2004年度 卒業論文

確率モデル遺伝的アルゴリズム EHBSAにおける戦略、パラメータの調査

提出日　：2005年2月2日
指導　　：上田　和紀 教授

早稲田大学理工学部情報学科
学籍番号：1G01P046-7

酒井　大輔

概 要

本研究では、確率モデル GA の一種である EHBSA(edge histogram based sampling algorithm) における適切な戦略、パラメータを、主に解の精度の観点から調べた。

確率モデル GA とは、従来の GA (genetic algorithm) のような交叉と突然変異を基本としたものではなく、集団の個体分布を確率モデルで近似して、そのモデルに基づいて次世代の集団を生成する手法で、高い探索能力を持つと期待されているが、実際の問題への適用はあまり進んでいない。確率モデル GA の実際の問題への応用に関する研究として、巡回セールスマン問題について筒井茂義氏が提案した EHBSA ([4]) があるが、従来の GA で用いられている局所的な改善法や集団の選択法は用いられていず、実際に実装する上で基本的なデータの蓄積が無い。

そこで、本研究では関数型の言語のひとつである OCaml を用いて、EHBSAWO (EHBSA without template)、EHBSAWT (EHBSA with template) とその並列版である MPIEHBSAWO、MPIEHBSAWT のプログラムを実装し、様々な戦略、パラメータを指定して実行し、その結果を調べた。また測定と実験結果の分析の手間を削減するために、補助のプログラムを実装し、できるだけ効率良くデータを分析できるようにした。

実験の結果、実効速度以外は EHBSAWO のほうが EHBSAWT より優れていること、初期集団の選択法としてはエリート戦略、トーナメント戦略が優れていること、EHM の重みづけ戦略は有効であること、集団数としては都市数の 4 分の 1 程度が適当であり、これ以上大きくなると実行時間、解の精度共に悪くなること、解の集束には大体 100 世代程度みておけばよく、これは都市数に依存しないこと、複数の EHM を持ち、世代毎にいくつかの集団の要素を交換する並列化ではエリート戦略が有効であり、解の精度を 1.3 倍から 1.6 倍向上させることができたが、安定性に欠け、戦略の工夫も加え改良の余地が有ること、などが分かった。

目 次

第 1 章　はじめに **1**
1.1　研究の目的と背景 1
1.2　論文構成 2

第 2 章　TSP と GA **3**
2.1　はじめに 3
2.2　TSP 3
2.3　GA 3
2.4　確率モデル GA 3

第 3 章　EHBSA アルゴリズム **5**
3.1　はじめに 5
3.2　概要 5
3.3　EHM 5
3.4　EHBSAWO 6
3.5　EHBSAWT 7
3.6　sampling のプログラム 7

第 4 章　実装した戦略 **8**
4.1　はじめに 8
4.2　全体の流れ 9
4.3　初期集団の選択 12
4.3.1　エリート戦略 12
4.3.2　ルーレット戦略 13
4.3.3　トーナメント戦略 13
4.4　template の選択 15
4.4.1　ランダム戦略 15
4.4.2　ルーレット戦略 15
4.4.3　エリート戦略 16
4.5　sampling node の選択 16
4.5.1　ランダム戦略 16
4.5.2　ルーレット戦略 17

4.6 EHM の戦略 19
4.6.1 重み無し 19
4.6.2 重みあり 20
4.7 局所的改善手法 21
4.8 送信における戦略 22
4.8.1 ランダム戦略 22
4.8.2 ルーレット戦略 23
4.8.3 エリート戦略 23
4.8.4 トーナメント戦略 24
4.9 受信における戦略 25
4.9.1 ランダム戦略 25
4.9.2 ルーレット戦略 25
4.9.3 エリート戦略 26
4.9.4 トーナメント戦略 26

第 5 章 実装したシステム **28**
5.1 はじめに 28
5.2 ehbsawo.ml 28
5.3 execute_ehbsawo.ml 31
5.4 read_ehbsawo.ml 31

第 6 章 実験結果 **35**
6.1 実験方針 35
6.2 個別の戦略評価 35
6.2.1 初期集団の選択 35
6.2.2 EHM 37
6.2.3 choose template , choose sampling node 38
6.2.4 2opt 40
6.2.5 まとめ 40
6.3 複数の戦略を組み合わせたときの評価 42
6.3.1 EHBSAWO 42
6.3.2 EHBSAWT 42
6.4 パラメータを変化させたときの評価 44
6.4.1 実行時間と cut 数 44
6.4.2 集団数、都市数、世代数 48
6.5 並列化させたときの評価 52
6.5.1 戦略 52
6.5.2 パラメータ 58

第 7 章　まとめと今後の課題 **66**
7.1 実験結果のまとめ 66
7.2 今後の課題 66

謝辞 **68**

参考文献 **69**

付 録 A パラメーター覧 **70**
A.1 EHBSAWO 70
A.1.1 指定できるパラメータ 70
A.1.2 サンプル 70
A.2 EHBSAWT 71
A.2.1 指定できるパラメータ 71
A.2.2 サンプル 71
A.3 MPIEHBSAWO 72
A.3.1 指定できるパラメータ 72
A.3.2 サンプル 72
A.4 MPIEHBSAWT 73
A.4.1 指定できるパラメータ 73
A.4.2 サンプル 73

付 録 B OCamlMPI **74**
B.1 はじめに 74
B.2 基本的な関数 74
B.3 基本的な使いかたと C との比較 75
B.4 送受信の例 81
B.4.1 リストの送受信の例 81
B.4.2 タプルの送受信の例 82
B.4.3 レコードの送受信の例 82
B.4.4 バリアントの送受信の例 83
B.4.5 インスタンスの送受信の例 83
B.4.6 インスタンスのリストの送受信の例 84
B.4.7 関数の送受信の例 85

図 目 次

5.1 system_graph1.eps 32
5.2 system_graph2.eps 32
5.3 system_graph3.eps 32

6.1 EHBSAWO における初期集団の選択 36
6.2 EHBSAWT における初期集団の選択 37
6.3 EHBSAWO の EHM 戦略 38
6.4 EHBSAWT の EHM 戦略 38
6.5 choose template, choose sampling node 39
6.6 EHBSAWO の 2opt 41
6.7 EHBSAWT の 2opt 41
6.8 EHBSAWO の複合戦略 1 43
6.9 EHBSAWO の複合戦略 2 43
6.10 EHBSAWT の複合戦略 1 45
6.11 EHBSAWT の複合戦略 2 45
6.12 EHBSAWO のパラメータ 1 49
6.13 EHBSAWT のパラメータ 1 50
6.14 EHBSAWO のパラメータ 2 50
6.15 EHBSAWT のパラメータ 2 51
6.16 EHBSAWO のパラメータ 3 51
6.17 EHBSAWT のパラメータ 3 52
6.18 EHBSAWO:都市数 200 53
6.19 EHBSAWT:都市数 200 53
6.20 EHBSAWO:都市数 400 54
6.21 EHBSAWT:都市数 400 54
6.22 EHBSAWO:都市数 600 55
6.23 EHBSAWT:都市数 600 55
6.24 EHBSAWO:都市数 800 56
6.25 EHBSAWT:都市数 800 56
6.26 EHBSAWO:都市数 1000 57
6.27 EHBSAWT:都市数 1000 57
6.28 MPIEHBSAWO の戦略 58

6.29 MPIEHBSAWT の戦略 59
6.30 MPIEHBSAWO のパラメータ 1 60
6.31 MPIEHBSAWT のパラメータ 1 60
6.32 MPIEHBSAWO のパラメータ 2 63
6.33 MPIEHBSAWO のパラメータ 3 63
6.34 MPIEHBSAWO のパラメータ 4 64
6.35 MPIEHBSAWO のパラメータ 5 64

表 目 次

6.1 cut 数と計測時間、および解の精度 1 47
6.2 cut 数と計測時間、および解の精度 2 47
6.3 MPIEHBSAWO の実験結果 1 62
6.4 MPIEHBSAWO の実験結果 2 65

第1章　はじめに

1.1　研究の目的と背景

近年、最適解を求めることの難しい問題について、生物学の知見を応用したアルゴリズムによってその近似解を求める手法が注目を集めている。このような問題の代表的なものとしてスケジューリング問題と巡回セールスマン問題 (TSP) があるが、本研究では TSP を取りあげる。

TSP の近似解を、生物学的なアルゴリズムを利用して解くことは広く研究されている。このようなアルゴリズムは大きく分けて、ニューラルネットを利用するものと遺伝的アルゴリズムを利用するものがある。前者は、行列を神経細胞の集合に見立てて、エネルギー計算をくり返すことによって行列がある状態に集束し、その状態が TSP における極小解を表していることを利用する。後者は、都市の配列をゲノムの集合と考え、これらの都市を順番にまわったときの距離の逆数を適応関数とし、これらの集合 (ゲノムの集合の集合) を作り、これ等に対して交叉と突然変異を起こさせ、世代を重ねていくことにより結果としてより良い解を探索する。

しかし、前者のホップフィールド型のニューラルネットは、数学的に集束が保証されているものの様々な問題がある。例えば、パラメータの定性的な性質が明らかでなく、アルゴリズムを改良しようとしたときどのようにパラメータをいじればよいのかが不明であること、並列化したとき集束せず同じ状態をくり返すことがあること、最適解とは程遠い極小解に陥る可能性が大きいことなどである (参考文献 [3])。

また、後者の突然変異と交叉演算を用いる通常の GA も TSP について応用しようとするといくつかの問題がある。例えば、交叉確率、交叉手法を変えたとき結果がどうなるかが人間にはイメージしにくい、というパラメータ調節の難しさに関する問題がある。また、突然変異を単純に行なった場合に、非常に近い都市間の関係を壊してしまう確率と非常に遠い都市間の間を変える確率が等しく、結果として上手く集束しなかったり集束しても非常に時間がかかるという問題もある。結局のところ、これらの問題は TSP に対するモデル化と生物が実際に行なっているアルゴリズムが非常に異なっていることに原因の大半がある。つまり、生物の進化のアルゴリズムは多様な環境に適応することに重点が置かれていて、都市を 1 回ずつ訪れるという非常に非生物学的な制約条件や、全ての都市を訪れる

ときの距離のみが適応度を示すという、生物が受けるのとは比べ物にならないくらい単純な選択圧と GA は相性が悪い。自然、TSP に GA を応用し突然変異や交叉演算を適用しようと思えば、人工的ななんらかの手法を併用しなくてはならない。これは結局パラメータの効果を複雑にし、高性能の GA のアルゴリズムの実現を難しくする (参考文献 [1])。

このような状況の中、生物のアルゴリズムの高い探索能力をコンピュータ上に実現する新たな手法として確率モデル GA と呼ばれるものが注目されている。これは、従来のような交叉オペレータにより子個体を生成せず、集団の個体分布を確率モデルで近似し、そのモデルに基づいて子個体を生成する。確率モデル GA としては、PBIL, compactGA, ECGA(extended compact GA)(参考文献 [5][6]) などがあるが、その適用はほとんど toy problem に限られ、従来の GA の代表的な応用であるスケジューリング問題や TSP についての応用はほとんど見られない。確率モデル GA に基づいた TSP についての数少ない応用としては EHBSA (参考文献 [4]) があるが、アルゴリズムが提案されているだけで十分に調べ尽くされているとは言い難い。確率モデル GA は従来の GA と同じように、集団数などの多くのパラメータや様々な戦略が存在し、これらの組み合わせによってその性能が大きく変化することを考えると、EHBSA における効果的な戦略、パラメータを調査することの意義は十分にあると考え、本研究を行なった。

計測は、SGI Altix 350 (Itanium2 × 8 , memory:40GB) 上で行ない、処理系に OCaml-3.08、拡張モジュールとして ocamlmpi-1.01 を用い、コンパイルは ocamlopt コマンドを使い行なった。また、計測は基本的に 1 回のみである。

1.2 論文構成

本論文の構成は次の通りである。

第 2 章 TSP と GA について簡単な説明をし、その後、確率モデル GA について自分の意見を述べる。

第 3 章 EHBSA のアルゴリズムについて述べる。

第 4 章 自分が実装した戦略について述べる。

第 5 章 自分が実装したシステムについて述べる。

第 6 章 実験結果について述べる。

第 7 章 実験結果のまとめと今後の課題についての考察について述べる。

第2章　TSPとGA

2.1　はじめに

本章では、本研究のテーマであるTSP(巡回セールスマン問題) とGA(遺伝的アルゴリズム) について、簡単に説明し、その後確率モデルGAについて自分の意見を交えながら述べる。

2.2　TSP

巡回セールスマン問題とは、セールスマンが与えられた都市すべてを1回ずつ訪問して、その経路を最短にしようという問題である。問題自体は単純なのだが、都市の数が増えると処理時間が爆発的に増えて、厳密解を求めることは実際上不可能になる。いわゆるNP完全問題として、ジョブショップスケジューリング問題 (JSP) と並んで有名な問題である。本研究では単純に乱数によって都市の位置を生成し、それらを全て訪問するように、という形でこの問題を調べた。

2.3　GA

GA(遺伝的アルゴリズム) とは、Darwinの進化論をそのままアルゴリズムにしたものだ。すなわち、与えられた環境下で、個体集団が交配と突然変異をくり返し、その自然集団に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ことで、主に厳密解を求めることの難しい問題の近似解を求める手法である。GAは様々な場所で利用されており、TSPの解法はその中でもJSPと並んでもっとも有名なもので、古くから研究されており、様々な仕事がなされている([1])。

2.4　確率モデルGA

2.3で述べたように、GAは突然変異と自然淘汰を基本としたアルゴリズムを直接応用し、計算機上に実装するが、これはいろいろな面で不自然な部分が有るように思える。

まず、Darwin の進化論による突然変異と自然淘汰は、外的要因によって適応できないものは淘汰され、適応できるもののみが子孫を残すことになる、という考え方だが、当然これは自然環境で純粋に生存できるかがを基準としているもので、そのままコンピュータに移そうとすると、戦略、パラメータの意味が人間には理解しにくい。また、当然通常の計算機上での計算量は概して多くなる。

また、実際の遺伝では遺伝子の交叉確率、突然変異確率は一律ではなく、遺伝子ごとに依存関係がある。これらを連鎖 (linkage) といい、遺伝子診断や遺伝子病の原因遺伝子の発見に利用される性質だがこれらのことも例えば、TSP に対する通常の GA の解法では考えることが困難である。

このようなことを考えると、生物のアルゴリズムの高い探索能力と柔軟性をそのままに、人間に理解しやすいよう、また計算機上に効率的に実装可能な手法として確率モデル GA の意味というのがあるのだと思う。このような確率モデル GA の中で実際の問題への適用が可能なものとして TSP に対する EHBSA が存在する。具体的な EHBSA のアルゴリズムの説明は第 3 章で述べる。

第3章　EHBSAアルゴリズム

3.1　はじめに

本章ではEHBSAの論文に基づいてEHBSAのアルゴリズムについて説明する。またsamplingを行なう関数のOCamlのソースを示す。

3.2　概要

EHBSA(edge histogram based sampling algorithm)は確率モデルGAのアルゴリズムの一種であり、順序問題に対する解を探索するのに用いる。確率モデルGAは、第2章で述べたように、集団の個体分布を確率モデルで近似するが、EHBSAではこれをEHM(edge histogram matrix)という行列で行なう。つまり、世代tの集団があったとき、これからEHM^tをつくり、これを用いてsamplingすることで、世代$t+1$の集団をつくる。これをくり返すことでより良い解を探索する。

EHBSAのアルゴリズムの簡単な流れは次のようになる。

1. 個体をランダムに生成
2. 選択オペレータを用い、有望集団をつくる
3. この集団からEHMを作成
4. このEHMを基に子個体をサンプリングする
5. 終了条件が満たされるまで、ステップ2から4をくり返す。

3.3　EHM

世代tの集団を$P(t)$、そのk番目の要素をs_k^tとする。s_k^tはN(=都市の数)の長さの順列であり、$s_k^t = (\pi_k^t(0), \pi_k^t(1), \ldots, \pi_k^t(N-1))$で表す。
このとき$EHM^t(e_{i,j}^t)(i,j = 0,1,2,\ldots,L-1)$の要素$e_{i,j}^t$は次のように表せる。

$$e_{i,j}^t = \begin{cases} \sum_{k=1}^{L}(\delta_{i,j}(s_k^t) + \delta_{j,i}(s_k^t)) + \epsilon & \text{if } i \neq j \\ 0 & \text{if } i = j \end{cases}$$

ただし、L は集団数、$\delta_{i,j}(s_k^t)$ は次の式で表されるデルタ関数。

$$\delta_{i,j}(s_k^t) = \begin{cases} 1 & \text{if } \exists h[h \in 0,1,...N-1 \wedge \pi_k^t(h) = i \wedge \pi_k^t((h+1) mod L) = j] \\ 0 & otherwise \end{cases}$$

ϵ は、パラメータ B_{ratio} によって次のように定義され、これはサンプリング圧を制御する。つまり、この値が大きい程ランダムにサンプリングされ、少ない程、EHM の内容どおりにサンプリングされる可能性が高まる。

$$\epsilon = \frac{2L}{N-1} B_{ratio}$$

これだけでは分かりにくいと思うので EHM の持つ意味について、少し補足すると $e_{i,j}^t$ は世代 t の集団の中に i、j が隣接しているものが多い程、大きくなる。世代 t の集団がそれまでに選択を受けて、優秀な個体が生き残っていることを考えると、その中に多く見られる特長はより良く適応するのに有用であると考えられる。よって EHM の要素 $e_{i,j}^t$ が他の要素の値より高ければ、i、j は隣接させておいた方が良いだろうと想像できる。EHBSA では、このような考えを基に EHM を用いて sampling する。

3.4 EHBSAWO

EHM を作成したら、それを基に sampling を行ない次世代の集団をつくる。EHBSAWO(EHBSA without template) では、次のようなアルゴリズムに基づいて sampling を行なう。sampling する集団の要素を c とする (c は 0 から $N-1$ までの順列)。

1. position counter: p を 0 にセット
2. c[0] を 0 から N-1 までからランダムに選ぶ
3. roulette wheel vector: rw[] を作成。ただし、$rw[j] = e_{c[p],j}^t (j = 0, 1, \ldots, N-1)$
4. すでに sampling されているノードを 0 にする。$(rw[c[i]] = 0 \, for \, i = 0, \ldots, p)$
5. sampling を行なう。つまり $\frac{rw[x]}{\sum_{j=0}^{N-1} rw[j]}$ の確率で $c[p+1] = x (x \in (y | rw[y] \neq 0))$ とする。
6. $p \leftarrow p+1$
7. $p < N-1$ ならば、ステップ 3 へ、そうでないなら終了

3.5 EHBSAWT

EHBSAWT は基本的には EHBSAWO と同じだが、一部分だけを sampling によって変更するところが違う。つまり、cut 数に template を分割し、その中から 1 つの部分だけ選び、それを sampling し、他の部分はそのままコピーする。

3.6 sampling のプログラム

```
let get_sampling_gene rw =
  let sum_of_rw = ref 0.0 in
    for i=1 to Array.length rw do
      sum_of_rw := !sum_of_rw +. rw.(i-1);
    done;
    let a = Random.float  !sum_of_rw in
    let sum1 = ref 0.0 and sum2 = ref rw.(0) in
    let i = ref 1 and b = ref 0.0 in
      while( not (!sum1 <= a && a < !sum2) ) do
        sum1 := !sum2;
        if !i = pred (Array.length rw) then sum2 := !sum_of_rw
        else sum2 := !sum2 +. rw.(!i);
        i := !i + 1;
      done;
      !i

let sampling newgene ehm =
  let len = Array.length newgene in
  let rw = Array.make len 0.0 in
    newgene.(0) <- (Random.int len) + 1;
    for i=1 to pred len do
      for j=0 to pred len do
        rw.(j) <- ehm.(newgene.(i-1)-1).(j);
      done;
      for j=0 to pred i do
        rw.(newgene.(j) - 1) <- 0.0;
      done;
      newgene.(i) <- get_sampling_gene rw;
    done
```

第4章　実装した戦略

4.1　はじめに

実装した、EHBSA の全体の流れは次のようになる。

EHBSA のアルゴリズム

1. t=0
2. 初期集団 $P(t)$ の生成
3. 初期集団の選択により、 $P(t) \rightarrow S(t)$ を生成
4. (a) EHBSAWO なら次へ進む
   (a) EHBSAWT なら template を選ぶ
   (b) cut 数のぶんだけ分割し、その中から sampling node を選ぶ。
5. EHM を作成
6. サンプリングにより、$S(t) \rightarrow P(t+1)$ を生成
7. 2opt などの局所的改善方を行なう
8. 並列版の場合、集団の中から一定の割合を交換
9. $t \leftarrow t+1$
10. $t \leq max\_generation$ ならばステップ 3 へ。そうでないなら終了

これらの各ステップにおいて、いくつかの戦略を実装した。簡単にふれる。

まず上のステップ 3 の初期集団の選択においては、まったく選択しない no_population_select、優れた上位半分を使う (つまり同じ内容のものが 2 つずつくられる) population_select_half、適応度の割合に応じて子孫をのこす population_select_roulette、ランダムに 2 つ選び、そのうちの優れた方が受け継がれ、それを集団の数だけくり返す population_select_tournament の 4 種類の戦略を実装した。

４の a のステップでは、ランダムに template を選ぶ choose_template_randomly、適応度に応じて選択する choose_template_roulette、最も良いものを template として選択する choose_template_elite の３種類を実装した。

４の b のステップでは、ランダムに sampling node を選ぶ choose_node_randomly、適応度に応じて選択する choose_node_roulette の２種類を実装した。

５のステップでは EHBSA の論文 ([4]) に基づいたデルタ関数を使う make_ehm_delta と、独自の重みづけを行なう EHM を作成する make_ehm_weight の２種類を実装した。

７のステップでは、2opt 法という従来の GA に使われているヒューリスティックスを使う use_twoopt と使わない non_use_twoopt の２種類を実装した。

８のステップでは送信用の戦略、受信用の戦略のそれぞれに対してランダムに送受信する send_randomly、receive_randomly、適応度に応じて送受信するものを選ぶ send_roulette、receive_roulette、上位の何個かを送る send_elite、自分の中で最も適応度の低いものと受信したものを取り換える receive_elite、ランダムに２つ選びそのうち適応度の高いものを送信する send_tournament、ランダムに１つ選び、それと受信したものを比較し、優れているときのみ交換する receive_tournament の計８種類を実装した。

なお、これらの工夫は通常の GA における戦略を参考にしている ([1][2])。本章では、これらの戦略の実装について OCaml のソースと共に説明する。

4.2 全体の流れ

これから個々の戦略について説明する前に、これらをどのような形で呼んでいるのかを示す。ソースは mpiehbsawt の場合だが、逐次版では、これに時間の計測が加わり、exchange_population の部分が抜ける。まだ他にも細かい差は存在するが、参考にはなると思う。

mpiehbsawt における execute のソース

```
let execute par ex_par population1 population2
  max_generation cut_num distance dis_total ehm city_pos =
  let population1_new = ref false in
  let pop_num = Array.length population1 in
  let min_distance_array = Array.make max_generation 0.0 in
  let copy_population genes1 genes2 index1 index2 =
    for i=index1 to index2 do
      genes2.(i) <- genes1.(i);
    done
  in
    for i=1 to max_generation do
      if par.a = Population_select_half then (
       let new_population = if !population1_new
         then population1 else population2 in
```

```
 let old_population = if !population1_new
   then population2 else population1 in
    select_population1 old_population
     new_population distance;
   population1_new := not (!population1_new);
) else if par.a = Population_select_roulette then (
   let new_population = if !population1_new
    then population1 else population2 in
   let old_population = if !population1_new
    then population2 else population1 in
    select_population2 old_population
     new_population distance;
    population1_new := not (!population1_new);
) else if par.a = Population_select_tournament then (
   let new_population = if !population1_new
    then population1 else population2 in
   let old_population = if !population1_new
    then population2 else population1 in
     select_population3 old_population
      new_population distance;
     population1_new := not (!population1_new);
) else if par.a = No_population_select then (
);

let a = ref 0 in
let new_population = if !population1_new
 then population1 else population2 in
let old_population = if !population1_new
 then population2 else population1 in
 (match par.b with
   Make_EHM_delta (b_ratio) -> (
    make_EHM1 old_population b_ratio ehm;
   )
  | Make_EHM_weight(b_ratio) -> (
   make_EHM2 old_population distance dis_total b_ratio ehm;
 ));

 if par.d = Choose_template_randomly then (
  a := choose_template1 old_population;
 ) else if par.d = Choose_template_roulette then (
  a := choose_template2 old_population distance;
 ) else raise ParameterError;

 for j=1 to pop_num do
  let b,c =
 if par.e = Choose_node_randomly then (
```

```
      choose_sampling_node1 old_population.(!a) cut_num;
     ) else if par.e = Choose_node_roulette then (
      choose_sampling_node2 old_population.(!a)
       distance cut_num;
     ) else raise ParameterError in

     copy_population old_population.(!a)
      new_population.(j-1) b c;


     (* check_ehm ehm distance; *)
     sampling new_population.(j-1) ehm b c;

     if par.c = Use_twoopt
      then twoopt new_population.(j-1) distance;

done;

if !population1_new then (
  exchange_population population1 distance ex_par;
) else (
  exchange_population population2 distance ex_par;
);


if !population1_new then (
 let min_dist = reduce_float (min_distance population1 distance)
     Float_min 0 comm_world in
   if myrank=0 then (
     min_distance_array.(i-1) <- min_dist;
     printf "generation:%d ,
        min_distance:%f\n" i min_dist;flush stdout;
   );
) else (
 let min_dist = reduce_float (min_distance population2 distance)
     Float_min 0 comm_world in
   if myrank=0 then (
     min_distance_array.(i-1) <- min_dist;
     printf "generation:%d ,
        min_distance:%f\n" i min_dist;flush stdout;
   );
 );

 population1_new := not (!population1_new);

done;
```

```
  if myrank=0 then
   (printf "\n************** execute end **************\n\n");

    match par.f with
      Record(filename) ->
    (
     if myrank=0 then (
       write_state filename
        (if !population1_new then population2 else population1)
        city_pos distance min_distance_array par ex_par;
     );
    )
      | Non_record -> ()
```

4.3 初期集団の選択

4.3.1 エリート戦略

説明

初期集団の選択法のひとつ。世代 t の初期集団 $P(t)$ から選択された集団 $S(t)$ を作るときに、集団の要素の上位半分を、それぞれ２個ずつつくる。評価は距離の逆数によって行なっている。計算量のオーダは集団数 L、都市数 N とすると $L \times N$ に比例する。

ソース

```
let select_population1 population1 population2 distance =
  let copy_population genes1 genes2 =
    for i=1 to Array.length genes1 do
      genes2.(i-1) <- genes1.(i-1);
    done in
  let fitness = Array.make (Array.length population1) (0.0,0) in
  let pop_num = Array.length population1 and
      city_num = Array.length population1.(0) in
    for i=0 to pred pop_num do
      fitness.(i) <-
        (1.0 /. (get_distance population1.(i) distance),i);
    done;
    Array.sort (fun a b-> let a1,a2 = a and
      b1,b2 = b in if a1>=b1 then ~-1 else 1) fitness;
    for i=0 to (pred pop_num) / 2 do
      let a,b = fitness.(i) in
```

```
        copy_population population1.(b) population2.(2*i);
        copy_population population1.(b) population2.(2*i+1);
    done
```

4.3.2 ルーレット戦略

説明

初期集団の選択法のひとつ。世代 t の初期集団 $P(t)$ から選択された集団 $S(t)$ を作るときに、適応度に応じた割合で選択する。つまり、集団の i 番目の要素の適応度を f_i とすると、$\frac{f_i}{\sum_{k=1}^{N} f_k}$ の確率で選択される。

ソース

```
let select_population2 population1 population2 distance =
  let copy_population genes1 genes2 =
    for i=1 to Array.length genes1 do
      genes2.(i-1) <- genes1.(i-1);
    done in
  let pop_num = Array.length population1 in
  let city_num = Array.length population1.(0) in
  let sum_fitness = ref 0.0 in
  let fitness = Array.make pop_num 0.0 in
    for i=0 to pred pop_num do
      fitness.(i) <-
        1.0 /. (get_distance population1.(i) distance);
      sum_fitness := !sum_fitness +. fitness.(i);
    done;
    for j=0 to pred pop_num do
      let a = (Random.float 1.0) *. !sum_fitness in
      let sum = ref 0.0 and i = ref 0 in
        while( !sum < a && !i < pop_num ) do
          sum := !sum +. fitness.(!i);
          i := !i + 1;
        done;
        copy_population population1.(!i-1) population2.(j);
    done
```

4.3.3 トーナメント戦略

説明

初期集団の選択法のひとつ。世代 t の初期集団 $P(t)$ から選択された集団 $S(t)$ を作るときに、ランダムに 2 つ選び、その 2 つのうち適応度の高いものを選択す

る。これを集団の数 L の分だけくり返す。プログラムの中では計算量を適応度を一度この関数内で計算したものはハッシュに格納し、計算していないもののみを計算するようにしている。

実行時間の観点からいえば、他の戦略において適応度をその都度計算している部分は改善の余地があるような気がするが、関数の依存関係を少なくし、可読性を高める観点から無理に計算量を減らそうとはしていない。

ソース

```
let select_population3 population1 population2 distance =
  let copy_population genes1 genes2 =
    for i=1 to Array.length genes1 do
      genes2.(i-1) <- genes1.(i-1);
    done in
  let pop_num = Array.length population1 in
  let city_num = Array.length population1.(0) in
  let dist_hash = Hashtbl.create pop_num in
   for i=0 to pred pop_num do
    let select_population_index1 = Random.int pop_num in
    let select_population_index2 = Random.int pop_num in
    let distance1 =
     (try
       Hashtbl.find dist_hash select_population_index1
      with
       Not_found -> (
        let tmp_dist =
         (get_distance
           population1.(select_population_index1) distance) in
          Hashtbl.add dist_hash select_population_index1 tmp_dist;
          tmp_dist )) in
    let distance2 =
     (try
       Hashtbl.find dist_hash select_population_index2
      with
       Not_found -> (
        let tmp_dist =
         (get_distance
           population1.(select_population_index2) distance) in
          Hashtbl.add dist_hash select_population_index2 tmp_dist;
          tmp_dist)) in
         if distance1 >= distance2
         then
          copy_population
           population1.(select_population_index2) population2.(i)
         else
```

```
        copy_population
         population1.(select_population_index1) population2.(i);
  done
```

4.4 template の選択

4.4.1 ランダム戦略

説明

EHBSAWT のアルゴリズムにおいて template を集団の中から選ぶときに関する戦略のひとつ。当然、EHBSAWO では存在しない。集団の中からランダムにひとつ選び、それを template として、sampling を行なう。

ソース

```
let choose_template1 population =
  Random.int (Array.length population)
```

4.4.2 ルーレット戦略

説明

EHBSAWT のアルゴリズムにおいて template を集団の中から選ぶときに関する戦略のひとつ。当然、EHBSAWO では存在しない。適応度に応じて、集団の中から template を選ぶ。基本的には初期集団の選択におけるルーレット戦略と同様である。

ソース

```
let choose_template2 population distance =
  let pop_num = Array.length population in
  let city_num = Array.length population.(0) in
  let sum_fitness = ref 0.0 in
  let fitness = Array.make pop_num 0.0 in
    for i=0 to pred pop_num do
      fitness.(i) <-
        1.0 /. (get_distance population.(i) distance);
      sum_fitness := !sum_fitness +. fitness.(i);
    done;
    let a = (Random.float 1.0) *. !sum_fitness in
    let sum = ref 0.0 and i = ref 0 in
      while ( !sum < a && !i < pop_num ) do
```

```
      sum := !sum +. fitness.(!i);
      i := !i + 1;
    done;
    !i-1
```

4.4.3 エリート戦略

説明

EHBSAWT のアルゴリズムにおいて template を集団の中から選ぶときに関する戦略のひとつ。当然、EHBSAWO では存在しない。最も、適応度の高いものを template として選択する。

ソース

```
let choose_template3 population distance =
  let pop_num = Array.length population in
  let city_num = Array.length population.(0) in
  let min_distance = ref (get_distance population.(0) distance) in
  let min_index = ref 0 in
    for i=1 to pred pop_num do
      let a = get_distance population.(i) distance in
       if !min_distance > a then (
          min_distance := a;
          min_index := i;
       );
    done;
    !min_index
```

4.5 sampling node の選択

4.5.1 ランダム戦略

説明

EHBSAWT のアルゴリズムでは、template を cut 数の数だけ分割し、その中からひとつ選び、それだけを sampling する。このときに sampling する node をランダムに選ぶ戦略。

プログラムの説明を簡単にする。配列 index を作成し、ここに各 cut の添字情報を記録している。つまり、最初の cut は genes.(index.(0)) から genes.(index.(1)) までとなり、次の cut は genes.(index.(2)(=index.(1)+1)) から genes.(index.(3)) となる。このとき cut の位置は重複してはいけないので、cut の位置として適切なものをつなげたリストを作成し、その中からランダムに要素を選び、それが選択されたらその要素をリストから削除するというアルゴリズムで本戦略を実装した。

ソース

```
let choose_sampling_node1 genes cut_num =
  let delete_list element list_ref =
    let rec sub_fun1 element list =
      if list = [] then []
      else if (List.hd list) = element then
       sub_fun1 element (List.tl list)
      else
       (List.hd list) :: (sub_fun1 element (List.tl list)) in
        list_ref := (sub_fun1 element !list_ref)
  in
  let len = Array.length genes in
  let index = Array.make (cut_num*2) ~-1 in
  let remain = ref [] in
    if cut_num * 2 > len then raise Cut_Num_Error;
    index.(0) <- 0;
    index.(pred (Array.length index))
     <- pred (Array.length genes);
    for i=1 to (Array.length genes)-3 do
      remain := i :: !remain;
    done;
    for i=0 to pred (cut_num-1) do
      let a = List.nth !remain
        (Random.int (List.length !remain)) in
        delete_list a remain;
        delete_list (a+1) remain;
        if (a-1) > 0 then delete_list (a-1) remain;
        index.(2 * i + 1) <- a;
        index.(2 * i + 2) <- a+1;
    done;

    Array.sort (fun x y-> if x>=y then 1 else ~-1) index;

    let a = Random.int cut_num in
      (index.(2 * a),index.(2 * a + 1))
```

4.5.2 ルーレット戦略

説明

EHBSAWT のアルゴリズムでは、template を cut 数の数だけ分割し、その中からひとつ選び、それだけを sampling する。sampling されない部分は次の世代でも変わらずに存在するため、sampling する場所はできるだけ改善の余地のある部分の方が良いのではないかという考えを基に実装した。各 cut において隣接する 2 つの要素の距離の平均を使い、ルーレット戦略を行なう戦略。

ソース

```
let choose_sampling_node2 genes distance cut_num =
  let delete_list element list_ref =
    let rec sub_fun1 element list =
      if list = [] then []
      else if (List.hd list) = element then
       sub_fun1 element (List.tl list)
      else
       (List.hd list) :: (sub_fun1 element (List.tl list)) in
        list_ref := (sub_fun1 element !list_ref)
  in
  let len = Array.length genes in
  let index = Array.make (cut_num*2) ~-1 in
  let remain = ref [] in
    if cut_num * 2 > len then raise Cut_Num_Error;
    index.(0) <- 0;
    index.(pred (Array.length index))
      <- pred (Array.length genes);
    for i=1 to (Array.length genes)-3 do
      remain := i :: !remain;
    done;
    for i=0 to pred (cut_num-1) do
      let a = List.nth !remain
        (Random.int (List.length !remain)) in
        delete_list a remain;
        delete_list (a+1) remain;
        if (a-1) > 0 then delete_list (a-1) remain;
        index.(2 * i + 1) <- a;
        index.(2 * i + 2) <- a+1;
    done;

    Array.sort (fun x y-> if x>=y then 1 else ~-1) index;

    let sum_dist = ref 0.0 and
        dist_array = Array.make cut_num 0.0 in
    let cut_index = ref 0 in
      for i=0 to pred cut_num do
       for j= index.(2*i) to pred index.(2*i+1) do
        let dis = distance.(genes.(j)-1).(genes.(j+1)-1) in
         dist_array.(i) <- dist_array.(i) +. dis;
       done;
      done;
      for i=0 to pred (Array.length dist_array) do
       (* normalization *)
       dist_array.(i) <- dist_array.(i) /.
```

```
      (float_of_int (index.(2*i+1) - index.(2*i) + 1));
      sum_dist := !sum_dist +. dist_array.(i);
    done;

    (* neglect distance between cuts *)

    (* select roulette *)
    let a = Random.float !sum_dist in
    let sum1 = ref 0.0 and sum2 = ref dist_array.(0) in
    let i = ref 1 in
      while ( not ( !sum1 <= a && a < !sum2) ) do
        sum1 := !sum2;
        sum2 := !sum2 +. dist_array.(!i);
        i := !i + 1;
      done;
      ( index.((!i-1)*2),index.((!i-1)*2 + 1) )
```

4.6 EHMの戦略

4.6.1 重み無し

説明

delta関数を使って、集団からEHM(edge histogram matrix)を作成する。この際b_ratioというパラメータが存在するが、本研究ではほとんど調べていない。

ソース

```
let make_EHM1 population b_ratio ehm =
  let pop_num = Array.length population in
  let city_num = Array.length population.(0) in
  let floatN = float_of_int pop_num and
      floatL = float_of_int city_num in
  let ehm_parameter = 2.0 *. floatN /.
      (floatL -. 1.0) *. b_ratio in
    for i=0 to pred city_num do
      for j=0 to pred city_num do
       if i <> j then ehm.(i).(j) <- ehm_parameter;
      done;
    done;
    for i=0 to pred pop_num do
      for j=0 to pred city_num do
        if i = j then ()
        else if j = (pred city_num) then (
```

```
        let a = population.(i).(j) and
            b = population.(i).(0) in
          ehm.(a-1).(b-1) <- ehm.(a-1).(b-1) +. 1.0;
          ehm.(b-1).(a-1) <- ehm.(b-1).(a-1) +. 1.0;
       ) else (
          let a = population.(i).(j) and
              b = population.(i).(j+1) in
           ehm.(a-1).(b-1) <- ehm.(a-1).(b-1) +. 1.0;
           ehm.(b-1).(a-1) <- ehm.(b-1).(a-1) +. 1.0;
       );
     done;
   done
```

4.6.2 重みあり

説明

集団の中に隣接している 2 つの要素が存在したとき、その距離の逆数を EHM に足す。この際、全ての都市間の距離の総和を定数項として掛けている。

ソース

```
let make_EHM2 population distance dis_total b_ratio ehm =
  let pop_num = Array.length population in
  let city_num = Array.length population.(0) in
  let floatN = float_of_int pop_num
  and floatL = float_of_int city_num in
  let ehm_parameter = 2.0 *. floatN /.
    (floatL -. 1.0) *. b_ratio in
    for i=0 to pred city_num do
      for j=0 to pred city_num do
       if i <> j then ehm.(i).(j) <- ehm_parameter
       else ehm.(i).(i) <- 0.0;
      done;
    done;
    for i=0 to pred pop_num do
      for j=0 to pred city_num do
       if j=(pred city_num) then (
        let a = population.(i).(j) and
            b = population.(i).(0) in
        let c = dis_total /. distance.(a-1).(b-1) in
          ehm.(a-1).(b-1) <- c;
          ehm.(b-1).(a-1) <- c;
       ) else (
        let a = population.(i).(j) and
```

```
            b = population.(i).(j+1) in
        let c = dis_total /. distance.(a-1).(b-1) in
          ehm.(a-1).(b-1) <- c;
          ehm.(b-1).(a-1) <- c;
       )
      done;
    done
```

4.7 局所的改善手法

説明

従来の GA による TSP 解法のときに用いられる、2opt 法と呼ばれる局所的改善手法を使う戦略。改善できるときは、改善し、もしも改善できないときは何もしない。

ソース

```
let reverse genes i1 i2 =
  let a = ref 0 and j1 = ref i1 and j2 = ref i2 in
    while !j1 < !j2 do
      a := genes.(!j1);
      genes.(!j1) <- genes.(!j2);
      genes.(!j2) <- !a;
      j1 := !j1 + 1;
      j2 := !j2 - 1;
    done

let twoopt newgenes distance =
  let i1 = ref 0 and i2 = ref 0 and
      j1 = ref 0 and j2 = ref 0 in
  let len = Array.length newgenes in
    i1 := (Random.int len);
    j2 := if (!i1 + 1) >= len then 0 else (!i1 + 1);
    j1 := (Random.int len);
    while (!j1 = !i1 || !j1 = !j2) do
      j1 := (Random.int len)
    done;
    if (!j1 > !j2) then (
      i2 := !j1 + 1;
      if (!i2 >= len) then i2 := 0;
    )else(
      i2 := !j1;
      j1 := !j1 - 1;
```

```
    if (!j1 < 0) then j1 := len - 1;
  );
  if (distance.(newgenes.(!i1) - 1).(newgenes.(!j2) - 1) +.
      distance.(newgenes.(!j1) - 1).(newgenes.(!i2) - 1)
      > distance.(newgenes.(!i1) - 1).(newgenes.(!j1) - 1) +.
      distance.(newgenes.(!i2) - 1).(newgenes.(!j2) - 1) )
  then (
    if ( !i1 < !j2 && !j1 < !i2) then (
     if (!j2 < !j1)
     then reverse newgenes !j2 !j1
      else reverse newgenes !i2 !i1
    ) else (
    if ( not (!i1 < !j2) )
      then reverse newgenes !j1 !i2
      else reverse newgenes !i1 !j2
    )
  )
```

4.8 送信における戦略

4.8.1 ランダム戦略

説明

並列版の実装において、他のプロセスに集団の中からなにを送るかをランダム選ぶ戦略。パラメータとしていくつの要素を送るかを示す send_num をとる。要素数 send_num の配列を返す。

ソース

```
let get_send_population_index1 population send_num =
  let sub_func1 population =
    Random.int (Array.length population) in
  let send_index = Array.make send_num 0 in
    for i=0 to pred send_num do
      send_index.(i) <- (sub_func1 population);
    done;
    send_index
```

4.8.2 ルーレット戦略

説明

並列版の実装において、他のプロセスに集団の中からなにを送るかをルーレット戦略に基づいて選ぶ戦略。パラメータとしていくつの要素を送るかを示す send_num をとる。要素数 send_num の配列を返す。

ソース

```
let get_send_population_index2 population distance send_num =
  let sub_func1 population distance =
    let pop_num = Array.length population in
    let city_num = Array.length population.(0) in
    let fitness = Array.make pop_num 0.0 in
    let sum_fitness = ref 0.0 in
      for i=0 to pred pop_num do
        fitness.(i) <-
         1.0 /. (get_distance population.(i) distance);
        sum_fitness := !sum_fitness +. fitness.(i);
      done;
      let a = Random.float !sum_fitness in
      let sum1 = ref 0.0 and sum2 = ref fitness.(0) in
      let i = ref 1 in
       while ( not ( !sum1 <= a && a < !sum2) ) do
        sum1 := !sum2;
        sum2 := !sum2 +. fitness.(!i);
        i := !i + 1;
       done;
       !i - 1 in
       let send_index = Array.make send_num 0 in
        for i=0 to pred send_num do
         send_index.(i) <- (sub_func1 population distance);
        done;
        send_index
```

4.8.3 エリート戦略

説明

並列版の実装において、自分の集団の中の上位 send_num 個の要素を他のプロセスに送る戦略。プログラムとしては集団の各要素の適応度を求め、これを index と共に記録する。この配列を適応度に応じてソートし、上位 send_num の分の index を返り値に代入すし、これを返す。

ソース

```
let get_send_population_index3 population distance send_num =
  let pop_num = Array.length population in
  let city_num = Array.length population.(0) in
  let fitness = Array.make pop_num (0.0,0) in
  let return_index_array = Array.make send_num 0 in
    if pop_num < send_num then raise Send_Num_Error;
    for i=0 to pred pop_num do
      fitness.(i) <-
       (1.0 /. (get_distance population.(i) distance),i);
    done;
    Array.sort (fun a b -> let a1,a2 = a and
                b1,b2 = b in if a1>=b1 then ~-1 else 1) fitness;
    for i=0 to pred send_num do
      let a,b = fitness.(i) in
       return_index_array.(i) <- b;
    done;
    return_index_array
```

4.8.4 トーナメント戦略

説明

並列版の実装において、他のプロセスに集団の中からなにを送るかをトーナメント戦略に基づいて選ぶ戦略。つまり自分の集団の要素をランダムにふたつ選び、これらのより適応度の高いものを配列に記録。これを send_num の数だけくり返す。

ソース

```
let get_send_population_index4 population distance send_num =
  let return_index_array = Array.make send_num 0 in
  let pop_num = Array.length population in
    for i=0 to pred send_num do
      let a = Random.int pop_num and b = Random.int pop_num in
      let dis1 = get_distance population.(a) distance in
      let dis2 = get_distance population.(b) distance in
      if dis1 >= dis2 then return_index_array.(i) <- b
      else return_index_array.(i) <- a;
    done;
    return_index_array
```

4.9 受信における戦略

4.9.1 ランダム戦略

説明

並列版の実装において、他のプロセスから送られてきたものを自分の集団の中のランダムの要素と交換する戦略。

ソース

```
let receive_population1 population genes =
  let pop_num = Array.length population in
  let city_num = Array.length population.(0) in
  let a = Random.int pop_num in
    for i=0 to pred city_num do
      population.(a).(i) <- genes.(i);
    done
```

4.9.2 ルーレット戦略

説明

並列版の実装において、他のプロセスから送られてきたものをルーレット戦略に基づいて交換する戦略。

ソース

```
let receive_population2 population genes distance =
  let pop_num = Array.length population in
  let city_num = Array.length population.(0) in
  let dist_array = Array.make pop_num 0.0 in
  let dis_total = ref 0.0 in
    for i=0 to pred pop_num do
      dist_array.(i) <- (get_distance population.(i) distance);
      dis_total := !dis_total +. dist_array.(i);
    done;
    let a = Random.float !dis_total in
    let sum1 = ref 0.0 and sum2 = ref dist_array.(0) in
    let i = ref 1 in
      while ( not ( !sum1 <= a && a < !sum2 ) ) do
       sum1 := !sum2;
       sum2 := !sum2 +. dist_array.(!i);
       i := !i + 1;
```

```
    done;
    (* population.(!i-1) is selected *)
    for j=0 to pred city_num do
     population.(!i-1).(j) <- genes.(j);
    done
```

4.9.3 エリート戦略

説明

並列版の実装において、他のプロセスから送られてきたものを自分の集団の中で最も適応度の低いものと交換する戦略。

ソース

```
let receive_population3 population genes distance =
  let pop_num = Array.length population in
  let city_num = Array.length population.(0) in
  let dis_max_index = ref 0 in
  let dis_max = ref (get_distance population.(0) distance) in
    for i=1 to pred pop_num do
      let dis = get_distance population.(i) distance in
       if dis > !dis_max then (
        dis_max := dis;
        dis_max_index := i;
       );
    done;
    for i=0 to pred city_num do
      population.(!dis_max_index).(i) <- genes.(i);
    done
```

4.9.4 トーナメント戦略

説明

並列版の実装において、他のプロセスから送られてきたものをトーナメント戦略に基づいて交換する戦略。つまり、ランダムに自分の集団の中から 1 つの要素を選び、それと受け取った要素を比較。もし、受け取った要素の適応度の方が高ければ、要素を交換、そうでなければ何もしない。

ソース

```
let receive_population4 population genes distance =
  let pop_num = Array.length population in
  let city_num = Array.length population.(0) in
  let a = Random.int pop_num in
  let dis1 = get_distance population.(a) distance in
  let dis2 = get_distance genes distance in
    if dis1 > dis2 then
      for i=0 to pred city_num do
       population.(a).(i) <- genes.(i);
      done
```

第5章　実装したシステム

5.1　はじめに

4章の説明で分かるようにGAには多数の戦略、パラメータが存在する。これら全てをいちいちコマンドラインオプションなどで指定して、その結果をそのつどグラフにして、それをもとにまたコマンドラインオプションで戦略、パラメータを指定して実行、というようなことをしていては、本研究の目的である適切な戦略、パラメータの調査という目的はとても達成できない。そこで、本研究ではEHBSAWO、EHBSAWT、MPIEHBSAWO、MPIEHBSAWTの各アルゴリズムに対してそれぞれ3種類のプログラムを作成し、計測、分析の効率を高めた。本章ではこの内容について説明する。なお、以下の説明では簡単のためEHBSAWOを例にして説明するが、基本的に他のEHBSAWT、MPIEHBSAWO、MPIEHBSAWTも同様である。

5.2　ehbsawo.ml

ehbsawo.mlというプログラムはパラメータファイルから戦略や初期条件などのパラメータを読み、実行する。

例えば次のようなパラメータファイルehbsawo_parameter1があったとする。

ehbsawo_parameter1の内容

```
population_select:no_population_select
make_ehm_strategy:make_ehm_delta:0.04
use_twoopt:non_use_twoopt
record_final_state:non_record
```

コマンドラインから次のように打ち込むことで、その戦略、パラメータに基づいたGAが実行される。なお、第２、第３、第４引数はそれぞれ、集団数、都市数、最大の世代数を表している。

```
ocamlopt str.cmxa ehbsawo.ml
./a.out get_parameter_from_file 30 30 30 ehbsawo_parameter1
```

この実行結果は次のようになる。

```
city number : 30
population number : 30
max generation : 30
population selecct strategy : no population select
make EHM strategy : make EHM delta
use twoopt strategy : non use twoopt

generation:1 , min_distance:4108.858416 , time:0.000000
generation:2 , min_distance:3766.920148 , time:0.010000
generation:3 , min_distance:3813.282691 , time:0.010000
generation:4 , min_distance:3595.132699 , time:0.010000

        /* 省略 */

generation:27 , min_distance:3856.722093 , time:0.080000
generation:28 , min_distance:3891.668244 , time:0.080000
generation:29 , min_distance:3969.077524 , time:0.080000
generation:30 , min_distance:4113.855390 , time:0.080000

************** execute end **************
```

結果をファイルに記録しておくことも可能で、result というファイルに結果を格納したいときはパラメータファイルを次のようにする。

ehbsawo_parameter2 の内容

```
population_select:no_population_select
make_ehm_strategy:make_ehm_delta:0.04
use_twoopt:non_use_twoopt
record_final_state:record:result
```

こうすると実行した後に、result というファイル名が出来ている。この内容は次のような感じになる。なお、各戦略のかかった時間と、population の記録は遂次版のみ実装した。

result の内容

```
population_number:30
city_number:30
total time:0.070000
generation change mean time:0.002333
min_distance: population(27):3720.962301
algorithm:
population select strategy: no population select:
 mean time=0.000000
make EHM strategy: make EHM delta: delta=0.040000,
 mean time=0.030000
twoopt strategy: non use twoopt: mean time=0.000000
min_distance:
generation:1 , min_distance:3691.558133
generation:2 , min_distance:3788.393055
generation:3 , min_distance:4119.590014
 /*  省略  */
generation:27 , min_distance:3819.663246
generation:28 , min_distance:3378.718650
generation:29 , min_distance:3612.802789
generation:30 , min_distance:3720.962301
city_position:
275.227295,297.223719
159.199218,14.568445
293.090348,45.535741
111.211363,20.637708
144.457566,117.711827
 /*  省略  */
78.386624,298.653678
19.419848,99.720997
population:
population(0):distance=4254.404667
3,17,1,9,24,7,8,6,11,25,2,21,4,23,10,13,12,19,
18,20,29,22,27,30,26,15,16,28,14,5,
population(1):distance=4248.203993
26,21,13,29,7,28,27,19,15,8,9,16,30,14,18,4,12,
2,3,11,17,23,10,24,5,20,6,25,22,1,
/*  省略  */
population(28):distance=4331.170656
16,28,4,2,5,23,18,29,7,8,13,24,14,6,11,3,22,25,19,
15,20,27,30,21,17,10,26,9,1,12,
population(29):distance=3935.545140
17,3,15,11,20,22,18,23,2,5,9,29,21,12,1,24,14,8,
10,13,28,4,16,26,30,7,27,25,19,6,
```

5.3 execute_ehbsawo.ml

execute_ehbsawo.ml はファイルを読みこみ、各行のパラメータを要素とし、その直積集合を求め、その要素の数の分だけ ehbsawo.ml を実行する。その際にはある規則性をもってサブディレクトリ、ファイル名を指定して結果をファイルに格納する。

例えば次のような、makeparameter というファイルによって execute_ehbsawo.ml を実行したとする。

makeparameter の内容

```
pop_num;20,30,40
city_num;50
max_generation;30,40
population_select;no_population_select,population_select_half
make_ehm_strategy;make_ehm_weight:0.04
use_twoopt;use_twoopt,non_use_twoopt
```

具体的な実行の仕方は次のようなかたちになる。

```
ocaml str.cma unix.cma execute_ehbsawo.ml makeparameter
```

この結果実行される ehbsawo.ml の回数は $3 \times 2 \times 2 \times 2 = 24$ 回で、ehbsawo20_50_30、ehbsawo20_50_40、ehbsawo30_50_30、ehbsawo30_50_40、ehbsawo40_50_30、ehbsawo40_50_40 の 6 つのディレクトリがつくられる。これらのディレクトリ名は実行したアルゴリズムの種類、集団数、都市数、最大の世代数を表している。これらのサブディレクトリの中の例えば、result121 などというファイルに実験結果が格納される。result の後の数字は選択された戦略を表していて、この場合、それぞれ no_population_select、make_ehm_weight、use_twoopt が選択されたことを意味している。指定できるパラメータについては付録を参照してほしい。

5.4 read_ehbsawo.ml

execute_ehbsawo.ml によって作成されたデータはそのままではただのテキストデータなので、グラフなど人間に分かりやすい形式に直す必要がある。これを行なうのが、read_ehbsawo.ml になる。read_ehbsawo.ml は execute_ehbsawo.ml と同形式のパラメータファイルを受け取り、横軸に世代数、縦軸に集団の中の最小距離をとったグラフを gnuplot により作成する。まず、例を挙げる。例えば、前節の例で実行した結果を graph というファイル名でグラフ化したいとする。この場合、次のように入力する。

```
ocaml str.cma unix.cma read_ehbsawo.ml graph makeparameter
```

このようにすると、graph.dat、graph.eps、graph.png、graph.gp の４種類のファイルが出力される。このうち、eps、png 形式が実際のグラフで、拡張子が gp のものが gnuplot のスクリプトファイルで、拡張子が dat のものがプロットしたデータのデータファイルになる。

パラメータファイルは、それが一度実行されていることを、つまり結果を格納したファイルが存在することを前提としているが、それがいつ、どのようなかたちで実行されたかは問わないので、例えば、上の例で、世代数を一定にしたときのグラフがほしければ、次のような２つのパラメータファイル sub_makeparameter1、sub_makeparameter2 を作り、２回 read_ehbsawo.ml を実行すれば良い。makeparameter、sub_makeparameter1、sub_makeparameter2 で作ったグラフはそれぞれ、図 5.1、図 5.2、図 5.3 である。

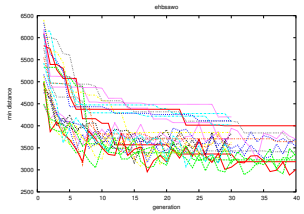

図 5.1: system_graph1.eps

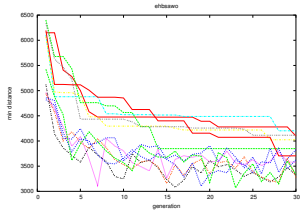

図 5.2: system_graph2.eps

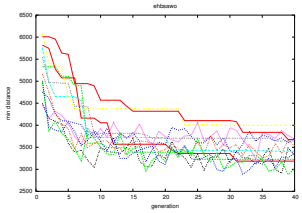

図 5.3: system_graph3.eps

sub_makeparameter1 の内容

```
pop_num;20,30,40
city_num;50
max_generation;30
population_select;no_population_select,population_select_half
make_ehm_strategy;make_ehm_weight:0.04
use_twoopt;use_twoopt,non_use_twoopt
```

sub_makeparameter2 の内容

```
pop_num;20,30,40
city_num;50
max_generation;40
population_select;no_population_select,population_select_half
make_ehm_strategy;make_ehm_weight:0.04
use_twoopt;use_twoopt,non_use_twoopt
```

実際のグラフ化の方法

```
ocaml unix.cma str.cma read_ehbsawo.ml system_graph1
  makeparameter
ocaml unix.cma str.cma read_ehbsawo.ml system_graph2
  sub_makeparameter1
ocaml unix.cma str.cma read_ehbsawo.ml system_graph3
  sub_makeparameter2
```

拡張子が dat のファイルについて補足すると、execute_ehbsawo.ml は指定されたパラメータに応じてファイルを上書きしてしまうので、実験をくり返すとプロットデータが消えてしまう。このような仕様は将来なんらかの不都合を生じさせるような気がしたので、データのバックアップという意味で dat 形式のファイルを作成するようにした。

これは、ディレクトリとファイル名をつなげた文字列の後に各世代の最小の距離がはいっているだけの次のようなテキストファイルなので、グラフを見てなにか調べたいことがあれば、awk などのツールを利用して簡単に調べることが出来る。

system_graph1.dat の内容

```
plotdata_ehbsawo20_50_30_result121
1 4856.678892
2 4802.995254
3 4170.658105
4 3783.815914
5 4042.554403
6 4003.361729
7 3723.309786
8 3564.487545
9 3544.818842
  /*  省略  */
28 3147.709878
29 3602.794195
30 3339.144686

plotdata_ehbsawo20_50_30_result221
1 6195.930750
2 5123.658308
  /*  省略  */
```

例えば、グラフを見て距離が 3000 以下になっている戦略が一体なんなのかを手軽に調べたいとする。これには単純に次のように打ち込めば良い。

距離が 3000 以下のものを表示

```
awk '$2 < 3000' system_graph1.dat
plotdata_ehbsawo20_50_30_result121

plotdata_ehbsawo20_50_30_result122

plotdata_ehbsawo20_50_30_result221

plotdata_ehbsawo20_50_30_result222

plotdata_ehbsawo20_50_40_result121
28 2989.230889
29 2953.186065
32 2890.580643
33 2989.199151

plotdata_ehbsawo20_50_40_result122
17 2954.878127
36 2961.944815
37 2992.673868
39 2882.762760

plotdata_ehbsawo20_50_40_result221

 /*  省略  */
```

第6章　実験結果

6.1　実験方針

第4章で述べた戦略と都市数などのパラメータをしらみ潰しに計測しては、意味ある結果をそこから抽出するのは難しい。そこで次のような基本方針をたて、計測する。

1. まず戦略を中心に調べる。そこである戦略が他の戦略よりも明らかに優れていたらそれを固定して他の戦略を調べる。

2. まず逐次について調べる。その次に逐次で調べた結果を活用し、有効な並列GAの戦略、パラメータをさぐる。

6.2　個別の戦略評価

まずは、個別の戦略について調べる。つまり、性能に影響を与えそうな戦略を一種類にしぼり解の精度を調べる。

6.2.1　初期集団の選択

初期集団の戦略についてを調べる。結果は図6.1、6.2である。

図6.1、6.2を見ると、エリート戦略とトーナメント戦略が優れていることが読み取れる。解くに解の精度としてはエリート戦略の方が優れているように見えるが、それほど大きな違いは見られない。この程度の差ならば、どちらかが完全に上であるとはいえないだろう。選択をしない、もしくはルーレット戦略はほとんど精度の向上が見られない。

また、全体としてEHBSAWOの方が優れているように思えるが、これはcut数が大きく関わっている。つまり、EHBSAWTでは一部分のみをsamplingすることになるので計算時間は少なくなるが、精度自体は低くなる。このことについては適切なパラメータを調べるときにもう一度取り上げる。

図 6.1 のパラメータファイル

```
pop_num;150
city_num;200
max_generation;50
population_select;no_population_select,population_select_half,
 population_select_roulette,population_select_tournament
make_ehm_strategy;make_ehm_delta:0.04
use_twoopt;non_use_twoopt
```

図 6.2 のパラメータファイル

```
pop_num;150
city_num;200
max_generation;50
cut_num;5
population_select;no_population_select,population_select_half,
 population_select_roulette,population_select_tournament
choose_template_strategy;choose_template_randomly
choose_sampling_node_strategy;choose_node_randomly
make_ehm_strategy;make_ehm_delta:0.04
use_twoopt;non_use_twoopt
```

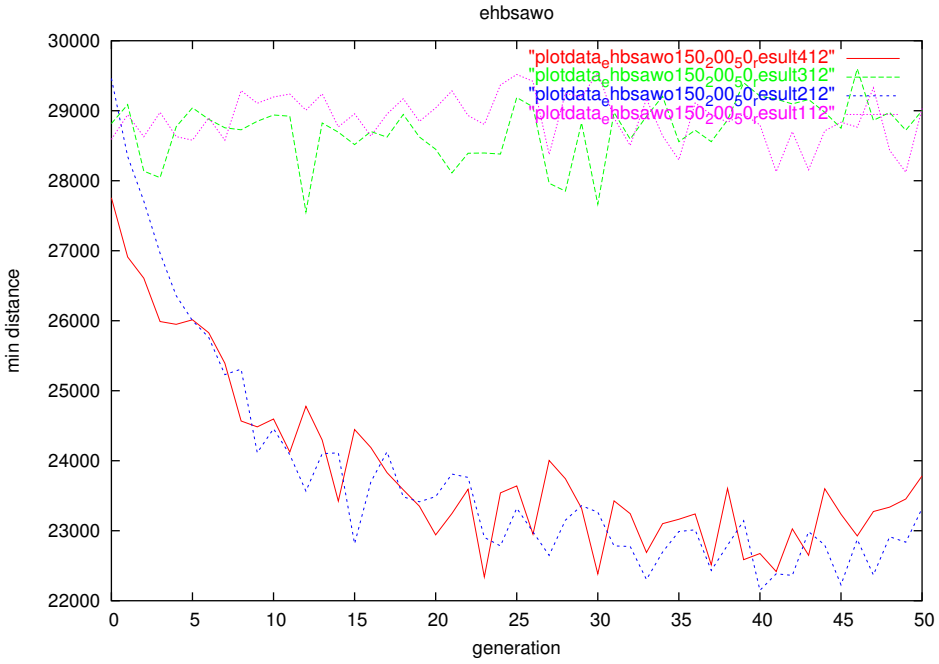

図 6.1: EHBSAWO における初期集団の選択

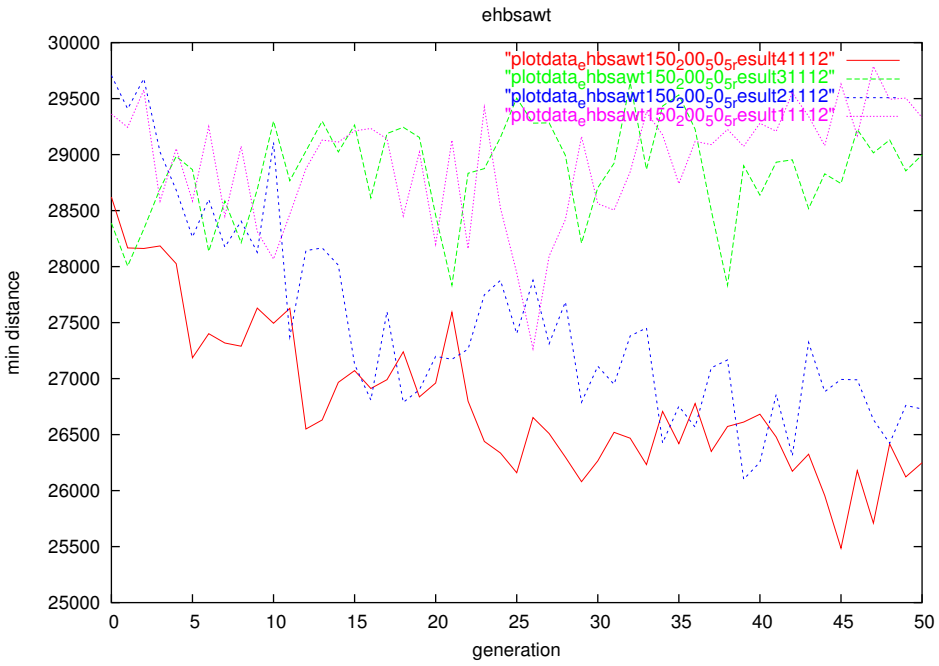

図 6.2: EHBSAWT における初期集団の選択

6.2.2 EHM

EHM の戦略について調べる。結果は図 6.3、6.4 である。これは重みづけをした方がはるかに良い性能を示した。初期集団の選択をしなくても解の精度が向上しているのが分かる。

図 6.3 のパラメータファイル

```
pop_num;150
city_num;200
max_generation;50
population_select;no_population_select
make_ehm_strategy;make_ehm_delta:0.04,make_ehm_weight:0.04
use_twoopt;non_use_twoopt
```

図 6.4 のパラメータファイル

```
pop_num;150
city_num;200
max_generation;50
cut_num;5
population_select;no_population_select
choose_template_strategy;choose_template_randomly
choose_sampling_node_strategy;choose_node_randomly
make_ehm_strategy;make_ehm_delta:0.04,make_ehm_weight:0.04
use_twoopt;non_use_twoopt
```

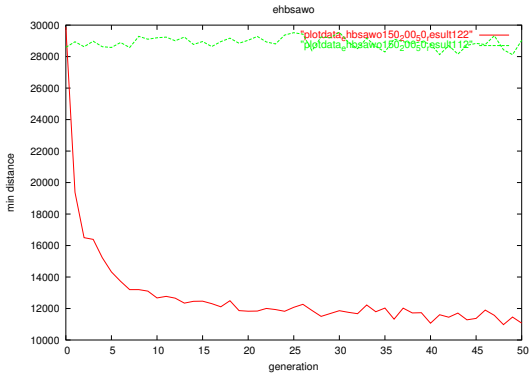

図 6.3: EHBSAWO の EHM 戦略

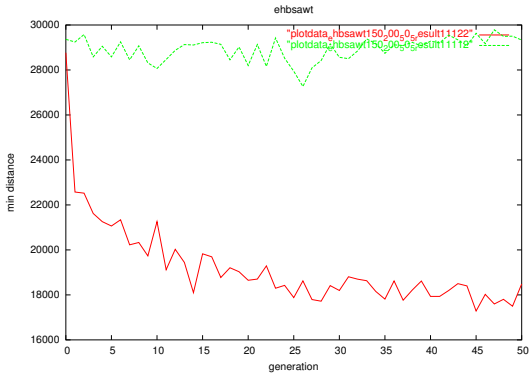

図 6.4: EHBSAWT の EHM 戦略

6.2.3 choose template , choose sampling node

template, sampling node を選択するときの戦略について調べる。結果は図 6.5 である。

図 6.5 の結果はいまいち分析が難しい。強いて云えば、template を選ぶときにはルーレット戦略、sampling node を選ぶときにはランダム戦略が最も良く、template を選ぶときにはエリート戦略、sampling node を選ぶときにはランダム戦略が最も悪いように見えるが、この程度は誤差の範囲だろう。

図 6.5 のパラメータファイル

```
pop_num;150
city_num;200
max_generation;50
cut_num;5
population_select;no_population_select
choose_template_strategy;choose_template_randomly,
 choose_template_roulette,choose_template_elite
choose_sampling_node_strategy;choose_node_randomly,
 choose_node_roulette
make_ehm_strategy;make_ehm_delta:0.04
use_twoopt;non_use_twoopt
```

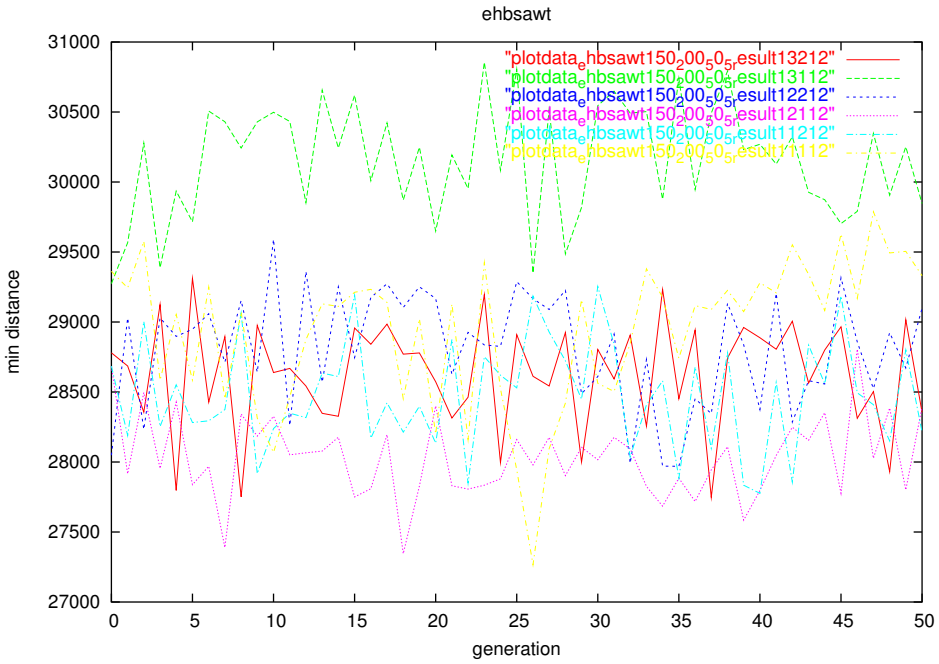

図 6.5: choose template, choose sampling node

6.2.4 2opt

2opt 法を使ったときと、使わなかったときの解の精度の差について調べる。この結果は図 6.6、6.7 である。

EHBSAWO は、ある程度の改善が見られるが、EHBSAWT の方はほとんど変わらないように見える。これはおそらく、EHBSAWT がランダムに template を選んでいるせいで、2opt の効果が次の世代に伝わらないせいだと考えられる。

図 6.6 のパラメータファイル

```
pop_num;150
city_num;200
max_generation;50
population_select;no_population_select
make_ehm_strategy;make_ehm_delta:0.04
use_twoopt;non_use_twoopt,use_twoopt
```

図 6.7 のパラメータファイル

```
pop_num;150
city_num;200
max_generation;50
cut_num;5
population_select;no_population_select
choose_template_strategy;choose_template_randomly
choose_sampling_node_strategy;choose_node_randomly
make_ehm_strategy;make_ehm_delta:0.04
use_twoopt;non_use_twoopt,use_twoopt
```

6.2.5 まとめ

今までの結果より、EHBSAWO は集団の選択に、エリート戦略、トーナメント戦略のどちらか、EHM に重みづけ戦略、局所的改善法の 2opt 法を使用する戦略が良いように思える。EHBSAWT は集団の選択に、エリート戦略、トーナメント戦略のどちらか、EHM に重みづけ戦略、2opt 法を使用しない、という戦略が良いと思える。だが、これらの結果のみからは choose template と choose sampling node はどれが良いともいえない。

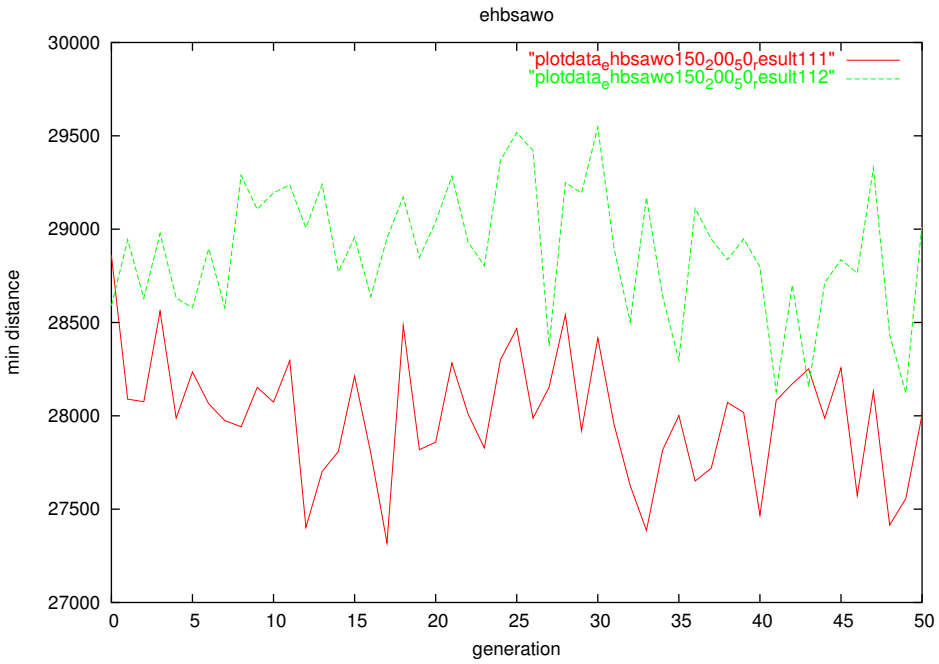

図 6.6: EHBSAWO の 2opt

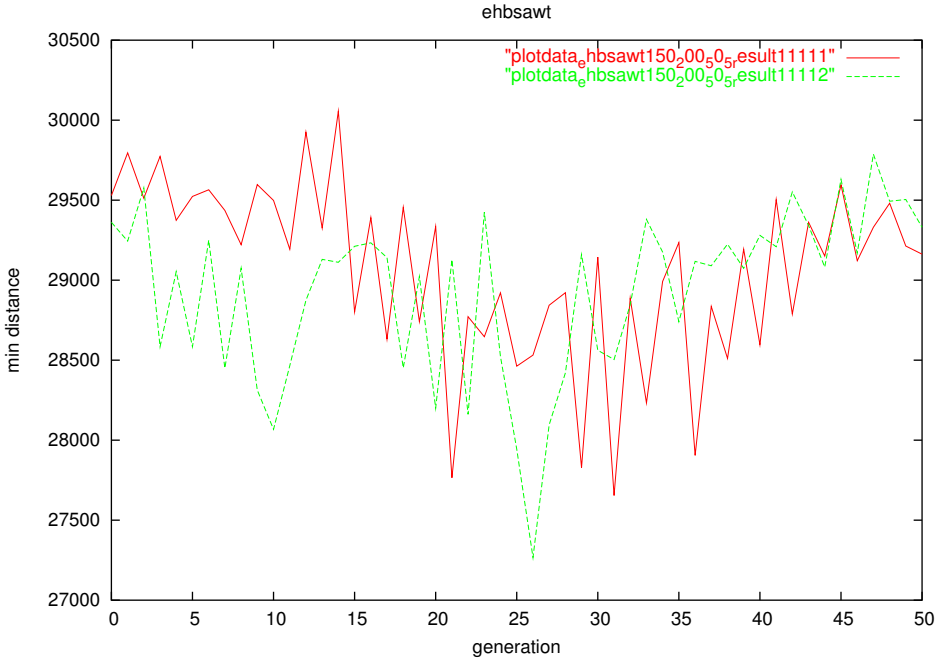

図 6.7: EHBSAWT の 2opt

6.3 複数の戦略を組み合わせたときの評価

6.3.1 EHBSAWO

前節の結果をより、EHBSAWO は集団の選択法としてエリート戦略、トーナメント戦略のどちらか、EHM に重みづけ戦略、2opt 法は有り、という戦略が優れていると考えられる。これを実際に確かめてみる。

次のようなパラメータファイル ehbsawo_parameter4_1 を使い計測、グラフ化し、その次に ehbsawo_parameter4_2 を使い、求める戦略のグラフを抽出する。

図 6.8 のパラメータファイル

```
pop_num;150
city_num;200
max_generation;50
population_select;no_population_select,population_select_half,
 population_select_roulette,population_select_tournament
make_ehm_strategy;make_ehm_weight:0.04
use_twoopt;non_use_twoopt,use_twoopt
```

図 6.9 のパラメータファイル

```
pop_num;150
city_num;200
max_generation;50
population_select;population_select_half,
 population_select_tournament
make_ehm_strategy;make_ehm_weight:0.04
use_twoopt;use_twoopt
```

この結果は、図 6.8、6.9 となる。

少し見にくいので説明すると、最終状態は優れている順に select_elite,twoopt 有り、次は select_elite,twoopt 無し、select_tournament,twoopt 有り、となっており、想定が正しかったことが分かる。

6.3.2 EHBSAWT

前節の結果をふまえ、他の戦略と比較したときも template、sampling node を選ぶ戦略が効果をもたないかを調べる。

そのために、次のようなパラメータファイル ehbsawt_parameter4_1 を使い計測、グラフ化し、その次に ehbsawt_parameter4_2 を使い、求める戦略のグラフを抽出する。

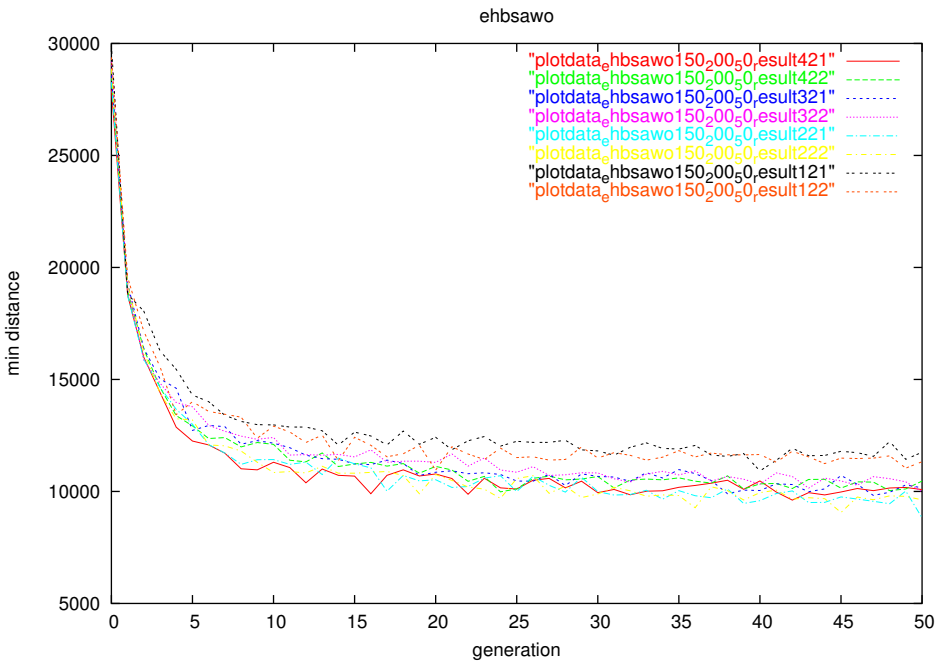

図 6.8: EHBSAWO の複合戦略 1

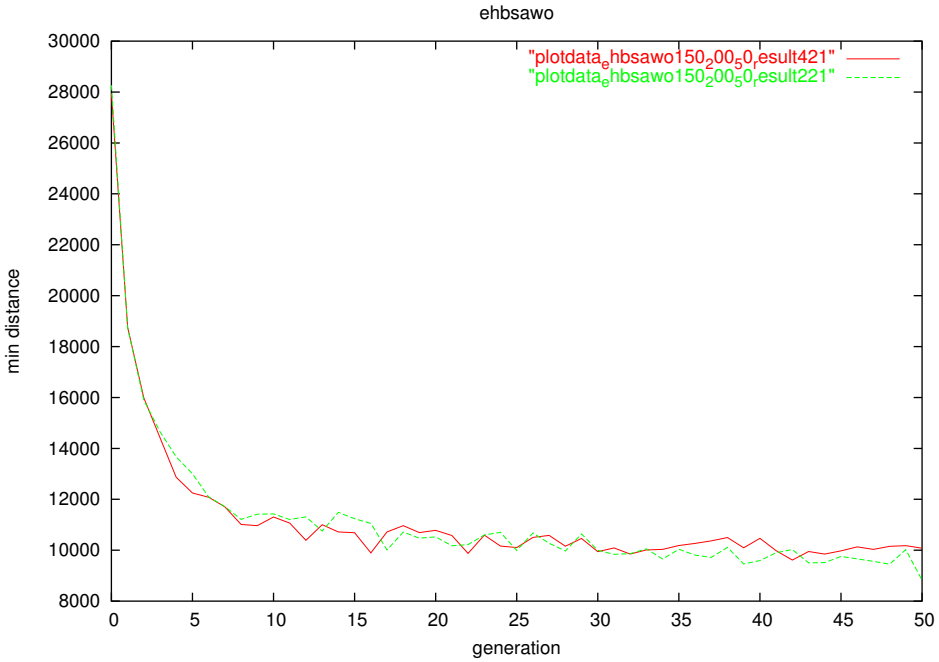

図 6.9: EHBSAWO の複合戦略 2

図 6.10 のパラメータファイル

```
pop_num;150
city_num;200
max_generation;50
cut_num;5
population_select;no_population_select,population_select_half,
 population_select_roulette,population_select_tournament
choose_template_strategy;choose_template_randomly,
 choose_template_roulette,choose_template_elite
choose_sampling_node_strategy;choose_node_randomly,
 choose_node_roulette
make_ehm_strategy;make_ehm_weight:0.04
use_twoopt;non_use_twoopt
```

図 6.11 のパラメータファイル

```
pop_num;150
city_num;200
max_generation;50
cut_num;5
population_select;population_select_half,
 population_select_tournament
choose_template_strategy;choose_template_randomly,
 choose_template_roulette,choose_template_elite
choose_sampling_node_strategy;choose_node_randomly,
 choose_node_roulette
make_ehm_strategy;make_ehm_weight:0.04
use_twoopt;non_use_twoopt
```

この結果は、図 6.10、6.11 となる。

choose_template と choose_sampling_node の戦略はほとんど性能に影響を与えないことが分かる。

6.4 パラメータを変化させたときの評価

6.4.1 実行時間と cut 数

cut 数は実行時間に大きく関わっている。例えば、上の例の cut 数が 5 のとき、同等の EHBSAWO の約 4 分の 1 の実行時間である。そこで、本節では cut 数と集団数を変化させたときの実行時間と解の精度を調べる。EHBSAWT が EHBSAWO と同程度の実行時間でそれよりも良い解の精度を出すことが出来るのかを調べるのが目的である。まず、次のようなパラメータファイルを用意し、計測する。

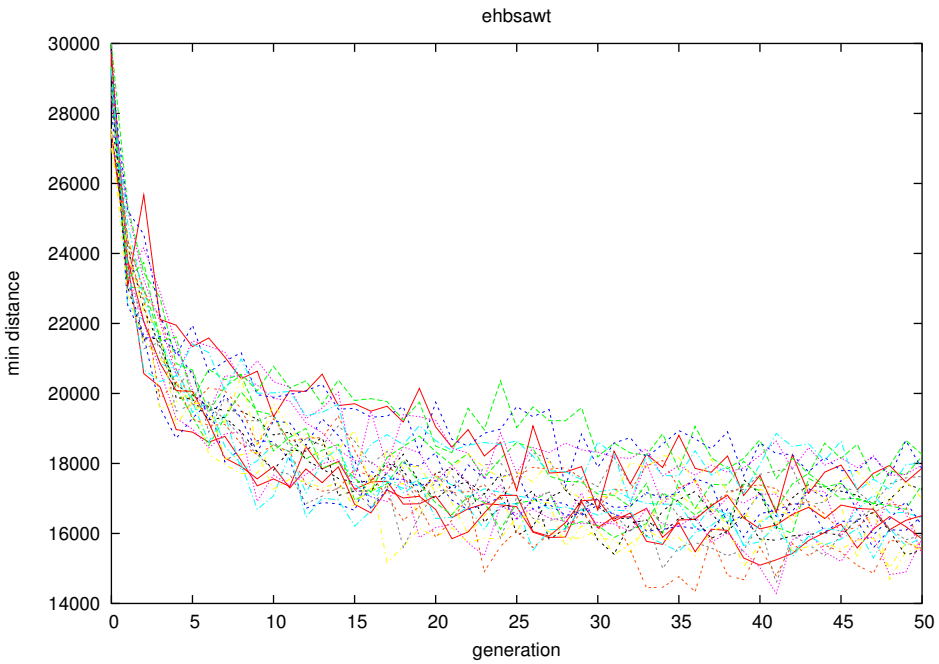

図 6.10: EHBSAWT の複合戦略 1

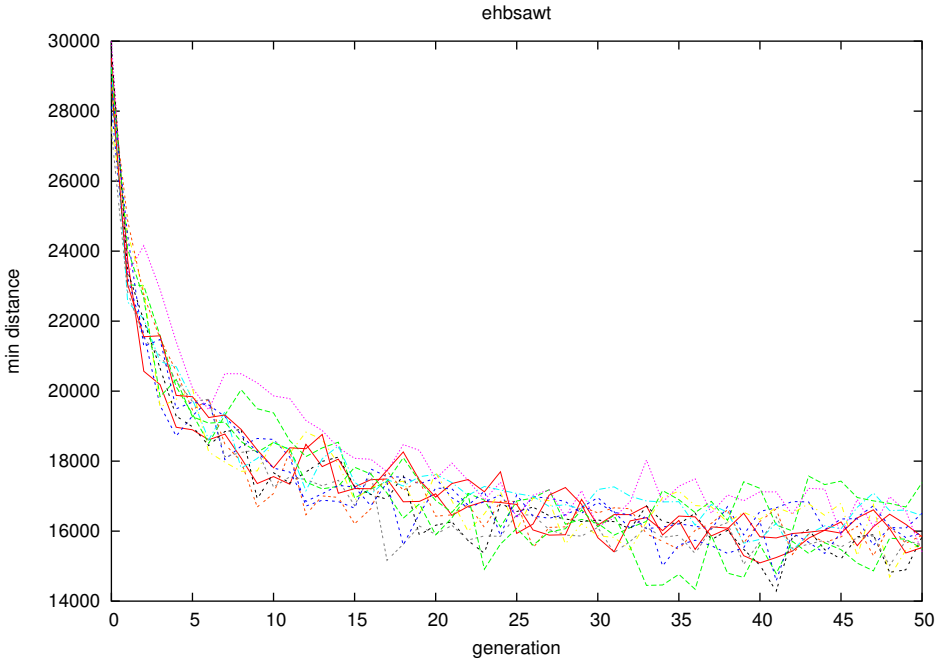

図 6.11: EHBSAWT の複合戦略 2

cut の効果を調べるためのパラメータファイル

```
pop_num;150
city_num;200
max_generation;50
cut_num;1,2,3,4
population_select;population_select_half
 ,population_select_tournament
choose_template_strategy;choose_template_randomly
choose_sampling_node_strategy;choose_node_randomly
make_ehm_strategy;make_ehm_weight:0.04
use_twoopt;non_use_twoopt
```

これらの実行結果を基に調べると表 6.1、6.2 が出来る。ただし、EHBSAWT1/1 は EHBSAWT1 の戦略で cut 数が 1 であることを表す。また、EHBSAWO1、EHBSAWO2、EHBSAWT1、EHBSAWT2 で固定されているパラメータは以下の通り。

EHBSAWO1 のパラメータ

```
pop_num;150
city_num;200
max_generation;50
population_select;population_select_half,
make_ehm_strategy;make_ehm_weight:0.04
use_twoopt;use_twoopt
```

EHBSAWO2 のパラメータ

```
pop_num;150
city_num;200
max_generation;50
population_select;population_select_tournament
make_ehm_strategy;make_ehm_weight:0.04
use_twoopt;use_twoopt
```

EHBSAWT1 のパラメータ

```
city_num;200
max_generation;50
population_select;population_select_half
choose_template_strategy;choose_template_randomly
choose_sampling_node_strategy;choose_node_randomly
make_ehm_strategy;make_ehm_weight:0.04
use_twoopt;non_use_twoopt
```

EHBSAWT2 のパラメータ

```
city_num;200
max_generation;50
population_select;population_select_tournament
choose_template_strategy;choose_template_randomly
choose_sampling_node_strategy;choose_node_randomly
make_ehm_strategy;make_ehm_weight:0.04
use_twoopt;non_use_twoopt
```

表 6.1: cut 数と計測時間、および解の精度 1

	集団数/最小距離/時間 (s)	集団数/最小距離/時間 (s)
EHBSAWO1	150/8823.9/16.6	150/8823.9/16.6
EHBSAWT1/1	150/9792.9/16.7	150/9792.9/16.7
EHBSAWT1/2	150 /11824.8/7.63	300/12462.9/15.4
EHBSAWT1/3	150/13729.6/5.35	450/14720.0/16.0
EHBSAWT1/4	150/14436.0/4.43	600/17208.3 /17.9
EHBSAWT1/5	150/15950.1 /4.17	750/19280.9/21.3

表 6.2: cut 数と計測時間、および解の精度 2

	集団数/最小距離/時間 (s)	集団数/最小距離/時間 (s)
EHBSAWO2	150/10081.3/16.6	150/10081.3/16.6
EHBSAWT2/1	150/9730.2 /16.7	150/9730.2/16.7
EHBSAWT2/2	150/11588.5/7.70	300/13372.9/15.2
EHBSAWT2/3	150/14407.8/5.27	450/16118.3/16.0
EHBSAWT2/4	150/15035.6/4.48	600/17210.9/18.2
EHBSAWT2/5	150/15809/5/4.13	750/ 19376.0/21.1

表 6.1、6.2 を見ると、cut 数を増やしたときに単純に集団数を増やすとかえって解の精度が悪化していることが分かる。これは template をランダムに選ぶせいで、質の良くないものを template にしてしまっている可能性が高いことに原因が有ると思われた。そこで、template をエリート戦略で選んで実行すれば、集団の増加による解の精度の向上が望めるか、と考え実行したが、ほとんど結果は変わらなかった。結局これは cut することで、EHM の重みづけの効果が少なくなってしまうことに大きな原因があると考えられる。

6.4.2 集団数、都市数、世代数

次に集団数、都市数、世代数の関係について調べる。前節の結果をふまえて、cut の数は 1 に固定する。ただ、EHBSAWT においては template を選択するときにランダム戦略に加えてエリート戦略も調べる。これは集団数を増やしたときに、template をエリート戦略で選んだ方が解の精度が良くなる可能性を考慮したためである。

図 6.12 のパラメータファイル

```
pop_num;50,100,150,200,250,300,350,400
city_num;200
max_generation;150
population_select;population_select_half,
 population_select_tournament
make_ehm_strategy;make_ehm_weight:0.04
use_twoopt;use_twoopt
```

図 6.13 のパラメータファイル

```
pop_num;50,100,150,200,250,300,350,400
city_num;200
max_generation;150
cut_num;1
population_select;population_select_half,
 population_select_tournament
choose_template_strategy;choose_template_randomly,
 choose_template_elite
choose_sampling_node_strategy;choose_node_randomly
make_ehm_strategy;make_ehm_weight:0.04
use_twoopt;non_use_twoopt
```

図 6.12、6.13 を見ると、戦略、集団数によって集束する世代数がほとんど変わらないことが見て取れる。またデータを調べると、集団数がある程度以上増えると解の精度はかえって悪くなることも分かる。6.12、6.13 から、集団数 50 のもののみを抽出したのが、図 6.14、6.15 である。

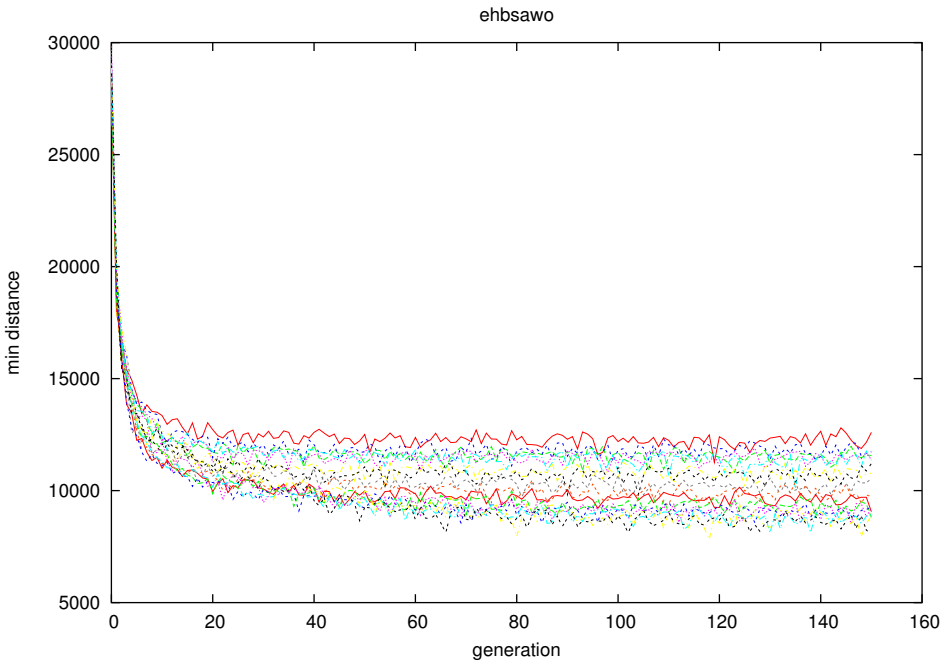

図 6.12: EHBSAWO のパラメータ 1

図 6.14、6.15 を見ると、集団数の大きいものに比べてあきらかに解の精度の良いことが分かる。よって次はどの程度の集団数が都市数 200 のときには適当なのかを調べる。この結果は図 6.16、6.17。これは図 6.14、6.15 のパラメータで集団数以外を等しくし、集団数を 10 から 100 まで 10 ずつ変化させて実行したものである。

図 6.16、6.17 を見ると、EHBSAWO は 40 以上ならばそれほど差はない。EHBSAWT なら 50 以上ならばそれほど差は見られない。これ以上はこのデータからは読み取れない。

そこで、集団数と都市の数の関係についてさらに調べる。都市の数を変化させ、適切な集団の数を探索する。上記の結果より、集団数は都市の数の 4 分の 1 から同等の数を目安にして探索する。集束は戦略、パラメータにそれほど依存しないようなので、世代数はほぼ集束しきっているように見える 150 で固定する。

都市の数を 200,400,600,800,1000 のときを調べる。それぞれに対して次の集団数を調べた。都市数 50 のものは集団数 10 から 10 ずつ、200 まで、都市数 400 のものは集団数 100 から 20 ずつ 400 まで、都市数 600 のものは集団数 150 から 50 ずつ 600 まで、都市数 800 のものは集団数 200 から 100 ずつ 800 まで、都市数 1000 のものは集団数 250 から 150 ずつ 1000 まで。

この結果は図 6.18 から図 6.27。

図 6.18 から 6.27 を見ると、都市の数を変化させたときも、特に変化は見られない。つまり、やはり集団数が都市数の 4 分の 1 以上を増えると、解の精度は悪くなる。また、cut 数 1 の EHBSAWT と EHBSAWO は解の精度という点ではほとんど差は見られなかった。またそれほど明確な差はみられないが、EHBSAWT においては template はランダム戦略、集団の選択はエリート戦略が優れているよ

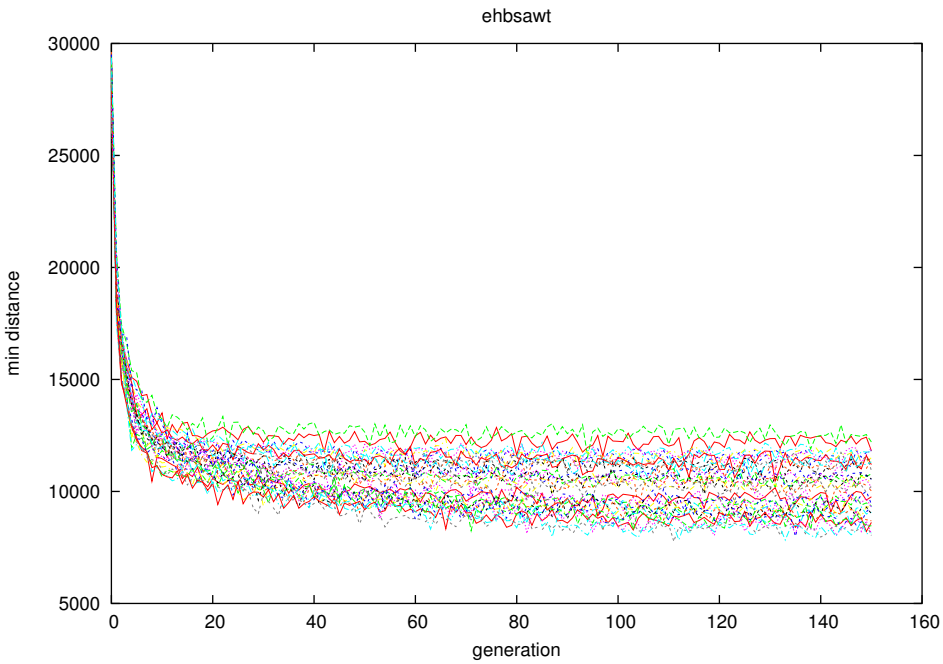

図 6.13: EHBSAWT のパラメータ 1

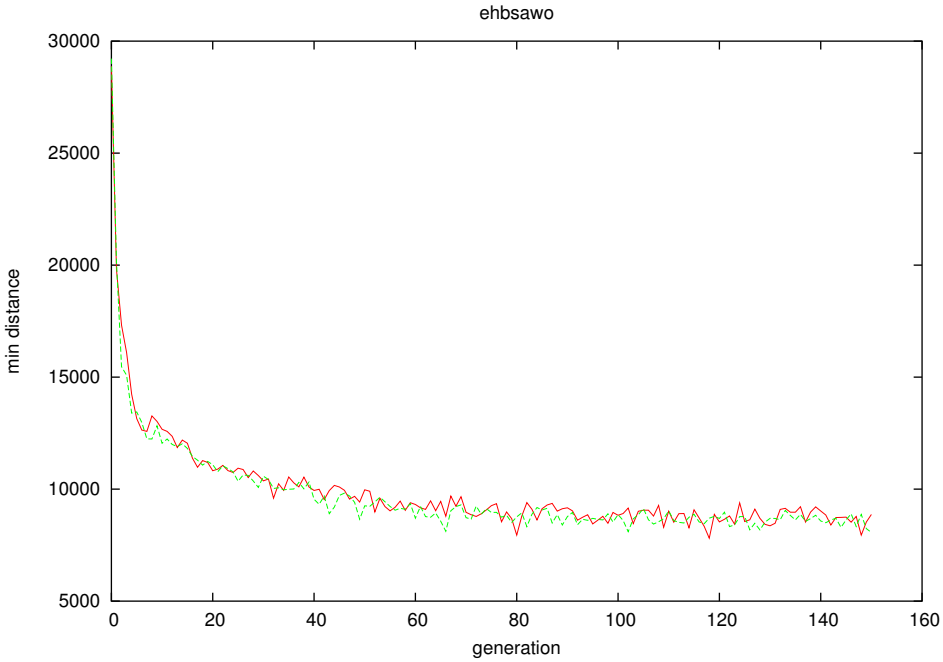

図 6.14: EHBSAWO のパラメータ 2

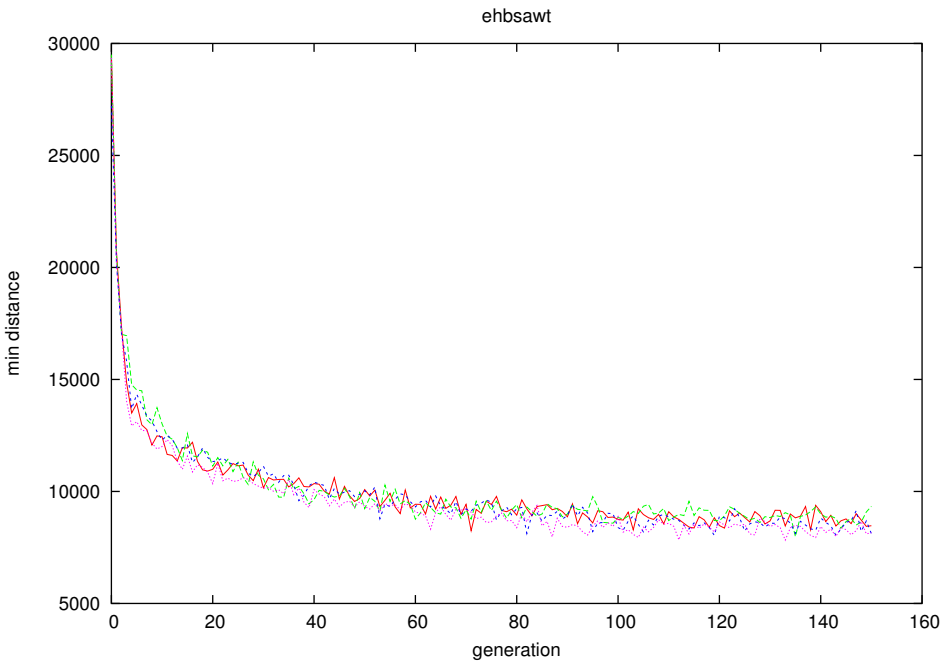

図 6.15: EHBSAWT のパラメータ 2

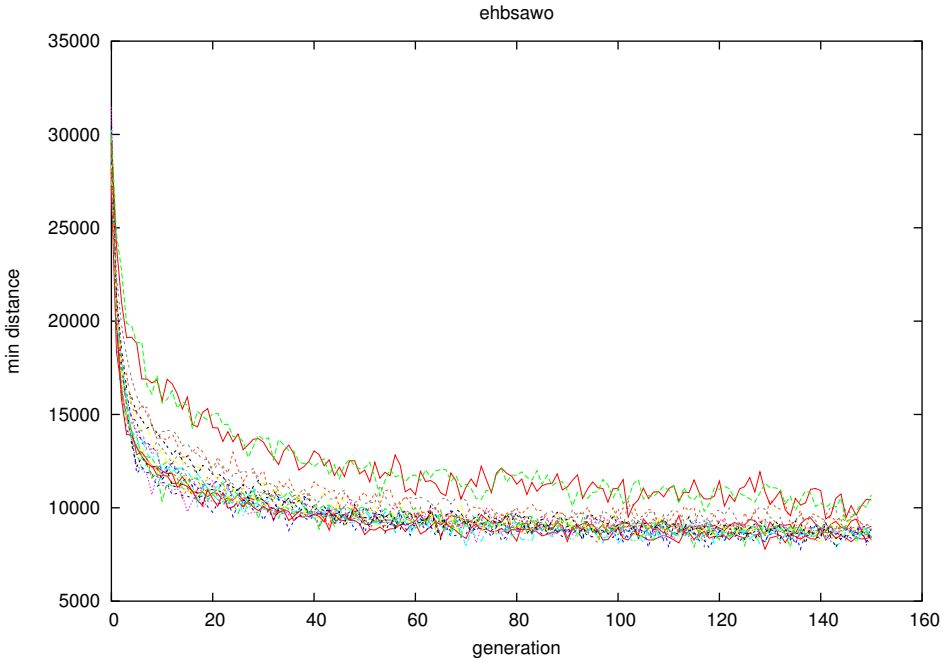

図 6.16: EHBSAWO のパラメータ 3

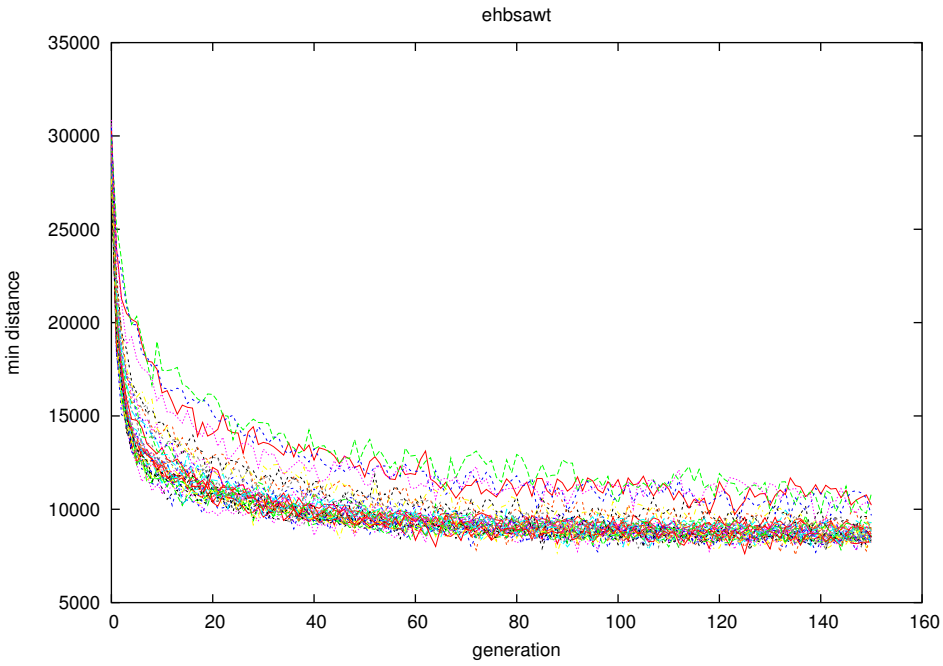

図 6.17: EHBSAWT のパラメータ 3

うだ。EHBSAWO においても集団の選択はどちらかといえばエリート戦略が優れているようにみえる。

6.5 並列化させたときの評価

6.5.1 戦略

次に並列化の評価を行なう。送受信以外の戦略、パラメータを固定し、解の精度を調べる。送受信以外の戦略は、逐次版で平均して最も優れた性能を出した次の戦略で行なう。

EHBSAWO の戦略

```
population_select:population_select_half
make_ehm_strategy:make_ehm_weight:0.04
use_twoopt:use_twoopt
```

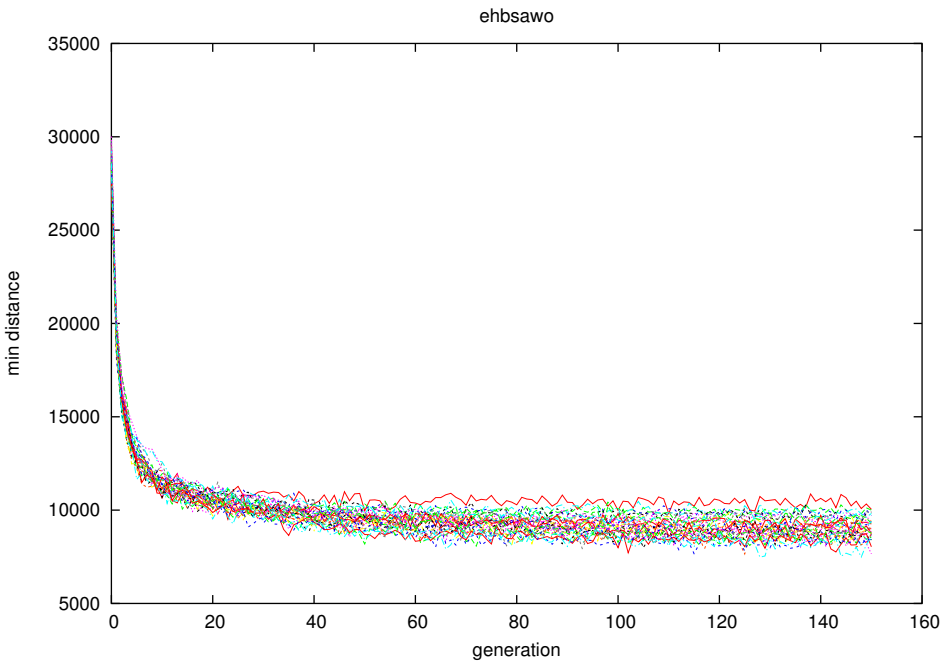

図 6.18: EHBSAWO:都市数 200

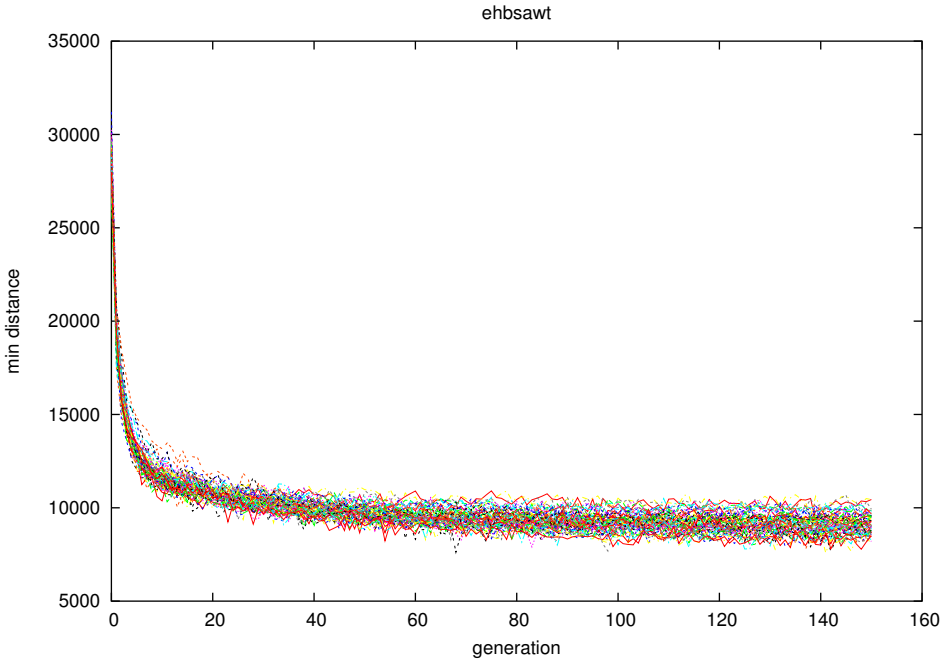

図 6.19: EHBSAWT:都市数 200

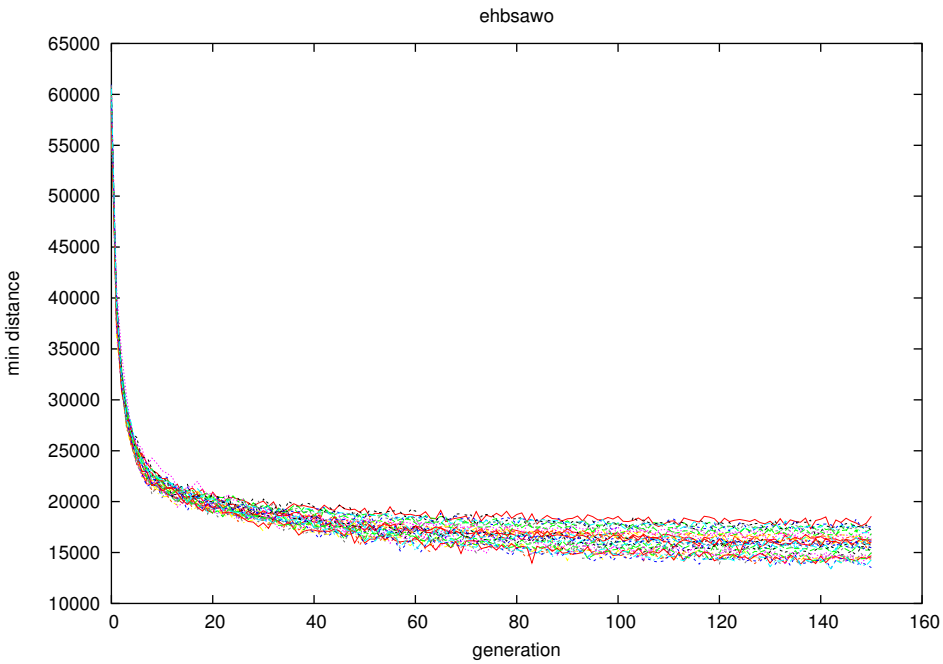

図 6.20: EHBSAWO:都市数 400

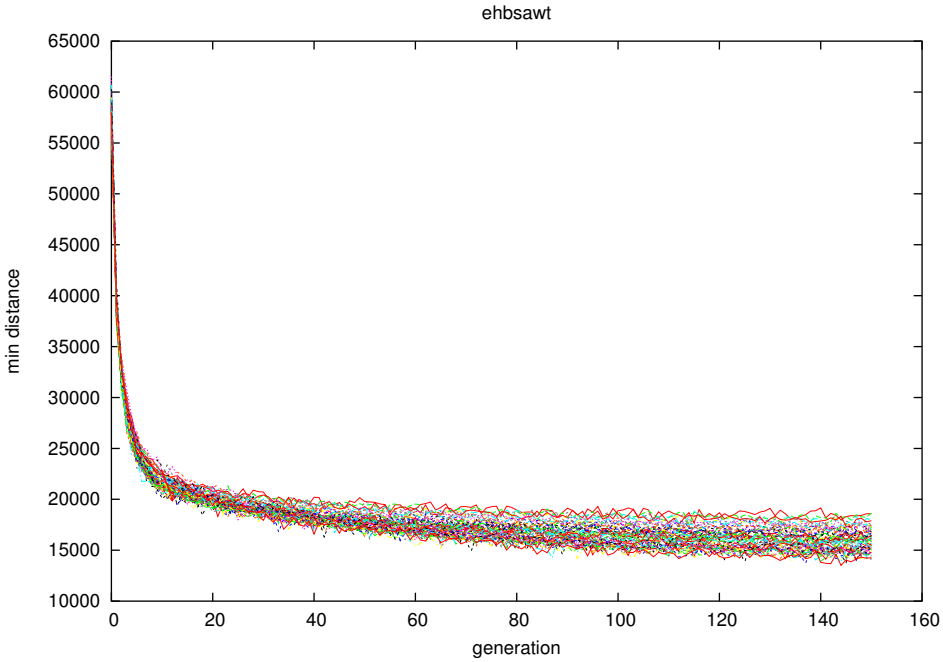

図 6.21: EHBSAWT:都市数 400

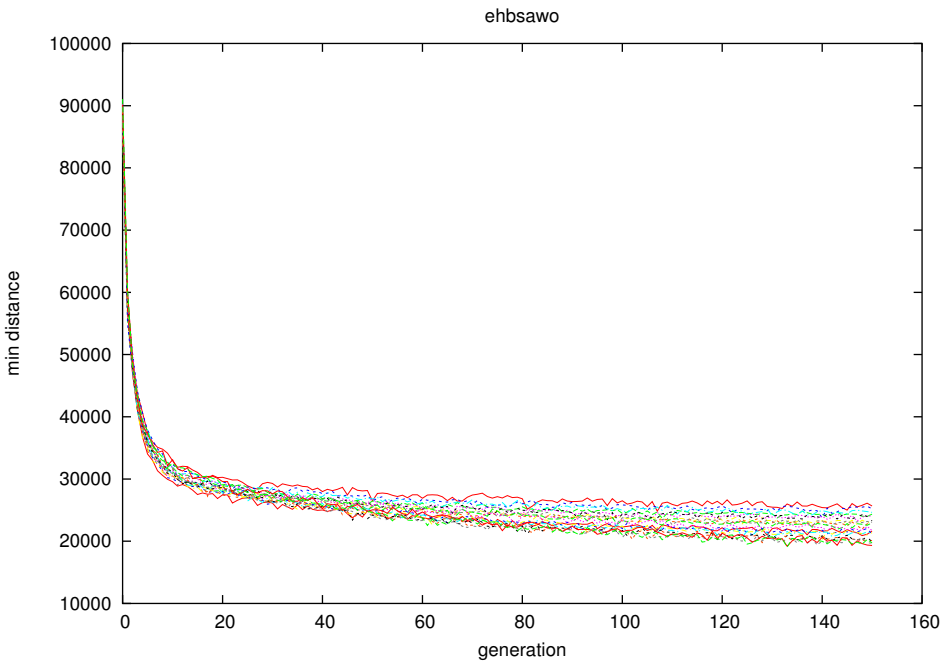

図 6.22: EHBSAWO:都市数 600

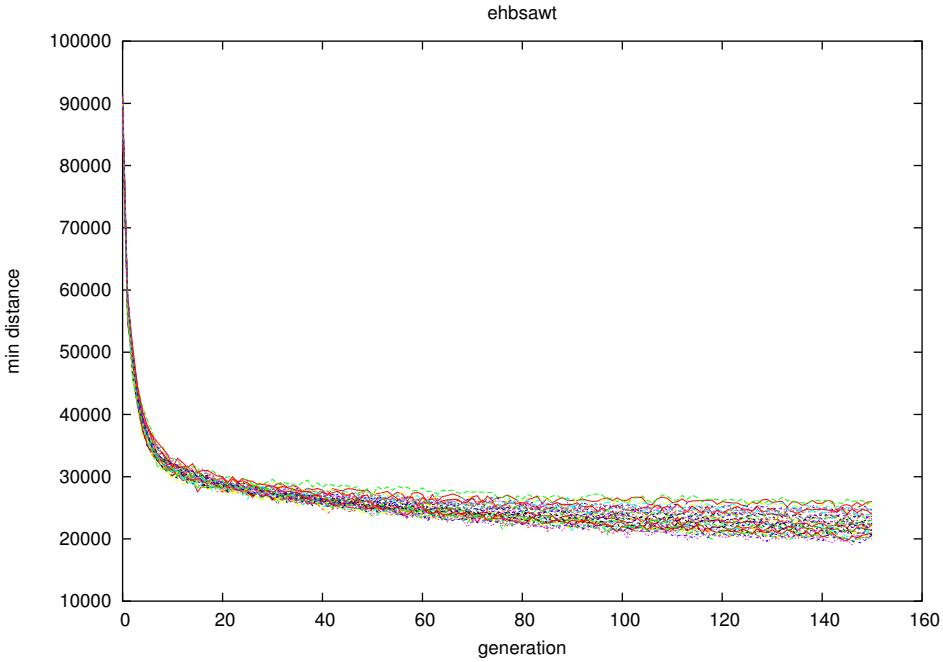

図 6.23: EHBSAWT:都市数 600

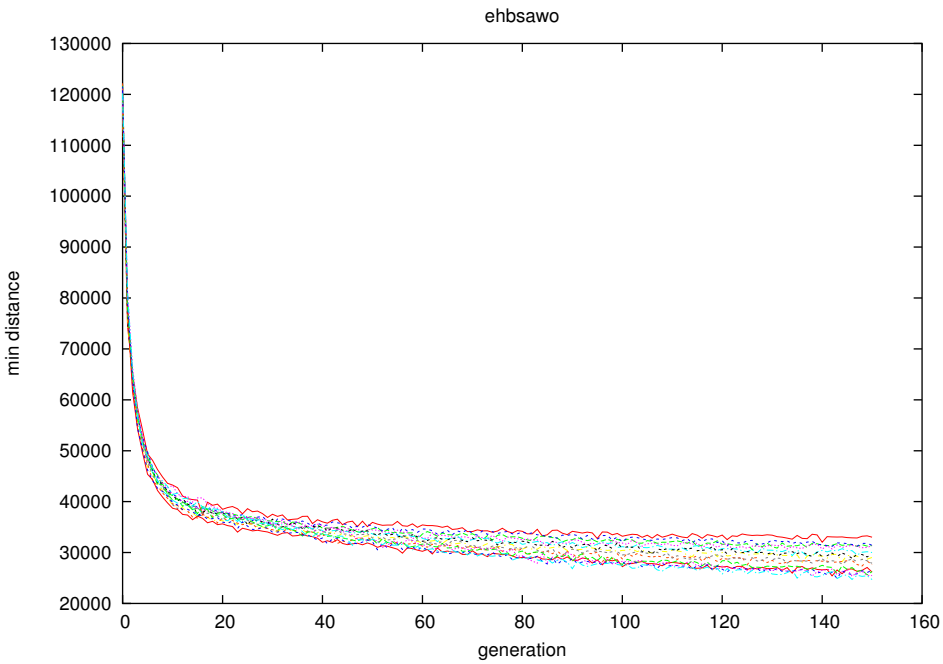

図 6.24: EHBSAWO:都市数 800

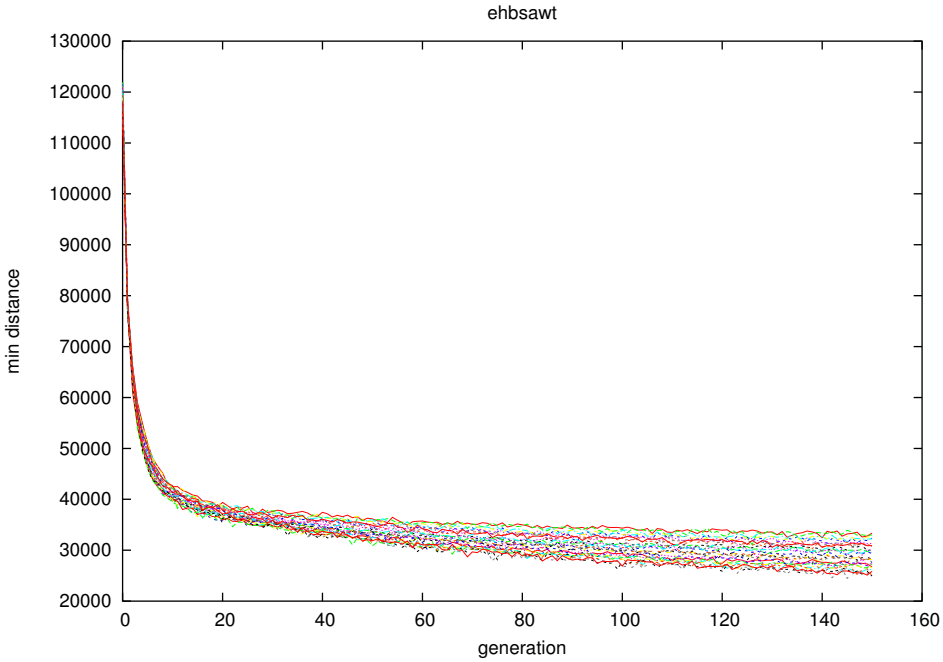

図 6.25: EHBSAWT:都市数 800

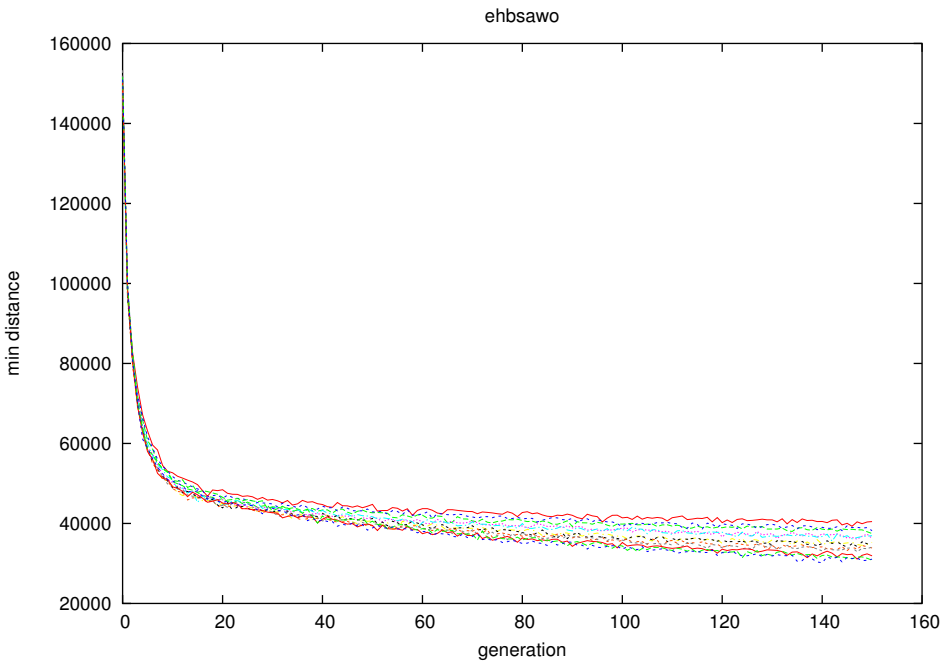

図 6.26: EHBSAWO:都市数 1000

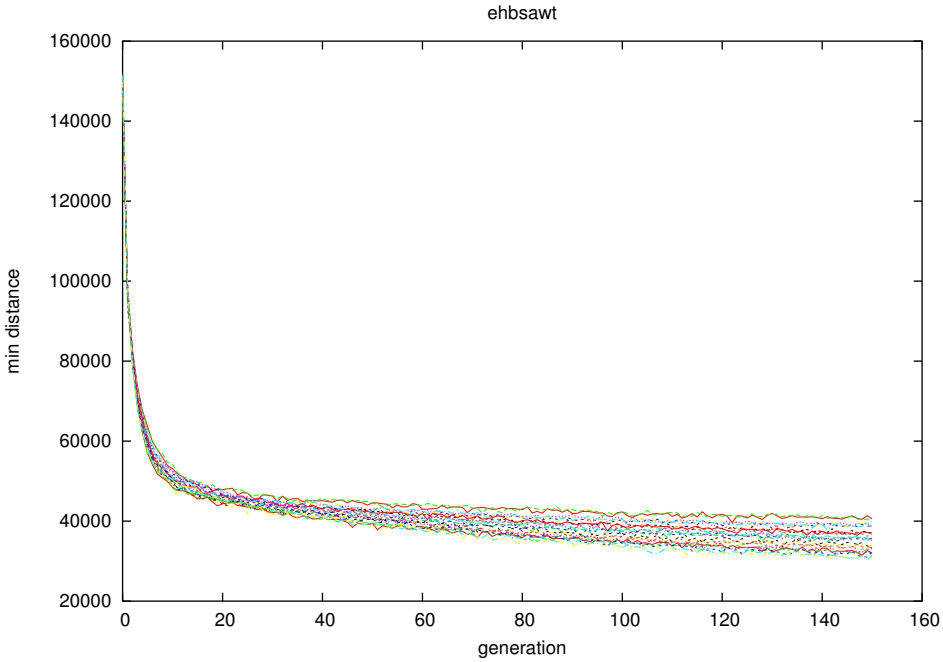

図 6.27: EHBSAWT:都市数 1000

EHBSAWT の戦略

```
cut_num:1
population_select:population_select_half
choose_template_strategy:choose_template_randomly
choose_sampling_node_strategy:choose_node_randomly
make_ehm_strategy:make_ehm_weight:0.04
use_twoopt:non_use_twoopt
```

まずは、集団数 100、都市数 200、世代数 50、送信数 10、プロセス数 8、で各戦略を計測してみる。この結果は図 6.28、6.29 になる。

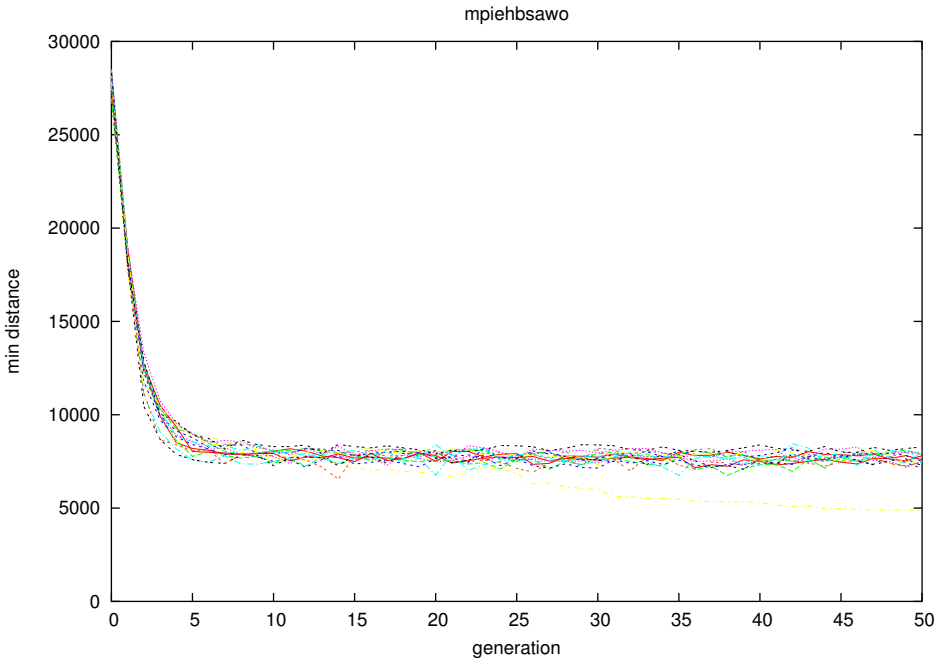

図 6.28: MPIEHBSAWO の戦略

これは明らかに送信の戦略、受信の戦略ともにエリート戦略のものが解の精度が良い。また、逐次では最高 8000 程度だった都市数 200 の解の精度が、5000 程度まで改善されている点も注目される。以下は送受信の戦略をエリート戦略に固定し、パラメータを変化させ解の精度を調べる。

6.5.2 パラメータ

都市数を 200、集団数を 50 に固定し、送受信数、プロセス数を変化させる。つまり、次のようなパラメータファイルを作り、実行する。

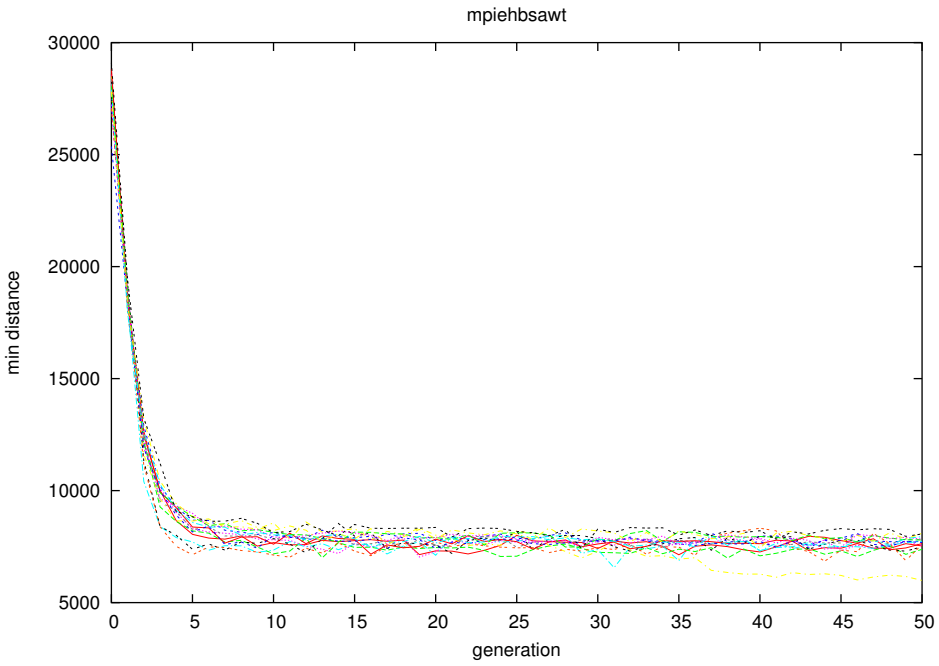

図 6.29: MPIEHBSAWT の戦略

図 6.30 のパラメータファイル

```
pop_num;50
city_num;200
max_generation;150
process_num;2,3,4,5,6,7,8
population_select;population_select_half
make_ehm_strategy;make_ehm_weight:0.04
use_twoopt;use_twoopt
send_strategy;send_elite
receive_strategy;receive_elite
send_num;5,10,15,20,25
```

図 6.31 のパラメータファイル

```
pop_num;50
city_num;200
max_generation;150
cut_num;1
process_num;2,3,4,5,6,7,8
population_select;population_select_half
choose_template_strategy;choose_template_randomly
choose_sampling_node_strategy;choose_node_randomly
make_ehm_strategy;make_ehm_weight:0.04
use_twoopt;non_use_twoopt
send_strategy;send_elite
receive_strategy;receive_elite
send_num;5,10,15,20,25
```

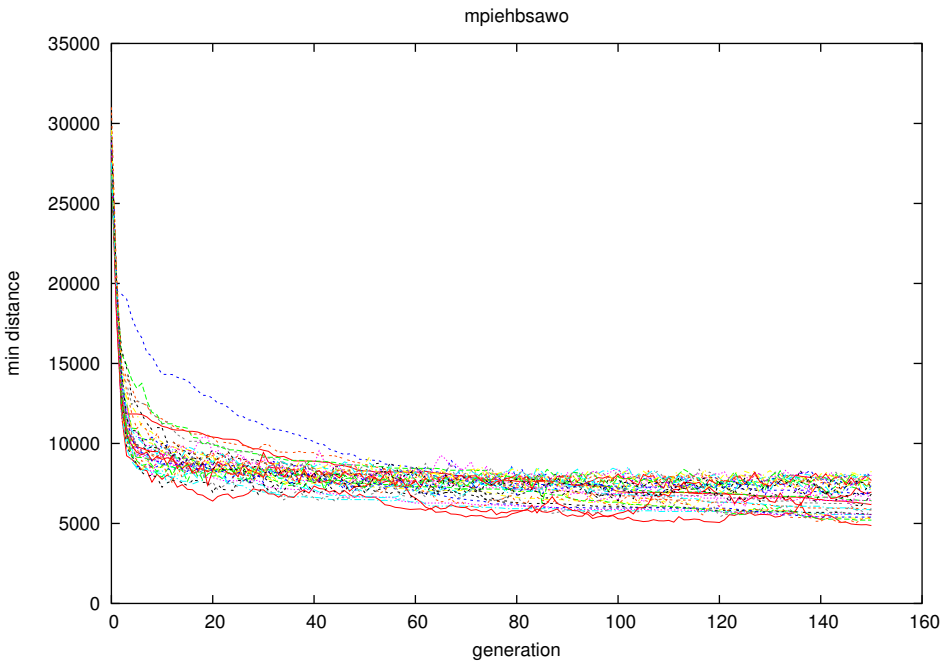

図 6.30: MPIEHBSAWO のパラメータ 1

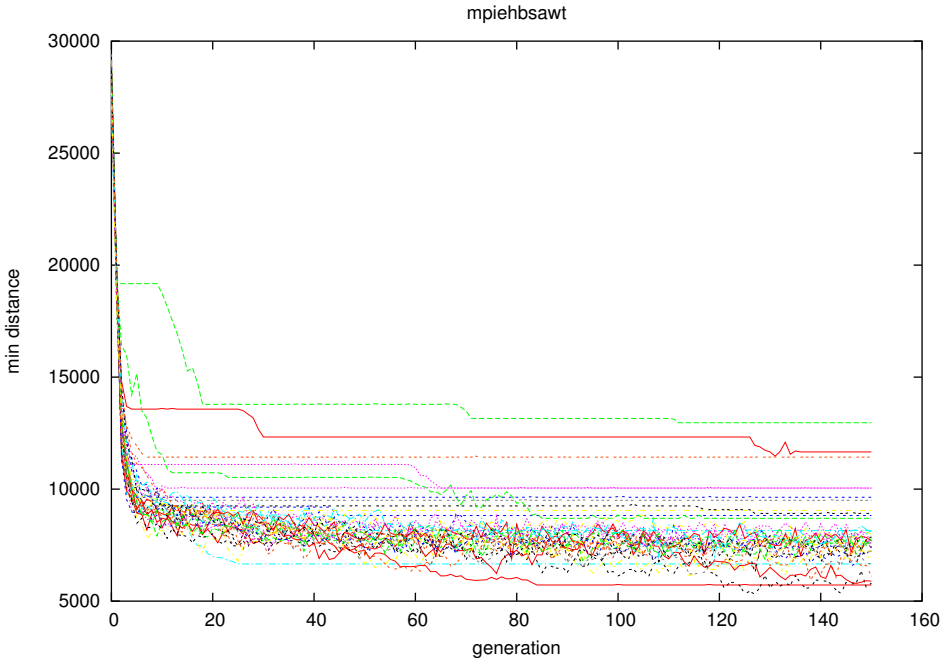

図 6.31: MPIEHBSAWT のパラメータ 1

MPIEHBSAWT のほうはプロセス数をふやすと著しく結果が悪くなる。また、最高の結果も MPIEHBSAWO とほとんど同程度である。よって以下では安定性において優る MPIEHBSAWO のほうのみを考える。

MPIEHBSAWO のデータを見ると、これだけでは良く分からない。単純にプロセス数を増やせばよいと云うわけではないようだし、単純に送受信数を増やせば良い (もしくは減らせば良い) というわけではないようだ。ただし、パラメータを適切に設定したときとそうでないときの解の精度の差はかなり大きい。図 6.30 における最良の最終状態はプロセス数 7 で送受信数 5 のデータだが、この解は 4864.1 と逐次版の 7657.1 より 1.6 倍程度優れている。

とりあえず、いくつか比較のためのデータをとってみる。

図 6.32 のパラメータファイル

```
pop_num;100
city_num;400
max_generation;150
process_num;2,3,4,5,6,7,8
population_select;population_select_half
make_ehm_strategy;make_ehm_weight:0.04
use_twoopt;use_twoopt
send_strategy;send_elite
receive_strategy;receive_elite
send_num;10,20,30,40,50
```

図 6.33 のパラメータファイル

```
pop_num;150
city_num;600
max_generation;150
process_num;6,7,8
population_select;population_select_half
make_ehm_strategy;make_ehm_weight:0.04
use_twoopt;use_twoopt
send_strategy;send_elite
receive_strategy;receive_elite
send_num;10,15,30,45,60,75
```

図 6.34 のパラメータファイル

```
pop_num;200
city_num;800
max_generation;150
process_num;6,7,8
population_select;population_select_half
make_ehm_strategy;make_ehm_weight:0.04
use_twoopt;use_twoopt
send_strategy;send_elite
receive_strategy;receive_elite
send_num;10,15,30
```

図 6.35 のパラメータファイル

```
pop_num;250
city_num;1000
max_generation;150
process_num;6,7,8
population_select;population_select_half
make_ehm_strategy;make_ehm_weight:0.04
use_twoopt;use_twoopt
send_strategy;send_elite
receive_strategy;receive_elite
send_num;20,30,40
```

これらの結果の中で最も性能の良かったものを、同じ都市数の最も性能の良かった逐次版と比較した表が表 6.3 になる。送受信数は少ない方が最高の性能を出しやすいことが見て取れる。また、逐次版と比べたときの性能比が都市数が多いときは悪いのは、パラメータを十分に調べ尽くしていないせいだと考えられる。

しかし、やはり単純にプロセス数を増やし、送受信数を減らせば良いというわけではないようだ。送受信数を固定し、プロセス数ごとの解を表にしたものが表 6.4 だが、これを見るとプロセス数が 1 違うと結果が大きく異なることがあることが分かる。

表 6.3: MPIEHBSAWO の実験結果 1

都市数	プロセス数	送受信数	並列版の解	逐次版の解	逐次版/並列版
200	7	5	4864.1	7657.1	1.57
400	8	10	8273.5	13525.5	1.63
600	6	10	15428.0	19317.1	1.25
800	8	10	17800.1	24701.0	1.38
1000	6	20	23639.5	30812.7	1.30

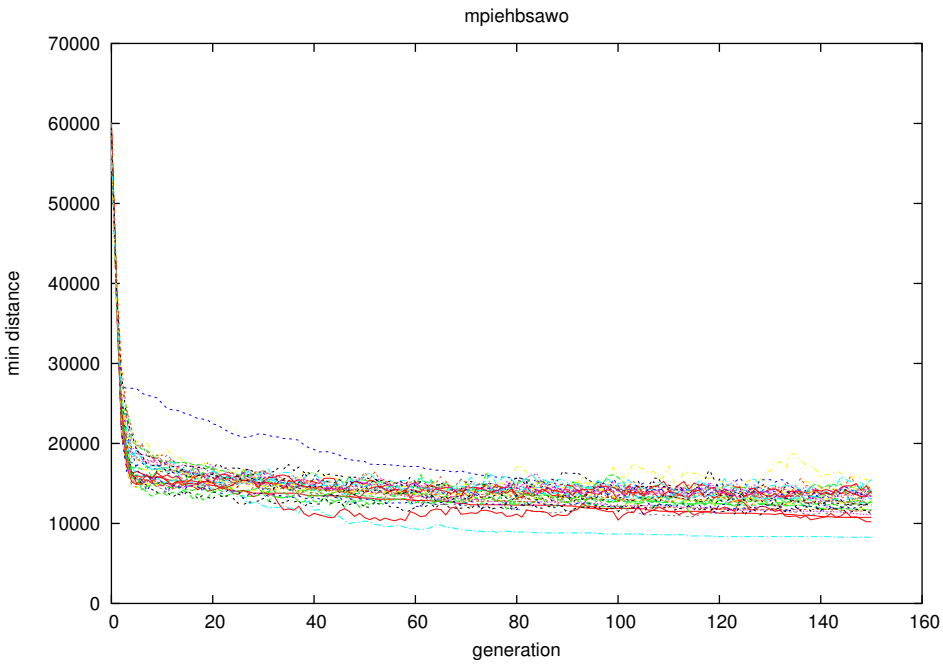

図 6.32: MPIEHBSAWO のパラメータ 2

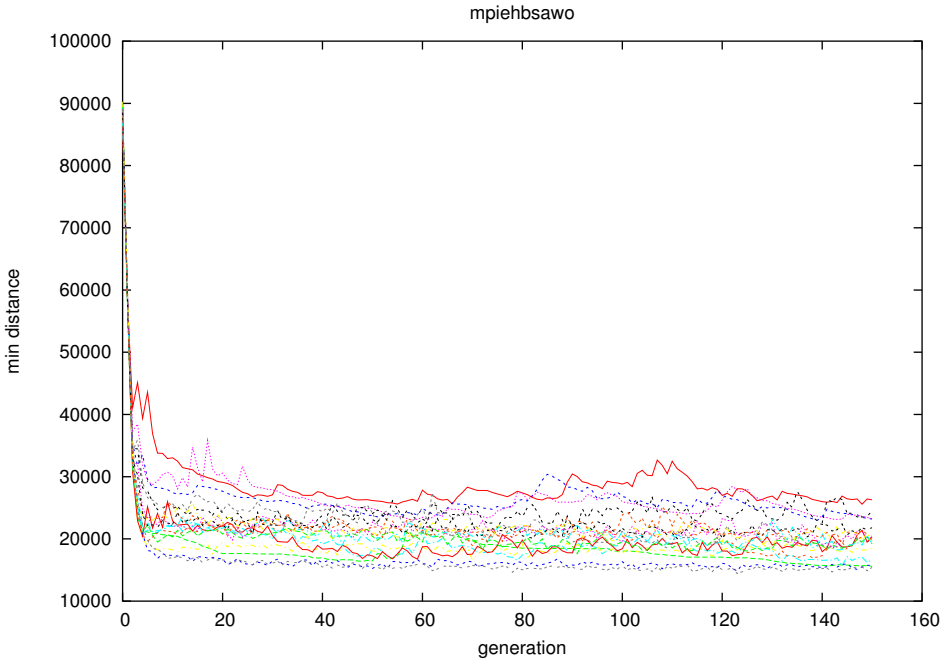

図 6.33: MPIEHBSAWO のパラメータ 3

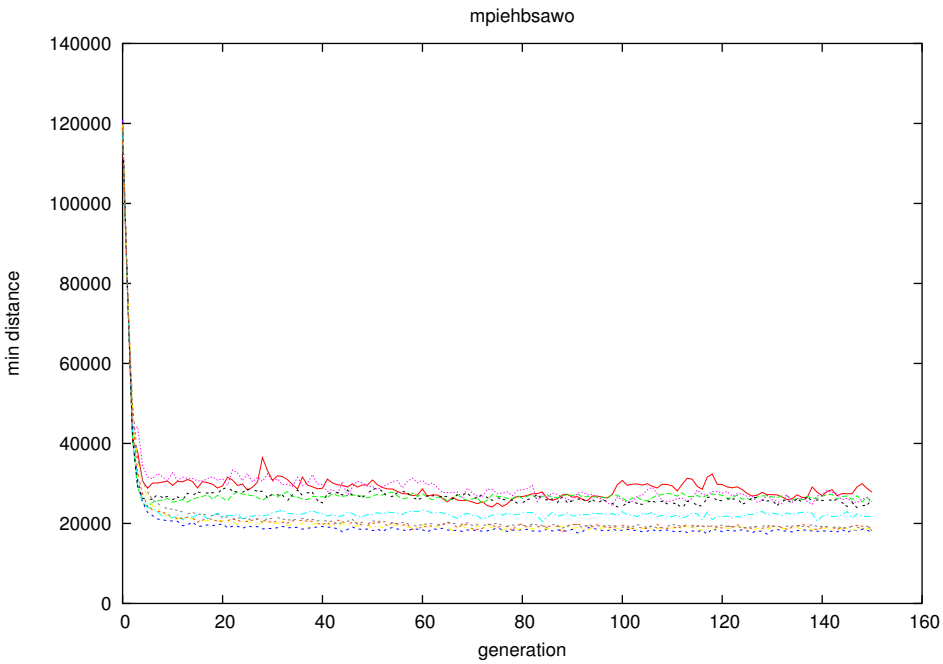

図 6.34: MPIEHBSAWO のパラメータ 4

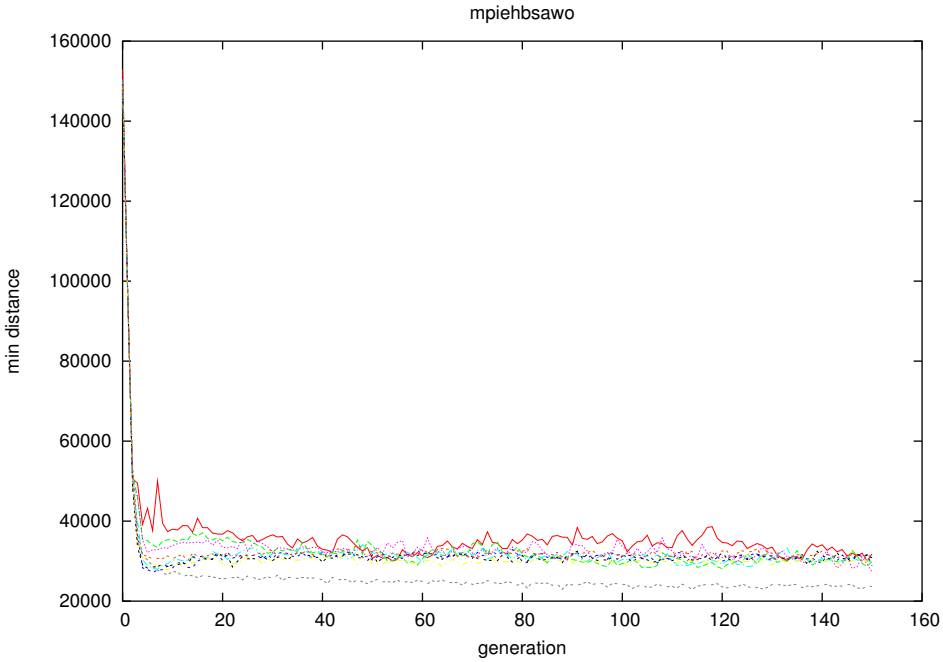

図 6.35: MPIEHBSAWO のパラメータ 5

表 6.4: MPIEHBSAWO の実験結果 2

都市数	プロセス数	送受信数	並列版の解
200	6	5	8284.6
200	7	5	4864.1
200	8	5	5562.4
400	6	10	14291.7
400	7	10	10221.9
400	8	10	8273.5
600	6	10	15428.0
600	7	10	15757.1
600	8	10	18500.4
800	6	10	18160.2
800	7	10	18956.0
800	8	10	17800.1
1000	6	20	23639.5
1000	7	20	30047.3
1000	8	20	28848.7

第7章　まとめと今後の課題

7.1　実験結果のまとめ

実験ではEHBSAWO、EHBSAWTのアルゴリズムについてまず、個々の戦略について調べ、その後適切なパラメータを調べ、その後にそれの並列版を調べた。EHBSAWTではcut数を増やすことで、実行時間を大幅に短くできるが、解の精度は悪くなる。たとえ、cut数を1にしても解の精度という点からみたとき、EHBSAWTがEHBSAWOを優ることはなかった。

初期集団の選択法としては、エリート戦略とトーナメント戦略が優れていて、EHMの作成法では重みづけ戦略が優れていた。前者のエリート戦略とトーナメント戦略は実行時間、解の精度ともにほぼ同等の性能を示した。

また、都市数が与えられたときの適切な集団数は大体その都市数の4分の1程度で、これ以上大きくなると計算時間がかかるだけでなく解の精度自体が低下する。また、都市数の変化によって解の集束にかかる世代数は大きく変動せず、大体１００世代も計算すればほぼ集束することが分かった。

EHBSAの並列化の実験は主にEHBSAWOで行なったが、最もよい集団の要素を送り、受け取ったほうは自分の持っている集団の要素の中の最も悪いものと交換するというエリート戦略が最も良い性能を示した。この並列化によって解の精度が1.3倍から1.6倍程度上昇した。並列化においては、プロセス数を多くし、送受信数をかなり少なめにとると解の精度が上昇することが多いという結果が出たが、ばらつきが大きく戦略の改善など、まださらなる工夫が必要である。

7.2　今後の課題

EHBSAにおける今後の課題としてはさまざなものが考えられるが、その中で特に大きいものを挙げる。

先ず1つはユーザーインターフェースがある。本研究では、結果を分析するのに簡単なユーティリティを作り、それを使ってデータを解析したがテキストデータをファイル名、ディレクトリ名を使って整理するという手法では、例えば、何回か同じパラメータを使い、その平均をとりたい、などという用途のときに使えない。また集団の状態を世代ごとに分析し、その結果によってパラメータ、戦略を変えるというようなことも、やりたかったのだが、本研究ではやれていない。これにはある程度しっかりした、例えばRDBのような形でデータを蓄え、取りだせるようなユーザーインターフェースが必要だ。

もうひとつにはアルゴリズムの改良がある。EHBSAは世代交代にかかる時間は通常のGAよりも短いが、解の精度自体は通常のGAに劣る。これは戦略にエリート戦略を用い、EHMを重みづけするというかなり強引に探索領域をせばめ

る戦略のため、解が局所解におちいってしまう可能性が高いせいである。また、並列化したときに解の精度が大幅に向上することがあるのは、全く異なるかたちの集団の要素を取り入れることで探索空間が拡大することが理由として考えられる。このようなことを考えると、アルゴリズムの改善のためには、探索空間を大きく取れるような戦略、パラメータを増やす必要があるだろう。

謝辞

本研究を進めるあたりに様々な方のお世話になりました。かなり自由にやらせてくれた上田 和紀教授、並列班の先輩方、ありがとうございました。特に稲垣良一氏には研究の環境整備から tex のテンプレートまでいろいろな面でお世話になりました。ここに深く感謝いたします。

また、OCaml というすばらしい処理系を作ってくれた INRIA の開発者の方々、また様々な素晴しいソフトを開発し、自由に使用させてくれているオープンソースコミュニティの開発者の方々に深く感謝いたします。

参考文献

[1] 平野廣美：応用事例で分かる遺伝的アルゴリズムプログラミング , パーソナルメディア , 1995.

[2] 伊庭斉志：遺伝的アルゴリズムの基礎 - GA の謎を解く - , オーム社 , 1994

[3] 平野廣美：C でつくるニューラルネットワーク , パーソナルメディア , 1991

[4] Shigeyoshi Tsutsui : Probabilistic Model-Building Genetic Algorithms in Permutation Representation Domain Using Edge Histogram , Proc. of the 7th Parallel Problem Solving from Nature - PPSN VII, pp.224-233 , 2002.

[5] Georges Harik : Linkage Learning via Probabilistic Modeling in the ECGA , Technical report , IIIiGAL Technical Report , No. 99010 , 1999.

[6] Georges R. Harik, G.Lobo , David E. Goldberg : The Compact Genetic Algorithm , Proceeding of the 1998 IEEE Conference on Evolutionary Computation, pp. 523-528, 1998

[7] 酒井大輔：ocaml 備忘録
`http://www.ueda.info.waseda.ac.jp/~sakai/ocaml/ocamlmemo.html`

[8] Xavier Leroy : The Objecttive Caml system release 3.08 Documentation and user's manual , 2004
`http://caml.inria.fr/ocaml/htmlman/`

[9] Peter S. Pacheco 著 , 秋葉博 訳：MPI 並列プログラミング , 培風館 , 2001.

[10] Marc Snir , Steve Otto , Steven Huss-Lederman , David Walker , Jack Dongarra : MPI - The Complete Reference Volume 1, The MPI Core second edition , The MIT Press , 1998.

[11] William Gropp , Steven Huss-Lederman , Andrew Lumsdaine , Ewing Lusk , Bill Nitzberg , William Saphir , Marc Snir : MPI - The Complete Reference Volume 2 , The MPI Extensions , The MIT Press , 1998.

付 録A　パラメーター一覧

本付録ではexecute_ehbsawo.ml、execute_ehbsawt.ml、execute_mpiehbsawo.ml、execute_mpiehbsawt.ml、read_ehbsawo.ml、read_ehbsawt.ml、read_mpiehbsawo.ml、read_mpiehbsawt.ml で用いるパラメータファイルで指定できるパラメータの一覧をサンプルと共に示す。

A.1　EHBSAWO

A.1.1　指定できるパラメータ

```
pop_num : 整数
city_num : 整数
max_generation : 整数
population_select : no_population_select
                    population_select_half
                    population_select_roulette
                    population_select_tournament
                    のいずれか
make_ehm_strategy : make_ehm_delta：整数 , make_ehm_weight：整数
                    のいずれか
use_twoopt : non_use_twoopt , use_twoopt
             のいずれか
```

A.1.2　サンプル

```
pop_num;150
city_num;200,400,600
max_generation;50
population_select;no_population_select,population_select_half
make_ehm_strategy;make_ehm_delta:0.04
use_twoopt;non_use_twoopt
```

A.2 EHBSAWT

A.2.1 指定できるパラメータ

```
pop_num : 整数
city_num : 整数
max_generation : 整数
cut_num : 整数
population_select : no_population_select
                    population_select_half
                    population_select_roulette
                    population_select_tournament
                    のいずれか
choose_template_strategy : choose_template_randomly
                           choose_template_roulette
                           choose_template_elite
                           のいずれか
choose_sampling_node_strategy : choose_node_randomly
                                choose_node_roulette
                                のいずれか
make_ehm_strategy : make_ehm_delta：整数 , make_ehm_weight：整数
                    のいずれか
use_twoopt : non_use_twoopt , use_twoopt
             のいずれか
```

A.2.2 サンプル

```
pop_num;150
city_num;200,400,600
max_generation;50
cut_num;5
population_select;no_population_select,population_select_half
choose_template_strategy;choose_template_randomly
choose_sampling_node_strategy;choose_node_randomly
make_ehm_strategy;make_ehm_delta:0.04
use_twoopt;non_use_twoopt
```

A.3 MPIEHBSAWO

A.3.1 指定できるパラメータ

```
pop_num : 整数
city_num : 整数
max_generation : 整数
process_num : 整数
population_select : no_population_select
                    population_select_half
                    population_select_roulette
                    population_select_tournament
                    のいずれか
make_ehm_strategy : make_ehm_delta：整数 , make_ehm_weight：整数
                    のいずれか
use_twoopt : non_use_twoopt , use_twoopt
             のいずれか
send_strategy : send_randomly , send_roulette
                send_elite , send_tournament
                のいずれか
receive_strategy : receive_randomly , receive_roulette
                   receive_elite , receive_tournament
                   のいずれか
send_num : 整数
```

A.3.2 サンプル

```
pop_num;100
city_num;200
max_generation;50
process_num;8
population_select;population_select_half
make_ehm_strategy;make_ehm_weight:0.04
use_twoopt;use_twoopt
send_strategy;send_randomly,send_roulette
receive_strategy;receive_randomly,receive_roulette
send_num;10
```

A.4 MPIEHBSAWT

A.4.1 指定できるパラメータ

```
pop_num : 整数
city_num : 整数
max_generation : 整数
cut_num : 整数
process_num : 整数
population_select : no_population_select
                    population_select_half
                    population_select_roulette
                    population_select_tournament
                    のいずれか
choose_template_strategy : choose_template_randomly
                           choose_template_roulette
                           choose_template_elite
                           のいずれか
choose_sampling_node_strategy : choose_node_randomly
                                choose_node_roulette
                                のいずれか
make_ehm_strategy : make_ehm_delta：整数 , make_ehm_weight：整数
                    のいずれか
use_twoopt : non_use_twoopt , use_twoopt
             のいずれか
send_strategy : send_randomly , send_roulette
                send_elite , send_tournament
                のいずれか
receive_strategy : receive_randomly , receive_roulette
                   receive_elite , receive_tournament
                   のいずれか
send_num : 整数
```

A.4.2 サンプル

```
pop_num;150
city_num;200
max_generation;50
cut_num;5
process_num;6
population_select;no_population_select
choose_template_strategy;choose_template_randomly
choose_sampling_node_strategy;choose_node_randomly
make_ehm_strategy;make_ehm_weight:0.04
use_twoopt;non_use_twoopt
send_strategy;send_randomly,send_roulette
receive_strategy;receive_randomly,receive_roulette
send_num;3,6,9
```

付 録B　OCamlMPI

B.1　はじめに

本付録ではOCamlMPIについて簡単に紹介する。なお本付録の内容は酒井の書いた参考文献[7]を一部抜粋し、修正したものである。OCaml自体については参考文献[8]などを、MPIについては参考文献[9][10][11]などを参照してほしい。

ocamlmpiはインストールされているものとする。また以下の作業は全てnfsでシェアされているディレクトリで行う。ocamlmpiのコンパイルは

```
ocamlc -I +ocamlmpi mpi.cma hoge.ml -o hoge
```

実行は

```
mpirun -np 3
```

で出来るとする。

B.2　基本的な関数

本付録でははは最も基本的な点対点の通信を見ていくが、その前にいくつかocamlmpiに定義されている変数や関数などを見ていく。当然全てmpiモジュール。なおMPI_InitとかMPI_Finalizeはない。これらは自動的にやってくれるが、barrier関数をプログラムの最初で呼ぼうとするとバグを起こすことがあるようだ。これは、内部で初期化されないままMPI_Barrier関数が呼ばれてしまうことが原因だ。

```
type communicator
type rank = int
val comm_world : communicator
external comm_size : communicator -> int = "caml_mpi_comm_size"
external comm_rank : communicator -> rank = "caml_mpi_comm_rank"
val barrier : communicator -> unit
external wtime : unit -> float = "coml_mpi_wtime"
```

ＭＰＩを知っている人は想像つくだろうが、とりあえず以下に説明する。

type communicator communicator のタイプ。communicator とはノードのグループ。つまりデータを交換できるプロセスの集まり。

type rank ノードのランク。ある communicator に属しているノードは０から順に正の整数値が割り当てられ、これをそのノードのランクという。

val comm_world グローバルな communicator。デフォルトでは全てのノードが参加している。

external comm_size communicator を引数に取り、その communicator の属しているノードの数を返す関数。

external comm_rank この関数を呼んだノードの与えられた communicator におけるランク数を返す関数。

val barrier 与えられた communicator の全ノードで同期をとる。

external wtime プログラム実行から数えた clock 時間を返す関数。

B.3 基本的な使いかたと C との比較

まずは基本的な点対点の通信を試してみる。点対点通信では送るデータを識別するのにタグという特別な整数値を使う。タグは ocamlmpi では次のように定義されている

```
type tag = int
```

しかし全ての整数値が使えるわけではなく、タグは０から３２７６７の間の整数値に限られる。このタグを用いて実際にメッセージをやりとりするには次のような send,receive という関数を使う。

```
val send : 'a -> rank -> tag -> communicator
val receive : rank -> tag -> communicator -> 'a
```

これを使ってプロセス０からプロセス１に"Hello World"という文字列を送ってみる。

```
(* mpi_test1.ml *)
open Mpi
let size = comm_size comm_world
let myrank = comm_rank comm_world
let _ =
 if myrank=0 then send "Hello,World" 1 0 comm_world;
 if myrank=1 then print_endline (receive 0 0 comm_world)
```

```
ocamlc -I +ocamlmpi mpi.cma mpi_test1.ml -o mpi_test1
mpirun -np 3 mpi_test1
Hello,World
```

これをCで書くと多分下のような感じになる。比べてもらえると分かると思うが、OCaml の方が短く簡単にかける。

```
/* mpi_test1.c */
#include <stdio.h>
#include <string.h>
#include <mpi.h>
int main(int argc, char **argv){
  int myid, numprocs;
  MPI_Status status;
  char msg[20];
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  if(myid == 0){
    strcpy(msg, "Hello,World");
    MPI_Send(msg, 20, MPI_CHAR, 1, 0, MPI_COMM_WORLD);
  }
  else if (myid == 1){
    MPI_Recv(msg, 20, MPI_CHAR, 0, 0, MPI_COMM_WORLD, &status);
    printf("%s\n", msg);
    fflush(stdout);
  }
  MPI_Finalize();
  return 0;
}
```

```
mpicc mpi_test1.c -o mpi_test1c
mpirun -np 3 mpi_test1c
Hello,World
```

後で見るようにこの send という関数はかなり汎用性が高く、点対点の送信関数としては最も一般的なものになる。

また、receive は任意の型を返すが、どこから送られてきたのか、何番のタグのデータだったのかは直接返すことはない（もちろん receive の中でタグと相手のランクを指定しているときは、簡単に調べられるのだが)。これらのデータが手軽に利用したいときには receive_status という関数がある。例を示す。

```
(* mpi_test2.ml *)
open Mpi
let size = comm_size comm_world
let myrank = comm_rank comm_world
let _ =
 if myrank=0 then send "Hello,World" 1 0 comm_world;
 if myrank=1 then
  begin
   let a,b,c = receive_status 0 0 comm_world in
    Printf.printf "rank:%d,tag:%d,%s\n" b c a
  end
```

```
ocamlc -I +ocamlmpi mpi.cma mpi_test2.ml -o mpi_test2
mpirun -np 3 mpi_test2
rank:0,tag:0,Hello,World
```

上のようにデータだけでなく、それがどこのノードの何番のタグのデータなのかを調べられる。これは any_tag,any_source という特別なタグとランクを使っているときに特に有用だ。any_tag と any_source はC言語のＭＰＩにあるようにあらゆるタグとソースにマッチする。以下に例を示す。

```
(* mpi_test3.ml *)
open Mpi
let size = comm_size comm_world
let myrank = comm_rank comm_world
let _ =
 if myrank=0 then
  begin
   send "Data1" 2 0 comm_world;
   send "Data2" 2 1 comm_world;
  end;
 if myrank=1 then
  begin
   send "Data3" 2 2 comm_world;
   send "Data4" 2 3 comm_world;
  end;
  if myrank=2 then
   begin
    print_endline (receive any_source any_tag comm_world);
    print_endline (receive any_source any_tag comm_world);
    let a,b,c = receive_status any_source any_tag comm_world and
        d,e,f = receive_status any_source any_tag comm_world in
     Printf.printf "rank:%d, tag:%d, %s\n" b c a;
     Printf.printf "rank:%d, tag:%d, %s\n" e f d;
   end
```

```
ocamlc -I +ocamlmpi mpi.cma mpi_test3.ml -o mpi_test3
mpirun -np 3 mpi_test3
Data1
Data2
rank:1, tag:2, Data3
rank:1, tag:3, Data4
```

このように any_source,any_tag を使った場合でも、相手ランクとタグが判別できる。send は最も一般的な送信関数であらゆるデータを送ることが出来るが、ocamlmpi には int,float,int_array,float_array 用の特別な送受信関数が用意されている。これらは send,recive を使うより高速である。とる型は次の通り。

```
val send_int : int -> rank -> tag -> communicator -> unit
val receive_int : rank -> tag -> communicator -> int
val send_float : float -> rank -> tag -> communicator -> unit
val receive_float : rank -> tag -> communicator -> float
val send_int_array : int array -> rank -> tag ->
  communicator -> unit
val receive_int_array : int array -> rank -> tag ->
  communicator -> unit
val send_float_array : float array -> rank -> tag ->
  communicator -> unit
val receive_falot_array : float_array -> rank -> tag ->
  communicator -> unit
```

注意して欲しいのは配列を受け取るときは受け皿用の配列を用意する必要があることだ。これらの関数を試してみる。

```
(* mpi_test4.ml *)
open Mpi
open Printf
let size = comm_size comm_world
let myrank = comm_rank comm_world
let _ =
 if myrank=0 then send_int 1 1 0 comm_world;
 if myrank=1 then printf "int: %d\n" (receive_int 0 0 comm_world);
 if myrank=0 then send_float 1.2 1 1 comm_world;
 if myrank=1 then printf "float: %f\n" (receive_float 0 1 comm_world);
 if myrank=0 then send_int_array [|1;2;3|] 1 2 comm_world;
 if myrank=1 then
  begin
   let a = Array.make 3 0 in
    receive_int_array a 0 2 comm_world;
```

```
    Array.iteri (fun x y -> printf "int_array[%d]: %d\n" x y) a;
  end;
 if myrank=0 then send_float_array [|1.2;3.1;4.5|] 1 3 comm_world;
 if myrank=1 then
  begin
   let a = Array.make 3 0.0 in
    receive_float_array a 0 3 comm_world;
    Array.iteri (fun x y -> printf "float_array[%d]: %f\n" x y) a;
  end
```

```
ocamlc -I +ocamlmpi mpi.cma mpi_test4.ml -o mpi_test4
mpirun -np 3 mpi_test4
int: 1
float: 1.200000
int_array[0]: 1
int_array[1]: 2
int_array[2]: 3
float_array[0]: 1.200000
float_array[1]: 3.100000
float_array[2]: 4.500000
```

点対点通信の最後として send によって文字列以外のさまざまなものを送受信してみる。もう一度 send,receive の型を確認しておくと

```
val send : 'a -> rank -> tag -> communicator -> unit
val receive : rank -> tag -> communicator -> 'a
```

である。'a というのは任意の型を取れるという意味なので、当然 send,receive は send_int,send_int_array などにも使える。そこで、mpi_test4.ml を send,receive を使って書き直してみる。

```
(* mpi_test5.ml *)
open Mpi
open Printf
let size = comm_size comm_world
let myrank = comm_rank comm_world
let _ =
 if myrank=0 then send 1 1 0 comm_world;
 if myrank=1 then printf "int: %d\n" (receive 0 0 comm_world);
 if myrank=0 then send 1.2 1 1 comm_world;
 if myrank=1 then printf "float: %f\n" (receive 0 1 comm_world);
 if myrank=0 then send [|1;2;3|] 1 2 comm_world;
 if myrank=1 then
  begin
```

```
  let a = receive 0 2 comm_world in
   Array.iteri (fun x y -> printf "int_array[%d]: %d\n" x y) a;
 end;
 if myrank=0 then send [|1.2;3.1;4.5|] 1 3 comm_world;
 if myrank=1 then
  begin
   let a = receive 0 3 comm_world in
    Array.iteri (fun x y -> printf "float_array[%d]: %f\n" x y) a;
  end
```

```
ocamlc -I +ocamlmpi mpi.cma mpi_test5.ml -o mpi_test5
mpirun -np 3 mpi_test5
int: 1
float: 1.200000
int_array[0]: 1
int_array[1]: 2
int_array[2]: 3
float_array[0]: 1.200000
float_array[1]: 3.100000
float_array[2]: 4.500000
```

ただ、このままだと receive で受け取る型が int array なのがソースに明示的に書いていないので少々気持ち悪い。そこで次のようにする。

```
(* mpi_test6.ml *)
open Mpi
open Printf
let size = comm_size comm_world
let myrank = comm_rank comm_world
let _ =
 if myrank=0 then send 1 1 0 comm_world;
 if myrank=1 then printf "int: %d\n" (receive 0 0 comm_world);
 if myrank=0 then send 1.2 1 1 comm_world;
 if myrank=1 then printf "float: %f\n" (receive 0 1 comm_world);
 if myrank=0 then send [|1;2;3|] 1 2 comm_world;
 if myrank=1 then
  begin
   let a : int array = receive 0 2 comm_world in
    Array.iteri (fun x y -> printf "int_array[%d]: %d\n" x y) a;
  end;
 if myrank=0 then send [|1.2;3.1;4.5|] 1 3 comm_world;
 if myrank=1 then
  begin
   let a : float array = receive 0 3 comm_world in
```

```
  Array.iteri (fun x y -> printf "float_array[%d]: %f\n" x y) a;
 end
```

```
ocamlc -I +ocamlmpi mpi.cma mpi_test6.ml -o mpi_test6
mpirun -np 3 mpi_test6
int: 1
float: 1.200000
int_array[0]: 1
int_array[1]: 2
int_array[2]: 3
float_array[0]: 1.200000
float_array[1]: 3.100000
float_array[2]: 4.500000
```

つまり上のようにして、変数を束縛するときに型の制約をかけておく。こうするとソースが読みやすくなる。

B.4 送受信の例

B.4.1 リストの送受信の例

リストを送受信してみる。

```
(* mpi_test7.ml *)
open Mpi
open Printf
let size = comm_size comm_world
let myrank = comm_rank comm_world
let _ =
 if myrank=0 then send [1;2;3] 1 0 comm_world;
 if myrank=1 then
  begin
   let a : int list = receive 0 0 comm_world in
   List.iter (fun a -> printf "List data: %d\n" a) a;
  end
```

```
ocamlc -I +ocamlmpi mpi.cma mpi_test7.ml -o mpi_test7
mpirun -np 3 mpi_test7
List data: 1
List data: 2
List data: 3
```

B.4.2 タプルの送受信の例

タプルを送受信してみる。

```
(* mpi_test8.ml *)
open Mpi
open Printf
let size = comm_size comm_world
let myrank = comm_rank comm_world
let _ =
 if myrank=0 then send (1.2,"test",3) 1 0 comm_world;
 if myrank=1 then
  begin
   let a : float * string * int = receive 0 0 comm_world in
   let b,c,d = a in
    printf "tuple data: %f, %s , %d\n" b c d
  end
```

```
ocamlc -I +ocamlmpi mpi.cma mpi_test8.ml -o mpi_test8
mpirun -np 3 mpi_test8
tuple data: 1.200000, test , 3
```

B.4.3 レコードの送受信の例

レコードを送受信してみる。

```
(* mpi_test9.ml *)
open Mpi
open Printf
let size = comm_size comm_world
let myrank = comm_rank comm_world
type complex = {re:float; im:float}
let _ =
 if myrank=0 then send {re=1.2;im=3.2} 1 0 comm_world;
 if myrank=1 then
  begin
   let a : complex = receive 0 0 comm_world in
    printf "complex: %f, %f\n" a.re a.im
  end
```

```
ocamlc -I +ocamlmpi mpi.cma mpi_test9.ml -o mpi_test9
mpirun -np 3 mpi_test9
complex: 1.200000, 3.200000
```

B.4.4　バリアントの送受信の例

バリアントを送受信してみる。

```
(* mpi_test10.ml *)
open Mpi
open Printf
let size = comm_size comm_world
let myrank = comm_rank comm_world
type var1 = Positive | Negative
type var2 = Data1 of int * int * int | Data2 of float * float * float
let _ =
 if myrank=0 then send Negative 1 0 comm_world;
 if myrank=1 then
  if Positive = (receive 0 0 comm_world)
  then printf "data is positive\n"
  else printf "data is negative\n";
 if myrank=0 then send (Data1(1,2,3)) 1 1 comm_world;
 if myrank=1 then
  let a : var2 = receive 0 1 comm_world in
   match a with
   | Data1(x,y,z) -> printf "Data1:%d,%d,%d\n" x y z
   | Data2(x,y,z) -> printf "Data2:%f,%f,%f\n" x y z
```

```
ocamlc -I +ocamlmpi mpi.cma mpi_test10.ml -o mpi_test10
mpirun -np 3 mpi_test10
data is negative
Data1:1,2,3
```

B.4.5　インスタンスの送受信の例

インスタンスを送受信してみる。

```
(* mpi_test11.ml *)
open Mpi
open Printf
class point(x_init,y_init) =
object
  val mutable x = x_init;
  val mutable y = y_init;
  method get_x = x;
  method get_y = y;
  method incr_x = x <- x+1;
```

```
  method incr_y = y <- y+1;
end
let size = comm_size comm_world
let myrank = comm_rank comm_world
let _ =
 if myrank=0 then send (new point (2,4)) 1 0 comm_world;
 if myrank=1 then
  begin
   let a : point = receive 0 0 comm_world in
    printf "object x:%d,y:%d\n" a#get_x a#get_y;
    a#incr_x; a#incr_x; a#incr_y;
    printf "object x:%d,y:%d\n" a#get_x a#get_y;
  end
```

```
ocamlc -I +ocamlmpi mpi.cma mpi_test11.ml -o mpi_test11
mpirun -np 3 mpi_test11
object x:2,y:4
object x:4,y:5
```

B.4.6 インスタンスのリストの送受信の例

インスタンスのリストを送受信してみる。

```
(* mpi_test12.ml *)
open Mpi
open Printf
class point(x_init,y_init) =
object
  val mutable x = x_init;
  val mutable y = y_init;
  method get_x = x;
  method get_y = y;
  method incr_x = x <- x+1;
  method incr_y = y <- y+1;
end
let size = comm_size comm_world
let myrank = comm_rank comm_world
let _ =
 if myrank=0 then
  begin
   let a = new point(2,4) and b = new point(3,1)
   and c = new point(5,7) in
    send [a;b;c] 1 0 comm_world;
```

```
 end;
if myrank=1 then
 begin
  let a : point list = receive 0 0 comm_world in
   List.iter (fun o ->
    printf "List data: %d %d\n" o#get_x o#get_y) a;
 end
```

```
ocamlc -I +ocamlmpi mpi.cma mpi_test12.ml -o mpi_test12
mpirun -np 3 mpi_test12
List data: 2 4
List data: 3 1
List data: 5 7
```

B.4.7 関数の送受信の例

関数を送受信してみる。

```
(* mpi_test13.ml *)
open Mpi
open Printf
let size = comm_size comm_world
let myrank = comm_rank comm_world
let _ =
 if myrank=0 then send (fun a b -> a + b) 1 0 comm_world;
 if myrank=1 then
  begin
   let f : int->int->int = receive 0 0 comm_world in
    printf "function result: %d\n" (f 2 3)
  end
```

```
ocamlc -I +ocamlmpi mpi.cma mpi_test13.ml -o mpi_test13
mpirun -np 3 mpi_test13
function result: 5
```