vector.push_back(l.str());
			vectorcount.push_back(1);

			//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();
					vectorcount.pop_back();
					vectorcount[j] += 1;
					break;
				}
```

```
                        }
                }
                sum = 0;//競合数の初期化
        }
}
```

elite.cpp

```
#include <iostream>
#include "set.h"
/**
*@param a[][NBIT] 外側の配列が集団数を格納し、内側の配列が駒の配置情報を格納している
*@param match[N] 遺伝子の競合数が格納されている
*/
void elite(int a[][NBIT],int match[N]){
        int cha[1][NBIT]; //交換用駒の配置保存用
        int chm=0; //交換用競合数保存用
        for(int i=0;i<N;i++){
                for(int j=0;j<N;j++){
                        //競合数の低いものから順番に並び替えを行っている
                        if(match[i]<match[j]){
                                //交換用遺伝子の保存
                                for(int p=0;p<NBIT;p++){
                                        cha[0][p]=a[i][p];
                                }
                                chm = match[i];
                                //遺伝子の交換
                                for(int q=0;q<NBIT;q++){
                                        a[i][q]=a[j][q];
                                }
                                match[i]= match[j];
                                //遺伝子の交換
                                for(int r=0;r<NBIT;r++){
                                        a[j][r]=cha[0][r];
                                }
                                match[j]= chm;
                        }
                }
        }
}
```

select.cpp

```
#include<iostream>
#include"set.h"
#include <vector>
#include<string>
#include<sstream>
/**
*@param a[][NBIT] 外側の配列が集団数を格納し、内側の配列が駒の配置情報を格納している
*@param match[N] 遺伝子の競合数が格納されている
*/
void select(int a[][NBIT],int match[N]){

        int newa[N][NBIT];

        std::vector<int> evector1;//エリート遺伝子の競合数保存用
        std::vector<std::string> evector2;//エリート遺伝子保存用

        //重複せず一定の競合度のみ保存
        for(int i=0;i<N;i++){
```

```
		//もし競合数が許容競合数以下なら
		if(match[i] <= permatch){

			std::ostringstream l;
			for(int j=0;j<NBIT;j++){
				l << a[i][j];
			}

			evector2.push_back(l.str());
			evector1.push_back(i);
			int size = evector2.size()-1;

			//重複がないかの確認
			for(int x=0;x<size;x++){
				if(evector2[x]==evector2[size]){
					evector2.pop_back();
					evector1.pop_back();
				}
				break;
			}
		}
	}

	int maxsize =0;
	//エリートの数が1以上なら
	if(evector1.size()>0){

		//保存したエリートが選ばれるエリートの数より少なかった場合
		//保存したエリートのだけが選ばれる
		if(evector1.size() < elitenumber){
			maxsize = evector1.size();
		}

		//保存したエリートが選ばれるエリートの数より多かった場合
		//保存したエリートは上から順に選ばれる
		else{
			maxsize = elitenumber;
		}

		int w1=0;//エリート数カウント用変数

		//エリートの選択を行っている
		for(int i=0;i<N;i++){
			if(w1 <= maxsize){
				for(int j=0;j<NBIT;j++){
					newa[i][j] = a[evector1[w1]][j];
				}
				if((w1+1)<evector1.size()){
					w1 +=1;
				}
			}
			else{
				for(int k=0;k<NBIT;k++){
					newa[i][k] = a[evector1[w1]][k];
				}
				w1 = 0;
			}
```

```
                }

        for(int i=0;i<N;i++){
                for(int j=0;j<NBIT;j++){
                        a[i][j] = newa[i][j];
                }
        }
        evector1.clear();
        evector2.clear();
        }
        //もしもエリートとなる遺伝子がない場合、上からエリート数だけ選択する
        else{

                int w2=0;//エリート数カウント用変数

                //エリートの選択を行っている
                for(int i=0;i<N;i++){
                        if(w2 <= maxsize){
                                for(int j=0;j<NBIT;j++){
                                        newa[i][j] = a[w2][j];
                                }
                                w2 +=1;
                        }
                        else{
                                for(int k=0;k<NBIT;k++){
                                        newa[i][k] = a[w2][k];
                                }
                                w2 = 0;
                        }
                }

                for(int i=0;i<N;i++){
                        for(int j=0;j<NBIT;j++){
                                a[i][j] = newa[i][j];
                        }
                }
                evector1.clear();
                evector2.clear();
        }

}
```

cross.cpp

```
#include<iostream>
#include"set.h"
/**
*@param a[][NBIT] 外側の配列が集団数を格納し、内側の配列が駒の配置情報を格納している
*/
void cross(int a[][NBIT]){


                int w[1][NBIT];//ランダム保管用配列
                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++){
                                w[0][j] = a[i][j];
                        }
                        int r=rand()%(N);
                        for(int s=0;s<NBIT;s++){
                                a[i][s] = a[r][s];
                        }
                        for(int t=0;t<NBIT;t++){
                                a[r][t] = w[0][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

                                if(crate<rnd0){
                                        //各遺伝子ごとに交叉判定を行う
                                        for(int p=0;p<NBIT;p++){
                                                int r = rand()%2;
                                                if(r==1){
                                                        int newa = a[x][p];
                                                        a[x][p] = a[x+1][p];
                                                        a[x+1][p] = newa;
                                                }
                                                else{
                                                }
                                        }
                                        x +=1;
                                }
                                else{
                                        x +=1;
                                }
                        }
}
```

mutation1.cpp

```
#include<iostream>
#include"set.h"
#include<cstdlib>
#include<ctime>
using namespace std;
/**
*@param a[][NBIT] 外側の配列が集団数を格納し、内側の配列が駒の配置情報を格納している
*/
void mutation1(int a[][NBIT]){

                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

                        //突然変異が起こった場合
                        if(p < q){

                                int y =rand()%(NBIT); //突然変異の起こる位置
                                int z =rand()%(NBIT); //コマの位置情報の置き換え

                                a[i][y] = z;       //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
                        }

                }

}
```

mutation2.cpp

```
#include<iostream>
#include"set.h"
#include<cstdlib> //rand()使用のために必要
#include<ctime> //rand()の初期化を行うためのtimeを呼び出す時に必要
using namespace std;
/**
*@param a[][NBIT] 外側の配列が集団数を格納し、内側の配列が駒の配置情報を格納している
*@param match[N] 遺伝子の競合数が格納されている
*/
void mutation2(int a[][NBIT],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

                        //突然変異が起こった場合
                        if(p < q){
                                int y =rand()%(NBIT); //突然変異の発生により、交換する駒の位置情
報
                                int w = a[i][y]; //交換する遺伝情報の保存
                                int z =rand()%(NBIT); //交換される駒の位置情報を乱数により決定

                                a[i][y] = a[i][z]; //駒の位置情報を交換する
                                a[i][z] = w;
                        }
                }
}
```

mutation3.cpp

```
#include<iostream>
#include"set.h"
#include<cstdlib> //rand()使用のために必要
#include<ctime> //rand()の初期化を行うためのtimeを呼び出す時に必要
using namespace std;
/**
```

```
*@param a[][NBIT] 外側の配列が集団数を格納し、内側の配列が駒の配置情報を格納している
*@param match[N] 遺伝子の競合数が格納されている
*/
void mutation3(int a[][NBIT],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 g = a[i][e];
                                        int j =rand()%(NBIT); //コマの位置情報の置き換え

                                        a[i][e] = a[i][j]; //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[i][j] = 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 w = a[i][y];
                                                  int z =rand()%(NBIT); //コマの位置情報の置き換え

                                                  a[i][y] = a[i][z]; //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[i][z] = w;
                                        }
                              }
                    }
}
```

mutation4.cpp

```
#include<iostream>
#include"set.h"
#include<cstdlib> //rand()使用のために必要
#include<ctime> //rand()の初期化を行うためのtimeを呼び出す時に必要
#include<vector>
#include<string>
#include<sstream>
using namespace std;
/**
*@param a[][NBIT] 外側の配列が集団数を格納し、内側の配列が駒の配置情報を格納している
*@param match[N] 遺伝子の競合数が格納されている
*@param vector 解となった配置を格納している
*@param vectorcount 解の出現回数が格納されている
*/
void mutation4(int a[][NBIT],int match [N],std::vector<std::string>& vector,std::vector<int>&
```

```
vectorcount){

                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);
                        std::ostringstream l;

                        //競合数0の場合
                        if(match[i]==0){
                                //既出解かどうかの判定をするためコマの配置所得
                                for(int f=0;f<NBIT;f++){
                                        l<<a[i][f];
                                }
                                int size = vector.size();
                                int hitpoint=1;
                                for(int s=0;s<size;s++){
                                        if(vector[s]==l.str()){
                                                hitpoint = vectorcount[s];
                                                if(hitpoint>NBIT){
                                                        hitpoint = NBIT;
                                                }
                                                break;
                                        }

                                }

                                for(int o=0;o<hitpoint;o++){
                                        int y =rand()%(NBIT); //突然変異の起こる位置
                                        int w = a[i][y];
                                        int z =rand()%(NBIT); //コマの位置情報の置き換え

                                        a[i][y] = a[i][z]; //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[i][z] = w;
                                        //sum =1;//強制的に突然変異させる場合はカウントから除
外する
                                }
                        }
                        else{
                                if(p < q){
                                        int y =rand()%(NBIT); //突然変異の起こる位置
                                        int w = a[i][y];
                                        int z =rand()%(NBIT); //コマの位置情報の置き換え

                                        a[i][y] = a[i][z]; //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[i][z] = w;
                                }
                        }
                }
}
```

mutation5.cpp

```
#include<iostream>
#include"set.h"
#include<cstdlib> //rand()使用のために必要
#include<ctime> //rand()の初期化を行うためのtimeを呼び出す時に必要
#include<vector>
#include<string>
#include<sstream>
using namespace std;
/**
*@param a[][NBIT] 外側の配列が集団数を格納し、内側の配列が駒の配置情報を格納している
*@param match[N] 遺伝子の競合数が格納されている
*@param vector 解となった配置を格納している
*@param vectorcount 解の出現回数が格納されている
*/
void mutation5(int a[][NBIT],int match [N],std::vector<std::string>& vector,std::vector<int>&
vectorcount){

                int sum=0;
                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);
                        double matpoint =0.0;
                        std::ostringstream l;

                        //競合数0の場合
                        if(match[i]==0){

                                //既出解かどうかの判定をするためコマの配置所得
                                for(int f=0;f<NBIT;f++){
                                        l<<a[i][f];
                                }
                                int size = vector.size();
                                int hitpoint=1;
                                //もし基準値以上出現していれば最大NBIT回混ぜるように設定する
                                for(int s=0;s<size;s++){
                                        if(vector[s]==l.str()){
                                                hitpoint = vectorcount[s];
                                                if(hitpoint>NBIT){
                                                        hitpoint = NBIT;
                                                }
                                                break;
                                        }

                                }
                                //重複回数だけ遺伝子内で遺伝情報が交換される
                                for(int o=0;o<hitpoint;o++){
                                        int y =rand()%(NBIT); //突然変異の起こる位置
                                        int w = a[i][y];
                                        int z =rand()%(NBIT); //コマの位置情報の置き換え

                                        a[i][y] = a[i][z]; //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[i][z] = w;
                                }
                        }
                        else{
                                //突然変異を起こす許容範囲によって発生率を決定する
                                if(match[i]<=NBIT/4+pernumber){
                                        matpoint = mrate +
(1.0-(1.0/(1.0+(double)match[i])))/10.0;
                                }
                                else{
                                        matpoint = mrate - 1.0-(1.0/(1.0+(double)match[i]));
                                }

                                if(matpoint < q){
                                        int y =rand()%(NBIT); //突然変異の起こる位置
                                        int w = a[i][y];
                                        int z =rand()%(NBIT); //コマの位置情報の置き換え

                                        a[i][y] = a[i][z]; //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[i][z] = w;
                                }
                        }

                }
}
```

mutation6.cpp

```
#include<iostream>
#include"set.h"
#include<cstdlib> //rand()使用のために必要
#include<ctime> //rand()の初期化を行うためのtimeを呼び出す時に必要
#include<vector>
#include<string>
#include<sstream>
using namespace std;
/**
*@param a[][NBIT] 外側の配列が集団数を格納し、内側の配列が駒の配置情報を格納している
*@param match[N] 遺伝子の競合数が格納されている
*@param vector 解となった配置を格納している
*@param vectorcount 解の出現回数が格納されている
*@param muvector 競合数が許容範囲内となった駒の配置情報を格納している
*@param mucount 競合数が許容範囲内となった駒の配置の出現回数を格納している
*/
void mutation6(int a[][NBIT],int match [N],std::vector<std::string>& vector,std::vector<int>&
vectorcount,std::vector<std::string>& muvector,std::vector<int>& mucount){

                int sum=0;

                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);
                        double matpoint =0.0;
                        std::ostringstream l;

                        //競合数0の場合
                        if(match[i]==0){
                                //既出解かどうかの判定をするためコマの配置所得
                                for(int f=0;f<NBIT;f++){
                                        l<<a[i][f];
                                }
                                int size = vector.size();
                                int hitpoint=1;
                                for(int s=0;s<size;s++){
                                        if(vector[s]==l.str()){
                                                hitpoint = vectorcount[s];
                                                if(hitpoint>NBIT){
                                                        hitpoint = NBIT;
                                                }
                                                break;
                                        }

                                }

                                for(int o=0;o<hitpoint;o++){
                                        int y =rand()%(NBIT); //突然変異の起こる位置
                                        int w = a[i][y];
                                        int z =rand()%(NBIT); //コマの位置情報の置き換え

                                        a[i][y] = a[i][z]; //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[i][z] = w;
                                }
                        }
                        else{
                                //突然変異を起こす許容範囲によって発生率を決定する

                                //許容範囲以下の場合
                                if(match[i]<NBIT/4+pernumber){

                                        std::ostringstream m;
                                        for(int e=0;e<NBIT;e++){
                                                m << a[i][e];
                                        }
                                        muvector.push_back(l.str());
                                        mucount.push_back(1);

                                        //発生回数の確認。発生回数はmucountに保存されている
                                        int u = muvector.size();
                                        int mu =0;
                                        for(int j=0;j<u-1;j++){
                                                if(muvector[j]==muvector[u-1]){
                                                        muvector.pop_back();
                                                        mucount.pop_back();
                                                        mu = mucount[j];
                                                        mucount[j] += 1;
                                                        break;
                                                }
```

```
                                        }

                                        //発生回数が上回っている場合
                                        if(mu>permu){
                                                matpoint = mrate -
1.0-(1.0/(1.0+(double)match[i]));
                                        }
                                        //発生回数が下回っている場合
                                        else{
                                                matpoint = mrate +
(1.0-(1.0/(1.0+(double)match[i])))/10.0;
                                        }

                                }

                                //許容範囲以上の場合
                                else{
                                        matpoint = mrate - 1.0-(1.0/(1.0+(double)match[i]));
                                }

                                //突然変異の発生判定
                                if(matpoint < q){
                                        int y =rand()%(NBIT); //突然変異の起こる位置
                                        int w = a[i][y];
                                        int z =rand()%(NBIT); //コマの位置情報の置き換え

                                        a[i][y] = a[i][z]; //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[i][z] = w;
                                }
                        }
                }
}
```

print0.cpp

```
#include<iostream>
#include"set.h"
#include<vector>
#include<string>
using namespace std;
/**
*@param vector 解となった配置を格納している
*/
void print0(std::vector<std::string> vector){
        int w = vector.size();
        for(int o=0; o<w;o++){
                cout <<"解:" <<o+1 <<"番目は" <<vector[o] <<endl;
        }

}
```

set.h

```
#define N 10 //初期集団の数
#define NBIT 8//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 fitness 5 //縦における競合度の大きさ
```

```
#define fitpoint 5 //競合度保存用の大きさ

#define pernumber 2 //突然変異発生率の許容競合数

#define tournumber 10 //トーナメントを実施する数
#define elitenumber 5 //選択するエリートの数

#define permatch 2 //エリートとする許容競合数
#define permu 15 //突然変異における許容重複回数

#define MAX 1000 //何世代まで求めるか
```