TR-AC-0029 018

WCNetにおける動的セグメンテーションに関する研究報告

野崎　正典

1999. 3. 31

ATR環境適応通信研究所

目次

概要 ---------- 1

1. はじめに ---------- 1

2. 動的セグメンテーションの概要 ---------- 1

2.1 WACNetの特徴 ---------- 1

2.2 動的セグメンテーションスキームの必要性 ---------- 2

2.3 動的セグメンテーションスキームの動作 ---------- 2

2.4 動的セグメンテーションスキームの課題 ---------- 4

3. WACNetの分割手法 ---------- 4

3. 1 グラフ分割問題 ---------- 4

3. 2 ヒューリスティック手法 ---------- 5

3. 2.1 Min-Cut法 ---------- 5

3. 2.2 EPMC法 ---------- 5

3. 2.3 シミュレーション結果 ---------- 5

3. 3 遺伝的アルゴリズム ---------- 6

3. 3.1 遺伝的アルゴリズムの概要 ---------- 6

3. 3.2 シミュレーション結果 ---------- 7

4. マルチセグメント化への対応 ---------- 8

4.1 マルチセグメンテーションGA ---------- 8

4.1.1 マルチセグメント化における問題点 ---------- 8

4.1.2 適応度の定義 ---------- 8

4.1.3 シミュレーション結果 ---------- 9

4.2 ヒューリスティック手法 ---------- 10

4.2.1 EPMC法の改良 ---------- 10

4.2.2 シミュレーション結果 ---------- 10

4.3 並列分散化GA ---------- 12

4.3.1 並列分散化GAの概要 ---------- 12

4.3.2 シミュレーション結果 ---------- 12

5. まとめ ---------- 14

参考文献 ---------- 14

付録 A. WACNet評価システム ---------- 15

付録 B. シミュレーションプログラム ---------- 17

概要

我々は、一時的に集合する人々の間での、アドホックなワイヤレスネットワークを構築することを目的としている。このようなアドホックネットワークでは、ユーザの振る舞いは予め予測できないため、ネットワークシステムは適応的な機能を備えることにより、これらユーザの振る舞いに対処することが必要となる。

そこで本報告書ではこの適応的な機能として動的セグメンテーションスキームの検討を行い、そのセグメントへの分割アルゴリズムとしてヒューリスティックな手法、遺伝的アルゴリズムを用いた手法を提案する。計算機シミュレーションにより、これらの手法がネットワークリソースの利用効率に有効的であることを検証する。

1. はじめに

近年、ポータブルやラップトップコンピュータに代表されるモバイルホストの性能向上と、無線ネットワーキング技術の発展に伴い、それらを利用したモバイルコンピューティングの需要が高まってきている。モバイルホスト同士の通信を実現するためには、既存のセルラーフォンシステムの設備を利用したり、ISMバンドを用いた無線LAN等の予め設置されたインフラストラクチャが必要となる。しかしながら、必要なインフラストラクチャの導入が物理的、もしくは経済的に困難である場所においても、モバイルユーザ同士が通信を望むという要求がしばしば起こることがある。このような状況において、無線ネットワークインタフェースを有したモバイルホスト群が、確立したインフラストラクチャなしで一時的なネットワークを形成することがあり得る。このようなタイプの無線ネットワークはアドホックネットワークとして知られており、これについては多くの研究が行われている[1][2][3]。

そこで我々はこのアドホックネットワークの一形態として WACNet(Wireless Ad-Hoc Community Network)を提案している[4][5]。WACNetは国際会議や展示会など、何らかの共通の目的を持った人々が、臨時的、一時的に集合する場所での新たなコミュニケーションの形態として考えており、従来の音声、ビデオ等による一方的な情報伝達とは異なり、ワイヤレス技術を用いることによって参加者間のデータ通信ネットワークの構築をスムーズに行うことを目的としている。WACNetは特徴的な機能として、ネットワークの利用者数の増減に伴うモバイルホスト間の通信トラヒック環境の変化や、利用者の移動などによって生じるワイヤレスリンク環境の変化などに追随して、ネットワークトポロジーを再構成する動的セグメンテーションスキームを備えている。このような適応的な機能をネットワークに備えることにより、限られたネットワークリソースを有効的に利用することが可能となる。

我々は、このセグメントへの分割アルゴリズムとしてヒューリスティックな手法、遺伝的アルゴリズムを用いた手法を考案し、それらの有効性を計算機シュミレーションによって確かめたので以下に報告する。

2. 動的セグメンテーションの概要

2.1 WACNetの特徴

現在、我々が研究のターゲットとしている通信システムは、通信環境の変化に対して、適応的にその機能・構成などを変えることにより、弾力性、ロバスト性、自己組織性などの特徴を兼ね備えた環境適応通信システムである。ここでの通信環境の変化とは、ネットワークの負荷状況や、端末間の性能格差、端末の物理的な位置状態などが挙げられる。そこで我々はこの環境適応通信システムを実現するプラットホームとしてWACNetの研究を行っている。

WACNetの利用者はアドホックな集まりを対象としているため、ネットワークへの参加者数や利用頻度、またサービスの利用形態などが、その都度場所や時間によって異なることが予想され

る。それゆえ、これらのユーザの多種多様な振る舞いに対してネットワークが正常に動作するためには、ネットワークは柔軟なシステムであることが好ましい。そこでWACNetでは様々な適応性を持たすことにより、これらの問題に対処している。

以下にWACNetが持つ適応性を示す。

1) 利用者のサービスイン／アウトに対する適応性
2) 通信サービスへの適応性
3) 通信トラヒックへの適応性

1)の特徴は、新たなユーザがWACNetが供給するサービスエリアに入った際に、ネットワークサイドで即座にそれを検出し、その新たなユーザをメンバーとして登録し、その情報をすべてのユーザに告知することである。もちろん、サービスエリアを退出する時も同様の処理を行うことが必要となる。これによりユーザによる登録作業などの負荷が軽減でき、シームレスなネットワークが構築できる。

2)の通信サービスへの適応性の実施例は、例えば、ユーザがインターネットなどの外部ネットワークへ接続を要求する場合や、ユーザ間でのブロードキャスト、マルチキャストなどの通信サービスを要求する場合などが挙げられる。この場合、WACNetはネットワークトポロジーや通信プロトコルを変更することによって、それらの新しい通信サービスに適応することが可能となる。

最後の3)の特徴はユーザ数の増減に伴うネットワーク全体の通信トラヒックの増減や、ユーザの移動などによるネットワーク内での通信トラヒックの偏りに対する適応性である。これらの通信トラヒックの変動に適応してネットワークトポロジーを再構成することにより、ネットワークリソースの有効利用が実現可能となる。

そこで我々は今回、3)の特徴である通信トラヒックへの適応性を実現するための機能として、動的セグメンテーションスキームを考案し、その機能についての利点、課題、解決方法の検討を行った。

2.2 動的セグメンテーションスキームの必要性

WACNetでは一時的に形成するネットワークを想定しているため、モバイルホスト間のトラヒック分布や負荷、またそれらの相対的な接続関係を予め予測することは難しい。そのため、従来のようにネットワーク初期設計時にネットワークトポロジーや基地局の配置などを決定するのは困難である。そこでWACNetでは階層化やドメインのセグメント化などの処理を初期状態では行わず、ネットワークの使用状況に応じて網トポロジーを最構成する動的セグメンテーションスキームの概念を導入する。これによりネットワークはその都度最適なトポロジーを形成することが可能となり、余分な帯域を割り当てることがなくなるため、ネットワークリソースの有効利用が実現できる。また、初期設計時には人的なコストが削減できるため、ネットワーク設計の容易化も可能となる。

2.3 動的セグメンテーションスキームの動作

WACNetでのメディアアクセス制御は対等分散方式であり、CSMA/CAライクなキャリア検出機能を備えることにより、各モバイルホストは一つの無線チャネルを共有する。今ネットワークが初期状態からある程度の時間が経過して、モバイルホスト間のトラヒックの増加、偏りにより、一つの無線チャネル内でのスループットが著しく低下したとする。動的セグメンテーションスキームの目的は、この飽和した一つの無線チャネルを二つに分割することによって、チャネルあたりのモバイルホストの数を減少させ、他のモバイルホストからのトラヒックの影響を緩和することである。これにより、全体のスループットを向上させることが可能となる。この様子を図1に示す。

図1では8台のモバイルホストが共有していた1つの無線チャネルを、2つのチャネルに分割し、それぞれ分割されたチャネルを4台のモバイルホストが共有している。分割したチャネル同士は空間的に干渉が起こらないような多重化を行うため、分割後の全体のチャネル帯域は2倍となる。具体的には各チャネル毎に、CDMAにおける異なる拡散コード系列を与えることにより多重化を実現する。

しかしこの結果、異なるチャネル間では直接的な通信が出来なくなる。そのため異なるチャネル間にパケットを中継するためのブリッジ機能を備えたモバイルホストが各チャネルにそれぞれ一つずつ必要となる。この中継動作を図2を用いて説明する。図2ではモバイルホストCとFが各チャネル間の中継ホストとして選定されている。それぞれモバイルホストCがチャネル#0からチャネル#1への中継を、モバイルホストFがチャネル#1からチャネル#0への中継を担当する。ここでチャネル#1に属するモバイルホストEが、チャネル#0に属するモバイルホストDにパケットを転送する場合を例に挙げて説明する。

まずモバイルホストEは、ディスティネーションアドレスフィールドにモバイルホストDのアドレスをセットしたパケットを、チャネル#1で使用されている拡散コード#1で変調しチャネル#1内にブロードキャストする。 … (1)

次に、中継ホストとして選定されたモバイルホストFは、チャネル#1上でブロードキャストされたすべての拡散コード#1で変調されたパケットをいったん内部に取り込む。 … (2)

モバイルホストFは、取り込んだパケットのディスティネーションアドレスの解決を行い、もし、そのアドレスが示すモバイルホストが、他チャネル(このケースではチャネル#0)に属するのであれば、モバイルホストFはそのパケットを拡散コード#0で再変調し、チャネル#0にブロードキャストする。 … (3)

そうでない場合、すなわち取り込んだパケットのディスティネーションがチャネル#1内に属するのであれば、モバイルホストFはそのパケットを廃棄する。

この一連の動作により、モバイルホストEからモバイルホストDへのパケットは中継ホストFによって、2ホップでルーチングされる。

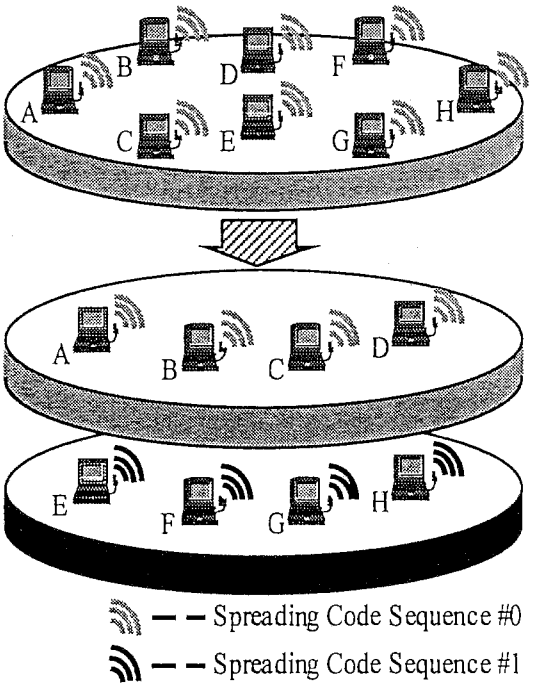


図1. チャネル分割の様子

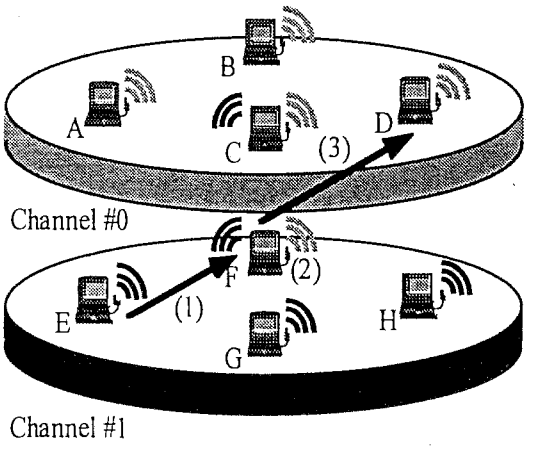


図2. 動的セグメンテーションの動作例

2.4 動的セグメンテーションスキームの課題

この動的セグメンテーションスキームで問題となるのはチャネルの分割方法である。すなわちどのような基準で各チャネルにモバイルホストを割り当て、どのモバイルホストを中継ホストとして選定するかということである。なぜならば、そのモバイルホストの割り当て方によっては、チャネル間を中継されるトラヒック量が異なるからである。この中継されるトラヒックはチャネル#0 とチャネル#1 の両方の帯域を使用するため、同じチャネル内のモバイルホスト同士の直接通信と比較して、2 倍のネットワークリソースを使用することになる。

このようにモバイルホストの割り当て方によって、ネットワークリソースの利用効率が変わってくるため、この利用効率を考慮した最適な分割方法を生成する必要がある。次章ではこの分割方法についての検討を行う。

3. WACNet の分割方法

3.1 グラフ分割問題

グラフ分割問題とは幾つかのノードとノード間を結ぶ何本かのリンクで構成されたグラフのノードを幾つかのグループに分割し(各グループに割り当てられるべきノード数は決められている)、グループ間に跨がるリンク数が最小となるような分割を求める問題であり、LSI 設計でのレイアウト等の分野では非常に重要な問題である。この様子を図 3 に示す。この図ではノード ABEF、ノード CDGH がそれぞれ別のグループに分割されることで、グループ間のリンク数が最小化されていることが分かる。

そこで、我々はこのグラフ分割問題にヒントを得て、WACNet での分割方法はトラヒックの交流度が強いモバイルホスト同士を同じチャネルに割り当て、異なるチャネル間を跨るトラヒック量が最小となるようなチャネル分割の方法を採用した。これはグラフ分割問題におけるノードとノード間を結ぶリンクの数をそれぞれ、WACNet でのモバイルホストとモバイルホスト間のトラヒック量に置き換えたものである。分割後のチャネル間を中継されるトラヒック量を最小とすることにより、動的セグメンテーション時の分割ロスを低く抑えることができ、ネットワークリソースの有効利用が実現可能となる。

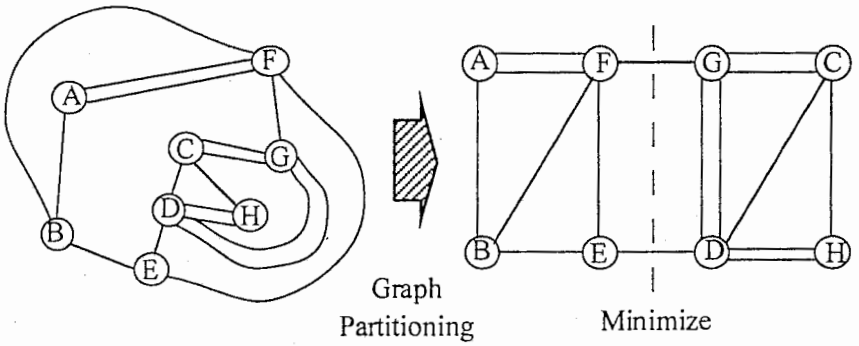


図 3. グラフ分割問題

3.2 ヒューリスティック手法

3.2.1 Min-Cut 法

グラフ分割問題の代表的なアルゴリズムとしてヒューリスティックな手法であるMin-Cut法が挙げられる[6]。Min-Cut 法には様々な改良版が提案されているが、おおよその動作は以下のようである。このアルゴリズムは基本的に二つのループからなり、内側のループでは、二つに分割したグラフにおいて、グラフ間に跨がるリンクの数(カット数)が一番少なくなるようなノードを選び他方のグラフに移動する。この操作をすべてのノードについて行う。すなわち、内側のループの最初の状態と最後の状態では丁度グラフが入れ替わった状態となる。一方、外側のループでは、前の内側のループの中で、最もカット数が少なくなった状態を選び、内側のループを再び適用する。この操作をカット数の減少が得られなくなるまで繰り返す。

Min-Cut 法は比較的良好な解を得ることができるが、初期分割(一般的に、乱数によって生成される)に対する依存性が高く、ローカルミニマムに陥りやすいという問題がある。また、Min-Cut 法をそのまま動的セグメンテーションのアルゴリズムに適用した場合に、分割後のチャネル間の負荷バランスが考慮されていないという問題もある。これはMin-Cut 法では、チャネル内のトラヒックの総和をチャネル同士で比較した時、その差が大きくなるようなケースが存在するためである。そのため負荷が均衡化されず、チャネル間の公正さが保てない恐れがある。

そこで我々は、これら Min-Cut 法の問題点を解決するために、Min-Cut 法に改良を加えたEPMC(Enhanced Pairing Min-Cut)法の検討を行った。

3.2.2 EPMC 法

EPMC 法のアルゴリズムは上述したMin-Cut 法をベースに、以下の項目についての性能向上を目的としている。

1)解空間探索能力の強化
2)分割後の負荷バランスの考慮

1)の解空間探索能力の強化に関しては、Min-Cut 法での内側のループのオペレーションに変更を加えることで対処している。従来のMin-Cut 法ではノードを他のグラフに移動した状態で、次のノードの移動を行っているため、ノードを移動させる順番の違いによっては、得られるカット数が異なってくる。なぜならば、探索する解空間はノードを移動させる順番に依存してそれぞれ異なっているからである。そこでEPMC 法ではノードを移動しカット数を求めた後、いったんそのノードをもとのグラフに戻し、その後に次のノードの移動を行う。このオペレーションは全ノードで行われる。これにより探索する解空間はノードの移動の順番に依存することなく、Min-Cut 法に比べて2乗倍の解空間を探索することができる。

2)の分割後の負荷バランスの考慮に関しては、ペアリングという前処理を加えることで対処している。この前処理は、初期分割を生成する際に、ノード間のリンク数が多いノード同士のペアを作成し、そのペアをリンク数が多いものから順番に2つのグラフに交互に割り当てていく処理である。この処理により、初期分割時のグラフ内のリンク数の総和が、グラフ間で均衡状態となる。それゆえ、分割後も乱数によって生成された初期分割に比べて、結びつきの強いノード同士が同じグラフに存在する確率が高くなり、その均衡が大きく壊されることがなくなる。

3.2.3 シミュレーション結果

EPMC 法の性能評価のために計算機によるシミュレーションを行った。シミュレーション条件として、モバイルホストの数を128、分割するチャネル数を2とし、モバイルホスト間のトラヒ

ックは、ショートパケットとロングパケットが混在する2種類のポアソン到着過程を使用した。この結果を図4に示す。

図4の横軸は探索の試行回数で、Min-Cut法での外側のループの実行回数を示している。縦軸は全体のトラヒックの総和に対する中継されるトラヒックの比率である。シミュレーションの結果、従来のMin-Cut法に比べ、EPMC法では解空間探索能力に関しては約3倍程度の改善が見られ、それに伴い中継されるトラヒックの量も減少していることが分かった。

しかしながら、EPMC法はMin-Cut法の解空間探索能力の性能向上は実現できたが、ロカールミニマムに陥る危険性への問題はまだ対処できていない。なぜならば根本的にこのヒューリスティックな手法は、今ある解の近辺を探索する局所的な探索しか行っていないからである。またEPMC法では探索する解空間が広がったために最適解を得るまでの処理時間が増大するという問題も発生した。この処理時間の増大は、即時性が要求される動的セグメンテーションスキームにとって大きなボトルネックとなる。

そこで我々は、動的セグメンテーションの新たなアルゴリズムとして、大域的探索に有利性を持つ遺伝的アルゴリズム(GA)を用いたアプローチを導入することとした。

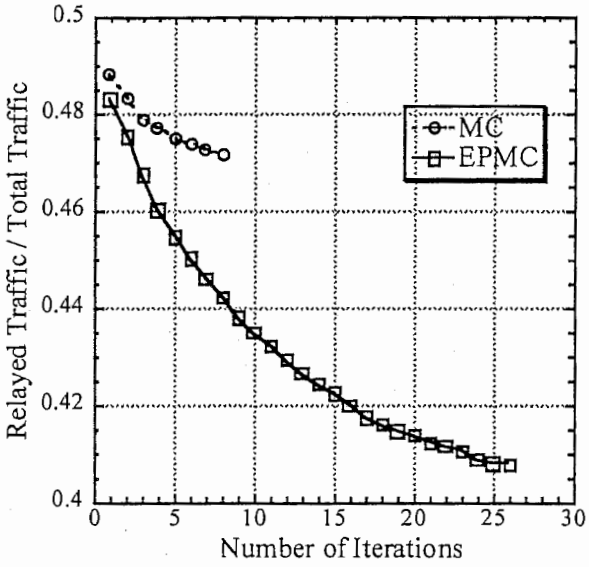


図4. Min-Cut法とEPMC法の探索能力の比較

3.3 遺伝的アルゴリズム

3.3.1 遺伝的アルゴリズムの概要

GAとは、生物進化(選択淘汰・突然変異)の原理に着想を得た、確率的検索・学習・最適化の一手法である。GAでは実際の生物での染色体や遺伝子座をそれぞれ、データ領域や配列、そのアドレスなどに対応して考えることができる。以下にGAに基本フローを示す。

a) 初期集団の生成 …… 決められた個体数の染色体をランダムに生成する。
b) 適応度の評価 …… 何らかの評価関数を用いるなりをして各々の個体の適応度を決定し評価する。
c) 選択 …… 優れた個体を形成する遺伝子を集団に広げるために交配させる個体を選択する。
d) 交叉 …… 選択された個体の染色体の組み替えを行う。

e) 突然変異 …… ある確率で染色体の一部の値を変更する。

b)から e)までの操作はある終了条件(例えば世代数、最適解の閾値など)が満たされるまでループする。このGAによる手法を動的セグメンテーションのアルゴリズムとして使用するためには、まず遺伝子情報を決定する必要がある。ここではチャネルの分割状態を"1001010..."のバイナリコードの配列で表現し、これを遺伝子情報とみなすこととした。このバイナリコードは、その配列の値(1もしくは0)がどちらのチャネルに属するのかを示し、その配列のアドレスがモバイルホストの番号を示している。このバイナリコードを用いて、GAの基本フローに基づいた計算機によるシミュレーションを行った。

3.3.2 シミュレーション結果

シミュレーション条件として、選択交配手法にはエリート保存戦略とルーレット戦略を統合したものを使用した。これはその集団中で最も適応度の高い個体はそのまま次世代に残し、残りの個体は適応度に比例した確率で選択される戦略である。交叉手法は、交叉する位置を一つ決めてその前と後ろで、どちらの親の遺伝子を受け継ぐかを変える一点交叉を使用した。突然変異は、適当な確率で遺伝子の0と1のビットを反転させることで実現した。これにより常に同じパターンの遺伝子が次世代に残ることを回避し、ローカルミニマムに陥る危険性を低減する。シミュレーションによる結果を図5に示す。

図5の縦軸は、図4と同様に全体のトラヒックの総和に対する中継されるトラヒックの比率であり、横軸は最適解を得るまでの計算機の処理時間である。尚、各種パラメータとして、個体数を20、交叉確率を0.3、突然変異確率を0.1とした。シミュレーションの結果、動的セグメンテーションのアルゴリズムにGAを用いることにより、EPMC法と比較して約4倍の速さの処理時間で最適解を得ることができることが分かった。

また各種パラメータを変更した場合のシミュレーションも行った。この結果を図6に示す。図6の横軸はGAオペレーションにおける世代数を示している。縦軸の0.47付近で留まっているデータは、交叉を行わず突然変異だけで探索したものである。これは、突然変異だけに依存しているとランダムサーチ化してしまい、最適解になかなか収束していないことが分かる。また突然変異なしで交叉だけで行った探索では、特定のパターンを持った遺伝子だけが残ることによりローカルミニマムに陥ることが分かった。これらの結果より、GAを用いたアプローチでは、適切な交叉確率と突然変異確率の選定が必要であるということが言える。

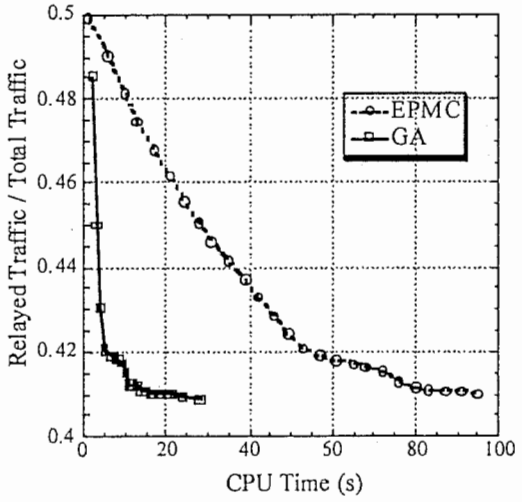


図5. EPMC法とGAとの処理時間の比較

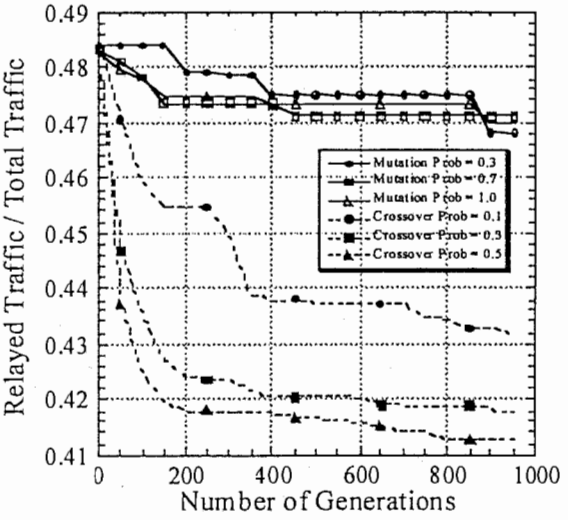


図6. 各パラメータ値における評価

4. マルチセグメント化への対応

4.1 マルチセグメンテーションGA

4.1.1 マルチセグメント化における問題点

次に、一度の動的セグメンテーションスキームの実行で、チャネルを複数のセグメントに分割することを目的とするマルチセグメンテーションについての検討を行った。任意の数でセグメント分割を行う場合、従来の2分割とは異なり、必ずしも分割後のセグメントあたりのモバイルホストの数が一定になるとは限らない。また、セグメントあたりのモバイルホストの数が不均一である状態で最適解があるようなトラヒックパターンの存在も考えられる。そこでマルチセグメンテーションスキームでは、2分割GAで用いていたセグメントあたりのモバイルホスト数が固定という制約をなくすこととした。これにより従来の2分割手法と比較してより広い解空間を探索することが可能となる。

しかし、この制約を解除したGAを実行した場合、従来の2分割GAで用いていたセグメント間を跨がるトラヒック量の最小化という適応度では、片方のセグメントだけにモバイルホストが集中するような解を探索することとなり、アルゴリズムが発散するという問題が発生した。

4.1.2 適応度の定義

上記した問題を解決するためにマルチセグメンテーションGAで用いる適応度を決定する必要がある。そこでまず、n個に分割されたセグメントのセグメントiでのトラヒック量S_iを以下の様に定義する。

$$S_i = \sum_{j=0}^{n} T_{ij} + \sum_{j=0}^{n} T_{ji} - T_{ii}$$

ここでT_{ii}はセグメントi内部における交流トラヒック量を、T_{ij}はセグメントiからjへの交流トラヒック量を、T_{ji}はセグメントjからiへの交流トラヒック量を示している。この様子を図7に示す。

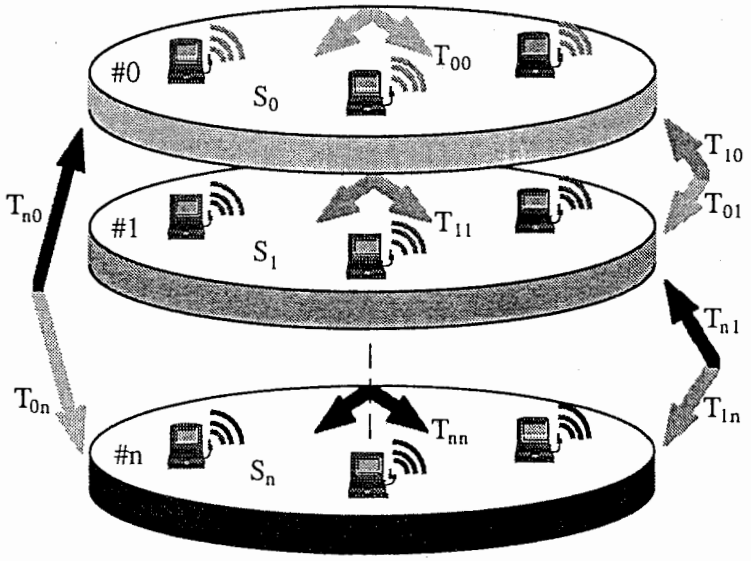


図7. トラヒック定義の様子

マルチセグメンテーションGAでの適応度は、このトラヒック量S_iを各セグメント毎に比較し、その最大値である$Max(S_i)$を最小化するものとした。この適応度を得るためには、S_iの定義式の前半の加算部の値を小さくする、もしくは後半の減算部の値を大きくする解を探索しなければならない。つまりこの適応度は、セグメントiにおける他セグメントとの交流トラヒック量を減少し、セグメント内部の交流トラヒック量を増加させることを意味している。この結果、セグメント間を中継するトラヒック量が減少され、2分割でのオペレーションと同様にネットワークリソースの有効利用が実現できる。この適応度では常に最大値を持つセグメントを最小化しようとするため、問題点としていた一つのセグメントにトラヒックが集中しアルゴリズムが発散してしまうことはなくなる。また他の利点として、セグメント毎のトラヒック量S_iがセグメント同士で平滑化されるため、各セグメントの負荷バランスが均等になり、新たな増加トラヒックに対する安全性が確保される。以上これらの特性を検証するために計算機によるシミュレーションを行った。

4.1.3 シミュレーション結果

シミュレーション条件は2分割GAの戦略と同様に、選択交配手法にはエリート保存戦略とルーレット戦略を統合したものを、交叉手法には一点交叉を使用した。各パラメータには、モバイルホスト数を64、個体数を30、交叉確率を0.3、突然変異確率を0.1とし、乱数のシードを変えたシミュレーションをそれぞれ50回試行した。またアルゴリズムの効果を検証するために最悪の分割ケースを探索するシミュレーションも同様に行った。この最悪ケースの場合、適応度を$\sum_{i=0}^{n}(S_i)$とすることで、交流トラヒック量が最大となる組合わせを探索している。

これらの結果を図8に示す。図では50回の試行のうちそれぞれ最良のものをプロットしている。

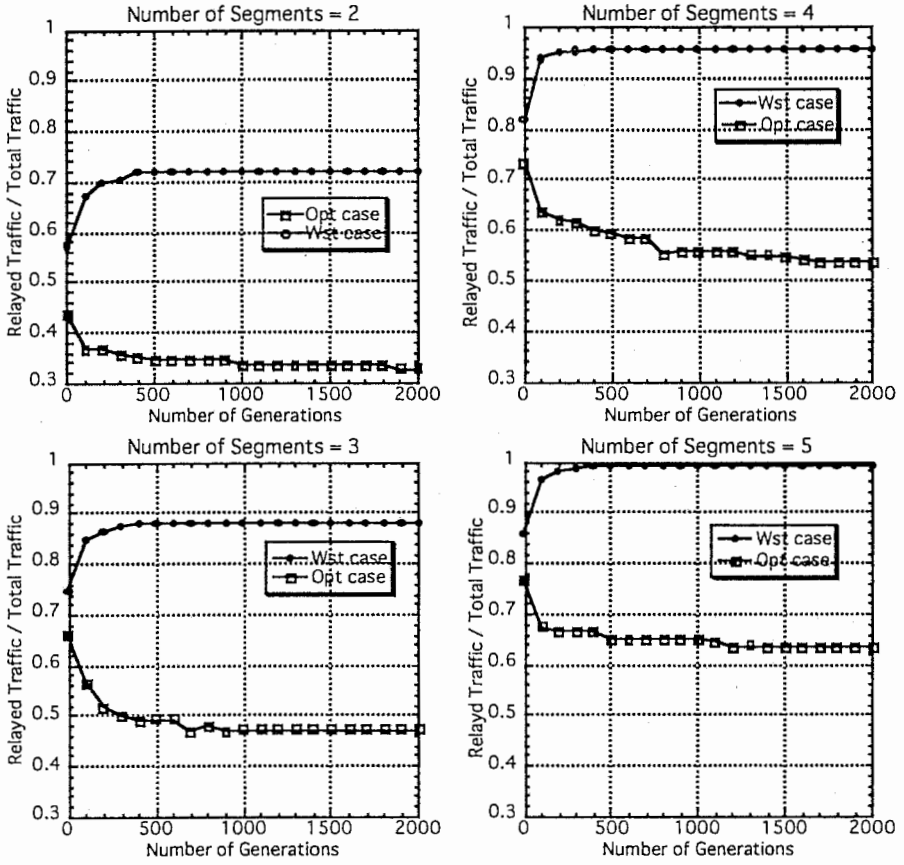


図8. マルチセグメンテーションGAによる評価

図の横軸は世代数を、縦軸は全体のトラヒックに対する中継トラヒックの比率を示している。図より各セグメント分割数における最適の分割状態は、最悪ケースの状態と比べて中継トラヒック量をおよそ1/2に抑えれることが分かる。しかし、分割数が増加するにつれてその効果は若干減少している。これは分割数が増えるにつれて組合わせのパターンが増加し、探索すべき空間が広がるため、同じ世代数では最適解に到達していないためだと考えられる。

次にマルチセグメンテーションGAとヒューリスティックな手法との比較を行うために、マルチセグメンテーションを考慮したEPMC法の検討を行った。

4.2 ヒューリスティック手法

4.2.1 EPMC法の改良

従来のEPMC法では、初期分割の段階でセグメントに属するモバイルホスト数を一定として解空間を探索するため、その数が不均一な状態で最適解が存在する場合には、最適解に辿り着く前に処理が終了してしまう。そこでマルチセグメンテーションで用いるEPMC法では、異なるモバイルホストの組み合わせパターンにおいても探索が可能となるような改良を加えた。これは均等分割で得られた最適解の近傍分割パターンを探索することで実現する。この様子を図9に示す。

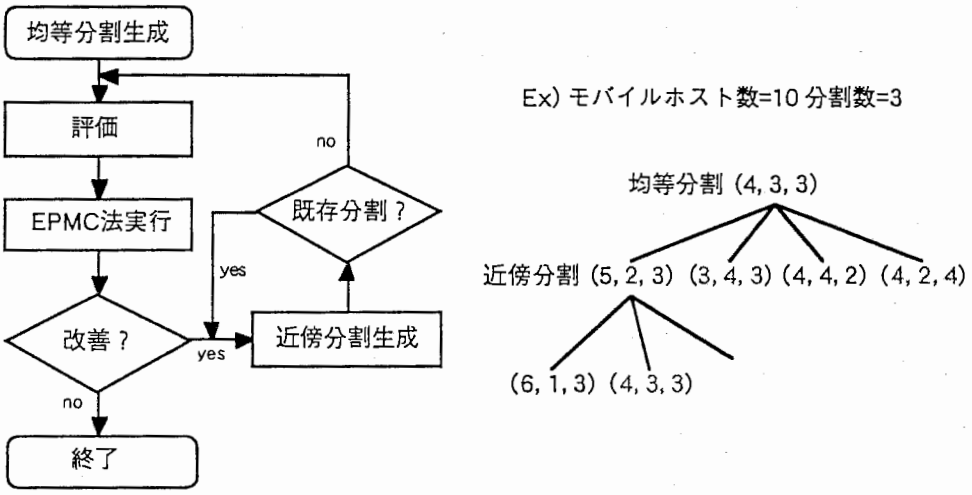


図9. ヒューリスティック手法のフロー

例えばモバイルホストの数が10、セグメントへの分割数が3の場合、均等分割におけるセグメントあたりのモバイルホストの数は(4,3,3)となる。まずはこの均等分割の状態でEPMC法を実行し、均等分割状態での最適解の探索を行う。均等分割状態での探索が終了すると、次にその近傍分割である(5,2,3)、(4,4,2)の組み合わせでの探索を行う。この時、既に探索を終えた分割パターンや、同じ分割パターンとなる組み合わせは除外する。この近傍分割の探索で得られた最適解と以前の探索で得られた最適解とを比較し、解が改善している場合は再びその近傍分割パターンの探索を行う。この操作を解の改善がなくなるまで行う。これによりセグメントあたりのモバイルホストの数が不均一な状態での探索を行なうことが可能となる。

4.2.2 シミュレーション結果

上記したヒューリスティック手法とマルチセグメンテーションGAとの性能評価を計算機シミュレーションにより行った。この様子を図10に示す。図の縦軸は全体のトラヒックに対する中継

トラヒックの比率を、横軸は解が得られるまでの計算機の処理時間を示している。GA のシミュレーション条件はは 4.1.3 節と同様のものを用い、50 回の試行のうち最良のデータをオプティマムケースに、最悪のデータをワーストケースとしてプロットしている。図より、分割数が少ない場合はヒュリスティック手法と GA を用いた手法との性能に格差はないが、分割数が増加するにつれてその差が顕著になっていることが分かる。これは上記したヒューリスティック手法では、分割数の増加に伴い探索する近傍分割パターンが多くなるため、一つの探索が終了するまでの処理時間が増大するためだと思われる。

マルチセグメンテーション GA でのオプティマムケースはおおむね良好な解を示しているが、ワーストケースでは局所解に陥りなかなか収束していないことが分かる。また実際の動的セグメンテーションスキームでアルゴリズムを実行する場合、一つのホストだけでこれらの計算を行うことは、負荷の集中という観点からはあまり望ましくない。そこで計算処理の負荷分散と、局所解に陥ることなく解の多様性を維持するために、GA の並列分散化の検討を行った。

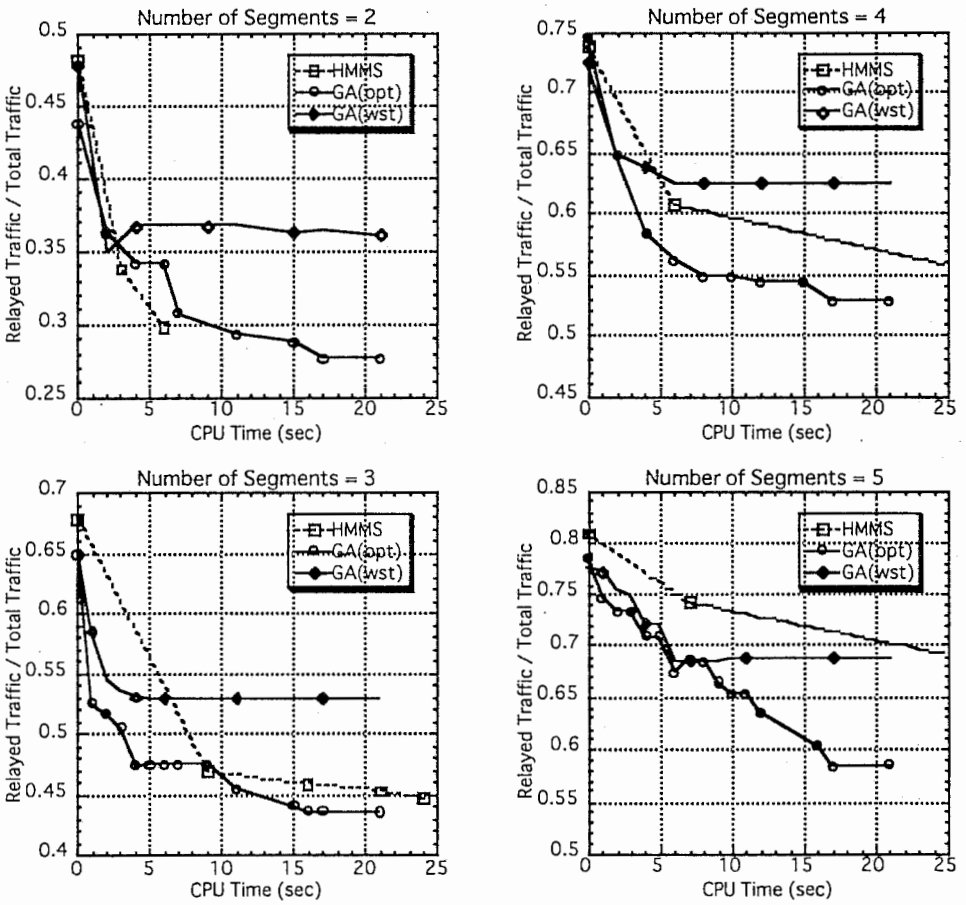


図 10. ヒューリスティック手法との比較

4.3 並列分散化GA

4.3.1 並列分散化GAの概要

生物学では、孤立した環境(例えば離島や山脈で区切られた平野)では、それぞれの環境の特性に適応して独自の進化を遂げることが知られている。並列分散GAでは、いくつかのサブ集団を競い合わせることによって、すべての個体が一つの集団を形成するのに比べて、より効率的な探索が可能となるという仮説の基に生まれている。これにより探索する解に対して多様性を維持することが可能となる。また並列分散化GAでは、サブ集団毎に複数の計算機上での探索が行なえるため、一つの計算機上でアルゴリズムを実行するのに比べて短時間で最適解を取得することが可能となる。

並列分散化GAの特徴として、サブ集団間で遺伝子を交換する移住操作が挙げられる。これは単一のサブ集団内で世代交代を繰り返していくと、次第に多様性が失われ探索が進まなくなる傾向があるため、ある世代毎に他のサブ集団から優秀な遺伝子の導入を行うものである。この様子を図11a)に示す。

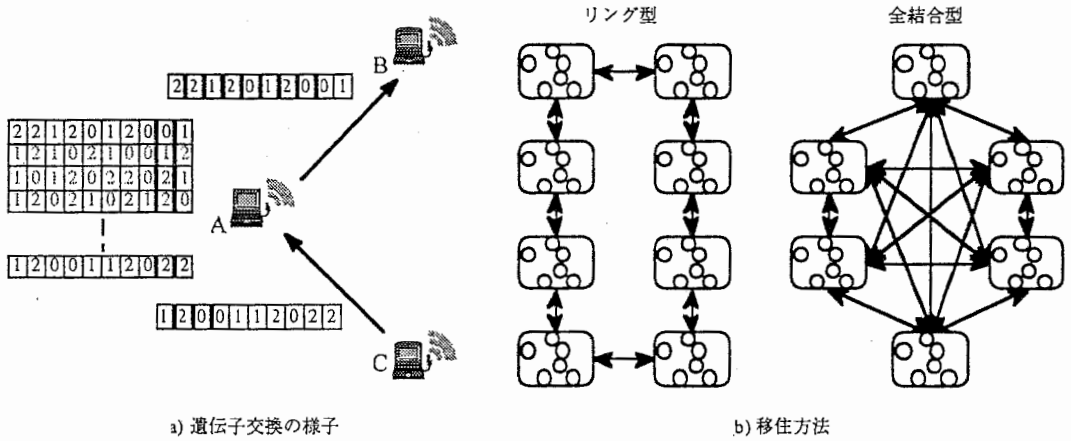


図11. 並列分散化GA

図ではモバイルホストCの優秀な遺伝子(1200112022)がモバイルホストAへ、モバイルホストAの優秀な遺伝子(2212012001)がモバイルホストBへ移住していることが分かる。この移住により、各々異なる環境下で世代交代を行った優秀な遺伝子同士が、交叉などでより優れた遺伝子を生み出すことが考えられる。

図11b)は移住の方法を示しており、隣接するサブ集団間で移住を行うリング型や、すべてのサブ集団同士で移住を行う全結合型、また図示していないがサブ集団が2次元、3次元の格子状に結合する格子型などがある。

4.3.2 シミュレーション結果

シミュレーションは、一つのサブ集団の個体数を10、世代数を1000、交叉確率を0.3、突然変異確率を0.1として、60台のモバイルホストを6つのセグメントに分割する条件で行い、それぞれサブ集団数を1,10,20,60と変化させ100回の試行を行った。サブ集団間の移住はリング型を実装し、移住の頻度は20世代毎とした。この結果を図12に示す。

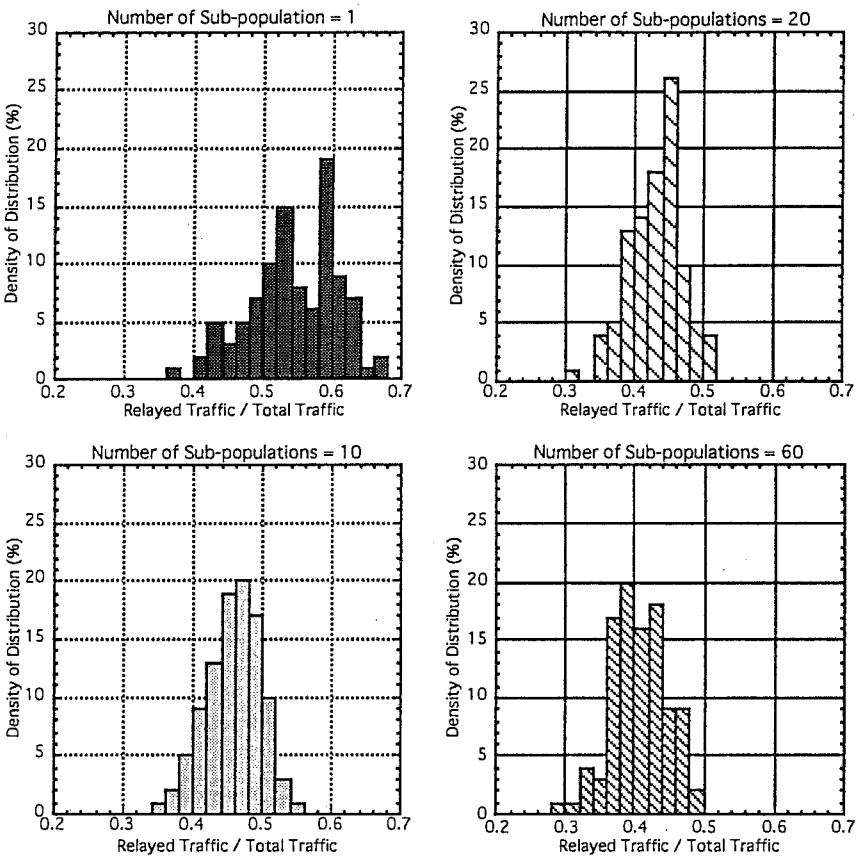


図 12. 並列分散化 GA による評価

図 12 は、横軸が全体のトラヒックに対する中継トラヒックの比率を示し、縦軸が 100 回の試行で得られた最適解の分布密度を示したヒストグラムである。図より、サブ集団数が 1 の場合では、0.4 以下の最適解に辿り着いたケースも見られるが、0.6 以上の局所解に留まっているケースが多く全体的にその分散も大きい。それに比べサブ集団数が増加すると、全体の分散も小さくなり、局所解に留まりにくくなっているのが分かる。これらの分布の平均値を縦軸に、サブ集団数を横軸にとったものを図 13 に示す。

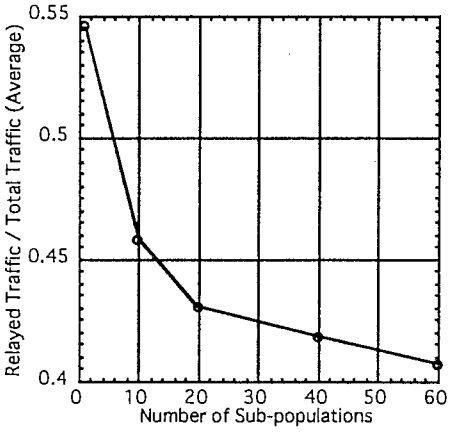


図 13. 平均値の向上

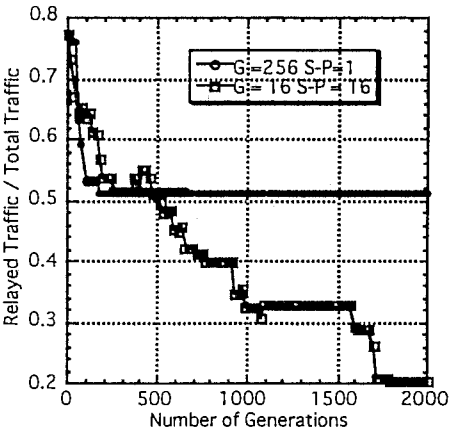


図 14. サブ集団化による効果

図13よりサブ集団数が増加するにつれて全体のトラヒックに対する中継トラヒックの比率の平均値が減少していることが分かる。これによりGAの並列分散化によって同じ世代数でもより良い最適解を得ることが可能であるということが言える。

また、全体の個体数が同じになる条件で、個体数が256でサブ集団数が1、個体数が16でサブ集団数が16のパラメータでシミュレーションを行った結果を図14に示す。図より、サブ集団数が1のデータでは0.5付近で局所解に留まり、最適解を得ることが出来ていないが、サブ集団数が16のデータでは局所解に留まることなく探索が進んでいることが分かる。これはサブ集団化することにより、解の探索能力の多様性が高まり、局所解に陥る確率が減少するためだと思われる。

以上、GAの並列分散化を行うことにより、最適解の平均値の向上と、局所解に留まることない探索の多様性の維持が可能となり、マルチセグメンテーションにおけるネットワークリソースの有効利用に対して非常に効果があるといえる。

5. まとめ

WACNetにおける動的セグメンテーションスキームにおいて、セグメントへの分割ルールとしてグラフ分割問題が適用可能であることを示し、ヒューリスティックな手法であるMin-Cut法を改良したEPMC法の検討を行った。また、EPMC法の処理速度の向上とローカルミニマムへ陥る危険性を回避するために、GAによるセグメントへの分割ルールを生成し、これらの問題に対して解決可能であることを示した。

また、一度の動的セグメンテーションスキームの実行で、チャネルを複数のセグメントに分割するマルチセグメンテーションスキームでは、分割アルゴリズムとして並列分散化GAの検討を行い、計算機シミュレーションによるアルゴリズムの有効性を検証した。

参考文献

[1] C-K Toh, "WIRELESS ATM and AD-HOC NETWORKS," Kluwer Academic Publishers, 1997.

[2] C Perkins and P.Bhagwat, "Highly Dynamic Destination Se-quenced Distance Vector Routing (DSDV) for Mobile Comput-ers," Proc. of ACM SIGCOMM'94, pp. 234-244, Sep 1994.

[3] D.Johnson, "Routing in Ad-Hoc Networks of Mobile Hosts," Proc. of IEEE Workshop in Mobile Computing & Applications, pp.158-163, Dec 1994.

[4] M.Kawai and M.Nozaki, "Wireless Ad-Hoc Community Net-work," Proc. of the 1997 IEICE General Conference, B-7-234, Mar 1997.

[5] M.Nozaki and M.Kawai, "A Study of Dynamic Segmentation in a Wireless Ad-Hoc Community Network," Proc. of the 1997 Communication Society Conference of IEICE, B-7-138, Sep 1997.

[6] Kernighan.B.W and Lin.S, "An Efficient Heuristic Procedure for Partitioning Graphs," Bell Systems Technical Journal, vol.49, pp.291-307, 1970.

付録A. WACNet評価システム

上記したWACNetの特徴を、実際のネットワークシステムとして反映させるために、小規模なWACNetの評価システムの試作を行った。ただし、97年度の試作においては既存のハードウェア／プロトコルに基づいて実装したため、すべての特徴を完全には実装しきれていない。

A.1 評価システムの構成

本試作におけるハードウェア構成を図A-1に示す。本来のWACNetでの仕様では、基地局はフリーで、各モバイルホストがブリッジ機能を備える自律分散型のネットワークシステムを想定しているが、既存のハードウェアを用いて実装した関係上、本試作におけるシステムは、4台のモバイルホストと2つの無線チャネルを供給するアクセスポイント2台、アクセスポイントの状態を監視するアクセスポイントモニタ1台、モバイルホスト間のトラヒックを収集するLANモニタ1台、及び動的セグメンテーションのアルゴリズムを実行する固定ホスト1台によって構成されている。

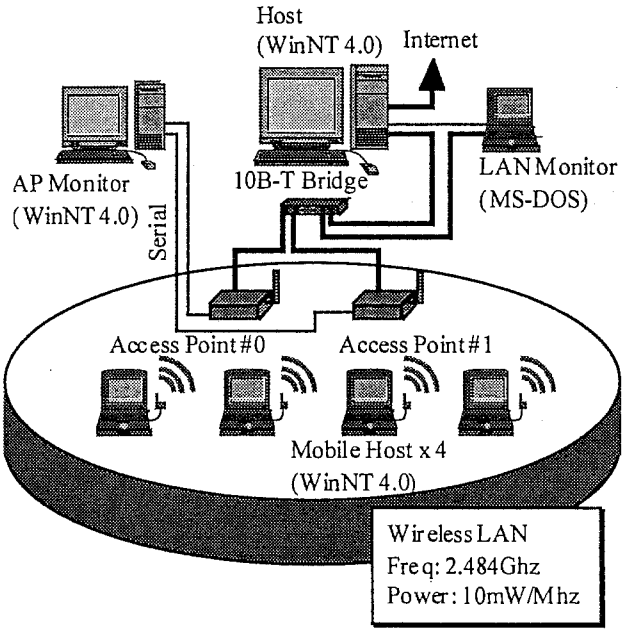


図 A-1 ハードウェア構成図

A.2 評価システムの特徴

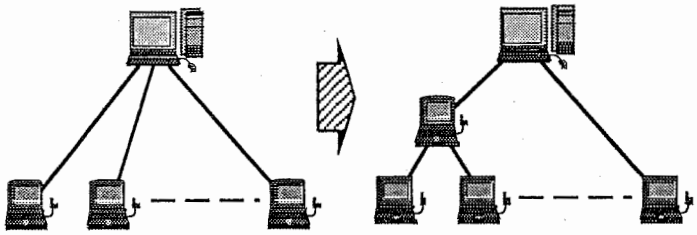
本評価システムで実現された機能及び特徴を以下に示す。

- 各モバイルホストのサービスエリアの入出管理及び位置表示
- 仮想モバイルホストの生成及びシミュレーション
- ネットワークセグメント生成／解除
- ネットワーク構成及びトラヒックフローのリアルタイム表示

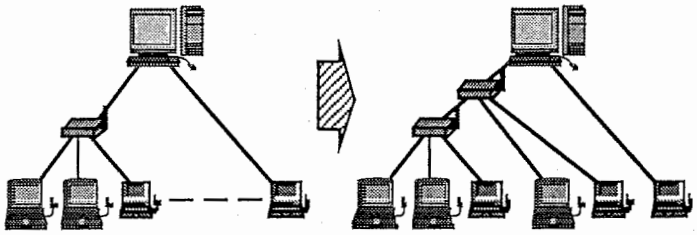
固定ホストでは、仮想モバイルホストが16台生成され、その仮想モバイルホストと実際の4台のモバイルホストとの間に仮想的なトラヒックを生成する。固定ホストでは、この仮想的なトラヒックデータとLANモニタで収集された実際のトラヒックデータを用いて、チャネル分割パターンを生成する計算が行われる。この生成されたチャネル分割パターンは各モバイルホストに通知

され、各モバイルホストは、このパターンに従い共有する無線チャネルを変更する。これによりネットワークの再構成が行われる。

図A-2 a)に本評価システムにおける動的セグメンテーション実行時の論理的なネットワーク構成図を示す。図では、ブリッジ機能を備えたモバイルホストがチャネル間のトラヒックを中継することで、動的セグメンテーションスキームを実行している。しかしながら、現状のモバイルホストにはこのブリッジ機能が備わっていないため、その代わりにチャネル間に2つの無線チャネルを供給することができるアクセスポイントを2台置くことによりチャネル間のトラヒックの中継を実現している。この様子を図A-2 b)に示す。固定ホストによって生成されたチャネル分割パターンに従って、モバイルホスト及び仮想モバイルホストはどちらのアクセスポイントに属するかを決定する。これによって動的セグメンテーションスキームにおける2つのチャネルへの分割を実現している。



a) Logical Network Structure



b) Network Structure using Access Points

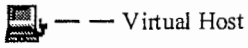


図 A-2 評価システムにおける動的セグメンテーションスキーム

付録B. シミュレーションプログラム

並列分散化GAのシミュレーションで用いたプログラムを添付する。プログラムは3つのファイルから構成され、makeコマンドにより実行ファイルgawacが作成される。
gawacにはオプションとして、 -w ワーストケース、-s モバイルホスト数、-d セグメント数、-p 個体数、-g 世代数、-c サブ集団数、-f トラヒックファイルをわたすことができる。その他のパラメータはinclude.hファイルで定義されている。以下に実行例を示す。

```
#gawac -s 60 -d 6 -p 10 -g 1000 -c 6 -f data.txt

   Genetic Alogrithm start.
   P_CROSS = 0.300000 P_TURNOVER = 0.100000 P_MUTATION = 0.100000
   N_SELECT = 7 PLOT_INTERVAL = 100 MIG = 1 MIG_INTERVAL = 20
   total_sum = 1091
   0th simulation start.
   generation,    max traffic,  cut ratio,        average,     time(s)
   0,             345,          0.8313,           0.3052       0,
   100,           321,          0.7342,           0.2890       2,
   200,           319,          0.6975,           0.2829       4,
   300,           308,          0.6544,           0.2757       5,
   400,           304,          0.6572,           0.2762       7,
   500,           301,          0.6077,           0.2679       9,
   600,           291,          0.5280,           0.2547       11,
   700,           291,          0.5280,           0.2547       13,
   800,           286,          0.5179,           0.2530       15,
   900,           285,          0.5005,           0.2501       16,
   1000,          277,          0.4895,           0.2482       18,

   ================   optimize case result.  ================

   genetic ->
   2222222200000002220511143155514344033213513145333351155404 44

   segment#0 traffic ratio-> 0.2447    station -> 9
   segment#1 traffic ratio-> 0.2521    station -> 10
   segment#2 traffic ratio-> 0.2521    station -> 10
   segment#3 traffic ratio-> 0.2466    station -> 11
   segment#4 traffic ratio-> 0.2401    station -> 10
   segment#5 traffic ratio-> 0.2539    station -> 10
   total cut ratio -> 0.4895
   average segment traffic -> 0.2482
   total time = 18.31(s)

   ==============================================================
```

実行結果は、60のモバイルホストを6分割、個体数10でサブ集団数を6、移住頻度を20世代毎とし、1000世代実行させたものである。結果、総トラヒックに対する中継トラヒック比率が0.8313から0.4895に減少していることが分かる。
なお、トラヒックデータであるdata.txtは各モバイルホストの送受信トラヒック量の行列ファイルであり、makematrixコマンドで作成可能である。

```
--- Makefile file ---
# Makefile for gawac

CC = gcc

CFLAGS = -O2

OFILES = main.o

all: gawac

gawac: $(OFILES)
	$(CC) $(CFLAGS) $(OFILES) -o gawac -lm

clean:
	rm -f *.o core *~ gawac


--- include.h file ---
/*****************************************************

          Genetic Algorithm for WACNet

        include.h    ver1.2.1   Feb.12.1999

                M.nozaki(nozaki@acr.atr.co.jp)

*****************************************************/

/* トラヒックデータを配列として使用 */
#define ARY_POS(row,col) (traffic + (station * row + col))
#define SWAP(a,b) {int tmp; tmp=a; a=b; b=tmp;} /* スワップ */
#define CUT_INDEX 1 /* 適応度計算時の乗数 */
#define P_CROSS 0.3 /* 交叉確率 */
#define P_TURNOVER 0.1 /* ターンオーバー確率 */
#define P_MUTATION 0.1 /* 突然変異確率 */
#define N_SELECT (int)(population - P_CROSS*population)/* 選択される個体数 */
#define PLOT_INTERVAL 100 /* ログ書き出しの期間 */
#define AVERAGE 1 /* シミュレーションの実行回数 */
#define MIG 1 /* 移住の判定 */
#define MIG_INTERVAL 20 /* 移住の頻度 */

#define GENE 0
#define S_GENE 1
#define CUT 2
#define FIT 3

#define TIMEUSE /* システム時間による乱数の発生 */

int station; /* ステーション数 */
int population; /* 個体数 */
```

```
int generation; /* 世代数 */
int segment; /* セグメント数 */
int group; /* サブ集団数 */

int worst; /* ワーストケースの判定 */
int total_sum; /* 全トラヒックの総和 */

int *traffic; /* トラヒックデータ */
int *genetic; /* 各個体毎の遺伝子の配列 */
int *s_genetic; /* *genetic の保持 */
int *segtra; /* トラヒックテーブル */
int *s_segtra; /* *segtra の保持 */
int *sort_ary; /* ソート配列 */
int *segtra_fit; /* 適応度 */
int *segtra_fit_tbl; /* 適応度のテーブル */
int *segtra_fit_sum; /* 適応度の合計 */

/* 関数のプロトタイプ宣言 */
void init_rand();
unsigned int get_rand(int);
void init_genetic();
int calc_segtra(int);
void calc_fitness();
void sort_s_segtra(int);
void selection();
void crossover();
void chk_gene();
void change_gene(int,int,int);
void mutation();
void migration();

void display(int);
```

--- main.c ---

```
/*****************************************************

          Genetic Algorithm for WACNet

        main.c    ver1.2.1   Feb.12.1999
                (ring type migration)

                M.nozaki(nozaki@acr.atr.co.jp)

*****************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <unistd.h>
#include <sys/times.h>
```

```
#include "include.h"

main(argc, argv)
int argc;
char *argv[];
{

  int i,j,k;
  double seg_ratio; /* クロストラヒックの比率 */
  int seg_sum; /* 分割後の総トラヒック */
  int chk_seg_sum,ofs; /* seg_sumのチェック用、オフセット */
  int *seg_dis; /* 各セグメントに属するステーション数 */

  /* 計算時間計測 */
  struct tms at_start,at_end;
  long clock_tick=sysconf(_SC_CLK_TCK);

  /* トラヒックデータ読込用ファイルポインタ */
  FILE *fp;

  /* オプション変数 */
  int c;
  int errflg = 0;
  extern char *optarg;
  extern int optopt;
  worst = 0;

  while((c = getopt(argc, argv, ":ws:d:p:g:c:f::")) != -1)
    switch(c){
    case 'w': /* ワーストケース */
      worst = 1;
      break;
    case 's': /* ステーション数 */
      station = atoi(optarg);
      break;
    case 'd': /* セグメント数 */
      segment = atoi(optarg);
      break;
    case 'p': /* 個体数 */
      population = atoi(optarg);
      break;
    case 'g': /* 世代数 */
      generation = atoi(optarg);
      break;
    case 'c': /* サブ集団数 */
      group = atoi(optarg);
      break;
    case 'f':
      fp = fopen(optarg,"r");
      break;
```

```
    case ':':
      fprintf(stderr, "option -%c requires an argument¥n",optopt);
      errflg++;
      break;
    case '?':
      fprintf(stderr, "Unrecognized option: - %c¥n",optopt);
      errflg++;
    }

  /* オプションが不適切時の処理 */
  if(errflg > 0 || segment >=  station || fp == NULL){
    fprintf(stderr,"¥nUsage: gawac ¥t-w worst case¥n¥t¥t-s [station
number]¥n¥t¥t-d [segment number]¥n¥t¥t-p [population number]¥n¥t¥t-g
[generation number]¥n¥t¥t-c [group number]¥n¥t¥t-f matrix file¥n");
    return(1);
  }

  traffic = (int *)malloc(sizeof(int)*station*station);
  genetic = (int *)malloc(sizeof(int)*station*population*group);
  s_genetic = (int *)malloc(sizeof(int)*station*population*group);
  segtra = (int *)malloc(sizeof(int)*segment*population*group);
  s_segtra = (int *)malloc(sizeof(int)*population*group);
  seg_dis = (int *)malloc(sizeof(int)*segment);
  sort_ary = (int *)malloc(sizeof(int)*population*group);
  segtra_fit = (int *)malloc(sizeof(int)*N_SELECT*group);
  segtra_fit_tbl = (int *)malloc(sizeof(int)*N_SELECT*group);
  segtra_fit_sum = (int *)malloc(sizeof(int)*group);

  /* データファイルからの読みだし */
  for(i=0;i<station;i++)
    for(j=0;j<station;j++)
      fscanf(fp,"%d",ARY_POS(i, j));

  fclose(fp);

  total_sum = 0;

  /* 総トラヒック量の計算 */
  for(i=0;i<station;i++)
    for(j=0;j<station;j++)
      total_sum += *ARY_POS(i,j);

  /* ログのラベル表示 */
  printf("¥t¥nGenetic Alogrithm start.¥n");
  printf("P_CROSS = %f P_TURNOVER = %f P_MUTATION = %f¥n",
      P_CROSS,P_TURNOVER,P_MUTATION);
  printf("N_SELECT = %d PLOT_INTERVAL = %d MIG = %d MIG_INTERVAL = %d¥n",
      N_SELECT,PLOT_INTERVAL,MIG,MIG_INTERVAL);
  printf("total_sum = %d¥n",total_sum);

  for(i=0;i<AVERAGE;i++){
```

```
    printf("%dth simulation start.¥n",i);
    printf("generation,¥tmax traffic,¥tcut ratio,¥taverage,¥ttime(s)¥n");

    init_rand(); /* 乱数の初期化 */
    init_genetic(); /* 遺伝子の初期化 */
    calc_fitness(); /* 適応度の計算 */

    times(&at_start); /* 処理時間の計測開始 */

    for(j=0;j<=generation;j++){

      crossover(); /* 交叉 */
      mutation(); /* 突然変異 */
      calc_fitness();

      if(MIG && j%MIG_INTERVAL == 0 && j != 0 && group > 1)
      migration(); /* サブ集団間の移住 */

      if(j%PLOT_INTERVAL == 0){ /* ログの表示タイミング */

      if(worst){ /* 各サブ集団の最大トラヒックをチェック */
        chk_seg_sum = 0;
        for(k=0;k<group;k++)
          if(*(s_segtra + population*k) > chk_seg_sum ){
            chk_seg_sum = *(s_segtra + population*k);
            ofs = k;
          }
      }
      else{ /* 各サブ集団の最小トラヒックをチェック */
        chk_seg_sum = total_sum;
        for(k=0;k<group;k++)
          if(*(s_segtra + population*k) < chk_seg_sum ){
            chk_seg_sum = *(s_segtra + population*k);
            ofs = k;
          }
      }

      seg_sum = 0;

      /* トラヒックの総和の計算 */
      for(k=0;k<segment;k++)
        seg_sum += *(segtra + segment*(population*ofs + *(sort_ary +
population*ofs)) + k);

      /* クロストラヒックの比率の計算 */
      seg_ratio = (double)(seg_sum - total_sum)/total_sum;

      times(&at_end); /* 現在までの処理時間 */

      /* ログデータの表示 */
      printf("%d,¥t¥t%d,¥t¥t%.4f,¥t¥t%.4f¥t¥t%.f,¥n",
```

```
            j, /* 世代 */
            *(s_segtra + population*ofs), /* 最大トラヒック */
            seg_ratio, /* クロストラヒックの比率 */
            (double)seg_sum/(total_sum*segment), /* セグメントの平均負荷 */
            (at_end.tms_utime - at_start.tms_utime)/(double)clock_tick); /*
処理時間 */

      }
    }
  }

  for(i=0;i<segment;i++)
    *(seg_dis + i) = 0;

  if(worst)
    printf("¥n================   worst case
result.================¥n¥n");
  else
    printf("¥n================  optimize case result.
================¥n¥n");

  printf("genetic -> ");
  for(i=0;i<station;i++){
    /* 各セグメントに属するステーション数の計算 */
    *(seg_dis + *(s_genetic + i)) = *(seg_dis + *(s_genetic + i)) + 1;
    /* 遺伝子パターンの表示 */
    printf("%d",*(s_genetic + station*population*ofs + i));
  }
  printf("¥n¥n");

  for(i=0;i<segment;i++)
    printf("segment#%d traffic ratio-> %.4f¥t station -> %d¥n",
        i, /* セグメント番号 */
        /* 各セグメントのトラヒック負荷 */
        (double)*(segtra + segment*(population*ofs + *(sort_ary +
population*ofs)) + i)/total_sum,
        *(seg_dis + i)); /* 各セグメントのステーション数 */

  /* クロストラヒックの比率 */
  printf("total cut ratio -> %.4f¥n",(double)(seg_sum -
total_sum)/total_sum);
  /* セグメントの平均負荷 */
  printf("average segment traffic -
> %.4f¥n",(double)seg_sum/(total_sum*segment));
  /* トータル処理時間 */
  printf("total time = %.2f(s)¥n¥n",
      (at_end.tms_utime - at_start.tms_utime)/(double)clock_tick);

  printf("============================================================¥n")
;
```

```
}

/******************/
/** 乱数の初期化 **/
/******************/

void init_rand()
{

#ifdef TIMEUSE
  int seed = (int)time(NULL);
#else
  int seed = 1;
#endif

  srand(seed);

  return;
}

/****************************/
/** width の範囲で乱数発生 **/
/****************************/

unsigned int get_rand(width)
int width;
{
  return( (unsigned int)((double)rand() / 32768.0 * width) );
}

/********************/
/** 遺伝子の初期化 **/
/********************/

void init_genetic()
{
  int quot,rest,offset;
  int cnt = 0;
  int set = 0;
  int i,j,k;
  int r1,r2;

  quot = station/segment; /* 商 */
  rest = station%segment; /* 余 */

  /* 均等分割で遺伝子を作成 */
  while(cnt < station){
    offset = cnt;
    if(rest > 0)
      for(i=0;i<=quot;i++){
      *(genetic + offset + i) = set;
      cnt++;
```

```
      }
    else
      for(i=0;i<quot;i++){
      *(genetic + offset + i) = set;
      cnt++;
      }
    rest--;
    set++;
  }

  /* 個体数 x サブ集団分コピー */
  for(i=1;i<population*group;i++)
    for(j=0;j<station;j++)
      *(genetic + (station*i + j)) = *(genetic + j);

  /* 一個体におけるの遺伝子パターンの変更 */
  for(i=0;i<population*group;i++)
    for(j=0;j<station*station;j++){
      r1 = get_rand(station);
      do{
      r2 = get_rand(station);
      }while(*(genetic + station*i + r1) == *(genetic + station*i + r2));
      SWAP(*(genetic + (station*i +r1)),*(genetic + (station*i + r2)));
    }

}

/******************/
/** 適応度の計算 **/
/******************/

void calc_fitness()
{
  int i,j,k;
  int max;

  /* calc_segtra(int)で得られたセグメント毎の最大トラヒック量(最適ケース)
     もしくは総トラヒック量(最悪ケース)を保持 */
  for(i=0;i<group;i++)
    for(j=0;j<population;j++){
    *(s_segtra + population*i + j) = calc_segtra(population*i + j);
    *(sort_ary + population*i + j) = j;
  }

  /* s_segtra を小さい順(最適ケース)もしくは大きい順(最悪ケース)にソート */
  for(i=0;i<group;i++)
    sort_s_segtra(i);

  /* ソートされた順に遺伝子を s_genetic にセーブ */
  for(k=0;k<group;k++)
    for(i=0;i<population;i++)
      for(j=0;j<station;j++)
```

```
      *(s_genetic + station*(population*k + i) + j) =
        *(genetic + station*(population*k + *(sort_ary + population*k + i))
+ j);

  /* セグメント毎のトラヒック量を適応度に変換 */
  for(k=0;k<group;k++){
    max = 0;
    for(i=0;i<N_SELECT;i++){
      *(segtra_fit + N_SELECT*k + i) =
      (int)abs(pow(*(s_segtra + population*k + N_SELECT)
                - *(s_segtra + population*k + i), CUT_INDEX));
      max += *(segtra_fit + N_SELECT*k + i);
    *(segtra_fit_tbl + N_SELECT*k + i) = max;
    }
    *(segtra_fit_sum + k) = max;
  }

}

/*********************************/
/** トラヒックテーブルのソート **/
/*********************************/

void sort_s_segtra(ofs)
int ofs;
{
  int i,j;
  int tmp;

  if(worst) /* 最悪ケースでは総トラヒック量が大きい順にソート */
    for(i=1;i<population;i++)
      for(j=i;j>=1;j--)
      if(*(s_segtra + population*ofs + j - 1) < *(s_segtra + population*ofs
+ j))
        {
          SWAP(*(s_segtra + population*ofs + j - 1), *(s_segtra +
population*ofs + j));
          SWAP(*(sort_ary + population*ofs + j - 1), *(sort_ary +
population*ofs + j));
        }
      else
        break;
  else /* 最適ケースでは最大トラヒック量が小さい順にソート */
    for(i=1;i<population;i++)
      for(j=i;j>=1;j--)
      if(*(s_segtra + population*ofs + j - 1) > *(s_segtra + population*ofs
+ j))
        {
          SWAP(*(s_segtra + population*ofs + j - 1), *(s_segtra +
population*ofs + j));
          SWAP(*(sort_ary + population*ofs + j - 1), *(sort_ary +
population*ofs + j));
```

```
      }
    else
      break;

}

/************************/
/** トラヒック量の計算 **/
/************************/

int calc_segtra(popu)
int popu;
{
  int tra =0;
  int segofs,staofs;
  int i,j;
  segofs = segment*popu;
  staofs = station*popu;

  /* トラヒックテーブルの初期化 */
  for(i=0;i<segment;i++)
    *(segtra + segofs + i ) = 0;

  /* トラヒック量の計算 */
  for(i=0;i<station;i++){
    for(j=0;j<station;j++){
      *(segtra + segofs +  *(genetic + staofs + i)) += *ARY_POS(i, j);
      if(*(genetic + staofs + i) != *(genetic + staofs + j))
      *(segtra + segofs + *(genetic + staofs + i)) += *ARY_POS(j, i);
    }
  }

  /* 総トラヒック量(最悪ケース)、最大トラヒック量(最適ケース)の計算 */
  for(i=0;i<segment;i++)
    if(worst)
      tra += *(segtra + segofs + i);
    else if(*(segtra + segofs + i) > tra)
      tra = *(segtra + segofs + i);

  return(tra);

}

/**********/
/** 交叉 **/
/**********/

void crossover()
{
  int i,j,k;
  int rnd1,rnd2;
  int sel_gen1,sel_gen2,c_point;
```

```
  for(k=0;k<group;k++){
    for(i=0;i<population;i++)
      /* エリート保存戦略 */
      /* 優秀な遺伝子はそのまま保存 */
      if(i < N_SELECT)
      for(j=0;j<station;j++)
        *(genetic + station*(population*k + i) + j) =
          *(s_genetic + station*(population*k + i) + j);
      else if(*(segtra_fit_sum + k) == 0)
      break;
      else{
      /* ルーレット戦略 */
      /* 適応度の合計の範囲で乱数を発生 */
      rnd1 = get_rand(*(segtra_fit_sum + k));
      /* 適応度に応じて個体を選択 */
      for(j=0;j<N_SELECT;j++)
        if(*(segtra_fit_tbl + N_SELECT*k + j) > rnd1){
          sel_gen1 = j;
          break;
        }
      rnd2 = get_rand(*(segtra_fit_sum + k));
      for(j=0;j<N_SELECT;j++)
        if(*(segtra_fit_tbl + N_SELECT*k + j) > rnd2){
          sel_gen2 = j;
          break;
        }

      /* 一点交叉のポイントを決定 */
      c_point = get_rand(station);

      /* 選択された個体の遺伝子を交換 */
      for(j=0;j<station;j++)
        if(j<c_point)
          *(genetic + station*(population*k + i) + j) =
            *(s_genetic + station*(population*k + sel_gen1) + j);
        else
          *(genetic + station*(population*k + i) + j) =
            *(s_genetic + station*(population*k + sel_gen2) + j);
      }
  }
 }

/**************/
/** 突然変異 **/
/**************/

void mutation()
{
  int i,j,k;
  int rnd1,rnd2;

  for(k=0;k<group;k++){
```

```
    for(i=1;i<population;i++)
      if(P_TURNOVER > 0 && get_rand(100) < P_TURNOVER*100){
      /* ターンオーバー */
      /* 特定の個体の遺伝子の値をインクリメント */
      for(j=0;j<station;j++)
        if(*(genetic + station*(population*k + i) + j) == segment - 1)
          *(genetic + station*(population*k + i) + j) = 0;
        else
          *(genetic + station*(population*k + i) + j) =
          *(genetic + station*(population*k + i) + j)  + 1;
      }
      else if(P_MUTATION > 0 && get_rand(100) < P_MUTATION*100){
      /* 突然変異 */
      /* 特定の個体の遺伝子の位置を交換 */
      rnd1 = get_rand(station);
      rnd2 = get_rand(station);
      SWAP(*(genetic + station*(population*k + i) + rnd1),
           *(genetic + station*(population*k + i) + rnd2));
      }
  }
}

/**********/
/** 移住 **/
/**********/

void migration()
{
  int i,j;

  /* 隣接するサブ集団間で一番優秀な遺伝子を交換 */
  for(i=1;i<group+1;i++)
    for(j=0;j<station;j++)
      if(i == group)
      *(s_genetic + station*(population*i - 1) + j) =
        *(s_genetic + j);
      else
      *(s_genetic + station*(population*i - 1) + j) =
        *(s_genetic + station*population*i + j);

}

/******************************/
/** 各状態の表示(デバック用) **/
/******************************/

void display(type)
int type;
{
  int i,j,k;

  switch(type){
```

```
  case GENE:
    for(k=0;k<group;k++){
      printf("group%d¥n",k);
      for(i=0;i<population;i++){
      printf("genetic%d = ",i);
      for(j=0;j<station;j++){
        if(j%4 == 0)
          printf("_");
        printf("%d",*(genetic + station*population*k + (station*i + j)));
      }
      printf("¥n");
      }
    }
    break;
  case S_GENE:
    for(k=0;k<group;k++){
      printf("group%d¥n",k);
      for(i=0;i<population;i++){
      printf("s_genetic%d = ",i);
      for(j=0;j<station;j++){
        if(j%4 == 0)
          printf("_");
        printf("%d",*(s_genetic + station*(population*k + i) + j));
      }
      printf("¥n");
      }
    }
    break;
  case CUT:
    for(k=0;k<group;k++){
      printf("group%d¥n",k);
      for(i=0;i<population;i++){
      printf("segtra%d = ",i);
      for(j=0;j<segment;j++)
        printf("%d, ",*(segtra + segment*population*k + segment*i + j));
      printf("s_cut = %d sort_ary = %d¥n",
            *(s_segtra + population*k + i),*(sort_ary + population*k + i));
      }
      printf("¥n");
    }
    break;
  case FIT:
    for(k=0;k<group;k++){
      printf("¥ngroup%d¥n",k);
      for(i=0;i<N_SELECT;i++)
      printf("fit = %d,¥tfit_tbl = %d¥n",
            *(segtra_fit + N_SELECT*k + i), *(segtra_fit_tbl + N_SELECT*k +
i));
      printf("seg_tra_fit_sum = %d¥n",*(segtra_fit_sum + k));
    }
  }
}
```