卒業研究報告書

題目

遺伝アルゴリズムによるNQueen解法

～交叉と選択方法の改良による解探索の研究～

指導教員

石水 隆　助教

報告者

08-1-037-0144

奥野 裕太

近畿大学理工学部情報学科

平成24年1月31日提出

概要

制約条件問題の解法の一つに遺伝的アルゴリズムの利用が考えられる。遺伝的アルゴリズムとは厳密な定式を必要とせず、評価関数を用いて解探索を行う確率的最適化法である。しかしながら、複数の最適解が存在する場合、単純な遺伝的アルゴリズムを用いて解探索を行っても限定的な解しか発見できず、探索能力がすぐれているとは言い難い。最適解が複数存在しても求解できるよう遺伝的アルゴリズムを改良する必要がある。本研究では複数の最適解が存在する場合でも求解できるように、遺伝的アルゴリズムの改良を試みる。複数の最適解が存在する問題としてはNQueen問題をとりあげる。これは駒の配置と判定という非常にシンプルな問題構造をとっており、既知の情報があり、解探索能力の比較をしやすいためである。複数の最適解を求めるということで、遺伝子に多様性をもたせる必要だと考えられる。また、解の生成率を上げるためにも遺伝子の淘汰も必要だと考えられる。そこで本研究では遺伝的アルゴリズムにおける「選択」と「交叉」に注目し、改良を行った。

目次

1　序論 5

1.1　本研究の背景 5
1.2　本研究の目的 5
1.3　NQueen 問題 5
1.3.1　組み合わせ最適化問題とは 5
1.3.2　NQueen 問題とは 5
1.3.3　NQueen 問題の既知の結果 6
1.4　本報告書の構成 8

2　遺伝的アルゴリズム 9

2.1　遺伝的アルゴリズムとは 9
2.2　選択 10
2.3　交叉 10
2.4　突然変異 11

3　遺伝的アルゴリズムを用いた NQueen 問題の解探索方法 12

3.1　遺伝子集団の設定 12
3.2　遺伝子コーディング 12
3.3　遺伝子の評価方法 13
3.4　選択方法 13
3.5　交叉方法 13
3.6　突然変異方法 14

4　選択方法の改良 16

4.1　ルーレット選択 16
4.2　エリート選択 17
4.3　トーナメント選択 17
4.4　選択方法の改良 18

5　交叉方法の改良 19

5.1　一点交叉(前後の交換) 19
5.2　二点交叉 20
5.3　一様交叉 21

6　結果・考察 22

7　結論・今後の課題 26

謝辞 26

参考文献 26

付録 26

1　序論

1.1 本研究の背景

制約条件問題の解法の一つに遺伝的アルゴリズム[1]の利用が考えられる。遺伝的アルゴリズムとは厳密な定式を必要とせず、評価関数を用いて解探索を行う確率的最適化法である。しかしながら、複数の最適解が存在する場合、単純な遺伝的アルゴリズムを用いて解探索を行っても限定的な解しか発見できず、探索能力が優れているとは言い難い。最適解が複数存在しても求解できるよう遺伝的アルゴリズムを改良する必要がある。

1.2 本研究の目的

本研究では複数の最適解が存在する場合でも求解できるように、遺伝的アルゴリズムの改良を試みる。複数の最適解が存在する問題としては NQueen 問題をとりあげる。これは駒の配置と判定という非常にシンプルな問題構造をとっており、また、既知の結果が多くあり、解探索能力の比較をしやすいためである。遺伝アルゴリズムを用いて複数の最適解を求めるためには、遺伝子に多様性をもたせる必要があると考えられる。また、解の生成率を上げるためには遺伝子の淘汰も必要だと考えられる。そこで本研究では遺伝的アルゴリウムにおける「選択」[2]と「交叉」[2]に注目し、改良を行った。

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[7]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 1 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。

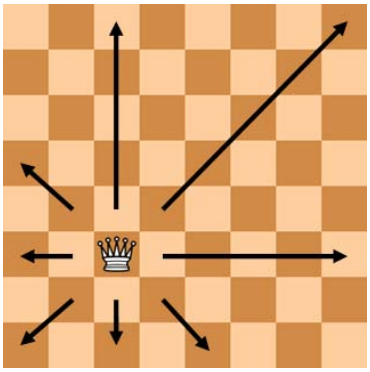


図 1：クイーンの利き筋

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[4]。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[4]までの解探索を行っている。表 1 に[3]に掲載されている解の探索結果を示す。また表 2 に今現在解明されている解の数を示す。

表 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 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	[8]
25	2005.06.11	ProActive	不明	2,207,893,435,808,350	[6]
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 において表 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

2 章には今回用いた遺伝的アルゴリズムについて述べる。3 章では本研究で作成した遺伝的アルゴリズムを用いた NQueen 問題の解探索方法を述べる。4 章では本研究で改良した選択方法について述べる。5 章では本研究で改良した交叉方法について述べる。6 章では本研究の結果および考察を述べる。7 章では本研究の結論および今後の課題を述べる。

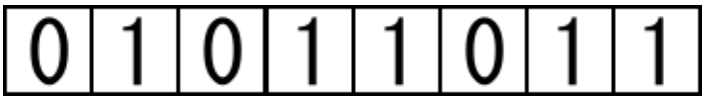


図 2： GTYPE の例

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

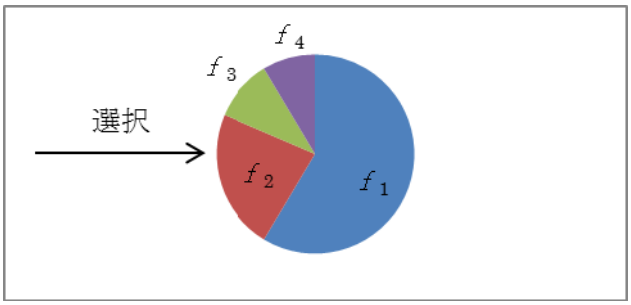


図 3：ルーレット法

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

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 3 である。図 3 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 に N=8 の場合の遺伝子コーディング例を示す。

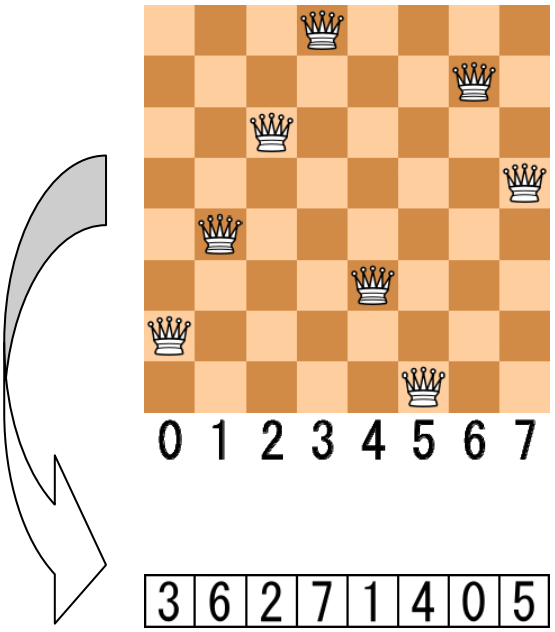


図 4：遺伝子コーディング例

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 5 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。

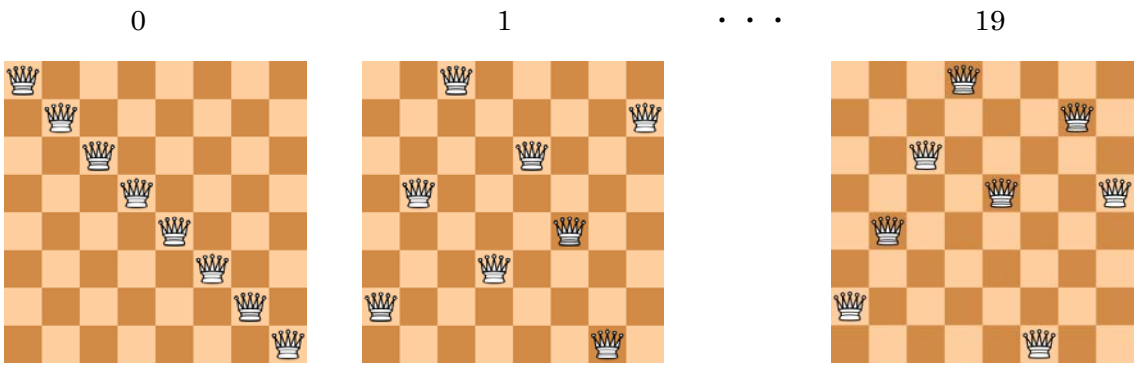


図 5：初期集団の状態

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。

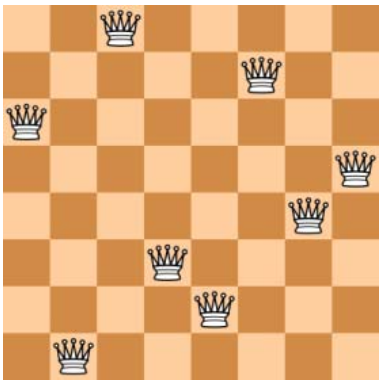


図 6：の配置状態の例

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 7 に示す親 1 と親 2 で交叉が発生したとする。

親 1　　親 2

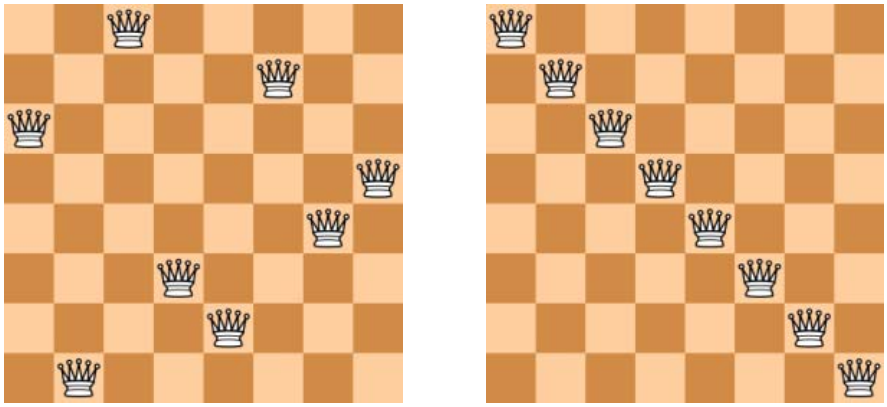


図 7：交叉前

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 8 に示す

子 1　　子 2

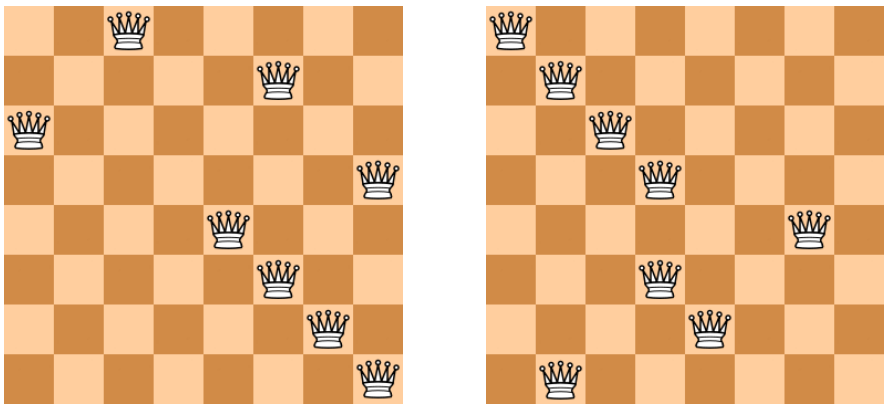


図 8：交叉後

図 8 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。

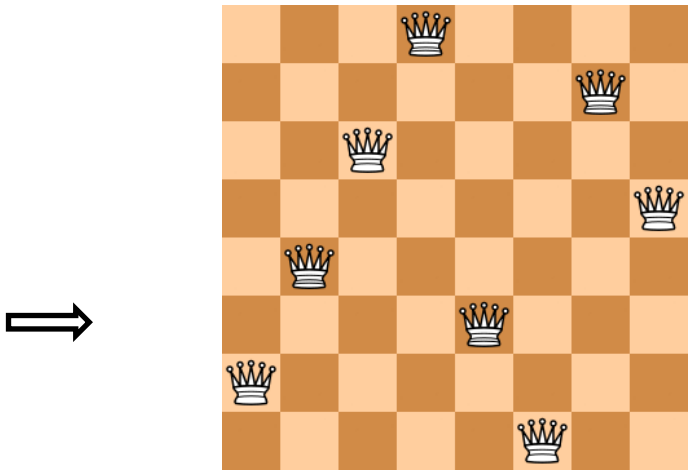


図 9：突然変異発生前

図 9 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 10 に示す。

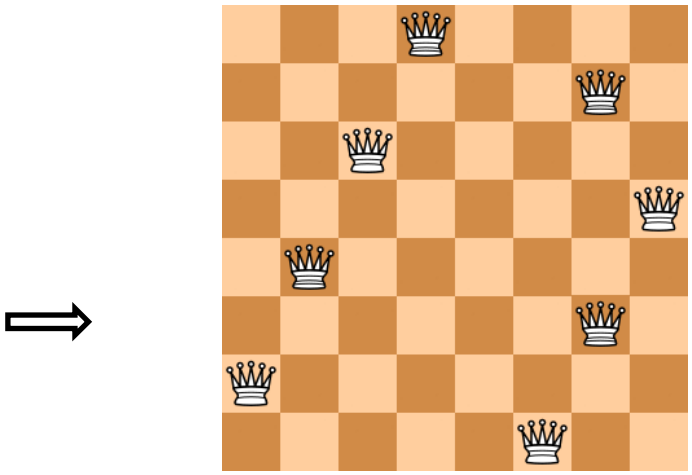


図 10：突然変異発生後

図 9 および図 10 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　選択方法の改良

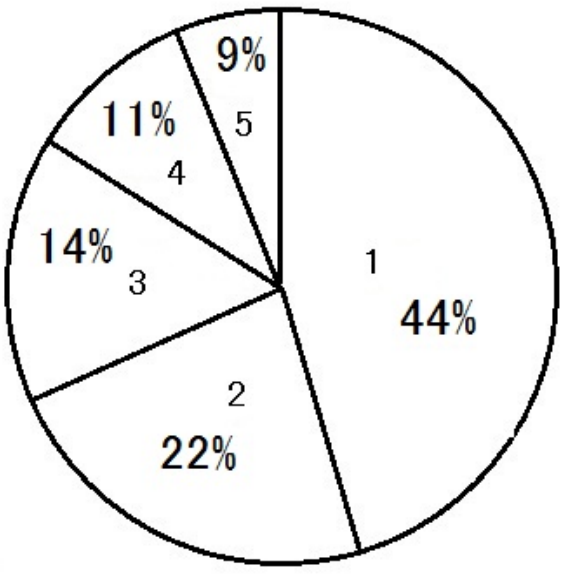
遺伝アルゴリズムでは、多くの場合選択方法としてルーレット[1]選択が用いられる。しかし、NQueen 問題に対して遺伝アルゴリズムを用いる場合、ルーレット選択では、競合数の大きい遺伝子があまり淘汰されず、なかなか解に収束しない。そこで淘汰の速度がはやいとされるエリート選択[2]と、遺伝アルゴリズムの選択方法としてよく用いられるトーナメント選択[2]を用いて改良にあたる。

4.1 ルーレット選択

ルーレット選択とは各個体の選択確率に基づいたルーレット盤を作成し、そのルーレット盤を利用してランダムに個体を選択する選択方法である。集団数を G、個体 i $(1 \leqq i \leqq G)$ の競合数を c_i とした場合個体 i の選択確率 p_i は次の式(1)で定義される。

$$p_i = \frac{\frac{1}{1+C_i}}{\sum_{k=1}^{G} \frac{1}{1+C_k}} \quad \cdots(1)$$

図 1 にルーレット選択の例を示す。G=5 として、仮に競合数 0,1,2,3,4 の個体 1,2,3,4,5 の選択確率が、式(1)よりそれぞれ 45%,25%,15%,10%,5%となるとする。その場合、図 11 ルーレット選択の例では個体 1 が選択される確率が高くなるが、あくまで確率が高くなるだけであり競合数が高いものが選択される可能性もある。つまりはランダムで選択され、解の個数が安定しない可能性がある。そこでルーレット選択は次に述べるエリート選択との組み合わせて使われるのが一般的である。



個体 i	競合数 c_i	$1/(1+c_i)$	選択確率 p_i
1	0	60/137	44%
2	1	30/137	22%
3	2	20/137	14%
4	3	15/137	11%
5	4	12/137	9%

図 11 ルーレット選択の例

4.2 エリート選択

エリート選択は本研究では集団から 3 章で述べた競合数の少ない個体を抽出していく選択方法である。具体的には競合数の低い順番に集団を並び替え、並び替えた順番に遺伝子を選択していく。この選択方法はランダムに個体を選択した結果、競合数の低い個体が選択されないということを回避できる。ただし、同じ個体ばかり選択されてしまい局所解に陥りやすいという欠点がある。また本研究の場合、ほぼ間違いなく局所解に陥ってしまうため改良の必要がある。改良した部分に関しては 4.4 節で説明する。

4.3 トーナメント選択

トーナメント選択は実施するトーナメントの数（トーナメントサイズ）を設定し、集団から個体をそのトーナメントに選ばれた個体から最も優秀な個体を保存する選択方法である。図 2 に NQueen 問題に対するトーナメントサイズ 2 のトーナメント選択の例を示す。図 12 のように本研究では競合数の少ない遺伝子が保存される仕様を用いる。一般的にトーナメントサイズは 2^k とされていて、本研究では、トーナメントサイズを 2^k で変更できる仕様にすることで解の個数などがどうなるか検証し、トーナメント選択が有効かどうか判断する。結果が極端に変化する場合、この選択方法は用いないこととする。

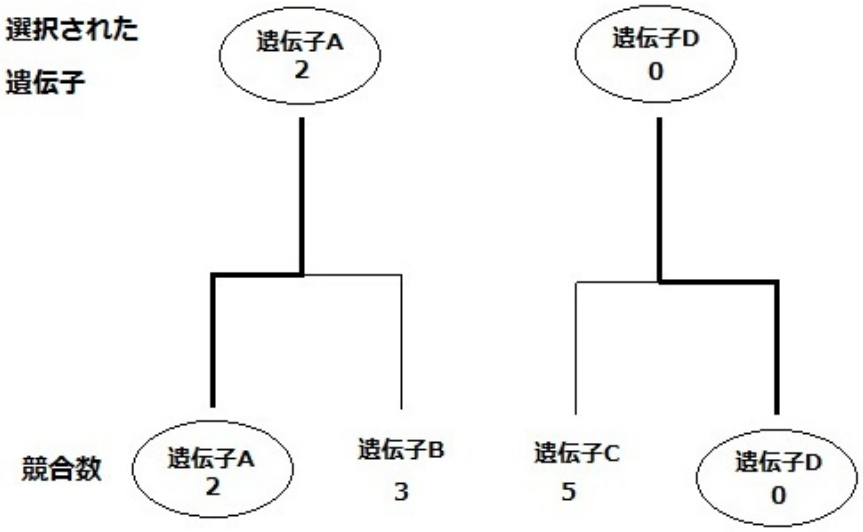


図 12: トーナメント選択の例

4.4 選択方法の改良

上記選択方法を同条件として解の探索を行った結果、エリート選択は、ほぼ安定して解を生成できたのに対して、トーナメント選択は確率的要素がある分、解の生成が安定しなかった。そこで本研究ではエリート選択を改良し、解探索を行う。

本研究で提案するエリート選択の改良点を述べる。

① 重複遺伝子がないように選択する

局所解に陥るということはすなわち解が重複しているということである。そこで生成された解の遺伝子が過去の世代で生成された過去の遺伝子とある一定の回数以上重複する場合解として保存し、なおかつ突然変異が起こるようにする。

② エリートとなる競合数の設定

エリート選択は優秀でない遺伝子ばかりの集団では優秀でない遺伝子のなかで最も優れた遺伝子しか残せない。そこでエリートとなる競合数を設定することでそうなることを避ける。エリートと認められる競合数の設定については４に設定した。これはエリートとしてあまり数字の高いものが選ばれるのを避けるためと、４以降に設定してもあまり解の探索精度が上昇しなかったためである。集団に競合数が４以降のものしか存在しない場合、突然変異のみで次世代に移る。

5　交叉方法の改良

従来の遺伝アルゴリズムで多く用いられる一点交叉、二点交叉をはじめとする交叉方法では NQueen 問題に対しては十分な求解能力が見込めなかった。そこで以下の新たな交叉方法を提案する。

5.1 一点交叉(前後の交換)

通常の一点交叉では交叉点（図 13 のように遺伝子を交換する点）以降の遺伝情報を交換する仕様になっている。 これでは交叉点以前の駒の配置情報は変化しておらず、遺伝子に多様性があまりないと考えられる。そこで交叉点の前後の遺伝情報を交換することを提案する。なお従来の一点交叉のようにランダムに交叉点を決めてしまうと、前後の長さが一致しないことがあるので、N/2 の点で交叉させるようにする。

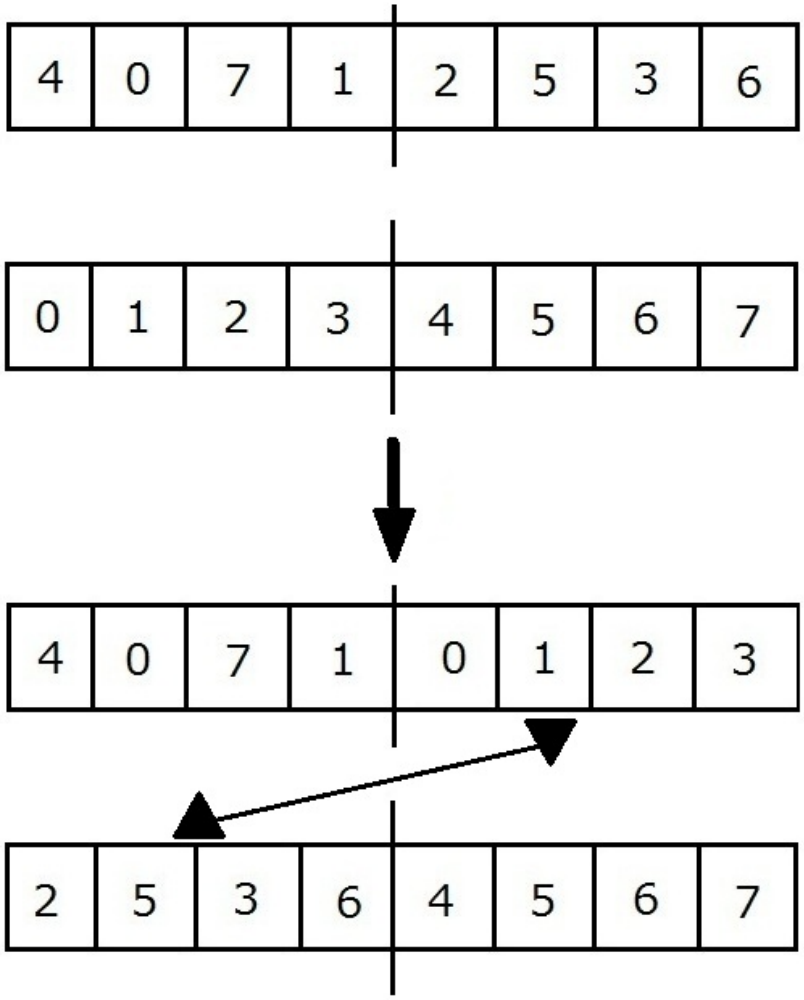


図 13：一点交叉（改良）の例

5.2 二点交叉

二点交叉は、交叉点を２点用意し、交叉点に挟まれた遺伝子配列を交換する交叉法である。同様に、交叉点の数を 3,4…点にしたものを三点交叉、四点交叉等と呼び、これらを総称して多点交叉と言う。多交叉では交叉点を多くするほど親の遺伝子とはかけ離れた子の遺伝子を得ることができる。しかし一般に、交叉点を多くすると、後述する一様交叉とあまり変わらない結果となるため、本研究では二点交叉を用いる。NQueen 問題に対する二点交叉は、ランダムに２点交叉点を用意し交叉点に挟まれた配列を交換する仕様とする。なお２点が同じ交叉点を用意した場合通常の一点交叉を行う。交叉点を本研究では、図 14 に NQueen 問題に対する二点交叉の例を示す。図 4 では遺伝子配列の２番目と５番目を基準として、その間の遺伝子を交換している。

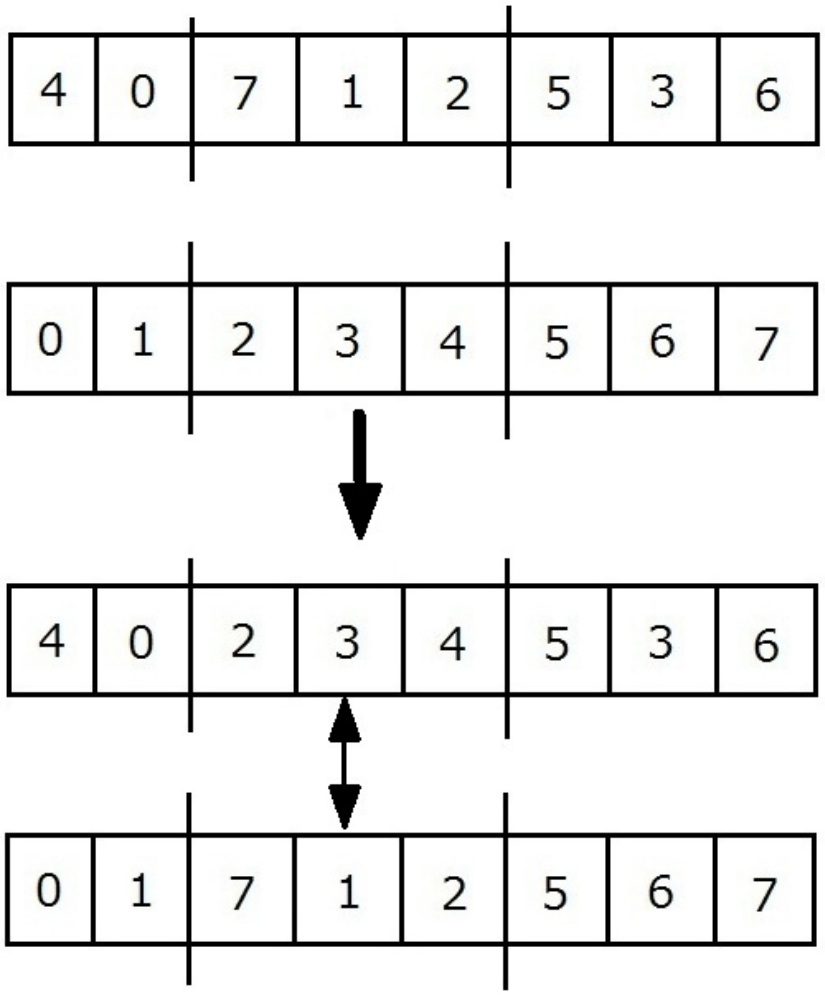


図 14：二点交叉の例

5.3 一様交叉

一様交叉は、任意の値で遺伝子の各配列を交換する交叉法である。今回の場合図 15 のようにマスクパターンを作成しそのマスクに従い遺伝子配列の交換を行う。マスクにはランダムに”1”と”0”を代入し、”1”の場合は交換し、”0”の場合は交換しない。また遺伝子の各配列毎にランダムで”1”か”0”を与え交叉を行うか判別する仕様にすることで、遺伝子の各配列を偏りなく無作為に交叉させることができる。交叉により生成された子の遺伝子は親の遺伝子の配列を継承しにくい。つまり一様にランダムに遺伝子配列の交換ができるため解の数が一番多くでる交叉法だと思われる。

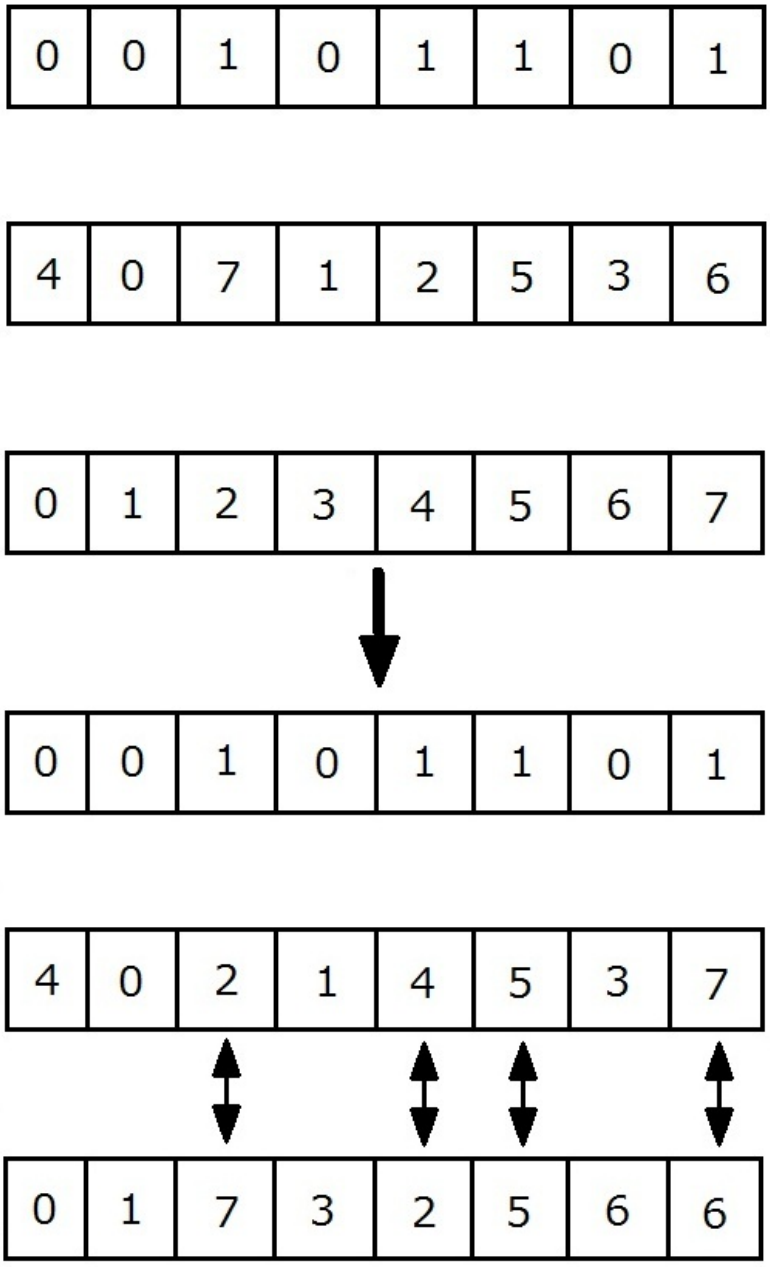


図 15：一様交叉の例

6　結果・考察

本章では、本研究で作成した NQueen 問題を解くプログラムを用いて得た結果について述べる。本研究では OS(WindowsXP Pro),CPU(InterCorei5 2.27GHz),RAM(3.42GB)の PC 上で全解探索プログラムを実行する。実験を行った際の環境は以下の表 3 の通りである。また各選択方法/交叉方法で発見できた解の個数の平均(試行回数 100 回)を表 4 に示す。

表 4 より各交叉方法において、解の発見数が最も多いのは一点交叉を用いたルーレット・エリート複合選択であることが示される。まず選択方法の研究結果について述べる。選択方法で注目する部分は解の発見数もあるが、突然変異発生率・標準偏差も見て行かなければならない。なお、選択方法に着目した実験を行う際は交叉法などその他の条件はすべて統一する。

選択方法としてルーレット選択を用いた場合、どの交叉を用いてもランダム性が高く 2.2.1 節でも述べたように安定して解の個数を発見することができなかった。その結果を表 5 に示す。

そのためエリートとルーレットの複合した選択方法を用いたところ、解は比較的安定したが、突然変異が 30%近く起こっていることが判明した。突然変異が多発すると解探索のランダム性が増してしまう。我々グループで定めた突然変異の起こる妥協できる確率は 10%と定め、この結果は本研究には適さないと判断した。その結果は表 4 に示す。

次にトーナメント選択だが、本研究で作成したトーナメント選択はトーナメントサイズである 2^k の k を変更できる仕様にした。その結果トーナメントサイズによって解の発見数の結果が極端にかわり、突然変異率も安定しないことも判明した。表 4 のトーナメント選択の結果は何度か遺伝アルゴリズムを実施した結果、目標である突然変異率が最も 10%に近づいた時の解の個数の平均値である。しかしそれでも突然変異率が 20%前後のため、この選択も使えない。また行われたトーナメントの数が少ない場合、突然変異率などが安定しない結果となった。集団数を”100”と設定しているため行われるトーナメントの数は 32 までとする。その結果を表 6 に示す。

表 3：実験環境

N	集団数	世代数	交叉率	突然変異率	試行回数
8	100	1000	0.9	0.4	100

表 4：各選択方法/交叉方法での
解の発見数の標準偏差(N=8)

	一点交叉	二点交叉	一様交叉
ルーレット	63.1±6.7	4.3±2.3	49.7±6.3
トーナメント	12.6±10.3	1.6±1.0	5.3±2.9
複合	86.2±2.9	9.2±2.3	70.4±4.5
エリート改良	69.7±3.6	65.2±4.0	79.5±3.4

表 5：各選択方法/各交叉での
突然変異率の平均標準偏差(N=8)

	一点交叉	二点交叉	一様交叉
ルーレット	20.3±0.5%	45.4±50.3%	19.2±0.4%
トーナメント	24.7±14.2%	30.17±36.2%	21.9±11.1%
複合	27.2±0.9%	26.6±13.1%	25.3±0.7%
エリート改良	8.6%±0.2%	13.87±0.6%	9.7±0.09%

表 6：ルーレット選択
試行時のランダム性

	平均値	標準偏差
一様交叉	49.7	±6.3
二点交叉	4.3	±2.3
一点交叉	63.1	±6.7

表 7: トーナメント毎における
解の発見数と突然変異率の平均と標準偏差（N=8）

	解の発見数	突然変異率
2	1.3±2.0	54±67.0%
4	1±1.1	37.3±39.7%
8	1.4±1.3	47.5±43.5%
16	5.3±2.9	21.9±11.1%
32	64±1.7	27.3±3.7%

よって本研究ではエリート選択を用いることとした。しかし単純なエリート選択を用いた場合、局所解に陥ってしまうと、突然変異も多く発生してしまう。そのため、改良の必要がある。改良した点は、前述の 4.4 節で述べた通り。

また表 4 よりエリート選択を用いた時の解の発見数が最も多いのは一様交叉であることが示される。一般的に一様交叉が得意とする問題を二点交叉は苦手としている。そのことは表 1 に顕著に表れている。また一様交叉が用いられた選択方法の突然変異率は比較的低いことも表 5 からわかる。一点交叉も比較的解の平均発見数は多いが、ランダム性が高いので本研究には適さない。従って、遺伝アルゴリズムを用いた NQueen 問題の解法において、交叉方法は一様交叉を用いるのが最適であると言える。一様交叉が NQueen 問題に適した交叉法であるのはランダム性が高く世代毎に異なった集団が生成されるためである、つまり最適解である競合数 0 の個体が多く抽出されるためであると思われる。

表 8：単純遺伝アルゴリズムと
改良遺伝アルゴリズムの比較(N=8)

	初期集団数	世代数	解の個数	処理時間[ms]
改良前	100	1000	1.1	4132.4
改良後	100	1000	72.2	5851.7

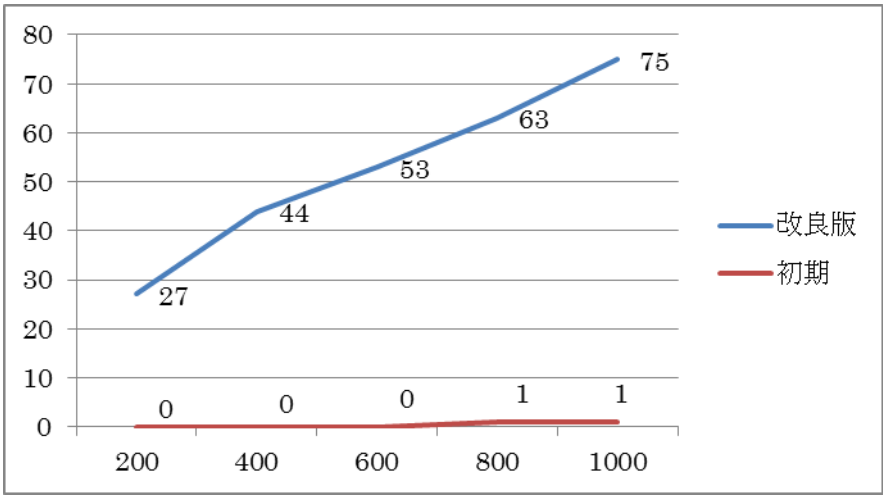


図 16：各世代における解の平均出現数

本研究では、競合数の許容範囲を変化させながらプログラムを実行し、最適な許容範囲の値を求めた。本研究により得られた性能向上結果を表 5 に示す。また各世代における解の出現回数を図 16 に表す。

改良前の単純遺伝アルゴリズムは、一点交叉・ルーレット選択を用い、突然変異も研究初期に用いたもので、改良後の遺伝アルゴリズムは一様交叉・改良エリート選択・突然変異を完成させたものである。

なお交叉・選択方法以外についてはグループ研究として共同研究したメンバーの各研究内容のオペレーティングを用いた。表 8 から、改良遺伝アルゴリズムを用いることにより発見できる解の個数は大幅に上昇した一方、処理時間の増加が示された。また問題サイズ N を大きくすることで処理時間も増加した。処理時間のグラフを図 17 に示す。

また改良した遺伝アルゴリズムで様々なクイーン数でも実験してみた。その結果 N の値が”9”からは集団数”100”、世代数”1000”では満足のいく結果が得られなくなった。その結果を表 9 および図 18 に示す。世代数や集団数の数値を上げるとクイーン数が 9 以降も求まることから汎用性のある遺伝アルゴリズムが作成されているといえる。

既知の結果との比較をすると表 1 からもわかるように、処理速度も解の探索精度も劣っていると思われる。

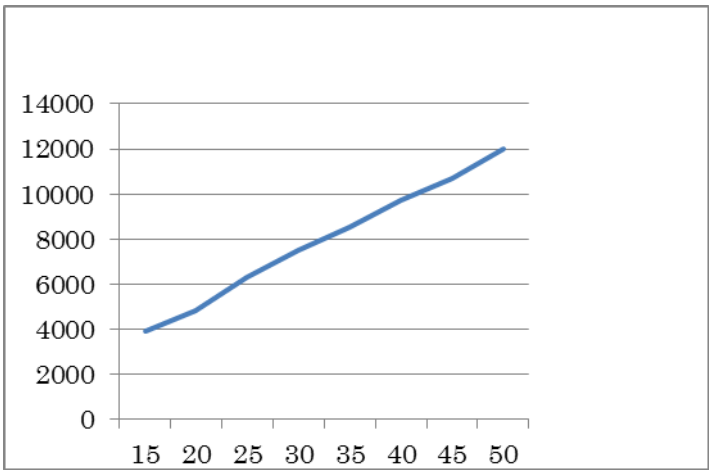


図 17：問題サイズにおける実行時間

(集団数:100,世代数:1000)

表 9：クイーン数と解の個数

クイーン数	4	5	6	7	9
解の個数	2	10	2.1	39.6	80.3
全解数	2	10	4	40	352

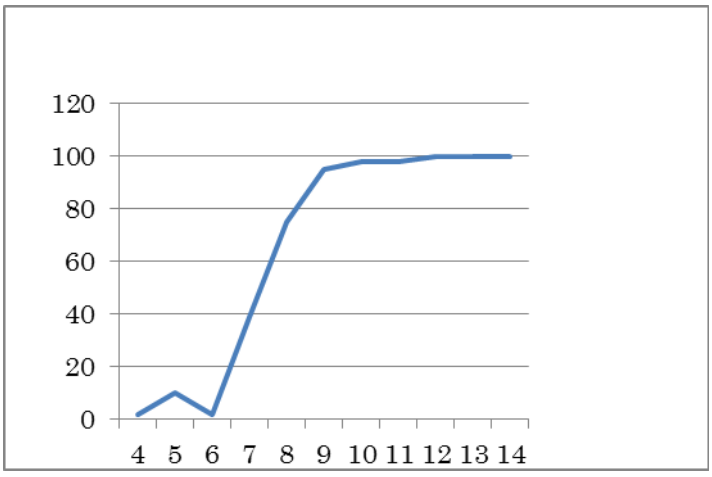


図 18：問題サイズによる解探索結果

(集団数:100,世代数:1000)

7　結論・今後の課題

本研究で、NQueen 問題に対し最適な交叉と選択方法は「一様交叉」と「エリート選択」であることが示された。ただし、今回改良したエリート選択・一様交叉と設定した世代数ではすべての解を抽出することができなかった。また、ルーレット・エリートの複合選択、トーナメント選択に関して突然変異の確率が高く交叉があまり行われていないというだけで、全解探索に近い精度のものも作成できており、より交叉が行われる選択方法の改良を研究する必要がある。また今回は GPU に実装後処理速度をあげるということで処理速度をまったく気にしなかったため、解の個数を求める精度は高いものが作成できたが、改良前に比べると遅いものができてしまった。より処理速度に影響のない改良を見つける必要があるかもしれない。

謝辞

本論文は近畿大学理工学部情報学科在籍中に、同学科石水研究室にて行った研究活動の成果をまとめたものである。遺伝アルゴリズムや NQueen 問題の知識、研究内容、プログラム作成、本論文の書き方ならびに就職活動までに至る御指導、御教示を賜った石水　隆助教に深く感謝し、また研究活動を進めるにあたり、同研究室のメンバーに大いに励まされたことに感謝する。最後に本大学に入学時から現在に至るまであらゆる場で御指導していただいた情報学科のすべての先生方に敬意と感謝の意を表すとともに素晴らしい学習環境を用意して頂いた近畿大学に感謝する。

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] 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

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

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

[6] 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.

[7] 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

[8] 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

[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>//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

#include "set.h"

using namespace std;

void main(){//各メソッドを受け取り実行するメソッド

    int a[N][NBIT]; // 初期集団
    int      match[N]; // 競合数を保存ための配列
    int save[NBIT*fitpoint]; //出現した競合数を保存する配列
    int gene[1]; //解の出現世代保存用
    int geneblockcheck[N]; //阻害したかどうか比較するための配列
    double mcount=0; //突然変異の発生回数を調べる
    int crosscount=0;
    int mutacount=0;
    double sprob[N]; // 選択確率を保存するための配列
    double cprob[N]; // 累積確率を保存するための配列

    srand((unsigned)time(NULL));

    vector<int> vector1; // 解出現世代保管用
    vector<int> vectorcount; // 解の出現回数保管用
    vector<int> geneblockcount; // 世代ごとに阻害した数をカウントする
    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
    vector<int> tourna; //トーナメントで選ばれた遺伝子の番号を保存する
    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
```

```
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],int [NBIT*fitpoint]);  //initメソッド呼び出し、initに配列を渡す
void print0(int [NBIT*fitpoint],std::vector<std::string>,std::vector<int>,std::vector<int>);
void print1(int,int);
void print2(std::vector<int>&); //解が発生した世代と平均発生世代数を表示する
void print3(std::vector<int>&,double &); //突然変異により進化阻害がどの程度発生したか確認する

void func(int [][NBIT],int [N],std::vector<std::string>&,std::vector<int>&);
void func1(int [][NBIT],int [N],std::vector<std::string>&,std::vector<int>&,int [1],std::vector<int>&);

void cfunc(int [][NBIT],int [N],std::vector<std::string>&,std::vector<int>&,int &);
void mfunc(int [][NBIT],int [N],std::vector<std::string>&,std::vector<int>&,int &);
void mutation( int[][NBIT], int[N], double
&,std::vector<std::string>&,std::vector<int>&,std::vector<std::string>&,std::vector<int>&);

void elite(int [][NBIT],int [N]);
void roulette(int [N],double [N],double [N]);
void tournament(int [][NBIT],int [N],std::vector<int>&);
void select(int [][NBIT],int [N]);
void select_r(int [][NBIT],double []);
void select_t(int [][NBIT],std::vector<int>&);
void select_h(int [][NBIT],int [N],double [N]); //ルーレットとエリートの複合型
void select_e(int [][NBIT],int [N]);//エリート改良前

void cross(int [][NBIT]);//一様交叉
void cross1(int [][NBIT]);//一点交叉
void cross2(int [][NBIT]);//二点交叉

void count(int[NBIT*fitpoint],int [N]); // 出現競合数カウント

void genecheck(int [][NBIT],int [1],std::vector<int>&); //解の出現した世代を確認する

void geneblockcheck1(int [N],int [N]); //突然変異前の競合度を確認する
void geneblockcheck2(int [N],int [N],std::vector<int>&); //突然変異により競合度が増加した遺伝子を数える

/*時間計測用*/
```

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

init(a,save); //初期集団の生成

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

        //エリート選択改良型
        //elite(a,match);
        //select(a,match);

        //エリートとルーレットの複合型
        //elite(a,match);
        //roulette(match,sprob,cprob);
        //similcheck(a,simil);
        //select_h(a,match,cprob);
        //print1(a,match,simil,sprob,cprob);

        //ルーレット選択
        roulette(match,sprob,cprob);
        select_r(a,cprob);

        //トーナメント選択
        //tournament(a,match,tourna);
        //select_t(a,tourna);

        //エリート改良前
        //elite(a,match);
        //select_e(a,match);

        //cross(a);
        //cross1(a);
        cross2(a);
        gene[0] = i+1;
        func1(a,match,vector,vectorcount,gene,vector1);
        cfunc(a,match,vector,vectorcount,crosscount);
        count(save,match);
```

```
            //genecheck(a,gene,vector1);
            geneblockcheck1(match,geneblockcheck);
            mutation(a,match,mcount,vector,vectorcount,muvector,mucount);
            gene[0] = i+1;
            //genecheck(a,gene,vector1);
            func1(a,match,vector,vectorcount,gene,vector1);
            mfunc(a,match,vector,vectorcount,mutacount);
            geneblockcheck2(match,geneblockcheck,geneblockcount);
            count(save,match);
    }

    QueryPerformanceCounter(&time_end);//計測時間停止

    print0(save,vector,vectorcount,vector1);
    print1(crosscount,mutacount);
    print2(vector1);
    print3(geneblockcount,mcount);
    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数値
*@return 0～1の間の数値を返す
*/
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],int counta[NBIT*fitpoint]){ //初期化

    //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;
            }
    }

    for(int y=0;y<NBIT*fitpoint;y++){

            counta[y]=0;
    }
}
```

・func1.cpp

```
#include <iostream>
#include "set.h"
#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 vector1 解の出現した世代が格納されている
*/
void func1(int a[][NBIT],int match[N],std::vector<std::string>& vector,std::vector<int>& vectorcount,int
    gene[1],std::vector<int>& vector1){
    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
                //ここから競合数を判定します
                //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);
```

```
                        vector1.push_back(gene[0]);

                        //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();
                                        vector1.pop_back();
                                        vectorcount[j] += 1;
                                        break;
                                }
                        }
                }
                sum = 0;//競合数の初期化
        }
}
```

・cross(一様交叉)

```
/*
* 一様交叉 ver1.0
* 交叉対象同士の各遺伝子ごとに交叉判定する
*/

#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
	//奇数であれば集団の最後の配列は交叉は起こらない

	//偶数パターン
	if((N%2)==0){
		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;
			}
```

```
                }
        }

        //奇数パターン
        else {
                for(int x=0;x<N-1;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;
                        }
                }
        }
}
```

・cross1（一点交叉）

```
/*
* 一点交叉 ver1.1
* 交叉対象 1 は交叉ポイント以降の駒の並びを
* 交叉対象 2 は交叉ポイント以前の駒の並びを入れ替える
*/

#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 cross1(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
```

```
//偶数パターン
if((N%2)==0){
	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){
			//交叉点の決定
			int cross_point = rand()%(NBIT);
			//交叉対象1の交叉点以降を保存する
			for(int x1=cross_point;x1<NBIT;x1++){
				crossa[0][x1] = a[x][x1];
			}
			//交叉対象1に交叉対象2の交叉点以前の駒の並びを保存する
			for(int x2=0;x2<(NBIT-cross_point);x2++){
				a[x][x2] = a[x+1][x2];
			}
			for(int x3=0;x3<(NBIT-cross_point);x3++){
				a[x+1][x3] = crossa[0][(cross_point+x3)];
			}
			x +=1;
		}
		else{
			x +=1;
		}
	}
}

//奇数パターン
else {
	for(int x=0;x<N-1;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){
                        //交叉点の決定
                        int cross_point = rand()%(NBIT);
                        //交叉対象1の交叉点以降を保存する
                        for(int x1=cross_point;x1<NBIT;x1++){
                            crossa[0][x1] = a[x][x1];
                        }
                        //交叉対象1に交叉対象2の交叉点以前の駒の並びを保存する
                        for(int x2=0;x2<(NBIT-cross_point);x2++){
                            a[x][x2] = a[x+1][x2];
                        }
                        for(int x3=0;x3<(NBIT-cross_point);x3++){
                            a[x+1][x3] = crossa[0][(cross_point+x3)];
                        }
                        x +=1;
                    }
                    else{
                        x +=1;
                    }
                }
            }
}
```

・cross2（二点交叉）

```
#include<iostream>
#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
*/
void cross2(int a[][NBIT]){

    int cross_p1 = rand()%(NBIT);
    int cross_p2 = rand()%(NBIT);
    int change=0;
    int w[1][NBIT];//ランダム保管用配列

    if(cross_p2<=cross_p1){
            change = cross_p1;
            cross_p1 = cross_p2;
            cross_p2 = change;
    }
    //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];
                    }
            }

    int croa[N][NBIT];
    for(int x=0;x<N;x++){
                    for(int y=0;y<NBIT;y++){
                            croa[x][y]=0;
```

```
                    }
          }

if((N%2)==0){


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

                    for(int x1=0;x1<cross_p1;x1++){
                              croa[x][x1] = a[x][x1];
                              croa[x+1][x1] = a[x+1][x1];
                    }
                    for(int x2=cross_p1;x2<=cross_p2;x2++){
                              croa[x][x2] = a[x+1][x2];
                              croa[x+1][x2] = a[x][x2];
                    }
                    for(int x3=cross_p2+1;x3<NBIT;x3++){
                              croa[x][x3] = a[x][x3];
                              croa[x+1][x3] = a[x+1][x3];
                    }

                    for(int h=0;h<NBIT;h++){
                              a[x][h]=croa[x][h];
                              a[x+1][h]=croa[x+1][h];
                    }

                    x += 1;
          }
}
else{
          for(int x=0;x<N-1;x++){
                    for(int x1=0;x1<cross_p1;x1++){
                              croa[x][x1] = a[x][x1];
                              croa[x+1][x1] = a[x+1][x1];
                    }
                    for(int x2=cross_p1;x2<=cross_p2;x2++){
                              croa[x][x2] = a[x+1][x2];
                              croa[x+1][x2] = a[x][x2];
```

```
                        }
                        for(int x3=cross_p2+1;x3<NBIT;x3++){
                                croa[x][x3] = a[x][x3];
                                croa[x+1][x3] = a[x+1][x3];
                        }
                        for(int h=0;h<NBIT;h++){
                                a[x][h]=croa[x][h];
                                a[x+1][h]=croa[x+1][h];
                        }
                        x += 1;
                }
        }
}
```

・roulette（ルーレット）

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

using namespace std;
/**
*@param match[N] 遺伝子の競合数が格納されている
*@param sprob[N] 各個体の計算した選択確率を格納している
*@param cprob[N] 各個体の計算した累積確率を格納している
*/
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];
    }
}
```

・elite（エリート）

```
/*
競合数が低いものから順に並び替えを行っている
*/

#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++){
                    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;
                        }
                }
        }
}
```

・tournament（トーナメント）

```
#include <iostream>
#include "set.h"
#include<vector>
/**
*@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 tourna  トーナメントで選ばれた遺伝子の番号を格納している
*/
void tournament(int a[][NBIT],int match[N],std::vector<int>& tourna){

    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

    int tcount =0;
    int matchnumber=0;

    //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];
            }
            int x = match[i];
            match[i] = match[r];
            match[r]=x;
    }

    //偶数の場合
    if((N%2) == 0){
            for(int i=0;i<N;i++){
                    if(tcount != tournumber){
                            if(match[i]>match[i+1]){
                                    matchnumber = i+1;
                            }
                            tcount +=1;
                    }
                    else{
                            if(match[i]>match[i+1]){
                                    matchnumber = i+1;
                            }
                            tourna.push_back(matchnumber);
                            tcount =0;
                            matchnumber = i+1;
                    }
                    i +=1;
            }
    }
}
```

・select(エリート改良後選択)

```
/*
* ・エリートとする競合度を設定することにより
*   一定の競合度を持ったエリートのみを選択できる
*/

#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;
    std::vector<std::string> evector2;

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

                    for(int x=0;x<size;x++){
                            if(evector2[x]==evector2[size]){
                                    evector2.pop_back();
```

```
                                        evector1.pop_back();
                                }
                                break;
                        }
                }
        }

        if(evector1.size()>0){
                int maxsize =0;

                if(evector1.size() < elitenumber){
                        maxsize = evector1.size();
                }
                else{
                        maxsize = elitenumber;
                }

                int w=0;

                for(int i=0;i<N;i++){
                        if(w <= maxsize){
                                for(int j=0;j<NBIT;j++){
                                        newa[i][j] = a[evector1[w]][j];
                                }
                                if((w+1)<evector1.size()){
                                        w +=1;
                                }
                        }
                        else{
                                for(int k=0;k<NBIT;k++){
                                        newa[i][k] = a[evector1[w]][k];
                                }
                                w = 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();

        }
        else{
                evector1.clear();
                evector2.clear();
        }
}
```

・select_r（ルーレット選択）

```
#include<iostream>
#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 cprob[N] 各個体の計算した累積確率を格納している
*/
void select_r(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];
                }
        }
}
```

・select_e（エリート選択改良前）

```
#include<iostream>
#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
*/
void select_e(int a[][NBIT],int match[N]){

        int enumber = N/4;//選択するエリートの数
```

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

    for(int i=0;i<N;i++){
            for(int j=0;j<NBIT;j++){
                    newa[i][j] = a[cnum][j];
                    newm[i] = match[cnum];
            }
            cnum +=1;
            if(cnum==enumber){
                    cnum = 0;
            }
    }
    for(int s=0;s<N;s++){
            for(int t=0;t<NBIT;t++){
                    a[s][t] = newa[s][t];
                    match[s] = newm[s];
            }
    }
}
```

・select_t（トーナメント選択）

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

#include <vector>
/**
*@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 vector 解となった配置を格納している
*@param tourna  トーナメントで残った遺伝子を格納している
*/
void select_t(int a[][NBIT],std::vector<int>& tourna){

    int newa[N][NBIT];
    int w =0;
    int size = tourna.size();
```

```
}
for(int i=0;i<N;i++){
		if(w != size-1){
			for(int j=0;j<NBIT;j++){
				newa[i][j] = a[tourna[w]][j];
			}
			w +=1;
		}
		else{
			for(int j=0;j<NBIT;j++){
				newa[i][j] = a[tourna[w]][j];
			}
			w = 0;
		}
	}

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

	tourna.clear();
```

・select_h(複合型の選択)

```
#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
*@param cprob[N] 各個体の計算した累積確率を格納している
*/
void select_h(int a[][NBIT],int match[N],double cprob[N]){

	int enumber = N/4;//選択するエリートの数
	int cnum = 0;
	int newa[N][NBIT];
```

```
    int ns[N/2];

    float rnd(short int);
    float rnd0;

    //保存するエリートを選択
    for(int i=0;i<enumber;i++){
            for(int j=0;j<NBIT;j++){
                    newa[i][j]=a[i][j];
                    newa[i+enumber][j]=a[i][j];
            }
    }

    //ルーレットにより保存する遺伝子を選択
    for(int w=0;w<N/2;w++){
            rnd0 = rnd(1);
            for(int e=0;e<N;e++){
                    if(rnd0<cprob[e]){
                            ns[w] = e;
                            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/2);s++){
            for(int t=0;t<NBIT;t++){
                    newa[s+(N/2)][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];
            }
    }
}
```

・print0

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

using namespace std;
/**
*@param counta 競合数の出現回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 vector1 解の出現した世代が格納されている
*/
void  print0(int  counta[NBIT*fitpoint],std::vector<std::string>  vector,std::vector<int>  vectorcount,std::vector<int>
    vector1){

    for(int i=0;i<NBIT*fitpoint;i++){

            cout << "競合数" << i <<"の個数"<< counta[i] << endl;
    }

    int w = vector.size();
    for(int o=0; o<w;o++){
            cout <<"解:" <<o+1 <<"番目は" <<vector[o] << " 出現世代"<< vector1[o] << "  出現回数"<< vectorcount[o] <<endl;
    }

}
```

・set

```
#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度

#define tournumber 32 //トーナメントを実施する数
#define elitenumber 3 //選択するエリートの数

#define permatch 2 //エリートとする許容競合度
#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
```