UNIVERSIDADE FEDERAL DE SANTA CATARINA UFSC
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ANDERSON ANDREI DE BONA

ALGORITMO DE OTIMIZAÇÃO COMBINATORIAL: UMA PROPOSTA HÍBRIDA UTILIZANDO OS ALGORITMOS SIMULATED ANNEALING E GENÉTICO EM AMBIENTE MULTIPROCESSADO

Dissertação submetida à Universidade Federal de Santa Catarina como parte dos requisitos para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Luis Fernando Friedrich, Dr.

Florianópolis, Julho de 2005.

ALGORITMOS DE OTIMIZAÇÃO COMBINATORIAL: UMA PROPOSTA HÍBRIDA UTILIZANDO OS ALGORITMOS SIMULATED ANNEALING E GENÉTICO EM AMBIENTE MULTIPROCESSADO

Anderson Andrei De Bona

Esta Dissertação foi julgada adequada para a obtenção do título de Mestre em Ciência da Computação Área de Concentração Sistemas de Computação e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

_____

Prof. Raul Sidnei Wazlawick, Dr.

Coordenador

Banca Examinadora

_____

Prof. Luis Fernando Friedrich, Dr. (Orientador)

_____

Prof. José Mazzucco Júnior, Dr.

_____

Prof. Mário Antonio Ribeiro Dantas, Dr.

_____

Prof. Rômulo Silva de Oliveira, Dr.

“A mente que se abre a uma nova idéia,
jamais voltará ao seu tamanho original”
(Albert Einstein)

Ofereço este trabalho a todos os Professores do PPGCC-UFSC, em especial aos colaboradores do LACPAD (Laboratório de Computação Paralela e Distribuída).

Dedico este trabalho ao meu pai Itacir De Bona.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

A busca por soluções de problemas envolvendo otimização combinatorial tem sido motivo de estudos e pesquisas há muito tempo. Grande parte dos métodos propostos para a resolução de problemas desse tipo, que buscam soluções ótimas, está baseada em técnicas conhecidas como *branch-and-bounds.* Entretanto, o principal problema desse tipo de abordagem consiste no esforço computacional exigido. O tempo de computação necessário para a determinação de uma solução pode atingir níveis impraticáveis, tornando-os muitas vezes inviáveis em aplicações práticas.

Como alternativa, atualmente, diversos métodos de aproximação estão sendo propostos. São abordagens que buscam soluções aceitáveis, próximas às soluções ótimas, porém, com tempos de processamento viáveis. Como exemplos típicos dessa abordagem podem ser citados os algoritmos das Formigas, Genéticos, *Simulated Anneling*, etc.

Nesta dissertação é apresentado um novo algoritmo de aproximação que poderá ser empregado em problemas dessa natureza. Basicamente, o que está sendo proposto é a utilização do algoritmo *Simulated Annealing* em sua forma original, combinado com os operadores *crossovers* dos Algoritmos Genéticos. Além da hibridização dos algoritmos aludidos, também é explorada neste trabalho a potencialidade da concorrência e paralelização dos mesmos em um ambiente multiprocessado.

Na implementação e nos testes do modelo proposto foi utilizado o clássico Problema do Caixeiro Viajante que é um dos representantes desta classe de problema de otimização combinatorial, mais utilizados como *benchmark.*

ABSTRACT

The search for solutions in combinatorial optimization problems has been considered as a research goal for a long time. Most of the methods that have been proposed for resolution of this kind of problem seek for optimal solution and are based on branch-and-bounds technique. However, this type of the approach always demands a great computational effort. The computation time required to find a solution could be out of an acceptable level.

Nowadays many approximation methods have been proposed. These are approaches that seek for an acceptable solution, close to optimal solution, with an acceptable computation time. For instance, the ant algorithm, genetic algorithm and simulated annealing are frequently used as approximation methods.

This dissertation presents a new approximation algorithm that can be used for combinatorial optimization problem. Basically, we propose the utilization of simulated annealing algorithm in its original form, combined by the crossover operators of genetic algorithm. In addition, we have explored the potentiality of concurrent and parallel execution in a multiprocessor machine.

In implementing and the testing of the proposed model one has used the classic Traveling Salesman Problem which is a very representative and used benchmark of combinatorial optimization problems.

CAPÍTULO I
INTRODUÇÃO

Apesar da computação estar se alastrando pelos mais diversos ramos da nossa vida, resolvendo inúmeros problemas que nos cercam nas mais diferentes atividades, ainda deparamos com problema relativamente simples que, entretanto, não são resolvidos por ela na sua forma final. A classe dos problemas de otimização combinatorial é um exemplo da afirmação acima. Nessa classe podem ser encontrados problemas simples que, entretanto, apresentam tempos de execução com grandezas inimagináveis na busca de suas soluções ideais. Muitos desses problemas são classificados, na teoria da computação, como NP-completos. Mesmo com a capacidade atual de processamento que os computadores apresentam, ainda assim não é possível resolver problemas que possuam essa classificação. Em decorrência desse fato, vem sendo dada muita ênfase na pesquisa e desenvolvimento de heurísticas que forneçam soluções de boa qualidade em tempo de processamento compatível com as necessidades requeridas. Segundo MAZZUCCO (1999), "novas técnicas, especialmente as metaheurísticas, tais como *Tabu Search*, *Simulated Annealing*, algoritmos Genéticos, algoritmos das Formigas etc, são de muita importância para a solução destes problemas de otimização combinatorial e têm se mostrados bastantes eficientes em muitos casos". Essas técnicas buscam, portanto, não a identificação da solução exata, mais sim de uma solução aceitável, que se aproxime da solução ótima, em tempo computacional viável.

Propõe-se nesse trabalho um modelo que se baseia na utilização de duas técnicas conhecidas para a solução de problemas de otimização, que são os Algoritmos Genéticos (AG) e o algoritmo *Simulated Annealing* (SA). A proposta, basicamente, tem como objetivo aumentar a potencialidade do algoritmo SA, através de uma abordagem do mesmo em paralelo, juntamente com o operador *crossover* dos AG, utilizando uma arquitetura multiprocessada para a realização dos testes. Para avaliação do modelo proposto foram utilizadas instâncias públicas do problema do caixeiro viajante disponíveis na *Internet* cujas soluções são conhecidas.

1.1 Objetivo do Trabalho

O principal objetivo dessa dissertação foi a realização de um estudo da utilização de dois métodos utilizados na resolução de problemas de otimização combinatorial, os algoritmos: Genético (AG) e *Simulated Annealing* (SA). O trabalho realizado esteve focado na pesquisa de diferentes maneiras de se realizar hibridização desses dois algoritmos na tentativa de se aumentar as suas potencialidades.

1.2 Estrutura do Trabalho

Esta dissertação é composta por sete capítulos organizados da forma como segue.

O capítulo II apresenta uma fundamentação básica conceitual a respeito de problemas de otimização combinatória, problemas NP-Completos e alguns dos métodos utilizados no tratamento dos mesmos.

No capítulo III, é introduzida uma fundamentação teórica a respeito dos algoritmos *Simulated Annealing* e Genético, onde são apresentadas suas concepções básicas.

O capítulo IV apresenta de forma abrangente os principais conceitos referentes à computação paralela e distribuída, dando ênfase à programação *multithread* na linguagem de programação *Java*, uma vez que essa foi a tecnologia utilizada na implementação do modelo proposto neste trabalho.

No capítulo V é apresentado o modelo proposto e implementado que combina os Algoritmos *Simulated Annealing* e Genético objetivando o aumento da potencialidade do primeiro.

O capítulo VI apresenta uma análise comparativa dos testes realizados, visando demonstrar o desempenho do algoritmo proposto na resolução de problemas de otimização combinatorial.

Concluindo a dissertação, são apresentados no capítulo VII, as considerações finais e sugestões para trabalhos futuros.

CAPÍTULO II
PROBLEMAS DE OTIMIZAÇÃO

Neste capítulo são apresentados, inicialmente, de forma básica, os conceitos da teoria da complexidade, buscando classificar o tipo do problema que está sendo tratado. Em seguida, são abordados conceitos fundamentais relacionados com a resolução de problemas de otimização combinatorial, que é o assunto diretamente relacionando com o trabalho proposto.

2.1 Introdução

Construir um programa de computador capaz de solucionar um determinado problema exige que o mesmo seja modelado em termos de um algoritmo computacionalmente executável. Problemas aparentemente triviais para os seres humanos, podem se tornar extremamente complexos quando da tentativa de modelá-los para uma solução via computador, como por exemplo, o simples fato de reconhecer uma pessoa numa foto. Em muitos casos o problema é a construção do algoritmo computacional, em outros, o problema é o tempo de execução desses algoritmos, e já em outros, o problema reside na forma como os computadores atuais lidam com as informações. Uma boa parte da pesquisa em ciência da computação consiste em projetar e analisar algoritmos.

A busca de solução para problemas de elevado nível de complexidade computacional tem sido um desafio constante para pesquisadores das mais diversas áreas. Particularmente, em otimização combinatorial, pesquisa operacional, ciência da computação, matemática etc. Muitos problemas de otimização combinatorial possuem soluções algorítmicas, mas geralmente sua viabilidade restringe-se a instâncias de pequeno e médio porte.

De forma genérica, pode-se classificar os problemas de otimização combinatorial como sendo a seleção, a partir de um conjunto discreto e finito de dados,

do melhor subconjunto que satisfaça a determinados critérios pré-definidos (GOLDBERG & LUNA, 2000).

2.2 Complexidade Computacional

Embora muitas vezes confundido, a teoria da complexidade computacional, uma importante área da ciência da computação, estuda a complexidade dos problemas e não dos algoritmos para resolvê-los. Segundo ROUTO (1991), “a complexidade de um problema é medida através do consumo do tempo ou do número de operações executadas para resolvê-lo”.

Segundo DALLE MOLE (2002), “Um modelo computacional que pode ser empregado no estudo da complexidade computacional é a máquina de Turing, que proporciona uma maneira de demonstrar a existência de problemas insolúveis (que não são computáveis). A máquina de Turing não somente pode ser empregada na determinação dos limites da computabilidade, como também, na classificação dos problemas que são computáveis e os que não são computáveis – os que têm e os que não têm algoritmos para encontrar suas soluções – pelo trabalho computacional necessário para processar esses algoritmos”. Assim sendo de acordo com GERSTING (1993), “Pela tese de Church, qualquer algoritmo pode ser expresso na forma de uma máquina de Turing. Desta forma, a quantidade de trabalho necessária é o numero de passos da máquina de Turing (um por ciclo de tempo) que são necessários para que a máquina de Turing pare”. De acordo com DALLE MOLE (2002), “Podemos distinguir duas classes de problemas distintos, os que são computáveis e os que não são. Os problemas computáveis são aqueles para os quais pode-se definir um algoritmo computacionalmente executável. Mesmo como a existência de um algoritmo, entretanto, alguns problemas são considerados intratáveis computacionalmente, dado o tempo necessário para a execução do mesmo. Problemas de otimização combinatorial são exemplos desses casos. Uma solução computacional para um problema qualquer é um algoritmo ou conjunto de algoritmos interdependentes articulados em um programa, de tal forma que sua execução fornece a solução ou o conjunto de solução do problema em questão”.

De acordo com LEWIS & PAPADIMITRIOU (2000), “Como problema não computável podemos citar a própria matemática, uma vez que já foi provado que não se pode criar um algoritmo que gere as deduções para todas as verdades da matemática”. Analisar a complexidade computacional de um problema é uma questão crítica quando o objeto de estudo é um problema cujo espaço de soluções cresce de forma não polinomial. Essa complexidade diz respeito à análise dos recursos computacionais necessários para a computação do algoritmo tais como memória e número de operações a serem executadas. O tempo de execução real pode variar muito de uma máquina para outra, dessa forma não é um bom parâmetro para a análise da complexidade. Por outro lado, o número de operações necessárias se mantém fixo independentemente da máquina onde o algoritmo é executado. Dessa forma, determinar o tempo computacional de um algoritmo significa determinar o tempo de execução em termos de operações de máquina realizadas (MAZZIERO, 2003).

O conceito de instância na teoria da complexidade computacional é bastante importante e corresponde a uma entrada para um determinado algoritmo. Dessa forma, dado um problema, o algoritmo proposto deve ser capaz de resolver todas as instâncias possíveis para aquele problema.

Podemos imaginar, sem perda de generalidade, para efeito de simplificação do raciocínio, que estamos tratando com problemas cujas instâncias são cadeias de caracteres e cujas soluções também o são. Dessa forma, o comprimento de uma instância de um problema passa a ser o número de caracteres que a compõe, ou seja, o comprimento da cadeia de entrada para o algoritmo. Normalmente, o tamanho de uma instância é denotado por N.

Também, para efeito de simplificação do raciocínio e novamente sem perda da generalidade, podemos supor que estamos tratando com problemas de decisão, ou seja, problemas cujas instâncias têm apenas soluções SIM ou NÃO. Uma instância cuja solução é SIM é considerada uma instância positiva, caso contrário, trata-se de uma instância negativa.

Dentro desse contexto podemos definir então complexidade de um problema, como sendo o consumo de tempo (operações) de um algoritmo ótimo para aquele problema. O consumo sendo medido em função do tamanho da instância, ou seja, *N*. É importante observar que foi utilizado o termo “algoritmo ótimo para aquele problema” uma vez que não necessariamente necessita ser o melhor algoritmo que resolva aquele problema, que inclusive, pode nem ter sido ainda determinado. Um problema é considerado como *polinomial* se a complexidade do seu algoritmo poder ser expressa por $O(N^k)$, para algum *k* e uma instância *N*. Em outras palavras, se o número de operações consumidas na execução do algoritmo, para aquela instância *N*, puder ser expressa por uma função polinomial de um grau determinado *k* PAPADIMITRIOU (1982). Da mesma forma, define-se um problema como sendo *não polinomial* se não existir um *k* tal que a complexidade do problema seja $O(N^k)$, para uma instância *N*.

Com as definições acima fornecidas pode se compreender mais facilmente a conhecida divisão da complexidade de problemas nas três classes tradicionais: classe P, NP e NP-completo. A classe P de problemas é o conjunto formado por todos os problemas polinomiais, ou seja, é o conjunto dos problemas cujos algoritmos possuem a complexidade $O(N^k)$, para algum *k* e uma instância *N*. Não é trivial determinar se um problema está ou não em P, para muitos problemas, inclusive, essa questão ainda é indeterminada, pois não foram determinados algoritmos polinomiais que os resolvam e também não se consegue afirmar que esses algoritmos não existem.

A classe NP de problemas, entretanto, muitas vezes confundida com “Não Polinomial”, tem a sua definição mais complexa e para se alcançá-la é necessário o conhecimento do conceito de algoritmo verificador de um problema de decisão. Um algoritmo verificador para um problema de decisão é um algoritmo que recebe uma instância supostamente positiva, ou seja, pertencente ao problema em questão, e uma cadeia de caracteres que representa uma forma de certificação de que aquela instância é realmente positiva. Como saída anuncia se o certificado recebido atesta realmente que a instância é positiva. O algoritmo verificador por sua vez é um algoritmo polinomial, ou seja, sua complexidade poder ser expressa por $O(N^k)$, para algum *k* e uma instância *N*.

Agora podemos entender a classe dos problemas NP como sendo o conjunto de todos os problemas de decisão que admitem um algoritmo verificado, em outras palavras, a classe NP de problemas é formada por todos os problemas cujas instâncias positivas apresentam um certificado que é aceito por um algoritmo verificador. Facilmente pode se verificar que a classe P de problemas está contida na classe NP.

Para se chegar à definição da classe NP-completo faz se necessário o entendimento de redução entre problemas. Diz-se que um problema de decisão PL' é redutível a um problema PL se PL' constitui um subproblema de PL, ou seja, PL' é um caso particular de PL. Mais precisamente, pode se afirmar que um problema PL' é redutível a um problema PL se existir um algoritmo capaz de transformar todas as instâncias *i'* de PL' em instâncias *i* de PL, tal que, uma instância *i'* de PL' tem solução SIM, se e somente se a instância *i* correspondente em PL tenha solução SIM.

Um problema PL pertence à classe NP-completo se PL está em NP e qualquer outro problema em NP puder ser reduzido à PL através de um algoritmo polinomial. Em outras palavras, um problema PL é NP-completo se, para qualquer problema PL' em NP, existir um algoritmo polinomial para PL que possa ser adaptado para resolver PL' em um tempo polinomial.

Segundo LEWIS & PAPADIMITRIOU (2000), "A classe NP-completo constitui a classe dos problemas considerados difíceis e se um problema está em NP-completo então não existe um algoritmo polinomial capaz de resolvê-lo". Um expressivo representante dessa classe é o Problema do Caixeiro Viajante utilizado nesta dissertação como *benchmark* para o modelo proposto e implementado.

2.3 Espaço de busca

Como já ressaltado anteriormente, grande parte dos métodos propostos para a resolução de problemas de otimização combinatorial, que busca soluções ótimas, está baseada em técnicas conhecidas como *branch-and-bounds.* A grande desvantagem

desse tipo de abordagem consiste no esforço computacional exigido (MAZZUCCO, 1999).

Segundo MAZZUCCO (1999), “Atualmente, diversos métodos de aproximação estão sendo propostos. São abordagens, também conhecidas como métodos de busca, que objetivam alcançar soluções aceitáveis, próximas às soluções ótimas, porém, com tempos de processamento viáveis. Como exemplos típicos dessa abordagem podem ser citados os algoritmos das Formigas, Algoritmos Genéticos, *Simulated Anneling* etc”.

Um dos principais obstáculos encontrados nesse tipo de abordagem no mundo real é a magnitude do espaço de busca do problema a ser tratado. O espaço de busca de um problema é o conjunto de todas as possíveis soluções para o mesmo.

2.4 Paradigmas de solução de problemas

Encontrar uma solução de um problema consiste em buscar um valor ou um conjunto de valores que satisfaça o enunciado do mesmo. Dessa forma, pode-se encontrar soluções através de métodos de resolução que, segundo ZUBEN (2000), podem ser classificados em: métodos fortes, métodos específicos e métodos fracos.

Métodos fortes: são concebidos para resolver problemas genéricos, mas foram desenvolvidos para operar em um mundo específico, onde se exige propriedades restritivas das funções matemáticas envolvidas.

Métodos específicos: são concebidos para resolver problemas específicos em mundos específicos.

Métodos fracos: são concebidos para resolverem problemas genéricos em mundos genéricos. Operam em mundos não-lineares e não-estacionários e embora não garantindo eficiência total na obtenção da solução ótima, geralmente, garantem a obtenção de uma “boa aproximação” para a mesma.

Existem três métodos que podem ser utilizados para pesquisar o espaço de soluções de um problema de otimização combinatorial: métodos numéricos, métodos enumerativos e métodos probabilísticos. Todos com seus respectivos derivados e ainda um grande número de métodos híbridos (TANOMARU, 1995).

Segundo ZUBEN (2000), “Independente da aplicação, métodos fracos devem ser considerados se e somente se métodos fortes (soluções clássicas) e métodos específicos (soluções dedicadas) não existem, não se aplicam, ou falham quando aplicados”.

Segundo DALLE MOLE (2002), “A utilização de métodos fracos ou probabilísticos tem ganhado ênfase como proposta de solução para problemas antes considerados como computacionalmente intratáveis. A idéia de obter uma solução aproximada em um tempo computacional aceitável tem levado pesquisadores a desenvolverem e aperfeiçoarem essa técnica”.

CAPÍTULO III
ALGORITMOS SIMULATED ANNEALING E GENÉTICO

São abordados neste capítulo os tópicos fundamentais relacionados com os algoritmos que constituem a base da proposta deste trabalho, ou seja, o algoritmo *Simulated Annealing* (SA), e o Algoritmo Genético (AG).

3.1 O algoritmo *Simulated Annealing*

3.1.1 Introdução

Segundo RODRIGUES (2000), “Na física da matéria condensada, recozimento (*annealing*) é um processo térmico utilizado para obtenção de estados de baixa energia em um sólido. Esse processo consiste em duas etapas: na primeira, a temperatura do sólido é aumentada para um valor máximo no qual ele se funde; na segunda, a temperatura é reduzida lentamente até que o material se solidifique. Na segunda fase, o resfriamento deve ser realizado muito lentamente, possibilitando aos átomos que compõem o material, tempo suficiente para se organizarem em uma estrutura uniforme com energia mínima. Se o sólido for resfriado bruscamente, seus átomos formarão uma estrutura irregular e fraca, com alta energia, em conseqüência do esforço interno gasto”.

Segundo MAZZUCCO (1999), “Computacionalmente, o recozimento pode ser visto como um processo estocástico de determinação de uma organização dos átomos de um sólido, que apresente energia mínima. Em temperatura alta, os átomos se movem livremente e, com grande probabilidade, podem se mover para posições que incrementarão a energia total do sistema. Quando se baixa a temperatura, os átomos gradualmente se movem em direção a uma estrutura regular e, somente com pequena probabilidade, incrementarão suas energias”.

Assim sendo, ao realizar o resfriamento, os átomos gradualmente se movem em direção a uma estrutura regular, e somente com pequena probabilidade incrementarão suas energias. Esse processo foi simulado em computador, com sucesso, por METROPOLIS et al. (1953). Segundo RODRIGUES (2000), “O algoritmo utilizado

baseava-se em métodos de Monte Carlo e gerava uma seqüência de estados de um sólido da seguinte maneira: dado um estado corrente *i* do sólido com energia *Ei,* um estado subseqüente era gerado pela aplicação de um mecanismo de perturbação, o qual transformava o estado corrente em um próximo estado por uma pequena distorção, por exemplo, pelo deslocamento de uma única partícula. A energia no próximo estágio passa a ser *Ei.* Se a diferença de energia fosse menor ou igual a zero, o estado *j* era aceito como estado corrente". Se a variação fosse maior que zero, o estado *j* era aceito com uma probabilidade dada por:

$$exp\ ((Ei - Ej)/(kB*T)),$$

onde *T* representa a temperatura atual do sistema e *kB* é uma constante física conhecida como constante de Boltzmann. Essa regra de aceite é conhecida como critério de Metropolis e o algoritmo também leva o seu nome.

No início dos anos 80, KIRKPATRICK et al. (1983) desenvolveram um algoritmo de utilização genérica, análogo ao de Metropolis, denominado Algoritmo *Simulated Annealing*. Na figura 3.1, pode ser visualizada a função *P*, utilizada no critério de aceite:

$$P_{c_k}(aceitar\ j) = \begin{cases} 1 & se\ g(j) \le g(i) \\ exp\left(\dfrac{-[g(j) - g(i)]}{c_k}\right) & se\ g(j) > g(i) \end{cases}$$

Figura 3.1 – Função de aceite *Simulated Annealing* KIRKPATRICK et al. (1983).

Onde *g* é a função a ser otimizada (minimizada) e *i* e *j* são, respectivamente, a solução corrente e uma solução candidata. C_k é um parâmetro representando a temperatura *T*.

De acordo com RODRIGUES (2000), "Se uma solução candidata *j* é melhor que a solução corrente *i*, ou seja, *(g(j)£g(i))*, esta é aceita com probabilidade 1. Caso contrário, a solução candidata ainda poderá ser aceita com uma dada probabilidade. Essa probabilidade é calculada em função da variação da energia, definida por *D=g(j)-g(i)* e da temperatura (C_k). A medida em que a temperatura diminui o algoritmo torna-se mais seletivo, passando a aceitar com menor freqüência soluções que apresentem grande

aumento na variação de energia, isto é, soluções que sejam muito piores que a solução corrente. Essa probabilidade tende a zero à medida que a temperatura se aproxima do ponto de congelamento.”

O algoritmo *Simulated Annealing* pode ser considerado como uma extensão do método original de busca local. A busca local requer somente a definição de um esquema de vizinhança e um método de avaliação do custo de uma solução em particular, sempre apresenta uma solução final (LEE, 1995). Entende-se por um esquema de vizinhança um mecanismo apropriado, através do qual se obtém uma nova solução, também pertencente ao espaço de soluções do problema, realizando uma pequena alteração na solução corrente. Esse mecanismo é altamente dependente do problema que está sendo tratado.

O método de busca local é ineficiente quanto às armadilhas constituídas pelos ótimos locais, fazendo deste método uma heurística pobre para muitos problemas de otimização combinatorial (CORRÊA, 2000). Uma propriedade desejável de qualquer algoritmo é a habilidade de encontrar uma boa solução, independente do ponto de partida. Um ótimo local se caracteriza quando o algoritmo atinge uma região correspondente ao fundo de um vale (um ponto de mínimo), em se tratando de um problema de minimização, que não contém a solução ótima e dele não consegue sair, uma vez que todas as soluções naquela vizinhança possuem valores maiores do que a solução corrente.

Uma estratégia para escapar da armadilha do ótimo local é executar diversas vezes o algoritmo com diferentes soluções iniciais, sendo adotado como solução ótima a melhor solução encontrada. Entretanto, esse procedimento conduz a um novo problema que é o de determinar quando parar o algoritmo, além de poder ser inviável em se tratando de grandes problemas (ARAUJO, 2001).

O algoritmo *Simulated Annealing*, por sua vez, consegue escapar de um ótimo local uma vez que o aceite de uma nova solução não depende única e exclusivamente do seu valor (HIROYASU et al, 2000). Mesmo sendo inferior à solução corrente, uma nova

solução pode ser aceita de forma probabilística. Como já aludido anteriormente, esse aceite ou não aceite é determinado por uma função que também leva em consideração a diferença entre as duas soluções envolvidas e a temperatura atual do processo.

3.1.2 O Algoritmo *Simulated Annealing* em pseudocódigo

Uma outra propriedade importante do algoritmo *Simulated Annealing*, conforme mostra a figura 3.2, através de um pseudocódigo, é a simplicidade de sua implementação computacional.

```
Algoritmo Simulated Annealing
Início
 obtenha a  constante α  e o número de repetições NR;
  S ← S0;
  T ← LS;              // Limite Superior
  TMIN ← LI;         // Limite Inferior
   enquanto (T  > TMIN) faça
     para I de 1 até NR faça
         gerar uma solução S' de N(S);        // S' recebe uma solução gerada na vizinhança de S
       avaliar a variação de energia          // ΔE = f (S') – f (S);
          se ΔE <=  0 então
            S ← S';
          senão
            gerar    rand ∈  random [ 0, 1 ];
              se  rand < exp (-ΔE  /  T)  então
                     S ← S';
               fimse
          fimse
     fimpara
                T ← T * α;
        fimenquanto
```

Figura 3.2 - Pseudocódigo do algoritmo *Simulated Annealing,* ARAUJO (2001).

Procurando evitar uma convergência precoce para um mínimo local, o algoritmo inicia com um valor de *T* relativamente alto. Esse parâmetro é gradualmente diminuído e, para cada um dos seus valores, são realizadas várias tentativas (*NR*) de se alcançar uma melhor solução, nas vizinhanças da solução corrente.

Na figura 3.2, a expressão ($T \leftarrow T * a$) corresponde ao processo de diminuição da temperatura, normalmente o parâmetro α é uma constante menor do que um.

De acordo com MAZZUCCO (1999),"matematicamente, o algoritmo *Simulated Annealing* pode ser modelado através da teoria de cadeias de Markov. Utilizando esse modelo, vários resultados importantes, tratando de condições suficientes para a convergência, têm surgido na literatura. A grande maioria desses trabalhos, entretanto, não leva em consideração o número de iterações necessárias para se atingir essa convergência. Uma vez que o tamanho do espaço de solução *E* cresce exponencialmente com o tamanho do problema, o tempo de execução de um algoritmo desse tipo pode alcançar níveis inviáveis. Um resultado muito importante é fornecido no trabalho publicado por (HAJEK, 1988), no qual, não só as condições necessárias e suficientes para a convergência assintótica do algoritmo para um conjunto de soluções ótimas globais são fornecidas, mas também o número de iterações necessário para que essa convergência possa ocorrer".

3.2 Algoritmo Genético

3.2.1 Introdução

No final da década de 1970, o professor John Holland da Universidade de Michigan (HOLLAND, 1993), em suas explorações dos processos adaptativos de sistemas naturais e suas possíveis aplicabilidades em projetos de *softwares* de sistemas artificiais, conseguiu incorporar características da evolução biológica natural a um algoritmo para constituir um método extremamente simples de resolução de problemas complexos, imitando o processo natural da evolução dos seres vivos. Inicialmente o seu trabalho envolvia algoritmos que manipulavam cadeias de dígitos binários, as quais chamou de cromossomos. Seus algoritmos realizavam evoluções simuladas em populações de tais cromossomos. Tal como nos processos biológicos, os algoritmos desconheciam totalmente a natureza do problema que estava sendo resolvido. As únicas informações que lhes eram fornecidas consistiam das avaliações de cada cromossomo que os mesmos produziam. Essas informações eram utilizadas na condução do processo de seleção dos cromossomos que iriam reproduzir para formar a próxima população. Os cromossomos que melhor evoluíam apresentavam uma tendência a se reproduzirem mais freqüentemente do que aqueles cuja evolução era ruim (MAZZUCCO, 1999).

Esse tipo de algoritmo, conhecido por Algoritmo Genético, que através de um simples mecanismo de reprodução sobre um conjunto de soluções codificadas de um determinado problema, consegue produzir solução de boa qualidade, constitui hoje uma importante técnica de busca, empregada na resolução de uma grande variedade de problemas, nas mais diversificadas áreas (MAZZUCCO, 1999).

Uma das principais características dos AG, assim como do SA, é a sua simplicidade de implementação. A figura 3.3 apresenta o fluxo básico de um Algoritmo Genético (AG) com os seus três operadores fundamentais: seleção, recombinação (*crossover*) e mutação.

```
Procedimento Algoritmo Genético
início
        t ← 0
        Inicializar P(t)
        Avaliar P(t)
        enquanto (Condição <> Fim) faça
                t ← t + 1
                Selecionar P(t) a partir de P(t-1)
                Recombinar e mutar P(t)
                Avaliar P(t)
        fimenquanto
fim.
```

Figura 3.3 - Fluxo básico do Algoritmo Genético, (TANOMARU, 1995).

Nesse algoritmo, figura 3.3, t é um índice que designa cada geração e P(t) é a população correspondente a aquela geração. Assim sendo, quando t = 0, trata-se da geração inicial, ou seja, P(0) é a população que inicia o processo e normalmente é escolhida de forma aleatória. “Avaliar P(t)” tem o significado da varredura de todas as soluções (cromossomos) que compõem aquela população da geração t para determinar a probabilidade que cada solução terá de se reproduzir para gerar (t+1). Quanto melhor a

sua qualidade maior a probabilidade de se reproduzir. Entretanto, uma solução com baixa probabilidade, não deixa de ter sua chance de se reproduzir. O processo de reprodução apresenta-se no comando “Recombinar P(t)” onde as soluções selecionadas serão combinadas duas a duas para produzirem um novo par de soluções. Ou seja, parte do cromossomo que representa uma solução que vai combinar com parte de outro cromossomo para formar dois novos cromossomos. Esse operador, conhecido por *crossover*, será abordado a seguir com mais profundidade uma vez que o mesmo constitui uma parte importante no modelo proposto nessa dissertação. “Mutar p(t)”, representa um outro operador genético muito importante em alguns casos. Trata-se da mutação de alguns indivíduos tomados aleatoriamente na população corrente. Essa mutação representa uma simples inversão de alguns elementos (genes) que compõem a solução, resultando na criação de um novo indivíduo (TANOMARU, 1995).

3.2.2 Princípios básicos dos Algoritmos Genéticos

Algoritmos Genéticos (AG) são métodos de busca baseados nos mecanismos de evolução natural e na genética. Em AG, uma população de possíveis soluções para o problema em questão evolui de acordo com operadores probabilísticos concebidos a partir de metáforas biológicas, de modo que há uma tendência de que, na média, os indivíduos representem soluções cada vez melhores à medida que o processo evolutivo continua (TANOMARU, 1995).

Segundo MAZZUCCO (1999), “Diferentemente do algoritmo *Simulated Annealing* que trabalha sempre com uma única solução corrente, o algoritmo genético trabalha simultaneamente com um conjunto de soluções, sobre um rico banco de pontos (uma população de cromossomos), subindo vários picos em paralelo e reduzindo, dessa forma, a probabilidade de ser encontrado um falso pico”.

A utilização do algoritmo genético na resolução de um determinado problema depende fortemente da realização de dois importantes passos iniciais: encontrar uma forma adequada de se representar soluções possíveis do problema em forma de cromossomo e determinar uma função de avaliação que forneça uma medida do valor

(da importância) de cada cromossomo gerado, no contexto do problema (MAZZUCCO, 1999). Assim como no SA, esses dois quesitos são de vital importância no sucesso de uma aplicação desses algoritmos na solução de um problema e muito provavelmente é uma das principais barreiras nas suas aplicações.

3.2.3 Representação Cromossômica

A representação cromossômica constitui o primeiro passo para a aplicação de um algoritmo genético na resolução de um problema. Consiste na determinação de uma maneira de se representar cada possível solução S do espaço de busca, como uma seqüência de símbolos gerados a partir de um alfabeto finito A. No caso geral, tanto o método de avaliação quanto o alfabeto genético dependem de cada problema. Porém uma vez definida a estrutura mais apropriada para representar a solução do problema, esta deve poder representar todas as soluções possíveis do problema univocamente, e permanecer imutável no decorrer do processo (MICHALEWICZ, 1999).

Na maioria dos algoritmos genéticos assume-se que cada indivíduo é constituído por um único cromossomo (fato que não ocorre na genética natural), razão pela qual é comum utilizar os termos cromossomos e indivíduos indistintamente. A maior parte dos algoritmos genéticos, proposta na literatura, utilizam uma população de tamanho fixo, com cromossomos também de tamanho constantes (TANOMARU, 1995).

3.2.4 População

É o conjunto de cromossomos representando soluções candidatas a sofrerem os efeitos dos operadores genéticos ao longo do processo evolutivo. As populações caminham em direção à formação de uma população de soluções mais apropriadas ao problema que está sendo resolvido. A população inicial consiste de n indivíduos, normalmente criados de forma aleatória, a partir dos quais outros mais refinados serão determinados. Dessa forma, a população inicial deve ser a mais diversificada possível, de modo que os mais variados pontos do espaço de busca possam ser amostrados.

3.2.5 Operadores Genéticos

São classificados como os mecanismos responsáveis pelas modificações sofridas pelos indivíduos de uma população, sendo os responsáveis diretos pela geração de novas soluções. Os operadores genéticos de um AG devem ser definidos considerando a forma de representação adotada e a natureza do problema em questão. Segundo GOLDBERG (1989), três classes de operadores considerados como básicos, figuram na grande maioria das implementações, são eles: mutação, recombinação e seleção.

3.2.6 Operadores de Mutação

Classificados como operadores que tem a finalidade de simular o fenômeno natural da mutação genética de indivíduos, onde em sua versão básica consiste do operador mudar o valor de alguns genes selecionados aleatoriamente.

Estes operadores conduzem à exploração de novas regiões do espaço de busca do problema, gerando nova informação genética, assim ao sofrer a mutação, um cromossomo passa a mapear um novo ponto do espaço de busca, possivelmente em área ainda não explorada, transferindo a busca empreendida pelo cromossomo para esta área. Estes operadores consistem em um importante mecanismo de manutenção da diversidade da população e de cobertura do espaço de busca. Quando os indivíduos de uma população vão se tornarem muito semelhantes, o efeito do operador de *crossover* vai se anulando gradativamente fazendo com que as populações em gerações sucessivas se tornem cada vez mais semelhantes (MAZZUCCO, 1999).

Entre os vários operadores de mutação, destacam-se o *Swap* e os baseados no operador *Mutation* (DRECHSLER et al., 1995).

3.2.7 Operadores de Recombinação, *Crossovers*

Esses operadores simulam o processo natural de reprodução sexuada e são responsáveis pela transferência de carga genética dos genitores aos seus descendentes. São selecionados pontos de corte nos cromossomos genitores e através de combinações dos fragmentos resultantes são formados os seus descendentes.

Qualquer que seja a variante escolhida, estes operadores atuam sobre o espaço de busca realizando um refinamento das soluções codificadas pelos cromossomos genitores, uma vez que, a recombinação “preserva” a informação genética de boa qualidade dos mesmos, configurando assim, uma busca local a partir dos genitores (TANOMARU, 1995).

Entre os operadores mais sofisticados e vastamente utilizados pode-se citar: *Order Crossover* (OX), *Cycle Crossover* (CX), *Partially-Mapped Crossover* (PMX).

O operador OX (*Order Crossover*) atua sobre os cromossomos genitores através de duas etapas distintas. Na figura 3.4 é exemplificado o funcionamento deste operador. Primeiramente, selecionam-se dois pontos de corte de forma aleatória nos cromossomos genitores e então se copiam os genes situados entre estes pontos do primeiro cromossomo genitor para o primeiro cromossomo descendente, mantendo integralmente, as posições e ordens dos genes. Na Figura 3.4, esse comportamento se refere ao passo 1. Posteriormente as posições restantes deste descendente são preenchidas com os genes do segundo genitor, do segundo ponto de corte em diante (passo 2). Após essa etapa, este procedimento passa atuar a partir de sua primeira posição, configurando um ciclo, encerrando quando todas as posições do descendente forem preenchidas (TANOMARU, 1995).

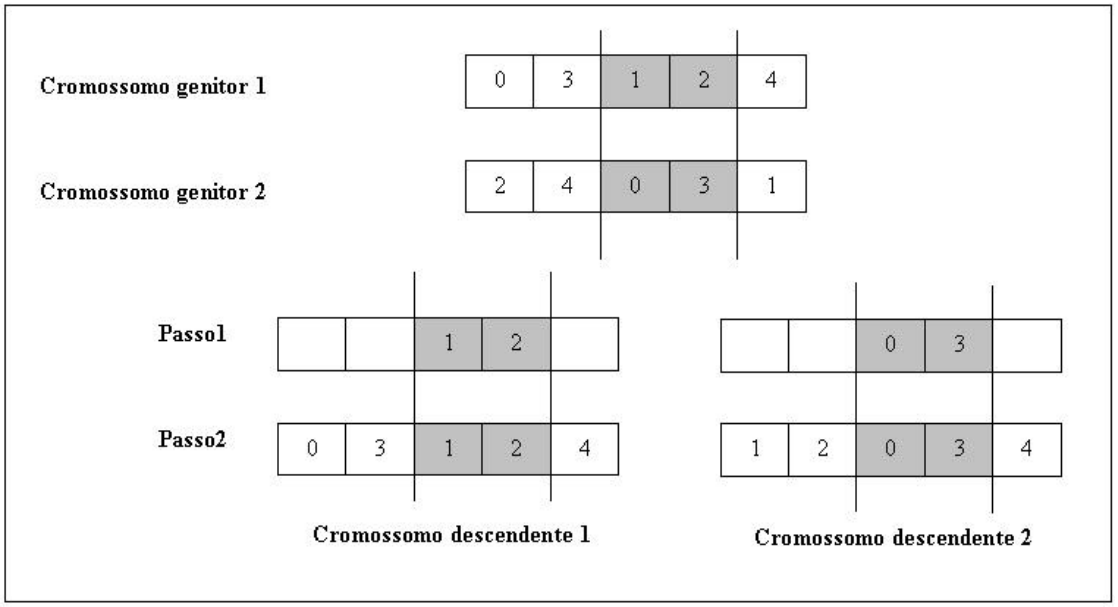

Figura 3.4 - Funcionamento Operador OX (*Order Crossover*).

O operador PMX (*Partially Mapped Crossover*) gera cromossomos correspondentes às permutações dos seus genitores, ao preservar intervalos de recombinação e embaralhar os demais genes. Considerando-se os seguintes cromossomos Cr*1* e Cr*2*, inicialmente dois pontos de corte são escolhidos de forma aleatória, e os genes presentes entre os pontos de corte **[*4* - *7* - *5*]** e **[*6* - *2* - *8*]** são trocados de forma que ambos recebem partes de seqüência de informações genéticas novas, conforme na figura 3.5.

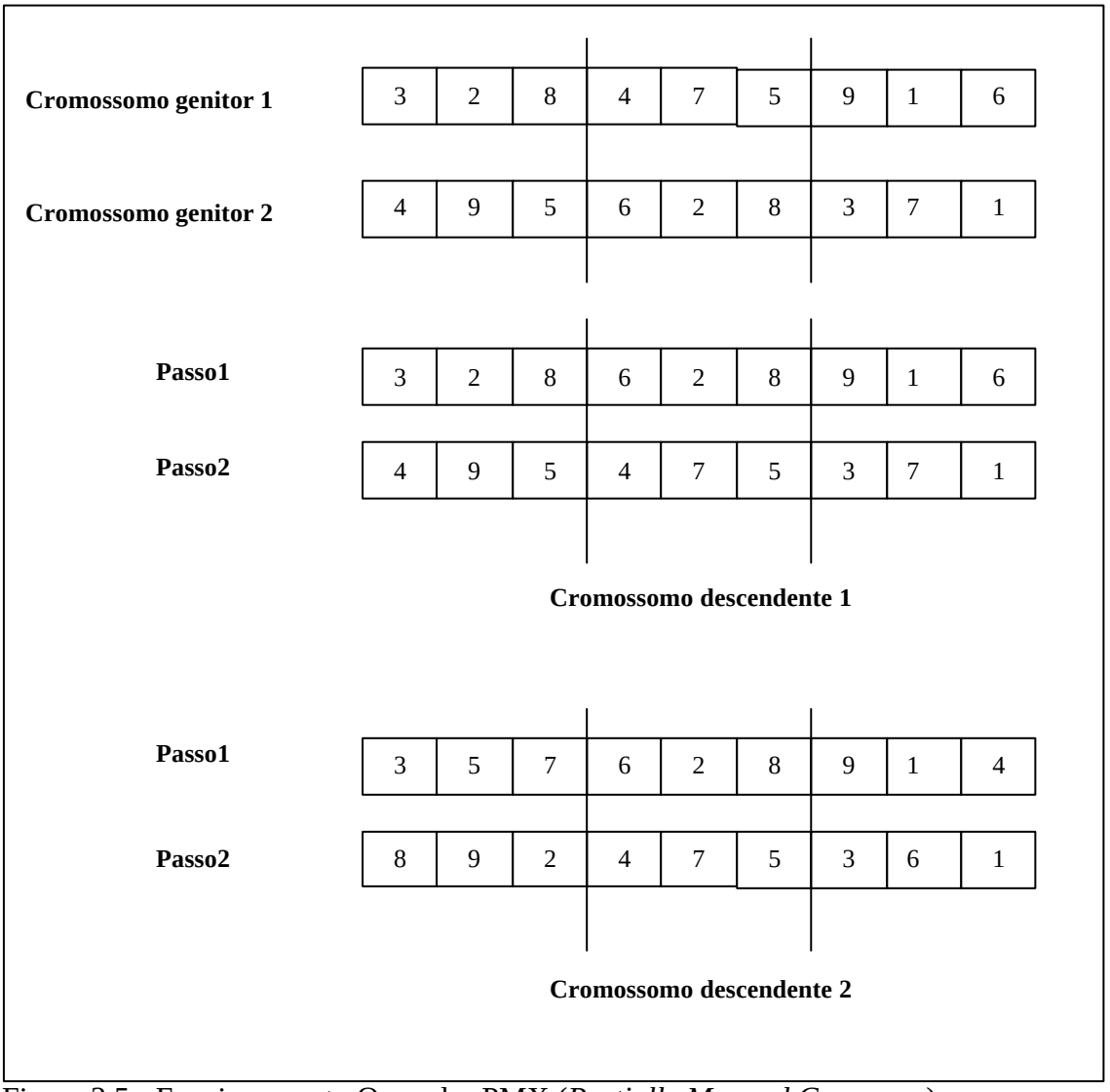

Figura 3.5 - Funcionamento Operador PMX (*Partially Mapped Crossover*).

Por sua vez**, o operador CX (*Cycle Crossover*)** apresenta um esquema de cruzamento totalmente diferente dos operadores OX e PMX já apresentados, uma vez que o mesmo não utiliza pontos de corte. As recombinações são executadas de forma que cada um dos genes de seus descendentes venha de posição correspondente de qualquer um dos pais.

O procedimento se baseia em percorrer os genes dos pais, selecionando quais não podem ser cruzados, sendo cruzados os genes restantes sem que ocorra inversão de posições. Na figura 3.6, pode ser visto que o cromossomo descendente 1 leva o valor do primeiro Cromossomo genitor 1, assim o gene da mesma posição do Cromossomo

genitor 2 possui valor 4. Procurando esse valor no Cromossomo 1, verifica-se que o mesmo encontra se coincidentemente no quarto gene, este valor deve então ser elevado para a quarta posição do Cromossomo descendente 1. Posteriormente, verifica-se que o gene de mesma posição no Cromossomo 2 possui valor 6, e que este valor encontra-se na mesma posição do Cromossomo 1, dessa forma esse valor é transportado para a nona posição do Cromossomo descendente 1. O gene que ocupa a mesma posição no Cromossomo 2 possui valor 1, o qual encontra-se na oitava posição no Cromossomo 1. Transporta-se então este valor para o Cromossomo descendente 1. O processo continua colocando o valor 7 no quinto gene do Cromossomo descendente 1; o valor 2 no segundo gene; o valor 9 no sétimo gene e ao tentar colocar o valor 3 no primeiro gene, observa-se que esta operação já foi realizada. Assim observamos que o clico está completo e o passo seguinte é completar os genes do Cromossomo descendente 1, com os respectivos elementos do Cromossomo 2. O segundo descendente (Cromossomo descendente 2) é obtido através do mesmo processo, começando pelo Cromossomo 2. Todo processo pode ser visto através da figura 3.6.

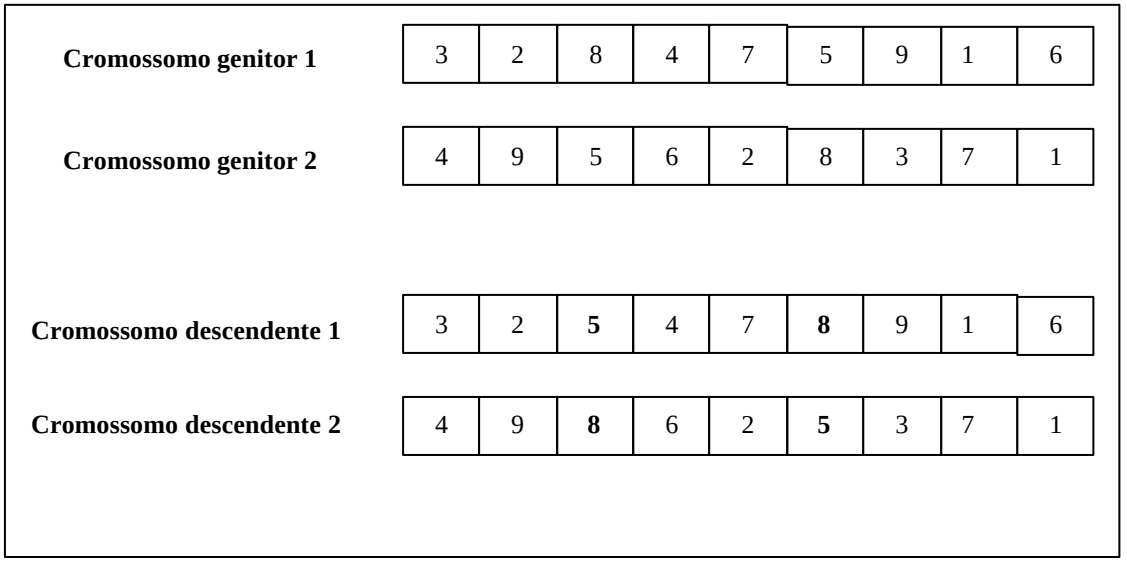

Figura 3.6 - Funcionamento Operador CX (*Cycle Crossover*).

Os resultados teóricos e empíricos que comparam os operadores PMX, OX e CX são encontrados em (GOLDBERG, 1989) e (BLUM & ROLI, 2001). Na realidade os operadores PMX e o OX são semelhantes, a diferença é que o PMX tende a preservar a posição absoluta do gene (devido ao mapeamento ponto a ponto), enquanto o OX tende

a preservar a posição relativa do gene (devido o preenchimento seqüencial dos espaços vazios) (GOLDBERG, 1989).

3.2.8 Operadores de Seleção

Esses operadores são os responsáveis pela escolha do conjunto de indivíduos que participarão do processo de procriação para a formação da próxima geração. Um operador desse tipo normalmente realiza uma seleção probabilística, através de uma função de avaliação, que atribui a cada indivíduo da geração corrente, um valor inteiro que representa o grau de aptidão do indivíduo ao problema que está sendo tratado. Como exemplo, no caso do problema do Caixeiro Viajante, que será utilizado neste trabalho, o valor retornado da função de avaliação poderia ser o tamanho da rota que cada indivíduo está representando.

O valor de cada aptidão individual somente terá utilidade após ter sido transformado em expectativa que seria o número esperado de descendentes que cada indivíduo poderá gerar. Existem vários métodos para realizar a conversão do grau de aptidão em expectativa de descendentes. Segundo MAZZUCCO (1999), “No algoritmo original de Holland, (HOLLAND, 1993), essa expectativa individual era calculada através da divisão da aptidão individual pela média das aptidões de toda a população da geração corrente. Desta forma, a expectativa *e* de um indivíduo *i*, é calculada por:

$$ei = apti / maptt ,$$

onde *apti* é a aptidão do indivíduo *i* e *maptt* é a aptidão média da população no tempo *t*.

As expectativas assim produzidas têm as seguintes características: um indivíduo com aptidão acima da média terá expectativa maior do que 1, enquanto que um indivíduo com aptidão abaixo da média terá uma expectativa menor do que 1; a soma dos valores de todas as expectativas individuais será igual ao tamanho da população”.

Como pode ser observado, de acordo com o método de conversão acima descrito, o número de descentes de um indivíduo não pode ser diretamente considerado como sendo o seu valor de expectativa, uma vez que esse valor é um número real. Assim sendo, é necessário ainda converter o valor esperado em um número inteiro que finalmente represente efetivamente o número de descentes de cada indivíduo. Para tanto, também, diversas técnicas são propostas. Uma técnica muito simples e bastante utilizada em diversos métodos pode ser encontrada em MAZZUCCO (1999), “Supondo que o valor esperado de cada indivíduo de uma geração seja representado em um círculo, ocupando uma fatia proporcional a seu número (a área total do círculo é igual a soma de todos os valores esperados) e imaginando, ainda, que o círculo, assim obtido, represente um alvo sobre o qual serão arremessados dardos, um descendente será, então, alocado ao indivíduo cuja fatia for atingida por um dardo arremessado sobre o círculo alvo. Quanto maior o tamanho de uma fatia (quanto maior o valor esperado), maior a probabilidade do dardo atingir esta fatia e, portanto, do indivíduo correspondente procriar. O número de dardos arremessados será igual ao tamanho da população”.

Condições de Término: Para todo algoritmo de otimização, o ideal seria que o processo terminasse assim que o ponto ótimo fosse descoberto. Na prática, entretanto, não se pode afirmar com certeza que o ponto ótimo encontrado pelo algoritmo seja o ponto “ótimo global”. Como conseqüência, geralmente o critério para o término utilizado é um número máximo de gerações ou um tempo limite de processamento, ou o que ocorrer primeiro (KOZA & RICE, 1994) e (TANOMARU, 1995).

Outro critério que também é empregado é a idéia de “estagnação”, no qual o processo se encerra quando nenhuma melhoria for encontrada na população, após várias gerações consecutivas (TANOMARU, 1995) e (MICHALEWICZ, 1999).

3.3 Considerações Finais

Os Algoritmos Genéticos constituem ainda um instrumento de pesquisa, capaz de ser utilizado em diversas áreas das atividades humanas. Sua principal vantagem reside na diversidade genética: a cada passo do processo de busca, um conjunto de

candidatos é considerado e envolvido na criação de novos candidatos. Existem outras importantes vantagens, mas apesar disto, a utilização desta técnica heurística deve estar sempre restrita aos problemas cujas características sejam coerentes com seu campo de aplicação. Em termos de comparação com outros métodos heurísticos, pode-se frisar que apesar dos muitos estudos e experimentações feitos, estes ainda são insuficientes para estabelecer conclusões precisas e rigorosas (REEVES, 1995).

CAPÍTULO IV
COMPUTAÇÃO PARALELA E DISTRIBUÍDA

Neste capítulo serão analisados de forma abrangente os principais conceitos referentes à computação paralela e distribuída. Maior ênfase será dado à programação *multithreads* na linguagem de programação *Java*, uma vez que foi essa a tecnologia utilizada na implementação do modelo proposto neste trabalho.

4.1 Introdução

A computação paralela consiste, basicamente, na combinação de elementos de processamento, que se cooperam e se comunicam para solucionarem problemas complexos, de maneira mais rápida do que se estivessem sendo solucionados seqüencialmente (ALMASI & GOTTLIEB, 1994). Por outro lado, o paralelismo também proporciona o surgimento de uma série de novas características em relação ao gerenciamento e manutenção da coerência das informações processadas, características essas próprias da computação paralela e geralmente não encontradas em sistemas de computação seqüencial (SANTOS, 2001).

A programação paralela pode ser considerada como sendo a atividade de se escrever programas computacionais compostos por múltiplos processos cooperantes, atuando no desempenho de uma determinada tarefa (TANENBAUM & STEEN, 2002). De acordo com MAZZUCCO (1999), “Embora ainda conceitualmente intrigante, a computação paralela já se torna essencial no projeto de muitos sistemas computacionais, seja pela própria natureza inerentemente paralela apresentada por um sistema, seja na busca pela redução do tempo de processamento, ou mesmo na busca de uma melhor estruturação ou segurança”.

4.2 Arquiteturas Paralelas

Computação paralela pode ser realizada de diferentes formas em um sistema computadorizado. Segundo FLYNN (1972) os diferentes tipos de arquiteturas de computador, disponíveis são:

SISD (*Single Instruction, Single Data Stream*) – instruções simples, seqüências de dados simples. A categoria SISD, Figura 4.1, apresenta um único fluxo de instruções e de dados. Essa categoria compreende as máquinas de *Von Newmann* tradicionais mono processadas com execução seqüencial.

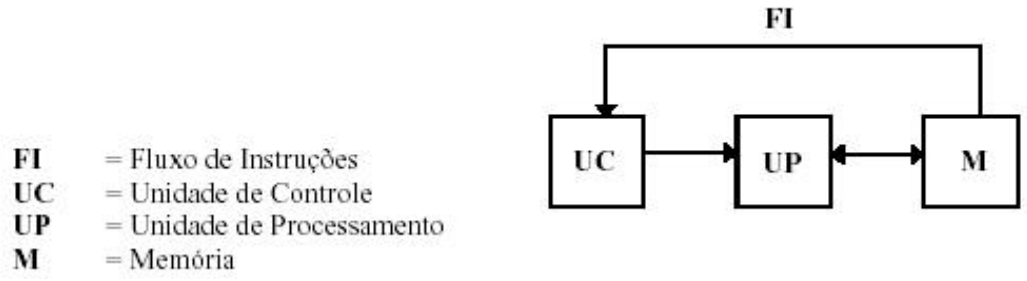

Figura 4.1 – Arquitetura SISD, DE ROSE & NAVAUX (2003).

SIMD (*Single Instruction, Multiple Data Stream*) – instruções simples, seqüências de dados múltiplas. Essa categoria apresenta um único fluxo de instruções atuando sobre múltiplos fluxos de dados, figura 4.2. Dessa forma, essa arquitetura possui várias unidades de processamento supervisionadas por uma única unidade de controle. A arquitetura SIMD engloba máquinas com vários processadores organizados na forma de vetor de processadores e máquinas matriciais.

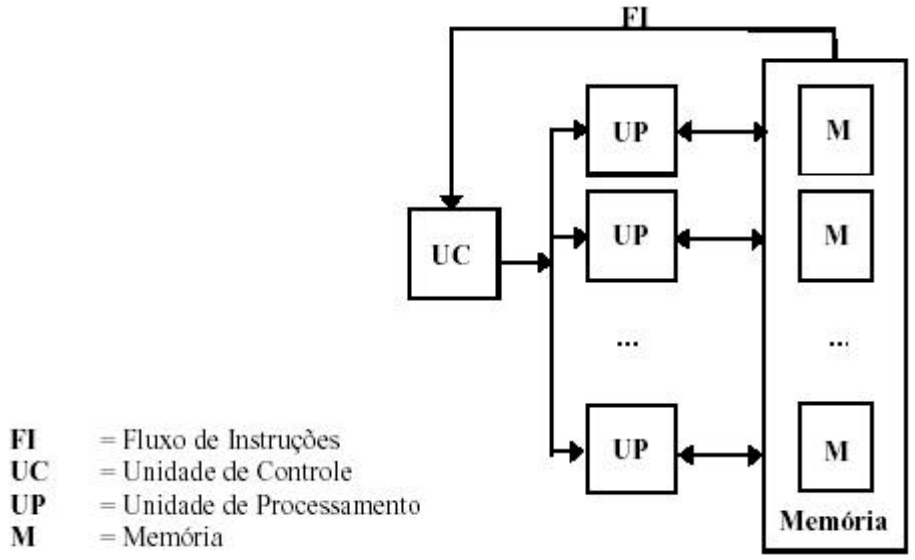

Figura 4.2 - Arquitetura SIMD, DE ROSE & NAVAUX (2003).

MISD (*Multiple Instruction, Single Data Stream*) – a categoria MISD, apresenta múltiplos fluxos de instruções atuando em um único fluxo de dados. Dessa forma, o fluxo de dados passaria por todas as unidades de processamento, sendo que o resultado de uma, seria a entrada para a próxima unidade, figura 4.3. Embora não existam muitos exemplos precisos de máquinas MISD na literatura, ALMASI & GOTTLIEB (1994) consideram o *pipeline* como representante dessa categoria.

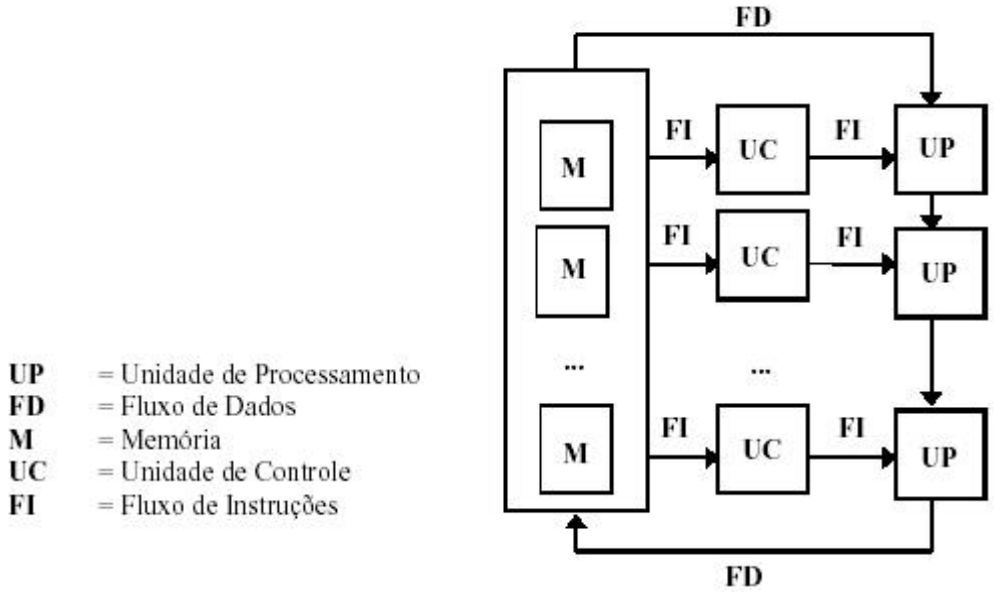

Figura 4.3 - Arquitetura MISD, DE ROSE & NAVAUX (2003).

MIMD (*Multiple Instruction, Multiple Data Stream*) - a categoria MIMD apresenta múltiplos fluxos de instruções atuando em múltiplos fluxos de dados, figura 4.4. Essa arquitetura envolve várias unidades de processamento executando diferentes conjuntos de dados, de maneira independente. Essa classe inclui os tradicionais sistemas multiprocessados, assim como redes de *workstations* e *clusters*.

Cada uma destas combinações caracteriza uma classe de arquitetura e corresponde a um tipo de paralelismo. Mais especificamente, as arquiteturas SIMD, por apresentarem fluxo único de instruções, oferecem facilidades para a programação e depuração de programas paralelos. Além disso, seus elementos de processamento são simples, pois são destinados à computação de baixa granulação. Por outro lado, arquiteturas MIMD apresentam grande flexibilidade para a execução de algoritmos paralelos, e bom desempenho em virtude de seus elementos de processamento serem assíncronos (DUNCAN, 1990).

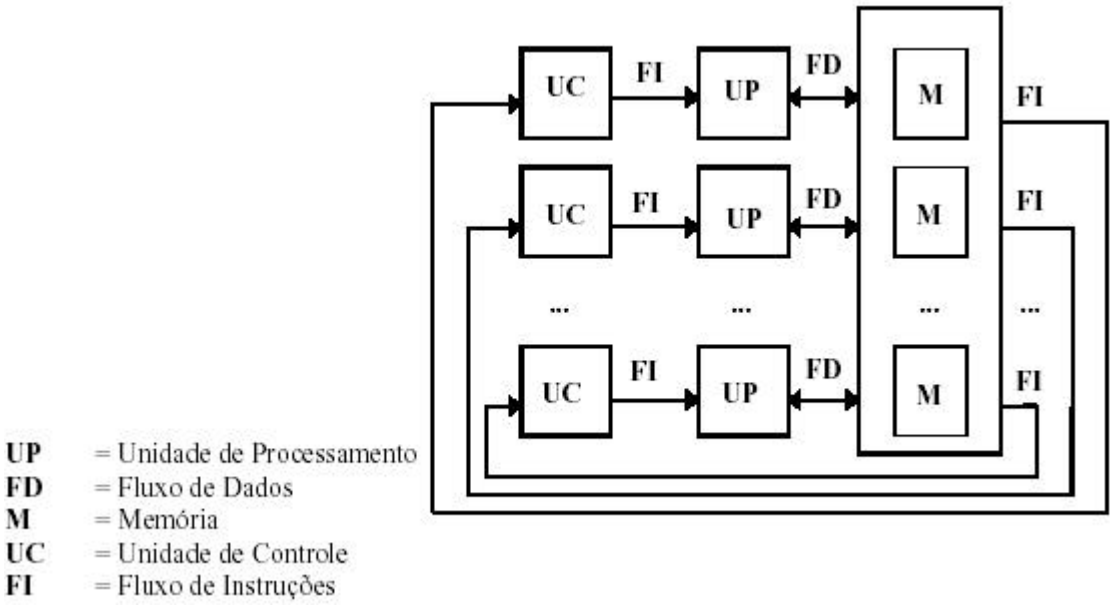

Figura 4.4 - Arquitetura MIMD, DE ROSE & NAVAUX (2003).

4.3 Programação Concorrente

Apesar de não haver um consenso absoluto estabelecido com relação a esse termo na literatura, será considerado nesse texto como sendo concorrente aquele sistema que possui memória compartilhada e que o número de processos em execução, não

necessariamente, coincida com o numero real de processadores. Também será considerado para efeito de simplicidade que processo e *thread* representam a mesma entidade computacional, embora haja diferenças entre os termos.

De uma forma geral, pode-se argumentar que a programação concorrente é regida pela idéia de se dividir determinadas aplicações em partes menores e que cada parcela resolva uma porção do problema. Consequentemente, com essa técnica de programação surge a necessidade de recursos adicionais de ativação, comunicação e sincronização de processos, não necessários na programação seqüencial. Assim sendo, a utilização da programação concorrente exige mecanismo de coordenação de processos para que possa existir entre eles perfeita sincronização e comunicação. Algumas linguagens de programação disponibilizam recursos para a execução de processos concorrentes.

A linguagem *Java,* a qual foi utilizada para o desenvolvimento do modelo proposto, disponibiliza os recursos necessários para a programação paralela permitindo a construção de programas com várias *threads* executados em máquinas mono ou multiprocessadas.

4.3.2 Sincronização de Processos

A sincronização de processos em um programa concorrente pode ser efetivada através da utilização de ferramentas que vão desde os primitivos semáforos até os monitores.

A linguagem *Java* utiliza o conceito de monitores para realizar sincronização entre *threads*. Na realidade, a linguagem *Java*, como uma tecnologia disponibiliza uma ferramenta que fica teoricamente muito distante do conceito de monitor proposto pelo seu autor Hoare em (HOARE, 1974). É considerado, no seu contexto, que todo o objeto com pelo menos um método *synchronized* transforma-se em um monitor. A exclusão mútua de um monitor em *Java* se dá através do qualificador *synchronized* aplicado aos seus métodos. Todos os métodos sincronizados de um monitor são excludentes, ou seja,

a partir do momento em que uma *thread* consegue acesso a um desses métodos ela passa a ter a posse (o *lock*) desse monitor de forma exclusiva. Somente os métodos não sincronizados desse objeto permanecem com o acesso livre (DEITEL & DEITEL, 2002). Diferentemente do monitor proposto por Hoare (HOARE, 1974), onde todos os métodos públicos de um monitor, por *default*, são excludentes.

Por sua vez, a sincronização nos monitores em *Java* se dá através dos métodos *wait, notify* e *notifyAll*. Uma *thread* que está em execução em um método *synchronized* pode determinar que não deve prosseguir e então executa voluntariamente o método *wait*. O efeito dessa execução consiste na liberação imediata do monitor envolvido, do bloqueio da execução da *thread* e da sua inserção em uma lista única, associada a aquele monitor, conhecida como lista de *wait*. Essa *thread* permanecerá bloqueada até que receba uma notificação de uma outra *thread* que, executando em um método *synchronized*, termina ou satisfaz a condição pela qual a *thread* anterior pode estar esperando. A notificação aludida pode ser feita através da execução do método *notify*, o qual atinge, de forma aleatória, uma única *thread* na lista de *wait*, ou através do método *notifyAll*, o qual atinge todas as *threads* bloqueadas na lista de *wait* daquele monitor. Quando uma *thread* foi notificada, a mesma passa do estado de "bloqueado" para o estado de "pronto para executar" e assim que for escalonada, disputará novamente o acesso ao monitor pelo mesmo método que a fez bloquear. Uma *thread* da lista de *wait* quando escolhida pela máquina virtual *Java* (*JVM-Java Virtual Machine*) para receber o *lock* do monitor, obviamente, executará o próximo comando após o método *wait* no método correspondente.

Diferentemente, no monitor proposto por HOARE (1974), quando uma *thread* notifica ou sinaliza uma segunda ela deixa imediatamente o monitor, bloqueando-se em uma lista especial chamada lista de sinalizadores. A *thread* sinalizada imediatamente recebe o *lock* do monitor e volta a executar o próximo comando após o método *wait*. A lista de sinalizadores, por sua vez, possui alta prioridade e sempre que o monitor se fizer vazio a máquina virtual (*JVM*) escolherá uma *thread* aleatoriamente dessa lista para lhe ceder o *lock* do monitor. Outra grande diferença na semântica do monitor proposto por HOARE (1974) com relação ao implementado pela linguagem *Java* é o tipo primitivo "condição". Uma variável do tipo condição, interna a um monitor, exerce uma função

muito útil uma vez que pode representar uma condição específica na lógica da programação. Na realidade, uma variável condição, operada somente através dos métodos *wait* e *signal*, possui o significado de uma lista. Quando um processo executa um método *wait* utilizando uma variável condição *C1*, *wait(C1)*, ele incondicionalmente e imediatamente perde o *lock* do monitor e é inserido na lista correspondente à variável *C1*. Um segundo processo ao executar o método *signal* sobre *C1*, *signal(C1*), estará perdendo o *lock* do monitor para um processo da lista *C1* e se inserindo na lista de sinalizadores vista anteriormente. Dessa forma, um monitor conforme proposto por HOARE (1974) pode possuir diversas listas: a lista do *lock* do monitor, utilizada uma única vez por cada processo, a lista de sinalizador para onde vão se posicionar os processos que executaram *signal* e diversas listas de *wait*, uma para cada variável do tipo condição declaradas no interior do um monitor. A linguagem *Java*, simplesmente aboliu o tipo condição, conferindo ao seu monitor, apenas duas listas: a lista do *lock* do monitor e uma única lista de *wait*. No monitor da linguagem *Java* não há mais condições específicas uma vez que qualquer execução do método *wait* conduz a *thread* correspondente a uma lista comum de *wait*. Também não há mais a lista de sinalizadores, já que a *thread* que sinaliza uma segunda, através dos métodos *notify*, segue o seu processamento normal dentro do monitor. O efeito da execução de um método *notify* é simplesmente retirar, de forma aleatória, uma das *threads* da lista de *wait* e inseri-la novamente na lista de *lock* do monitor. A semântica dos métodos *notify* e *notifyAll*, muito provavelmente, foi inspirada na linguagem de programação Mesa (LAMPSON & REDELL, 1980).

4.3.3 Considerações Finais

Existem muitas críticas com relação aos mecanismos de sincronização de *threads* oferecidos pela linguagem *Java*, alguns até bastante contundentes como o apresentado em (HANSEN, 1999) no seu artigo intitulado “Paralelismo inseguro de *Java*”. Entretanto, a experiência com a sua utilização neste trabalho foi bastante positiva, apresentou-se como uma linguagem de fácil utilização, muito clara, robusta e consistente, principalmente com relação à programação concorrente e paralela.

CAPÍTULO V
MODELO PROPOSTO

Neste capítulo é apresentado o modelo proposto e implementado nesta dissertação, o qual combina os Algoritmos *Simulated Annealing* e Genético, objetivando obter melhora no desempenho do primeiro. Será abordado inicialmente o ambiente computacional utilizado e então apresentada a descrição do algoritmo desenvolvido.

5.1 Configurações básicas dos ambientes utilizados

Para o desenvolvimento do modelo proposto, foram adotados os ambientes denominados de *"Memória compartilhada em ambiente paralelo real"* e *"Memória compartilhada em ambiente pseudoparalelo*. O primeiro ambiente é constituído de um computador executando uma quantidade de *threads* igual à quantidade de processadores, onde cada *thread* representa uma instância do algoritmo proposto. O segundo ambiente utilizado é constituído de um computador executando uma quantidade de *threads* superior a quantidade de processadores, com cada *thread* também representando uma instância do algoritmo proposto.

5.2 Detalhamento do modelo proposto

O algoritmo proposto tem como base o *Simulated Annealing* na sua forma original. Sua implementação baseou-se no modelo proposto em HANSEN (1995). Dessa forma, foi definido um conjunto de parâmetros específicos do problema a ser tratado utilizando os parâmetros propostos por (HANSEN, 1995). O SA adota esquemas nos quais a temperatura *T* é retirada de uma faixa definida pelos limites superior (valor inicial da temperatura) e inferior, próximo a zero. Dada uma temperatura, a mesma é mantida constante por um número pré-determinado de repetições e então multiplicada por uma constante, normalmente menor do que um, denominado, fator de redução. O processo se repete até que *T* atinja o limite inferior da faixa estipulado. Como já apresentado no Capítulo III, trata-se de um algoritmo interativo que, a cada passo gera uma nova solução que pode se tornar a solução corrente. Essa condição de troca de

soluções tem probabilidade igual a *1*, caso a solução gerada seja melhor que a corrente. Caso contrário, a probabilidade passa a valer 1 / (*e**DE/T*). Onde *DE* representa a diferença entre os valores das duas soluções consideradas.

A estrutura básica do método desenvolvido é um algoritmo *Simulated Annealing*, que foi apresentado na figura 3.2.

A hibridização dos algoritmos SA e AG não é um fato inédito, em MAZZUCCO (1999), por exemplo, pode ser encontrada a combinação do algoritmo *Simulated Anneling* com os operadores *crossover* dos algoritmos Genéticos, gerando uma nova técnica empregada na resolução de problemas de escalonamento de produção do tipo *job shop*. Entretanto, a maneira como essa combinação foi realizada neste trabalho pode-se afirma que apresenta um grau importante de ineditismo, com resultados consideráveis, como será visto no capítulo correspondente à análise dos resultados. Na realidade, como poderá ser notada, a nova proposta transforma o algoritmo SA, que originalmente trabalha com apenas uma solução corrente, em um novo algoritmo que passa a atuar sobre um banco de soluções correntes. Esse banco de soluções, atualizado a cada laço correspondente a uma dada temperatura, é formado pelas diversas *threads* que passam a compor o novo algoritmo. Cada *thread*, utilizando os mesmos parâmetros, implementa, independentemente das demais, uma instância do algoritmo SA e, a cada laço de temperatura, deposita nesse banco a sua solução corrente. Uma *thread* especial, chamada de *thread* Gerente, tem a incumbência de realizar o cruzamento dessas soluções no banco, através do emprego dos operadores *crossovers* dos Algoritmos Genéticos. Após a realização dos cruzamentos das soluções correntes, feitos aos pares, e de forma totalmente aleatória, as novas soluções voltam a ser distribuídas entre as *threads*. Cada *thread*, então, volta para realizar mais um laço de temperatura adotando a solução recebida como sua solução corrente. É importante observar que a questão de sincronização entre as *threads* é um fator primordial na implementação do modelo proposto. A *thread* gerente deve ser bloqueada em uma barreira até que todas as *threads* terminem os seus laços e depositem as suas soluções correntes no banco de soluções. Na medida em que as *threads* vão depositando suas soluções no banco, também vão sendo bloqueadas nessa mesma barreira. Quando a última *thread* terminar o seu deposito ela então liberará a *thread* Gerente que inicia o processo de cruzamento. Após o término do

processo de cruzamento, cada *thread* receberá uma nova solução e será desbloqueada, executando até o findar de um novo laço de temperatura.

O algoritmo proposto e implementado, referenciado a partir de agora nesse texto por SACC (*Simulated Anealing com Crossover*), é apresentado na sua forma geral através do seu pseudocódigo, figura 5.1.

```
O Algoritmo Simulated Annealing com Crossover (SACC)
Início
   obtenha a constante α e o número de repetições NR;
   S ← S0;
  T ← LS;                  // Limite Superior
  TMIN ← LI;               // Limite Inferior
   enquanto ( T > TMIN ) faça
     para I de 1 até NR    faça
        gerar uma solução S' de N(S);    // S' recebe uma solução gerada na vizinhança de S
       avaliar a variação de energia      // ΔE = f(S') – f(S);
          se ΔE <= 0 então
            S ← S'
          senão
            gerar rand ∈ random [ 0,1 ];
              se rand < exp (-ΔE / T) então
                    S ← S';
               fimse
          fimse
        Banco de Soluções ← S;
        Ponto de Bloqueio da Thread;
        S ← S'                    // S' extraída do banco de soluções após os cruzamentos
      fimpara                          realizados pela thread Gerente.
                T ← T * α;
       fimenquanto
```

Figura 5.1 - Pseudocódigo do algoritmo *Simulated Annealing* com *Crossover*.

5.3 Considerações finais sobre o modelo desenvolvido

Na realidade, o modelo proposto neste trabalho pode ser vislumbrado sob duas visões diferentes. A primeira como sendo um Algoritmo Genético modificado uma vez que o banco de soluções proposto nada mais seria do que uma população. Quanto maior o número de *threads* disparadas, maior será o tamanho dessa população. O trabalho que a *thread* Gerente realiza nos cruzamentos também não deixa de ser a transformação da geração corrente na sua sucessiva. Essa visão, no entanto, poderia ser considerada como uma inovação dos AG uma vez que as *threads* fariam uma espécie de refinamento nos indivíduos de cada população que participariam da formação da próxima geração,

através da aplicação do algoritmo SA, o que no AG original não ocorre. É importante observar que o operador de seleção e até mesmo de mutação são, de certa forma, realizado pelas *threads*. Por outro lado, sob a segunda visão, o algoritmo proposto poderia também ser encarado como um SA modificado. Além das transformações normais da solução corrente em uma outra escolhida na sua vizinhança, o algoritmo sofre uma grande perturbação quando recebe uma nova solução da *thread* Gerente para refinar durante a próxima iteração correspondente à nova temperatura. Apesar de ser uma atribuição aleatória, a probabilidade da solução recebida ser de boa qualidade é grande uma vez que foi gerada em um banco de soluções com elementos genéticos cada vez melhores para o problema que está sendo tratado.

Essa idéia de gerar perturbações nos algoritmos SA também não é inédita e tem como propósito realizar um possível deslocamento controlado da busca de uma região do espaço de soluções, para uma outra. Em MAZZUCCO (1999) foi proposto um novo esquema de controle da redução da temperatura cujo propósito foi o mesmo, ou seja, procurar evitar convergência para solução ótima local, através do abandono de uma certa região do espaço de soluções. Também, em LOURENÇO (1995) é proposta uma abordagem que utiliza uma outra estratégia, mas, cujo objetivo é novamente abandonar regiões do espaço de soluções intensivamente exploradas, através de perturbações realizadas no algoritmo *Simulated Annealing* original. A abordagem proposta no último trabalho aludido é conhecida por método de otimização de passo largo. Essa denominação é conseqüente de sua forma de execução onde a cada iteração, um passo largo é realizado, seguido da execução de um método de otimização de busca local. O algoritmo empregado no passo largo fica responsável pela modificação acentuada realizada na solução corrente. Cabe ao método de otimização de busca local, o refinamento da solução encontrada no passo largo.

Finalmente, a idéia de se utilizar *threads* na abordagem proposta, foi extremamente útil, principalmente com relação à concepção e implementação do modelo. A organização do programa tornou-se muito clara e seu desenvolvimento mais simples. Além disso, assim como em MAZIERO (2003) o interesse pela programação *multithreads*, influenciou substancialmente na concepção do modelo proposto. Sem a

utilização de *threads,* ficaria muito difícil imaginar a produção do banco de soluções a cada iteração, conforme descrito anteriormente. Dessa forma, embora ainda não muito utilizada, pode se afirma que a computação paralela vem se tornando cada vez mais essencial no projeto de muitos sistemas computacionais, seja pela própria natureza inerentemente paralela apresentada pelo sistema, seja na minimização do tempo de processamento, ou mesmo na busca de uma estruturação melhor e/ou mais segura.

No caso do problema tratado neste trabalho, pode-se considerar que os três objetivos foram almejados. O Algoritmo Genético é intrinsecamente paralelo, a máquina utilizada é multiprocessada e finalmente, como já foi dito, a concepção do método e a estruturação do algoritmo foram fortemente influenciadas pela programação paralela.

CAPÍTULO VI
ANÁLISE DOS TESTES REALIZADOS

A análise dos testes realizados tem como principal finalidade atestar, de forma prática, a potencialidade do modelo do algoritmo proposto para solucionar problemas de otimização combinatorial. A análise foi baseada em resultados obtidos nas resoluções de instâncias públicas do problema do Caixeiro Viajante, cujas soluções já são conhecidas, facilitando assim o emprego das métricas aplicadas na mesma.

As métricas aplicadas basearam-se na complexidade do problema, ou seja, na quantidade de pontos do problema (cidades a serem percorridas pelo Caixeiro Viajante) e na quantidade de *threads* aplicadas. O desempenho do método proposto foi comparado, em todos os testes realizados, com o do Algoritmo *Simulated Annealing* na sua foram original.

6.1 Métrica e Problemas utilizados

Nas análises dos resultados, optou-se por utilizar a métrica que apenas leva em consideração a qualidade da solução obtida a qual fornece em porcentagem a qualidade média do resultado encontrado em comparação com o melhor resultado já conhecido do problema em questão. A métrica que leva em consideração o número de acertos no resultado conhecido foi abandonada, pois, normalmente os problemas envolvidos ou são muito simples, assim a quantidade de acerto seria total ou são muito difíceis e raramente o resultado conhecido poderá ser alcançado.

Os problemas que foram tratados nesta dissertação são de domínio público. Fazem parte da biblioteca TSPLIB, que é mantida por Gerhard Reinelt - Universität Heidelberg (REINELT, 2005).

Foram escolhidos problemas de forma que todas as faixas de complexidades em TSPLIB fossem atingidas. Na Tabela 6.1 abaixo se encontram dispostos os problemas analisados.

Tabela 6.1 – Lista dos problemas a serem utilizados.

Ordem de Execução	Problema TSPLIB	Quantidade Pontos
1	brasil58	58
2	att532	532
3	d2103	2.103
4	pla7397	7.397
5	usa13509	13.509
6	pla33810	33.810

Como serão observados nos gráficos que seguem, cada problema foi resolvido pelos dois métodos: o SA puro e o SACC (*Simulated Annealing* com *crossover*) que é o método proposto e implementado neste trabalho. Para cada problema, foram efetuadas cinco rodadas de cada algoritmo, calculado a média aritmética dos resultados obtidos por cada um deles e então calculado o percentual dessa média em relação ao resultado conhecido.

Em cada gráfico, a seguir apresentado, cada ponto tem como coordenadas: no eixo x, o número de *threads* utilizadas no algoritmo SACC e no eixo y, o percentual obtido da comparação do resultado conhecido, com a média dos resultados obtidos com cinco rodadas do algoritmo proposto com aquele número de *threads*. No caso do algoritmo SA puro o número de *threads* corresponderia ao número de rodadas das quais é retirada a melhor solução encontrada.

Todos os testes foram realizados em uma máquina multiprocessada contando com seis processadores com memória compartilhada. Foi utilizada a tecnologia *Java*, como já foi dito, e a distribuição das *threads* disparadas pelo algoritmo proposto, foi realizada pela máquina virtual de forma transparente. Especificamente, o computador utilizado nos testes possui as seguintes características: é uma máquina da Sun Microsystem, modelo Sun Fire V880 Ultra Sparc III com seis processadores 1.2 GHz UltraSPARC® III RISC, com 8 MB de cache por processador, sistema operacional Solaris 9 com doze discos Sun StorEdge A1000.

Apenas para o problema *pla33810* o número de execuções limitou-se a duas, pois a quantidade de pontos que o problema apresenta exige um esforço computacional extremamente elevado e consequentemente uma disponibilidade de máquinas que não pode ser viabilizada.

Todas as simulações foram realizadas utilizando os três tipos de *crossover* apresentados anteriormente. Dessa forma, cada simulação conterá quatro gráficos, o do SA correspondente ao algoritmo original, os dos SACCOX, SACCCX e o SACCPMX, correspondentes ao algoritmo proposto utilizando, respectivamente os *crossovers* OX, CX e PMX.

Na figura 6.1, é apresentado o desempenho do modelo proposto perante o primeiro problema da Tabela 6.1. Trata-se do problema *brasil58* que apresenta pequeno grau de complexidade onde a solução conhecida foi alcançada diversas vezes durante os testes realizados.

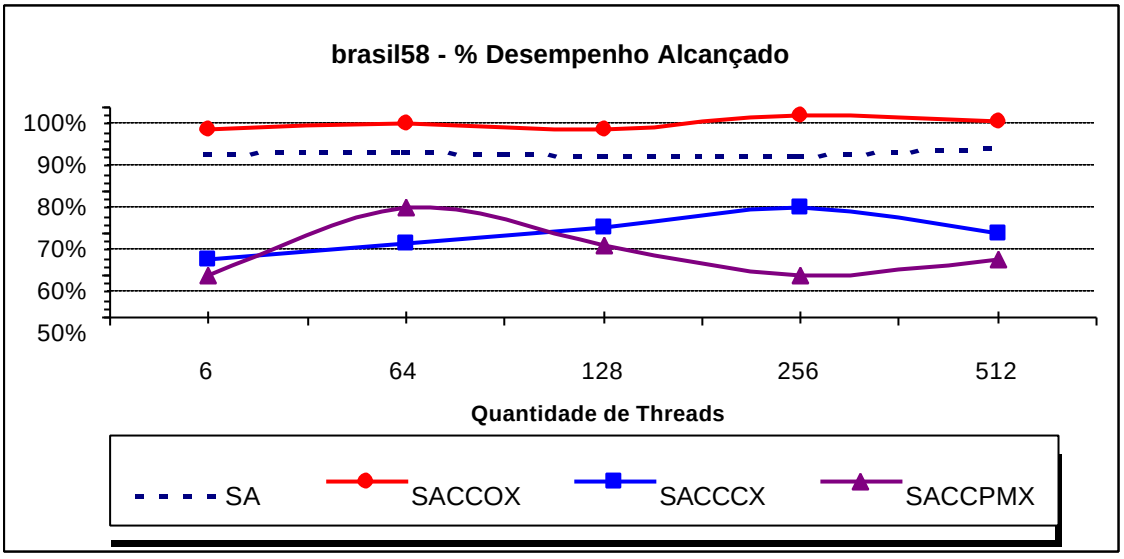

Figura 6.1 - Problema brasil58 – Desempenho dos Grupos por quantidade de *threads.*

Em todos os testes apresentados será adotada a mesma legenda: SA (*Simulated Annealing* puro) representado nos gráficos pela linha tracejada; SACCOX (método proposto com *crossover* OX) representado pela linha cheia em vermelho; SACCCX (método proposto com *crossover* CX) representado pela linha cheia em azul;

SACCPMX (método proposto com *crossover* PMX) representado pela linha cheia em bordo, conforme legenda presente nos gráficos.

Pela figura 6.1 acima fica evidenciado o desempenho do modelo proposto e implementado, principalmente quando utilizando o operador *crossover* denominado OX.

A figura 6.2 mostra um gráfico com escala mais precisa, a comparação dos desempenhos apenas do método proposto utilizando o *crossover* OX, com o do método SA original.

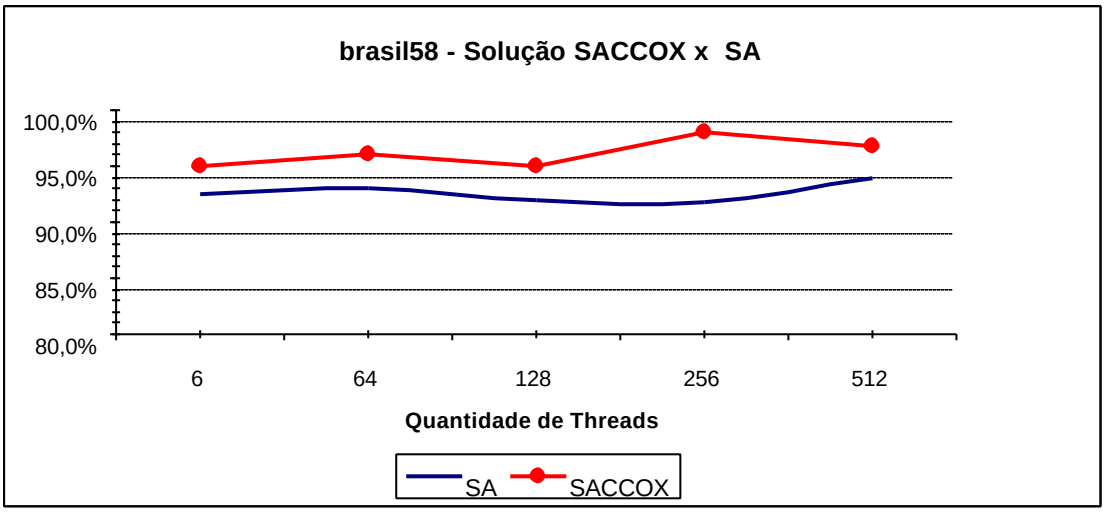

Figura 6.2 - Problema brasil58 – Comparação apenas dos métodos *Simulated Annealing* puro com SACCOX.

Seguindo os testes propostos na Tabela 6.1, o problema a ser abordado a seguir será o *att532*.

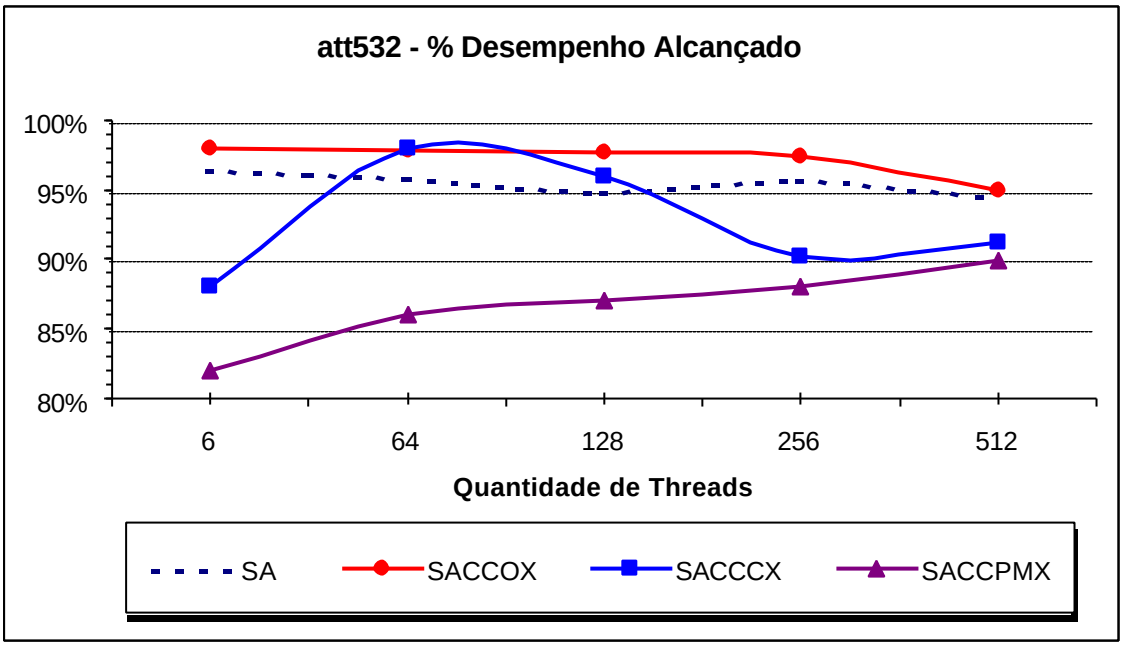

Figura 6.3 - Problema att532 – Desempenho dos Grupos por quantidade de *threads.*

Para o problema *att532* verifica-se novamente, figura 6.3, que o algoritmo que obteve desempenho superior aos demais foi o grupo SACCOX desde a utilização do paralelismo real (6 *threads*) até (512 *threads*), pseudoparalelismo. Outra característica importante que diferencia esse algoritmo dos demais é a sua homogeneidade, a variação do seu desempenho com relação ao aumento do número de *threads* utilizadas permanece praticamente constante, chegando a diminuir a sua qualidade quando o número de *threads* ultrapassa 256.

A seguir, na figura 6.4, é apresentado um resultado importante. Os gráficos apresentados representam as simulações dos algoritmos SA e SACCOX agora com os números de *threads* disparadas, variando entre 500 a 1000. Foi surpreendente o aumento do desempenho do SACCOX com relação ao acréscimo de *threads* utilizadas nas simulações. Esse fato pode, de certa forma, mostrar que dependendo do problema a ser tratado, o número de *threads* utilizadas pode ter uma influência decisiva. É importante salientar aqui que o aumento do número de *threads* corresponde ao aumento do tamanho da população do algoritmo genético que, como já foi visto, estaria indiretamente sendo utilizado no algoritmo proposto. Em outras palavras, o aumento do número de *threads* corresponde ao acréscimo do tamanho do banco de soluções que serão cruzadas para constituírem o próximo laço do algoritmo proposto.

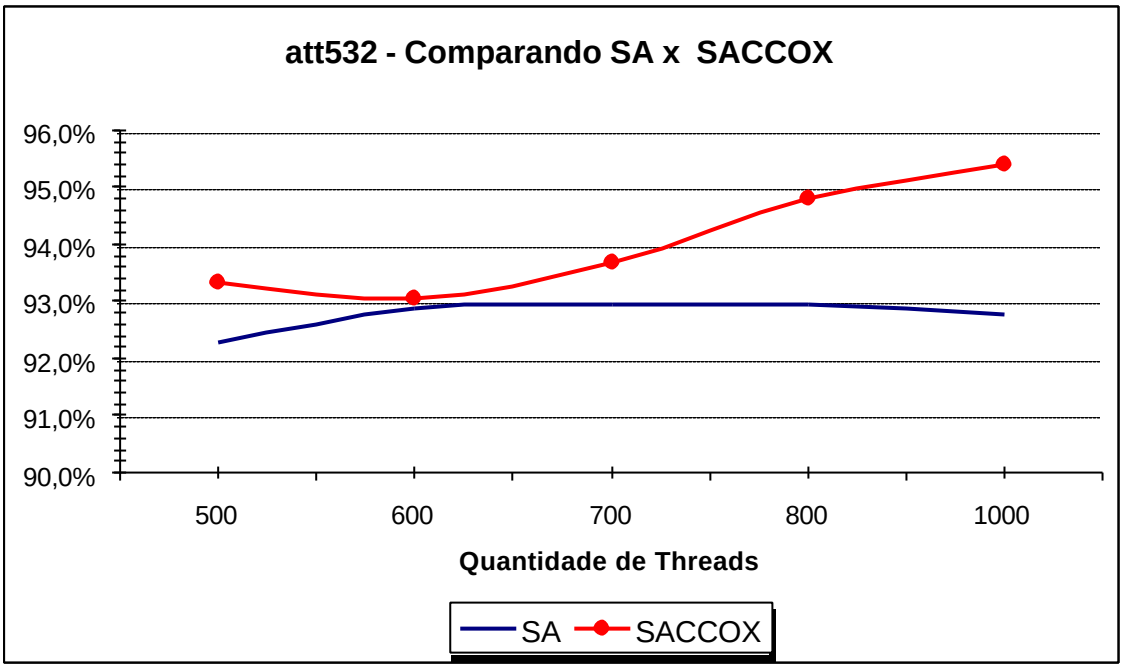

Figura 6.4 - Problema att532 – Comparação da SA com SACCOX focada no aumento do número de *threads* empregadas.

Seguindo a ordem da Tabela 6.1, o problema a ser tratado a seguir será o *pla7397*, que apresenta uma complexidade aproximadamente treze vezes superior ao estudado anteriormente, representado pela Figura 6.5.

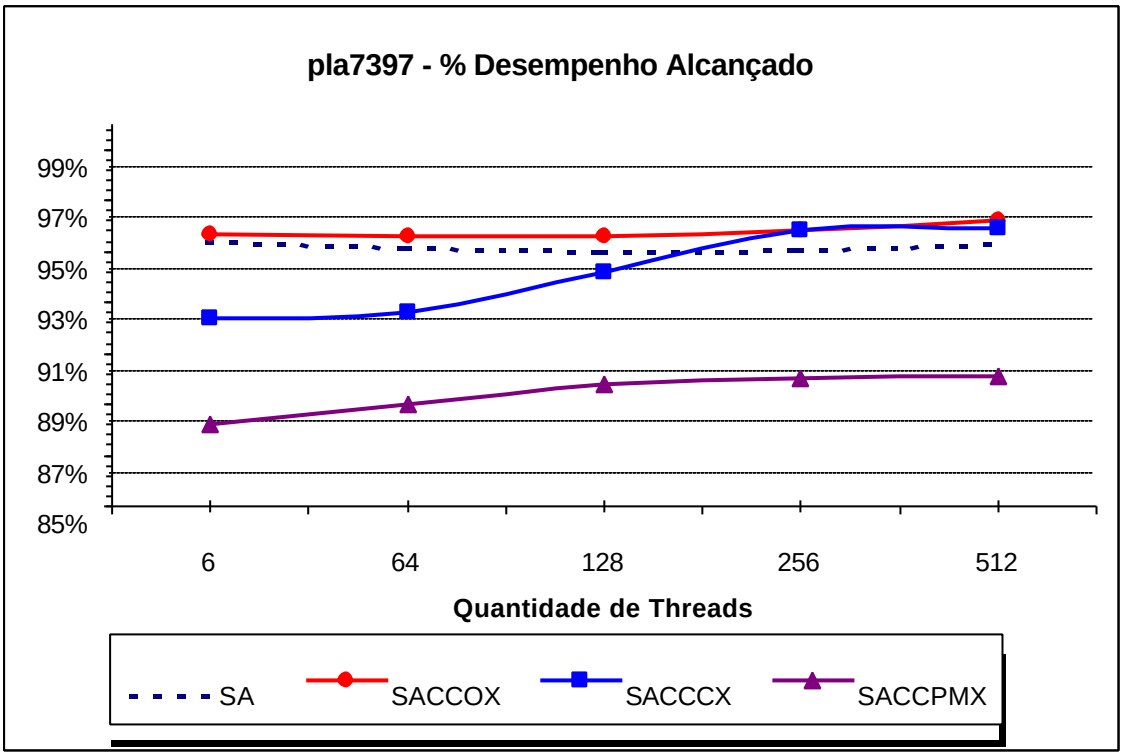

Figura 6.5 - Problema pla7397 - Desempenho dos Grupos por quantidade de *threads.*

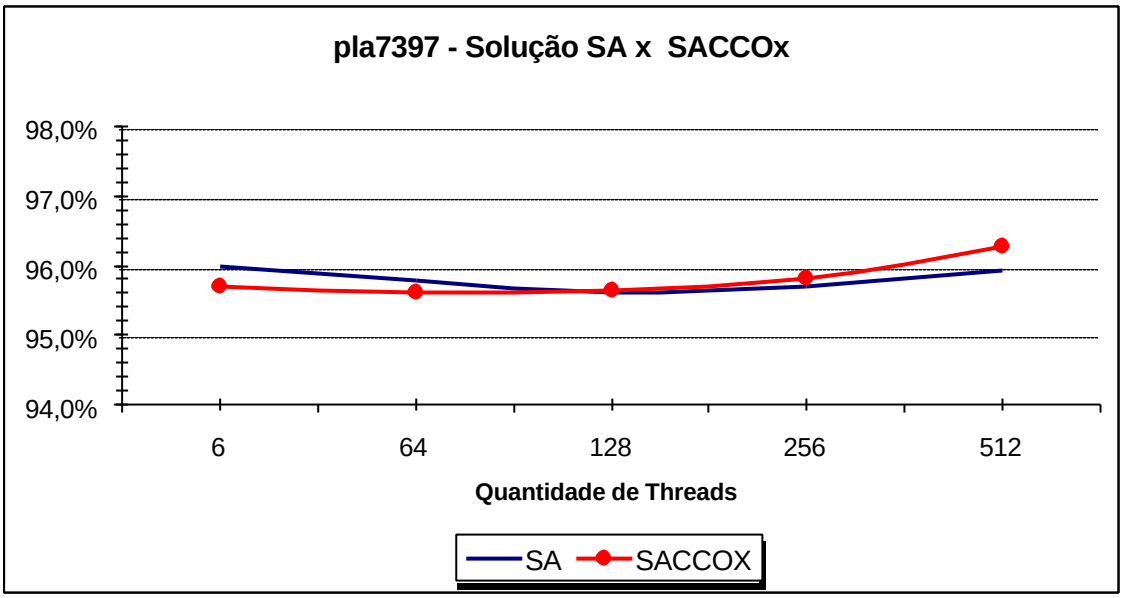

Figura 6.6 - Problema pla7397 – Comparação apenas do *Simulated Annealing* puro com SACCOX.

O problema *pla7397* apresenta uma complexidade considerável, o número de cidades que compõe a rota do caixeiro viajante passa a ser de 7.397 pontos. Nas Figuras 6.5 e 6.6 pode ser observado que o desempenho, principalmente do SACCOX, começa a se destacar a partir de 128 *threads*, mais uma vez confirmando a importâncias do aumento do número de *threads* empregado, ou seja, do tamanho do banco de soluções existente. Objetivando confirmar essa afirmação, foi realizada uma simulação envolvendo apenas o algoritmo original SA e o SACCOX, focando no aumento do número de *threads* empregadas. O resultado pode ser observado na figura seguinte, figura 6.7, onde se verifica que o desempenho do SACCOX se distancia do SA a medida em que o número de *threads* empregadas aumenta.

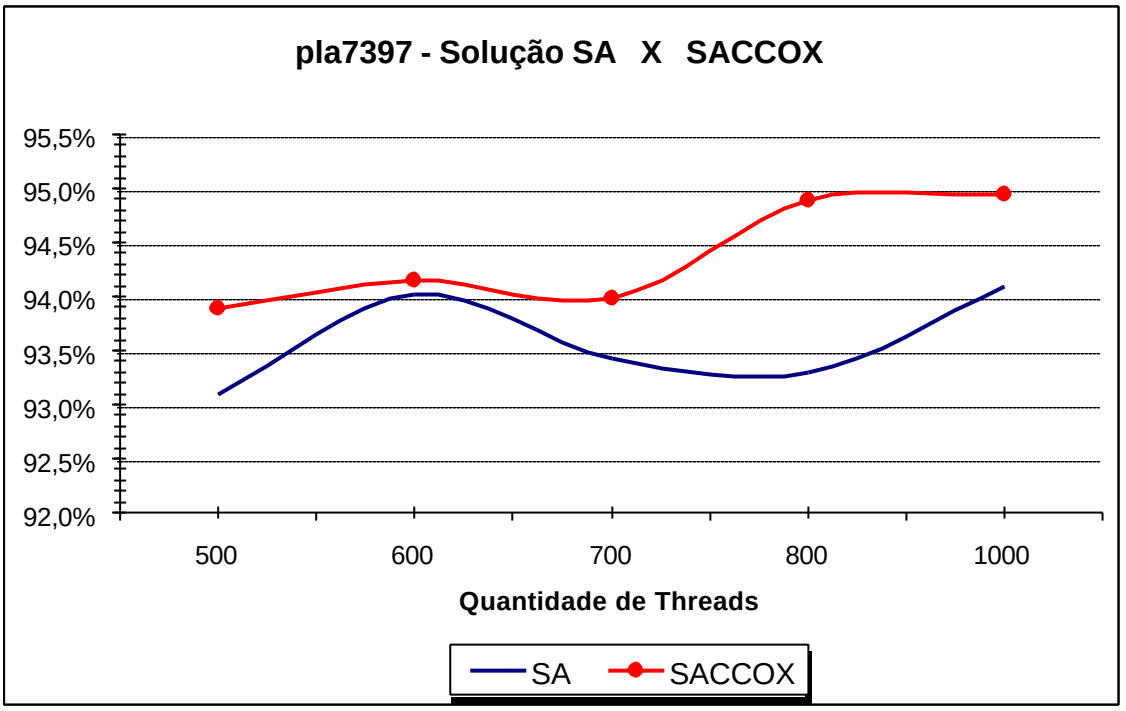

Figura 6.7 - Problema pla7397 – Desempenhos dos algoritmos SA e SACCOX focado no número de *threads* empregadas.

Para o problema *usa13509*, penúltimo problema da tabela apresentada na Tabela 6.1, agora com 13.509 pontos, a situação se alterou consideravelmente. Conforme pode ser observado na figura 6.8, o algoritmo SA, *Simulated Annealing* puro, alcançou o melhor desempenho com o número de *threads* inferior a aproximadamente 160. Deve ser considerado que esse problema apresenta um grau alto de complexidade, muito embora o SA tenha conseguido melhor desempenho com até 160 *threads*, o valor médio das soluções por ele alcançado atinge aproximadamente 97% do valor da melhor solução conhecida para esse problema. A partir da utilização de mais 160 *threads*, o algoritmo proposto com *crossover* OX começa a se destacar, novamente, confirmando a importância do tamanho do banco de soluções empregado.

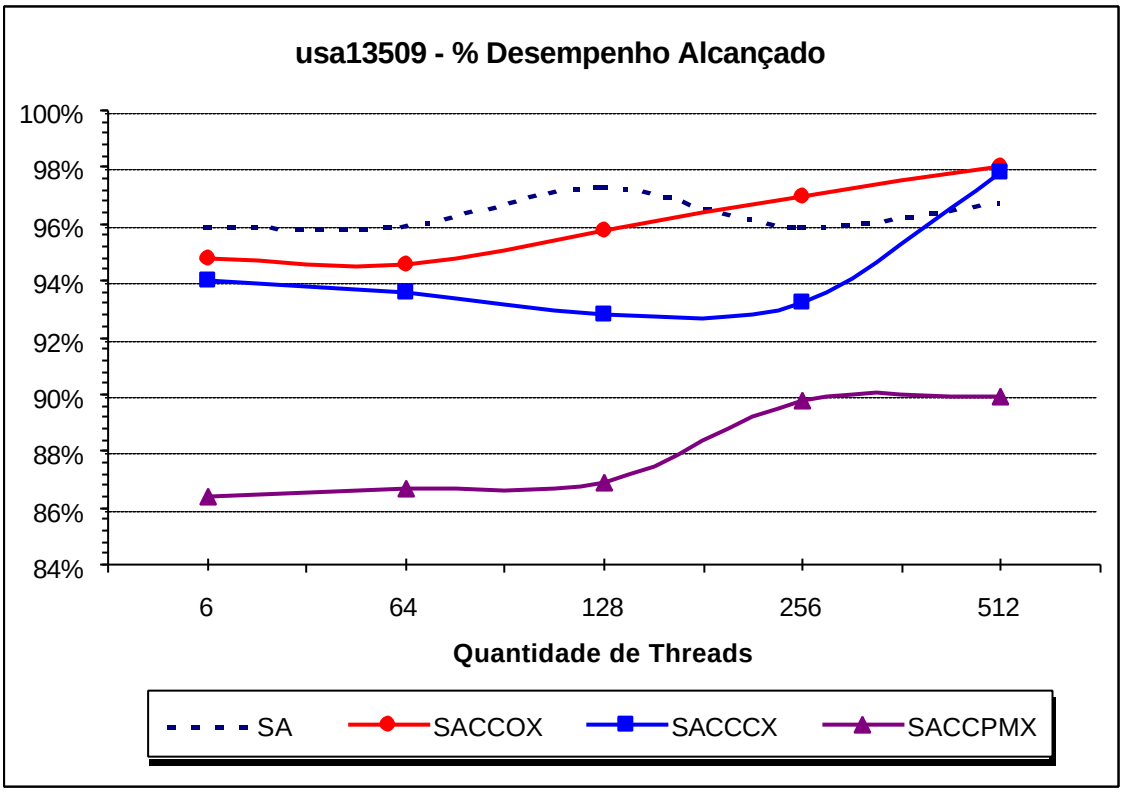

Figura 6.8 - Problema usa13509 – Desempenho dos Grupos por quantidade de *threads*.

Como nos problemas anteriores, na figura 6.9, é feita uma comparação apenas dos desempenhos dos algoritmos SA e do SACCOX, muito embora, nesse caso particular, deva ser destacado o desempenho alcançado também pelo algoritmo proposto com a utilização do *crossover* CX, gráfico SACCCX , Figura 6.8.

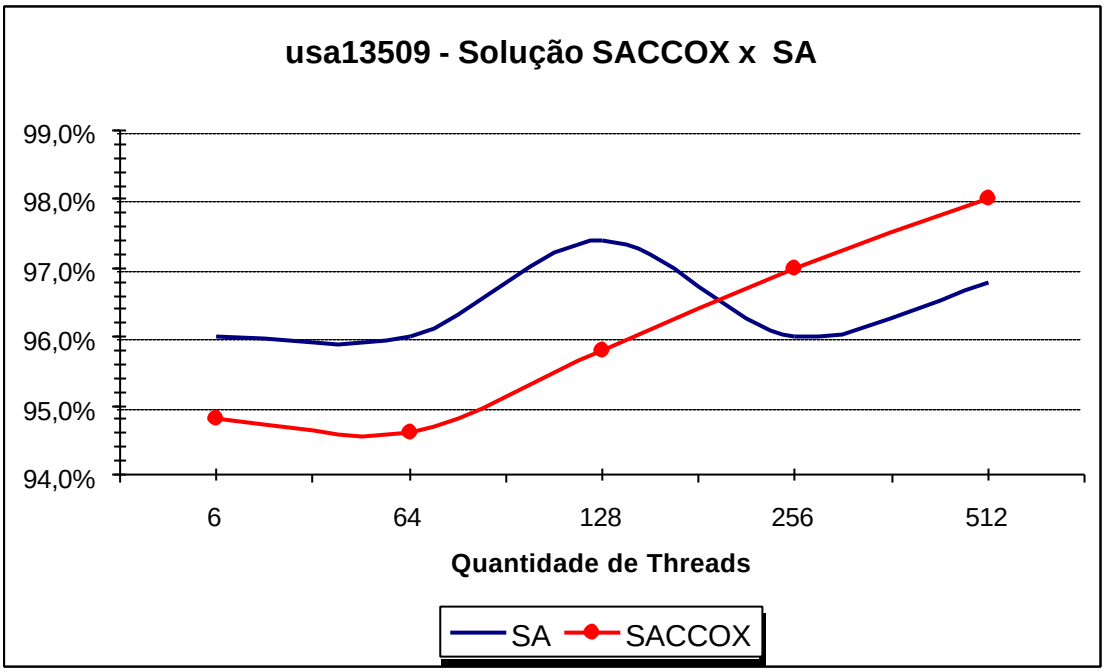

Figura 6.9 - Problema usa13509 – Comparação dos desempenhos de SA e SACCOX.

Finalmente, o ultimo teste proposto na Tabela 6.1 é alcançado. Trata se do problema *pla33810*, com trinta e três mil, oitocentos e dez pontos na rota do caixeiro viajante. Conforme já mencionado, nesse caso particular o número de rodadas para cada algoritmo, em cada conjunto de *threads* utilizadas ficou em apenas duas, uma vez que, a quantidade de pontos que o problema apresenta exige um esforço computacional extremamente elevado e consequentemente uma disponibilidade de máquinas que não pode ser viabilizada.

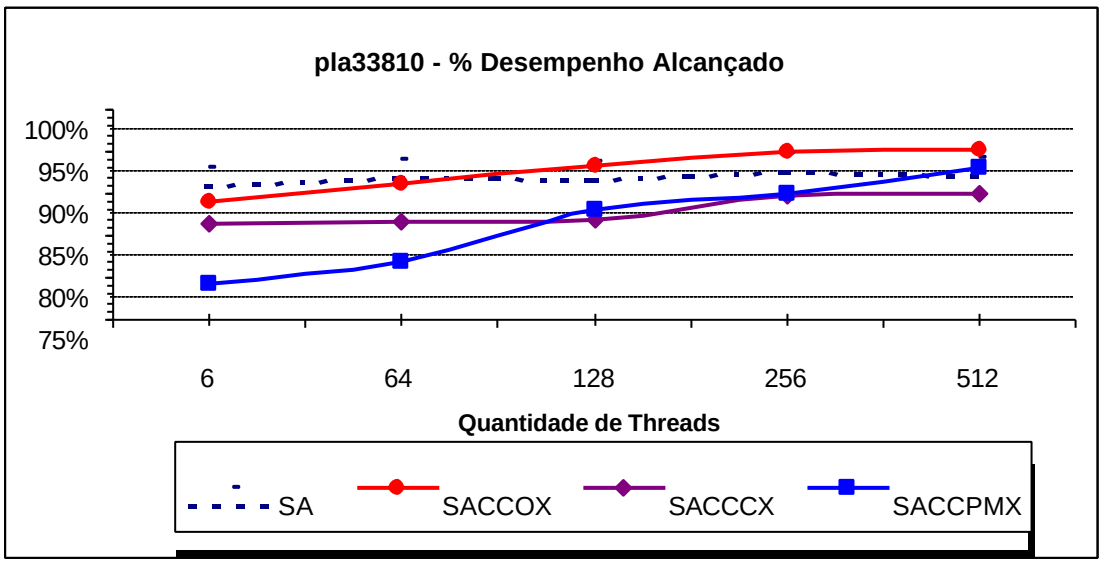

Figura 6.10 - Problema pla33810 – Desempenho dos Grupos por quantidade de *threads*.

Para o problema *pla33810*, figura 6.10, os desempenhos dos algoritmos tiveram comportamentos semelhantes ao problema anterior. O SA segue com um desempenho superior a todos, muito embora atingindo apenas 95% da solução ótima conhecida para esse difícil problema. Na medida em que o número de *threads* cresce, o desempenho, principalmente do SACCOX, aumenta ultrapassando o SA ao atingir 256 *threads*.

CAPÍTULO VII
CONSIDERAÇÕES FINAIS

Serão abordadas neste capítulo as considerações e conclusões do trabalho desenvolvido nessa dissertação, incluindo comentários sobre os resultados alcançados e propostas para pesquisas futuras no assunto abordado.

7.1 Conclusões

O modelo proposto e implementado neste trabalho teve como base a utilização de duas técnicas conhecidas para a solução de problemas de otimização combinatorial, que são os Algoritmos Genéticos (AG) e o algoritmo *Simulated Annealing* (SA). A proposta, basicamente, tinha como objetivo fundamental aumentar a potencialidade do algoritmo SA, através de uma abordagem do mesmo em paralelo, juntamente com o operador *crossover* dos AG, utilizando uma arquitetura multiprocessada na realização dos testes. Para avaliação do modelo proposto foram utilizadas instâncias públicas do problema do Caixeiro Viajante disponíveis na *internet* cujas soluções são conhecidas.

Os resultados obtidos nas simulações realizadas, embora ainda não conclusivos, projetam uma potencialidade do algoritmo híbrido proposto superior ao do algoritmo *Simulated Annealing* original, em certos problemas e situações determinadas. Obviamente, embora não tenha sido levado em consideração, o emprego de *threads*, fundamental na implementação do algoritmo proposto, deve aumentar consideravelmente o tempo de processamento. Além do tempo adicional gasto pelo próprio sistema operacional no escalonamento dessas *threads*, a existência da *thread* Gerente, conforme aludida no trabalho, também consome tempo no cruzamento das soluções. Entretanto, considerando a evolução atual da computação, não há dificuldade em se imaginar uma máquina ou um *cluster* contando com mil ou mais processadores para realizar tal tarefa.

Um outro aspecto muito importante que indiretamente pode ser constado com esse trabalho foi a da importância da utilização da programação paralela na concepção de um sistema computacional. Como já afirmado anteriormente, embora ainda não

muito divulgada e utilizada, a computação paralela vem se tornando cada vez mais uma ferramenta essencial no projeto e no desenvolvimento de muitos sistemas computacionais, seja pela própria natureza inerentemente paralela do sistema a ser implementado, seja na minimização do tempo de processamento, ou mesmo na busca de uma estruturação melhor e ou mais segura. No caso do problema tratado nesta dissertação, pode-se considerar que os três objetivos foram almejados. O Algoritmo Genético é intrinsecamente paralelo, a máquina utilizada é multiprocessada e finalmente, a concepção do método e a estruturação do algoritmo, foram fortemente influenciados pela programação paralela.

7.2 Sugestões para trabalhos futuros

Atualmente pode ser observada uma grande tendência em se utilizar métodos aproximados na resolução de problemas envolvendo otimização combinatorial, dessa forma, essa área se transformou em um campo muito fértil de pesquisa e desenvolvimento. Diretamente relacionado com o método proposto poderia se indicar como alternativas de estudos, novas operações de *crossover* em outros testes e até mesmo em outros tipos de problemas, outras maneiras de seleção de soluções para realização dos cruzamentos etc.

Entretanto, talvez a proposta mais interessante na continuidade desse trabalho seja a implementação da proposta em um ambiente fisicamente distribuído, permitindo que sejam realizados testes com problemas maiores em tempos de processamentos viáveis.

REFERÊNCIAS BIBLIOGRÁFICAS

ABNT. ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS. Referências Bibliográficas, NBR 6023, NBR 6027, NBR 6028, NBR 6034, NBR 10520, Rio de Janeiro, 2005.

ALMASI, G. S.; GOTTLIEB A., Highly Parallel Computing, The Benjamin Cummings Publishing Company Inc., ed. 2, 1994.

ARAUJO, H.A. Algoritmo Simulated Annealing: Uma nova abordagem, Dissertação de Mestrado, Florianópolis-SC: PPGEP/UFSC, 2001.

BLUM, C.; ROLI, A. Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison, Technical Report TR/IRIDIA/2001-13, IRIDIA, Université Libre de Bruxelles, Belgium, 2001.

CORRÊA, E. S. Algoritmos Genéticos e Busca Tabu Aplicados ao Problema das P-medianas, Dissertação de Mestrado, Coritiba-PR: UFPR, 2000.

DALLE MOLE, V.L. Algoritmos Genéticos – Uma Abordagem Paralela Baseada em Populações Cooperantes, Dissertação de Mestrado, Florianópolis-SC: PPGCC/UFSC, 2002.

DEITEL, H.M.; DEITEL P.J. Java How to Program, Prentice Hall, ed. 4, 2002.

DE ROSE, C. A. F.; NAVAUX, P. O. A. Arquiteturas Paralelas, Porto Alegre: Sagra Luzzatto, 2003.

DRECHSLER, R.; BECKER, B.; GÖCKERL, N.; et al. A genetic algorithm for variable ordering of obdds, ACM/IEEE International Workshop on Logic Synthesis, may 1995.

DUNCAN, R. A Survey of Parallel Computer Architectures, IEEE Computer, p. 5-16, 1990.

FLYNN, M. J. Some Computer Organizations and Their Effectiveness, IEEE Transactions on Computers, v. C-21, p. 948-60, 1972.

GOLDBERG, M.C.; LUNA, H.P.L. Otimização Combinatória e Programação Linear: Modelos e Algoritmos, Editora Campos, Rio de Janeiro, 2000.

GOLDBERG, D.E. Genetic Algorithms in Search, Optimization, and Machine Learning, Massachusetts, Addilson Wesley Publisher Company, 1989.

HAJEK, B. Cooling Schedules for Optimal Annealing, Mathematics of Operations Research, v. 13, p. 311-329, 1988.

HANSEN, P.B. Studies in Computational Science - Parallel Programming Paradigms, Prentice Hall, 1995.

HANSEN, P. B. Java's Insecure Parallelism, ACM Sigplan Notices, v. 34, n.4, p. 38-45, april 1999.

HIROYASU, T.; MIKI, M.; OGURA, M.. et al. Parallel Simulated Annealing using Genetic Crossover, International Conference Parallel and Distributed Computing and System, Las Vegas, Nevada, 2000.

HOARE, C. A. R. Monitor: An Operating System Structuring Concept, Comm. ACM, 17(10), p. 549-557, 1974.

HOLLAND, J. H. When will a genetic algorithm outperform a hill climbing, Proceedings of the Fifth International Conference on Genetic Algorithms, Forrest, S., Morgan Kaufmann, 647, 1993.

KIRKPATRICK, S.; GELLAT, D. C.; VECCHI, M. P.; et al. Optimization by Simulated Annealing, Science, p. 671-680, 1983.

KOZA, J.R; RICE, J.P. Genetic Programming II: Automatic Discovery of Reusable Programs, Cambridge: MIT Press, 1994.

LAMPSON, B.; REDELL, D. Experience with Processes and Monitors in Mesa, Commun, ACM 23(2), p. 105-117, 1980.

LEE, F.A. Parallel Simulated Annealing on a Message-Passing Multi-Computer, Dissertation to Doctor of Philosophy in Electrical Engineering, Utah State University, Logan, 1995.

LEWIS, H.R.; PAPADIMITRIOU, C.H. Elementos de Teoria da Computação, Trad. Edson Furmankiewicz, Porto Alegre: Bookman, ed. 2, 2000.

LOURENÇO, H. R. Job Shop Scheduling: Computational Study of Local Search and Large-Step Optimization Methods, European Journal of Operational Research, v. 83, p. 347-364, 1995.

MAZZIERO, A. E. Simulated Annealing em Paralelo + Genético Crossover, Dissertação de Mestrado, Florianópolis-SC: PPGCC/UFSC, 2003.

MAZZUCCO, J. Uma Abordagem Híbrida do Problema da Programação da Produção através dos Algoritmos Simulated Annealing e Genético, Tese de Doutorado, Florianópolis-SC: PPGEP/UFSC, 1999.

METROPOLIS, W.; ROSENBLUTH, A.; ROSENBLUTH, M.; et al. Equation of State Calculations by Fast Computing Machines, Journal of Chemical Physics, vol. 21, 1953.

MICHALEWICZ, Z. Genetic Algorithms + Data Structures = Evolution Programs, Department of Computer Science, University of North Carolina, Charlotte, USA, 1999.

PAPADIMITRIOU, C. H. Combinatorial Optimization: Algorithms and Complexity, Prentice-Hall, Massachusets, 1982.

REEVES, C.R. Modern heuristic techniques for combinatorial problems, Advanced topics in computer science, McGraw-Hill Book Co Europe, Berkshire, 1995.

REINELT, G. http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/, Universität Heidelberg, Institut für Informatik, 2005.

RODRIGUES, M.A.P. Problema do Caixeiro Viajante – Um Algoritmo para Resolução de Problemas de Grande Porte baseado em Busca Local Dirigida, Dissertação de Mestrado, Florianópolis-SC: PPGEP/UFSC, 2000.

ROUTO, T. Desenvolvimento de Algoritmos e Estrutura de Dados, McGraw-Hill, Brasil, 1991.

SANTOS, R. R. Escalonamento de Aplicações Paralelas: Interface AMIGOCORBA, Dissertação de Mestrado, Instituto de Ciências Matemática e de Computação, Universidade de São Paulo, 2001.

TANENBAUM, A. S.; STEEN, M.V. Distributed Operating Systems, Prentice Hall, ed. 1, 2002.

TANOMARU, J. Motivação, Fundamentos e Aplicações de Algoritmos Genéticos, II Congresso Brasileiro de Redes Neurais, p. 373-403, Curitiba-PR, 1995.

ZUBEN, F.J.V. Computação Evolutiva: Uma Abordagem Pragmática, DCA/FEEC, UNICAMP, Campinas-SP, 2000.