DEPARTAMENTO DE COMPUTAÇÃO

D E C O M

Aplicação de Algoritmos Genéticos ao Problema Job-Shop

Wendrer Carlos Luz Scofield – 98.2.4996

Prof. Dr. Marcone Jamilson Freitas Souza

Relatório Técnico 02 / 2001

U F O P

UNIVERSIDADE FEDERAL DE OURO PRETO

FOLHA DE APROVAÇÃO DA BANCA EXAMINADORA

Aplicação de Algoritmos Genéticos ao Problema Job-Shop

Wendrer Carlos Luz Scofield

Monografia apresentada ao Departamento de Computação do Instituto de Ciências Exatas e Biológicas da Universidade Federal de Ouro Preto como requisito da disciplina COM 951 – Projeto Orientado II, do curso de Bacharelado em Ciência da Computação, e aprovada pela Banca Examinadora composta pelos seguintes membros:

_____

Prof. Dr. Marcone J. F. Souza
Orientador
Departamento de Computação – UFOP

Ouro Preto, 06 de maio de 2002

A Deus, com todo o meu carinho

Agradecimentos

Agradeço, em especial, a minha mãe, a minha sogra Tereza Rosa e a minha namorada Daniela Rosa pelo apoio que foi relevante durante o desenvolvimento do projeto.

Agradeço a Deus por todos os momentos juntos ao longo dessa trajetória.

Agradeço também ao meu orientador Marcone Souza, pela oportunidade oferecida com este trabalho em atuar em outras áreas do curso Ciência da Computação combinando Pesquisa Operacional e Inteligência Artificial, áreas em que eu não tinha conhecimento prático até então.

Resumo

Dentre os principais fatores que compõem o controle e o gerenciamento de uma produção industrial encontra-se a programação dessa produção.

Este fator, especialmente em se tratando de programação da produção do tipo Job-Shop, se traduz em uma tarefa árdua devido a sua grande complexidade. Trata-se de um dos problemas mais difíceis de otimização combinatorial conhecidos.

O problema da programação da produção do tipo Job-Shop pode ser descrito da seguinte forma. São dados um conjunto de jobs e um conjunto de máquinas. Cada job consiste de uma cadeia de operações em que cada uma das quais deve ser processada durante um período de tempo ininterrupto, de um dado tamanho, em uma dada máquina. Cada máquina pode processar no máximo uma operação por vez. Um programa consiste da alocação das operações em cada máquina. O problema é determinar um programa que realize todas as operações, no menor tempo possível.

Atualmente observa-se uma forte tendência em se utilizar métodos aproximados na resolução desta classe de problemas. Uma abordagem dessa natureza, apesar de não garantir uma solução ótima para um problema, é capaz de oferecer uma solução de boa qualidade, em um tempo de processamento aceitável.

Neste projeto, foi proposto e investigado a potencialidade do método de aproximação Algoritmos Genéticos, com o objetivo de obter boas soluções a curto prazo para o problema Job-Shop sem exigir grandes recursos computacionais.

Para utilizamos como referência nos estudos sobre os pontos atacados dos Algoritmos Genéticos tradicionais, foram desenvolvidos o Algoritmo Aleatório para criação de soluções aleatórias e o Algoritmo Guloso que gera uma única solução com um certo teor de qualidade, segundo o critério de escolha Menor Tempo Acumulativo. Os critérios de escolha adotados são amplamente discutidos nesse projeto, uma vez que, constitui o principal fator de controle de qualidade na geração de soluções, para o problema Job-Shop, obtidas pelo Algoritmo Genético sofisticado. Um método de aproximação foi desenvolvido combinando tais conceitos herdados dos algoritmos Guloso e Aleatório com os conceitos de Evolução Natural herdados dos Algoritmos Genéticos puros com o objetivo de avaliarmos a eficiência adquirida pelo Algoritmo Genético híbrido resultante. Serão comparados resultados obtidos pelos algoritmos Guloso e Aleatório, o Algoritmo Genético puro e o Algoritmo Genético híbrido.

Palavra Chave

Job-Shop – job – Algoritmos Genéticos – Algoritmo Aleatório – Algoritmo Guloso – Menor Tempo Acumulativo – Evolução Natural

Lista de Figuras

Lista de Tabelas

Índice

1 Introdução

Seqüenciamento são formas de tomada de decisão que possuem um papel crucial nas empresas, tanto de manufatura como de serviços. No atual ambiente competitivo, o efetivo seqüenciamento se tornou uma necessidade para sobrevivência no mercado. Companhias devem esforçar-se ao máximo para cumprir as datas firmadas com seus clientes. O fracasso deste comprometimento pode resultar em uma perda significante da imagem da empresa perante os clientes (Pinedo&Chão,1999).

O seqüenciamento das atividades ou *scheduling* é uma das atividades que compõem o planejamento da produção (programação da produção). Na programação da produção são levados em consideração uma série de elementos que disputam vários recursos por um período de tempo, recursos esses que possuem capacidade limitada.

Os elementos a serem processados são chamados de ordens de fabricação ou jobs (trabalho) e são compostos de partes elementares chamadas tarefas ou operações. Os principais objetivos tratados no problema de seqüenciamento podem ser resumidos no atendimento de prazos (ou datas de entrega), na minimização do tempo de fluxo dos estoques intermediários e na maximização da utilização da capacidade disponível, ou mesmo na combinação destes objetivos (Walter,1999).

A maioria dos problemas de programação da produção estudados aplica-se ao ambiente conhecido como Job-Shop.

O problema Job-Shop (JSP) é um problema de alocação de um conjunto de jobs para as máquinas, de tal forma que os jobs sejam executados em um menor intervalo de tempo. Cada job pode consistir de diversas tarefas e cada tarefa deve ser processada numa máquina particular. Além disso, as tarefas em cada job estarão sujeitas à restrições de precedência..

Devido aos grandes esforços computacionais exigidos por modelos matemáticos de otimização para obtenção de solução ótima para o problema, propomos o estudo de um algoritmo heurístico, que tem como principio a Evolução Natural, para obtenção de boas soluções sem exigir grandes esforços computacionais.

Os métodos genéticos podem ser vistos como técnica de otimização de soluções. Eles são baseados na evolução natural, utilizando uma estratégia de gerar-e-testar muito elegante, permitindo uma pesquisa paralela no espaço de soluções possíveis. “Quanto melhor um indivíduo se adaptar ao seu meio ambiente, maior será sua chance de sobreviver e gerar descendentes” : este é o conceito básico da evolução genética biológica. A área biológica mais proximamente ligada aos Algoritmos Genéticos é a Genética Populacional. Estes métodos são estocásticos e não garantem a determinação de uma solução ótima, entretanto, na pratica, eles são amplamente utilizados, com sucesso, em inúmeras aplicações. (Mitchell,1997).

No Apêndice A “Glossário Genético” é exibido um glossário voltado para a área de Algoritmos Genéticos definindo termos comuns utilizados pela Biologia.

No Apêndice B “Código Fonte dos Programas” é exibido o código-fonte do Algoritmo Genético híbrido implementado na linguagem de programação C++.

No Apêndice C “O Programa Job-Shop” um breve comentário a respeito de entrada/saída de dados para o programa Job-Shop é relatado. É exibido uma instância do Problema Job-Shop e o melhor indivíduo gerado para um número de gerações percorridas pelo Algoritmo Genético híbrido.

2 Atividades e Metodologias

O desafio nesse projeto é produzir, em tempo mínimo, soluções tão próximas quanto possível da solução ótima para o problema JSP, utilizando como heurística os Algoritmos Genéticos que deverão passar por processos de sofisticação.

Atividades a serem realizadas:

1. Implementação Básica do Problema JSP

O problema Job-Shop (JSP) apresenta como implementação básica os algoritmos Guloso e Aleatório e o Algoritmo Genético tradicional implementados para utilizarmos como referência na avaliação e implementação do Algoritmo Genetico sofisticado. Testes serão realizados comparando resultados obtidos dos algoritmos implementados.

A justificativa para o desenvolvimento dos algoritmos Guloso e Aleatório :

- Utilização dos algoritmos Guloso e Aleatório para comparar os resultados obtidos com os resultados adquiridos dos Algoritmos Genéticos tradicional e sofisticado.
- A combinação dos critérios de escolha adotados pelos algoritmos Guloso e Aleatório ao Algoritmo Genético básico com a finalidade de obtermos um algoritmo genético híbrido que ofereça melhores possibilidades de geração de melhores soluções a um curto prazo de tempo para o problema tratado.

2. Implementação Sofisticada

Algumas modificações foram realizadas no Algoritmo Genético tradicional, sem perder a característica evolucionária do algoritmo heurístico, com a intenção de obtermos soluções cada vez mais próximas da solução ótima para o problema em menor intervalo de tempo. As modificações realizadas implementarão o Algoritmo Genético sofisticado.

Neste projeto apresentamos um algoritmo baseado em regras heurísticas, Algoritmo Genético, sofisticado e híbrido que herda conceitos dos algoritmos Guloso e Aleatório desenvolvidos (que não adotam heurísticas). Essa é a principal modificação adotada no Algoritmo Genético tradicional. A combinação dos conceitos adotados pelo Algoritmo Aleatório que utiliza o critério de escolha Aleatória e dos conceitos adotados pelo Algoritmo Guloso que utiliza o critério de escolha Menor Tempo Acumulativo, ambos os conceitos abordados no projeto, faz uso de uma metodologia adotada pelos algoritmos heurísticos GRASP, também abordado nesse trabalho, para geração de soluções iniciais a serem utilizadas como parâmetro pelo Algoritmo Genético que passa a ter uma natureza híbrida (combinação de metodologias GRASP e Algoritmo Genético).

3. Experimentos Computacionais

Os experimentos serão realizados comparando os resultados obtidos das implementações básicas, para o problema Job-Shop, com os resultados obtidos pelo Algoritmo Genético sofisticado e híbrido, gerado pela combinação dos algoritmos básicos, apresentado nesse projeto.

As implementações básicas não heurísticas são : Algoritmo Aleatório e Algoritmo Guloso. A implementação básica heurística é o Algoritmo Genético tradicional.

Avaliaremos as modificações sofridas pelo Algoritmo Genético tradicional sem que o algoritmo genético perca a sua natureza evolucionária na busca de melhores soluções.

3 A Classe do Problema Tratado

Apresentaremos, especificamente, a classe do problema, conhecida como Job-Shop, que se pretende tratar nesse projeto. Para tanto, recorreu-se às definições e premissas existentes na teoria clássica da programação.

3.1 Definições e Premissas

Descreve-se, a seguir, um problema da programação da produção do tipo Job-Shop, em sua forma básica.

Um conjunto de *n jobs* $J = \{ J_1, J_2, \ldots , J_n \}$ deve ser processado em um conjunto de *m* máquinas $M = \{ M_1, M_2, \ldots , M_m \}$ disponíveis. Cada job possui uma ordem de execução específica entre as máquinas, ou seja, um job é composto de uma lista ordenada de operações, cada uma das quais definida pela máquina requerida e pelo tempo de processamento na mesma. As restrições que devem ser respeitadas são:

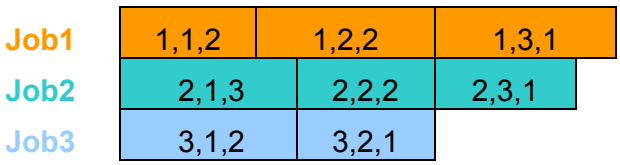
- Operações não podem ser interrompidas e cada máquina pode processar apenas uma operação de cada vez;
- Cada job só pode estar sendo processado em uma única máquina de cada vez.
- Cada job é fabricado por uma seqüência conhecida de operações.
- Não existe restrição de precedência entre operações de diferentes jobs.
- Não existe qualquer relação de precedência entre as operações executadas por uma mesma máquina.

Uma vez que as seqüências de máquinas de cada job são fixas, o problema a ser resolvido consiste em determinar as seqüências dos jobs em cada máquina, de forma que o tempo de execução transcorrido, desde o início do primeiro job até o término do último, seja mínimo. Essa medida de qualidade de programa, conhecida por *makespan*, não é a única existente, porém é o critério mais simples e o mais largamente utilizado. (Rodammer&White,1988). O objetivo é encontrar a solução com menor *makespan* (tempo de finalização), ou seja, a melhor solução para o problema.

Normalmente o número de restrições é muito grande, o que torna o Job-Shop um dos mais difíceis problemas. Testes para problemas maiores são mais difíceis de obter o agendamento ótimo conhecido, ou seja, mais difícil é obter o arranjo de todas as tarefas nas máquinas que satisfaz as restrições de precedência cujo processamento se faz em um menor tempo. É por isso que é tão difícil avaliar a performance de algoritmos de agendamento. (Ansi,1997).

Um Problema Job-Shop é normalmente referenciado como um *n/m* JSP, onde *n* é número de trabalhos e *m* o número de máquinas.

Será apresentado, a seguir (Figura 3.1) um exemplo ilustrativo de um problema de programação do tipo Job-Shop, problema *3/3* JSP, e um agendamento para o problema na Tabela 3.1.



- Cada linha (job) representa um trabalho e cada operação X,Y,Z representa uma tarefa do trabalho.
- Para cada operação X,Y,Z :
  - Z – máquina em que a operação deverá ser executada.
  - Y – número da operação.
  - X – job.
- O tamanho de cada bloco é proporcional ao tempo de processamento requerido para executar a operação. A tabela a seguir exibe os tempos de processamento de cada operação em unidades de tempo u.

Operação (X,Y,Z)	Tempo de Processamento (u)
1,1,2	5
1,2,2	7
1,3,1	7
2,1,3	7
2,2,2	5
2,3,1	5
3,1,2	7
3,2,1	5

Figura 3.1 – Problema 3/3 JSP

A seguir uma solução para o Problema *3/3* JSP.

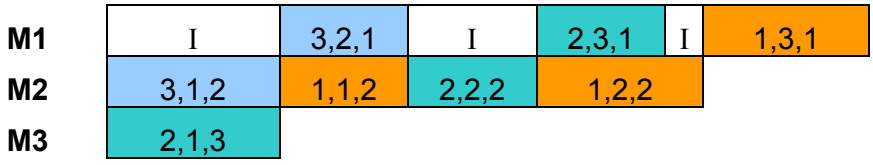


Tabela 3.1 – Agendamento para o Problema 3/3 JSP

- I – intervalo de ociosidade.

A Tabela 3.1 é o gráfico de Gantt que representa uma das soluções para o problema expresso na figura 3.1. É importante observar que a ordem requerida das operações dentro de cada job (a seqüência tecnológica) foi preservada na solução apresentada. Por exemplo, o job 2 foi processado nas máquinas 3, 2 e 1, respectivamente, como especificado na figura 3.1.

Sob as suposições, usualmente consideradas, de que todos os jobs estão prontos para serem processados no início do processamento e de que todas as operações requerem o uso exclusivo de uma máquina que não pode ser compartilhada, a solução exemplo apresentada na tabela 3.1 representa uma seqüência de operações que requer um tempo total de 31 unidades de tempo para completar o processamento de todos os 3 jobs.

3.2 Scheduling num Ambiente Job-Shop

O problema Job-Sshop Scheduling é classificado na literatura como NP-difícil, isto é, é um problema para o qual não existem algoritmos que o resolva em tempo polinomial. Trata-se de um "problema de otimização combinatória" (Souza,2001). Utiliza-se de uma enumeração explícita ou implícita de todas as alternativas possíveis a procura de soluções satisfatórias ou a procura da solução ótima, dependendo da classe do algoritmo de otimização utilizado.

Os algoritmos de otimização podem ser classificados em duas classes: os modelos matemáticos (procura a solução ótima) e os heurísticos (procura por aproximação).

- **Modelos Matemáticos** : Modelo que enfatiza a obtenção de resultados ótimos em função de algum parâmetro de desempenho. Este pode ser, por exemplo, a minimização dos tempos de produção ou a maximização do uso dos recursos. Dependendo da complexidade do problema tratado, pode consumir muito tempo para obter a solução ótima.
- **Modelos Ponto-a-Ponto** : A busca pela melhor solução se processa sempre de um único ponto para outro (inicia-se com um único candidato), no espaço de decisão, através da aplicação de alguma regra de transição, ou seja, o método realiza a busca sempre na vizinhança do ponto corrente, constituindo, portanto, um método de busca local. Essa característica dos métodos ponto-a-ponto constitui um fator de risco, uma vez que, em um espaço com vários picos, é grande a probabilidade de um falso pico ser retornado como solução.
- **Modelos Heurísticos** : Modelo baseado em regras práticas (indutivas) de escalonamento que enfatiza a obtenção de "boas" soluções, próxima da solução ótima, sem, no entanto, garantir a solução ótima. Os modelos híbridos são caracterizados pela obtenção de boas soluções para um problema em tempos de computação viáveis.

A escolha do modelo mais adequado à solução de problemas reais do tipo Job-Shop deve levar em consideração alguns aspectos como :

- O número de máquinas envolvidas.
- Os roteiros das ordens de fabricação (jobs).
- O regime de chegada das ordens.
- A variabilidade dos tempos de processamento entre outras características.

(Pacheco&Santoro,1999).

Devido ao grande número de soluções possíveis e à complexidade do problema de agendamento (*scheduling*), torna-se difícil e até impossível modelar todas as variáveis envolvidas no processo utilizando modelos matemáticos. Além disso, o tempo de resposta para os modelos matemáticos eleva-se consideravelmente, tornando impraticável a obtenção de soluções ótimas em tempos satisfatórios. Dessa forma, algoritmos otimizantes matemáticos são computacionalmente viáveis quando aplicados a problemas reais pequenos, com objetivos limitados.

Para problemas de porte similar aos encontrados no ambiente real, costuma-se sacrificar a obtenção de uma solução ótima por métodos heurísticos, que resultem em uma solução subótima ("boa" solução), com tempo computacional aceitável.

Nos modelos heurísticos, ditos satisfatorizantes, pois não garantem que a solução encontrada seja a ótima, mas sim muito próxima desta, quando há mais de uma atividade disputando um recurso, cabe a uma regra selecionar a atividade a atender primeiro, de forma a otimizar o programa de acordo com algum critério. (Morton&Pentico,1993).

Em virtude das dificuldades intrínsecas aos problemas de otimização combinatória, pesquisadores têm concentrado esforços na utilização de heurísticas para solucionar problemas desse nível de complexidade e é também o motivo pelo qual utilizaremos heurísticas para o problema Job-Shop, nesse projeto, a fim de encontrarmos soluções satisfatórias.

3.3 Heurística Aplicada

Como heurística, iremos adotar os *Algoritmos Genéticos* como meio de obtermos resultados melhores para o problema JSP sem exigir grandes recursos computacionais.

Os Algoritmos Genéticos fundamentam-se em uma analogia com processos da evolução biológica natural, nos quais, dada uma população, os indivíduos com características genéticas melhores têm maiores chances de sobrevivência e de produzirem filhos cada vez mais aptos, enquanto indivíduos menos aptos tendem a desaparecer. Tais algoritmos operam numa população de soluções potenciais aplicando o princípio de *ajustamento de sobrevivência*, que possibilita as soluções mais aptas terem maiores condições de serem selecionadas para produzir cada vez mais aproximações melhores para a solução. A cada geração, um novo conjunto de aproximações é criado pelo processo de seleção de indivíduos nos quais são aplicados os operadores de natureza genética de reprodução. Este processo cobre a evolução de populações de indivíduos que são mais compatíveis com seus ambientes do que indivíduos que os criaram, tal como da adaptação natural (Neto,1997).

4 Algoritmos Genéticos

Para que se consiga entender o algoritmo genético é conveniente que se conheça um pouco dos processos biológicos observados na evolução natural.

4.1 Evolução Natural

O cientista inglês Charles Darwin estudando as espécies e suas evoluções, coletou durante anos um volumoso material que demonstrou, principalmente, a existência de inúmeras variações em cada espécie. Seus estudos, aliados às pesquisas de outros cientistas no assunto, tornaram evidente que as espécies realmente se modificam. (Darwin,1953).

Um dos principais pontos dos estudos de Darwin foi sem dúvida o aspecto das variações apresentadas entre indivíduos da mesma espécie. Através de estudos de pombos, por exemplo, ele verificou a enorme variedade que se obtém cruzando uma mesma espécie. Segundo Darwin(1953), novas variedades são produzidas por meio da seleção natural. As variações mais favoráveis de cada espécie conseguem sobreviver com mais facilidade, uma vez que têm mais chance de se reproduzirem.

Muitas experiências comprovam que a seleção natural é um fato incontestável, podendo não ser o único mecanismo evolutivo, porém sem dúvida um dos fatores principais do processo.

Uma das principais constatações dessa evolução foi alcançada através de uma experiência realizada em laboratório, onde uma placa contendo cem milhões de bactérias foi posta em contato com uma dose de penicilina insuficiente para o combate total das mesmas. Observou-se que as dez bactérias que haviam sobrevivido se reproduziram na própria placa, e que essa nova geração sobreviveram ao meio, mesmo com a presença da penicilina. Dobrou-se então a quantidade do antibiótico e observou-se que quase todas as bactérias morreram. Novamente as poucas bactérias que sobreviveram a nova dosagem passaram a se multiplicar, sendo novamente expostas à dosagens cada vez mais altas. Esse processo se repetiu por cinco gerações e ao final da experiência, apresentou-se uma linhagem de bactérias resistentes a uma dose de penicilina duas mil e quinhentas vezes maior do que a inicial, utilizada na primeira cultura. É importante observar que as bactérias não desenvolveram individualmente resistência à penicilina, eram descendentes das poucas bactérias que herdaram uma característica favorável à sua sobrevivência naquele meio, havendo, portanto, um processo biológico de adaptação ao meio.

Embora os mecanismos que conduzem a evolução natural não estejam ainda plenamente compreendidos, algumas de suas importantes características são conhecidas. A evolução dos seres vivos se processa nos cromossomos (dispositivos orgânicos onde as estruturas desses seres são codificadas). Parte da criação de um ser vivo é realizada através de um processo de decodificação de cromossomos.

Muito embora, os processos específicos de codificação e decodificação de cromossomos também não estejam ainda totalmente esclarecidos, existem algumas características gerais na teoria desse assunto que estão 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 se reproduzir mais freqüentemente do que os mal sucedidos;
- É no processo de reprodução que a evolução se realiza. Através de mutação (alterações aleatórias de genes) o cromossomo de um ser descendente podem ser diferente do cromossomo de seu gerador. Através do processo de recombinações (*crossover*) dos cromossomos de dois seres geradores, é também possível que os cromossomos do ser descendente se tornem muito diferentes daqueles dos seus geradores e
- A evolução biológica não possui memória. A produção de um novo indivíduo depende apenas de uma combinação de genes da geração que o produz.

4.2 Introdução aos Algoritmos Genéticos

Holland (1993), convenceu-se de que as características da evolução biológica poderiam ser incorporadas a um algoritmo para constituir um método extremamente simples de resolução de problemas complexos, imitando o processo natural, ou seja, através da evolução.

Os algoritmos genéticos foram inventados por John Holland nos anos 60 e desenvolvido por ele e seus alunos na Universidade de Michigan em meados de 1970. O principal objetivo de Holland não foi desenvolver algoritmos para solucionar problemas específicos, mas dedicar-se ao estudo formal do fenômeno da evolução, como ocorre na natureza, e desenvolver maneiras de importá-lo aos sistemas de computação.

Porém, este algoritmo poderia solucionar qualquer problema que apresentasse as mesmas características da evolução. O sistema trabalhava com uma população de algumas cadeias de bits (0´s e 1´s) denominadas cromossomos.

Semelhante à natureza, o sistema evoluiu até o melhor cromossomo para atender um problema específico, mesmo sem saber que tipo de problema estava sendo solucionado. A solução foi encontrada de um modo automático e não-supervisionado, as únicas informações dadas ao sistema foram os ajustes de cada cromossomo produzido por ele.

Desde então, os algoritmos genéticos vêm sendo aplicados com sucesso nos mais diversos problemas de otimização e aprendizado de máquina.

4.3 Motivação

Algoritmos genéticos são amplamente utilizados devido a vários fatores. Dentre eles podemos citar :

- **Algoritmos genéticos são facilmente paralelizados** : cada indivíduos da população pode ser alocado em um processador diferente ou processador virtual em uma máquina paralela (Berson&Stephen,1997). A aptidão de cada indivíduo é avaliada localmente em seu processador correspondente. Um indivíduo que possui um valor de aptidão baixo pode ser substituído na próxima geração. A seleção de indivíduos é realizada da mesma forma que a seleção em um algoritmo genético não paralelizado. Os operadores genéticos podem ser aplicados também, da mesma forma que são aplicados em algoritmos genéticos não paralelizados.
- **Algortimos genéticos são baseados na teoria da evolução natural** : a teoria da evolução natural é um sucesso. Uma prova disto somos nós humanos. (Mitchell,1997).
- **Algoritmos genéticos são estratégias de otimização global** : algoritmos genéticos não otimizam (evoluem) apenas um indivíduo a cada geração e sim uma população de vários indivíduos. Um individuo é a representação de uma solução. A cada geração os algoritmos genéticos retornam várias representações de soluções. Isto significa que os algoritmos genéticos executam uma busca no espaço de soluções de forma paralela, porque buscam várias soluções simultaneamente. (Mitchell,1997).

4.4 Características Gerais

Toda tarefa de busca e otimização possui vários componentes, entre eles: o espaço de busca, onde são consideradas todas as possibilidades de solução de um determinado problema e a função de avaliação, uma maneira de avaliar os membros do espaço de busca.

As técnicas de computação evolucionária operam sobre uma população de candidatos em paralelo.

Em contraste aos modelos ponto-a-ponto, onde a busca pela melhor solução se processa sempre de um único ponto para outro no espaço de decisão, os Algoritmos Genéticos trabalham simultaneamente sobre um rico banco de pontos (uma população de cromossomos), subindo vários picos em paralelo e reduzindo, dessa forma, a probabilidade de ser encontrado um falso pico.

Os Algoritmos Genéticos (AGs) são técnicas de computação evolucionária que diferem dos métodos tradicionais de busca e otimização, principalmente em quatro aspectos:

- AGs trabalham com uma codificação do conjunto de parâmetros e não com os próprios parâmetros.
- AGs trabalham com uma população e não com um único ponto.
- AGs utilizam informações de custo ou recompensa e não derivadas ou outro conhecimento auxiliar.
- AGs utilizam regras de transição probabilísticas e não determinísticas.

Algoritmos Genéticos são muito eficientes para busca de soluções ótimas, ou aproximadamente ótimas, em uma grande variedade de problemas, pois não impõem muitas das limitações encontradas nos métodos de busca tradicionais.

Além de ser uma estratégia de gerar-e-testar muito elegante, por serem baseados na evolução biológica, são capazes de identificar e explorar fatores ambientais e convergir para soluções ótimas, ou aproximadamente ótimas em níveis globais.

"Quanto melhor um indivíduo se adaptar ao seu meio ambiente, maior será sua chance de sobreviver e gerar descendentes" : este é o conceito básico da evolução genética biológica.

Algoritmos genéticos são algoritmos de otimização global, baseados nos mecanismos de seleção natural. Eles empregam uma estratégia de busca paralela e estruturada, mas aleatória, que é voltada em direção ao reforço da busca de pontos de "alta aptidão", ou seja, pontos nos quais a função a ser minimizada (ou maximizada) tem valores relativamente baixos (ou altos).

Apesar de aleatórios, eles não são caminhadas aleatórias não direcionadas, pois exploram informações históricas para encontrar novo pontos de busca onde são esperados melhores desempenhos. Isto é feito através de processos interativos, onde cada iteração é chamada de geração.

Durante cada iteração, os princípios de seleção e reprodução são aplicados a uma população de candidatos que pode variar, dependendo da complexidade do problema e dos recursos computacionais disponíveis. Através da seleção, se determina quais indivíduos conseguirão reproduzir, gerando um número determinado de descendentes para a próxima geração, com uma probabilidade determinada pelo seu índice de aptidão. Em outras palavras, os indivíduos com maior adaptação relativa têm maiores chances de se reproduzir.

A representação do individuo, na forma de cromossomo, é feita normalmente através de um vetor, onde cada posição do vetor, chamada de gene, possui um elemento que denota uma característica do individuo : o seu genótipo. Esses elementos podem ser combinados ou sofrer alterações formando a característica real de um individuo : o seu fenótipo. Genótipos diferentes podem resultar em fenótipos satisfatórios para sobrevivência de um indivíduo.

A utilização de representações em níveis de abstração mais altos tem sido investigada. Como estas representações são mais fenotípicas, facilitariam sua utilização em determinados ambientes, onde essa transformação "fenótipo – genótipo" é muito complexa. Neste caso, precisam ser criados operadores genéticos específicos para utilizar estas representações.

4.4.1 Funcionamento

Os algoritmos genéticos básicos funcionam da seguinte forma:

1. Determinar uma população inicial de cromossomos.
2. Avaliar cada indivíduo da população através do cálculo de sua aptidão, utilizando a função de avaliação.
3. Selecionar os cromossomos que produzirão descendentes para a próxima geração, em função dos seus desempenhos apurados no passo anterior. (etapa de seleção).
4. Gerar descendentes através da aplicação das operações de cruzamento, *crossover* e mutação sobre os cromossomos selecionados no passo anterior, criando a próxima geração. (etapa de reprodução) e.
5. Se o critério de parada for satisfeito, termina e retorna o melhor cromossomo até então gerado, senão volta ao passo 2.

4.4.1.1 Inicialização da População

Uma população de indivíduos é composta por soluções iniciais codificadas na forma de cromossomos.

Inicialmente uma população de n indivíduos (cromossomos) é gerada de forma aleatória nos algoritmos genéticos básicos.. Também pode ser gerada através de algoritmos heurísticos.

4.4.1.1.1 Representando Geneticamente as Soluções Viáveis do Problema

As soluções a comporem a população inicial devem ser codificadas geneticamente, ou seja, representadas por cromossomos. Somente soluções viáveis (válidas) do problema a ser tratado devem ser codificadas.

O cromossomo representa uma particularidade do problema tratado por codificar soluções para o problema.

De acordo com a Biologia, cromossomos são extensas cadeias de compostos químicos que contém a descrição da composição genética dos seres vivos.

Do ponto de vista dos Algoritmos Genéticos, o cromossomo é uma solução candidata, ou seja, um ponto no espaço de busca. Cada cromossomo codifica uma solução do problema e seu valor contribui para o cálculo da função objetivo que determina o *makespan* de uma solução para uma instância do problema Job-Shop.

Em muitas aplicações, um cromossomo consiste de um *string* de codificação binária.

Exemplo : problema do caixeiro viajante, temos :

Número de cidades = 3

Cidade	Código Genético
A	001
B	010
C	100

Tabela 4.1 – Esquema de Codificação

Como sabemos, o problema consiste em encontrar o menor caminho, partindo da cidade A, percorrendo todas as outras cidades e retornando à cidade A. Um indivíduo seria representado por quatro atributos cidades. O número de bits do atributo cidade é igual ao número de valores diferentes que o atributo cidade pode receber de cada vez. O atributo cidade pode receber três valores diferentes (cidade A,B,C) de cada vez. Portanto, o número de bits do atributo é igual a três.

Um exemplo de cromossomo e indivíduo para o problema seria :

	Cidade1	Cidade2	Cidade3	Cidade4
Individuo	A	C	B	A
Cromossomo	001	100	010	001

Tabela 4.2 – Indivíduo & Cromossomo

Cromossomo = 001100010001

Indivíduo = ACBA

O cromossomo 001100010001 é uma solução válida para o problema, ou seja, representa um individuo para o problema.

As soluções candidatas também podem ser implementadas em codificações não binárias, com o objetivo de melhorar a adaptação ao tipo de problema abordado.

4.4.1.2 Avaliação

Uma função de aptidão atribui a cada indivíduo um valor, denominado valor de aptidão do indivíduo.

A função de aptidão é uma função de avaliação.

A função de aptidão aplicada a cada indivíduo retorna um valor que representa sua adaptabilidade ao meio, isto é, sua aproximação da solução ótima. É criada de acordo com a necessidade do problema a ser otimizado, portanto, ela serve para problemas específicos. Como vimos anteriormente, a característica assumida pelo cromossomo é conhecida como genótipo de um indivíduo. A característica que o cromossomo atribui ao individuo é conhecido como fenótipo. Daí conclui-se que a função de aptidão analisa o fenótipo de um indivíduo de acordo com o problema, e retorna um valor correspondente.

4.4.1.3 Seleção

Selecionar indivíduos da população.

Indivíduos são selecionados de acordo com seu valor de aptidão. O princípio básico do funcionamento dos algoritmos genéticos é que um critério de seleção vai fazer com que, depois de muitas gerações, o conjunto inicial de indivíduos gere indivíduos mais aptos.

4.4.1.3.1 Métodos de seleção

A escolha dos indivíduos geradores (os pais) pode ser feita de inúmeras formas. Dentre os métodos mais importantes, destacam-se :

a) **Seleção por Ranking** (*Rank Selection*):

   Os indivíduos da população são ordenados de acordo com seu valor de adequação e então sua probabilidade de escolha é atribuída conforme a posição que ocupam.

b) **Seleção por Giro de Roleta** (*Roulette Wheel Selection*):

   Um método de seleção muito utilizado é o Giro de Roleta, onde indivíduos de uma geração são escolhidos para fazer parte da próxima geração, através de um sorteio de roleta. Os indivíduos são representados na roleta proporcionalmente ao seu índice de aptidão. Finalmente, a roleta é girada um determinado número de vezes, dependendo do tamanho da população, e são escolhidos como indivíduos que participarão da próxima geração, aqueles sorteados na roleta. Os indivíduos de melhores aptidão têm maiores tendências de serem escolhidos, ou seja, a seleção por Giro de Roleta segue os princípios naturais da evolução.

c) **Seleção por Torneio** (*Tournament Selection*):

Grupos de soluções são escolhidos sucessivamente e as mais adaptadas dentro de cada um destes são selecionadas (Goldberg,1989) (Geyer,1997).

d) **Seleção Uniforme**:

Todos indivíduos possuem a mesma probabilidade de serem selecionados (Goldberg,1989). Obviamente, esta forma de seleção possui uma probabilidade muito remota de causar a evolução da população sobre a qual atua.

e) **Remainder Stochastic Selection** :

A partir da fórmula:

$s(I) = Fadap(I) / Madap_t$

: onde *Fadap(I)* é a aptidão do indivíduo *I*, $Madap_t$ é a aptidão média da população no tempo *t,*

se assume que a parte inteira da função s(I) determina quantas cópias do indivíduo I são selecionadas diretamente. O restante de indivíduos a serem selecionados é escolhido aplicando um método de seleção como o giro de roleta sobre a parte decimal do valor s(I) de cada indivíduo. (Goldberg,1989) (Geyer,1997).

4.4.1.4 Reprodução

Criar novos indivíduos a partir dos indivíduos selecionados.

A capacidade de reprodução é o principal fator de evolução dos seres vivos. Por evolução entende-se a adaptação de um individuo ao meio ambiente. Os Algoritmos Genéticos são sistemas adaptativos por excelência, ou seja, adaptam-se à determinadas condições que resultam na soluço de um problema específico. Os Algoritmos Genéticos forma inventados numa tentativa de mimetizar os processos observados na evolução natural.

A adaptação por reprodução incorre no fato dos seres vivos modificarem-se a medida que vão se reproduzindo (Darwinismo). A adaptação não ocorre de forma direta , ou seja, os indivíduos se modificam de acordo com as necessidades e essa modificação passa diretamente para os descendentes (Lamarkismo). O principio adotado pelos Algoritmos Genéticos é o processo de evolução por seleção natural, ou seja, os indivíduos mais adaptados ao meio ambiente têm maiores chances de sobreviver. Evolução biológica não possui memória. Todavia ela sabe como produzir seres vivos que se adaptam melhor ao seu meio ambiente. Este conhecimento está armazenado no grupo genético do indivíduo, responsáveis pelas características do indivíduo.

A etapa de reprodução consiste na aplicação dos operadores genéticos para formar os novos indivíduos a partir dos indivíduos selecionados.

A reprodução é efetuado em duas etapas. Na primeira são feitas cópias dos indivíduos selecionados como geradores da próxima população. O número de cópias feitas de cada indivíduo gerador será igual ao seu número de descendentes individual, sorteados por algum critério de seleção (mais comum o Giro de Roleta). Na segunda etapa, essas cópias serão agrupadas, duas a duas, de forma aleatória e sem reposição, constituindo os pares de geradores (os pais). Cada par de indivíduos, através da aplicação dos operadores genéticos,

produzirá dois descendentes para a geração seguinte. Os operadores genéticos básicos, neste passo, são cruzamento (crossover) e mutação e serão abordados com mais detalhes adiante.

Existem duas principais formas de reprodução :

- **Reprodução por Gerações**

  Nesta técnica de reprodução, a nova geração substitui inteiramente o lugar da primeira. Isto é, a partir dos indivíduos selecionados são gerados novos indivíduos que são copiados para a próxima geração. A população corrente é exterminada. Os novos indivíduos correspondem à população da próxima geração. Esta reprodução recebe este nome porque toda uma geração é substituída por uma nova geração.

- **Reprodução Elitista**

  Neste tipo de reprodução, os indivíduos selecionados são copiados automaticamente para a próxima geração através do operador genético clonagem.

  Os operadores genéticos geram novos indivíduos a partir dos indivíduos selecionados. Estes novos indivíduos também são copiados para a população da próxima geração.

  Esta reprodução recebe este nome porque os indivíduos selecionados na segunda etapa, isto é, os indivíduos de elite (valor de aptidão mais alto) são copiados para a população da próxima geração.

4.4.1.4.1 Operadores Genéticos

O principio básico dos operadores genéticos é transformar a população através de sucessivas gerações, estendendo a busca até chegar a um resultado satisfatório. Os operadores genéticos são necessários para que a população se diversifique e mantenha características de adaptação adquiridas pelas gerações anteriores.(Goldberg,1989).

Os operadores genéticos são responsáveis pela reprodução dos indivíduos de uma população. Mais precisamente, são responsáveis pela formação da população da próxima geração. Os operadores genéticos são:

- Clonagem
- Recombinação Genética ou Cruzamento
- Mutação

4.4.1.4.1.1 Clonagem

Este operador genético é responsável pela cópia de um ou mais indivíduos para a população da próxima geração. É utilizado principalmente na reprodução elitista.

Exemplo : Dado o cromossomo 001010100001, se ele for um indivíduo com alto valor de aptidão então será copiado para a próxima geração. A população sobrevivente a segunda geração conterá tal indivíduo.

4.4.1.4.1.2 Recombinação Genética ou Cruzamento

O operador genético cruzamento gera dois novos indivíduos a partir de dois indivíduos pais (selecionados na etapa de seleção).

O cruzamento é o operador responsável pela recombinação de características dos pais durante a reprodução, permitindo que as próximas gerações herdem essas características. Ele é considerado o operador genético predominante, por isso é aplicado com probabilidade dada pela taxa de cruzamento, que deve ser maior que a taxa de mutação.

O cruzamento é realizado por meio de uma máscara chamada máscara de cruzamento. A máscara é uma seqüência de bits sendo que o seu número de bits é igual ao número de dígitos de um cromossomo. Esta máscara é o modelo (molde) para o novo indivíduo. Os dois indivíduos pais são ordenados como primeiro e segundo para formar o primeiro indivíduo. Para formar o segundo indivíduo a ordem é trocada, Istoé, o primeiro passa a ser o segundo e assim por diante. Cada posição da máscara que possuir o bit 1 será preenchida pelo dígito de posição equivalente do primeiro cromossomo pai. E cada posição da máscara que possuir o bit 0 será preenchida pelo dígito de posição equivalente do segundo cromossomo pai. Portanto, um indivíduo será formado por partes dos dois pais, isto é, um indivíduo herda código genético dos dois pais.

Existem três tipos de cruzamento :

- Ponto Único
- Dois Pontos
- Uniforme

Ponto Único

Este método consiste em separar os cromossomos pais em duas partes. O ponto de separação nos cromossomos pai é determinado pela posição do último bit 1 da máscara, como é apresentado a seguir. O primeiro cromossomo filho é formado pela concatenação da primeira parte do primeiro cromossomo pai com a segunda parte do segundo cromossomo pai. O segundo cromossomo filho é formado pela concatenação da primeira parte do segundo cromossomo pai com a segunda parte do primeiro cromossomo pai. Neste tipo de cruzamento,

a máscara sempre será uma seqüência inicial de bits 1 terminando com uma seqüência de bit 0.

Cromossomos Pai	Máscara	Cromossomos Filho
001100010001	111111000000	001100100001
001010100001		001010010001

Tabela 4.3 – Cruzamento Ponto Único

Dois Pontos

Este método consiste em dividir os cromossomos pai em três partes. Os dois pontos de separação nos cromossomos pai são determinados pela posição do primeiro e último bits 1 da máscara, como é mostrado a seguir. O primeiro cromossomo filho é formado pela concatenação da primeira parte do segundo cromossomo, com a segunda parte do primeiro pai e com a terceira parte do segundo cromossomo pai. Neste tipo de cruzamento a máscara sempre será uma seqüência de bits 0, seguida de uma seqüência de bits 1 e terminando com uma seqüência de bits 0.

Cromossomos Pai	Máscara	Cromossomos Filho
001100010001	001111110000	001100010001
001010100001		001010100001

Tabela 4.4 – Cruzamento Dois Pontos

Uniforme

Segue a mesma idéia dos dois métodos anteriores.

Porém, neste tipo de cruzamento a máscara é gerada de forma aleatória. Portanto, os cromossomos pai são separados em n partes, como apresentado no exemplo a seguir.

Cromossomos Pai	Máscara	Cromossomos Filho
001100010001	100111010101	001100110001
001010100001		001010000001

Tabela 4.5 – Cruzamento Uniforme

4.4.1.4.1.3 Mutação

O operador genético mutação gera um novo indivíduo a partir de um único indivíduo pai.

O operador de mutação é necessário para a introdução e manutenção da diversidade genética da população, alterando arbitrariamente um ou mais componentes de uma estrutura escolhida, como é mostrado a seguir, fornecendo assim, meios para introdução de novos elementos na população. Desta forma, a mutação assegura que a probabilidade de se chegar a qualquer ponto do espaço de busca nunca será zero, além de contornar o problema de mínimos locais, pois com este mecanismo, altera-se levemente a direção da busca. O operador de mutação é aplicado aos cromossomos com uma probabilidade dada pela taxa de mutação : geralmente se utiliza uma taxa de mutação pequena, pois é um operador genético secundário, aplicado nos descendentes da população após uma operação crossover.

A mutação consiste em modificar o valor de um ou mais dígitos (genótipo) de uma determinada posição (gene) do cromossomo pai. Esta(s) posição(ões) são determinadas aleatoriamente.

Exemplo : Dado o cromossomo pai 001100010001 , aplicando a mutação na posição dois e dez teremos o cromossomo 011100010101.

4.4.1.5 Condição de Término

As etapas anteriores (exceto criação da população inicial) são repetidas até que uma condição seja satisfeita. Esta condição pode ser o número de gerações ou o valor de aptidão de algum individuo da população.

Condições de parada normalmente utilizados :

- Tempo de execução.
- Numero de gerações.
- Falta de diversidade.
- Ultimas k gerações sem melhora (convergência).

4.4.2 Parâmetros Genéticos

É importante também, analisar de que maneira alguns parâmetros influem no comportamento dos Algoritmos Genéticos, para que se possa estabelecê-los conforme as necessidades do problema e dos recursos disponíveis.

Tamanho da População

O tamanho da população afeta o desempenho global e a eficiência dos Algoritmos Genéticos. Uma população pequena oferece uma pequena cobertura do espaço de busca, causando uma queda no desempenho. Uma grande população fornece uma melhor cobertura representativa do domínio do problema e previne a convergência prematura para soluções locais. Entretanto, com uma grande população tornam-se necessários recursos computacionais maiores, ou um tempo maior de processamento do problema.

Taxa de Cruzamento

Quanto maior for esta taxa, mais rapidamente novas estruturas serão introduzidas na população. Entretanto, isto pode gerar um efeito indesejado, pois a maior parte da população será substituída podendo ocorrer perda de estruturas de alta aptidão. Com um valor baixo, o algoritmo pode tornar-se muito lento.

Taxa de Mutação

Uma baixa taxa de mutação previne que uma dada posição fique estagnada em um valor, além de possibilitar que se chegue em qualquer ponto do espaço de busca. Com uma taxa muito alta a busca se torna essencialmente aleatória.

Intervalo de Geração

Controla a porcentagem da população que será substituída durante a próxima geração. Com um valor muito baixo, o algoritmo provavelmente não conseguirá retornar indivíduos satisfatórios devido ao pouco número de combinações genéticas realizadas ao longo da execução.

4.4.3 Exemplo de um Algoritmo Genético Básico

Abaixo é mostrado um exemplo de Algoritmo Genético. Durante esse processo, os melhores indivíduos, assim como alguns dados estatísticos, podem ser coletados e armazenados para avaliação.

```
AG
{
 t = 0;
 inicia_população (P, t);
 avaliação (P, t);
 repita até (t = d)
 {
  t = t + 1;
  seleção_dos_pais (P, t);
  recombinação (P, t);
  mutação (P, t);
  avaliação (P, t);
  sobrevivem (P, t);
 }
}
```

Onde:

t : tempo atual.

d : tempo determinado para finalizar o algoritmo.

P : população.

Esse ciclo é repetido um determinado número de vezes e o algoritmo é finalizado quando satisfeito o critério de parada.

4.5 Aplicações

A seguir, é exibido algumas aplicações dos Algoritmos Genéticos :

Evolução Interativa de Imagens

Utilizados no reconhecimento de imagens.

Economia

Algoritmos Genéticos usados para modelar processos de inovação, o desenvolvimento de estratégias de lances, e na predição de mercados econômicos emergentes.

Indução e Otimização da Base de Regras

Algoritmos Genéticos têm sido usados em uma grande variedade de problemas de otimização, incluindo otimização de problemas numéricos e combinatoriais como o escalonamento de trabalho e o desenvolvimento de layouts para circuitos.

Programação Automática

Algoritmos Genéticos têm sido usados para desenvolver programas de computador para tarefas específicas e outras estruturas computacionais, tais como, autômatos celulares e redes de ordenação.

Aprendizagem de Máquina

Algoritmos Genéticos usados para muitas aplicações em aprendizagem, incluindo classificação e predição de tarefas, tais como : previsão do tempo ou estrutura de proteínas. Usados também para desenvolver aspectos particulares de sistemas de aprendizagem, tais como : pesos para redes neurais, regras de sistemas de aprendizagem classificadores ou sistemas de produção simbólicos, e sensores para robôs.

Ecologia

Algoritmos Genéticos usados para modelar fenômenos ecológicos como competição entre nichos biológicos, co-evolução parasita-hospedeiro, simbiose, e fluxo de recursos.

5 Aplicando Algoritmos Genéticos ao Problema Job-Shop

Este capítulo é dedicado ao tratamento do problema Job-Shop aplicando Algoritmos Genéticos.

Na seção 4.4.1 “Funcionamento” apresentamos como funcionam os algoritmos genéticos básicos. Durante a implementação do problema Job-Shop três fatores na modelagem do algoritmo genético foram considerados chaves na tentativa de sofisticarmos o algoritmo e que merecem especial atenção.

São eles :

- O indivíduo (agendamento) e a população inicial.
- A função de avaliação dos indivíduos e o processo seletivo.
- Os operadores genéticos adotados.

5.1 O Indivíduo e a População Inicial

As partes que relacionam o Algoritmo Genético com o problema Job-Shop são a codificação de uma solução, na forma de cromossomo, e a função de avaliação.

Cada solução do problema representa um agendamento, ou geneticamente um indivíduo.

A população inicial do problema é formada por um conjunto de cromossomos.

5.1.1 O Indivíduo (Scheduling)

O indivíduo representa a decodificação de um cromossomo. A seção 5.1.1.1 “O Cromossomo” trata a forma como as soluções são representadas geneticamente.

5.1.1.1 O Cromossomo

Se o problema pode ser representado por um conjunto de parâmetros (genes), estes podem ser unidos para formar uma cadeia de valores (cromossomo).

Este processo se chama codificação. Em genética este conjunto representado por um cromossomo em particular é referenciado como genótipo, contendo a informação necessária para construir um organismo (indivíduo), conhecido como fenótipo.

Esquema de Codificação para o Problema Jop-Shop

O cromossomo representa um indivíduo, mas não é um indivíduo. O cromossomo é apenas a codificação genética de uma solução viável para o problema e a solução é o próprio indivíduo.

A idéia básica é representar geneticamente o cromossomo como um vetor de permutações onde cada permutação (genoma) corresponde a uma máquina. As permutações, que correspondem à ordem em que as operações se encontram em uma máquina do problema Job-Shop, foram representadas por cadeias de estruturas. Cada estrutura (gene), ou operação, foi representada por valores inteiros únicos (genótipo) que pudessem identificar a operação no problema (esquema de codificação da estrutura).

Para tornar mais simples e legível a demonstração de cromossomos nos exemplos a serem citados, representaremos cada operação por uma estrutura de parâmetro X, Y e Z que reperesentam respectivamente o número do job, o número da operação no job e a máquina em que a operação deverá ser agendada.

5.1.1.2 O Processo de Agendamento

Segundo Ansi (1997), um agendamento possível, ou solução, é representado por permutações múltiplas definindo o indivíduo. A ordem em que as operações se encontram agendadas nas máquinas, definem a ordem em que as operações deverão ser executadas pelas máquinas. Em toda máquina existe um número fixo de operações que devem ser executadas. A restrição de capacidade requer que apenas uma dessas operações possa ser executada em um instante, e assim cada permutação da operação representará uma possível ordem de processamento das operações na máquina, depois de agendadas, e a combinação das permutações de todas as máquinas representará um agendamento possível (uma solução). Em cada máquina, durante a formação do indivíduo, deverão ser agendadas as operações seguindo às restrições de precedência estabelecidas pelo problema Job-Shop.

Uma classe de operadores crossover, os quais produzem permutações válidas a partir de permutações antigas, podem ser usados. Os operadores genéticos são aplicados a todas as permutações do cromossomo ao mesmo tempo, sempre resultando em cromossomos legítimos. (Ansi,1997).

De acordo com Ansari(1997), o agendamento (scheduling) é o arranjo de todas as operações dos jobs a serem executas pelas máquinas, na ordem estabelecida, respeitando as restrições de precedência do problema Job-Shop.

O agendamento (scheduling) é realizado durante a criação de cada indivíduo na formação da população inicial dos Algoritmos Genéticos e após o crossover, no cromossomo descendente, ordenando todas as operações atribuídas às máquinas seguindo às restrições de precedência para obter a aptidão do descendente. O agendamento também é realizado quando pretendemos obter os tempos de agendamento de cada operação codificada em um cromossomo e após a mutação sofrida por um cromossomo descendente.

O agendamento em um problema Job-Shop é para os Algoritmos Evolucionários (seguidores dos princípios da Evolução Natural), a decodificação de um cromossomo gerando um indivíduo. Representa o indivíduo.

As restrições de precedência estão intrínsecas nas operações de cada job. Uma operação de um job (trabalho) é agendada na máquina j em um tempo t se t é maior ou igual ao tempo de finalização de todas as outras operações do mesmo job que a precedem.

5.1.1.2.1 Tipos de Agendamento

Existe um número infinito de agendamentos para um simples problema Job-Shop, uma vez que, o tempo de espera pode ser inserido em um dado agendamento de infinitas maneiras. Tempo de espera, é o intervalo de tempo em que uma máquina do problema fica ociosa à espera de uma operação a ser executada. Entretanto, agendamento com tempo de espera não são interessantes, pois normalmente eles não otimizam a função objetiva (não minimizam o tempo de finalização de um agendamento).(Ansari,1997)

Certos agendamentos podem ser vistos como agendamentos dominantes entre um número infinito de agendamentos, destacamos entre eles :

- **Agendamento semiativo** (semiactive schedule): um agendamento semiativo pode ser obtido de um agendamento arbitrário movendo cada operação para a esquerda o máximo possível sem mudar a ordem das operações. Por exemplo, um agendamento semiativo de um problema 2/2 é mostrado na figura 5.1.
- **Agendamento ativo** (active schedule): cada operação no agendamento semiativo pode ser movida para a esquerda por um pulo sobre outra operação e dentro de um intervalo de tempo de espera, uma vez que o intervalo é grande o bastante para acomodar a operação e que as restrições de precedência ainda estão satisfeitas. Um agendamento obtido pela execução de pulos em um agendamento semiativo é chamado de agendamento ativo. Por exemplo, na figura 5.2, um agendamento ativo (agendamento 2) é obtido a partir do agendamento 1 fazendo com que a operação (2,1,2) pule a operação (1,2,2).
- **Agendamento sem espera** (non-delayed schedule): Se para cada instância de tempo de espera t e para cada máquina j em um agendamento ativo, não há operação agendada em j no tempo t, então o agendamento é chamado de agendamento sem espera. Ou seja, não há instante no qual um trabalho é atrasado quando a máquina para processá-lo está disponível e em espera. Uma máquina estará ociosa, quando não há operação que a requer naquele intervalo de tempo. Um agendamento ótimo não é necessariamente um agendamento sem espera. Por exemplor, um problema 2/2 é mostrado na figura 5.3. O agendamento 1 não é um agendamento sem espera, uma vez que a máquina 2 está inicialmente em espera enquanto a operação (2,1,2) é agendada. Por outro lado o agendamento 2 é um agendamento sem espera e o agendamento 1 termina antes do agendamento 2.

- Mn – máquina n
- Operação X,Y,Z : X : job; Y : número operação no job; Z : máquina.
- I : indica que a máquina está ociosa no intervalo de tempo.
- Tamanho do bloco está associado ao tempo de processamento da operação.

M1	2,1,1	I	1,2,1	I
M2	1,1,2	I	1,3,2	

Tabela 5.1 – Agendamento 1

M1	2,1,1	I	1,2,1	I
M2	1,1,2	I	1,3,2	

Tabela 5.2 – Agendamento 2

Figura 5.1 – Agendamento 2 é um agendamento semiativo obtido do agendamento 1.

M1	1,1,1	I	I	2,2,1
M2	I	1,2,2	2,1,2	I

Tabela 5.3 – Agendamento 1

M1	1,1,1	2,2,1
M2	2,1,2	1,2,2

Tabela 5.4 – Agendamento 2

Figura 5.2 – Agendamento 2 é um agendamento ativo obtido do agendamento 1.

M1	1,1,1	I	1,3,1
M2	I	1,2,2	2,1,2

Tabela 5.5 – Agendamento 1

M1	1,1,1	I	1,3,1
M2	2,1,2	1,2,2	I

Tabela 5.6 – Agendamento 2

Figura 5.3 – Agendamento com (1) e sem (2) espera.

5.1.1.2.2 Agendando com Eficiência

A ordem das operações codificadas no cromossomo, indica a ordem em que as operações deverão ser executadas por cada máquina.

Evitaremos agendamentos que propiciem maiores intervalos de ociosidade nas máquinas. Não será permitido, durante o agendamento, uma máquina se encontrar ociosa enquanto existir operação que a requer no intervalo de ociosidade. Não queremos atrasar a execução dos jobs.

No projeto, só consideramos as gerações de agendamentos ativos e sem espera. O objetivo foi encontrar o agendamento com o término em menor tempo.

Os agendamentos das operações realizadas em cada máquina são de natureza paralela. As máquinas trabalham em paralelo na execução das operações. Foi simulado, neste projeto, o paralelismo, durante o agendamento, que segue o seguinte modelo:

Cada máquina, durante o agendamento, possui o seu tempo de disponibilidade, ou seja, tempo em que a máquina estará disponível para ser agendada uma operação. Inicialmente o tempo de disponibilidade delas é o tempo em que o agendamento começará a ser realizado. (O algoritmo de agendamento assume tempo inicial igual a zero para cada máquina).

O tempo corrente é o tempo real avançado sucessivamente durante o agendamento. No agendamento, corresponde ao menor tempo de disponibilidade entre as máquinas não ociosas do problema.

Para não termos agendamentos com espera, ao iniciar o agendamento, será agendada em cada máquina a operação inicial de um job a ser executada por tal máquina. Isso acontece se existir uma operação inicial de job que requer a máquina. No caso em que houver mais de uma operação inicial de jobs a serem executadas por uma mesma máquina, será agendada a primeira operação na ordem em que se encontra no cromossomo.

Ao ser agendada uma operação em uma máquina, avançamos o tempo de disponibilidade da máquina, ou seja, somamos o tempo corrente ao tempo de processamento da operação agendada, a fim de estabelecermos o próximo tempo em que a máquina estará disponível.

Se não foi agendada uma operação à máquina, significa que não há neste instante operação que a requer, então teremos um intervalo de ociosidade. Nesse caso, o tempo de disponibilidade da máquina será o menor tempo de disponibilidade encontrado entre as máquinas depois de um processo de verificação de ociosidade entre todas as máquinas que possuem o mesmo tempo corrente. Veja a seção 5.1.1.2.3 a seguir "Tratamento da Ociosidade" que detalha o processo de verificação de ociosidade e que tem como objetivo minimizar o tempo de espera em uma máquina.

Após o agendamento de uma operação ou tratamento da ociosidade de uma máquina, caso não haja operação a ser agendada no momento, é localizada a máquina, no problema, que possuir o menor tempo de disponibilidade e que não estiver ociosa ou agendada para o agendamento de uma nova operação.

Ao definir a próxima máquina para agendamento, se tornam candidatas todas as operações que sucedem as últimas operações agendadas nas máquinas disponíveis em tempo real (tempo corrente igual ao tempo de disponibilidade da máquina). Dentre as operações

candidatas, é agendada aquela operação, na máquina, que for a primeira operação encontrada em ordem no cromossomo.

O agendamento é realizado enquanto houver operação a ser agendada.

5.1.1.2.3 Tratamento da Ociosidade

É importante tratarmos com eficiência a ociosidade a fim de termos o menor tempo de espera em uma máquina.

O tratamento da ociosidade é realizado da seguinte forma:

É verificado entre as máquinas do problema, se há alguma outra máquina com o mesmo tempo corrente da máquina ociosa, ou seja, outra máquina que esteja disponível em tempo real.

Se não houver, é avançado o tempo corrente da máquina para o menor tempo de disponibilidade correspondente a uma máquina não ociosa do problema. É retirado em seguida, o status de ociosidade da máquina.

Se houver, localiza-se a próxima operação a ser agendada em tal máquina, nesse instante. Caso não haja operação a ser agendada nesse momento, é repetido o processo atribuindo status de ociosidade à máquina e procurando por outra máquina de mesmo tempo corrente. Caso contrário (foi encontrada operação a ser agendada à máquina nesse momento), é agendada a operação e avançado o tempo corrente da máquina somado ao tempo de processamento da operação e, para as demais máquinas de mesmo tempo corrente, continua o processo de verificação de ociosidade. Após a verificação completa entre todas as máquinas de mesmo tempo corrente, aquelas máquinas ociosas terão seus tempos avançados para o menor tempo de disponibilidade encontrado correspondente a uma máquina não ociosa do problema. É retirado em seguida, o status de ociosidade das máquinas.

Quando procuramos avançar o tempo corrente de uma máquina ociosa para o menor tempo de disponibilidade de uma máquina não ociosa no problema, nossa intenção é de tornar o intervalo de ociosidade o menor possível para a máquina. Não realizamos esse processo diretamente ao encontrar uma máquina ociosa, porque podemos encontrar máquinas, nesse mesmo tempo (máquinas trabalham em paralelo), com operações sendo agendadas podendo contribuir para a definição do menor tempo de disponibilidade a ser atribuída à máquina ociosa.

Em relação aos tempos de disponibilidade assumidos pelas máquinas, não significa, com certeza, que uma operação será agendada nesse tempo em que a máquina se encontra disponível. Quando o tempo corrente for igual ao tempo de disponibilidade (avanço do tempo real), a máquina se encontra disponível, mas não podemos garantir que haja uma operação a ser agendada nesse instante, ou seja, que requer a máquina nesse momento. Nesse caso, a máquina se torna ociosa.

A seção a seguir, 5.1.2 “População Inicial”, dará uma maior ênfase a descrição dos cromossomos implementados para o problema Job-Shop, onde são abordados critérios utilizados no problema para criação de indivíduos, exemplos de cromossomos, e exemplos de indivíduos gerados durante o agendamento das operações às máquinas.

5.1.2 População Inicial

A população inicial para algoritmos genéticos significa o seu espaço de pesquisa inicial. É formada por indivíduos que passarão por processos de seleção e reprodução seguindo os mecanismos de evolução natural.

Uma população é formada por cromossomos onde cada cromossomo representa um indivíduo. Assim teremos que criar cromossomos para termos uma população inicial.

Cada cromossomo está associado a um único indivíduo e é a codificação genética de uma solução viável para o problema Job-Shop.

Para as populações iniciais estudadas nesse projeto, a ordem que diz qual a próxima operação do problema JSP a ser escolhida para ser agendada em uma máquina disponível, durante a formação do cromossomo, está associada à escolhas : **Randômica** – operação é escolhida aleatoriamente; **Menor Tempo Acumulativo** – é escolhida a operação candidata que possuir o menor tempo de processamento no conjunto de operações candidatas da máquina; e escolha **GRASP** - que une escolha randômica e escolha por menor tempo acumulativo. Um valor $\propto$ (percentual) escolhe as operações candidatas que possuírem os Menores Tempos Acumulativos para a máquina disponível.

5.1.2.1 Gerando a População Inicial

Segue-se os passos para geração de uma solução (agendamento) para o problema Job-Shop :

Passo 1 :

Cada máquina do problema JSP possui inicialmente um conjunto formado pelas primeiras operações de cada job a serem executadas. Esses conjuntos são chamados Conjunto de Operações Agendáveis de uma máquina.

Passo 2 :

Serão escolhidas as máquinas disponíveis no tempo corrente *t*.

Se houver máquina disponível :

Para todas as máquinas do problema, caso a última operação agendada possui tempo de finalização menor ou igual a *t*, busca-se então, caso exista, a operação do mesmo job sucessora a última operação agendada à máquina e a inseri no conjunto agendável correspondente à máquina em que a operação deverá ser executada. A operação torna-se agendável a partir do instante *t*.

Se não houver máquina disponível, vá para o Passo 6.

Passo 3 :

Os conjuntos de operações candidatas serão os conjuntos de operações agendáveis de cada máquina escolhida.

Se um conjunto de candidatas estiver vazio, significa que não há operação a ser agendada na máquina escolhida, logo esta se tornará ociosa.

Passo 4 :

Em seguida, para cada conjunto de candidatas, é escolhida uma operação do conjunto de operações candidatas não vazio a ser agendada na sua máquina, seguindo o critério de escolha Randômico ou Menor Tempo Acumulativo. Para o critério de escolha GRASP, cria-se uma lista de candidatos restrita (LCR), para cada conjunto de candidatas, que irá conter as operações candidatas com Menores Tempos Acumulativos (menor tempo de processamento das operações) no intervalo definido por uma taxa gulosa e a partir daí uma escolha aleatória definirá a operação a ser agendada.

Passo 5 :

Uma vez sorteada as operações dos conjuntos de operações candidatas (ou dos conjuntos LCR para o critério GRASP), essas operações são retiradas dos conjuntos agendáveis das máquinas em que elas serão atribuídas. Atribui-se as operações às máquinas. As máquinas tornam-se não disponíveis e cada máquina tornará disponível quando o tempo *t* atingir o tempo de finalização da operação agendada.

Passo 6 :

O processo percorrido para geração de solução é finalizado quando todas as operações do problema estiverem sido agendadas às suas respectivas máquinas.

Se não foi finalizado o processo, avançar o tempo *t* e voltar ao Passo 2 repetindo os passos.

Ao finalizar os passos percorridos teremos uma solução codificada, ou seja, um cromossomo gerado.

Executando n vezes o algoritmo que implementa esse mecanismo para gerar um cromossomo, teremos uma população inicial de n indivíduos.

O problema 3/3 JSP apresentado na seção 3.1 “Definições e Premissas” será utilizado para gerarmos, como exemplo, cromossomos para a População Inicial Aleatória, População Inicial Menor Tempo Acumulativo e População Inicial GRASP.

5.1.2.1.1 População Inicial Aleatória

Consulte antes a seção 5.1.2.1 “Gerando a População Inicial” para se informar como criar uma população inicial para os Algoritmos Genéticos.

Na População Inicial Aleatória uma operação do conjunto de operações candidatas é escolhida seguindo o critério randômico – escolha aleatória. Cada operação candidata possui a mesma probabilidade de ser escolhida no conjunto.

5.1.2.1.1.1 Gerando Cromossomo Aleatório

Como foi dito na seção 5.1.2.1 "Gerando a População Inicial" o conjunto de operações candidatas representa o conjunto de operações agendáveis da máquina disponível que foi escolhida para agendamento no tempo corrente *t*.

Após a formação dos conjuntos de operações candidatas, sorteamos aleatoriamente nos conjuntos de candidatas as operações a serem inseridas nas suas respectivas máquinas, no momento disponíveis para agendamento. Assim, é retirada cada operação do conjunto da máquina a qual pertence e atribuída à máquina.

Esse processo acontece sucessivamente ao longo do tempo *t* até que todos as operações do problema já estejam agendadas formando assim o cromossomo aleatório.

Veja o pseudo-código para criar um cromossomo aleatório :

```
criarCromossomoRandomico
{
  t = inicialisar;
  while ( não todos os jobs finalizados )
  {
    avançar t;
    /* check o status de cada máquina */
    for ( cada máquina )
      if ( máquina  está ociosa ou
           operação corrente dessa máquina foi finalizada )
      {
        máquina está disponível;
       /* atualizar o conjunto de operações agendáveis dessa máquina */
       if (há oper sucessora de mesmo job à última oper agendada nessa
máquina e oper sucessora não está no conjunto de opers agendáveis )
       {
           proxOp = operação sucessora;
           maq = máquina em que proxOp deverá ser atribuída;
           inserir proxOp no conjunto de operações da maquina maq;
       }
      }
    /* agendar nova operação para cada máquina */
    for (cada máquina )
     if ( essa máquina está disponível ) {
      conjunto de candidatas é conjunto de operações agendáveis da máquina;
      if ( não vazio o conjunto de operações candidatas )
      {
          escolher randomicamente operação do conjunto de candidatas;
          retirar do conjunto a operação;
          atribuir a nova operação escolhida à máquina;
          máquina não disponível;
      }
     } // if
  }  // while
} // criarCromossomoRandomico
```

Executando esse pseudo-código n vezes criamos uma população inicial de n indivíduos.

5.1.2.1.1.2 Exemplo de Cromossomo Aleatório

Veja um exemplo de cromossomo aleatório gerado a partir do problema 3/3 JSP, figura 3.1 na seção 3.1 "Definições e Premissas". Cada iteração corresponde a uma operação agendada.

- A <= B : incluir em A o conteúdo de B.
- A = B : o conteúdo de A é igual ao conteúdo de B.

- t : tempo real (tempo corrente). Representação de medida u : unidades de tempo.
- o : operação
- Mn : máquina n do problema 3/3 JSP.
- Conj. Ops Maq(n) : conjunto de operações agendáveis da máquina n.
- Conj. A : um conjunto qualquer.
- Random (Conj. A) : operação sorteada aleatoriamente do conjunto A.
- Retirar (a,Conj. A) : retirar operação a do conjunto A.
- Crom(m) : máquina m do cromossomo.
- TProc(o) : tempo de processamento da operação o.
- TAg(o) : tempo de agendamento de uma operação o. Acontece sempre no tempo corrente t.
- ProxTDisp(Mn) : tempo em que a máquina n se encontrará disponível para agendar uma operação.
    - ProxTDisp(Mn) = t + TProc(o) : após agendamento de uma operação na máquina n no tempo t.
    - ProxTDisp(Mn) = ProxTDisp(Mn') : quando não há agendamento de uma operação em Mn no tempo t. Verifica-se antes a possibilidade de agendamento em todas as outras máquinas n' no tempo t (se estiverem disponíveis nesse tempo). Se houver possibilidade de agendamento, agende-as. Em seguida, a maquina n' que possuir o menor ProxTDisp será atribuída esse valor ao ProxTDisp da máquina n. Mn' é diferente de Mn.
- I : intervalo de ociosidade.

Condições Iniciais:

t = 0u.

Conj. Ops Maq(1) = {}; Conj. Ops Maq(2) = {}; Conj. Ops Maq(3) = {}

Iteração 1:

1- Conj. Ops Maq(1) = {}; Conj. Ops Maq(2) = {1,1,2; 3,1,2}; Conj. Ops Maq(3) = {2,1,3}

2- Verificar Conj. Ops Maq(1).

- Nenhuma operação encontrada. Logo M1 é ociosa.

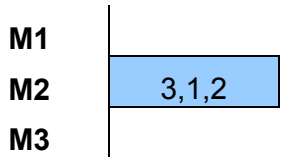
3- Verificar Conj. Ops Maq(2).

- Crom(2) <= (Random (Conj. Ops Maq(2)) = {3,1,2})

4- Retirar ({3,1,2}, Conj. Ops Maq(2)) : Conj. Ops Maq(2) = {1,1,2}

5- TAg({3,1,2}) = 0u

6- ProxTDisp(M2) = 0 + (Tproc({3,1,2}) = 7) = 7u.



Iteração 2:

1- Conj. Ops Maq(1) = {}; Conj. Ops Maq(2) = {1,1,2}; Conj. Ops Maq(3) = {2,1,3}

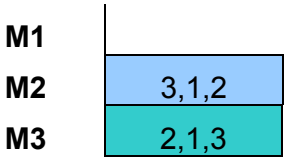
2- Verificar Conj. Ops Maq(3).

- Crom(3) <= (Random (Conj. Ops Maq(3)) = {2,1,3})

3- Retirar ({2,1,3}, Conj. Ops Maq(3)) : Conj. Ops Maq(3) = {}

4- TAg({2,1,3}) = 0u

5- ProxTDisp(M3) = 0 + (Tproc({2,1,3}) = 7) = 7u.



Iteração 3:

1- Conj. Ops Maq(1) = {}; Conj. Ops Maq(2) = {1,1,2}; Conj. Ops Maq(3) = {}

2- Verificar ociosidade em M1. ProxTDisp(M1) = 7u. (menor ProxTDisp)

3- t = 7u. (menor ProxTDisp).

4- Atualizar todos Conj. Ops Maq.

- Conj. Ops Maq(1) <= {3,2,1} : Conj. Ops Maq(1) = {3,2,1}
- Conj. Ops Maq(2) <= {2,2,2} : Conj. Ops Maq(2) = {1,1,2; 2,2,2}

5- Verificar Conj Ops Maq(1).

- Crom(1) <= (Random (Conj. Ops Maq(1)) = {3,2,1})

6- Retirar ({3,2,1}, Conj. Ops Maq(1)) : Conj. Ops Maq(1) = {}

7- TAg({3,2,1}) = 7u

8- ProxTDisp(M1) = 7 + (Tproc({3,2,1}) = 5) = 12u.

Iteração 4:

1- Conj. Ops Maq(1) = {}; Conj. Ops Maq(2) = {1,1,2; 2,2,2}; Conj. Ops Maq(3) = {}

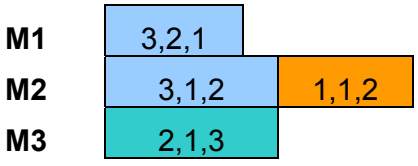
2- Verificar Conj Ops Maq(2).

- Crom(2) <= (Random (Conj. Ops Maq(2)) = {1,1,2})

3- Retirar ({1,1,2}, Conj. Ops Maq(2)) : Conj. Ops Maq(2) = {2,2,2}

4- TAg({1,1,2}) = 7u

5- ProxTDisp(M2) = 7 + (Tproc({1,1,2}) = 5) = 12u.



Iteração 5:

1- Conj. Ops Maq(1) = {}; Conj. Ops Maq(2) = {2,2,2}; Conj. Ops Maq(3) = {}

2- t = 12u. (menor ProxTDisp)

3- Atualizar todos Conj. Ops Maq.

- Conj. Ops Maq(2) <= {1,2,2} : Conj. Ops Maq(2) = {2,2,2; 1,2,2}

4- Verificar Conj. Ops Maq(1).

- Nenhuma operação encontrada. Logo M1 é ociosa.

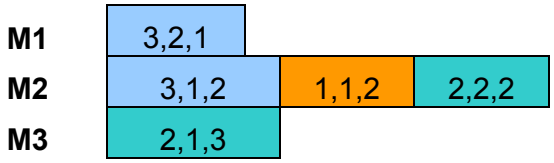
5- Verificar Conj. Ops Maq(2).

- Crom(2) <= (Random (Conj. Ops Maq(2)) = {2,2,2})

6- Retirar ({2,2,2}, Conj. Ops Maq(2)) : Conj. Ops Maq(2) = {}

7- TAg({2,2,2}) = 12u

8- ProxTDisp(M2) = 12 + (Tproc({2,2,2}) = 5) = 17u.



Iteração 6:

1- Conj. Ops Maq(1) = {}; Conj. Ops Maq(2) = {1,2,2}; Conj. Ops Maq(3) = {}

2- Verificar ociosidade em M1. ProxTDisp(M1) = 17u. (menor ProxTDisp)

3- t = 17u. (menor ProxTDisp)

4- Atualizar todos Conj. Ops Maq.

- Conj. Ops Maq(1) <= {2,3,1} : Conj. Ops Maq(1) = {2,3,1}

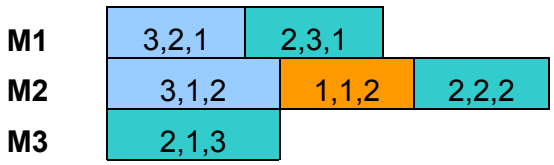
5- Verificar Conj. Ops Maq(1).

- Crom(1) <= (Random (Conj. Ops Maq(1)) = {2,3,1})

6- Retirar ({2,3,1}, Conj. Ops Maq(1)) : Conj. Ops Maq(1) = {}

7- TAg({2,3,1}) = 17u

8- ProxTDisp(M1) = 17 + (Tproc({2,3,1}) = 5) = 22u.



Iteração 7:

1- Conj. Ops Maq(1) ={}; Conj. Ops Maq(2) = {1,2,2}; Conj. Ops Maq(3) = {}

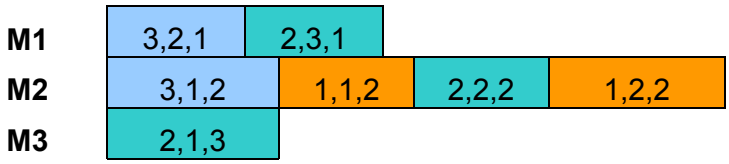
2- Verificar Conj. Ops Maq(2).

- Crom(2) <= (Random (Conj. Ops Maq(2)) = {1,2,2})

3- Retirar ({1,2,2}, Conj. Ops Maq(2)) : Conj. Ops Maq(2) = {}

4- TAg({1,2,2}) = 17u

5- ProxTDisp(M2) = 17 + (Tproc({1,2,2}) = 7) = 24u.



Iteração 8:

1- Conj. Ops Maq(1) ={}; Conj. Ops Maq(2) = {}; Conj. Ops Maq(3) = {}

2- t = 22u. (menor ProxTDisp)

3- Atualizar todos Conj. Ops Maq. Nenhum conjunto atualizado.

4- Verificar Conj. Ops Maq(1).

- Nenhuma operação encontrada. Logo M1 é ociosa.

5- t = 24u. (menor ProxTDisp)

6- Atualizar todos Conj. Ops Maq.

- Conj. Ops Maq(1) <= {1,3,1} : Conj. Ops Maq(1) = {1,3,1}

7- Verificar Conj. Ops Maq(1).

- Crom(1) <= (Random (Conj. Ops Maq(1)) = {1,3,1})

8- Retirar ({1,3,1}, Conj. Ops Maq(1)) : Conj. Ops Maq(1) = {}

9- TAg({1,3,1}) = 24u

10- ProxTDisp(M1) = 24 + (Tproc({1,3,1}) = 7) = 31u.

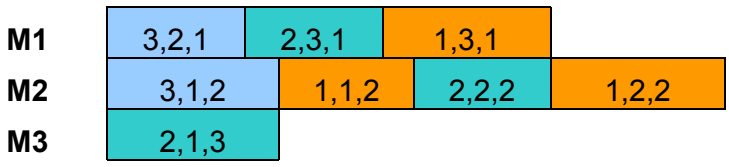


Tabela 5.7 – Cromossomo Aleatório

O cromossomo gerado pelo pseudo-código **criarCromossomoRandomico** é totalmente aleatório. Uma operação é escolhida aleatoriamente do conjunto de operações da máquina disponível (conjunto de operações candidatas).

O cromossomo gerado representa a ordem em que as operações deverão ser executadas pelas máquinas. Voltamos a lembrar que as operações são codificadas na forma de valores inteiros únicos (genótipos) que as representa.

O indivíduo é representado abaixo com tempo de finalização igual a 31 unidades de tempo.

M1	I	3,2,1	I	2,3,1	I	1,3,1
M2	3,1,2	1,1,2	2,2,2	1,2,2		
M3	2,1,3					

Tabela 5.8 – Indivíduo Aleatório

5.1.2.1.2 População Inicial Menor Tempo Acumulativo

Consulte antes a seção 5.1.2.1 "Gerando a População Inicial" para se informar como criar uma população inicial para os Algoritmos Genéticos.

Para a população inicial constituída de cromossomos gerados à partir do critério de escolha Menor Tempo Acumulativo, a ordem que diz qual a próxima operação a ser escolhida para inserir na máquina diponível, durante a formação do cromossomo, será a operação candidata que possuir o menor tempo de processamento dentre as operações candidatas que se encontram no conjunto de operações agendáveis da máquina.

5.1.2.1.2.1 Gerando Cromossomo Menor Tempo Acumulativo

Como foi dito na seção 5.1.2.1 "Gerando a População Inicial" o conjunto de operações candidatas representa o conjunto de operações agendáveis da máquina disponível que foi escolhida para agendamento no tempo corrente *t*.

Após a formação dos conjuntos de operações candidatas, serão escolhidas as operações, de cada conjunto, que possuirem os menores tempos de processamento.

Observe a descrição formal matemática que define a operação candidata com Menor Tempo Acumulativo.

- Conj. A : conjunto de operações candidatas.
- c : operação candidata. $c \in Conj.\ A$.
- o : operação.
- Crom(m) : maquina *m* do cromossomo.
- TProc(o) : tempo de processamento da operação *o*.
- Maq(o) : máquina em que a operação o deverá ser executada.
- MinTProc(Conj A) : operação de menor tempo *TProc* dentre as operações contidas no conjunto A.
- MTA(Conj A) : operação do conjunto A com "Menor Tempo Acumulativo".

  MTA (Conj A) = MinTProc(Conj A).

Depois que as operações candidatas foram escolhidas, estas são retiradas dos conjuntos de operações agendáveis da máquina a qual pertence e em seguida cada operação é atribuída à sua repectiva maquina, no momento disponível para agendamento.

Esse processo acontece sucessivamente ao longo do tempo *t* até que todos as operações do problema já estejam agendadas formando assim o cromossomo Menor Tempo Acumulativo.

Veja o pseudo-código :

```
criarCromossomoMTA
{
  t = inicialisar;
  while ( não todos os jobs finalizados )
  {
    avançar t;
    /* check o status de cada máquina */
    for ( cada máquina )
     if ( máquina  está ociosa ou
          operação corrente dessa máquina foi finalizada )
     {
       máquina está disponível;
       /* atualizar o conjunto de operações agendáveis dessa máquina */
       if (há oper sucessora de mesmo job à última oper agendada nessa
máquina e oper sucessora não está no conjunto de opers agendáveis )
       {
           proxOp = operação sucessora;
           maq = máquina em que proxOp deverá ser atribuída;
           inserir proxOp no conjunto de operações da maquina maq;
       }
      }
    /* agendar nova operação para cada máquina */
    for (cada máquina )
     if ( essa máquina está disponível ) {
      conjunto de candidatas é conjunto de operações agendáveis da máquina;
      if ( não vazio o conjunto de operações candidatas )
      {
          escolher operação do conjunto com menor tempo de processamento;
          retirar do conjunto a operação;
          atribuir a nova operação escolhida à máquina;
          máquina não disponível;
      }
     } // if
  }  // while
} // criarCromossomoMTA
```

Executando esse pseudo-código n vezes podemos criar uma população inicial de n indivíduos. Um detalhe importante é que a população inicial será totalmente homogênea constituída de indivíduos com iguais valores de aptidão, segundo o critério guloso Menor Tempo Acumulativo.

5.1.2.1.2.2 Exemplo de Cromossomo Menor Tempo Acumulativo

Veja um exemplo de cromossomo Menor Tempo Acumulativo gerado a partir do problema 3/3 JSP, figura 3.1 na seção 3.1 “Definições e Premissas”. Cada iteração corresponde a uma operação agendada.

- A <= B : incluir em A o conteúdo de B.

- A = B : o conteúdo de A é igual ao conteúdo de B.

- t : tempo real (tempo corrente). Representação de medida u : unidades de tempo.

- o : operação

- Mn : máquina n do problema 3/3 JSP.

- Conj. Ops Maq(n) : conjunto de operações agendáveis da máquina n.

- Conj. A : um conjunto qualquer.

- MTA (Conj. A) : operação com Menor Tempo Acumulativo no conjunto A.

- Retirar (a,Conj. A) : retirar operação a do conjunto A.

- Crom(m) : máquina m do cromossomo.

- TProc(o) : tempo de processamento da operação o.

- TAg(o) : tempo de agendamento de uma operação o. Acontece sempre no tempo corrente t.

- ProxTDisp(Mn) : tempo em que a máquina n se encontrará disponível para agendar uma operação.

    - ProxTDisp(Mn) = t + TProc(o) : após agendamento de uma operação na máquina n no tempo t.
    - ProxTDisp(Mn) = ProxTDisp(Mn') : quando não há agendamento de uma operação em Mn no tempo t. Verifica-se antes a possibilidade de agendamento em todas as outras máquinas n' no tempo t (se estiverem disponíveis nesse tempo). Se houver possibilidade de agendamento, agende-as. Em seguida, a maquina n' que possuir o menor ProxTDisp será atribuída esse valor ao ProxTDisp da máquina n. Mn' é diferente de Mn.

- I : intervalo de ociosidade.

Condições Iniciais:

t = 0u.

Conj. Ops Maq(1) = {}; Conj. Ops Maq(2) = {}; Conj. Ops Maq(3) = {}

Iteração 1:

1- Conj. Ops Maq(1) = {}; Conj. Ops Maq(2) = {1,1,2; 3,1,2}; Conj. Ops Maq(3) = {2,1,3}

2- Verificar Conj. Ops Maq(1).

- Nenhuma operação encontrada. Logo M1 é ociosa.

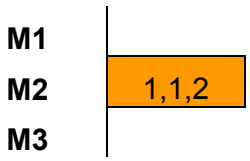
3- Verificar Conj. Ops Maq(2).

- Crom(2) <= (MTA(Conj. Ops Maq(2)) = {1,1,2})

4- Retirar ({1,1,2}, Conj. Ops Maq(2)) : Conj. Ops Maq(2) = {3,1,2}

5- TAg({1,1,2}) = 0u

6- ProxTDisp(M2) = 0 + (Tproc({1,1,2}) = 5) = 5u.



Iteração 2:

1- Conj. Ops Maq(1) = {}; Conj. Ops Maq(2) = {3,1,2}; Conj. Ops Maq(3) = {2,1,3}

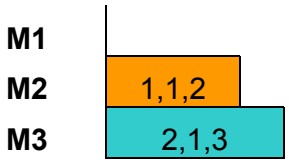
2- Verificar Conj. Ops Maq(3).

- Crom(3) <= (MTA(Conj. Ops Maq(3)) = {2,1,3})

3- Retirar ({2,1,3}, Conj. Ops Maq(3)) : Conj. Ops Maq(3) = {}

4- TAg({2,1,3}) = 0u

5- ProxTDisp(M3) = 0 + (Tproc({2,1,3}) = 7) = 7u.



Iteração 3:

1- Conj. Ops Maq(1) = {}; Conj. Ops Maq(2) = {3,1,2}; Conj. Ops Maq(3) = {}

2- Verificar ociosidade em M1. ProxTDisp(M1) = 5u. (menor ProxTDisp)

3- t = 5u. (menor ProxTDisp).

4- Atualizar todos Conj. Ops Maq.

- Conj. Ops Maq(2) <= {1,2,2} : Conj. Ops Maq(2) = {3,1,2; 1,2,2}

5- Verificar Conj Ops Maq(1).

- Nenhuma operação encontrada. Logo M1 é ociosa.

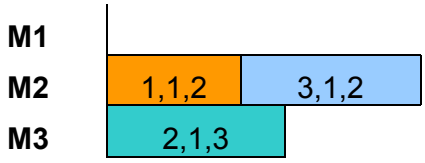
6- Verificar Conj Ops Maq(2).

- Crom(2) <= (MTA(Conj. Ops Maq(2)) = {3,1,2})

7- Retirar ({3,1,2}, Conj. Ops Maq(2)) : Conj. Ops Maq(2) = {1,2,2}

8- TAg({3,1,2}) = 5u

9- ProxTDisp(M2) = 5 + (Tproc({3,1,2}) = 7) = 12u.



Iteração 4:

1- Conj. Ops Maq(1) = {}; Conj. Ops Maq(2) = {1,2,2}; Conj. Ops Maq(3) = {}

2- Verificar ociosidade em M1. ProxTDisp(M1) = 7u. (menor ProxTDisp)

3- t = 7u. (menor ProxTDisp)

4- Atualizar todos Conj. Ops Maq.

- Conj. Ops Maq(2) <= {2,2,2} : Conj. Ops Maq(2) = {1,2,2; 2,2,2}

5- Verificar operação do Conj. Ops Maq(1). Nenhuma operação encontrada. Logo M1 é ociosa.

6- Verificar ociosidade em M1. ProxTDisp(M1) = 12u. (menor ProxTDisp)

7- t = 12u. (menor ProxTDisp).

8- Atualizar todos Conj. Ops Maq.

- Conj. Ops Maq(1) <= {3,2,1} : Conj. Ops Maq(1) = {3,2,1}

9- Verificar operação do Conj. Ops Maq(1).

- Crom(1) <= (MTA(Conj. Ops Maq(1)) = {3,2,1})

10- Retirar ({3,2,1}, Conj. Ops Maq(1)) : Conj. Ops Maq(1) = {}

11- TAg({3,2,1}) = 12u

12- ProxTDisp(M1) = 12 + (Tproc({3,2,1}) = 5) = 17u.

Iteração 5:

1- Conj. Ops Maq(1) = {}; Conj. Ops Maq(2) = {1,2,2; 2,2,2}; Conj. Ops Maq(3) = {}

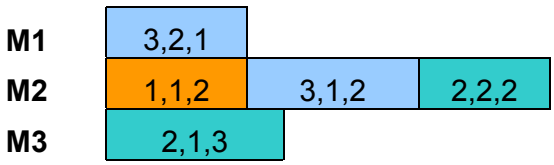
2- Verificar operação do Conj. Ops Maq(2).

- Crom(2) <= (MTA(Conj. Ops Maq(2)) = {2,2,2})

3- Retirar ({2,2,2}, Conj. Ops Maq(2)) : Conj. Ops Maq(2) = {1,2,2}

4- TAg({2,2,2}) = 12u

5- ProxTDisp(M2) = 12 + (Tproc({2,2,2}) = 5) = 17u.



Iteração 6:

1- Conj. Ops Maq(1) = {}; Conj. Ops Maq(2) = {1,2,2}; Conj. Ops Maq(3) = {}

2- t = 17u. (menor ProxTDisp).

3- Atualizar todos Conj. Ops Maq.

- Conj. Ops Maq(1) <= {2,3,1} : Conj. Ops Maq(1) = {2,3,1}

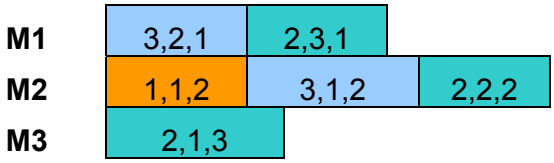
4- Verificar Conj Ops Maq(1).

- Crom(1) <= (MTA(Conj. Ops Maq(1)) = {2,3,1})

5- Retirar ({2,3,1}, Conj. Ops Maq(1)) : Conj. Ops Maq(1) = {}

6- TAg({2,3,1}) = 17u

7- ProxTDisp(M1) = 17 + (Tproc({2,3,1}) = 5) = 22u.



Iteração7:

1- Conj. Ops Maq(1) = {}; Conj. Ops Maq(2) = {1,2,2}; Conj. Ops Maq(3) = {}

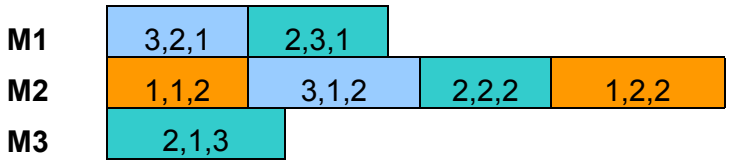
2- Verificar Conj Ops Maq(2).

- Crom(2) <= (MTA(Conj. Ops Maq(2)) = {1,2,2})

3- Retirar ({1,2,2}, Conj. Ops Maq(2)) : Conj. Ops Maq(2) = {}

4- TAg({1,2,2}) = 17u

5- ProxTDisp(M2) = 17 + (Tproc({1,2,2}) = 7) = 24u.



Iteração 8:

1- Conj. Ops Maq(1) = {}; Conj. Ops Maq(2) = {}; Conj. Ops Maq(3) = {}

2- t = 22u. (menor ProxTDisp).

3- Atualizar todos Conj. Ops Maq. Nenhum conjunto atualizado.

4- Verificar Conj. Ops Maq(1).

- Nenhuma operação encontrada. Logo M1 é ociosa.

5- t = 24u. (menor ProxTDisp)

6- Atualizar todos Conj. Ops Maq.

- Conj. Ops Maq(1) <= {1,3,1} : Conj. Ops Maq(1) = {1,3,1}

7- Verificar Conj. Ops Maq(1).

- Crom(1) <= (MTA(Conj. Ops Maq(1)) = {1,3,1})

8- Retirar ({1,3,1}, Conj. Ops Maq(1)) : Conj. Ops Maq(1) = {}

9- TAg({1,3,1}) = 24u

10- ProxTDisp(M1) = 24 + (Tproc({1,3,1}) = 7) = 31u.

M1	3,2,1	2,3,1	1,3,1	
M2	1,1,2	3,1,2	2,2,2	1,2,2
M3	2,1,3			

Tabela 5.9 – Cromossomo Menor Tempo Acumulativo

O pseudo-código **criarCromossomoMTA** gera um único cromossomo.

A operação de menor tempo de processamento é escolhida do conjunto de operações da máquina disponível (conjunto de operações candidatas).

O cromossomo gerado representa a ordem em que as operações deverão ser executadas pelas máquinas. Voltamos a lembrar que as operações são codificadas na forma de valores inteiros únicos (genótipos) que as representa.

O indivíduo é representado abaixo com tempo de finalização igual a 31 unidades de tempo.

M1	I	I	3,2,1	2,3,1	I	1,3,1
M2	1,1,2	3,1,2	2,2,2	1,2,2		
M3	2,1,3					

Tabela 5.10 – Indivíduo Menor Tempo Acumulativo

5.1.2.1.3 População Inicial GRASP

Uma abordagem híbrida é obtida com o Algoritmo Genético que cria a sua população inicial adotando conceitos do algoritmo de otimização GRASP, com a finalidade de flexibilizar a obtenção de melhores soluções para o problema JSP.

A criação da População Inicial GRASP tem como objetivo sofisticar o algoritmo genético puro. Testes serão realizados com as soluções obtidas do algoritmo genético híbrido comparando às soluções obtidas pelo algoritmo genético que cria uma População Inicial Aleatória (algoritmo genético puro) e às soluções obtidas pelo algoritmo genético que cria uma População Inicial Menor Tempo Acumulativo.

Antes de falarmos a respeito da População Inicial GRASP, introduziremos o conceito de GRASP, que é uma outra técnica que adota heurística, além de Algoritmos Genéticos, a procura de boas soluções para um dado problema.

5.1.2.1.3.1 GRASP

GRASP - *Greedy Randomized Adaptive Search Procedure* – traduzido para Procedimento de Busca Adaptativa Gulosa e Randomizada é um método interativo, proposto por Feo & Resende(1995), que consiste de duas fases: uma fase de construção, na qual uma solução é gerada, elemento a elemento, e de uma fase de busca local, na qual um ótimo local na vizinhança da solução construída é pesquisado. A melhor solução encontrada ao longo de todas as interações GRASP realizadas é retornada como resultado do algoritmo de otimização GRASP. (Souza,2001).

Fase de Construção

Na fase de construção, uma solução é interativamente construída, elemento por elemento. A cada iteração dessa fase, os próximos elementos candidatos a serem incluídos na solução são colocados em uma lista C de candidatos, seguindo um critério de ordenação pré-determinado. O processo de seleção é baseado em uma função adaptativa gulosa $g : C \rightarrow R$, que estimula o benefício da seleção de cada um dos elementos. A heurística é adaptativa porque os benefícios associados com a escolha de cada elemento são atualizados em cada iteração da fase de construção para refletir as mudanças oriundas da seleção do elemento anterior. A componente probabilística ($\alpha \in [0,1]$) reside no fato de que cada elemento é selecionado de forma aleatória a partir de um subconjunto restrito formado pelos melhores elementos que compõem a lista de candidatos. Este subconjunto recebe o nome de lista de candidatos restrita (LCR). Esta técnica de escolha permite que diferentes soluções sejam geradas em cada iteração GRASP.

O parâmetro α controla o nível de gulosidade e aleatoriedade na geração das soluções durante as interações GRASP. Um valor $\alpha = 0$ faz gerar soluções puramente gulosas, enquanto $\alpha = 1$ faz produzir soluções totalmente aleatórias.

Assim como em muitas técnicas determinísticas, as soluções geradas pela fase de construção do GRASP provavelmente não são localmente ótimas com respeito à definição de vizinhança adotada. Daí a importância da fase de busca local, a qual objetiva melhorar a solução construída.

(Souza,2001)

Fase de Busca Local

As técnicas de busca local em problemas de otimização constituem uma família de técnicas baseadas na noção de vizinhança.

Observe a modelagem de um problema qualquer de otimização : Dado um conjunto S de soluções s e uma função objetivo $f : S \rightarrow R$, que associa cada solução $s \in S$ a um valor real $f(s)$, encontre a solução $s^* \in S$, dita ótima, para a qual $f(s)$ é mínima.

Ou seja, S é o espaço de pesquisa de um problema de otimização e f a função objetivo a minimizar. A função N, a qual depende da estrutura do problema tratado, associa a cada solução viável $s \in S$, sua vizinhança $N(s) \subseteq S$. Cada solução $s' \in N(s)$ é chamado de vizinho de s.

Em linhas gerais, uma técnica de busca local, começando de uma solução inicial s_o , neste caso obtida pela fase de construção GRASP, navega pelo espaço de pesquisa, passando, interativamente, de uma solução para outra que seja sua vizinha, em busca de um ótimo local.

(Souza,2001)

Pseudo-código GRASP

O pseudo-código abaixo descreve o algoritmo GRASP.

```
GRASP (α)
{
     f* = ∞;  // f* menor valor real da função objetiva f aplicada a
              // uma solução s.
     while ( critério de parada não satisfeito) {
        s = construção (α);
        s = buscaLocal (s, f(.));
        if ( f(s) < f* ){
           s* = s;
           f* = f(s);
        }
     }
     s = s*;
     print(s);
} // GRASP
```

```
construção (α)
{
     s = vazio;
     inicialize o conjunto C de candidatos;
     while ( C não vazio) {
        tMin = min{g(t) | t ∈ C};
        tMax = Max{g(t) | t ∈ C};
        LCR = {t ∈ C | g(t) <= tMin + α(tMax – tMin)};
        selecione aleatoriamente um elemento t ∈ LCR;
        s = s U {t};
        atualize o conjunto C de candidatos;
     }
     retorne s;
}

buscaLocal(s, f(.))
{
     s* = s; // melhor solução encontrada.
     V = {s' ∈ N(s) | f(s') < f(s)};
     while ( |V| > 0) {
        selecione s ∈ V;
        if (f(s) < f(s*)) s* = s;
        V = {s' ∈ N(s) | f(s') < f(s)};
     }
     s = s*;
     retorne s;
}
```

(Souza,2001)

5.1.2.1.3.2 Sobre a População Inicial GRASP

Consulte antes seção 5.1.2.1 "Gerando a População Inicial" para se informar como criar uma população inicial para os Algoritmos Genéticos.

A População Inicial GRASP mescla metodologias adotadas na criação da População Inicial Aleatória e metodologias adotadas na criação da População Inicial Menor Tempo Acumulativo. Herda da População Inicial Menor Tempo Acumulativo o critério adotado na escolha da operação a ser inserida na sua máquina, durante a formação do cromossomo, criando a lista de candidatos restrita LCR com as melhores operações candidatas que satisfazem o critério de escolha. Também herda da População Inicial Aleatória, o critério randômico adotado na escolha da operação candidata a ser inserida na sua máquina, com apenas uma diferença em que a escolha será realizada sobre a lista de candidatos restrita.

Devemos levar em consideração as diferenças existentes na escolha de uma operação candidata a ser inserida no cromossomo, durante a sua formação, realizada pelo pseudo-código **criarCromossomoRandomico** e pelo pseudo-código **criarCromossomoMTA**.

A escolha da operação candidata realizada pelo pseudo-código **criarCromossomoMTA** é totalmente gulosa, ou seja, a operação com menor tempo de processamento é escolhida, segundo o critério Menor Tempo Acumulativo.

Na escolha da operação realizada pelo pseudo-código **criarCromossomoRandomico** cada operação candidata possui a mesma probabilidade de ser escolhida e uma dessas operações é escolhida aleatoriamente.

Os pontos positivos para uma escolha gulosa é que teremos indivíduos com certo teor de qualidade, onde o teor é estabelecido de acordo com a regra adotada para formação do cromossomo (critério de escolha adotado). Isso faz com que eliminamos as possibilidades de gerarmos indivíduos muito ruins (pouca qualidade).

Para uma escolha aleatória, temos como ponto positivo a diversidade dos indivíduos gerados.

Os pontos negativos da escolha gulosa é que temos pequena ou nenhuma diversidade dos indivíduos gerados, obtendo um mínimo local para uma população que tende a homogeneidade. Raramente nesse caso teremos bons indivíduos globais (mínimo global), uma vez que, o espaço de pesquisa se refere a indivíduos que tem pouca variação em sua qualidade (aptidão).

Os pontos negativos de uma escolha aleatória se refere às grandes possibilidades de gerarmos indivíduos ruins, ou seja, de pouca qualidade, dificultando a obtenção de bons indivíduos durante um lento processo de convergência. Freqüentemente torna-se necessário um maior tempo de pesquisa para obtenção de melhores indivíduos. (Resende,1997).

Assim, com o intuito de herdamos os pontos positivos de cada um destes métodos de escolha, dispensando os pontos negativos de cada, criaremos uma população inicial utilizando a "Fase de Construção" (seção 5.1.2.1.3.1 "GRASP") do algoritmo de otimização GRASP como metodologia para gerarmos os cromossomos. Não nos interessa nesse projeto, a "Fase de Busca Local" (seção 5.1.2.1.3.1 "GRASP") do algoritmo de otimização GRASP.

Como pontos positivos da escolha gulosa Menor Tempo Acumulativo, nossa intenção é herdar a capacidade de gerar indivíduos com algum teor de qualidade considerável e as rápidas convergências nas pesquisas (menor tempo para obtenção de bons indivíduos) e tentaremos dispensar os pontos negativos pequena ou nenhuma diversidade, a fim de evitarmos convergências prematuras. Para a escolha aleatória, nossa intenção é herdar a diversidade dos indivíduos e tentaremos dispensar a geração de grandes quantidades de indivíduos de qualidades ruins e as lentas convergências nas pesquisas.

A combinação dos pontos positivos de cada criterio de escolha permitirá criarmos uma populacao inicial de indivíduos com certa qualidade variando em torno de um valor guloso determinado pelo critério Menor Tempo Acumulativo. A variação da qualidade dos indivíduos é controlada por uma taxa gulosa que não permite termos uma população muito diversificada, mas também não homogênea.

5.1.2.1.3.3 Gerando Cromossomo GRASP

Como foi dito na seção 5.1.2.1 "Gerando a População Inicial" o conjunto de operações candidatas representa o conjunto de operações agendáveis da máquina disponível que foi escolhida para agendamento no tempo corrente *t*.

Após a formação dos conjuntos de operações candidatas, a função gulosa aplicada a cada operação de cada conjunto de operacões candidatas definirá quais operações farão parte das listas de candidatos restrita (LCR) de cada conjunto de candidatas.

A função gulosa retorna o tempo de processamento da operação passada como parâmetro a ser executada pela máquina disponível. Um valor α percentual, aplicado sobre a diferença entre o maior e o menor valor retornado pela função gulosa, permitirá apenas a inserção, na lista de candidatos restrita do conjunto de candidatas, aquelas operações candidatas que tiverem os menores valores retornados pela função gulosa, ou seja, os menores tempos correspondentes ao tempo de processamento da operação candidata.

Observe a descrição formal matemática que cria a lista de candidatos restrita para um conjunto de candidatas à máquina disponível.

- Conj. A : conjunto de operações candidatas.
- c : operação candidata. $c \in Conj.\ A$.
- o : operação.
- Crom(m) : maquina *m* do cromossomo.
- TProc(o) : tempo de processamento da operação *o*.
- Maq(o) : máquina em que a operação *o* deverá ser executada.
- MinTProc(Conj A) : operação de menor tempo *TProc* dentre as operações contidas no conjunto A.
- MaxTProc(Conj A) : operação de maior tempo *TProc* dentre as operações contidas no conjunto A.
- MTA(Conj A) : operação do conjunto A com "Menor Tempo Acumulativo".

  MTA (Conj A) = MinTProc(Conj A).
- G(c) : função gulosa. G(c) = TProc(c).
- α : taxa de gulosidade. Percentual que controla a gulosidade durante a escolha das operações candidatas a fazerem parte da lista restrita de candidatos.
- Conj LCR(α, Conj A) : lista de candidatos restrita do conjunto A com taxa de gulosidade α. Deverá conter somente as operações candidatas *c* em que $G(c) \leq (G(MTA(Conj\ A)) + \alpha(G(MaxTProc(Conj\ A)) - G(MTA(Conj\ A))))$.

Uma vez formada as listas de candidatos restrita, aleatoriamente é escolhida de cada lista a operação a ser atribuída a sua máquina.

Depois que as operações foram escolhidas das listas de candidatos restrita, estas são retiradas dos conjuntos de operações agendáveis da máquina a qual pertence e em seguida cada operação é atribuída à sua repectiva maquina, no momento disponível para agendamento.

Esse processo acontece sucessivamente ao longo do tempo *t* até que todos as operações do problema já estejam agendadas formando assim o cromossomo GRASP.

Veja o pseudo-código :

```
criarCromossomoGRASP (α)
{
  t = inicialisar;
  while ( não todos os jobs finalizados )
  {
    avançar t;
    /* check o status de cada máquina */
    for ( cada máquina )
     if ( máquina  está ociosa ou
          operação corrente dessa máquina foi finalizada )
     {
       máquina está disponível;
       /* atualizar o conjunto de operações agendáveis dessa máquina */
       if (há oper sucessora de mesmo job à última oper agendada nessa
máquina e oper sucessora não está no conjunto de opers agendáveis )
       {
           proxOp = operação sucessora;
           maq = máquina em que proxOp deverá ser atribuída;
           inserir proxOp no conjunto de operações da maquina maq;
       }
      }
    /* agendar nova operação para cada máquina */
    for (cada máquina )
     if ( essa máquina está disponível ) {
      conjunto de candidatas é conjunto de operações agendáveis da máquina;
      if ( não vazio o conjunto de operações candidatas )
      {
          opMinTProc = opMaxTProc = primeira operação do conj.candidatas;
          /* definir operação com menor tempo de processamento (opMinTProc)
             e operação com maior tempo de processamento (opMaxTProc) */
          for ( cada operação do conjunto de candidatas ) {
            if (fgulosa(operação) < fgulosa(opMinTProc) )
               opMinTempoProc = operação;
            if (fgulosa(operação) > fgulosa(opMaxTProc) )
               opMaxTempoProc = operação;
          }
          /* criar lista de candidatos restrita (LCR) */
          for ( cada operação do conjunto de candidatas )
            if (fgulosa(operacao) <= (fgulosa(opMinTProc) +
                  α(fgulosa(opMaxTProc) – fgulosa(opMinTProc))) )
               inserir operacao no conjunto LCR;
          escolher randomicamente operação do conjunto LCR;
          esvaziar LCR;
          retirar do conjunto de candidatas a operação escolhida;
          atribuir a nova operação escolhida à máquina;
          máquina não disponível;
      }
     } // if
  }  // while
} // criarCromossomoGRASP

fgulosa ( operação )  /* função gulosa */
{ retorne ( tempo proc. da operação); }
```

Executando esse pseudo-código n vezes podemos criar uma população inicial de n indivíduos.

5.1.2.1.3.4 Exemplo de Cromossomo GRASP

Veja um exemplo de cromossomo GRASP gerado a partir do problema 3/3 JSP, figura 3.1 na seção 3.1 "Definições e Premissas". Cada iteração corresponde a uma operação agendada.

- A <= B : incluir em A o conteúdo de B.

- A = B : o conteúdo de A é igual ao conteúdo de B.

- t : tempo real (tempo corrente). Representação de medida u : unidades de tempo.

- o : operação

- Mn : máquina n do problema 3/3 JSP.

- Conj. Ops Maq(n) : conjunto de operações agendáveis da máquina n.

- Conj. A : um conjunto qualquer.

- Random (Conj. A) : operação sorteada aleatoriamente do conjunto A.

- Retirar (a,Conj. A) : retirar operação a do conjunto A.

- Conj. LCR (α, Conj A) : lista de candidatos restritos do Conj A com taxa de gulosidade α.

- Crom(m) : máquina m do cromossomo.

- TProc(o) : tempo de processamento da operação o.

- TAg(o) : tempo de agendamento de uma operação o. Acontece sempre no tempo corrente t.

- ProxTDisp(Mn) : tempo em que a máquina n se encontrará disponível para agendar uma operação.
    - ProxTDisp(Mn) = t + TProc(o) : após agendamento de uma operação na máquina n no tempo t.
    - ProxTDisp(Mn) = ProxTDisp(Mn') : quando não há agendamento de uma operação em Mn no tempo t. Verifica-se antes a possibilidade de agendamento em todas as outras máquinas n' no tempo t (se estiverem disponíveis nesse tempo). Se houver possibilidade de agendamento, agende-as. Em seguida, a maquina n' que possuir o menor ProxTDisp será atribuída esse valor ao ProxTDisp da máquina n. Mn' é diferente de Mn.

- I : intervalo de ociosidade.

Condições Iniciais:

t = 0u.

Conj. Ops Maq(1) = {}; Conj. Ops Maq(2) = {}; Conj. Ops Maq(3) = {}

Iteração 1:

1- Conj. Ops Maq(1) = {}; Conj. Ops Maq(2) = {1,1,2; 3,1,2}; Conj. Ops Maq(3) = {2,1,3}

2- Verificar Conj. Ops Maq(1).

- Nenhuma operação encontrada. Logo M1 é ociosa.

3- Verificar Conj. Ops Maq(2).

- Conj. LCR (0.5, Conj. Ops Maq(2)) = {1,1,2}
- Crom(2) <= (Random(Conj. LCR) = {1,1,2})

4- Retirar ({1,1,2}, Conj. Ops Maq(2)) : Conj. Ops Maq(2) = {3,1,2}

5- TAg({1,1,2}) = 0u

6- ProxTDisp(M2) = 0 + (Tproc({1,1,2}) = 5) = 5u.

Iteração 2:

1- Conj. Ops Maq(1) = {}; Conj. Ops Maq(2) = {3,1,2}; Conj. Ops Maq(3) = {2,1,3}

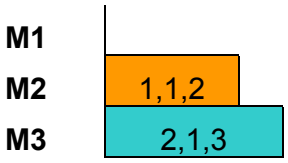
2- Verificar Conj. Ops Maq(3).

- Conj. LCR (0.5, Conj. Ops Maq(3)) = {2,1,3}
- Crom(3) <= (Random(Conj. LCR) = {2,1,3})

3- Retirar ({2,1,3}, Conj. Ops Maq(3)) : Conj. Ops Maq(3) = {}

4- TAg({2,1,3}) = 0u

5- ProxTDisp(M3) = 0 + (Tproc({2,1,3}) = 7) = 7u.



Iteração 3:

1- Conj. Ops Maq(1) = {}; Conj. Ops Maq(2) = {3,1,2}; Conj. Ops Maq(3) = {}

2- Verificar ociosidade em M1. ProxTDisp(M1) = 5u. (menor ProxTDisp)

3- t = 5u. (menor ProxTDisp).

4- Atualizar todos Conj. Ops Maq.

- Conj. Ops Maq(2) <= {1,2,2} : Conj. Ops Maq(2) = {3,1,2; 1,2,2}

5- Verificar Conj Ops Maq(1).

- Nenhuma operação encontrada. Logo M1 é ociosa.

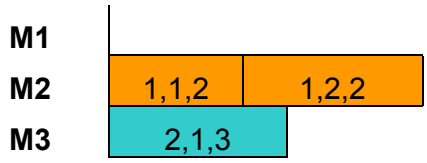
6- Verificar Conj Ops Maq(2).

- Conj. LCR (0.5, Conj. Ops Maq(2)) = {3,1,2; 1,2,2}
- Crom(2) <= (Random(Conj. LCR) = {1,2,2})

7- Retirar ({1,2,2}, Conj. Ops Maq(2)) : Conj. Ops Maq(2) = {3,1,2}

8- TAg({1,2,2}) = 5u

9- ProxTDisp(M2) = 5 + (Tproc({1,2,2}) = 7) = 12u.



Iteração 4:

1- Conj. Ops Maq(1) = {}; Conj. Ops Maq(2) = {3,1,2}; Conj. Ops Maq(3) = {}

2- Verificar ociosidade em M1. ProxTDisp(M1) = 7u. (menor ProxTDisp)

3- t = 7u. (menor ProxTDisp)

4- Atualizar todos Conj. Ops Maq.

- Conj. Ops Maq(2) <= {2,2,2} : Conj. Ops Maq(2) = {3,1,2; 2,2,2}

5- Verificar operação do Conj. Ops Maq(1). Nenhuma operação encontrada. Logo M1 é ociosa.

6- Verificar ociosidade em M1. ProxTDisp(M1) = 12u. (menor ProxTDisp)

7- t = 12u. (menor ProxTDisp).

8- Atualizar todos Conj. Ops Maq.

- Conj. Ops Maq(1) <= {1,3,1} : Conj. Ops Maq(1) = {1,3,1}

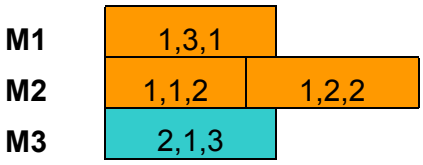
9- Verificar operação do Conj. Ops Maq(1).

- Conj. LCR (0.5, Conj. Ops Maq(1)) = {1,3,1}
- Crom(1) <= (Random(Conj. LCR) = {1,3,1})

10- Retirar ({1,3,1}, Conj. Ops Maq(1)) : Conj. Ops Maq(1) = {}

11- TAg({1,3,1}) = 12u

12- ProxTDisp(M1) = 12 + (Tproc({1,3,1}) = 7) = 19u.



Iteração 5:

1- Conj. Ops Maq(1) = {}; Conj. Ops Maq(2) = {3,1,2; 2,2,2}; Conj. Ops Maq(3) = {}

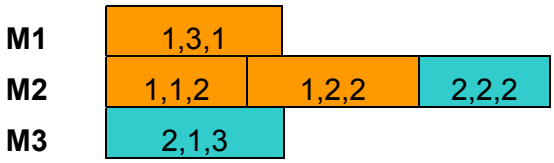
2- Verificar operação do Conj. Ops Maq(2).

- Conj. LCR (0.5, Conj. Ops Maq(2)) = {2,2,2}
- Crom(2) <= (Random(Conj. LCR) = {2,2,2})

3- Retirar ({2,2,2}, Conj. Ops Maq(2)) : Conj. Ops Maq(2) = {3,1,2}

4- TAg({2,2,2}) = 12u

5- ProxTDisp(M2) = 12 + (Tproc({2,2,2}) = 5) = 17u.



Iteração 6:

1- Conj. Ops Maq(1) = {}; Conj. Ops Maq(2) = {3,1,2}; Conj. Ops Maq(3) = {}

2- t = 17u. (menor ProxTDisp).

3- Atualizar todos Conj. Ops Maq.

- Conj. Ops Maq(1) <= {2,3,1} : Conj. Ops Maq(1) = {2,3,1}

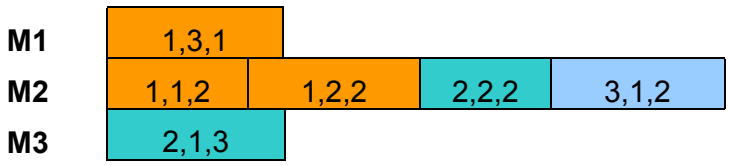
4- Verificar Conj Ops Maq(2).

- Conj. LCR (0.5, Conj. Ops Maq(2)) = {3,1,2}
- Crom(2) <= (Random(Conj. LCR) = {3,1,2})

5- Retirar ({3,1,2}, Conj. Ops Maq(2)) : Conj. Ops Maq(2) = {}

6- TAg({3,1,2}) = 17u

7- ProxTDisp(M2) = 17 + (Tproc({3,1,2}) = 7) = 24u.



Iteração7:

1- Conj. Ops Maq(1) = {2,3,1}; Conj. Ops Maq(2) = {}; Conj. Ops Maq(3) = {}

2- t = 19u. (menor ProxTDisp).

3- Atualizar todos Conj. Ops Maq. Nenhum conjunto atualizado.

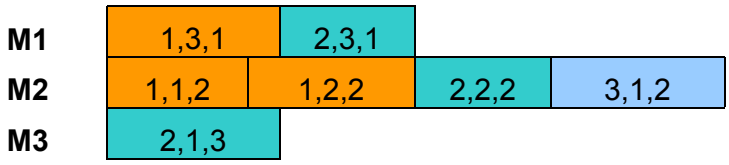
4- Verificar Conj Ops Maq(1).

- Conj. LCR (0.5, Conj. Ops Maq(2)) = {2,3,1}
- Crom(1) <= (Random(Conj. Ops Maq(1)) = {2,3,1})

5- Retirar ({2,3,1}, Conj. Ops Maq(1)) : Conj. Ops Maq(1) = {}

6- TAg({2,3,1}) = 19u

7- ProxTDisp(M1) = 19 + (Tproc({2,3,1}) = 5) = 24u.



Iteração 8:

1- Conj. Ops Maq(1) = {}; Conj. Ops Maq(2) = {}; Conj. Ops Maq(3) = {}

2- t = 24u. (menor ProxTDisp).

3- Atualizar todos Conj. Ops Maq.

- Conj. Ops Maq(1) <= {3,2,1} : Conj. Ops Maq(1) = {3,2,1}

4- Verificar Conj. Ops Maq(1).

- Conj. LCR (0.5, Conj. Ops Maq(1)) = {3,2,1}
- Crom(1) <= (Random(Conj. LCR) = {3,2,1})

5- Retirar ({3,2,1}, Conj. Ops Maq(1)) : Conj. Ops Maq(1) = {}

6- TAg({3,2,1}) = 24u

7- ProxTDisp(M1) = 24 + (Tproc({3,2,1}) = 5) = 29u.

M1	1,3,1	2,3,1	3,2,1	
M2	1,1,2	1,2,2	2,2,2	3,1,2
M3	2,1,3			

Tabela 5.11 – Cromossomo GRASP

Durante a formação do cromossomo, o pseudo-código **criarCromossomoGRASP** utiliza um valor α percentual que controla a gulosidade na escolha da operação a ser atribuída à máquina disponível.

A lista de candidatos restrita de uma máquina disponível é criada a partir do conjunto de operações candidatas à serem agendadas na máquina contendo as operações que possuírem os menores tempos de processamento de acordo com a taxa gulosa. Uma operação é escolhida aleatoriamente da lista de candidatos restrita.

O cromossomo gerado representa a ordem em que as operações deverão ser executadas pelas máquinas. Voltamos a lembrar que as operações são codificadas na forma de valores inteiros únicos (genótipos) que as representa.

O indivíduo é representado abaixo com tempo de finalização igual a 29 unidades de tempo.

M1	I	I	1,3,1	2,3,1	3,2,1
M2	1,1,2	1,2,2	2,2,2	3,1,2	
M3	2,1,3				

Tabela 5.12 – Indivíduo GRASP

5.1.2.2 Relevâncias sobre a População Inicial

Durante a formação do cromossomo, inserimos nos conjuntos de operações de cada máquina (conjunto de operações agendáveis da máquina) apenas as operações que seguem a ordem de execução estabelecidas pelo seu job. Isso porque queremos gerar permutações válidas, ou seja, indivíduos válidos para o problema Job-Shop e é por isso que durante a formação do cromossomo agendamos as operações possibilitando somente a criação de cromossomos legítimos idênticos ao indivíduo quanto à ordem das operações agendadas em cada máquina. Cada permutação (ou máquina), no cromossomo formado, definirá a ordem em que as operações deverão ser executadas. Como o cromossomo é apenas uma solução codificada, os tempos de agendamento de cada operação são visíveis somente no processo de decodificação do cromossomo gerando o indivíduo (solução).

No critério guloso Menor Tempo Acumulativo somente é escolhida a operação a ser agendada na máquina disponível que pertença ao conjunto de operações agendáveis da

máquina e que possua menor tempo de processamento dentre as operações candidatas no conjunto.

O fato de podermos escolher uma operação segundo o critério de escolha gulosa Menor Tempo Acumulativo gera uma população inicial de indivíduos idênticos em aptidão, que tende a uma rápida convergência prematura, prejudicando a evolução natural.

À medida que a escolha se faz aleatória a tendência é aumentarmos a diversidade entre os indivíduos gerados, ou seja, termos uma população com grandes variações nos valores de aptidão dos seus indivíduos e possibilidades de obtermos melhores soluções. Entretanto, a diversidade dos indivíduos deverá ser controlada.

Nossa intenção aqui é fugir de métodos totalmente aleatórios que podem prejudicar a eficiência dos algoritmos genéticos devido a grandes diversidades de indivíduos gerados podendo, inclusive, gerar muitos indivíduos ruins (de pouca aptidão). Nesse caso, o algoritmo levaria maior tempo para gerar melhores indivíduos como solução conduzindo a uma lenta convergência durante o processo evolutivo. Por outro lado, também não queremos populações iniciais geradas por um critério guloso convergindo prematuramente a população.

Quanto maior o grau de gulosidade (no caso do algoritmo guloso, a gulosidade é total) maiores são as possibilidades de obtermos uma população homogênea.

Gerar indivíduos homogêneos significa gerar cromossomos que após passarem pelo processo de agendamento acabam gerando o mesmo indivíduo ou semelhantes em valores de aptidão. Uma população inicial com indivíduos muito parecidos ou idênticos tornam o espaço de pesquisa para os algoritmos genéticos ineficiente, com pouca diversidade, o que aumenta a possibilidade de aproximação para um mínimo local, ou seja, aproximação para a melhor solução do problema Job-Shop (solução com menor tempo de execução) que poderia ser melhor se o algoritmo de otimização explorasse uma população mais diversificada.

Um mínimo local é gerado pelo fato dos indivíduos muito semelhantes quando selecionados para reprodução gerar descendentes que se assemelham com seus pais (os filhos herdam genes dos seus pais) e algumas vezes idênticos, o que dificulta a evolução dos indivíduos, ou seja, a tendência é termos pouca variação nos valores de aptidão dos indivíduos gerados, que farão parte da população sobrevivente. Isso foi comprovado muitas vezes durante os testes que utilizaram o critério de escolha Menor Tempo Acumulativo.

Assim, dá para perceber que a população inicial é muito importante para geração de melhores soluções para um problema.

Uma população inicial homogênea (pouca variação nos valores de aptidão dos indivíduos) tende a gerar um mínimo local. Quanto maior a variação dos valores de aptidão entre os indivíduos maior é a possibilidade de gerar indivíduos mais diversificados. No entanto, uma diversificação muito grande entre os indivíduos tende a exigir maiores gerações na execução do algoritmo genético para reproduzir indivíduos que representem boas soluções para o problema. Um maior número de convergências se faz necessário para que os valores de aptidão dos indivíduos sobreviventes tenham uma menor variação tornando a população mais homogênea e daí obtemos o melhor indivíduo, dentre as gerações percorridas.

Um critério eficiente de fim de execução do algoritmo (critério de parada), nesse caso, poderia ser o de controle de homogeneidade, ou seja, no momento em que um determinado valor representando a variação dos valores de aptidão na população for menor que a constante de variação de aptidão definida (constante de homogeneidade) o algoritmo finalizaria retornando o melhor indivíduo na geração. Outro critério de parada também interessante poderia envolver um número limite de interações sem melhora no valor de aptidão do melhor indivíduo gerado até então.

Para controle de diversidade utilizamos o critério de escolha gulosa Menor Tempo Acumulativo combinado com o critério de escolha Randômica, ambos com aplicação controlada por meio de uma taxa (taxa gulosa) na formação dos cromossomos durante a criação da população inicial. A escolha por Menor Tempo Acumulativo, ajudará a evitar a geração de indivíduos muito variados proporcionados pela escolha Randômica. Essa combinação de critérios adotada para a população inicial damos o nome de População Inicial GRASP. O método heurístico GRASP (Procedimento de Busca Adaptativa Gulosa e Randomizada) é abordado de forma mais detalhada na seção 5.1.2.1.3 " População Inicial GRASP".

As escolhas randômicas adotadas possuem o papel de principal diversificador entre os indivíduos gerados na população inicial, para problemas de pequenos ou grandes tamanhos, e as escolhas gulosas possuem o papel de controlador de diversidade.

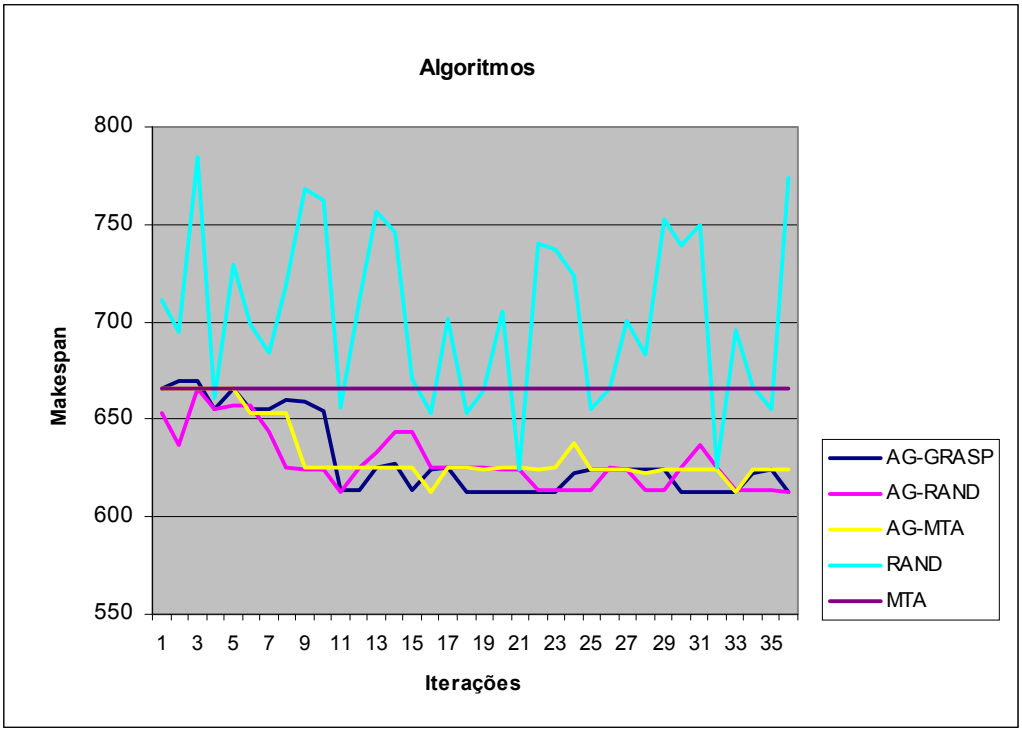
Criamos a População Inicial Aleatória e a População Inicial Menor Tempo Acumulativo para podermos comparar a eficiência dos indivíduos gerados pelo algoritmo genético que utiliza uma População Inicial GRASP.

5.1.2.3 Testes

A figura 5.4 define um gráfico que exibe a evolução das melhores soluções geradas a cada iteração pelos Algoritmos Genéticos (AG's). Uma iteração corresponde, para os Algoritmos Genéticos, a uma geração onde a melhor solução obtida na geração é exibida pelo gráfico no valor Makespan (tempo de finalização na execução dos jobs de uma instância do problema Job-Shop).

Cada Algoritmo Genético adota como população inicial : População Inicial Aleatória (AG-RAMD); População Inicial Menor Tempo Acumulativo (AG-MTA); e População Inicial GRASP (AG-GRASP).

O gráfico também exibe as soluções produzidas a cada iteração pelo Algoritmo Guloso (MTA) e pelo Algoritmo Aleatório (RAND). Cada iteração, para os algoritmos guloso MTA e aleatório RAND, é gerada apenas uma única solução exibida pelo gráfico no valor Makespan.



Problema Job-Shop	
Jobs	*Maquinas*
10	5

Melhor Solução : 613 u (unidades de tempo).

Algoritmos	**Parâmetros**					
AG's	*Taxas*			*População*		*Gerações Percorridas*
	Crossover	Mutação	Gulosa*	Inicial	Tamanho Máximo	37
	0.95	0.05	0.40	15	30	
MTA e RAND	*Iterações Percorridas*					
	37					

Nota : * Taxa Gulosa é utilizada somente pelo algoritmo AG-GRASP

Figura 5.4 – Gráfico Makespan / Iterações

Veja no gráfico (figura 5.4) a tendência a convergências prematuras do Algoritmo Genético (AG-MTA) que possui população inicial totalmente gulosa enquanto que o Algoritmo Genético (AG-RAND), que possui população inicial totalmente aleatória, tende a variar mais as soluções obtidas durante as gerações comparando com os outros Algoritmos Genéticos. O Algoritmo Aleatório (RAND) exibe as diversas qualidades de soluções produzidas a cada iteração e o Algoritmo Guloso (MTA) exibe a única solução gerada.

A seguir, críticas serão realizadas para cada algoritmo exposto no gráfico.

5.1.2.3.1 Os Algoritmos Genéticos

Algumas críticas foram realizadas para os Algoritmos Genéticos implementados em função dos resultados obtidos pelos testes realizados durante a execução de tais algoritmos.

A população inicial se tornou o principal ponto de referência para sofisticação dos Algoritmos Genéticos nesse projeto e é por isso que as críticas estão relacionadas ao critério utilizado para geração da população inicial nos Algoritmos Genéticos.

5.1.2.3.1.1 Críticas para População Inicial Aleatória

Consulte antes a seção 5.1.2.1.1 "População Inicial Aleatória" para saber como é gerada a população inicial aleatória. A seção exibe um exemplo de criação do cromossomo aleatório.

Segue-se para os problemas:

- *Para problemas de tamanho pouco consideráveis*

O algoritmo genético AG-RAND não mostra resultados ruins.

Durante a criação do cromossomo aleatório, é pequeno o número de operações para escolha no conjunto de operações de cada máquina (o que aumenta as chances de escolhas repetitivas acontecerem), assim, a escolha aleatória da operação no conjunto agendável de candidatas é um bom critério para aumentarmos a diversidade entre os indivíduos a serem gerados, uma vez que, a tendência é termos indivíduos semelhantes quanto menor for o tamanho do problema.

- *Para problemas de tamanho consideráveis*

Os indivíduos gerados tendem a se tornar bastante diversificados.

Durante a criação do cromossomo aleatório, são maiores os espaços de pesquisa, para escolha aleatória, que representam o número de operações no conjunto de operações agendáveis de cada máquina, o que aumenta o poder de diversificação do algoritmo genético AG-RAND na geração de indivíduos para a população inicial.

Os testes realizados vieram a comprovar uma maior variedade de indivíduos gerados (maior variação nos valores de aptidão dos indivíduos) e um lento processo de convergência a melhores soluções durante as iterações. Seria interessante adotar um critério de controle de diversidade para impedir a geração de muitos indivíduos de pouca aptidão evitando um lento processo de convergência.

5.1.2.3.1.2 Críticas para População Inicial Menor Tempo Acumulativo

Consulte antes a seção 5.1.2.1.2 "População Inicial Menor Tempo Acumulativo" para saber como é gerada a população inicial que adota o critério guloso Menor Tempo Acumulativo. A seção exibe um exemplo de criação do cromossomo Menor Tempo Acumulativo.

Assim segue-se para os problemas:

- *Independentemente do tamanho do problema*

O algoritmo genético AG-MTA gera um único indivíduo para a população inicial.

Por gerar um único indivíduo, a criação da população inicial realizada por meio desse algoritmo é completamente homogênea com enormes possibilidades de obtermos uma convergência prematura durante o processo evolucionário.

Um único indivíduo é gerado durante a formação do cromossomo na população inicial, porque o algoritmo escolhe somente a operação gulosa com Menor Tempo Acumulativo no conjunto de operações candidatas a ser inserida na máquina disponível. A escolha entre as operações do conjunto de operações candidatas não tem nenhum teor aleatório que possa diversificar os indivíduos gerados.

Os resultados não são satisfatórios, pois são grandes as possibilidades de termos um mínimo local com a convergência prematura. Nesse caso, um aumento controlado da taxa de mutação (troca aleatória de informações contidas nos genes dos cromossomos) aplicada sobre os descendentes, após o crossover, poderia contribuir para geração de indivíduos com qualidades variadas e assim minimizar a possibilidade de termos a convergência prematura. Entretanto, taxas de mutação elevadas não faz nenhum sentido, pois teríamos no meio mais mutação que reprodução dos indivíduos fugindo às regras da evolução natural.

5.1.2.3.1.3 Críticas para População Inicial GRASP

Consulte antes a seção 5.1.2.1.3 "População Inicial GRASP" para saber como é gerada a população inicial que adota o critério herdado do método heurístico GRASP. A seção exibe um exemplo de criação do cromossomo GRASP.

Assim segue-se para os problemas:

- *Para problemas de tamanho pouco consideráveis*

Pelo fato de ser mais restrita a área de pesquisa no conjunto de operações agendáveis de cada máquina, durante a formação do cromossomo GRASP, é considerável que, grande quantidade das operações do conjunto de operações candidatas (conjunto de operações agendáveis da máquina disponível para agendamento) façam parte da lista de candidatos restrita aumentando o número de operações para escolha randômica. Para isso, basta apenas que a taxa de gulosidade seja α grande o suficiente [0.7,0.9] diminuindo as possibilidades de gerar indivíduos homogêneos na população inicial. Para $\alpha = 1$, o algoritmo genético **AG-GRASP**, adotaria o critério aleatório do algoritmo genético **AG-RAND.**

É bom lembrarmos que, quanto menor o tamanho do problema e menor for o valor α , maiores são as possibilidades de obtermos indivíduos homogêneos na população inicial convergendo prematuramente as soluções para o problema ao longo da evolução natural.

- *Para problemas de tamanho consideráveis*

Nesse caso, o critério aleatório (fator de diversidade) geraria indivíduos muito diversificados em aptidão aumentando as possibilidades de obtermos muitos indivíduos pobres para a população inicial.

Assim, um valor α mediano controlaria a diversidade produzida pelas escolhas aleatórias, que ao invés de serem realizadas sobre uma maior área nos conjuntos de operações agendáveis de cada máquina (maior número de operações nos conjuntos), durante a formação do cromossomo GRASP, serão realizadas sobre uma lista de candidatos restrita contento apenas as operações candidatas que satisfazem o criterio de escolha gulosa aplicado com taxa de gulosidade α. O valor α pequeno adotaria um rígido critério de escolha gulosa Menor Tempo Acumulativo sobre as operações no conjunto de operações agendáveis candidatas, diminuindo o número de operações a fazerem parte da lista de operações restritas. Quanto menor α, maior é a gulosidade, ou seja, tende a se tornar menor a lista de candidatos restrita priorizando as operações gulosas. Para $\alpha = 0$, é escolhida sempre a operação com menor tempo de processamento (gulosidade total), segundo o critério de escolha Menor Tempo Acumulativo.

O algoritmo genético **AG-GRASP**, considerando problemas de tamanhos consideráveis, adotaria o critério Menor Tempo Acumulativo do algoritmo genético **AG-MTA,** sob uma taxa gulosa mediana, para evitar um lento processo de convergência. Para um valor α mediano, os resultados obtidos pelo algoritmo genético híbrido são bastante satisfatórios, uma vez que, os indivíduos gerados para a população inicial não são muito diversificados e também não são homogêneos facilitando a convergência do algoritmo para boas soluções.

O critério de escolha Menor Tempo Acumulativo tem a função de controlar a diversidade produzida pelas escolhas aleatórias sobre um maior número de operações nos conjuntos agendáveis, criando uma lista restrita. Em contrapartida, a forte tendência homogênea do critério de escolha Menor Tempo Acumulativo é amenizado pelas escolhas randômicas realizadas na lista de candidatos restrita.

Quanto maior o problema a ser tratado, maiores as dificuldades intrínsecas envolvidas, maiores as restrições de precedência, se tornando assim maiores as dificuldades em obter melhores soluções para o problema. A possibilidade de obtermos maiores quantidades de indivíduos ruins em aptidão também é maior. Nesse caso, para escolhas puramente randômicas, o processo de convergência para melhores soluções se tornaria muito lento

durante as interações e para escolhas puramente gulosas teríamos uma convergência prematura.

Para tornar mais evidente as críticas realizadas para a População Inicial GRASP, Feo & Resende(1995), diante dos resultados obtidos pelos métodos heurísticos GRASP, anunciaram o seguinte : "valores de α que levam a uma lista de candidatos restrita de tamanho muito limitado (ou seja, valor de α próximo da escolha gulosa) implicam em soluções finais de qualidade muito próxima àquela obtida de forma puramente gulosa, obtidas com um baixo esforço computacional. Em contrapartida, provocam uma baixa diversidade de soluções construídas. Já uma escolha próxima da seleção puramente aleatória leva a uma grande diversidade de soluções construídas mas, por outro lado, muitas das soluções construídas são de qualidade inferior, tornando mais lento a convergência".

Assim, o parâmetro α , que determina o tamanho da lista de candidatos restrita, é basicamente o único parâmetro a ser ajustado na implementação de uma População Inicial GRASP.

A População Inicial GRASP adotada aos Algoritmos Genéticos procura conjugar bons aspectos dos algoritmos puramente gulosos, com aqueles dos algoritmos aleatórios na construção dos indivíduos. A População Inicial GRASP gera indivíduos que constitui bons pontos de partida para os Algoritmos Genéticos, permitindo assim acelerá-lo.

Enfim, seria interessante o algoritmo genético **AG-GRASP** adotar uma taxa de gulosidade (valor α) flexível podendo aumentar ou diminuir de acordo com as variações no tamanho dos problemas Job-Shop's, contribuindo para criação de uma população inicial que possa ajudar na obtenção de melhores indivíduos para os Algoritmos Genéticos.

5.1.2.3.2 Os Algoritmos Guloso e Aleatório

Os algoritmos Gulosos e Aleatório foram desenvolvidos com o principal objetivo de herdarmos seus pontos positivos no processo de busca de melhores soluções para o problema Job-Shop através de Algoritmos Genéticos.

Seus pontos positivos foram utilizados pelo Algoritmo Genético híbrido que implementa uma população inicial chamada População Inicial GRASP modelada nos princípios adotados pelo método heurístico GRASP, que combina tais pontos positivos.

Aproveitamos também os algoritmos Guloso e Aleatório durante os testes para compararmos a eficiência do algoritmo genético híbrido.

5.1.2.3.2.1 Algoritmo Aleatório

Dificilmente poderemos obter soluções satisfatórias para um problema, devido à falta de controle de diversidade sobre esse algoritmo.

A cada iteração uma solução é gerada aleatoriamente e comparada com a solução obtida da iteração anterior. Esse algoritmo, não implementa uma lógica para obtenção de boas soluções. As soluções são simplesmente geradas ao acaso. Não significa que não possamos obter boas soluções, até mesmo porque o processo randômico varia a qualidade das soluções obtidas, o problema é que, escolhas totalmente aleatórias podem consumir muito tempo para

gerar uma boa solução para o problema devido a grande variedade de soluções produzidas, entre elas, várias soluções pobres.

Escolhas totalmente randômicas geram uma grande variedade de soluções. A diversidade é um ponto positivo quando desejamos melhorar o espaço de pesquisa na população dos Algoritmos Genéticos sem aumentar o tamanho da população. Evita convergências prematuras.

5.1.2.3.2.2 Algoritmo Guloso Menor Tempo Acumulativo

Esse algoritmo gera uma única solução de acordo com a regra de escolha Menor Tempo Acumulativo. A solução é denominada solução puramente gulosa.

A vantagem desse algoritmo é que teremos uma solução com algum teor de qualidade. Não podemos dizer que a solução gerada tem uma boa qualidade, que seja uma solução satisfatória para uma determinada instância do Problema Job-Shop, entretanto, a solução gerada não é uma solução totalmente pobre, em virtude da regra utilizada.

Utilizamos princípios desse algoritmo para criar soluções que têm qualidades que variam em torno da qualidade da solução puramente gulosa para constituir uma população inicial dos Algoritmos Genéticos. Esses princípios são manifestados no critério de escolha GRASP adotado para criar a população inicial do Algoritmo Genético híbrido. A variação da qualidade das soluções em torno da solução puramente gulosa ocorre mediante um valor $\propto$ denominado taxa de gulosidade responsável por controlar a variedade dos indivíduos gerados durante a formação da população inicial.

Essa técnica GRASP adotada na população inicial certamente diminuirá o tempo de convergência dos Algoritmos Genéticos durante a procura de melhores soluções, apesar de não podermos dizer, com rigor, o quanto essa técnica é eficiente em relação a uma população inicial gerada aleatoriamente. Isso porque uma série de fatores devem ser considerados durante os processos de convergências nos Algoritmos Genéticos, como por exemplo, os valores dos parâmetros definidos, os operadores de convergência adotados, a função de avaliação, entre outros.

Algoritmos Genéticos que utilizam um Algoritmo Guloso, sem variação na gulosidade do algoritmo (gulosidade total), terão uma população inicial totalmente homogênea e assim uma convergência prematura, prejudicando a evolução natural dos Algoritmos Genéticos.

5.2 A Função de Avaliação e o Processo Seletivo

Um dos itens estudados para aplicação no problema Job-Shop que merece atenção especial é a função de avaliação adotada e o processo de seleção de indivíduos para reprodução.

5.2.1 A Função de Avaliação

A avaliação consiste simplesmente na aplicação de uma função de avaliação em cada um dos indivíduos da população corrente. Como visto anteriormente, essa função deve expressar a qualidade de cada indivíduo daquela população, no contexto do problema considerado.

Essa função é extremamente importante, uma vez que constitui o único elo entre o algoritmo genético e o problema Job-Shop, depende fortemente da forma como as soluções foram representadas.

Como os Algoritmo Genéticos trabalham sobre uma população de soluções codificadas, após o agendamento das operações codificadas no cromossomo obtemos o tempo de finalização do indivíduo (tempo gasto para que todos os jobs sejam executados pelas máquinas) que é fornecida pela função objetiva aplicada ao cromossomo.

O problema Job-Shop é um problema de minimização, ou seja, desejamos obter o individuo de menor tempo de finalização (makespan), de menor valor retornado pela função objetiva. Dessa forma, a função de avaliação para o problema deverá atribuir ao indivíduo de menor tempo de finalização a melhor aptidão. Quanto menor o tempo de finalização de um indivíduo mais apto ele é.

Observe o gráfico da figura 5.5 que mostra como foram obtidos os valores de aptidão dos indivíduos na população, para o problema Job-Shop, de acordo com o seu tempo de finalização.

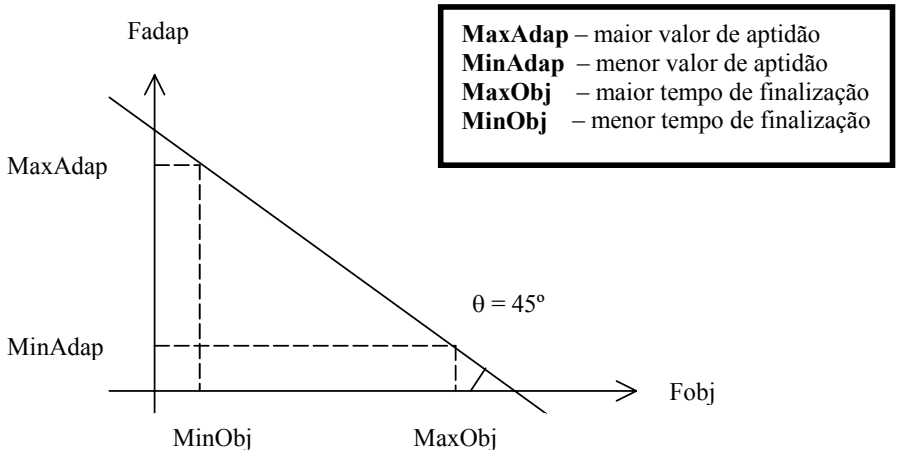


Figura 5.5 – Gráfico Fobj x Fadap

O gráfico da figura 5.5 aplica um valor de aptidão Y para um individuo de tempo de finalização X, onde Fobj (Função Objetiva) é a coordenada X que determina o tempo de finalização do indivíduo e Fadap (Função de Avaliação) é a coordenada Y que determina o valor de aptidão.

Definimos como função de avaliação de um individuo a seguinte equação matemática:

(1) : Fadap(X) = (tg135º)X + B = (-1)X + B

(2) : MinAdap = (-1)MaxObj + B

: B = MinAdap + MaxObj

Logo: para as equações (1) e (2) temos (3) :

(3) : Fadap(X) = (-1)X + MinAdap + MaxObj

Para o problema Job-Shop, consideramos que, o individuo de maior tempo de finalização (MaxObj) possui o menor valor de aptidão (MinAdap) que corresponde ao menor tempo de finalização (MinObj) encontrado entre os indivíduos da população. Logo temos :

(4) : MinAdap = MinObj

Assim, para as equações (3) e (4) temos (5) :

(5) : Fadap(I) = (-1)Fobj(I) + MinObj + MaxObj

: onde I representa um indivíduo.

Uma vez obtida a avaliação de cada individuo na população poderemos mediante um critério de seleção natural, selecionar os indivíduos para reprodução, ou seja, para aplicarmos os operadores genéticos para geração de descendentes. Lembre-se que os indivíduos mais aptos têm maiores chances de serem escolhidos. Não significa que os menos aptos não serão escolhidos, apenas a sua chance de escolha é menor.

Os valores de aptidão dos indivíduos na população é o único parâmetro para o processo seletivo que determina quais indivíduos serão escolhidos para serem os geradores, ou seja, que passarão pelo processo de reprodução para geração de descendentes a compor a nova população sobrevivente. Portanto, a análise dos valores de aptidão dos indivíduos são muito importantes para a composição da população a sobreviver durante gerações. Em vista disso, enfatizamos o processo seletivo na seção seguinte, onde a função de aptidão (ou avaliação) será argumentada com mais detalhes.

5.2.2 O Processo Seletivo

Esta etapa de um algoritmo genético é responsável, como o próprio nome indica, pela seleção de indivíduos de uma população. Os indivíduos serão selecionados de acordo com sua aptidão.

O Operador de Seleção dos Algoritmos Genéticos é uma analogia com a seleção biológica natural de Darwin.

Realizar uma seleção significa escolher dentre os indivíduos pertencentes à população, àqueles que servirão para criar descendentes para a próxima geração e quantos descendentes cada um deverá gerar. Uma seleção pode ser denominada *muito fraca*, se resultar em evolução muito baixa e *muito forte* se resultar em populações de indivíduos mais fortes (o termo *forte* aqui utilizado, refere-se aos indivíduos mais adaptados ao meio ambiente; esta adaptação é que direciona o processo evolutivo). Isto significa que os indivíduos mais adequados vivem

mais e geram mais descendentes, os quais herdam suas qualidades. Fica portanto evidente, que o mecanismo da Seleção tem por finalidade enfatizar um filtrador de indivíduos na população, para aumentar a perspectiva de gerar indivíduos melhores segundo algum critério.

A dominância dos cromossomos com alto valor de aptidão nas populações pode fazer com que o algoritmo venha a convergir muito rapidamente (convergência prematura) para um ponto de mínimo local no espaço de soluções para o problema Job-Shop. Isso acontece porque teremos seleções apenas de indivíduos muito fortes em aptidão, tornando a população pobre em diversidade. A tendência nesse caso, é de gerarmos indivíduos, após aplicação dos operadores de reprodução, muito semelhantes ou homogêneos. Essa tendência de escolha de somente indivíduos fortes é chamada de pressão de seleção.

Deste fato, surge a necessidade de modificar a função de avaliação em prol da contribuição dos indivíduos menos aptos. O processo inicia-se com a conversão de cada aptidão individual em uma expectativa que é o número esperado de descendentes que cada indivíduo poderá gerar.

A seguir, serão apresentadas duas maneiras para o cálculo dos valores das expectativas.

5.2.2.1 Expectativas

Nesse projeto, daremos prioridade ao mecanismo Proporcionalidade do *Fitness* e ao mecanismo Escala Truncamento Sigma que foram os dois mecanismos estudados e avaliados para obtenção das expectativas dos indivíduos.

5.2.2.1.1 Proporcionalidade do Fitness

No algoritmo original de Holland (1993), a expectativa individual era calculada através da divisão da aptidão individual pela média das aptidões de toda a população da geração corrente. Desta forma, a expectativa *Exp,* de um indivíduo *I*, é calculada por:

$Exp(I) = Fadap(I) / Madap_t$,

: onde *Fadap(I)* é a aptidão do indivíduo *I*, e $Madap_t$ é a aptidão média da população no tempo *t*.

Em seguida, utilizava-se o mecanismo de seleção por Giro de Roleta (mais comum) para sorteio dos indivíduos baseados em suas expectativas. Na seção 4.4.1.3.1, "Métodos de Seleção" é argumentado o método.

O mecanismo Proporcionalidade do *Fitness* (aptidão) pode apresentar efeitos indesejáveis em duas ocasiões durante o seu processamento na seleção dos indivíduos :

- Logo no início, quando a população é aleatoriamente calculada e a média das aptidões é baixa, indivíduos mais aptos podem receber um número desordenado de descendentes, dispensando outras regiões do espaço de busca, conduzindo a uma convergência prematura. Posteriormente, quando existe muita semelhança entre os indivíduos da população, não existem diferenças de aptidão que sejam capazes de continuar a evolução.

- No decorrer do processamento, esse método tem uma forte tendência em alocar a todos os indivíduos um descendente, mesmo para aqueles indivíduos com expectativa muito baixa (indivíduos ruins), resultando em uma exploração ineficiente do espaço de busca e consequentemente em um retardo acentuado na convergência do método.

5.2.2.1.2 Sigma Truncation Scaling

Um segundo método de conversão de valores de aptidão em números esperados de descendentes, referenciado em Goldberg (1989) como o método Sigma Truncation Scaling, é baseado no valor de aptidão do indivíduo, na aptidão média de toda a população e no desvio padrão das aptidões sobre a população. Sua idéia básica é também muito simples:

$Exp(I) = Fadap(I) – Madap_t – c \times desvio_t$

: onde *Fadap(I)* é a aptidão do indivíduo *I*, $Madap_t$ é a aptidão média da população no tempo *t, desvio* é o desvio padrão das aptidões na população no tempo *t* e c é uma constante de desvio pertencente ao conjunto dos reais maior que zero.

: todos os valores negativos são alterados para zero.

Em seguida, poderíamos utilizar a seleção por Giro de Roleta para sorteio dos indivíduos baseados em suas expectativas. Na seção 4.4.1.3.1, "Métodos de Seleção" é argumentado o método.

O mecanismo Sigma Truncation Scaling permite que sejam evitados a escolha de indivíduos de aptidão muito ruins que estejam abaixo da linha limite ($Madap_t – c \times desvio_t$) na população, atribuindo o valor de expectativa igual a zero para tais indivíduos. O valor c controla a linha limite.

Este método mantém a pressão de seleção quase constante durante o processo de busca da melhor solução, sem depender da variância da aptidão da população.

O método foi escolhido para gerar os valores de expectativa para os indivíduos, durante a implementação do problema Job-Shop, como meio para amenizar os problemas de convergências, evitando a escolha, para reprodução, de indivíduos muito ruins e também a convergência prematura.

5.2.2.2 Método de Seleção Utilizado

Após obtermos os valores de expectativa dos indivíduos, utilizamos estes no processo efetivo da escolha dos indivíduos geradores.

O Giro de Roleta foi o método de seleção adotado para sorteio dos indivíduos, com base nos valores de expectativa dos indivíduos, ao invés de adotarmos os valores de aptidão. O método Giro de Roleta foi escolhido por se inspirar fortemente nos princípios da evolução natural.

O método de seleção por Giro de Roleta é apresentado na seção 4.4.1.3.1 "Métodos de Seleção".

5.3 Os Operadores Genéticos Adotados

Nessa seção, serão discutidos os operadores crossover e mutação adotados ao problema.

5.3.1 Mutação

O operador de mutação é relativamente simples: Uma máquina é randomicamente escolhida e dois elementos da permutação da máquina também são randomicamente escolhidos e trocados. (Ansi,1997).

Aplicamos o operador mutação a uma taxa de 5% de probabilidade de ocorrência sobre os indivíduos descendentes, mediante um sorteio quando gerado o descendente.

5.3.2 Clonagem

O operador clonagem, ou cópia do cromossomo para geração seguinte, é realizado quando uma operação de crossover não ocorre sobre os cromossomos pais selecionados. Então os pais são copiados para a geração seguinte fazendo parte da população sobrevivente.

A sua taxa de aplicação corresponde a (100% - taxa crossover).

5.3.3 Crossover

O operador crossover é um pouco mais complicado.

Uma vez que o agendamento de um n/m JSP é representado por m permutações, os operadores crossover de reordenação como PMX, OX, e CX podem ser usados com uma pequena modificação.

Em primeiro lugar, essas operações crossover de reordenação devem ser executadas em permutações de mesmo tamanho. Entretanto, cada máquina normalmente tem o número diferente de operações, então a operação crossover é aplicada somente na mesma máquina. (Ansi,1997).

Em segundo lugar, esses operadores de reordenação têm uma coisa em comum, a qual é a escolha de dois pontos de crossover e a preservação da informação de ordenação. Pelo fato das máquinas rodarem em paralelo é razoável fazer com que os pontos de crossover de todas as máquinas sejam relacionados. Isto é feito pela escolha de operações como pontos de crossover de cada máquina em aproximadamente os mesmos tempos de começo e fim de execução dessas operações. Assim a informação ordenada entre esse período de tempo pode ser preservada após o crossover. (Ansi,1997).

O exemplo seguinte ilustra a operação de crossover discutida acima. Considere o problema 3/3 JSP (figura 3.1) na sessão 3.1 “Definições e Premissas” e os dois cromossomos selecionadas para o crossover.

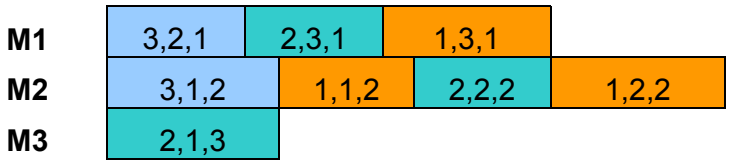


Tabela 5.13 – Cromossomo A

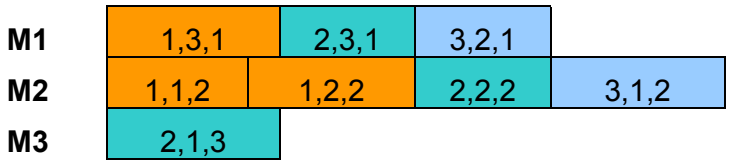


Tabela 5.14 – Cromossomo B

Suponha que a permutação da máquina 1 (M1), cromossomo A, a posição 1 é randomicamente selecionada como dois pontos de crossover. O tempo de início da operação 3,2,1, que é a posição 1, e o tempo de término da operação 3,2,1, que também é a posição 1, pode ser encontrado a partir do agendamento. (Ansi,1997). Veja o agendamento do Cromossomo A na tabela 5.8 "Indivíduo Aleatório" seção 5.1.2.3.1.2 "Exemplo de Cromossomo Aleatório".

Para a permutação da máquina 2 (M2), cromossomo A, duas operações correspondentes podem ser encontradas para que o tempo de início da primeira operação seja aproximadamente igual ao tempo de início da operação 3,2,1, e o tempo de término da segunda operação seja aproximadamente igual ao da operação 3,2,1. Essas duas posições serão utilizadas como pontos de crossover na máquina 2 (M2). Similarmente, os pontos de crossover na máquina 3 (M3) poderão ser encontrados. (Ansi,1997).

Suponha que os pontos de crossover encontrados sejam marcados " | " onde cada máquina Mn possui seus pontos Pn como se segue.

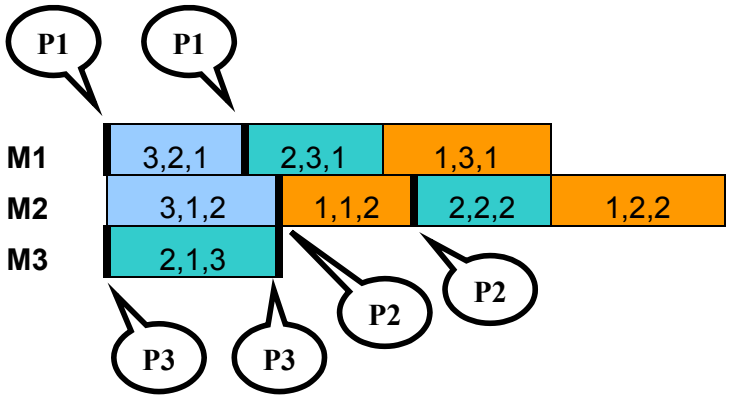


Tabela 5.15 – Pontos de Crossover do Cromossomo A

Para esses pontos de crossover, a operação 3,2,1 na máquina 1, a operação 1,1,2 na máquina 2, e a operação 2,1,3 na máquina 3 tem aproximadamente o mesmo tempo de início. A operação 3,2,1 na máquina 1, a operação 1,1,2 na máquina 2, e a operação 2,1,3 na máquina 3 tem aproximadamente o mesmo tempo de término. (Ansi,1997).

Após serem decididos os pontos de crossover, as operações crossover de reordenação são executadas em cada permutação correspondente as máquinas de cada cromossomo pai para formar dois novos conjuntos de permutações, que, depois de passarem pelo processo de agendamento, gerarão o descendente A' e o descendente B'.

5.3.3.1 O Operador Order Crossover - OX

O operador crossover utilizado para cruzamento foi o operador OX – *Order Crossover* por preservar a ordem relativa das operações (ordem relativa das informações codificadas – genoma) herdadas dos cromossomos pais durante o crossover e por não permitir a geração de indivíduos inválidos para o problema Job-Shop.

O operador OX realiza o crossover da seguinte forma :

Após os cortes (pontos de crossover) realizados em cada máquina no cromossomo A, o descendente direto de A, o cromossomo A', herda a seqüência de operações entre os cortes do pai A. Então a seqüência de operações do segundo pai, o cromossomo B, são preenchidas no descendente A' partindo do segundo corte, preservando-se a ordem das operações. Repete-se esse processo para o segundo descendente (descendente B').(Neto,1997).

5.3.3.1.1 Um Exemplo de Crossover OX

Iremos mostrar uma operação crossover apenas para máquina 2 (M2) do cromossomo A para gerar a permutação correspondente a maquina 2 (M2) do descendente A'. O processo é análogo para as outras máquinas do cromossomo A.

M2	3,1,2	1,1,2	2,2,2	1,2,2

Tabela 5.16 – Pontos de Crossover na máquina M2 do Cromossomo A

M2	1,1,2	1,2,2	2,2,2	3,1,2

Tabela 5.17 – Máquina M2 do Cromossomo B

M2		1,1,2		

Tabela 5.18 – Composição inicial do Descendente A' - máquina M2

O descendente A' é composto inicialmente pela operação 1,1,2 herdada do cromossomo A na mesma posição em que se encontra. As posições seguintes deverão ser preenchidas da seguinte forma.

Crie uma lista de operações, seguindo a ordem em que elas se encontram no cromossomo B, partindo da terceira posição. Partimos da terceira posição porque a última operação no cromossomo A, da seqüência entre os pontos de crossover herdados por A', ocupa a segunda posição.

Assim teremos a seguinte lista ordenada de todas as operações de B : { 2,2,2; 3,1,2; 1,1,2; 1,2,2 }. Exclua da lista, mantendo a ordem, as operações que foram herdadas pelo descendente A'. A lista de operações de B ficará assim : { 2,2,2; 3,1,2; 1,2,2 }. Nesse caso, a única operação herdada foi a operação 1,1,2 excluída da lista. Em seguida, essa lista, seguindo a ordem das operações, deverá ser preenchida em A', a partir da última operação herdada, até o preenchimento completo.

A permutação correspondente a máquina 2 (M2) do descendente A' ficará assim :

M2	1,2,2	1,1,2	2,2,2	3,1,2

Tabela 5.19 – Composição final do Descendente A' - máquina M2

Após aplicar o operador genético OX nos pontos de crossover em todas as permutações do cromossomo A, teremos as seguintes permutações das máquinas :

M1	3,2,1	2,3,1	1,3,1	
M2	1,2,2	1,1,2	2,2,2	3,1,2
M3	2,1,3			

Agora, só nos resta agendar as operações de cada permutação obedecendo às ordens de precedência, ou seja, a próxima operação da permutação a ser agendada será aquela que, na ordem em que se encontra na permutação, obedecer as restrições de precedência. A ordem, em cada permutação, que define qual a próxima operação a ser agendada parte da primeira operação agendada da permutação.

Veja a seguir, o agendamento das operações criando o cromossomo descendente A'. Cada iteração corresponde a uma operação agendada.

- A <= B : incluir em A o conteúdo de B.

- A = B : o conteúdo de A é igual ao conteúdo de B.

- t : tempo real (tempo corrente). Representação de medida u : unidades de tempo.

- o : operação

- Mn : máquina n do problema 3/3 JSP.

- Conj. A : um conjunto qualquer.

- Retirar (a,Conj. A) : retirar operação a do conjunto A.

- Crom(m) : máquina m do cromossomo.

- TProc(o) : tempo de processamento da operação o.

- TAg(o) : tempo de agendamento de uma operação o. Acontece sempre no tempo corrente t.

- ProxTDisp(Mn) : tempo em que a máquina n se encontrará disponível para agendar uma operação.

    - ProxTDisp(Mn) = t + TProc(o) : após agendamento de uma operação na máquina n no tempo t.
    - ProxTDisp(Mn) = ProxTDisp(Mn') : quando não há agendamento de uma operação em Mn no tempo t. Verifica-se antes a possibilidade de agendamento em todas as outras máquinas n' no tempo t (se estiverem disponíveis nesse tempo). Se houver possibilidade de agendamento, agende-as. Em seguida, a maquina n' que possuir o menor ProxTDisp será atribuída esse valor ao ProxTDisp da máquina n. Mn' é diferente de Mn.

- Conj. Ag : conjunto de operações diponíveis para agendamento seguindo as restrições de precedência. Durante o agendamento, é escolhida para agendar na máquina n a operação do Conj Ag que obedecer a ordem em que se encontra na permutação correspondente a máquina n, partindo da primeira operação agendada da permutação.

- Op1Ag(Mn) : primeira operação agendada da permutação correspondente a máquina n

- I : intervalo de ociosidade.

Condições Iniciais:

t = 0u.

Conj. Ag = {}

Iteração 1:

1- Conj. Ag <= {1,1,2; 2,1,3; 3,1,2} : primeiras operações de um job.

2- Verificar operação do Conj Ag na permutação M1.

- Nenhuma operação encontrada. Logo M1 é ociosa.

3- Verificar operação do Conj. Ag na permutação M2. Operação encontrada = 1,1,2.

- Crom(2) <= {1,1,2}
- Op1Ag(2) = {1,1,2}

4- Retirar ({1,1,2}, Conj. Ag) : Conj Ag = {2,1,3; 3,1,2}

5- TAg({1,1,2}) = 0u. ProxTDisp(M2) = 0 + (TProc({1,1,2}) = 5) = 5u.

Iteração 2:

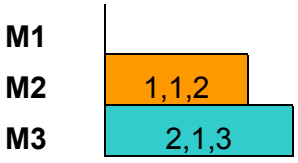
1- Conj Ag = {2,1,3; 3,1,2}

2- Verificar operação do Conj. Ag na permutação M3. Operação encontrada = 2,1,3

- Crom(3) <= {2,1,3}
- Op1Ag(3) = {2,1,3}

3- Retirar ({2,1,3}, Conj. Ag) : Conj Ag = {3,1,2}

4- TAg({2,1,3}) = 0u. ProxTDisp(M3) = 0 + (TProc({2,1,3}) = 7) = 7u.



Iteração 3:

1- Conj Ag = {3,1,2}

2- Verificar ociosidade em M1. ProxTDisp(M1) = 5u. (menor ProxTDisp)

3- t = 5u. (menor ProxTDisp).

4- Atualizar Conj. Ag. : Conj Ag <= {1,2,2} : Conj Ag = {3,1,2; 1,2,2}

5- Verificar operação do Conj. Ag na permutação M1.

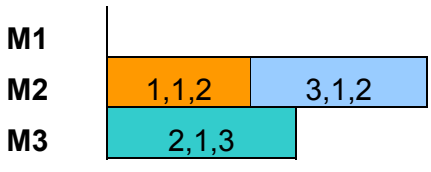
- Nenhuma operação encontrada. Logo M1 é ociosa.

6- Verificar operação do Conj. Ag na permutação M2 a partir de (Op1Ag(2) = {1,1,2}).

- Operação encontrada = 3,1,2. : Crom(2) <= {3,1,2}

7- Retirar ({3,1,2}, Conj. Ag) : Conj Ag = {1,2,2}

8- TAg({3,1,2}) = 5u. ProxTDisp(M2) = 5 + (TProc({3,1,2}) = 7) = 12u.



Iteração 4:

1- Conj Ag = {1,2,2}

2- Verificar ociosidade em M1. ProxTDisp(M1) = 7u. (menor ProxTDisp)

3- t = 7u. (menor ProxTDisp)

4- Atualizar Conj. Ag. : Conj Ag <= {2,2,2} : Conj Ag = {1,2,2; 2,2,2}

5- Verificar operação do Conj Ag na permutação M1.

- Nenhuma operação encontrada. Logo M1 é ociosa.

6- Verificar ociosidade em M1. ProxTDisp(M1) = 12u. (menor ProxTDisp)

7- t = 12u. (menor ProxTDisp)

8- Atualizar Conj. Ag. : Conj Ag <= {3,2,1} : Conj Ag = {1,2,2; 2,2,2; 3,2,1}

9- Verificar operação do Conj Ag na permutação M1. Operação encontrada = 3,2,1

- Crom(1) <= {3,2,1}
- Op1Ag(1) = {3,2,1}

10- Retirar ({3,2,1}, Conj. Ag) : Conj Ag = {1,2,2; 2,2,2}

11- TAg({3,2,1}) = 12u. ProxTDisp(M1) = 12 + (TProc({3,2,1}) = 5) = 17u.

Iteração 5:

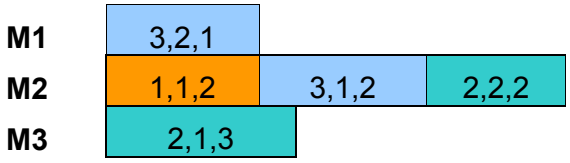
1- Conj Ag = {1,2,2; 2,2,2}

2- Verificar operação do Conj Ag na permutação M2 a partir de (Op1Ag(2) = {1,1,2}).

- Operação encontrada = 2,2,2 : Crom(2) <= {2,2,2}

3- Retirar ({2,2,2}, Conj. Ag) : Conj Ag = {1,2,2}

4- TAg({2,2,2}) = 12u. ProxTDisp(M2) = 12 + (TProc({2,2,2}) = 5) = 17u.



Iteração 6:

1- Conj Ag = {1,2,2}

2- t = 17u. (menor ProxTDisp)

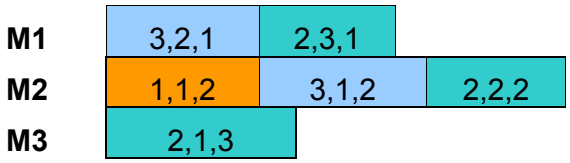
3- Atualizar Conj. Ag. : Conj Ag <= {2,3,1} : Conj Ag = {1,2,2; 2,3,1}

4- Verificar operação do Conj Ag na permutação M1 a partir de (Op1Ag(1) = {3,2,1}).

- Operação encontrada = 2,3,1 : Crom(1) <= {2,3,1}

5- Retirar ({2,3,1}, Conj. Ag) : Conj Ag = {1,2,2}

6- TAg({2,3,1}) = 17u. ProxTDisp(M1) = 17 + (TProc({2,3,1}) = 5) = 22u.



Iteração 7:

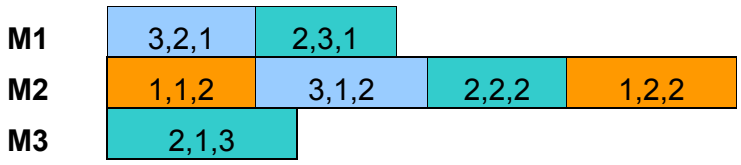
1- Conj Ag = {1,2,2}

2- Verificar operação do Conj Ag na permutação M2 a partir de (Op1Ag(2) = {1,1,2}).

- Operação encontrada = 1,2,2 : Crom(2) <= {1,2,2}

3- Retirar ({1,2,2}, Conj. Ag) : Conj Ag = {}

4- TAg({1,2,2}) = 17u. ProxTDisp(M2) = 17 + (TProc({1,2,2}) = 7) = 24u.



Iteração 8:

1- Conj Ag = {}

2- t = 22u. (menor ProxTDisp)

3- Atualizar Conj. Ag. : Nenhuma operação adicionada.

4- Verificar operação do Conj. Ag na permutação M1 a partir de (Op1Ag(1) = {3,2,1}).

- Nenhuma operação no Conj. Ag. Logo M1 é ociosa.

5- Verificar ociosidade em M1. ProxTDisp(M1) = 24u. (menor ProxTDisp)

6- t = 24u. (menor ProxTDisp)

7- Atualizar Conj. Ag. : Conj Ag <= {1,3,1} : Conj Ag = {1,3,1}

8- Verificar operação do Conj Ag na permutação M1 a partir de (Op1Ag(1) = {3,2,1}).

- Operação encontrada = 1,3,1 : Crom(1) <= {1,3,1}

9- Retirar ({1,3,1}, Conj. Ag) : Conj Ag = {}

10- TAg({1,3,1}) = 24u. ProxTDisp(M1) = 24 + (TProc({1,3,1}) = 7) = 31u.

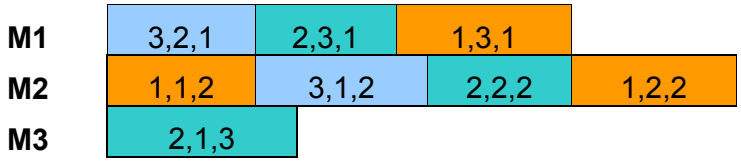


Tabela 5.20 – Descendente A'

Após o agendamento do descendente A' (Tabela 5.20), temos o indivíduo A' (Tabela 5.21) gerado com tempo de finalização igual a 31 unidades de tempo.

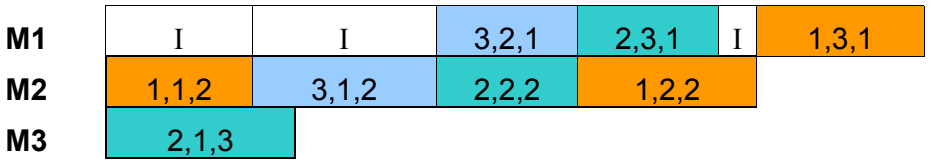


Tabela 5.21 – Indivíduo A'

O operador OX prioriza a ordem das operações codificadas e não suas posições no cromossomo. Sabe-se que a posição das operações codificadas no cromossomo não é importante para o problema Job-Shop e sim a ordem de visitas das operações, ou seja, para o problema Job-Shop é importante saber quem é o vizinho de uma dada operação *o* a ser agendado.

6 Conclusões

As pesquisas aplicadas aos Algoritmos Genéticos ainda são recentes. São considerados um instrumento novo de pesquisa. Podem ser utilizados em diversas áreas de atividade humana.

Apesar dos muitos estudos e experimentações feitos, estes ainda são insuficientes para estabelecer conclusões precisas e rigorosas.

Um bom resultado a ser obtido pelos Algoritmos Genéticos, dependerá do rendimento de detalhes como o método de codificação das soluções candidatas, os tipos de operadores genéticos utilizados, os ajustes de parâmetros e outros critérios particulares. Um Algoritmo Genético na sua forma básica pode ser insuficiente para um determinado tipo de problema, por exemplo: a codificação *bit-string* pode ser inadequada; a seleção através da Proporcionalidade do *Fitness* pode conduzir à convergência prematura; os operadores de cruzamento e de mutação podem tornar inviáveis as soluções descendentes, bem como podem destruir bons esquemas dos cromossomos atuais e um ajuste inadequado dos parâmetros pode diminuir a performance global.

Por este motivo, as decisões sobre cada passo da elaboração do algoritmo devem ser analisadas cuidadosamente, visto que existem muitas possibilidades de implementação e o relacionamento entre os objetos de decisão geralmente não estão suficientemente claros e simples.

Tem-se observado grandes esforços nas pesquisas que adotam metodologias híbridas em busca de melhores soluções para o problema a curto prazo, como por exemplo Algoritmos Genéticos com o método heurístico Simulated Annealing, Algoritmos Genéticos com o método heurístico Busca Tabu, entre outros.

6.1 Dificuldades Encontradas

Não foi encontradado nenhum material que oferecesse de forma mais clara o mecanismo de codificação de soluções para o problema tratado e exemplos de aplicações crossover para o problema.

Também um tempo consideravel foi dedicado de forma a entender uma função de avaliação que pudesse avaliar os individuos da população, que não prejudicasse a seleção dos indivíduos diante das grandes e pequenas variações nas qualidades dos indivíduos.

6.2 Sugestões Futuras

Como sugestões futuras para o projeto é sugerido um estudo aprofundado dos parâmetros a serem utilizados pelo Algoritmo Genético híbrido implementado como : Tamanho da População e o Controle Flexível da Taxa de Gulosidade.

- **Controle Flexível da Taxa de Gulosidade** : baseado nas soluções obtidas em gerações anteriores e no tamanho do Problema Job-Shop, a taxa de gulosidade

poderá sofrer alterações, de forma automatizada, a fim de evitar convergências prematuras na obtenção de soluções para o problema.

- **Tamanho Máximo da População** : também denominada biologicamente *Capacidade de Suporte*. Como determinar o tamanho máximo da população durante as gerações de indivíduos? Quais fatores deveremos levar em consideração para o crescimento da população? Como adaptar os ideais naturais da Biologia referentes a Capacidade de Suporte ao Problema Job-Shop?. São questões a serem estudadas.

  Veja a seguir uma breve descrição a respeito de Capacidade de Suporte, segundo Piank (1982) e Ricklefs (1993).

  Capacidade de Suporte : ou capacidade máxima de suporte do ambiente, representa o número de indivíduos que o ambiente suporta.

  Em um ambiente finito, nenhuma população pode crescer exponencialmente durante muito tempo. O crescimento populacional diminuirá diante das difíceis condições ambientais ou a escassez de recurso para sua reprodução.

  Na prática, a capacidade de suporte pode ser ilustrada com o exemplo de pequenos roedores herbíveros em um ambiente limitado. Em um primeiro momento os indivíduos crescem livremente, mas, chegará um momento em que o espaço e os recursos alimentares não serão suficientes para todos resultando em um declínio populacional. Com a diminuição no número de indivíduos, o ambiente terá o tempo necessário para renovar a vegetação, o que permitiria um novo aumento populacional. Assim, a população em equilíbrio oscila constantemente em torno da capacidade máxima ambiental.

7 Referências Bibliográficas

ANSARI, N. e HOU, E. *Computational Intelligence for Optimization*. Kluwer Academic Publishers, 1997.

BERSON, A., STEPHEN, J. S. *Data Warehousing, Data Mining, and OLAP*. McGraw-Hill, 1997.

DARWIN, C. *The Origin of Species.* New York. Mentor Books, 1953.

DAVIS, L. *Handbook on Genetic Algorithms.* Van Nostrand Reinhold. NY, 1991.

FEO, T. A., RESENDE, M. G. C. *Greendy randomized adaptive search procedures.* Jornal of Global Optimization, 6:109-133, 1995.

GEYER, A. *Fuzzy Rule-Based Expert Systems and Genetic Machine Learning*. Heidelberg : Physica-Verlag, 1997.

GOLDBERG, D.E. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley, 1989.

HAX, A., CANDEA, D. *Production and Inventory Management.* Englewood Cliffs, N.J., Prentice-Hall, 1984.

HOLLAND, J. H. *Adaptation in Natural and Artificial Systems*. Cambridge. MIT Press, p. 211, 1993.

KOZA, J. R. *Genetic programming on programming of computers by means of natural selection*. MIT, 1992

MITCHELL, T. M. *Machine Learning.* McGraw-Hill, 1997.

MORTON, T.E., PENTICO, D.W. ***Heuristic Scheduling Systems.*** John Wiley & Sons. N.Y., 1993.

NETO, J. F. B. *Os Métodos Heurísticos com Base em Algoritmos Genéticos.* Relatório Técnico Maio/1997, UFRJ.

PIANK, E. R. *Ecologia Evolutiva.* Editora Omega. Barcelona, 1982, p. 108-115.

PACHECO, R.F. e SANTORO, M.C. *Proposta de Classificação Hierarquizada dos Modelos de Solução para o Problema de* ***Job Shop Scheduling.*** GESTÃO E PRODUÇÃO, Revista do Departamento de Engenharia de Produção, Universidade de São Carlos, Abril de 1999. p. 1-15.

PINEDO, M., CHAO, X. *Operations Scheduling With Applications in Manufacturing and Services.* Irwin McGrawn-Hill, 1999.

RESENDE, M. G. C. *A GRASP for Job Shop Scheduling.* AT&T Labs Research, May 1997.

RICKLEFS, R. E. *A Economia da Natureza.* 3ª edição.

Editora Guanabara Koogan. 1993, p. 215-229

RODAMMER, F. A., WHITE, K. P., *A Recent Survey of Production Scheduling*, IEEE Transactions on Systems, Man, and Cybernetics, v. 118/6, 1988, p. 841-851.

SOUZA, M. J. F. *Tópicos Especiais em Inteligência Artificial.* Apostila 2001/1. DECOM/ICEB/UFOP

TAILLARD, E. *Benchmarks for basic scheduling problems.* European Journal of Operational Research 64, 1993, 278-285

WALTER, C. *Planejamento e Controle da Produção - PCP.* Apostila de aula.- Mestrado em Engenharia de Produção - UFRGS, 1999

Anexo A. Glossário Genético

É importante ressaltar que este glossário é voltado para a área de algoritmos genéticos e por isso faz uso de um considerável grau de liberdade na definição de termos com relação a seus correlatos na biologia.

Adaptação – no campo da biologia, o nível de adaptação mede o quanto um indivíduo está apto para sobreviver no meio em que reside. Em sua analogia com a natureza, os Algoritmos Genéticos fazem uso do grau da adaptação para representar quão bem um determinado indivíduo (solução) responde ao problema proposto.

Convergência – o grau de convergência é característica das populações dos Algoritmos Genéticos, e diz respeito à diferença entre a média de adaptação de sua geração atual e suas anteriores. A ascensão deste índice indica que o processo de evolução está efetivamente promovendo a melhora da média de adaptação da população, e sua estabilização em torno de um mesmo valor por muitas gerações normalmente indica que a população se estacionou em um determinado valor médio de adaptação, caso em que a continuação do processo de evolução se torna improdutiva.

Convergência Prematura – a convergência prematura é um problema bastante recorrente na técnica de algoritmos genéticos: ela ocorre quando a diversidade de uma população cai de tal forma que o processo de reprodução gera a cada geração filhos muito semelhantes aos pais, o que causa a convergência da população com média de adaptação em muitos casos pouco adequada e retarda (ou até mesmo estanca por completo) a evolução.

Cromossomo – um cromossomo consiste de uma cadeia de genes. Visto que tradicionalmente são utilizados genomas com uma cadeia simples de genes, muitas vezes este termo é utilizado, com certa liberdade, como sinônimo para genoma.

Cruzamento – durante o processo de cruzamento os indivíduos previamente selecionados são cruzados com seus pares para gerar filhos.

Diversidade (ou biodiversidade) – característica de uma população de indivíduos, a diversidade diz respeito ao grau de semelhança entre eles.

Evolução – o processo de evolução é o responsável pela busca efetuada pelos Algoritmos Genéticos. A cada passo os indivíduos da população a ele submetida passam pelos processos de seleção, reprodução e mutação, criando assim uma nova geração a partir de sua anterior.

Função de aptidão – a função objetivo recebe como entrada um indivíduo e retorna o grau de adaptação que este apresenta.

Fenótipo – resultado do processo de decodificação do genótipo de um indivíduo. Em um Algoritmo Genético o fenótipo é a solução propriamente dita que este representa.

Gene – um gene serve como um *container*, um espaço para a alocação de um valor. As características dos indivíduos são definidas a partir dos valores contidos no conjunto de um ou mais genes que as codificam.

Genótipo – para um algoritmo genético o genótipo de um indivíduo é a informação codificada nos genes e manipulada durante o processo de evolução e o fenótipo, resultante do processo de decodificação, é a solução que o genótipo representa.

Genoma – um genoma é o conjunto de genes de um determinado indivíduo. Tradicionalmente genomas são formador por uma cadeia única de genes (um cromossomo),

mas existem representações alternativas como a de árvores (bastante utilizada em problemas de programação genética) de listas e de matrizes multidimensionais.

Geração – a cada passo do processo de evolução uma nova geração é criada a partir da população anterior, e esta última é atualizada. Para que o processo de evolução possa ser considerado eficiente é necessário que em cada nova geração tenda a ser melhor (mais adaptada) que suas anteriores.

Indivíduo – um indivíduo pertencente a um algoritmo genético representa uma possível solução para o problema a ser tratado.

Mutação – na natureza, falhas no processo de reprodução podem facilmente ocorrer, fazendo com que os filhos apresentem características que não seriam atingíveis através de uma combinação de características dos pais. A este fenômeno dá-se o nome de mutação, e nos algoritmos genéticos esta transformação ocorre sobre indivíduos provindos do processo de reprodução.

População – uma população consiste em um conjunto de indivíduos. Ela apresenta características não observáveis nos indivíduos, tais como grau de diversidade e de convergência.

Reprodução – o processo de reprodução consiste da alocação em pares dos indivíduos selecionados e do cruzamento entre estes. Ele é o responsável pela perpetuação de características ao longo do processo de evolução.

Scaling – é sabido que a presença de indivíduos com alto grau de aptidão e a baixa diversidade numa população são fatores que facilmente levam ao problema de convergência prematura. Vários métodos são utilizados para diminuir a probabilidade de que isto ocorra, entre eles os de *scaling*. Estes últimos operam aplicando uma transformação ao grau de adaptação de cada indivíduo da população, atenuando as discrepâncias nestes valores, e valorizando indivíduos diferentes da média.

Seleção – nesta etapa onde indivíduos são escolhidos para a reprodução de acordo com seu grau de adaptação. Para isto cada indivíduo é avaliado e os mais aptos da população possuem maior probabilidade de serem selecionados.

Anexo B. Código Fonte dos Programas

Os algoritmos foram implementados na linguagem C++.

Exibimos a seguir o código-fonte que implementa o algoritmo genético híbrido.

```
/*
   CLASSE PARA EXIBIR MENSAGEM DE ERRO E ABORTAR O PROGRAMA.
   ARQUIVO : ERRO.H
*/

// ------------------------------------------------------------
#define ERROLIB
#define ABORTAR "O PROGRAMA SERA ABORTADO"
// ------------------------------------------------------------

// bibliotecas do sistema:
// ------------------------------------------------------------
#include <conio.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
// ------------------------------------------------------------

class Erro
{
   private:
   typedef char* Mensagem;

   void printMensg ( Mensagem mens);
   void procurarIndArg ( Mensagem erro,Mensagem &erro0_Ind,Mensagem
&erroInd_Fim );
   void desalocarMemo ( Mensagem &erro0_Ind,Mensagem &erroInd_Fim);

   public:
   Erro (void) { }
   void printErro  ( Mensagem erro);
   void printErro  ( Mensagem erro, int arg);
   void printErro  ( Mensagem erro, char arg);
   void printErro  ( Mensagem erro, char* arg);
}; // classe Erro
void Erro:: desalocarMemo ( Mensagem &erro0_Ind, Mensagem &erroInd_Fim)
{
  delete erro0_Ind;    // desalocar memoria
  delete erroInd_Fim;
  return;
}

void Erro:: printMensg ( Mensagem mens)
{
  fprintf(stderr,"\n%s\n",mens);
  return;
}

void Erro:: procurarIndArg ( Mensagem erro, Mensagem &erro0_Ind,
                             Mensagem &erroInd_Fim )
{
   int i,j,k,
       TAMMENSGERRO,
```

```
        TAMMENSG_0_IND,
        TAMMENSG_IND_FIM;

   TAMMENSGERRO=strlen(erro);

   i=0;
   while (erro[i] &&
        erro[i]!='%')
   {
     i++;
   }
   TAMMENSG_0_IND=i+1;
   if (!(erro[i]))
   {
     TAMMENSG_IND_FIM=1;
   }
   else
   {
     TAMMENSG_IND_FIM=(TAMMENSGERRO-1)-i+1;
   }
   erro0_Ind= new char[TAMMENSG_0_IND];
   erroInd_Fim= new char[TAMMENSG_IND_FIM];

   for (j=0; j<i; j++)
   erro0_Ind[j]=erro[j];
   erro0_Ind[i]='\0';

   for (k=0,j=i+1; j<TAMMENSGERRO; j++,k++)
   erroInd_Fim[k]=erro[j];
   erroInd_Fim[k]='\0';
   return;
} // encontra o indice do Argumento

void Erro:: printErro (Mensagem erro)
{
  fprintf(stderr,"\n%s\n",erro);
  printMensg(ABORTAR);
  getch(); exit(1);
  return;
} // printErro (<Mensagem>)

void Erro:: printErro (Mensagem erro, char* arg)
{
  char *erro0_Ind, *erroInd_Fim;

  procurarIndArg(erro,erro0_Ind,erroInd_Fim);
  fprintf(stderr,"\n%s%s%s\n",erro0_Ind,arg,erroInd_Fim);
  desalocarMemo(erro0_Ind,erroInd_Fim);
  printMensg(ABORTAR);
  getch(); exit(1);
  return;
} // printErro (<Mensagem><char*>)

void Erro:: printErro (Mensagem erro, char arg)
{
  char *erro0_Ind, *erroInd_Fim;

  procurarIndArg(erro,erro0_Ind,erroInd_Fim);
  fprintf(stderr,"\n%s%c%s\n",erro0_Ind,arg,erroInd_Fim);
  desalocarMemo(erro0_Ind,erroInd_Fim);
  printMensg(ABORTAR);
  getch();
  exit(1);
```

```
  return;
} // printErro (<Mensagem><char>)

void Erro:: printErro (Mensagem erro, int arg)
{
  char *erro0_Ind, *erroInd_Fim;

  procurarIndArg(erro,erro0_Ind,erroInd_Fim);
  fprintf(stderr,"\n%s%d%s\n",erro0_Ind,arg,erroInd_Fim);
  desalocarMemo(erro0_Ind,erroInd_Fim);
  printMensg(ABORTAR);
  getch();
  exit(1);
  return;
} // printErro (<Mensagem><int>)


/*
   DEFINICAO DE UMA CLASSE TIPICA PARA UM OBJETO DO TIPO LISTA.
   ARQUIVO : LISTA.H

*/

// ------------------------------------------------------------
#define LISTALIB
// ------------------------------------------------------------

// bibliotecas especiais:
// ------------------------------------------------------------
#ifndef ERROLIB
#include "c:\temp\wendrer\programa\heads\erro.h"
#endif
// ------------------------------------------------------------

template <class TItem>
class Lista
{
  private:
  struct SCelula {
    TItem item;
    struct SCelula *prox;
  };
  SCelula *primeiro, *ultimo;
  int quantos;
  Erro erro;

  public:
  Lista        (void);
  ~Lista       (void);
  short vazia  (void);
  int tam      (void);
  void inserir (TItem item);
  void retirar (TItem &item, int indice=0);  // primeiro item:indice = 0
  void obter   (TItem &item, int indice=0);  // primeiro item:indice = 0
};

template <class TItem>
Lista<TItem> :: Lista (void)
{
  if((primeiro=ultimo=new (SCelula))!=NULL)
  {
   ultimo->prox = NULL;
   quantos=0;
```

```
  }
  else erro.printErro("ERRO: Memoria nao alocada");
}

template <class TItem>
Lista<TItem> :: ~Lista (void)
{
  SCelula *temp =primeiro;
  while (primeiro)
  {
     primeiro=primeiro->prox;
     temp->prox=NULL;
     delete temp;
     temp=primeiro;
  }
  ultimo=primeiro;
  quantos=0;
}

template <class TItem>
short Lista<TItem> :: vazia (void)
{ return (primeiro==ultimo); }

template <class TItem>
int Lista<TItem> :: tam (void)
{ return (quantos); }

template <class TItem>
void Lista<TItem> :: inserir ( TItem item)
{
  if ((ultimo->prox=new (SCelula))!=NULL)
  {
    ultimo=ultimo->prox;
    ultimo->item = item;
    ultimo->prox=NULL;
    quantos++;
  }
  else erro.printErro("ERRO: Memoria nao alocada");
  return;
}

template <class TItem>
void Lista<TItem> :: retirar ( TItem &item, int indice )
{
  if (!(vazia()))
  {
    SCelula *aux=primeiro, *temp;

    for (int i=0; i<indice; i++)
    aux=aux->prox;

    if (indice==tam()-1)
    {
      ultimo=aux;
    }
    temp=aux->prox;
    item=temp->item;
    aux->prox=temp->prox;
    temp->prox=NULL;
    delete temp;
    quantos--;
  }
  return;
```

```
}

template <class TItem>
void Lista<TItem> :: obter ( TItem &item, int indice )
{
  if (!(vazia()))
  {
    SCelula *aux=primeiro;

    for (int i=0; i<indice; i++)
    aux=aux->prox;

    item=aux->prox->item;
  }
  return;
}


/*
   CLASSE PARA CRIAR UMA INSTÂNCIA JOB-SHOP PARA O PROGRAMA.
   ARQUIVO : PROB.H
*/

// ------------------------------------------------------------------
#define PROBLIB
// ------------------------------------------------------------------

// bibliotecas C/C++:
// ------------------------------------------------------------------
#include <stdio.h>
#include <conio.h>
#include <string.h>
#include <fstream.h>
#include <math.h>

// bibliotecas especiais:
// ------------------------------------------------------------------
#ifndef ERROLIB
#include "c:\temp\wendrer\programa\heads\erro.h"
#endif
// ------------------------------------------------------------------

struct SProblema {
    long seedTime;         /* semente para tempo de processamento */
    long seedMaq;          /* semente para maquinas */
    int numJobs;           /* numero de jobs */
    int numMaq;            /* numero de maquinas */
 };

 class Problema {
 private:
 typedef char* Arquivo;
 struct SOperacao { int tempo; int maq; };

 int *numOpsJob,           // nr de operacoes de cada job.
     *numOpsMaq;           // nr de operacoes atribuidas a cada maquina.

 Erro erro;                // tratamento de erro.

 SOperacao **op;           // operacoes.
 SProblema prob;           // dados do problema.

 void setValoresOps (void);
```

```
 void swap (int &A,int &B);
 int  obterRandNrUnif (long *seed,int baixo,int alto);

 public:
 Problema  (void);
 Problema  (Arquivo obterDadosProb);
 Problema  (long seedTime,long seedMaq,int numJobs,int numMaq);
 ~Problema (void);
 int obterNrJobs   (void);
 int obterNrMaqs   (void);
 int obterNrOpsJob (int nJob);
 int obterNrOpsMaq (int nMaq);
 int obterTempoOp  (int nJob, int nOp);
 int obterMaqOp     (int nJob, int nOp);
 void gerarProblema (long seedTime, long seedMaq,int numJobs, int numMaq );
 void gerarProblema (SProblema p);
 SProblema obterDadosProblema (Arquivo obterDadosProb);
 void gravarProblema (Arquivo printProblema);
}; // class Problema

Problema :: Problema (void)
{
    gerarProblema (0,0,0,0);
} // construtor

Problema :: Problema (long seedTime,long seedMaq,int numJobs,int numMaq )
{
    gerarProblema (seedTime,seedMaq,numJobs,numMaq);
} // construtor

Problema :: Problema (Arquivo obterDadosProb)
{
    gerarProblema (obterDadosProblema(obterDadosProb));
} // construtor

Problema :: ~Problema (void)
{
  for(int nJob=0; nJob<prob.numJobs; nJob++)
   delete [](op[nJob]);
  delete []op;
  delete []numOpsJob;
  delete []numOpsMaq;
}  // destrutor

int Problema :: obterNrJobs (void)
{  return (prob.numJobs); }

int Problema :: obterNrMaqs (void)
{  return (prob.numMaq); }

int Problema :: obterNrOpsJob (int nJob)
{
   if (prob.numJobs!=0)
     if (nJob >=0 && nJob < prob.numJobs)
     return (numOpsJob[nJob]);
   return (-1);
}

int Problema :: obterNrOpsMaq (int nMaq)
{
   if (prob.numMaq!=0)
     if (nMaq >=0 && nMaq < prob.numMaq)
     return (numOpsMaq[nMaq]);
```

```
   return (-1);
}

int Problema :: obterTempoOp (int nJob,int nOp)
{
  if (prob.numJobs!=0)
     if ((nJob >=0 && nJob < prob.numJobs) &&
       (nOp >=0 && nOp < numOpsJob[nJob]))
     return (op[nJob][nOp].tempo);
  return (-1);
}

int Problema :: obterMaqOp (int nJob, int nOp)
{
  if (prob.numJobs!=0)
     if ((nJob >=0 && nJob < prob.numJobs) &&
         (nOp >=0 && nOp < numOpsJob[nJob]))
     return (op[nJob][nOp].maq);
  return (-1);
}

/*
  ----------------------------------
   Determinar o Problema e atribuir
   valores ramdomicos  as operacoes
  ----------------------------------
*/
void Problema :: gerarProblema (SProblema p)
{
    gerarProblema(p.seedTime,p.seedMaq,p.numJobs,p.numMaq);
    return;
} // gerarProblema

void Problema :: gerarProblema ( long seedTime, long seedMaq,int
numJobs,int numMaq )
{
    prob.seedTime = seedTime;
    prob.seedMaq  = seedMaq;
    prob.numJobs  = numJobs;
    prob.numMaq   = numMaq;

    if (seedTime >0 && seedMaq >0 &&
      numJobs >0 && numMaq >0 )
    {
      setValoresOps();
    }
    else
    {
        op           =(SOperacao**)NULL;
        prob.numJobs =prob.numMaq=0;
        numOpsJob    =(int*)NULL;
        numOpsMaq    =(int*)NULL;
    }
    return;
} // gerarProblema

/*
  ---------------------------------------
   Leitura do Problema no arquivo de dados
  ---------------------------------------
*/
SProblema Problema :: obterDadosProblema (Arquivo obterDadosProb)
{
```

```
  SProblema auxProb;
  const int MAXTAMBUFF=100;
  char *campo,
       *dado,
       buff[MAXTAMBUFF];

  ifstream fin (obterDadosProb);
  if (fin)
  while (fin.getline(buff,MAXTAMBUFF))
  {
    campo=strtok(buff," ");
    if (campo)
    {
     if (campo[0]=='!')
     continue;
     else
     {
      dado=strtok(NULL," ");
      if (dado)
      {
       if (dado[0]=='=')
       {
      if (strlen(dado)==1)
      dado=strtok(NULL," ");
      else dado++;
       }
       else campo=strtok(campo,"=");
      }
      else
      {
       campo=strtok(campo,"=");
       dado=strtok(NULL," ");
      }
      if (campo && dado)
      {
       if (stricmp(campo,"Time_seed")==0)
       {
      if(!(auxProb.seedTime=atol(dado)))
      erro.printErro("ERRO: valor <Time_seed> invalido.");
       }
       else
      if (stricmp(campo,"Machine_seed")==0)
      {
       if(!(auxProb.seedMaq=atol(dado)))
       erro.printErro("ERRO: Valor <Machine_seed> invalido.");
      }
      else
       if (stricmp(campo,"Jobs")==0)
       {
        if(!(auxProb.numJobs=atol(dado)))
        erro.printErro("ERRO: Valor <Jobs> invalido.");
       }
       else
        if (stricmp(campo,"Machines")==0)
        {
         if(!(auxProb.numMaq=atol(dado)))
         erro.printErro("ERRO: Valor <Machines> invalido.");
        }
        else erro.printErro("ERRO: Campo no arquivo %
irreconhecivel.",obterDadosProb);
      } // if
      else erro.printErro("ERRO: CAMPO ou DADO do arquivo %
irreconhecivel.",obterDadosProb);
```

```
     } // else
    } // if
  } // while
  else erro.printErro("ERRO: Nao foi possivel abrir o arquivo
",obterDadosProb);
  return (auxProb);
} // obterDadosProblema

/* ---------------------------------------------
   Gerar um numero randomico uniformemente entre
   os valores dos parametros baixo e alto
   ---------------------------------------------
*/
int Problema :: obterRandNrUnif (long *seed,int baixo,int alto)
{
  const long m = 2147483647,
             a = 16807,
             b = 127773,
             c = 2836;
  double  valor_0_1;
  long k;

  k= *seed/b;
  *seed= a*(*seed%b)-k*c;
  if (*seed < 0)
  *seed= *seed+m;
  valor_0_1= *seed/(double)m;

  return (baixo+floor(valor_0_1*(alto-baixo+1)));
} // gerarRandNumUnif

/*  ----------------------------
    Atribuir valores as operacoes
    ----------------------------
*/
void Problema :: swap ( int &A, int &B )
{
  if (A!=B)
  {
   int temp;

   temp=A;  A=B; B=temp;
  }
  return;
} // swap

void Problema :: setValoresOps (void)
{
  int nOp,     // nr da operacao.
      nJob,    // nr do job.
      nMaq;    // nr da maquina.

  long seedMaq,
       seedTime;


  seedMaq=prob.seedMaq;
  seedTime=prob.seedTime;

  numOpsJob=new int[prob.numJobs];
  numOpsMaq=new int[prob.numMaq];

  op=new SOperacao*[prob.numJobs];
```

```
  for(nMaq=0; nMaq<prob.numMaq; nMaq++)
  numOpsMaq[nMaq]=0;
  for(nJob=0; nJob<prob.numJobs; nJob++)
  {
    numOpsJob[nJob]=prob.numMaq;
    op[nJob]=new SOperacao[numOpsJob[nJob]];

    for(nOp=0; nOp<numOpsJob[nJob]; nOp++)
    {
      /*
       determina o tempo de processamento (max=99)
       e a maquina para todas operacoes do job=nJob.
      */
      op[nJob][nOp].tempo=obterRandNrUnif(&seedTime,1,99);
      op[nJob][nOp].maq=nOp;        // maquina = nr da operacao.

      numOpsMaq[op[nJob][nOp].maq]++;
    }
    for(nOp=0; nOp<numOpsJob[nJob]; nOp++) // troca aleatoria das maquinas.
    swap(op[nJob][nOp].maq,
       op[nJob][obterRandNrUnif(&seedMaq,nOp,numOpsJob[nJob]-1)].maq);
  }
  return;
} // setValoresOps

/*  ----------------------------
    Gravar no arquivo o problema
    ----------------------------
*/
void Problema :: gravarProblema (Arquivo printProblema)
{
  FILE *fout = NULL;

  if(!(fout=fopen(printProblema,"a+t")))
  erro.printErro("\nNao foi possivel gravar o arquivo %\n",printProblema);

  fprintf(fout,"\n\n PROBLEMA %d/%d JSP:\n\n",prob.numJobs,prob.numMaq);

  if (op)
  for (int nJob=0; nJob<prob.numJobs; nJob++)
  {
    fprintf(fout," J%-4d- ",nJob+1);
    for (int nOp=0; nOp<numOpsJob[nJob]; nOp++)
    fprintf(fout,"%3d-%#02d ",op[nJob][nOp].maq+1,op[nJob][nOp].tempo);

    fprintf(fout,"\n");
  }
  else fprintf(fout,"%d",0);
  fclose(fout);
  return;
} // gravarProblema
```

```
/*
   CLASSE PARA CRIAR E MANIPULAR CROMOSSOMOS.
   ARQUIVO : CROGRASP.H
*/

  // ------------------------------------------------
  #define CROMOSLIB
  // ------------------------------------------------

  // bibliotecas do sistema:
  // ------------------------------------------------
  #include <conio.h>
  #include <stdio.h>
  #include <fstream.h>
  // ------------------------------------------------

  // bibliotecas especiais:
  // ------------------------------------------------
  #ifndef ERROLIB
  #include "c:\temp\wendrer\programa\heads\erro.h"
  #endif
  #ifndef PROBLIB
  #include "c:\temp\wendrer\programa\heads\prob.h"
  #endif
  #ifndef LISTALIB
  #include "c:\temp\wendrer\programa\heads\lista.h"
  #endif
  // ------------------------------------------------------------------------

        const int TEMPOINICIAL = 0;  // tempo inicial p/ agendamento.

  // ------------------------------------------------------------------------

  typedef int Genotipo;
  struct Operacao { int op; int job; };

  class Cromossomo {
    private:

    typedef char* Arquivo;
    typedef Genotipo* Genoma;

    Genoma *crom;              // cromossomo.

    Problema *prob;            // referencia ao problema nJob/nMaq JSP.
    Erro erro;                 // tratamento de erro.

    int   numMaqs,             // numero de maquinas.
          numJobs,             // numero de jobs.
       *numOpsMaq;             // numero de operacoes/maquina.

    float tempoFim;            // tempo de execucao do individuo.

    void iniciar                (Problema &p);
    int proxMaqDisponivel       (float* &tempoMaq, int* &maqOciosa,
                                 int* &numOpsAgMaq);
    float avancarTempoMaqOciosa (float tempo, int nMaq, float* &tempoMaq,
                                 int* &maqOciosa, int* &numOpsAgMaq);
    void agendar                (void);
    float obterTempoAgOp        (Operacao op);
    void randPontosCorte        (int* &corte1, int* &corte2);
    void setGene                (int nMaq,int indOp, Genotipo genotipo);
    void obterGenotipo          (int nMaq,int indOp, Genotipo &genotipo);
```

```
  short existeOpAg            (int a, int b, Operacao op);
  Genotipo genotipo           (Operacao op);
  Operacao decGenotipo        (Genotipo genotipo);
  int fgulosa                 (Operacao op,int tempoOpsMaq);

  public:
  Cromossomo                (void);
  void criarIndividuo       (float taxaGulosa, Problema &p);
  float obterTempoFimJobs   (void);
  int obterNrMaqs           (void);
  int obterNrOpsMaq         (int nMaq);
  void deletar              (void);
  void mutacao              (void);
  void crossover            (Cromossomo &pai,Cromossomo &filho);
  short igual               (Cromossomo &indiv);
  void copiar               (Cromossomo &indiv);
  Problema* obterProblema   (void);
  void gravarIndividuo      (Arquivo printIndividuo);

}; // classe Cromossomo

Cromossomo :: Cromossomo (void)
{
  numMaqs=numJobs=0;
  tempoFim =TEMPOINICIAL;
  prob=NULL;
  crom=NULL;
} // construtor

void Cromossomo :: iniciar (Problema &p)
{
  if(crom)
   deletar();
  prob=&p;

  if(prob)
  {
   numMaqs=prob->obterNrMaqs();
   numJobs=prob->obterNrJobs();
  }
  if(numMaqs!=0)
  {
   numOpsMaq     = new    int[numMaqs];
   crom          = new Genoma[numMaqs];

   for (int nMaq=0; nMaq<numMaqs; nMaq++)
   {
    numOpsMaq[nMaq]   = prob->obterNrOpsMaq(nMaq);
    crom[nMaq]        = new Genotipo[numOpsMaq[nMaq]];
   }
  }
  else erro.printErro("ERRO: Cromossomo nao criado.");
  return;
} // iniciar

void Cromossomo :: deletar (void)
{
  if(crom)
  {
   delete []numOpsMaq;

   for (int nMaq=0; nMaq<numMaqs; nMaq++)
    delete[] (crom[nMaq]);
```

```
   delete[] crom;
   crom=NULL; prob=NULL;
   numMaqs = numJobs =0;
   tempoFim = TEMPOINICIAL;
  }
  return;
} // deletar

void Cromossomo :: copiar (Cromossomo &indiv)
{
  if(crom!=indiv.crom)
  {
   if(crom)
    deletar();
   if(indiv.crom)
   {
    iniciar(*(indiv.prob));
    tempoFim = indiv.tempoFim;
    for (int nMaq=0; nMaq<numMaqs; nMaq++)
     for(int indOp=0; indOp<numOpsMaq[nMaq]; indOp++)
     crom[nMaq][indOp] = indiv.crom[nMaq][indOp];
   }
  }
  return;
} // copiar

Problema* Cromossomo :: obterProblema (void)
{  return (prob);  }

short Cromossomo :: igual (Cromossomo &indiv)
{
  /*
     verifica se cromossomos sao iguais.
  */
  if(crom==indiv.crom)
   return 1;
  else
  {
    if(prob!=indiv.prob)
     return 0;
    else
    {
     for (int nMaq=0; nMaq<numMaqs; nMaq++)
       for(int indOp=0; indOp<numOpsMaq[nMaq]; indOp++)
      if (crom[nMaq][indOp] != indiv.crom[nMaq][indOp])
        return 0;
    }
  }
  return 1;
} // igual

int Cromossomo :: obterNrMaqs (void)
{ return (numMaqs); }

int Cromossomo :: obterNrOpsMaq (int nMaq)
{
  if (nMaq >=0 && nMaq <numMaqs)
   return (numOpsMaq[nMaq]);
  return (0);
}

float Cromossomo :: obterTempoFimJobs (void)
{ return tempoFim;  }
```

```
  void Cromossomo :: setGene (int nMaq, int indOp, Genotipo genotipo)
  {
    if((nMaq>=0 && nMaq<numMaqs) &&
       (indOp>=0 && indOp<numOpsMaq[nMaq]))
      crom[nMaq][indOp]=genotipo;
    else erro.printErro("ERRO: campo nao encontrado para inserir
operacao.");
    return;
  } // setGene

  void Cromossomo :: obterGenotipo(int nMaq, int indOp, Genotipo &genotipo)
  {
    if(nMaq<numMaqs && indOp<numOpsMaq[nMaq])
     genotipo=crom[nMaq][indOp];
    else erro.printErro("ERRO: Operacao nao encontrada.");
    return;
  } // obterGenotipo

  short Cromossomo :: existeOpAg (int a, int b, Operacao op)
  {
    Genotipo auxGenotipo = genotipo(op);
    int nMaq=prob->obterMaqOp(op.job,op.op);

    if (a>=0 && (b>=a && b< numOpsMaq[nMaq]))
    for(int indOp=a; indOp<=b; indOp++)
    {
      if ( auxGenotipo==crom[nMaq][indOp])
      return 1;
    }
    return 0;
  } // existeOpAg

  void Cromossomo :: randPontosCorte(int* &corte1, int* &corte2)
  {
    float tempoC1, tempoC2,   // tempo de corte1 e tempo de corte2.
          indOp, nMaq;
    Operacao auxOp1, auxOp2;

    if (numMaqs==0) erro.printErro ("ERRO: Cromossomo nao definido.");
    corte1 = new int[numMaqs];
    corte2 = new int[numMaqs];

    corte1[0]=rand()%numOpsMaq[0];
    corte2[0]=(rand()%(numOpsMaq[0]-corte1[0]))+corte1[0];

    auxOp1 = decGenotipo(crom[0][corte1[0]]);
    auxOp2 = decGenotipo(crom[0][corte2[0]]);
    tempoC1=obterTempoAgOp(auxOp1);
    tempoC2=obterTempoAgOp(auxOp2)+
            prob->obterTempoOp(auxOp2.job,auxOp2.op);
    for (nMaq=1; nMaq<numMaqs; nMaq++)
    {
     indOp=0;
     while(indOp<numOpsMaq[nMaq] &&
           obterTempoAgOp(decGenotipo(crom[nMaq][indOp])) < tempoC1)
     indOp++;
     if (indOp<numOpsMaq[nMaq])
     {
      if(indOp-1>=0)
      {
       if( (obterTempoAgOp(decGenotipo(crom[nMaq][indOp]))-tempoC1) <
           (tempoC1-obterTempoAgOp(decGenotipo(crom[nMaq][indOp-1]))))
```

```
        corte1[nMaq]=indOp;
        else
        corte1[nMaq]=indOp-1;
       }
       else corte1[nMaq]=indOp;
       while(indOp<numOpsMaq[nMaq] &&
           (obterTempoAgOp(decGenotipo(crom[nMaq][indOp])) +
              prob->obterTempoOp(decGenotipo(crom[nMaq][indOp]).job,
                                 decGenotipo(crom[nMaq][indOp]).op)) <
                                                                  tempoC2)
       indOp++;
       if(indOp<numOpsMaq[nMaq])
       {
        if(indOp-1>=0)
        {
          Operacao auxOp     = decGenotipo(crom[nMaq][indOp]),
                   auxOpPred = decGenotipo(crom[nMaq][indOp-1]);

          if( ((obterTempoAgOp(auxOp)+
                prob->obterTempoOp(auxOp.job,auxOp.op))- tempoC2) <
                (tempoC2-(obterTempoAgOp(auxOpPred)+
                 prob->obterTempoOp(auxOpPred.job,auxOpPred.op))) )
          corte2[nMaq]=indOp;
          else corte2[nMaq]=indOp-1;
        }
        else corte2[nMaq]=indOp;
       }
       else corte2[nMaq]=indOp-1;
      }
     else corte1[nMaq]=corte2[nMaq]=indOp-1;
    } // for
    return;
  } // randPontosCorte

  void Cromossomo :: crossover( Cromossomo &pai, Cromossomo &filho)
  {
    /*
      Operacao crossover que gera um descendente direto
      do pai corrente. Filho herda do pai corrente operações
      entre corte1 e corte2 e do outro pai as demais operações
      nao herdadas Do pai corrente segundo o operador genetico OX.
    */

    int nMaq, indOp, auxIndOp,
      *corte1, *corte2;
    Genotipo auxGenotipo; Cromossomo desc;

    if (crom)
    {
     if ( prob!=pai.prob )
      erro.printErro("ERRO: Cromossomos pais nao se referem ao mesmo
problema p/ crossover.");
     desc.iniciar(*prob);
    }
    else erro.printErro("ERRO: Cromossomo inexistente p/ crossover");

    randPontosCorte(corte1,corte2);   // cortes no pai corrente.

    for(nMaq=0; nMaq<numMaqs; nMaq++)
    {
      for(indOp=corte1[nMaq]; indOp<=corte2[nMaq]; indOp++)
      {
       obterGenotipo(nMaq,indOp,auxGenotipo);
```

```
       desc.setGene(nMaq,indOp,auxGenotipo);
      }
      auxIndOp=corte2[nMaq]+1;
      for(indOp=corte2[nMaq]+1; indOp<numOpsMaq[nMaq]; indOp++)
      {
        pai.obterGenotipo(nMaq,indOp,auxGenotipo);
        if ( (!(desc.existeOpAg(corte1[nMaq],corte2[nMaq],
                                decGenotipo(auxGenotipo)))) )
        {
          desc.setGene(nMaq,auxIndOp,auxGenotipo);
          auxIndOp++;
        }
      }
      for(indOp=0; indOp<=corte2[nMaq]; indOp++)
      {
        if (auxIndOp >= numOpsMaq[nMaq]) auxIndOp=0;
        pai.obterGenotipo(nMaq,indOp,auxGenotipo);
        if ( (!(desc.existeOpAg(corte1[nMaq],corte2[nMaq],
              decGenotipo(auxGenotipo)))) )
        {
          desc.setGene(nMaq,auxIndOp,auxGenotipo);
          auxIndOp++;
        }
      }
    }
    delete[] corte1;
    delete[] corte2;
    desc.agendar();
    filho.copiar(desc);
    desc.deletar();
    return;
  } // crossover.

  void Cromossomo :: mutacao (void)
  {
     Genotipo auxGenotipo;
     int randMaq, rand1OpMaq, rand2OpMaq;

     if(crom==NULL) erro.printErro("ERRO: Cromossomo nao criado p/
mutacao.");

     randMaq = rand()%numMaqs;
     rand1OpMaq = rand()%numOpsMaq[randMaq];
     do {
       rand2OpMaq = rand()%numOpsMaq[randMaq];
     } while ( rand1OpMaq == rand2OpMaq);
     auxGenotipo = crom[randMaq][rand1OpMaq];
     crom[randMaq][rand1OpMaq]= crom[randMaq][rand2OpMaq];
     crom[randMaq][rand2OpMaq]= auxGenotipo;
     agendar();
     return;
  } // mutacao

  Genotipo Cromossomo :: genotipo ( Operacao op)
  {
    int primeiraOpJob = 0;

    if( op.job>=numJobs ||
        op.op>=prob->obterNrOpsJob(op.job) )
     erro.printErro("operacao invalida");
    for(int nJob=1;nJob <= op.job; nJob++)
     primeiraOpJob += prob->obterNrOpsJob(nJob-1);
    return(primeiraOpJob + op.op);
```

```
  } // codifica operacao

  Operacao Cromossomo :: decGenotipo ( Genotipo genotipo)
  {
    Operacao auxOp;
    int nJob=0; int ultimaOpJob=0;
    short acheiJob=0;
    if (genotipo>=0)
    {
     while (!acheiJob && nJob<numJobs)
     {
       ultimaOpJob += prob->obterNrOpsJob(nJob);
       if(genotipo < ultimaOpJob)
      acheiJob=1;
       else nJob++;
     }
     if (genotipo>=ultimaOpJob) erro.printErro("valor invalido para
operacao");
    }
    else erro.printErro("valor invalido para operacao");
    auxOp.job= nJob;
    auxOp.op = (prob->obterNrOpsJob(nJob)-(ultimaOpJob-genotipo-1))-1;
    return auxOp;
  } // decodifica genotipo

  int Cromossomo :: fgulosa ( Operacao op,int tempoOpsMaq)
  {
    return (tempoOpsMaq+prob->obterTempoOp(op.job,op.op));
  } // funcao gulosa

  void Cromossomo :: criarIndividuo (float taxaGulosa, Problema &p)
  {
    Genotipo auxGenotipo;
    Operacao auxOp, auxOp1;
    short cromossomoGerado;
    Lista<Genotipo>  *conjOpsAgMaq,
                conjOpsCandidatas,
                       conjOpsLCR;
    int nMaq, nJob, auxMaq, maq,
                   *gene,
            *proxOpAgJob,
              *maqOciosa,
          minTempoOpsMaq,
          maxTempoOpsMaq;
    float      *tempoMaq;


    iniciar(p);

    gene         = new int[numMaqs];
    tempoMaq     = new float[numMaqs];
    maqOciosa    = new int[numMaqs];
    proxOpAgJob  = new int[numJobs];
    conjOpsAgMaq = new Lista<Genotipo>[numMaqs];

    for(nMaq=0; nMaq<numMaqs; nMaq++)
    {
     gene[nMaq]      = 0;
     maqOciosa[nMaq] = 0;
     tempoMaq[nMaq] = TEMPOINICIAL;
    }

    for (nJob=0; nJob<numJobs; nJob++)
```

```
{
  auxOp.job = nJob;
  auxOp.op  = 0;
  conjOpsAgMaq[prob->obterMaqOp(nJob,0)].inserir(genotipo(auxOp));

  proxOpAgJob[nJob] = 0;
}
do
{
 maq=proxMaqDisponivel(tempoMaq,maqOciosa,gene);
 for (nMaq=0; nMaq<numMaqs; nMaq++)
 {
  if (gene[nMaq]>0 && (tempoMaq[maq]>=tempoMaq[nMaq]) )
  {
   auxOp = decGenotipo(crom[nMaq][gene[nMaq]-1]);

   if (proxOpAgJob[auxOp.job]==auxOp.op)
   {
    if ( ((auxOp.op)+1) < prob->obterNrOpsJob(auxOp.job) )
    {
     auxOp1.job = auxOp.job;
     auxOp1.op = (auxOp.op)+1;
     conjOpsAgMaq[prob->obterMaqOp(auxOp1.job,
                                   auxOp1.op)].inserir(genotipo(auxOp1));
    }
    proxOpAgJob[auxOp.job]++;
   }
  }
 }
 if (!(conjOpsAgMaq[maq].vazia()))
 {
   int verifica=0;
   while (!(conjOpsAgMaq[maq].vazia()))
   {
    conjOpsAgMaq[maq].retirar(auxGenotipo);
    conjOpsCandidatas.inserir(auxGenotipo);
    auxOp = decGenotipo(auxGenotipo);
    if (verifica==0)
     minTempoOpsMaq = maxTempoOpsMaq = fgulosa(auxOp,tempoMaq[maq]);
    else
    {
     if (minTempoOpsMaq > fgulosa(auxOp,tempoMaq[maq]))
      minTempoOpsMaq = fgulosa(auxOp,tempoMaq[maq]);
     if (maxTempoOpsMaq < fgulosa(auxOp,tempoMaq[maq]))
      maxTempoOpsMaq = fgulosa(auxOp,tempoMaq[maq]);
    }
    verifica++;
   }
   while (!(conjOpsCandidatas.vazia()))
   {
    conjOpsCandidatas.retirar(auxGenotipo);
    auxOp = decGenotipo(auxGenotipo);

    if(fgulosa(auxOp,tempoMaq[maq]) <= (minTempoOpsMaq +
            (taxaGulosa/100.0)*(maxTempoOpsMaq-minTempoOpsMaq)))
      conjOpsLCR.inserir(auxGenotipo);
    else conjOpsAgMaq[maq].inserir(auxGenotipo);
   }
   conjOpsLCR.retirar(auxGenotipo,rand()%conjOpsLCR.tam());
   auxOp = decGenotipo(auxGenotipo);
   crom[maq][gene[maq]]=auxGenotipo;
   tempoMaq[maq]=fgulosa(auxOp,tempoMaq[maq]);
   for(auxMaq=0; auxMaq<numMaqs; auxMaq++)
```

```
      maqOciosa[auxMaq]=0;
      gene[maq]++;

      while (!(conjOpsLCR.vazia()))
      {
       conjOpsLCR.retirar(auxGenotipo);
       conjOpsAgMaq[maq].inserir(auxGenotipo);
      }
    }
    else  // atualizar tempo para maquina ociosa.
    tempoMaq[maq]=avancarTempoMaqOciosa(tempoMaq[maq],
                                        maq,tempoMaq,maqOciosa,gene);

    cromossomoGerado=1; nMaq=0;
    while (nMaq<numMaqs && cromossomoGerado) {
     if (!(gene[nMaq]==numOpsMaq[nMaq]))
      cromossomoGerado=0;
     nMaq++;
    }
  } while (!(cromossomoGerado));

  tempoFim=tempoMaq[0];
  for (nMaq=1; nMaq<numMaqs; nMaq++)
    if (tempoMaq[nMaq] > tempoFim)
     tempoFim=tempoMaq[nMaq];

  delete []conjOpsAgMaq;
  delete []gene;
  delete []tempoMaq;
  delete []proxOpAgJob;
  delete []maqOciosa;

  return;
} // criarIndividuo

void Cromossomo :: agendar (void)
{
  Operacao auxOp = {-1,-1};
  tempoFim = obterTempoAgOp(auxOp);
} // agendar

float Cromossomo :: obterTempoAgOp (Operacao op)
{
 short todasOpsAgendadas,
       acheiOp;

 int nMaq, nJob,
     maq, indOp,
     *proxOpAgJob,
     *maqOciosa,
     *numOpsAgMaq,
     *indPartida;

 float    tempo,
      *tempoMaq;

 if(crom==NULL)
 erro.printErro("ERRO: Nao e'possivel agendar. Cromossomo nao criado.");

 numOpsAgMaq   = new    int[numMaqs];
 tempoMaq      = new  float[numMaqs];
 maqOciosa     = new    int[numMaqs];
 proxOpAgJob   = new    int[numJobs];
```

```
indPartida    = new    int[numMaqs];

for (nMaq=0; nMaq<numMaqs; nMaq++)
{
  tempoMaq[nMaq]    = TEMPOINICIAL;
  maqOciosa[nMaq]   = 0;
  numOpsAgMaq[nMaq] = 0;
  indPartida[nMaq]  = 0;
}
for (nJob=0; nJob<numJobs; nJob++)
proxOpAgJob[nJob]=0;

acheiOp = 0;

do
{
  maq=proxMaqDisponivel(tempoMaq,maqOciosa,numOpsAgMaq);
  for (nMaq=0; nMaq<numMaqs; nMaq++)
   if (numOpsAgMaq[nMaq]>0 && (tempoMaq[maq]>=tempoMaq[nMaq]) )
   {
    auxOp = decGenotipo(crom[nMaq][numOpsAgMaq[nMaq]-1]);

    if (proxOpAgJob[auxOp.job]==auxOp.op)
    {
     proxOpAgJob[auxOp.job]++;
    }
   }
  int i = numOpsAgMaq[maq];
  indOp = indPartida[maq];
  while (i < numOpsMaq[maq] &&
       proxOpAgJob[decGenotipo(crom[maq][indOp]).job] !=
       decGenotipo(crom[maq][indOp]).op )
  {
   indOp = (((indOp+1)%numOpsMaq[maq])==0) ? numOpsAgMaq[maq] : indOp+1;
   i++;
  }
  if (i < numOpsMaq[maq])
  {
    Genotipo auxGenotipo = crom[maq][indOp];
    auxOp = decGenotipo(auxGenotipo);

    if (numOpsAgMaq[maq] == 0)
    indPartida[maq] = ( ((indOp+1)%numOpsMaq[maq])==0) ? 1 : indOp+1;
    else if (indOp >= indPartida[maq] )
    indPartida[maq] = ( ((indPartida[maq]+1)%numOpsMaq[maq])==0) ?
                          numOpsAgMaq[maq]+1 : indPartida[maq]+1;

    if (op.job==auxOp.job &&
      op.op ==auxOp.op )
    {
     tempo = tempoMaq[maq];
     acheiOp=1;
    }
    else
    {
     if(op.job==-1 && op.op==-1)
     {
      while ((indOp-1)>=numOpsAgMaq[maq])
      {
      crom[maq][indOp]=crom[maq][indOp-1];
      indOp--;
      }
      crom[maq][indOp]=auxGenotipo;
```

```
        }
        tempoMaq[maq]+=prob->obterTempoOp(auxOp.job,auxOp.op);
        for(int auxMaq=0; auxMaq<numMaqs; auxMaq++)
        maqOciosa[auxMaq]=0;
        numOpsAgMaq[maq]++;
       }
      }
      else
       tempoMaq[maq]=avancarTempoMaqOciosa(tempoMaq[maq],
                                    maq,tempoMaq,maqOciosa,numOpsAgMaq);

     todasOpsAgendadas=1; nMaq=0;
     while (nMaq<numMaqs && todasOpsAgendadas) {
      if (!(numOpsAgMaq[nMaq]==numOpsMaq[nMaq]))
       todasOpsAgendadas=0;
      nMaq++;
     }
   } while (!(todasOpsAgendadas) && (!acheiOp));

   if( (!(op.job==-1 && op.op==-1)) && !acheiOp )
    erro.printErro("operacao inexistente no cromossomo");

   if(op.job==-1 && op.op==-1)
   {
    tempo=tempoMaq[0];
    for (nMaq=1; nMaq<numMaqs; nMaq++)
     if (tempoMaq[nMaq] > tempo)
      tempo=tempoMaq[nMaq];
   }

   delete []proxOpAgJob;
   delete []numOpsAgMaq;
   delete []tempoMaq;
   delete []maqOciosa;
   delete []indPartida;

   return (tempo);
  } // obterTempoAgOp


  int Cromossomo :: proxMaqDisponivel (float* &tempoMaq,int* &maqOciosa,
int* &numOpsAgMaq )
  {
    /*
       Buscar a maquina disponivel para agendamento,  que nao seja
       uma maquina onde todas as suas operacoes ja foram agendadas
       ou que nao seja uma maquina ociosa,pois, uma maquina so' se
       torna ociosa quando nao ha' operacao que a requer neste
       instante de tempo.
    */

    int nMaq, maq, auxMaq,
      acheiOp;

    do
    {
     acheiOp = 1;
     nMaq=maq=numMaqs-1;
     while (nMaq>0)
     {
      if ((!(maqOciosa[maq])) && (!(numOpsAgMaq[maq]==numOpsMaq[maq])) )
      {
       auxMaq=nMaq-1;
```

```
       if ((!(maqOciosa[auxMaq])) &&
           (!(numOpsAgMaq[auxMaq]==numOpsMaq[auxMaq])) )
       {
        if (tempoMaq[maq] >= tempoMaq[auxMaq])
        maq=auxMaq;
       }
      }
      else maq--;
      nMaq--;
     }
     if (maq==0 && ((numOpsAgMaq[maq]==numOpsMaq[maq]) || maqOciosa[maq]) )
     {
      // temos somente maquinas ociosas e/ou agendadas.
      acheiOp = 0;
      for(int auxMaq=0; auxMaq<numMaqs; auxMaq++)
       maqOciosa[auxMaq]=0;
     }
   } while(!acheiOp);
   return (maq);
  } // proxMaqDisponivel

  float Cromossomo :: avancarTempoMaqOciosa (float tempo, int nMaq,
                      float* &tempoMaq, int* &maqOciosa, int* &numOpsAgMaq)
  {
    /*
      calcular o tempo mais proximo do tempo <tempo> para que a maquina
      <nMaq> se encontre disponivel.
    */

    float menorTempo=-1;

    for (int auxMaq=0; auxMaq<numMaqs; auxMaq++)
     if (nMaq!=auxMaq)
     {
      if( (tempoMaq[auxMaq]>tempo) ||
          ((tempoMaq[auxMaq]==tempo) && (!(maqOciosa[auxMaq])) &&
           (!(numOpsAgMaq[auxMaq]==numOpsMaq[auxMaq]))) )
      {
        if (menorTempo==-1) menorTempo=tempoMaq[auxMaq];
        else
         if (tempoMaq[auxMaq]< menorTempo) menorTempo=tempoMaq[auxMaq];
      }
     }
     if (tempo==menorTempo) maqOciosa[nMaq]=1;
     else
       for(int auxMaq=0; auxMaq<numMaqs; auxMaq++)
       maqOciosa[auxMaq]=0;
    return (menorTempo);
  } // avancarTempoMaqOciosa

  void Cromossomo :: gravarIndividuo (char* printIndividuo)
  {
    FILE *fout=NULL;
    Operacao auxOp;

    if(!(fout=fopen(printIndividuo,"a+t")))
    erro.printErro("\nNao foi possivel gravar o arquivo
%\n",printIndividuo);

    fprintf(fout,"\n\n INDIVIDUO %d/%d JSP:\n\n",numJobs,numMaqs);

    if (crom==NULL) erro.printErro("ERRO: Cromossomo nao criado.");
    for (int nMaq=0; nMaq<numMaqs; nMaq++)
```

```
    {
      fprintf(fout," M%-4d- ",nMaq+1);
      for (int indOp=0; indOp<numOpsMaq[nMaq]; indOp++)
      {
      auxOp = decGenotipo(crom[nMaq][indOp]);
      fprintf(fout,"%3d-%-3d %-4g ",
            auxOp.job + 1,auxOp.op + 1,obterTempoAgOp(auxOp) );
      }
      fprintf(fout,"\n");
    }
    fclose(fout);
    return;
  } // gravarIndividuo


/*
   CLASSE POPULACAO.
   ARQUIVO : POP.H
*/

  // -------------------------------------------------
  #define POPLIB
  // -------------------------------------------------

  // bibliotecas do sistema:
  // -------------------------------------------------
  #include <math.h>
  // -------------------------------------------------

  // bibliotecas especiais:
  // -------------------------------------------------
  #ifndef ERROLIB
  #include "c:\temp\wendrer\programa\heads\erro.h"
  #endif
  #ifndef PROBLIB
  #include "c:\temp\wendrer\programa\heads\prob.h"
  #endif
  #ifndef CROMOSLIB
  #include "c:\temp\wendrer\programa\heads\croGRASP.h"
  #endif
  #ifndef LISTALIB
  #include "c:\temp\wendrer\programa\heads\lista.h"
  #endif
  // ---------------------------------------------------------------------

              const float DESVIO=2.0;   // desvio padrao.

  // ---------------------------------------------------------------------

 class Populacao
 {
   private:

   typedef char* Arquivo;

   Problema *prob;
   Erro erro;

   Lista<Cromossomo> conjIndivs;

   Cromossomo menosAptoIndiv,  // individuo menos apto.
              maisAptoIndiv;   // individuo mais apto.
```

```
  int nrIndivPopInicial, nrIndivPop;

  double  somaExpect,
          maxTempo,
          minAdap,
          mediaAdap, desvioAdap;

  void gerarExpectIndivs        (void);

  public:
  Populacao  (void);
  Populacao  (float taxaGulosa, int tamPopInicial, Problema &p );
  ~Populacao (void);

  short vaziaPop                (void);
  int obterTamPop               (void);
  int obterTamPopInicial        (void);
  void criarPopInicial          (float taxaGulosa,int tamPopInicial,
                                 Problema &p);
  void sortearIndivsRoletaRussa (Populacao &novaPop);
  void sortearIndivRoletaRussa  (Cromossomo &indiv);
  void sortearIndivSemRep       (Cromossomo &indiv);
  void crossover                (float taxaCrossover, Populacao &descPop);
  void mutacao                  (float taxaMutacao);
  void obterMaisAptoIndiv       (Cromossomo &indiv);
  void obterMenosAptoIndiv      (Cromossomo &indiv);
  void inserirIndiv             (Cromossomo &indiv);
  void retirarIndiv             (Cromossomo &indiv, int indice=0);
  void obterIndiv               (Cromossomo &indiv, int indice=0);
  void inserirIndivsPop         (Populacao  &pop);
  void excluirTodosIndivs       (void);
  double fobjetiva              (Cromossomo &indiv);
  double fexpectativa           (Cromossomo &indiv);
  void gravarMelhorIndividuo    (Arquivo printIndividuo);

}; // class Populacao

Populacao :: Populacao (void)
{
  nrIndivPopInicial=nrIndivPop=0;
  prob=NULL;
} // construtor

Populacao :: Populacao (float taxaGulosa, int tamPopInicial, Problema &p )
{
  nrIndivPopInicial=nrIndivPop=0;
  criarPopInicial(taxaGulosa,tamPopInicial,p);
} // construtor

Populacao :: ~Populacao (void)
{
  while(!(vaziaPop()))
    excluirTodosIndivs();
  nrIndivPopInicial=nrIndivPop=0;
  prob=NULL;
} // destrutor

short Populacao :: vaziaPop (void)
{  return (nrIndivPop==0); }

int Populacao :: obterTamPop (void)
{  return (nrIndivPop); }
```

```
int Populacao :: obterTamPopInicial (void)
{  return (nrIndivPopInicial); }

void Populacao :: obterMaisAptoIndiv (Cromossomo &indiv)
{
  if (!(vaziaPop()))
   indiv.copiar(maisAptoIndiv);
  return;
} // obterMaisAptoIndiv

void Populacao :: obterMenosAptoIndiv (Cromossomo &indiv)
{
  if (!(vaziaPop()))
   indiv.copiar(menosAptoIndiv);
  return;
} // obterMaisAptoIndiv

void Populacao :: criarPopInicial (float taxaGulosa, int tamPopInicial,
                                   Problema &p)
{
   Cromossomo auxIndiv;
   prob=&p;

   while (!(vaziaPop()))
    excluirTodosIndivs();
   nrIndivPopInicial = tamPopInicial;
   for( int nIndiv=0; nIndiv<nrIndivPopInicial; nIndiv++)
   {
     auxIndiv.criarIndividuo(taxaGulosa,*prob);
     inserirIndiv(auxIndiv);
   }
   auxIndiv.deletar();
   return;
} // criarPopInicial

double Populacao :: fobjetiva (Cromossomo &indiv)
{
  if (!(vaziaPop()))
  {
   Cromossomo auxIndiv;
   int nIndiv=0; short achouIndiv=0;

   while (nIndiv<nrIndivPop && (!(achouIndiv)) )
   {
    obterIndiv(auxIndiv,nIndiv);
    if(indiv.igual(auxIndiv))  achouIndiv=1;
    else nIndiv++;
   }
   auxIndiv.deletar();
   if(achouIndiv) return (indiv.obterTempoFimJobs());
  }
  return (-1.0);
} // fobjetiva.

double Populacao :: fexpectativa (Cromossomo &indiv)
{
  //   Obter a expectativa(chance) do individuo ser escolhido.

  if (!(vaziaPop()))
  {
   double aptidao, expectativa, tempo;
   if ((tempo=fobjetiva(indiv))!=-1.0)
   {
```

```
      aptidao = (minAdap+maxTempo) - tempo;
      expectativa = aptidao-(mediaAdap-DESVIO*desvioAdap);

      if (expectativa >0.0) return (expectativa/somaExpect);
      return (0.0);
    }
   }
   return (-1.0);
} // fexpectativa.

void Populacao :: gerarExpectIndivs (void)
{
 Cromossomo auxIndiv;
 int nIndiv;
 double aux,
      somaAdap,
      expectativa,
      *aptidao;

 aux=maxTempo=somaAdap=somaExpect=0.0;
 minAdap=mediaAdap=desvioAdap=0.0;

 if(!(vaziaPop()))
 {
  for (nIndiv=0; nIndiv < nrIndivPop; nIndiv++)
  {
   obterIndiv(auxIndiv,nIndiv);
   if (nIndiv==0) {
    menosAptoIndiv.copiar(auxIndiv);
    maisAptoIndiv.copiar(auxIndiv);
   }
   else
   {
     if (fobjetiva(auxIndiv) > fobjetiva(menosAptoIndiv))
      menosAptoIndiv.copiar(auxIndiv);
     if (fobjetiva(auxIndiv) < fobjetiva(maisAptoIndiv))
      maisAptoIndiv.copiar(auxIndiv);
   }
  }
  maxTempo= fobjetiva(menosAptoIndiv);
  minAdap = fobjetiva(maisAptoIndiv);
  aptidao = new double[nrIndivPop];

  for (nIndiv=0; nIndiv < nrIndivPop; nIndiv++)
  {
   obterIndiv(auxIndiv,nIndiv);
   aptidao[nIndiv] = (minAdap+maxTempo) - fobjetiva(auxIndiv);
   somaAdap += aptidao[nIndiv];
  }
  mediaAdap=somaAdap/nrIndivPop;
  for (nIndiv=0; nIndiv < nrIndivPop; nIndiv++)
   aux+=pow(mediaAdap-aptidao[nIndiv],2.0);
  desvioAdap=sqrt(aux);
  for (nIndiv=0; nIndiv < nrIndivPop; nIndiv++)
  {
   expectativa = aptidao[nIndiv]-(mediaAdap-DESVIO*desvioAdap);
   somaExpect+= (expectativa<=0.0)? 0.0 : expectativa;
  }
  delete aptidao;
  auxIndiv.deletar();
 }
 return;
} // gerarExpectIndivs
```

```
 void Populacao :: inserirIndiv (Cromossomo &indiv)
 {
   Cromossomo auxIndiv;

   if ( prob==NULL )
     prob=indiv.obterProblema();
   else
     if (prob!=indiv.obterProblema())
     erro.printErro("ERRO: Populacao deve incluir individuos que se referem
ao mesmo problema.");
   auxIndiv.copiar(indiv);
   conjIndivs.inserir(auxIndiv);
   nrIndivPop++;
   gerarExpectIndivs();
   return;
 } // inserirIndiv.

 void Populacao :: inserirIndivsPop (Populacao &pop)
 {
   if (!(pop.vaziaPop()))
   {
     Cromossomo auxIndiv;

     for (int nIndiv=0; nIndiv<pop.obterTamPop(); nIndiv++)
     {
      pop.obterIndiv(auxIndiv,nIndiv);
      inserirIndiv(auxIndiv);
     }
     gerarExpectIndivs();
     auxIndiv.deletar();
   }
   return;
 } // inserirIndivsPop.

 void Populacao :: obterIndiv (Cromossomo &indiv, int indice)
 {
    if(!(vaziaPop()))
     if(indice >=0 && indice < nrIndivPop)
     {
      Cromossomo auxIndiv;

      conjIndivs.obter(auxIndiv,indice);
      indiv.copiar(auxIndiv);
     }
    return;
 } // obterIndiv.

 void Populacao :: retirarIndiv (Cromossomo &indiv, int indice)
 {
    if(!(vaziaPop()))
     if(indice >=0 && indice < nrIndivPop)
     {
      Cromossomo auxIndiv;

      conjIndivs.retirar(auxIndiv,indice);
      nrIndivPop--;
      indiv.copiar(auxIndiv);
      auxIndiv.deletar();
      gerarExpectIndivs();
     }
    return;
 } // retirarIndiv.
```

```
void Populacao :: excluirTodosIndivs (void)
{
  Cromossomo auxIndiv;
  while(!(vaziaPop()))
   retirarIndiv(auxIndiv);
  auxIndiv.deletar();
  return;
} // excluirTodosIndivs.

void Populacao :: sortearIndivsRoletaRussa (Populacao &novaPop)
{
  Cromossomo auxIndiv;

  if(!(novaPop.vaziaPop()))
   novaPop.excluirTodosIndivs();
  if(!(vaziaPop()))
  {
   for (int lance=0; lance<nrIndivPop; lance++)
   {
     sortearIndivRoletaRussa(auxIndiv);
     novaPop.inserirIndiv(auxIndiv);
   }
   auxIndiv.deletar();
  }
  else erro.printErro("ERRO: Populacao vazia. Individuo nao sorteado");
  return;
} // sortearIndivsRoletaRussa

void Populacao :: sortearIndivRoletaRussa( Cromossomo &indiv)
{
  /*
     A selecao de individuos da populacao e'baseado no
     criterio de roleta russa, onde os individuos de maior expectativa
     tem maiores chances de serem selecionados.
  */

  Cromossomo auxIndiv;
  double valorSorteado, somatorioExpect;

  if(!(vaziaPop()))
  {
    valorSorteado = somatorioExpect=0.0;

    for(int i=1; i<=20; i++)    // precisao de 20 casas decimais
    valorSorteado+=(rand()%10)/pow(10.0,i);    // lance do dado.

    int nIndiv=0; short achouIndivSorteado=0;
    while ( nIndiv<nrIndivPop && (!(achouIndivSorteado)) )
    {
       obterIndiv(auxIndiv,nIndiv);
       somatorioExpect+=fexpectativa(auxIndiv);
       if (valorSorteado < somatorioExpect)
       {
          achouIndivSorteado=1;
          indiv.copiar(auxIndiv);
       }
       nIndiv++;
    }
    if (somatorioExpect==0)
     indiv.copiar(maisAptoIndiv);
    auxIndiv.deletar();
  }
```

```
  else erro.printErro("ERRO: Populacao vazia. Individuo nao sorteado");
  return;
} // sortearIndivRoletaRussa.

void Populacao :: sortearIndivSemRep ( Cromossomo &indiv )
{
  // sortear aleatoriamente um individuo sem reposicao.

  int valorSorteado;
  if (!(vaziaPop()))
  {
   valorSorteado = rand()%nrIndivPop;
   retirarIndiv(indiv,valorSorteado);
  }
  else erro.printErro("ERRO: Populacao vazia. Individuo nao sorteado");
  return;
} // sortearIndivSemRep.

void Populacao :: crossover (float taxaCrossover, Populacao &descPop )
{
  int sorteioTaxa, sorteioIndiv,
      limiteSorteio, maxTentativas, nTentativa, tamPop;
  Cromossomo pai1, pai2, filho1, filho2;

  if(!(vaziaPop()))
  {
   tamPop=limiteSorteio=nrIndivPop;
   nTentativa=0; maxTentativas=limiteSorteio-(limiteSorteio/2);
   while (nTentativa<maxTentativas)
   {
    sorteioIndiv = rand()%limiteSorteio;
    retirarIndiv(pai1,sorteioIndiv);
    limiteSorteio--;
    if((nTentativa+1!=maxTentativas) || tamPop%2==0)
    {
      sorteioIndiv = rand()%limiteSorteio;
      retirarIndiv(pai2,sorteioIndiv);
      limiteSorteio--;
      sorteioTaxa = (rand()%100)+1;
      if(sorteioTaxa <= taxaCrossover)
      {
       pai1.crossover(pai2,filho1);
       descPop.inserirIndiv(filho1);
       pai2.crossover(pai1,filho2);
       descPop.inserirIndiv(filho2);
      }
      else {
            descPop.inserirIndiv(pai1);   // processo chamado clonagem
            descPop.inserirIndiv(pai2);   // de individuos.
      }
      inserirIndiv(pai2);
    }
    else descPop.inserirIndiv(pai1);
    inserirIndiv(pai1);
    nTentativa++;
   }
   pai1.deletar();   pai2.deletar();
   filho1.deletar(); filho2.deletar();
  }
  return;
} // crossover.

void Populacao :: mutacao (float taxaMutacao )
```

```
 {
   int sorteioTaxa;
   int nTentativa, maxTentativas, nIndiv;
   Cromossomo auxIndiv;

   nTentativa=0; maxTentativas=nrIndivPop;
   nIndiv=nTentativa;

   if (!(vaziaPop()))
   {
    for( ; nTentativa<maxTentativas; nTentativa++)
    {
     sorteioTaxa = (rand()%100)+1;
     if(sorteioTaxa <= taxaMutacao)
     {
       retirarIndiv(auxIndiv,nIndiv);
       auxIndiv.mutacao();
       inserirIndiv(auxIndiv);
     }
     else nIndiv++;
    }
    auxIndiv.deletar();
   }
   return;
 } // mutacao.

 void Populacao :: gravarMelhorIndividuo (Arquivo printIndividuo)
 {
   if (vaziaPop())
     erro.printErro("ERRO: Populacao vazia. Nao pode ser gravado
individuo.");
   maisAptoIndiv.gravarIndividuo(printIndividuo);
   return;
 } // gravarMelhorIndividuo.
```

```
/*
   PROGRAMA PRINCIPAL

   ALGORITMO JOB-SHOP COM ALGORITMO GENETICO:
   POPULACAO INICIAL GRASP - MTA

   ARQUIVO : JOBSHOPAG-GRASP.CPP

*/

 // bibliotecas do sistema:

 #include <iostream.h>
 #include <conio.h>
 #include <time.h>
 #include <iomanip.h>
 #include <stdio.h>
 #include <dos.h>
// ---------------------------------------------------------------------
     const float TAXAGULOSA  =   40;  // taxa gulosa.
// ---------------------------------------------------------------------
     const int TAMPOPINICIAL =   15;  // populacao inicial.
     const int MAXTAMPOP     =   15;  // maximo tamanho população.
     const int MAXGERACOES   =   50;  // nr. maximo de gerações.
     const float TAXACROSSOVER = 95;  // taxa percentual de crossover.
     const float TAXAMUTACAO   =  5;  // taxa percentual de mutacao.

// ---------------------------------------------------------------------
// bibliotecas especiais:

 #include "c:\temp\wendrer\programa\heads\erro.h"
 #include "c:\temp\wendrer\programa\heads\prob.h"
 #include "c:\temp\wendrer\programa\heads\croGRASP.h"
 #include "c:\temp\wendrer\programa\heads\pop.h"
// ------------------------------------------------

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

   char ch;
   clock_t iniciarTempo, fimTempoGeracao;
   double tempoMelhorIndiv, tempo;
   int geracaoMelhorIndiv, geracaoPiorIndiv;

   char* dados= "c:\\temp\\wendrer\\programa\\dados\\dado.dat";
   char* saida= "c:\\temp\\wendrer\\programa\\dados\\saida\\agGRASP.dat";
   char* teste= "c:\\temp\\wendrer\\programa\\dados\\testes\\tagGRASP.dat";

   Populacao pop, reprodPop, descPop, sobrevivePais;
   Cromossomo indiv, auxIndiv,
            piorIndivGeracao,melhorIndivGeracao,
            piorIndivGeral,melhorIndivGeral;
   Erro erro;

   FILE *fsaida=NULL, *fteste=NULL;
   if(!(fteste=fopen(teste,"a+t")))
   erro.printErro("\nNao foi possivel gravar o arquivo %\n",teste);

   cout << "\n\n";
   cout << "\n           ALGORITMO JOB-SHOP COM ALGORITMO GENETICO:"
```

```
     << "\n                POPULACAO INICIAL GRASP - MTA";
cout << "\n\n";
do {
 cout << "               <ENTER>-EXECUTAR  <ESC>-CANCELAR";
 ch = getch();
 cout << '\r'; clreol();
} while (ch!=13 && ch!=27);
if (ch == 13)
{
cout << "\n\n PROCESSAMENTO : POR [" << MAXGERACOES << "] GERACOES.";
cout << "\n\n AGUARDE, PROCESSANDO ...\n";

iniciarTempo = clock();

Problema prob(dados);
pop.criarPopInicial(TAXAGULOSA,TAMPOPINICIAL,prob);

fimTempoGeracao = clock();
tempo = (double)(fimTempoGeracao-iniciarTempo)/CLK_TCK;

pop.obterMaisAptoIndiv(melhorIndivGeracao);
pop.obterMenosAptoIndiv(piorIndivGeracao);

fprintf(fteste,"\n GERACAO [0]: MELHOR MAKESPAN = %gu : SEGUNDOS = %gs",
               melhorIndivGeracao.obterTempoFimJobs(),tempo);

cout << "\n GERACAO [" << setfill('0') << setw(3) << 0 << "] : ";
cout << "MELHOR MAKESPAN = "  << setfill(' ') << setw(4) <<
         melhorIndivGeracao.obterTempoFimJobs();
cout << " : SEGUNDOS = " << tempo;

piorIndivGeral.copiar(piorIndivGeracao);
geracaoPiorIndiv   = 0;
melhorIndivGeral.copiar(melhorIndivGeracao);
tempoMelhorIndiv   = tempo;
geracaoMelhorIndiv = 0;

int nGeracao =0;

while (nGeracao < MAXGERACOES)
{
 pop.sortearIndivsRoletaRussa(reprodPop);
 pop.excluirTodosIndivs();
 reprodPop.crossover(TAXACROSSOVER,descPop);
 descPop.mutacao(TAXAMUTACAO);
 while(!(descPop.vaziaPop()))
 {
    descPop.retirarIndiv(auxIndiv);
    pop.inserirIndiv(auxIndiv);
 }
 if( pop.obterTamPop() < MAXTAMPOP )
 {
   while ( (!(reprodPop.vaziaPop())) &&
         (sobrevivePais.obterTamPop() < (MAXTAMPOP-pop.obterTamPop())) )
   {
    reprodPop.sortearIndivSemRep(indiv);

    short indivIgual = 0; int nIndiv=0;
    while (nIndiv<sobrevivePais.obterTamPop() && (!(indivIgual)) )
    {
    sobrevivePais.obterIndiv(auxIndiv,nIndiv);
    if (indiv.igual(auxIndiv))
     indivIgual=1;
```

```
       else nIndiv++;
       }
       if (!(indivIgual)) sobrevivePais.inserirIndiv(indiv);
      }
      while( !(sobrevivePais.vaziaPop()) )
      {
       sobrevivePais.retirarIndiv(auxIndiv);
       pop.inserirIndiv(auxIndiv);
      }
    }
    fimTempoGeracao = clock();
    tempo = double(fimTempoGeracao-iniciarTempo) / CLK_TCK;

    pop.obterMaisAptoIndiv(melhorIndivGeracao);
    pop.obterMenosAptoIndiv(piorIndivGeracao);

    reprodPop.excluirTodosIndivs();
    sobrevivePais.excluirTodosIndivs();
    descPop.excluirTodosIndivs();

    fprintf(fteste,"\n GERACAO [%d]:MELHOR MAKESPAN = %gu :SEGUNDOS = %gs",
          nGeracao+1,melhorIndivGeracao.obterTempoFimJobs(),tempo);

    cout << "\n GERACAO[" <<setfill('0') << setw(3) << nGeracao+1 << "]: ";
    cout << "MELHOR MAKESPAN = "  << setfill(' ') << setw(4) <<
             melhorIndivGeracao.obterTempoFimJobs();
    cout << " : SEGUNDOS = " << tempo;

    if ( melhorIndivGeracao.obterTempoFimJobs() <
         melhorIndivGeral.obterTempoFimJobs() )
    {
      melhorIndivGeral.copiar(melhorIndivGeracao);
      tempoMelhorIndiv   = tempo;
      geracaoMelhorIndiv = nGeracao+1;
    }
    if ( piorIndivGeracao.obterTempoFimJobs() >
       piorIndivGeral.obterTempoFimJobs() )
    {
      piorIndivGeral.copiar(piorIndivGeracao);
      geracaoPiorIndiv = nGeracao+1;
    }

    nGeracao++;
   }
   fprintf(fteste,"\n\n MELHOR SOLUCAO: GERACAO [%d] - MAKESPAN = %gu -
SEGUNDOS = %gs",geracaoMelhorIndiv,
                melhorIndivGeral.obterTempoFimJobs(),
                tempoMelhorIndiv);
   fprintf(fteste,"\n\n MAIOR MAKESPAN = %gu - GERACAO [%d]",
                   piorIndivGeral.obterTempoFimJobs(),geracaoPiorIndiv);
   fclose(fteste);

   cout << "\n\n";
   cout << "\n MELHOR SOLUCAO GERADA :";
   cout << "\n\n GERACAO = [" << geracaoMelhorIndiv << "] : ";
   cout     << "MAKESPAN = " << setfill(' ') << setw(4) <<
                melhorIndivGeral.obterTempoFimJobs();
   cout << " : SEGUNDOS = " << tempoMelhorIndiv;
   cout << "\n";
   cout << "\n MAIOR MAKESPAN = " << setfill(' ') << setw(4) <<
               piorIndivGeral.obterTempoFimJobs();
   cout << " : GERACAO = [" << geracaoPiorIndiv << "]";
   cout << "\n";
```

```
   prob.gravarProblema(saida);

   if(!(fsaida=fopen(saida,"a+t")))
   erro.printErro("\nNao foi possivel gravar o arquivo %\n",saida);
   fprintf(fsaida,"\n MELHOR INDIVIDUO GERADO : \n");
   fclose(fsaida);

   melhorIndivGeral.gravarIndividuo(saida);

   if(!(fsaida=fopen(saida,"a+t")))
   erro.printErro("\nNao foi possivel gravar o arquivo %\n",saida);

   fprintf(fsaida,"\n");
   fprintf(fsaida,"\n POPULACAO INICIAL    = %-4d
INDIVIDUOS",pop.obterTamPopInicial());
   fprintf(fsaida,"\n POPULACAO ATUAL      = %-4d
INDIVIDUOS",pop.obterTamPop());
   fprintf(fsaida,"\n GERACOES PERCORRIDAS = %-4d GERACOES",nGeracao);
   fprintf(fsaida,"\n");
   fprintf(fsaida,"\n INDIVIDUO : GERACAO = %d - MAKESPAN = %gu - TEMPO =
%gs",geracaoMelhorIndiv,
     melhorIndivGeral.obterTempoFimJobs(),
     tempoMelhorIndiv );
   fprintf(fsaida,"\n");

   fclose(fsaida);

   indiv.deletar();
   auxIndiv.deletar();
   melhorIndivGeracao.deletar();
   melhorIndivGeral.deletar();

   pop.excluirTodosIndivs();

   cout << "\n\n ARQUIVO GERADO COM SUCESSO...\n\n RESULTADO : " << saida;
   cout << "\n DADOS     : " << teste;
   } // if(ch==13)
   else
   cout << "\n\n PROGRAMA CANCELADO.";
   cout << "\n\n TECLE <QUALQUER TECLA> PARA FINALIZAR.";

   getch();
   return (0);
 }
```

Anexo C. O Programa Job-Shop

Como entrada de dados para o algoritmo Job-Shop (JSP), iremos gerar instâncias testes que são muito comuns na literatura.

Instâncias para o Programa Job-Shop

Instâncias são problemas JSP gerados para que algum algoritmo de otimização possa utilizá-las como entrada de dados para obtenção de uma solução. A solução é obtida para o problema (instância) em questão.

Uma instância consiste de uma linha contendo o número de jobs e o número de máquinas e uma linha para cada job listando o número da máquina e o tempo de processamento para cada tarefa (operação) do job.

A máquina e o tempo de processamento atribuídos a cada operação de um job são definidos segundo uma semente randômica (uma semente para a máquina e outra semente para o tempo de processamento) e são dados que devem ser fornecidos pelo usuário para gerar uma instância, além do número de jobs e o número de máquinas para o problema. Cada instância vem acompanhada de sua semente de tempo e semente de máquina. As instâncias que utilizamos como teste, com suas sementes de tempo e máquina, podem ser encontradas no "Benchmarks for basic scheduling problems" elaborado por Taillard (1993) ou na sua home page http://www.idsia.ch/~eric/.

Assim, os seguintes dados devem ser fornecidos pelo usuário para gerar uma instância de um problema <numJobs>/<numMaq> JSP :

- Semente Tempo <Time_seed>
- Semente Máquina <Machine_seed>
- Número de Jobs <Jobs>
- Número de Máquinas <Machines>

Para obtenção destes dados, foi criado um arquivo de dados denominado DADO.dat que já possui estes campos, bastando apenas inserir os dados para gerar a instância do problema.

Veja a figura B.1, o arquivo DADO.dat que gera uma instância de um problema 10/5 JSP com semente de tempo <Time_seed> = 1166510396 e semente de máquina <Machine_seed> = 164000672.

```
! PROBLEMA JOB SHOP

 ! Gerar instancias do Problema JOB SHOP seguindo os dados fornecidos
 ! por Taillard, E. D.: "Benchmarks for basic scheduling problems",
 ! EJOR vol. 64, pp. 278-285, 1993

 ! Para gerar e exibir uma instancia do problema devera ser
 ! editado este arquivo contendo os dados do problema.
 !
 ! EDITAR OS CAMPOS: <Time_seed>,<Machine_seed>,<Jobs> e <Machines>.
 ! ----------------------------------------------------------------------------------------
 !                                DADOS DO PROBLEMA:
 ! ----------------------------------------------------------------------------------------
 ! Problema numJobs/numMaq JSP:

            Time_seed       = 1166510396
            Machine_seed    = 164000672
            Jobs            = 10
            Machines        = 5

 ! ----------------------------------------------------------------------------------------
 ! A instancia a ser gerada sera exibida no arquivo de nome
 ! escolhido pelo usuario quando executar o programa JOB SHOP.

 ! FIM
```

Figura B.1 – Arquivo DADO.dat com campos preenchidos.

A instância gerada a partir do arquivo DADO.dat é exibida logo a seguir no tópico Saída para o Programa Job-Shop.

Saída para o Programa Job-Shop

Como saída para o Programa Job-Shop, que utiliza o algoritmo genético híbrido para obtenção de boas soluções, é gerado dois arquivos.

- O arquivo agGRASP.dat : arquivo de saída contendo a instânica do problema Job-Shop e a melhor solução obtida para o problema.
- O arquivo tagGRASP.dat : o arquivo contendo uma lista de dados coletados durante a evolução dos indivíduos gerados a cada geração. A lista exibe o melhor makespan/geração.

- O arquivo agGRASP.dat :

PROBLEMA 10/5 JSP:

J1 - 3-54 4-34 2-61 5-02 1-09
J2 - 4-15 1-89 5-70 2-38 3-19
J3 - 3-28 2-87 1-95 4-34 5-07
J4 - 4-29 2-79 5-25 1-59 3-95
J5 - 2-39 3-66 1-89 4-91 5-53
J6 - 4-01 5-14 3-66 1-50 2-99
J7 - 5-46 4-11 3-57 2-68 1-67
J8 - 4-11 3-35 2-51 1-05 5-46
J9 - 2-27 1-46 4-28 3-40 5-64
J10 - 4-11 2-14 1-89 5-33 3-62

---------- MELHOR INDIVIDUO GERADO : ----------

INDIVIDUO 10/5 JSP:

M1 - 2-2 15 9-2 104 5-3 150 4-4 239 1-5 298 10-3 307 6-4 396 8-4 446 3-3 451 7-5 546
M2 - 9-1 0 5-1 27 4-2 66 1-3 145 2-4 206 10-2 244 3-2 258 8-3 345 7-4 396 6-5 464
M3 - 3-1 0 1-1 28 5-2 82 7-3 148 9-4 205 2-5 245 8-2 264 6-3 299 4-5 365 10-5 460
M4 - 2-1 0 4-1 15 8-1 44 7-2 55 6-1 66 10-1 67 1-2 82 9-3 150 5-4 239 3-4 546
M5 - 7-1 0 6-2 67 2-3 104 4-3 174 1-4 206 9-5 245 5-5 330 10-4 396 8-5 451 3-5 580

POPULACAO INICIAL = 15 INDIVIDUOS
POPULACAO ATUAL = 15 INDIVIDUOS
GERACOES PERCORRIDAS = 50 GERACOES

INDIVIDUO : GERACAO = 29 - MAKESPAN = 613u - TEMPO = 36.044s

O arquivo agGRASP exibe uma instância 10/5 JSP e a melhor solução gerada em 50 gerações percorridas pelo algoritmo genético híbrido. A melhor solução foi gerada na geração = 29 e possui tempo de finalização (makespan) = 613 u (unidades de tempo).

- O arquivo tagGRASP :

GERACAO [0] : MELHOR MAKESPAN = 655u : SEGUNDOS = 0.10989s

GERACAO [1] : MELHOR MAKESPAN = 666u : SEGUNDOS = 1.20879s

GERACAO [2] : MELHOR MAKESPAN = 666u : SEGUNDOS = 2.36264s

GERACAO [3] : MELHOR MAKESPAN = 666u : SEGUNDOS = 3.40659s

GERACAO [4] : MELHOR MAKESPAN = 666u : SEGUNDOS = 4.50549s

GERACAO [5] : MELHOR MAKESPAN = 666u : SEGUNDOS = 5.38462s

GERACAO [6] : MELHOR MAKESPAN = 627u : SEGUNDOS = 6.31868s

GERACAO [7] : MELHOR MAKESPAN = 627u : SEGUNDOS = 7.47253s

GERACAO [8] : MELHOR MAKESPAN = 655u : SEGUNDOS = 8.51648s

GERACAO [9] : MELHOR MAKESPAN = 666u : SEGUNDOS = 9.61538s

GERACAO [10] : MELHOR MAKESPAN = 633u : SEGUNDOS = 10.7692s

GERACAO [11] : MELHOR MAKESPAN = 633u : SEGUNDOS = 11.978s

GERACAO [12] : MELHOR MAKESPAN = 654u : SEGUNDOS = 13.1319s

GERACAO [13] : MELHOR MAKESPAN = 664u : SEGUNDOS = 14.2308s

GERACAO [14] : MELHOR MAKESPAN = 656u : SEGUNDOS = 15.4945s

GERACAO [15] : MELHOR MAKESPAN = 677u : SEGUNDOS = 18.2967s

GERACAO [16] : MELHOR MAKESPAN = 699u : SEGUNDOS = 19.7253s

GERACAO [17] : MELHOR MAKESPAN = 654u : SEGUNDOS = 20.989s

GERACAO [18] : MELHOR MAKESPAN = 654u : SEGUNDOS = 22.8571s

GERACAO [19] : MELHOR MAKESPAN = 654u : SEGUNDOS = 23.9011s

GERACAO [20] : MELHOR MAKESPAN = 630u : SEGUNDOS = 25.1099s

GERACAO [21] : MELHOR MAKESPAN = 654u : SEGUNDOS = 26.4835s

GERACAO [22] : MELHOR MAKESPAN = 654u : SEGUNDOS = 27.8571s

GERACAO [23] : MELHOR MAKESPAN = 654u : SEGUNDOS = 28.9011s

GERACAO [24] : MELHOR MAKESPAN = 624u : SEGUNDOS = 30.2198s

GERACAO [25] : MELHOR MAKESPAN = 654u : SEGUNDOS = 31.4286s

GERACAO [26] : MELHOR MAKESPAN = 654u : SEGUNDOS = 32.5824s

GERACAO [27] : MELHOR MAKESPAN = 624u : SEGUNDOS = 33.7363s

GERACAO [28] : MELHOR MAKESPAN = 624u : SEGUNDOS = 34.8901s

GERACAO [29] : MELHOR MAKESPAN = 613u : SEGUNDOS = 36.044s

GERACAO [30] : MELHOR MAKESPAN = 613u : SEGUNDOS = 37.2527s

GERACAO [31] : MELHOR MAKESPAN = 622u : SEGUNDOS = 38.5165s

GERACAO [32] : MELHOR MAKESPAN = 613u : SEGUNDOS = 39.8901s

GERACAO [33] : MELHOR MAKESPAN = 622u : SEGUNDOS = 41.2088s

GERACAO [34] : MELHOR MAKESPAN = 645u : SEGUNDOS = 42.5275s

GERACAO [35] : MELHOR MAKESPAN = 645u : SEGUNDOS = 43.8462s

GERACAO [36] : MELHOR MAKESPAN = 645u : SEGUNDOS = 45.1648s

GERACAO [37] : MELHOR MAKESPAN = 645u : SEGUNDOS = 46.5934s

GERACAO [38] : MELHOR MAKESPAN = 645u : SEGUNDOS = 48.022s

GERACAO [39] : MELHOR MAKESPAN = 645u : SEGUNDOS = 49.2857s

GERACAO [40] : MELHOR MAKESPAN = 645u : SEGUNDOS = 50.8242s

GERACAO [41] : MELHOR MAKESPAN = 645u : SEGUNDOS = 52.1978s

GERACAO [42] : MELHOR MAKESPAN = 630u : SEGUNDOS = 53.4615s

GERACAO [43] : MELHOR MAKESPAN = 630u : SEGUNDOS = 54.8901s

GERACAO [44] : MELHOR MAKESPAN = 630u : SEGUNDOS = 56.044s

GERACAO [45] : MELHOR MAKESPAN = 645u : SEGUNDOS = 57.3077s

GERACAO [46] : MELHOR MAKESPAN = 630u : SEGUNDOS = 58.4066s

GERACAO [47] : MELHOR MAKESPAN = 613u : SEGUNDOS = 59.5604s

GERACAO [48] : MELHOR MAKESPAN = 645u : SEGUNDOS = 60.8791s

GERACAO [49] : MELHOR MAKESPAN = 644u : SEGUNDOS = 62.2527s

GERACAO [50] : MELHOR MAKESPAN = 645u : SEGUNDOS = 63.4615s

MELHOR SOLUCAO : GERACAO [29] - MAKESPAN = 613u - SEGUNDOS = 36.044s

MAIOR MAKESPAN = 862u - GERACAO [14]

O arquivo tagGRASP exibe a evolução das melhores soluções geradas a cada geração, pelo algoritmo genético híbrido, durante a execução do programa Job-Shop. Veja que a pior solução foi gerada na geração = 14 com tempo de finalização (makespan) = 862 u (unidades de tempo).