卒業研究報告書

題目

遺伝アルゴリズムによるNQueen解法

～遺伝補修飾を用いた解探索の性能評価～

指導教員

石水 隆　助教

報告者

08-1-037-0050

馬場 亮輔

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、最適解探索の方法として注目されているものに遺伝的アルゴリズムがある。遺伝的アルゴリズムは進化論をもとに考案されたアルゴリズムであり、定式化された式から最適解を探索するのではなく、環境変数を変化させていくことにより最適解を探索する。遺伝アルゴリズムは定式化が難しい問題に対しても解探索がおこなえるという利点がある。しかしながら問題によっては遺伝オペレーティングが複雑化することがある。このことにより、計算能力の低下や、解探索能力があまり向上しないのではないかと考えられる。そこで本研究では問題の性質に応じた新たな遺伝オペレーティングを導入することによりアルゴリズムが複雑化するのをさけるために、遺伝オペレーティング自体は基本的なもののみを用い、遺伝子自身に特殊な性質を持たせる遺伝補修飾を導入することによって、解探索能力を向上させる方法を提案する。

本研究では、最適化問題として NQueen 問題を用いる。NQueen 問題は複数の最適解を持つ問題であり、従来の遺伝アルゴリズムでは全ての解を得るのは難しい。本研究では、遺伝補修飾を導入することにより NQueen 問題の全最適解を得る手法を提案する。

目次

1　序論

1.1 本研究の背景

近年、最適解探索の方法として注目されているものに遺伝的アルゴリズム[1] [1]がある。遺伝的アルゴリズムは進化論をもとに考案されたアルゴリズムであり、定式化された式から最適解を探索するのではなく、環境変数を変化させていくことにより最適解を探索する。遺伝アルゴリズムは定式化が難しい問題に対しても解探索がおこなえるという利点がある。しかしながら問題によっては複雑なパラメタ設定や遺伝オペレーティングの複雑化という解決しなくてはならない問題点が発生する。遺伝アルゴリズムは最適解探索問題を解くアルゴリズムとして、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、提唱した考え方である。遺伝アルゴリズム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。 遺伝アルゴリズムを用いることにより解け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。

1.2 本研究の目的

1.1 節で述べたように、遺伝アルゴリズムを用いる場合、対象の問題によっては、その問題の性質に応じた新たな遺伝オペレーティングを導入する必要がある。例えば、新たな遺伝オペレーティングとして転座[4]を導入したとする。転座とは、遺伝子の一部分を同じ遺伝子内の他の部分に置き換える操作である。操作自体は単純で、交叉や突然変異以外の方法で遺伝子の種類に幅を持たせることができるが、発生条件の決定や、どの操作の後で起こすべきかを問題によって検討しなくてはならない。またさらに新たな遺伝オペレーティングを追加するとさらに多くの条件を考えなくてはならない。このように遺伝オペレーティングを加えることによりアルゴリズムが複雑化するおそれがある。そこで本研究では、遺伝オペレーティング自体は基本的なもののみを用い、遺伝子自身に特殊な性質を持たせる遺伝補修飾を用いることにより解探索能力を向上させるよう遺伝アルゴリズムに改良を施す。

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[5]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。

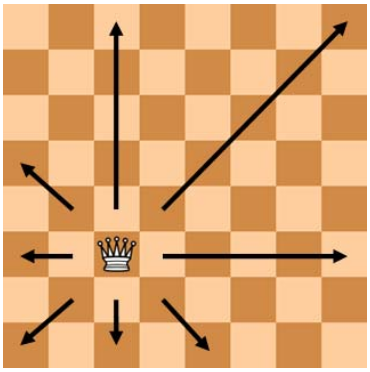

図 1.3.2　クイーンの利き筋

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 1.3.2 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。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[8]までの解探索を行っている。表 1.3.3.1 に[7]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	[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[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

本報告書の構成は以下の通りである。まず 2 章では遺伝アルゴリズムについて説明する。次に 3 章では本研究で提案する遺伝補修飾について述べ、4 章では提案した遺伝補修飾の実装方法について述べる。

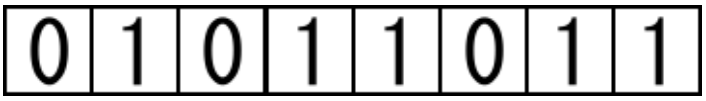

図 2.1 GTYPE の例

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[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

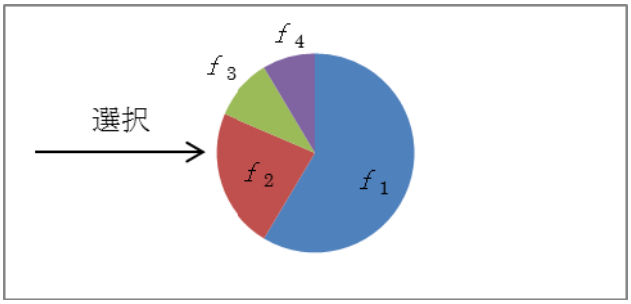

図 2.2　ルーレット法

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≦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。

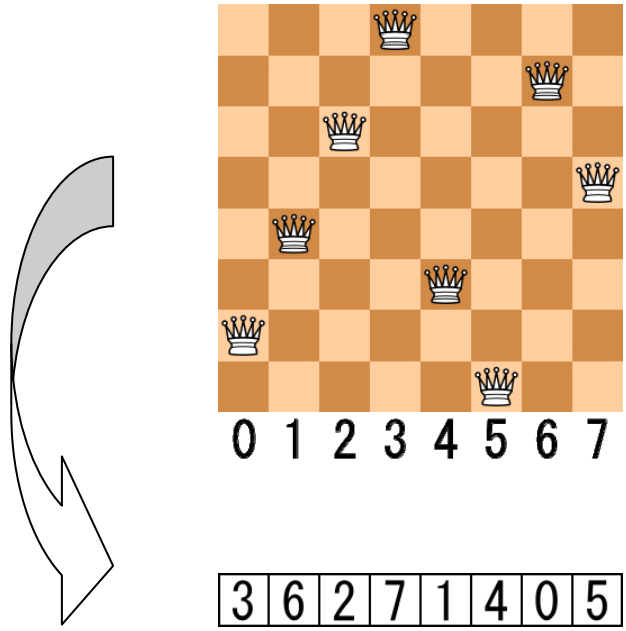

図 4.1 遺伝子コーディング例

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≦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。

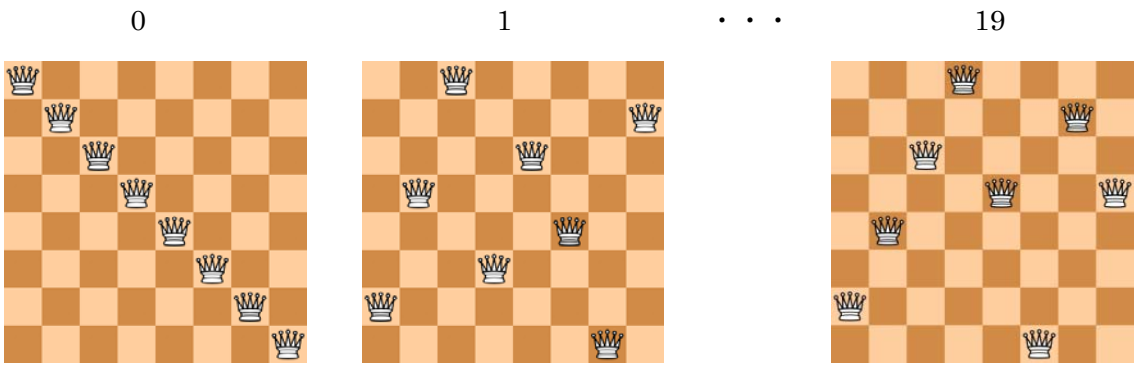

図 4.2　初期集団の状態

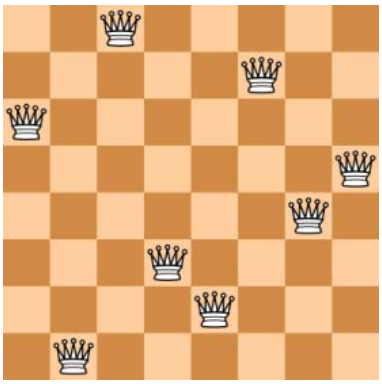

図 4.3　駒の配置状態の例

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。

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

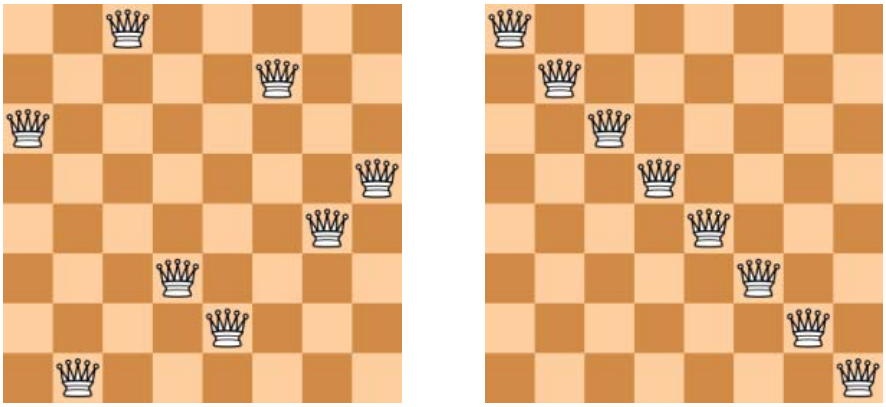

図 4.5.1　交叉前

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

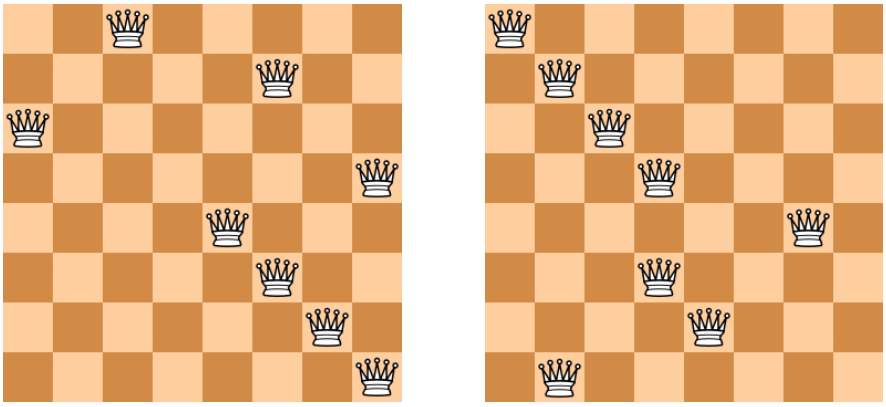

図 4.5.2　交叉後

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。

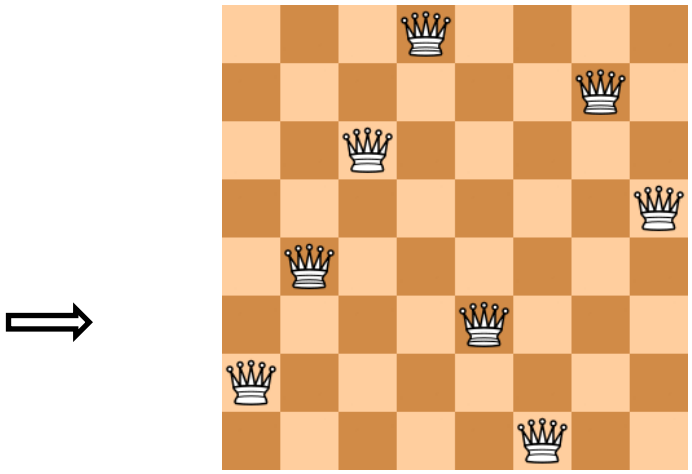

図 3.6.1 突然変異発生前

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。

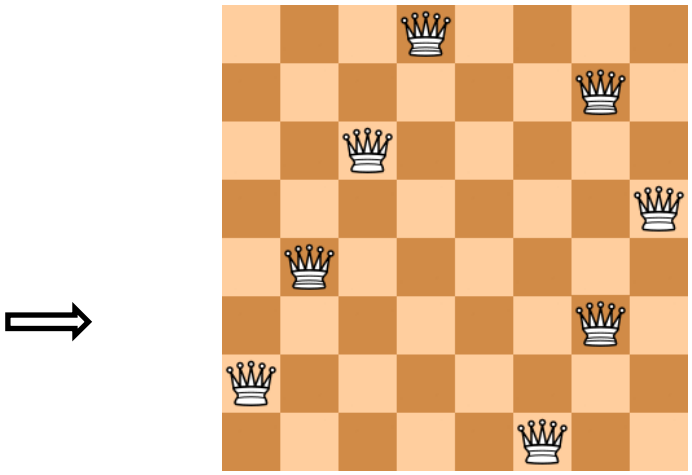

図 3.6.2 突然変異発生後

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

本研究では遺伝アルゴリズムに対し、新たな遺伝オペレーティングを加えることなく解の探索能力の向上を得るために、遺伝子にある種の性質を加える(以下、遺伝補修飾と呼ぶ)手法を提案する。

4.1 遺伝補修飾とは

通常、遺伝アルゴリズム上で扱われる遺伝情報は、バイナリ表現においてはじめてその性質を示すが、遺伝情報そのものには環境に対しての性質はない。環境の対しての性質とは、例えば次の世代への生き残りやすさなどである。巡回セールスマン問題[1]を例にすると、遺伝情報である回らなくてはならない場所(以下巡回地点と呼ぶ)自体に性質はなく、巡回地点の並びである経路として表現されてはじめて効率が良いのか、悪いのかと性質をもってくる。ここでいう遺伝情報に性質を持たせるというのは、巡回セールスマン問題の場合、ある場所は回らなくてもよい(以下巡回不要地点と呼ぶ)ことや、この場所をまわると次は別の違う場所から回り始めることができることなどを循環地点自体が持つ性質として情報を加えることである。こうすることにより、最適化された経路ではないがそれに近い経路がとれるのではないかと考えられる。つまり遺伝子情報に何かしらの特別な性質を与えることができると考えられる。以降、遺伝情報に何かしらの性質を与えることを遺伝補修飾と定義する。また遺伝的アルゴリズムにおいて遺伝補修飾を使用する場合、上記の巡回セールスマン問題において、例えば巡回不要地点が大量に存在すると経路自体が成り立たなくなってしまうことが考えられる。このことより特殊な性質を持つ遺伝子が大量に存在すると遺伝的アルゴリズムが正常に動作しなくなるおそれがあるので、使用する際は、原則集団数に対して少量の割合でのみ含ませる。NQueen 問題に対する解法として遺伝アルゴリズムを用いる場合、多くの遺伝アルゴリズムでは遺伝子が持つ遺伝情報として駒の配置場所のみを保持しており、遺伝情報自体にはこれといって特別な性質があるとはいえない。そこで以降に NQueen 問題に対して有効であると考えられる 2 種類の遺伝補修飾を提案する。

4.2 劣性遺伝

多くの場合、NQueen 問題を解く遺伝アルゴリズムでは、盤上に N 個の駒が配置されている配置パターンを遺伝子として用いる。本節では、遺伝アルゴリズムで用いる遺伝子の中に劣勢遺伝子を加える手法を提案する。劣勢遺伝子とは、盤上に駒が配置された駒の数が N 個未満であり、駒の配置されていない行が存在する配置パターンを持つ遺伝子である。図 1 に劣勢遺伝子となる配置パターンの例を示す。図 1 の場合、遺伝子座の 5 番目に遺伝補修飾がかかっている。

劣性遺伝をもつ遺伝子を集団内に混ぜることで、劣性遺伝を含む遺伝子と交叉が発生した場合、交叉対象の遺伝子の競合数が下がるように作用する性質がある。例えば、N＝8 の問題において、7 個は正解の配置ができているが、1 個が不正解の配置をとってしまい、競合が発生している部分解があるとする。この不正解部分が劣性遺伝子であれば競合数は 0 となり、最適解ではないが近似的な解(以降本稿ではこれを準解とする)を求めることができる。このことから劣性遺伝子は近似解探索に活用できるのではないかと考えられる。

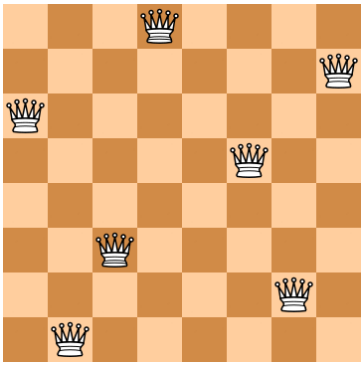

図 4.2.　劣性遺伝子となる配置パターンの例

4.3 完全遺伝

本節では、遺伝アルゴリズムで用いる遺伝子の中に完全遺伝子を加える手法を提案する。完全遺伝子とは、盤上のある行の全ての列に駒が配置された配置パターンを持つ遺伝子である。図 2 に完全遺伝子となる配置パターンの例を示す。図には擬似的な状態を示しており、実際はクイーンがすべて配置されているわけではなく、図の場合、X 軸において右から順番に、5 番目の遺伝子座に駒が配置された、8 つの状態が並列して存在していることをさす。

通常、NQueen 問題の解となる遺伝子は、N 個の駒を競合すること無く配置された配置パターンを持つもののみである。しかし、完全遺伝子を用いた場合では、N-1 個の駒が競合することなく配置された配置パターンでも、解となる場合がある。完全遺伝子を用いるとこのことからサイズ N の問題に対して、実質サイズ N-1 として解探索を行うことができるので解空間を狭め、解探索能力の向上が図れるのではないかと考えられる。

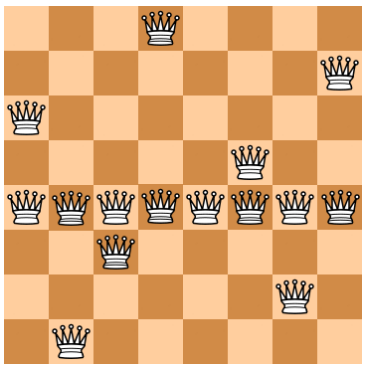

図 4.3　完全遺伝子となる配置パターンの例

5　遺伝修飾の実装

本章では、4 章で述べた劣勢遺伝子および完全遺伝子を用いた遺伝補修飾の実装方法について述べる。

5.1 劣性遺伝の実装方法

統劣性遺伝を実装するにあたり以下の仕様を規定する。

① 劣性遺伝の表現方法
② 致死性遺伝子の回避方法
③ 劣性遺伝を含む競合判定

まず、劣性遺伝の表現方法の仕様を規定する。通常の遺伝子では、各遺伝座が持つ駒の位置情報は、1 以上 N 以下の整数で表現されている。劣勢遺伝子では、遺伝補修飾により駒が無い行が存在するため、その行を表す遺伝座には駒が無いことを示す位置情報を与える必要がある。そこで、本研究では、駒が無いことを位置情報 N+1 で表現する。これにより、劣勢遺伝子用に新たな遺伝オペーレーションを加えることなく、他の遺伝子と同様に劣勢遺伝子を交叉などの判定で使用できる。

次に致死性遺伝子の回避方法を規定する。致死性遺伝子[1]とはその遺伝子情報を含むだけで解とはなりえなくなってしまう遺伝子である。劣勢遺伝子は配置された駒の数が不足するため、NQueen 問題の解とは成り得ない。したがって劣勢遺伝子が遺伝子集団内の多数を占めてしまうと、最適解を得ることができなくなる。よって、交叉により劣性遺伝を大量に含む遺伝子が発生することを抑えなくてはならない。最悪の場合、駒の配置が全くない遺伝子が誕生し、遺伝子集団全体に影響すると考えられる。この致死性遺伝子を回避するために、交叉において、劣性遺伝子同士の交叉が発生した場合、劣性遺伝子の遺伝補修飾部分にはランダムで駒を配置し、劣勢遺伝子ではない通常の遺伝子が生成されるようにする。

最後に劣性遺伝を含む競合の判定を規定する。前述した通り、劣勢遺伝子は NQueen 問題の解とは成り得ない。従って、劣性遺伝子場合、競合数が 0 となっても解保存を行わないようにする。

5.2 完全遺伝の実装方法

完全性遺伝の実装条件として以下の仕様を規定する。

① 完全遺伝の表現方法
② 致死性遺伝子の回避方法
③ 完全遺伝を含む競合判定

まず、完全遺伝の表現方法の仕様を規定する。本研究では劣性遺伝と同じように遺伝補修飾を施した遺伝子座が持つ位置情報を N+1 とする。また、完全遺伝子に対する判定の際には遺伝補修飾を施した列は駒の配置がないものとして扱う。すなわち、判定の際には、完全遺伝子は劣勢遺伝子として扱う。ただし、無条件で解では無いと判定する劣勢遺伝子とは異なり、完全遺伝子では、遺伝補修飾を施した列以外の駒の配置パターンを部分解として、競合数を算出し、算出した部分解の競合数が 0 の場合、解判定を行う。

次に致死性遺伝子方法を規定する。完全遺伝は最悪の場合、全ての遺伝子が完全遺伝となり、判定部分で全ての状態を判定してしまうということが考えられる。そこで、完全遺伝子に対しても、交叉時に劣勢遺伝子と同様の処理を行う。すなわち、交叉において、完全遺伝同士が交叉するとき、完全遺伝子による遺伝補修飾部分にはランダムに駒が配置されるようにする。

最後に完全遺伝の競合の判定を規定する。完全遺伝を含む遺伝子の競合度が 0 となった場合、完全遺伝子による遺伝補修飾部分の行に対し、N 個のマスのいずれか 1 つに駒を配置した遺伝子を N 通り発生させ、発生させた各遺伝子に対して最適解であるか判定する。

5.3 遺伝補修飾を用いた NQueen 問題を解くプログラム

付録に本研究で作成した遺伝補修飾を用いた NQueen 問題を解く遺伝アルゴリズムを実装した c++プログラムを示す。

main.cpp

・main.cpp は各メソッド実行部分である。

perfunc.cpp

・perfunc.cpp は遺伝補修飾である完全遺伝を含む場合、完全遺伝の遺伝情報が保存されている配列部分に、全ての駒の配置方法を実行し、もし解が発見されれば解を保存するプログラムである。

refunc.cpp

・refunc.cpp は遺伝補修飾である劣性遺伝を含む場合、劣性遺伝により発生した準解を保存するプログラムである。

recross.cpp

・recross.cpp は遺伝補修飾同士が交叉し、致死性遺伝になるのを防ぐため、遺伝補修飾同士の交叉の場合、遺伝補修飾部分にランダムに選択された駒の配置情報が格納されるプログラムである。

reinit.cpp

・reinit.cpp は初期集団に遺伝補修飾を混ぜるプログラムである。

本研究では、上記のプログラムを用いて遺伝補修飾を用いた *n*=(*n* の範囲)に対する *n* クイーン個数問題の解および準解を求めた。以下に本研究で用いた計算機のスペックを示す。

CPU:Core2Duo　L7800　2.0GHz

OS:windouwsVista

6　結果・考察

6.1 結果

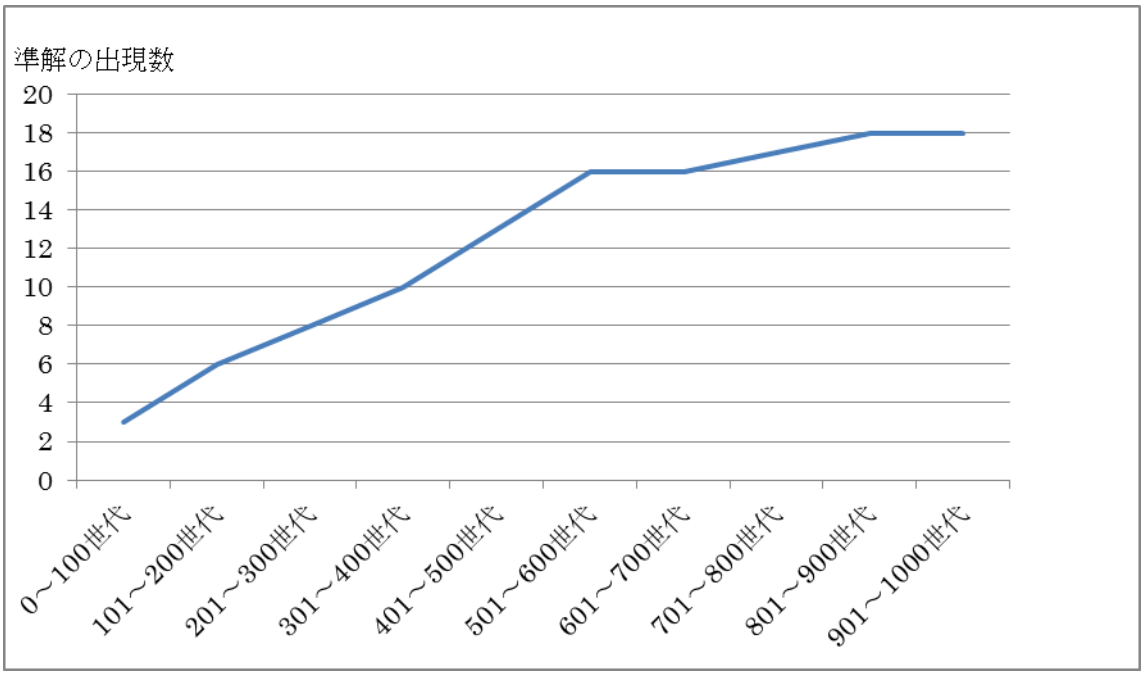

図 6.1.1　準解の出現数

表 6.1.1 プログラムを実行した条件

N	初期集団数	終了世代数	選択方法	交叉方法	突然変異率	遺伝補修飾含有率	試行回数
8	100	1000	ルーレット選択	一点交叉	0.1	20%	1000

本研究では、表 6.1 に示す条件で劣勢遺伝による遺伝補修飾を用いた NQueen 問題の順解を付録の示すプログラムにより求めた。本研究で得られた結果を図 6.1.1 に示す。図 6.1.1 から、世代が進むにつれて、準解の生成率が増えていることがわかる。次にこの準解が安定して生成されているかを調べるために、図 6.1.2 に劣性遺伝が消滅した世代の割合を示す。

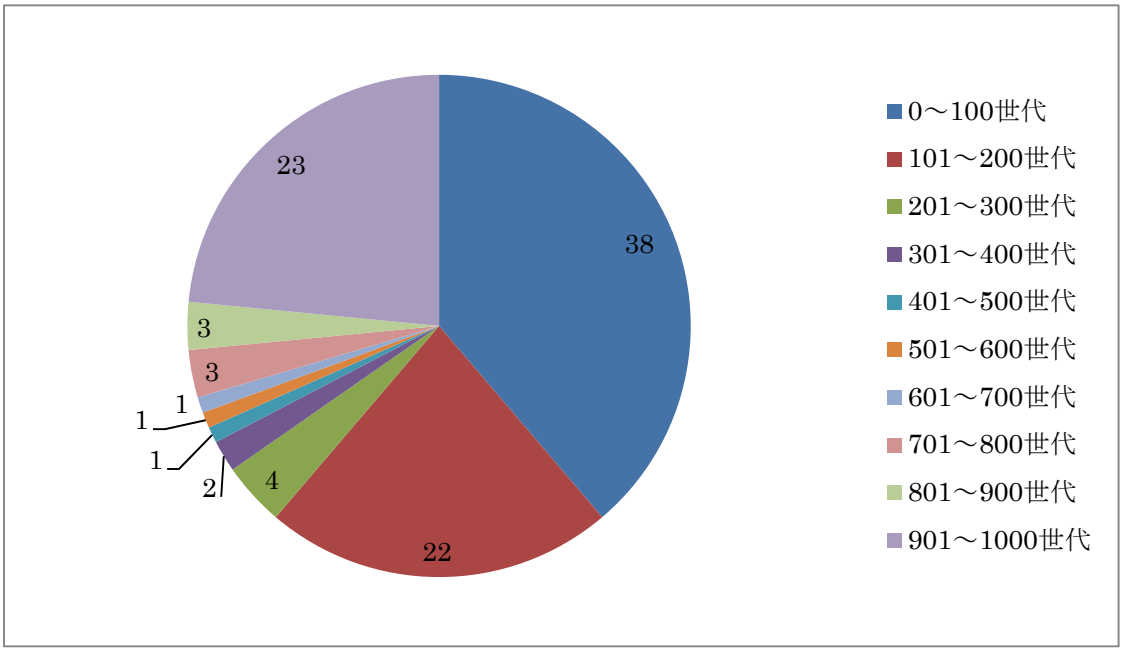

図 6.1.2 劣性遺伝の消滅世代の割合

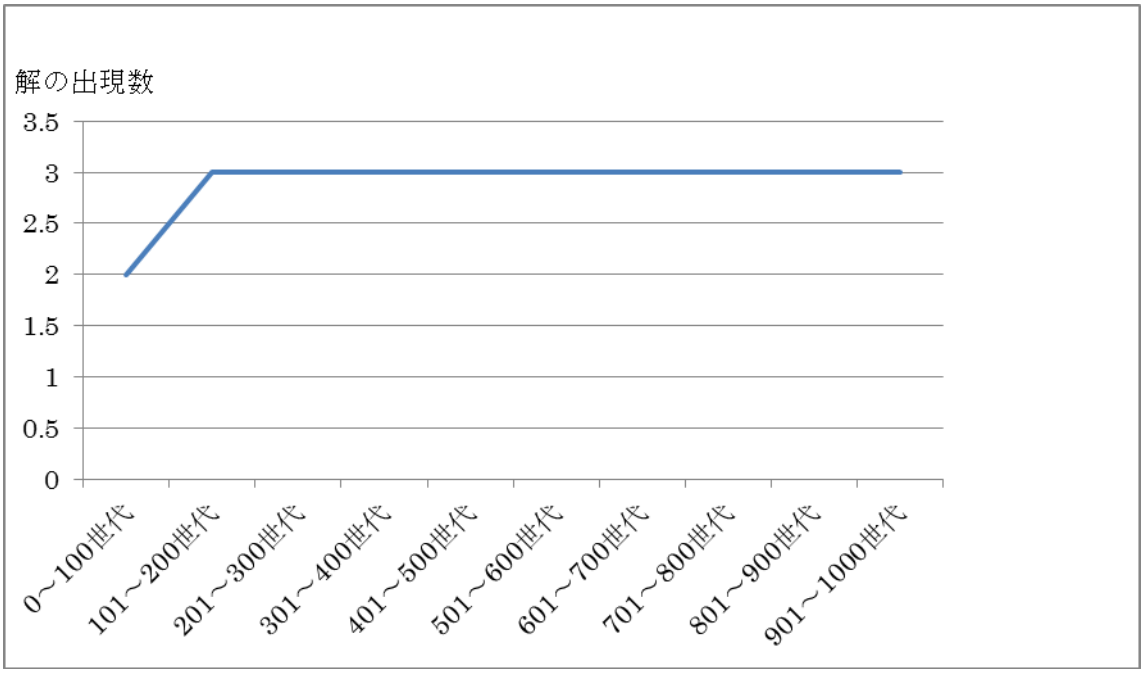

図 6.1.3 完全遺伝を含む解探索による解の個数

図 6.1.2 から初期に劣性遺伝が消滅か、終盤まで生き残るかに二極化しているのがわかる。これは選択方法がルーレットだからということも考えられるが、初期の方が終盤より多いのは、初期の方が終盤に比べ、劣性遺伝同士での交叉が起こりやすく消滅したのではないかと考えられる。しかしながら、目的としていた準解の生成は確認できた。

次に、完全遺伝を用いた解探索を行う。本研究においては、劣性遺伝と同じく表 6.1 の条件を用いて実行した。その実行結果を図 3 に示す。問題サイズ 8 において初期のプログラムは平均して約 2 個程度しか解の生成ができなかった。図 6.1.3 からわかる通り、完全遺伝を用いても効果があったとは言い難い。当初考えていたときとは違い、完全遺伝子を含有しても発見する解に変化はなかった。図 6.1.1 において、終盤では少なくとも 18 個の準解が生成されており、解がでると予想していた。しかしながら図 3 に示す通り、各世代に変化は

なかった。原因として考えられたのは、完全遺伝で解は生成できているが、重複解を生成しているので解としてカウントされていないことである。完全遺伝を含む場合とそうでない場合の解の生成は変化がなかった。そこで、①少なくとも初期には完全遺伝が生存している。②完全遺伝は解探索を行い解の生成を行っている。③完全遺伝子の有無にかかわらず、生成される解の個数は同じ。以上のことから今度は解の生成スピードに着目した。現在の条件ではあまり解が生成されず、結果がわかりにくいので、共同研究者で作成した、もっとも解を生成するプログラムに完全遺伝を適用した。表 6.1.2 にプログラムを実行した条件を示す。完全遺伝による解の生成スピードを検証するため、完全遺伝が消滅するまでに発見された解の何パーセントを完全遺伝が発見できかを調査した。共同研究で作成したプログラムでは解への収束を早めるため、選択で競合数に制約がかかるなど、初期に作成したプログラムにくらべて、遺伝子の淘汰が急速に行われるため 400 世代を超えて生き残れなかった。そこで 400 世代までの割合を図 6.1.4 に示す。図 6.1.4 からわかるように、発見できた解の約 90%が完全遺伝によるものであった。このことから、完全遺伝は解の生成スピードを向上させたと考えられる。世代が進むにつれ割合が低下しているのは、淘汰と交叉により、完全遺伝子が減少したためだと考えられる。また図 6.1.5 に N の大きさ別に実行した結果を記載する。図 6.1.5 より N の大きさが 7 を境に割合が 90%をこえはじめている。これは問題サイズが小さいと解の発見が容易になり、完全遺伝の効果があまりあらわれなかったのではないかと考える。

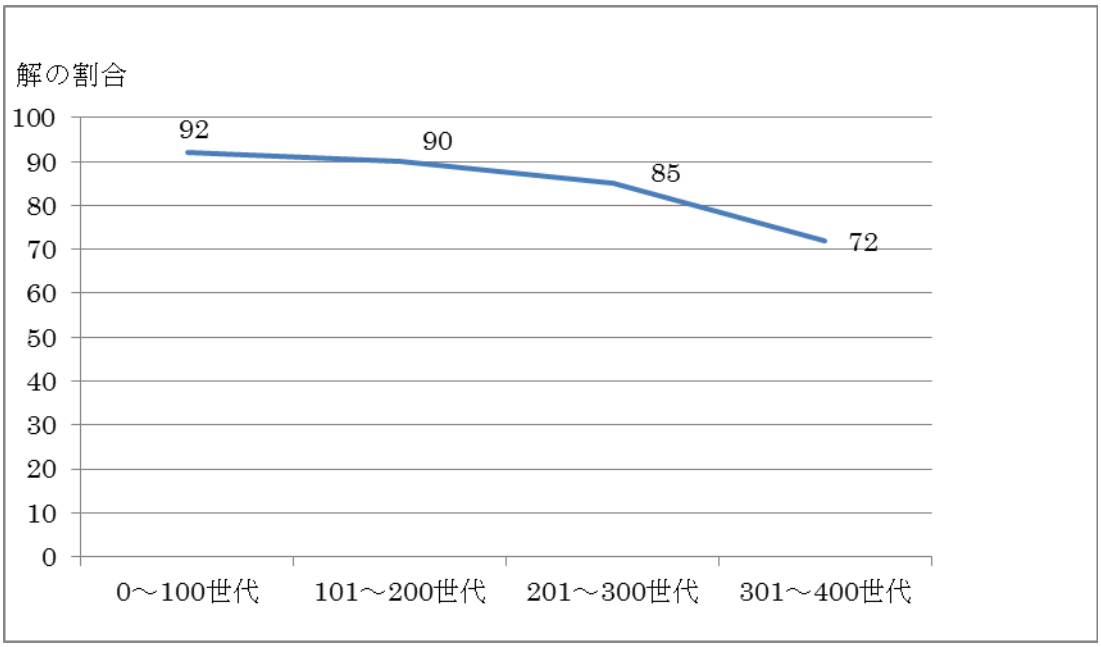

図 6.1.4　完全遺伝消滅までに発見された解に対する完全遺伝の割合

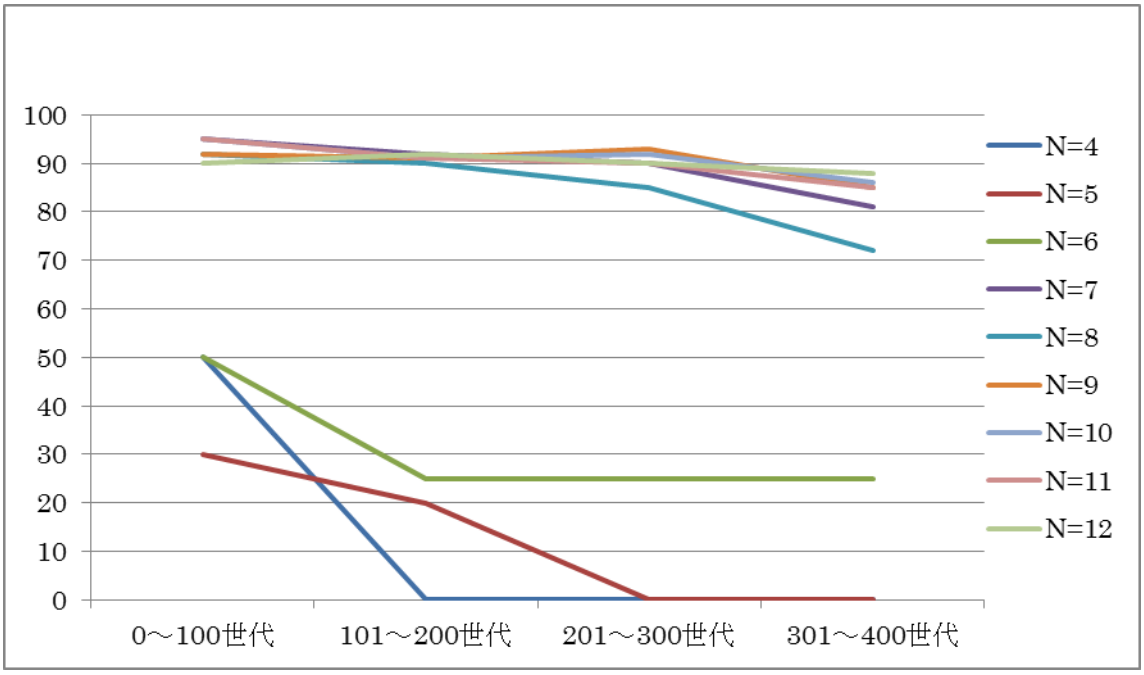

図 6.1.5　N の大きさ別の完全遺伝消滅までに発見された解に対する完全遺伝の割合

6.2 考察

今回の研究結果から劣性遺伝は近似解である準解を生成する効果、完全遺伝子は解の生成スピードを向上させる効果があるとわかった。しかしながら当初目的としていた全解探索にはいたれなかった。初期に作成したプログラムの選択方法、交叉方法、突然変異方法などが、解探索にあまり効果がなく、遺伝補修飾の性能を十二分に引き出せなかったことも考えられるが、遺伝補修飾において特に致死性遺伝子発生を防ぐために提案した、性質を持つ遺伝子同士が交叉によって消滅するという条件がさらに性能低下を招いたと考えられる。

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。これは、遺伝補修飾である完全遺伝で解の生成スピードは向上できたかもしれないが、解の生成自体が確率てきで、その時の遺伝情報によって生成されるためではないだろうか。しかしながら、1.3 章のプログラムは毎回同じ解が生成されるため、その時により、生成される最適解が異なる本プログラムとは単純に比較できないとも考えられる。

7　結論・今後の課題

7.1 結論

本研究で新たなに遺伝オペレーティングを用いず、遺伝補修飾という手法を用いて NQueen 問題の解探索を行った。当初目的としていた解探索能力の向上とはいかなかったが、劣性遺伝では近似解の探索能力の提示が行えた。完全遺伝においては解の生成速度向上がはかれた。

7.2 今後の課題

本研究では遺伝補修飾の消滅が特に性能向上のカギと感じた。今後は集団内において、いかに一定量の遺伝補修飾を保つかが問題となる。消滅回避の方法としては、一定量以下になると突然変異によって遺伝補修飾が誕生したりするなどが考えられる。

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多大なるご迷惑をかけましたお詫びと並々ならぬ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] 知的システムデザイン研究室 GA グループ,
“卒論・修論作成のための基礎シリーズ 遺伝的アルゴリズム,“同志社大学生命医科学部医療情報システム研究室,2009. http://www.is.doshisha.ac.jp/text/ga20090504.pdf

[5] 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

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

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

[8] 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.

[9] 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

[10] 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
	double sprob[N];//選択確率保存用配列
	double cprob[N];//累積確率保存用配列

	vector<int> vectorcount; // 解の出現回数保管用
	vector<string> revector;//劣性遺伝用出現準解保存リスト
	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 reinit(int [][NBIT]);
	void refunc(int [][NBIT],int [N],std::vector<std::string>&,std::vector<std::string>&);
	void perfunc(int [][NBIT],int [N],std::vector<std::string>&);
	void mutation(int [][NBIT],int [N]);//状態交換の突然変異
	void roulette(int [N],double [N],double [N]);
	void select(int[][NBIT],double [N]);
	void recross(int [][NBIT]);//一点交叉


	srand((unsigned)time(NULL));
```

```
	/*時間計測用*/
	QueryPerformanceFrequency(&freq);
	QueryPerformanceCounter(&time_start);//時間計測開始

	reinit(a);//遺伝補修飾用集団の初期化
	//perfunc(a,match,vector);//完全遺伝用競合数判定
	refunc(a,match,vector,revector);//劣性遺伝用競合数判定

	//世代数だけ遺伝オペレーティング実行
	for(int i=0;i<MAX;i++){
		roulette(match,sprob,cprob);
		select(a,cprob);
		recross(a);//遺伝補修飾用一点交叉
		//perfunc(a,match,vector);//完全遺伝用競合数判定
		refunc(a,match,vector,revector);//劣性遺伝用競合数判定
		mutation(a,match);
		//perfunc(a,match,vector);//完全遺伝用競合数判定
		refunc(a,match,vector,revector);//劣性遺伝用競合数判定
	}
	QueryPerformanceCounter(&time_end);//計測時間停止
	int count1 = vector.size();//解となった数を保存
	int count2 = revector.size();//劣性遺伝により発生した準解数を保存
	printf("解：%d 準解：%d¥n",count1,count2);
	printf("処理時間:%d[ms]¥n",(time_end.QuadPart-time_start.QuadPart)*1000 / freq.QuadPart);

}

/**
*@param x 乱数を発生させるもととなる値
*@return  0～1までの値を乱数として返す
*/
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);
}
```

reinit.cpp

```
#include <iostream>
#include "set.h"
/**
*@param a[][NBIT] 集団の各遺伝子およびその遺伝子の駒の配置情報を格納している。
*/
void reinit(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;
                }
        }

        //初期集団にR個劣性遺伝を配置する
        for(int i=0;i<R;i++){
                int r1 = rand()%NBIT;
                a[i][r1] = NBIT+1;
        }

        if(R>0){
                //交換遺伝子保存用
                int inita[1][NBIT];

                //集団ないで遺伝子を混ぜる
                for(int i=0;i<N;i++){
                        int r2 = rand()%N;

                        for(int j=0;j<NBIT;j++){
                                inita[0][j] = a[i][j];
                        }

                        for(int j1=0;j1<NBIT;j1++){
                                a[i][j1] = a[r2][j1];
```

```
                        }

                        for(int j2=0;j2<NBIT;j2++){
                                a[r2][j2] = inita[0][j2];
                        }
                }
        }
}
```

refunc.cpp

```
#include <iostream>
#include "set.h"
#include<vector>
#include<string>
#include<sstream>
using namespace std;
/**
*@param a[][NBIT] 集団の各遺伝子およびその遺伝子の駒の配置情報を格納している。
*@param match[N] 遺伝子集団の各遺伝子の競合数を格納している
*@param vector  競合数判定により競合数0となった駒の配置情報を解として保存する。
*@param revector 劣性遺伝子によって発生した競合数0の準解を保存する
*/
void refunc(int a[][NBIT],int match[N],std::vector<std::string>& vector,std::vector<std::string>& revector){
        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
        std::ostringstream m;//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;
}


//劣性遺伝の有無を確認する
int judge=0;
for(int d=0;d<NBIT;d++){
		if(a[x][d]==NBIT+1){
			judge =1;
		}
}


//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

		//ここから競合数を判定します

		//もしaの値が劣勢遺伝子でなければ判定を行う
		if(a[x][y] != (NBIT+1)){

			//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){

        //劣勢遺伝判定が0なら実行
        if(judge != 1){
                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());

                //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;
                        }
```

```
                        }
                    }
                    else {
                        std::ostringstream m;//string型を連結保存できる変数
                        for(int i=0;i<NBIT;i++){
                            m<<a[x][i];
                        }
                        revector.push_back(m.str());

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


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

perfunc.cpp

```
#include <iostream>
#include "set.h"
#include<vector>
#include<string>
#include<sstream>
using namespace std;
/**
*@param a[][NBIT] 集団の各遺伝子およびその遺伝子の駒の配置情報を格納している。
*@param match[N] 遺伝子集団の各遺伝子の競合数を格納している
*@param vector  競合数判定により競合数0となった駒の配置情報を解として保存する。
*/
void perfunc(int a[][NBIT],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
int perfect_point=0;//完全遺伝が存在する場所を保存する
int p1[NBIT*2-1]; //完全遺伝の競合数判定における右斜め上判定用配列
int q1[NBIT*2-1]; //完全遺伝の競合数判定における右斜め下判定用配列
int r1[NBIT]; //完全遺伝の競合数判定における縦列判定用配列
int sum1=0; //完全遺伝の競合数判定における競合数保存用変数

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


        //完全遺伝の有無を確認する
        int judge=0;
        for(int d=0;d<NBIT;d++){
                if(a[x][d]==NBIT+1){
                        judge =1;
                        perfect_point=d;
                        break;
                }
        }
```

```
//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

        //ここから競合数を判定します

        //もしaの値が劣勢遺伝子でなければ判定を行う
        if(a[x][y] != (NBIT+1)){

                //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){

        //完全遺伝が含まれていない場合
        if(judge != 1){
                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());

                //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;
                        }
                }
        }

        //完全遺伝が含まれている場合
        else {

                //完全遺伝部分の全駒の配置を確認する
                for(int z1=0;z1<NBIT;z1++){
                        a[x][perfect_point] = z1;

                        //右斜め判定用配列の初期化を行っている
                        for(int xe1=0;xe1<NBIT*2-1;xe1++){
                                p1[xe1]=0;
                                q1[xe1]=0;
                        }
                        //縦列判定用配列の初期化を行っている
```

```
for(int xe2=0;xe2<NBIT;xe2++){
		r1[xe2]=0;
}

//駒の数(NBIT個)だけ各駒について競合数を算出する
for(int y1=0;y1<NBIT;y1++){

		int i1=a[x][y1];// x集団のy列目コマの配置情報

		//右斜め上判定
		if(p1[y1+i1]==0){
			p1[y1+i1]=1;
		}
		else{
			sum1+=1;
		}

		//右斜め下判定
		if(q1[y1-i1+(NBIT-1)]==0){
			q1[y1-i1+(NBIT-1)]=1;
		}
		else{
			sum1+=1;
		}

		//縦の判定
		if(r1[i1]==0){
			r1[i1]=1;
		}
		else{
			sum1+=1;
		}

}

if(sum1==0){
		cout << "完全遺伝子で解が発生しました" << endl;
		std::ostringstream m;//string型を連結保存できる変数
		for(int i=0;i<NBIT;i++){
```

```
                                        m<<a[x][i];
                                }
                                vector.push_back(m.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]){
                                                cout << "しかし重複解でした" << endl;
                                                vector.pop_back();
                                                break;
                                        }
                                }
                        }
                        sum1=0;
                }
        }
        sum = 0;//競合数の初期化
    }
}
```

roulette.cpp

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

using namespace std;
/**
*@param match[N] 遺伝子集団の各遺伝子の競合数を格納している。
*@param sprob 各遺伝子の選択確率が格納されている。
*@param cprob 各遺伝子の累積確率が格納されている。
*/
void roulette(int match[N],double sprob[N],double cprob[N]){

    //選択確立を計算し、sprob配列に保存する
    for(int y=0;y<N;y++){
        sprob[y] = 1/(1+(double)match[y]);
    }
```

```
	double sum=0.0;//選択確立の合計値を保存するための変数

	//累積確立のために選択確立の合計値を算出する
	for(int i = 0;i<N;i++){
		sum += sprob[i];
	}


	//累積確立の計算

	//累積のため、最初の部分は単独で算出
	cprob[0] = sprob[0]/sum;
	//累積のため、残りは合計していく
	for(int t=1;t<N;t++){
		cprob[t]= sprob[t]/sum+cprob[t-1];
	}
}
```

select.cpp

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

using namespace std;
/**
*@param a[][NBIT] 集団の各遺伝子およびその遺伝子の駒の配置情報を格納している。
*@param cprob 各遺伝子の累積確率が格納されている。
*/
void select(int a[][NBIT],double cprob[N]){

	int newa[N][NBIT];//選択した集団をa[N][NBIT]に入れ替えるための保存用の配列
	int ns[N];//何番の盤面が選ばれたかを保存

	float rnd(short int);
	float rnd0;


	//乱数を発生させて集団を選択
	for(int w=0;w<N;w++){
```

```
                rnd0 = rnd(1);//rnd()に1を入れてその１から乱数を発生させてrnd0に代入している

                //選択方法によりどの盤が選ばれたかを確認する
                for(int e=0;e<N;e++){
                        if(rnd0<cprob[e]){
                                ns[w] = e;//選ばれた盤の番号を保存する
                                break;
                        }
                }

        }

        //aからnewaに代入するためにanewに保存する
        for(int s=0;s<N;s++){
                for(int t=0;t<NBIT;t++){

                        newa[s][t] = a[ns[s]][t];
                }

        }

        //新しく決まった交叉対象をaに移し変える
        for(int r=0;r<N;r++){
                for(int p=0;p<NBIT;p++){

                        a[r][p] = newa[r][p];
                }
        }
}
```

recross.cpp

```
#include<iostream>
#include"set.h"
/**
*@param a[][NBIT] 集団の各遺伝子およびその遺伝子の駒の配置情報を格納している。
*/
void recross(int a[][NBIT]){
```

```
int w[1][NBIT];//ランダム保管用配列
int crossa[1][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++){
                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
//奇数であれば集団の最後の配列は交叉は起こらない

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[0][e]=0;
        }

        if(crate<rnd0){
                //劣勢遺伝子の有無を確認
                int judge=0;
                for(int ge=0;ge<NBIT;ge++){
                        if(a[x][ge]==NBIT+1){
                                judge += 1;
                        }
```

```
                    if(a[x+1][ge]==NBIT+1){
                        judge += 1;
                    }
                }
                    //劣勢遺伝子同士の交叉であれば消滅させる
                if(judge==2){
                    for(int h1=0;h1<2;h1++){
                        for(int h2=0;h2<NBIT;h2++){
                            if(a[x+h1][h2]==NBIT+1){
                                a[x+h1][h2] = rand()%NBIT;
                            }
                        }
                    }
                }

                //一点交叉
                int cross_point = rand()%(NBIT);
                for(int x1=cross_point;x1<NBIT;x1++){
                    crossa[0][x1] = a[x][x1];
                }
                for(int x2=cross_point;x2<NBIT;x2++){
                    a[x][x2] = a[x+1][x2];
                }
                for(int x3=cross_point;x3<NBIT;x3++){
                    a[x+1][x3] = crossa[0][x3];
                }
                x +=1;
            }
            else{
                x +=1;
            }
        }

}
```

mutation.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] 集団の各遺伝子およびその遺伝子の駒の配置情報を格納している。
*@param match[N] 遺伝子集団の各遺伝子の競合数を格納している
*/
void mutation(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;
                        }
                }
}
```

set.h

```
#define N 100 //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 //縦における競合数の大きさ
#define fitpoint 5 //競合度保存用の大きさ
#define MAX 1000 //何世代まで求めるか
```

```
#define R 20//初期状態に含める遺伝補修飾の数
```