View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Repositório Institucional da UFSC

UNIVERSIDADE FEDERAL DE SANTA CATARINA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

VILSON LUIZ DALLE MOLE

ALGORITMOS GENÉTICOS – UMA ABORDAGEM PARALELA BASEADA EM POPULAÇÕES COOPERANTES

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

Prof. José Mazzucco Junior, Dr.

Orientador

Florianópolis, Dezembro de 2002

ALGORITMOS GENÉTICOS – UMA ABORDAGEM PARALELA BASEADA EM POPULAÇÕES COOPERANTES

VILSON LUIZ DALLE MOLE

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 Curso de Pós-Graduação em Ciência da Computação – CPGCC

____

Prof. Fernando A. Ostuni Gauthier, Dr. – Coordenador.

Banca Examinadora:

____

Prof. José Mazzucco Junior, Dr. – Orientador.

____

Prof. Thadeu Botteri Corso, Dr.

____

Prof. Luiz Fernando Friedrich, Dr.

Dedico este trabalho a minha amada esposa e a meus queridos filhos,

que sempre estiveram ao meu lado oferecendo seu apoio,

e que souberam suportar minha ausência.

Agradecimentos

Agradeço primeiramente a Deus que me permitiu realizar este trabalho, estou certo de que sem sua benção não teria sido capaz de fazê-lo.

Em segundo lugar a minha querida esposa, ela que sofreu com minha ausência mas que mesmo assim sempre esteve ao meu lado oferecendo seu apoio e seu amor.

Agradeço ao meu orientador por ter sido mais que orientador, sua postura amiga possibilitou a interação necessária para desvelar os caminhos a serem trilhados.

Agradeço ao CEFET/PR por ter disponibilizado os equipamentos utilizados no desenvolvimento do protótipo e na realização dos testes.

Agradeço aos que foram meus alunos em 2001 e 2002, por terem sabido compreender as minhas dificuldades em conciliar estudo e trabalho. Eles que muitas vezes se dispuseram a repor aulas nos feriados e contra-turnos.

Agradeço aos meus colegas de trabalho e em especial ao meu amigo Betzek, pelo apoio recebido.

Resumo

Este trabalho tem como objetivo principal o desenvolvimento de uma abordagem paralela para algoritmos genéticos. O foco principal está na estrutura de paralelização, ou seja, o modelo de algoritmo genético paralelo.

A arquitetura modelada está baseada nos conceitos de programação orientada a objetos e explora o paralelismo de sistemas multiprocessados ao mesmo tempo em que permite a distribuição através de uma rede de workstations.

O algoritmo genético paralelo está modelado segundo o conceito de populações cooperantes, coordenadas por um nó central. Nesse modelo, cada máquina é responsável por processar uma população com N indivíduos onde a troca de informações, entre populações, se realiza através da migração de indivíduos. Os indivíduos foram modelados como unidades autônomas com a responsabilidade da aplicação dos operadores de *crossover* e mutação. Cada indivíduo é também responsável pela determinação da sua aptidão. Este enfoque torna simples a exploração do paralelismo encontrado nos sistemas multiprocessados.

A hipótese defendida é a de que um conjunto de populações cooperantes, executando em paralelo, equivale a uma grande população. Para avaliar o desempenho desta nova abordagem, foram realizados testes utilizando um protótipo construído para simular o ambiente distribuído e o paralelismo dos sistemas multiprocessados, através de programação concorrente. Como corpo de provas foram utilizadas instâncias de um dos mais conhecidos *benchmarks* na área de otimização, o problema do caixeiro viajante. Os resultados obtidos estão descritos neste trabalho.

Abstract

This work has as main objective the development of a parallel approach for genetic algorithms. The main focus is in the parallel structure, in other words, the model of parallel genetic algorithm.

The modeled architecture is based on the concepts of object oriented programming and it explores the parallelism of systems endowed with multiple processors at the same time in that it allows the distribution through a workstations net.

The parallel genetic algorithm is modeled according to the concept of cooperative populations coordinated by a central node. In that model, each machine is responsible for processing one population with N individuals where the change of information among the populations takes place through the individuals' migration. The individuals were modeled about autonomous units with the responsibility of the application of the crossover and mutation operators. Each individual is also responsible for the determination of his fitness. This focus turns simple the exploration of the parallel systems.

The protected hypothesis is that a group of cooperative populations executing in parallel, it is equal to a great population. To evaluate the acting of this new approach, tests were accomplished using a prototype built to simulate the distributed atmosphere and the parallelism of the parallel systems, through concurrent programming. As proof body was used one of the more acquaintances benchmarks in the optimization area, the traveling salesman problem. The obtained results are described in this work.

Sumário

1 INTRODUÇÃO 12

1.1 MOTIVAÇÃO 13
1.2 OBJETIVOS DO TRABALHO 13
1.3 ORGANIZAÇÃO DO TEXTO 14
1.4 LIMITAÇÕES DA PESQUISA 15

2 COMPLEXIDADE COMPUTACIONAL 16

2.1 INTRODUÇÃO 16
2.2 ANALISANDO A COMPLEXIDADE 17
2.2.1 Tempo Computacional 18
2.2.2 Eficiência dos algoritmos 19
2.3 CLASSIFICAÇÃO DOS PROBLEMAS 21
2.3.1 Classificação do Problema do Caixeiro Viajante 22
2.4 ESPAÇO DE BUSCA 24
2.5 PARADIGMAS DE SOLUÇÃO DE PROBLEMAS 24

3 COMPUTAÇÃO EVOLUTIVA 27

3.1 INTRODUÇÃO 27
3.2 TEORIA DA EVOLUÇÃO 28
3.3 PARADIGMA COMPUTACIONAL 29
3.4 ALGORITMOS GENÉTICOS 30
3.4.1 Representação 32
3.4.2 Seleção 33
3.4.3 Função de Avaliação 34
3.4.4 Crossover 34
3.4.4.1 Order Crossover (OX) 35
3.4.4.2 Partial-Mapped Crossover (PMX) 35
3.4.4.3 Cycle Crossover (CX) 36
3.4.5 Mutação 37
3.4.5.1 Exchange Mutation (EM) 38
3.4.5.2 Displacement Mutation (DM) 38
3.4.5.3 Simple Inversion Mutation (SIM) 38
3.4.5.4 Insertion Simple Mutation (ISM) 39
3.4.5.5 Inversion Displacement Mutation (IDM) 39
3.4.6 O Algoritmo Genético 40
3.4.7 O Problema da Convergência Prematura 41

4 COMPUTAÇÃO PARALELA E CONCORRENTE 43

4.1 INTRODUÇÃO 43
4.2 COMPUTAÇÃO PARALELA 44
4.2.1 Programa Paralelo 45
4.2.2 Arquiteturas Paralelas 45
4.3 COMPUTAÇÃO CONCORRENTE 48
4.4 CONSTRUÇÃO DE PROGRAMAS PARALELOS 49
4.4.1 Criação e Término de Processos 51
4.4.2 Sincronização de Processos 52
4.5 ALGORITMOS GENÉTICOS PARALELOS (PGAS) 52

5 MÉTODO DESENVOLVIDO 55
5.1 O PROBLEMA DO CAIXEIRO VIAJANTE 55
5.2 TRABALHOS RELACIONADOS 56
5.2.1 Algoritmo de Lohn 56
5.2.2 Algoritmo de Mühlembein 57
5.2.3 Algoritmo de Alaoui 58
5.2.4 Discussão 58
5.3 ESTRUTURA DO AG PARALELO IMPLEMENTADO 59
5.3.1 Representação Adotada 60
5.3.2 Função de Avaliação da Aptidão 61
5.3.3 Seleção para Reprodução 61
5.3.4 Troca de Informação entre as Populações 62
5.3.5 Controle da Estagnação 62
5.3.6 Processo de Seleção 63
5.3.7 Exploração do Paralelismo 63
5.4 PROTÓTIPO IMPLEMENTADO 64
5.4.1 Estrutura do Protótipo 64
5.4.1.1 Classe Supervisor 65
5.4.1.2 Classe Population 65
5.4.1.3 Classe Indivíduo 67
5.4.2 Simulação do Paralelismo 70
6 ANÁLISE DOS RESULTADOS OBTIDOS 71
6.1 INSTANCIA ATT48 71
6.2 INSTANCIA BERLIN52 73
6.3 INSTANCIA KROC100 75
6.4 INSTANCIA EIL101 76
6.5 INSTANCIA CH150 78
6.6 INSTANCIA GR202 79
6.7 INSTANCIA TS225 81
6.8 INSTANCIA A280 83
6.9 INSTANCIA PCB442 85
6.10 INSTANCIA GR666 86
6.11 INSTANCIA PR1002 87
6.12 DISCUSSÃO DOS RESULTADOS 88
7 CONSIDERAÇÕES FINAIS 90
7.1 CONCLUSÃO 90
7.2 SUGESTÕES PARA NOVAS PESQUISAS 91
8 REFERÊNCIAS BIBLIOGRÁFICAS 92

Lista de Figuras

Figura 1 – Uma instancia do PCV representada na forma de grafo 23
Figura 2 – Aplicação do operador – *Order Ccrossover* – (OX)........ 36
Figura 3 – Aplicação do operador – P*artial-Mapped Crossover* – (PMX)........ 36
Figura 4 – Aplicação do operador – C*ycle Crossover* – (CX) 37
Figura 5 – Aplicação do Operador – *Exchange Mutation* – EM........ 38
Figura 6 – Aplicação do Operador – *Displacement Mutation* – DM 38
Figura 7 – Aplicação do operador –*Simple Inversion Mutation* – SIM 39
Figura 8 – Aplicação do operador – *Insertion Simple Mutation* – ISM 39
Figura 9 – Aplicação do operador – *Inversion Displacement Mutation* – IDM........ 40
Figura 10 – Fluxograma genérico dos Algoritmos Genéticos........ 41
Figura 11 – Diagrama básico de uma máquina SISD........ 46
Figura 12 – Diagrama básico de uma máquina SIMD 46
Figura 13 – Diagrama básico de uma máquina MISD 47
Figura 14 – Diagrama básico de uma máquina MIMD........ 48
Figura 15 – Um *Thread* / Contexto de Execução........ 51
Figura 16 – AG Paralelo – *Master - Slave* 57
Figura 17 – Representação adotada........ 61
Figura 18 – Intervalo para seleção aleatória dos pares........ 62
Figura 19 – Hierarquia de Comando 64
Figura 20 – Evolução da Instância Att48 73
Figura 21 – Evolução da Instancia Berlin52 74
Figura 22 – Evolução da Instancia Kroc100 76
Figura 23 – Evolução da Instancia EIL101 77
Figura 24 – Evolução da Instancia CH150........ 79
Figura 25 – Evolução da Instancia GR202........ 81
Figura 26 – Evolução da Instancia TS225........ 83
Figura 27 – Evolução da Instancia A280 84
Figura 28 – Melhor Resultado da Instancia GR666 87

Lista de Tabelas

Tabela 1 – Ordens de grandeza usadas pra expressar a complexidade de um algoritmo 20
Tabela 2 – Comparação dos tempos de várias funções de complexidade 21
Tabela 3 – Resultados da Instancia ATT48 72
Tabela 4 – Resultados da Instancia Berlin52 73
Tabela 5 – Resultados da instância KROC100 75
Tabela 6 – Resultados da Instancia EIL101 76
Tabela 7 – Resultados da instância CH150 78
Tabela 8 – Resultados da Instancia GR202 79
Tabela 9 – Resultados da Instancia TS225 81
Tabela 10 – Resultados da Instancia A280 83
Tabela 11 – Resultados da Instancia PCB442 85
Tabela 12 – Resultados da Instancia GR666 86
Tabela 13 – Resultados da Instancia PR1002 87

Lista de Gráficos

Gráfico 1 – Att48 – 1 População 50 Indivíduos 72
Gráfico 2 – Att48 – 10 Populações 50 Indivíduos 72
Gráfico 3 – Att48 – 1 População 500 Indivíduos 72
Gráfico 4 – Att48 – Comparativo da Região Critica 72
Gráfico 5 – Berlin52 – 1População 50 Indivíduos 74
Gráfico 6 – Berlin52 – 10 Populações 50 Indivíduos 74
Gráfico 7 – Berlin52 – 1 População 500 Individuos 74
Gráfico 8 – Berlin52 – Comparativo da Região Critica 74
Gráfico 9 – Kroc100 – 1 População 100 Indivíduos 75
Gráfico 10 – Kroc100 – 10 Populações 100 Indivíduos 75
Gráfico 11 – Kroc100 – 1População 1000 Indivíduos 75
Gráfico 12 – Kroc100 – Comparativo da Região Critica 75
Gráfico 13 – Eil101 – 1 População 100 Indivíduos 77
Gráfico 14 – Eil101 – 10 Populações 100 Indivíduos 77
Gráfico 15 – Eil101 – 1 População 1000 Indivíduos 77
Gráfico 16 – Eil101 – Comparativo da Região Critica 77
Gráfico 17 – CH150 – 1 População 150 Indivíduos 78
Gráfico 18 – CH150 – 10 Populações 150 Indivíduos 78
Gráfico 19 – CH150 – 1População 1000 Indivíduos 78
Gráfico 20 – CH150 – Comparativo da Região Crítica 78
Gráfico 21 – GR202 – 1 População 100 Indivíduos 80
Gráfico 22 – GR202 – 10 Populações 100 Indivíduos 80
Gráfico 23 – GR202 – 1 População 1000 Indivíduos 80
Gráfico 24 – GR202 – Comparativo da Região Critica 80
Gráfico 25 – TS225 – 1 População 100 Indivíduos 82
Gráfico 26 – TS225 – 10 Populações 100 Indivíduos 82
Gráfico 27 – TS225 – 1 População 1000 Indivíduos 82
Gráfico 28 – TS225 – Comparativo da Região Critica 82
Gráfico 29 – A280 – 1 População 200 Indivíduos 84
Gráfico 30 – A280 – 10 Populações 200 Indivíduos 84
Gráfico 31 – A280 – 1 População 1000 Indivíduos 84
Gráfico 32 – A280 – Comparativo da Região Critica 84
Gráfico 33 – PCB442 – 1 População 100 Indivíduos 85
Gráfico 34 – PCB442 – 10 Populações 100 Indivíduos 85
Gráfico 35 – PCB442 – 1 População 1000 Indivíduos 85
Gráfico 36 – PCB442 – Comparativo da Região Critica 85
Gráfico 37 – GR666 – 1 População 100 Indivíduos 86
Gráfico 38 – GR666 – 10 Populações 100 Indivíduos 86
Gráfico 39 – GR666 – 1 População 1000 Indivíduos 86
Gráfico 40 – GR666 – Comparativo da Região Critica 86
Gráfico 41 – PR1002 – 1Populaçao 200 Indivíduos 88
Gráfico 42 – PR1002 – 5 Populações 200 Indivíduos 88
Gráfico 43 – PR1002 – 1 População 1000 Indivíduos 88
Gráfico 44 – PR1002 – Comparativo da Região Critica 88

1 Introdução

A pesquisa por métodos de otimização combinatorial tem se intensificado nas últimas décadas. No entanto, apesar dos esforços em propor modelos matemáticos que solucionem, de forma exata, problemas dessa natureza, como o Problema do Caixeiro Viajante -PCV. Essa questão permanece em aberto, uma vez que esses modelos alem de serem dependentes do problema, apresentam tempos de execução extremamente proibitivos. Num outro sentido, novas técnicas da computação evolutiva, tais como *Simulated Annealing*, *Tabu Search* e Algoritmos Genéticos, têm se apresentado como alternativas bastante otimistas, mesmo considerando que seus resultados são aproximados, não exatos.

A computação paralela consiste, basicamente, em elementos de processamento, que cooperam e comunicam-se entre si para solucionarem problemas mais complexos, de maneira mais rápida do que se estivessem sendo solucionados seqüencialmente (ALMASI & GOTTLIEB, 1994). Esta definição explica claramente os fatores que motivam o direcionamento de esforços no sentido de desenvolver mecanismos de paralelização de processos computacionais em geral. Por outro lado, o paralelismo também ocasiona o surgimento de uma série de novas características em relação ao gerenciamento e manutenção da coerência das informações processadas (SANTOS, 2001).

Os Modelos baseados em Algoritmos Genéticos apresentam-se como intrinsecamente paralelos. A geração e avaliação dos novos indivíduos durante o processo de evolução para a solução ótima podem ser executadas em paralelo. Dessa forma pode ser possível reduzir o tempo de execução adicionando mais processadores. Se por um lado, a adição de mais

processadores aumenta o tempo computacional sem aumentar o tempo de execução, por outro, com execução paralela torna-se possível avaliar e miscigenar várias populações simultaneamente. O presente trabalho, enfoca o uso da computação paralela como forma de redução do tempo de execução, aceleração da convergência e melhoria nas soluções encontradas através de métodos baseados em algoritmos genéticos.

1.1 Motivação

Este trabalho foi motivado pelo desejo de desenvolver uma estrutura de algoritmo genético capaz de tirar proveito do poder de processamento encontrado em máquinas multiprocessadas e/ou redes de computadores.

1.2 Objetivos do Trabalho

Este trabalho tem como objetivo desenvolver um mecanismo de exploração do poder de processamento de máquinas dotadas de múltiplos processadores, para métodos baseados em Algoritmos Genéticos. O ponto central é a definição de uma estrutura capaz de explorar o paralelismo encontrado em máquinas multiprocessadas, ao mesmo tempo em que permite a distribuição em uma rede de computadores para a exploração do paralelismo de máquina.

Verificar a hipótese de que várias populações de pequeno e médio porte, evoluindo em paralelo, mas de forma não isoladas, apresentem o mesmo resultado que uma grande população.

Verificar a hipótese de que a troca de informações – material genético – entre as populações acarreta uma melhor performance do algoritmo acelerando a convergência.

1.3 Organização do texto

Este trabalho, dividido em 7 capítulos. O capitulo 1 faz uma introdução geral sobre a problemática em questão. Descreve os objetivos gerais e específicos do trabalho; os aspectos e elementos que motivaram a escolha do tema; as limitações do trabalho; e as diretrizes que nortearam a pesquisa.

No capítulo 2, são abordadas: a questão da complexidade computacional e sua quantificação, bem como as métricas usadas para classificar os algoritmos quanto à sua eficiência, sendo apresentada uma discussão sobre a classificação dos problemas quanto aos seus níveis de dificuldade. Nesse capítulo há também uma breve descrição de algumas técnicas de busca usadas para obtenção da solução ótima ou uma aproximação da mesma.

O capítulo 3 aborda a computação evolutiva como paradigma para resolução de problemas considerados difíceis. Nesse capítulo é dada uma introdução à teoria da evolução proposta por Charles Darwin no século XIX e seu uso como paradigma computacional de otimização. O capítulo cobre ainda os algoritmos genéticos, sua base teórica inspirada na teoria da evolução genética, seus operadores básicos e a seqüência típica de operações que compõem um algoritmo genético.

O capítulo 4 enfoca a computação paralela, abrangendo os pontos essenciais do processamento paralelo e concorrente. Esse capítulo trás o embasamento teórico básico para a compreensão e utilização efetiva da computação paralela e da computação concorrente.

O capítulo 5 descreve o método desenvolvido. Inicialmente é dada uma definição para o problema do caixeiro viajante, em seguida são apresentados alguns trabalhos relacionados; seguindo-se com a descrição da estrutura de AG paralelo proposto e do protótipo implementado.

O capítulo 6 é dedicado à análise dos resultados obtidos, os quais estão dispostos em seqüência de acordo com o tamanho da instância. Para cada instância, é apresentada uma tabela com os melhores resultados obtidos, os gráficos da curva de convergência de cada tipo de teste e um gráfico comparativo entre os testes.

O capítulo 7 apresenta as considerações finais e sugestões para novos trabalhos na área.

1.4 Limitações da pesquisa

O presente trabalho não teve como pretensão desenvolver um novo método para a resolução de problemas de otimização combinatorial. Este trabalho limita-se ao desenvolvimento de uma estrutura de implementação paralela para Algoritmos Genéticos capaz de explorar o paralelismo existente em máquinas multiprocessadas e com possibilidade de distribuição em rede.

O paralelismo foi simulado pelo uso da programação concorrente, com a utilização de *threads*.

O corpo de prova é o clássico problema do caixeiro viajante com instâncias colhidas na internet, através do site http://www.math.princeton.edu/tsp.

A análise comparativa entre operadores de *crossover,* mutação e seleção, bem como a comparação de tempo de processamento entre as versões *standard* e paralela, estão fora do escopo deste trabalho.

2 Complexidade Computacional

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 humanos, podem se tornar extremamente complexos quando da tentativa de modela-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, noutros 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. Um Algoritmo é importante, do ponto de vista da complexidade computacional, se ele é especial de algum modo. Geralmente provendo um modo surpreendentemente rápido de resolver um problema simples ou importante. (COOK, 1983).

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 pouco tamanho.

2.2 Analisando a Complexidade

Como um modelo computacional, a máquina de Turing[1] fornece uma maneira de demonstrar a existência de problemas insolúveis (que não são computáveis). A máquina de Turing não ajuda apenas a encontrar os limites da computabilidade, mas também pode ajudar a classificar os 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. Segundo Gersting (GERSTING, 1993) "Pela tese de Church – Turing[2], 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" Assim 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 assim, tendo-se um algoritmo, alguns problemas são considerados intratáveis computacionalmente, dado o tempo necessário para a execução do mesmo. Problemas de otimização combinatorial são casos de exemplo.

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 solução do problema em questão. Assim a complexidade diz respeito a esse algoritmo ou conjunto de algoritmos.

Os problemas que não podem ser computacionalmente resolvidos são aqueles para os quais a teoria atual da ciência da computação não consegue produzir um algoritmo. É desnecessário

[1] A máquina de Turing foi proposta pelo matemático inglês Alan M. Turing em 1936. Consiste essencialmente de uma máquina de estado finito com a habilidade de ler suas entradas mais de uma vez e de apagar ou substituir os valores de suas entradas, ela também possui uma memória auxiliar ilimitada.

[2] Uma função número-teórica é computável por um algoritmo se, e somente se, ela for Turing-computável.

dizer que as técnicas tradicionais disponíveis são insuficientes para resolver esse tipo de problema. “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” (LEWIS & PAPADIMITRIOU, 2000).

2.2.1 Tempo Computacional

A análise da complexidade computacional de um algoritmo é um dos pontos mais críticos, quando o objeto de estudo é um problema cujo espaço de soluções cresce de forma não polinomial. Analisar a complexidade diz respeito à análise dos recursos computacionais necessários 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 ponto para a análise da complexidade. Por outro lado o número de operações necessárias se mantém fixo independentemente da máquina onde o algoritmo é executado, logo pode ser considerado como um ponto para a análise da complexidade computacional.

Determinar o tempo computacional de um algoritmo significa determinar o tempo de execução em termos de operações de máquina. O tempo real, ou seja, o tempo de execução medido por cronômetro, pode apresentar variações de uma máquina para outra. Fatores como processador e memória disponível influenciam diretamente no tempo medido pelo cronômetro. Já o tempo computacional é independente do tempo do relógio, ele é contado em termos de ciclos de CPU e não em segundos, milisegundos, etc. A determinação do tempo computacional é realizada através de métodos analíticos que determinam a ordem de grandeza do tempo de execução. O objetivo desses métodos é determinar uma função matemática que traduza o comportamento do tempo do algoritmo, que ao contrário do método empírico, visa aferir o tempo de execução de forma independente do ambiente utilizado.

Para um dado problema, a complexidade é uma função que relaciona o tamanho de uma instância ao tempo necessário para resolvê-la. Essa função normalmente expressa a quantidade das operações elementares realizadas. A noção de complexidade de tempo é descrita a seguir segundo Routo, (ROUTO, 1991).

Seja **A** um algoritmo, **{E_1,..., E_m}** o conjunto de todas as entradas possíveis de **A**. Denote por **t_i**, o número de passos efetuados por **A**, quando a entrada for **E_i** . Definem-se:

Complexidade do pior caso = $\mathbf{max\ \ Ei \in E \{ ti \}}$,

Complexidade do melhor caso = $\mathbf{min\ \ Ei \in E \{ ti \}}$,

Complexidade do caso médio = $\mathbf{\Sigma\ \ 1 \leq i \leq m\ \ pi \cdot ti}$,

onde **p_i** é a probabilidade de ocorrência da entrada **E_i**.

É importante observar que a complexidade associada ao pior caso fornece um limite superior ao número de operações que o algoritmo vai executar. Esse limite superior é o ponto central de referência para a complexidade computacional.

2.2.2 Eficiência dos algoritmos

Um algoritmo eficiente é aquele que minimiza o número de operações elementares. É comum expressar a complexidade de um algoritmo através da ordem de grandeza da função que o modela. Quando se expressa a complexidade através de sua ordem de grandeza, pode-se desprezar constantes aditivas ou multiplicativas. Uma notação comum para simbolizar essa ordem de grandeza é a letra O, seguida por uma função entre parênteses que representa a ordem de grandeza. A Tabela 1, apresenta as ordens de grandezas mais comuns.

Definição: Diz-se que um algoritmo tem ordem de grandeza ou complexidade O(f(n)) quando o número de operações elementares executadas para obter a solução à instância do problema de tamanho n não exceder uma constante vezes f(n), para n suficientemente grande, ou seja, diz-se que g(n) é O(f(n)) se existirem duas constantes K e n' tal que |g(n)| = K |f(n))|, para todo n = n'.

Definição: Diz-se que um algoritmo tem complexidade de tempo polinomial se sua função de complexidade g(n) é um polinômio ou se é limitada por outra função polinomial.

Função	Ordem de Grandeza
Constante	O(1)
Logarítmica	O(log n)
Linear	O(n)
Quadrática	$O(n^2)$
Cúbica	$O(n^3)$
Polinomial	$O(n^c)$, c real
Exponencial	$O(c^n)$, c real e > 1
Fatorial	O(n!)

Tabela 1 – Ordens de grandeza usadas pra expressar a complexidade de um algoritmo

“Algoritmos com complexidade polinomial são aceitos como de origem de problemas solúveis na prática e, portanto, algoritmos computacionalmente viáveis” (LEWIS & PAPADIMITRIOU, 2000). Na prática alguns problemas, como o do caixeiro viajante, apresentam algoritmos cuja complexidade não é limitada por um polinômio. Para problemas desse tipo, o espaço das soluções cresce de forma explosiva à medida que **n** cresce e, portanto seus algoritmos são computacionalmente inviáveis.

> **Definição:** Diz-se que um algoritmo tem complexidade de tempo exponencial quando sua função de complexidade g(n) não é limitada por uma função polinomial.

Apesar de parecerem bem comportadas, algumas funções da Tabela 1, que representam as respectivas complexidades, podem apresentar surpresas, e por vezes desagradáveis, quando o tamanho de **n** fica suficientemente grande. Na Tabela 2, podem-se observar as diferenças na velocidade de crescimento de várias funções típicas de complexidade. Os valores expressam os tempos de execução quando cada operação elementar no algoritmo é executável em um décimo de microssegundo.

	Valores para n		
Função	**20**	**40**	**60**
N	0,0002 s	0,0004 s	0,0006 s
n log n	0,0009 s	0,0021 s	0,0035 s
n^2	0,004 s	0,016 s	0,036 s
n^3	0,08 s	0,64 s	2,26 s
2^n	10 s	127 dias	3.660 séculos
3^n	580 min	38.550 séculos	$1,3 \times 10^{14}$ séculos

Tabela 2 – Comparação dos tempos de várias funções de complexidade

Fonte: (ROUTO, 1991)

2.3 Classificação dos problemas

Estabelecer a complexidade de um problema, é um dos primeiros passos para determinar se há ou não um algoritmo para resolve-lo, que seja viável computacionalmente. A classificação atual está dividida em 3 grandes grupos ***P***, ***NP*** e ***NP-Completos***.

> **Definição**: A classe P é formada pelo conjunto de todos os problemas que podem ser resolvidos por um algoritmo deterministico em tempo polinomial.

A computação limitada polinomialmente é um conceito atraente e produtivo para uma teoria elegante e útil, já que inclui a maioria dos algoritmos viáveis e exclui a maioria dos impraticáveis. Essa fronteira entre os algoritmos viáveis e os impraticáveis ainda é bastante cinzenta, uma vez que é possível encontrar algoritmos polinomiais que são impraticáveis em termos práticos em função de seus elevados índices.

> **Definição:** Problemas NP são aqueles problemas que podem ser resolvidos por uma máquina não-deterministica em tempo polinomial.

A idéia de encontrar a solução para um problema checando todas as possíveis soluções em paralelo, para determinar qual é a melhor “correta” é chamada de não-determinismo. Algoritmos com tal característica são chamados de algoritmos não-deterministicos. Qualquer problema para o qual exista um algoritmo que executa em uma máquina não-deterministica em tempo polinomial, pertence ao conjunto NP.

A classe NP engloba todos os problemas para os quais não é possível um algoritmo deterministico com tempo polinomial, mas que, dada uma solução, essa pode ser verificada em tempo polinomial. A classe NP é formada pelo conjunto de todos os problemas de decisão[3] D, para os quais, existe uma justificativa à resposta SIM para D, cujo passo de verificação pode ser realizado por um algoritmo polinomial do tamanho da entrada de D.

> “Se uma máquina de Turing pode determinar em tempo polinomial quando uma cadeia arbitrária pertence a um conjunto, certamente pode verificar um elemento do conjunto em tempo polinomial. Portanto $P \subseteq NP$. No entanto, não se sabe se esta inclusão é própria, isto é, não se sabe se $P = NP$” (GERSTING, 1993).

> Um conjunto S de strings é P-redutível (P para polinomial) para um conjunto T de strings se existir uma máquina de consulta M e um polinômio Q(n) tal que para cada string de entrada w, A computação de T por M com entrada w, termina dentro de Q(|w|) passos (|w| é o comprimento de w) e o fim está num estado aceitável se $w \in S$ (COOK, 1971).

2.3.1 Classificação do Problema do Caixeiro Viajante

A Figura 1, apresenta uma instância do PCV com cinco cidades ABCDE, representada pelo grafo G(V,E) no qual cada vértice V representa uma cidade e cada arco ***e*** representa um caminho entre um par de cidades.

Encontrar a solução para o PCV consiste em encontrar o menor caminho passando por todas as cidades uma única vez e retornando ao local de partida. Um exemplo de viajem seria ABCDEA com custo total 12, onde o custo é a soma dos pesos dos arcos em seqüência. Outro caminho poderia ser ABCEDA com custo 11. Este problema não pode ser resolvido em tempo

[3] Problemas de Decisão são aqueles que podem ser respondidos com sim ou não.

polinomial com uma máquina não-deterministica, no entanto, dado um ciclo candidato é possível checar rapidamente se é ou não um ciclo com a forma apropriada, entretanto não há uma forma simples de saber se de fato é o ciclo mais curto.

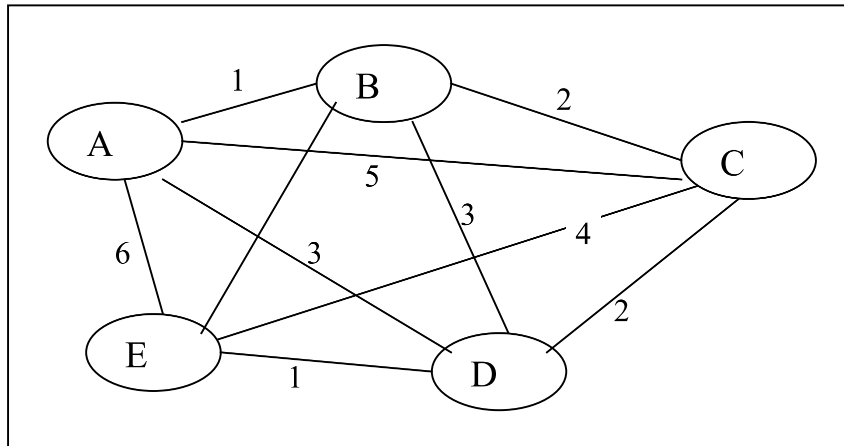


Figura 1 – Uma instancia do PCV representada na forma de grafo

Por outro lado, é possível resolver uma variante - PCV’ - deste problema, apresentado sob a forma de um problema de decisão, cuja resposta é sim ou não.

PCV’

Entrada: Um grafo dirigido completo G(*V,E*) com distancias associadas a cada um de seus arcos, e um inteiro K.

Saída: SIM, existe um ciclo simples com custo total < = K, contendo todos os vértices de G, e NÃO caso contrário.

Essa versão do PCV pode ser resolvida em tempo polinomial por uma máquina não-deterministica. O algoritmo simplesmente checa todos os possíveis subconjuntos de arcos no grafo em paralelo. Se qualquer subconjunto de arcos é um ciclo apropriado de comprimento total menor ou igual a K, a resposta é SIM, caso contrário é NAO. Checar um subconjunto em particular, é realizado em tempo polinomial, somando-se os comprimentos de cada arco no ciclo que visita cada vértice exatamente uma vez. Infelizmente, existem (V-1)!/2 possibilidades a checar. E deste modo não é possível um algoritmo deterministico com tempo polinomial executando numa máquina regular. Assim como o PCV há uma grande coleção de problemas com esta propriedade, para os quais são conhecidos algoritmos não deterministas eficientes, no entanto não é sabido se existe algum algoritmo determinista com tempo

computacional aceitável. Também não há provas de que tal algoritmo não exista. Esta classe de problemas é denominada NP-Completos. Desta forma o PCV é um problema NP-Completo.

> **Definição:** Um problema X é NP-Completo, se X está em NP e qualquer outro problema em NP pode ser reduzido a X em tempo polinomial.(GERSTING, 1993)

2.4 Espaço de busca

A resolução de problemas é fundamental para a maioria das aplicações. Existem basicamente dois tipos de problemas. O primeiro tipo pode ser resolvido por algum procedimento deterministico. São os problemas computáveis (ver introdução). Contudo, existem inúmeros problemas do mundo real que não possuem soluções deterministicas. Esses problemas estão alocados na segunda categoria, ou seja, a dos problemas não computáveis. Para encontrar uma solução aceitável, para esses problemas lança-se mão da aplicação de métodos de busca. Um dos obstáculos mais difíceis de superar quando são aplicadas técnicas busca nos problemas do mundo real é a magnitude e complexidade da maioria das situações.

O espaço de busca de um problema é o conjunto de todas as possíveis soluções para o mesmo. Em notação matemática, o espaço é denotado pelo subconjunto S de R^n, ou seja, $S \subseteq R^n$. Encontrar a solução de um determinado problema significa encontrar o conjunto S' de R^n tal que uma dada função f**:** $S' \rightarrow R$ satisfaz a um determinado critério, geralmente valor máximo ou mínimo. Encontrar S' tal que $f(x') = f(x)$, $\forall\ x \in S$, $x' \in S'$, no caso de mínimo.

2.5 Paradigmas de solução de problemas

A solução de um problema consiste em encontrar um valor ou conjunto de valores que

satisfaça o enunciado do problema e suas restrições. Em muitos casos a solução é única, em outros pode haver um conjunto de soluções que podem ser classificadas como ruins, boas e ótimas. A natureza do problema determina o método a ser utilizado na sua solução.

Segundo Zuben (ZUBEN, 2000), os métodos de busca 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 impera linearidade, continuidade, diferenciabilidade e/ou estacionariedade. São exemplos: método do gradiente e técnicas de programação linear (busca iterativa).
>
> **Métodos específicos:** são concebidos para resolver problemas específicos em mundos específicos. Exemplo: toda técnica que conduz a uma solução na forma fechada.
>
> **Métodos fracos:** são concebidos para resolverem problemas genéricos em mundos genéricos. Operam em mundos não-lineares e não-estacionários, embora não garantam eficiência total na obtenção da solução, geralmente garantem a obtenção de uma "boa aproximação" para a solução ótima, sendo que a complexidade algorítmica cresce a uma taxa menor que exponencial com o aumento do "tamanho" do problema. Exemplo: técnicas baseadas em computação evolutiva.

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

> **Métodos numéricos:** são bastante utilizados e apresentam soluções exatas em muitos tipos de problemas, entretanto são ineficientes para otimização de funções multimodais[4]. Além disso, em problemas de otimização

[4] Funções multimodais são aquelas cujo espaço de busca apresenta mais de um mínimo/máximo, dessa forma a solução encontrada pode se caracterizar como uma solução ótima local, o que não significa ser a solução ótima para o problema.

combinatorial em espaços discretos, geralmente a função a ser otimizada é não somente multimodal, mas também não diferenciável e/ou não contínua.

Métodos enumerativos: examinam cada ponto do *espaço de busca*, um por um, em busca de pontos ótimos. A idéia pode ser intuitiva, mas é obviamente impraticável quando há um número infinito ou extremamente grande de pontos a examinar. Esse método poderia ser chamado de método da força bruta, pois é exatamente esse o seu princípio.

Métodos probabilísticos: vêm ganhando espaço e estão presentes nas abordagens mais modernas. Estes métodos empregam a idéia de busca probabilística, o que não quer dizer que sejam métodos totalmente baseados em sorte, como é o caso dos chamados métodos aleatórios. Dentre as abordagens mais conhecidas, estão os algoritmos *Simulated Annealing* e Genético.

A utilização de métodos fracos ou probabilísticos tem ganhado força 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 desenvolver e aperfeiçoar essa técnica.

No caso do PCV, a explosão combinatória gerada no espaço de busca torna o método enumerativo computacionalmente intratável. Uma instancia de 10 cidades, por exemplo, gera um espaço de (10-1)/2!, ou seja, 9! = 9*8*7*6*5*4*3*2 = 362880/2 = 181440 caminhos possíveis. Assim como o PCV, uma grande parte dos problemas científicos pode ser formulada como problemas de otimização combinatorial. Desta forma o PCV pode ser utilizado como um corpo de teste para o desenvolvimento e aperfeiçoamento de algoritmos de aproximação.

3 Computação Evolutiva

3.1 Introdução

A sabedoria da natureza manifestada através dos princípios de evolução natural genética e seleção natural tem inspirado e provado ser um mecanismo poderoso para o aparecimento e aperfeiçoamento de seres vivos no nosso planeta.

Sistemas evolutivos artificiais são algoritmos computacionais com certas propriedades inspiradas na evolução natural. Esta classe de algoritmos recebe também o nome de computação evolutiva. "Pode-se definir algoritmos evolutivos como programas de computador de caráter geral estocástico que executam *otimização* através de um processo de aprimoramento de um conjunto de soluções, empregando *operadores* inspirados na evolução biológica" (ESBENSEN, 1998 *apud* ZEBULUM, 1999).

A *computação evolutiva* é uma área de estudo que busca a síntese de algoritmos inspirados em aspectos essenciais da evolução natural e os aplica na solução de problemas em geral. Segundo esse paradigma, abre-se mão da garantia de obtenção da solução ótima e passa-se a aceitar soluções aproximadas, obtidas através de uma ferramenta de propósito geral. Estes *algoritmos evolutivos* recebem diferentes denominações de acordo com certas

particularidades que eles apresentem: algoritmos genéticos, programação genética, programação evolutiva, estratégias evolutivas, entre outros. Existem três aspectos da evolução biológica nos quais estes algoritmos, em geral, se baseiam: a seleção natural, a recombinação de material genético e a mutação (RIDLEY, 1996).

Zuben (ZUBEN, 2000), esclarece que em termos históricos, três algoritmos para computação evolutiva foram desenvolvidos independentemente:

1. Algoritmos genéticos: Holland (1962), Bremermann (1962) e Fraser, (1957);

2. Programação evolutiva: Fogel (1962);

3. ·Estratégias evolutivas: Rechenberg (1965) e Schwefel (1965).

Dentre os algoritmos mencionados acima, os algoritmos genéticos de três operadores (HOLLAND, 1975) (GOLDBERG, 1989) e (ZUBEN, 2000), encontraram uma maior difusão, principalmente em aplicações relacionadas à otimização.

Neste capítulo é apresentado um breve histórico sobre a teoria da evolução e sua aplicação como paradigma computacional – *computação evolutiva.* Posteriormente são abordados os *algoritmos genéticos*, seus componentes principais e seu mecanismo básico de funcionamento.

3.2 Teoria da Evolução

A *Teoria da Evolução* foi primeiramente proposta pelo naturalista britânico Charles Darwin no século XIX. De 1831 a 1836, Darwin participou de uma expedição de ciência britânica ao redor do mundo. Na América do Sul, Darwin encontrou fósseis de animais extintos que eram semelhante a espécies modernas. Darwin também observou que muitas variações entre plantas e animais do mesmo tipo geral, presentes na América do Sul também se faziam presente nas Ilhas de Galápagos no Oceano Pacífico. De volta a Londres, Darwin registrou sua pesquisa e estabeleceu os pontos chaves da teoria da evolução das espécies.

- ✓ Todos os seres estão em evolução;
- ✓ A mudança evolutiva é gradual, requerendo de milhares a milhões de anos;
- ✓ O mecanismo primário para evolução é um processo chamado de seleção natural;
- ✓ As espécies de seres vivos existentes hoje surgiram a partir de uma única forma de vida original por um processo chamado "especialização".

A teoria de Darwin explica as variações existentes numa mesma espécie. Essas variações surgem fortuitamente através de processos evolutivos que buscam garantir a sobrevivência. A perpetuação ou extinção de cada organismo está determinada pela habilidade desse organismo se adaptar ao seu ambiente. Darwin publicou sua teoria em seu livro *"On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life"* (1859).

A evolução natural implementa mecanismos evolutivos e adaptativos de otimização que evoluem lentamente, a cada nova geração de indivíduos. Na evolução natural o problema que cada espécie enfrenta é a busca por adaptações benéficas a sua inserção no meio ambiente. O conhecimento adaptativo que cada espécie adquiri fica embutido no seu DNA.

3.3 Paradigma Computacional

O paradigma computacional da computação evolutiva ou evolucionária (CE) imita um modelo rudimentar e simplificado do modelo da natureza como um processo adaptativo de busca e otimização que possibilite implementações computacionais. A CE sugere um mecanismo em que uma população inicial de indivíduos (soluções) evolui aproximando-se, em média, a sua adequação em relação ao ambiente, ou seja, o seu desempenho geral com respeito a um dado problema (GOLDBERG, 1989).

A vantagem mais significativa da computação evolutiva está na possibilidade de resolver problemas pela simples descrição matemática do que se quer ver presente na solução, não havendo necessidade de se indicar explicitamente os passos até o resultado, que certamente seriam específicos para cada caso. É lógico que os algoritmos evolutivos correspondem a uma seqüência de passos até a solução, mas estes passos são os mesmos, para uma ampla gama de problemas, fornecendo robustez e flexibilidade. Sendo assim, a computação evolutiva deve ser entendida como um conjunto de técnicas e procedimentos genéricos e adaptáveis, a serem aplicados na solução de problemas complexos, para os quais outras técnicas conhecidas são ineficazes ou nem mesmo são aplicáveis.

3.4 Algoritmos Genéticos

Por anos, vários métodos de busca foram efetivamente usados para resolver problemas de otimização. Porém, o espectro de problemas hoje é tal que nenhum desses métodos de otimização provou ser adequado a todas as classes de problemas. “Métodos de busca enumerativos, como programação dinâmica, mantêm uma perspectiva global enquanto procuram evitar a convergência para um ótimo local, mas são ineficientes em problemas grandes” (BELLMAN, 196l *apud* KARR, 1993). Ainda, segundo Karr, Métodos baseados no cálculo do gradiente ou procedimentos *zero-finding* convergem para soluções de qualidade na otimização de funções unimodais com espaços de busca regulares, mas geralmente convergem para soluções sub ótimas ou locais quando aplicados a espaços de busca ruidosos, mal-comportados.

Há uma clara necessidade de pesquisa por métodos robustos que possam superar o problema de convergência local e localizar a solução ótima global de uma maneira simples e eficiente. Esses métodos robustos também deveriam poder resolver, efetivamente, um grande espectro de problemas de otimização.

Nesse contexto, o *Algoritmo Genético* (AG) apresenta-se como uma alternativa com boas perspectivas de sucesso. AGs são algoritmos de busca baseados nos mecanismos da genética, eles fazem uso de operações encontradas na genética natural para guiar a busca sobre o

espaço (HOLLAND, 1975). AGs apresentam convergência rápida, mesmo sobre grandes espaços de busca, enquanto exigem apenas o valor dado por uma função objetivo como parâmetro guia da busca. (KARR, 1993). Esta é uma característica convidativa porque as técnicas de busca mais comumente usadas requerem: informações derivadas, continuidade do espaço de busca, ou o conhecimento completo da função objetiva para guiar a pesquisa. Além disso, por causa do nível de processo associado com os AGs, estes agregam uma visão mais global do espaço de busca que muitos métodos encontrados na prática de otimização (GOLDBERG, 1989). Essas características favoráveis dos AGs foram inicialmente investigadas por Holland (HOLLAND, 1975).

Os algoritmos genéticos foram inicialmente desenvolvidos por J.H. Holland (HOLLAND, 1975), na *University of Michigan* nas décadas de 50 e 70, com refinamentos posteriores por D. Whitley, D.E. Goldberg, K. De Jong e J. Grefenstette;

Os *AGs* são técnicas não-deterministicas de busca e otimização que manipulam um espaço de soluções potenciais utilizando mecanismos inspirados nas teorias de seleção natural de C. Darwin e na genética de G. Mendel. Os *AGs* são robustos e eficientes em espaços de procura irregulares, multidimensionais e complexos, e caracterizam-se por: (GOLDBERG, 1994 *apud* COELHO, 1999):

- ✓ Operarem em uma população de pontos;
- ✓ Não requerem derivadas;
- ✓ Trabalharem com a codificação de seu conjunto de parâmetros, não com os próprios parâmetros (representação binária);
- ✓ Realizarem transições probabilísticas, não regras deterministicas;
- ✓ Necessitarem apenas de informação sobre o valor de uma função objetivo para cada integrante da população de indivíduos.

Segundo Mazzucco (MAZZUCCO, 1999) e Goldberg (GOLDBERG, 1989), os processos específicos de codificação e decodificação de cromossomos ainda não estão totalmente esclarecidos, mas existem algumas características gerais plenamente consolidadas:

- ✓ A evolução é um processo que se realiza nos cromossomos e não nos seres que os mesmos codificam;

- ✓ A seleção natural é a ligação entre os cromossomos e o desempenho de suas estruturas decodificadas. Os processos da seleção natural determinam que aqueles cromossomos bem sucedidos devem ser reproduzidos mais freqüentemente do que os mal sucedidos;

- ✓ É no processo de reprodução que a evolução se realiza. Através da mutação (*mutation*) o cromossomo de um ser descendente pode ser diferente do cromossomo de seu gerador. Através do processo de cruzamento (c*rossover*) é possível que o cromossomo do ser descendente se torne muito diferente daqueles dos seus geradores;

- ✓ A evolução biológica não possui memória. A produção de um novo indivíduo depende apenas de uma combinação de genes da geração que o produz.

Algoritmos Genéticos diferem das técnicas citadas anteriormente, devido ao fato de eles trabalharem com uma população de soluções e de usarem regras de transição probabilísticas. Além disso, AGs não usam, em princípio, nenhuma informação auxiliar sobre o espaço de busca, tal como o emprego de derivadas. Desta forma, algoritmos genéticos são cegos quando empregados em sua forma mais pura (GOLDBERG, 1989).

O funcionamento dos algoritmos genéticos pode ser explicado em termos da representação do problema; do uso de três operadores, seleção, *crossover* e mutação; e da aplicação de uma função de avaliação da aptidão (ZEBULUM, 1999).

3.4.1 Representação

Segundo Mazzucco (MAZZUCCO, 1999), A utilização do algoritmo genético na resolução de um determinado problema depende fortemente da realização de dois importantes passos iniciais:

1. Encontrar uma forma adequada de se representar soluções possíveis do problema em forma de cromossomo,

2. Determinar uma função de avaliação que forneça uma medida do valor (da importância) de cada cromossomo gerado, no contexto do problema.

A representação do problema diz respeito ao mapeamento das possíveis soluções presentes no espaço de busca em uma estrutura de dados que possa ser manipulada computacionalmente. Tipicamente, algoritmos genéticos codificam possíveis soluções em palavras binárias. Enquanto que estas palavras binárias recebem também a denominação de *cromossomos* ou *genótipos*, o ponto do espaço de busca codificado pelos mesmos recebe o nome de *fenótipo*. Desta forma, a implementação computacional de um AG inicia-se pela geração aleatória de uma população de genótipos. Sucessivas gerações de indivíduos são produzidas a partir da aplicação dos operadores genéticos.

A forma de representação das soluções possíveis em cromossomos varia de acordo com o problema. Nos trabalhos originais de Holland, essas representações eram feitas somente através de codificação utilizando cadeias de *bits.* Algoritmos que fazem uso dessa forma de codificação são tidos por muitos como algoritmos genéticos puros. Hoje as representações são elaboradas fazendo uso das mais diversas estruturas de dados.

3.4.2 Seleção

O operador de seleção baseia-se no princípio de sobrevivência dos mais aptos, por meio de uma metáfora aos processos de seleção natural observados na evolução biológica. A medida de desempenho de um indivíduo em relação a uma determinada especificação determinará a probabilidade deste indivíduo ser selecionado e contribuir para a criação de indivíduos em uma próxima geração. A característica mais importante do operador de seleção é o fato de ele ser probabilístico. Os indivíduos são selecionados de acordo com uma probabilidade dada pela sua função de adequação.

A probabilidade de seleção de um indivíduo é determinada pela sua medida de desempenho, também chamada de aptidão. A avaliação dos indivíduos é feita através da aplicação de uma

função de aptidão, que é definida com base na especificação do problema. Tipicamente, trata-se de uma função a ser otimizada. Após a aplicação desta função, cada indivíduo tem associado a si um valor escalar, real ou inteiro, que quantifica a aptidão do indivíduo em relação à solução do problema.

3.4.3 Função de Avaliação

A determinação da função de avaliação da aptidão é, em geral, simples quando apenas um critério ou especificação deve ser atendido. Esta especificação é então representada por uma função objetivo e esta é utilizada como função de avaliação da aptidão. Porém, a maior parte dos problemas de interesse prático deve atender a múltiplos objetivos, uma vez que diversas especificações devem ser levadas em consideração (GOLDBERG, 1989) e (FONSECA, 1995). A função de avaliação retorna um escalar como medida de desempenho do indivíduo.

A função de avaliação constitui-se no único elo de ligação entre o algoritmo genético e o problema a ser resolvido. Recebendo um cromossomo como entrada, essa função retorna um número ou uma lista de números que exprime a medida do desempenho daquele cromossomo, no contexto do problema a ser resolvido. A função de avaliação, no algoritmo genético, desempenha o mesmo papel que o ambiente no processo de evolução natural. Assim como, a interação de um indivíduo com seu meio ambiente fornece uma medida de sua aptidão, a interação de um cromossomo com uma função de avaliação também resulta em uma medida de aptidão, medida essa que o algoritmo genético utiliza na realização do processo de reprodução (MAZZUCCO, 1999).

3.4.4 Crossover

Após o processo de seleção, pares de indivíduos são escolhidos de forma aleatória para execução do operador de recombinação (*crossover*). O operador de *crossover* executa uma troca de informações entre dois indivíduos, a partir da miscigenação do conteúdo de seus respectivos cromossomos. O cruzamento de dois indivíduos via *crossover*, produz dois descendentes híbridos, ambos apresentando material genético dos dois progenitores. A

aplicação ou não do *crossover* após a escolha de dois cromossomos é probabilística, isto é, dois novos híbridos podem ser produzidos, ou então os dois indivíduos inicialmente selecionados são preservados.

"O operador genético de recombinação *(crossove*r) é responsável pela troca de material genético entre os indivíduos com probabilidade de reproduzirem mais freqüentemente bons indivíduos, ou seja, indivíduos mais aptos ao ambiente". (COELHO, 1999).

Segundo Miki, (MIKI *et all,* 1999), a performance de cada algoritmo genético é dependente do quão boa é a escolha do operador de *crossover* e da taxa de aplicação do operador de mutação. A escolha desses operadores deve considerar o escopo do problema a ser solucionado e sua respectiva representação genética.

No artigo intitulado *Genetic Algorithms for Travelling Saelsman Probem : A Review of Representation and Operators*, Larraña *et all* (LARRAÑA *et all,* 1998) descrevem diversos operadores de *crossover* e mutação propostos no correr de anos de pesquisas. Dentre eles destacam-se os operadores de *crossover* OX, PMX, CX e de mutação EM, DM, SIM, ISM, IDM.

3.4.4.1 Order Crossover (OX)

Segundo Larraña, o operador OX foi proposto por Davis (1985). Este operador explora a representação na forma de caminho, na qual a o importante é a ordem das cidades, não sua posição. Um descendente é construído tomando-se um retalho do cromossomo de um dos pais e preservando a ordem relativa de suas cidades no outro pai, como mostra a Figura 2.

3.4.4.2 Partial-Mapped Crossover (PMX)

Sugerido por Goldberg e Lingle (1985), o operador *partial-mapped corssover* cria os indivíduos descentes escolhendo dois segmentos de ambos os pais e então realiza o intercambio dos mesmos. No segundo passo, os elementos pertencentes ao segmento enviado são mapeados sobre os elementos do cromossomo, que coincidem com os do segmento recebido. O mapeamento evita a geração de indivíduos inválidos. O processo é mostrado na Figura 3.

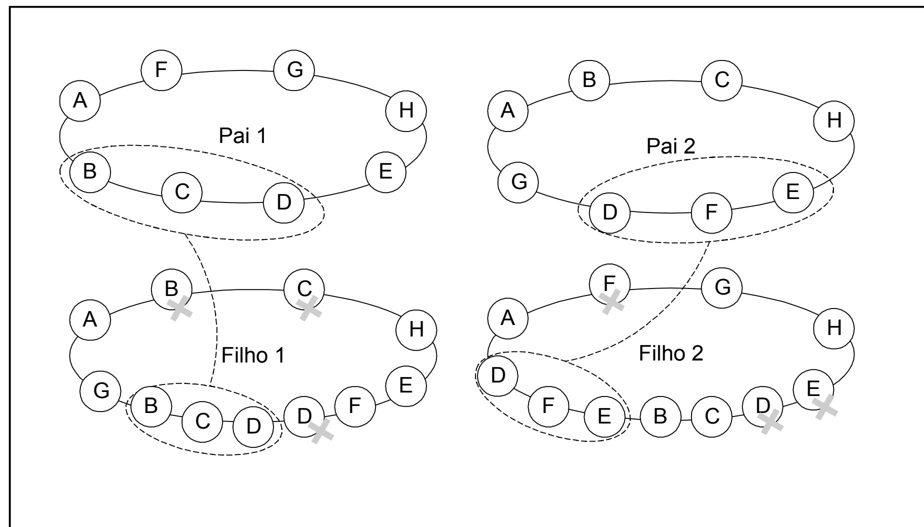


Figura 2 – Aplicação do operador – *Order Ccrossover* – (OX)

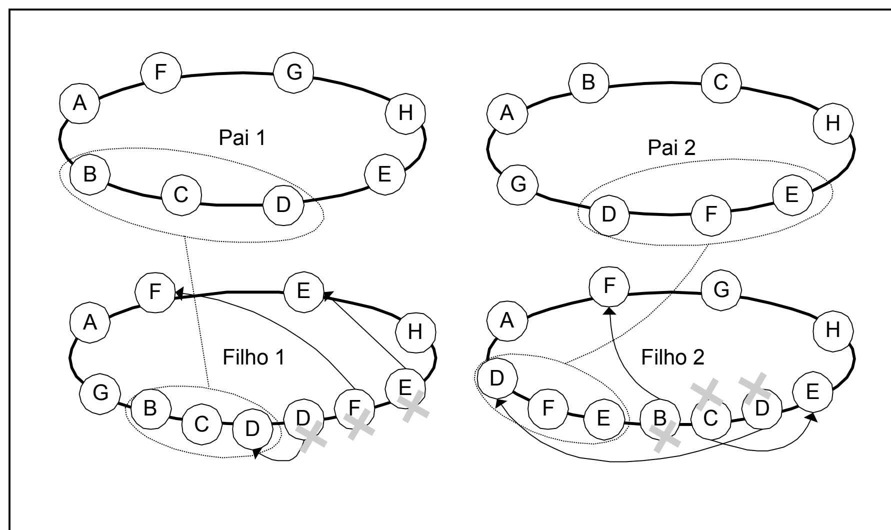


Figura 3 – Aplicação do operador – P*artial-Mapped Crossover* – (PMX)

3.4.4.3 Cycle Crossover (CX)

O operador *cycle crossover* foi proposto por Oliver *et all* (1987). Esse operador tenta criar descendentes onde cada posição é ocupada por um elemento correspondente a um dos pais. O primeiro elemento é escolhido aleatoriamente como sendo o primeiro elemento de um dos pais. Em seguida verifica-se a posição deste elemento no segundo pai e então se toma o elemento correspondente no primeiro pai. O processo se repete até formar um ciclo, neste ponto toma-se o primeiro elemento, do segundo pai, que ainda não está no filho repetindo o

processo novamente até completar o ciclo ou até a geração do filho ter sido concluída. A Figura 4 ilustra o processo.

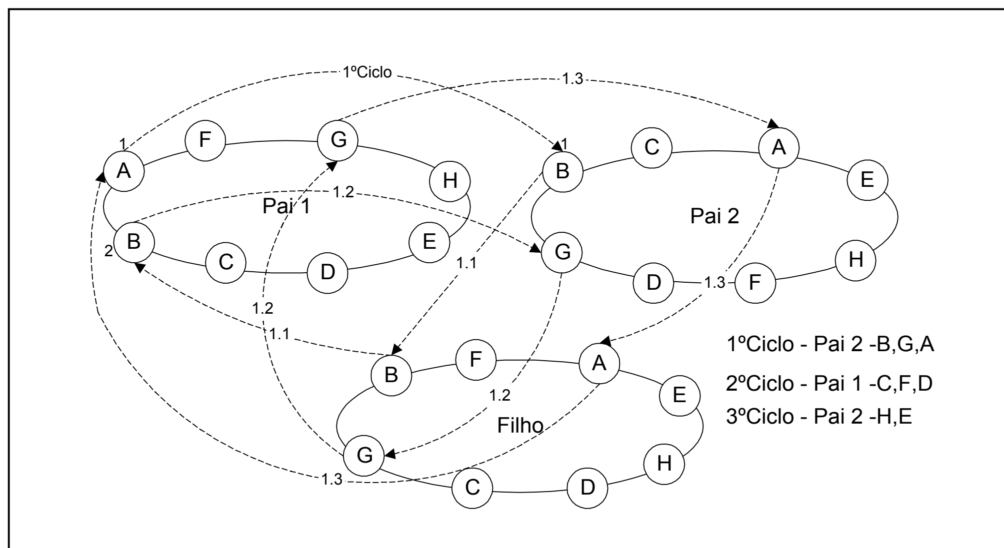


Figura 4 – Aplicação do operador – C*ycle Crossover* – (CX)

3.4.5 Mutação

Após a aplicação do operador de *crossover*, a nova geração de indivíduos passa pelo processo de mutação. O operador de mutação implica na modificação do valor de um ou mais genes de um indivíduo e visa restaurar o material genético perdido ou não explorado em uma população visando prevenir a convergência prematura do AG para soluções sub-ótimas.

A taxa de aplicação deste operador é usualmente muito baixa, sendo em geral empregada com menos de 1% de probabilidade. A literatura clássica de algoritmos genéticos considera o operador de *crossover* como sendo o principal no mecanismo de funcionamento de AGs, ao passo que o operador de mutação teria apenas um caráter secundário (GOLDBERG, 1989). A mutação pode ocorrer de diversas formas, de acordo com o operador aplicado. A seguir são apresentados os operadores EM, DM, SIM, ISM, IDM conforme descritos em Larraña (LARRAÑA *et all,* 1998).

3.4.5.1 Exchange Mutation (EM)

Este operador consiste em selecionar aleatoriamente dois genes no cromossomo e então permutar suas posições, como mostra a Figura 5.

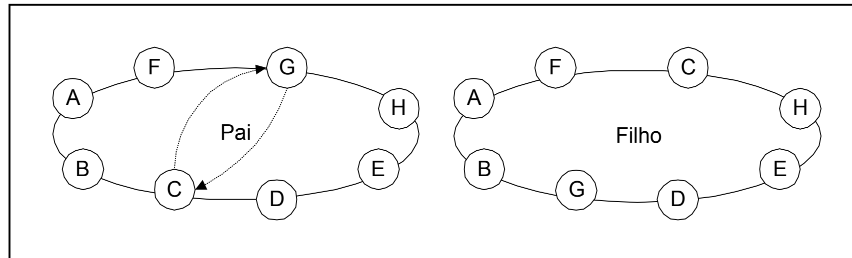


Figura 5 – Aplicação do Operador – *Exchange Mutation* – EM

3.4.5.2 Displacement Mutation (DM)

O operador *displacement mutation* Michalewicz (1992), consiste em remover um subconjunto de genes do cromossomo e depois inseri-los novamente na seqüência, em uma nova posição aleatória. A Figura 6 ilustra o processo.

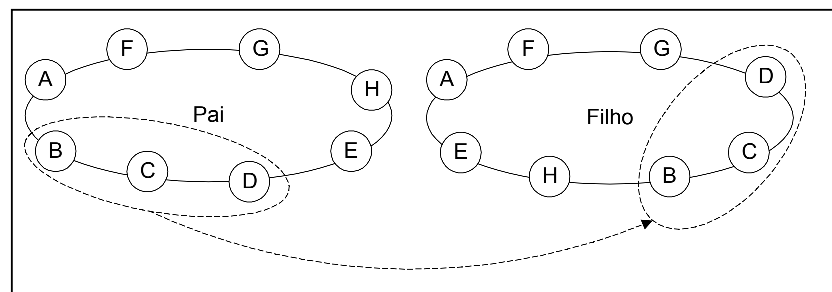


Figura 6 – Aplicação do Operador – *Displacement Mutation* – DM

3.4.5.3 Simple Inversion Mutation (SIM)

A aplicação deste operador, consiste em selecionar dois pontos aleatórios na seqüência de genes e então inverter a subseqüência permutando os genes entre os dois pontos, como demonstrado na Figura 7.

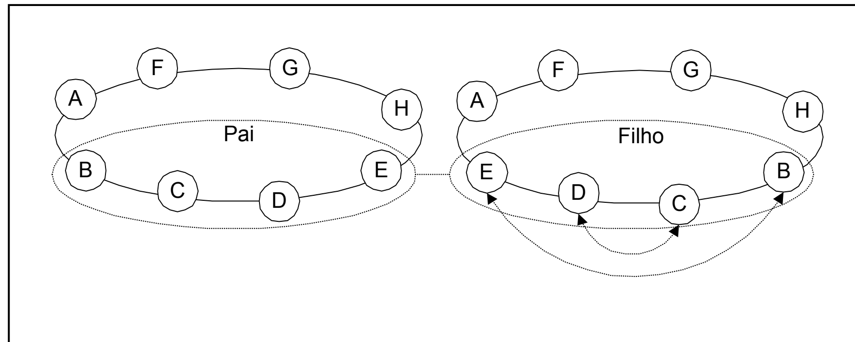


Figura 7 – Aplicação do operador –*Simple Inversion Mutation* – SIM

3.4.5.4 Insertion Simple Mutation (ISM)

O operador *insertion simple mutation* consiste em selecionar um gene aleatório e sortear uma nova posição para o mesmo no cromossomo. A aplicação do operador ISM é demonstrada na Figura 8.

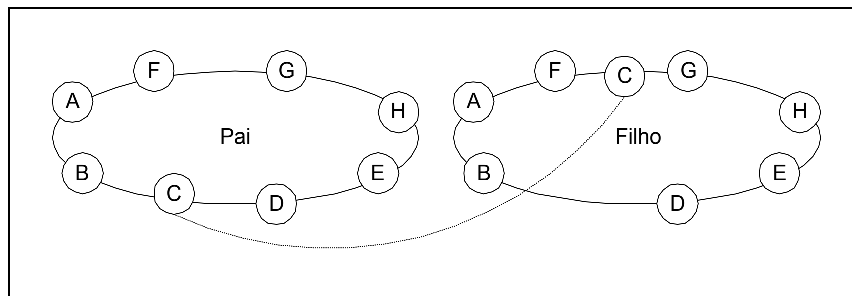


Figura 8 – Aplicação do operador – *Insertion Simple Mutation* – ISM

3.4.5.5 Inversion Displacement Mutation (IDM)

O operador IDM é similar ao operador DM, sua aplicação consiste em selecionar uma subseqüência de genes do cromossomo, remover a subseqüência e então inserir novamente em uma nova posição. Assim como no operador DM, a nova posição é sorteada de forma aleatória, no entanto o conjunto de genes é inserido em ordem inversa, conforme mostra a Figura 9.

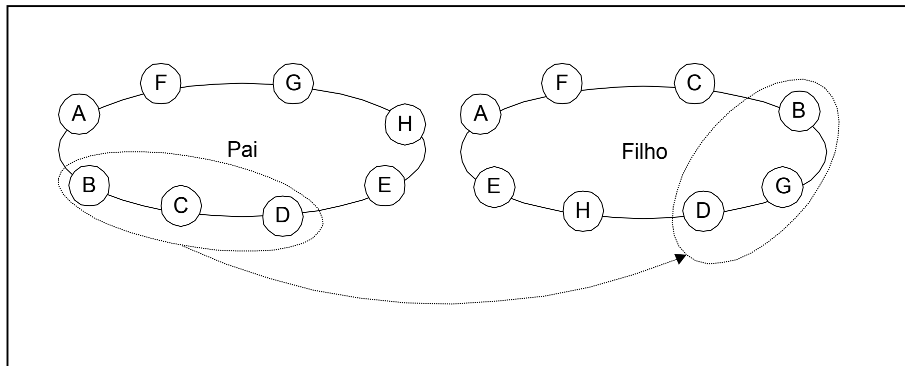


Figura 9 – Aplicação do operador – *Inversion Displacement Mutation* – IDM

3.4.6 O Algoritmo Genético

O ponto inicial para fazer uso de algoritmos genéticos, como mecanismo de solução de um dado problema, é definir uma representação genética para as possíveis soluções do problema. Após a definição da representação genética, o algoritmo genético, apresenta basicamente os passos da Figura 10.

O 1º passo é gerar a população inicial, um conjunto de indivíduos gerados aleatoriamente.

No 2º passo é aplicada a função de avaliação, para determinar a aptidão de cada indivíduo no espaço de soluções do problema.

O 3º passo consiste na aplicação do operador de seleção, este determina quais indivíduos irão servir como reprodutores da nova população e quais serão eliminados.

No 4º passo são aplicados os operadores *crossover* e *mutação*. A aplicação desses operadores produz uma nova população.

No 5º passo, realiza-se a checagem do(s) critério (s) de parada. O ciclo evolutivo será encerrado, se pelo menos um dos critérios estabelecidos tiver sido atingido, caso contrário todo processo é repetido a partir do 2º passo.

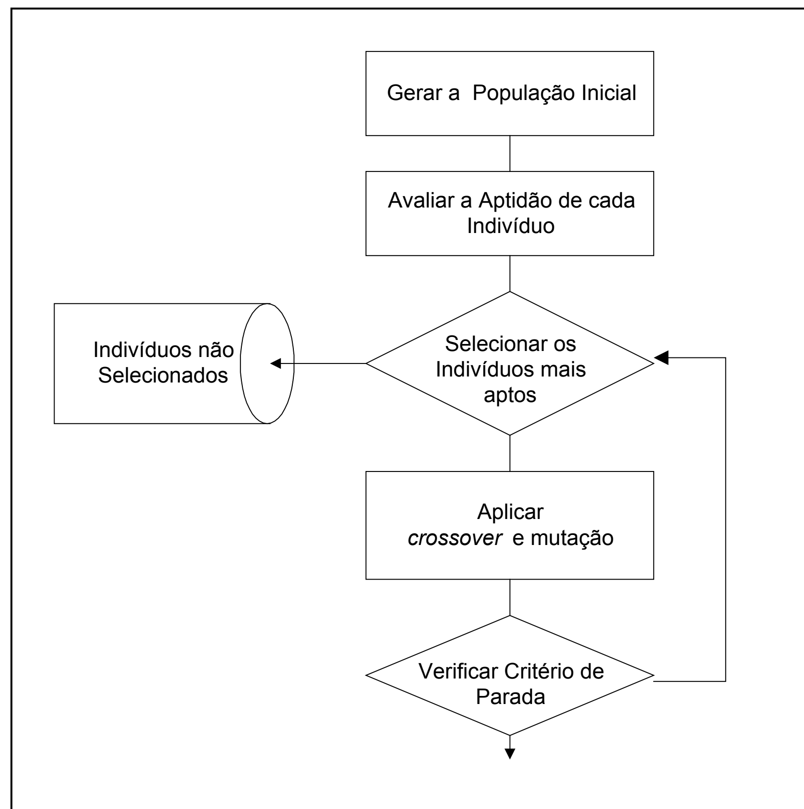


Figura 10 – Fluxograma genérico dos Algoritmos Genéticos

Uma das vantagens dos AGs é o fato de não ser necessário muito conhecimento a respeito do problema. A utilização dos métodos tradicionais de otimização, implica na necessidade de conhecimentos específicos sobre o problema e a solução do mesmo. Em muitos casos, como no PCV, o problema é plenamente entendido, no entanto não se tem o conhecimento de uma estratégia tal que aplicada a uma instância do mesmo, produza a respectiva solução ótima. Já no caso do algoritmo genético basta definir um item genérico através de um conjunto de parâmetros, composto de respostas a características específicas pré-determinadas, criar alguns exemplares deste item e testá-los uns contra os outros até que gerem itens satisfatórios, sendo que esse processo pode levar de alguns minutos a algumas horas ou mesmo, alguns dias (PRICE, 1994).

3.4.7 O Problema da Convergência Prematura

Algoritmos genéticos padecem do problema da convergência prematura. Para resolver esse problema, muitos algoritmos genéticos forçam a diversificação explicitamente, violando a

metáfora biológica. Um método muito popular é aceitar somente descendentes com um grau de diferença maior que um determinado fator verificado sobre os demais membros da população. Segundo Miki, (MIKI *et all,* 1999), a convergência prematura pode ser evitada através de boas meta-GA estratégias, porem não poderá ser evitada quando a convergência prematura é causa da própria meta-estratégia que esta ruim.

“A meta-estratégia de um AGP é dirigida pelos seus três componentes – A estrutura de distribuição espacial da população, o operador *crossover* e estratégias de busca local.” (Mühlembein, 2001)

A estratégia de busca de um algoritmo genético pode ser explicada em termos simples. O operador *crossover* produz um espalhamento da busca para novos pontos fora da área definida pela população anterior. O *crossover* implementa um passo adaptativo controlado, porém explorando o espaço de busca continuamente. Já o operador de mutação pode provocar saltos para outras regiões do espaço, sendo útil como mecanismo de escape de mínimos locais.

Problemas combinatoriais como o PCV, contém uma característica de blocos de construção. Uma solução otimizada é produzida pela junção de sub cadeias das soluções anteriores “pais”, e é justamente isso que o operador *crossover* faz. (Mühlembein, 2001).

4 Computação Paralela e Concorrente

4.1 Introdução

De uma forma geral pode-se dizer que a idéia básica atrás de paralelismo é: empregar muitos recursos para resolver um problema em particular. Para isso, é necessário dividir o problema em subproblemas de menor complexidade, alocando os recursos necessários à resolução dos subproblemas, e posteriormente combinando as soluções parciais em uma solução global. Os subproblemas podem ser independentes ou sobreporem-se, dependendo da situação, uma técnica diferente deve ser empregada. (METAXAS, 1995)

A computação concorrente é uma variação do paralelismo real. Na computação concorrente, processos diversos concorrem entre si pelos recursos computacionais. A programação concorrente pode ser utilizada tanto em máquinas com múltiplos processadores como em máquinas mono processadas.

4.2 Computação Paralela

A computação paralela consiste, basicamente, em elementos de processamento, que cooperam e comunicam-se entre si para solucionarem problemas mais complexos, de maneira mais rápida do que se estivessem sendo solucionados seqüencialmente (ALMASI & GOTTLIEB, 1994). Esta definição explica claramente os fatores que motivam o direcionamento de esforços no sentido de desenvolver mecanismos de paralelização de processos computacionais em geral. Por outro lado, o paralelismo também ocasiona o surgimento de uma série de novas características em relação ao gerenciamento e manutenção da coerência das informações processadas, essas características são próprias da computação paralela e geralmente não são encontradas em sistemas de computação seqüencial (SANTOS, 2001).

Dividir e conquistar, é um dos paradigmas fundamentais da computação seqüencial, e num outro modo de ver, é inerentemente paralelo. Nesta técnica, os subproblemas são independentes e podem se resolvidos sem comunicação entre os processos. *"Divide-and-conquer* é um paradigma natural para algoritmos paralelos. Depois de dividir o problema em dois ou mais subproblemas, os subproblemas podem ser resolvidos em paralelo" (BLELLOCH & MAGGS, 1996). Esse paradigma fornece o norte para a implementação de algoritmos paralelos.

Diversas classificações de paralelismo podem ser facilmente encontradas na literatura, dentre elas tem-se: (SANTOS, 2001)

Paralelismo de Dados, onde o processador executa as mesmas instruções sobre dados diferentes;

Paralelismo Funcional, onde o processador executa diferentes instruções que podem ou não operar sobre o mesmo conjunto de dados;

Paralelismo de Objetos, que utiliza o conceito de objetos distribuídos por uma rede, capazes de serem acessados por métodos em diferentes processadores, para uma determinada finalidade.

4.2.1 Programa Paralelo

Um programa paralelo é uma coleção de módulos separados, denominados processos, que se comunicam e cooperam para a execução de uma determinada tarefa. A carga total de processamento necessária à execução da tarefa é distribuída no sistema, sobre seus N processadores. Uma vez que a cada processador é dado para executar um ou mais processos, e que cada processo não é necessariamente do mesmo tamanho. A estratégia de mapeamento da distribuição de carga sobre o sistema multiprocessado, influi na eficiência do uso do poder computacional e no alcance do objetivo final. (ALAOUI, 2000). Por outro lado, um ambiente de execução paralela é formado por uma ou mais máquinas conectadas por uma rede física que fornece o suporte de execução aos programas paralelos (PREUS, 1998). O ambiente, portanto, é responsável pela implementação dos mecanismos que permitem a comunicação, sincronização, gerência de processos, dentre outros.

Em computação distribuída, o algoritmo a ser computado é dividido em pequenos sub-algoritmos e cada um desses pedaços é associado para ser executado em um processador separado, quando possível (MADISSETI *et all,* 1991).

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 (TANNEMBAUM, 1991). 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 minimização do tempo de processamento, ou mesmo na busca de uma estruturação melhor - ou mais tolerante a falhas - de um sistema.

4.2.2 Arquiteturas Paralelas

Computação paralela ou concorrente pode ser realizada de diferentes formas em um sistema computadorizado. Segundo Flynn, (FLYNN, 1996) 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 11, apresenta um único fluxo de instruções e de dados. Essa categoria compreende as máquinas de *Von Newmann,* amplamente utilizadas, as tradicionais máquinas mono -processadas com execução seqüencial.

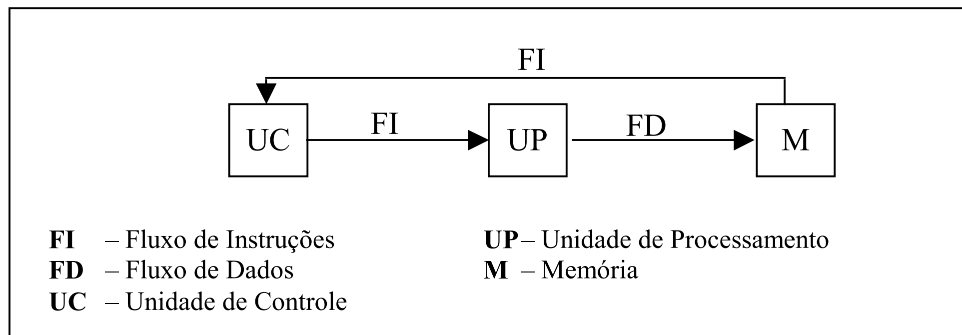


Figura 11 – Diagrama básico de uma máquina SISD

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 12. 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, máquinas matriciais.

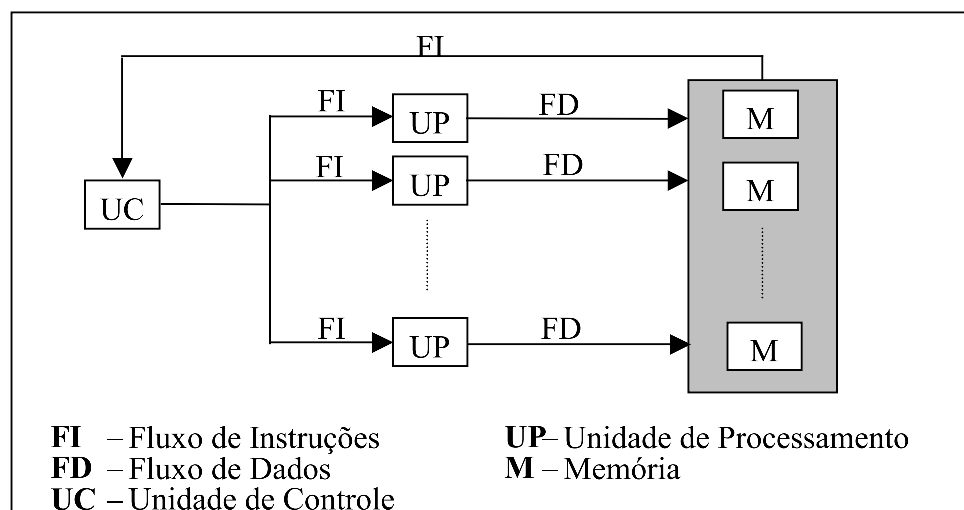


Figura 12 – Diagrama básico de uma máquina SIMD

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 13. Embora não existam muitos exemplos precisos de máquinas MISD na literatura, (ALMASI & GOTTLIEB, 1994) consideram o *pipeline* como representante dessa categoria.

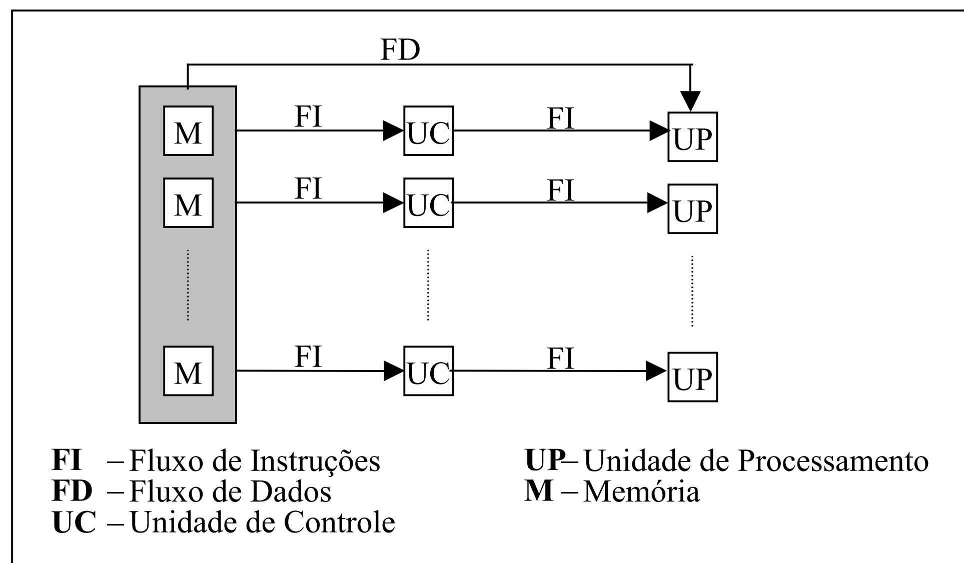


Figura 13 – Diagrama básico de uma máquina MISD

MIMD (Multiple Instruction, Multiple Data Stream). A categoria MIMD, apresenta múltiplos fluxos de instruções atuando em múltiplos fluxos de dados, Figura 14. 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.

Cada uma destas combinações caracteriza uma classe de arquiteturas 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 (ALMASI & GOTTLIEB, 1994).

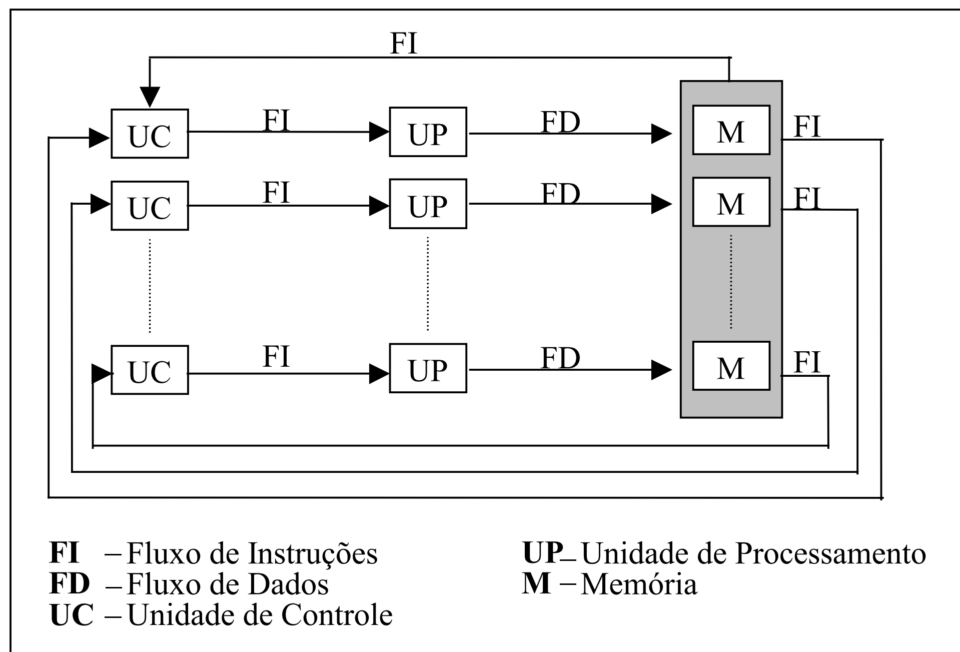


Figura 14 – Diagrama básico de uma máquina MIMD

4.3 Computação Concorrente

O termo *concorrente* refere-se à potencialidade de execução paralela de partes de uma computação. Em uma computação concorrente, os componentes de um programa podem ser executados seqüencialmente ou em paralelo. A concorrência existe quando, em um determinado instante, dois ou mais processos (executando em um ou mais processadores) começaram a sua execução, mas, devido à disputa pela utilização do processador e/ou outros recursos do sistema, ainda não terminaram. (ALMASI & GOTTLIEB, 1994).

A programação concorrente é uma parte fundamental no apoio à construção de programas paralelos, pelo fato de oferecer recursos que podem ser utilizados visando obter melhor desempenho e melhor utilização do hardware paralelo disponível, (QUINN, 1994).

Computação concorrente proporciona a flexibilidade de intercalar a execução dos componentes de um programa, sobre um único processador, ou distribui-los sobre vários processadores. A concorrência abstrai alguns detalhes de uma execução, permitindo que o

programador se concentre na parte conceitual do problema, sem precisar se preocupar com uma ordem particular de execução, (AGHA, 1990).

O corrente desenvolvimento da programação objeto orientada concorrente (COOP) – *concurrent object-oriented programing* – está promovendo uma sólida base de software para computação concorrente sobre máquinas multiprocessadas. As futuras gerações de sistemas provavelmente serão baseadas nos fundamentos desenvolvidos por essa tecnologia de software (AGHA, 1990).

4.4 Construção de Programas Paralelos

Quinn (QUINN, 1994) sugere três maneiras de se construir um algoritmo paralelo:

1. Detectar e explorar algum paralelismo inerente a um algoritmo seqüencial existente.
2. Criar um algoritmo paralelo novo.
3. Adaptar outro algoritmo paralelo que resolva um problema similar.

Ainda segundo Quinn, a primeira opção, embora muito utilizada, não produz bons resultados em termos de eficiência. A segunda opção possui o inconveniente de requerer o projeto de um algoritmo totalmente novo. A terceira opção exige menos trabalho e em geral a solução final apresenta boa performance.

De uma forma geral, pode-se argumentar que a programação concorrente é regida pela idéia de dividir determinadas aplicações em partes menores e que cada parcela resolva uma porção do problema. Assim, a programação concorrente pode ser considerada como outra manifestação do paradigma "dividir e conquistar". Entretanto, há a necessidade de recursos adicionais de ativação, comunicação e sincronização não necessárias na programação seqüencial.

Os recursos de ativação são responsáveis pela inicialização e término de processos concorrentes. Algumas linguagens de programação disponibilizam recursos para a execução

de processos concorrentes. A linguagem JAVA, por exemplo, disponibiliza o recurso de multithreads, que permite o encadeamento múltiplo de um programa executado em máquinas mono ou multiprocessadas, obtendo-se dessa forma a concorrência.

Outro ponto importante em programação concorrente é a coordenação e especificação da cooperação entre os diversos processos. O estabelecimento de um mecanismo de comunicação possibilita a um processo em execução comunicar-se com outro. Essa comunicação pode se dar através do compartilhamento de variáveis em memória (sistema com memória compartilhada) ou através de mensagens (sistema distribuído).

Segundo Preus (PREUS, 1998), o paralelismo em um programa pode ser explorado de duas formas: implicitamente através de compiladores paralelizantes. Neste caso o compilador encarrega-se da tarefa de descobrir trechos de programa que podem ser executados em paralelo automaticamente; explicitamente, através de primitivas inseridas no programa, ou através de uma linguagem de programação, com suporte a concorrência.

Ainda segundo Preus, a programação paralela é significativamente mais difícil do que a seqüencial, principalmente porque envolve a necessidade de sincronização entre tarefas, como também a análise da dependência de dados. A utilização de um sistema paralelizante minimiza estas dificuldades, e permite também o reaproveitamento de programas seqüenciais já implementados. Por outro lado, o paralelismo explícito permite que fontes não passíveis de detecção por um sistema paralelizante possam ser exploradas.

O paralelismo expresso pelo usuário pode ser especificado em um programa através de comandos específicos de uma linguagem de programação paralela, por chamadas a rotinas de uma biblioteca que gerencia os recursos de paralelismo da máquina ou pelo uso de diretivas do compilador.

Entre estas formas, a utilização de uma linguagem de programação paralela é a que oferece um ambiente mais adequado de programação. Nesse contexto linguagens como JAVA, que oferecem suporte ao conceito de *thread*, Figura 15, são excelentes candidatas. Um *thread* compreende três partes principais: 1- a CPU virtual, 2- o código que a CPU está executando e 3- os dados sobre os quais o código trabalha. (SUN, 1997). Um *thread* pode executar um código que seja igual ao de outro *thread* ou pode executar um código diferente. O mesmo é válido também para os dados.

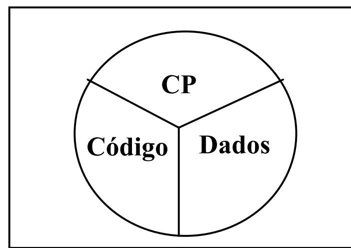


Figura 15 – Um *Thread* - Contexto de Execução

O uso de bibliotecas específicas, também tem sido uma forma muito utilizada de implementar programas concorrentes. Atualmente, existem várias bibliotecas específicas para multiprocessadores como PVM - Parallel Virtual Machine, MPI - Message Passing Interface (PREUS, 1998).

Um dos aspectos mais importantes da paralelização de programas está relacionado com as operações de gerenciamento de processos, criação, comunicação, sincronização e o término dos mesmos.

4.4.1 Criação e Término de Processos

Em um programa paralelo, os processos podem ser criados de maneira implícita ou explícita (PREUS, 1998). Na criação implícita os processos são associados a um tipo de dado e são instanciados com a declaração de uma variável desse tipo no programa. A criação explícita de processos é realizada por uma função, designada especialmente para realizar esta tarefa.

A criação de processos também pode ser estática ou dinâmica. Na criação estática, o conjunto de processos que irão compor o programa paralelo deve ser definido antes da execução, e não pode ser alterado até o final do programa. Este tipo de criação apresenta a desvantagem de exigir que o número total de processos seja conhecido antecipadamente. Já na criação dinâmica, os processos podem ser instanciados a qualquer momento durante a execução do programa sendo, portanto, mais flexível que a criação estática.

O término de processos é uma questão importante e ligada diretamente à criação. Quando um processo termina, podem ocorrer dois tipos de situação: 1- o término do processo causa

automaticamente a morte de todos os seus processos filhos; 2- o processo fica bloqueado, aguardando que os processos filhos também terminem.

4.4.2 Sincronização de Processos

A sincronização de processos pode ser efetivada através do uso de semáforos. Um semáforo consiste de um objeto com duas primitivas básicas *signal* e *wait* um processo esperando por outro, executa a primitiva *wait*, nesse ponto o processo entra em estado de espera até que outro processo emita uma primitiva *signal*. Enquanto está em estado de espera, o processo não faz uso do processador.

4.5 ALGORITMOS GENÉTICOS PARALELOS (PGAs)

Como exposto no capitulo 3, Algoritmos genéticos (GAs) são algoritmos de pesquisa inspirados na teoria da seleção natural ou teoria da evolução. As mais importantes fases de um AG são: reprodução, mutação, avaliação de aptidão e seleção. Reprodução é o processo pelo qual o material genético de dois ou mais parentes são combinados (*crossover*) para obter descendentes. O operador de mutação é normalmente aplicado a um indivíduo para produzir uma nova versão desse mesmo indivíduo, o material genético do indivíduo é aleatoriamente re-locado. A avaliação de aptidão é o processo no qual a qualidade do indivíduo é quantificada em relação a sua contribuição para o alcance do objetivo. O processo de avaliação geralmente é o passo que mais consome recursos computacionais. (NOWOSTAWSKI & POLI, 1999). Seleção é uma operação utilizada para decidir quais os indivíduos serão aceitos e quais serão descartados, eliminados do processo de reprodução e mutação sendo extremamente centralizada.

“Algoritmos genéticos seqüenciais tem se mostrado verdadeiramente bons em muitas aplicações em diferentes domínios. Entretanto ainda existem alguns problemas na sua

utilização que podem ser resolvidos com alguma forma de paralelismo" (NOWOSTAWSKI & POLI, 1999).

Existem algumas razões práticas para a pesquisa de mecanismos de paralelização da execução em algoritmos genéticos. (NOWOSTAWSKI & POLI, 1999), (LOHN *et all*, 2000)

- ✓ Para alguns conjuntos de problemas, a população precisa ser muito grande e a memória requerida para armazenar cada indivíduo pode ser considerável. Em alguns casos isso torna impossível rodar uma aplicação eficientemente usando uma única máquina, nesses casos faz-se necessário um GA paralelo.

- ✓ O processo de avaliação da aptidão, geralmente consome uma fatia considerável do tempo computacional. Na literatura podem ser encontrados relatos de tempos computacionais superiores a 1 ano de CPU para rodar aplicações em domínios complexos. Para aplicações dessa magnitude, o único caminho para prover mais tempo computacional é o uso de processamento paralelo.

- ✓ Algoritmos genéticos seqüenciais podem ficar presos em uma região sub ótima do espaço de busca ficando assim impossibilitado de encontrar melhores soluções. PGAs podem pesquisar diferentes sub-espaços do espaço de busca em paralelo, reduzindo a influencia de sub-espaços ruins.

As primeiras duas razões sugerem o estudo de PGAs, para uso em aplicações executando sobre máquinas maciçamente paralelas e também sobre sistemas distribuídos. Entretanto a mais importante vantagem de PGAs é que em muitos casos, eles proporcionam melhor performance que um algoritmo baseado em uma população simples. A razão é que múltiplas populações permitem especificar um processo no qual as diferentes populações evoluem em diferentes direções, ou seja, diversas formas de atingir a solução ótima. Por esta razão GAs Paralelos não podem ser vistos como uma simples extensão do modelo tradicional de Algoritmo Genético seqüencial. PGAs definem uma nova classe de algoritmos de busca (NOWOSTAWSKI & POLI, 1999).

De uma forma geral, Algoritmos Genéticos Paralelos podem ser categorizados em três tipos: *standard*, *coarse grained* e *fine grained* (padrão, granularidade grossa e granularidade fina) (TURTON *et all*, 1994).

Standard - GAs podem ser implementados em uma arquitetura paralela, pela distribuição do processo de avaliação sobre um conjunto de processadores.

Coarse grained - Várias populações rodam em paralelo. Após um certo número de gerações cada população exporta um conjunto de indivíduos que migram para as populações vizinhas. Normalmente um processo central controla as populações e o processo de migração, efetivando a troca de indivíduos entre as populações. A vantagem do esquema *coarse grained*, aparece quando é permitido a várias populações compartilharem indivíduos. Se as múltiplas populações permanecerem independentes, então esta técnica se torna equivalente à execução de múltiplas instâncias GAs *standard*.

Fine grained – A população é dividida sobre um conjunto de processadores. Cada processador atua sobre um ou mais indivíduos da mesma população. O operador *crossover* é executado tomando-se o vizinho mais próximo.

Um algoritmo genético é uma pesquisa paralela com controle centralizado. A parte centralizada é a agenda da seleção. O mecanismo de seleção por sua vez, necessita da média de aptidão entre todos os indivíduos, o resultado é um algoritmo altamente sincronizado, o que dificulta a implementação eficiente em computadores paralelos (MÜHLENBEIN, 2000).

5 Método desenvolvido

Este capítulo inicia com uma breve apresentação do problema do caixeiro viajante – PCV – usado como corpo de prova ao método proposto. Em seguida são apresentados alguns trabalhos relacionados. O restante do capítulo descreve a estrutura do algoritmo genético paralelo proposto e o protótipo implementado.

5.1 O Problema do Caixeiro Viajante

O problema do caixeiro viajante (PCV) ou TSP (*Traveling Salesman Problem*), pode ser enunciado como: Dado um número finito de "cidades" junto com o custo de viagem entre cada par delas, achar o caminho mais barato de visitar todas as cidades exatamente uma vez e voltando ao local de partida.

Em termos da teoria dos grafos, as cidades são representadas pelos vértices e os custos das viagens são pesos associados aos arcos que determinam o grafo. O grafo assim constituído é um grafo completo[5] finito. O problema passa então a ser: encontrar um ciclo de Hamilton[6]

[5] Um grafo G é dito completo se para todo vértice V de G existe um arco partindo de V para todo e qualquer outro vértice V' pertencente a G

[6] Um ciclo de Hamilton consiste em um caminho cíclico que passa por todos os vértices do grafo, passando uma única vez em cada vértice

com peso total mínimo. Assim o PCV pode ser visto como uma versão em linguagem não matemática do problema de encontrar um ciclo de Hamilton em um grafo completo, cujo peso seja mínimo.

Neste trabalho foi assumido que os custos de viagem entre duas cidades A e B são simétricos, de tal modo que viajando da cidade A para cidade B vale o mesmo que viajando de B até A. A bateria de testes foi realizada com instâncias publicas coletadas na internet[7]. Tendo sido escolhidas as instancias: ATT48, BERLIN52, KROC100, EIL101, CH150, GR202, TS225, A280, PCB442, GR666, PR1002. A escolha foi aleatória tomando-se apenas a precaução de escolher instancias de tamanhos crescentes.

5.2 Trabalhos Relacionados

5.2.1 Algoritmo de Lohn

Lohn, (LOHN *et all*, 2000), propõem uma estrutura de *mater-slave* (controlador – escravo), na qual o controlador é responsável pela manutenção da população e execução dos operadores genéticos. A avaliação de aptidão dos indivíduos é distribuída para os demais nós "escravos". O controlador envia um indivíduo a cada um dos nós subjacentes, "escravos". Estes por sua vez realizam a decodificação/interpretação dos indivíduos segundo a representação do problema e então aplicam a função de cálculo da aptidão do indivíduo. Posteriormente, ao terminar a avaliação, cada nó escravo envia seus resultados ao controlador que executará a seleção dos indivíduos e a geração da nova população. O processo então se repete como um todo. A Figura 16 ilustra o mecanismo proposto.

[7] Site - http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/tsp/ disponível em 29-04-2002.

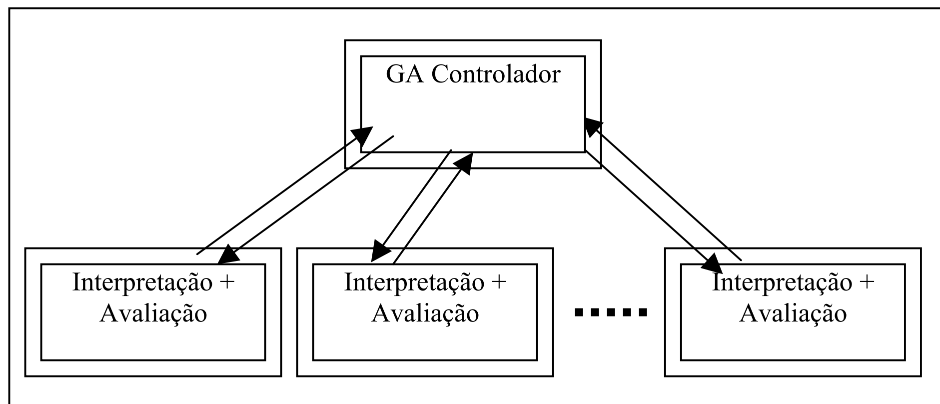


Figura 16 – AG Paralelo – *Master - Slave*

5.2.2 Algoritmo de Mühlembein

Mühlembein, (MÜHLENBEIN, 2000), propõem um método de paralelização para algoritmos genéticos, onde cada indivíduo é o próprio responsável por encontrar um parceiro "casamento" em sua vizinhança, bem como sua reprodução. Dessa forma os indivíduos são elementos ativos no processo evolutivo. O controle de seleção e reprodução se dispersa pela população e o algoritmo deixa de ser centralizado diminuindo o grau de sincronia e facilitando a paralelização. "GAs com múltiplas sub-populações relativamente isoladas, podem manter uma maior diversidade de indivíduos nas populações envolvidas" (MIKI *et all*,1999). Esse modelo parece adaptar-se extremamente bem ao paradigma de programação objeto orientado.O algoritmo sugerido está transcrito abaixo

Algoritmo Genético Paralelo (AGP) Genérico de Mühlembein

Passo0- Definir uma representação genética para o problema

Passo1 – Criar uma população inicial e sua estrutura de vizinhança

Passo2 – Cada indivíduo faz uma busca local "*hill-climbing*".

Passo3 – Cada indivíduo seleciona um parceiro, em sua vizinhança, para casar.

Passo4 - Uma geração descendente é criada com o operador genético *crossover* dos pais.

Passo5 – A nova geração faz uma busca local "*hill-climbing*" e substitui os pais, se forem melhores, segundo algum critério de aceitação.

Passo6 – Se não terminou, retorna ao passo3.

5.2.3 Algoritmo de Alaoui

Alaoui, (ALAOUI *et all,* 2000), propõem um modelo *master-slave* no qual cada um dos nós subjacentes "escravos" é responsável por uma população inteira. O nó central "mestre" cria as populações iniciais e as distribui para os nós escravos. Cada nó subjacente processa a evolução da população por um determinado número de gerações e então a submete ao mestre. Nesse estágio o mestre seleciona os melhores indivíduos dentre todas as sub populações e os distribui novamente aos nós escravos. Em cada nó subjacente, o novo grupo de indivíduos será inserido na população corrente e o processo de evolução recomeça. O processo de migração, controlado pelo nó mestre, implementa o mecanismo que regula a velocidade da convergência e oferece os meios de escape de mínimos locais. Entretanto como relatado, a migração das populações dos nós subjacentes para o mestre e vice versa, pode impor um certo grau de *overhead* dependente do meio de comunicação entre os nós.

5.2.4 Discussão

O método proposto por Lohn não é eficaz no sentido de que explora apenas o paralelismo da avaliação dos indivíduos, ou seja, o cálculo de sua aptidão. Esse modelo peca ao não explorar o paralelismo da reprodução e da mutação.

O modelo proposto por Mühlembein sugere a possibilidade de se empregar vários métodos de busca local em indivíduos de uma mesma população. Indivíduos parentes executando métodos de busca local diferentes seriam úteis nos casos em que a eficiência dos métodos de busca se mostrar dependente da instância do problema. Por outro lado Mühlembein explica que a troca de informações entre as populações realiza-se pela sobreposição das vizinhanças.

O modelo de Alaoui apresenta-se com boas perspectivas, no sentido de que proporciona uma divisão em populações de pequeno ou médio porte. No entanto o método está preso ao controle do nó central o que diminui sua flexibilidade. Alem disso como descrito acima o processo de seleção é centralizado, demandando grande capacidade de processamento e armazenamento no nó central. Por outro lado não há menção sobre a exploração do paralelismo em máquinas dotadas de múltiplos processadores.

5.3 Estrutura do AG Paralelo Implementado

A estrutura do algoritmo proposto segue em parte a proposta de Alaoui, (ALAOUI *et all,*2000). Nesta, como descrito em 5.2.3, um nó supervisor cria e distribui as populações para nós escravos. Cada nó escravo processa a evolução de uma população e então a submete ao supervisor, este por sua vez escolhe os melhores representantes e os redistribui novamente para um novo ciclo evolutivo.

A estrutura de Alaoui serviu de inspiração, entretanto a estrutura proposta neste trabalho apresenta uma estrutura *coarse-grained,* como discutido em 4.5. A estrutura proposta apresenta um mestre cujas funções se restringem a:

1- Criar as populações distribuindo a cada uma delas o conjunto de genes codificados e um conjunto de parâmetros iniciais. (número máximo e mínimo de indivíduos, operadores de *crossover* e mutação*,* periodicidade de envio e aceite de indivíduos migrantes, desvio padrão a ser considerado como mínimo na detecção de estagnação).

2- Servir de Intermediário na migração de indivíduos entre as populações. O supervisor propicia uma sala de espera onde os indivíduos migrantes permanecem até serem requisitados por uma outra população que não a de origem, ou até serem substituídos por novos migrantes mais aptos. Apenas um migrante de cada população é mantido.

As populações por sua vez, caracterizam-se por serem elementos independentes. Ao ser inicializada, a população gera o conjunto inicial de indivíduos tomando como base o conjunto de genes recebidos do supervisor. Após a formação inicial, a população executa o algoritmo genético aplicando os operadores de *crossover*, mutação e seleção em ciclos de evolução. A aplicação dos operadores de *crossover* e mutação é um processo independente entre as populações, desse diferentes populações podem aplicar diferentes operadores.

Os indivíduos são elementos ativos sob a coordenação da população e são responsáveis pela efetivação da aplicação dos operadores de *crossover* e mutação. Esse mecanismo permite que operadores distintos sejam aplicados dentro da mesma população. Entretanto ao contrário da proposta de Mühlembein (MÜHLENBEIN, 2000), os indivíduos não têm a capacidade de encontrar seus parceiros, a tarefa de seleção dos casais é realizada pela população. O indivíduo também é responsável pela quantificação de sua aptidão em relação à solução do problema em questão. Essas características dos indivíduos tornam o processo descentralizado como um todo o que amplia as possibilidades de exploração do paralelismo.

5.3.1 Representação Adotada

A representação adotada toma como base um caminho cíclico no qual estão dispostas todas as referências às cidades que compõem a instância. Os operadores de *crossover* e mutação atuam diretamente sobre as referências, alterando a respectiva seqüência. Assim o espaço de busca é o conjunto de todas as combinações possíveis sem repetição, observado-se que, por exemplo, (1, 2, 3) é o mesmo que (3, 1, 2) ou que (2, 3, 1) . Matematicamente, sendo N o número de cidades, o número de pontos P no espaço de busca S é dado por: $P_S = (N-1)!/2$.

A Figura 17 ilustra a estrutura de dados utilizada. As cidades são mantidas em um vetor, enquanto que cada solução candidata é composta por uma lista circular cuja ordem dos nodos representa a seqüência de percurso.

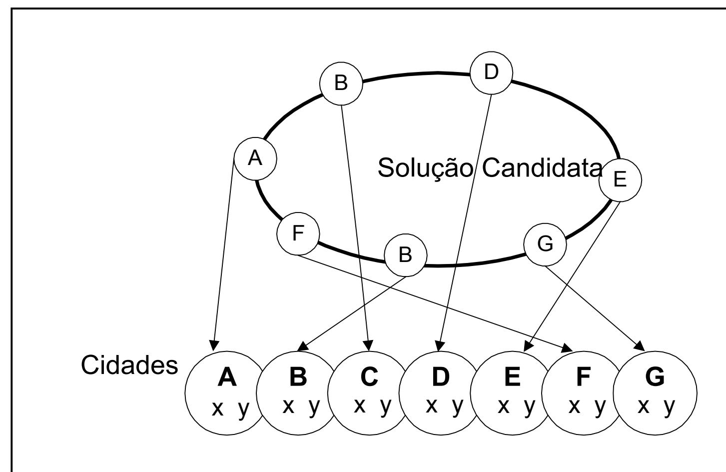


Figura 17 – Representação adotada

5.3.2 Função de Avaliação da Aptidão

Especificamente, no problema do caixeiro viajante, a forma mais simples de quantificar a aptidão de cada indivíduo consiste numa função que calcule o custo total de cada solução candidata. Esse custo foi tomado como o somatório das distâncias, em linha reta, calculadas segundo as coordenadas de cada cidade. Considerando que a cada cidade correspondem duas coordenadas num plano cartesiano, a distância entre duas cidades A e B é dada pela Equação 1 Como o objetivo é minimizar esse custo, inverteu-se o valor final de forma que quanto menor o custo maior a aptidão, ou seja, $aptidão_i = 1/custo_i$.

$$d = \sqrt[2]{(A.x - B.x)^2 + (A.y - B.y)^2}$$

Equação 1 – Cálculo da distância entre duas cidades

5.3.3 Seleção para Reprodução

A seleção dos pares para reprodução através da aplicação do operador de *crossover* segue o sistema de alvo, Figura 18, descrito por Goldberg (GOLDBERG, 1989) sobre o qual são lançados dardos, os dardos são números aleatórios gerados entre zero e a soma total das aptidões.

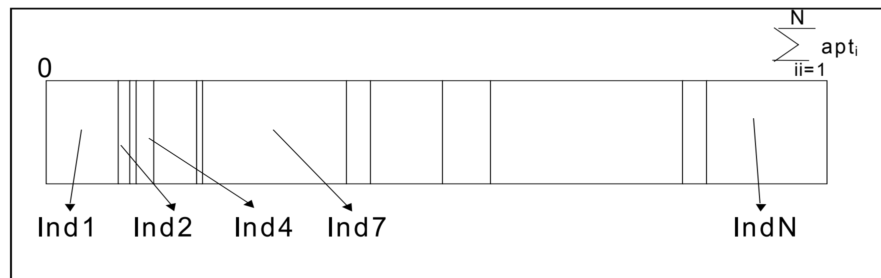


Figura 18 – Intervalo para seleção aleatória dos pares

No alvo representado na Figura 18, a área de cada região é proporcional à expectativa de cada indivíduo, ou seja, indivíduos mais aptos possuem uma faixa maior e, por conseguinte uma chance maior de ser selecionado.

5.3.4 Troca de Informação entre as Populações

A troca de material genético entre as populações é realizada através de indivíduos que migram de uma população para outra. Um parâmetro determina o intervalo de envio, enquanto outro determina o intervalo de aceite de indivíduos migrantes. Os indivíduos migrantes são clones dos melhores indivíduos de cada população. Esses indivíduos migrantes são enviados pelas populações ao mestre que os mantém em espera. Quando uma população solicita um indivíduo reprodutor, o mestre escolhe aleatoriamente dentre os migrantes, um indivíduo originário de qualquer uma das demais populações. Esse mecanismo de troca de indivíduos busca acelerar a convergência disseminando as características dos melhores indivíduos no conjunto de populações, ao mesmo tempo em que tenta evitar a convergência para um mínimo local.

5.3.5 Controle da Estagnação

O parâmetro de controle da detecção de estagnação estabelece o desvio padrão mínimo a ser considerado como ponto de estagnação. Quando ocorre uma estagnação, a evolução está presa num mínimo local e as diferenças entre os indivíduos são mínimas. Como um segundo mecanismo de escape de mínimos locais, a população é submetida a uma redução até o

mínimo e posterior crescimento por mutação e sem seleção até atingir o número máximo de indivíduos. O processo de redução garante apenas a sobrevivência do melhor individuo. Os demais indivíduos são selecionados de acordo com seu desvio padrão dando preferência à diversidade conforme as meta-GA estratégias sugeridas por Miki, (MIKI *et all,* 1999). O crescimento que segue a redução é obtido através da geração de indivíduos mutantes, espera-se que estes apresentem características totalmente novas, dando novos rumos ao processo evolutivo.

5.3.6 Processo de Seleção

O processo de seleção reduz a população a um tamanho igual ao número máximo de indivíduos permitido. Nesse processo os indivíduos menos aptos são eliminados em favor dos mais aptos que sobrevivem, o que não significa que irão reproduzir, pois a seleção para reprodução é realizada em outro momento, como descrito em 5.3.3.

5.3.7 Exploração do Paralelismo

A estrutura proposta permite a exploração do paralelismo existente em cada máquina, no caso de máquinas multiprocessadas e do paralelismo oferecido pelos ambientes distribuídos, como é o caso das redes de computadores. O paralelismo local pode ser explorado através da implementação dos indivíduos como *threads* que executam em paralelo. Já o paralelismo do ambiente distribuído pode ser explorado distribuindo uma população para cada máquina da rede. O AG paralelo proposto foi testado através da simulação do paralelismo com o uso intensivo de *threads*.

5.4 Protótipo Implementado

Para desenvolvimento do protótipo foi escolhida a linguagem JAVA, por sua independência de plataforma e facilidade de exploração das características intrinsecamente paralelas dos AGs através da simulação do paralelismo com a utilização de *threads*. A bateria de testes foi realizada a partir de instancias publicas do PCV, coletadas via internet, sendo: ATT48, BERLIN52, KROC100, EIL101, CH150, GR202, TS225, A280, PCB442, GR666, PR1002. Os testes foram realizados em microcomputadores Pentium IV 1.5GHZ com 256MB RAM/400MHZ.

5.4.1 Estrutura do Protótipo

O protótipo desenvolvido tomou como base a tecnologia de orientação a objetos, tendo sido implementadas três classes abstratas básicas para suportar o método proposto, simulando o paralelismo através de *threads*. A cada uma dessas classes abstratas há uma correspondente que implementa os métodos dependentes do problema. Um conjunto de classes auxiliares implementa elementos de interface e apresentação de resultados. As três classes que suportam o método proposto são: Supervisor, Population e Individuo. A Figura 19 ilustra a hierarquia de comando na estrutura do protótipo.

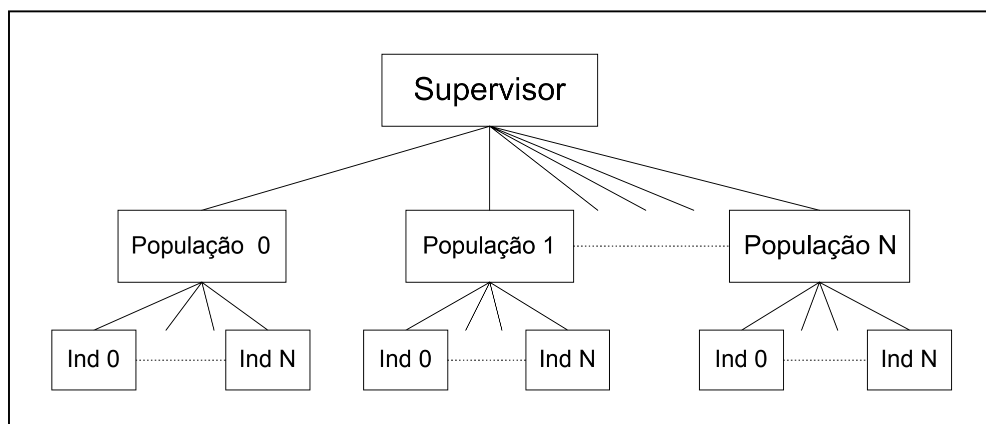


Figura 19 – Hierarquia de Comando

5.4.1.1 Classe Supervisor

Essa classe é responsável por carregar a instância do problema, capturar os parâmetros para o algoritmo e inicializar as populações. Posteriormente o supervisor age como intermediário na migração de indivíduos entre as populações. Os métodos de trabalho incluem:

- ✓ public Supervisor(BenchMark bench); – Construtor, recebe como parâmetro um objeto que implementa a carga da instância.
- ✓ public void setParametros(ParametersSupervisor supParam); – Permite a interface gráfica enviar os parâmetros setados pelo operador.
- ✓ public void run( ); – Metodo padrão da *thread* executa o comportamento do supervisor;
- ✓ public final synchronized void insertReprodutor(Individuo ind); – Permite às populações enviarem seus indivíduos migrantes, implementando uma sala de espera controlada, onde permanece apenas um representante de cada população.
- ✓ public final synchronized Individuo getReprodutor(Population pop); – Permite às populações requisitarem um reprodutor, com vistas a melhorar as características genéticas dos indivíduos internos.
- ✓ public Vector getGenes( ); – Permite a requisição do conjunto de genes que compõem a instância.
- ✓ public abstract Population createPopulation(); – Este método deve ser redefinido pois trabalha com características dependentes do problema. Sua implementação consiste de uma única linha: Population pop = new Populacao(); necessária para permitir o polimorfismo na classe Population.

5.4.1.2 Classe Population

A classe Population é responsável pela criação dos indivíduos iniciais da população e pela execução do AG aplicando os operadores de *crossover*, mutação e seleção. As populações são *threads* que executam um laço infinito, sendo que um sinal de parada e/ou recomeço pode ser

enviado a qualquer momento através da interface gráfica. Também está a cargo da classe Population o controle da migração de indivíduos e dos mecanismos de detecção de mínimos locais. A interface gráfica permite ainda alternar entre os operadores disponíveis, a qualquer tempo, bastando para isso escolher um novo operador. Também é possível alterar os parâmetros que regem o algoritmo, como número mínimo/máximo de indivíduos, intervalo de migração. Os métodos principais da classe population incluem:

✓ public Population ( ); – Construtor, realiza as inicializações necessárias ao inicio do algoritmo, criando os objetos necessários.

✓ public void setParameters(ParametersPopulation popParam); – Permite ao supervisor enviar os parâmetros iniciais. Esse método é também invocado pela interface gráfica para atualizar os parâmetros durante o processo evolutivo.

✓ public void setSemaforo(Semaforo semaf); – Este método permite ao supervisor enviar um semáforo para a sincronização das populações. O semáforo controla a distribuição do processador entre as populações, garantindo que todas terão acesso ao mesmo. Esse mecanismo foi necessário ao protótipo visto que o mesmo simula o paralelismo pelo uso intensivo de *threads,* desse modo foi necessário implementar um mecanismo de garantia de execução para todas as populações. Em uma implementação distribuída esse mecanismo é desnecessário.

✓ private synchronized void crossover(String crossover); – Este método faz uso de outros métodos para montar os pares e então inicia o processo de *crossover*. Após iniciar o processo, executa um laço de espera (*wait*) pela sinalização de término do processo reprodutivo. Iniciar o processo significa indicar aos indivíduos a tarefa a ser executada.

✓ private synchronized void mutation(String mutation); – Semelhante ao método *crossover*, este método é responsável pelo processo de reprodução por mutação.

✓ private void createFileReults( ); – Cria um arquivo de *log* das atividades da população durante o processo evolutivo.

✓ public synchronized void run( ); – Implementa o comportamento da população implementando o algoritmo genético. Método responsável por controlar as atividades da população e dos seus indivíduos.

✓ private synchronized void reducePopulation( ); – Quando invocado, reduz a população ao seu número mínimo de indivíduos.

✓ private void santoAntonio( ); – Método responsável por realizar a eleição dos pares. Um mesmo individuo pode formar par com vários outros, o processo embora aleatório privilegia os indivíduos mais aptos.

✓ public Individuo getReprodutor( ); – A função básica desse método é varrer a população em busca do individuo mais apto.

✓ private void avaliaPopulation( ); – Um dos mais importantes métodos, responsável pela seleção dos indivíduos mais aptos eliminando os demais. Mantém a população estável no seu número máximo de indivíduos.

✓ protected void createAlvo( ); – Cria um pseudo alvo, sobre o qual serão lançados os dardos durante o processo de seleção dos pares que irão reproduzir.

✓ public void create( ); – Gera o conjunto inicial de indivíduos;

✓ public abstract Individuo createIndividuo( ); – Deve ser redefinido para permitir o suporte ao polimorfismo e ás características dependentes do problema.

5.4.1.3 Classe Indivíduo

A classe Indivíduo é responsável pela implementação dos operadores de *crossover*, mutação e avaliação da aptidão (*fitness*). Essa classe também é responsável pela codificação e representação da solução do problema. Uma solução é tratada como um conjunto de objetos organizados em uma determinada ordem. Sendo que é suportada apenas a representação na forma de caminho cíclico (*path*), No entanto uma representação binária pode ser mapeada para objetos, onde cada objeto pode assumir somente 0 e 1, enquanto que a redefinição dos operadores de *crossover* e mutação permitiria o tratamento adequado.

Estão predefinidos cinco operadores de *crossover* (OX, PMX, CX, HALF e INTERLACED) e cinco de mutação (ISM, EM, DM, SIM e IDM), enquanto que outros cinco operadores, estão apenas declarados permitindo uma implementação personalizada de acordo com o problema. A classe Indivíduo apresenta os seguintes métodos básicos.

✓ public Individuo(Population pop); – Construtor sua função é inicializar alguns objetos internos e definir a população a qual o individuo pertence;

✓ public void setAction(String action); – permite indicar qual a ação a ser executada: *crossover* ou mutação. A ação também define qual o operador a ser usado, dessa forma é possível ter indivíduos executando operadores distintos dentro da mesma população. Por exemplo “ACT_CROSS_OX” define a aplicação do operador de *crossover* Order_OX.

✓ public void setParceiro(Individuo parc) throws NullPointerException; – Invocado pelo objeto que representa a população, para determinar qual o individuo parceiro no processo de *crossover*. Esse processo pressupõe um individuo ativo M e outro passivo H. Como na biologia humana, M concebe os novos indivíduos, enquanto que H participa apenas com sua codificação. O mesmo individuo desempenha os dois papéis, sendo que a definição ocorre no momento da eleição dos casais.

✓ public void run( ); – Coordena a execução do comportamento indicado durante a invocação do método **setAction**.

✓ public static void HALF(Individuo indA, Individuo indB); – Implementa o operador de *croossover* mais simples. Divide os cromossomos de ambos os indivíduos obtendo quatro metades e então as recombina gerando dois filhos.

✓ public static void crossoverOX (Individuo indA, Individuo indB); – Implementa o operador de *crossover* OX – *Order Crossover* – proposto por DAVIS (DAVIS, 1985 *apud* LARRÑAGNA *et all,*1998), como descrito em 3.4.4.1.

✓ public static void crossoverPMX(Individuo indA, Individuo indB); – Implementa o operador de *crossover* PMX – *Partial Mapped Crossover* – sugerido por Goldberg e Lingle em 1985 como descrito em0.

✓ public static void CX(Individuo indA, Individuo indB); – Implementa o operador de *crossover CX – Cycle Crossover* – de Oliver *et all* (1987), descrito em 3.4.4.3

✓ public static void crossoverInterlace(Individuo indA, Individuo indB); – Implementa um operador que constrói dois indivíduos filhos, intercalando pedaços da seqüência dos indivíduos pais.

✓ public Object clone( ); – Esse método é responsável por efetuar o clone do individuo para ser utilizado pelos operadores de *crossover* e mutação, bem como para gerar os indivíduos migrantes. O método precisa ser redefinido na subclasse.

✓ public void ISM( ); – Implementa o operador de mutação *Insertion Simple Mutation* descrito em 3.4.5.4.

✓ public void EM( ); – Implementa o operador de mutação *Exchange Mutation* descrito em 3.4.5.1.

✓ public synchronized void DM( ); – Implementa o operador de mutação *Displacement Mutation* descrito em 3.4.5.2.

✓ public synchronized void SIM( ); – Implementa o operador de mutação *Simple Inversion Mutation* descrito em 3.4.5.3.

✓ public abstract void print(); – Método destinado a realizar a impressão do cromossomo que codifica o Individuo. Este método permite extrair a seqüência que representa a melhor solução candidata no momento de sua invocação.

✓ public abstract void calcFitness(); – Método responsável pelo cálculo da aptidão do indivíduo.

5.4.2 Simulação do Paralelismo

O paralelismo foi simulado com a utilização de *threads*. Tanto as populações quanto os indivíduos que as compõem são *threads*. Cada população após selecionar os indivíduos para *crossover,* inicializa um grupo de *threads* para a execução da operação e então executa um laço de espera (*wait)* aguardando todos os casais completarem a reprodução. O mesmo ocorre no caso da operação de mutação, entretanto essa é uma operação unária produzindo apenas um descendente.

Um escalonador garante uma parcela de tempo de processador para cada população, a efetivação dessa garantia foi posta a carga de um semáforo multivalorado. Esse mecanismo faz com que cada população tenha de esperar sua vez para executar um ciclo e então ceder o processador para outra. O tempo de cada população é variável, de acordo com o seu número de indivíduos e operadores a serem aplicados. O semáforo mantém um contador cíclico que é incrementado toda vez que uma população recebe o processador. A população necessariamente precisa solicitar ao semáforo permissão para executar um ciclo evolutivo e deve necessariamente indicar ao mesmo quando do seu término.

Inicialmente, o supervisor inicializa todas as populações com o mesmo número de indivíduos e os mesmos operadores. No entanto é possível alterar todos os parâmetros iniciais de cada população, a partir de uma interface gráfica individual para cada população. Os parâmetros disponíveis incluem os operadores de *crossover* e mutação, o número de indivíduos na população, os intervalos de aceite e envio de indivíduos migrantes e de amostragem.

6 Análise dos Resultados Obtidos

Neste capítulo são apresentados, os resultados obtidos, os dados básicos referentes aos testes desenvolvidos encontram-se resumidos e tabelados por instância. Para cada instância são apresentados três gráficos individuais que sintetizam a curva de convergência obtida em cada conjunto de teste. O quarto gráfico apresenta a comparação entre as três formas testadas. O eixo Horizontal representa o número de gerações e o eixo vertical o custo do melhor caminho em cada geração. Em seguida, um conjunto de figuras apresenta o ciclo evolutivo para cada instancia. Uma discussão sobre os resultados obtidos frente aos objetivos do trabalho encerra o capítulo.

6.1 Instancia ATT48

A instancia ATT48, é uma instancia pequena e com baixo grau de dificuldade e sua solução ótima foi alcançada diversas vezes. Observou-se, no entanto que com pequeno número de indivíduos, sempre se obteve apenas uma aproximação, sendo que a solução ótima foi alcançada com o algoritmo executando diversas populações. A solução ótima foi, às vezes, alcançada aumentando-se o tamanho da população de modo a equivaler a todas as populações juntas. Outra observação é a de que com populações grandes obteve-se uma convergência

mais veloz do que múltiplas populações, mas na maioria dos testes com populações grandes ocorreram mínimos locais.Os resultados obtidos estão resumidos na Tabela 3.

ATT48 – 48 Cidades							
Populações	**Indivíduos**	**Intervalo de aceite**	**Intervalo de envio**	**Desv. Pad. Estagnação**	**Resultado conhecido**	**Resultado Obtido**	**Gerações**
1	10-50	-	-	30	33523,708	35675,061	825
10	10-50	25	24	30	33523,708	33523,708	350
1	10-500	-	-	30	33523,708	33701,261	225

Tabela 3 – Resultados da Instancia ATT48

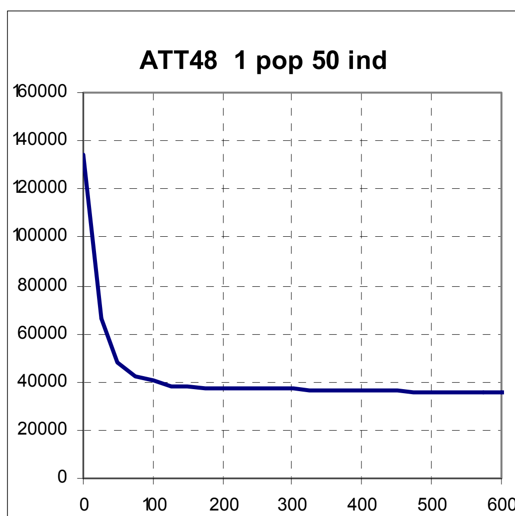


Gráfico 1 – Att48 – 1 População 50 Indivíduos

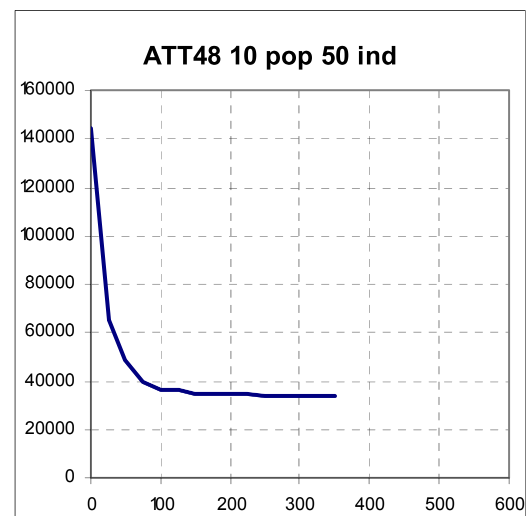


Gráfico 2 – Att48 – 10 Populações 50 Indivíduos

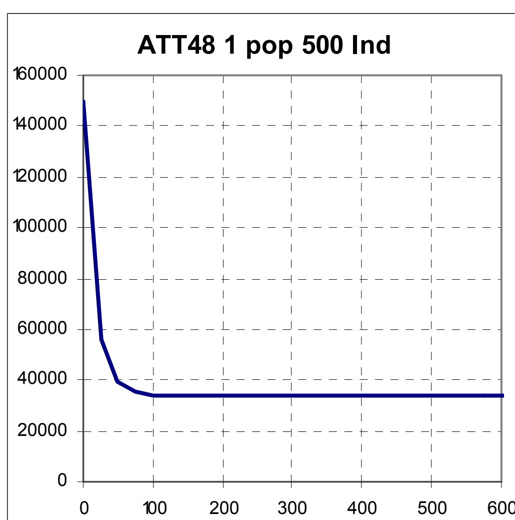


Gráfico 3 – Att48 – 1 População 500 Indivíduos

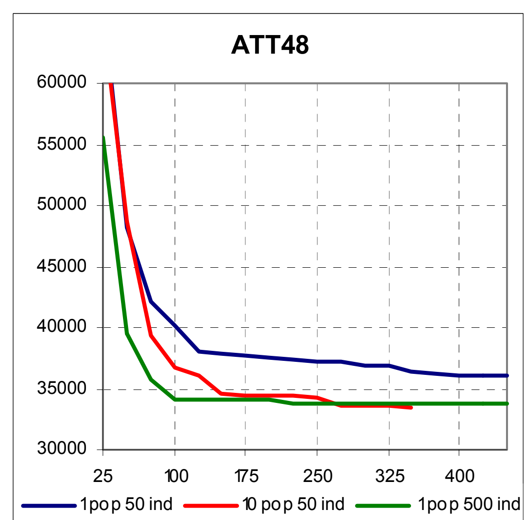


Gráfico 4 – Att48 – Comparativo da Região Critica

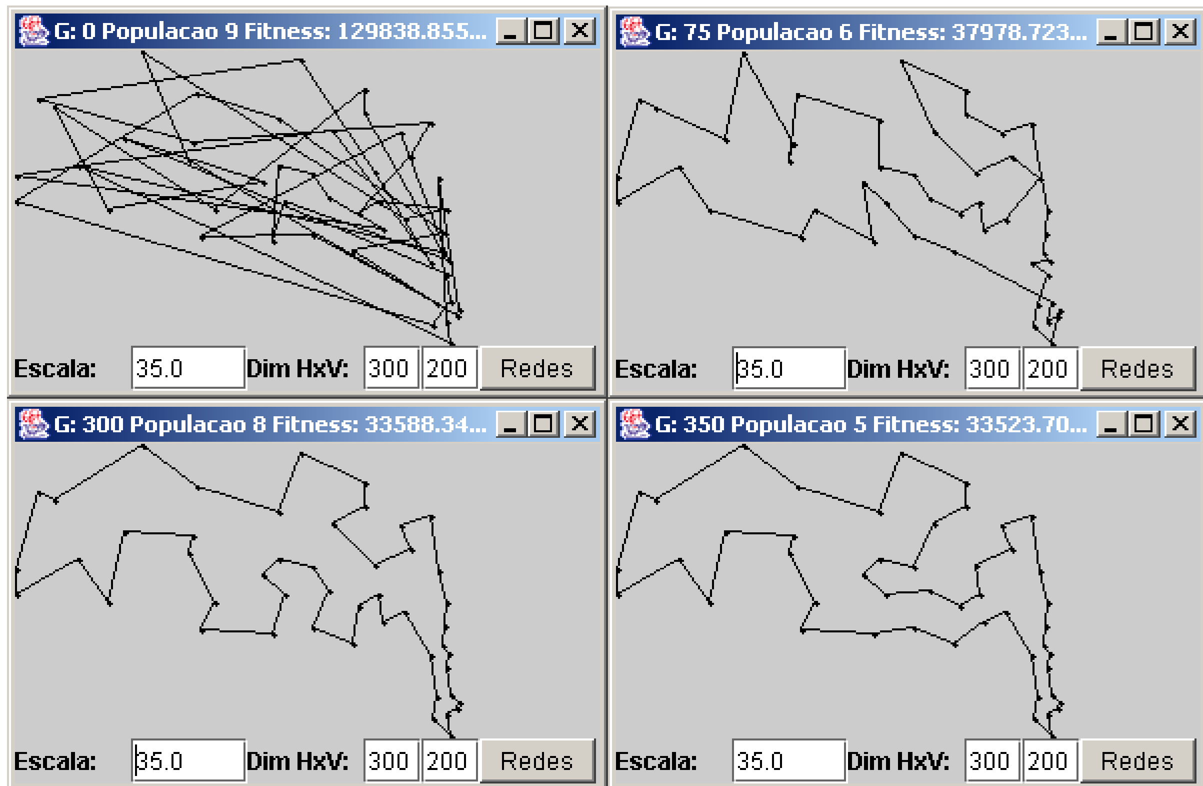


Figura 20 – Evolução da Instância Att48

6.2 Instancia BERLIN52

Berlin52 apresenta baixo grau de dificuldade e os resultados obtidos são muito similares a ATT48. A solução ótima foi alcançada na geração de número 450 com 10 populações de 50 indivíduos e na geração 1325 com 1 população de 500 indivíduos. Com uma população de 50 indivíduos, ocorreu um mínimo local em 775 gerações, como mostra a Tabela 4

Berlin52 – 52 Cidades							
Populações	**Indivíduos**	**Intervalo de aceite**	**Intervalo de envio**	**Desv. Pad. Estagnação**	**Resultado conhecido**	**Resultado Obtido**	**Gerações**
1	10-50	-	-	15	7544,3659	8015,7771	775
10	10-50	50	49	15	7544,3659	7544,3659	450
1	10-500	-	-	15	7544,3659	7544,3659	1325

Tabela 4 – Resultados da Instancia Berlin52

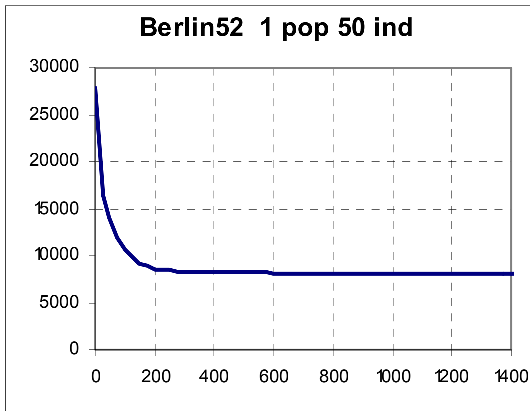


Gráfico 5 – Berlin52 – 1População 50 Indivíduos

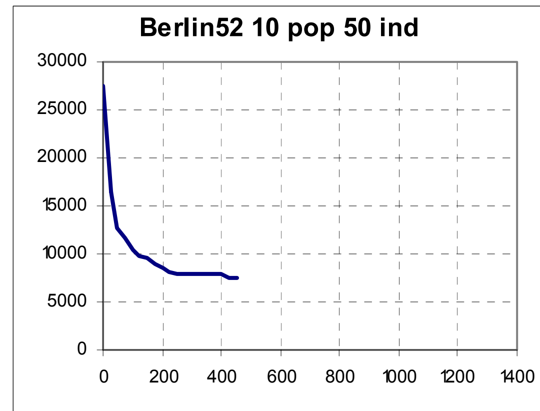


Gráfico 6 – Berlin52 – 10 Populações 50 Indivíduos

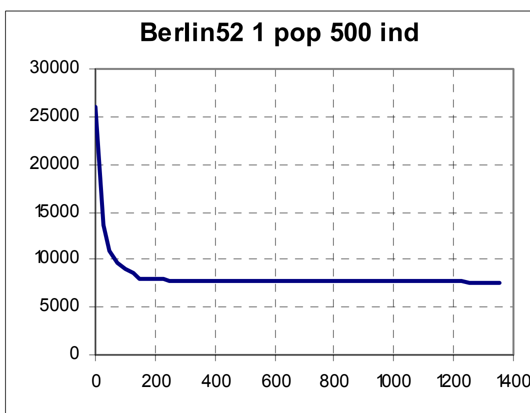


Gráfico 7 – Berlin52 – 1 População 500 Indivíduos

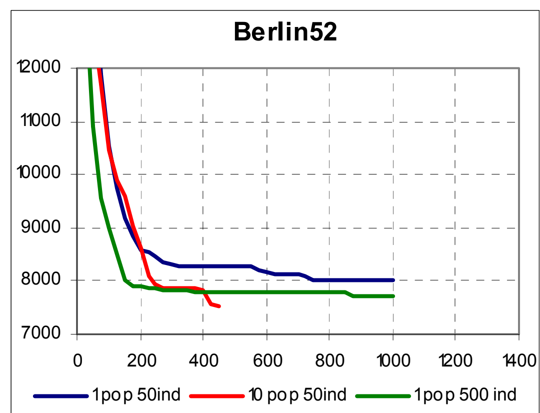


Gráfico 8 – Berlin52 – Comparativo da Região Critica

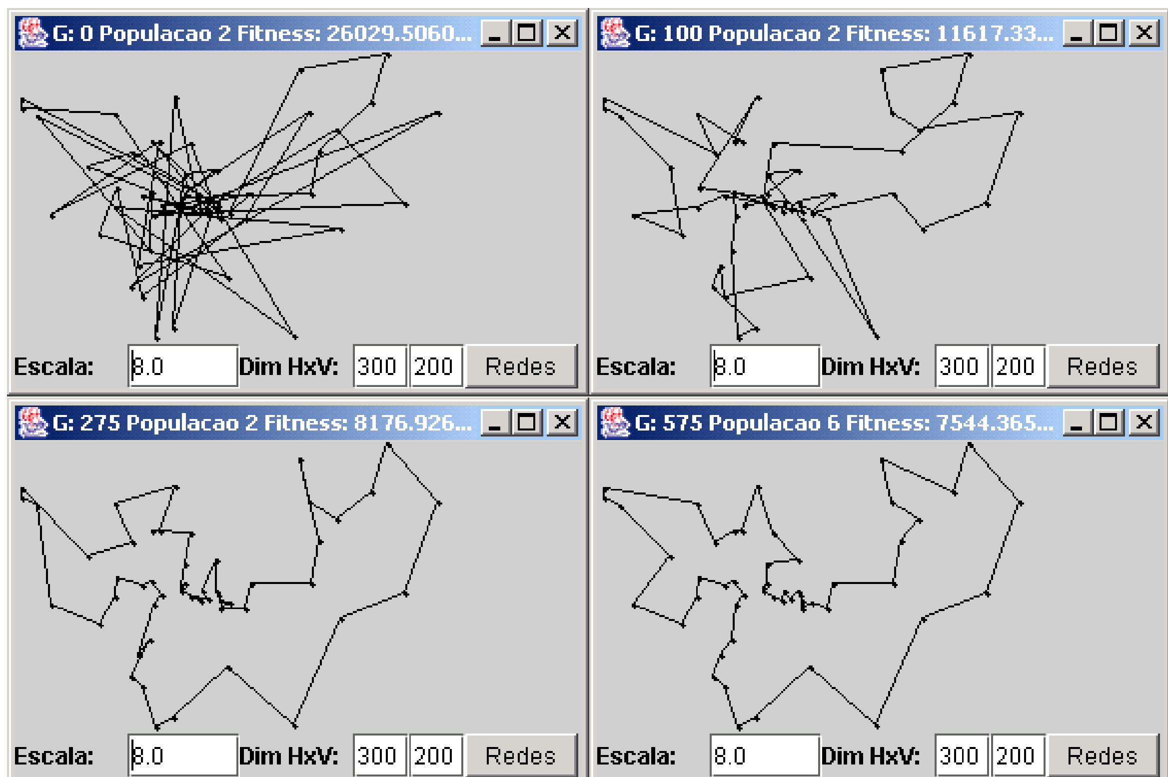


Figura 21 – Evolução da Instancia Berlin52

6.3 Instancia KROC100

Para a instância Kroc100, foram obtidos os resultados mostrados na Tabela 5. Com 1 população de 100 indivíduos, a melhor aproximação foi obtida após 1300 gerações, já com 10 populações de 100 indivíduos cada, a obteve-se uma boa aproximação em 450 gerações e com uma única população de 1000 indivíduos, a melhor aproximação ocorreu após 1125 gerações.

Kroc100 – 100 Cidades							
Populações	**Indivíduos**	**Intervalo de aceite**	**Intervalo de envio**	**Desv. Pad. Estagnação**	**Resultado conhecido**	**Resultado Obtido**	**Gerações**
1	10-100	-	-	20	20750,762	23314,153	130
10	10-100	25	25	20	20750,762	20852,278	450
1	10-1000	-	-	20	20750,762	20914,785	1125

Tabela 5 – Resultados da instância KROC100

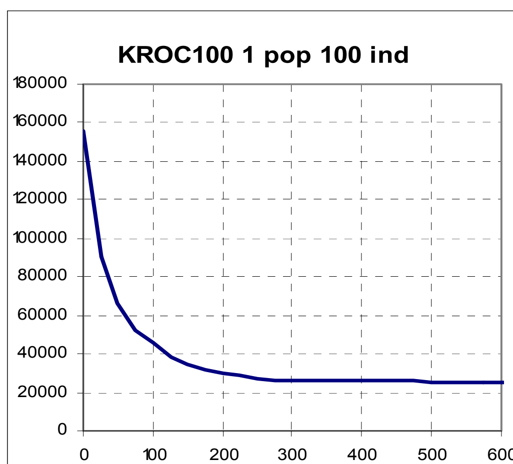


Gráfico 9 – Kroc100 – 1 População 100 Indivíduos

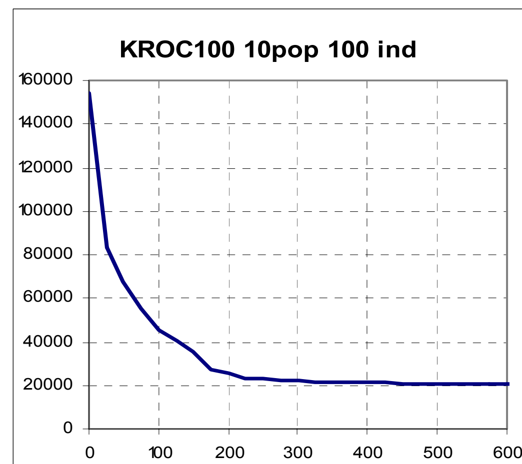


Gráfico 10 – Kroc100 – 10 Populações 100 Indivíduos

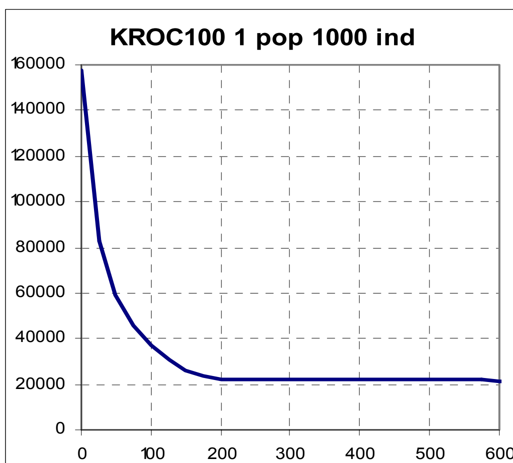


Gráfico 11 – Kroc100 – 1População 1000 Indivíduos

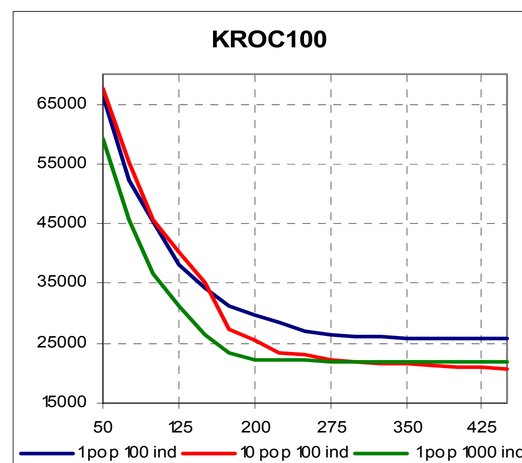


Gráfico 12 – Kroc100 – Comparativo da Região Critica

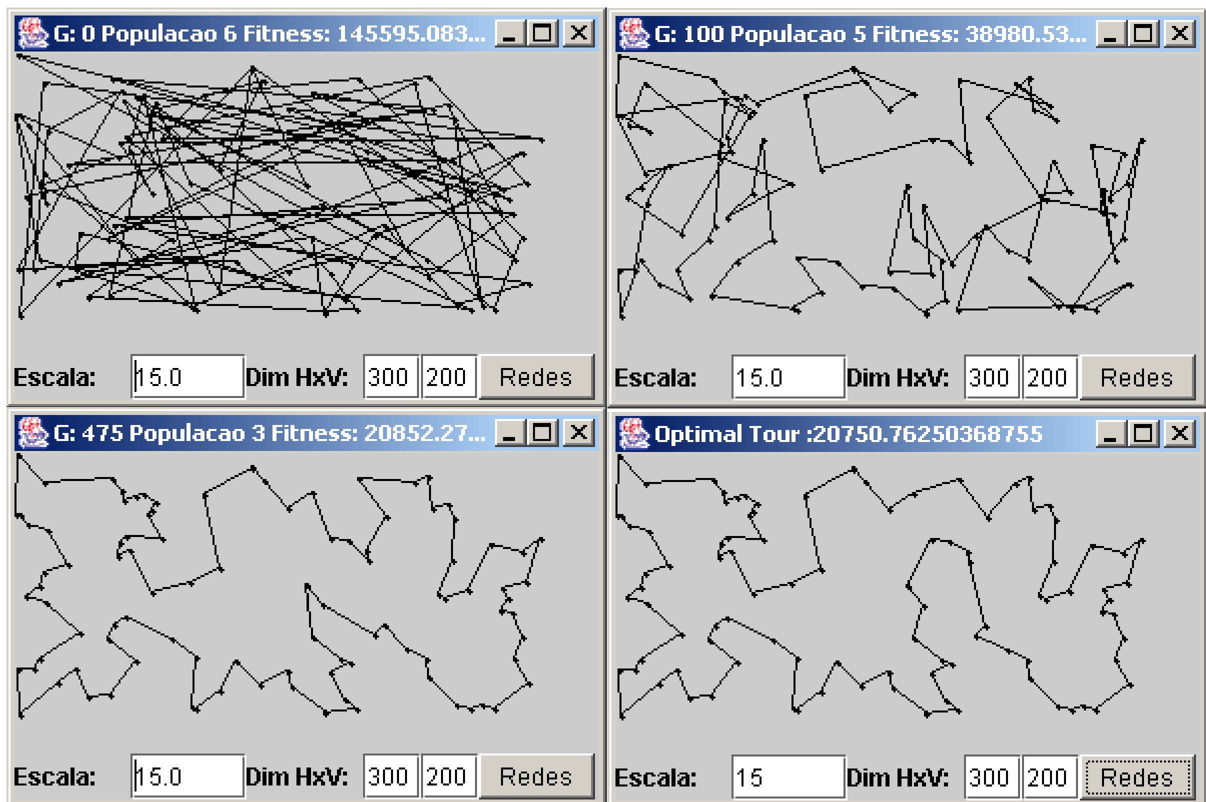


Figura 22 – Evolução da Instancia Kroc100

6.4 Instancia EIL101

A instancia Eil101 embora pequena, apresenta um grau de dificuldade um pouco maior que as anteriores, o que é devido à disposição das cidades. Para essa instância, não se obteve a solução ótima em nenhum dos testes realizados. A Tabela 6 resume os melhores resultados, sendo que a melhor aproximação foi obtida com 10 populações de 100 indivíduos.

EIL101 – 101 Cidades							
Populações	**Indivíduos**	**Intervalo de aceite**	**Intervalo de envio**	**Desv. Pad. Estagnação**	**Resultado conhecido**	**Resultado Obtido**	**Gerações**
1	10-100	-	-	2	642,309	797,486	1625
10	10-100	25	24	2	642,309	653,543	625
1	10-1000	-	-	2	642,309	674,967	1725

Tabela 6 – Resultados da Instancia EIL101

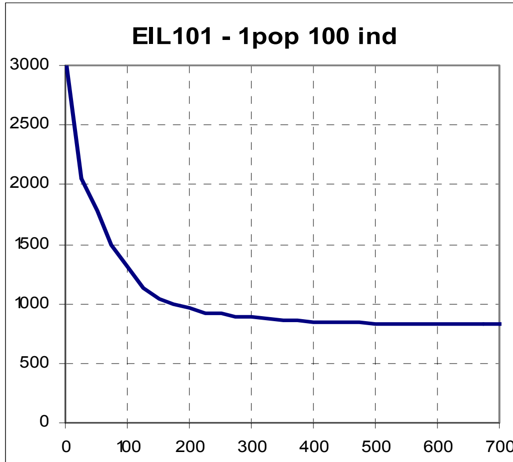


Gráfico 13 – Eil101 – 1 População 100 Indivíduos

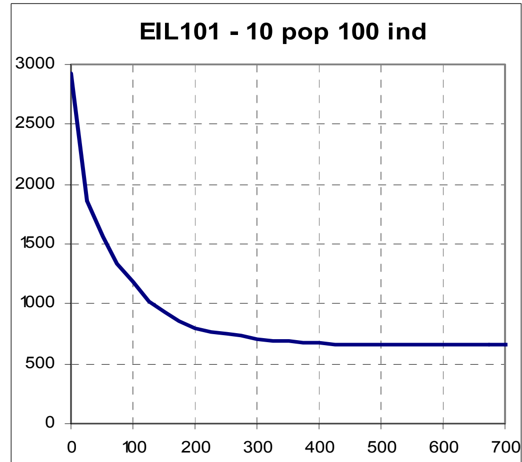


Gráfico 14 – Eil101 – 10 Populações 100 Indivíduos

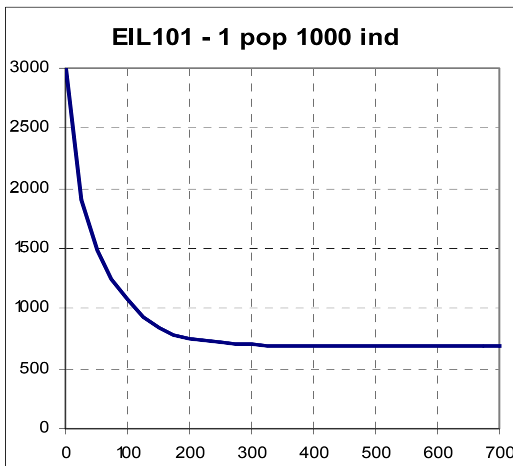


Gráfico 15 – Eil101 – 1 População 1000 Indivíduos

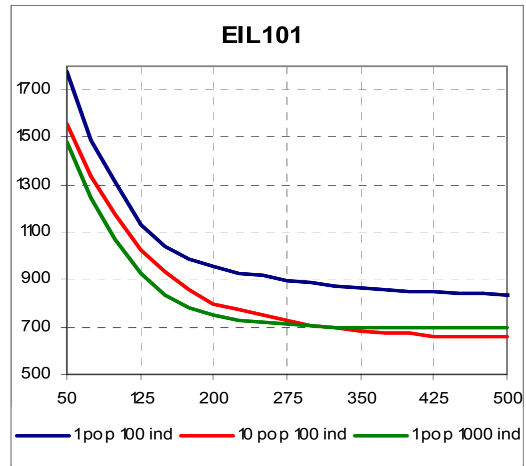


Gráfico 16 – Eil101 – Comparativo da Região Critica

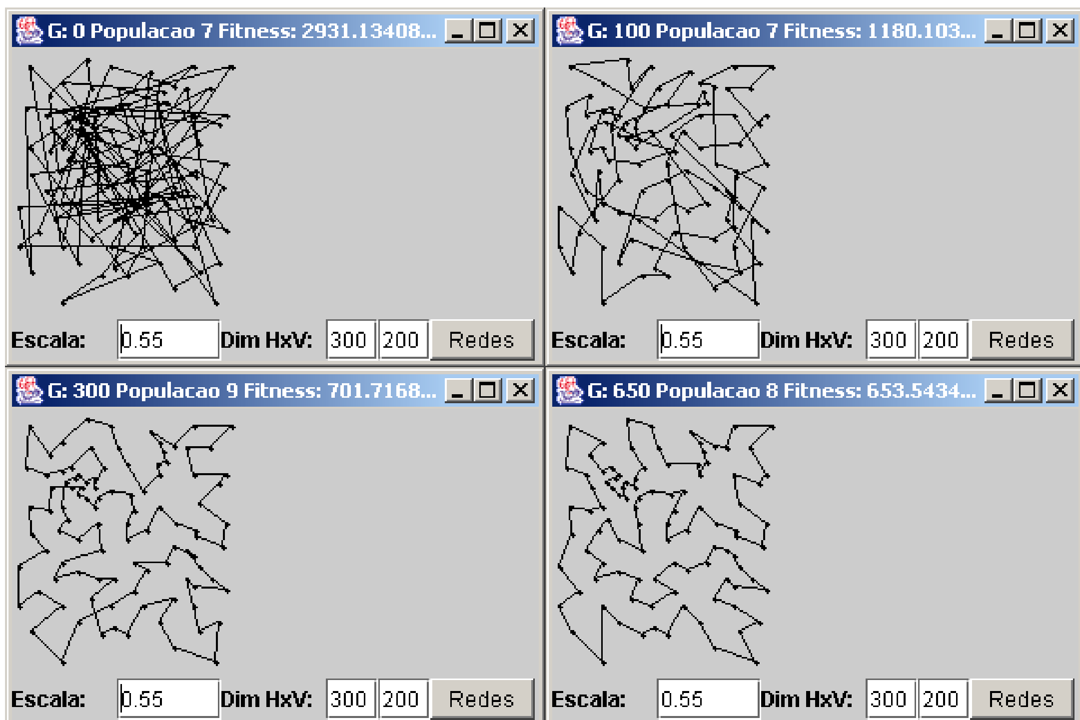


Figura 23 – Evolução da Instancia EIL101

6.5 Instancia CH150

Esta instancia é ligeiramente mais complexa que as anteriores. Embora o número de cidades não seja elevado, a disposição das mesmas incrementa razoavelmente a sua complexidade. Para essa instancia, obteve-se apenas uma aproximação da solução conhecida, conforme o demonstrado na Tabela 7.

CH 150 – 150 Cidades							
Populações	**Indivíduos**	**Intervalo de aceite**	**Intervalo de envio**	**Desv. Pad. Estagnação**	**Resultado conhecido**	**Resultado Obtido**	**Gerações**
1	15-150	-	-	3	6532,280	6843,238	3000
20	10-150	50	49	3	6532,280	6620,480	900
1	10-1000	-	-	3	6532,280	6897,699	1450

Tabela 7 – Resultados da instância CH150

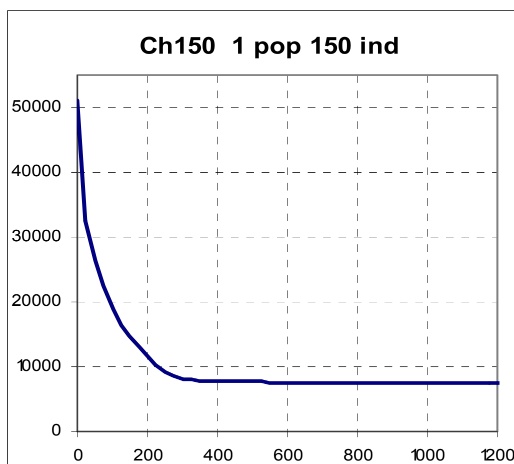


Gráfico 17 – CH150 – 1 População 150 Indivíduos

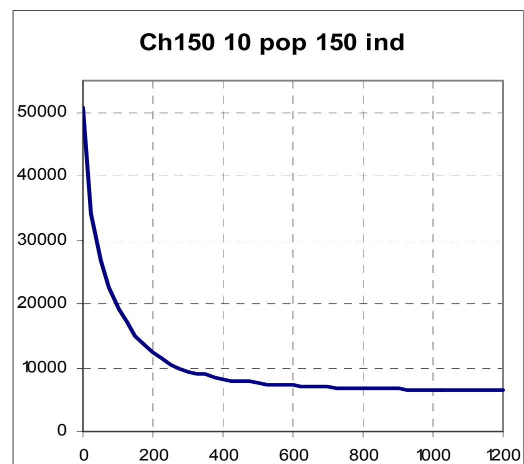


Gráfico 18 – CH150 – 10 Populações 150 Indivíduos

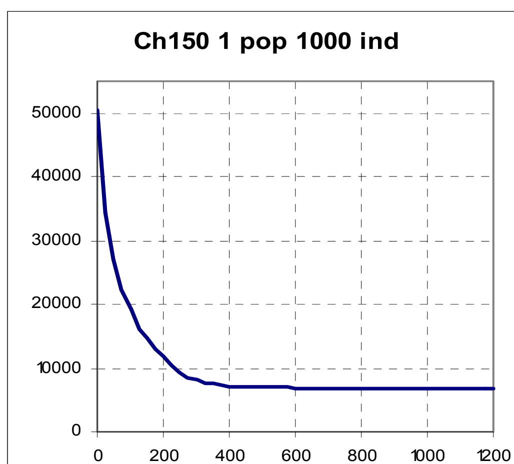


Gráfico 19 – CH150 – 1População 1000 Indivíduos

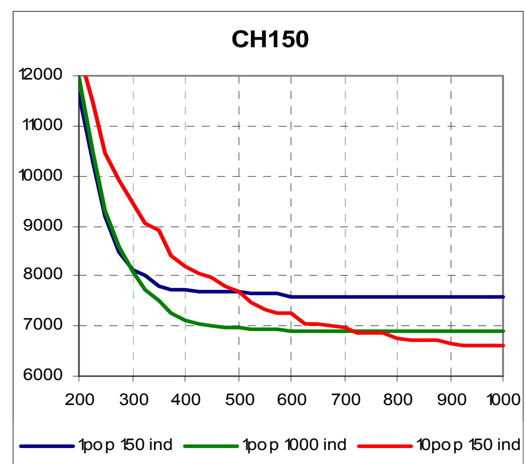


Gráfico 20 – CH150 – Comparativo da Região Crítica

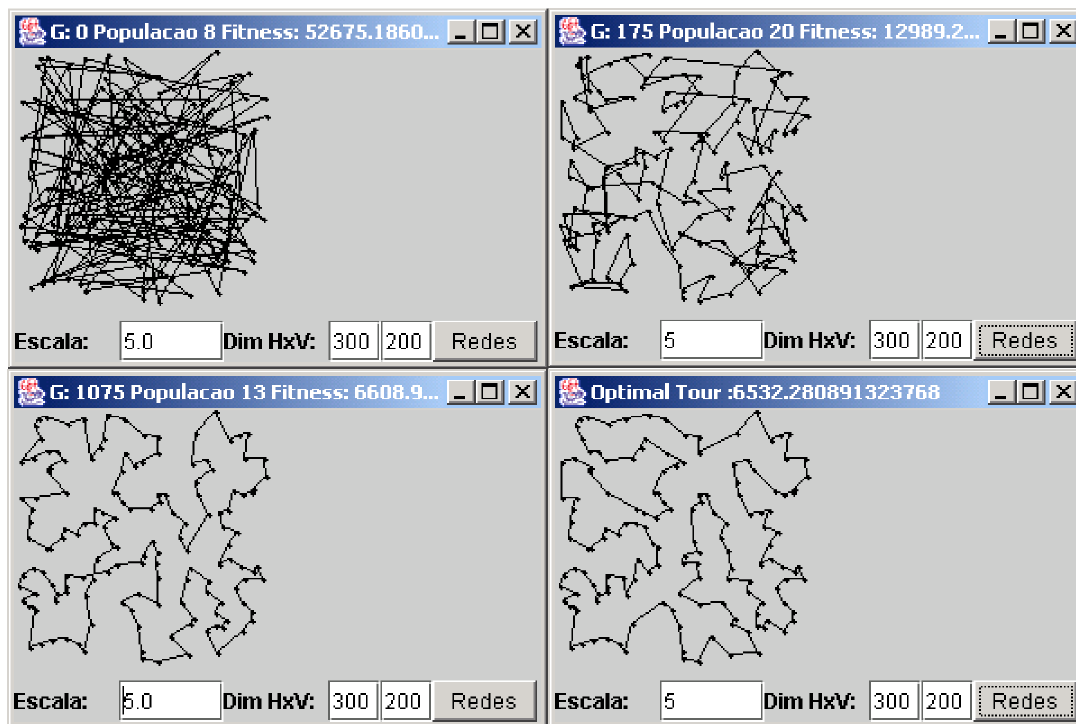


Figura 24 – Evolução da Instancia CH150

6.6 Instancia GR202

Os resultados obtidos com essa instancia foram surpreendentes, o valor mínimo tido inicialmente como solução ótima (ver Figura 25), foi ultrapassado, e obteve-se um mínimo sensivelmente abaixo do mesmo. Os resultados obtidos encontram-se resumidos na Tabela 8, como pode ser observado, foi obtido um mínimo de 502,961 após 3800 gerações, contra 549,998 dado como solução ótima da instancia.

GR202 – 202 Cidades							
Populações	**Indivíduos**	**Intervalo de aceite**	**Intervalo de envio**	**Desv. Pad. Estagnação**	**Resultado conhecido**	**Resultado Obtido**	**Gerações**
1	10-100	-	-	10	549,998	701,965	2000
10	10-100	50	49	10	549,998	502,961	3800
1	10-1000	-	-	10	549,998	550,719	4225

Tabela 8 – Resultados da Instancia GR202

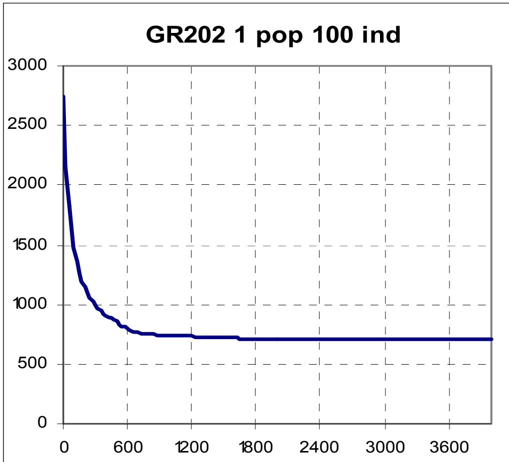


Gráfico 21 – GR202 – 1 População 100 Indivíduos

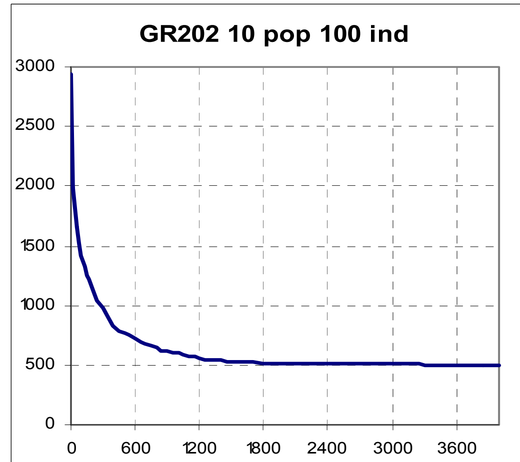


Gráfico 22 – GR202 – 10 Populações 100 Indivíduos

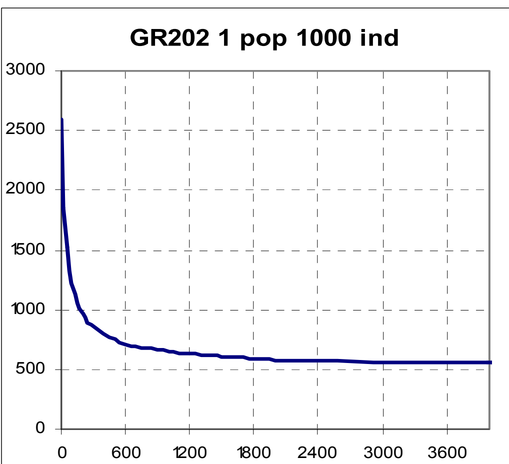


Gráfico 23 – GR202 – 1 População 1000 Indivíduos

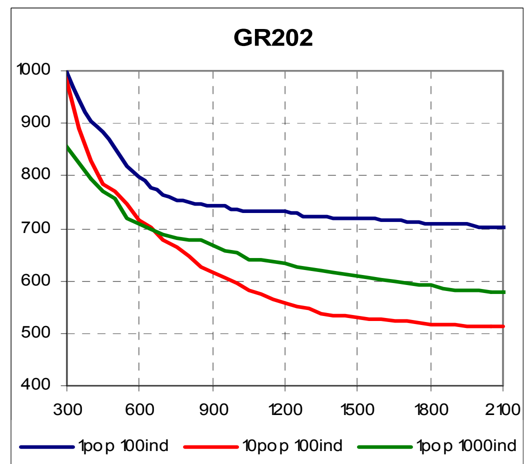
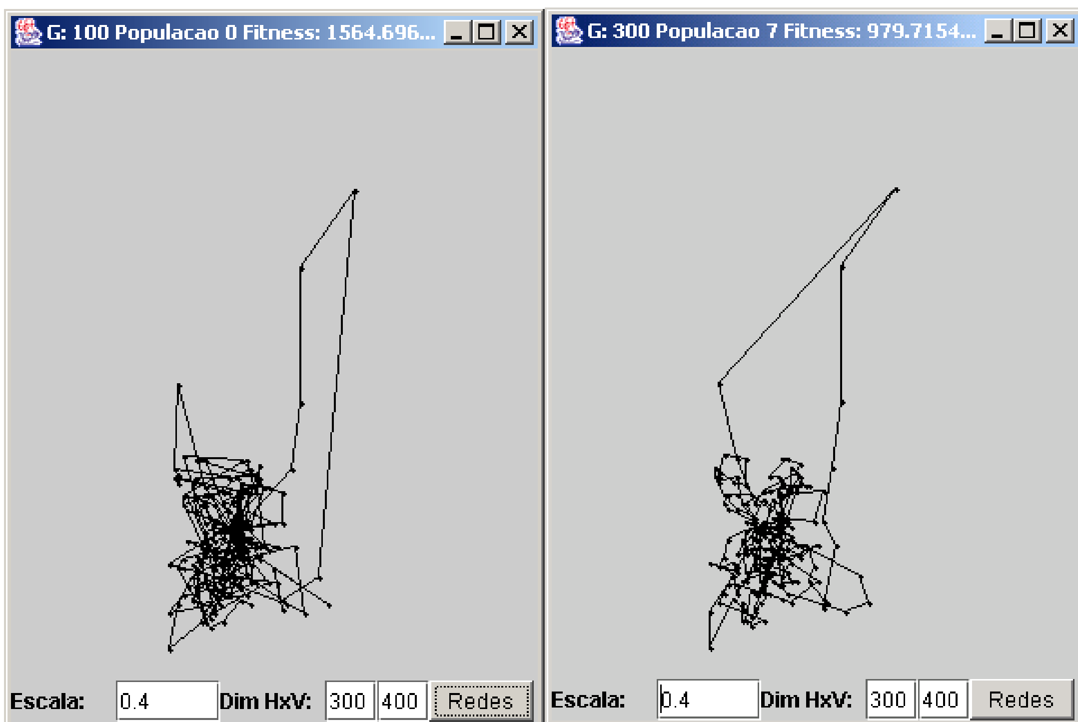


Gráfico 24 – GR202 – Comparativo da Região Critica



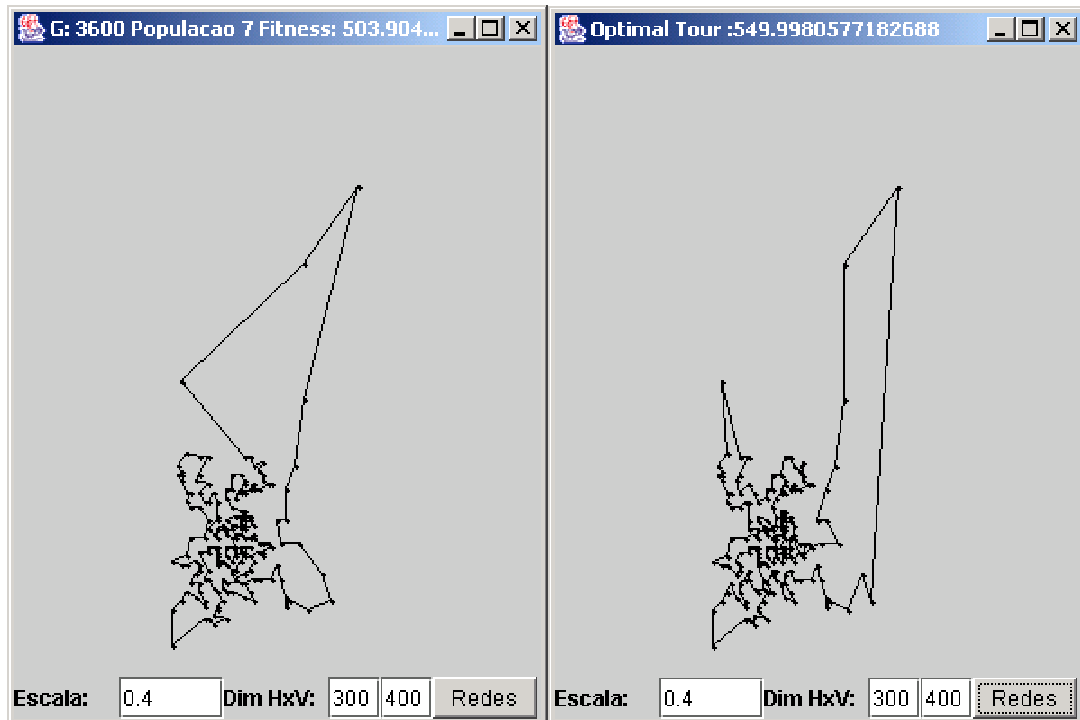


Figura 25 – Evolução da Instancia GR202

6.7 Instancia TS225

As cidades desta instância estão dispostas em uma forma geométricas retangular que a torna relativamente fácil, no entanto foram obtidas apenas aproximações da solução ótima. A melhor aproximação foi de 128928,53 após 1400 gerações, como mostra a Tabela 9.

TS225 – 225 Cidades							
Populações	**Indivíduos**	**Intervalo de aceite**	**Intervalo de envio**	**Desv. Pad. Estagnação**	**Resultado conhecido**	**Resultado Obtido**	**Gerações**
1	10-100	-	-	50	126643	194371,57	2825
10	10-100	50	49	50	126643	128928,53	1400
1	10-1000	-	-	50	126643	131497,52	1100

Tabela 9 – Resultados da Instancia TS225

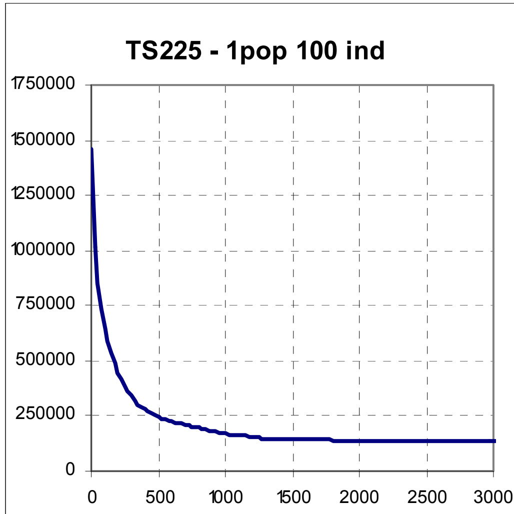


Gráfico 25 – TS225 – 1 População 100 Indivíduos

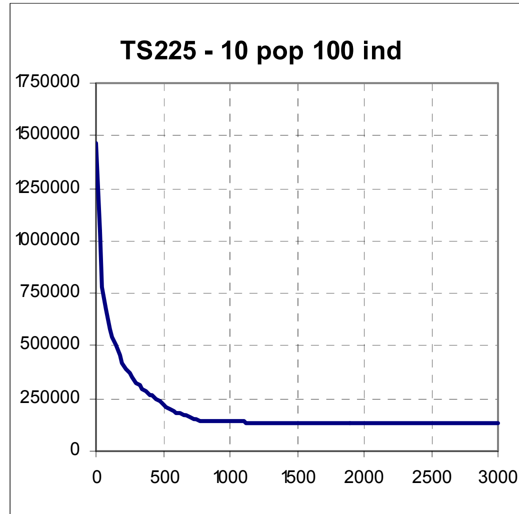


Gráfico 26 – TS225 – 10 Populações 100 Indivíduos

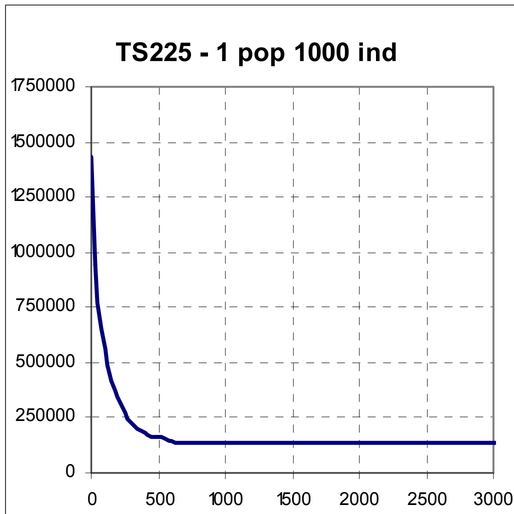


Gráfico 27 – TS225 – 1 População 1000 Indivíduos

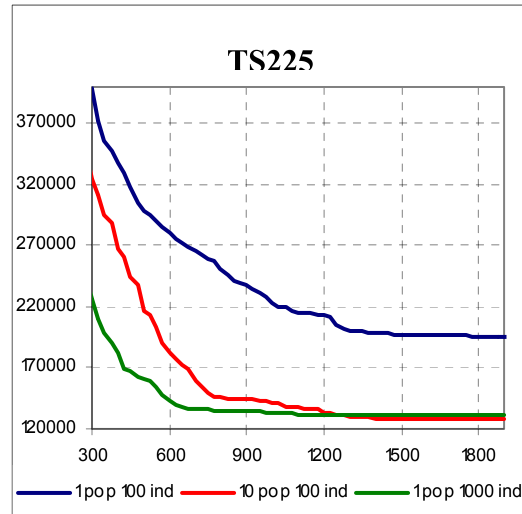
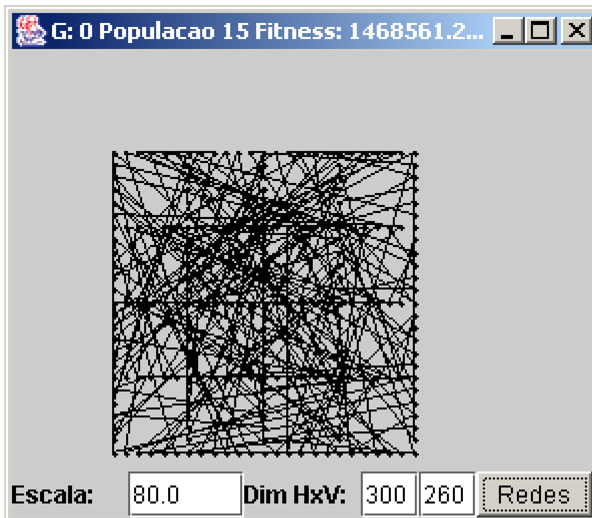
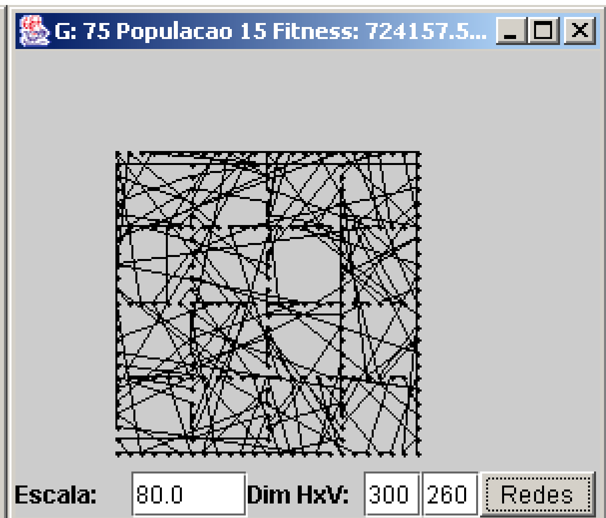


Gráfico 28 – TS225 – Comparativo da Região Critica





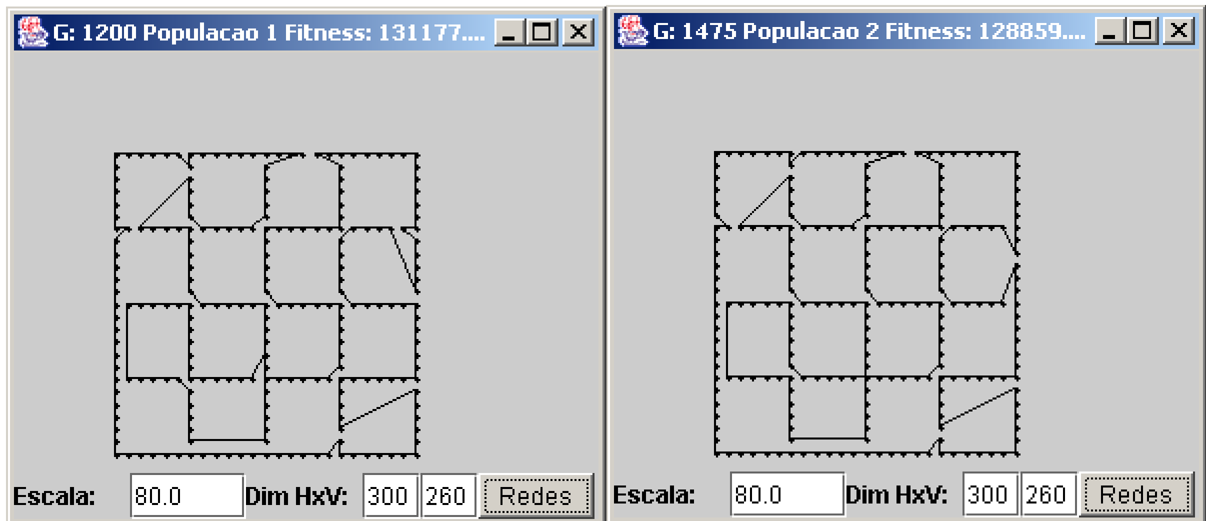


Figura 26 – Evolução da Instancia TS225

6.8 Instancia A280

De todas as instancias testadas, esta instancia foi uma das mais trabalhadas. Seu grau de dificuldade surpreende. Embora não tenha muitos pontos, esses estão dispostos de tal maneira que em nenhum teste se conseguiu mais que uma aproximação. Os resultados obtidos encontram-se na Tabela 10.

A280 – 280 Cidades							
Populações	**Indivíduos**	**Intervalo de aceite**	**Intervalo de envio**	**Desv. Pad. Estagnação**	**Resultado conhecido**	**Resultado Obtido**	**Gerações**
1	20-200	-	-	10	2587,831	3326,400	4200
10	20-200	50	49	10	2587,831	2735,863	4075
1	20-1000	-	-	10	2587,831	3144,564	4100

Tabela 10 – Resultados da Instancia A280

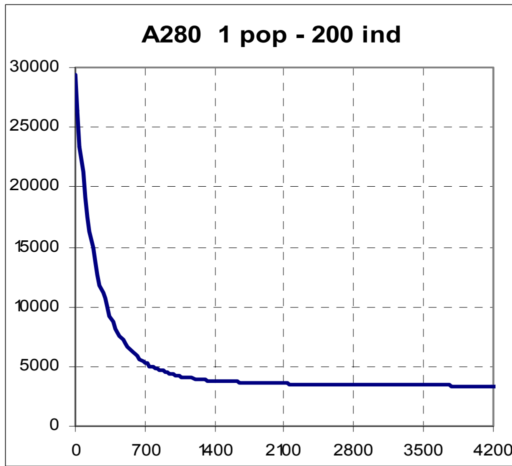


Gráfico 29 – A280 – 1 População 200 Indivíduos

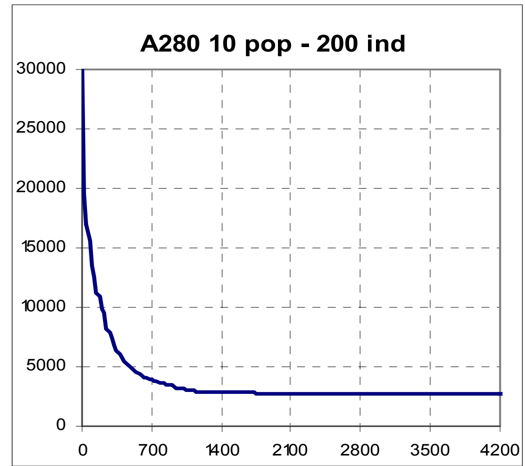


Gráfico 30 – A280 – 10 Populações 200 Indivíduos

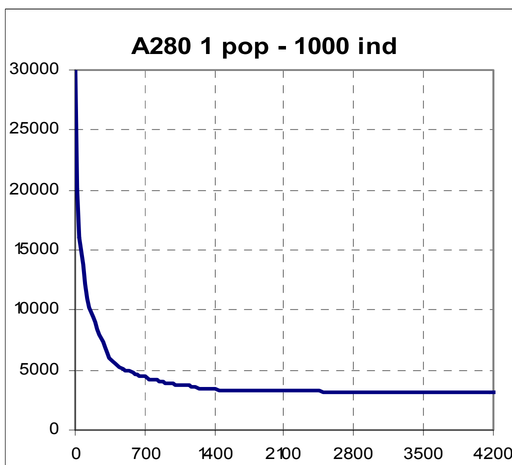


Gráfico 31 – A280 – 1 População 1000 Indivíduos

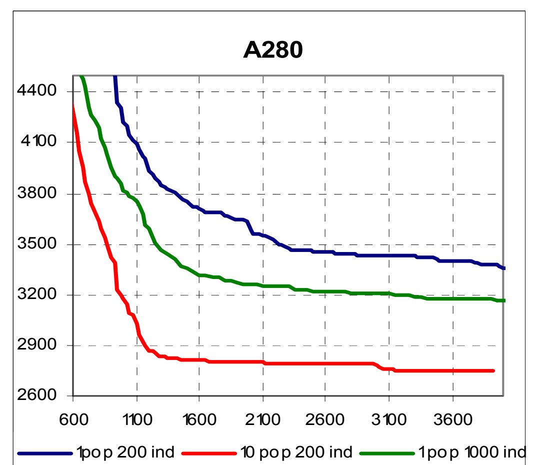


Gráfico 32 – A280 – Comparativo da Região Critica

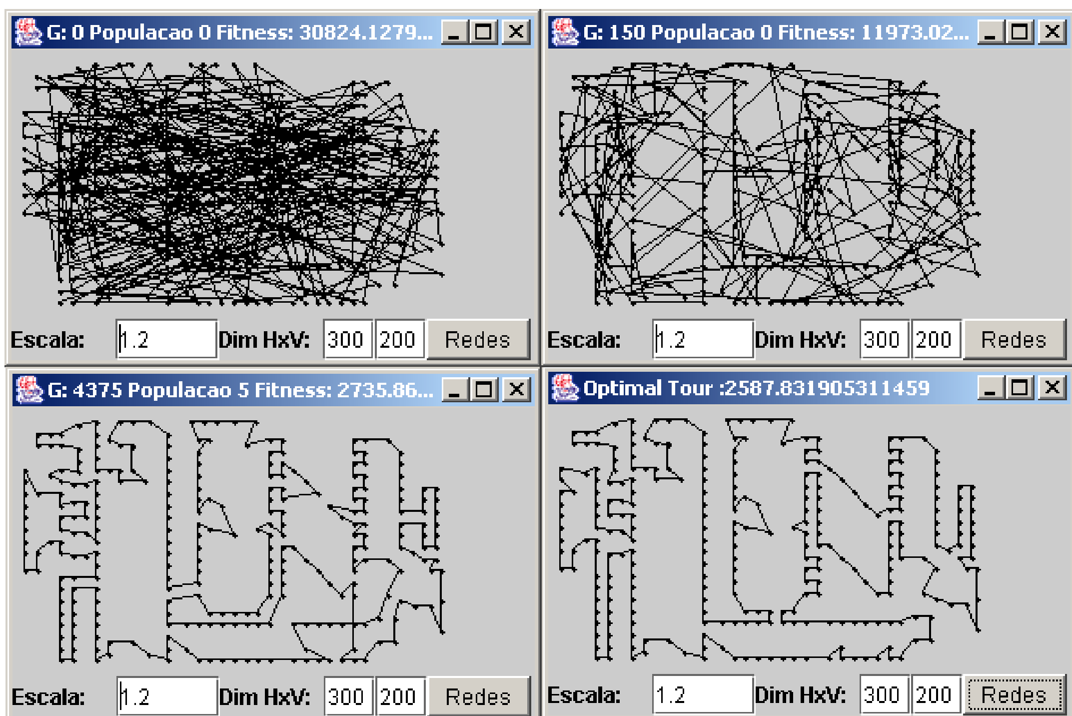


Figura 27 – Evolução da Instancia A280

6.9 Instancia PCB442

Para essa instância, mesmo com as limitações impostas pelo hardware, pode-se considerar que o algoritmo teve um bom desempenho e que foi possível realizar a verificação da hipótese em teste. Os resultados desta instancia estão resumidos na Tabela 11.

PCB442 – 442 Cidades							
Populações	**Indivíduos**	**Intervalo de aceite**	**Intervalo de envio**	**Desv. Pad. Estagnação**	**Resultado conhecido**	**Resultado Obtido**	**Gerações**
1	10-100	-	-	100	50783,547	77159,268	6000
10	10-100	75	69	100	50783,547	57466,313	6000
1	20-1000	-	-	100	50783,547	58767,826	6000

Tabela 11 – Resultados da Instancia PCB442

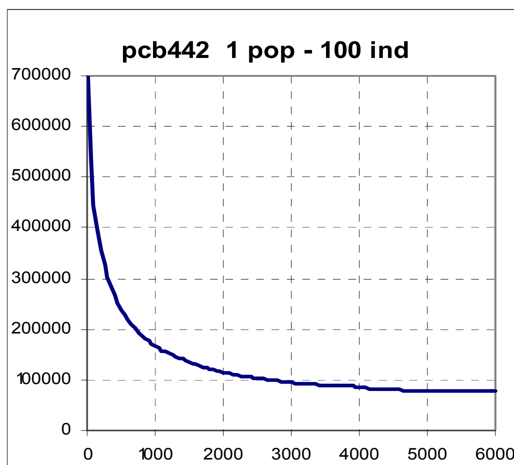


Gráfico 33 – PCB442 – 1 População 100 Indivíduos

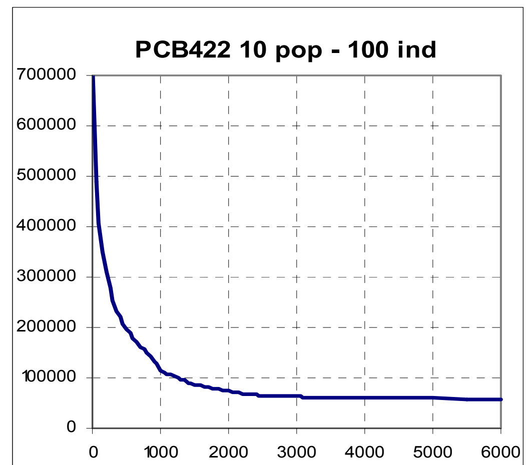


Gráfico 34 – PCB442 – 10 Populações 100 Indivíduos

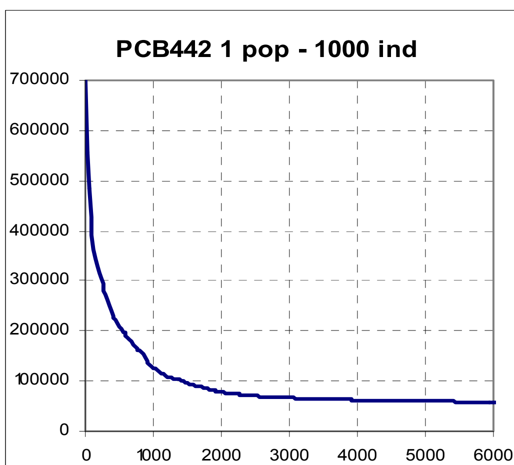


Gráfico 35 – PCB442 – 1 População 1000 Indivíduos

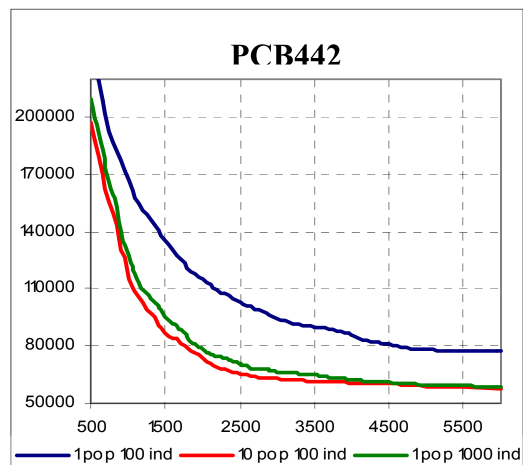


Gráfico 36 – PCB442 – Comparativo da Região Critica

6.10 Instancia GR666

Os teste desta instancia foram surpreendentes, pois mesmo com as limitações impostas pelo hardware, foi possível produzir uma solução melhor do que a tida como ótima, conforme mostra a Tabela 12.

GR666 – 666 Cidades							
Populações	**Indivíduos**	**Intervalo de aceite**	**Intervalo de envio**	**Desv. Pad. Estagnação**	**Resultado conhecido**	**Resultado Obtido**	**Gerações**
1	10-100	-	-	100	3952,53	6050,33	14825
10	10-100	75	69	100	3952,53	3699,73	14800
1	10-1000	-	-	100	3952,53	3961,43	14950

Tabela 12 – Resultados da Instancia GR666

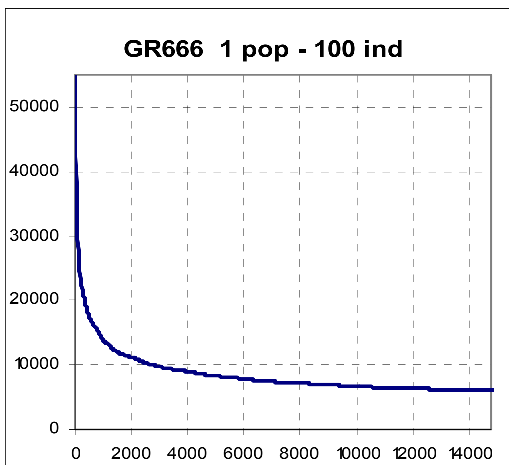


Gráfico 37 – GR666 – 1 População 100 Indivíduos

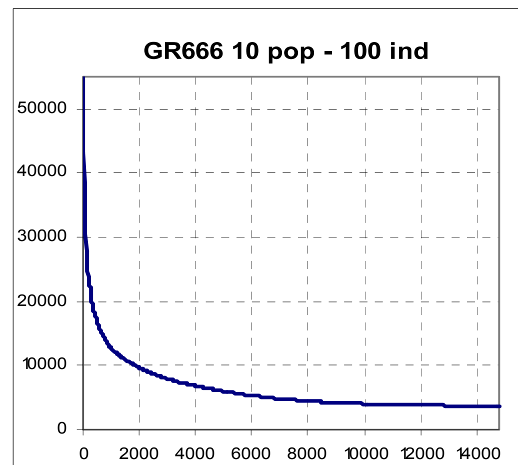


Gráfico 38 – GR666 – 10 Populações 100 Indivíduos

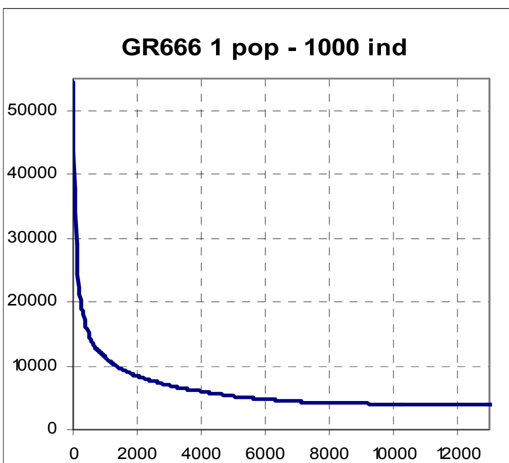


Gráfico 39 – GR666 – 1 População 1000 Indivíduos

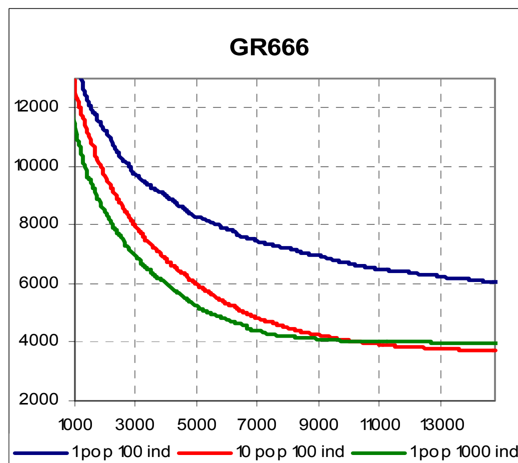


Gráfico 40 – GR666 – Comparativo da Região Critica

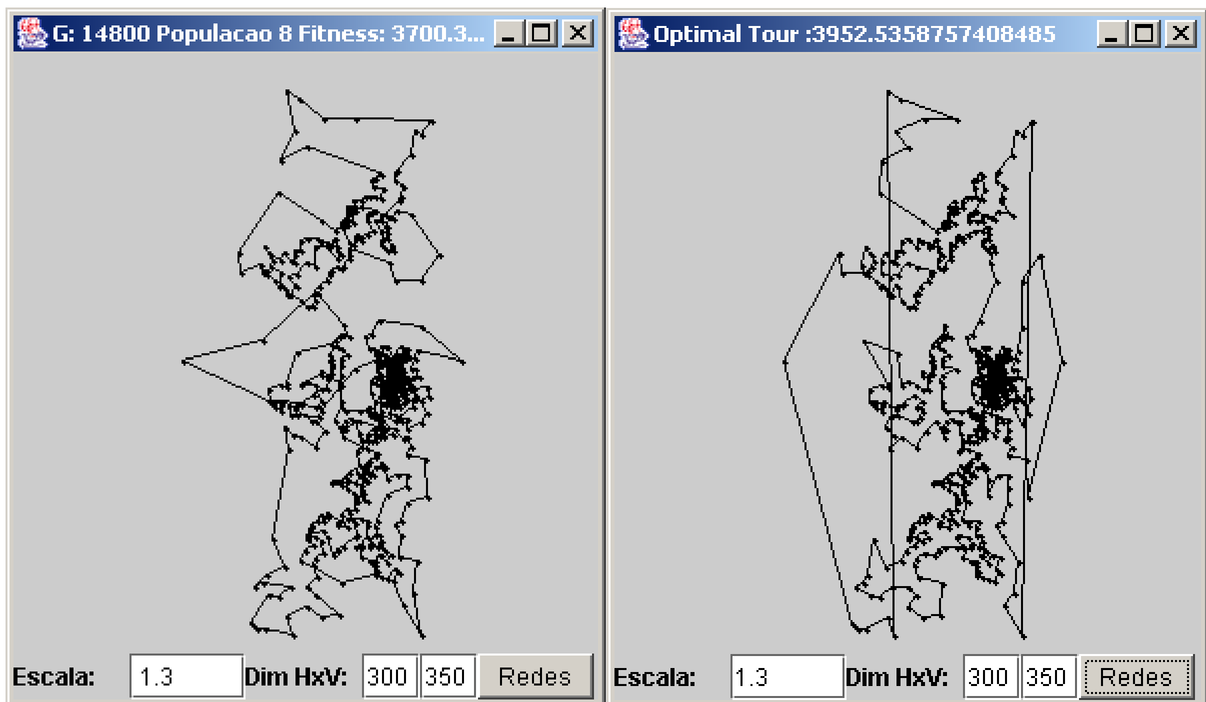


Figura 28 – Melhor Resultado da Instancia GR666

6.11 Instancia PR1002

Com esta instância, os limites de hardware foram atingidos, dessa forma foi a ultima instância testada. Os resultados obtidos estão muito aquém de uma boa aproximação, mas podem ser considerados como efetivos na validação da estrutura de AG paralelo proposta. Os resultados obtidos encontram-se na Tabela 13.

PR1002 – 1002 Cidades							
Populações	**Indivíduos**	**Intervalo de aceite**	**Intervalo de envio**	**Desv. Pad. Estagnação**	**Resultado conhecido**	**Resultado Obtido**	**Gerações**
1	20-200	-	-	100	259066,66	370542,41	14075
10	20-200	75	69	100	259066,66	291364,1	17750
1	20-1000	-	-	100	259066,66	349566,6	11925

Tabela 13 – Resultados da Instancia PR1002

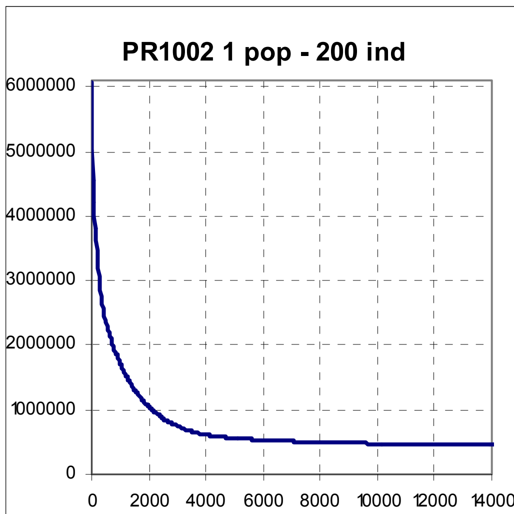


Gráfico 41 – PR1002 – 1Populaçao 200 Indivíduos

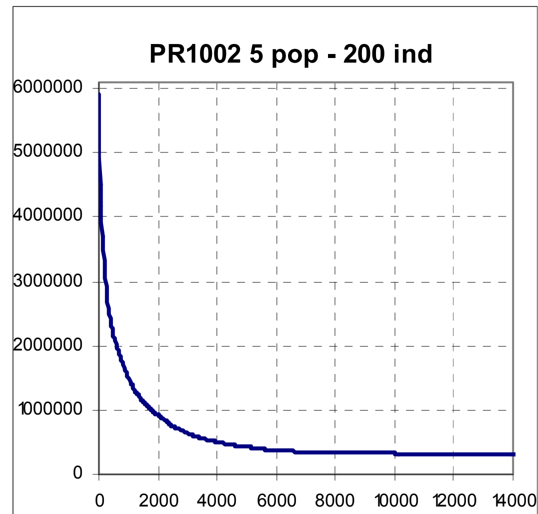


Gráfico 42 – PR1002 – 5 Populações 200 Indivíduos

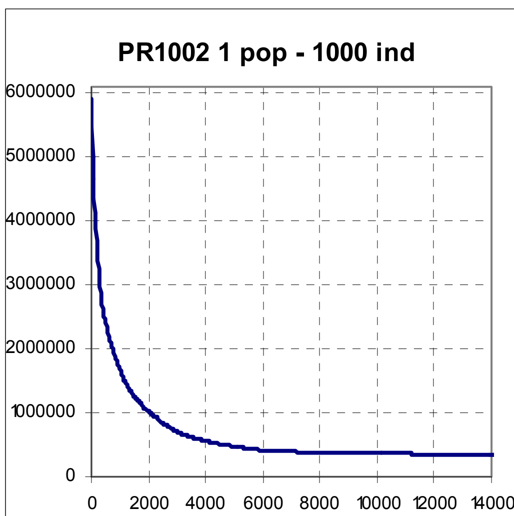


Gráfico 43 – PR1002 – 1 População 1000 Indivíduos

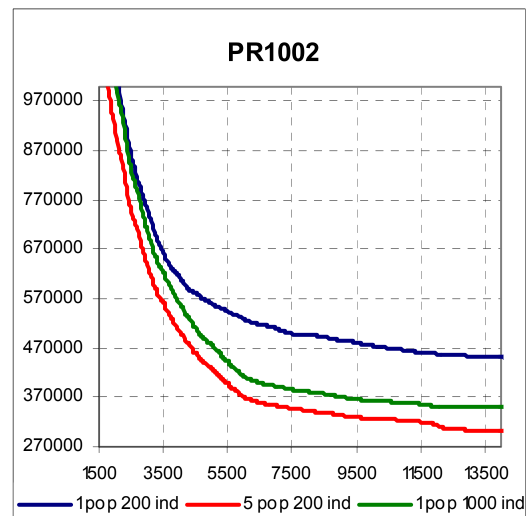


Gráfico 44 – PR1002 – Comparativo da Região Critica

6.12 Discussão dos Resultados

Como demonstrado nos gráficos, tabelas e figuras acima, em todos os testes realizados obteve-se boas aproximações sendo que para algumas instâncias o resultado obtido equivale à solução ótima e em dois casos – instancia GR202 e GR666 – a solução obtida superou a solução dada como ótima no *benchmark*.

A aplicação dos operadores de mutação ocorreu de forma randômica, enquanto que optou-se pelo operador de *crossover* OX, dado que o mesmo apresentou resultados significativamente melhores. A comparação entre os operadores de *crossover* e mutação está fora do escopo deste trabalho, deste modo a opção pelo operador OX se deu com base apenas na observação dos resultados de testes preliminares.

A validação da proposta apresentada neste trabalho vem da análise comparativa dos resultados obtidos com a versão de populações evoluindo em paralelo *versus* resultados obtidos com o mesmo algoritmo executando apenas uma única população, conforme mostram os gráficos acima, em todos os testes, a versão paralela obteve um ciclo evolutivo mais longo, gerando uma aproximação maior da solução ótima, enquanto que a versão simples apresentou condições de estagnação mais prematuras. Por outro lado, a proposta de paralelização mostrou-se adequada sugerindo que sua implementação sobre ambientes distribuídos tornaria o algoritmo apto a trabalhar com instâncias de grande porte. A estrutura do AGP proposto foi completamente validada, o protótipo implementado demonstrou que a proposta apresentada neste trabalho apresenta boa performance nos resultados obtidos, mas mais que isso é extremamente flexível e capaz de tirar proveito das características paralelas do algoritmo, aproveitando o paralelismo oferecido pelos recursos computacionais disponíveis.

7 Considerações Finais

Este capítulo é destinado a apresentar a conclusão geral da pesquisa realizada, bem como apresentar algumas sugestões para novas pesquisas na área.

7.1 Conclusão

Embora não tenham sido realizados testes exaustivos, e mesmo não tendo sido feito um estudo para identificar os melhores operadores de *crossover* e mutação. Os resultados obtidos sugerem que a implementação de AGs paralelos segundo a estrutura proposta neste trabalho, tem boas perspectivas. Os resultados obtidos com a simulação do paralelismo indicam que a troca periódica de indivíduos não acelera o processo de convergência, entretanto regulando o intervalo de migração se obtém maior diversidade de indivíduos, o que possibilita alcançar melhores resultados antes da estagnação.

As baterias de testes mostraram ainda que diversos fatores podem influenciar nos resultados finais, entre eles estão: o tamanho de cada população, o número de populações e a dificuldade intrínseca de cada instancia. Esta ultima se caracteriza pela distribuição espacial das cidades e pelo tamanho da instância. A definição de tamanho e número das populações deve ser norteada por uma análise do grau de dificuldade da instância. O aumento no tamanho da

população depende da capacidade de processamento local, enquanto que um aumento no número de populações, num ambiente distribuído, dar-se-ia pela simples adição de mais nós de processamento, sendo que a estrutura testada e validada, pode assimilar qualquer número de populações.

7.2 Sugestões Para Novas Pesquisas

Novas investigações podem ser realizadas no sentido de implementar a estrutura proposta em um ambiente distribuído, onde cada máquina seria responsável por uma única população, tornando possível investigar a aplicação do algoritmo sobre instâncias maiores.

Outras pesquisas podem ainda investigar mecanismos de seleção não otimistas, ou seja, mecanismos que privilegiem a diversidade e não apenas os indivíduos mais aptos.

O mecanismo de redução/expansão carece ser trabalhado, juntamente com os operadores de mutação. Esses quando aplicados deveriam produzir maior variedade genética mantendo um certo grau de qualidade.

8 Referências Bibliográficas

(AGHA, 1990)

AGHA, GUL - **Concurrent object-oriented programming** - ACM Press New York, NY, USA. p 125 - 141 Periodical-Issue-Article, 1990

(ALAOUI *et all,* 2000)

ALAOUI, MOUNIR S. , FRIDER O., EL-GHAZAWI T. – **A Parallel Genetic Algorithm for Task Mapping on Parallel Machines** – Florida Institute of Technology, Melbourne. Florida, USA. 2000

(ALMASI & GOTTLIEB, 1994)

ALMASI, G.S., GOTTLIEB, A – **Highty Parallel Computing** . Benjamin Cummings Publishing, 2ªEdição,1994.

(APPLEGATE *et all*, 2002)

APPLEGATE, DAVID, *et all,* – **Solution of a 15,112-city Traveling Salesman Problem**, Web-Site http://www.math.princeton.edu/tsp/d15sol/ (20-03- 2002)

(ARAÚJO, 2000)

ARAÚJO, HAROLDO A., – **Algoritmo Simulated Annealing – Uma Nova Abordagem** – Dissertação de Mestrado, Florianópolis UFSC, 2000;

(BELLMAN, 1961)

BELLMAN, R. – **Adaptive control processes: A guided tour** – Princeton, PA: Princeton University Press. Washington, DC. 1961.

(BLELLOCH & MAGGS, 1996)

BLELLOCH, GUY E., MAGGS, BRUCE M – **Parallel algorithms** – ACM Computing Surveys (CSUR), ACM Press New York, NY, USA, p 51 - 54 Periodical-Issue-Article, 1996 .

(COELHO, 1999)

COELHO, LEANDRO S e COELHO, ANTONIO A R – **Algoritmos Evolutivos em Identificação e Controle de Processos: Uma Visão Integrada e Perspectivas.** – SBA Controle & Automação Vol. 10 1999

(COOK, 1971)

COOK, STEPHEN A. – **The Complexity of Theorem-Proving Procedures** – University of Toronto,1971 – Proced 3ª Annual ACM Symposium on Theory of Computing, p.151-158, May 03-05, 1971, Shaker Heights, Ohio, United States

(COOK, 1983)

COOK,STEPHEN A. – **An overview of computational complexity** – University of Toronto, 1983, ACM Press New York, NY, USA p. 400 - 408 Periodical-Issue-Article.

(ESBENSEN, 1998)

ESBENSEN, H., DRESCHER, R. – **Evolutionary Algorithms in Computer Aided Design of Integrated Circuits** – Escola de Inverno em Algoritmos Evolutivos, Notas do Curso, Denmark Technical University, Janeiro, 1998.

(FOMNSECA, 1995)

FONSECA, C.M., FLEMING, P.J., – **An Overview of Evolutionary Algorithms in Multiobjective Optimization. Evolutionary Computation** – The MIT Press, vol. 3 1995.

(GAREY & JOHSON, 1979)

GAREY, M. R., JOHSON, D. S., – **Computers and Intractability: A Guide to the Theory of NP-Completeness** – Ed. W. H. Freeman, San Francisco, 1979.

(GERSTING, 1993)

GERSTING, JDITH L. – **Fundamentos Matemáticos para a Ciência da Compu*tação*** – 3ªEd. LTC – Livros Técnicos e Científicos Editora S.A., Rio de Janeiro – RJ, 1993.

(GOLDBERG, 1989)

GOLDBERG, D.E. – **Genetic Algorithms in Search, Optimization and Machine Learning** – Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1989.

(GOLDBERG, 1994)

GOLDBERG, D.E. – **Genetic and Evolutionary Algorithms Come of Age** – Communications of the ACM, vol. 37, nº3, pp. 113-119.

(HOLLAND, 1975)

HOLLAND, J., – **Adaptation in Natural and Artificial Systems** – University of Michigan Press, Ann Arbor, EUA, 1975. MIT Press Cambridge, MA, USA 1992

(KARR, 1993)

CHARLES L. KARR – **Genetic Algorithms For Modelling, Design, And Process Control** – Proceedings of the second international conference on Information and knowledge management December 1993

(KOZA, 1992)

KOZA, J. R., – **Genetic Programming On the Programming of Computers by Means of Natural Selection** – MIT Press, 1992.

(LARRAÑAGA *et all*, 1998)

LARRANÃGA, P. *et all* – **Genetic Algorithm for the Travelling Saelsman Problem – A Review of Representations and Operators** – ACM Press New York, NY, USA, Issue 2 1998.

(LEWIS & PAPADIMITRIOU, 2000)

LEWIS, H. R., PAPADIMITRIOU, C. H., – **Elementos de Teoria da Computação** – trad. Edson Furmankiewicz, 2ed, Bookman, Porto Alegre-RS, 2000.

(LOHN *et all*, 2000)

LOHN, JASON D., COLOMBANO, SILVANO P., HAITH, GARY L., STASSINOPOULOS, DIMITRIS – **A Parallel Genetic Altgorithm for Automated Eletronic Circuit Design** – Computacional Sciences Division – Recom Technologies, Inc. NASA Ames Research Center. Moffett Field, CA. Email: jlohn@ptolemy.arc.nasa.gov

(MADISSETI *et all,* 1991)

MADISSETI VIJAV k, WARLAND, JEAN C., MESSERSCHMITT, DAVID G – **Asynchronous algorithms for the parallel simulation of event-driven dynamical systems** – ACM Transactions on Modeling and Computer Simulation (TOMACS) ACM Press New York, NY, USA,Volume 1 , Issue 3 (July 1991) .

(MAZZUCCO, 1999)

MAZZUCCO, JOSÉ JUNIOR. – **Uma abordagem híbrida do problema da programação da produção através dos algoritmos genéticos e Simulated Annealing** – Tese de Doutorado, Florianópolis : UFSC, 1999.

(METAXAS, 1995)

METAXAS, P. TAKIS – **Fundamental ideas for a parallel computing co*urse*** – . ACM Press New York, NY, USA. p 284 - 286 Periodical-Issue-Article, 1995

(MIKI *et all,* 1999)

MIKI, M., HIROYASU, T., KANEKO, M., IIATANAKA, K. – **A Parallel Genetic Algorithm with Distributed Enviroment Scheme** – Departament of Knowledge Engineering, Doshiha University, Kyo-tanabe, Kyto, Japan 1999. IEEE press 0-7803-5731-0/99

(MÜHLENBEIN, 2000)

MÜHLENBEIN, HEINZ – **Evolution in Time and Space – The Parallel Genetic Algorithm** – GMD Schloss Birlinghoven, D-5205 Sankt Augustin l.

(MÜHLENBEIN, 2000)

MÜHLENBEIN, HEINZ – **Parallel Genetic Algorithm in Combinatorial Optimization** – GMD Schloss Birlingghoven, D-5205 Sankt Augustin l.

(NOWOSTAWSKI & POLI, 1999)

NOWOSTAWSKI, MARIUSZ & POLI, RICCARDO – **Parallel Genetic Algorithm Taxonomy** – University of Otago, New Zealand, 1999. The University of Birmingham, Birmingham 1999

(PRICE, 1994)

PRICE, KENNETH V. – **Genetic Annealing** – Dr Dobb's Journal, October 1994 p.127–132.

(PRINCETON, 2002)

WEB-SITE Universidade de Princeton – **http://www.math.princeton.edu/tsp/history.html** – Visitado em 27/02/2002.

(PREUS, 1998)

PREUS, EVANDRO – **MDX: Um Ambiente de Programação Paralela, Baseado em Memória Virtual Distribuída** – Dissertação de Mestrado em Informática, PUC – Pontifícia Universidade Católica, Porto Alegre RS, 1998

(QUINN, 1994)

QUINN, M. J. – **Parallel Computing: theory and pratice** – McGraww-Hill, New York, 2ªEd. 1994.

(RIDLEY, 1996)

RIDLEY, M. – **Evolution** – Second Edition, Blackwell Science, EUA, 1996.

(ROUTO, 1991)

ROUTO, Terada. – **Desenvolvimento de algoritmos e estruturas de dados** – São Paulo, McGraw-Hill, Makron, 1991.

(SANTOS, 2001)

SANTOS, RICARDO R. – **Escalonamento de Aplicações Paralelas: Interface AMIGO-CORBA** – Dissertação de Mestrado, Instituto de Ciências Matemática e de Computação, USP - Universidade de São Paulo, SP, Novembro/2001.

(SUN, 1997)

SUN EDUCATIONAL SERVICES – **Desenvolvimento de Aplicações Java SL-276 Apostila do Estudante** – Sun Microsystems do Brasil, São Paulo, 1997.

(TANNEMBAUM, 1991)

TANNEMBAUM, ANDREWS S. – **Sistemas Operacionais Modernos** –, Makron Books, São Paulo, SP. 1991.

(TANOMARU, 1995)

TANOMARU, J., – **Motivação, Fundamentos e Aplicações de Algoritmos Genéticos** – II Congresso Brasileiro de Redes Neurais,Curitiba-Pr, 1995.

(TURTON *et all*, 1994)

TURTON, B.C.H, ARSLAN, T., HORROCKS, D.H. – **A Hardware Archieture for a Parallel Genetic Algorithm fo Image Registration** – School of Engineering, University of Wales College of Cardiff, Cardiff U.K., Printed and Published by the IEE, Savoy Place, London WC2R OBL, UK. 1994.

(ZEBULUN, 1999)

ZEBULUM, RICARDO S. – **Sintese de Circuitos Eletrônicos por Computação Evolutiva** – Tese de doutorado, PUC-RIO - Departamento de Engenharia Elétrica, Rio de Janeiro, 15 de Abril de 1999.

(ZUBEN, 2000)

ZUBEN, FERNANDO J. VON. – **Computação Evolutiva: Uma Abordagem Pragmática** – Tese de Doutorado, DCA/FEEC/Unicamp, 2000.