免疫アルゴリズムのための Immune 言語の開発

Development of Immune Language for Immune Algorithm

森山 賀文　Yoshifumi MORIYAMA
鹿児島大学 工学部 情報工学科
Department of Information and Computer Science, Faculty of Engineering, Kagoshima University

飯村 伊智郎　Ichiro IIMURA
熊本県立大学 総合管理学部 総合管理学科
Department of Administration, Faculty of Administration, Prefectural University of Kumamoto

中山　茂　Shigeru NAKAYAMA
鹿児島大学 工学部 情報工学科
Department of Information and Computer Science, Faculty of Engineering, Kagoshima University

要　旨

組合せ最適化問題の近似的解法である遺伝的アルゴリズム（Genetic Algorithm: GA）は，進化の過程において探索を停滞させる過剰な収束を生じる場合がある。それに対して，多様性のある抗体産生機構と増え過ぎた類似抗体の産生を抑制する自己調節機構により過剰な収束の回避を期待できる免疫アルゴリズム（Immune Algorithm: IA）が提案されているが，そのIAを実装する場合，免疫システムに関する知識が必要であり，それらをシミュレートするための煩雑なコーディング作業をプログラム開発者が行う必要がある。そこで本論文では，IAを実装する際のコーディングの煩雑さを軽減し，免疫システムに関する知識を有しない一般ユーザでもIAを容易に適用できる環境構築を目的として，IAのためのImmune言語を提案する。

Abstract

Genetic algorithm (GA) is often used to calculate quasi-optimal solution of combinational optimization problems (COPs). However, in the GA, undesirable phenomenon of excess convergence can often occur. Once the excess convergence occurs, the search by the GA becomes meaningless. Immune algorithm (IA) is expected to avoid the excess convergence and maintain the diversity of antibodies. However, knowledge concerning immune system is indispensable for programmers. Therefore, complex coding work is demanded from programmers. In this paper, we propose "Immune Language" in order to reduce the complex coding work and to construct an environment which general users who are not specialists can easily apply IA to COPs.

1. はじめに

組合せ最適化問題とは，数多くの組合せの中から最適な解を求める問題であり，問題の規模が大きくなるにつれてその組合せ数は爆発的に増大する。そのため現実的な時間内に厳密な解を求めることが困難となる場合がある。そこで，遺伝的アルゴリズム（Genetic Algorithm: GA）などの近似的解法を用いることによって，近似的な最適解（準最適解）を可能な限り高速に求める研究がなされている。筆者らはこれまでに，コーディングの煩雑さを軽減し，一般ユーザでも容易に利用できる環境構築を目的として，GAに特化したGene言語[1]を開発してきた。しかしながら，このGAは，進化の過程において集団内で同じ個体が急増するなどして，集団の多様性が失われる過剰な収束[2][3]という探索を停滞させる現象が生じる場合がある。それに対して，大局的最適解を含む複数の局所的最適解を得ることを目的とした免疫アルゴリズム（Immune Algorithm: IA）[4]~[9]がある。このIAは，生体の免疫システムを模倣したアルゴリズムであり，多様性のある抗体産生機構と，必要以上に増え過ぎた類似抗体の産生を抑制する自己調節機構により多様性のある抗体を産生し，過剰収束の回避を期待できる。

しかしながら，IAはGA同様，選択や交叉，突然変異などの遺伝的操作に関する知識が必要であり，それらをシミュレートするための煩雑なコーディング作業をプログラム開発者が行う必要がある。コーディングの煩雑さを軽減する策としては，IA用のクラスを開発しそれらをプログラム開発者に提供する，またはIA専用のDLL（Dynamic Link Library）を開発し提供するなどが容易に考えられる。しかしながら，Java言語が詳しくないと，IA用のクラスの使用や専用のDLLの使用さえもIAが複雑なために難しいと思われる。そこで本論文では，そのコーディングの煩雑さを軽減し，一般ユーザでもIAを容易に適用できる環境構築を目的としてIAのためのImmune言語を提案する。そして，組合せ最適化問題の代表例であるナップザック問題（Knapsack Problem: KP）と巡回セールスマン問題（Traveling Salesman Problem: TSP）に対して実際に適用することにより，Immune言語の効果を検証する。提案するImmune言語は，Java言語にIA処理に関するキーワードを追加したもので，Java言語の基本的な知識さえあれば容易にIAを取り扱うことができる。

GA と IA とを比較すると，その起源が異なるが，計算機に実装した場合，基本的な処理過程は類似した部分が多いという特徴があるため，Gene 言語同様，Espace 言語 [10] [11] との融合により，分散並列 IA に拡張可能な Immune 言語への展開も期待できる。情報文化との関連性としては，Java 言語の初心者であっても提案する Immune 言語による IA のプログラミングは容易であると考え，今後，先に提案した Espace 言語との融合により，グリッドコンピューティングのような分散並列処理を加えて分散並列 IA とすることで一般ユーザも参加できる点が挙げられる。

以下，IA の概要，提案する Immune 言語の詳細およびコーディング事例について述べる。その後，Immune 言語を KP と TSP へ適用した場合の実験結果を示し，その結果をもとに考察を加える。

2. 免疫アルゴリズムの概要

2.1 概要

IA とは，獲得免疫と呼ばれる免疫システムをもとに考え出されたアルゴリズムである。以下，まず免疫システムについて説明し，次に IA の概要を説明する。

生体には，未知の抗原に対応するための獲得免疫と呼ばれる免疫システムが備わっている。この免疫システムには，抗体産生細胞，記憶細胞，サプレッサー T 細胞と呼ばれる重要な役割を担う細胞がある。抗体産生細胞とは抗体を産生する細胞であり，交叉や突然変異を繰り返すことで多種多様な抗原に特異的に反応する抗体を産生することができる。抗原を排除する有益な抗体はさらに産生され，一部は記憶細胞として記憶され 2 度目の侵入ではより迅速に対応する。また，体内に必要以上に増えすぎた抗体はサプレッサー T 細胞により抑制され定常状態に戻される。この未知の抗原に遭遇することで新たな免疫が備わることを獲得免疫と呼び，「麻疹」や「おたふく風邪」に一度かかると二度かからない，もしくはかかったとしても軽度で済むのはこの獲得免疫によるものである。

IA では，最適化問題における目的関数や前提条件を抗原 (antigen)，解候補を抗体（antibody）で表し，探索により発見された解を記憶細胞（memory cell）として記憶する。抗体の集合を母集団 (population) と呼び，各抗体は設計変数をコーディングした遺伝子（gene）の一次元配列で表現される。抗原と抗体の親和性，つまり目的関数に対する抗体の評価は親和度（affinity）で表され，抗体の設計変数を読み出し目的関数の値を計算することで得られる。また，抗体同士の類似性を表す指標として類似度（similarity）を設定し，ある程度似かよった抗体を類似抗体とみなす。さらに，母集団に占める類似抗体の割合を濃度（density）とし，ある濃度に達した抗体はサプレッサー T 細胞 (suppressor T cell) により抑制 (suppression) される。

世代が更新されるごとに母集団内で親和度の高い抗体の選択 (selection) が行われ，他は淘汰される。淘汰されずに残った抗体を親として，交叉（crossover）や突然変異（mutation）

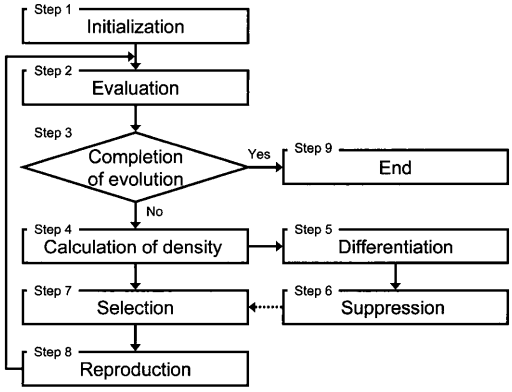


図 1 免疫アルゴリズムの処理手順
Fig. 1 Processing procedure of immune algorithm.

により次世代の母集団を形成する。このとき，親和度に比例し濃度に反比例する期待値（expectation）に応じて抗体を増殖することにより，母集団の多様性を維持しつつ解探索を行うことができる。

2.2 処理手順

IA の処理手順を図 1 に示し，各処理の詳細を以下に説明する。以下で使用される T_Ψ, T_Θ, $T^S_{\Psi 1}$, $T^S_{\Psi 2}$ は抗体の多様性に関する閾値であり，任意に設定できる。実際の免疫システムでは，抗体産生細胞が遺伝子の再構成を行い多様性のある抗体を産生するが，本研究では抗体産生細胞と抗体を同じ細胞とみなし，抗体自身が交叉や突然変異を繰り返すことで，多様な抗体を産生するものとする。また，3.1 節で述べるように，親和度計算メソッドと類似度計算メソッドは，引数に細胞名と細胞番号を設定し汎用性を持たせる必要があるため，記憶細胞とサプレッサー T 細胞を，抗体同様，遺伝子配列で表現する。

Step 1〔初期化〕 第 t 世代の抗体群が形成する母集団を $P(t)$ とする。まず，ランダムに初期世代（t=0）の母集団 $P(0)$ を生成する。KP の場合，抗体には全荷物についてナップザックに入れる (1) か入れない (0) かをバイナリ型でコーディングする。つまり，1 または 0 を一次元的に並べた配列を抗体とする。また，TSP の場合，抗体には整数で表現された都市名を巡回する順番にコーディングする。つまり，都市名を一次元的に並べた配列を抗体とする。

Step 2〔評価〕 現在の母集団 $P(t)$ 内の各抗体 v に対して親和度 Φ_v を計算する。KP の場合，親和度は抗体から荷物の組合せを読み出し，荷物の価値の合計を計算することで得られ，価値の合計が高い抗体ほど高くなる。TSP の場合，親和度は抗体から都市の巡回路を読み出し，巡回路長を計算することで得られ，巡回路長が短い抗体ほど高くなる。提案する Immune 言語では，親和度計算メソッドは問題によって異なるためにユーザ自身が定義するものとする。

Step 3〔終了判定〕 予め指定された進化の終了条件を満たしていれば Step 9〔進化終了〕へ，そうでなければ Step 4〔濃度計算〕へ進む。

Step 4〔濃度計算〕 各抗体 v について $P(t)$ 内の他の抗体 w と

の類似度Ψ_{vw}を計算する。KP の場合，類似度は 2 つの抗体間に共通する荷物の個数をもとに計算され，共通する荷物が多いほど高くなる。TSP の場合，類似度は 2 つの抗体間に共通する経路（枝）数をもとに計算され，共通した経路（枝）が多いほど高くなる。提案する Immune 言語では，類似度計算メソッドは問題によって異なるためにユーザ自身が定義するものとする。

また$\Psi_{vw} \geq T_{\Psi}$のときv, wを類似抗体とみなし，式（1）で定義される濃度Θ_vを求める。但し，N_vは総抗体数でありπ_{vw}は式（2）で定義される単位ステップ関数である。

$$\Theta_v = \frac{1}{N_v} \sum_{w=1}^{N_v} \pi_{vw} \tag{1}$$

$$\pi_{vw} = \begin{cases} 1 & (\Psi_{vw} \geq T_{\Psi}) \\ 0 & (otherwise) \end{cases} \tag{2}$$

Step 5〔分化〕 $\Theta_v \geq T_{\Theta}$のときvは記憶細胞候補v^*へ分化する。記憶細胞が上限数Mに達するまでは，v^*をそのまま記憶細胞mに分化する。上限数Mに達した場合は，v^*とmとの類似度Ψ_{v^*m}が最も高い記憶細胞mを選び出し，$\Phi_{v^*} > \Phi_m$のときのみv^*でmを更新する。v^*がmに分化した場合，v^*は類似抗体を抑制するサプレッサー T 細胞sに分化する。

Step 6〔抑制〕 $P(t)$内の各抗体vとサプレッサー T 細胞sとの類似度Ψ_{vs}を計算し，$\Psi_{vs} \geq T^S_{\Psi 1}$のとき$v$を$P(t)$から消滅させる。その後，消滅した抗体に代わる新しい抗体を Step 1 と同じ方法で産生し，Step 2, Step 4 と同じ計算処理を行う。

Step 7〔選択〕 $P(t)$内でΦ_vの低い抗体を淘汰する。残った抗体について式（3）で定義される期待値E_vを計算し，E_vに応じたルーレット選択により親抗体p_1, p_2を選び出す。但し，N_{sp}は淘汰されずに残った抗体数，N_Sはサプレッサー T 細胞の総数であり，Ξ_{vs}は，式（4）で定義されるステップ関数である。

$$E_v = \frac{\Phi_v}{\Theta_v \sum_{w=1}^{N_{sp}} \Phi_w} \prod_{s=1}^{N_S} (1 - \Xi_{vs}) \tag{3}$$

$$\Xi_{vs} = \begin{cases} \Psi_{vs} & (\Psi_{vs} \geq T^S_{\Psi 2}) \\ 0 & (otherwise) \end{cases} \tag{4}$$

Step 8〔生殖〕 親抗体として選ばれたp_1, p_2に交叉，突然変異を作用させ 2 つの子抗体を産生する。親抗体の選出と子抗体の産生を繰り返し，淘汰された抗体と同じ数だけ補充し次世代の母集団$P(t+1)$を生成する。その後，Step 2〔評価〕へ戻る。

Step 9〔進化終了〕 予め指定された条件を満足したため，進化を終了する。

3. 提案する Immune 言語の仕様およびそのコーディング事例

Immune 言語とは，Java 言語に **immune** キーワードによる **immune** 構文が追加定義されたものである。この **immune** 構文では，IA 特有の処理を自動補完するために必要な情報を指定することになる。Immune 言語で記述されたプログラムは，筆者らが開発した Immune 言語コンパイラにより Java 言語のプログラムに変換され，その後 Java 言語コンパイラにより Java バイトコードとなる。なお，Immune 言語のような新たなプログラミング言語の提案は，分散並列処理の分野においては OpenMP や HPF（High Performance Fortran）など種々のものが提案されているが，IA に関しては筆者らの知る限りにおいて従来手法は存在しない。

以下，提案する Immune 言語の仕様について述べ，その後 Immune 言語による開発の流れについて述べる。

3.1 文法

Immune 言語では，IA の処理を開始するための情報と，対象とする組合せ最適化問題に依存する情報を記述し，それらの情報を **immune** 構文で指定する。Immune 言語コンパイラは，この **immune** 構文が指定する情報をもとに IA の処理に必要なメソッド等を補完して Java 言語プログラムを自動生成する。

以下，Immune 言語で記述すべき「IA 処理開始メソッドの呼び出し」，「前提条件変数の定義」，「親和度メソッドの定義」，「類似度メソッドの定義」，「**immune** 構文による必要情報の指定」について述べ，Immune 言語の各文法を，今回評価の対象とした整数型遺伝子が使われる TSP とバイナリ型遺伝子が使われる KP を例に説明する。

IA 処理開始メソッドの呼び出し　IA の処理を開始するためのメソッドを呼び出す。ここではメソッドの呼び出しのみを行い，メソッド自体の定義は Immune 言語コンパイラが行う。

開始メソッドの引数には抗体の遺伝子数，抗体数，1 回の実行で更新される世代数，探索結果を保存するためのファイル名がある。世代数とファイル名は省略することができ，世代数が省略された場合は 1 回の実行で 1 世代ごとの更新となり，ファイル名が省略された場合は探索結果は保存されない。

抗体の遺伝子数を **nGenes**，抗体数を **nAntibodies**，1 回の実行で更新される世代数を **nSteps**，探索結果を保存するためのファイル名を **fName** とし，開始メソッド名を **start** としたオーバーロード記述例を以下に示す。

```
// 記述例 1
start(nGenes, nAntibodies,
                    nSteps, fName);
// 記述例 2
start(nGenes, nAntibodies, nSteps);
// 記述例 3
```

```
start(nGenes, nAntibodies, fName);
// 記述例 4
start(nGenes, nAntibodies);
```

前提条件変数の定義　組合せ最適化問題の前提条件は対象とする問題によって異なるため，前提条件を格納するための変数をユーザが定義する必要がある。例えば，KP の場合は各荷物の質量および価値が前提条件となり，TSP の場合は各都市の位置を表す x 座標，y 座標が前提条件となる。前提条件は，親和度の計算処理と視覚的な探索結果の描画処理のために用いられる。

KP を対象とした **int** 型変数名を **luggage**，TSP を対象とした **double** 型変数名を **city** とした記述例を以下に示す。

```
// ナップザック問題の記述例
int luggage[][] = {{50, 10}, ...
                   ... , {100, 70}};
// 巡回セールスマン問題の記述例
double city[][] = {{37, 52}, ...
                   ... , {30, 40}};
```

親和度メソッドの定義　親和度を計算する処理は，対象とする組合せ最適化問題の目的関数や前提条件に依存し，問題によって処理が異なるため，ユーザが問題に応じて定義する必要がある。例えば，KP の場合は重量と価値をもとに，TSP の場合は各都市間の距離をもとに親和度を算出することになる。

親和度メソッドは，特定の細胞に対して 1 世代当たり数回呼び出されるため，計算対象となる細胞情報が必要となる。そのため，親和度メソッドの引数に細胞名と細胞番号を設定する必要があり，筆者らが既に提案した Gene 言語とは異なり，「抗体を格納する変数」が省略できる。細胞名には抗体，記憶細胞またはサプレッサー T 細胞のうちいずれかの細胞を指定し，細胞番号には各細胞に割り振られている細胞の番号を指定する。ここで，細胞を格納する変数は，何番目の細胞の遺伝子座がいくつの遺伝子なのかを指定できるようにするために，配列の第 1 添え字を細胞番号，第 2 添え字を遺伝子座とした二次元配列で定義する。バイナリ型 IA の場合は **boolean** 型の二次元配列，整数型 IA の場合は **int** 型の二次元配列となる。

細胞名を **cName**，細胞番号を **nCell**，親和度の定義メソッド名を **setAffinity** とし，**double** 型の親和度を戻り値として返す記述例を以下に示す。

```
// ナップザック問題の記述例
public double setAffinity(
      boolean[][] cName, int nCell) {
  double affinity = 0.0d;
  int weight = 0;
  int value = 0;
  int limit = 550;
  for(int g = 0; g < nGenes; g++) {
    if(cName[nCell][g]) {
      weight += luggage[g][0];
      value += luggage[g][1];
    }
  }
  affinity = value;
  if(weight > limit)
       affinity = affinity / weight;
  return affinity;
}
// 巡回セールスマン問題の記述例
public double setAffinity(
          int[][] cName, int nCell) {
  double affinity = 0.0d;
  double len = 0;
  for(int g = 0; g < nGenes; g++) {
    double dx;
    double dy;
    int gNext = g + 1;
    if(gNext == nGenes) gNext = 0;
    dx = city[cName[nCell][g]][0]
     - city[cName[nCell][gNext]][0];
    dy = city[cName[nCell][g]][1]
     - city[cName[nCell][gNext]][1];
    len += Math.sqrt(Math.pow(dx, 2)
                + Math.pow(dy, 2));
  }
  affinity = 1.0d / len;
  return affinity;
}
```

類似度メソッドの定義　類似度を計算する処理は遺伝子をコーディングする手法に依存し，対象とする組合せ最適化問題によって処理が異なるため，ユーザが問題に応じて定義する必要がある。例えば，KP の場合は同じ荷物の有無をもとに，TSP の場合は都市間の繋がりをもとに類似度を算出することになる。

類似度メソッドは，特定の 2 つの細胞に対して 1 世代当たり数回呼び出されるため，計算対象となる 2 つの細胞情報が必要となる。そのため，類似度メソッドの引数に 2 組の細胞名と細胞番号を設定する必要があり，筆者らが既に提案した Gene 言語とは異なり，「抗体を格納する変数」が省略できる。細胞名には抗体，記憶細胞またはサプレッサー T 細胞のうちいずれかの細胞を指定し，細胞番号には各細胞に割り振られている細胞の番号を指定する。ここで，細胞を格納する変数は，何番目の細胞の遺伝子座がいくつの遺伝子なのかを指定できるようにするために，配列の第 1 添え字を細胞番号，第 2 添え字を遺伝子座とした

二次元配列で定義する。バイナリ型IAの場合は**boolean**型の二次元配列，整数型IAの場合は**int**型の二次元配列となる。

1つの細胞の名前を**cName1**，番号を**nCell1**，もう1つの細胞の名前を**cName2**，番号を**nCell2**，類似度の定義メソッド名を**setSimilarity**とし，**double**型の類似度を戻り値として返す記述例を以下に示す。

```
//ナップザック問題の記述例
public double setSimilarity(
   boolean[][] cName1, int nCell1,
   boolean[][] cName2, int nCell2) {
  double similarity = 0.0d;
  int commonLug = 0;
  for(int g = 0; g < nGenes; g++) {
    if(cName1[nCell1][g]
              == cName2[nCell2][g])
                    commonLug++;
  }
  similarity = (double)
                commonLug / nGenes;
  return similarity;
}
//巡回セールスマン問題の記述例
public double setSimilarity(
       int[][] cName1, int nCell1,
       int[][] cName2, int nCell2) {
  double similarity = 0.0d;
  int commonPath = 0;
  int commonPath_Rev = 0;
  for(int g = 0; g < nGenes; g++) {
    int gNext1 = g + 1;
    if((g + 1) == nGenes)gNext1 = 0;
    int gSame = 0;
    while(cName1[nCell1][g]
          != cName2[nCell2][gSame])
                    gSame++;
    int gNext2 = gSame + 1;
    if(gNext2 == nGenes) gNext2 = 0;
    int gNext2_Rev = gSame - 1;
    if(gNext2_Rev < 0)
          gNext2_Rev = nGenes - 1;
    if(cName1[nCell1][gNext1]
        == cName2[nCell2][gNext2])
                  commonPath++;
    if(cName1[nCell1][gNext1]
      == cName2[nCell2][gNext2_Rev])
                commonPath_Rev++;
  }
  if(commonPath < commonPath_Rev)
        commonPath = commonPath_Rev;
  similarity =
       (double) commonPath / nGenes;
  return similarity;
}
```

immune構文による必要情報の指定　**immune**構文では，IAの型名，つまりバイナリ型IAの場合は**boolean**型，整数型IAの場合は**int**型と，これまで定義してきたメソッドや変数に関する情報を指定する。Immune言語コンパイラは，この情報をもとにIA処理に必要なメソッド等を補完してJava言語プログラムを自動生成する。

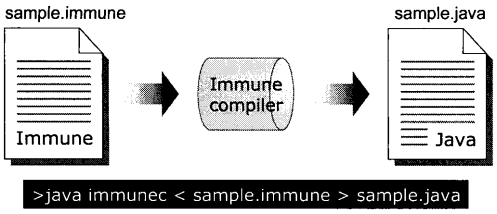
「IA処理開始メソッドの呼び出し」で述べたIAの処理を開始するために呼び出すメソッド名を**start**，「親和度メソッドの定義」で述べたIAの親和度の算出方法を定義したメソッド名を**setAffinity**，「類似度メソッドの定義」で述べたIAの類似度の算出方法を定義したメソッド名を**setSimilarity**，「前提条件変数の定義」で述べた対象とする組合せ最適化問題に関する前提条件を格納する変数名をKPの場合**luggage**，TSPの場合**city**とした記述例を以下に示す。

```
//ナップザック問題の記述例
immune start(boolean, setAffinity,
             setSimilarity, luggage);
//巡回セールスマン問題問題の記述例
immune start(int, setAffinity,
               setSimilarity, city);
```

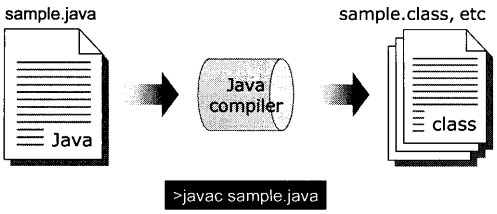
3.2　Immune言語コンパイラとImmune言語によるプログラム開発の流れ

本研究では，Immune言語で記述されたプログラムをJava言語のプログラムに変換するためのImmune言語コンパイラ“immunec”を開発した。Immune言語コンパイラは，Gene言語コンパイラ“genec”同様，UCLAのComputer Science Departmentによる“The Cooperative Database Project Web Page”[(1)]で公開されている“java1_4c.jj”をベースに，Immune言語で必要な字句や構文が追加定義されたもので，Java言語のすべての文法(JDK1.4仕様)に則した構文解析が可能である。

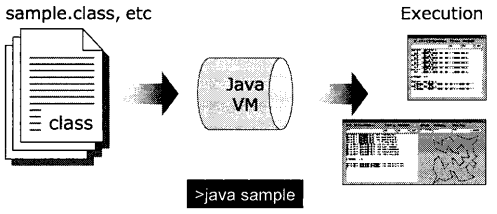
図2にImmune言語による開発の流れを示す。3.1節で述べたImmune言語の文法に基づいて記述されたプログラムは，Immune言語コンパイラを通すことで構文解析が行われ，文法的にミスがなければIA処理に必要なメソッド等が補完されたJava言語プログラムに変換される。補完されるメソッドは，免疫システム特有の処理に関するメソッドと，GUI（Graphical User Interface）による視覚的な結果表示を行うためのメソッドである。その後，変換されたJava言語プログラムはJava言語コンパイラを通すことでJavaバイトコードに変換され，Javaバイトコードが Java仮想マシン上で実行されることになる。



(a) Immune file to Java file by Immune compiler "immunec".



(b) Java file to class files by Java compiler "javac".



(c) Executing class files on Java VM.

図 2 Immune 言語による開発の流れ
Fig. 2 Procedure of developement in Immune language.

4. 実験

本節では，Immune 言語で記述されたプログラムと，そのプログラムを Immune 言語コンパイラで変換した Java 言語で記述されたプログラムとを比較することにより，Immune 言語による開発効率を考察する。本実験では，組合せ最適化問題の代表例である KP と TSP を対象に，それぞれ各言語により記述された IA プログラムのステップ数を比較する。また，ステップ数ではコーディングする際の書式によって多少増減することが考えられるため，併せてプログラムのファイルサイズによる比較を行うものとする。

さらに，被験者 7 名を対象として，Immune 言語を用いて KP と TSP を解くプログラムを実際に作成してもらう実験を行い，その後，Java 言語の理解度や IA に関する知識，Immune 言語の利便性に関する 5 段階評価と自由記述によるアンケート調査を行った。このアンケート結果をもとに，免疫システムに関する知識の有無と Immune 言語を用いた IA の取り扱いの関係について考察する。

参考までに，KP と TSP について，Immune 言語プログラムの実行結果の画面例を図 3，図 4 に，それらのもととなる Immune 言語で記述されたプログラムを付録の A.1 節および A.2 節にそれぞれ示す。

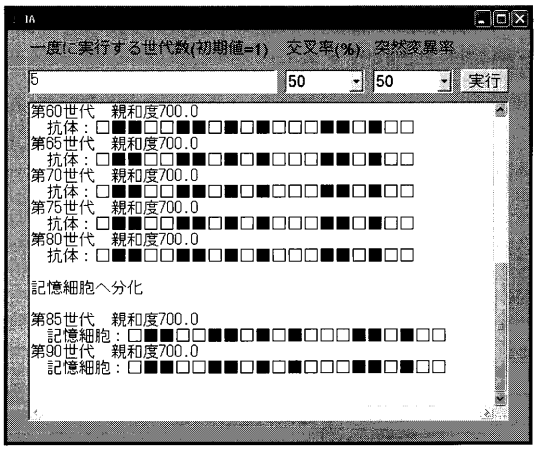


図 3 KP を対象とした Immune 言語プログラムの実行画面例
Fig. 3 An executing example of a Immune language program for KP.

4.1 Java 言語との比較実験と考察

Immune 言語で記述されたプログラムと Java 言語で記述されたプログラムとを，ステップ数とファイルサイズで比較した結果を図 5 に示す。

まず，KP をバイナリ型 IA で解くためのプログラムをステップ数で比較すると，Immune 言語での記述に要した 34 ステップは，Java 言語での記述に要した 403 ステップの約 1/12 倍であることが分かる。また，プログラムのファイルサイズについて比較すると，Immune 言語での記述に要した 1.23KB は，Java 言語での記述に要した 13.4KB の約 1/11 倍であることが分かる。なお，交叉は一点交叉とし，交叉率によって作用させる抗体数を決定する。突然変異は各遺伝子をランダムに対立遺伝子で置き換える操作を採用し，交叉により産生された抗体に対して突然変異を作用させる。突然変異を作用させる確率は突然変異率によって決定する。荷物の組合せがナップザックの許容重量を超えた抗体は，荷物の価値を総重量で除した値を親和度とした。

次に，TSP を整数型 IA で解くためのプログラムをステップ数で比較すると，Immune 言語での記述に要した 45 ステップは，Java 言語での記述に要した 623 ステップの約 1/14 倍であることが分かる。また，プログラムのファイルサイズについて比較すると，Immune 言語での記述に要した 2.09KB は，Java 言語での記述に要した 22.0KB の約 1/11 倍であることが分かる。なお，交叉は順序交叉とし，交叉率によって作用させる抗体数を決定する。突然変異はランダムに選んだ 2 つの遺伝子を入替える操作を採用し，交叉により産生された抗体に対して突然変異を作用させる。突然変異を作用させる確率は突然変異率によって決定する。また，これら遺伝的オペレータとは別に，経験的な解の改良として，2-opt 法という局所的な改良法を Immune 言語コンパイラが自動的に組み込んでいる。

以上の結果から，Immune 言語を用いることでプログラム開発者によるコーディングのステップ数，ファイルサイズ共に大幅に削減でき，IA の開発効率を向上できると考えられる。さらに，Immune 言語コンパイラにより変換されたプログラムは Java 言語で記述されているため，Java 言語の知識さえ

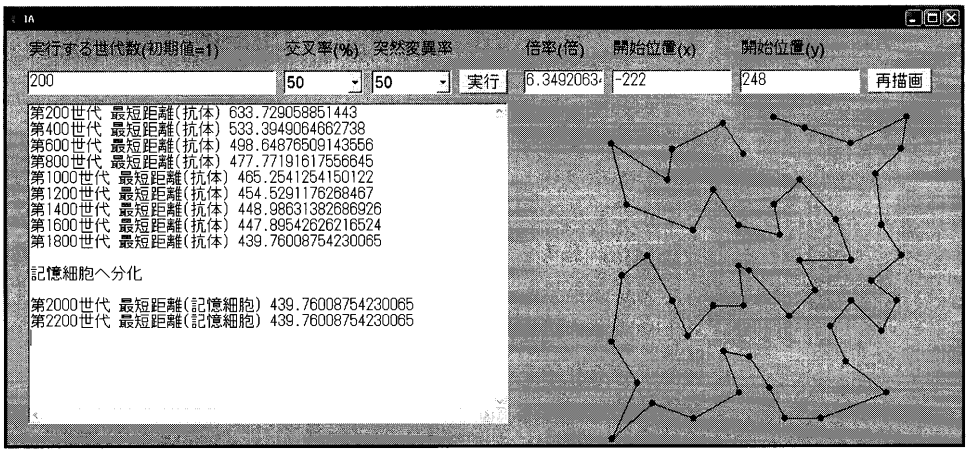


図 4　TSP を対象とした Immune 言語プログラムの実行画面例
Fig. 4　An executing example of a Immune language program for TSP.

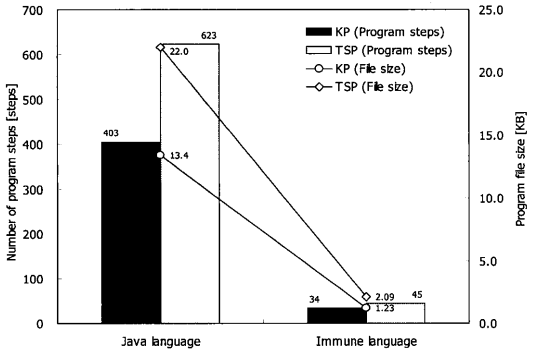


図 5　Java 言語と Immune 言語におけるプログラム・ステップ数とファイル・サイズの比較
Fig. 5　Comparison of the proguram step numbers and file sizes in "Java language" and "Immune language".

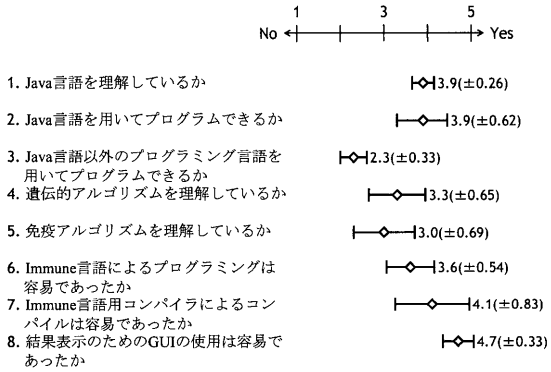


図 6　アンケート結果
Fig. 6　Results of questionnaires.

あればプログラムの変更も可能であり，容易に IA を取り扱うことができる。そのため，IA による組合せ最適化問題の解法に対するハードルを下げ，より多くの一般ユーザに対して IA 活用の機会を拡大させる効果も期待できる。

4.2　アンケートによる主観評価

Java 言語の基本文法の学習経験がある社会科学系の学部生 7 人を被験者として，アンケートによる主観評価を行った。本実験では，まず被験者に対して筆者らが作成したマニュアルを用いて Immune 言語および対象問題である KP と TSP について説明し，その後，Immune 言語を用いたコーディングからプログラム実行までの作業を被験者にしてもらうものとした。アンケート調査では，Java 言語の理解度や IA に関する知識，Immune 言語の利便性について，5 段階の評価を行った。そのアンケート結果を図 6 に示す。図中のエラーバーは，95％の信頼区間を表す。図 6 より，Java 言語をある程度理解していれば免疫システムに関する知識がなくとも，Immune 言語を用いることで IA を取り扱えることが分かる。また，自由記述を集計した結果，「思いの外，プログラムがしやすかった。詳しい内容まではしっかり理解できなくとも簡単。短いコード記述で実行ができた。」，「難しいプログラムが簡単に作れた。使いやすかった。」，「Immune 言語でソースを書くと，Java 言語と比べてコードがとても短くすんだので驚いた。」，「慣れればすごく使いやすい言語だと思った。」などの回答が得られた。さらに，「エラーがある時，Immune コンパイラに表示されれば便利だと思った。」という回答も得られた。

以上の結果より，Immune 言語を用いることにより，免疫システムに関する知識がなくとも IA を KP や TSP に適用できると考えられる。しかしながら，現在の Immune 言語用コンパイラでは Immune 言語の文法的なミスを検出するものの，そのエラーメッセージは画面に表示されるのではなくファイルに出力される。そのため，エラーが発生した行番号やその内容などを，エラーメッセージとして画面に出力する機能を Immune 言語用コンパイラに追加し，ユーザの負担を軽減する必要があると考える。

5.　おわりに

本論文では，Java 言語に **immune** キーワードを追加定義した Immune 言語を提案した。Immune 言語を用いることでコーディングの煩雑さを軽減でき，また，Java 言語の知識さえあれば IA を容易に取り扱うことができる。IA の基本的な処理は GA と似ているため，Gene 言語同様 Espace 言語との融合により分散並列 IA を可能にする Immune 言語への展開も可能である。つまり，Immune 言語プログラムを Java 言語プログラムに変換する際，Immune 言語コンパイラが Espace 言語に関

するキーワードを自動補完することで，容易に分散並列 IA を実装できると期待できる。

今回開発した Immune 言語では，Gene 言語同様，KP と TSP を解くことに焦点を絞り IA の実装を行ったため，交叉や突然変異の処理がそれぞれの問題に対して固定的に定められている。しかしながら，Immune 言語の汎用性を向上させるために，親和度計算メソッドや類似度計算メソッド同様，交叉や突然変異の方法についてもユーザ自身が記述できるよう拡張を行う必要があると考えている。また，これらの対象とする組合せ最適化問題に依存する処理に対して，予め多種多様な問題を想定した処理を Immune 言語に組み込み，ユーザが単純に選択することによる自動補完を可能とすることで，さらなる開発効率の向上が見込まれる。さらに，本研究では，2.2 節で述べた IA 特有の多様性に関する閾値を固定値として扱ったが，その閾値を任意に決めるための機能を GUI として付加するなど，さらなる機能の充実が必要であると思われる。

注

(1) http://www.cobase.cs.ucla.edu/ を参照のこと。

参考文献

[1] 吉永孝二，飯村伊智郎，中山茂：遺伝的アルゴリズムのための Gene 言語の開発，情報文化学会論文誌，Vol.12, No.1, pp. 18−25, 2005.

[2] 飯村伊智郎，池端伸哉，中山茂：オブジェクト共有空間を用いた並列遺伝的アルゴリズムにおけるノアの箱舟戦略の検討，情報知識学会誌，Vol.13, No.2, pp. 1−17, 2003.

[3] 飯村伊智郎，松岡賢一郎，中山茂：1 次元トーラス網状離島モデルに基づく遺伝的局所探索における島間距離戦略の検討，電気学会論文誌 C, Vol.125, No.1, pp. 84−92, 2005.

[4] 森一之，築山誠，福田豊生：多様性をもつ免疫的アルゴリズムの提案と負荷割り当て問題への応用，電気学会論文誌 C, Vol.113, No.10, pp. 872−878, 1993.

[5] 森一之，築山誠，福田豊生：免疫アルゴリズムによる多峰性関数最適化，電気学会論文誌 C, Vol.117, No.5, pp. 593−598, 1997.

[6] 飯村伊智郎，杜暁冬，中山茂：免疫アルゴリズムによる複数画像領域探索の検討，情報処理学会論文誌，Vol.46, No.6, pp. 1512−1515, 2005.

[7] 中山茂，飯村伊智郎，伊藤登志也：免疫アルゴリズムにおける量子干渉交叉法の検討，電子情報通信学会論文誌 D-I, Vol.J88-D-I, No.12, pp. 1795−1799, 2005.

[8] 中山茂，伊藤登志也，飯村伊智郎，小野智司：免疫アルゴリズムにおける混合干渉交叉法の提案，電子情報通信学会論文誌 D, Vol. J89-D, No.6, pp. 1449−1456, 2006.

[9] 三井和男，大崎純，大森博司，田川浩，本間俊雄：発見的最適化手法による構造のフォルムとシステム，コロナ社，2004.

[10] 原崇，飯村伊智郎，武田和大，鶴沢偉伸，中山茂：インターネット・ユーザ参加型の分散並列処理のための Espace 言語の開発とその応用，情報文化学会論文誌，Vol.10, No.1, pp. 11−18, 2003.

[11] 岩川建彦，小野智司，中山茂：分散並列処理プログラミング言語 Espace の開発，システム制御情報学会論文誌，Vol.19, No.7, pp. 296−298, 2006.

付録

A.1 KP のための Immune 言語プログラム記述例

以下に，KP を解くプログラムを Immune 言語で記述した例を示す。

```
public class IA_bin {
  static int nGenes = 20;
  static int nAntibodies = 50;
  static public void main(String ar[])throws Exception{
    IA_bin ia_bin = new IA_bin();
    //IA 処理開始メソッドの呼び出し
    ia_bin.start(nGenes, nAntibodies);
  }
  // 前提条件変数の定義
  int luggage[][] = {{50, 10}, {30, 40},
                     ..... , {120, 100}, {100, 70}};
  // 親和度メソッドの定義
  public double setAffinity(boolean[][] cName,
                                          int nCell) {
    double affinity = 0.0d;
    int weight = 0;
    int value = 0;
    int limit = 550;
    for(int g = 0; g < nGenes; g++) {
      if(cName[nCell][g]) {
        weight += luggage[g][0];
        value += luggage[g][1];
      }
    }
    affinity = value;
    if(weight > limit) affinity = affinity / weight;
    return affinity;
  }
  // 類似度計算メソッドの定義
  public double setSimilarity(boolean[][] cName1,
          int nCell1, boolean[][] cName2, int nCell2) {
    double similarity = 0.0d;
    int commonLug = 0;
    for(int g = 0; g < nGenes; g++) {
      if(cName1[nCell1][g] == cName2[nCell2][g])
                                          commonLug++;
    }
    similarity = (double) commonLug / nGenes;
    return similarity;
  }
  //immune 構文による必要情報の指定
  immune start(boolean, setAffinity,
                              setSimilarity, luggage);
}
```

A.2 TSP のための Immune 言語プログラム記述例

以下に，TSP を解くプログラムを Immune 言語で記述した例を示す。

```
public class IA_int {
  static int nGenes = 51;
  static int nAntibodies = 100;
  static public void main(String ar[])throws Exception{
```

```
  IA_int ia_int = new IA_int();
  //IA処理開始メソッドの呼び出し
  ia_int.start(nGenes, nAntibodies);
}
//前提条件変数の定義
double city[][] = {{37, 52}, {49, 49},
                   ..... , {56, 37}, {30, 40}};
//親和度メソッドの定義
public double setAffinity(int[][] cName, int nCell) {
  double affinity = 0.0d;
  double len = 0;
  for(int g = 0; g < nGenes; g++) {
    double dx;
    double dy;
    int gNext = g + 1;
    if(gNext == nGenes) gNext = 0;
    dx = city[cName[nCell][g]][0]
                    - city[cName[nCell][gNext]][0];
    dy = city[cName[nCell][g]][1]
                    - city[cName[nCell][gNext]][1];
    len += Math.sqrt(Math.pow(dx, 2)
                                  + Math.pow(dy, 2));
  }
  affinity = 1.0d / len;
  return affinity;
}
//類似度計算メソッドの定義
public double setSimilarity(
                       int[][] cName1, int nCell1,
                       int[][] cName2, int nCell2) {
  double similarity = 0.0d;
  int commonPath = 0;
  int commonPath_Rev = 0;
  for(int g = 0; g < nGenes; g++) {
    int gNext1 = g + 1;
    if((g + 1) == nGenes) gNext1 = 0;
    int gSame = 0;
    while(cName1[nCell1][g] != cName2[nCell2][gSame])
                                          gSame++;
    int gNext2 = gSame + 1;
    if(gNext2 == nGenes) gNext2 = 0;
    int gNext2_Rev = gSame - 1;
    if(gNext2_Rev < 0) gNext2_Rev = nGenes - 1;
    if(cName1[nCell1][gNext1]
             == cName2[nCell2][gNext2]) commonPath++;
    if(cName1[nCell1][gNext1]
     == cName2[nCell2][gNext2_Rev]) commonPath_Rev++;
  }
  if(commonPath < commonPath_Rev)
                        commonPath = commonPath_Rev;
  similarity = (double) commonPath / nGenes;
  return similarity;
 }
 //immune構文による必要情報の指定
 immune start(int, setAffinity, setSimilarity, city);
}
```