UNIVERSIDADE ESTADUAL DE CAMPINAS
FACULDADE DE ENGENHARIA ELÉTRICA E DE COMPUTAÇÃO
DEPARTAMENTO DE ENGENHARIA DE SISTEMAS

# ALGORITMOS GENÉTICOS PARA MINIMIZAÇÃO DE *MAKESPAN* EM UM *FLOW SHOP* FLEXÍVEL

LUÍS HENRIQUE SACCHI

Orientador:
Prof. Dr. Vinícius Amaral Armentano

Tese apresentada à Faculdade de Engenharia Elétrica e de Computação da Universidade Estadual de Campinas -UNICAMP, como parte dos requisitos exigidos para a obtenção do título de MESTRE EM ENGENHARIA ELÉTRICA.

SETEMBRO 1997

Este exemplar corresponde a redação final da tese defendida por LUÍS HENRIQUE SACCHI e aprovada pela Comissão Julgada em 12 / 9 / 1997.
Vinícius A. Armentano
Orientador

Sa14a
32563/BC

UNICAMP
BIBLIOTECA CENTRAL

UNIDADE 30
N.º CHAMADA :
T/UNICAMP
Sa14a
V. Ex.
TOMBO BC/ 32563
PROC. 395/98
C [ ] D [X]
PREÇO R$ 11,00
DATA 6/03/98
N.º CPD

CM-00104992-3

FICHA CATALOGRÁFICA ELABORADA PELA
BIBLIOTECA DA ÁREA DE ENGENHARIA - BAE - UNICAMP

Sa14a

Sacchi, Luís Henrique
Algoritmos genéticos para minimização de *makespan* em um *flow shop* flexível / Luís Henrique Sacchi-- Campinas, SP: [s.n.], 1997.

Orientador: Vinicius Amaral Armentano.
Dissertação (mestrado) - Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. Heurística. 2. Pesquisa operacional. 3. Inteligência artificial. 4. Administração da produção - Modelos matemáticos. I. Armentano, Vinicius Amaral. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. III. Título.

À minha família e
à memória de Oswaldo Perina

*"Então, como podes orientar a tua mente retorcida, purificar teu coração e estar em harmonia com a atividade de todas as coisas que existem na natureza? Em primeiro lugar, deves* ***tornar teu*** *o coração de Deus. É um grande amor onipresente; encontra-se em todos os lugares e em todos os tempos do universo. No amor não existe discórdia. O amor não tem inimigos. Uma mente impregnada de discórdia, que pensa que os inimigos existem, não está em absoluto de acordo com a vontade de Deus."*

Morihei Uyeshiba

# AGRADECIMENTOS

A todos que colaboraram na realização deste trabalho e em especial

- a Deus pela vida que me deu.
- ao Professor Vinicius A. Armentano pelo incentivo à pesquisa e pela paciência que teve comigo.
- ao Walcir e à Luciana pela atenção e pela rede estável.
- à Débora pelo artigo do Polonês, que chegou em boa hora.
- à Franklina pelos favores urgentes e pelo asterisco do Latex.
- à Cíntia pelas discussões sobre grafos.
- à Renata pelas discussões e sugestões sobre algoritmos genéticos.
- aos meus pais pelo incentivo e pelo apoio financeiro.
- à Luciana Soares pela solidez do seu amor.
- ao Caratê pela força interior.
- a CAPES pelo apoio financeiro.

# RESUMO

Este trabalho aborda o problema de programação de tarefas no ambiente de produção *flow shop* flexível, também conhecido como *flow shop* com máquinas paralelas. Algoritmos genéticos são utilizados para minimizar o tempo de processamento de todas as tarefas, isto é, o *makespan.* Implementações clássicas, baseadas em conhecimento e híbridas são apresentadas. Os algoritmos genéticos são comparados com as principais heurísticas da literatura e com um limitante inferior. Estratégias de busca local também são analisadas.

- Palavras chave: programação da produção, *flow shop* flexível, *makespan,* algoritmos genéticos, heurística.

# ABSTRACT

This work addresses the scheduling of jobs in a flexible flow shop or flow shop with parallel machines. The problem of minimizing the makespan is tackled by genetic algorithms. Classical, knowledge based and hybrid implementations are presented. The genetic algorithms are compared with the main heuristics from the literature and also with a lower bound. Local search strategies are also analysed.

- Key words: production scheduling, flexible flow shop, makespan, genetic algorithms, heuristic.

# Conteúdo

# Lista de Figuras

# Lista de Tabelas

# INTRODUÇÃO

## A PROGRAMAÇÃO DA PRODUÇÃO

*Scheduling*[1] é uma forma de tomada de decisão que ocupa um papel crucial na produção. No cenário atual, o investimento em programação da produção pelas empresas tornou-se uma necessidade de sobrevivência no mercado. As empresas devem programar atividades de forma a utilizar os recursos disponíveis de maneira eficiente, a fim de minimizar custos e respeitar compromissos de entrega estabelecidos com os clientes, onde falhas resultam em multas contratuais e na perda de credibilidade.

A programação da produção passou a ser questionada seriamente na produção no começo do século XX com os trabalhos de Henry Gantt e outros pioneiros. Entretanto, vários anos se passaram para que as primeiras publicações aparecessem na literatura. Algumas dessas primeiras publicações apareceram na *Naval Research Logistics Quarterly* no início dos anos 50, contendo resultados de W. E. Smith, S. M. Johnson, e J. R. Jackson. Durante os anos 60 uma quantidade significativa de trabalho foi dispendida na obtenção de métodos ótimos como a programação dinâmica e *branch and bound.* Como o esforço computacional para alcançar a solução ótima em problemas de programação da produção é muito grande e o tempo para tomada de decisões é bastante curto, podendo ser o período de um dia ou de um turno, os pesquisadores têm se concentrado no desenvolvimento e aperfeiçoamento de métodos heurísticos para resolução dos problemas. Em geral, esses métodos não garantem a obtenção de uma solução ótima. Mas, usando informações específicas sobre a estrutura do sistema de manufatura, pode-se definir heurísticas capazes de gerar soluções muito boas, e em questão de minutos.

[1] Para evitar o uso da palavra *scheduling,* ela será substituída por programação da produção ou simplesmente, programação, quando não comprometer o entendimento do texto.

## O PAPEL DA PROGRAMAÇÃO

A programação se preocupa com a alocação de recursos limitados para tarefas ao longo do tempo. É um processo de tomada de decisão que visa otimizar um ou mais objetivos. Os recursos e tarefas podem se apresentar de diferentes formas. Os recursos podem ser máquinas em uma fábrica, pistas em um aeroporto, unidades de processamento em um ambiente computacional, etc. As tarefas podem ser operações em um processo de produção, decolagens e aterrissagens em um aeroporto, execuções de programas, etc. Cada tarefa é caracterizada por um tempo de processamento, um nível de prioridade, data de entrega, etc. Os objetivos podem tomar também várias formas. Um exemplo de objetivo é minimizar o *makespan,* que corresponde ao tempo de término da última tarefa deixando o sistema. A minimização do *makespan* é equivalente à minimização do número médio de tarefas em processamento e do tempo médio ocioso de máquinas (French, 1982). Outros critérios são baseados em datas de entrega, tais como, atraso médio, atraso máximo e número de tarefas atrasadas.

## O LUGAR DA PROGRAMAÇÃO DA PRODUÇÃO DENTRO DE UMA ORGANIZAÇÃO

A função da programação da produção dentro de uma organização ou sistema envolve inter-relacionamento com muitas outras funções. Estes relacionamentos são, é claro, dependentes do sistema e podem diferir substancialmente de uma situação para outra. A seguir, uma descrição é dada de um ambiente de produção e o papel do processo de programação da produção dentro de tal ambiente. A Figura 0-1 traduz o texto abaixo de uma forma esquemática e hierarquizada.

Em sistemas de manufatura as ordens de sistema devem ser liberadas e traduzidas em tarefas. As tarefas freqüentemente devem ser processadas pelas máquinas em um ou mais centros de trabalho em uma dada ordem ou seqüência. As tarefas podem ter que esperar por máquinas que estão ocupadas, e interrupções podem ocorrer quando tarefas de alta prioridade chegam na máquina e devem ser processadas imediatamente. A programação detalhada das tarefas a serem processadas em um sistema de produção é necessária para manter a eficiência e o controle das operações.

A programação da produção é afetada pelo processo de planejamento da produção, que trata com o planejamento de médio a longo prazo de toda a organização. Este processo deve

considerar níveis de estoque, previsões de demanda, e requisição de recursos para planejar em um nível mais alto a produção dos produtos envolvidos e a alocação de recursos de longo prazo. Decisões que são feitas por esta função do planejamento têm impacto na programação da produção. A programação da produção também recebe informações do controle de chão de fábrica. Eventos não esperados no chão de fábrica, tais como a quebra de uma máquina ou tempos de processamento superiores aos previstos, devem ser levados em conta, pois terão um impacto maior na programação da produção.

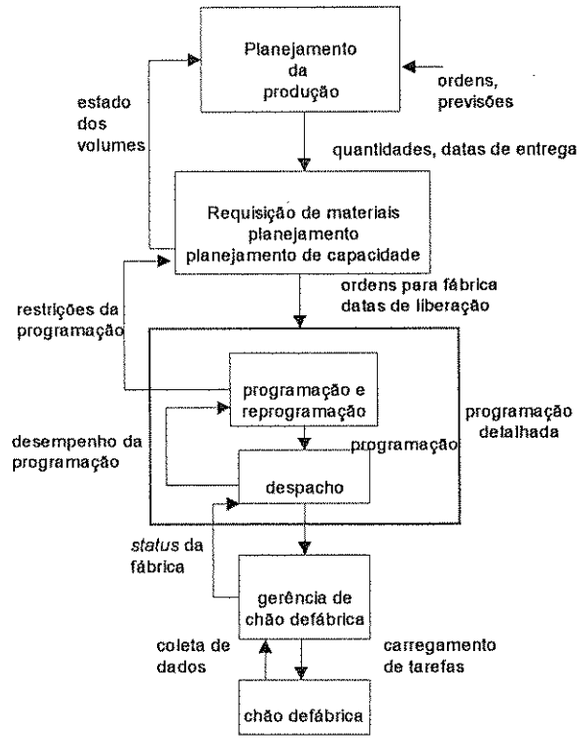



Figura 0-1: Diagrama de fluxo de informação em um sistema de manufaturas. Extraído de Pinedo (1995).

## FLOW SHOPS

Em muitos ambientes de produção e montagem um número de operações deve ser feito em cada tarefa. Em diversas situações, estas operações devem ser feitas em todas as tarefas na

mesma ordem, o que implica que as tarefas devem seguir a mesma rota, como mostra a Figura 0-2. As máquinas assumem uma disposição em série e o ambiente é referido como *flow shop.* O buffer entre máquinas sucessivas pode, às vezes, ser ilimitado por considerações práticas. Este é o caso quando os produtos processados são fisicamente pequenos (circuitos integrados, por exemplo); é fácil então armazenar grandes quantidades entre duas máquinas. Quando os produtos são fisicamente grandes (televisores, por exemplo), então o espaço de armazenamento entre duas máquinas sucessivas pode ter uma capacidade limitada. Esta dissertação considera apenas buffers ilimitados.

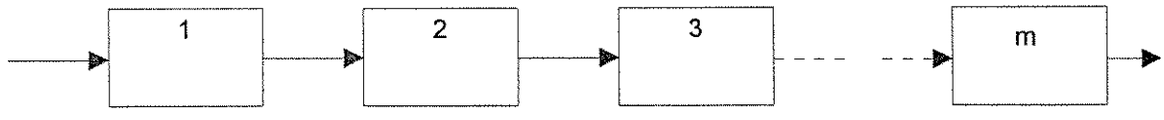



Figura 0-2: Ambiente *flow shop.*

## FLOW SHOPS DE PERMUTAÇÃO

Quando se procura uma solução ótima para uma instância[2] do problema de *flow shop* surge a questão: é suficiente determinar meramente uma permutação em que as tarefas atravessam todo o sistema? Fisicamente, é possível para uma tarefa "passar na frente" de outra tarefa esperando na fila por uma máquina que está ocupada. As máquinas não precisam operar de acordo com o princípio: *primeiro a chegar - primeiro a ser servido,* e a seqüência em que as tarefas passam pelas máquinas pode mudar de uma máquina para outra. A troca da seqüência de tarefas esperando na fila entre duas máquinas pode, às vezes, resultar em um programa de melhor qualidade.

Encontrar uma programação ótima quando alterações na seqüência são permitidas, é significativamente mais difícil do que encontrar uma programação ótima quando não o são. *Flow shops* nos quais mudanças na seqüência não são permitidas são chamados *flow shops* de permutação (a mesma permutação de tarefas é mantida através do *flow shop*).

[2]A palavra instância é usada no texto com o significado da palavra inglesa *instance.*

## FLOW SHOPS FLEXÍVEIS

O *flow shop* flexível ou *flow line* flexível, é considerado uma extensão de dois ambientes clássicos de manufatura, que são o *flow shop* e o de máquinas paralelas. Há um conjunto de tarefas e um conjunto de estágios de processamento, cada um contendo um conjunto de máquinas paralelas idênticas, como mostrado na Figura 0-3. Uma tarefa consiste de uma seqüência de operações processadas em estágios sucessivos, e todas as tarefas passam pelos estágios na mesma ordem (indo do primeiro ao último). Em um estágio, uma tarefa pode ser processada em qualquer máquina.

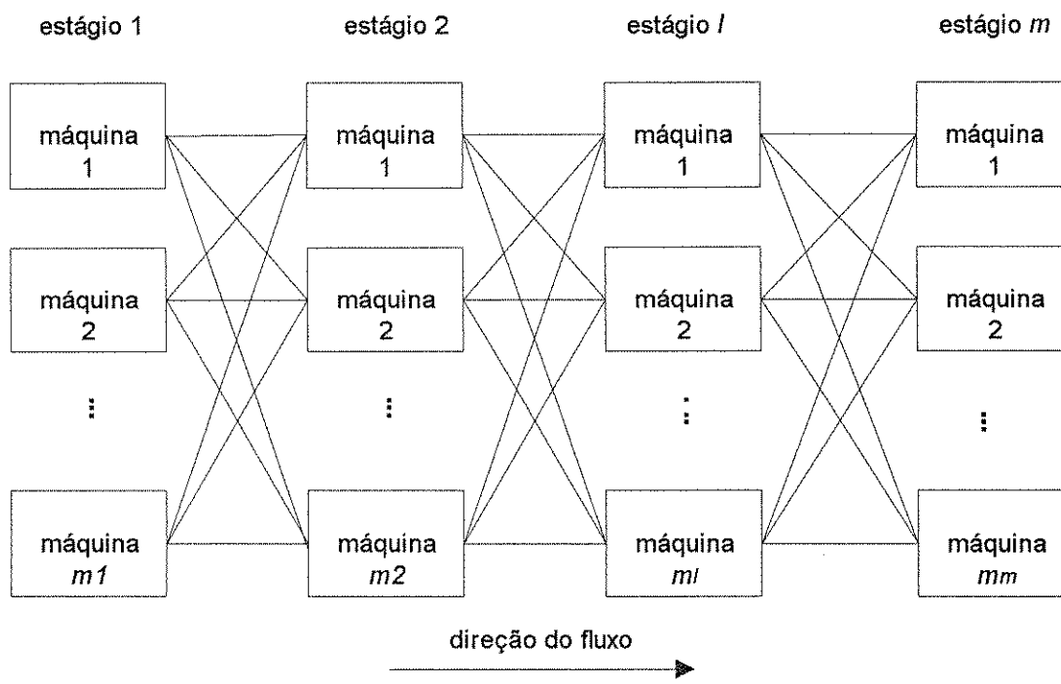



Figura 0-3: Estrutura do *flow shop* flexível.

Cada máquina pode processar no máximo uma tarefa por vez, e cada tarefa é processada no máximo em uma máquina por vez.

O objetivo deste trabalho é aplicar algoritmos genéticos ao ambiente de *flow shop* flexível, para se obter soluções de boa qualidade quando o critério de otimização for o *makespan*.

Quando se fala em meta-heurísticas, pode-se considerar aquelas que exploram o espaço de

solução de forma pontual. Uma vizinhança é determinada a cada iteração e uma solução única é escolhida desta vizinhança. Este tipo de varredura do espaço de busca gera um caminho de soluções, obtido da transição de uma solução para outra. Fazem parte desta subclasse de meta-heurísticas, a busca tabu (Glover, 1989 e Glover, 1994) e *simulated annealing* (Kirkpatrick, 1983 e Eglese, 1990).

Em contraposição a esses métodos, os algoritmos genéticos fazem parte de outra classe de meta-heurísticas, aquela em que os métodos são capazes de explorar várias regiões do espaço de busca simultaneamente. Ao longo das iterações, não se constrói um caminho único de busca pois novas soluções são obtidas através de combinações de soluções anteriores.

## APLICAÇÕES DO PROBLEMA

As aplicações deste tipo de problema ocorrem mais freqüentemente do que pode-se imaginar. Muitos ambientes de manufatura de grandes volumes têm vários *flow shops* separados, porém similares, e as máquinas em tais processos de produção podem ser intercambiadas em cada estágio. Salvador (1973) reconheceu o problema no processo químico de polimerização em indústrias petroquímicas, onde há várias instalações idênticas, cada uma podendo ser considerada um *flow shop,* e as tarefas podem ser processadas praticamente em qualquer instalação em cada estágio de processamento. Em linhas de montagem nas quais mais de um produto pode ser produzido e cada estação de trabalho tem múltiplas máquinas, é também óbvia a aplicação do problema. De modo similar, a situação onde máquinas paralelas são adicionadas em um ou mais estágios do *flow shop* para aliviar a pressão nas máquinas gargalo, e/ou para aumentar a capacidade de produção, também pode ser vista como uma aplicação do problema.

Há situações análogas a sistemas de produção onde a similaridade com um *flow shop* flexível pode ser estabelecida. Considere, por exemplo, a execução de um programa em um computador. Os três passos de compilação, ligação e execução são realizados em uma seqüência fixa. Se há múltipas tarefas (programas de computador) requisitando os três processos, cada um com múltiplos processadores (softwares), o problema se enquadra em um *flow shop* flexível.

## ORGANIZAÇÃO DOS CAPÍTULOS

Esta dissertação está assim organizada. No capítulo 1, encontra-se a revisão bibliográfica e no capítulo 2 o problema é representado através de um grafo direcionado. O capítulo 3 descreve os fundamentos sobre algoritmos genéticos. As propostas de soluções implementadas são apresentadas no capítulo 4. O capítulo 5 relata os resultados computacionais e a conclusão é discutida no capítulo 6.

# Capítulo 1

# REVISÃO BIBLIOGRÁFICA

A literatura sobre *flow shop* flexível é relativamente escassa e muito trabalho ainda precisa ser feito. Estudos aprofundados fazem-se necessários para determinar métodos de resolução para problemas de médio a grande porte. Quase todos os trabalhos realizados utilizam o *makespan* como critério de otimização. Portanto, será apenas citada a função objetivo, quando outro critério de otimização for adotado.

Garey e Johnson (1978) provaram que o problema de otimizar o *makespan* no ambiente de máquinas paralelas é $\mathcal{NP}$-difícil. Garey, Johnson e Sethi (1976) provaram que o problema de otimizar o *makespan* no ambiente *flow shop* com mais de duas máquinas, é $\mathcal{NP}$-difícil. Com esses resultados, conclui-se que otimizar o *makespan* no ambiente *flow shop* flexível também é $\mathcal{NP}$-difícil. Existem poucos algoritmos ótimos na literatura e todos eles utilizam a técnica *branch & bound*. A grande dificuldade enfrentada por estes algoritmos está na qualidade relativamente pobre dos limitantes inferiores propostos. Além disso, estes algoritmos resolvem somente problemas de pequeno porte. Por estes motivos, a maior parte das pesquisas está centrada em métodos heurísticos.

## 1.1 MÉTODOS ÓTIMOS

Os primeiros a abordarem o problema foram Arthanari e Ramaswamy (1971), que propuseram um modelo de programação inteira mista e desenvolveram um algoritmo *branch & bound* em um *flow shop* flexível com dois estágios, com duas máquinas paralelas idênticas no

primeiro estágio, e uma máquina no segundo estágio. Outro pioneiro, Salvador (1973), sugeriu um algoritmo *branch & bound* para resolver o problema de *flow shop* flexível *no-wait* [1]. A solução é impraticável mesmo para um número moderado de tarefas. Trabalhos posteriores foram realizados por Brah e Hunsucker (1991) no desenvolvimento de um algoritmo de *branch & bound* mais geral. O algoritmo pode tratar vários estágios, qualquer número de máquinas paralelas idênticas por estágio e o buffer entre os estágios é considerado infinito. Entretanto, a resolução de um problema com 8 tarefas, 5 estágios e 2 ou 3 máquinas paralelas por estágio, consumiu mais de 12 horas em um IBM-XT. Gupta e Tunc (1991), propuseram uma heurística cuja solução foi utilizada como limitante superior no algoritmo *branch & bound* desenvolvido por Brah e Hunsucker (1991).

## 1.2 MÉTODOS HEURÍSTICOS CONSTRUTIVOS

Heurísticas construtivas são aquelas que constroem uma solução progressivamente. O algoritmo sempre começa com uma solução parcial nula, e a cada passo, um elemento é adicionado à solução parcial. No final do algoritmo, tem-se uma solução para o problema.

Todas as heurísticas desenvolvidas para o *flow shop* flexível podem ser classificadas em três grupos:

- **Seqüenciamento seguido de alocação:** primeiro o algoritmo encontra uma seqüência, considerando apenas um *flow shop* de permutação (com apenas uma máquina por estágio). Depois, o algoritmo faz a alocação para as máquinas paralelas em cada estágio, segundo algum critério aplicado à seqüência encontrada no primeiro passo.

- **Alocação seguida de seqüenciamento:** na primeira fase, o algoritmo determina quais máquinas serão responsáveis pelo processamento de cada tarefa. Na segunda fase, o algoritmo determina a seqüência de processamento das tarefas em cada máquina.

- **Alocação e seqüenciamento simultaneamente:** à medida que as tarefas passam a fazer parte do programa, ficam determinadas as alocações e a seqüências em cada uma das máquinas pelas quais as tarefas passam.

[1] Programas *no-wait* são aqueles cujas tarefas não podem esperar entre os estágios. A transição de um estágio para outro deve ser contínua.

### 1.2.1 MÉTODOS RESTRITOS A 2 ESTÁGIOS

**Seqüenciamento seguido de alocação**

Considerando 2 máquinas idênticas no primeiro estágio e apenas uma máquina no segundo, Gupta (1988), propôs fazer a programação das tarefas como na regra de Johnson (1954), usando uma máquina em cada estágio. As tarefas são então alocadas às máquinas do primeiro estágio de forma a minimizar o tempo ocioso no segundo estágio. Gupta e Tunc (1991) consideraram o trabalho de Gupta (1988) no contexto reverso: uma máquina no primeiro estágio e múltiplas máquinas no segundo estágio. Os testes computacionais desses dois trabalhos consideram até 100 tarefas.

Chen (1995) desenvolveu heurísticas para dois estágios, onde um estágio qualquer possui uma única máquina e o outro várias máquinas paralelas idênticas. Classes de heurísticas são estudadas e desempenhos de pior caso são estabelecidos. Testes computacionais envolvem até 8 máquinas paralelas e 100 tarefas. Sriskandarajah e Sethi (1989) estudaram o *flow shop* flexível para 2 estágios. O problema com 1 máquina no primeiro estágio, e um número maior que 2 máquinas no segundo estágio foi considerado, assim como um número arbitrário de máquinas nos dois estágios. Heurísticas e desempenho no pior caso foram estabelecidos. Os testes computacionais lidam com no máximo 10 máquinas paralelas e 100 tarefas.

Tratando número arbitrário de máquinas em cada estágio, Guinet *et al.* (1996) propuseram uma heurística que, inicialmente, seqüencia as tarefas de acordo com uma regra de prioridade. A seguir as tarefas são alocadas às máquinas disponíveis em cada estágio. Sete regras são descritas para a fase de seqüenciamento. Na segunda fase, a heurística aloca cada tarefa da seqüência à máquina com menor tempo de espera em cada estágio. Resultados computacionais são relatados para problemas envolvendo 300 tarefas e 4 máquinas por estágio.

### 1.2.2 MÉTODOS PARA MÚLTIPLOS ESTÁGIOS

**Alocação e seqüenciamento simultaneamente**

Sawik (1993) propôs uma heurística construtiva na qual a cada iteração a rota completa de uma tarefa ao longo do *flow shop* flexível é determinada. A seleção da tarefa e a rota a seguir no sistema são obtidas baseadas na programação parcial obtida até o momento. As

decisões em cada iteração são tomadas usando um procedimento de otimização local baseado na minimização do tempo total ocioso ao longo da rota da tarefa escolhida. O método considera buffer limitado. Sawik (1995), usa o mesmo método, porém considerando buffer zero. Chegam a ser considerados 5 estágios, 100 tarefas e até 4 máquinas paralelas por estágio.

Ding e Kittichartphayak (1994) desenvolveram um heurística similar à de Sawick (1993). Os testes tratam até 20 tarefas, 8 estágios, e um número aleatório entre 1 e 3 de máquinas paralelas.

Hunsucker e Shah (1994) aplicaram várias regras de prioridade ao *flow shop* flexível: LPT (*longest processing time)*, SPT (*shortest processing time)*, MWR (*most work remaining*), LWR (*least work remaining)*, FIFO (*first in first out)* e LIFO (*last in first out).* Além do *makespan*, os tempos de fluxo médio e máximo são utilizados como critérios de otimização. Em seu trabalho, limitam o número de tarefas que pode coexistir dentro do sistema. Até 16 estágios, 5 máquinas por estágio e 320 tarefas são tratados.

Guinet e Solomon (1996) desenvolveram uma heurística para vários estágios de processamento equivalente ao trabalho de Sawik (1993), além de estudarem o critério de máximo atraso. Os testes computacionais envolvem no máximo 30 tarefas, 5 estágios e 5 máquinas por estágio.

### Alocação seguida de seqüenciamento

Wittrock (1985 e 1988) realizou trabalhos desenvolvendo heurísticas. A solução é baseada na decomposição do problema em três subproblemas a serem resolvidos seqüencialmente:

- Alocação de máquinas (determinando quais tarefas visitam cada máquina individual em cada estágio)
- Seqüenciamento das tarefas (especificando a ordem em que as tarefas devem entrar na linha de processamento)
- Especificação dos tempos em que as tarefas devem entrar na linha de processamento.

A solução proposta por Wittrock (1985) é mais simples por considerar a programação de um pequeno conjunto de tarefas que é repetida de forma periódica. Ambos trabalhos tratam até 51 tarefas em 3 estágios de processamento que não possuem mais que três máquinas paralelas.

**Seqüenciamento seguido de alocação**

Ding e Kittichartphayak (1994), desenvolveram 2 variações da heurística de Campbell *et al.* (1970). Guinet e Solomon (1996) desenvolveram heurísticas para vários estágios de processamento semelhantes às propostas por Ding e Kittichartphayak (1994).

Kochhar e Morris (1987) realizaram trabalhos no desenvolvimento de heurísticas que consideram tempo de preparação das máquinas, buffers finitos e tempo de ociosidade nas máquinas. Elaboraram um simulador para resolver o problema. Primeiro, uma seqüência é determinada para entrar no simulador. Depois as tarefas caminham através dos estágios, segundo regras que tentam minimizar tempos de bloqueios, tamanho das filas nos buffers e tempo de máquina ociosa. A seqüência de entrada pode ser determinada por várias heurísticas, e depois é melhorada através de busca local. Um problema real é considerado, consistindo de 7 estágios, 45 tarefas e 16 máquinas ao todo.

Santos *et al.* (1996) aplicaram heurísticas desenvolvidas para o *flow shop* de permutação tradicional (uma máquina por estágio) no ambiente *flow shop* flexível. A seqüência gerada por cada heurística é aplicada na entrada do sistema. A partir daí, as tarefas percorrem o ambiente de acordo com a política *fifo*. Os maiores problemas tratados, possuem 5 estágios, 20 tarefas e 3 máquinas por estágio.

## 1.3 MÉTODOS META-HEURÍSTICOS

Apenas um trabalho utiliza metódos meta-heurísticos. Nowicki e Smutnicki (1995) aplicaram busca tabu ao *flow shop* flexível, obtendo resultados muito bons. Usam vizinhança do tipo inserção de tarefa na mesma máquina, e inserção de tarefa em qualquer outra máquina do mesmo estágio. Conhecimento para evitar movimentos de baixa qualidade é utilizado. Um algoritmo especial para calcular o *makespan* é aplicado quando a vizinhança de uma solução é percorrida. 300 tarefas chegam a ser tratadas, bem como 6 máquinas por estágio e até 10 estágios.

# Capítulo 2

# DEFINIÇÃO DO PROBLEMA

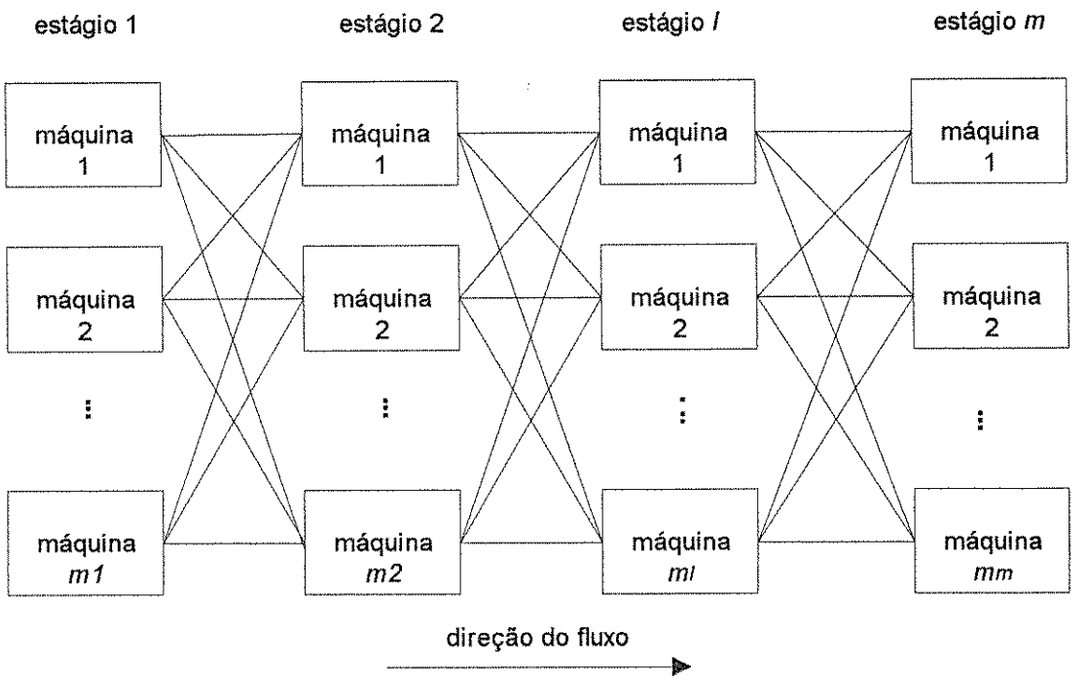



Figura 2-1: Estrutura do *flow shop* flexível.

O sistema de produção tem a arquitetura mostrada na Figura 2-1. Consiste de $m$ estágios de máquinas, onde cada estágio $l$ está equipado com $m_l \geq 1$ máquinas idênticas. O conjunto de estágios é representado por $M = \{1, \ldots, m\}$ e o conjunto de máquinas no estágio $l$ é denotado

Tabela 2.1: Tempos de processamento das tarefas.

| $l$ | $j$ | $p_{lj}$ | $l$ | $j$ | $p_{lj}$ | $l$ | $j$ | $p_{lj}$ |
|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 60 | 1 | 4 | 30 | 1 | 7 | 80 |
| 2 | 1 | 60 | 2 | 4 | 40 | 2 | 7 | 40 |
| 3 | 1 | 30 | 3 | 4 | 50 | 3 | 7 | 100 |
| 1 | 2 | 30 | 1 | 5 | 20 | | | |
| 2 | 2 | 10 | 2 | 5 | 90 | | | |
| 3 | 2 | 40 | 3 | 5 | 70 | | | |
| 1 | 3 | 40 | 1 | 6 | 30 | | | |
| 2 | 3 | 30 | 2 | 6 | 50 | | | |
| 3 | 3 | 40 | 3 | 6 | 30 | | | |

por $M_l = \{1, \ldots, m_l\}, l \in M$. Um conjunto de $n$ tarefas representado por $N = \{1, 2, \ldots, n\}$, deve ser processado no sistema. Todas as tarefas passam pelos estágios na mesma ordem $1, 2, \ldots, m$. A tarefa $j, j \in N$, consiste de uma seqüência de $m$ operações, cada uma delas correspondendo ao seu processamento no estágio $l$ durante um tempo de processamento não sujeito a interrupções $p_{lj} > 0$. Em cada estágio $l$, a tarefa $j$ pode ser processada em qualquer das $m_l$ máquinas. Cada máquina pode processar, no máximo, uma tarefa por vez, e cada tarefa é processada, no máximo, em uma máquina por vez. Um programa factível é definido como uma coleção de pares $(i_{lj}, C_{lj}), j \in N, l \in M$, onde $i_{lj} \in M_l$ é a máquina que processa a tarefa $j$ no estágio $l$, e $C_{lj} \geq 0$ é o instante de término da tarefa $j$ no estágio $l$ (na máquina $i_{lj}$), tal que as restrições acima são satisfeitas. O problema consiste em encontrar um programa factível que minimize o *makespan*, $\max_{j \in N} C_{mj}$.

Para ilustrar o problema, considere um exemplo com $m = 3$ estágios, 2 máquinas no primeiro estágio, 2 no segundo e 3 no terceiro estágio, isto é, $m_1 = m_2 = 2$, $m_3 = 3$. Um conjunto de $n = 7$ tarefas, com tempos de processamento dados na Tabela 2.1, deve ser processado no sistema.

Um programa factível para a instância é mostrado no diagrama de Gantt da Figura 2-2. Por exemplo, a tarefa 1 no estágio 2 é processada na máquina 1 e é completada no tempo 190,

isto é $i_{21} = 1, C_{21} = 190$. O *makespan* é 310.

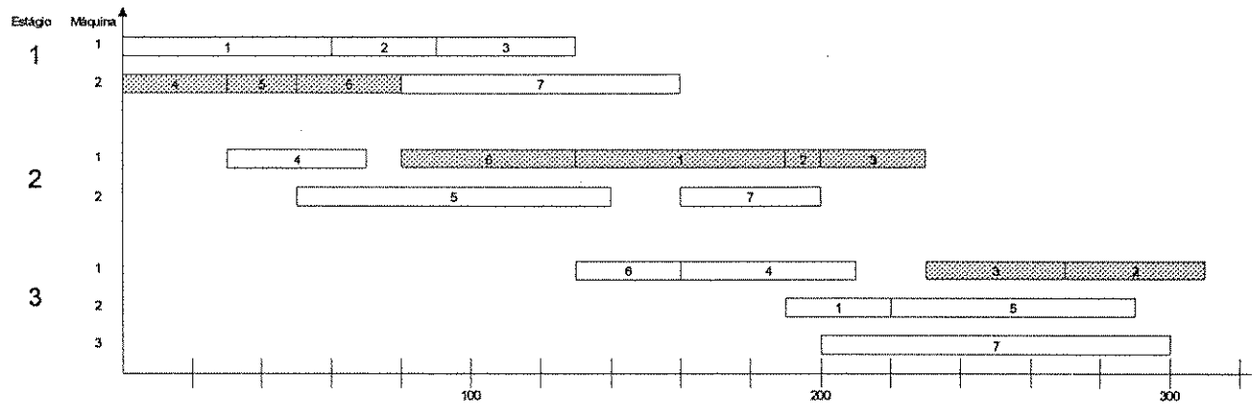



Figura 2-2: Diagrama de Gantt para o exemplo.

A seguir, o problema é representado através de um grafo. Com este propósito são introduzidas duas novas noções: grupos e ordem de processamento. Um grupo é um subconjunto de tarefas designado a uma máquina em um estágio. Em cada estágio $l$, o conjunto de tarefas $N$ é particionado em $m_l$ grupos $N_{li} \subset N, i \in M_l; n_{li} = |N_{li}|; i \in M_l, l \in M$.

A ordem de processamento das tarefas do grupo $N_{li}$ pode ser expressa por uma permutação $\pi_{li} = (\pi_{li}(1), \ldots \pi_{li}(n_{li})) \in \mathcal{P}(N_{li})$, onde $\pi_{li}(k)$ denota o elemento de $N_{li}$ que ocupa a posição $k$ em $\pi_{li}$, e $\mathcal{P}(N_{li})$ é o conjunto de todas as permutações do conjunto $N_{li}$. Assim, o processamento das tarefas no estágio $l$ pode ser completamente determinado pelo conjunto de $m_l$ permutações $\pi_l = \{\pi_{l1}, \ldots \pi_{lm_l}\}$. A ordem de processamento das tarefas é definida pela $m-tupla$ $\pi = (\pi_1, \ldots, \pi_m)$. O conjunto de todas as ordens de processamento é representado por:

$$\Pi = \{\pi = (\pi_1, \ldots, \pi_m) |\ \pi_l = \{\pi_{l1}, \ldots \pi_{lm_l}\}, (\pi_{li} \in \mathcal{P}(N_{li}),\ i \in M_l) \tag{2.1}$$

Para uma dada permutação $\pi \in \Pi$, os instantes de término das tarefas podem ser calculados em $\mathcal{O}(nm)$ usando a seguinte fórmula recursiva para $k = 1, \ldots, n_{li}; i = 1, \ldots, m_l; l = 1, \ldots, m$

$$C_{l,\pi_{li}(k)} = \max\{C_{l,\pi_{li}(k-1)}, C_{l-1,\pi_{li}(k)}\} + p_{l,\pi_{li}(k)} \tag{2.2}$$

onde, $\pi_{li}(0) = 0, i = 1, \ldots m_l$ ; $C_{l,0} = 0, l = 1, \ldots m$; $C_{0,j} = 0, j = 1, \ldots n$.

O valor de *makespan* associado a $\pi$ é denotado por $C_{\max}(\pi)$. Assim, $C_{\max}(\pi) = \max_{j \in N} C_{mj}$. Para determinar o programa factível a partir da ordem de processamento $\pi$, é preciso alocar grupos $N_{li}$, $i \in M_l$, a máquinas no centro $l$, $l \in M$. Desde que as máquinas no estágio são idênticas, a alocação pode ser feita de qualquer maneira. Assim, um programa factível $(i_{lj}, C_{lj}), j \in N, l \in M$, pode ser obtido da ordem de processamento $\pi$ da seguinte maneira: faz-se a alocação $i_{lj} \leftarrow i$, onde $i$ é tal que $i \in M_l$, $j \in N_{li}$, e encontra-se $C_{lj}$ através de (2.2). É claro que qualquer $\pi$ representa $\Pi_{l=1}^{m}(m_l)!$ programas factíveis com o mesmo valor de *makespan* para as várias alocações de grupos a máquinas. Finalmente, o problema pode ser estabelecido como encontrar a permutação $\pi \in \Pi$ que minimize $C_{\max}(\pi)$.

Para o exemplo dado são criados dois grupos $N_{11} = \{1, 2, 3\}$ e $N_{12} = \{4, 5, 6, 7\}$ no estágio 1; dois grupos $N_{21} = \{1, 2, 3, 4, 6\}$, $N_{22} = \{5, 7\}$ no estágio 2; e três grupos $N_{31} = \{2, 3, 4, 6\}$, $N_{32} = \{1, 5\}$, $N_{33} = \{7\}$ no estágio 3. Os grupos são processados nas ordens dadas por $\pi_{11} = (1, 2, 3)$, $\pi_{12} = (4, 5, 6, 7)$, $\pi_{21} = (4, 6, 1, 2, 3)$, $\pi_{22} = (5, 7)$, $\pi_{31} = (6, 4, 3, 2)$, $\pi_{32} = (1, 5)$, $\pi_{33} = (7)$, respectivamente. A ordem de processamento das tarefas é $\pi = (\pi_1, \pi_2, \pi_3)$, onde $\pi_1 = \{\pi_{11}, \pi_{12}\}$, $\pi_2 = \{\pi_{21}, \pi_{22}\}$, $\pi_3 = \{\pi_{31}, \pi_{32}, \pi_{33}\}$. Esta ordem de processamento gera $2! \cdot 2! \cdot 3! = 24$ programas factíveis com o mesmo valor de *makespan*. Um destes programas é mostrado na Figura 2-2.

É conveniente fazer a análise através de um grafo associado a uma ordem de processamento estabelecida. Este grafo foi definido por Nowicki e Smutnicki. Para uma dada permutação $\pi \in \Pi$ cria-se o grafo $G(\pi) = (O, E^* \bigcup E(\pi))$ com um conjunto de nós $O = M \times N$ e um conjunto de arcos $E^* \bigcup E(\pi)$, onde

$$E^* = \bigcup_{j=1}^{n} \bigcup_{l=1}^{m-1} \{((l, j), (l+1, j))\},$$

e

$$E(\pi) = \bigcup_{l=1}^{m} \bigcup_{i=1}^{m_l} \bigcup_{k=1}^{n_{li}-1} \{((l, \pi_{li}(k)), (l, \pi_{li}(k+1)))\}.$$

Os arcos do conjunto $E^*$ representam as rotas das tarefas pelos estágios, enquanto os arcos do conjunto $E(\pi)$ representam a ordem de processamento dos grupos. Cada nó $(l, j) \in O$ representa a $l-ésima$ operação da tarefa $j$ e tem peso $p_{lj}$. Todo arco tem peso zero. O instante de término da tarefa $C_{lj}$ é encontrado através de (2.2) e é igual ao comprimento do maior

caminho até o nó $(l, j)$, incluindo $(p_{lj})$ em $G(\pi)$. O *makespan* $C_{\max}(\pi)$ é igual ao comprimento do maior caminho (caminho crítico) em $G(\pi)$.

O grafo $G(\pi)$ para a ordem de processamento $\pi$ do exemplo anterior é mostrado na figura 2-3. Os arcos de $E^*$ são pontilhado, e os arcos de $E(\pi)$ são desenhados com traçado contínuo. O caminho crítico está hachurado nas Figuras 2-2 e 2-3.

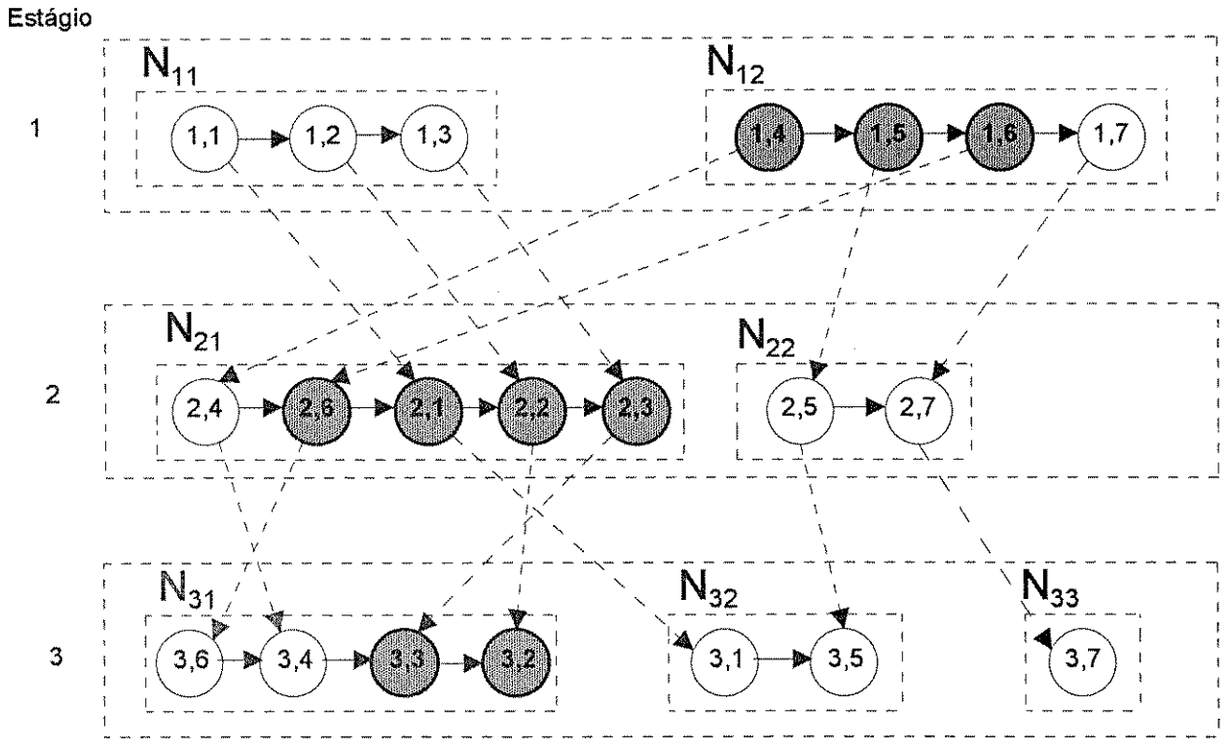



Figura 2-3: O grafo $G(\pi)$ do exemplo, com o caminho crítico em círculos hachurados.

A seqüência de nós do caminho crítico é obtida dos instantes de término das tarefas $C_{lj}$, $l \in M$, $j \in N$ determinados pela equação (2.2), ao se percorrer regressivamente a partir do nó final, os nós responsáveis pelo máximo caminho. O caminho crítico passa pelos nós associados aos estágios $1, 2, \ldots, m$ nesta ordem. No estágio $l$, o caminho crítico passa pelos nós correspondentes às tarefas de um único grupo $N_{lh_l}$, indo consecutivamente da tarefa na posição $e_l$ até a tarefa na posição $f_l$ $(1 \leq e_l \leq f_l \leq n_{lh_l})$ na permutação $\pi_{lh_l}$, $l \in M$. Assim, o caminho

crítico pode ser escrito como a seqüência de nós:

$$(1, \pi_{1h_1}(e_1)), (1, \pi_{1h_1}(e_1+1)), \ldots, (1, \pi_{1h_1}(f_1)),$$
$$(2, \pi_{2h_2}(e_2)), (2, \pi_{2h_2}(e_2+1)), \ldots, (2, \pi_{2h_2}(f_2)), \ldots,$$
$$(m, \pi_{mh_m}(e_m)), (m, \pi_{mh_m}(e_m+1)), \ldots, (m, \pi_{mh_m}(f_m))$$

onde $\pi_{lh_l}(f_l) = \pi_{l+1,h_{l+1}}(e_{l+1})$ para $l = 1, \ldots, m-1$. Claramente, $e_1 = 1$, $f_m = n_{mh_m}$. A seqüência de tarefas

$$B_l = (\pi_{lh_l}(e_l), \pi_{lh_l}(e_l+1), \ldots, \pi_{lh_l}(f_l))$$

associadas com o subgrafo $(l, \pi_{lh_l}(e_l), (l, \pi_{lh_l}(e_l+1), \ldots, \pi_{lh_l}(f_l)$ será chamada de um bloco de tarefas no estágio $l$. Por $b_l$, denota-se o número de tarefas no bloco $B_l$; claramente $b_l = f_l - e_l + 1$, $l \in M$. Nota-se que a última tarefa em $B_l$ é simultaneamente a primeira em $B_{l+1}$, $l = 1, \ldots, m-1$. Usando a notação de bloco, pode-se reescrever a fórmula de $C_{\max}(\pi)$ na forma de $C_{\max}(\pi) = \sum_{l \in M} \sum_{j \in B_l} p_{lj}$.

Para o grafo $G(\pi)$ do exemplo, o caminho crítico está marcado na Figura 2-3 com nós hachurados, bem como as respectivas operações na figura 2-2. Existem os blocos $B_1 = (4, 5, 6)$, $B_2 = (6, 1, 2, 3)$ e $B_3 = (3, 2)$. As tarefas do bloco $B_1$ pertencem ao grupo $N_{12}$ ($h_1 = 2$), as do bloco $B_2$ a $N_{21}$ ($h_2 = 1$), e as do bloco $B_3$ a $N_{31}$ ($h_3 = 1$). Assim, tem-se $e_1 = 1$, $f_1 = 3$, $e_2 = 2$, $f_2 = 5$, $e_3 = 3$, $f_3 = 4$.

## 2.1 LIMITANTES INFERIORES PARA O FLOW SHOP FLEXÍVEL

Um limitante inferior para o *makespan* foi estabelecido por Sawik (1988):

$$LI1C_{\max} = \max_{1 \le l \le m} \left\{ \sum_{j=1}^{n} \frac{p_{lj}}{m_l} + \sum_{l' \neq l} \min_{j \in N} \{p_{l'j}\} \right\} \tag{2.3}$$

Observe em (2.3) que o primeiro termo vem da idéia da ocupação média de cada máquina em um estágio, e o segundo vem do tempo mínimo gasto nos demais estágios.

Santos e Hunsucker (1995) propuseram um novo limitante inferior para o *flow shop* flexível:

$$LI2C_{\max} = \max_{1 \le l \le m} \left\{ \frac{1}{m_l} \left( \sum_{y=1}^{m_l} LSA(y,l) + \sum_{j=1}^{n} p_{lj} + \sum_{y=1}^{m_l} RSA(y,l) \right) \right\} \tag{2.4}$$

Seja $RS(j,l)$ a soma dos tempos de processamento a partir do estágio $l$+1 até o último, para a tarefa

$$RS(j,l) = \begin{cases} \sum_{l'=l+1}^{m} p_{l'j} & \text{se } l < m \\ 0 & \text{se } l = m \end{cases}$$

e $LS(j,l)$ a soma dos tempos de processamento do primeiro estágio, até o estágio $l-1$ para a tarefa j:

$$LS(j,l) = \begin{cases} \sum_{l'=1}^{l-1} p_{l'j} & \text{se } l > 1 \\ 0 & \text{se } l = 1 \end{cases}$$

Para um estágio $l$, $RSA(l)$ é a lista ordenada em ordem ascendente dos valores $RS(j,l)$ tomados sobre todas as tarefas. $RSA(1,l)$ é o primeiro valor nesta lista e corresponde ao $RS(k,l)$ para algum número particular de tarefa $k$.

$$\begin{aligned} RSA(l) &= \{RSA(1,l), RSA(2,l), \ldots, RSA(n,l)\} \\ \text{onde } RSA(1,l) &\le RSA(2,l) \le \ldots \le RSA(n,l) \end{aligned}$$

A lista ordenada $LSA(l)$ é definida de forma similar:

$$\begin{aligned} LSA(l) &= \{LSA(1,l), LSA(2,l), \ldots, LSA(n,l)\} \\ \text{onde } LSA(1,l) &\le LSA(2,l) \le \ldots \le LSA(n,l) \end{aligned}$$

O limitante inferior (2.4) pode ser explicado da seguinte forma. Num programa qualquer, cada máquina $i$, em cada estágio $l$, processa um grupo de tarefas $N_{li}$. O instante máximo de término das tarefas em $N_{li}$ não pode ser menor que o instante de término da última tarefa processada pela máquina $i$. Em outras palavras, o instante de término da última tarefa a ser processada pela máquina $i$ estabelece um limitante baseado em máquina nas tarefas em $N_{li}$.

Antes da última tarefa processada pela máquina $i$ no estágio $l$ deixar o *flow shop,* a primeira tarefa de $N_{li}$ deve chegar nesta máquina, todas as tarefas em $N_{li}$ devem ser processadas pela

máquina $i$, e então a última tarefa a ser processada pela máquina $i$ deve ser processada nos estágios restantes. Como não se sabe *a priori* que tarefas são alocadas a cada grupo $N_{li}$, não se pode determinar o máximo limitante baseado em máquinas. Entretanto, sabe-se que este máximo deve ser maior ou igual à média sobre os $m_l$ limitantes sobre máquinas deste estágio. A demonstração desse resultado pode ser encontrada em (Santos e Hunsucker, 1995). Demonstra-se neste trabalho que o limitante inferior $LI2C_{\max}$ é superior ao limitante inferior $LI1C_{\max}$.

Seja $l''$ um dado estágio. Então de (2.3):

$$\begin{aligned} & \sum_{j=1}^{n} \frac{p_{l''j}}{m_{l''}} + \sum_{l' \neq l''} \min_{j \in N} \{p_{l'j}\} \\ = & \sum_{j=1}^{n} \frac{p_{l''j}}{m_{l''}} + \sum_{l'=1}^{l''-1} \min_{j \in N} \{p_{l'j}\} + \sum_{l'=l''+1}^{m} \min_{j \in N} \{p_{l'j}\}. \end{aligned}$$

Para o estágio $l''$, tem-se de (2.4):

$$\begin{aligned} & \frac{1}{m_{l''}} \left( \sum_{y=1}^{m_{l''}} LSA(y, l'') + \sum_{j=1}^{n} p_{l''j} + \sum_{y=1}^{m_{l''}} RSA(y, l'') \right) \\ = & \sum_{y=1}^{m_{l''}} \frac{LSA(y, l'')}{m_{l''}} + \sum_{j=1}^{n} \frac{p_{l''j}}{m_{l''}} + \sum_{y=1}^{m_{l''}} \frac{RSA(y, l'')}{m_{l''}} \end{aligned}$$

Como

$$LSA(y, l'') \geq \sum_{l'=1}^{l''-1} \min_{j \in N} \{p_{l'j}\} \, \forall y \in N \Rightarrow \sum_{y=1}^{m_{l''}} \frac{LSA(y, l'')}{m_{l''}} \geq \frac{\sum_{l'=1}^{l''-1} \min_{j \in N} \{p_{l'j}\}}{m_{l''}} \cdot m_{l''}$$

e

$$RSA(y, l'') \geq \sum_{l'=l''+1}^{m} \min_{j \in N} \{p_{l'j}\} \, \forall y \in N \Rightarrow \sum_{y=1}^{m_{l''}} \frac{RSA(y, l'')}{m_{l''}} \geq \frac{\sum_{l'=l''+1}^{m} \min_{j \in N} \{p_{l'j}\}}{m_{l''}} \cdot m_{l''}.$$

então

$$\frac{1}{m_{l''}} \left( \sum_{y=1}^{m_{l''}} LSA(y, l'') + \sum_{j=1}^{n} p_{l''j} + \sum_{y=1}^{m_{l''}} RSA(y, l'') \right) \geq \sum_{j=1}^{n} \frac{p_{l''j}}{ml''} + \sum_{l' \neq l''} \min_{j \in N} \{p_{l'j}\}$$

e portanto

$$LI2C_{\max} \geq LI1C_{\max}$$

## 2.2 O FLOW SHOP FLEXÍVEL E A DIFICULDADE EM SE ENCONTRAR A SOLUÇÃO ÓTIMA

É interessante calcular o número de programas existentes para um dada instância do problema, em função do número de estágios, do número de máquinas por estágio, e do número de tarefas, pois assim pode-se estimar o tempo necessário para encontrar a solução ótima através da enumeração completa das soluções possíveis. Para tanto é útil considerar o seguinte formato de armazenamento da informação de um estágio, ilustrado pelo seguinte exemplo. Considere um estágio $l$ com 3 máquinas paralelas ($m_l = 3$), e dez tarefas ($n = 10$) que devem ser processadas segundo a ordem:

- máquina 1 processa as tarefas 2, 3 e 7 nesta ordem, $N_{l1} = \{2, 3, 7\}$, $\pi_{l1} = \{2, 3, 7\}$.
- maquina 2 processa as tarefas 5 e 1 nesta ordem, $N_{l2} = \{5, 1\}, \pi_{l2} = \{5, 1\}$.
- maquina 3 processa as tarefas 8, 9, 10, 6 e 4 nesta ordem, $N_{l3} = \{8, 9, 10, 6, 4\}, \pi_{l3} = \{8, 9, 10, 6, 4\}$.

Para representar a informação , utilizam-se dois vetores:

- um vetor para armazenar a ordem de processamento das tarefas (**vetor de tarefas**),
- um vetor formado por um conjunto de apontadores indicando para cada máquina o conjunto de tarefas a ela alocado (**vetor de máquinas**).

O exemplo anterior é representado na Figura 2-4.

Com esta representação sabemos que a máquina 1 processa as tarefas 2,3 e 7, nesta ordem. A máquina 2 processa as tarefas 5 e 1, e a maquina 3 processa as tarefas 8, 9, 10, 6 e 4.

O conjunto formado pelos vetores de tarefas e de máquinas de um estágio codifica a informação de um estágio. Para codificar um *flow shop* flexível composto de $m$ estágios, são necessárias $m$ estruturas deste tipo.

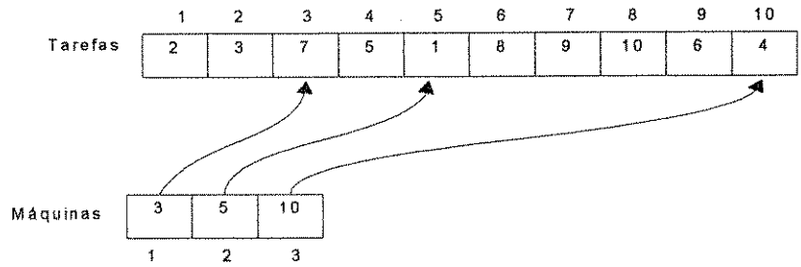



Figura 2-4: Informação de um estágio.

Aqui faz-se uma hipótese importante: uma máquina deve processar no mínimo uma tarefa. Observa-se que:

1. O **vetor de tarefas** é uma permutação no número de tarefas $n$.
2. O último elemento do **vetor de máquinas** deve apontar sempre para a *n_ésima* posição do vetor de tarefas, senão sobrariam tarefas que não estariam alocadas a nenhuma máquina.
3. Dois elementos do **vetor de máquinas** quaisquer, não podem apontar para uma mesma posição no **vetor de tarefas**; senão uma das máquinas não estaria processando nenhuma tarefa, o que fere a hipótese assumida.

Com estas considerações, mantendo-se o **vetor de máquinas** fixo, tem-se $n!$ programas diferentes. A partir da observação 2, ao manter-se o vetor de tarefas fixo, tem-se $\binom{n-1}{m_l-1}$ diferentes possibilidades de distribuir as $n-1$ tarefas restantes às $m_l-1$ máquinas restantes. Com o último elemento do vetor de máquinas fixado, sobram $\binom{n-1}{m_l-1}$ possibilidades de tomar $m_l-1$ números distintos dentre $n-1$. Assim se houver $m$ estágios, o número total de programas que podem ser gerados com esta representação será:

$$\Pi_{l=1}^{m}\left\{\binom{n-1}{m_l-1}n!\right\}$$

Como em cada estágio há $m_l$ máquinas idênticas, existem $m_l!$ programas idênticos, onde só mudam os índices das máquinas. Assim o número total de soluções distintas para o problema

é:

$$\Pi_{l=1}^{m} \left\{ \binom{n-1}{m_l - 1} \frac{n!}{m_l!} \right\}$$

Para se ter uma idéia da dificuldade em se tentar enumerar todas as soluções, é interessante analisar uma instância como exemplo: considerando $n = 20$, $l = 2$, e $m_1 = m_2 = 2$, tem-se a expressão:

$$\left\{ \binom{19}{1} \frac{20!}{2!} \right\}^2 = \left\{ \frac{19 \cdot 20!}{2} \right\}^2 \cong 5,342 \times 10^{38}$$

Supondo que um computador possa calcular o *makespan* de uma solução a cada $\mu s$, e levando-se em conta que um ano tem $3600 \times 24 \times 365,25 = 31.557.600s$, em um milhão de anos, será possível enumerar $3,15576 \times 10^{19}$ soluções. Isto implica que serão necessários $1,693 \times 10^{19}$ milhões de anos para enumerar todas as soluções! Algo impraticável.

O número 100!, como calculou F. Thoman (Tahan, 1987), contém 456.572 algarismos. Daí a necessidade de procurar métodos baseados em conhecimento, que sejam capazes de encontrar soluções de boa qualidade. Da impotência em lidar com a explosão combinatória, surgiram as meta-heurísticas como algoritmos genéticos e busca tabu, que são capazes de encontrar soluções muito boas, pesquisando de forma *inteligente* apenas uma pequena fração de um gigantesco espaço de soluções.

# Capítulo 3

# ALGORITMOS GENÉTICOS

## 3.1 INTRODUÇÃO

Os métodos tradicionais de otimização não são capazes de transcender ótimos locais. Os algoritmos genéticos são um tipo de meta-heurística capaz de superar esta limitação. Em geral, eles fornecem soluções de boa qualidade para problemas difíceis, em tempo computacional viável.

Algoritmos genéticos são algoritmos estocásticos cujos mecanismos de busca são baseados em fenômenos naturais: populações, herança genética e luta pela sobrevivência darwiniana.

A idéia por trás dos algorimos genéticos é tentar reproduzir o que a natureza faz. Considere, por exemplo, uma população de coelhos em um dado momento. Alguns deles são mais rápidos e mais espertos do que outros coelhos. Estes coelhos mais rápidos e mais espertos têm uma probabilidade menor de serem comidos por raposas, e assim mais deles sobrevivem para fazer o que os coelhos fazem melhor: mais coelhos. É claro que alguns dos coelhos mais lentos e mais tolos irão sobreviver por terem mais sorte. Esta população de coelhos sobreviventes inicia a procriação. O resultado da procriação é uma boa mistura de material genético de coelhos: alguns coelhos lentos procriam com coelhos rápidos, alguns coelhos rápidos com outros coelhos rápidos, alguns espertos com coelhos tolos, e assim por diante. Ocorrem também mutações no material genéticos de alguns coelhos. Os filhotes dos coelhos serão, em média, mais rápidos e mais espertos que os da população original, porque os pais mais rápidos e mais espertos sobreviveram às raposas. É uma coisa boa o fato das raposas estarem sofrendo um processo

similar, caso contrário, os coelhos se tornariam rápidos e espertos demais para as raposas os apanharem.

Os algoritmos genéticos usam um vocabulário emprestado da genética natural. Os termos mais usados são:

- **Indivíduos, genótipos** ou **estruturas**: em uma população, freqüentemente são também denominados *strings* ou cromossomos. Isto pode parecer um pouco confuso, pois cada célula de qualquer organismo de uma espécie carrega um certo número de cromossomos. O homem por exemplo, tem 46 cromossomos agrupados em pares, entretanto em algoritmos genéticos costuma-se usar indivíduos com apenas um cromossomo.

- **Genes**: unidades que formam os cromossomos. Os genes controlam a herança de uma ou várias características e situam-se em locais específicos dos cromossomos, que são chamados **loci** (posição na *string*). Qualquer característica dos indivíduos, como cor do cabelo, pode se manisfestar diferentemente. Diz-se que o gene tem vários estados, chamados **alelos.**

- **Fitness:** valor numérico associado a um cromossomo. Representa a sua adaptabilidade ao meio ambiente. Quanto maior o seu *fitness*, maior a sua probabilidade de sobreviver. Em problemas de otimização, representam o valor da função objetivo.

Cada indivíduo representa uma solução potencial para um problema. Um processo evolucionário realizado sobre uma população de indivíduos, corresponde à busca em um espaço de soluções potenciais. Tal busca requer balanceamento entre dois objetivos aparentemente conflitantes: intensificação da busca em regiões com boas soluções e diversificação no espaço de busca. *Hillclimbing* é um exemplo de uma estratégia que intensifica a busca na direção de máxima melhoria; por outro lado, a diversificação no espaço de busca é desconsiderada. Busca aleatória é um exemplo típico de uma estratégia que explora a diversificação no espaço de soluções, ignorando a intensificação em regiões promissoras do espaço. Algoritmos genéticos são uma classe de métodos de busca de propósito geral que conseguem conciliar a diversificação e a intensificação no espaço de busca.

Os algoritmos genéticos têm sido aplicados com sucesso em problemas de otimização como roteamento de fios, programação da produção, controle adaptativo, jogos, problemas de trans-

porte, problemas do caixeiro viajante, problemas de controle ótimo (Michalewicz, 1992 e Dowsland, 1996).

Os algoritmos genéticos foram desenvolvidos por John Holland, seus colegas, e alunos na universidade de Michigan. Os objetivos da pesquisa deles eram: (1) abstrair e explicar rigorosamente o processo adaptativo de sistemas naturais, e (2) projetar sistemas de software artificiais que utilizem os mecanismos dos sistemas naturais. A primeira monografia no assunto é devida a Holland (1975) e intitulada *Adaptation in Natural and Artificial Systems.* Fundamentos teóricos sobre algoritmos genéticos apresentados a seguir, são baseados nos trabalhos de Goldberg (1989), Michalevicz (1992), Beasley, Bull e Martin (1993a,b) e Whitley (1993). Uma introdução e perspectivas futuras de aplicação de algoritmos genéticos em pesquisa operacional podem ser encontradas no artigo recente de Dowsland (1996).

### 3.1.1 DIFERENÇAS ENTRE ALGORITMOS GENÉTICOS E MÉTODOS TRADICIONAIS

Os algoritmos genéticos diferem dos métodos de otimização mais comuns e procedimentos de busca em quatro aspectos:

1. Os algoritmos genéticos trabalham com uma codificação do conjunto de variáveis, não com as variáveis propriamente ditas.

2. Trabalham sobre uma população de soluções.

3. Utilizam informação da função objetivo, e não derivadas ou outros conhecimentos auxiliares.

4. Utilizam regras de transição probabilísticas, e não regras determinísticas.

Os algoritmos genéticos requerem que o conjunto das variáveis naturais do problema de otimização seja codificado em uma *string* de tamanho finito sobre algum alfabeto. Como exemplo, considere o problema de maximizar a função $f(x) = x^2$ no intervalo de inteiros $[0, 31]$. Com algoritmos genéticos, o primeiro passo para o processo de otimização é codificar a variável $x$ como uma *string* de tamanho fixo. Pode-se usar o alfabeto binário $\{0,1\}$, para construir a *string.* Em muitos métodos de otimização tais como o *simulated annealing* e a busca tabu, move-se

de um único ponto no espaço de decisão para o próximo usando alguma regra de transição para determinar o próximo ponto. Em contraste, os algoritmos genéticos trabalham a partir de um conjunto de pontos simultaneamente (uma população de *strings*), Um algoritmo genético começa com uma população de *strings* e depois disso gera sucessivas populações de *strings*. Uma população inicial para o problema do exemplo a ser otimizado pode ser formada por 4 *strings*, representadas por 01101, 11000, 01000 e10011.

Muitas técnicas de busca requerem muita informação auxiliar com o objetivo de trabalhar corretamente. Por exemplo, técnicas baseadas em gradiente precisam das derivadas com o objetivo de escalar o pico corrente. Em contraste, os algoritmos genéticos não necessitam de nenhuma dessas informações auxiliares. Para realizar uma busca eficaz, precisam apenas dos valores da função objetivo (*fitness*) associados a cada *string* individual. Usam regras de transição probabilísticas para guiar a busca e o uso de probabilidade não implica que o método seja uma busca simplesmente aleatória. As escolhas aleatórias são uma ferramenta para guiar o processo através de regiões do espaço de busca com probabilidade de melhoria. Os quatro aspectos mencionados anteriormente, contribuem para a robustez dos algoritmos genéticos.

### 3.1.2 UM ALGORITMO GENÉTICO SIMPLES

O funcionamento de um algoritmo genético simples é surpreendentemente trivial, envolvendo nada mais complexo do que copiar *strings* e trocar *strings* parciais. A explicação do porquê do funcionamento do processo é muito mais sutil. Dada a população inicial, é preciso definir um conjunto de operações simples que gere populações sucessivas. A qualidade dos indivíduos da população deve, na média, melhorar ao longo do tempo.

Um algoritmo genético que produz bons resultados em muitos problemas práticos é composto de três operadores descritos a seguir.

#### Reprodução

É o processo no qual *strings* individuais são copiadas de acordo com seus valores de função objetivo, $f$ (*fitness*). Intuitivamente, pode-se pensar na função $f$ como uma medida de ganho, ou utilidade que se deseja otimizar. Copiando *strings* de acordo com seus valores de *fitness*, significa que *strings* com maior valor têm maior probabilidade de contribuir para um ou mais

Tabela 3.1: *Fitness* de cada indivíduo da população inicial.

| No. | string | fitness | % do total | % acumulada |
|---|---|---|---|---|
| 1 | 01101 | 169 | 14,4 | 14,4 |
| 2 | 11000 | 576 | 49,2 | 63,6 |
| 3 | 01000 | 64 | 5,5 | 69,1 |
| 4 | 10011 | 361 | 30,9 | 100,0 |
| **Total** | | 1170 | 100,0 | ——— |

descendentes na próxima geração. Este operador, é claro, é uma versão artificial da **seleção natural** que simula a sobrevivência darwiniana da mais adaptada dentre as *strings*. Em populações naturais o *fitness* é determinado pela habilidade de sobreviver a predadores, pestes, e outros obstáculos da vida adulta. No algoritmo genético, a função objetivo é o árbitro que decide a vida ou morte de uma *string*. O operador de reprodução pode ser implementado de diferentes formas. Talvez a mais fácil seja criar uma roleta onde cada *string* existente na população tem um espaço na roleta proporcional ao seu *fitness*. A população do exemplo apresenta os valores de *fitness* listado na Tabela 3.1. Somando os valores de *fitness* das quatro *strings*, obtém-se 1170. A porcentagem do *fitness* de cada *string* em relação ao total da população também é mostrada na Tabela 3.1. A Figura 3-1 mostra a proporção do *fitness* de cada *string* na roleta. Para reproduzir, simplesmente roda-se a roleta, assim definida, quatro vezes. A *string* 1 tem probabilidade 0,144 de ser escolhida em cada giro da roleta. Cada vez que deseja-se uma outra *string* para reprodução, gira-se a roleta. Girar a roleta significa gerar um número aleatório $r \in [0, 1]$ e procurar o intervalo em que se encaixa na coluna **% acumulada**, presente na Tabela 3.1. Por exemplo, se $r = 0,620$, a *string* 2 é selecionada pois $0,144 < 0,620 < 0,636$. Uma réplica exata da *string* é construída e colocada numa população intermediária. Aqui o processo de seleção está sendo realizado. O processo de ir da população corrente para a próxima população constitui uma **geração** na execução de um algoritmo genético. A Figura 3-2 ilustra a formação da população intermediária.

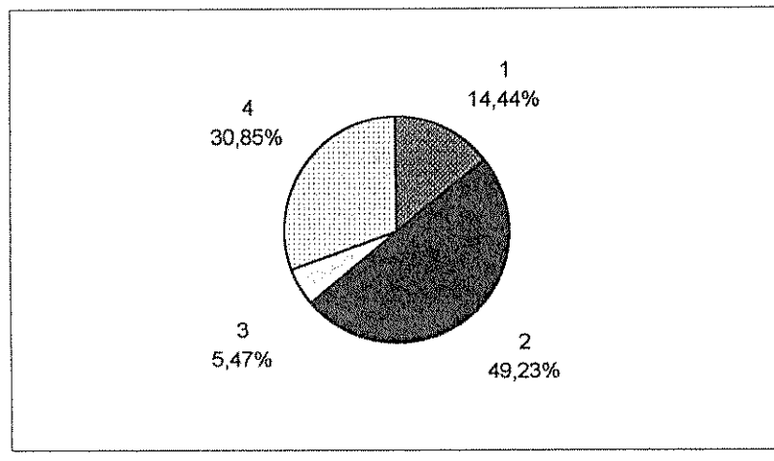



Figura 3-1: Roleta com divisão proporcional ao *fitness*.

### Crossover

O operador *crossover* é um mecanismo pelo qual a informação é passada entre os cromossomos. Isto é feito através de trocas parciais de *strings*. Há uma analogia óbvia aqui com sistemas naturais. Dois cromossomos pais, são combinados para gerar cromossomos filhos, com mostra a Figura 3-2. Os cromossomos filhos herdam características recebidas dos pais. A maioria, ou até mesmo todos cromossomos podem sofrer *crossover*. Se ambos os pais possuen *fitness* bom, há uma chance razoável de que pelo menos um deles possua *fitness* maior que o dos pais. Assim, o *crossover* provê um mecanismo pelo qual novos cromossomos são criados e também promove o aumento na qualidade dos cromossomos da população.

O exemplo a seguir ilustra um processo de *crossover* aplicado em *strings* definidas sobre um alfabeto binário. Primeiro, os membros da população intermediária são agrupados dois a dois de forma aleatória. Segundo, cada par de *strings* sofre *crossover* com probabilidade $p_c$, da seguinte forma. Gera-se um número $r \in [0, 1]$:

- Se $r > p_c$, as strings são copiadas integralmente na população intermediária.
- Se $r \leq p_c$, uma posição inteira $k$, ao longo da *string* é escolhida de forma aleatória a partir de uma distribuição uniforme no intervalo $[1, l-1]$, onde $l$ é o tamanho da *string*. Duas novas *strings* são criadas ao se trocar todos os caracteres entre as posições $k+1$ e $l$

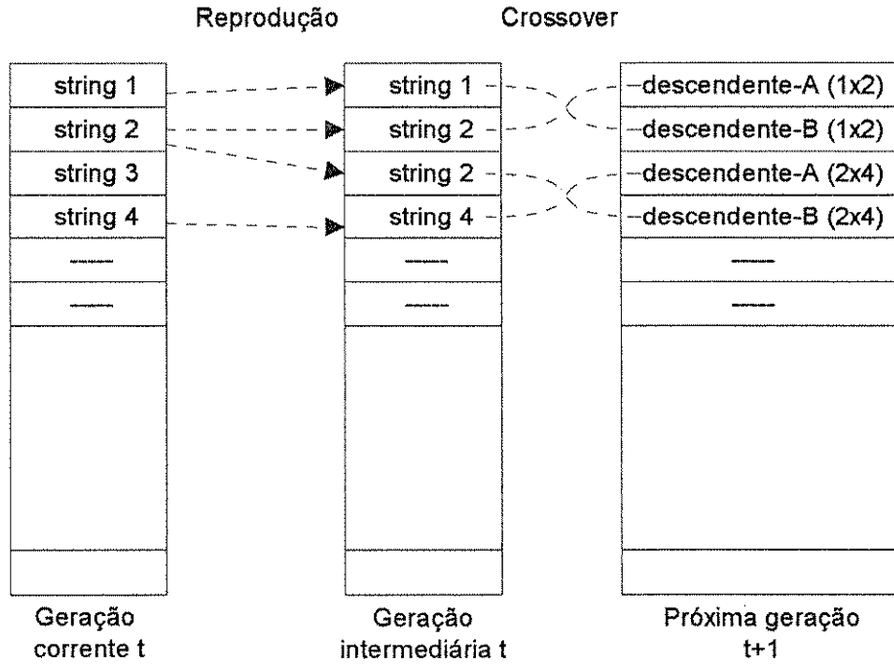



Figura 3-2: Uma geração de reprodução e *crossover*.

inclusive. Considere as *strings* $\mathbf{A}_1$e $\mathbf{A}_2$ do exemplo:

$$\begin{aligned} \mathbf{A}_1 &= 0\ 1\ 1\ 0\ |\ 1 \\ \mathbf{A}_2 &= 1\ 1\ 0\ 0\ |\ 0 \end{aligned}$$

Suponha que ao escolher um número aleatório entre 1 e 4, obtém-se $k = 4$, (indicado pelo símbolo separador |). O *crossover* resultante gera duas novas *strings* $\mathbf{A}'_1$ e $\mathbf{A}'_2$, integrantes da nova geração:

$$\begin{aligned} \mathbf{A}'_1 &= 0\ 1\ 1\ 0\ |\ 0 \\ \mathbf{A}'_2 &= 1\ 1\ 0\ 0\ |\ 1 \end{aligned}$$

### Mutação

Embora o *crossover* gere novas *strings,* ele o faz recombinando cromossomos existentes, sem introduzir informações novas na população. É necessário então, um outro operador cujo papel

seja introduzir informações nas *strings*. Este operador é a mutação, que tem duas funções:

- provê um mecanismo pelo qual informações que não estavam na população inicial sejam geradas.
- previne a estagnação da população.

Em sua forma mais simples, a mutação é a alteração aleatória do valor em alguma posição da *string*, com probabilidade $p_m$. Gera-se um número aleatório $r \in [0, 1]$ e se $r \leq p_m$, a mutação deve ocorrer. No alfabeto binário, significa simplesmente trocar o 1 pelo 0 e vice-versa. Vários autores alegam que a taxa de mutação deve ser baixa comparada com o *crossover*, no entanto outros autores consideram que a taxa de mutação deve ser tão alta quanto a taxa de *crossover*.

O *crossover* e a mutação são aplicados na população intermediária para criar a próxima população.

### 3.1.3 ESQUELETO DO ALGORITMO

O esqueleto do algoritmo genético proposto por Holland, é apresentado a seguir:

*Passo 1*

1. $t \leftarrow 0$, zera o contador de gerações.
2. Defina o tamanho da população ($P$).
3. Estabeleça as probabilidades de mutação ($p_m$) e *crossover* ($p_c$).
4. Inicialize a população $A(t)$. Todos os cromossomos são gerados aleatoriamente.
5. Ordene $A(t)$ de acordo com o *fitness* e construa a roleta de seleção.

*Passo 2*

1. Se a condição de término for satisfeita, vá para o *Passo 3*.
2. $t \leftarrow t + 1$.
3. Reproduza $P$ indivíduos de $A(t-1)$ aos pares usando a roleta.

4. Aplique o *crossover* e a mutação em cada par selecionado, segundo as probabilidades $p_c$ e $p_m$.

5. Introduza cada novo par na população $A(t)$.

6. Ordene $A(t)$ de acordo com o *fitness* e construa a roleta de seleção.

7. Volte para o *Passo 2.*

*Passo 3*

1. Fim.

## 3.2 TÓPICOS ADICIONAIS

### 3.2.1 USO DO FITNESS EM PROBLEMAS DE MINIMIZAÇÃO

Quando o problema é de maximização, a probabilidade de um indivíduo $x_i$ ser selecionado é:

$$p(x_i) = \frac{f(x_i)}{\sum_{j=1}^{P} f(x_j)}$$

onde $P$ é o tamanho da população. Note que $\sum_{i=1}^{P} p(x_i) = 1$. Quando o problema é de minimização, pode-se atribuir uma probabilidade $p'(x_i) = (1 - p(x_i))/(P-1)$:

$$p'(x_i) = \frac{\left(1 - \frac{f(x_i)}{\sum_{j=1}^{P} f(x_j)}\right)}{P-1} \tag{3.1}$$

É preciso dividir por $(P-1)$, para que $\sum_{i=1}^{P} p'(x_i) = 1$. O exemplo a seguir ilustra por que este mecanismo não se mostrou satisfatório.

Em um teste realizado com uma população de 400 indivíduos, na primeira geração, o melhor cromossomo possuia $f(400) = 6938$, o pior cromossomo possuia $f(1) = 7796$, e $\sum_{j=1}^{400} f(x_j) = 2962157$. Note que $\frac{7796}{6938} \cong 1,1237$. Por outro lado, $p'(1) \cong 0,002499669$, $p'(400) \cong 0,002500395$ e $\frac{p'(400)}{p'(1)} \cong 1,0003$. Para o algoritmo genético, o melhor indivíduo passa a ser apenas 0,03% melhor que o pior indivíduo da população. Não é possível que ocorra evolução na população a

partir da sobrevivência dos mais adaptados, pois todos os indivíduos possuem probabilidade de seleção muito parecidas. O algoritmo genético age como se fosse apenas uma busca aleatória.

Um outro mecanismo que apresentou bons resultados no problema de minimizar o *makespan* no ambiente de *flow shop* de permutação com uma máquina por estágio, proposto por Murata *et al.* (1996) foi adotado:

$$p'(x_i) = \frac{[f_M - f(x_i)]^2}{\sum_{j=1}^{P}[f_M - f(x_j)]^2} \tag{3.2}$$

onde (3.3)

$$f_M = \max_{j=1}^{P}\{f(x_j)\}$$

Note, que com este tipo de estratégia, a probabilidade do pior indivíduo ser selecionado é igual a zero. Para a mesma instância de teste, obteve-se $\frac{p'(400)}{p'(2)} \cong 9088,43$.

### 3.2.2 MECANISMOS DE CLASSIFICAÇÃO (RANK)

São mecanismos baseados em procedimentos de seleção (reprodução) não paramétricos e que ordenam a população de acordo com seu valor de função objetivo. A probabilidade de um indivíduo ser escolhido depende do seu valor de classificação. Estes métodos são baseados na crença de que a convergência prematura é devida à presença de superindivíduos que são muito melhores que o *fitness* médio da população. Estes superindivíduos geram um grande número de descendentes (devido ao tamanho constante da população), impedindo outros indivíduos de contribuírem com descendentes nas gerações seguintes.

Uma forma de criar um sistema de classificação proposto por Baker (1985), consiste em ordenar a população de acordo com o valor do *fitness*. Define-se um valor MAX como o limitante superior para o número de descendentes, e uma reta, de acordo com a equação 3.4, passando por MAX, é tomada de modo que a área sob a curva se iguale ao tamanho da população $P$. A Figura 3-3 ilustra a reta de classificação.

$$f(x) = \begin{cases} \frac{MAX}{b-1}(b-x),\ x \in [1,b] \\ 0, \text{ caso contrário.} \end{cases} \tag{3.4}$$

Os métodos de classificação, embora mostrem melhoria no comportamento dos algoritmos

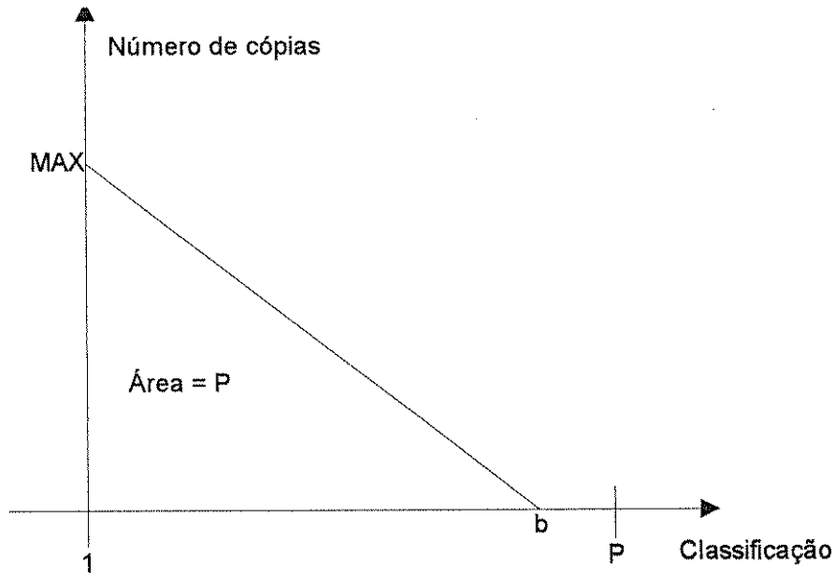



Figura 3-3: Curva de classificação, 1 é o melhor cromossomo, enquanto $P$ é o pior.

genéticos em alguns casos, apresentam a desvantagem de ignorar a informação sobre avaliações relativas de cromossomos diferentes, tratando todas as gerações uniformemente, independente dos valores relativos de *fitness*.

Nas implementações de algoritmos genéticos que usaram este mecanismo de classificação, não foi usada a roleta para selecionar os cromossomos pais. O critério utilizado foi tratar os cromossomos a partir do melhor classificado, ou seja o de número 1. Introduz-se $\max(\lfloor f(1) \rfloor^1, 1)$ cópias do primeiro cromossomo na população intermediária, e assim sucessivamente para os demais cromossomos, até que haja $P$ cromossomos na população intermediária. Para realizar o *crossover*, dois cromossomos são tomados aleatoriamente da população intermediária.

### 3.2.3 DETERMINAÇÃO DOS PARÂMETROS DOS ALGORITMOS GENÉTICOS

O tamanho da população e a influência da probabilidade dos operadores no desempenho do sistema têm sido estudados por muitos pesquisadores. Os resultados obtidos mostram que:

[1] $\lfloor x \rfloor$ é o maior número inteiro menor ou igual a $x$.

- A mutação tem um papel mais importante do que se admitia previamente (como um operador secundário).
- A estratégia de busca baseada em seleção e mutação apenas, pode ser um procedimento potente de busca, mesmo sem a assistência do *crossover*.

### 3.2.4 TÉCNICAS BASEADAS EM CONHECIMENTO

Enquanto muitos pesquisadores utilizam operadores tradicionais de *crossover* e mutação, outros têm defendido o uso de novos operadores para tarefas específicas, usando conhecimento. Isto torna cada algoritmo genético mais específico para a tarefa (menos robusto), mas pode melhorar o desempenho consideravelmente. Quando um algoritmo genético é projetado para dedicar-se a um problema do mundo real, e deve competir com outras técnicas de otimização, a incorporação de conhecimento é justificada.

Conhecimento específico do problema pode ser incorporado ao operador de *crossover* de forma eficiente, impedindo a criação de cromossomos obviamente ruins, ou aqueles que violam restrições do problema. Isto evita perda de tempo. Também é possível usar inicialização heurística da população, de forma que a busca comece com alguns pontos razoavelmente bons, ao contrário de um conjunto aleatório. Costuma-se também aplicar técnicas híbridas de algoritmos genéticos com outros métodos de otimização.

### 3.2.5 SUBSTITUIÇÃO EM REGIME PERMANENTE

A maioria dos trabalhos provem da proposta de Holland, no qual toda a população é substituída a cada geração. Entretanto, uma tendência mais recente tem favorecido substituições em regime permanente. Ela opera no extremo oposto, ou seja, em cada geração apenas uns poucos, tipicamente dois, indivíduos são substituídos.

Isto pode ser um modelo melhor do que acontece na natureza. Em espécies de vida curta, incluindo insetos, os pais põem os ovos, e morrem antes de seus descendentes chocarem. Mas em espécies de vida longa, incluindo os mamíferos, os descendentes e os pais vivem concorrentemente. Isto permite que os pais eduquem e ensinem seus descendentes, mas também aumenta a competição entre eles.

Em substituições em regime permanente, não deve-se apenas considerar como selecionar os dois indivíduos a serem os pais, mas também deve-se selecionar os indivíduos da população que serão mortos, para dar espaço para os descendentes. Vários esquemas são possíveis:

- Seleção dos pais de acordo com o *fitness*, e seleção aleatória dos que serão substituídos.
- Seleção aleatória dos pais, e seleção dos que serão substituídos inversamente ao *fitness*.
- Seleção dos pais e dos que serão substituídos de acordo com *fitness*/inverso do *fitness*.

No algoritmo genético com substituição em regime permanente, os novos descendentes estão imediatamente disponíveis assim que são criados. O primeiro a estudar este tipo de algoritmo genético foi Whitley (1989) com o algoritmo Genitor. O esquelto do algoritmo genético com substituição em regime permanente vem a seguir:

*Passo 1*

1. $t \leftarrow 0$, zera o contador de gerações.
2. Defina o tamanho da população ($P$).
3. Estabeleça as probabilidades de mutação ($p_m$) e *crossover* ($p_c$).
4. Inicialize a população $A(t)$. Todos os cromossomos são gerados aleatoriamente.
5. Ordene $A(t)$ de acordo com o *fitness* e construa a roleta de seleção.

*Passo 2*

1. Se a condição de término for satisfeita, vá para o *Passo 3.*
2. $t \leftarrow t + 1$.
3. Reproduza um par de pais de $A(t-1)$
4. Escolha de $A(t-1)$ um par de indivíduos que será removido.
5. Produza um par de filhos, aplicando o *crossover* e a mutação no par de pais selecionado, segundo as probabilidades $p_c$ e $p_m$.

6. Em $A(t-1)$, introduza o par de filhos gerados no lugar do par de indivíduos escolhidos para morrer, formando assim $A(t)$.

7. Ordene $A(t)$ de acordo com o *fitness* e construa a roleta de seleção.

8. Volte para o *Passo 2.*

*Passo 3*

1. Fim.

### 3.2.6 ALGORITMO GENÉTICO DE ACKLEY

Um tipo de algoritmo genético com substituição em regime permanente cujos resultados empíricos são encorajadores, foi proposto por Ackley (1987). Ackley utiliza seleção aleatória de ambos os pais, mas após o casamento, os descendentes substituem cromossomos existentes cujos *fitness* estão abaixo da média. A principal vantagem prática desta solução é que cromossomos acima da média com certeza sobrevivem. No algoritmo original de Holland, é possível descartar o melhor cromossomo quando passa-se de uma geração para outra. Isto é indesejável no contexto de um problema de otimização.

#### Mecanismo de seleção e remoção de Reeves

A idéia de Reeves (1995) é uma variação da proposta de Ackley (1987). Ao contrário dos dois pais serem selecionados aleatoriamente, um deles é selecionado de acordo com seu *fitness.* Um mecanismo de classificação é utilizado para realizar a seleção, um dos pais é escolhido aleatoriamente e o outro é selecionado através da roleta montada de acordo com a distribuição de probabilidades:

$$p([i]) = \frac{2i}{P(P+1)}. \tag{3.5}$$

Na expressão 3.5, $[i]$ é o $i-ésimo$ cromossomo em ordem ascendente de *fitness* (ordem decrescente de valor de *fitness*, se o problema for de minimização). Isto implica que o valor mediano tem probabilidade $1/P$ de ser selecionado, enquanto que o $P-ésimo$ (maior *fitness*) tem probabilidade $2/(P+1)$, quase o dobro do valor mediano.

O mecanismo de classificação é também utilizado na seleção do cromossomo a ser eliminado da população. Ackley (1987) escolhe cromossomos que estão abaixo da média do *fitness*. Reeves efetua uma escolha aleatória de dois cromossomos que estão abaixo do mediano.

## ESQUELETO DA IMPLEMENTAÇÃO DO ALGORITMO GENÉTICO DE ACKLEY + REEVES

*Passo 1*

1. $t \leftarrow 0$, zera o contador de gerações.
2. Defina o tamanho da população ($P$).
3. Defina o limite máximo de casamentos ($Cas$)
4. Estabeleça as probabilidades de mutação ($p_m$) e *crossover* ($p_c$).
5. Inicialize a população $A(t)$. Todos os cromossomos são gerados aleatoriamente.
6. Prepare $A(t)$ para a reprodução, montando a roleta de seleção.

*Passo 2*

1. Se o limite máximo de casamentos foi atingido, $t > Cas$, vá para o *Passo 3*.
2. $t \leftarrow t + 1$.
3. Selecione um par de pais de $A(t-1)$, usando a classificação de Reeves + roleta para um pai, e distribuição uniforme de probabilidades para o outro.
4. Selecione de $A(t-1)$ um par de indivíduos com *fitness* abaixo do mediano para ser removido.
5. Produza um par de filhos, aplicando o *crossover* e a mutação no par de pais selecionado, segundo as probabilidades $p_c$ e $p_m$.
6. Em $A(t-1)$, introduza o par de filhos gerados no lugar do par de indivíduos escolhidos para morrer, formando assim $A(t)$.

7. Prepare $A(t)$ para a reprodução, montando a roleta de seleção.

8. Volte para o *Passo 2.*

*Passo 3*

1. Fim.

# Capítulo 4

# IMPLEMENTAÇÕES DE ALGORITMOS GENÉTICOS

Nesta tese foram implementados e comparados vários dos algoritmos genéticos descritos no capítulo 3. Os algoritmos genéticos implementados podem ser agrupados segundo o diagrama da Figura 4-1.

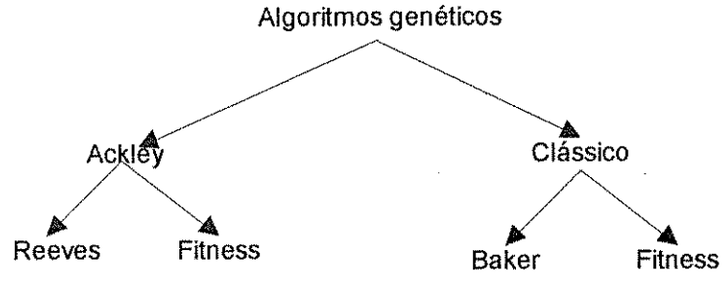



Figura 4-1: Diagrama de critérios de seleção

- **Ackley + Reeves:** Implementação na qual o critério de seleção é o proposto por Reeves (1995), e o corpo do algoritmo genético segue a proposta de substituição em regime permanente proposta por Ackley (1987).

- **Ackley + Fitness**: Implementação na qual o critério de seleção é baseado no valor do *fitness* (3.2), e o corpo do algoritmo genético segue a proposta de substituição em regime permanente proposta por Ackley (1987).

- **Clássico + Baker**: Implementação na qual o critério de seleção é baseado em classificação, como ilustrado na Figura 3-3, e o corpo do algoritmo genético segue a proposta de Holland (1975).

- **Clássico + Fitness**: Implementação na qual o critério de seleção é baseado em *fitness* (3.2), e o corpo do algoritmo genético segue a proposta de Holland (1975).

Em todas as implementações, o mecanismo de seleção utilizado foi o da roleta, com exceção da implementação **clássico + Baker**. Quanto ao grau de conhecimento embutido, as soluções estudadas podem ser agrupadas segundo o diagrama da Figura 4-2. Na implementação do tipo **B** cromossomos que geram qualquer seqüência de entrada de tarefas em cada estágio são aceitos, enquanto na implementação do tipo **A** o processo de formação dos cromossomos permite apenas aqueles que geram a mesma seqüência de tarefas em todos os estágios. Na implementação do tipo **B** há duas propostas de representação da informação na forma de cromossomos, que são o cromossomo I, e o cromossomo II. As implementações híbridas usam o algoritmo genético para gerar seqüências de entrada de tarefas, delegando a designação de tarefas a máquinas para regras de alocação.

## 4.1 IMPLEMENTAÇÕES DE ALGORITMOS GENÉTICOS PUROS

### 4.1.1 CODIFICAÇÃO DA INFORMAÇÃO EM CROMOSSOMOS

#### Tipo I

A representação da informação do programa das tarefas na forma de cromossomos deve ser feita de forma bastante natural, para que os operadores de *crossover* e mutação sejam fáceis de ser implementados e interpretados. Considere, por exemplo, um estágio $l$ com 3 máquinas paralelas ($m_l = 3$), e dez tarefas ($n = 10$) que devem ser processadas segundo a ordem:

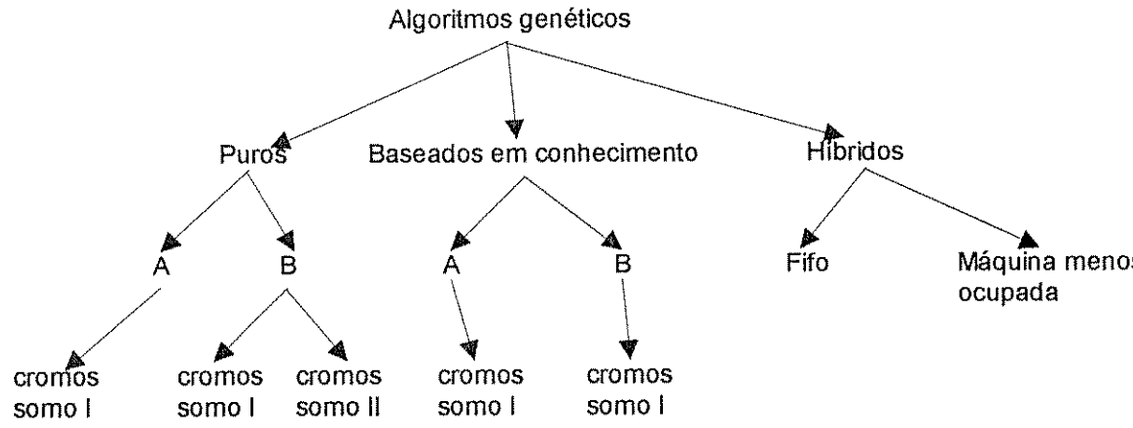



Figura 4-2: Diagrama de implementações estudadas

- máquina 1 processa as tarefas 2, 3 e 7 nesta ordem, $N_{l1} = \{2, 3, 7\}$, $\pi_{l1} = \{2, 3, 7\}$.
- maquina 2 processa as tarefas 5 e 1 nesta ordem, $N_{l2} = \{5, 1\}, \pi_{l2} = \{5, 1\}$.
- maquina 3 processa as tarefas 8, 9, 10, 6 e 4 nesta ordem, $N_{l3} = \{8, 9, 10, 6, 4\}, \pi_{l3} = \{8, 9, 10, 6, 4\}$.

Qual seria uma forma simples e econômica de codificação dessas informações na forma de cromossomos? A solução encontrada foi valer-se de dois vetores:

- um para armazenar a ordem de processamento das tarefas (**vetor de tarefas**),
- e outro formado por um conjunto de apontadores indicando a máquina responsável pelo processamento das tarefas (**vetor de máquinas**).

Assim, o exemplo anterior é representado na Figura 4-3. Com esta representação sabemos que a máquina 1 processa as tarefas 2,3 e 7, nesta ordem. A máquina 2 processa as tarefas 5 e 1, e a máquina 3 processa as tarefas 8, 9, 10, 6 e 4.

O conjunto formado pelos vetores de tarefas e de máquinas de um estágio será chamado de codificação de um estágio ou estágio, desde que não comprometa o sentido do texto. Para codificar um *flow shop* flexível composto de $m$ estágios, serão necessárias $m$ codificações de estágios, que passarão a compor juntas, uma unidade chamada cromossomo. A Figura 4-4

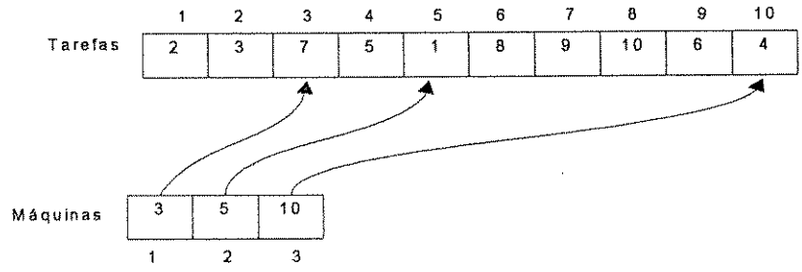



Figura 4-3: Exemplo de codificação de um estágio.

ilustra um cromossomo de um *flow shop* flexível com 3 estágios e 10 tarefas. Observe que o vetor de tarefas de um estágio é uma permutação no número de tarefas $n$.

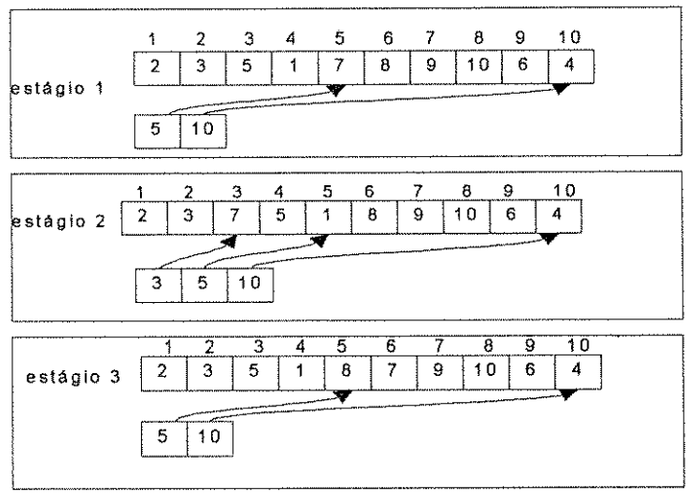



Figura 4-4: Cromossomo com 3 estágios

### Tipo II

Este tipo de codificação usa apenas um **vetor de estágio** para representar a informação de um estágio. Números positivos representam tarefas, e números negativos representam máquinas. A Figura 4-5 ilustra esta representação para o exemplo da Figura 4-3.

- A máquina 1 processa as tarefas 2, 3 e 7 nesta ordem.

- A máquina 2 processa as tarefas 5 e 1 nesta ordem.
- A máquina 3 processa as tarefas 8, 9, 10, 6 e 4 nesta ordem.

| -1 | 2 | 3 | 7 | -2 | 5 | 1 | -3 | 8 | 9 | 10 | 6 | 4 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

Figura 4-5: Exemplo de codificação de estágio tipo II

Para representar $m$ estágios são necessários $m$ vetores deste tipo.

### 4.1.2 IMPLEMENTAÇÃO PURA DO TIPO A + CROMOSSOMO TIPO I

Para um bom funcionamento do algoritmo genético é necessária a elaboração de operadores de *crossover* e mutação que sejam eficientes na exploração do espaço de soluções, de forma a permitir a convergência da população para soluções de alta qualidade. Para tanto, várias propostas de mutação e *crossover* foram elaboradas e implementadas.

#### Cromossomos permitidos

No *flow shop* de permutação com uma máquina por estágio, a seqüência de processamento das tarefas é repetida em todos os estágios de processamento. Assim, se houver 5 tarefas para serem processadas, e a ordem de entrada for 3, 2, 1, 4, 5 no primeiro estágio, esta mesma ordem deve ser repetida em todos os estágios subseqüentes. Usando esta idéia, na solução tipo **A**, o **vetor de tarefas** é repetido em todos os estágios de um cromossomo. O espaço de soluções não é amplamente coberto e muitas regiões boas podem ficar inexploradas ao se repetir o **vetor de tarefas** em todos os estágios.

#### Crossover tipo A.1

É uma generalização da proposta de Reeves (1995), que implementou um algoritmo genético para o problema de *flow shop* de permutação com uma máquina em cada estágio. O *crossover* é realizado da seguinte forma:

- Dois cromossomos filhos são gerados a partir de dois cromossomos pais.
- Um número aleatório inteiro no intervalo $[\mathbf{1}, \mathbf{n}]$ gerado com distribuição uniforme é atribuído à variável **ponto_cross**. Assim, tem-se o índice do local onde deve-se fracionar o **vetor de tarefas** de um estágio.
- O **filho1** recebe as partes iniciais dos **vetores de tarefas** dos genes do **pai1** que compreendem as tarefas entre índices [**1**, **ponto_cross**]. As demais posições dos **vetores de tarefas** são preenchidas com as tarefas remanescentes, respeitando as posições relativas em que se encontram no **vetor de tarefas** do **pai2**. Para cada estágio, percorre-se o **vetor de tarefas** do **pai2** do início ao fim. Ao se encontrar uma tarefa no **vetor de tarefas** do **pai2** que ainda não se encontra no respectivo estágio do **filho1**, copia-se esta tarefa para o **vetor de tarefas** do **filho1**.
- Os **vetores de tarefas** dos estágios do **filho2** são preenchidos de forma similar ao do **filho1**. As partes iniciais vêm do **pai2** e as partes finais vêm do **pai1**.
- O **filho1** herda os **vetores de máquinas** do **pai1** em todos os estágios.
- O **filho2** herda os **vetores de máquinas** do **pai2** em todos os estágios.

O ponto de *crossover* é o mesmo para todos os estágios, gerando assim, **vetores de tarefas** idênticos em todos os estágios do cromossomo. A Figura 4-6 ilustra a operação de *crossover*.

**Crossover tipo A.2**

O crossover tipo A.2 altera o **vetor de máquinas** de um dado estágio, ao contrário do anterior que altera o **vetor de tarefas**. Dados **pai1**, **pai2** e o **ponto_cross** seguem abaixo as regras de formação dos estágios dos filhos criados via *crossover*, acompanhadas de um exemplo para cada regra aplicada a um estágio $l$ dos cromossomos pais.

1. O *crossover* ocorre em apenas um estágio. O **filho1** recebe integralmente os estágios do **pai1** que não sofreram *crossover*, e o **filho2** recebe os do **pai2**. Cada estágio de um cromossomo, tem a mesma probabilidade de sofrer o *crossover*, e apenas um estágio é escolhido.

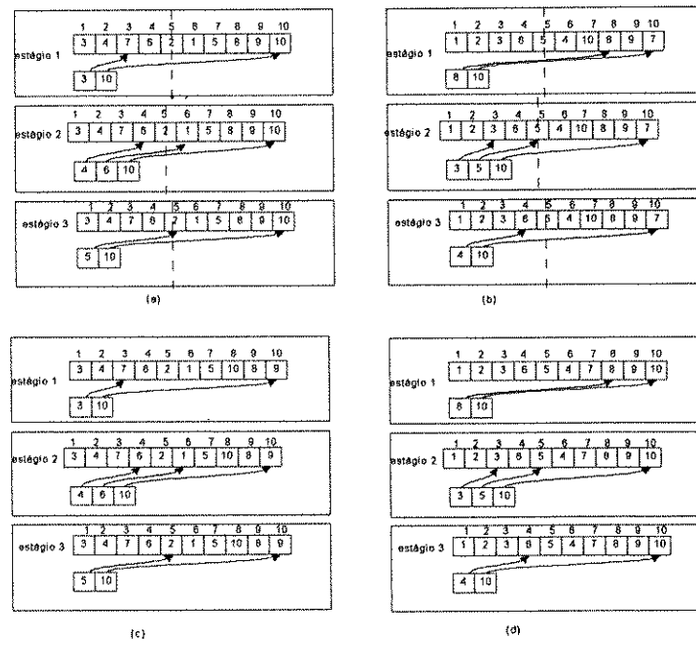



Figura 4-6: *Crossover* tipo A.1. (a) e (b) são os pais 1 e 2. (c) e (d) são os filhos 1 e 2.

2. Para o estágio que sofre o *crossover*, tem-se para o **filho1:**

    1. O **filho1** recebe o **vetor de tarefas** do **pai1** integralmente. De forma similar o **filho2** recebe o **vetor de tarefas** do **pai2**, como mostra a Figura 4-7.
    2. O **vetor de maquinas** do **filho1** é formado, copiando do **pai1** a fração inicial do **vetor de máquinas** cujos valores armazenados em seus elementos forem menores que a variável **ponto_cross**. A parte final do **vetor de máquinas** vem do **pai2**. Assim, se houver $m_l$ elementos no **vetor de máquinas** e os $k$ primeiros vierem do **pai1**, os $(m_l - k)$ finais serão copiados do final do **vetor de máquinas** do **pai2**, como poder ser visto na Figura 4-8.
    3. Estando completo o vetor de máquinas do **filho1**, é necessário colocar em ordem crescente os seus elementos, como ilustra a Figura 4-9.
    4. Muitas vezes, este *crossover* pode deixar algumas máquinas ociosas. Como a ociosidade não é algo desejável, um algoritmo de não ociosidade é acionado para que cada máquina ociosa processe pelo menos uma tarefa. Primeiro, tenta-se

passar para a máquina ociosa a última tarefa da máquina cujo índice antecede o da máquina ociosa: a máquina ociosa $i$, passa a processar a tarefa $\pi_{l,i-1}(n_{l,i-1})$. Se não for possível, atribui-se à máquina ociosa a primeira tarefa da máquina cujo índice sucede o da ociosa: a máquina ociosa $i$, passa a processar a tarefa $\pi_{l,i+1}(1)$, como pode ser visto nas Figuras 4-10 e 4-11. É importante observar que na transição da Figura 4-10 para a Figura 4-11, ao se remover a ociosidade da máquina 4, cria-se temporariamente ociosidade na máquina 3. Deve-se neste caso, aplicar novamente a regra, para que se remova a ociosidade. Caso a ociosidade não seja removida desta forma, aloca-se à máquina ociosa a primeira tarefa da máquina seguinte, e repete-se este passo até a remoção da ociosidade. No exemplo, seria necessário passar para a máquina 4, a primeira tarefa da máquina 5, ou seja, a tarefa de número 6 passaria a ser processada pela máquina 4.

O **vetor de tarefas** do **filho2** é gerado de forma similar. Porém, a parte inicial vem do **pai2** e a parte final vem do **pai1**.

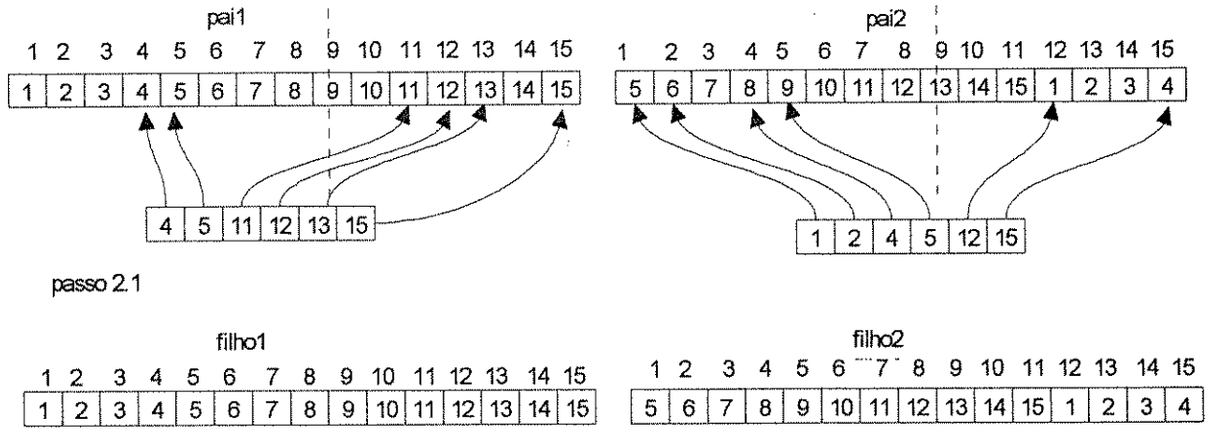



Figura 4-7: *Crossover* tipo A.2, passo 2.1

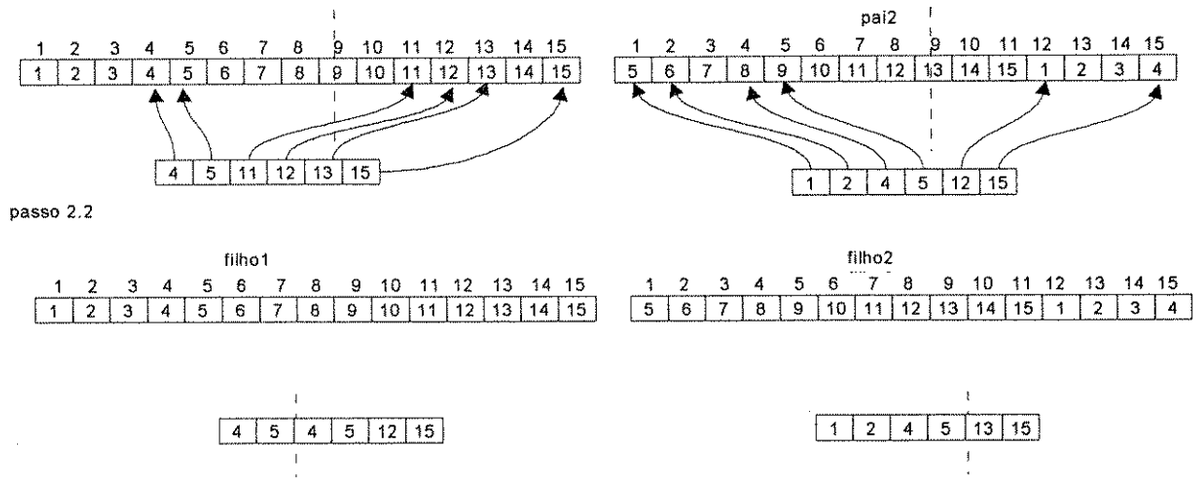



Figura 4-8: *Crossover* tipo A.2, passo 2.2

**Mutação tipo A.1 - inserção**

O objetivo deste tipo de mutação é alterar a ordem de processamento das tarefas e a distribuição de tarefas entre as máquinas, por isso trabalha-se com o **vetor de tarefas** e o **vetor de máquinas**. Dois índices do **vetor de tarefas** são escolhidos aleatoriamente. O primeiro é o ponto de inserção e o segundo é o ponto de remoção. As tarefas são rearranjadas através da operação de inserção. A mutação é aplicada em todos os estágios, com o mesmo ponto de inserção e remoção. Com isto o **vetor de tarefas** fica idêntico em todos os estágios. O **vetor de máquinas** pode ser alterado por este tipo de mutação, pois este operador faz inserção de uma tarefa na mesma máquina ou inserção de uma tarefa em outra máquina, como pode-se ver na Figura 4-12. Quando o ponto de inserção e o ponto de remoção caem em grupos de tarefas processados por máquinas distintas, o **vetor de máquinas** é alterado. Na Figura 4-12, no estágio 2, o ponto de remoção cai no interior do grupo $N_{22}$, e o ponto de inserção cai no interior do grupo $N_{23}$, com isto a máquina 2 deixa de processar a tarefa de número 2, que passa para o domínio da máquina 3. Toma-se o cuidado de não deixar uma máquina ociosa, isto é, se uma máquina $i$ num estágio $l$ é responsável pelo processamento de apenas uma tarefa, esta tarefa não pode ser removida de seu grupo $N_{li}$.



UNICAMP
BIBLIOTECA CENTRAL

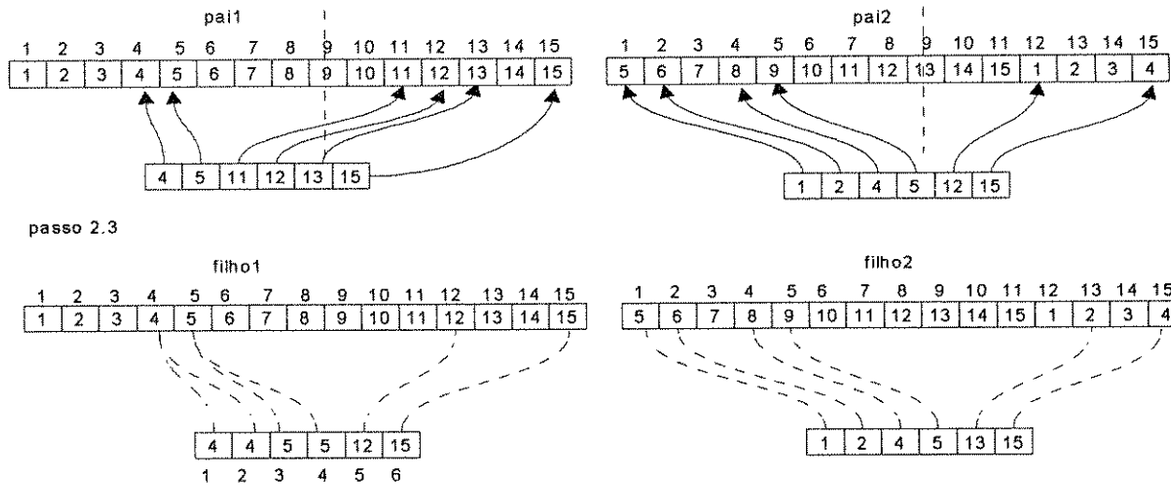



Figura 4-9: *Crossover* A.2, passo 2.3. As máquinas 2 e 4 no filho 1 estão ociosas.

**Mutação tipo A.2 - troca**

Este tipo de mutação é do tipo **troca** e trabalha apenas com o **vetor de tarefas**. Dois números $t_1$ e $t_2$ são sorteados no intervalo $[1, n]$; os conteúdos dos elementos dos **vetores de tarefas** correspondentes aos índices $t_1$ e $t_2$, são trocados entre si, em todos os estágios, como mostra a Figura 4-13.

### 4.1.3 IMPLEMENTAÇÃO PURA DO TIPO B + CROMOMOSSOMO TIPO I

**Cromossomos permitidos**

- **Vetor de tarefas**: em cada estágio do cromossomo, este vetor conterá qualquer permutação do conjunto de tarefas, ao contrário do *crossover* tipo A.1 que forçava a repetição do **vetor de tarefas** em todos os estágios do cromossomo. Esta decisão permite uma melhor exploração do espaço de soluções.

- **Vetor de máquinas:** funcionamento igual ao da solução A.1.

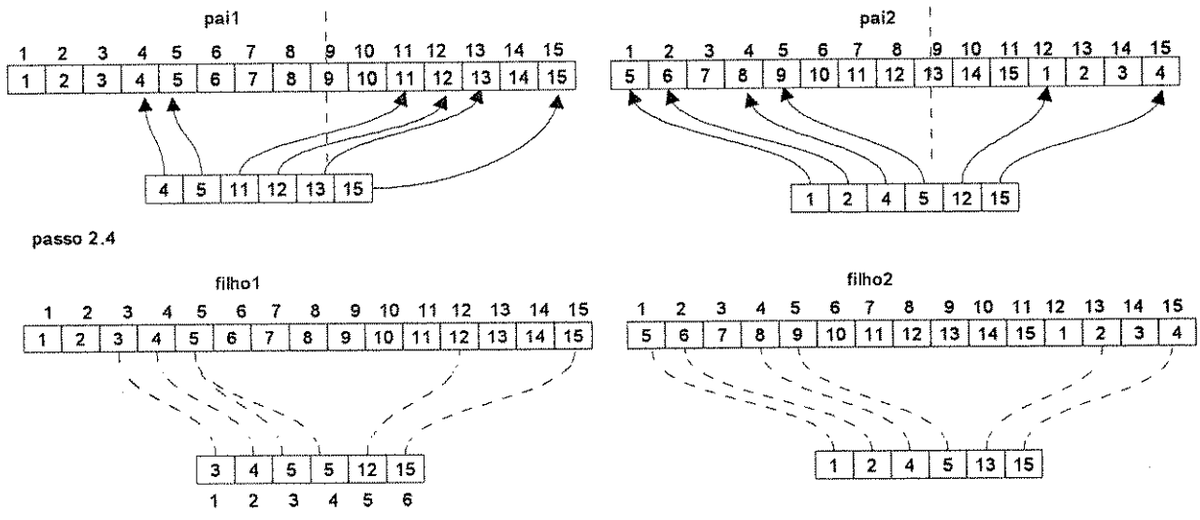



Figura 4-10: *Crossover* A.2, passo 2.4. A máquina 4 no filho 1 está ociosa.

### Crossover tipo B.1

É uma variação do *crossover* tipo A.1, porém o ponto de *crossover* não é necessariamente o mesmo para todos os estágios. A Figura 4-14 ilustra a operação de *crossover*.

### Crossover tipo B.2

O *crossover* B.2 foi igual ao *crossover* do tipo A.2 usado na implementação pura do tipo A.

### Mutação tipo B.1

É similar à mutação do tipo A.1, porém os pontos de inserção e remoção não são necessariamente os mesmos para todos os estágios. A Figura 4-15 ilustra a operação de mutação.

### Mutação tipo B.2

Realizada da mesma forma que a mutação do tipo A.2, exceto pelo fato dos índices de troca $t_1$ e $t_2$ não serem necessariamente os mesmos para todos os estágios. A Figura 4-16 ilustra este tipo de mutação.

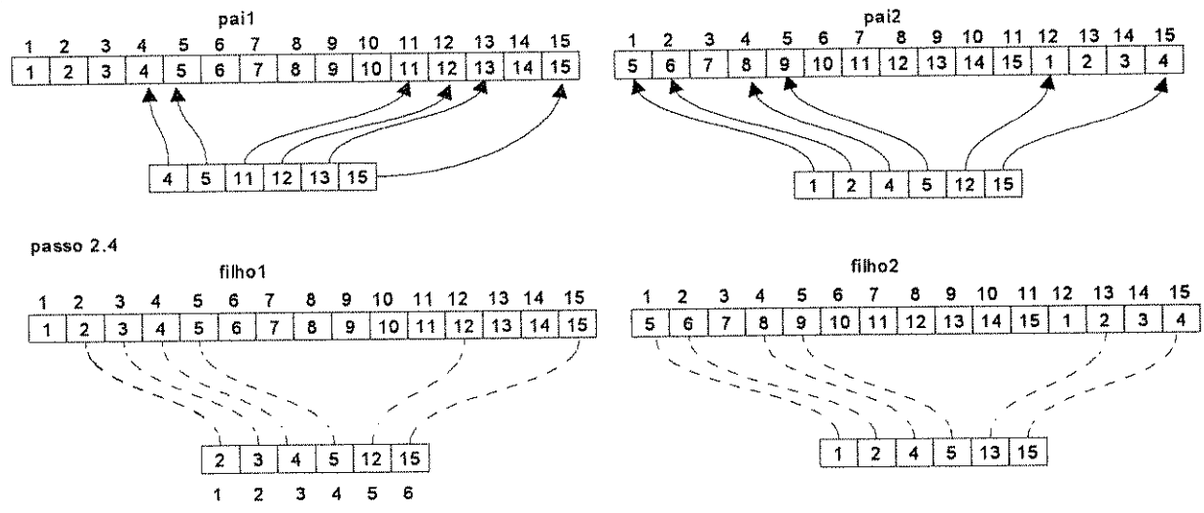



Figura 4-11: *Crossover* A.2, final do passo 2.4. Nenhuma máquina ociosa em nenhum dos filhos.

### 4.1.4 SOLUÇÃO PURA TIPO B + CROMOSSOMO TIPO II

**Cromossomos permitidos**

- **Vetor de estágio:** A primeira posição do vetor sempre deve conter o valor -1, para que as tarefas seguintes estejam alocadas a uma máquina. As demais posições do vetor estão livres para receber máquinas (números negativos) ou tarefas (números positivos), desde que não surjam situações de máquinas ociosas. Se duas posições consecutivas no vetor contiverem números negativos, a primeira das máquinas não será responsável pelo processamento de nenhuma tarefa, ficando ociosa. A última posição do vetor também não pode abrigar um número negativo, senão esta máquina também estará ociosa, como mostra a Figura 4-17.

**Crossover tipo B.3**

Funciona de forma semelhante ao *crossover* tipo B.1, porém não existe o tratamento para o **vetor de máquinas** que aqui inexiste. Muitas vezes, este *crossover* pode gerar soluções com máquinas ociosas. Caso isto aconteça, o cromossomo é convertido em um cromossomo do tipo

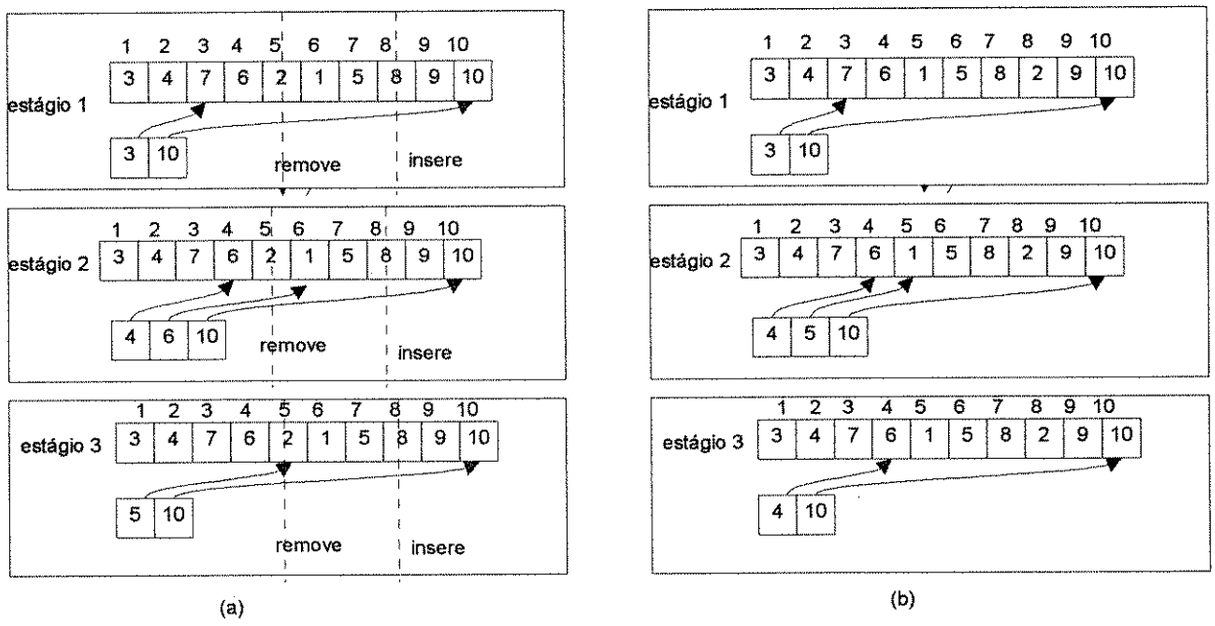



Figura 4-12: Mutação A.1 (a) cromossomo original; (b) cromossomo mutado

I, e o passo 4 do *crossover* tipo A.2 é aplicado ao cromossomo para remover a ociosidade.

**Mutação tipo B.3**

A mutação utilizada é do tipo inserção, com o cuidado de não provocar o surgimento de máquina ociosa.

## 4.2 IMPLEMENTAÇÕES DE ALGORITMOS GENÉTICOS BASEADOS EM CONHECIMENTO + CROMOSSOMO TIPO I

### 4.2.1 CODIFICAÇÃO DA INFORMAÇÃO EM CROMOSSOMOS

Esta codificação é uma extensão da proposta na implementação de algoritmo genético puro. O conhecimento sobre o caminho crítico associado ao programa que um cromossomo codifica é armazenado junto ao mesmo, com o objetivo de eliminar-se *crossovers* e mutações que *a*

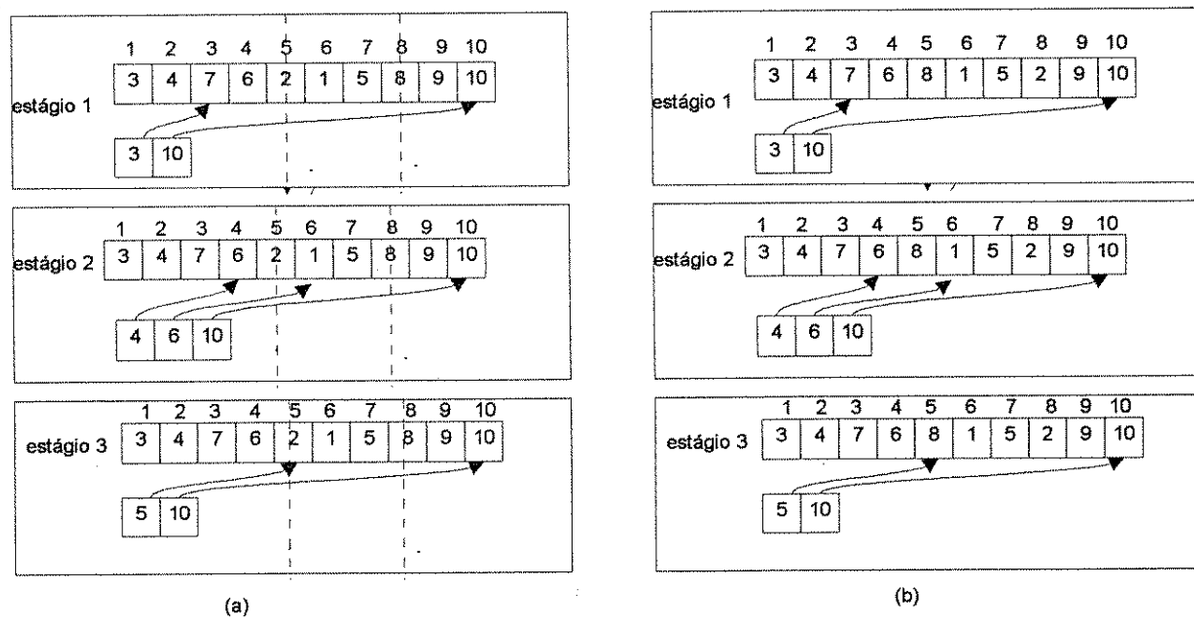



Figura 4-13: Mutação A.2 (a) cromossomo original; (b) cromossomo mutado

*priori* podem ser classificados como pouco promissores no tocante à melhoria da qualidade dos descendentes. Considere o diagrama de Gantt do exemplo do capítulo 3 e repetido na Figura 4-18 com seu caminho crítico hachurado. A Figura 4-19 mosta o cromossomo com o caminho crítico associado.

### 4.2.2 ELIMINANDO MUTAÇÕES DO TIPO INSERÇÃO NÃO PROMISSORAS - NOWICKI E SMUTNICKI

Tendo o caminho crítico associado a um cromossomo, é possível eliminar mutações do tipo inserção que não trarão melhorias no *makespan* do cromossomo mutado. Isto é possível determinar pois há mutações que levam a cromossomos cujos caminhos críticos dos seus programas contêm todos os nós do caminho crítico do cromossomo não mutado, isto é, nós $(l, j)$, $j \in B_l$, $l \in M$. Obviamente, o *makespan* do cromossomo mutado será maior ou igual ao *makespan* do cromossomo original, podendo ser descartado.

Seja $l$ um estágio selecionado, $a$ e $b$ grupos selecionados neste estágio e $x, y$ posições nas

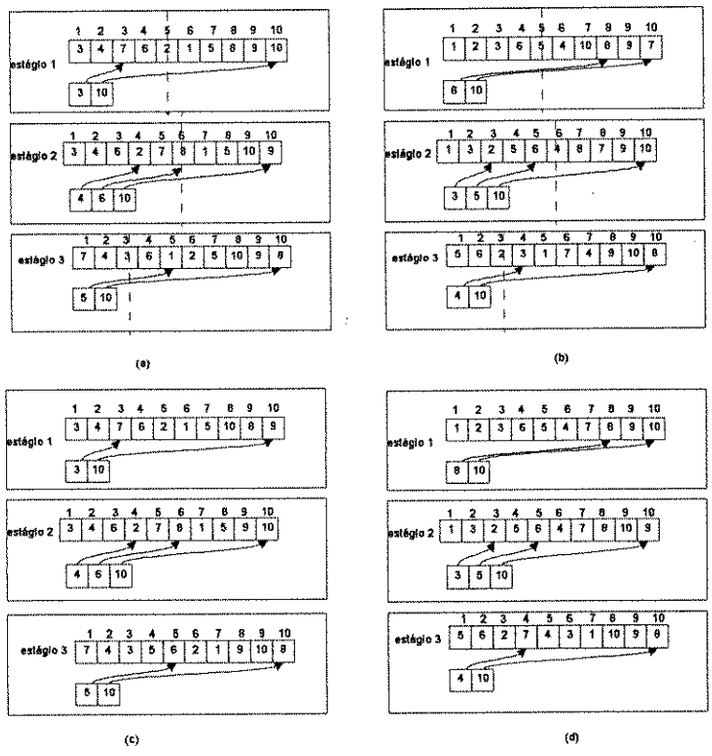



Figura 4-14: *Crossover* tipo B.1; (a) pai 1; (b) pai 2; (c) filho 1; (d) filho 2

permutações $\pi_{la}$ e $\pi_{lb}$. Um grupo é chamado crítico se o caminho crítico passa por ele. O vetor $v = (l, a, x, b, y)$ associado à ordem de processamento $\pi \in \Pi$ define uma mutação do tipo inserção em um cromossomo, de tal forma que a tarefa $z = \pi_{la}(x)$ é removida do grupo $N_{la}$ (precisamente da posição $x$ na permutação $\pi_{la}$) e então, é inserida em $N_{lb}$ na posição $y$ na permutação $\pi_{lb}$.

Uma mutação $v = (l, a, x, b, y)$ não diminui o *makespan*, se pelo menos uma das condições estabelecidas por Nowicki e Smutnicki (1995) for satisfeita:

- $(I_i)$ A tarefa $z$ não pertence ao bloco $B_l$, isto é: $a \neq h_l$ ($a$ não é o grupo crítico) ou $a = h_l$ e ($x < e_l$ ou $x > f_l$) ($a$ é o grupo crítico mas $x$ não está no interior do grupo crítico). Pode-se observar na Figura 4-18 que remover a tarefa 1 da máquina 1 do estágio 1, e inseri-la na máquina 2 antes da tarefa 4, piora o *makespan* (aumenta o caminho crítico). Também pode-se ver que remover a tarefa 7 da máquina 2 do estágio 1, e inseri-la antes

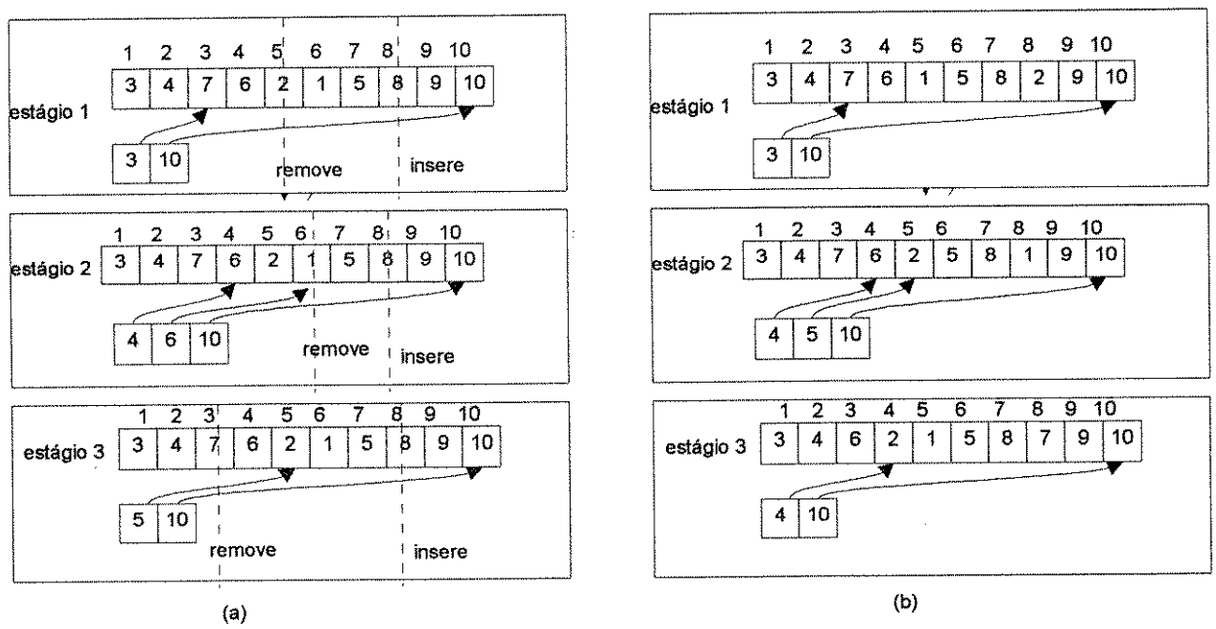



Figura 4-15: (a) cromossomo original; (b) cromossomo mutado

da tarefa 4, na mesma máquina, piora o *makespan*.

- ($I_{ii}$) A tarefa $z$ pertence ao bloco $B_l$, mas o bloco contém apenas uma tarefa, isto é, $a = h_l$ e $x = e_l = f_l$.

- ($I_{iii}$) A tarefa $z$ pertence ao interior do bloco $B_l$ e é movida dentro deste bloco, isto é $e_l < x < f_l$, $e_l < y < f_l$. Pela Figura 4-18 pode-se ver que o *makespan* não se altera ao inserir a tarefa 1 no lugar da tarefa 2, na máquina 1 do estágio 2.

- ($I_{iv\prime}$) A tarefa $z$ é a primeira no bloco $B_l$ e é movida para a esquerda, isto é: $x = e_l$, $y < e_l$. Na Figura 4-18, inserir a tarefa 3 no lugar da tarefa 4, na máquina 1 do estágio 3, piora o *makespan*.

- ($I_{iv\prime\prime}$) A tarefa $z$ é a última no bloco $B_l$ e é movida para a direita, isto é: $x = f_l$, $y > f_l$. Na Figura 4-18, inserir a tarefa 6 no lugar da tarefa 7, na máquina 1 do estágio 1, piora o *makespan*.

- ($I_{v\prime}$) $B_l$ é o primeiro bloco, a tarefa $z$ é a primeira neste bloco e é movida para a direita

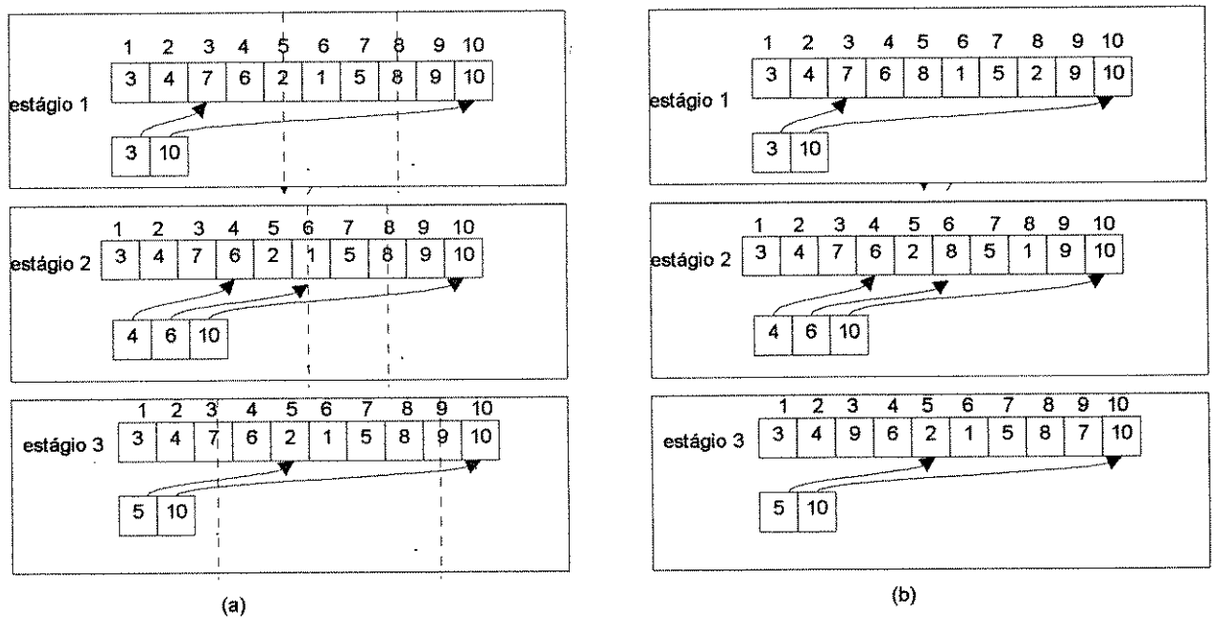



Figura 4-16: Mutação tipo B.1 (a) cromossomo original; (b) cromossomo mutado

dentro do bloco, isto é: $l = 1$, $x = e_1$, $y < f_1$. Na Figura 4-18, inserir a tarefa 4 no lugar da tarefa 5, na máquina 2 do estágio 1, não altera o *makespan*.

- $(I_{vii})$ $B_l$ é o último bloco, a tarefa $z$ é a última neste bloco e é movida para a esquerda dentro deste bloco, isto é: $l = m$, $x = f_m$, $y > e_m$.

Se em todos os estágios em que a mutação do tipo inserção for aplicada, uma das regras acima for satisfeita para cada estágio, a mutação com certeza é inutil, deve ser descartada e novas mutações devem ser realizadas no cromosssomo original, em busca de melhorias.

### 4.2.3 ELIMINANDO MUTAÇÕES DO TIPO TROCA NÃO PROMISSORAS

Seja $l$ um estágio selecionado, $a$ e $b$ grupos selecionados neste estágio e $x, y$ posições nas permutações $\pi_{la}$ e $\pi_{lb}$. O vetor $v = (l, a, x, b, y)$ associado com a ordem de processamento $\pi \in \Pi$ define uma mutação do tipo troca em um cromossomo, de tal forma que a tarefa $z = \pi_{la}(x)$

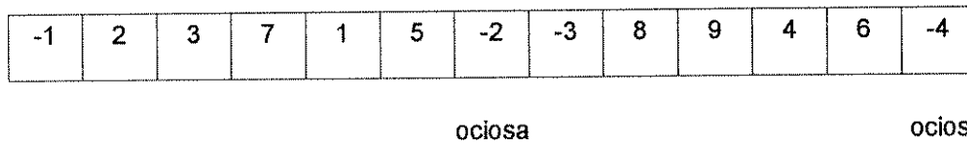



Figura 4-17: Exemplo de estágio com máquina ociosa

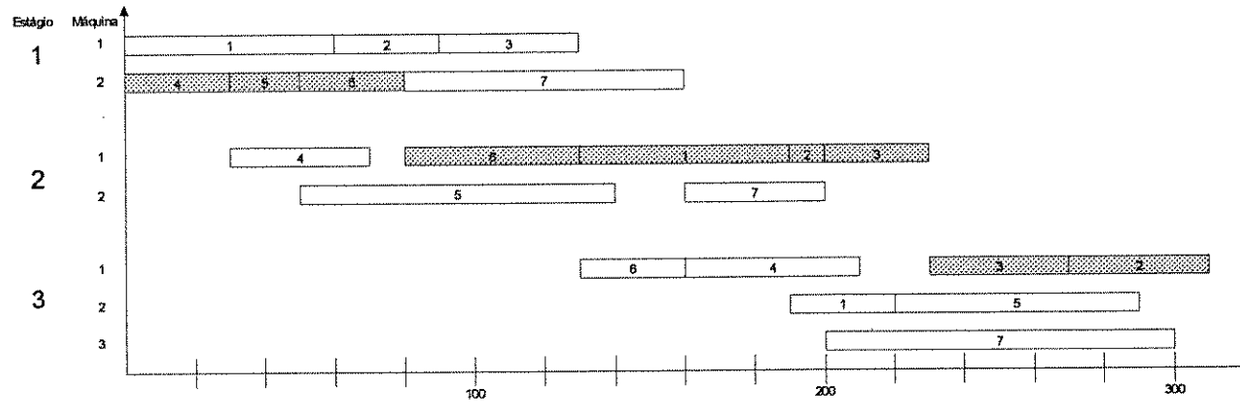



Figura 4-18: Diagrama de Gantt para o exemplo, o caminho crítico está sombreado

do grupo $N_{la}$ (precisamente da posição $x$ na permutação $\pi_{la}$) troca de posição com a tarefa em $N_{lb}$ na posição $w = \pi_{lb}(y)$.

Uma mutação $v = (l, a, x, b, y)$ é inútil, se pelo menos uma das condições for satisfeita:

- $(T_i)$ As tarefas $z$ **e** $w$ não pertencem ao bloco $B_l$, isto é, $a \neq h_l$ ($a$ não é o grupo crítico) ou $a = h_l$ e ($x < e_l$ ou $x > f_l$) ($a$ é o grupo crítico mas $x$ não está no interior do grupo crítico) e $b \neq h_l$ ($b$ não é o grupo crítico) ou $b = h_l$ e ($y < e_l$ ou $y > f_l$) ($b$ é o grupo crítico mas $y$ não está no interior do grupo crítico)

- $(T_{ii})$ A tarefa $z$ **ou** $w$ pertence ao bloco $B_l$, mas o bloco contém apenas uma tarefa, isto é, $a = h_l$ e $x = e_l = f_l$ **ou** $b = h_l$ e $y = e_l = f_l$

- $(T_{iii})$ As tarefa $z$ **e** $w$ pertencem ao interior do bloco $B_l$, isto é, $e_l < x < f_l$, $e_l < y < f_l$.

- $(T_{iv\prime})$ A tarefa $z$ **ou** $w$ é a primeira no bloco $B_l$ e é movida para a esquerda, isto é, $x = e_l$,

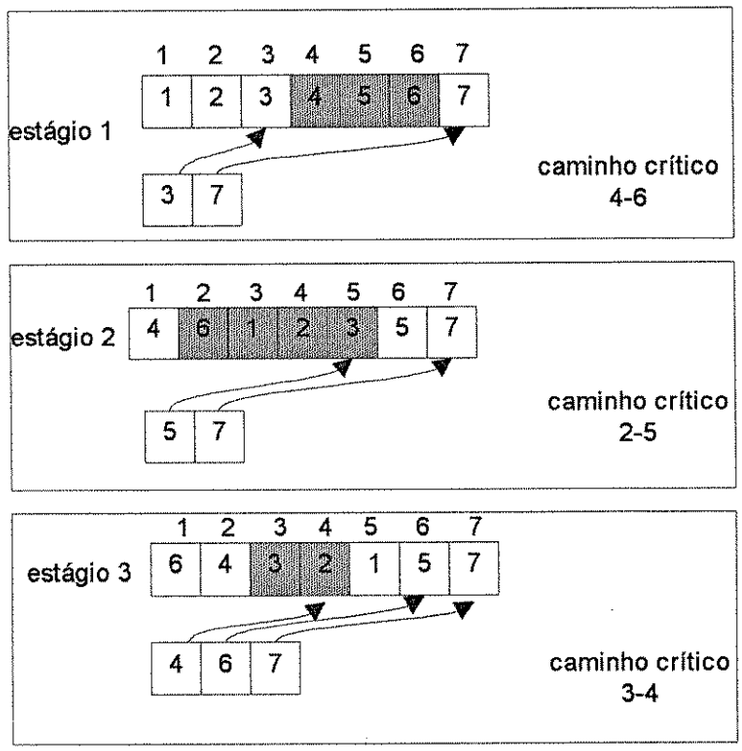



Figura 4-19: Cromossomo com caminho crítico associado

$y < e_l$ **ou** $y = e_l$, $x < e_l$.

- ($T_{iv//}$) A tarefa $z$ **ou** $w$ é a última no bloco $B_l$ e é movida para a direita, isto é, $x = f_l$, $y > f_l$ **ou** $y = f_l$, $x > f_l$.

- ($T_{v//}$) $B_l$ é o primeiro bloco, a tarefa $z$ **ou** $w$ é a primeira neste bloco e é movida para a direita dentro do bloco, isto é, $l = 1$, $x = e_1$, $y < f_1$, **ou** $y = e_1$, $x < f_1$.

- ($T_{vi//}$) $B_l$ é o último bloco, a tarefa $z$ **ou** $w$ é a última neste bloco e é movida para a esquerda dentro deste bloco, isto é, $l = m$, $x = f_m$, $y > e_m$ **ou** $y = f_m$, $x > e_m$.

Se em todos os estágios em que a mutação do tipo troca for aplicada, uma das regras acima for satisfeita para cada estágio, a mutação com certeza é inutil, deve ser descartada e uma nova mutação deve ser realizada no cromosssomo original, em busca de melhorias.

### 4.2.4 IMPLEMENTAÇÕES BASEADAS EM CONHECIMENTO DOS TIPOS A E B

A implementação baseada em conhecimento do tipo A, é uma extensão da implementação pura do tipo A, com o acréscimo da informação sobre caminho crítico que realiza as mutações utilizando esse conhecimento. O *crossover* do tipo 1 é realizado de tal forma que o ponto de *crossover* caia pelo menos no interior de um bloco $B_l$, para que pelo menos uma parte do caminho crítico seja envolvida no processo de *crossover* (aqui o ponto de *crossover* é o mesmo para todos os vetores de tarefa, como já foi discutido).

A implementação baseada em conhecimento do tipo B, de forma análoga vem da implementação pura do tipo B. Aqui no processo de *crossover* do tipo 1, todos os **vetores de tarefas** são particionados no interior de seu $B_l$, para cada estágio $l$. Assim o caminho crítico é envolvido em todos os estágios.

## 4.3 IMPLEMENTAÇÕES HÍBRIDAS: ALGORITMOS GENÉTICOS + REGRAS DE ALOCAÇÃO

### 4.3.1 CODIFICAÇÃO DA INFORMAÇÃO EM CROMOSSOMOS

Nestas implementações, a codificação da informação na forma de cromossomo é a mais econômica em termos de memória. Aqui o **vetor de máquinas** é dispensado, pois a função de atribuir tarefas a máquinas é retirada do escopo do algoritmo genético e passada para uma regra de alocação que trabalha em conjunto com o algoritmo genético. É necessário apenas um **vetor de tarefas** para codificar o cromossomo. A Figura 4-20 ilustra um cromossomo. A forma como as tarefas são processadas depende da regra de alocação que acompanha a solução.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| Tarefas | 2 | 3 | 7 | 5 | 1 | 8 | 9 | 10 | 6 | 4 |

Figura 4-20: Exemplo de cromossomo

### 4.3.2 CROSSOVER

O *crossover* é implementado de forma similar aos *crossovers* do tipo A.1 nas implementações de algoritmos genéticos puro e baseado em conhecimento. A diferença é que aqui não há vários estágios que podem sofrer o *crossover*, existe apenas um **vetor de tarefas**. A Figura 4-21 ilustra a operação.

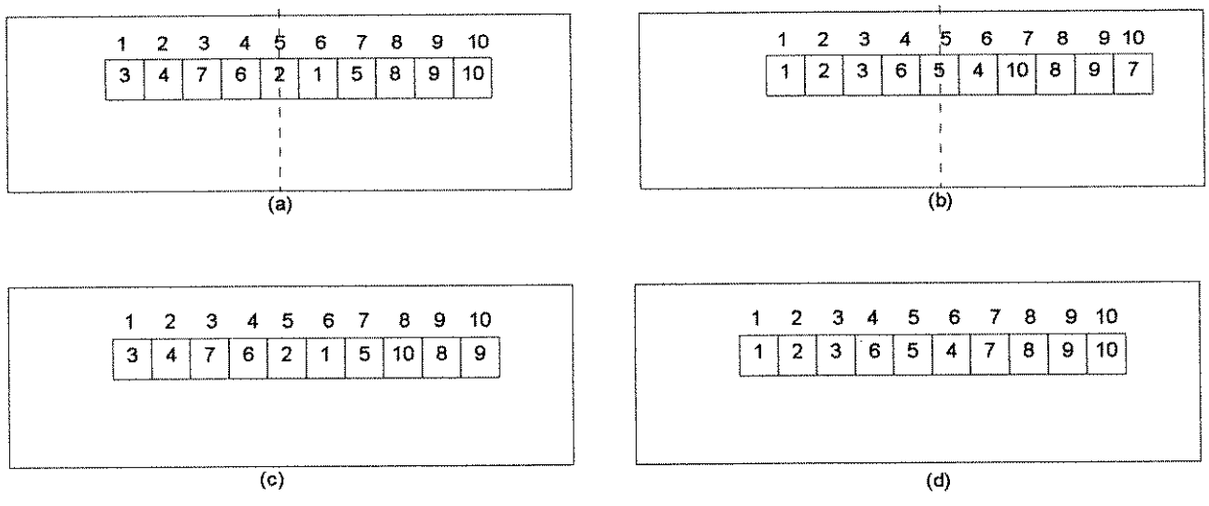



Figura 4-21: Exemplo de *crossover*. (a), (b) são os cromossomos pais, (c) e (d) são os cromossomos filhos

### 4.3.3 MUTAÇÃO DO TIPO 1

É a mutação do tipo inserção já utilizada nas implementações de algoritmos genéticos puro e baseado em conhecimento. A Figura 4-22 mostra um exemplo.

### 4.3.4 MUTAÇÃO DO TIPO 2

É a mutação do tipo troca já utilizada nas implementações de algoritmos genéticos puro e baseado em conhecimento. A Figura 4-23 mostra um exemplo.

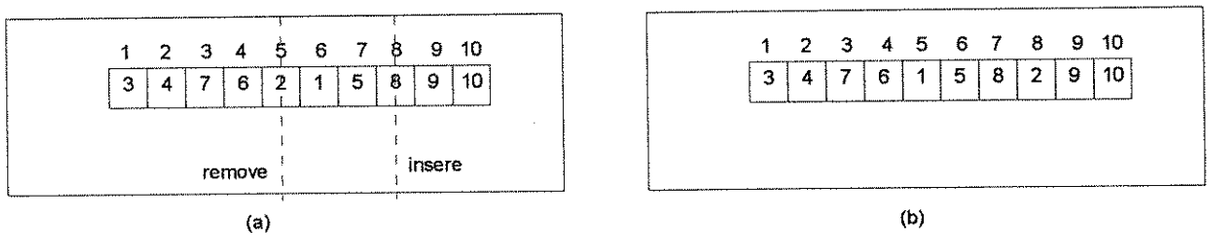



Figura 4-22: Mutação do tipo 1. (a) cromossomo original, (b) cromossomo mutado

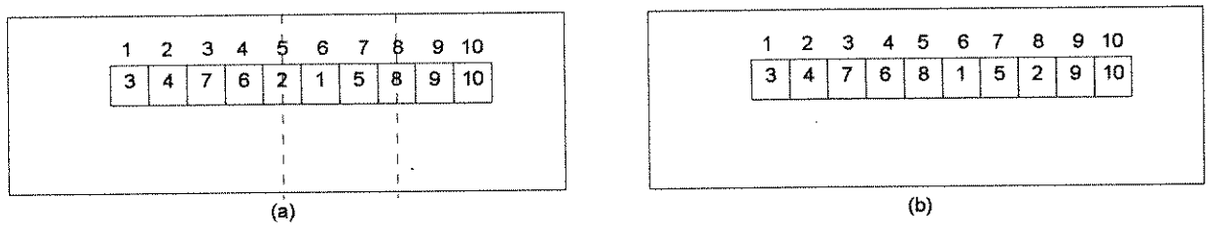



Figura 4-23: Mutação do tipo 2. (a) cromossomo original, (b) cromossomo mutado

### 4.3.5 IMPLEMENTAÇÃO HÍBRIDA DO TIPO FIFO

Na Figura 4-20, as tarefas estão ordenadas: $2, 3, 7, 5, \ldots, 4$. Esta implementação considera que as tarefas entram no sistema nesta ordem, e são atendidas por ordem de chegada (*first in-first out*). No primeiro estágio, a tarefa 2 será a primeira a ser alocada. Esta alocação é realizada conduzindo a tarefa para a máquina menos ocupada[1] do estágio. No segundo estágio, a primeira tarefa a ser processada, será a primeira a deixar o estágio 1. Note que não é necessariamente a tarefa de número 2. A informação contida no cromossomo está influenciando apenas a ordem de entrada das tarefas no primeiro estágio, nos estágios subseqüentes, as tarefas são designadas segundo a ordem de chegada (política *fifo*), e são alocadas imediatamente para a máquina menos ocupada daquele estágio, no instante que chegam nele. O algoritmo genético procura a melhor forma de iniciar o processamento das tarefas segundo a regra *fifo*.

[1]A máquina menos ocupada é aquela com menor tempo de processamento alocado.

### 4.3.6 IMPLEMENTAÇÃO HÍBRIDA DO TIPO MÁQUINA MENOS OCIOSA

Esta solução considera o **vetor de tarefas** em todos os estágios de processamento. Em cada estágio, as tarefas são alocadas para as máquinas de acordo com a ordem em que se encontram no **vetor de tarefas**. A tarefa é designada à máquina menos ocupada do estágio. Na Figura 4-20 as tarefas estão ordenadas assim: $2, 3, 7, 5, \ldots, 4$. No primeiro estágio, a primeira tarefa a ser alocada será a 2, depois a 3, e assim sucessivamente. A tarefa 2 é designada para a máquina menos ocupada do primeiro estágio (no caso qualquer uma, pois todas estarão ociosas no início do processamento). Depois a tarefa 3 é alocada para a máquina menos ocupada. No segundo estágio, a ordem de alocação das tarefas $(2, 3, 7, 5, \ldots, 4)$ se mantém, independente de qual foi a primeira tarefa a deixar o primeiro estágio. Assim, no segundo estágio, a tarefa 2 também será a primeira a ser alocada. Nesta implementação, o algoritmo genético determina a ordem de entrada das tarefas em cada estágio, e a regra de alocação atribui as tarefas à máquina menos ocupada. Aqui o algoritmo genético tem um papel diferente em relação à solução híbrida do tipo *fifo*, na qual o algoritmo genético apenas determinava a seqüência de entrada das tarefas no sistema, deixando o restante do trabalho para a heurística.

Esta implementação surgiu da analogia feita com o *flow shop* de permutação com uma máquina por estágio. A mesma seqüência de tarefas definida no **vetor de tarefas** é aplicada em todos os estágios, e a regra de alocação existe para tirar proveito das máquinas paralelas. Apesar da analogia, esta implementação permite situações nas quais máquinas ficam ociosas mesmo quando há tarefas disponíveis para processamento. Na Figura 4-24, as tarefas 3,4 e 5 terminam o processamento antes das tarefas 1 e 2 no estágio 1, porém não podem ser imediatamente processadas no segundo estágio. Apesar deste aspecto restritivo, esta implementação foi a que apresentou os melhores resultados computacionais.

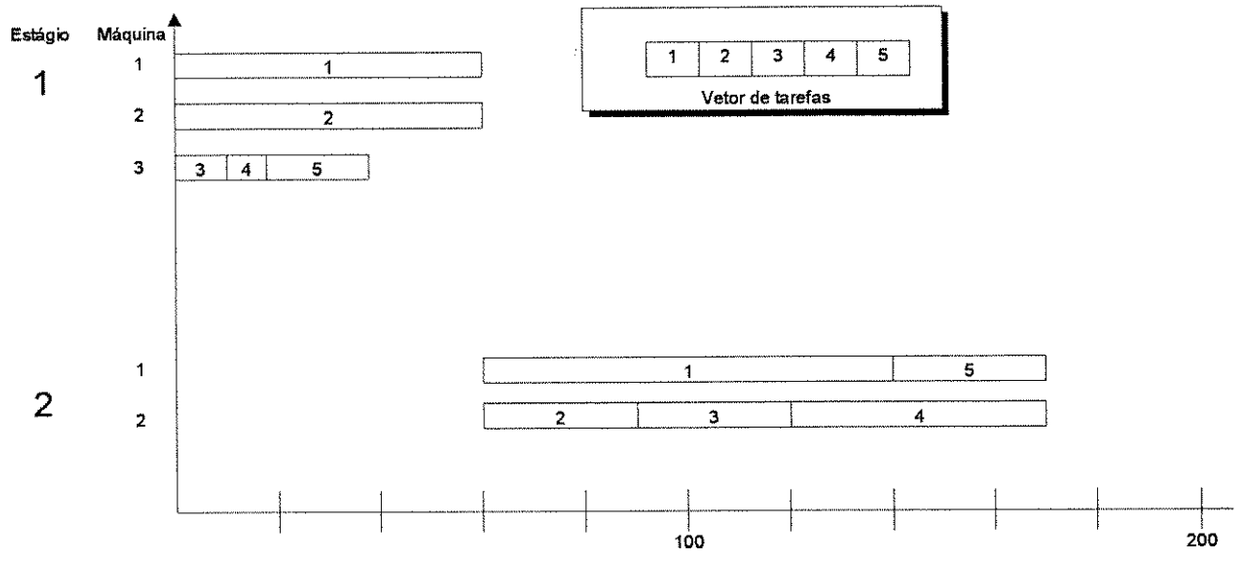



Figura 4-24: Máquinas ociosas no estágio 2, mesmo havendo tarefas disponíveis para processamento

# Capítulo 5

# RESULTADOS COMPUTACIONAIS

## 5.1 GERAÇÃO DAS INSTÂNCIAS

As instâncias foram geradas de forma similar àquela apresentada no trabalho de Nowicki e Smutnicki (1995). Consideraram-se situações de 2, 4, 6, 8 e 10 estágios. Para cada configuração de estágios, foram geradas instâncias com 2, 4, 6 e número aleatório de 3 a 5 máquinas por estágio. Isto define 20 grupos de instâncias diferentes, como pode ser observado na Tabela 5.1. Para cada um desses grupos, foram geradas 5 instâncias de 20 tarefas, 5 instâncias de 50 tarefas, 5 instâncias de 100 tarefas e 5 instâncias de 150 tarefas, totalizando $20 \times 5 \times 4 = 400$ instâncias geradas. O tempo de processamento de cada tarefa é inteiro e foi gerado aleatoriamente com distribuição uniforme no intervalo [1,100]. Os algoritmos foram implementados em linguagem C, e executados em um PC 486 com *clock* de 100 Mhz.

## 5.2 PLANO DE COMPARAÇÃO DOS RESULTADOS

### 5.2.1 COMPARAÇÃO COM OUTRAS HEURÍSTICAS

Os resultados obtidos pelas implementações de algoritmos genéticos foram comparados com os resultados de outras heurísticas desenvolvidas para o *flow shop* flexível.

Tabela 5.1: Dimensões das instâncias geradas.

| | Número de máquinas por estágio | | | |
|---|---|---|---|---|
| | 2 | 4 | 6 | aleatório, 3 . . . 5 |
| | 20/2 | 20/2 | 20/2 | 20/2 |
| | 20/4 | 20/4 | 20/4 | 20/4 |
| $n/m$ | 20/6 | 20/6 | 20/6 | 20/6 |
| | 20/8 | 20/8 | 20/8 | 20/8 |
| | 20/10 | 20/10 | 20/10 | 20/10 |
| | 50/2 | 50/2 | 50/2 | 50/2 |
| | 50/4 | 50/4 | 50/4 | 50/4 |
| $n/m$ | 50/6 | 50/6 | 50/6 | 50/6 |
| | 50/8 | 50/8 | 50/8 | 50/8 |
| | 50/10 | 50/10 | 50/10 | 50/10 |
| | 100/2 | 100/2 | 100/2 | 100/2 |
| | 100/4 | 100/4 | 100/4 | 100/4 |
| $n/m$ | 100/6 | 100/6 | 100/6 | 100/6 |
| | 100/8 | 100/8 | 100/8 | 100/8 |
| | 100/10 | 100/10 | 100/10 | 100/10 |
| | 150/2 | 150/2 | 150/2 | 150/2 |
| | 150/4 | 150/4 | 150/4 | 150/4 |
| $n/m$ | 150/6 | 150/6 | 150/6 | 150/6 |
| | 150/8 | 150/8 | 150/8 | 150/8 |
| | 150/10 | 150/10 | 150/10 | 150/10 |

### Algoritmos genéticos × busca local de Nowicki e Smutnicki

Nowicki e Smutnicki (1995) aplicaram buscas local e tabu ao problema de minimizar o *makespan* no ambiente de *flow shop* flexível. As buscas foram comparadas com as heurísticas **DK1**, **DK2** e **DK3** (Ding e Kittichartphayak,1994); **S** (Sawik, 1993); **HS** (Hunsucker e Shah, 1994) e (Wittrock, 1988). O melhor resultado obtido por estas heurísticas definiu o valor de referência com o qual os resultados das buscas foram comparados. As heurísticas **DK1**, **DK2**, **DK3** e **S** estão descritas nos APÊNDICES A e B.

Com exceção da heurística de Wittrock que trata um caso específico de produção de circuito impresso e que não é completamente descrita pelo autor, e de algumas regras de despacho que apresentam mau desempenho, segundo Hunsucker e Shah, todas as demais heurísticas e a busca local foram implementadas nesta dissertação. Assim, os resultados obtidos pelos algoritmos genéticos foram comparados com as principais heurísticas desenvolvidas até o momento, e com a busca local de Nowicki e Smutnicki.

### Algoritmos genéticos × heurística de Guinet

Guinet *et al.* (1995) propuseram uma heurística para dois estágios de processamento. A heurística foi implementada e os algoritmos genéticos também foram comparados com esta heurística que é descrita no APÊNDICE C.

### Algoritmos genéticos × busca local II[1]

Um outro tipo de busca local foi proposto, no qual seqüências de entrada de tarefas são geradas através de uma vizinhança do tipo inserção. A seqüência de tarefas gerada é tratada como nas implementações híbridas de algoritmos genéticos do tipo máquina menos ocupada. A seqüência que serve como ponto de partida para a busca local é obtida da heurística **DK1**. O critério de parada estabelecido foi interromper a busca se pelo menos uma das condições for satisfeita:

- Um ótimo local for alcançado.

[1]Note que a vizinhança utilizada não corresponde à definição clássica de uma vizinhança de solução como ocorre na busca local de Nowicki e Smutnicki.

Tabela 5.2: Dimensões das instâncias geradas para aplicar o *Flowmult*.

| | Número de máquinas por estágio | | | |
|---|---|---|---|---|
| | 2 | 4 | 6 | aleatório, 3...5 |
| $n/m$ | 9/2 | 9/2 | 9/2 | 9/2 |
| | 9/4 | 9/4 | 9/4 | 9/4 |
| | 9/6 | 9/6 | 9/6 | 9/6 |
| | 9/8 | 9/8 | 9/8 | 9/8 |
| | 9/10 | 9/10 | 9/10 | 9/10 |

- Após uma vizinhança ser analisada, o total acumulado de programas analisados exceder 30000. Este é o número de programas analisados pelos algoritmos genéticos.

**Algoritmos genéticos × heurística Flowmult**

Santos *et al.* (1996), desenvolveram uma heurística que procura um programa cujo *makespan* esteja dentro de um desvio $\delta$ em relação ao limitante inferior (2.4). A heurística chega a analisar até $n!$ programas, onde $n$ é o número de tarefas. Esta heurística trata o *flow shop* flexível da mesma forma que a implementação de algoritmo genético híbrido do tipo *fifo*. Uma permutação sobre as $n$ tarefas é colocada na entrada do sistema, que passa a processá-las de acordo com a regra *fifo*. Foram considerados 100 instâncias com apenas 9 tarefas, 5 para cada grupo $n/m$, de acordo com a Tabela 5.2. Apesar de 9 ser um número pequeno, $9! = 362880$, e instâncias com 10 estágios e 6 máquinas paralelas por estágio consumiram, em média, 443 segundos. Se fossem tratadas 10 tarefas, seriam gerados $10 \times 9!$ programas, e o tempo seria de aproximadamente 4430 segundos, ou 73,83 minutos para uma única instância.

### 5.2.2 COMPARAÇÃO COM LIMITANTES INFERIORES

O limitante (2.4) foi utilizado para avaliar o resultado dos algoritmos genéticos.

Tabela 5.3: Desvio médio de uma heurística particular × melhor heurística, quando se varia o número de estágios.

| Estágios | Valores percentuais do desvio | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | DK1 | DK2 | DK3 | S | SPT | LPT | MWR | LWR |
| 2 | 0,68 | 2,46 | 7,02 | 3,90 | 7,70 | 26,83 | 10,30 | 11,85 |
| 4 | 1,66 | 3,69 | 8,25 | 3,98 | 7,26 | 26,72 | 10,12 | 11,23 |
| 6 | 2,78 | 5,02 | 6,86 | 3,27 | 4,86 | 20,30 | 8,06 | 7,88 |
| 8 | 3,43 | 5,09 | 9,43 | 2,29 | 4,24 | 20,42 | 7,20 | 7,54 |
| 10 | 5,93 | 7,20 | 9,00 | 2,02 | 3,00 | 16,84 | 7,05 | 6,02 |
| Média | 2,90 | 4,69 | 8,11 | 3,09 | 5,41 | 22,22 | 8,55 | 8,90 |

## 5.3 ALGORITMOS GENÉTICOS × BUSCA LOCAL DE NOWICKI E SMUTNICKI

### 5.3.1 RESULTADOS DAS HEURÍSTICAS

Foram implementadas as heurísticas **DK1**, **DK2**, **DK3**, **S**, e as seguintes regras de despacho de **HS**: **SPT**, **LPT**, **MWR**, **LWR**. Cada heurística foi comparada com o resultado obtido pela melhor heurística. A comparação foi feita através do desvio relativo:

$$desvio = (\frac{Z_h - Z_r}{Z_r}) \cdot 100\%$$

Na expressão, $Z_h$ é o valor obtido para uma heurística e $Z_r$ é o valor de referência, no caso o resultado da melhor heurística. A Tabela 5.3 mostra a qualidade percentual de cada heurística em relação à melhor, quando se varia o número de estágios. Em cada célula encontra-se a média dos resultados para 20, 50, 100 e 150 tarefas.

A Tabela 5.4 mostra o desvio relativo das heurísticas em relação à melhor heurística, quando se varia o número de tarefas. Cada célula contém resultados médios para 2, 4, 6, 8 e 10 estágios.

A heurística **DK1** mostrou ser, na média, a mais poderosa dentre todas, porém mostra-se sensível ao número de estágios crescente, quando ocorre degradação na qualidade. Embora

Tabela 5.4: Desvio médio de uma heurística particular × melhor heurística, quando se varia o número de tarefas.

| Tarefas | Valores percentuais do desvio | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | **DK1** | **DK2** | **DK3** | **S** | **SPT** | **LPT** | **MWR** | **LWR** |
| **20** | 2,84 | 6,09 | 11,06 | 5,11 | 7,82 | 21,39 | 11,20 | 13,05 |
| **50** | 3,03 | 4,60 | 8,76 | 4,06 | 5,59 | 21,12 | 8,59 | 8,14 |
| **100** | 2,97 | 4,52 | 7,39 | 1,70 | 4,47 | 21,86 | 7,44 | 6,41 |
| **150** | 2,74 | 3,56 | 5,23 | 1,50 | 3,77 | 24,51 | 6,96 | 8,01 |
| **Média** | 2,90 | 4,69 | 8,11 | 3,09 | 5,41 | 22,22 | 8,55 | 8,90 |

Tabela 5.5: Porcentagem do melhor resultado de cada heurístca.

| Valores percentuais de melhores resultados | | | | | | | |
|---|---|---|---|---|---|---|---|
| **DK1** | **DK2** | **DK3** | **S** | **SPT** | **LPT** | **MWR** | **LWR** |
| 31,75 | 11,25 | 6,75 | 40,25 | 13,75 | 1,75 | 3,50 | 4,75 |

as heurísticas **DK3** e **S** sejam muito parecidas, os resultados de **DK3** foram insatisfatórios. A heurística **S** apresenta melhores resultados à medida que o número de estágios e tarefas aumentam, como pode ser observado para 8 e 10 estágios, e 100 e 150 tarefas. Com relação às regras de despacho, nota-se que as regras **SPT**, **MWR** e **LWR** geram soluções de boa qualidade quando o número de estágios é superior a 6 e o número de tarefas é maior ou igual a 100. Em particular, a regra de despacho **SPT** apresenta um desempenho muito bom.

A Tabela 5.5 mostra para cada heurística a porcentagem das 400 instâncias em que apresentou o melhor resultado. Observe que **S** supera as outras heurísticas na maioria das vezes, embora não seja a heurística mais eficiente quando se considera o desvio relativo como parâmetro de comparação. É importante notar também que a soma das porcentagens de ganho é superior a 100%, revelando a existência de empates.

Tabela 5.6: Desvio médio da busca local com soluções iniciais distintas × busca local de melhor resultado, quando se varia o número de estágios.

| **Estágios** | **Valores percentuais do desvio** | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | **DK1** | **DK2** | **DK3** | **S** | **SPT** | **LPT** | **MWR** | **LWR** |
| **2** | 0,08 | 1,94 | 5,44 | 3,72 | 6,02 | 25,86 | 9,43 | 10,06 |
| **4** | 0,46 | 2,63 | 8,83 | 5,27 | 7,52 | 27,96 | 11,13 | 11,46 |
| **6** | 0,25 | 3,45 | 9,47 | 5,88 | 7,59 | 23,30 | 10,78 | 10,42 |
| **8** | 0,25 | 3,06 | 11,62 | 4,58 | 6,12 | 23,29 | 10,03 | 9,42 |
| **10** | 0,32 | 3,20 | 11,93 | 4,44 | 5,79 | 20,01 | 9,99 | 8,90 |
| **Média** | 0,27 | 2,86 | 9,46 | 4,78 | 6,61 | 24,09 | 10,27 | 10,05 |

### 5.3.2 ANÁLISE DA BUSCA LOCAL

A busca local definida por Nowicki e Smutnicki (1995), foi aplicada às soluções iniciais geradas pelas heurísticas **DK1**, **DK2**, **DK3**, **S**, **SPT**, **LPT**, **MWR** e **LWR**. Ela utiliza vizinhança do tipo inserção de tarefa na mesma máquina, e inserção de tarefa em qualquer outra máquina do mesmo estágio. Conhecimento a respeito do caminho crítico, definido no capítulo 5 através de regras, é usado para evitar movimentos de baixa qualidade. O resultado da busca local aplicada a cada solução foi comparado com o melhor resultado de busca local. Para tal foi usado o desvio relativo:

$$desvio = (\frac{Z_{bl} - Z_r}{Z_r}) \cdot 100\%$$

Na expressão, $Z_{bl}$ é o valor obtido pela busca local e $Z_r$ é o valor de referência, no caso o melhor resultado de busca local. A Tabela 5.6 mostra o desvio médio quando se varia o número de estágios. Cada célula mostra a média dos resultados para 20, 50, 100 e 150 tarefas. A Tabela 5.7 mostra o desvio médio quando se varia o número de tarefas.

A busca local aplicada ao resultado da heurística **DK1** mostrou ser a mais eficiente dentre todas. As soluções geradas pelas heurísticas **DK2**, **S** e **SPT** são também pontos de partida promissores, para número de estágios e tarefas maior que 6 e 50, respectivamente.

Tabela 5.7: Desvio médio da busca local com soluções iniciais distintas × busca local de melhor resultado, quando se varia o número de tarefas.

| **Tarefas** | **Valores percentuais do desvio** | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | **DK1** | **DK2** | **DK3** | **S** | **SPT** | **LPT** | **MWR** | **LWR** |
| **20** | 0,58 | 3,53 | 11,36 | 6,28 | 8,33 | 22,15 | 12,24 | 12,69 |
| **50** | 0,15 | 2,69 | 10,36 | 6,14 | 7,22 | 23,69 | 10,98 | 9,69 |
| **100** | 0,14 | 3,09 | 9,41 | 3,69 | 5,87 | 24,25 | 9,52 | 8,39 |
| **150** | 0,21 | 2,12 | 6,71 | 3,01 | 5,01 | 26,25 | 8,36 | 9,20 |
| **Média** | 0,27 | 2,86 | 9,46 | 4,78 | 6,61 | 24,09 | 10,27 | 10,05 |

Tabela 5.8: Porcentagem do melhor resultado da busca local partindo de soluções iniciais distintas.

| **Valores percentuais de melhores resultados** | | | | | | | |
|---|---|---|---|---|---|---|---|
| **DK1** | **DK2** | **DK3** | **S** | **SPT** | **LPT** | **MWR** | **LWR** |
| 85,75 | 14,00 | 3,00 | 11,75 | 4,50 | 1,75 | 2,00 | 1,50 |

Tabela 5.9: Ganho médio do melhor resultado de busca local × melhor heurística.

| Tarefas | Número de estágios | | | | | |
|---|---|---|---|---|---|---|
| | **2** | **4** | **6** | **8** | **10** | **Média** |
| **20** | 1,92 | 3,43 | 5,65 | 3,72 | 3,70 | 3,68 |
| **50** | 2,32 | 2,51 | 4,71 | 4,56 | 3,92 | 3,60 |
| **100** | 0,84 | 2,28 | 3,30 | 3,22 | 4,03 | 2,73 |
| **150** | 0,44 | 1,68 | 1,71 | 2,81 | 3,10 | 1,95 |
| **Média** | 1,69 | 2,74 | 3,84 | 3,58 | 3,69 | 2,99 |

A Tabela 5.8 mostra que ao se usar a solução de **DK1** com ponto de partida para a busca local, atinge-se o melhor resultado em 87,75% das instâncias. Observe também que embora a heurística **S** apresente a melhor solução na maioria das instâncias, como ilustrado na Tabela 5.5, esta não representa o melhor ponto de partida para a busca local, já que alcança o melhor resultado em apenas 11,75% dos casos.

### 5.3.3 BUSCA DE MELHOR RESULTADO × MELHOR HEURÍSTICA

A Tabela 5.9 tem o propósito de revelar o ganho obtido com a aplicação da busca local que gerou o melhor resultado em relação à melhor heurística. O ganho é expresso por:

$$ganho = (\frac{Z_r - Z_{bl}}{Z_r}) \cdot 100\%$$

Na expressão, $Z_{bl}$ é o valor obtido pelo melhor resultado de busca local, e $Z_r$ é o valor de referência, no caso o resultado da melhor heurística. Em geral, ao se aumentar o número de tarefas, o benefício da busca local tende a se tornar menor, embora para número de estágios maior ocorra o oposto.

### 5.3.4 TEMPO COMPUTACIONAL EXIGIDO PELA BUSCA LOCAL

Os tempos computacionais das heurísticas implementadas foram inferiores a 1 segundo, e não são relatados. A Tabela 5.10 apresenta as médias de tempo de execução, em segundos,

Tabela 5.10: Médias do tempo de execução da busca local.

| Tarefas | Número de estágios | | | | |
|---|---|---|---|---|---|
| | **2** | **4** | **6** | **8** | **10** |
| **20** | 0 | 1 | 2 | 3 | 6 |
| **50** | 8 | 17 | 30 | 49 | 69 |
| **100** | 40 | 134 | 204 | 402 | 619 |
| **150** | 289 | 651 | 1335 | 2676 | 3200 |

da busca local aplicada aos resultados das heurísticas. É mostrado no APÊNDICE D que o tamanho da vizinhança da busca local quando se usa o conhecimento do caminho crítico é $\mathcal{O}(\frac{n^2}{c})$,onde $m_l = c$ para todo $l$, é o número de máquinas por estágio. Para uma dada alocação de tarefas a máquinas, o custo para calcular o *makespan* é $mn$, como mostrado no capítulo 3. Portanto, a complexidade de execução da busca local para uma iteração é da ordem de $\mathcal{O}(\frac{n^3 m}{c})$. Espera-se então, que o tempo computacional cresça linearmente com o aumento do número de estágios, e de forma cúbica com o aumento das tarefas. Isto pode ser verificado na Figura 5-1 que mostra o tempo médio computacional como função do número de estágios, quando o número de tarefas é mantido fixo em 100, e na Figura 5-2 que mostra o tempo como função do número de tarefas quando o número de estágios é mantido fixo em 10.

### 5.3.5 DETERMINAÇÃO DOS PARÂMETROS DOS ALGORITMOS GENÉTICOS

**Tamanho da população**

Alguns testes foram feitos para a determinação do tamanho da população, tendo-se observado que para valores abaixo de 400 perdia-se qualidade. O algoritmo convergia muito rápido para uma solução apenas razoável. Valores acima de 400 não trouxeram benefício algum, apenas atrasaram o processo de convergência. Assim, em todas as implementações o tamanho da população foi mantido em 400 e a população inicial foi gerada de forma aleatória.

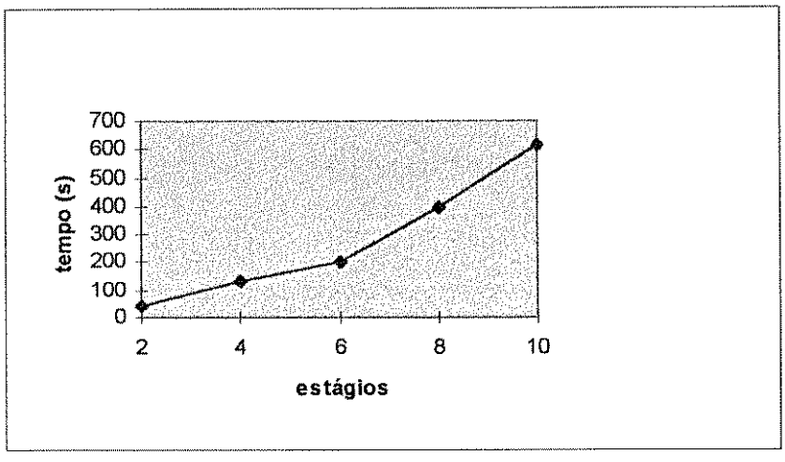



Figura 5-1: Variação do tempo computacional com o número de estágios para 100 tarefas.

Tabela 5.11: Desvio relativo do algoritmos genético × melhor heurística, quando se variam as taxas de mutação e *crossover*.

| | | Crossover | | |
|---|---|---|---|---|
| | | **0,6** | **0,8** | **1,0** |
| **Mu** | **0,6** | 5,29 | 5,53 | 5,24 |
| **ta** | **0,8** | 5,66 | 5,64 | 5,63 |
| **ção** | **1,0** | 6,10 | 6,12 | 6,15 |

**Taxas de crossover e mutação**

Algumas instâncias foram escolhidas para determinar as taxas de *crossover* e mutação. Foram consideradas situações com 2, 4, 6, 8 e 10 estágios, 20, 50 e 150 tarefas, e 2, 4 e 6 máquinas paralelas por estágios. Para cada uma dessas possíveis configurações, foi escolhido uma instância de problema, totalizando $5 \times 3 \times 3 = 45$ instâncias. Foram consideradas taxas de 0,6; 0,8 e 1,0 tanto para mutação como para *crossover*. A implementação de algoritmo genético escolhida para realizar a determinação dos parâmetros foi **Cls + Bak**. A Tabela 5.11 mostra o desvio relativo em relação à melhor heurística dentre as apresentadas na Tabela 5.3.

O melhor resultado foi obtido com 100% de mutação e *crossover*. Observa-se também que o

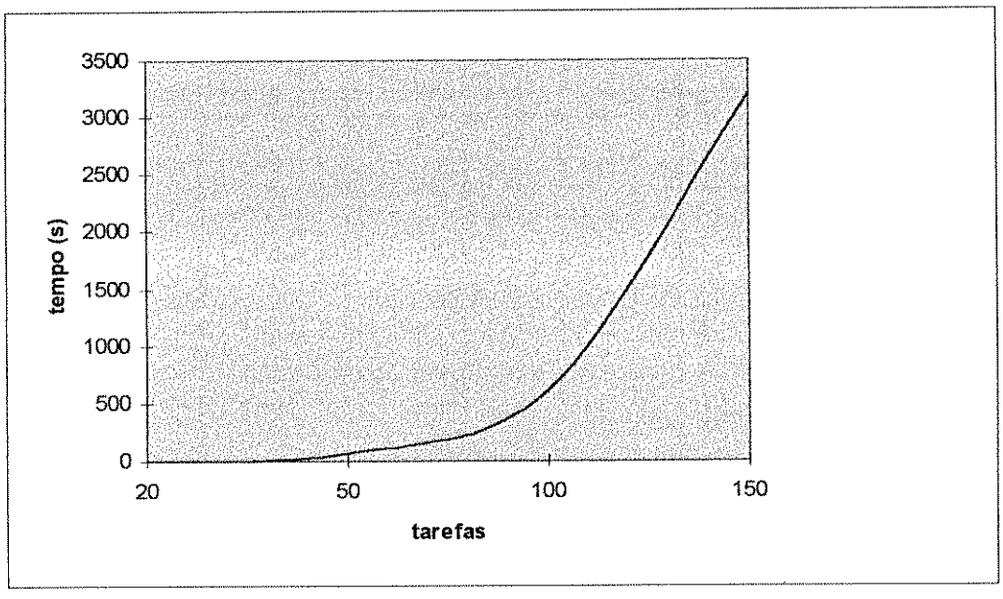



Figura 5-2: Variação do tempo computacional com o número de tarefas para 10 estágios.

algoritmo genético mostra-se mais sensível à variação da taxa de mutação. Baseado nestes resultados, foram adotadas as taxas de 100% de mutação e *crossover* em todas as implementações de algoritmos genéticos. Este resultado também foi observado por Murata *et al.* (1996), que aplicaram algoritmos genéticos ao problema de minimizar o *makespan* no ambiente de *flow shop* de permutação com uma máquina por estágio.

### Critério de parada

O critério de parada utilizado foi estabelecer um limite máximo de 15000 casamentos para todas as implementações. Sendo assim, 30000 programas foram analisados para cada instância, desde que cada casamento gera dois descendentes. As figuras 5-3, 5-4 e 5-5 mostram a evolução da busca para 20, 50 e 100 tarefas respectivamente, e para número de estágios igual a 10. A implementação de algoritmo genético em questão é **Cls + Bak.** Nesta implementação são produzidas 75 gerações a partir de uma população de 400 indivíduos, totalizando $400 \times 75 = 30000$ programas analisados. Observe que por volta de 35 gerações a busca já está praticamente estabilizada. Um teste adicional envolveu 75 e 150 iterações da implementação **Cls + Bak.** Para 75 gerações, o algoritmo genético foi 5,65% superior à melhor heurística

dentre as apresentadas na Tabela 5.3, e para 150 gerações, o algoritmo genético foi 6,04% melhor do que o valor de referência, mostrando uma diferença de apenas 0,39%.

Ao resultado da implementação **Cls + Bak** com 75 gerações, aplicou-se a busca local de Nowicki e Smutinicki. O valor de referência utilizado foi o melhor resultado de busca local de Nowicki e Smutnicki, quando os pontos de partida para a busca eram os resultados das heurísticas apresentadas na Tabela 5.3. Para 20, 50 e 100 tarefas, o algoritmo genético foi 3,34% melhor que a referência, enquanto que a busca local partindo do resultado do algoritmo genético foi 3,40% melhor que a referência, mostrando que o ganho obtido ao se aplicar esta busca local é muito pequeno.

A busca local II também foi aplicada ao resultado da implementação **Cls + Bak** com 75 gerações. O valor de referência utilizado foi o melhor resultado da busca local de Nowicki e Smutnicki, quando os pontos de partida para a busca eram os resultados das heurísticas apresentadas na Tabela 5.3. Para 20, 50, 100 e 150 tarefas, o algoritmo genético foi 5,65% melhor que o valor de referência, enquanto que a busca local II partindo do resultado do algoritmo genético foi 5,71% melhor que a referência, mostrando que o ganho aqui também é muito pequeno.

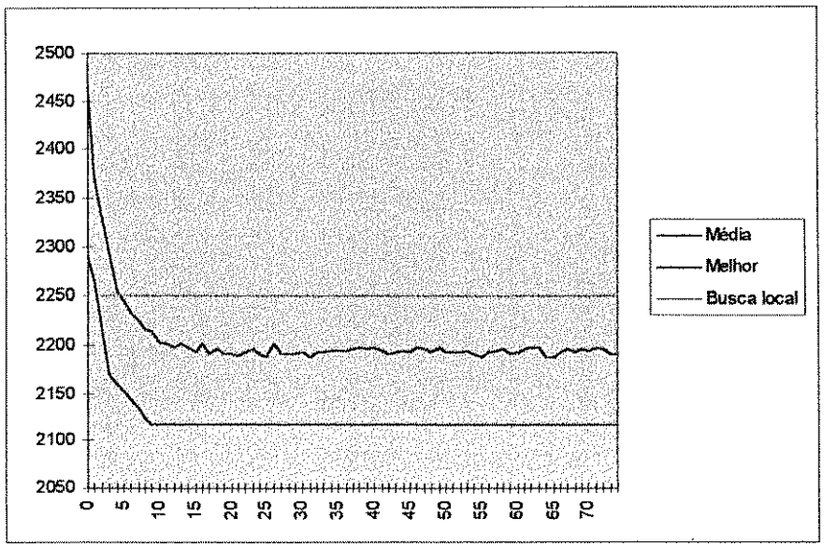



Figura 5-3: Evolução do algoritmo genético ao longo das gerações, para 10 estágios e 20 tarefas.

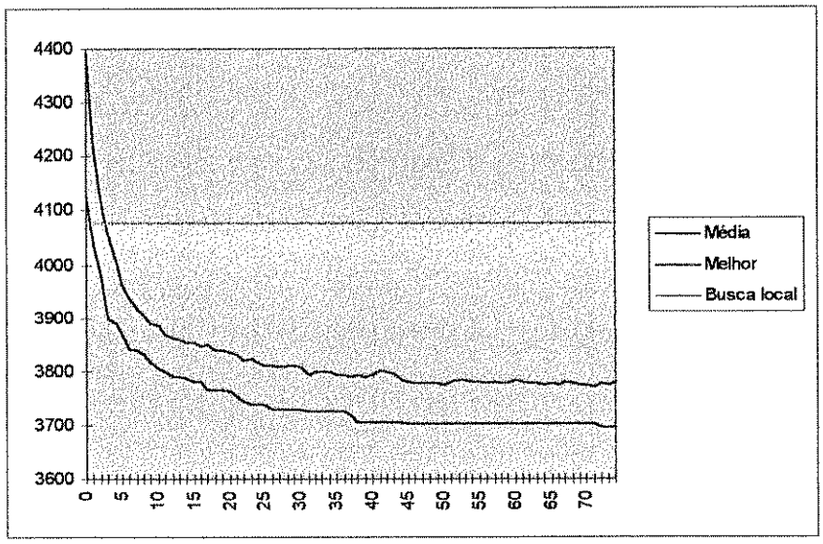



Figura 5-4: Evolução do algoritmo genético ao longo das gerações, para 10 estágios e 50 tarefas.

### 5.3.6 ANÁLISE DOS ALGORITMOS GENÉTICOS MAL-SUCEDIDOS

Os algoritmos genéticos que não apresentaram resultados satisfatórios foram:

- **Puro A:** Implementação pura do tipo A com cromossomo do tipo I.
- **Puro $B_I$:** Implementação pura do tipo B com cromossomo do tipo I.
- **Puro $B_{II}$:** Implementação pura do tipo B com cromossomo do tipo II.
- **BC + A:** Implementação do tipo A com cromossomo do tipo I, baseada em conhecimento do caminho crítico.
- **BC + $B_I$:** Implementação do tipo B com cromossomo do tipo I, baseada em conhecimento do caminho crítico.
- **Fifo:** Implementação híbrida com uso da regra de alocação *fifo*.

Os algoritmos genéticos mal sucedidos foram comparados com o melhor resultado obtido dentre todas as heurísticas apresentadas na Tabela 5.4. A comparação foi feita usando a expressão de desvio relativo:

$$desvio = (\frac{Z_g - Z_r}{Z_r}) \cdot 100\%$$

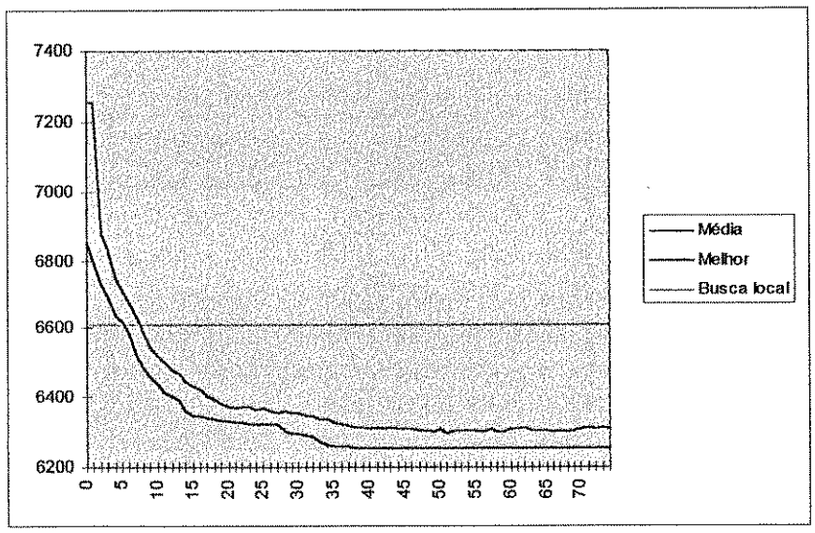



Figura 5-5: Evolução do algoritmo genético ao longo das gerações, para 10 estágios e 100 tarefas.

Na expressão, $Z_g$ é o valor obtido para um algoritmo genético, e $Z_r$ é o valor de referência, no caso o resultado da melhor heurística. A Tabela 5.12 mostra a qualidade percentual de cada algoritmo genético em relação à melhor heurística, quando se varia o número de estágios. Cada célula indica a média dos resultados para 20, 50 e 100 tarefas.

A Tabela 5.13 mostra o desvio relativo dos algoritmos genéticos mal sucedidos em relação à melhor heurística, quando se varia o número de tarefas.

Os algoritmos genéticos **Puro A**, **Puro $B_I$**, **BC + A**, **BC + $B_I$** e **Puro $B_{II}$** são muito ruins e podem ser descartados. O único a conseguir melhoria em relação à melhor heurística foi a implementação do tipo **Fifo**, porém o benefício é pequeno, não justificando também o seu uso. A implementação do tipo **Fifo** obtém um ganho médio de 3,60% sobre a melhor heurística enquanto que a busca local obtém um ganho médio de 3,34%, em relação à melhor heurística. Este valor é obtido da Tabela 5.9, e corresponde à média dos ganhos para 20, 50 e 100 tarefas.

### Tempo computacional

A Tabela 5.14 mostra as médias de tempo de execução, em segundos, dos algoritmos genéticos. Como já mencionado, a ordem de execução do cálculo do *makespan* é $\mathcal{O}(nm)$, exceto para as implementação híbridas para as quais o cálculo do *makespan* é $\mathcal{O}(n\sum_{l=1}^{m} m_l)$. Se $m_l$

Tabela 5.12: Desvio médio dos algoritmos genéticos mal sucedidos × melhor heurística, quando se varia o número de estágios.

| **Estágios** | **Valores percentuais de erro** | | | | | |
|---|---|---|---|---|---|---|
| | **Puro A** | **Puro $B_I$** | **BC + A** | **BC + $B_I$** | **Puro $B_{II}$** | **Fifo** |
| **2** | 8,40 | 35,70 | 5,38 | 21,42 | 18,01 | -1,02 |
| **4** | 26,71 | 116,32 | 26,64 | 112,80 | 47,10 | -2,02 |
| **6** | 52,74 | 175,86 | 47,21 | 161,23 | 77,32 | -5,25 |
| **8** | 68,50 | 227,87 | 59,94 | 205,54 | 94,11 | -4,08 |
| **10** | 83,96 | 265,91 | 75,06 | 239,10 | 107,60 | -5,62 |
| **Média** | 48,06 | 164,33 | 42,85 | 148,02 | 68,83 | -3,60 |

Tabela 5.13: Desvio médio dos algoritmos genéticos mal sucedidos × melhor heurística, quando se varia o número de tarefas.

| **Tarefas** | **Valores percentuais de erro** | | | | | |
|---|---|---|---|---|---|---|
| | **Puro A** | **Puro $B_I$** | **BC + A** | **BC + $B_I$** | **Puro $B_{II}$** | **Fifo** |
| **20** | 20,96 | 81,46 | 17,96 | 72,15 | 30,97 | -5,30 |
| **50** | 52,79 | 173,64 | 47,67 | 158,38 | 74,67 | -3,44 |
| **100** | 70,44 | 237,89 | 62,90 | 213,53 | 100,85 | -2,05 |
| **Média** | 48,06 | 164,33 | 42,85 | 148,02 | 68,83 | -3,60 |

for constante e igual a $c$, tem-se $\mathcal{O}(ncm)$. Isto acontece, porque nas implementações híbridas, a regra de alocação precisa encontrar a máquina menos ocupada no estágio, devendo então analisar todas as $m_l$ máquinas de cada estágio para então alocar a tarefa. Espera-se então que o tempo computacional cresça linearmente com o número de estágios, e com o número de tarefas. O algoritmo que realiza a geração da população inicial tem complexidade não linear em relação ao número de tarefas, devido ao caráter probabilístico da geração de números aleatórios, e o algoritmo que realiza o *crossover*, tem complexidade $\mathcal{O}(n^2)$, como mostrado no APÊNDICE E. Estes comportamentos podem ser analisados no gráfico da Figura 5-6 que mostra o tempo computacional como função do número de estágios, quando o número de tarefas é mantido fixo em 100, e na Figura 5-7 que mostra o tempo como função do número de tarefas quando o número de estágios é mantido fixo em 10.

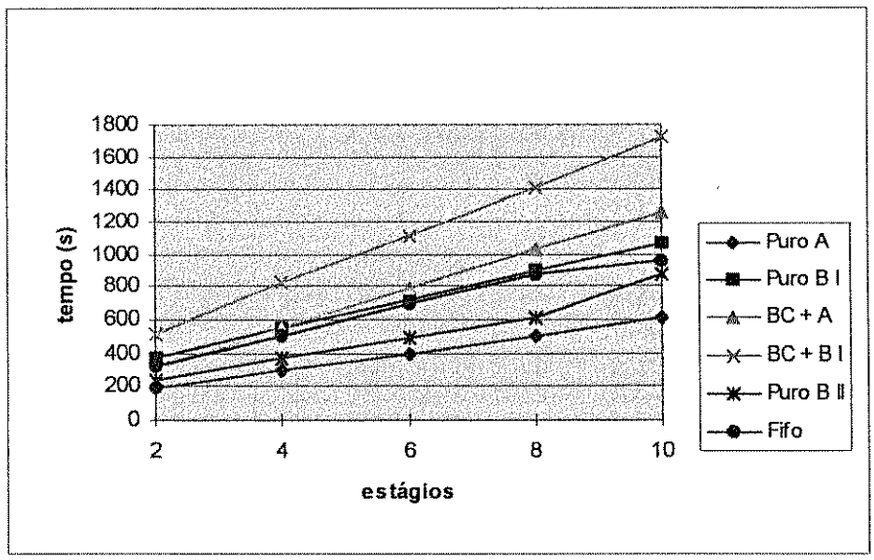



Figura 5-6: Variação do tempo computacional com o número de estágios para 100 tarefas.

### 5.3.7 ANÁLISE DOS ALGORITMOS GENÉTICOS BEM-SUCEDIDOS

As implementações de algoritmos genéticos que apresentaram os melhores resultados foram as implementações híbridas do tipo máquina menos ocupada, com as seguintes características:

Tabela 5.14: Médias de tempo de execução, em segundos, dos algoritmos genéticos mal sucedidos.

| Tempo (s) | Número de estágios×número de tarefas | | | | | | |
|---|---|---|---|---|---|---|---|
| | 2×20 | 2×50 | 2×100 | 4×20 | 4×50 | 4×100 | 6×20 |
| **Puro A** | 44 | 90 | 189 | 71 | 79 | 296 | 91 |
| **Puro $B_I$** | 66 | 153 | 378 | 105 | 238 | 550 | 144 |
| **BC + A** | 85 | 163 | 318 | 142 | 286 | 549 | 199 |
| **BC + $B_I$** | 130 | 223 | 514 | 179 | 454 | 830 | 253 |
| **Puro $B_{II}$** | 54 | 113 | 245 | 84 | 177 | 367 | 114 |
| **Fifo** | 52 | 135 | 330 | 89 | 222 | 510 | 126 |

| Tempo (s) | Número de estágios × número de tarefas | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 6×50 | 6×100 | 8×20 | 8×50 | 8×100 | 10×20 | 10×50 | 10×100 |
| **Puro A** | 203 | 402 | 124 | 260 | 510 | 150 | 317 | 616 |
| **Puro $B_I$** | 323 | 723 | 183 | 410 | 897 | 222 | 496 | 1070 |
| **BC + A** | 411 | 788 | 257 | 533 | 1027 | 315 | 659 | 1263 |
| **BC + $B_I$** | 531 | 1113 | 316 | 687 | 1421 | 387 | 841 | 1734 |
| **Puro $B_{II}$** | 242 | 492 | 145 | 308 | 618 | 174 | 373 | 881 |
| **Fifo** | 313 | 695 | 163 | 402 | 881 | 200 | 491 | 958 |

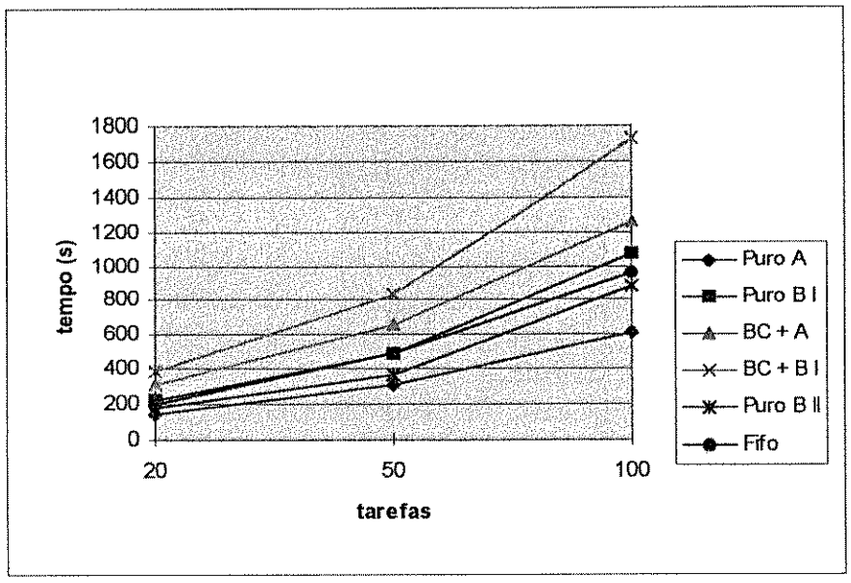



Figura 5-7: Variação do tempo computacional com o número de tarefas para 10 estágios.

- **Ack + Rvs:** Implementação na qual o critério de seleção é baseado em classificação (3.5), e o corpo do algoritmo genético segue a proposta de substituição em regime permanente proposta por Ackley (1987).

- **Ack + Fit:** Implementação na qual o critério de seleção é baseado no valor do *fitness* (3.2), e o corpo do algoritmo genético segue a proposta de substituição em regime permanente proposta por Ackley (1987).

- **Cls + Bak:** Implementação híbrida onde o critério de seleção é baseado em classificação, Figura 3-3, e o corpo do algoritmo genético segue a proposta de Holland (1975).

- **Cls + Fit:** Implementação híbrida onde o critério de seleção é baseado em *fitness* (3.2), e o corpo do algoritmo genético segue a proposta de Holland (1975).

O resultado de cada um foi comparado com o melhor resultado de busca local obtido. A comparação foi feita usando a expressão do ganho relativo:

$$ganho = (\frac{Z_r - Z_g}{Z_r}) \cdot 100\%$$

Tabela 5.15: Ganho médio dos algoritmos genéticos bem sucedidos.

| | **Número de estágios × número de tarefas** | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 2×20 | 2×50 | 2×100 | 2×150 | 4×20 | 4×50 | 4×100 | 4×150 | 6×20 | 6×50 |
| **Ack + Rvs** | 2,49 | 0,50 | 0,09 | -0,01 | 2,83 | 2,60 | 1,33 | 1,03 | 4,53 | 4,12 |
| **Ack + Fit** | 2,29 | 0,38 | 0,09 | 0,01 | 4,21 | 2,33 | 1,15 | 0,74 | 4,21 | 3,66 |
| **Cls + Bak** | 2,81 | 0,61 | 0,12 | 0,08 | 4,33 | 2,94 | 1,83 | 1,31 | 4,44 | 5,02 |
| **Cls + Fit** | 0,02 | -0,57 | -0,33 | -0,20 | 2,33 | 0,67 | -0,16 | 0,16 | 2,07 | 1,07 |

| | **Número de estágios × número de tarefas** | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 6×100 | 6×150 | 8×20 | 8×50 | 8×100 | 8×150 | 10×20 | 10×50 | 10×100 | 10×150 |
| **Ack + Rvs** | 2,56 | 1,35 | 4,03 | 4,59 | 2,63 | 1,48 | 2,57 | 4,20 | 3,08 | 1,75 |
| **Ack + Fit** | 1,87 | 0,83 | 3,90 | 3,96 | 2,49 | 1,46 | 3,22 | 3,81 | 2,73 | 1,62 |
| **Cls + Bak** | 3,34 | 1,41 | 3,91 | 5,10 | 3,43 | 2,22 | 3,44 | 4,95 | 4,18 | 2,99 |
| **Cls + Fit** | 0,20 | -0,65 | 1,58 | 3,43 | 0,53 | 0,13 | 1,00 | 1,38 | 0,74 | 1,29 |

Na expressão, $Z_g$ é o valor obtido para um algoritmo genético, e $Z_r$ é o valor de referência, no caso o resultado da busca local de melhor resultado, quando os pontos de partida são os resultados das heurísticas apresentadas na Tabela 5.3.

A Tabela 5.15 apresenta o ganho relativo dos algoritmos genéticos bem sucedidos. Cada célula na tabela mostra o ganho médio dos resultados das instâncias com 2, 4, 6 e número aleatório entre 3 e 5 máquinas paralelas.

A Tabela 5.16 mostra o ganho relativo quando se varia o número de estágios. Em cada célula encontra-se o ganho médio dos resultados para 20, 50, 100 e 150 tarefas. A Tabela 5.17 mostra o ganho relativo dos algoritmos genéticos bem sucedidos em relação ao melhor resultado de busca local, quando se varia o número de tarefas. Em cada célula encontra-se o ganho relativo médio dos resultados para 2, 4, 6, 8 e 10 estágios.

A partir da Tabela 5.15, conlui-se que **Cls + Bak** obteve os melhores resultados. Das Tabelas 5.16 e 5.17, conclui-se que todas as implementações apresentam ganhos maiores com o aumento do número de estágios e ganhos menores à medida que o número de tarefas aumenta. A implementação **Cls + Fit** é a menos eficiente, tendo sendo inferior à busca local quando 2

Tabela 5.16: Ganho médio dos algoritmos genéticos bem sucedidos × melhor resultado de busca local, quando se varia o número de estágios.

| **Estágios** | **Valores percentuais** | | | |
|---|---|---|---|---|
| | **Ack + Rvs** | **Ack + Fit** | **Cls + Bak** | **Cls + Fit** |
| **2** | 0,77 | 0,74 | 0,82 | -0,27 |
| **4** | 1,95 | 2,12 | 2,60 | 0,75 |
| **6** | 3,14 | 2,55 | 3,56 | 0,67 |
| **8** | 3,18 | 2,83 | 3,66 | 1,42 |
| **10** | 2,90 | 2,60 | 3,89 | 1,10 |
| **Média** | 2,39 | 2,17 | 2,91 | 0,73 |

Tabela 5.17: Ganho médio dos algoritmos genéticos bem sucedidos × melhor resultado de busca local, quando se varia o número de tarefas.

| **Tarefas** | **Valores percentuais** | | | |
|---|---|---|---|---|
| | **Ack + Rvs** | **Ack + Fit** | **Cls + Bak** | **Cls + Fit** |
| **20** | 3,29 | 3,48 | 3,79 | 1,40 |
| **50** | 3,20 | 2,77 | 3,72 | 1,20 |
| **100** | 1,94 | 1,67 | 2,58 | 0,20 |
| **150** | 1,12 | 0,75 | 1,60 | 0,14 |
| **Média** | 2,39 | 2,17 | 2,91 | 0,73 |

estágios são considerados.

A Tabela 5.18 confronta os resultados obtidos pelo algoritmo genético **Cls + Bak** com a melhor heurística dentre as apresentadas na Tabela 5.3. A comparação foi feita usando a expressão de ganho relativo:

$$ganho = (\frac{Z_r - Z_g}{Z_r}) \cdot 100\%$$

Na expressão, $Z_g$ é o valor obtido pelo algoritmo genético, e $Z_r$ é o valor de referência, no caso o resultado da melhor heurística dentre as apresentadas na Tabela 5.3. Esta Tabela também apresenta a iteração média na qual o melhor resultado foi alcançado.

Na média sobre estas instâncias, o algoritmo genético foi 5,63% melhor do que o valor de referência. Para instâncias geradas da mesma forma, Nowicki e Smutnicki relatam que a busca tabu obteve um ganho de 5,78% em relação ao valor de referência. Se ao invés de 75 gerações, forem consideradas 150 gerações, o algoritmo genético passa a ser 6,07% melhor do que o valor de referência.

Note que, em geral, à medida que o número de tarefas, estágios, e máquinas paralelas aumentam, são necessárias mais iterações para se alcançar a melhor solução.

### 5.3.8 TEMPO COMPUTACIONAL

A Tabela 5.19 revela as médias de tempo de execução, em segundos, dos algoritmos genéticos bem sucedidos. Não existe grande variação nos tempos para todas as implementações. As Figuras 5-8 e 5-9 trazem curvas que representam os resultados.

#### Uma implementação mais eficiente do algoritmo genético Cls + Bak

Com o intuito de obter resultados em tempo menor, uma implementação mais eficiente do algoritmo genético híbrido **Cls + Bak** foi codificada. A implementação envolveu melhoria no algoritmo de *crossover* que de complexidade quadrática, passou a ser linear em relação ao número de tarefas, e melhoria no algoritmo de geração da população inicial, que embora continue sendo não linear, ficou mais rápido, como pode ser observado nas Figuras 5-11 e 5-10. A descrição das complexidades dos algoritmos de *crossover* e geração da população inicial está no APÊNDICE E. Com esta implementação eficiente, é possivel mostrar que o algoritmo

Tabela 5.18: Ganho médio do algoritmo genético **Cls + Bak** quando o valor de referência é a melhor heurística e média da iteração que apresenta a melhor solução.

| **Cls + Bak** | | | | | | |
|---|---|---|---|---|---|---|
| | **ganho** | | | **iteração** | | |
| $n/m$ | **n. de máquinas** | | | **n. de máquinas** | | |
| | **2** | **4** | **6** | **2** | **4** | **6** |
| **20/2** | 0,6 | 4,8 | 5,1 | 5 | 31 | 38 |
| **20/4** | 5,7 | 8,9 | 3,4 | 24 | 35 | 8 |
| **20/6** | 9,7 | 11,2 | 7,4 | 45 | 58 | 19 |
| **20/8** | 9,3 | 7,7 | 4,4 | 39 | 37 | 12 |
| **20/10** | 8,9 | 5,8 | 3,4 | 34 | 27 | 15 |
| **50/2** | 0,6 | 4,8 | 5,1 | 8 | 30 | 57 |
| **50/4** | 2,6 | 5,6 | 7,5 | 19 | 58 | 64 |
| **50/6** | 7,9 | 9,7 | 9,9 | 56 | 67 | 69 |
| **50/8** | 9,5 | 10,2 | 8,7 | 54 | 70 | 70 |
| **50/10** | 9,0 | 8,1 | 7,7 | 52 | 74 | 69 |
| **100/2** | 0,3 | 0,7 | 1,7 | 24 | 43 | 47 |
| **100/4** | 2,3 | 4,6 | 6,6 | 47 | 60 | 72 |
| **100/6** | 4,9 | 7,3 | 8,4 | 52 | 67 | 72 |
| **100/8** | 5,2 | 7,3 | 8,4 | 59 | 70 | 70 |
| **100/10** | 6,5 | 7,9 | 8,6 | 68 | 73 | 71 |
| **150/2** | 0,0 | 0,0 | 0,4 | 45 | 39 | 56 |
| **150/4** | 1,3 | 1,7 | 2,4 | 59 | 62 | 71 |
| **150/6** | 1,7 | 1,8 | 1,6 | 55 | 74 | 73 |
| **150/8** | 3,0 | 4,4 | 2,8 | 62 | 69 | 73 |
| **150/10** | 3,1 | 4,2 | 3,2 | 71 | 71 | 73 |

Tabela 5.19: Média dos tempos computacionais dos algoritmos genéticos bem sucedidos.

| Tempo(s) | Número de estágios × número de tarefas | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 2×20 | 2×50 | 2×100 | 2×150 | 4×20 | 4×50 | 4×100 | 4×150 | 6×20 | 6×50 |
| **Ack+Rnk** | 46 | 119 | 296 | 679 | 58 | 191 | 445 | 860 | 109 | 223 |
| **Ack+Fit** | 48 | 125 | 308 | 719 | 60 | 200 | 462 | 911 | 116 | 236 |
| **Cls+Bak** | 52 | 126 | 312 | 722 | 84 | 202 | 466 | 915 | 117 | 280 |
| **Cls+Fit** | 49 | 120 | 296 | 677 | 79 | 192 | 444 | 858 | 111 | 267 |

| Tempo(s) | Número de estágios × número de tarefas | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 6×100 | 6×150 | 8×20 | 8×50 | 8×100 | 8×150 | 10×20 | 10×50 | 10×100 | 10×150 |
| **Ack+Rnk** | 599 | 1070 | 141 | 343 | 755 | 1284 | 172 | 417 | 906 | 1483 |
| **Ack+Fit** | 629 | 1133 | 148 | 360 | 793 | 1359 | 181 | 446 | 956 | 1570 |
| **Cls+Bak** | 631 | 1138 | 150 | 362 | 798 | 1435 | 183 | 448 | 961 | 1578 |
| **Cls+Fit** | 597 | 1067 | 143 | 343 | 751 | 1280 | 174 | 417 | 901 | 1479 |

genético **Cls + Bak** é superior à busca local, no tempo computacional. Enquanto a busca local tem complexidade de ordem cúbica em relação ao número de tarefas, o algoritmo genético é não linear na geração da população inicial em relação ao número de tarefas, e linear em relação ao número de tarefas e de máquinas nos demais processamentos, como mostra a Figura 5-9. Para número de tarefas abaixo de 50, a busca local é bem mais rápida, porém com o aumento do número de tarefas, o tempo computacional exigido torna a busca local muito demorada, como mostra a Figura 5-12. Para melhorar o desempenho de analisar toda a vizinhança de uma solução, Nowicki e Smutnicki (1995) propuseram um mecanismo de cálculo de *makespan* que torna o processo muito mais rápido. Como este mecanismo não é de fácil implementação, a busca tabu não foi implementada, pois o tempo computacional exigido seria inviável.

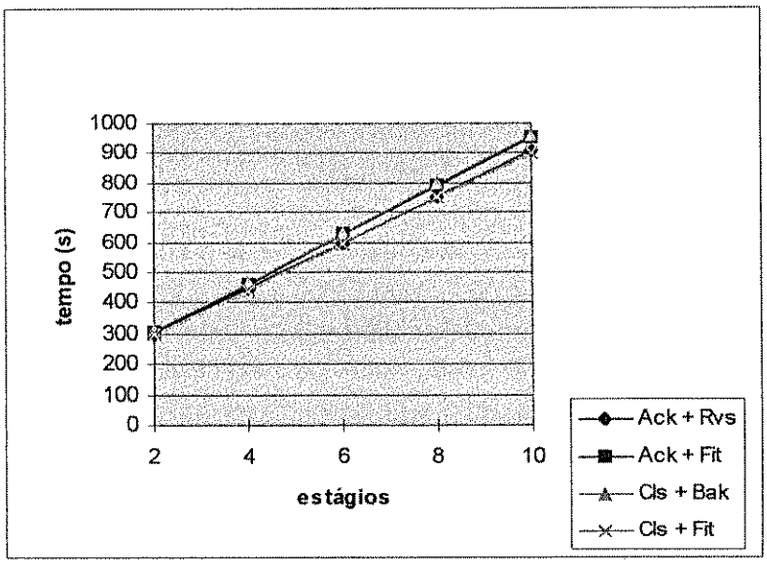



Figura 5-8: Variação do tempo computacional com o número de estágios para 100 tarefas.

## 5.4 ALGORITMO GENÉTICO × LIMITANTE INFERIOR

### 5.4.1 COMPARAÇÃO ENTRE OS LIMITANTES INFERIORES

A Tabela 5.20 tem o objetivo de mostrar quão melhor é o $LI2C_{\max}$ em relação ao $LI1C_{\max}$. A comparação foi feita usando a expressão de desvio relativo:

$$desvio = (\frac{Z_r - Z_{li}}{Z_r}) \cdot 100\%$$

Na expressão, $Z_{li}$ é o valor obtido por $LI2C_{\max}$, e $Z_r$ é o valor de referência, no caso, o limitante inferior $LI1C_{\max}$.

O limitante $LI2C_{\max}$, mostrou-se mais eficiente com o aumento do número de estágios, porém o desvio torna-se menor com o aumento do número de tarefas.

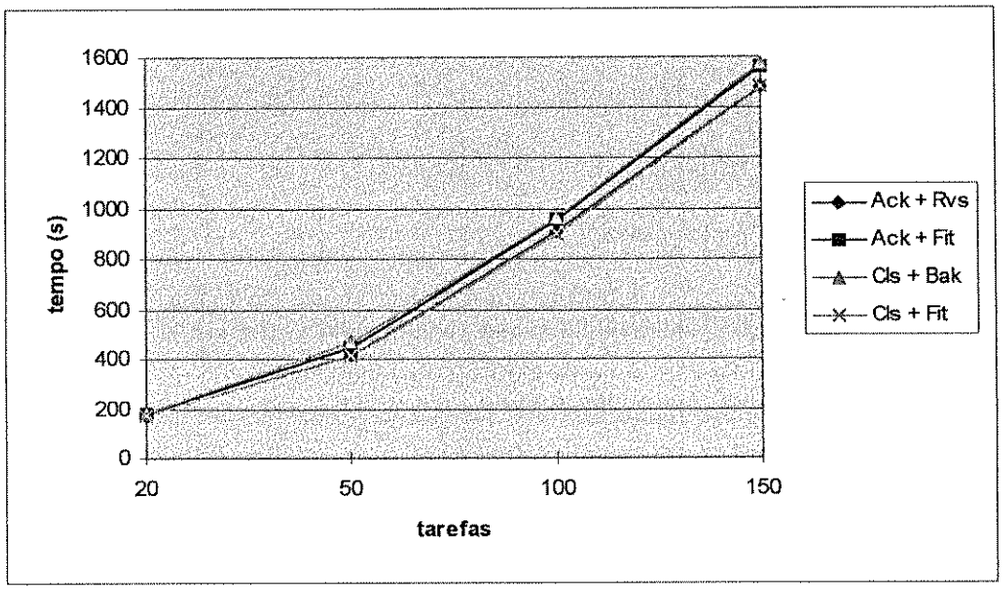



Figura 5-9: Variação do tempo computacional com o número de tarefas para 10 estágios.

### 5.4.2 LIMITANTE INFERIOR $LI2C_{\max}$× ALGORITMO GENÉTICO

O resultado da Implementação **Cls + Bak** foi comparado com o limitante inferior $LI2C_{\max}$. A comparação foi feita usando a expressão de desvio relativo:

$$desvio = (\frac{Z_g - Z_r}{Z_r}) \cdot 100\%$$

Na expressão, $Z_g$ é o valor obtido para o algoritmo genético, e $Z_r$ é o valor de referência no caso o limitante inferior $LI2C_{\max}$. A Tabela 5.21 mostra a qualidade percentual do algoritmo genético em relação ao limitante inferior, quando se varia o número de estágios e o número de tarefas. Em cada célula está a média dos resultados para 2, 4, 6 e número aleatório entre 3 e 5 máquinas por estágio.

Com o número crescente de tarefas, o resultado do algoritmo genético se aproxima do limitante inferior, porém a análise fica difícil com o aumento do número de estágios, pois em geral, limitantes inferiores degradam-se com o aumento do número de estágios. Em 23 instâncias, o resultado do algoritmo genético se igualou ao do limitante inferior, sendo então ótimas estas soluções.

Tabela 5.20: Desvio médio do limitante inferior $LI2C_{\max}$ em relação ao limitante inferior $LI1C_{\max}$, quando se varia o número de tarefas e estágios.

| **Tarefas** | **Número de estágios** | | | | | |
|---|---|---|---|---|---|---|
| | **2** | **4** | **6** | **8** | **10** | **Média** |
| **20** | 3,51 | 18,50 | 36,23 | 57,56 | 75,49 | 38,26 |
| **50** | 0,93 | 5,98 | 13,02 | 22,31 | 31,30 | 14,71 |
| **100** | 0,12 | 2,22 | 6,02 | 10,07 | 14,64 | 6,61 |
| **150** | 0,08 | 1,20 | 4,03 | 6,57 | 9,77 | 4,33 |
| **Média** | 1,16 | 6,98 | 14,82 | 24,13 | 32,80 | 15,98 |

Tabela 5.21: Desvio médio do algoritmo genético **Cls + Bak** em relação ao limitante inferior $LI2C_{\max}$, quando se varia o número de tarefas e estágios.

| **Tarefas** | **Número de estágios** | | | | | |
|---|---|---|---|---|---|---|
| | **2** | **4** | **6** | **8** | **10** | **Média** |
| **20** | 1,67 | 9,62 | 16,05 | 19,72 | 20,50 | 13,57 |
| **50** | 0,48 | 1,86 | 7,87 | 11,32 | 14,74 | 7,28 |
| **100** | 0,30 | 1,14 | 3,67 | 6,68 | 9,65 | 4,31 |
| **150** | 0,18 | 1,49 | 2,88 | 5,74 | 7,55 | 3,57 |
| **Média** | 0,66 | 3,53 | 7,62 | 10,87 | 13,11 | 7,16 |

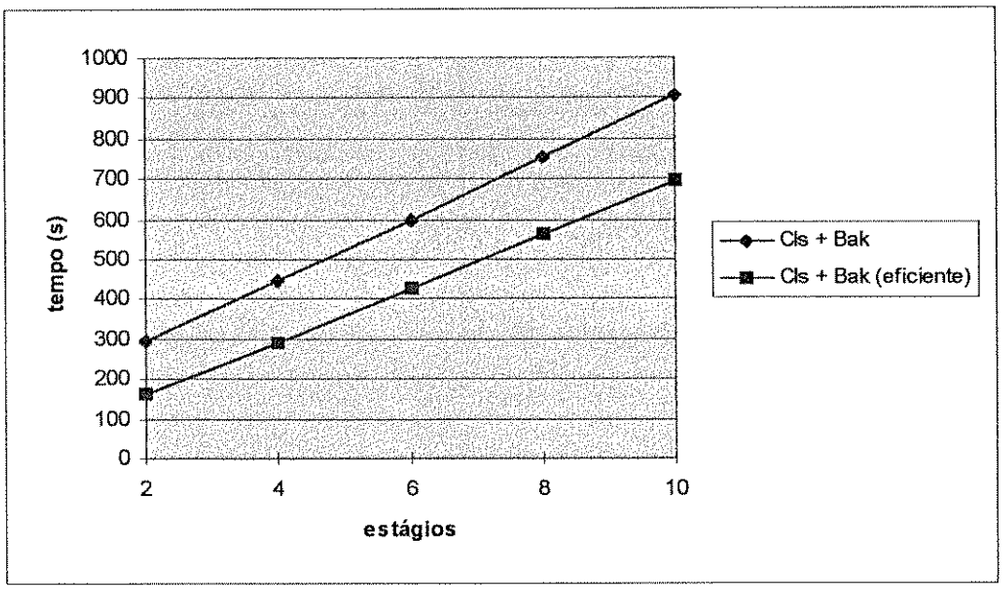



Figura 5-10: Variação do tempo computacional com o número de estágios para 100 tarefas.

## 5.5 ALGORITMO GENÉTICO × HEURÍSTICA DE GUINET

A heurística de Guinet para o *flow shop* flexível com 2 estágios, é um dos trabalhos mais recentes até o momento da composição desta dissertação, e merece ser comparada com os resultados dos algoritmos genéticos. A Tabela 5.22 traz a comparação dos resultados da implementação **Cls + Bak** com a heurística de Guinet. A comparação foi feita usando a expressão de ganho relativo:

$$ganho = (\frac{Z_r - Z_g}{Z_r}) \cdot 100\%$$

Na expressão, $Z_g$ é o valor obtido pelo algoritmo genético, e $Z_r$ é o valor de referência, no caso o resultado da heurística de Guinet, ou o resultado da heurística **DK1**.

Embora a heurística de Guinet seja mais elaborada do que a heurística **DK1**, o seu desempenho foi inferior. Note que para 20 tarefas, os resultados da heurística de Guinet são de baixa qualidade.

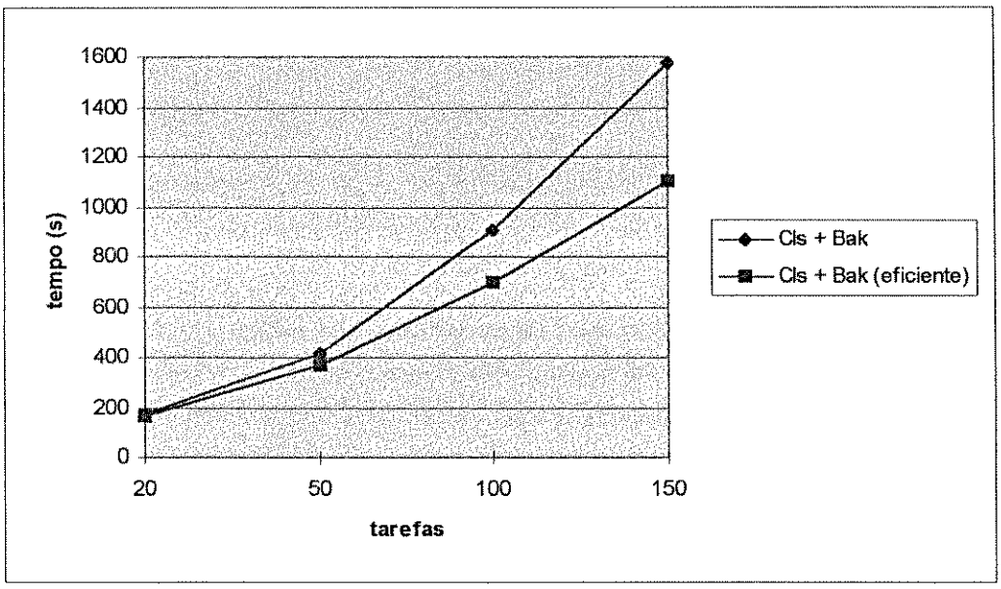



Figura 5-11: Variação do tempo computacional com o número de tarefas para 10 estágios.

## 5.6 ALGORITMO GENÉTICO × BUSCA LOCAL II

O propósito da busca local II foi verificar se uma busca local com uso de vizinhança do tipo inserção, e que analisa seqüências de entrada como as implementações híbridas de algoritmos genéticos do tipo máquina menos ocupada, seria capaz de gerar soluções que pudessem competir com aquelas dos algoritmos genéticos. A comparação foi feita usando a expressão de ganho relativo:

$$ganho = (\frac{Z_r - Z_g}{Z_r}) \cdot 100\%$$

Na expressão, $Z_g$ é o valor obtido pelo algoritmo genético, pela busca local de Nowicki e Smutnicki ou pela busca local II, e $Z_r$ é o valor de referência, no caso o resultado da heurística **DK1**.

A Tabela 5.23 mostra o ganho relativo quando se varia o número de tarefas. Em cada célula encontra-se o ganho médio dos resultados para 2, 4, 6, 8 e 10 estágios. A Tabela 5.24 mostra o ganho relativo da implementação **Cls + Bak**, da busca local II e da busca local de Nowicki e Smutnicki, em relação à heurística **DK1**, quando se varia o número de estágios. Em cada célula encontra-se o ganho relativo médio dos resultados para 20, 50, 100 e 150 tarefas.

Tabela 5.22: Ganho médio do algoritmo genético **Cls + Bak** em relação à heurística de Guinet e **DK1**.

| **Tarefas** | **Número de estágios** | |
|---|---|---|
| | **2** | |
| | **Guinet** | **DK1** |
| **20** | 8,92 | 4,93 |
| **50** | 4,54 | 4,71 |
| **100** | 1,69 | 1,55 |
| **150** | 1,18 | 1,07 |
| **Média** | 4,04 | 3,06 |

Tabela 5.23: Ganho médio em relação ao resultado da heurística **DK1** , quando se varia o número de tarefas.

| **Tarefas** | **Valores percentuais** | | |
|---|---|---|---|
| | **Busca local II** | **Busca local de Nowicki** | **Cls + Bak** |
| **20** | 8,11 | 5,48 | 9,38 |
| **50** | 6,22 | 6,02 | 9,76 |
| **100** | 4,46 | 5,21 | 7,71 |
| **150** | 3,65 | 4,21 | 5,92 |
| **Média** | 5,61 | 5,23 | 8,19 |

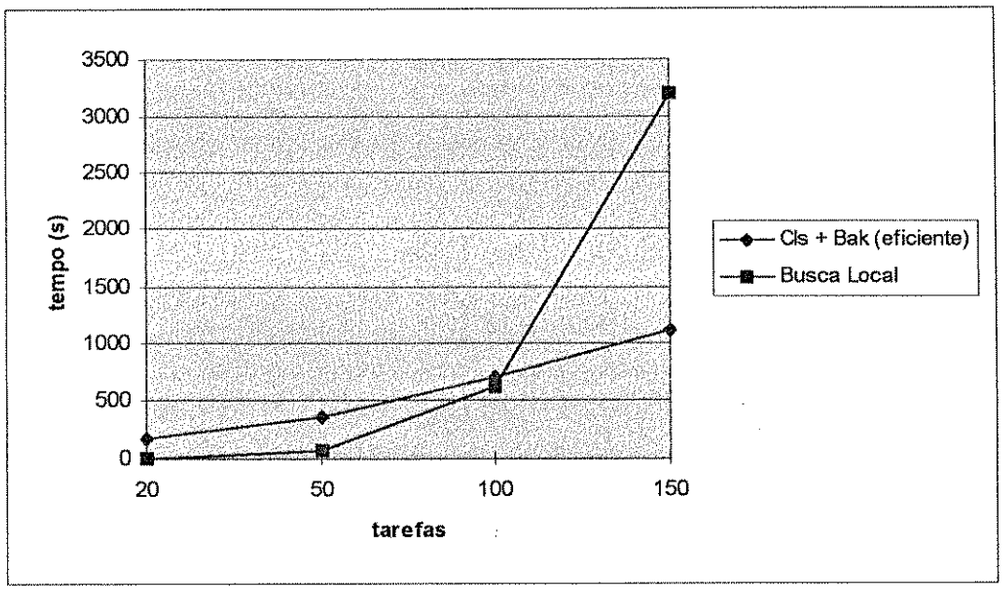



Figura 5-12: Variação do tempo computacional com o número de tarefas para 10 estágios.

Note que a implementação de algoritmo genético **Cls + Bak** apresenta os melhores resultados. Na média sobre todas as instâncias, a busca local II apresentou resultados melhores do que a busca local de Nowicki e Smutnicki. A busca local de Nowicki e Smutnicki só termina quando um mínimo local é alcançado. Por outro lado, a busca local II é interrompida se após a análise de uma vizinhança, o número de programas acumulados supera 30000. Para número pequeno de tarefas, 20, a busca local II é mais eficiente, porém ocorre perda de qualidade para 100 e 150 tarefas. Ou seja, com número de tarefas pequeno, a busca local II atinge um mínimo local antes de ser interrompida por excesso de movimentos. Com o aumento do número de tarefas, a busca é interrompida com poucas iterações. Sabe-se que a cardinalidade da vizinhança do tipo inserção é $(n-1)^2$. Para 150 tarefas, o número de vizinhos será $149^2 = 22201$. Neste caso apenas duas iterações da busca local II são realizadas. O desempenho bem superior da busca local II em relação à busca local de Nowicki e Smutnicki para 20 tarefas faz com que a primeira busca apresente um melhor desempenho ao se variar o número de estágios, exceto para 2 estágios.

A Tabela 5.25 mosta o número médio de iterações realizadas pela busca local II quando 20 e 50 tarefas são consideradas. Note que para 50 tarefas, em geral, a busca local é interrompida por excesso de movimentos, desde que $13 \times (49)^2 = 31212 > 30000$.

Tabela 5.24: Ganho médio em relação ao resultado da heurística **DK1** , quando se varia o número de estágios.

| **Estágios** | **Valores percentuais** | | |
|---|---|---|---|
| | **Busca local II** | **Busca local de Nowicki** | **Cls + Bak** |
| **2** | 1,84 | 1,89 | 3,06 |
| **4** | 4,31 | 3,44 | 6,35 |
| **6** | 6,16 | 5,93 | 9,19 |
| **8** | 6,73 | 6,34 | 9,99 |
| **10** | 9,00 | 8,55 | 12,38 |
| **Média** | 5,61 | 5,23 | 8,19 |

Tabela 5.25: Média de iterações realizadas pela busca local II, quando se varia o número de estágios e tarefas.

| **Tarefas** | **Número de estágios** | | | | | |
|---|---|---|---|---|---|---|
| | **2** | **4** | **6** | **8** | **10** | **Média** |
| **20** | 8 | 14 | 12 | 15 | 14 | 13 |
| **50** | 11 | 13 | 13 | 13 | 13 | 13 |
| **Média** | 10 | 14 | 13 | 14 | 14 | 13 |

Tabela 5.26: Desvio médio do algoritmo genético **Fifo** × heurística *Flowmult.*

| $n/m$ | **n. de máquinas** | | | |
|---|---|---|---|---|
| | **2** | **4** | **6** | **Aleatório 3..5** |
| **9/2** | 1,63 | 4,59 | 0,57 | 6,59 |
| **9/4** | 2,68 | 3,94 | 1,12 | 7,18 |
| **9/6** | 2,26 | 4,31 | 2,27 | 6,08 |
| **9/8** | 5,63 | 3,86 | 0,70 | 6,36 |
| **9/10** | 4,74 | 3,34 | 1,42 | 4,38 |

## 5.7 ALGORITMO GENÉTICO × HEURÍSTICA FLOWMULT

A Tabela 5.26 traz a comparação dos resultados da Implementação **Fifo** com a heurística *Flowmult.* A comparação foi feita usando a expressão de desvio relativo:

$$desvio = (\frac{Z_g - Z_r}{Z_r}) \cdot 100\%$$

Na expressão, $Z_g$ é o valor obtido pelo algoritmo genético, e $Z_r$ é o valor de referência, no caso o resultado da heurística *Flowmult.*

Ao se comparar o resultado da implementação **Fifo** com esta heurística, é possível avaliar a qualidade do algoritmo genético ao explorar o espaço de soluções que a implementação **Fifo** avalia, isto porque a heurística *Flowmult* avalia todos os programas que a implementação **Fifo** pode gerar. Na média, sobre todas as instâncias, o algoritmo genético **Fifo** foi 3,69% pior do que a heurística *Flowmult.*

Para avaliar a qualidade das implementações de algoritmos genéticos híbridos do tipo máquina menos ocupada, a heurística *Flowmult* foi ligeiramente modificada. Ao contrário de tratar as tarefas segundo a política *fifo,* ela passou a alocar as tarefas da mesma forma que as implementações híbridas do tipo máquina menos ocupada. A heurística *Flowmult*-modificada, passa a avaliar todos os programas que as implementações do tipo máquina menos ocupada consegue tratar. A Tabela 5.27 traz a comparação dos resultados da Implementação **Cls** +

Tabela 5.27: Desvio médio do algoritmo genético **Cls + Bak** ×heurística *Flowmult*-modificada.

| $n/m$ | **n. de máquinas** | | | |
|---|---|---|---|---|
| | **2** | **4** | **6** | **Aleatório 3..5** |
| **9/2** | 0,00 | 0,00 | 0,00 | 0,00 |
| **9/4** | 0,29 | 0,00 | 0,00 | 0,00 |
| **9/6** | 0,08 | 0,00 | 0,00 | 0,45 |
| **9/8** | 0,13 | 0,00 | 0,00 | 0,00 |
| **9/10** | 0,07 | 0,00 | 0,00 | 0,05 |

**Bak** com a heurística *Flowmult*-modificada. A comparação foi feita usando a expressão de desvio relativo:

$$desvio = (\frac{Z_g - Z_r}{Z_r}) \cdot 100\%$$

Na expressão, $Z_g$ é o valor obtido pelo algoritmo genético, e $Z_r$ é o valor de referência, no caso o resultado da heurística *Flowmult*-modificada. Na média, sobre todas as instâncias, o algoritmo genético **Cls + Bak** foi 0,05% pior do que a heurística *Flowmult*-modificada.

A Tabela 5.28 traz a comparação da heurística *Flowmult* com a heurística *Flowmult*-modificada. A comparação foi feita usando a expressão de desvio relativo:

$$desvio = (\frac{Z_h - Z_r}{Z_r}) \cdot 100\%$$

Na expressão, $Z_h$ é o valor obtido pela heurística *Flowmult*-modificada, e $Z_r$ é o valor de referência, no caso o resultado da heurística *Flowmult*.

Ao se comparar as heurísticas *Flowmult* e *Flowmult*-modificada, é possível avaliar a qualidade dos programas avaliados pelas implementações de algoritmos genéticos do tipo **Fifo** e máquina menos ocupada. Na média, sobre todas as instâncias, a heurística *Flowmult*-modificada foi 1,15% pior do que a heurística *Flowmult*. A implementação do tipo **Fifo** avalia um espaço de soluções de melhor qualidade, porém a implementação do tipo máquina menos ocupada é mais eficiente no seu mecanismo de busca, como pode ser visto nas Tabelas 5.26 e 5.27. A pior qualidade dos programas avaliados pela implementação do tipo máquina menos ocupada pode

Tabela 5.28: Desvio relativo da heurística *Flowmult*-modificada ×heurística *Flowmult*.

| $n/m$ | **n. de máquinas** | | | |
|---|---|---|---|---|
| | **2** | **4** | **6** | **Aleatório 3..5** |
| **9/2** | 1,00 | 0,57 | 0,00 | 0,68 |
| **9/4** | 1,26 | 1,31 | 0,00 | 1,54 |
| **9/6** | 2,88 | 1,71 | 0,00 | 3,65 |
| **9/8** | 4,02 | 0,49 | 0,00 | 0,10 |
| **9/10** | 3,68 | 0,21 | 0,00 | 1,19 |

ser explicada a partir da discussão feita sobre a Figura 4-24. É importante também atentar para o fato que este efeito é atenuado quando se aumenta o número de máquinas paralelas, como pode ser constatado na Tabela 5.27.

# Capítulo 6

# CONCLUSÕES

Assim como Kochhar e Morris (1987) procuram uma boa seqüência de entrada de tarefas no sistema, Santos *et al.* (1996) defendem que tratar segundo a política *fifo* uma boa seqüência de entrada de tarefas no *flow shop* flexível, é suficiente para se obter resultados próximos do ótimo. Este argumento está embasado em resultados obtidos com o uso da heurística *Flowmult.*

Este mesmo argumento pode ser utilizado para se analisar o comportamento das implementações de algoritmos genéticos. Note que as implementações de algoritmos genéticos mal sucedidos, com exceção da implementação **Fifo**, não buscam uma seqüência de entrada. Na verdade procuram distribuir tarefas às máquinas e determinar a seqüência de processamento das tarefas em cada uma delas. Com este tipo de estratégia, a primeira tarefa a ser processada em uma máquina de um estágio, pode ser a última a ser processada num outro estágio qualquer. O mecanismo de busca falha, pois utiliza os operadores genéticos em um estágio alterando a distribuição de tarefas nas máquinas e a ordem de processamento das tarefas sem atentar para o impacto global dessas alterações. Todas as implementações de algoritmos genéticos bem sucedidos usam a estratégia de procurar uma boa seqüência de tarefas. Para número crescente de tarefas os ganhos dos algoritmos genéticos tornam-se menores, mas por outro lado, os resultados dos algoritmos tornam-se mais próximos do limitante inferior.

Note que uma heurística simples como **DK1** apresentou os melhores resultados médios. A busca local II, que procura seqüências de entrada, apresenta resultados melhores do que a busca local de Nowicki e Smutnicki, que parte de uma única seqüência e altera a ordem de processamento e distribuição de tarefas nas máquinas.

As Tabelas 6.1, 6.2, 6.3 e 6.4 mostram de forma sintética os resultados obtidos com as heurísticas, buscas locais e implementações de algoritmos genéticos. Os dados são apresentados com o uso do desvio relativo. Números negativos indicam que houve ganho em relação ao valor de referência. Algumas explicações sobre a Tabelas 6.1, 6.2, 6.3 e 6.4 seguem abaixo:

- **MH**: heurística que apresenta o melhor resultado dentre **DK1**, **DK2**, **DK3**, **S**, **SPT**, **LPT**, **MWR** e **LWR**.
- **MRBL**: melhor resultado da busca local de Nowicki e Smutnicki quando o resultado das heurísticas **DK1**, **DK2**, **DK3**, **S**, **SPT**, **LPT**, **MWR** e **LWR** são utilizados como ponto de partida.
- **FM**: heurística *flowmult* para 9 tarefas.
- **FM-MOD**: heurística *flowmult* modificada para 9 tarefas.
- **BL-X**: busca loca de Nowicki e Smutnicki usando como ponto de partida a heurística **X**.
- $\mathbf{BL_{II}}$: busca local II para 20 e 50 tarefas.
- **BL-DK1** $\times$ **DK1**: busca loca de Nowicki e Smutnicki para 20 e 50 tarefas.
- **Puro A**, **Puro** $\mathbf{B_I}$, **Puro** $\mathbf{B_{II}}$, **BC** $+$ **A**, **BC** $+$ $\mathbf{B_I}$ e **Fifo**: não consideram instâncias com 150 tarefas.
- **Cls** $+$ **Bak** $\times$ **GUINET**: considera instâncias com apenas 2 estágios.
- Os demais resultados consideram instâncias com 2, 4, 6, 8 e 10 estágios, bem como 20, 50, 100 e 150 tarefas.

Da Tabela 6.1, conclui-se que heurística **DK1** mostrou ser a mais poderosa dentre todas. Embora as heurísticas **DK3** e **S** sejam muito parecidas, os resultados de **DK3** foram insatisfatórios. Com relação às regras de despacho, nota-se que a regra **SPT** apresenta um desempenho muito bom.

Da Tabela 6.2, conclui-se que a busca local de Nowicki e Smutnicki aplicada ao resultado da heurística **DK1** mostrou ser a mais eficiente dentre todas. As soluções geradas pelas heurísticas

Tabela 6.1: Síntese dos resultados das heurísticas.

| | MH | MRBL | LI2Cmax | DK1 | GUINET | FM | FM-MOD |
|---|---|---|---|---|---|---|---|
| **DK1** | 2,90 | - | - | - | - | - | - |
| **DK2** | 4,69 | - | - | - | - | - | - |
| **DK3** | 8,11 | - | - | - | - | - | - |
| **S** | 3,09 | - | - | - | - | - | - |
| **SPT** | 5,41 | - | - | - | - | - | - |
| **LPT** | 22,22 | - | - | - | - | - | - |
| **MWR** | 8,55 | - | - | - | - | - | - |
| **LWR** | 8,90 | - | - | - | - | - | - |

Tabela 6.2: Síntese dos resultados das buscas locais.

| | MH | MRBL | LI2Cmax | DK1 | GUINET | FM | FM-MOD |
|---|---|---|---|---|---|---|---|
| **BL-DK1** | - | 0,27 | - | -5,23 | - | - | - |
| **BL-DK2** | - | 2,86 | - | - | - | - | - |
| **BL-DK3** | - | 9,46 | - | - | - | - | - |
| **BL-S** | - | 4,78 | - | - | - | - | - |
| **BL-SPT** | - | 6,61 | - | - | - | - | - |
| **BL-LPT** | - | 24,09 | - | - | - | - | - |
| **BL-MWR** | - | 10,27 | - | - | - | - | - |
| **BL-LWR** | - | 10,05 | - | - | - | - | - |
| **MRBL** | -2,99 | - | - | - | - | - | - |
| **$BL_{II}$** | - | - | - | -5,61 | - | - | - |

Tabela 6.3: Síntese dos resultados dos algoritmos genéticoss mal-sucedidos.

| | MH | MRBL | LI2Cmax | DK1 | GUINET | FM | FM-MOD |
|---|---|---|---|---|---|---|---|
| **Puro A** | 48,06 | - | - | - | - | - | - |
| **Puro $B_I$** | 164,33 | - | - | - | - | - | - |
| **Puro $B_{II}$** | 68,83 | - | - | - | - | - | - |
| **BC + A** | 42,85 | - | - | - | - | - | - |
| **BC + $B_I$** | 148,02 | - | - | - | - | - | - |
| **Fifo** | -3,60 | - | - | - | - | 3,69 | - |

Tabela 6.4: Síntese dos resultados dos algoritmos genéticos bem-sucedidos.

| | MH | MRBL | LI2Cmax | DK1 | GUINET | FM | FM-MOD |
|---|---|---|---|---|---|---|---|
| **Ack +Rvs** | - | -2,39 | - | - | - | - | - |
| **Ack + Fit** | - | -2,17 | - | - | - | - | - |
| **Cls + Bak** | -5,63 | -2,91 | 7,16 | -8,19 | -4,04 | - | 0,05 |
| **Cls + Fit** | - | -0,73 | - | - | - | - | – |

**DK2**, **S** e **SPT** são também pontos de partida promissores. A busca local II apresenta um desempenho superior ao da busca de Nowicki e Smutnicki quando o ponto de partida é o resutado da heurística **DK1.**

A Tabela 6.3 mostra que os algoritmos genéticos **Puro A**, **Puro $B_I$**, **BC + A**, **BC + $B_I$** e **Puro $B_{II}$** são muito ruins e podem ser descartados. O único a conseguir melhoria em relação à melhor heurística foi a implementação do tipo **Fifo**, porém o ganho é pequeno, não justificando também o seu uso. Ao se comparar o resultado da implementação **Fifo** com a heurística *Flowmult*, é possível avaliar a qualidade do algoritmo genético, isto porque a heurística *Flowmult* gera um conjunto de programas que contém o conjunto de programas considerados pela versão **Fifo** de algoritmo genético. Na média, sobre todas as instâncias, o algoritmo genético **Fifo** foi 3,69% pior do que a heurística *Flowmult.*

A Tabela 6.4 mostra que a implementação **Cls + Bak** obteve os melhores resultados. Ao

se comparar o resultado da implementação **Cls + Bak** com a heurística *Flowmult* modificada, é possível avaliar a qualidade do algoritmo genético ao explorar o espaço de soluções que a implementação **Cls + Bak** avalia, isto porque a heurística *Flowmult* modificada gera um conjunto de programas que contém o conjunto de programas considerados pela versão **Cls + Bak** de algoritmo genético. Na média, sobre todas as instâncias, o algoritmo genético foi 0,05% pior do que a heurística *Flowmult* modificada. Os algoritmos genéticos estão próximos do limitante inferior, mostrando que os resultados obtidos estão suficientemente próximos do resultado ótimo.

## ESTUDOS FUTUROS

A partir das conclusões anteriores, é possível propor algumas linhas de pesquisa:

1. Utilizar o *crossover* proposto por Murata *et al.* (1996), pois apresentou bons resultados.

2. A implementação híbrida de algoritmos genéticos do tipo máquina menos ocupada, possui a característica restritiva de permitir que máquinas fiquem ociosas mesmo com tarefas disponíveis para processar. Para superar este problema, pode-se elaborar uma implementação híbrida que trabalhe em dois níveis. O primeiro nível seria responsável por encontrar seqüências de entrada com uso de uma implementação de algoritmo genético, e o segundo nível realizaria ajustes finos na distribuição de tarefas a máquinas, e na seqüência de processamento das tarefas em cada máquina com o uso da busca local ou busca tabu de Nowicki e Smutnicki.

# Apêndice A

# HEURÍSTICA RITM (ROUTE IDLE TIME MINIMIZATION) DE SAWIK

Esta heurística foi proposta por Sawik (1993, 1995), a primeira implementação tratando *buffer* limitado, e a segunda, *buffer* zero. Aqui, a heurística foi adaptada e implementada para tratar *buffer* infinito.

É uma heurística do tipo construtiva na qual a cada iteração a rota completa de uma tarefa é determinada. A seleção da tarefa e sua rota são baseadas no programa parcial obtido até o momento. As decisões em cada iteração são tomadas usando um procedimento de otimização local que busca minimizar o tempo ocioso total de máquinas para uma tarefa selecionada ao longo da rota.

## A.1 DESCRIÇÃO DA HEURÍSTICA

O programa das $n$ tarefas é determinado ao se especificar a ordem de entrada das tarefas no sistema $[\gamma_1, \gamma_2, \ldots, \gamma_n]$, assim como as variáveis temporais para detalhar o *programa individual* de cada tarefa. Em particular, para cada tarefa $\gamma_w$ que entra no sistema na posição $w$ ($w = 1, \ldots, n$), seu instante de início de processamento $s_{l\gamma_w}$ e instante de término $c_{l\gamma_w}$ em cada estágio $l$ podem ser determinados. Outras variáveis importantes são:

- $Y_{hlw}$ é o instante em que a máquina $h$ $(h = 1, \ldots, m_l)$ no estágio $l$ fica disponível após as primeiras $w$ tarefas terem sido seqüenciadas.

- $y_{lw}$ é o instante da **primeira** máquina no estágio $l$ a ficar disponível para processar outra tarefa, após as $w$ primeiras tarefas já terem sido seqüenciadas, $y_{lw} = \min_{1 \leq h \leq m_l}\{Y_{hlw}\}$

- $m(l, w)$ é a máquina no estágio $l$ a ficar disponível mais cedo após as $w - 1$ primeiras tarefas terem sido processadas, $m(l, w) = \arg\min_{1 \leq h \leq m_l}\{Y_{hl,w-1}\}$

Estas variáveis satisfazem as fórmulas apresentadas a seguir:

$$\begin{aligned} s_{l\gamma_w} &= \max\{c_{l-1,\gamma_w}, y_{l,w-1}\}, l = 2, \ldots, m \\ s_{1\gamma_w} &= y_{1,w-1}. \end{aligned} \tag{A.1}$$

A tarefa $\gamma_w$ não pode iniciar seu processamento no estágio $l$ antes de ter sido liberada do estágio $l-1$, e uma máquina estar disponível no estágio $l$ para processá-la. O processamento da tarefa $\gamma_w$ no primeiro estágio é iniciado quando uma máquina está disponível após as primeiras $w - 1$ terem sido processadas.

Desde que interrupções no processamento das tarefas não são permitidas, $s_{l\gamma_w}$ e $c_{l\gamma_w}$ para cada tarefa $\gamma_w$ estão conectadas por:

$$c_{l\gamma_w} = s_{l\gamma_w} + p_{l\gamma_w} \; l = 1, \ldots, m \tag{A.2}$$

Os instantes de disponibilidade $Y_{hlw}$ para as máquinas no estágio $l$ são calculados iterativamente:

$$Y_{hlw} = \begin{cases} Y_{hl,w-1} \text{ se } m \neq m(l, w) \\ c_{l\gamma_w} \text{ se } m = m(l, w) \end{cases} \tag{A.3}$$

Esta fórmula é derivada da hipótese de que em cada estágio $l$ a tarefa $\gamma_w$ é alocada à máquina $m(l, w)$ com instante mais cedo de disponibilidade.

Finalmente, nota-se que o tempo ocioso $t_{l\gamma_w}$ da tarefa $\gamma_w$ no estágio $l$ alocada à primeira máquina a ficar disponível é:

$$t_{l\gamma_w} = s_{l\gamma_w} - y_{l,w-1}, \tag{A.4}$$

isto é, o instante em que a tarefa $\gamma_w$ inicia seu processamento no estágio $l$ menos o instante da primeira máquina a ficar disponível para o processamento da tarefa $\gamma_w$, após $w-1$ tarefas terem sido processadas.

## A.2 ALGORITMO

Todas as tarefas são inicialmente ordenadas de forma não crescente em relação ao tempo ponderado total de processamento

$$\begin{aligned} \bar{P}_j &= \sum_{l=1}^{m}[1+a_1(l-1)]p_{lj}. \\ a_1 &= 0,5 \end{aligned} \tag{A.5}$$

A cada iteração uma tarefa é escolhida para entrar no sistema, bem como seu programa individual é determinado. Dado um programa parcial para $(w-1)$ tarefas já selecionadas, primeiro a melhor rota ao longo dos estágios é encontrada:

$$[m(1,w), m(2,w), \ldots, m(m,w)]\,,$$

sendo escolhida em cada estágio, a máquina com instante mais cedo de disponibilidade. Para cada tarefa esperando para entrar no sistema, seu programa individual é determinado ao longo desta melhor rota. Para avaliar o programa individual de cada tarefa considerada, o tempo ponderado total de máquinas ociosas para a tarefa $j$ ao longo da rota é computado:

$$\begin{aligned} \bar{t}_j &= \sum_{l=1}^{m}\frac{t_{lj}}{a_2^{m-l}} \\ a_2 &= 1,2 \end{aligned} \tag{A.6}$$

Finalmente, a tarefa com menor tempo ponderado total ocioso de máquinas $\bar{t}_j$ é selecionada para entrar no sistema e seu programa individual é adicionado ao programa parcial obtido até então.

Segue agora uma descrição do algoritmo, onde **J** é o conjunto de tarefas já introduzidas no sistema.

*Passo 0* - **Inicialização**

1. Ordenar todas as tarefas de forma não crescente de acordo com os valores do tempo de processamento total $\bar{P}_j$ (A.5), isto é $\bar{P}_1 \geq \bar{P}_2 \geq \ldots \geq \bar{P}_n$
2. atribuir: $Y_{hl0} = 0$; $h = 1, \ldots, m_l$, $l = 1, \ldots, m$
3. atribuir: $y_{l0} = 0$; $l = 1, \ldots, m$
4. atribuir: $m(l, 1) = 1$; $l = 1, \ldots, m$
5. atribuir: $\mathbf{J} = \emptyset$
6. atribuir: $w = 1$

*Passo 1* - **Seleção de uma tarefa para entrar no sistema**

1. Para cada tarefa $j \notin \mathbf{J}$ esperando para ser inserida no sistema, determine $s_{lj}$ (A.1), $c_{lj}$ (A.2), $t_{lj}$ (A.4), em cada estágio $l$, na máquina $m(l, w)$ com tempo mais cedo de disponibilidade $y_{l,w-1}$. Calcule o tempo ponderado total de máquina ociosa $\bar{t}_j$ (A.6).
2. Selecione a tarefa $\gamma_w$ para a qual o tempo ponderado total de máquina ociosa é mínimo.

$$\gamma_w = \arg\min_{j \notin \mathbf{J}} \{\bar{t}_j\}$$

Em caso de empate, escolha a tarefa com menor índice $\bar{P}_j$, isto é, aquela com maior tempo ponderado de processamento total.

*Passo 2* - **Determinação do programa individual da tarefa selecionada**

Para a tarefa selecionada, determinar $s_{l\gamma_w}$ (A.1), $c_{l\gamma_w}$ (A.2), $t_{l\gamma_w}$ (A.4), em cada estágio $l$, na máquina $m(l, w)$ com instante mais cedo de disponibilidade $y_{l,w-1}$. O programa individual da tarefa $\gamma_w$ é adicionado ao programa parcial das primeiras $(w-1)$ tarefas.

*Passo 3* - **Determinação da próxima rota**

Faça $\mathbf{J} = \mathbf{J} \cup \gamma_w$. Se $\mathbf{J} = \{1, \ldots, n\}$, fim. Caso contrário, determine para cada máquina $h$ ($h = 1, \ldots, m_l$, $l = 1, \ldots, m$) o instante de disponibilidade $Y_{hlw}$ (A.3) após as primeiras $w$ tarefas terem sido selecionadas. Para cada estágio $l$, encontre a máquina $m(l, w+1)$ com instante mais cedo de disponibilidade $y_{lw}$. Faça $w = w + 1$ e volte ao *Passo 1.*

### A.2.1 COMPLEXIDADE DO ALGORITMO

Há $n$ tarefas para serem seqüenciadas, e $m$ máquinas da melhor rota pelas quais cada uma delas deve passar. A cada iteração, a melhor rota deve ser encontrada. Isto envolve analisar todas as máquinas de todos os estágios, $\sum_{l=1}^{m} m_l$. Depois, uma tarefa é selecionada. Portanto na primeira iteração há $n$ tarefas para serem avaliadas, na segunda iteração, há $n-1$, e assim sucessivamente. Assim o número total de operações é:

$$[n + (n-1) + \ldots + 1] \cdot m + n \cdot \sum_{l=1}^{m} m_l = \frac{n \cdot (n+1) \cdot m}{2} + n \cdot \sum_{l=1}^{m} m_l.$$

O primeiro termo representa o custo de escolher tarefas ao longo da rota composta de $m$ máquinas, e o segundo termo o custo de encontrar a melhor rota para as $n$ iterações.

Considerando o processamento de uma tarefa como uma operação básica, a ordem de execução do algoritmo em função do número de tarefas, número de máquinas e número de estágios é de $\mathcal{O}(n^2 m + n\sum_{l=1}^{m} m_l) = \mathcal{O}(n(nm + \sum_{l=1}^{m} m_l))$. Como em todo estágio, o número tarefas é superior ao número de máquinas, $nm > \sum_{l=1}^{m} m_l$. Então a complexidade do algoritmo é $\mathcal{O}(n^2 m)$.

# Apêndice B

# HEURÍSTICAS DE DING E KITTICHARTPHAYAK

Ding e Kittichartphayak (1994) propuseram três heurísticas construtivas. Nas heurísticas **DK1** e **DK2**, uma seqüência de tarefas é determinada para o primeiro estágio. Esta seqüência de tarefas é aplicada em todos os estágios da seguinte forma: a primeira tarefa é alocada à máquina que fica liberada mais cedo no estágio, e assim sucessivamente para as demais tarefas da seqüência. A heurística **DK3** procura uma rota para cada tarefa, objetivando minimizar o tempo ocioso total.

## B.1 DK1

Esta heurística aplica a idéia de Campbell *et al.* (1970), originária do *flow shop* de permutação com uma máquina por estágio, no ambiente de *flow shop* flexível. Nas heurísticas **DK1** e **DK2**, o termo $P_{1j}$ é usado para designar o tempo de processamento total nos $l$ primeiros estágios da tarefa $j$, e $P_{2j}$ é o tempo de processamento total nos últimos $l$ estágios da tarefa $j$, onde $l$ é incrementado de uma unidade a cada iteração.Na heurística **DK1**, $r$ é o índice da tarefa que tem o menor valor entre todos os $P_{1j}$'s e $P_{2j}$'s. O procedimento é o seguinte:

1. Seja $l = 1$, calcule

$$P_{1j} = \sum_{y=1}^{l} p_{yj},$$

$$P_{2j} = \sum_{y=1}^{l} p_{m-y+1,j}$$

Encontre $\min_j\{P_{1j}, P_{2j}\}$ para as tarefas não seqüenciadas.

2. Se $P_{1r} = \min_j\{P_{1j}, P_{2j}\}$ para uma tarefa $r$, designe a tarefa $r$ para a primeira posição livre, caso contrário, se $P_{2r} = \min_j\{P_{1j}, P_{2j}\}$, designe a tarefa $r$ para a última posição livre.

3. Remova a tarefa $r$ e repita o *Passo 2* até que não sobre nenhuma tarefa.

4. Compare o *makespan* com a melhor solução corrente, e mantenha a melhor solução.

5. Se $l = m - 1$, pare, caso contrário, $l = l + 1$ e vá para o *Passo 1*.

O custo de calcular o *makespan* é $\mathcal{O}(n\sum_{l=1}^{m} m_l)$. Isto porque a regra de alocação precisa encontrar a máquina menos ocupada no estágio, devendo então analisar todas as $m_l$ máquinas de cada estágio para depois alocar a tarefa. Como $m-1$ programas são gerados, a complexidade da heurística é $\mathcal{O}(mn\sum_{l=1}^{m} m_l)$.

## B.2 DK2

De modo similar à heurística **DK1**, **DK2** gera $m - 1$ programas. A diferença entre estas duas heurísticas é a regra de colocação de uma tarefa no início ou final de uma seqüência. Na heurística **DK2**, $r$ é a tarefa que tem o menor valor entre todos os $P_{1j}$'s e também o mínimo entre todos os $P_{2j}$'s. O procedimento passo a passo é o seguinte:

1. Seja $l = 1$, calcule

$$P_{1j} = \sum_{y=1}^{l} p_{yj}$$

$$P_{2j} = \sum_{y=1}^{l} p_{m-y+1,j}$$

Encontre $\min_j\{P_{1j}\}$, e o $\min_j\{P_{2j}\}$ para as tarefas não seqüenciadas.

2. Se $P_{1r} = \min_j\{P_{1j}\}$, e $P_{2r} = \min\{P_{2j}\}$ para uma tarefa $r$, designe a tarefa $r$ para a última posição livre. Se $P_{1r} = \min_j\{P_{1j}\}$, mas $P_{2r} \neq \min_j\{P_{2j}\}$, designe a tarefa $r$ para a primeira posição livre.

3. Remova a tarefa $r$ e repita o *Passo 2* até que não sobre nenhuma tarefa.

4. Compare o *makespan* com a melhor solução corrente, e mantenha a melhor solução.

5. Se $l = m - 1$, pare, caso contrário, $l = l + 1$ e vá para o *Passo 1.*

A complexidade de **DK2** é idêntica à de **DK1**.

## B.3 DK3

Esta heurística constrói uma seqüência de tarefas e minimiza o tempo de ociosidade total a cada passo. O algoritmo foi inicialmente proposto por Gupta (1986) para ser aplicado no *flow shop* de permutação com uma máquina por estágio. O método é muito parecido com a heurística **RITM** descrita no APÊNDICE A. O procedimento passo a passo para a heurística **DK3** é descrito abaixo. Detalhes de como calcular o tempo de ociosidade total ao longo de uma rota no *flow shop* flexível para uma dada tarefa podem ser encontrados na descrição da heurística **RITM**.

1. Gere duas seqüências iniciais de tarefas obtidas a partir das heurísticas **DK1** e **DK2**. Escolha o resultado de **DK1** como seqüência parcial inicial para a heurística, e guarde o resultado de **DK2** para ser analisado depois.

2. Seqüencie apenas as duas primeiras tarefas da seqüência inicial. Os passos a seguir tratam as $n - 2$ tarefas restantes.

3. Encontra o mínimo $\bar{t}_j$ (APÊNDICE A), usando $a_1 = 0$ e $a_2 = 1$, onde $j$ é qualquer tarefa não seqüenciada. Se uma única tarefa produz o mínimo, vá para o *Passo 5.*

4. Desempates são estabelecidos em favor da tarefa com maior tempo de processamento no estágio $m$, $m-1, \ldots,$ até que o empate seja resolvido.

5. Adicione a tarefa ao programa parcial. Se restarem apenas 2 tarefas, vá para o *Passo 6*, caso contrário, vá para o *Passo 3*.

6. Gere dois programas completos alocando as duas tarefas restantes ao final dos programas parciais, e guarde o programa com menor *makespan* como sendo a melhor solução.

7. Se a solução inicial da heurística **DK2**, obtida no *Passo 1* não foi ainda utilizada, volte para o *Passo 2*, sendo ela a seqüência inicial. Caso contrário, fim.

A complexidade computacional desta heurística envolve resolver **RITM** para $n-4$ tarefas, **DK1** e **DK2:**

$$\mathcal{O}(mn\sum_{l=1}^{m} m_l + n^2 m) = \mathcal{O}(mn(\sum_{l=1}^{m} m_l + n)).$$

# Apêndice C

# HEURÍSTICA PARA DOIS ESTÁGIOS DE GUINET

Esta heurística construtiva foi proposta por Guinet *et al.* (1996). É uma variação da heurística **DK1**, e é descrita a seguir.

*Passo 1 - SEQÜÊNCIAMENTO*

Aplique a heurística **DK1**, descrita no APÊNDICE B, para obter uma seqüência de entrada.

*Passo 2 - ALOCAÇÃO*

Cada tarefa da seqüência estabelecida no *Passo 1* é alocada às máquinas dos dois estágios de forma a minimizar o *makespan*. Quando duas ou mais máquinas do mesmo estágio geram o mesmo instante de término para a tarefa, escolhe-se a liberada mais tarde, visando a minimização local do tempo ocioso de máquinas. Há duas situações a serem consideradas:

- Seja $\text{MIMMC}_l$ o instante mais cedo de liberação de uma máquina no estágio $l$, e $\text{RTMC}_{l,i}$ o instante de liberação da máquina $i$ do estágio $l$ antes da tarefa $j$ ser alocada. Se $\text{MIMMC}_1 + p_{1j} \geq \text{MIMMC}_2$, no estágio 1, a tarefa $j$ é alocada à máquina que gera $\text{MIMMC}_1$. Entre as máquinas do estágio 2 que poderiam ser consideradas para processar a tarefa $j$, isto é, aquelas para as quais $\text{MIMMC}_1 + p_{1j} > \text{RTMC}_{2,i}$, é escolhida aquela a ficar disponível mais tarde.

- Se $\text{MIMMC}_1 + p_{1j} < \text{MIMMC}_2$, no estágio 2, a tarefa $j$ é alocada à máquina que gera $\text{MIMMC}_2$. Entre as máquinas do estágio 1 que poderiam ser escolhidas para processar a tarefa $j$, isto é, aquelas para as quais $\text{MIMMC}_2 - p_{1j} > \text{RTMC}_{1,i}$, escolhe-se aquela a ficar disponível mais tarde.

A complexidade computacional pode ser derivada do resultado obtido para a heurística **DK1** para um número fixo de estágios e é dada por $\mathcal{O}(n \sum_{l=1}^{2} m_l)$.

# Apêndice D

# CARDINALIDADE DA VIZINHANÇA

Quando define-se uma vizinhança, é importante calcular quantos vizinhos possui uma dada solução x. O número total de vizinhos é chamado de cardinalidade da vizinhança e é simbolizado por $|\mathcal{N}(\mathbf{x})|$. Sabendo a cardinalidade da vizinhança, é possivel calcular a complexidade de uma iteração do algoritmo de busca local.

No algoritmo de busca local implementado para o *flow shop* flexível, usou-se vizinhança do tipo inserção, que é descrita a seguir. Seja $l$ um estágio selecionado, $a$ e $b$ grupos neste estágio, e $x, y$ posições nas permutações $\pi_{la}$ e $\pi_{lb}$. O vetor $v = (l, a, x, b, y)$ associado à ordem de processamento $\pi \in \Pi$ define um movimento do tipo inserção, de tal forma que a tarefa $\pi_{la}(x)$ é removida do grupo $N_{la}$ (mais precisamente, da posição $x$ na permutação $\pi_{la}$) e então é inserida em $N_{lb}$ na posição $y$ na permutação $\pi_{lb}$, Figura D-1. Usando uma descrição mais formal, seja $z = \pi_{la}(x)$. Se $a \neq b$, a remoção da tarefa $z$ da posição $x$ na permutação $\pi_{la}$ implica que as tarefas $\pi_{la}(x+1), \ldots, \pi_{la}(n_{la})$ são deslocadas uma posição para a esquerda em $\pi_{la}$, depois as tarefas $\pi_{lb}(y), \ldots, \pi_{lb}(n_{lb})$ são deslocadas uma posição para a direita em $\pi_{lb}$, e $z$ é inserida na posição $y$ da segunda permutação. Se $a = b$, a remoção da tarefa $z$ da posição $x$ na permutação $\pi_{la}$ implica que as tarefas $\pi_{la}(x+1), \ldots, \pi_{la}(y)$ são deslocadas uma posição para a esquerda se $x < y$, ou as tarefas $\pi_{la}(y), \ldots, \pi_{la}(x-1)$ são deslocadas uma posição para a direita se $x > y$, e depois $z$ é inserida na posição $y$ da permutação. A nova ordem de processamento obtida de

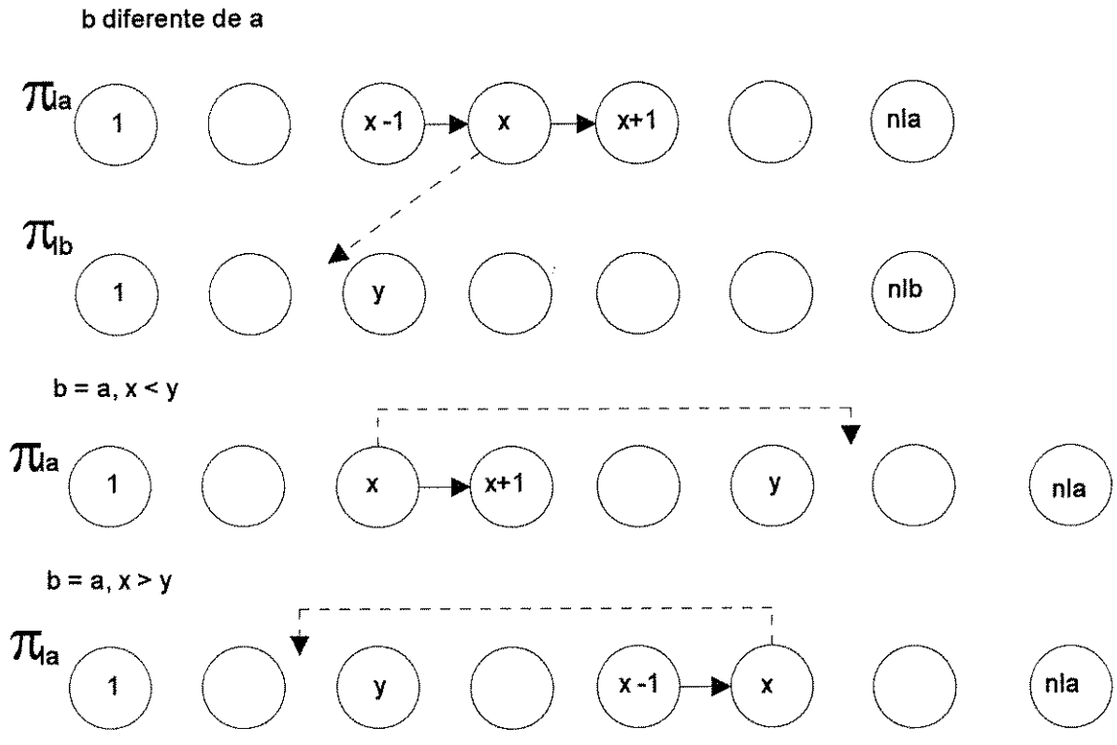



Figura D-1: Movimento $v = (l, a, x, b, y)$

$\pi$ pela aplicação do movimento $v$ será denotada por $\pi_v$.

A vizinhança de $\pi$ consiste das ordens de processamento $\pi_v$ geradas por movimentos $v$ de um dado conjunto. Este conjunto é do tipo: $\{\pi_v | v \in \mathcal{N}(\pi)\}$ gerados pelos movimentos:

$$\mathcal{N}(\pi) = \bigcup_{l \in M} \bigcup_{a \in M_l} \bigcup_{x=1}^{n_{la}} \bigcup_{b \in M_l} \mathcal{N}_{laxb}(\pi)$$

onde

$$\mathcal{N}_{laxb}(\pi) = \begin{cases} \{(l, a, x, b, y) \mid 1 \leq y \leq n_{lb},\ y \notin \{x-1, x\}\}, & \text{se } b = a \\ \{(l, a, x, b, y) \mid 1 \leq y \leq y \leq n_{lb} + 1\}, & \text{se } b \neq a \end{cases}$$

para $a, b \in M_l$, $x = 1, \ldots, n_{la}$, $l \in M$. O conjunto $\mathcal{N}_{laxb}(\pi)$ contém movimentos tais que a tarefa $\pi_{la}(x)$ é removida da posição $x$ na permutação $\pi_{la}$ e inserida em todas as posições possíveis

na permutação $\pi_{lb}$. A condição $y \neq x - 1$ é adicionada para evitar movimentos redundantes. Pode-se verificar que $|\mathcal{N}_{laxb}(\pi)| = n_{lb} + 1$ se $a \neq b$, e $|\mathcal{N}_{laxb}(\pi)| = n_{lb} - \min\{x, 2\}$ se $a = b$. Assim, levando em consideração a igualdade $\sum_{i \in M_l} n_{li} = n$, $l \in M$, tem-se que:

$$|\mathcal{N}(\pi)| = \sum_{l=1}^{m} \sum_{a=1}^{m_l} \sum_{x=1}^{n_{la}} \sum_{b=1}^{m_l} |\mathcal{N}_{laxb}(\pi)|$$

$$|\mathcal{N}(\pi)| = \sum_{l=1}^{m} \sum_{a=1}^{m_l} \sum_{x=1}^{n_{la}} (n + (m_l - 1) - \min\{x, 2\})$$

$$|\mathcal{N}(\pi)| = \sum_{l=1}^{m} \sum_{a=1}^{m_l} [n + (m_l - 1)] \cdot n_{la} - \sum_{x=1}^{n_{la}} \min\{x, 2\}$$

$$|\mathcal{N}(\pi)| = \sum_{l=1}^{m} \sum_{a=1}^{m_l} [n + (m_l - 1)] \cdot n_{la} - (1 + 2 \cdot (n_{la} - 1))$$

$$|\mathcal{N}(\pi)| = \sum_{l=1}^{m} (n + m_l - 1) \cdot n + m_l - 2n$$

$$|\mathcal{N}(\pi)| = (mn + \sum_{l=1}^{m} m_l - m) \cdot n + \sum_{l=1}^{m} m_l - 2nm$$

$$|\mathcal{N}(\pi)| = mn(n-3) + (n+1) \sum_{l=1}^{m} m_l$$

ou seja, o número de vizinhos é da ordem de $mn^2$.

O tempo de execução da busca local depende da complexidade de se analisar a vizinhança de uma solução. Como a complexidade de se calcular $C_{\max}(\pi)$, é igua a $\mathcal{O}(mn)$, a complexidade de calcular o *makespan* de todos os programas gerados por essa vizinhança será $\mathcal{O}(n^3 m^2)$. A vizinhança pode ser reduzida pelas regras propostas por Nowicki e Smutinicki (1995) e descritas no capítulo 5. Em Nowicki e Smutnicki (1995), há uma demonstração de que o uso da vizinhança reduzida diminui $mc$ vezes a complexidade de avaliação da vizinhança $\mathcal{N}(\pi)$, onde $c$ é um número constante de máquinas por estágio.

# Apêndice E

# GERAÇÃO DA POPULAÇÃO INICIAL E CROSSOVER

Seguem abaixo as descrições da complexidade da geração da população inicial e do *crossover*.

## E.1 POPULAÇÃO INICIAL

O algoritmo que gera de forma aleatória um indivíduo da população inicial para as implementações híbridas é descrito a seguir. As demais implementações são variações deste processo.

Sejam

- o vetor $\mathbf{C} = C[1] \ldots C[n]$: a estrutura que conterá a seqüência de entrada das tarefas, ou seja, é o próprio cromossomo.
- $n$: o número de tarefas.
- $i$, $temp$: índices em um vetor de cromossomo.

*Passo 1*

- $i = 1$.
- $temp = 0$.

*Passo 2* - *temp* recebe um número aleatório no intervalo de inteiros $1 \ldots n$.

*Passo 3* - Se $C[i'] \neq temp$, $\forall\ i' \in [1 \ldots i-1]$, $C[i] \leftarrow temp$, caso contrário volte ao *Passo 2.*

*Passo 4* - $i \leftarrow i + 1$.

*Passo 5* - Se $i > n$, vá para o *Passo 6*, caso contrário volte ao *Passo 2.*

*Passo 6* - Fim.

Não é possível fazer uma análise de complexidade de pior caso, devido à característica probabilística de geração de números aleatórios no *Passo 2.* Por outro lado, é possível tornar o algoritmo mais rápido, modificando o *Passo 3,* que no pior caso deve analisar $n-1$ elementos do vetor **C**. O algoritmo melhorado vem descrito a seguir.

*Passo 1*

- $\mathbf{C'} = C'[x] = 0$, $\forall\ x \in [1 \ldots n]$. Inicialmente, é um vetor de rótulos nulo.
- $i = 1$.
- $temp = 0$.

*Passo 2* - *temp* recebe um número aleatório no intervalo de inteiros $1 \ldots n$.

*Passo 3* - Se $C'[temp] = 0$, $C'[temp] \leftarrow 1$ e $C[i] \leftarrow temp$, caso contrário volte ao *Passo 2.*

*Passo 4* - $i \leftarrow i + 1$.

*Passo 5* - Se $i > n$, vá para o *Passo 6*, caso contrário volte ao *Passo 2.*

*Passo 6* - Fim.

Note que $C'[x] = 0$, significa que o número de tarefa $x$ ainda não foi gerado, e $C'[x] = 1$, rotula a tarefa $x$ como já existente. O *Passo 3* que tinha complexidade linear, passou a ter complexidade constante.

## E.2 CROSSOVER

O algoritmo que gera dois descendentes pelo *crossover* para as implementações híbridas é descrito a seguir. As demais implementações são variações deste processo.

Sejam:

- os vetores $\mathbf{P}_1$=$P_1[1]\ldots P_1[n]$, e $\mathbf{P}_2$=$P_2[1]\ldots P_2[n]$ que representam os pais.
- os vetores $\mathbf{F}_1$=$F_1[1]\ldots F_1[n]$, e $\mathbf{F}_2$=$F_2[1]\ldots F_2[n]$ que representam os filhos.
- *ponto*: o ponto de *crossover*.
- $n$: o número de tarefas.
- $i$, $j_1$, $j_2$: índices em um vetor de cromossomo.

*Passo 1*

- $i = 1$.
- $ponto = 0$.

*Passo 2* - *ponto* recebe um número aleatório no intervalo de inteiros $1\ldots n$.

*Passo 3* - $F_1[i'] \leftarrow P_1[i']$, $\forall\ i' \in [1\ldots ponto]$.

*Passo 4* - $F_2[i'] \leftarrow P_2[i']$, $\forall\ i' \in [1\ldots ponto]$.

*Passo 5* - $j_1 \leftarrow 1$.

*Passo 6* - $j_2 \leftarrow 1$.

*Passo 7*- $i \leftarrow ponto + 1$.

*Passo 8* - Se $i > n$, vá para o *Passo 12*.

*Passo 9* - Se $F_1[i'] \neq P_2[j_2]$, $\forall\ i' \in [1\ldots i-1]$, $F_1[i] \leftarrow P_2[j_2]$, caso contrário $j_2 \leftarrow j_2 + 1$,e volte ao *Passo 9*.

*Passo 10* - Se $F_2[i'] \neq P_1[j_1]$, $\forall\ i' \in [1\ldots i-1]$, $F_2[i] \leftarrow P_1[j_1]$, caso contrário $j_1 \leftarrow j_1 + 1$,e volte ao *Passo 10*.

*Passo 11* - $i \leftarrow i + 1$. Volte ao *Passo 8*.

*Passo 12* - Fim.

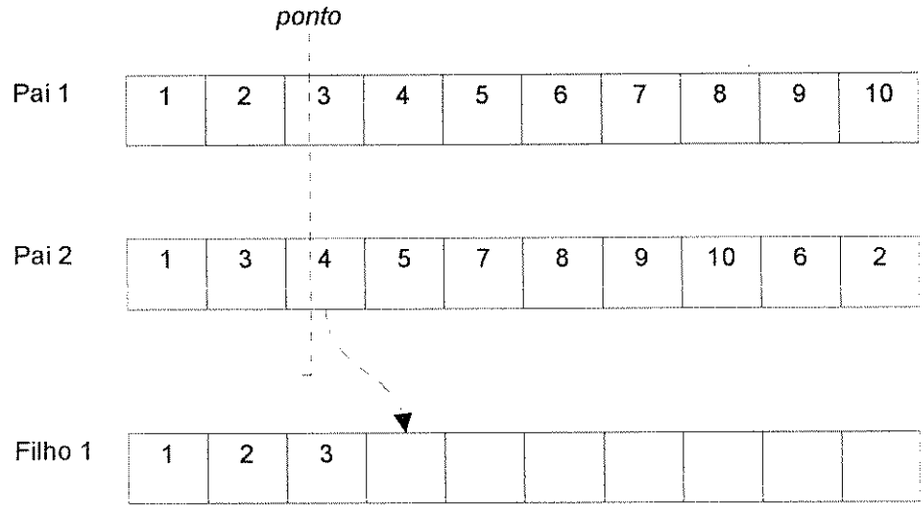


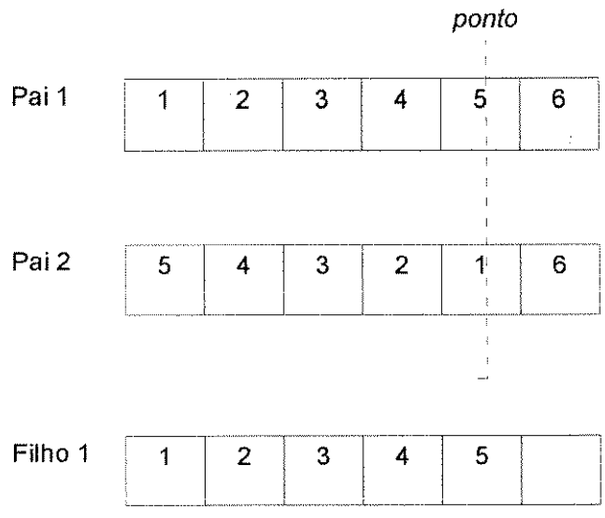

Figura E-1: *Crossover*.

A Figura E-1 mostra o estado do Filho 1, após ter recebido a parte inicial do Pai 1. O restante do Filho 1 deve ser completado com as tarefas do Pai 2 que ainda não estão presentes. A primeira tarefa do Pai 2 a ser incluída no Filho 1, será a tarefa de número 4, quando no *Passo 9*, a condição é satisfeita para $i = 4$ e $j_2 = 3$.

Note que um pior caso do algoritmo ocorre para $ponto = n - 1$, $P_1[n] = P_2[n]$ e $P_1[k] = P_2[n-k]\ \forall\ k \in [1, \ldots, n-1]$. Nesta condição, são realizadas $2 \times [(n-1) + \ldots + 2 + 1 + (n-1)] = (n-1)(n+2)$ comparações nos passos 9 e 10, como pode ser observado no exemplo da Figura E-2. O algoritmo tem complexidade quadrática, mas é possível torná-lo mais rápido, modificando os passos 9 e 10. O algoritmo melhorado vem descrito a seguir.

*Passo 1* - Sejam

- $\mathbf{F}'_1 = F'_1[x] = 0, \forall\ x \in [1 \ldots n]$. Inicialmente é um vetor de rótulos nulo.
- $\mathbf{F}'_2 = F'_2[x] = 0, \forall\ x \in [1 \ldots n]$. Inicialmente é um vetor de rótulos nulo.
- $i = 1$.
- $ponto = 0$.




| Tarefa | Compara com | Total |
|---|---|---|
| 5 | 1 2 3 4 5 | 5 |
| 4 | 1 2 3 4 | 4 |
| 3 | 1 2 3 | 3 |
| 2 | 1 2 | 2 |
| 1 | 1 | 1 |
| 6 | 1 2 3 4 5 | 5 |

Figura E-2: Pior caso do algoritmo de *crossover* para 6 tarefas.

*Passo 2* - *ponto* recebe um número aleatório no intervalo de inteiros $1 \ldots n$.

*Passo 3* - $F_1[i'] \leftarrow P_1[i']$, $F_1'[P_1[i']] \leftarrow 1$, $\forall\ i' \in [1 \ldots ponto]$.

*Passo 4* - $F_2[i'] \leftarrow P_2[i']$, $F_2'[P_2[i']] \leftarrow 1$, $\forall\ i' \in [1 \ldots ponto]$.

*Passo 5* - $j_1 \leftarrow 1$.

*Passo 6* - $j_2 \leftarrow 1$.

*Passo 7*- $i \leftarrow ponto + 1$.

*Passo 8* - Se $i > n$, vá para o *Passo 12.*

*Passo 9* - Se $F_1'[P_2[j_2]] = 0$ então $F_1[i] \leftarrow P_2[j_2]$ e $F_1'[P_2[j_2]] \leftarrow 1$, caso contrário $j_2 \leftarrow j_2 + 1$,e volte ao *Passo 9.*

*Passo 10* - Se $F_2'[P_1[j_1]] = 0$ então $F_2[i] \leftarrow P_1[j_1]$ e $F_2'[P_1[j_1]] \leftarrow 1$, caso contrário $j_1 \leftarrow j_1 + 1$,e volte ao *Passo 10.*

*Passo 11* - $i \leftarrow i + 1$. Volte para o *Passo 7.*

*Passo 12* - Fim.

Note que $F_1'[x] = 0$, significa que o número de tarefa $x$ ainda não faz parte do filho1, e $F_1'[x] = 1$, rotula a tarefa $x$ como já existente, como pode ser observado na Figura E-3. Os passos 9 e 10 realizam no máximo $2n$ comparações e portanto a complexidade do algoritmo é $\mathcal{O}(n)$.

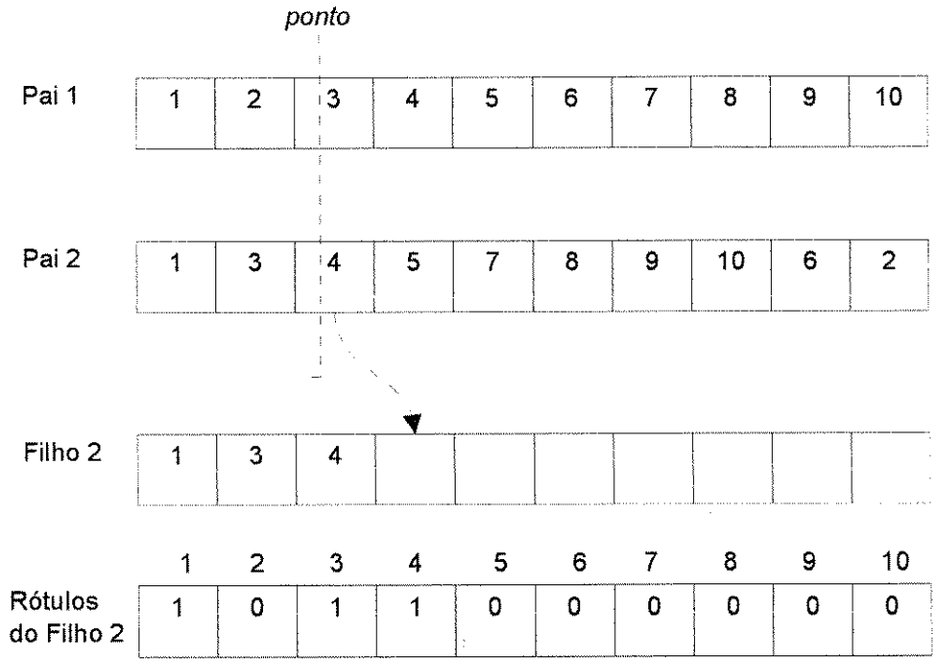



Figura E-3: Vetor de rótulos do filho 2

# Bibliografia


[1] Ackley, D. H., 1987. An empirical study of bit vector function optimisation. *In Genetic Algorithms and Simulated Annealing.* (Edited by L. Davis), 170-204. Pitman, London. *apud* Reeves, C. R., 1995, A genetic algorithm for flowshop sequencing. *Computers Ops. Res.*, vol 22, no 1, pp 5-13.

[2] Arthanari, T. S., and Ramaswamy, K. G., 1971. An extension of two machine sequencing problem. *Opsearch,* vol 8, pp 10-22. *apud* Hunsucker J.L., Shah J. R., 1994. Comparative performance analysis of priority rules in a constrained flow shop with multiple processors environment. *European J. Oper. Res.* vol 72, pp 102-114.

[3] Baker, J. E., 1985. Adaptive selection Methods for genetic algorithms. Proceedings of the first international conference on genetic algorithms, Lawrence Erlbaum associates, Hillsdale, NJ, pp 101-111. *apud* Michalewicz, Z., 1992. *Genetic Algorithms + Data Structures = Evolution Programs.* Springer-Verlag.

[4] Beasley, D., Bull, D. R. and Martin, R. R., 1993a. An overview of genetic algorithms: Part 1, fundamentals. *University Computing,* vol 15, no 2, pp 58-69.

[5] Beasley, D., Bull, D. R. and Martin, R. R., 1993b. An overview of genetic algorithms: Part 2, research topics. *University Computing,* vol 15 no 4, pp 170-181.

[6] Brah, S. A., and Hunsucker, J. L., 1991, Branch and bound algorithm for the flowshop with multiple processors. *European Journal of Operational Research,* vol 51, pp 88-99.

[7] Campbell, H. G.; Dudek, R. A. and Smith, M. L., 1970. A heuristic algorithm for the $n$ job machine sequencing problems. *Mgmt Sci.* vol 16, no 10, .

[8] Chen, B.,1995, Analysis of Classes of heuristics for scheduling a two-stage flow shop with parallel machines at one stage. *Journal of Operational Research Society*, vol 46, pp 234-244.

[9] Ding F. Y., Kittichartphayak D., 1994. Heuristics for scheduling flexible flow lines, *Computers Industrial Engineering* vol 26, pp 27-34.

[10] Downsland, K. A., 1996. Genetic algorithms - a tool for OR? *Journal of the Operational Reseach Society* vol 47, pp 550-561.

[11] Eglese, R. W., 1990. Simulatede Annealing: a tool for operational research, *European Journal of Operational Research*, vol 46, pp 271-281.

[12] French, S., 1982, *Sequencing and scheduling: An introduction to the mathematics of the job-shop* (Chichester, UK: Horwood).

[13] Garey, M. R. and Johnson, D. S., 1978. Strong NP completeness results: Motivations, examples and implications. *J. assoc comput. match*, vol 25, pp 499-508.

[14] Garey, M. R. and Johnson, D. S., and Sethi, R., 1976. The complexity of flow shop and job shop scheduling. *Math oper res.* vol 1, pp 117-129.

[15] Glover, F., 1994. Tabu Search for nonlinear and parametric optimization (with links to genetic algorithms), *Discrete applied mathmatics*, vol 7, pp 231-255.

[16] Glover, F., 1989, and Greenberg, H. J., 1989. New approaches for heuristic search: a bilateral linkage with artificial intelligence, *European Journal of Operational Research*, vol 39 no 2, pp 119-130.

[17] Goldberg, D. E., 1989. *Genetic algorithms in search, optimization and machine learning.* Addison-Wesley.

[18] Guinet A. G. P. and Solomon M. M., 1996. Scheduling hybrid flowshops to minimize maximum tardiness or maximum completion time. *International Journal of Production Reaserch* vol 34, no 6, pp 1643-1654.

[19] Guinet A. G. P., Solomon M. M., Kedia P. K. and Dussauchoy A., 1996. A computational study of heuristics for two-stage flexible flowshops. *International Journal of Production Reaserch* vol 34, no 5, pp 1399-11415.

[20] Gupta, J. N. D., and Tunc, E. A., 1991. Schedules for a two-stage hybrid flow shop with parallel machines at the second stage. *International Journal of Production Reaserch* vol 29, pp 1489-1502.

[21] Gupta, J. N. D., 1988. Two stage, hybrid flow shop scheduling problem. *Journal of the Operational Research Society* vol 39, pp 359-364.

[22] Holland, J. H., 1975. *Adaptation in Natural and Artificial Systems.* MIT Press.

[23] Hunsucker J.L., Shah J. R., 1994. Comparative performance analysis of priority rules in a constrained flow shop with multiple processors environment. *European J. Oper. Res.* vol 72, pp 102-114.

[24] Johnson, S. M., 1954. Optimal two and three-stage production schedules with setup time included. *Nav. Res. Log. Q.* vol 1 pp 61-68.

[25] Kirkpatrick, S., Gelatt, Jr., C. D., and Vecchi, M. P., 1983. Optimization by simulated annealing. *Science* vol 220, pp 671-680.

[26] Kochhar, S., and Morris, R. J. T., 1987, Heuristics methods for flexible flow lines scheduling. *Journal of Manufacturing Systems* vol 6 no 4, pp 299-314.

[27] Michalewicz, Z., 1992. *Genetic Algorithms + Data Structures = Evolution Programs.* Springer-Verlag.

[28] Murata, T., Ishibuchi, H. and Tanaka, H., 1996. Genetic algorithms for flowshop scheduling problems. *Computers ind Engng* vol 30 no 4 pp 1061-1071.

[29] Nowicki, E. and Smutnicki, C., 1995. The flow shop with parallel machines. A tabu search approach. *Instytut Cybernetyk Technicznej. Politechnik Wroclawskiej.*

[30] Pinedo, M, 1995. *Sheduling: theory, algorithms, and systems.* Prentice-Hall, Inc.

[31] Reeves, C. R., 1995, A genetic algorithm for flowshop sequencing. *Computers Ops. Res.*, vol 22, no 1, 5-13.

[32] Salvador, M. S., 1973, A solution to a special case of flowshop scheduling problems, in: *S.E. Elmaghraby (ed.), Symposium of the Theory of Scheduling and Applications*, Springer-Verlag, New York.

[33] Santos, D. L., Hunsucker, J. L. and Deal, D. E., 1995. Global lower bounds for flow shops with multiple processors, *European Journal of Operational Research* vol 80, pp 112-120.

[34] Santos, D. L., Hunsucker, J. L. and Deal, D. E., 1996. An evaluation of sequencing heuristics in flow shops with multiple processors. *Computers ind. Engng*, vol 30, no 4, pp 681-692.

[35] Sawik, T. J., 1993, A scheduling algorithm for flexible flow lines with limited intermediate buffers. *Applied Stochastic Models and Data Analysis, Special Issue on Manufacturing Systems*, vol 9, pp 127-138.

[36] Sawik, T. J., 1995, Scheduling flexible flow lines with no in-process buffers. *Int. J. Prod. Res*, vol 33, no. 5, 1357-1367.

[37] Sriskandarajah, C., and Sethi, S. P., 1989, Scheduling algorithms for flexible flow-shops: worst and average case performance. *European Journal of Operational Research*, vol 43, pp 143-160.

[38] Whitley, D., 1989. The GENITOR algorithm and selective pressure, *Proceedings of the Third International Conference on Genetic Algorithms*, pp 116-121. Morgan Kaufmann. *apud* Whitley, D., 1993. A genetic algorithm tutorial. Technical report CS-93-103. *Department of computer science, Colorado state university.*

[39] Whitley, D., 1993. A genetic algorithm tutorial. Technical report CS-93-103. *Department of computer science, Colorado state university.*

[40] Wittrock, R.J., 1985. Scheduling algorithms for flexible flow lines. *IBM J. Res. Dev.* vol 29, pp 401-412.

[41] Wittrock, R. J., 1988, An adaptable scheduling algorithm for flexible flow lines. *Operations Research*, vol 36, pp 445-453.