ALGORITMOS GENÉTICOS E DE SIMULATED ANNEALING: APLICAÇÃO AO PROBLEMA DE PLACEMENT E TÉCNICAS DE PARALELIZAÇÃO

Jonas Knopman

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO EM ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE DOUTOR EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovada por:

Prof. Nélson Maculan Filho, D.Habil.
(Presidente)

Prof. Júlio Salek Aude, PhD
(Orientador)

Prof. Marco Aurélio Cavalcanti Pacheco, PhD

Prof. Antonio C. Mesquita Filho, Dr. d'Etat

Prof. Adriano Joaquim de O. Cruz, PhD

RIO DE JANEIRO, RJ - BRASIL
ABRIL DE 1996

KNOPMAN, JONAS

Algoritmos genéticos e de simulated annealing: aplicação ao problema de placement e técnicas de paralelização (Rio de Janeiro) 1996.
viii, 127 p. 29,7 cm (COPPE/UFRJ, D.Sc. Engenharia de Sistemas e Computação, 1996)

Tese - Universidade Federal do Rio de Janeiro, COPPE.

1. Algoritmos paralelos
I. COPPE/UFRJ II. Título (série)

Para a próxima geração:
Eduardo e Ricardo.

Agradecimentos

"É preciso exprimir, traduzir, explicar.
Ninguém sente senão o que soube falar..."

Sou profundamente grato ao meu orientador, Professor Júlio Salek, pelo contínuo interesse com que acompanhou este trabalho e pelas inúmeras sugestões fornecidas. Sua capacidade de trabalho e dedicação à pesquisa são fonte constante de motivação para todos os seus alunos. A delicadeza no trato com as pessoas e o contínuo incentivo aos seus alunos fazem dele uma pessoa admirada e querida por todos nós. "Quando eu crescer vou ser como o Salek" é uma brincadeira freqüente entre seus alunos e colegas de trabalho, mas é também uma secreta aspiração a todos os que, como eu, têm o prazer de conviver com o Professor Salek.

Aos amigos do Núcleo de Computação Eletrônica pelo constante incentivo.

À minha mãe Sarah Knopman. Sua dedicação à família e os constantes exemplos de retidão de comportamento moldaram em mim atributos pelos quais serei eternamente grato. Gostaria também de agradecer minha irmã, Ester Knopman, pelo constante interesse com que acompanha minha vida acadêmica.

Finalmente, existe outra pessoa a quem devo agradecer: minha esposa Eliana Bayer Knopman. Além do constante incentivo, seu apoio traduziu-se materialmente ao assumir grande parte das minhas tarefas e obrigações. Eliana foi, nos últimos tempos, chefe da casa, mãe e pai dos meus filhos. Eduardo e Ricardo, por serem crianças adoráveis, tornaram muito duro este afastamento involuntário. Eliana, por tudo o que você fez por mim, minha eterna gratidão e amor.

Jonas Knopman

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para obtenção do grau de Doutor em Ciências (D.Sc.).

ALGORITMOS GENÉTICOS E DE SIMULATED ANNEALING: APLICAÇÃO AO PROBLEMA DE PLACEMENT E TÉCNICAS DE PARALELIZAÇÃO

Jonas Knopman

ABRIL, 1996

Orientador: Prof. Júlio Salek Aude

Programa: Engenharia de Sistemas e Computação

Os algoritmos genéticos e de *simulated annealing* são conhecidos por serem capazes de encontrar o mínimo global no espaço de solução de problemas de otimização a um custo computacional elevado. Esta tese se dedica à aplicação destes algoritmos à solução do problema de *placement* de macrocélulas em circuitos VLSI e ao estudo de técnicas de paralelização destes algoritmos considerando uma plataforma formada por *clusters* de estações de trabalho e o ambiente de programação paralela PVM.

Ambos os algoritmos demonstraram ser capazes de obter boas soluções para o problema de *placement* tanto na versão seqüencial como na versão paralela. A sintonia dos parâmetros de controle do algoritmo genético para solução do problema de *placement* mostrou-se muito mais difícil que a do algoritmo de *simulated annealing*. A implementação seqüencial do algoritmo genético é significativamente mais lenta do que a do algoritmo de *simulated annealing*. As implementações paralelas dos algoritmos genéticos proporcionaram, no entanto, ganhos significativos em tempo de processamento.

A tese tem como contribuição fundamental a apresentação de estudos experimentais detalhados da sintonia dos parâmetros de controle dos algoritmos seqüenciais ao problema de *placement* e dos efeitos produzidos por diferentes técnicas de paralelização dos algoritmos. Para os algoritmos genéticos fica demonstrado que o tamanho da população e a estratégia de incialização adotada para posicionamento dos genes no cromossoma são fundamentais para que o algoritmo convirja para soluções próximas à ótima. Além disso, a tese propõe uma versão otimizada da paralelização do algoritmo de *simulated annealing* especulativo, aplicável, com ganho de velocidade significativo, em plataformas multiprocessadas homogêneas. Mostra ainda que a paralelização de algoritmos de *simulated annealing* com base no método de cadeias de comprimento mínimo não conduz a soluções próximas da solução ótima Finalmente, propõe uma técnica de paralelização de algoritmos genéticos com *speed-up* aproximadamente linear com o número de processadores.

Abstract of Thesis presented to COPPE/UFRJ as partial fulfillment of the requirements for the degree of Doctor of Science (D.Sc.)

GENETIC AND SIMULATED ANNEALING ALGORITHMS: APPLICATIO TO THE PLACEMENT PROBLEM AND PARALLELIZATION TECHNIQUES

Jonas Knopman

APRIL, 1996

Thesis Supervisor: Prof. Júlio Salek Aude
Department: Systems and Computing Engineering

The genetic and simulated annealing algorithms are well known to be able to find the global minimum within the solution space of optimization problems with a high computational cost. This thesis is dedicated to the application of both algorithms to the solution of the VLSI macrocell placement problem and to the study of parallelization techniques of both algorithms considering the use of PVM on a cluster of workstations.

Both algorithms have shown to be able to find the optimal solution to the placement problem in the sequential and parallel versions. The tuning of the control parameters of the genetic algorithm to the solution of the placement problem has proved to be much more difficult than that of the simulated annealing algorithm. The sequencial implementation of the genetic algorithm has shown to be much slower than the simulated annealing algorithm. However, the parallel implementations of the genetic algorithms achieved very good speed-ups.

The fundamental contribution of this thesis is the detailed presentation of experimental studies on the tuning of the control parameters of the sequential algorithms to the placement problem and the effects produced by the different parallelization techniques of the algorithms. For the genetic algorithms it is shown that the population size and the initialization procedure adopted for the placement of genes within the cromossomes are fundamental for the algorithm to reach the optimal solution. In addition, this thesis proposes an optimized version of the parallelization of the speculative simulated annealing algorithm which can produce important speed-ups in homogeneous platforms. It also shows that the parallelization method of the simulated annealing algorithm based on the use of chains with minimum length are not able to reach the optimal solution. Finally, a parallel genetic algorithm with almost linear speed up is proposed and implemented.

ÍNDICE

Capítulo 1 - Introdução.

1.1 - O problema de placement 1
1.2 - O ambiente PVM 4
1.3 - Contribuições deste trabalho 7
1.4 - Organização do texto 8

Capítulo 2 - Simulated Annealing - O Algoritmo Seqüencial

2.1 - Introdução 9
2.2 - O algoritmo básico 11
2.3 - Implementação do algoritmo SA 12

Capítulo 3 - Paralelização do algoritmo Simulated Annealing.

3.1 - Introdução 14
3.2 - O conjunto mínimo [Krav87] 16
3.2.1 - Implementação 19
3.2.2 - Conclusões 25
3.3 - O Algoritmo Especulativo [Sohn95] 26
3.3.1 - Conclusões 30
3.4 - O Algoritmo Adaptativo 30
3.4.1 - Análise do speedup em presença do custo de comunicação 31
3.4.2 - As modificações propostas 33
3.4.3 - Resultados 41
3.5 - O Algoritmo em Altas Temperaturas 42
3.5.1 - Fundamentos 42
3.5.2 - Algumas considerações a respeito da estratégia de Cadeias Paralelas. 46
3.5.3 - Resultados 47
3.6 - Sumário e Conclusões 47

Capítulo 4 - O Algoritmo Genético Seqüencial

4.1 - Introdução 49
4.2 - O Modelo de População Única: Ajuste dos parâmetros e escolha dos operadores 53
4.2.1 - Representação das variáveis do problema 53
4.2.2 - A função custo 56

4.2.3 - A função aptidão 59
4.2.4 - O método de seleção 62
4.2.5 - O operador de crossover 62
4.2.6 - O operador de mutação 65
4.2.7 - Resultados 67
4.2.8 - Conclusões 69
4.3 - Organização da população: O modelo paralelo. 69
4.3.1 - O modelo paralelo convencional 69
4.3.2 - Populações superpostas. O algoritmo paralelo modificado (mpGA) 70
4.3.3 - Distribuição da população e processo de reprodução 71
4.3.4 - Pseudo código 73
4.3.5 - Resultados 73
4.4 - Algumas técnicas não convencionais 76
4.4.1 - População de Tamanho Variável 76
4.4.2 - O algoritmo híbrido 83
4.4.3 - O Operador de Inversão e o Crossover Cíclico 87
4.4.4 - Codificação Expandida 91
4.4.5 - Crossover por Redes 91
4.4.6 - Crossover por Regiões do Plano 93
4.4.7 - Pré alocação de módulos 94
4.4.8 - Efeito dos Operadores de Crossover e Mutação 99
4.5 - Estratégias Evolucionárias e outras estratégias 103
4.5.1 - A heurística TABU 103
4.5.2 - Aprendizado Incremental Baseado em Populações (AIBP) 106
4.6 - Conclusões 108

Capítulo 5 - Paralelização do Algoritmo Genético

5.1 - Introdução 110
5.2 - O modelo centralizado 110
5.3 - O modelo distribuído 113

Capítulo 6 - Conclusões

6.1 - Contribuições 118
6.2 - Trabalhos Futuros 120

Capítulo 7 - Bibliografia 121

CAPÍTULO 1

Introdução

O ambiente típico de CAD consiste de um certo número de estações de trabalho conectadas por meio de uma rede de alta velocidade, o que permite à equipe de projetistas o acesso conjunto à base de dados. A maioria dos problemas individuais é resolvida em uma única estação. Problemas computacionalmente intensivos tais como simulação de circuitos ou problemas de alocação de recursos são, algumas vezes, executados em máquinas remotas, mais potentes do que as estações de trabalho. Estas máquinas remotas podem ser super computadores ou alguma máquina especialmente projetada para resolver o problema em questão. No entanto, o poder computacional do ambiente paralelo representado pelas estações de trabalho raramente é usado. Esta tese explora a utilização de uma classe de algoritmos baseados na natureza (Algoritmos Genéticos e *Simulated Annealing*) os quais podem fazer efetivo uso deste recurso computacional. Os algoritmos propostos buscam reduzir o tempo de processamento pela distribuição do esforço computacional por todos os processadores da rede.

O problema particular atacado é o problema de posicionamento (*placement*) de módulos em circuitos impressos ou circuitos integrados. Este é um problema bastante estudado, com diversas soluções paralelas bem sucedidas descritas na literatura. Todas estas soluções requerem, no entanto, comunicação intensiva entre os processadores em máquinas paralelas especiais com memória compartilhada ou redes de interconexão dedicadas. A principal contribuição desta tese é mostrar como este problema pode ser resolvido pela classe de algoritmos propostos através de uma implementação paralela eficiente rodando em uma rede de estações de trabalho usando o ambiente PVM.

1.1) O problema de *placement*:

Fisicamente, o projeto de um circuito envolve o posicionamento de um conjunto de módulos eletrônicos em uma placa ou substrato. Os elementos do circuito existentes nos módulos levam à existência de pinos de conexão nas bordas dos módulos. Vários subconjuntos destes pinos devem ser interconectados para formar *redes*. Uma vez que os módulos contêm vários pinos, eles, geralmente, pertencem a mais de uma rede. Formalmente, o problema de *placement* consiste em encontrar a localização ótima dos módulos de modo a minimizar uma função custo segundo alguma norma.

Na prática, existem uma série de requisitos (geralmente conflitantes) que devem ser minimizados no projeto de um circuito eletrônico: o comprimento da fiação, ruído, dissipação de calor, área, etc. Entretanto, acima de tudo, o posicionamento utilizado deve prover fácil roteamento dos sinais para formar as redes.

Em termos práticos, é impossível incorporar todos os objetivos de projeto em uma única função custo. Entretanto, minimizar o comprimento total da fiação parece ser uma aproximação razoável dos demais objetivos de projeto. De fato, circuitos com ligações curtas entre pinos têm, em geral, baixa dissipação de calor, pequena área e baixo nível de ruído.

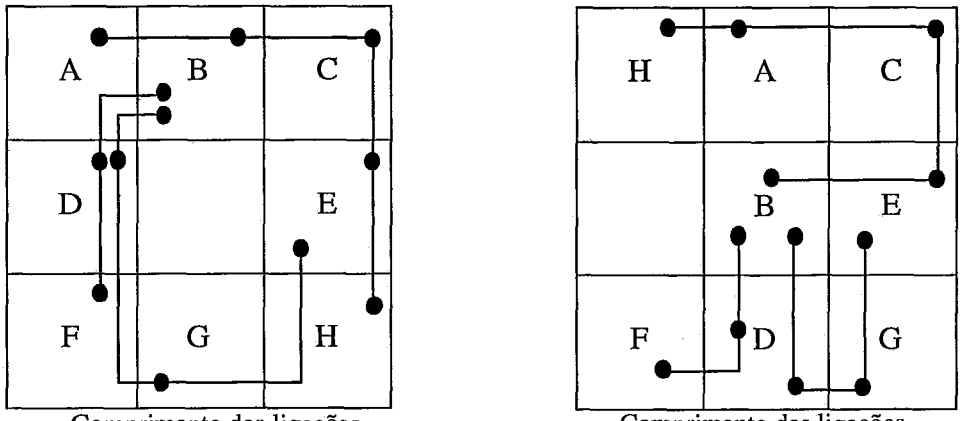
Finalmente, deve-se notar que a distância usada para medir o custo de um *placement* é a distância retilínea ou distância *Manhattan*, isto é, se (x_1, y_1) e (x_2, y_2) são dois pontos no plano a distância retilínea d é dada por:

$$d = |x_1 - x_2| + |y_1 - y_2| \tag{1.1}$$

Seja o seguinte exemplo composto por três redes[1] S_1, S_2 e S_3 e oito módulos A, B,..., H.

$S_1 = \{A, B, C, E, H\}$
$S_2 = \{B, D, E, G\}$
$S_3 = \{B, D, F\}$

A figura a seguir mostra duas alternativas para o *placement* dos módulos e a diferença resultante na função custo



Comprimento das ligações
13 unidades

Comprimento das ligações
9 unidades

Figura 1.1 - Diferença na função custo resultante de duas alternativas de *placement.*

O problema de *placement* de módulos é um problema NP completo e não pode, portanto, ser resolvido exatamente em tempo polinomial [Dona80, Leig83, Sahn80]. A tentativa de determinar a solução exata pela busca exaustiva em todas as configurações possíveis, demandaria tempo de processamento proporcional ao fatorial do número de módulos no circuito. Este método não tem aplicação prática, portanto, em circuitos com qualquer número razoável de módulos. Desta forma, a fim de orientar a busca entre um grande número de soluções candidatas, várias heurísticas têm sido desenvolvidas. A qualidade do *placement* obtido depende da heurística usada. A esperança é encontrar um bom *placement* com comprimento de fiação muito próximo do mínimo, sem a garantia de que o custo obtido é o mínimo global da função.

Os algoritmos de *placement* podem ser divididos em duas classes principais: algoritmos construtivos e algoritmos iterativos. Em algoritmos construtivos, os módulos são individualmente alocados (um por vez) e a sua posição permanece inalterada até o término do algoritmo. Nos métodos iterativos, o algoritmo inicia com um *placement*

[1] Em problemas reais o número dos pinos deveria aparecer ao lado da identificação dos módulos, i.e., A_1, B_5 e H_7. Para fins de clareza, a numeração dos pinos é omitida neste exemplo e as redes são definidas como conexões de módulos e não de pinos.

inicial (obtido, em geral, aleatoriamente) e repetidamente modifica-o em busca de reduções na função custo.

Outra classificação possível para os algoritmos de *placement* é em algoritmos determinísticos e probabilísticos. Algoritmos que constróem a solução usando regras de conectividade fixas ou que determinam o *placement* pela solução simultânea de sistemas de equações, são determinísticos e, para um dado problema particular, produzem sempre os mesmos resultados. Os algoritmos probabilísticos, por outro lado, examinam aleatoriamente o espaço de soluções e podem produzir resultados diferentes a cada vez que são executados. Algoritmos construtivos são, em geral, determinísticos, enquanto que os algoritmos iterativos usualmente são probabilísticos.

Alguns algoritmos clássicos para a solução do problema de *placement* são o "Método das Forças" e o "*Placement* por Partições" (*Min-Cut*). Os princípios gerais destes algoritmos serão vistos a seguir.

O método das forças.

O método das forças constitue uma classe de algoritmos que diferem grandemente em detalhes de implementação [Hana72]. O denominador comum a estes algoritmos é o método de calcular a localização "ideal" dos módulos. Este método é descrito a seguir.

Considere um *placement* inicial. Assuma que os módulos conectados por redes equipotenciais exerçam uma força de atração uns sobre os outros (Figura 1.2). A magnitude da força atraindo dois módulos é diretamente proporcional à distância entre eles (assim como na Lei de Hooke para a força exercida por molas distendidas), a constante de proporcionalidade sendo o somatório dos pesos das redes conectando os módulos. Se fosse permitido aos módulos neste sistema deslocarem-se livremente, eles moveriam-se na direção da força resultante até que o sistema atingisse o equilíbrio em uma configuração de energia mínima. Os algoritmos baseados no método das forças baseiam-se, portanto, no deslocamento dos módulos na direção da força total exercida sobre eles até que esta força seja nula.

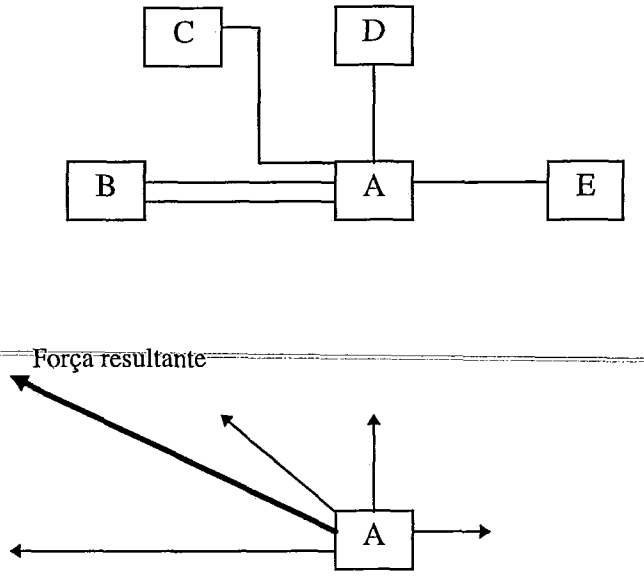


Figura 1.2 - O método das forças.

Um problema a ser resolvido é a ordem na qual os módulos devem ser avaliados e movidos. Na maioria das implementações o módulo com a maior força resultante é selecionado primeiro. Em outras implementações os módulos são selecionados aleatoriamente.

Um outro problema é para onde mover um módulo se a posição mais próxima à localização de força zero estiver ocupada (o que ocorre na maioria das vezes). Uma solução é selecionar o módulo previamente ocupando aquela posição para o próximo movimento. O processo continua até que a posição destino de um movimento esteja vazia. Um novo módulo é então selecionado.

Placement por partições (Min-cut).

O *placement* por partições constitui-se em uma importante classe de algoritmos baseados na divisão do circuito em subcircuitos densamente conectados de forma a que o número de redes seccionadas pelo particionamento seja minimizado. A maioria dos algoritmos *Min-cut* usa alguma forma modificada das heurísticas de Kernighan-Lin [Kern70] e Fiduccia-Mattheyses [Fidu82] para o particionamento.

O algoritmo de Kernighan-Lin tem início com um particionamento aleatório inicial, o qual divide o conjunto dos módulos em dois subconjuntos disjuntos A e B. Em seguida o *net cut* (número de redes conectando módulos em A a módulos em B) é calculado. Para todos os pares (a, b), a $\in A$, $b \in B$, é determinada a redução g no *net cut* obtida pela troca de a e b (a passa para o conjunto B e b passa para o conjunto A). O valor g é chamado de "ganho da troca". Se $g > 0$, a troca é benéfica. O par (a_1, b_1) com o maior ganho g_1 é selecionado. Os módulos a_1 e b_1 são retirados dos conjuntos A e B e um novo ganho máximo g_2 é determinado para um par (a_2, b_2). O processo continua até que os conjuntos A e B estejam vazios. Em seguida determina-se um valor k tal que o ganho total

$$G = \sum_{i=1}^{k} g_i \qquad (1.2)$$

seja maximizado e troca-se os pares de módulos (a_1, b_1), (a_2, b_2), ..., (a_k, b_k). O processo é repetido enquanto $G > 0$ e $k > 0$.

Diversos outros algoritmos foram propostos em literatura: o algoritmo de Goto [Goto86], o algoritmo de Dunlop [Dunl85] e algoritmos baseados em técnicas de otimização [Hall70, Chen84, Tsay88]. Todos estes algoritmos caracterizam-se por não possuir mecanismos para "escapar" de pontos de mínimos locais, característica comum aos métodos de *Simulated Annealing* e Algoritmos Genéticos. Estes algoritmos, no entanto, demandam tempos de processamento excessivamente longos, o que os torna um nicho atrativo para a aplicação de técnicas de paralelização. O estudo dos algoritmos genéticos e de *Simulated Annealing* aplicados ao problema de *placement*, bem como o estudo de técnicas de paralelização serão vistos em detalhes no restante desta Tese.

1.2) O ambiente PVM:

O ambiente PVM (*Parallel Virtual Machine*) constitui-se em um conjunto integrado de ferramentas de software e bibliotecas de funções que emulam uma arquitetura paralela a partir de uma coleção de máquinas heterogêneas, de alguma forma interligadas. O principal objetivo do ambiente PVM é permitir que esta coleção de

máquinas heterogêneas seja utilizada de forma cooperativa na solução concorrente ou paralela de uma dada aplicação.

Em linhas gerais, as principais características do ambiente PVM são:

- A arquitetura paralela é configurada pelo usuário: Os processadores integrantes da arquitetura paralela são escolhidos pelo usuário para a execução de uma dada aplicação. Máquinas compostas por uma única CPU ou máquinas multiprocessadas (memória compartilhada ou troca de mensagens) podem tomar parte da arquitetura paralela virtual. A arquitetura paralela pode ser alterada em tempo de execução, pela adição ou retirada de processadores.
- Acesso transparente ao hardware: As aplicações podem enxergar a arquitetura paralela como um conjunto indistinto de processadores ou podem preferir executar partes do código em máquinas específicas.
- Computação baseada em processos: A unidade de paralelismo no PVM é a tarefa (*task*) - usualmente um processo Unix. Não existe uma correspondência obrigatória entre um processo em um processador. Particularmente, vários processos podem executar em um único processador.
- Modelo de troca de mensagens: Um conjunto de tarefas, resultantes da decomposição de uma aplicação (decomposição funcional, de dados ou híbrida), cooperam através do envio e recebimento explícito de mensagens umas para as outras. O tamanho das mensagens é limitado apenas pela quantidade de memória disponível.
- Suporte à heterogeneidade: O ambiente PVM suporta heterogeneidade ao nível das máquinas, redes ou aplicações. Em particular, no que diz respeito à troca de mensagens, PVM permite o envio e recebimento de mensagens entre máquinas utilizando diferentes representações de dados.
- Suporte a máquinas multiprocessadas: De modo a tirar proveito do hardware existente em máquinas multiprocessadas, o sistema PVM utiliza, em tais casos, o ambiente nativo para a troca de mensagens.

O sistema PVM é composto por duas partes. A primeira delas é um programa residente (*daemon*), chamado de *pvmd*. Quando um usuário deseja executar uma aplicação PVM, ele primeiro cria a máquina paralela virtual disparando um processo *daemon* em cada um dos nós integrantes da máquina. A aplicação pode ser então iniciada a partir de uma linha de comando Unix em qualquer uma das máquinas. Uma mesma máquina pode tomar parte, simultaneamente, das máquinas paralelas de mais de um usuário. De forma similar, um mesmo usuário pode executar simultaneamente mais de uma aplicação PVM.

A segunda parte do sistema é uma biblioteca de rotinas PVM para a troca de mensagens, controle e configuração da máquina virtual e disparo de processos. Existem bibliotecas apropriadas para a chamada das funções a partir de programas escritos em C, C++ ou Fortran.

Todas as tarefas PVM são identificadas por um número inteiro (*Task Identifier* ou TID). Os remetentes ou destinatários das mensagens são identificados através do TID. Uma vez que os identificadores devem ser únicos para toda a máquina virtual, eles são atribuídos pelo *pvmd* local e não podem ser alterados pelo usuário. A biblioteca PVM contém várias rotinas que retornam o TID de um processo de modo que uma tarefa possa facilmente identificar os outros processos que fazem parte da aplicação.

Existem aplicações onde é natural pensar em "grupos de tarefas" e operações comuns a todo o grupo (por exemplo, uma mensagem enviada em *broadcasting* para todos os processos no grupo). Também podem ocorrer situações em que o programador gostaria de identificar suas tarefas pelos números inteiros $[0, p)$ onde p é o número de processos. Em resposta a esses requisitos, o ambiente PVM inclui o conceito de grupo onde o nome do grupo é atribuido pelo usuário. As tarefas são identificadas unicamente no grupo através de um identificador GID. A primeira tarefa a se juntar ao grupo recebe o identificador GID = 0. As demais tarefas recebem identificadores crescentes e consecutivos. Qualquer tarefa PVM pode entrar ou sair de qualquer grupo a qualquer tempo sem informar o ocorrido a qualquer tarefa participante dos grupos afetados. Um processo pode integrar simultaneamente mais de um grupo e uma tarefa pode mandar mensagens em *broadcasting* para grupos dos quais ela não faz parte.

O paradigma para a programação PVM pode ser expresso na seguinte forma: um usuário escreve um ou mais programas seqüenciais em C, C++ ou Fortran que contêm chamadas à biblioteca PVM. Estes programas correspondem ao código das tarefas que constituem a aplicação paralela (Não existe correspondência entre o número de códigos distintos e o número de tarefas. Em um modelo SIMD, por exemplo, todos os processos compartilham o mesmo código). Os programas são então compilados para as arquiteturas alvo e o código executável resultante é colocado em algum lugar acessível pelos processos *daemon*. Para executar uma aplicação, o usuário, tipicamente, dispara manualmente uma cópia de um dos processos (o processo mestre) a partir de um dos nós da máquina paralela virtual. O processo mestre então dispara os demais processos, o que resulta em um conjunto de tarefas ativas, processando localmente, e trocando mensagens no intuito de resolver o problema sendo abordado. A Figura 1.3 mostra um trecho de uma aplicação típica PVM. Os seguintes passos principais podem ser identificados na figura:

1. A chamada a *pvm_mytid( )* inscreve o processo mestre no ambiente PVM.

2. *pvm_spawn( )* cria o processo escravo.

3. O processo mestre prepara o *buffer* de transmissão e o envia para o escravo.

4. O processo escravo recebe a mensagem especificando o TID do mestre e o mesmo TAG usado por este no envio da mensagem.

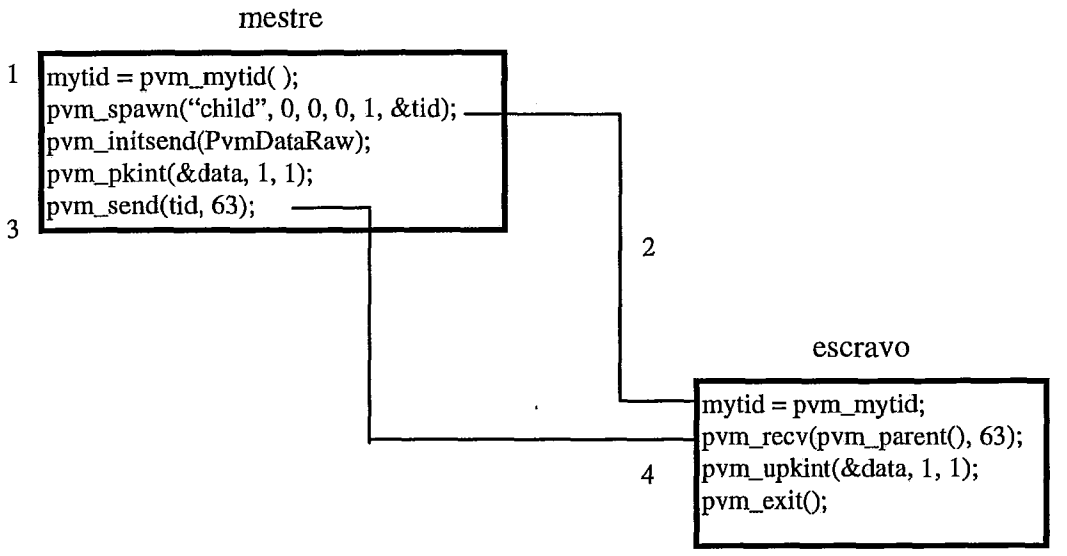


Figura 1.3 - Trecho de código de uma aplicação típica PVM.

O ambiente PVM utilizado em todas as implementações descritas neste trabalho constitui-se por um *cluster* de estações de trabalho IBM©25T equipadas com o processador PowerPC, utilizadas em modo exclusivo (isto é, não haviam outros processos rodando nas máquinas durante os testes). As estações são equipadas com uma interface padrão Ethernet e interligadas por uma rede TCP/IP. A rede de comunicação é um fator extremamente limitante em nosso ambiente de programação: O barramento Ethernet pode transmitir dados a uma taxa máxima de 10 Mbits/s e apenas uma mensagem pode ocupar o barramento de cada vez. Desta forma, qualquer estratégia de paralelização que pretenda ser efetiva neste ambiente, deve manter a quantidade de troca de mensagens entre os processadores reduzida a um mínimo. Este é, em verdade, o grande desafio a ser enfrentado: O alto custo de comunicação inviabiliza, neste ambiente, a maioria das soluções paralelas encontradas em literatura para o problema de *placement.*

1.3) Contribuições deste trabalho:

Pode-se enumerar como contribuições deste trabalho:

- A construção de um algoritmo genético paralelo efetivo para a solução do problema de *placement* no ambiente disponível.
- Explicitar a real contribuição dos operadores genéticos clássicos (*crossover* e mutação) e avaliar o impacto de variantes destes operadores propostas na literatura.
- Estudo detalhado do efeito dos diversos parâmetros dos algoritmos genéticos (tamanho da população, probabilidade de *crossover*, probabilidade de mutação, tamanho do plano de alocação, inicialização da população, etc.) sobre o problema de *placement*

- Estudo do efeito da aplicação de algumas técnicas não convencionais de algoritmos genéticos (populações de tamanho variável, algoritmos híbridos, codificação expandida, etc.) aplicadas ao problema de *placement.*
- Desenvolvimento de técnicas que permitiram lidar com a forte epistasia (a influência de um gene no cromossoma sobre a "qualidade" do indivíduo depende do valor de outros genes na cadeia) característica do problema de *placement.*
- Uma comparação crítica entre o desempenho das duas classes de algoritmos (algoritmos genéticos e *Simulated Annealing*) aplicados ao problema de *placement.*
- A construção de um *annealing* paralelo efetivo para a solução do problema de *placement* no ambiente disponível.
- Uma visão crítica a respeito de técnicas de paralelização do algoritmo de *annealing* freqüentemente referenciadas em literatura.

1.4) Organização do texto:

O restante desta Tese é organizado da seguinte forma:

O Capítulo II descreve os fundamentos do algoritmo *Simulated Annealing*. É apresentada ainda uma implementação serial, base dos esforços de paralelização do algoritmo.

O Capítulo III descreve a implementação paralela do algoritmo de *annealing*. São apresentados resultados e o algoritmo proposto é comparado com outras implementações descritas na literatura.

O Capítulo IV descreve os fundamentos dos algoritmos genéticos. São apresentadas as diversas alternativas experimentadas para a construção de um algoritmo serial efetivo para o problema de *placement.*

O Capítulo V descreve a implementação paralela do algoritmo genético. São apresentados os resultados em termos do *speedup* obtido em relação à versão serial e em comparação ao *Simulated Annealing* paralelo.

Finalmente, no Capítulo VI são apresentadas as conclusões e as direções para trabalhos futuros.

CAPÍTULO 2

Simulated Annealing - O algoritmo seqüencial

2.1) Introdução:

O algoritmo *Simulated Annealing* (SA) consiste em uma técnica de otimização baseada no processo físico de recozimento de metais. Sua principal característica é a possibilidade de explorar o espaço de estados do problema permitindo movimentos *hill climbing*, isto é, movimentos que piorem o custo nas vizinhanças de mínimos locais. Nos últimos anos o algoritmo SA tornou-se um método popular para a otimização de problemas complexos. O algoritmo tem um nicho permanente no arsenal de métodos disponíveis para Otimização Combinatória devido à sua simplicidade computacional. A desvantagem desta solução probabilística é o tempo necessário para atingir-se uma solução próxima da ótima.

De modo a permitir uma melhor compreensão do algoritmo, consideremos a analogia com o processo físico. Quando um pedaço de metal é moldado a estrutura cristalina interna é alterada. Uma parte da energia empregada para moldar o metal é absorvida pelo material sob a forma de imperfeições na cadeia cristalina. A nova estrutura é, tipicamente, mais sujeita a fraturas. Dessa forma, se for necessário moldar o metal ainda mais, a estrutura original deve ser recuperada. Isso é feito aquecendo-se o material a uma temperatura próxima à metade do seu ponto de fusão e mantendo-o nesta temperatura por um certo tempo. O metal é então resfriado lentamente. A energia térmica em altas temperaturas e o resfriamento lento possibilitam aos átomos do metal caminhar em direção à configuração de energia mínima, nominalmente cristais perfeitos.

Os métodos computacionais, em analogia, permitem, na fase inicial do algoritmo de *annealing*, que os parâmetros do problema variem de uma grande quantidade (simulando os efeitos de alta temperatura). O parâmetro controlando a quantidade de alteração é freqüentemente chamado de T ou temperatura. Nas fases seguintes do algoritmo (temperaturas mais baixas) reduz-se a variação permitida para os parâmetros do problema. A idéia é que em altas temperaturas evite-se que a solução caia num mínimo local. Então, uma vez no entorno do mínimo global, as fases de baixa temperatura permitem o refinamento da solução.

O algoritmo SA pode ser visto como um algoritmo probabilístico de busca por um mínimo global. Ao contrário dos métodos determinísticos de *hill climbing*, configurações que piorem o custo também podem ser aceitas. Partindo de uma configuração inicial, o algoritmo repetidamente gera novas configurações vizinhas. Configurações que contribuam para diminuir a função custo são aceitas. Configurações que aumentem a função custo de uma quantidade ΔC são aceitas com probabilidade $e^{-\Delta C/T}$. Valores altos de T levam à aceitação de grandes deteriorações na função custo ao passo que, em baixas temperaturas, somente configurações melhores são aceitas. Em cada uma das temperaturas devem ser gerados estados suficientes para atingir o equilíbrio térmico.

SA não é um algoritmo propriamente dito [Laar87]. Ele é melhor definido como um paradigma para se construir algoritmos para a solução de problemas específicos de otimização combinatória. O projeto de um algoritmo de *annealing* compreende 5 partes:

1. **Espaço de estados**: A representação dos estados possíveis para o sistema deve ser simples e facilitar a fácil geração de perturbações.
2. **Geração de novos estados:** As regras para a geração de novos estados devem ser flexíveis o bastante para permitir que qualquer solução seja atingida a partir de transformações no estado inicial. Além disso, os novos estados devem ser gerados a um custo relativamente baixo uma vez que muitos deles devem ser gerados durante a execução do algoritmo.
3. **Cálculo do custo de uma configuração:** O custo de uma configuração deve ser calculado incrementalmente de modo que o tempo para avaliar uma nova configuração seja mínimo.
4. **Resfriamento:** O esquema de resfriamento da temperatura T é crucial para o bom desempenho do algoritmo: temperaturas iniciais muito baixas ou um resfriamento muito rápido podem levar a soluções não satisfatórias. Temperaturas iniciais muito altas ou um resfriamento muito lento levam a um custo de processamento muito alto.
5. **Estruturas de dados:** A habilidade de propor e avaliar novas configurações eficientemente depende de uma representação eficiente dos dados do problema. Para o problema de *placement*, isto significa estruturas projetadas de modo que a localização dos módulos e a conectividade do circuito sejam facilmente acessadas.

O algoritmo SA pode ser descrito matematicamente por meio de cadeias de Markov. Pode-se mostrar que, se forem executados um número suficiente de passos em cada temperatura e a estratégia de resfriamento for apropriada, o algoritmo converge para a solução ótima com probabilidade 1 [Aart87, Mitr85]. Ainda que esta certeza forneça pouca informação em como escolher os parâmetros adequados para cada aplicação específica, ela serve para dar confiança no bom comportamento do algoritmo.

2.2) O algoritmo básico:

O algoritmo básico é descrito a seguir:

```
SimulatedAnnealing( )        {
/*
Variáveis:
s_i     ⇒     Estado corrente na iteração i do loop interno.
T_k     ⇒     Temperatura no passo k (loop externo).
C(s)    ⇒     Custo da configuração s.
*/
      Começa com um estado inicial s_i; k = 0;
      T_k = T_0;                    /* temperatura inicial                */
      Repita                        /* loop externo                       */
            Repita                  /* loop interno                       */
                  Gera uma nova configuração s_j;
                  Calcula ΔC = C(s_j) - C(s_i);
                  Se aceita(ΔC, T_k)
                        s_i = s_j;
            Até equilíbrio na temperatura;
            T_{k+1} = α_k * T_k;    /* redução de temperatura             */
            k = k + 1;
      Até congelamento;
}
```

A aceitação de um novo estado s_j é determinado pela função *aceita( )* cuja estrutura é mostrada a seguir.

```
aceita(ΔC, T)        {
/*
Retorna 1 se a configuração é aceita;
random( ) retorna um número real no intervalo [0,1]
*/
      Se (ΔC ≤ 0)        retorna 1;
      y = exp(-ΔC/T);
      r = random( );
      Se (r < y)         retorna 1;
      retorna 0;
}
```

Observe que novos estados caracterizados por $\Delta C \leq 0$ sempre satisfazem o critério de aceitação. No entanto, para estados caracterizados por $\Delta C > 0$ o parâmetro T tem um papel fundamental. Se T é muito grande, então r provavelmente será menor do que y e o novo estado é quase sempre aceito. Se T é pequeno, próximo de 0, então somente novos estados que são caracterizados por variações muito pequenas de $\Delta C > 0$ têm alguma chance de serem aceitos.

Nos testes, os melhores resultados foram obtidos inicializando-se o parâmetro T com um valor bastante alto, onde praticamente todos os estados propostos são aceitos. Além disso, permite-se que o sistema atinja o "equilíbrio" em cada temperatura. Isto é, um número suficiente de iterações é realizado no loop interno. O critério de parada é

satisfeito quando o valor da função custo permanece o mesmo após várias iterações do processo de annealing.

2.3) Implementação do algoritmo SA:

O algoritmo SA serial usado neste trabalho, base de nossas tentativas de paralelização, é inspirado no algoritmo TimberWolf [Sech85]. TimberWolf tem as seguintes características principais:

O espaço de estados: As células são dispostas em uma grade uniforme. É permitido às células ocuparem a mesma posição na grade, fato este refletido por um aumento na função custo.

Geração de novos estados: O algoritmo começa com um *placement* aleatório. Novos estados são obtidos trocando-se duas células de lugar (*swap*) ou movendo-se uma célula para outra localização (*displacement*). O uso de ambos os métodos é necessário para atingir os melhores resultados. O problema de duas células tenderem a ocupar o mesmo lugar no espaço é resolvido introduzindo-se uma função penalidade que aumenta a função custo quando houver sobreposição. A proporção entre o número de *displacements* e *swaps* tem um efeito pronunciado na qualidade final do *placement*. Os testes mostraram que uma proporção de 4 para 1 fornece os melhores resultados. Isto é implementado gerando-se dois números aleatórios. O primeiro entre 1 e o número de células e o segundo entre 1 e o número de células multiplicado por 5. Se o segundo número gerado for menor do que o número de células, procede-se um *swap* para produzir o novo estado. Se o segundo número não corresponde ao número de uma célula procede-se a um *displacement* da célula correspondente ao primeiro número.

O *displacement* de uma célula é controlado por uma janela delimitadora a qual limita o campo de deslocamento de uma célula. Isto se deve ao fato de que nos estágios finais do algoritmo o *displacement* de uma célula tem pouca chance de ser aceito a menos que ele seja muito pequeno. Limitando-se o campo de deslocamento nas últimas fases do algoritmo, as células passam a efetuar muitos deslocamentos pequenos buscando reduzir o número de sobreposições e o custo da fiação. A janela limitadora é definida como uma região retangular centrada na célula a ser movida. No início do algoritmo, quando T é muito grande, essa janela tem dimensões iguais a duas vezes a dimensão da grade. As dimensões da janela são reduzidas proporcionalmente ao logaritmo de T. A nova localização é escolhida randomicamente no interior da janela delimitadora. Os *swaps* de células são também controlados pelas janelas limitadoras. Duas células são trocadas de posição somente se a janela puder ser posicionada de forma a conter as duas células.

A função custo: A função custo para o problema de *placement* é baseada no comprimento estimado da fiação. O comprimento de um fio ligando vários pinos - uma rede - é estimado pelo semiperímetro do retângulo que envolve todos os pinos. Para uma rede de 2 células esta medida coincide com a distância Manhattan. A função *overlapping( )*, a qual penaliza sobreposições de células, também influencia o valor da função custo. Uma característica adicional do sistema é a de que é possível atribuir pesos diferentes à altura e à largura do retângulo usado para calcular o custo de uma rede. Com isso consegue-se privilegiar ligações horizontais ou verticais, conforme o caso. De modo a não ser necessário recalcular totalmente a função custo a cada novo

movimento, armazena-se, para cada módulo, a lista das redes às quais ele pertence. Dessa forma somente essas redes precisam ser recalculadas. Para reduzir ainda mais o processamento do cálculo da função custo, observa-se que uma rede só precisa ser totalmente recalculada em alguns casos, quando o módulo a ser movido estabelece uma das fronteiras do retângulo.

Estratégia de annealing: É adotado um esquema simples $T_{k+1} = \alpha_k * T_k$, $\alpha_k < 1$, com uma temperatura inicial arbitrariamente alta ($T_0 = 1000.0$), determinada empiricamente. A exemplo da estratégia empregada em [Sech85], α_k é maior (aproximadamente 0,95) durante as fases do algoritmo em que a função custo decresce mais rapidamente. Além disso α_k tem valores menores (aproximadamente 0,90) nas fases inicial e final do algoritmo. Um número constante de movimentos é efetuado em cada temperatura para atingir o equilíbrio. Este número é especificado como um múltiplo do número de células a serem alocadas. O critério de parada é implementado guardando-se o valor da função custo ao final de cada estágio do processo de annealing. O critério de parada é satisfeito quando o valor da função custo não se alterar por 20 iterações consecutivas.

Estruturas de dados: Os dados do problema são armazenados na forma de listas encadeadas. Uma lista para cada módulo, contendo as redes às quais o módulo pertence. Estas listas são usadas para se ter acesso apenas às redes que precisam ser recalculadas em função do deslocamento de um módulo. Existem ainda listas por rede contendo os módulos que pertencem a uma dada rede. Estas são usadas quando é necessário recalcular completamente o custo de uma rede.

CAPÍTULO 3

Paralelização do algoritmo Simulated Annealing

3.1) Introdução:

O algoritmo *Simulated Annealing* (SA) consiste em uma alternativa eficiente para a solução de problemas de Otimização Combinatória. Particularmente, no problema do projeto físico de circuitos, diversas implementações bem sucedidas têm sido reportadas na literatura [Leon85, Moor85, Sech85, Vecc83]. A vantagem dos métodos de *annealing* é a sua habilidade em escapar de pontos de mínimo local através da aceitação probabilística de estados que piorem a função custo. A desvantagem desta solução probabilística, característica comum de muitas soluções iterativas, é o tempo necessário para atingir-se uma solução próxima da ótima. Para um problema grande, exigindo avaliações de 10^7 estados computados em 5 ms cada um, seriam necessárias mais de 13 horas de CPU para completar o cálculo. Na medida em que as tendências apontam para uma crescente complexidade dos circuitos e requisitos de desempenho cada vez mais estritos, esta situação tende somente a piorar.

Esta combinação crítica de tempos de execução inaceitáveis e complexidade crescente dos circuitos conduziu, naturalmente, a soluções que buscavam reduzir o tempo de processamento pela utilização de *hardware* dedicado ou implementações paralelas. Neste estudo buscou-se alternativas de paralelização com baixo custo de comunicação entre os processos. Este, dado o ambiente de programação utilizado (estações de trabalho se comunicando via troca de mensagens em barramento Ethernet), é um requisito fundamental a qualquer estratégia de paralelização empregada.

Algumas tentativas de paralelizar o algoritmo SA podem ser encontradas na literatura para máquinas de memória compartilhada [Caso87] ou distribuída [Diek92]. Na maioria dos casos a paralelização é implementada ao nível dos dados. Os dados descrevendo o problema são divididos em pequenos subconjuntos e distribuídos entre os processadores que realizam de forma concorrente o SA seqüencial.

Em linhas gerais, pode-se identificar duas aproximações para acelerar o algoritmo SA empregando paralelismo. A primeira consiste em reduzir o tempo de processamento por movimento, dividindo-se a tarefa de propor e avaliar uma nova configuração entre vários processadores. Como os processadores precisam sincronizar-se ao menos uma vez por movimento, esta é uma solução de granularidade fina uma vez que envolve um grande número de mensagens trocadas entre os processadores. A segunda alternativa é processar vários movimentos completos em paralelo. Devido ao alto custo de comunicação imposto pela arquitetura alvo, não é possível explorar paralelismo de granularidade fina. Assim, as técnicas aqui propostas se concentram em algoritmos em que os processadores propõem e avaliam novos estados independentemente.

Uma restrição que se impõe é a de que os movimentos aceitos em paralelo não sejam contraditórios (por exemplo, mover a mesma célula para duas posições diferentes). Além disso, decisões erradas de aceitação/rejeição são possíveis quando os movimentos são gerados em paralelo. Durante a avaliação de cada movimento, os processadores, individualmente, não conhecem os movimentos de todas as células e assumem que somente as células envolvidas no movimento corrente vão mudar de posição. Algumas vezes a decisão local de aceitar ou rejeitar um movimento pode levar a uma decisão

global incorreta. Essa decisão incorreta pode levar não apenas a um comportamento oscilatório, como também afetar a convergência do método de *annealing*.

Existem duas alternativas para o problema das decisões erradas. A primeira é permitir movimentos paralelos contraditórios tendo em mente que eles podem afetar a convergência. Neste caso deve-se considerar a quantidade de paralelismo permissível, isto é, a fração dos módulos que podem ser movidos concorrentemente sem afetar as propriedades de convergência do algoritmo. A segunda alternativa é proibir a aceitação em paralelo de movimentos contraditórios. Para conseguir isso pode-se polarizar a seleção de movimentos - por exemplo, a grade poderia ser dividida em regiões e cada região ser alocada a um processador - ou pode-se filtrar os efeitos da aceitação de movimentos que colidem depois que eles tenham sido calculados. Essa última estratégia foi empregada nesse trabalho.

A maioria das estratégias de paralelização aqui descritas foram idealizadas a partir da observação de que o algoritmo SA não é estático. O parâmetro temperatura que controla os movimentos de *hill-climbing* altera profundamente o comportamento do algoritmo durante a sua execução. O desenvolvimento deste trabalho mostra que o comportamento dinâmico do algoritmo fornece novas oportunidades de explorar o paralelismo inerente ao processo. A estratégia de paralelização empregada consiste em mudar parâmetros de um algoritmo, ou mesmo o algoritmo usado de acordo com a fase do processo. O objetivo é construir um algoritmo paralelo eficiente, fornecendo uma solução de qualidade igual à do algoritmo seqüencial.

Uma das estratégias de paralelização empregadas, baseada no trabalho de Kravitz e Rutenbar [Krav87], consiste em manter vários processadores trabalhando simultaneamente na avaliação de uma única cadeia de Markov, preservando, segundo os autores, as propriedades de convergência do algoritmo seqüencial. Para aplicações típicas de SA, cerca de 90% de todos os movimentos gerados são rejeitados. A observação de que os movimentos rejeitados são independentes uns dos outros leva a uma implementação direta do método, uma vez que esses movimentos podem ser rejeitados independentemente em cada um dos processadores. Os movimentos aceitos levam a uma sincronização de todos os processadores envolvidos no cálculo da cadeia de Markov. Observa-se que, no início do algoritmo, um número muito grande de movimentos é aceito levando a uma baixa eficiência deste tipo de estratégia. A observação posterior de que o algoritmo de Kravitz e Rutenbar alterava a proporção entre o número de estados aceitos e rejeitados em uma cadeia de Markov (com a conseqüente perda de qualidade na solução), levou ao desenvolvimento do algoritmo especulativo baseado no trabalho de Sohn et al. [Sohn93, Sohn95] e do algoritmo especulativo adaptativo, uma proposta original aqui enunciada.

Um outro tipo de estratégia consiste em cadeias de Markov independentes, uma computada por cada processador. Como veremos adiante esta é uma estratégia adequada ao regime de temperaturas altas. O número de movimentos aceitos fornece a indicação do momento ideal para troca das estratégias.

Todos os algoritmos aqui descritos foram implementados utilizando-se um ambiente PVM [Geis94] composto por um *cluster* de estações de trabalho IBM©25T interligadas através de uma rede padrão Ethernet.

3.2) O conjunto mínimo: [Krav87]

A fim de evitar decisões equivocadas durante a avaliação de movimentos paralelos, vamos considerar a seleção de um sub-conjunto dos movimentos aceitos no qual todos os membros são mutuamente não conflitantes. Seja S o conjunto dos movimentos avaliados em paralelo e $S^n \subset S$ o subconjunto de todos os movimentos não conflitantes em S. Observe-se que pela aplicação concorrente dos movimentos em S^n a uma configuração inicial, atinge-se o mesmo resultado que seria obtido pela aplicação serial dos mesmos movimentos, em qualquer ordem. Os movimentos em S^n são serializáveis. Uma observação importante é a de que qualquer conjunto de movimentos rejeitados é serializável: nenhuma alteração é efetuada na base de dados como conseqüência de um movimento rejeitado. Podemos então redefinir o requisito de que os movimentos avaliados em paralelo sejam não conflitantes como a necessidade de que qualquer decisão de aceitação/rejeição de movimentos efetuada em paralelo, produza um resultado que possa também ser atingido pela serialização destes movimentos.

Se os movimentos são avaliados em paralelo, com alguns aceitos, alguns rejeitados e alguns simplesmente ignorados, a questão é como considerar a contribuição destes movimentos para o objetivo de alcançar o equilíbrio térmico em cada temperatura. A solução proposta por Kravitz e Rutenbar [Krav87] é encontrar algum sub-conjunto serializável $S^s \in S$ e somente considerar a contribuição destes movimentos no sentido do equilíbrio. As alternativas possíveis para a construção do conjunto serializável são melhor caracterizadas pela complexidade do processamento necessário a formar o sub-conjunto. Duas possibilidades extremas são prontamente identificadas. Na primeira delas poder-se-ia tentar determinar, a cada passo, o maior sub-conjunto dos movimentos calculados que não provocasse colisão. Essa tarefa, no entanto, parece ser extremamente difícil e, portanto, o cálculo deste sub-conjunto provavelmente anularia os potenciais benefícios advindos da exploração do paralelismo.

No outro extremo, existe um conjunto facilmente identificável e ainda grande o suficiente para ser interessante. Este conjunto pode ser determinado, praticamente, sem nenhum custo adicional de processamento. Sua construção é baseada na observação apresentada anteriormente: a seqüência formada por todos os movimentos rejeitados acrescida de um único movimento aceito é sempre serializável. É claro que, como apenas um movimento aceito é efetuado a cada passo, não há movimentos contraditórios. Esse conjunto de movimentos considerados (todos os recusados e um aceito), o mais simples dos conjuntos serializáveis, será a partir de agora referenciado como o *conjunto mínimo.* A Figura 3.1 ilustra a escolha do conjunto mínimo. É de se supor que, entre os dois extremos, hajam conjuntos não ótimos de movimentos não conflitantes os quais possam ser determinados em tempo razoável através de heurísticas adequadas. Esta possibilidade não foi, no entanto, aqui explorada.

Um modelo simples ilustra o desempenho do conjunto mínimo. Seja N o número de processadores e M o número de movimentos aceitos a cada passo. Na temperatura T, a probabilidade de aceitação de um movimento individual é $a(T)$. Uma vez que cada processador aceita um movimento com probabilidade $a(T)$ ou o rejeita com probabilidade $(1 - a(T))$, M tem uma distribuição de probabilidade binomial. $Pr(M = m)$. A probabilidade de que m movimentos sejam aceitos é dada então por:

$$Pr(M = m) = \binom{N}{m} * a(T)^m * [1 - a(T)]^{N-m} \qquad (3.1)$$

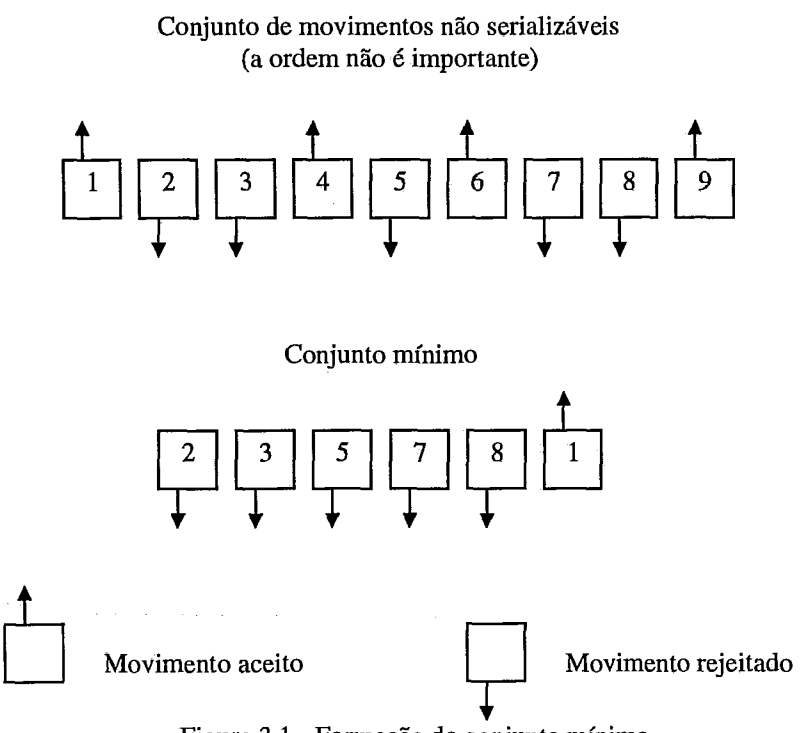


Figura 3.1 - Formação do conjunto mínimo

O tamanho esperado para o conjunto mínimo S é dado por:

$$E[S] = \Pr(M = 0) * N + \sum_{m=1}^{N} \Pr(M = m) * (N - m + 1) \qquad (3.2)$$

O gráfico da Figura 3.2 mostra o tamanho provável do conjunto mínimo como função da probabilidade de aceitação $a(T)$ para N = 10 e N = 20. Observe-se que, na hipótese do custo de comunicação entre os processos ser nulo, a função $E[S]$ representa o *speedup* máximo teórico pontual, para cada valor de $a(T)$.

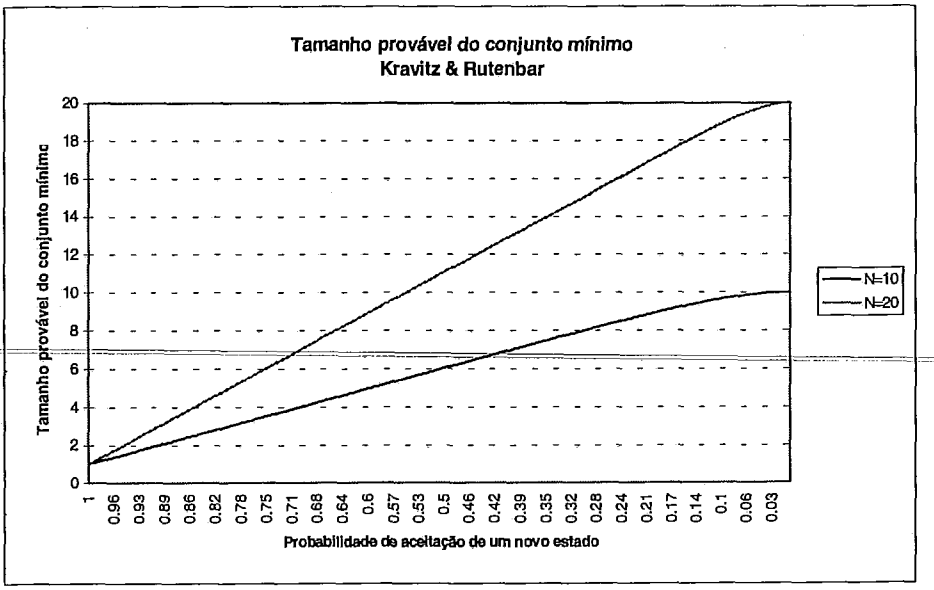


Figura 3.2 - Tamanho provável do conjunto mínimo.

A partir da observação da figura acima percebe-se que, em altas temperaturas, onde a maioria dos movimentos é aceita, não é efetivo alocar-se muitos processadores ao cálculo dos candidatos a movimento. A estratégia do conjunto mínimo parece ser mais adequada ao regime de baixas temperaturas. Isso dá margem a um esquema adaptativo com a utilização de algum outro algoritmo mais apropriado ao regime de altas temperaturas e a utilização do conjunto mínimo em temperaturas mais baixas, onde é menor a probabilidade de aceitação. Na hipótese de que o algoritmo apropriado a temperaturas altas exista, que o ponto de troca de estratégias ocorra em $a(T) = \infty$, que o custo de comunicação seja nulo e que a probabilidade de aceitação decaia linearmente com a temperatura, as integrais das curvas na Figura 3.2 entre $a(T) = \infty$ e $a(T) = 0$ forneceriam o *speedup* máximo teórico para a estratégia do conjunto mínimo. A integral da expressão em (3.2) fornece então:

$$speedup(\alpha) = \int_{a=\alpha}^{0} E[S]da \tag{3.3}$$

$$speedup(\alpha) = \left[-\frac{N}{N+1}(1-a)^{N+1} + \sum_{m=1}^{N}\binom{N}{m}(N-m+1)\int a^m(1-a)^{N-m}da \right]_{a=\alpha}^{0} \tag{3.4}$$

A integral do binômio em (3.4) pode ser resolvida recursivamente, obtendo-se:

$$\int a^m(1-a)^{N-m}da = \frac{1}{m+1}a^{m+1}(1-a)^{N-m} + \frac{N-m}{m+1}\int a^{m+1}(1-a)^{N-m-1}da \tag{3.5}$$

Nas equações 3.3, 3.4 e 3.5 a dependência de $a(T)$ com a temperatura foi tornada implícita, e nos referimos à probabilidade de aceitação como a. A Figura 3.3 mostra o *speedup* máximo teórico para a estratégia do conjunto mínimo para N = 10 e N = 20.

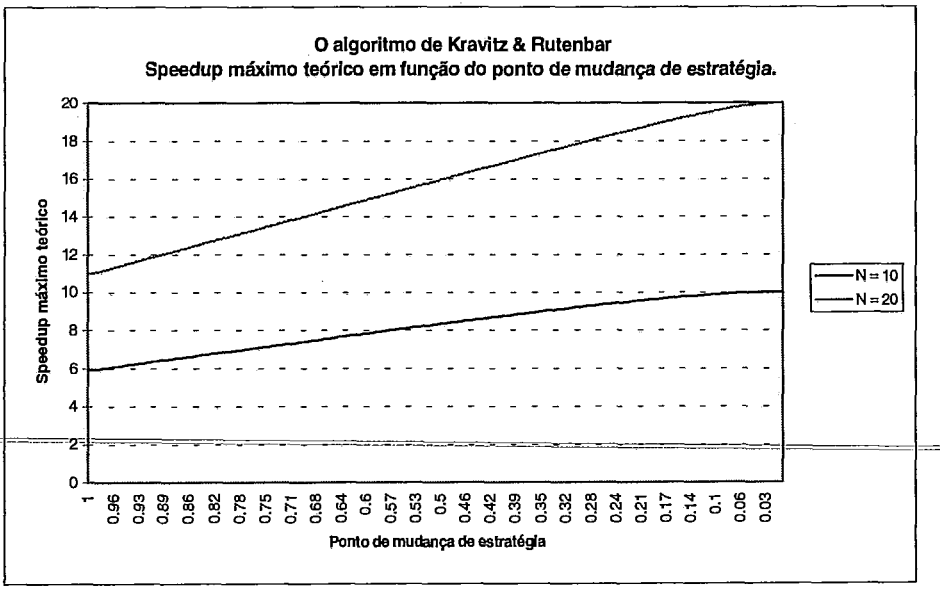


Figura 3.3 - O algoritmo do conjunto mínimo. Speedup máximo teórico em função do ponto de mudança de estratégia.

Observe-se o bom desempenho prometido pelo algoritmo. Ainda que a troca de estratégias não fosse utilizada e o algoritmo fosse empregado mesmo em altas

temperaturas, seriam obtidos *speedups* da ordem de 6 (10 processadores) ou 11 (20 processadores).

A seção seguinte descreve a implementação do algoritmo do conjunto mínimo.

3.2.1) Implementação:

- Pseudo código:

a) Mestre:

```
/* NRejects      ⇒      Contador do número de estados rejeitados    */

Repete até resfriamento      {
      Se existe mensagem a ser recebida      {
            Caso mensagem de estados rejeitados      {
                  Atualiza NRejects;
                  Se estabilizou na temperatura  {
                        Muda a senha de estados válidos;
                        T = ∝ T;
                        Broadcast da nova temperatura e senha;
                        NRejects = 0;
                  }
            }
            Caso escravo aceitou um estado      {
                  Se a senha é válida      {
                        Muda a senha de estados válidos;
                        Envia acknowledge para o processo que enviou a mensagem;
                        Espera bloqueante pela nova configuração;
                  }
            }
      }
      Gera nova configuração;
      Calcula o custo;
      Se aceita a nova configuração
            Muda a senha de estados válidos;
            Broadcast nova configuração e senha para os escravos;
      Senão      {
            Atualiza NRejects;
            Se estabilizou na temperatura      {
                  Muda a senha de estados válidos;
                  T = ∝ T;
                  Broadcast da nova temperatura e senha;
                  NRejects = 0;
            }
      }
}
```

a) Escravo:

```
/* NRejects      ⇒      Contador do número de estados rejeitados          */
/* KREJECTS     ⇒      Número máximo de estados rejeitados acumulados    */
                       antes de enviar mensagem ao mestre.               */
```

```
Repete até resfriamento      {
    Se existe mensagem a ser recebida                {
        Caso mensagem de nova temperatura        {
            Recebe a nova temperatura;
            Recebe a nova senha de estados válidos;
            NRejects = 0;
        }
        Caso mensagem de nova configuração       {
            Recebe a nova configuração;
            Recebe a nova senha de estados válidos;
        }
    }
    Gera nova configuração;
    Calcula o custo;
    Se aceita a nova configuração
        Pede validação da configuração ao mestre;
        Entra em espera bloqueante por nova mensagem;
        Caso mensagem de acknowledge vinda do mestre          {
            Broadcast da configuração aceita para os demais processadores;
        }
        Caso mensagem de nova temperatura        {
            Recebe a nova temperatura;
            Recebe a nova senha de estados válidos;
            NRejects = 0;
        }
        Caso mensagem de nova configuração       {
            Recebe a nova configuração;
            Recebe a nova senha de estados válidos;
        }
    Senão       {
        Atualiza NRejects;
        Se NRejects > KREJECTS          {
            Envia msg. de atualização de estados rejeitados para o mestre;
            NRejects = 0;
        }
    }
}
```

- Descrição do algoritmo e Resultados:

Foi introduzida uma relação de mestre-escravo entre os processadores. Um processador mestre cria os processos escravos e participa também da parte serial do processamento. Para atingir o equilíbrio térmico um grande número de movimentos deve ser executado em cada temperatura. Para aumentar a velocidade de processamento os estados são avaliados assincronamente. Os processadores continuamente geram novos movimentos, calculam o custo da configuração resultante e decidem localmente sobre a aceitação ou não desta configuração. Se um escravo detectar uma configuração aceitável ele pede a validação desta configuração ao processador mestre. Se o mestre decidir que aquela configuração gera um conjunto mínimo ele inicia uma atualização global da base de dados a fim de que todos os processadores recebam a nova configuração. Observe que a nova configuração é obtida incrementalmente pelos demais processadores. O

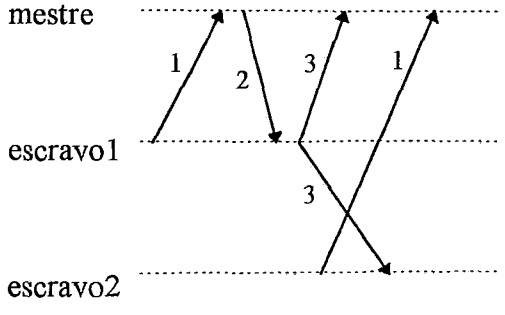
processador que gerou a configuração aceita envia para os demais apenas o movimento que gerou esta configuração. Os estados rejeitados em cada processador contribuem para a contagem de movimentos necessários a se atingir o equilíbrio numa dada temperatura. Para diminuir a comunicação entre os processadores, os escravos acumulam um certo número de estados rejeitados antes de enviar a contagem ao mestre. Assim, o mestre é responsável por atualizar a temperatura, manter a integridade da base de dados e monitorar o critério de parada do algoritmo. Idealmente, a cada instante, existe apenas uma configuração válida e essa é conhecida por todos os processadores. Observe que a maior parte das mensagens trocadas se deve à aceitação de novos estados e, quando este evento é raro, o custo de comunicação é reduzido. Por esse motivo, esse algoritmo parece ser mais adequado ao regime de temperaturas baixas. Esta estratégia geral foi melhorada com a utilização de alguns conceitos novos descritos a seguir.

Validação - Devido aos atrasos impostos pela rede, alguns dos processadores podem, momentaneamente, estar trabalhando sobre configurações desatualizadas. O mestre deve ser capaz de distinguir um pedido de validação gerado a partir de uma modificação na configuração corrente de um outro proveniente de uma configuração desatualizada. Para resolver-se este problema adotou-se um mecanismo de senhas. O mestre atribui uma senha para a próxima configuração válida. O primeiro estado que chegar ao mestre com a senha válida é aceito, os demais são ignorados. Quando o mestre faz a atualização global da base de dados ele envia a senha válida para o próximo estado. Um processador que tenha, nesse meio tempo, pedido validação de configuração ao mestre, percebe que o seu estado não foi aceito pela chegada de uma mensagem de nova configuração.

Recuperação de um movimento - A fim de reduzir a perda de boas configurações em temperaturas mais altas foi introduzido um mecanismo de recuperação de um movimento. Um escravo que tenha aceito uma configuração descobre se esta configuração foi usada ou não para a atualização global. Se ela não foi usada, ele tenta o mesmo movimento novamente. Os testes mostraram que esta estratégia leva a um comportamento ligeiramente melhor da convergência do algoritmo.

Existem, basicamente, quatro tipos de mensagens circulando na máquina virtual: a) Pedido de validação de um estado (escravo → mestre) e a provável aceitação deste estado (mestre → escravo); b) Mensagem de atualização do contador de estados rejeitados (escravo → mestre); c) Envio de nova configuração (qualquer processador para todos os demais); d) Mensagem de atualização de temperatura (mestre → escravos). Essas situações são vistas nos diagramas a seguir:

a) Escravo aceita um estado (pedido de validação ao mestre):



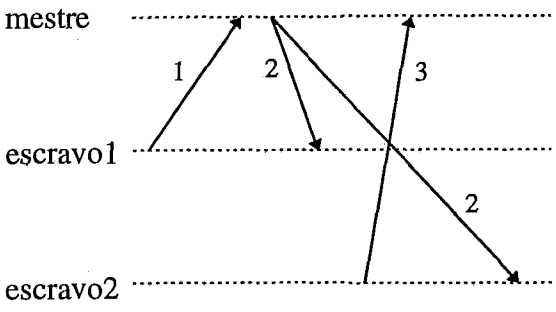
1) Escravo pede validação

2) Acknowledge

3) Broadcast da nova configuração

Observe que o escravo2 percebe que o seu pedido de validação não foi considerado através da chegada de uma mensagem de nova configuração.

b) Mensagem de atualização do contador de estados rejeitados:



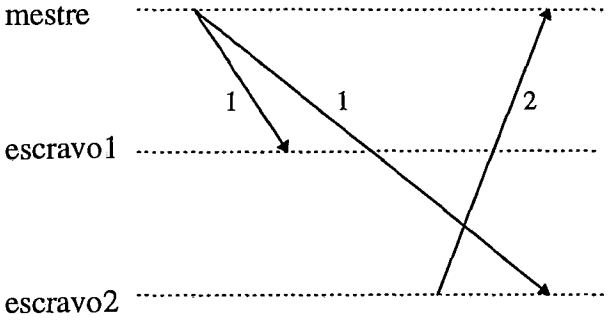
1) atualização de estados rejeitados

2) nova temperatura

3) pedido de validação de estado

Observe que o escravo2 sabe que o seu pedido de validação não foi considerado pela chegada de uma mensagem de atualização de temperatura.

c) Mestre aceita nova configuração



1) broadcast de nova configuração

2) pedido de validação

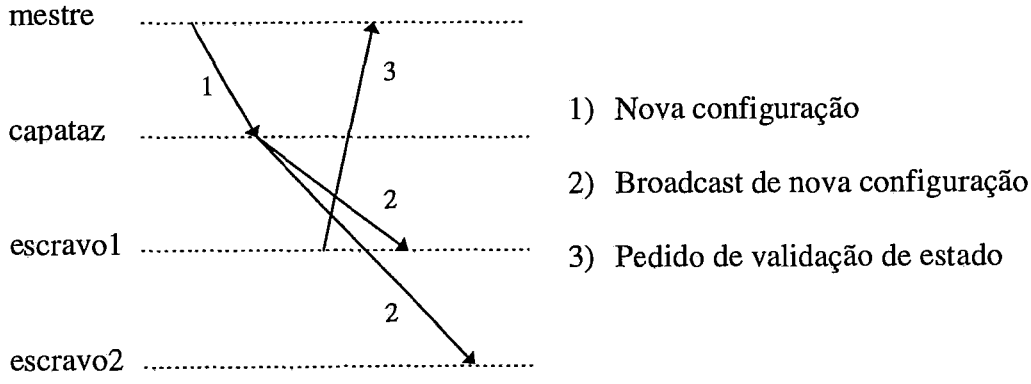
Observe que o escravo2 sabe que o seu pedido de validação não foi considerado pela chegada de uma mensagem de nova configuração.

O algoritmo do conjunto mínimo foi utilizado a partir de uma probabilidade de aceitação $a(T) < 20\%$ (O algoritmo usado para temperaturas altas será visto posteriormente). O circuito de teste é composto por 80 módulos e 30 redes equipotenciais. Através de resultados experimentais verificou-se uma efetiva contribuição dos escravos no cálculo das cadeias. Cerca de 60% ou 70% dos estados intermediários são calculados pelos escravos. Observou-se, no entanto, que a qualidade do *placement* obtido é sensivelmente pior do que aquele gerado pela versão serial. Além disso, o tempo de processamento paralelo mostrou-se muito maior do que o obtido na versão serial. As razões para a baixa qualidade da solução gerada serão investigadas posteriormente. O tempo excessivo de processamento deve-se, provavelmente, às seguintes razões:

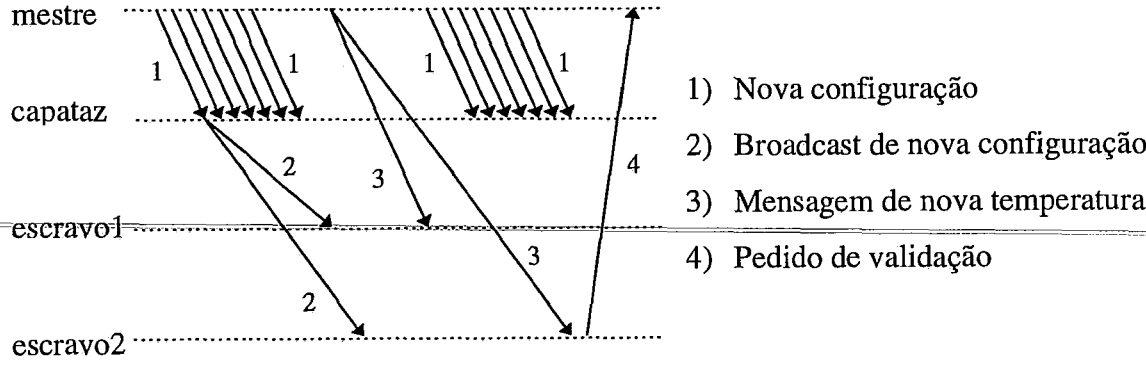
- O circuito de teste é muito pequeno. O tempo gasto para propor e avaliar uma nova configuração é, provavelmente, muito menor do que os tempos de troca de mensagens entre os processadores.

- A temperatura e, conseqüentemente, a probabilidade de aceitação de um novo estado ainda eram, provavelmente, muito altas quando da troca de estratégias resultando em um número excessivo de estados aceitos e em uma troca intensa de mensagens entre os processadores.

No entanto, buscando identificar as fontes de erro devido ao fato dos processos serem assíncronos, o ponto de troca de estratégia foi mantido e buscou-se reduzir o tempo de processamento modificando-se o esquema de comunicação entre os processadores. Primeiramente, observamos que 63% do tempo total de processamento era gasto com a mensagem de *broadcasting* de nova configuração do mestre para os escravos. Este mau desempenho talvez não deva ser creditado ao PVM, mas sim à configuração usada (*cluster* de estações de trabalho heterogêneas ligadas por um barramento Ethernet). De modo a não bloquear o mestre durante a operação de *broadcasting* pensamos então em criar um processo intermediário, que chamaremos aqui de capataz. O capataz recebe a mensagem de nova configuração do mestre (uma troca de mensagens rápida em PVM) e faz o *broadcasting* da nova configuração aos escravos. Esperava-se com isto que o nível de contribuição dos escravos caísse mas que, ao menos, eles não atrasassem o mestre na tarefa de computar uma cadeia. O novo esquema é visto na figura a seguir.



Um problema que logo se evidenciou com essa solução foi o acúmulo de mensagens no capataz. A figura a seguir ilustra este fato.



Observe-se que as mensagens de nova configuração se acumulam no capataz para serem enviadas aos escravos. Antes que todas sejam recebidas, o mestre completa o cálculo de uma temperatura e envia a nova configuração, temperatura e senha em *broadcast* para os escravos. Todas as mensagens acumuladas no capataz devem agora ser recebidas e descartadas pelos escravos, uma vez que elas se referem a uma configuração já

ultrapassada. Observou-se então que os escravos ficavam tão ocupados recebendo e descartando mensagens que o nível de contribuição ao cálculo das cadeias caiu a níveis próximos de zero. Isto é, as cadeias eram quase que inteiramente calculadas pelo processo mestre. (Ironicamente, nesta situação, a qualidade da solução gerada aproxima-se daquela obtida com a versão serial). Uma possível solução para o problema, que não chegou a ser implementada, é mostrada na figura a seguir.

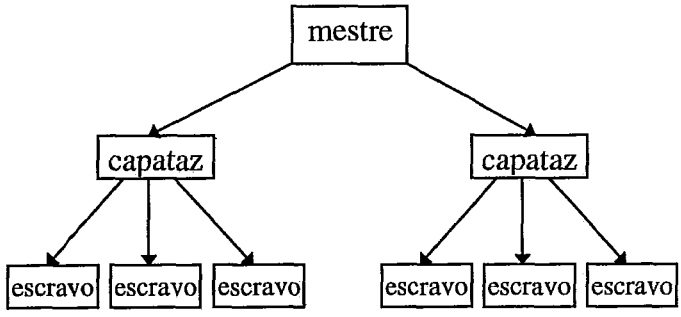
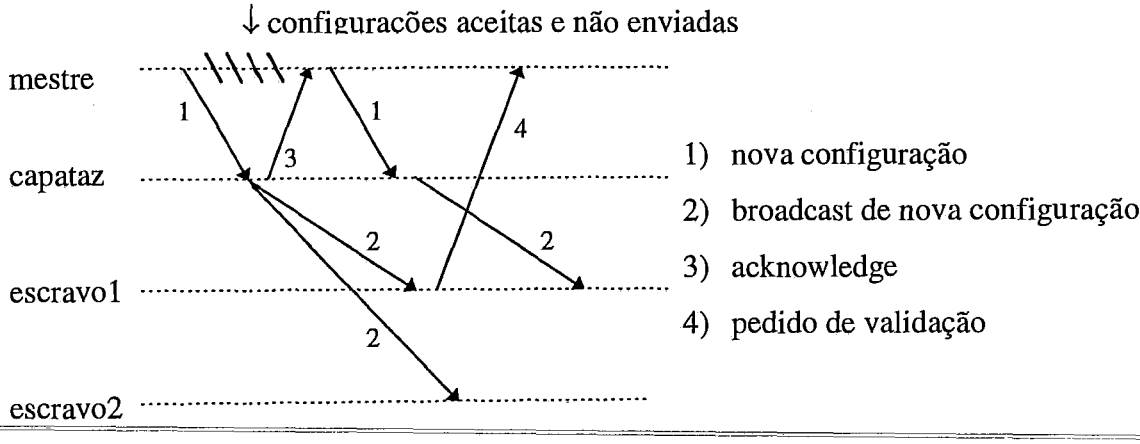


Figura 3.4 - Uma possível solução para o acúmulo de mensagens no processo capataz.

Aqui, o acúmulo de mensagens em cada um dos capatazes tende a diminuir, uma vez que é menor o número de escravos ligados a cada um deles. Além disso, os processos capatazes poderiam também participar do processamento ao invés de, simplesmente, rotear mensagens para os escravos.

Uma outra facilidade implementada é a de prover um nível de *acknowledge* entre o mestre e o capataz. O objetivo é o de não sobrecarregar o capataz e, em conseqüência, os escravos com mensagens que serão descartadas. Um dos problemas desta solução é o de que, uma vez que nem todos os movimentos aceitos chegam aos escravos, a base de dados não mais pode ser obtida incrementalmente. É necessário agora transmitir toda a configuração ao invés de apenas o movimento aceito.



Efetivamente, o nível de contribuição dos escravos para o cálculo da cadeia aumentou. No entanto, a convergência do algoritmo piorou. Pode-se compreender o motivo desta degradação pela figura acima. Observe que quando o escravo1 recebe a nova configuração, vinda do capataz, o mestre já aceitou muitas outras configurações. O escravo1 aceita um movimento baseado na configuração desatualizada e pede validação ao mestre. Este movimento é obviamente rejeitado, o que o escravo percebe pela chegada de outra mensagem de nova configuração. Percebe-se daí que os escravos conseguem contribuir apenas para a contagem de movimentos rejeitados. Além disso os

movimentos rejeitados pelos escravos são, em sua maioria, obtidos sobre configurações já desatualizadas.

3.2.2) Conclusões:

Em termos de velocidade de processamento, os resultados obtidos por nossa implementação do algoritmo de Kravitz e Rutenbar [Krav87] foram decepcionantes. É claro que, dada a natureza da plataforma utilizada (caracterizada por um alto custo de troca de mensagens), o algoritmo parece mais apropriado a circuitos várias ordens de grandeza maiores do que o utilizado e a temperaturas mais baixas. Ainda assim, o algoritmo é de difícil implementação e alguns problemas foram identificados devido à natureza completamente assíncrona dos processos. Mas, o aspecto mais importante, foi o fato da qualidade da solução paralela cair à medida em que aumentava a contribuição dos processos escravos ao cálculo da cadeia. Para explicar a interpretação a que se chegou a respeito deste fato, a Figura 3.1 (referenciada aqui como Figura 3.5) é reproduzida abaixo:

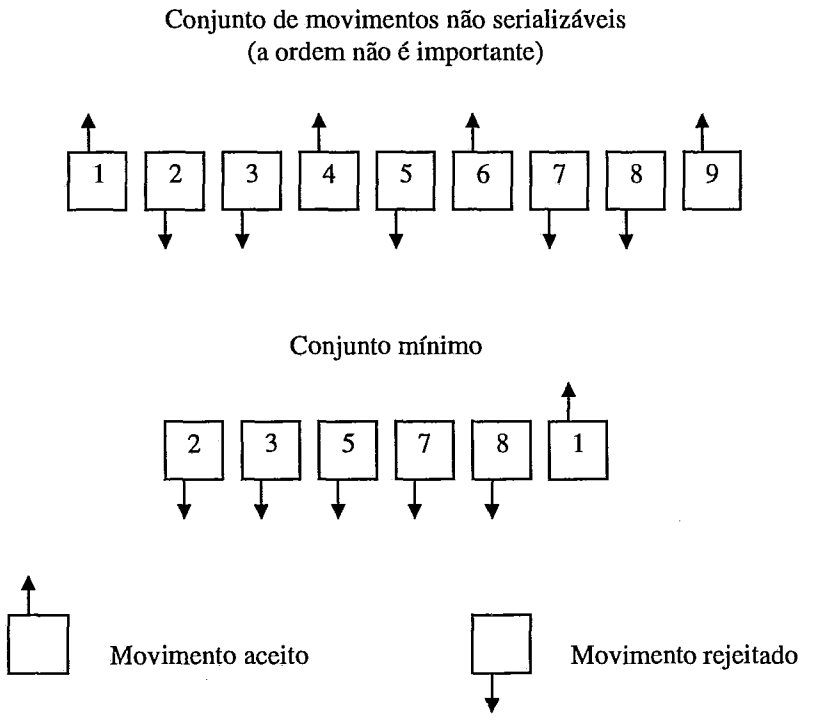


Figura 3.5 - A formação do conjunto mínimo.

Suponha que os movimentos [1..9] sejam escalados para serem processados pela versão serial do algoritmo, isto é, o movimento 2 é processado se o movimento 1 for recusado, o movimento 3 é processado se os movimentos 1 e 2 forem recusados e assim por diante. Uma vez que, no exemplo apresentado, o movimento 1 foi aceito, nenhum dos movimentos seguintes seria efetivado e o algoritmo prosseguiria a partir da nova configuração gerada. Na formação do conjunto mínimo, a inclusão dos movimentos 2, 3, 5, 7 e 8 têm o efeito apenas de provocar o encurtamento da cadeia de Markov uma vez que, obviamente, movimentos rejeitados não contribuem para a qualidade da solução gerada. Explica-se assim o motivo pelo qual a qualidade da solução aumentava quando era menor o nível de contribuição dos processadores escravos.

Conclui-se portanto que o algoritmo de Kravitz e Rutenbar, apesar de fartamente referenciado em literatura, incorre em grave erro conceptual fazendo com que o *speedup* seja obtido às custas da qualidade da solução gerada. O algoritmo apresentado na seção seguinte, baseado em trabalho de Andrew Sohn [Sohn95] buscará resolver este problema.

3.3) O Algoritmo Especulativo: [Sohn95]

O algoritmo de Sohn, ainda que não tenha sido idealizado em contraposição ao algoritmo de Kravitz e Rutenbar resolve os problemas deste. O algoritmo trabalha também com o conceito de conjunto serializável. No entanto, este é tomado em sentido estrito: o algoritmo paralelo reproduz exatamente a seqüência de movimentos (aprovados ou rejeitados) que seriam executados pela versão serial. O funcionamento do algoritmo pode ser melhor compreendido pela Figura 3.6.

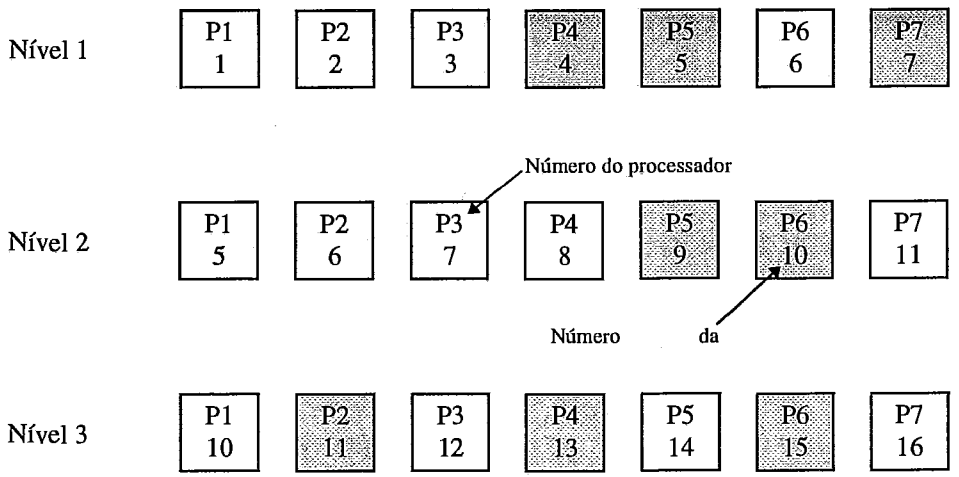


Figura 3.6 - O algoritmo especulativo. Os retângulos sombreados correspondem a estados aceitos.

A fim de assegurar que a versão paralela gere a **mesma** seqüência de decisões que a correspondente serial, os processadores usam o mesmo *seed* para gerar a seqüência de números pseudo randômicos. A Figura 3.6 mostra o algoritmo especulativo executando 11 (e não 16) iterações em 3 passos. O primeiro nível mostra $N = 7$ processadores executando 7 iterações simultaneamente. Suponha que os processadores 4, 5 e 7 resolvam pela aceitação de seus movimentos (os retângulos sombreados indicam estados aceitos). Entre os três, a decisão tomada pelo processador de índice mais baixo, processador 4, é aceita já que as decisões tomadas pelos processadores de índice mais alto são equivocadas. Ao propor um movimento, cada processador baseia-se na hipótese, ou especulação, de que todos os movimentos anteriores, no mesmo nível, serão rejeitados. A fim de seguir a mesma trajetória do algoritmo serial, o movimento gerado pelo processador 5 teria de ser aplicado à base de dados modificada pelo processador 4. O processador 5 não pode decidir pela aceitação ou não do estado proposto baseado na configuração corrente. Pelo mesmo motivo a decisão tomada pelo processador 7 é também equivocada.

O segundo nível é formado pelos sete processadores calculando as iterações de números 5 a 11. Cada processador recebe o índice do movimento efetivado no nível anterior, baseado no qual ele atualiza a sua cópia da base de dados. Em seguida, cada

processador calcula o índice de sua próxima iteração. Este índice pode ser facilmente obtido adicionando-se o índice do movimento efetivado no nível anterior à identificação do processador. Por exemplo, o processador 2 adiciona o seu número de identificação, 2, ao índice do movimento efetivado no nível anterior, 4, resultando no índice 6 para o próximo nível. Os outros processadores calculam de forma similar o índice de sua iteração. Observe que, a cada nível, todos os processadores geram toda a seqüência de números randômicos necessários à caracterização dos N movimentos propostos e, por este motivo, é necessário que eles conheçam apenas o índice do movimento efetivado a fim de reconstruir a base de dados.

Os processadores são organizados segundo o paradigma mestre-escravo. No exemplo dado, seja P_1 o processador mestre e $P_2..P_7$ os processadores escravos. O funcionamento do algoritmo é descrito a seguir.

1. No início do processamento o processador mestre envia o *seed* aos processadores escravos e o índice base 0 a fim de que eles determinem o índice de suas iterações.
2. Cada processador determina o índice de sua iteração somando o índice base ao número de identificação do processador e decide pela aceitação ou não do movimento proposto. Cada processador escravo envia então ao mestre o resultado de sua decisão e o seu número de identificação.
3. O processador mestre coleta os resultados das decisões dos escravos e efetiva, dentre os movimentos aceitos, aquele executado pelo processador de menor índice.
4. O processador mestre envia o índice do movimento efetivado para todos os outros processadores.
5. Todos os processadores atualizam a base de dados e prosseguem a partir do passo 2, sem interrupções.

O melhor caso ocorre se, em um dado nível, nenhum movimento é aceito ou se o processador de índice mais alto é o único a aceitar um movimento. Neste caso, o algoritmo especulativo pode calcular N iterações em um único nível, onde N é o número de processadores na arquitetura paralela. O pior caso ocorre quando o processador de índice mais baixo, P_0, decide pela aceitação de seu movimento. Este caso, se repetido em todos os níveis, seria equivalente à implementação serial do algoritmo em P_0 (em verdade, devido aos custos de comunicação entre os processos, é de se supor que o desempenho de tal versão paralela seria bem pior do que o equivalente serial).

O desempenho do algoritmo pode ser melhor entendido a partir do seguinte modelo matemático: Seja N o número de processadores e M o tamanho do conjunto serializável S em cada nível. Na temperatura T, a probabilidade de aceitação de um movimento individual é $a(T)$. A probabilidade $Pr(M = m)$ de que o conjunto S tenha tamanho m, é dada pela probabilidade de que os $(m\text{-}1)$ primeiros processadores rejeitem seus movimentos e que o processador de índice m o aceite. Assim,

$$Pr(M = m) = a(T)*[1-a(T)]^{m-1} \tag{3.6}$$

O tamanho esperado para o conjunto serializável S é então dado por:

$$E[S] = [1-a(T)]^{N} * N + \sum_{m=1}^{N} a(T)*[1-a(T)]^{m-1} * m \tag{3.7}$$

O gráfico da Figura 3.7 mostra o tamanho provável do conjunto serializável como função da probabilidade de aceitação $a(T)$ para N = 10 e N = 20. Observe-se que, na hipótese do custo de comunicação entre os processos ser nulo, a função $E[S]$ representa o *speedup* máximo teórico pontual, para cada valor de $a(T)$.

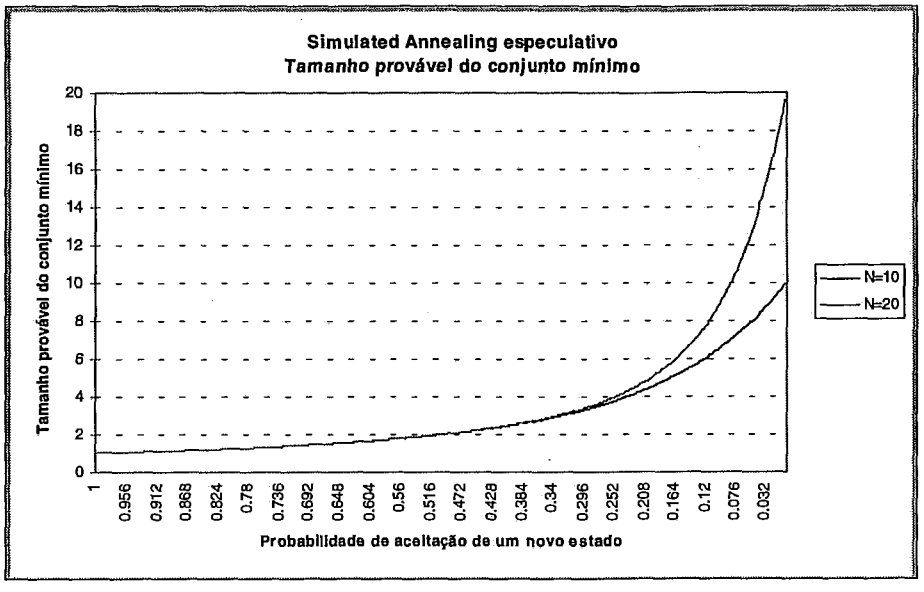


Figura 3.7 - Tamanho provável do conjunto serializável em função da probabilidade de aceitação de um novo estado.

Dois fatos chamam a atenção a partir da observação da Figura 3.7:

1. A baixa eficiência do algoritmo em altas temperaturas. De fato, o tamanho do conjunto serializável é menor do que 2 para $a(T) > 0{,}5$.
2. O reduzido benefício de alocar-se muitos processadores à arquitetura paralela em temperaturas altas. Observe que as duas curvas têm valores idênticos até, aproximadamente, $a(T) = 0{,}3$.

Vamos comparar aqui o conjunto serializável obtido pelo algoritmo especulativo de Sohn com aquele defendido por Kravitz e Rutenbar. As curvas para $N = 20$ podem ser vistas na Figura 3.8. Observe a diferença acentuada entre as duas curvas, resultado da inclusão equivocada de todos os movimentos rejeitados no conjunto serializável. Entende-se assim os resultados pobres obtidos por aquele algoritmo em comparação com a versão serial.

Aqui é também possível obter o *speedup* máximo teórico em função do ponto de mudança de estratégia $\propto$. Para isto vamos mais uma vez supor que exista um algoritmo apropriado a altas temperaturas, que o custo de comunicação seja nulo e que a probabilidade de aceitação decaia linearmente com a temperatura. Neste caso, a integral da expressão em (3.7) entre $a(T) = \propto$ e $a(T) = 0$ fornece o *speedup* máximo teórico para o algoritmo especulativo:

$$speedup(\alpha) = \int_{a=\alpha}^{0} E[S]da \qquad (3.8)$$

$$speedup(\alpha) = \left\{ -\frac{N}{N+1}(1-a)^{N+1} + \sum_{m=1}^{N}\left[\frac{m}{m+1}(1-a)^{m+1} - (1-a)^{m} \right] \right\}_{a=\alpha}^{0} \qquad (3.9)$$

A Figura 3.9 mostra o *speedup* máximo teórico para $N = 10$ e $N = 20$.

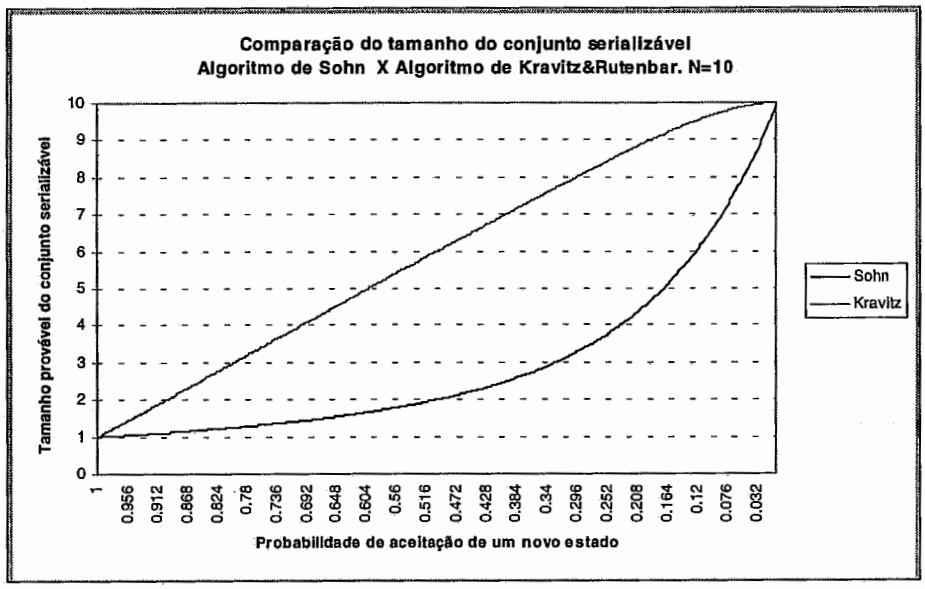


Figura 3.8 - Comparação dos algoritmos de Sohn e de Kravitz-Rutenbar.

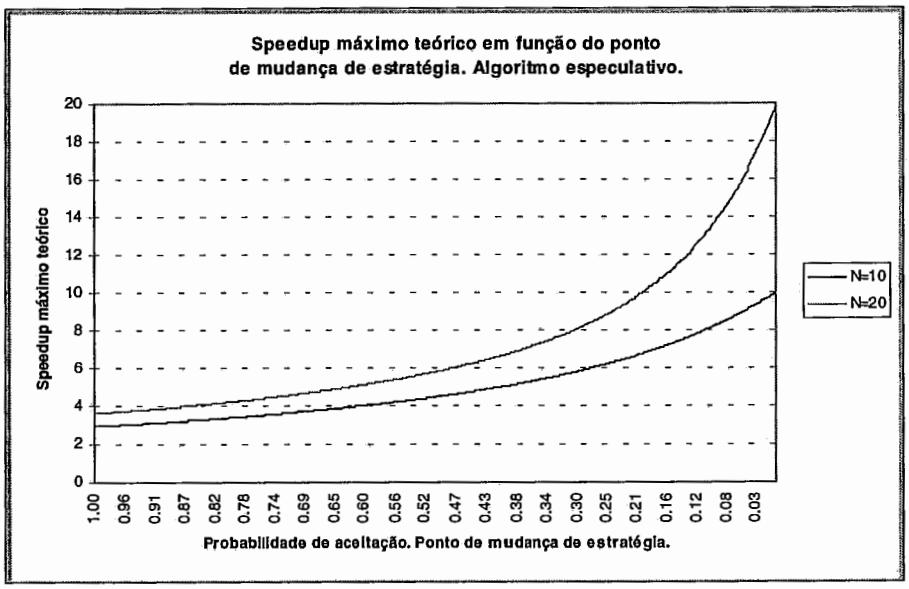


Figura 3.9 - Speedup máximo teórico em função do ponto de mudança de estratégia. Algoritmo Especulativo.

Aqui também é possível observar o reduzido benefício de alocar-se muitos processadores à máquina paralela e o fraco desempenho do algoritmo em temperaturas altas (O *speedup* máximo teórico para $N = 20$ processadores e $\propto = 1.0$ é de apenas 4 unidades e isso, considerando-se a hipótese pouco provável de custo 0 de comunicação).

3.3.1) Conclusões:

O algoritmo especulativo de Sohn tem o grande mérito de estabelecer um conjunto serializável coerente. No entanto, a restrição de que, a partir do mesmo *seed*, o algoritmo paralelo gere exatamente a mesma seqüência de movimentos que a versão serial é exageradamente forte. A característica marcante do algoritmo é manter a mesma proporção entre os estados aceitos/rejeitados verificada no algoritmo serial. Este é o ponto onde falha o algoritmo de Kravitz-Rutenbar e é a característica que deveria ser buscada em um algoritmo alternativo. A exigência de que o algoritmo paralelo reproduza a trajetória do serial obriga a que todos os processadores gerem toda a seqüência de números randômicos necessária à construção de todos os movimentos em um dado nível. Mesmo a suposta facilidade daí advinda - a possibilidade de atualização da base de dados a partir do recebimento de um único índice - é relativa, uma vez que as mudanças na configuração corrente são incrementais e o processador "vencedor" tem que transmitir apenas o movimento efetivado em cada nível.

Além disso, pela observação da Figura 3.7, percebe-se o comportamento dinâmico do algoritmo. Em temperaturas mais altas, onde é grande a probabilidade de aceitação de um novo estado, é absolutamente ineficaz alocar-se um número grande de processadores ao cálculo. Isto, no entanto, torna-se vantajoso próximo ao resfriamento. Isto sugere uma estratégia adaptativa onde o número de processadores e, talvez, outros parâmetros do algoritmo pudessem variar ao longo do tempo. Este e outros tópicos serão vistos na seção seguinte.

3.4) O Algoritmo Adaptativo:

A primeira modificação introduzida no algoritmo especulativo é a relaxação da exigência de que os processadores executem a mesma seqüência de movimentos da versão serial. Cada processador gera uma seqüência independente de números pseudo randômicos. O processador "vencedor" transmite aos demais apenas o movimento efetivado a cada nível.

A segunda modificação diz respeito à verificação da estabilidade em uma dada temperatura e o conseqüente resfriamento da temperatura. No algoritmo especulativo o processador mestre mantém um contador com o somatório dos tamanhos dos conjuntos serializáveis em cada nível. Ao verificar a efetivação de um número suficiente de estados, o processador mestre reduz a temperatura e envia uma mensagem em *broadcast* para que os processadores escravos realizem o mesmo procedimento. A modificação introduzida consiste em determinar, probabilisticamente, o tamanho do conjunto serializável, a cada nível. Seja *npassos* o número de movimentos que devem ser testados em uma dada temperatura para garantir o equilíbrio. De (3.7) conhecemos $E[S]$, o tamanho provável do conjunto serializável S a cada nível. Assim, se pudermos determinar a probabilidade de aceitação $a(T)$, podemos calcular o número de níveis que devem ser processados a cada temperatura (η):

$$\eta = \frac{npassos}{E[S]} \tag{3.10}$$

O valor de η é calculado pelo processador mestre no início do processamento de cada temperatura e enviado aos escravos. Desta forma, todos os processadores conhecem à *priore* o número de níveis que serão calculados na temperatura. O valor de $a[T]$ é obtido

por extrapolação dos valores conhecidos, referentes às últimas temperaturas processadas. Complementando este procedimento, relaxou-se também a exigência de que o movimento efetivado seja o movimento aceito pelo processador de menor índice. O primeiro movimento aceito que chega ao mestre é efetivado, independente do processador onde ele foi gerado. A vantagem destes procedimentos combinados é a de que, após o recebimento pelo mestre da primeira mensagem comunicando a aceitação de um estado, todas as demais mensagens tornam-se desnecessárias e não precisam ser recebidas e desempacotadas pelo processador mestre.

Uma primeira versão do algoritmo, contemplando estas modificações foi implementada e usada para a otimização de um circuito contendo 100 módulos e 300 redes equipotenciais. Um outro algoritmo, mais apropriado ao regime de temperaturas altas, foi utilizado durante a fase inicial do processo e o ponto de troca de estratégias ocorreu para $a(T) = 0{,}2$. Foram utilizados $N = 10$ processadores na máquina virtual. Observa-se, a partir do gráfico na Figura 3.9, que o *speedup* máximo teórico nessas condições é da ordem de 6 unidades (se considerarmos o custo de comunicação nulo). A tabela a seguir mostra os resultados obtidos e a comparação com a versão serial. As medidas de tempo referem-se somente ao domínio do algoritmo especulativo, qual seja, $a(T) > 0{,}2$. Os valores referem-se à media de 10 medidas.

	Simulated Annealing	
	serial	adaptativo
Custo	1915	1927
t (s)	63,78	418,47

Tabela 1 - Comparação de desempenho: annealing serial versus algoritmo adaptativo.

A qualidade da solução obtida é compatível com a versão serial. Isto é um bom resultado e confirma o fato de que as modificações introduzidas corrigem os problemas encontrados no algoritmo de Kravitz-Rutenbar. No entanto, o péssimo resultado obtido para o tempo de execução, nos dá a certeza de que o custo de comunicação não é desprezível no ambiente de programação usado. Para o circuito de teste, o tempo de processamento gasto para propor e avaliar um estado é, provavelmente, muito menor do que o tempo gasto em troca de mensagens. Assim, para obtermos sucesso com alguma estratégia paralela, é necessário reduzir a quantidade de dados circulando pela rede ou, ao menos, definir uma complexidade mínima para os circuitos a serem otimizados, abaixo da qual o algoritmo paralelo não é efetivo.

Na seção seguinte estudaremos o *speedup* para o algoritmo adaptativo levando-se em consideração o custo de comunicação.

3.4.1) Análise do *speedup* em presença do custo de comunicação:

Seja t_s, o tempo gasto para processar uma temperatura no algoritmo *simulated annealing* serial. t_s pode ser expresso como:

$$t_s = npassos * t_1 \tag{3.11}$$

onde:

- $npassos$ $\Rightarrow$ número de movimentos que devem ser testados em uma dada temperatura para assegurar o equilíbrio;
- t_1 $\Rightarrow$ tempo gasto para propor e avaliar um movimento;

O tempo gasto pela versão paralela do algoritmo (t_p) pode ser expresso como:

$$t_p = \frac{npassos}{E[S]}\left(t_1 + t_2 + Pr(M \neq 0)t_1\right) \tag{3.12}$$

onde:

$E[S]$ $\Rightarrow$ Tamanho provável do conjunto S, em cada nível. Dado pela equação (3.7).

t_2 $\Rightarrow$ Tempo gasto para que todos os processadores enviem ao mestre o resultado de suas avaliações e recebam deste o movimento efetivado;

$Pr(M \neq 0)$ $\Rightarrow$ Probabilidade de que ao menos um movimento seja aceito pelos N processadores, $Pr(M \neq 0) = \left[1 - (1 - a(T))^N\right]$

O último termo refere-se ao fato de que, se algum movimento for efetivado, todos os processadores precisam atualizar sua cópia da base de dados. Assim, o *speedup* é dado por:

$$speedup = \frac{t_s}{t_p} = \frac{t_1 * E[S]}{t_1 + t_2 + Pr(M \neq 0) * t_1} \tag{3.13}$$

Se fizermos,

$$t_2 = K * t_1 \tag{3.14}$$

vem:

$$speedup = \frac{E[S]}{1 + K + Pr(M \neq 0)} \tag{3.15}$$

Se o custo de comunicação for nulo, o valor de K é igual a zero resultando em um *speedup* que varia de $E[S]/2$ em altas temperaturas até $E[S]$ em temperaturas baixas.

Os valores de t_1 e t_2 foram medidos para o circuito de teste (100 módulos, 300 redes) e $N = 10$ processadores. Os valores encontrados foram $t_1 \cong 400$ µs e $t_2 \cong 20$ ms o que resulta em $K \cong 50$. Observe que o valor de t_1 é função da complexidade do circuito enquanto que t_2, o tempo para enviar e receber os movimentos aceitos, é função da arquitetura utilizada, sendo, no caso específico deste trabalho, função do número de processadores presentes à máquina virtual. Assim, para circuitos maiores, o valor de K tende a diminuir melhorando o valor de *speedup*. A Figura 3.10 mostra o *speedup* pontual, em função da probabilidade de aceitação $a(T)$, para $K = 50$ e $a(T) < 0{,}2$. Na hipótese de que $a(T)$ decaia linearmente com a temperatura, o *speedup* total ($\overline{sp}$) pode ser calculado por integração numérica da curva, obtendo-se $\overline{sp} = 0{,}1309$. Observe que este valor é muito próximo do valor medido encontrado na Tabela 1 onde $\overline{sp} = 0{,}1524$.

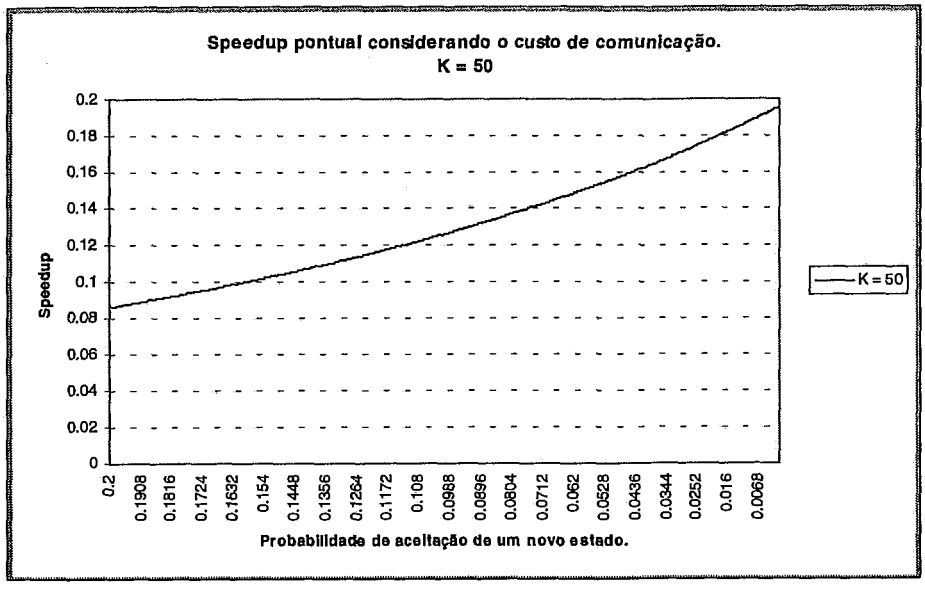


Figura 3.10 - Speedup pontual considerando o custo de comunicação. K =50.

3.4.2) As modificações propostas:

A primeira modificação visando acelerar o processamento do algoritmo foi inspirada pela constatação de que o tempo de comunicação é muito maior do que o tempo gasto para propor e avaliar um estado (ao menos para o circuito de teste até então usado).

Analisando novamente o gráfico na Figura 3.7, que, por comodidade, é repetido aqui na Figura 3.11, observa-se que, em baixas temperaturas, existe um benefício real em aumentar o número de estados avaliados entre sincronizações.

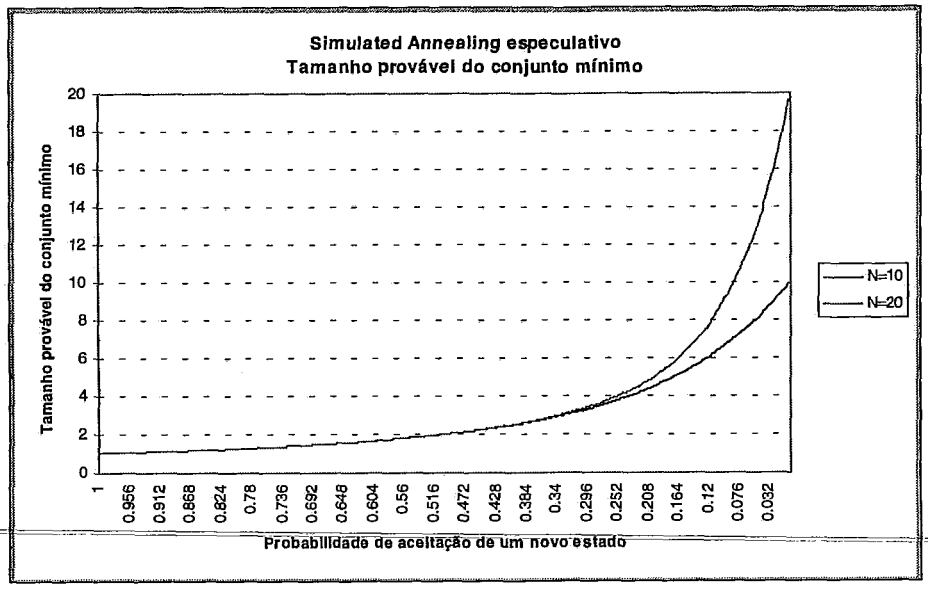


Figura 3.11 - Tamanho provável do conjunto serializável em função da probabilidade de aceitação de um novo estado.

A primeira alternativa para aumentar o número de estados avaliados é aumentar o número de processadores na máquina paralela. No entanto, este é um recurso limitado e o aumento do número de processadores tem o efeito de aumentar o custo de comunicação (t_2 na equação 3.13). Uma vez que o custo de propor e avaliar um estado é muito menor do que o tempo de comunicação, pode ser vantajoso aumentar o número

de estados avaliados através do cálculo, entre sincronizações, de mais de um estado por processador. Esta alternativa tem a vantagem de não alterar o valor de K uma vez que cada processador envia ao mestre um único estado aceito (se houver algum) a cada passo.

Seja n o número de estados avaliados por processador entre sincronizações. A equação (3.12) pode então ser reescrita como:

$$t_p = \frac{npassos}{E'[S]}\left(n*t_1 + t_2 + Pr'(M \neq 0)*t_1\right) \tag{3.16}$$

onde:

$Pr'(M \neq 0) \Rightarrow$ Probabilidade de que ao menos um movimento seja aceito pelos N processadores; $Pr'(M \neq 0) = \left[1-(1-a(T))^{n*N}\right]$, e

$$E'[S] = [1-a(T)]^{n*N} * n * N + \sum_{m=1}^{n*N} a(T)*[1-a(T)]^{m-1} * m \tag{3.17}$$

A equação para o *speedup* pode então ser expressa como:

$$speedup = \frac{t_s}{t_p} = \frac{t_1 * E'[S]}{n*t_1 + t_2 + Pr'(M \neq 0)*t_1} \tag{3.18}$$

ou:

$$speedup = \frac{E'[S]}{n + K + Pr'(M \neq 0)} \tag{3.19}$$

Observe que para valores muito pequenos de K, o aumento no numerador em (3.19) será menor do que o correspondente aumento no denominador. Assim, em tal circunstância, não deveria ser efetivo calcular mais de um movimento por processador, entre sincronizações. Por outro lado, se $K \rangle\rangle n$, o valor de *speedup* cresce linearmente com $E'[S]$. A Figura 3.12 mostra o *speedup* pontual, em função de $a(T)$, para alguns valores de n. Os gráficos referem-se ainda a $N = 10$ processadores, $K = 50$ e o ponto de mudança de estratégia em $a(T) < 0{,}2$.

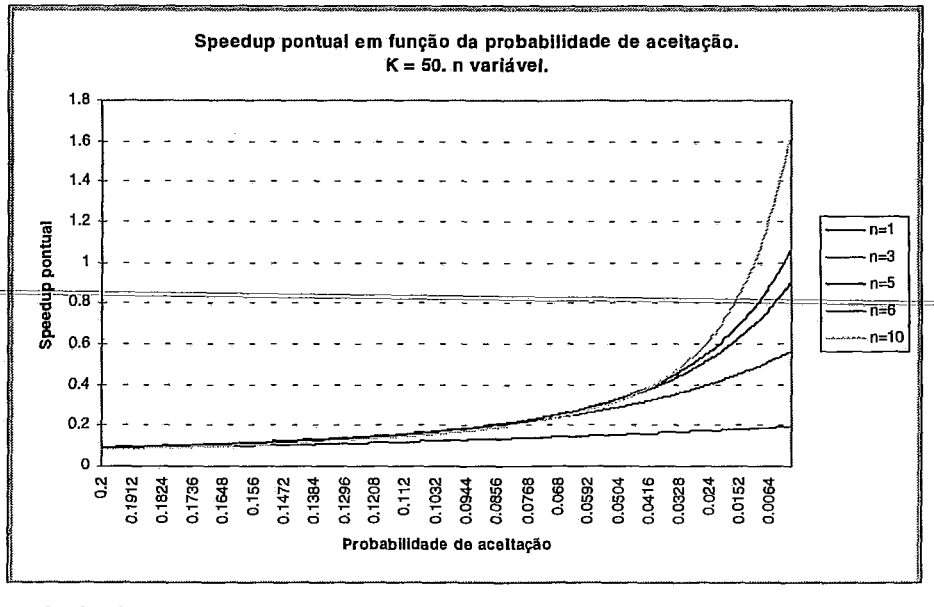


Figura 3.12 - Speedup pontual em função da probabilidade de aceitação de um novo estado. $K = 50$. n variável.

Mais uma vez, na hipótese de que a probabilidade de aceitação varie linearmente com a temperatura, as curvas em 3.12 podem ser integradas para obter o *speedup* médio no intervalo $0{,}2 < a(T) < 0$. O resultado é mostrado na Figura 3.13.

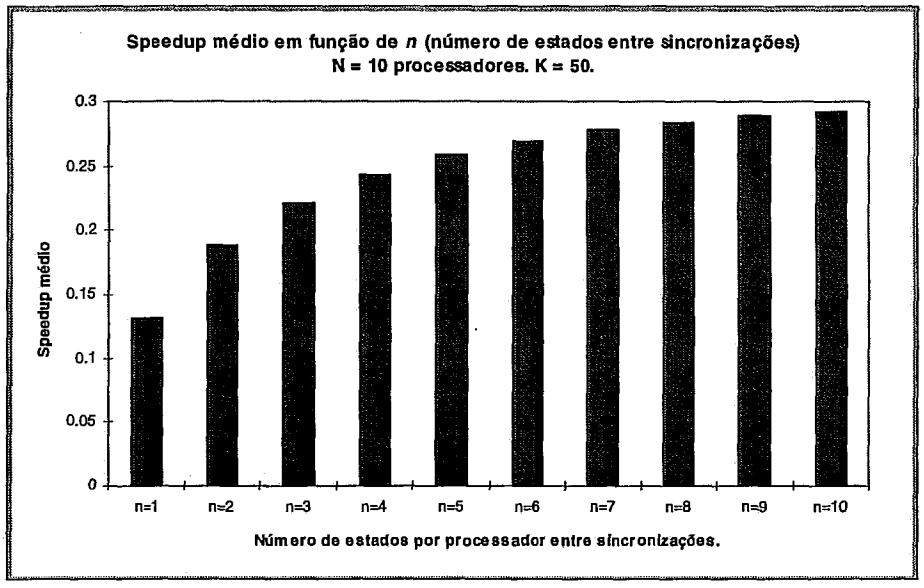


Figura 3.13 - Speedup médio em função de n.
$N = 10$ processadores. $K = 50$.

Os experimentos representados pelos gráficos nas Figuras 3.12 e 3.13 foram repetidos para um circuito maior composto por 1024 módulos e 3000 redes equipotenciais. O tempo para propor e avaliar um estado neste circuito foi medido, e vale $t_1 \cong 5{,}2$ ms resultando em $K = 3{,}8$. Os resultados podem ser vistos nas Figuras 3.14 e 3.15.

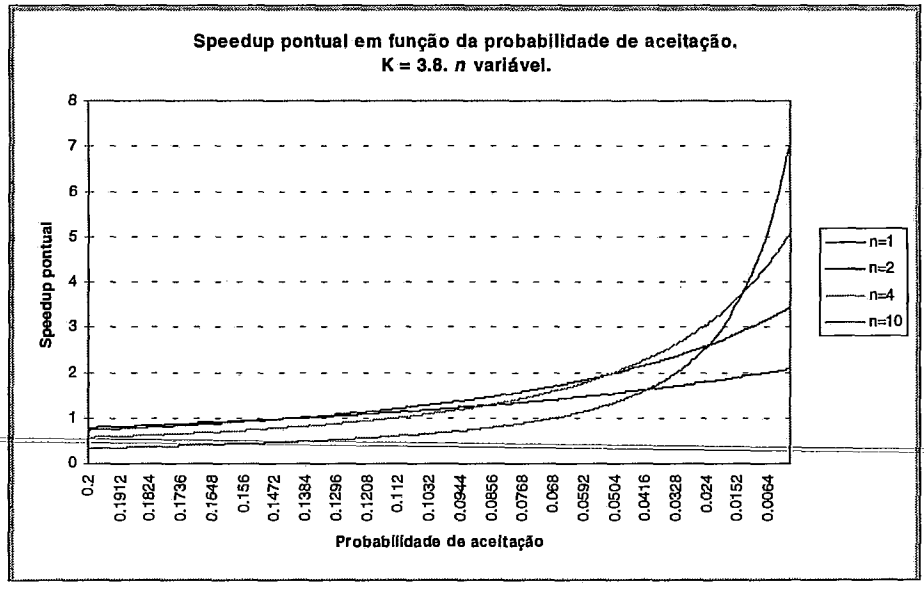


Figura 3.14 - Speedup pontual em função da probabilidade de aceitação de um novo estado.
$K = 3{,}8$. n variável.

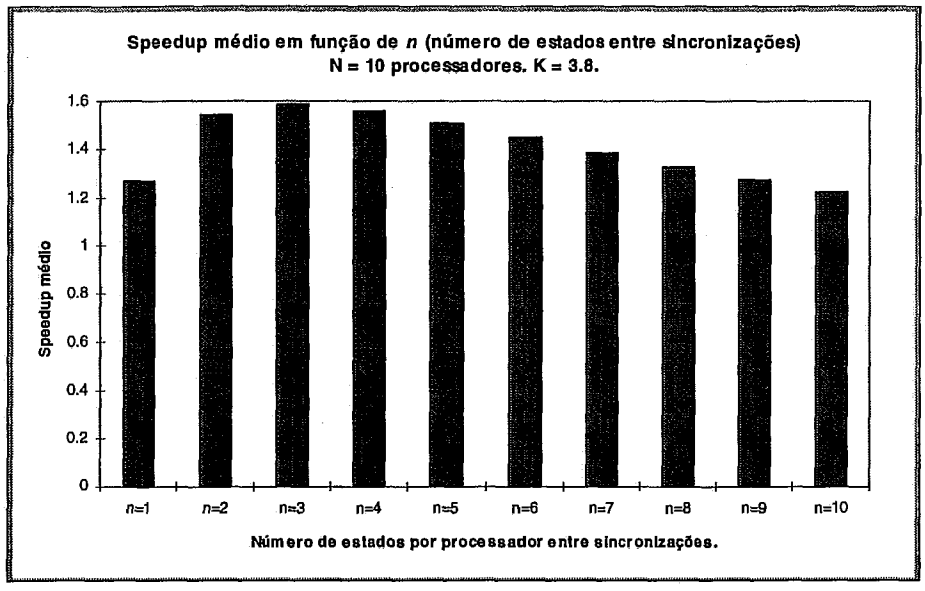


Figura 3.15 - Speedup médio em função de n.
$N = 10$ processadores. $K = 3.8$.

Pode-se tirar algumas conclusões importantes pela observação das Figuras 3.12 - 3.15:

- Se mantido constante durante todo o processamento, o valor ótimo de n é função de K.
- Para o mesmo valor de K, o valor ótimo de n é dinâmico, e varia com a probabilidade de aceitação $a(T)$.

A partir desta última observação pode-se pensar em uma estratégia adaptativa onde o valor de n varia com o decaimento da temperatura. Tomemos como exemplo o gráfico na Figura 3.14. Para cada valor de $a(T)$ tomamos o valor de n que maximiza o valor da função *speedup*. O valor ótimo de n em função de $a(T)$ pode ser visto na Figura 3.16.

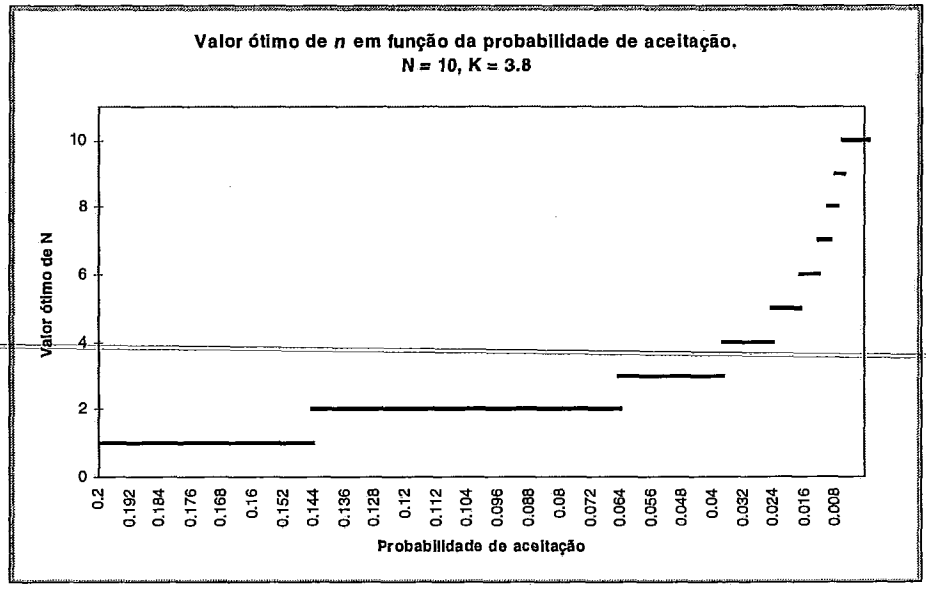


Figura 3.16 - Valor ótimo de n (número de estados calculados por processador entre sincronizações) em função da probabilidade de aceitação. N = 10 processadores, $K = 3{,}8$.

Observe-se a natureza exponencial do gráfico na Figura 3.16. Os gráficos nas Figuras 3.17 e 3.18 mostram a influência da escolha ótima de n sobre o *speedup* médio para $K = 50$ e $K = 3{,}8$. ($N = 10$ processadores)

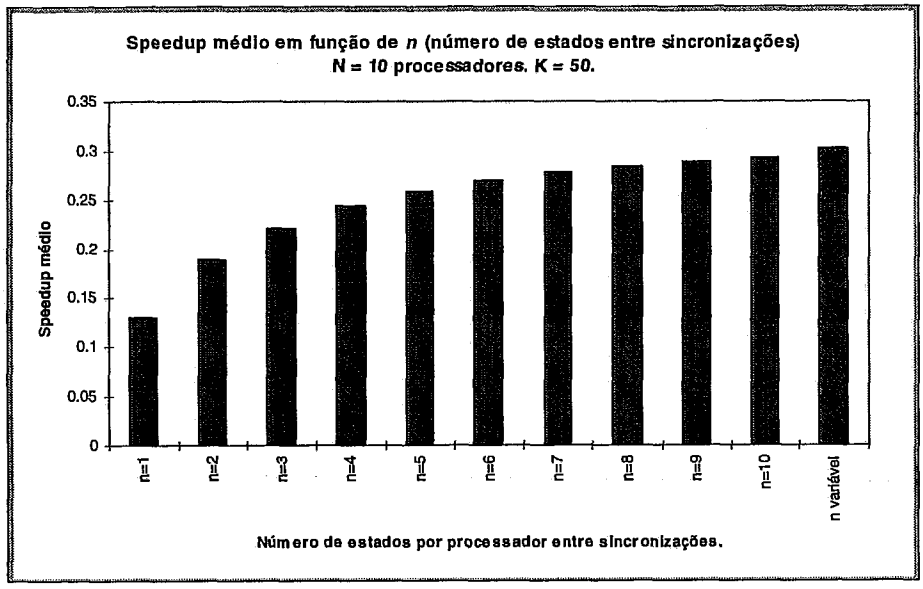


Figura 3.17 - Influência da escolha ótima de n sobre o *speedup* médio. $K = 50$.

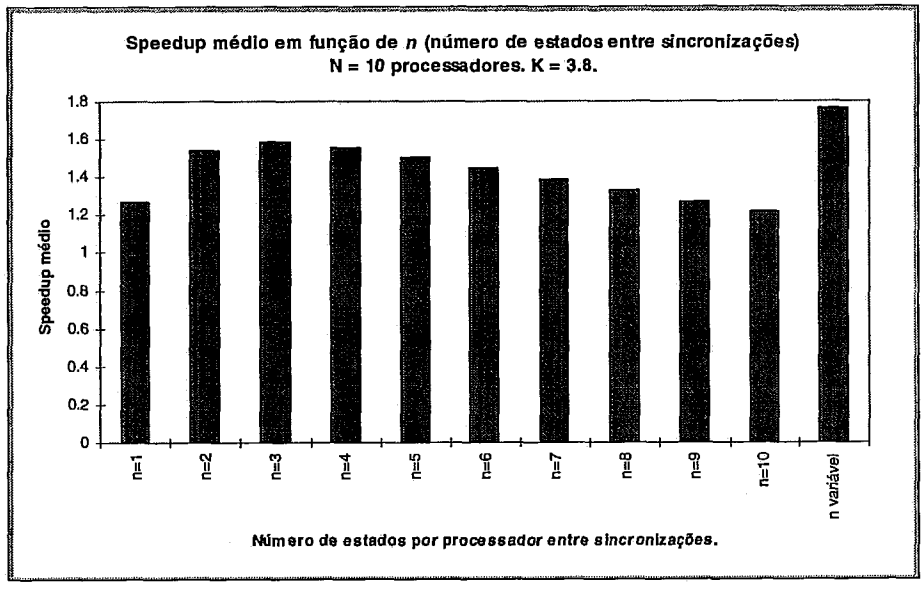


Figura 3.18 - Influência da escolha ótima de n sobre o *speedup* médio. $K = 3{,}8$.

Conforme vimos anteriormente, a simples observação da Figura 3.7 evidenciou a pouca eficiência de alocar, em temperaturas altas, um número grande de processadores à máquina paralela e nos sugeriu a utilização de uma estratégia adaptativa. Consideremos então esta possibilidade.

Seja a Equação para o *speedup* (Equação 3.19) repetida aqui como Equação 3.20:

$$speedup = \frac{E'[S]}{n + K + Pr'(M \neq 0)} \tag{3.20}$$

Nos exemplos anteriores o número de processadores na máquina paralela (N) foi mantido fixo em 10 processadores. Isto tem o efeito de tornar constante o valor de K

durante todo o processo de otimização do circuito. Uma vez que o valor de $E'[S]$ e de n variam com a probabilidade de aceitação, é razoável supor que o valor da função *speedup* poderia ser otimizado pela variação de K. A partir da Equação (3.14), recordamos que $K = t_2/t_1$ é uma constante de proporcionalidade, indicativa do tempo gasto para que todos os processadores enviem o resultado de suas avaliações ao mestre e deste recebam o movimento efetivado. Uma vez que , a cada instante de tempo, apenas uma mensagem pode circular pelo barramento, este constitui-se em um ponto de serialização do algoritmo e o valor de K é, obviamente, função do número de processadores trocando mensagens. Assim, a equação (3.14) pode ser reescrita como:

$$K[N] = \frac{t_2[N]}{t_1} \tag{3.21}$$

resultando na seguinte expressão para o *speedup*:

$$speedup(a(T), N, n) = \frac{E'[S]}{n + K[N] + Pr'(M \neq 0)} \tag{3.22}$$

onde:

$$E'[S] = [1 - a(T)]^{n*N} * n * N + \sum_{m=1}^{n*N} a(T) * [1 - a(T)]^{m-1} * m \tag{3.23}$$

$$Pr'(M \neq 0) = \left[1 - (1 - a(T))^{n*N}\right] \tag{3.24}$$

A Figura 3.19 mostra os valores medidos de $t_2[N]$ para N variando de 2 a 10 processadores. A reta assinalada corresponde à regressão linear dos pontos medidos. Observe que o tempo de comunicação varia linearmente com o número de processadores na máquina virtual.

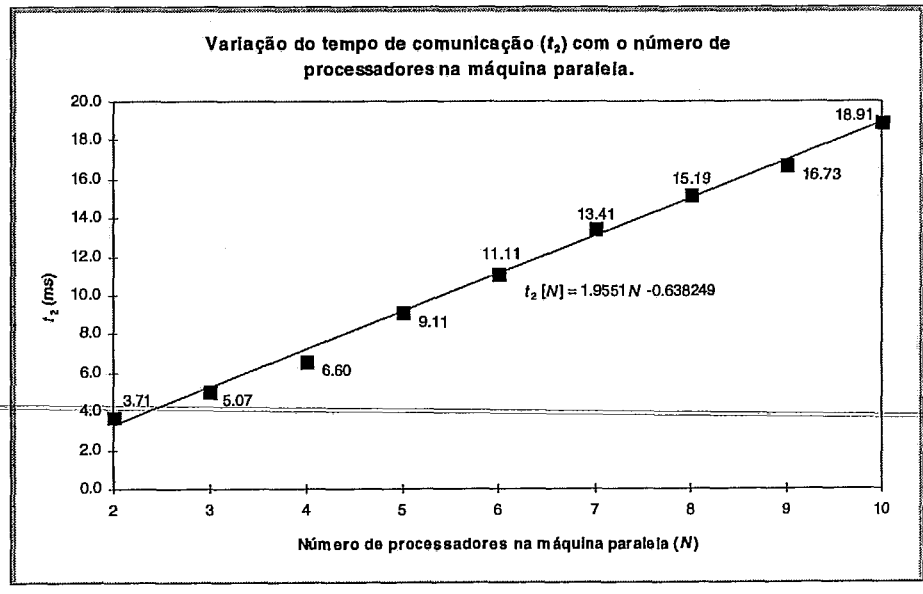


Figura 3.19 - Tempo de comunicação em função do número de processadores.

Os valores de $K[N]$ podem ser facilmente derivados a partir dos valores medidos para t_2. O problema agora consiste em, para cada valor de $a(T)$, maximizar a expressão do *speedup* (Equação 3.22) em relação ao número de processadores (N) e ao número de

estados calculados por processador entre sincronizações (n). Uma vez que, nos experimentos realizados, dispunha-se de um máximo de 10 processadores e o número de estados por processador foi limitado em 20, a otimização do valor do *speedup* foi calculada exaustivamente, para todos os valores de N e n. As Figuras 3.20 e 3.21 mostram, respectivamente, os valores ótimos de N e n para $K = 50$ e $K = 3{,}8$.

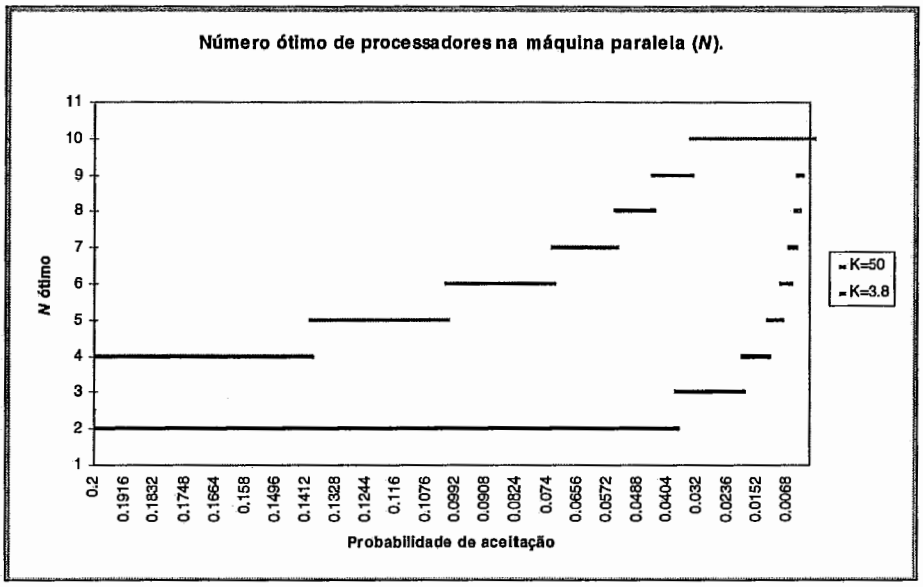


Figura 3.20 - Número ótimo de processadores na máquina paralela.

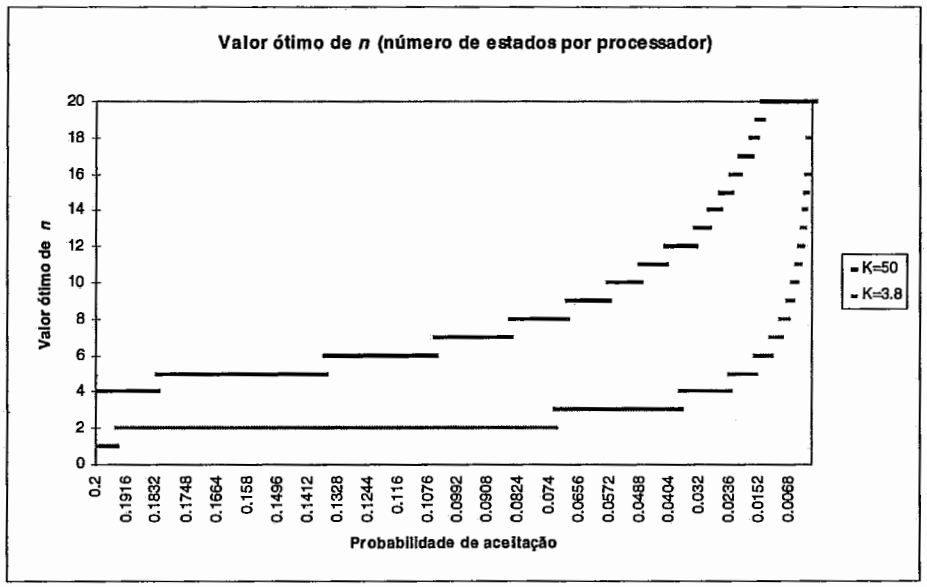


Figura 3.21 - Número ótimo de estados calculados por processador entre sincronizações.

Pela observação das figuras acima percebe-se que, para circuitos pequenos, parece ser interessante confiar em um crescimento mais rápido de n a fim de aumentar o número de estados calculados por nível. Ao contrário, para circuitos maiores, obtém-se melhores resultados quando o número de processadores tem um crescimento mais rápido do que o número de movimentos calculados por processador.

As curvas acima podem ser facilmente calculadas caso a caso, para cada circuito a ser otimizado. Para isto, basta determinar o valor de t_1, o tempo gasto para propor e avaliar um único movimento. A partir de t_1 derivam-se facilmente os valores de $K[N]$, necessários à otimização da equação (3.22). Alternativamente, pode-se proceder a uma

aproximação razoável a partir da observação da natureza exponencial das curvas. Qualquer estratégia em que o número de processadores e o número de movimentos calculados por processador cresça exponencialmente à medida em que a temperatura decaia, deve fornecer resultados satisfatórios. As Figuras 3.22 e 3.23 mostram o efeito da escolha ótima de N e n para $K = 50$ e $K = 3{,}8$, respectivamente.

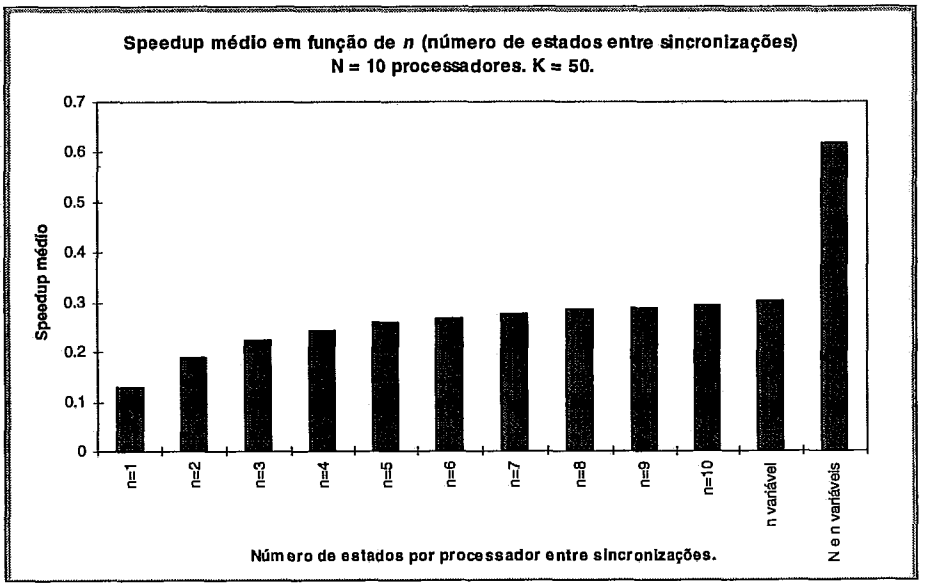


Figura 3.22 - Influência da escolha ótima de N e n sobre o *speedup* médio. $K = 50$.

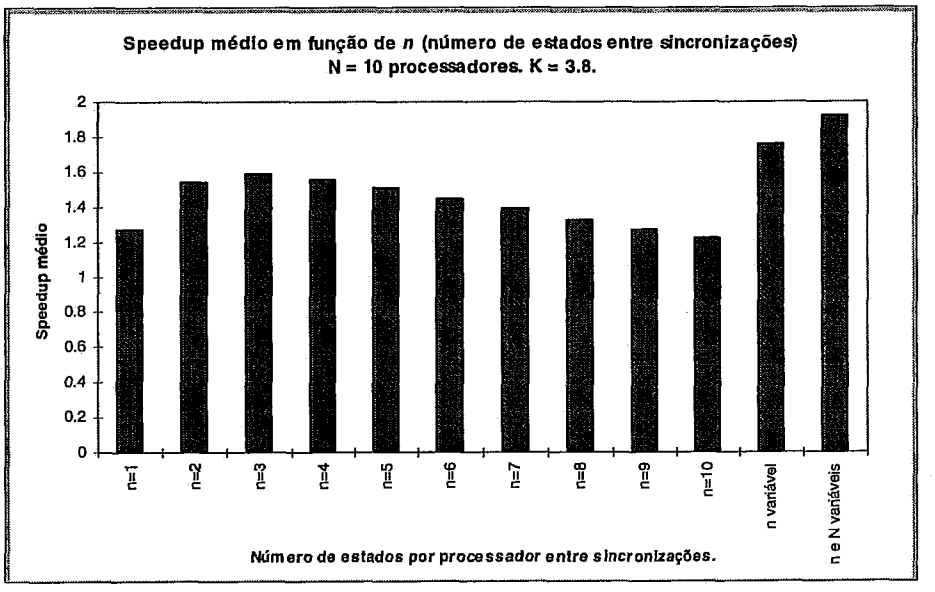


Figura 3.23 - Influência da escolha ótima de N e n sobre o *speedup* médio. $K = 3{,}8$.

Deve-se notar que, apesar dos resultados significativamente melhores obtidos, não foi possível obter *speedup* real para o circuito pequeno, onde $K = 50$. Isto sugere a existência de uma complexidade mínima para os circuitos otimizados, abaixo da qual não é efetivo aplicar-se a estratégia de paralelização.

Na seção seguinte são detalhados os resultados obtidos.

3.4.3) Resultados:

As tabelas 2 e 3 mostram os resultados obtidos para os circuitos com $K = 50$ (100 nós, 300 equipotenciais) e $K = 3{,}8$ (1024 nós, 3000 equipotenciais). Os valores referem-se apenas ao domínio do algoritmo adaptativo ($a(T) < 0{,}2$). O algoritmo para temperaturas altas será visto posteriormente.

	Serial		Adaptativo	
	Custo	tempo(s)	Custo	tempo(s)
Média	1915	63,78	1942	86,02
Desvio padrão	28	0,60	20	7,46
Mínimo	1884	63,17	1911	74,39
Máximo	1963	64,98	1965	95,94
	Speedup		0,7414	

Tabela 2 - Resultados para o circuito com 100 nós e 300 equipotenciais ($K = 50$).

	Serial		Adaptativo	
	Custo	tempo(s)	Custo	tempo(s)
Média	61929	10.343,17	62261	2671,00
Desvio padrão	1193	107,84	1288	82,01
Mínimo	59708	10.059,80	61021	2560,35
Máximo	63420	10.422,20	64904	2805,65
	Speedup		3,87	

Tabela 3 - Resultados para o circuito com 1024 nós e 3000 equipotenciais ($K = 3{,}8$).

Algumas observações podem ser feitas a partir dos valores nas tabelas:

1. No que diz respeito à qualidade da solução gerada, os resultados obtidos comprovam a eficiência do método adaptativo.

2. O valor do *speedup* obtido na Tabela 2 confirma a pouca eficiência do método para circuitos pequenos.

3. Para o circuito na Tabela 3, o valor do *speedup* obtido foi significativamente maior do que aquele previsto pelos estudos teóricos. Isto se deve ao fato de que, provavelmente, não é válida nossa hipótese de que a probabilidade de aceitação de um novo estado decaia linearmente com a temperatura.

A Figura 3.24 mostra o gráfico do decaimento de $a(T)$ com a temperatura. Os pontos no eixo das abcissas correspondem às temperaturas utilizadas em cada um dos níveis pelos quais passa o algoritmo em direção ao resfriamento. Pela observação do gráfico, percebe-se que o decaimento da probabilidade de aceitação é mais acentuado no início e que, portanto, o algoritmo passa menos tempo em temperaturas altas, onde ele é menos eficiente. Assim, para o cálculo preciso do *speedup* teórico, a integral da equação 3.22 teria de ser ponderada pelo tempo de duração de cada $a(T)$.

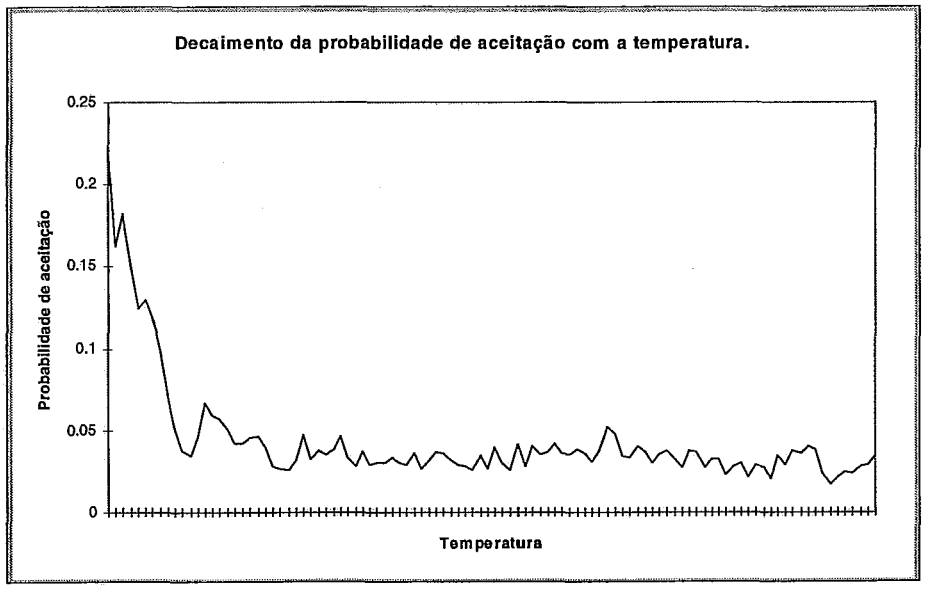


Figura 3.24 - Gráfico do decaimento da probabilidade de aceitação com a temperatura.

3.5) O algoritmo em altas temperaturas:

A ineficiência do algoritmo adaptativo em temperaturas elevadas deve-se ao fato de que, durante esta fase do *annealing*, a maioria dos estados propostos são aceitos, o que torna a cardinalidade do conjunto serializável muito pequena. Para superar tal dificuldade foi abandonada a premissa de que, em qualquer instante de tempo, a base de dados é única para todos os processadores. Em cada temperatura, cada um dos processadores executa o *annealing* serial na sua totalidade, em sua cópia local do problema. Depois que todas as cadeias são computadas é realizada uma sincronização global. A cadeia de mais baixo custo é escolhida como ponto de partida para a próxima temperatura.

A expectativa de obter-se algum ganho na velocidade de processamento é baseada na esperança de que as cadeias de Markov possam ser mais curtas do que no *annealing* serial.

Como veremos posteriormente, esta não parece ser uma estratégia apropriada ao regime de baixas temperaturas e, assim, parece natural combiná-la com o algoritmo adaptativo. Este e outros pontos serão vistos a seguir.

3.5.1) Fundamentos:

Em todos os experimentos descritos anteriormente, o equilíbrio em uma dada temperatura é obtido através do processamento de um número suficiente de movimentos propostos. Este número é obtido através de uma constante de proporcionalidade (*ESTBYCELL*) e é diretamente proporcional ao número de módulos no circuito.

$$npassos = ESTBYCELL * nmodulos \qquad (3.25)$$

onde:

npassos ⇒ número de movimentos calculados por temperatura;
nmodulos ⇒ número de módulos no circuito;

A constante *ESTBYCELL* foi determinada empiricamente através do estudo da qualidade da solução gerada em um número suficiente de execuções do algoritmo. A Tabela 4 mostra alguns valores obtidos para um circuito composto por 80 módulos e 30 redes equipotenciais. Todos os dados referem-se a uma média de 20 rodadas.

	Custo			
ESTBYONE	Média	Desvio Padrão	Mínimo	Máximo
1	105,67	9,71	89	139
5	93,40	5,55	85	103
15	83,67	3,34	77	93
30	78,95	2,70	75	84

Tabela 4 - Qualidade da solução gerada em função de *ESTBYONE*

A partir da observação da Tabela, foi utilizado *ESTBYONE* = 15 uma vez que este valor pareceu resultar em soluções de boa qualidade a um custo de processamento aceitável. Cabe salientar que tal critério de estabilidade foi utilizado para todos os circuitos, durante todo o processo de *annealing*. Ainda que não tenhamos tido a oportunidade de aprofundar este estudo, pensamos que seria válida uma investigação posterior no sentido de determinar se o tamanho ótimo da cadeia é função da temperatura ou da complexidade do circuito[1]. No entanto, ainda que tal dependência exista, ela não invalida os resultados qualitativos que pretendemos aqui demonstrar.

Conforme já assinalado, a expectativa em reduzir-se o tempo de processamento na versão paralela reside na esperança de que a sincronização entre os processos ao término do cálculo de cada cadeia, possibilite a utilização de cadeias mais curtas do que a correspondente versão serial. A fim de obtermos a comprovação de tal teoria, considere o enunciado da seguinte hipótese:

Hipótese 1:

Seja o algoritmo de *annealing* dividido em 2 fases: a fase 1 para $a(T) \leq \alpha$, chamada de *Parchain*, e a fase 2, para $a(T) > \alpha$, chamada de *OneChain*. Seja C a configuração final atingida por *Parchain*, correspondente ao estado inicial de *OneChain*. A qualidade de C é função do comprimento da cadeia usado em *Parchain*.

A fim de obtermos a comprovação da Hipótese 1, consideremos a Tabela 5. Nesta tabela é apresentado o custo médio de C em função da constante *ESTBYCELL*. (A partir deste ponto, a constante de proporcionalidade usada em *Parchain* passa a ser chamada de *ESTBYPAR*. De maneira similar, *ESTBYONE* passa a designar a constante de proporcionalidade em *OneChain*)

ESTBYPAR	Custo em C
1	218,6
5	142,5
15	122,9

Tabela 5 - Influência do comprimento da cadeia na qualidade de C.

[1]Uma possibilidade neste sentido é o trabalho desenvolvido por Huang, Romeo e Sangiovanni-Vincentelli em [Huan86], onde o comprimento da cadeia é dinamicamente determinado. Segundo os autores, foi possível obter significativos ganhos no tempo de processamento pela utilização de cadeias mais curtas em altas temperaturas

Os valores na Tabela 5 correspondem à média de 20 rodadas. A comprovação da Hipótese 1 torna-se evidente pela simples observação da Tabela. Consideremos agora o enunciado da Hipótese 2.

Hipótese 2:

Para um valor de α já suficientemente próximo ao resfriamento, a qualidade da solução gerada por *OneChain* (a solução final do processo de *annealing*) é função da qualidade de *C*.

A comprovação da Hipótese 2 foi obtida novamente por resultados experimentais os quais são apresentados na Tabela 6. Os valores referem-se à média de 20 rodadas do algoritmo para $\alpha = 20\%$ e $ESTBYONE = 15$.

ESTBYPAR	Custo em *C*	Custo Final
1	218,6	95,03
5	142,5	86,90
15	122,9	83,67

Tabela 6 - Influência da qualidade de *C* na solução final gerada pelo processo de *annealing*.

Consideremos finalmente a Hipótese 3, base da estratégia de paralelização do *annealing* em temperaturas altas.

Hipótese 3:

A utilização de cadeias paralelas para $a(T) \geq \alpha$ permite a obtenção de soluções *C* com qualidade equivalente a do *Parchain* serial, utilizando porém cadeias de comprimento mais curto.

O enunciado da Hipótese 3 é comprovado pelos resultados experimentais mostrados na Tabela 7. Os valores referem-se à média de 20 rodadas de cada um dos algoritmos.

	Custo em *C*
Serial $ESTBYPAR = 15$	122,9
Paralelo $ESTBYPAR = 15$	111,6
Paralelo $ESTBYPAR = 5$	123,4
Paralelo $ESTBYPAR = 3$	133,1
Paralelo $ESTBYPAR = 1$	156,0

Tabela 7 - Obtenção de configurações *C* equivalentes com a utilização de cadeias paralelas.

Baseado nos resultados acima, foi utilizado o valor de $ESTBYPAR = 5$ a fim de determinar o comprimento das cadeias paralelas. Com o objetivo de enfatizar os resultados contidos na Tabela 7, considere os histogramas nas Figuras 3.25 e 3.26.

O histograma em 3.25 mostra, para o algoritmo serial, a distribuição de temperaturas nas quais a probabilidade de aceitação de novos estados caiu abaixo do valor estipulado α. Observe, para *ESTBYPAR* = 5, o deslocamento do ponto de troca de estratégias para temperaturas mais baixas, conseqüência do aumento na função custo de *C*.

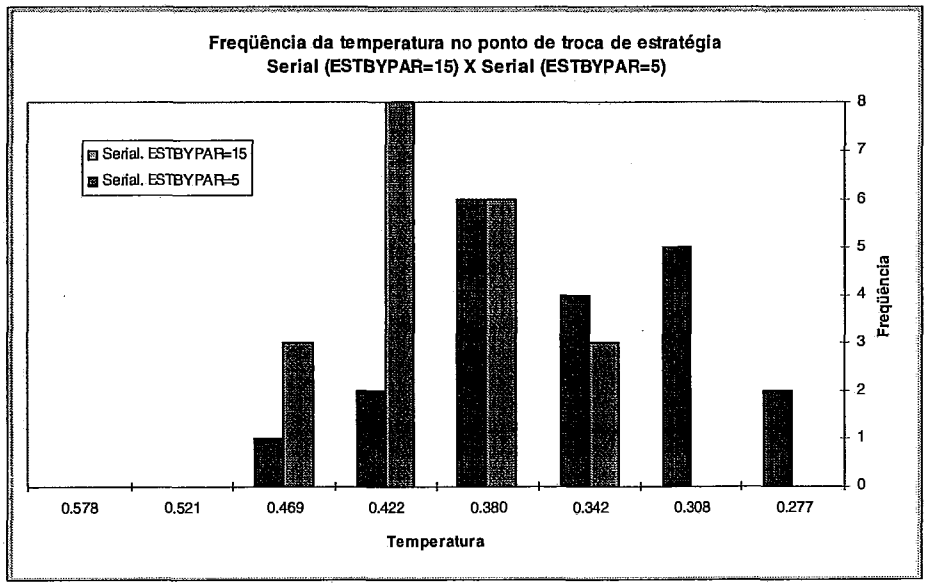


Figura 3.25 - Distribuição de temperaturas no ponto de troca de estratégia.
Serial (*ESTBYPAR* = 15) X Serial (*ESTBYPAR* = 5)

De maneira similar, o gráfico na Figura 3.26 mostra que a utilização de cadeias paralelas (ainda que mais curtas) resulta no deslocamento do ponto de troca de estratégias para temperaturas até mesmo superiores do que as registradas no equivalente serial.

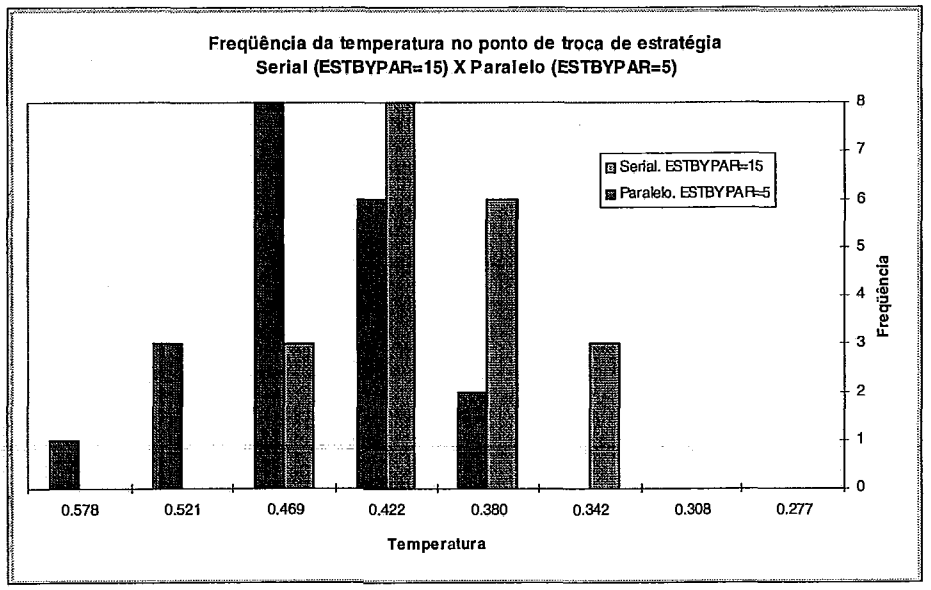


Figura 3.26 - Distribuição de temperaturas no ponto de troca de estratégia.
Serial (*ESTBYPAR* = 15) X Paralelo (*ESTBYPAR* = 5)

3.5.2) Algumas considerações a respeito da estratégia de Cadeias Paralelas:

A partir dos resultados mostrados na seção anterior é possível afirmar com certeza que, para o mesmo valor de *ESTBYPAR*, o algoritmo paralelo atinge o ponto de troca de estratégias α com uma configuração de mais baixo custo e, possivelmente, em uma temperatura mais alta. A partir daí divisou-se uma estratégia onde é possível obter ganho em velocidade de processamento a partir da redução do tamanho das cadeias. Cabem, no entanto, algumas questões e considerações a respeito da estratégia de Cadeias Paralelas.

A primeira questão a ser enunciada é: "Se o algoritmo permite a redução do comprimento das cadeias em até 3 vezes como no exemplo (o que aponta para um *speedup* da ordem de 3) por que é necessária uma estratégia para temperaturas baixas?" A fim de responder a esta pergunta o circuito de teste foi otimizado usando-se a estratégia das Cadeias Paralelas durante toda a duração do *annealing*. Os resultados podem ser vistos na Tabela 8 para *ESTBYPAR* = 5.

Média	87,50
Desvio Padrão	4,33
Mínimo	81
Máximo	96

Tabela 8 - Resultados da otimização do circuito de teste usando apenas a estratégia de Cadeias Paralelas.

Observa-se que o algoritmo não atingiu os mesmos resultados da versão serial. Conclui-se portanto que, em baixas temperaturas, talvez seja necessário usar valores maiores de *ESTBYPAR* resultando em cadeias de comprimento maior. Assim, teríamos um algoritmo cujo *speedup* diminui com a temperatura. Parece natural portanto, combinar esta estratégia com o algoritmo adaptativo visto anteriormente que caracteriza-se, justamente, por fornecer melhores valores de *speedup* à medida em que o processo aproxima-se do resfriamento. No entanto, a escolha do ponto de mudança de estratégia aqui utilizado ($\alpha = 20\%$) foi um tanto arbitrária e é lícito supor que pode haver uma melhoria significativa no desempenho das estratégias combinadas pela utilização do valor ótimo de α.

Um outro aspecto que deve ser considerado é o fato dos processos sincronizarem-se ao término do cálculo de cada temperatura. Isto traz o efeito indesejado de que o tempo de processamento é amarrado pelo mais lento dos processadores da máquina paralela. Uma possível solução para este problema é escolher-se a cadeia de menor custo entre os N primeiros processos que completem o cálculo em cada temperatura. A questão que deve então ser respondida é qual o tamanho mínimo do conjunto N para que o esquema de cadeias curtas seja ainda efetivo? De posse deste dado poder-se-ia montar um esquema de barreira em que os N primeiros processadores que chegassem ao ponto de sincronização determinassem a configuração inicial para a próxima temperatura. Os demais interromperiam o cálculo, receberiam a nova base de dados e recomeçariam o processamento para a nova temperatura. Esta alternativa não chegou a ser utilizada em nossos testes.

3.5.3) Resultados:

As tabelas 9 e 10 mostram os resultados obtidos para um circuito médio (100 nós, 300 equipotenciais) e um grande (1024 nós, 3000 equipotenciais). Os valores referem-se apenas ao domínio do algoritmo de Cadeias Paralelas ($a(T) > 0{,}2$) e dizem respeito à média de, no mínimo, 20 rodadas. Em todos os testes foram utilizados os seguintes parâmetros: $ESTBYPAR = 5$, $ESTBYONE = 15$, $\alpha = 20\%$, $N = 10$ (número de processadores na máquina paralela).

	Serial tempo(s)	Cadeias Paralelas tempo(s)
Média	26,16	14,96
Desvio padrão	0,54	0,26
Mínimo	25,13	14,68
Máximo	26,95	15,24
	Speedup	1,7487

Tabela 9 - Resultados do algoritmo de Cadeias Paralelas para o circuito com 100 nós e 300 equipotenciais.

	Serial tempo(s)	Cadeias Paralelas tempo(s)
Média	2329,94	808,47
Desvio padrão	37,75	10,68
Mínimo	2302,47	785,24
Máximo	2384,42	814,26
	Speedup	2,8819

Tabela 10 - Resultados do algoritmo de Cadeias Paralelas para o circuito com 1024 nós e 3000 equipotenciais.

Observe-se que, para circuitos maiores, o valor de *speedup* aproxima-se do fator de redução do tamanho da cadeia.

Em termos da qualidade da solução final gerada, os resultados obtidos foram sempre compatíveis com o da versão serial.

3.6) Sumário e Conclusões:

- A paralelização do algoritmo de *simulated annealing* não é trivial e, no ambiente de teste utilizado, produz *speedup* significativo apenas para problemas de tamanho muito grande (*speedup* de aproximadamente 3 para um circuito composto por 1024 nós e 3000 redes equipotencias, utilizando 10 processadores na máquina paralela).
- O algoritmo de Kravitz e Rutenbar [Krav87], freqüentemente citado em literatura como uma alternativa viável para a paralelização do *annealing*, é equivocado. O algoritmo baseia-se na idéia do conjunto serializável mínimo onde, em cada passo, todos os estados rejeitados e um único estado aceito são contabilizados no sentido do equilíbrio. Verificou-se que esta estratégia não preserva a proporção entre estados aceitos/rejeitados verificada no algoritmo serial e, por este motivo, as soluções geradas para o algoritmo paralelo são de baixa qualidade.

- O parâmetro temperatura, que controla os movimentos de *hill-climbing* no *Simulated Annealing*, altera profundamente o comportamento do algoritmo durante a sua execução. Este comportamento dinâmico fornece novas oportunidades de explorar o paralelismo inerente ao processo. A estratégia de paralelização empregada consiste em mudar parâmetros de um algoritmo, ou mesmo o algoritmo utilizado de acordo com a fase do processo.
- A estratégia de paralelização empregada para o *annealing* em baixas temperaturas consiste em uma variante do algoritmo especulativo proposto na literatura [Sohn95]. O algoritmo tem um comportamento inteiramente dinâmico onde o número de processadores integrantes da arquitetura paralela e o número de estados avaliados por processador entre sincronizações variam com a temperatura. Durante o domínio do algoritmo (probabilidade de aceitação de novos estados menor do que 20%), para um máximo de 10 processadores na máquina paralela, e um circuito composto por 1024 nós e 3000 redes, foi possível obter um *speedup* de 3,87 unidades.
- Para o algoritmo em temperaturas altas, partiu-se da constatação de que, para cadeias de mesmo comprimento, o algoritmo paralelo atinge o ponto de troca de estratégias com uma configuração de mais baixo custo e, possivelmente, em uma temperatura mais alta. A partir daí divisou-se uma estratégia onde é possível obter ganho em velocidade de processamento a partir da redução do tamanho das cadeias. Durante o domínio do algoritmo (probabilidade de aceitação de novos estados maior do que 20%), 10 processadores na máquina paralela, e um circuito composto por 1024 nós e 3000 redes, foi possível obter um *speedup* de 2,88 unidades.

CAPÍTULO 4

O Algoritmo Genético Seqüencial

4.1) Introdução:

Nos últimos anos, os algoritmos genéticos adquiriram posição de destaque como uma técnica prática e robusta de busca e otimização. Diversas áreas tais como projeto de circuitos VLSI, controle adaptativo, problemas de transporte, planejamento de estratégias e aprendizado de máquinas têm se beneficiado desta abordagem. A popularidade dos algoritmos genéticos pode ser medida através de uma conferência bianual, um jornal internacional e uma massa crescente de literatura dedicada à teoria, prática, e aplicações desta técnica.

Os algoritmos genéticos baseiam-se nos conceitos de genética e evolução das espécies encontrados na natureza. O interesse em métodos heurísticos baseados em fenômenos físicos ou naturais teve início na década de 70 quando Holland [Holl75] enunciou os princípios dos algoritmos genéticos. Este interesse foi reforçado pelo método de *Simulated Annealing* proposto por Kirkpatrick, Gelatt e Vechhi em 1983 [Kirk83]. O método de *Simulated Annealing* baseia-se em considerações termodinâmicas. Estratégias evolucionárias e algoritmos genéticos, por outro lado, buscam inspiração na natureza onde o fenômeno conhecido por seleção natural leva a sobrevivência dos indivíduos mais aptos. Tais técnicas têm em comum o uso de um mecanismo de busca probabilística orientada no sentido de diminuir uma dada função custo. Os três métodos apresentam uma probabilidade alta de localizar o mínimo global em um cenário composto por vários mínimos locais. O modo de operação de cada um deles é, no entanto, marcadamente diferente.

O algoritmo *Simulated Annealing*, descrito anteriormente, probabilisticamente gera uma seqüência de estados de acordo com uma estratégia de *resfriamento* buscando a convergência para o mínimo global. Estratégias evolucionárias usam um operador de mutação como um mecanismo de busca e um operador de seleção para direcionar a busca no sentido das regiões mais promissoras do espaço de estados. Os algoritmos genéticos geram uma seqüência de populações usando os operadores de *crossover* e mutação como mecanismos de busca e uma estratégia de seleção para fazer com que uma nova população seja composta, em média, por indivíduos mais aptos do que os integrantes das populações anteriores.

As seções seguintes deste Capítulo apresentam os princípios de funcionamento dos algoritmos genéticos e os detalhes de implementação de uma série de técnicas avaliadas no processo de busca por um algoritmo seqüencial que fosse adequado para a aplicação e avaliação de técnicas de paralelização. Por adequado entende-se neste ponto um algoritmo robusto, isto é, que, em sucessivas experiências, produza resultados próximos àqueles obtidos pelo algoritmo de *Simulated Annealing*.

Os algoritmos genéticos e a seleção natural:

Na natureza, em geral, sobrevivem os indivíduos mais capacitados para a competição por recursos escassos. A capacidade de adaptação a um ambiente permanentemente em mutação é essencial à sobrevivência das espécies. O conjunto de características de um indivíduo, que unicamente o distingue dos demais, determina a sua capacidade de sobrevivência. Estas características, por sua vez, são determinadas pelo material genético do indivíduo. Cada uma das características do indivíduo é controlada por uma unidade básica chamada de *gene*. Os conjuntos de genes formam os cromossomas, a chave para a sobrevivência do indivíduo em um ambiente competitivo.

Ainda que a evolução se manifeste sob a forma de uma sucessão de modificações nas características dos indivíduos, são as modificações no material genético das espécies que constituem-se na essência do processo de evolução. Assim, a evolução das espécies se dá através da ação conjunta da seleção natural e da recombinação de material genético que ocorre durante a reprodução.

Na natureza, a competição entre indivíduos por recursos escassos tais como comida, espaço ou mesmo parceiros para o ato sexual, resulta em que os indivíduos mais aptos têm mais chances de sobreviver e reproduzir-se. Desta forma, os genes dos indivíduos mais aptos sobrevivem enquanto que os genes dos indivíduos menos favorecidos desaparecem. O mecanismo de seleção natural leva à sobrevivência dos indivíduos mais aptos e, implicitamente, à sobrevivência dos genes mais aptos.

O processo de evolução inicia-se na recombinação de material genético proveniente dos pais durante a reprodução. Novas combinações de genes são geradas a partir das já existentes. A troca de material genético entre cromossomas é chamada de *crossover*. Pedaços dos cromossomas dos pais são trocados durante o *crossover* criando a possibilidade do surgimento da combinação "correta" de genes e, conseqüentemente, de indivíduos mais capacitados. A repetição do processo de seleção e *crossover* ao longo das gerações permite a evolução contínua do material genético da espécie e a geração de indivíduos que sobrevivem melhor em um ambiente competitivo.

Holland [Holl75] propôs os algoritmos genéticos como programas de computador que imitam o processo de evolução encontrado na natureza. Os algoritmos genéticos manipulam uma população de potenciais soluções para um problema de busca ou otimização. Eles operam sobre a representação codificada das soluções, equivalente ao material genético dos indivíduos na natureza. O algoritmo genético de Holland codifica as soluções em cadeias de *bits*. Cada solução é associada a uma medida de qualidade ou *fitness* que reflete a capacitação de um indivíduo em relação aos demais indivíduos da população. Assim como na natureza, um mecanismo de seleção força a contínua evolução da qualidade das gerações. Quanto maior o *fitness* de um indivíduo, maiores suas chances de reproduzir-se e de sobrevivência à próxima geração. A recombinação de material genético é simulada através de um mecanismo de *crossover* que troca pedaços entre as cadeias de bits dos pais. Outra operação, chamada de mutação, provoca alterações esporádicas e randômicas nas cadeias de bits. A operação de mutação tem também uma analogia direta com o fenômeno encontrado na natureza e tem o papel de reintroduzir na população material genético perdido.

A estrutura de um algoritmo genético simples pode ser vista na Figura 4.1. Durante a geração t o algoritmo mantém uma população de soluções potenciais (cromossomas) $P(t) = \{v_1^t, \ldots, v_n^t\}$. Cada solução v_i^t é avaliada para fornecer a medida da sua qualidade ou aptidão. Uma nova população $P(t+1)$ é então formada selecionando-se, probabilisticamente, os indivíduos mais aptos de $P(t)$. Alguns dos membros desta nova

população sofrem transformações por meio de operações de *crossover* e mutação. O operador de *crossover* combina características dos pais pela troca de segmentos dos seus cromossomas. O operador de mutação arbitrariamente altera um ou mais genes de um cromossoma selecionado com uma probabilidade igual à probabilidade de mutação.

Assim, o projeto de um algoritmo genético para uma dada aplicação deve prover os seguintes componentes:

- Uma representação na forma de cromossomas das soluções potenciais do problema.
- Uma forma de criar uma população inicial de soluções válidas.
- O projeto de uma função de avaliação que classifique os indivíduos em função de sua aptidão.
- Operadores genéticos que alterem a composição dos filhos.
- A atribuição de valores aos diversos parâmetros usados pelos algoritmos genéticos (tamanho da população, probabilidade de aplicação dos operadores genéticos, critério de parada, etc.)

```
/* t            ⇒    contador de gerações                                    */
/* P(t)         ⇒    População na geração t                                  */

t = 0;
Inicializa P(t);
Avalia P(t);              /* Calcula a função fitness para cada indivíduo na população.  */
Enquanto não atingiu o critério de terminação          {
        t = t + 1;
        Seleciona P(t) a partir de P(t-1);
        Altera P(t);      /* aplica os operadores de crossover e mutação.                */
        Avalia P(t);
}
Imprime resultados;
```

Figura 4.1 - Estrutura de um algoritmo genético simples.

Um exemplo:

Vamos supor uma população de POPSIZE=4 indivíduos. O cromossoma de cada um dos indivíduos é uma *string* de *bits* onde os bytes representam a posição no plano de cada um dos módulos.
Assuma que depois do procedimento de inicialização tenhamos a seguinte população:

$$\begin{array}{l} \qquad\quad x_1 \qquad y_1 \qquad x_2 \qquad y_2 \qquad \ldots \\ v_1 = (00001010,00011001,00011110,00000011,00010001,00001001,\ldots) \\ v_2 = (00010101,00001110,00011001,00000011,00001110,00010001,\ldots) \\ v_3 = (00000111,00011001,00000110,00011000,00001101,00011101,\ldots) \\ v_4 = (00001010,00001100,00011111,00000000,00000001,00010000,\ldots) \end{array}$$

Durante a fase de avaliação é atribuído a um valor a cada cromossoma, representativo do grau de aptidão do indivíduo correspondente (no caso do exemplo, um número real entre 0.0 e 1.0). Suponha que os seguintes valores sejam atribuídos aos indivíduos por uma função aptidão $f(\ )$ a ser definida:

$f(v_1) = 0.0$
$f(v_2) = 0.7434$
$f(v_3) = 1.0$
$f(v_4) = 0.2075$

Em seguida, pares de indivíduos são selecionados probabilisticamente para reprodução: quanto maior a aptidão de um indivíduo, maior sua probabilidade de ser selecionado. Assim, suponhamos que os seguintes pares de cromossomas sejam selecionados para reprodução:

v_4 X v_3
v_2 X v_3

Para cada um desses pares é gerado um número inteiro randômico *pos* entre 0 e o comprimento da *string* em bits. O primeiro par de cromossomas é repetido aqui:

$v_4 = (000010100|000110000011111000000000000000100010000...)$
$v_3 = (000001110|001100100000110000110000000110100011101...)$

Supondo que o número gerado $pos = 9$, os cromossomas são cortados depois do nono bit e os pedaços são trocados:

$v_5 = (000010100|001100100000110000110000000110100011101...)$
$v_6 = (000001110|000110000011111000000000000000100010000...)$

O segundo par de cromossomas é

$v_2 = (000101010000111000011001000000110000111000010001...)$
$v_3 = (000001110001100100000110000110000000110100011101...)$

Resultando em:

$v_7 = (000101010000111000011011000011000000011010001 1101...)$
$v_8 = (000001110001100100000100100000011000011100001000 1...)$

O operador de mutação é então aplicado a cada um dos filhos. Cada um dos *bits* nos cromossomas é invertido com probabilidade *pm* (*pm* é um valor baixo, tipicamente da ordem de 1%, ou menor). A população final é mostrada a seguir. Os bits que sofreram mutações aparecem em negrito.

$v_1 = (000010100001100100011110000000110001000100001001...)$
$v_2 = (000101010000111000011001000000110000111000010001...)$
$v_3 = (000001110001100100000110000110000000110100011101...)$
$v_4 = (000010100000110000011111000000000000000100010000...)$
$v_5 = (000010100001**0**0010000011000011000000011010001 1101...)$
$v_6 = (000001110000110000011111000000000000000100010000...)$
$v_7 = (000101010000111000010110000110000000110100011101...)$
$v_8 = (000001110001100100001001000000110000111000010**1**01...)$

Finalmente cada cromossoma é decodificado e é calculada a função custo da nova população. A nova geração é obtida com os POPSIZE indivíduos mais aptos. Os demais indivíduos são retirados da população. A esperança aqui é a de que cada geração seja composta, em média, por indivíduos mais aptos do que os da geração anterior.

4.2) O Modelo de População Única: Ajuste dos parâmetros e escolha dos operadores.

Esta seção descreve a implementação de um algoritmo genético usando o modelo clássico de população única. A população é dita centralizada, ou única, quando a aptidão de um indivíduo diz respeito a todos os demais indivíduos da população.

A concepção inicial deste algoritmo foi baseada no algoritmo genético clássico seqüencial descrito em [Mare94] (Figura 4.2). Em função do esforço despendido para o ajuste do algoritmo ao problema de *placement*, diversos operadores e parâmetros do problema (operador de *crossover*, operador de mutação, tamanho da população, cálculo da função aptidão, etc.) tiveram de ser testados ou ajustados. As diversas alternativas experimentadas e as justificativas para as escolhas efetuadas são apresentadas a seguir. Ao final da seção são apresentados os resultados obtidos para o algoritmo genético clássico (população única).

```
Genese ( );                    /* Gera população inicial com POPSIZE indivíduos */

Repete por T gerações          {
        Evaluation( );         /* Avalia o custo e fitness de todos os indivíduos */

        Gera POPSIZE/2 cruzamentos na forma:          {
                Seleção( );    /* Probabilisticamente escolhe um par de indivíduos */
                Crossover( );  /* Gera um par de filhos */
                Mutação( );    /* Probabilisticamente aplica mutação aos novos indivíduos */
        }

        Seleciona, dentre os novos indivíduos e a geração atual, os POPSIZE
        indivíduos mais aptos para formarem a nova geração.
}
Imprime resultados.
```

Figura 4.2 - O algoritmo genético clássico.

4.2.1) Representação das variáveis do problema

a) Representação em *strings* de *bits*:

Inicialmente as variáveis do problema foram codificadas utilizando-se a representação tradicional em *strings* de *bits*. O alfabeto binário oferece o maior número de "esquemas" por *bit* de informação [Gold89] e, por este motivo, a representação das soluções em *strings* de *bits* tem dominado as pesquisas em algoritmos genéticos. Esta codificação facilita ainda a análise teórica do algoritmo e permite a construção de elegantes operadores genéticos.

Seja, por exemplo, a solução v_1 representada pelos pares ordenados:

$v_1 = (10, 25); (30, 3); (17, 9), ...$

Cada um dos pares ordenados representa a localização no plano de um dos módulos componentes do circuito. A representação da solução v_1 em uma *string* de *bits* forneceria:

$$\begin{matrix} & x_1 & y_1 & x_2 & y_2 & \\ v_1 = (&00001010,&00011001,&00011110,&00000011,&00010001,00001001,...) \end{matrix}$$

O operador de *crossover* é implementado seccionando-se o cromossoma em um ponto arbitrário da cadeia. O operador de mutação, probabilisticamente, inverte o valor dos bits no cromossoma.

Conforme assinalado em [Srin94], uma desvantagem advinda da codificação das variáveis como *strings* de *bits* é a presença de *Hamming cliffs* (grandes distâncias de Hamming entre a representação binária de inteiros adjacentes). Por exemplo, 01111 e 10000 são as representações binárias dos números inteiros 15 e 16, respectivamente, e têm distância de Hamming de 5 unidades. A fim de que o algoritmo genético melhore uma dada solução passando o valor de uma variável de 15 para 16, ele deve alterar todos os bits da variável simultaneamente. Assim, a presença de *Hamming cliffs* introduz um problema de difícil solução para os algoritmos genéticos na medida em que os operadores tradicionais de mutação e *crossover* não conseguem lidar facilmente com esta situação.

Para minimizar esta dificuldade as variáveis do problema foram codificadas usando-se o código de Gray, no qual a representação binária de dois números inteiros adjacentes difere, no máximo, pelo valor de 1 bit.

A função *bin2gray( )* converte um número inteiro decimal I na sua representação usando o código de Gray:

$$bin2gray(I) = I \oplus (I/2)$$

As variáveis são armazenadas em código Gray e convertidas para representação decimal sempre que for necessário calcular a função custo associada a um cromossoma. A fim de diminuir o tempo de conversão os resultados são armazenados em uma tabela - *lookup* - consultada pela rotina *gray2bin( )*. A tabela *lookup* é construída no início do processamento utilizando as rotinas *InitGray( )* e *bin2gray( )*.

```
InitGray( )
{
int     i;

        for(i=0; i<DIMX; i++)
        lookup[bin2gray(i)]=i;
}
```

```
bin2gray(B)
unsigned charB;
{
        return B^(B>>1);
}


gray2bin(G)
unsigned char    G;
{
        return lookup[G];
}
```

As dimensões do plano de alocação são sempre potências de 2. Evita-se com isto a geração de estados inválidos pelos operadores de *crossover* e mutação. Suponha, por exemplo, que uma das dimensões do plano fosse equivalente a 12 unidades de medida. Seja uma coordenada, válida neste plano, equivalente a 8 unidades de medida. A representação em Gray desta coordenada é vista abaixo:

$$1100_{Gray} = 8 \text{ unidades}$$

Uma operação de mutação na coordenada acima poderia gerar:

$$1000_{Gray} = 15 \text{ unidades},$$

que é uma medida inválida no plano usado.

b) Pares ordenados:

Nesta representação as variáveis do problema (representadas pela localização no plano dos módulos do circuito) são mapeadas diretamente nos genes do cromossoma. O principal objetivo nesta representação é aproximar o algoritmo genético do espaço do problema. Esta aproximação permite a construção de operadores genéticos especiais, que incorporem conhecimento específico do problema a ser resolvido (Por exemplo, estados proibidos podem ser evitados ou pode-se, mais facilmente, estimular a geração de estados "promissores"). A favor desta abordagem, Goldberg escreveu em [Gold90]:

> "The use of real-coded or floating-point genes has a long, if controversial, history in artificial genetic and evolutionary search schemes, and their use as of late seems to be on the rise. This rising usage has been somewhat surprising to researchers familiar with fundamental genetic algorithm (GA) theory, because simple analyses seem to suggest that enhanced schema processing is obtained by using alphabets of low cardinality, a seemingly direct contradiction of empirical findings that real codings have worked well in a number of practical problems."

A representação de uma solução hipotética nos genes de um cromossoma pode ser vista na Figura 4.3. Observe que, uma vez que a localização dos módulos no plano é dada por números inteiros, a representação interna do cromossoma usando pares ordenados pouco difere da representação em *string* de *bits*. A diferença fundamental

reside na maneira pela qual os genes são tratados pelo algoritmo: um gene é uma unidade indivisível o que torna imediata a interpretação de seu conteúdo.

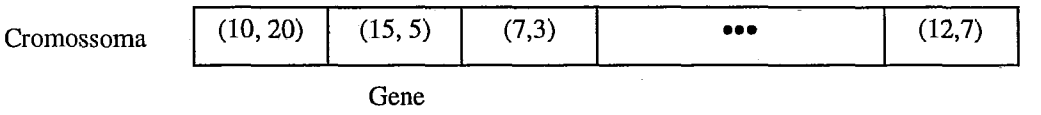


Figura 4.3 - Representação de um Indivíduo utilizando a localização dos módulos como genes.

Os experimentos conduzidos mostraram que a representação em pares ordenados, intuitivamente mais próxima do espaço do problema, possibilitou a obtenção de resultados mais consistentes pela utilização de operadores genéticos especialmente projetados.

4.2.2) A função custo.

a) Cálculo da função custo com violações (sobreposições de módulos):

Dado um cromossoma em particular, a função custo retorna uma "figura de mérito" deste cromossoma, ou seja, um valor numérico que, supõe-se, proporcional à "utilidade" ou "aptidão" do indivíduo representado pelo cromossoma. O custo associado a um cromossoma é uma característica individual, ou seja, ele não é influenciado pelo custo dos demais indivíduos da população.

A exemplo da abordagem utilizada no método *Simulated Annealing*, e, no que parece ser uma aproximação razoável dos demais objetivos de projeto (ruído, dissipação de calor, área, etc.), optou-se por minimizar o comprimento total da fiação. O comprimento de um fio ligando vários pinos - uma rede - é estimado pelo semiperímetro do retângulo que envolve todos os pinos (Figura 4.4). A restrição de que dois módulos não podem ocupar o mesmo lugar no plano é resolvida associando-se uma penalidade a cada violação. O somatório dessas penalidades é então incluído no cálculo da função custo.

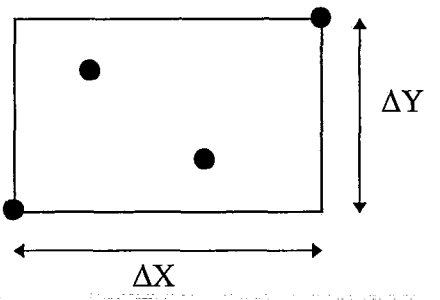


Figura 4.4 - Custo estimado da fiação

Assim, o custo total do placement para um circuito composto por i redes equipotenciais é dado por:

$$Cx = \sum_i \alpha_i * \left(\Delta X_i \cdot W_i^H + \Delta Y_i \cdot W_i^V\right) + p * \delta \qquad (4.1)$$

onde:

Cx	Custo total do placement.
α_i	Peso da rede i.
W_i^H e W_i^V	Custo das ligações horizontais e verticais, respectivamente.
p	número total de sobreposições.
δ	constante para penalizar sobreposições.

A esperança aqui é a de que, pela escolha apropriada de δ, o algoritmo evolua para configurações sem sobreposições de módulos. A função custo poderia ser facilmente modificada de forma a incorporar uma medida da densidade de fiação nas diversas regiões do plano. A fim de combinar o comprimento da fiação e a medida de congestionamento em uma única função objetivo, o custo do *placement* poderia ser estimado através de um histograma (Figura 4.5). O plano de alocação é dividido em fronteiras (no caso da Figura, as fronteiras são as áreas entre colunas de células adjacentes. Um segundo histograma seria montado para medir a densidade de fiação nas fronteiras horizontais). Os histogramas contém então o número de redes cruzando cada uma das fronteiras. É fácil verificar que o somatório das entradas nos histogramas corresponde ao somatório dos semiperímetros dos retângulos que envolvem cada uma das redes.

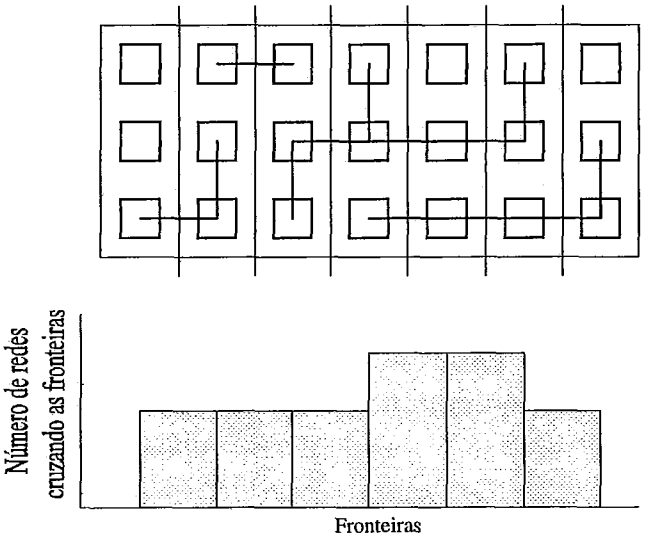


Figura 4.5- Construção do histograma de densidade de fronteiras verticais

A fim de evitar o congestionamento, as fronteiras horizontais e verticais com densidade maior do que um valor preestabelecido seriam penalizadas e estas penalidades seriam incluídas na função custo. A medida do congestionamento no plano é um parâmetro muito importante, e a sua inclusão levaria a *placements* de melhor qualidade, tendo em vista uma probabilidade maior de se obter 100% de sucesso no roteamento automático das conexões. Optamos, no entanto, por não incluí-lo uma vez que seu cálculo seria computacionalmente dispendioso e pouco contribuiria para o objetivo de distinguir a eficiência dos diversos algoritmos propostos.

b) Cálculo da função custo sem sobreposições de módulos:

A modificação introduzida aqui é a eliminação de sobreposições de células antes do cálculo da função custo.

Como visto anteriormente, uma forma de lidar com as restrições do problema - o fato de que duas células não possam sobrepor-se - é gerar configurações sem considerar as restrições e, posteriormente, associar uma penalidade a cada violação. Em outras palavras, um problema com restrições é transformado em um problema sem restrições associando-se uma penalidade a cada violação. Posteriormente, o somatório dessas penalidades é incluído no cálculo da função custo. Portanto, o problema original de otimizar a função $f(x_1, x_2, \ldots, x_n)$ transforma-se na otimização da função:

$$f(x_1, x_2, \ldots, x_n) + p * \delta \tag{4.2}$$

onde (x_i , $i = 1,..,n$) são os módulos a serem alocados, p é o número total de violações e δ é uma constante para penalizar sobreposições.

No entanto, ainda que a fórmula usada para avaliar um cromossoma seja, em geral, bem definida não existe uma metodologia aceita de como combiná-la com as penalidades. Neste sentido, reproduzimos aqui uma importante citação retirada de [Davi87]:

> "If one incorporates a high penalty into the evaluation routine and the domain is one in which production of an individual violating the constraint is likely, one runs the risk of creating a genetic algorithm that spends most of its time evaluating illegal individuals. Further, it can happen that when a legal individual is found, it drives the others out and the population converges on it without finding better individuals, since the likely paths to other legal individuals require the production of illegal individuals as intermediate structures, and the penalties for violating the constraint make it unlikely that such intermediate structures will reproduce. If one imposes moderate penalties, the system may evolve individuals that violate the constraint but are rated better than those that do not because the rest of the evaluation function can be satisfied better by accepting the moderate constraint penalty than by avoiding it".

Em conseqüência, de forma a eliminar sobreposições, pensou-se ser benéfico realinhar as células antes do cálculo da função custo. A favor desta idéia existe ainda o fato de que, de maneira diversa ao algoritmo *Simulated Annealing*, onde apenas dois módulos são movidos por vez e onde é possível calcular a função custo incrementalmente, nos algoritmos genéticos até metade dos módulos pode ser movida em uma única iteração, o que faz com que o comprimento da fiação tenha de ser completamente recalculado a cada *crossover*. Deste modo, não é necessário tolerar sobreposições na medida em que o tempo de processamento necessário a determinar-se as células que se sobrepõem é da ordem do tempo necessário à remoção destas sobreposições.

Uma vez detectada uma sobreposição de células, a função *EliminaOverlapping*( ) procura uma posição livre no entorno do ponto de sobreposição e uma das células é movida para esta posição livre. Na sua implementação a função constrói "quadrados concêntricos" centrados no ponto de sobreposição e de lado progressivamente maior até encontrar uma posição livre. (Figura 4.6).

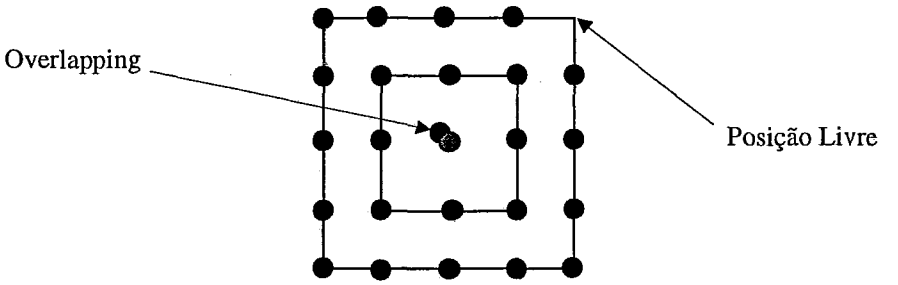


Figura 4.6 - Eliminação de sobreposições.

Os testes efetuados confirmaram a superioridade do cálculo da função custo sem sobreposições sobre a alternativa utilizando-se penalidades.

4.2.3) A função aptidão.

a) A Transformação Linear:

Ao contrário da função custo, a função aptidão fornece uma medida da qualidade de um indivíduo em relação ao restante da população. A medida da aptidão é usada durante o processo reprodutivo para fazer com que os indivíduos mais aptos participem de um número maior de cruzamentos e tenham também maior possibilidade de sobrevivência. Inicialmente, baseada no trabalho de Maresky em [Mare94], foi utilizada a seguinte transformação linear para o cálculo da função aptidão:

$$f = \frac{CxMax - Cx}{CxMax - CxMin} \tag{4.3}$$

Cx é o custo do indivíduo sendo avaliado. *CxMax* e *CxMin* são os custos máximo e mínimo, respectivamente, relativos a todos os indivíduos da população. A função atribui o valor 1 ao melhor indivíduo de uma geração e o valor 0 ao pior indivíduo desta geração.

O algoritmo implementado com a transformação acima não apresentou bons resultados. A função parece ter o problema de privilegiar exageradamente os indivíduos mais aptos. Seja o exemplo de uma população em que apenas um indivíduo tem função custo *CxMin* e todos os demais têm custo *CxMax*. A função aptidão em (4.3) atribuiria $f = 1$ ao indivíduo mais apto e $f = 0$ a todos os demais. Assim, um único indivíduo teria a chance de ser escolhido para reprodução o que eliminaria a saudável diversidade na população.

b) A função tradicional:

A fim de tentar contornar os problemas com os indivíduos super dotados, foram feitos testes com uma função mais tradicional, que leva em conta o valor médio da função custo na população e não os seus valores extremos. A nova função é mostrada na Equação 4.4.

$$f = \frac{\overline{C}x}{Cx} \tag{4.4}$$

Através dos testes, percebeu-se que a função aptidão levando em conta o custo médio da população também parece ter suas limitações. Próximo à convergência, o desvio padrão da função custo é muito pequeno em torno da média. Isto faz com que o valor de f seja bastante semelhante para toda a população o que reduz a pressão para que os melhores indivíduos sobrevivam e tenham maiores chances de reprodução.

c) A função bilinear:

De modo a ter um maior controle sobre a pressão seletiva aplicada à população, em qualquer fase do algoritmo, utilizou-se a seguinte transformação bilinear para o cálculo da função aptidão do indivíduo v:

Se $f[v] < FitAvg$

$$fitness[v] = FitAvg * \left[1.0 - a * \frac{FitAvg - f[v]}{FitAvg - FitMin}\right] \qquad 0.0 < a < 1.0 \tag{4.5}$$

Senão

$$fitness[v] = FitAvg * \left[1.0 + b * \frac{f[v] - FitAvg}{FitMax - FitAvg}\right] \qquad b > 0.0 \tag{4.6}$$

onde:

$f[v]$	É o valor da função aptidão antes da transformação, dado por $1.0/Cx[v]$;
$fitness[v]$	Valor da função aptidão após a transformação bilinear;
$FitAvg$	É o valor médio de $f[v]$;
$FitMax$	É o maior valor de $f[v]$ presente à população;
$FitMin$	É o menor valor de $f[v]$ presente à população;

Através da escolha apropriada dos valores de a e b, é possível controlar o espalhamento dos valores de $fitness[v]$ ao redor da média . Nos testes realizados foram utilizados os valores $a = 0.5$ e $b = 2.0$, o que resulta em $FitMax = 3.0*FitAvg$ e $FitMin = 0.5*FitAvg$, após a transformação. O gráfico para a transformação bilinear pode ser visto na Figura 4.7.

d) *Linear Ranking*

Uma vez que a simples utilização da transformação bilinear não foi capaz de impedir que o algoritmo continuasse convergindo para pontos de mínimo local, foram ainda efetuados testes com uma forma alternativa de atribuir a aptidão aos indivíduos. Uma das causas freqüentes da não convergência em algoritmos genéticos é a presença de indivíduos com aptidão muito alta comparada com o restante da população. Estes "super indivíduos" podem dominar completamente o processo reprodutivo arrastando desta forma o algoritmo para uma solução não ótima. Existem, tradicionalmente, duas formas de lidar-se com este problema:

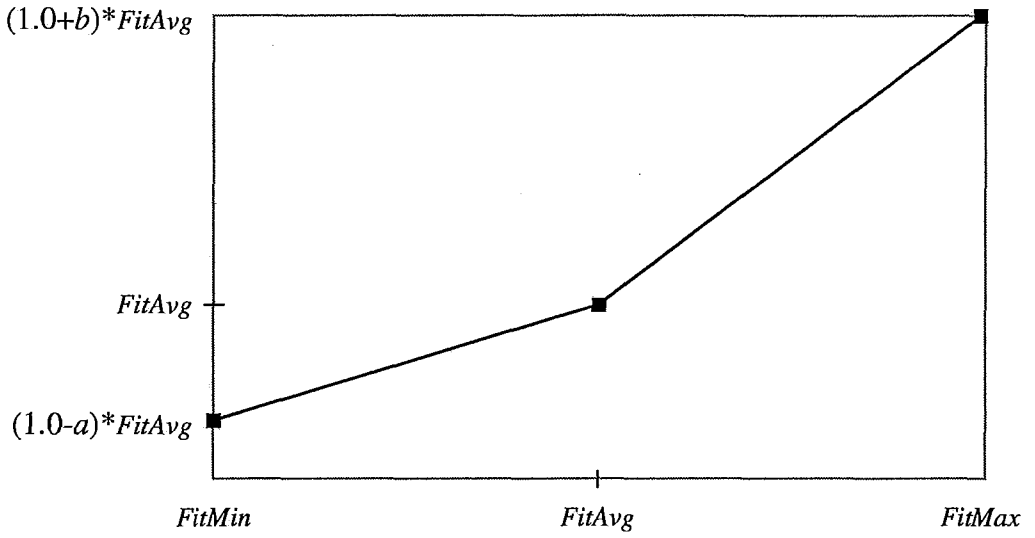


Figura 4.7 - Transformação bilinear

1. O escalamento ou transformação do valor da função aptidão. Por este método, a aptidão *f(v)* de um cromossoma *v* é mapeada em um novo valor *fitness(v)* antes de aplicar-se a seleção proporcional. O objetivo desta técnica é provocar um espalhamento nos valores de *f(v)*, se a população estiver próxima à convergência, e penalizar os "super indivíduos". A transformação bilinear apresentada na seção anterior é um exemplo de um mapeamento possível.
2. O uso de uma distribuição estática das probabilidades de seleção. Uma implementação possível é o chamado *linear ranking* descrito por Baker em [Bake85]. Os indivíduos são ordenados segundo o valor da função aptidão e as probabilidades de seleção são constantes atribuídas em função do *ranking* do indivíduo na população.

Resultados experimentais reportados na literatura ([Bäck91], [Bake85], [Whit89]) apresentam fortes indícios da vantagem do *ranking* sobre os demais métodos. A população é ordenada segundo o valor da função aptidão (o indivíduo mais apto sendo denominado v_0). As probabilidades de seleção são então valores constantes dados por:

$$p(v_i) = \frac{1}{\lambda}\left(\eta_{max} - (\eta_{max} - \eta_{min})\frac{i}{\lambda - 1}\right) \tag{4.7}$$

onde:

i	*ranking* do indivíduo na população;
λ	tamanho da população;
η_{min}	constante proporcional à menor probabilidade de seleção;
η_{max}	constante proporcional à maior probabilidade de seleção;

A partir da relação acima, deduz-se facilmente que

$$\frac{\eta_{min}}{\lambda} \leq p(v_i) \leq \frac{\eta_{max}}{\lambda} \tag{4.8}$$

Baker, em [Bake85], sugere as seguintes relações: $\eta_{min} = 2 - \eta_{max}$ e $1 \leq \eta_{max} \leq 2$. E sugerido ainda o valor $\eta_{max} = 1.1$.

Nos testes realizados o método do *linear ranking* não gerou qualquer resultado digno de referência. Por este motivo, fixamo-nos no método da transformação bilinear.

4.2.4) O método de seleção.

O método de seleção utilizado deve prover a pressão do meio sobre a população, isto é, os indivíduos mais aptos devem ter maior probabilidade de serem escolhidos para reprodução. Utilizamos o chamado método da roleta. A cada indivíduo é atribuído um setor de largura proporcional à aptidão deste indivíduo (Fig. 4.8).

Em seguida "gira-se a roleta", isto é, gera-se um número randômico *r* entre 0 e *FitSum* (soma das aptidões na população). O indivíduo escolhido (*j*) é aquele que atende à relação:

$$\sum_{i=0}^{j-1} f(i) < r \leq \sum_{i=0}^{j} f(i) \qquad (4.9)$$

Indivíduo	*aptidão*
0	0.0
1	0.1
2	0.1
3	0.2
4	0.25
5	0.35

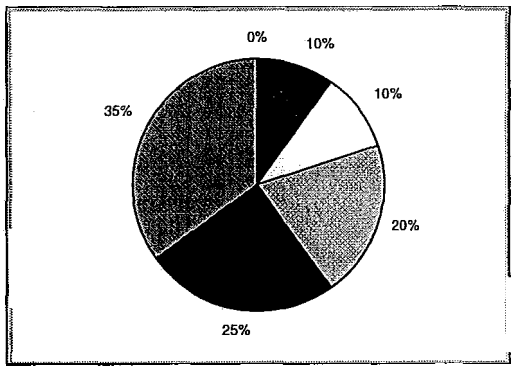


Figura 4.8 - O método da roleta

Baseado no trabalho de Bäck *et al* [Bäck91], foram obtidos bons resultados com uma estratégia elitista de seleção: somente $\mu <$ POPSIZE indivíduos (os mais aptos) participam do processo de seleção. Na implementação efetuada utilizou-se μ = POPSIZE/2 indivíduos.

4.2.5) O operador de *crossover*.

Diversos tipos de operadores de *crossover* foram testados na implementação do modelo de população única. Estes operadores são descritos a seguir:

a) *Crossover* aritmético:

O *crossover* aritmético busca tirar partido da representação direta da posição dos módulos nos genes do cromossoma. As diferentes estratégias de *crossover* empregadas são descritas em [Mich94] e foram implementadas por aquele autor no programa

GENOCOP (GEnetic algorithm for Numerical Optimization for COnstrained Problems). [Mich92]

Crossover aritmético (1 posição):

Se $s_v^t = (v_1, \ldots, v_m)$ e $s_w^t = (w_1, \ldots, w_m)$ devem ser combinados o *offspring* resultante tem a forma:

$$s_v^{t+1} = (v_1, \ldots, v'_k, \ldots, v_m) \tag{4.10}$$

e

$$s_w^{t+1} = (w_1, \ldots, w'_k, \ldots, w_m) \tag{4.11}$$

onde:

$k \in [1,m]$,

$v'_k = a \cdot w_k + (1-a) \cdot v_k$,

$w'_k = a \cdot v_k + (1-a) \cdot w_k$,

$a \in [0,1]$ é um número randômico.

Crossover aritmético (total):

É definido como a combinação linear de dois vetores: se s_v^t e s_w^t devem ser combinados, o *offspring* resultante tem a forma:

$$s_v^{t+1} = a \cdot s_w^t + (1-a) \cdot s_v^t \tag{4.12}$$

e

$$s_w^{t+1} = a \cdot s_v^t + (1-a) \cdot s_w^t \tag{4.13}$$

onde:

$a \in [0,1]$ é um número randômico.

Os operadores aritméticos são de processamento dispendioso e não geraram qualquer resultado notável em comparação com as alternativas convencionais. Alguns outros esquemas (média aritmética entre s_v e s_w, valor randômico intermediário entre o meio do intervalo e o indivíduo mais apto, etc.) foram também testados sem sucesso

b) *Crossover* de 1 ponto:

Se $s_v^t = (v_1, \ldots, v_m)$ e $s_w^t = (w_1, \ldots, w_m)$ devem ser combinados e trocam material genético a partir da posição k, o *offspring* resultante tem a forma (Figura 4.9):

$$s_v^{t+1} = (v_1, \ldots, v_k, w_{k+1}, \ldots, w_m) \tag{4.14}$$

e

$$s_w^{t+1} = (w_1, \ldots, w_k, v_{k+1}, \ldots, v_m) \tag{4.15}$$

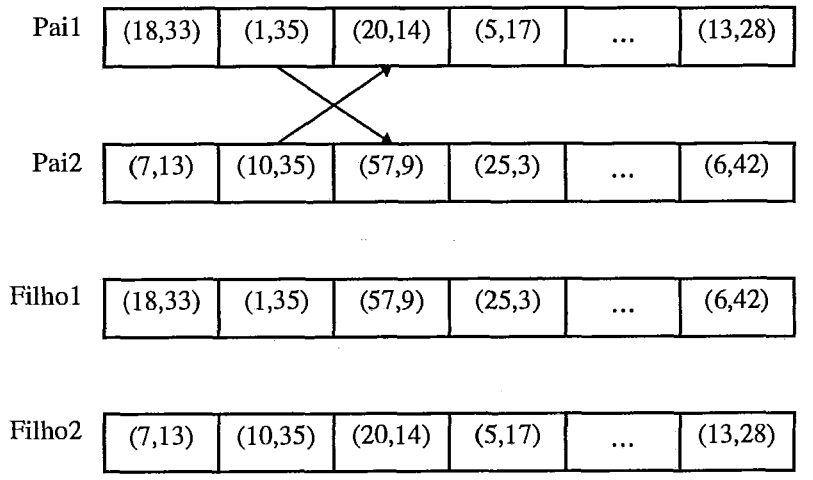


Figura 4.9 - Crossover de 1 ponto.

c) *Crossover* de 2 pontos:

O chamado *crossover* de 1 ponto é inspirado em um processo biológico. Ele apresenta, no entanto, algumas desvantagens. Assuma, por exemplo, que existam dois esquemas promissores na otimização de um dado problema:

$S_1 = (0\ 0\ 1\ *\ *\ *\ *\ *\ *\ *\ 0\ 1)$ e

$S_2 = (*\ *\ *\ *\ 1\ 1\ *\ *\ *\ *\ *\ *)$

Assuma ainda que existam dois cromossomas v_1 e v_2 na população que atendam o padrão definido por S_1 e S_2, respectivamente:

$v_1 = (\mathbf{0\ 0\ 1}\ 0\ 0\ 0\ 1\ 1\ 0\ 1\ 0\ \mathbf{0\ 1})$ e

$v_2 = (1\ 1\ 1\ 0\ \mathbf{1\ 1}\ 0\ 0\ 0\ 1\ 0\ 0\ 0)$

Claramente, através da utilização do *crossover* de 1 ponto, é impossível obter-se um cromossoma que atenda ao esquema:

$S_3 = (0\ 0\ 1\ *\ 1\ 1\ *\ *\ *\ *\ *\ 0\ 1)$

Para resolver este problema, outros tipos de *crossover* (*crossover* de 2 pontos, *crossover* multi-ponto, *crossover* uniforme, etc.) têm sido reportados na literatura.

[Eshe89] [Sisw89] [Scha87] [Spea91]. Em nosso experimentos foi utilizado o *crossover* de 2 pontos:

v_1 = (**0 0 1**|0 0 0 1 1|0 1 0 **0 1**) e

v_2 = (1 1 1|0 **1 1** 0 0|0 1 0 0 0)

Os cromossomas v_1 e v_2 podem agora gerar um filho v_3 que atenda ao esquema em S_3.

É claro que, de maneira similar, existirão esquemas os quais o *crossover* de dois pontos não conseguirá combinar. Uma extensão natural seria então pensar em *crossovers* multi-ponto, onde os filhos seriam formados por vários "pedaços" alternados tomados de cada um dos pais. Observe-se, no entanto, que esta estratégia tem a desvantagem de, potencialmente, destruir blocos promissores presentes em um dos pais.

Talvez, a melhor síntese do problema da escolha de um operador de *crossover* tenha sido apresentada por Michalewicz em [Mich94]:

> "(...)Eshelman [Eshe89] reports on several experiments for various crossover operators. The results indicate that the "loser" is one-point crossover; however, there is no clear winner. A general comment on the above experiments is that each of these crossovers is particularly useful for some classes of problems and quite poor for other problems. This strengthens our idea of problem dependent operators leading us towards evolution programs."

4.2.6) O operador de mutação.

Conforme descrito anteriormente, enquanto foi utilizada a codificação dos cromossomas em *strings* de *bits*, o operador de mutação era implementado pela inversão dos *bits* do cromossoma com probabilidade *pm*. De modo a não introduzir perturbações exageradas no material genético da população, a probabilidade de mutação (*pm*) é um valor baixo, tipicamente da ordem de 1% ou menos. A partir da representação da localização dos módulos diretamente nos genes do cromossoma, outros esquemas de mutação foram experimentados. A descrição destes esquemas é visto a seguir:

a) O operador não uniforme de mutação:

O operador não uniforme de mutação [Mich94] é definido da seguinte forma: se $s_x^t = (x_1, \ldots, x_m)$ é um cromossoma (t é o número da geração) e o elemento x_k foi selecionado para mutação, o resultado é um vetor $s_x^{t+1} = (x_1, \ldots, x'_k, \ldots, x_m)$ onde:

$$x'_k = x_k + \Delta(t, UB - x_k) \qquad \text{// com 50\% de probabilidade} \tag{4.16}$$

ou

$$x'_k = x_k + \Delta(t, x_k - LB) \qquad \text{// com 50\% de probabilidade} \tag{4.17}$$

LB e *UB* são, respectivamente, os limites inferior e superior do domínio da variável x_k. A função $\Delta(t,y)$ retorna um valor no intervalo [0, y] tal que a probabilidade de $\Delta(t,y)$ ser próximo de 0 aumenta a medida em que t cresce. Esta propriedade faz com que o

operador explore o espaço uniformemente no início (quando t é pequeno) e muito localmente em estágios posteriores. Deve-se notar que o parâmetro t tem um papel muito similar ao da temperatura no algoritmo de *Simulated Annealing*. O valor de Δ é expresso pela fórmula:

$$\Delta(t,y) = y*\left(1 - r^{\left(1-\frac{t}{T}\right)^b}\right) \tag{4.18}$$

onde r é um número randômico no intervalo [0..1], T é o número máximo de gerações e b é um parâmetro que determina o grau de dependência de Δ com o número da iteração. As Figuras 4.10 e 4.11 mostram curvas típicas para a expressão $\Delta(t,y)/y$ para $b = 1$ e $b = 5$.

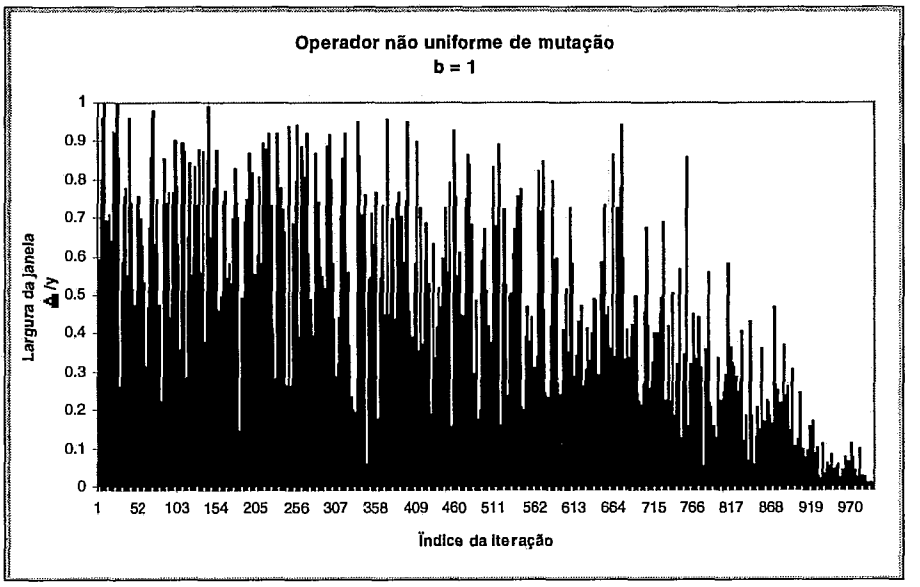


Figura 4.10 - Operador não uniforme de mutação. $b = 1$.

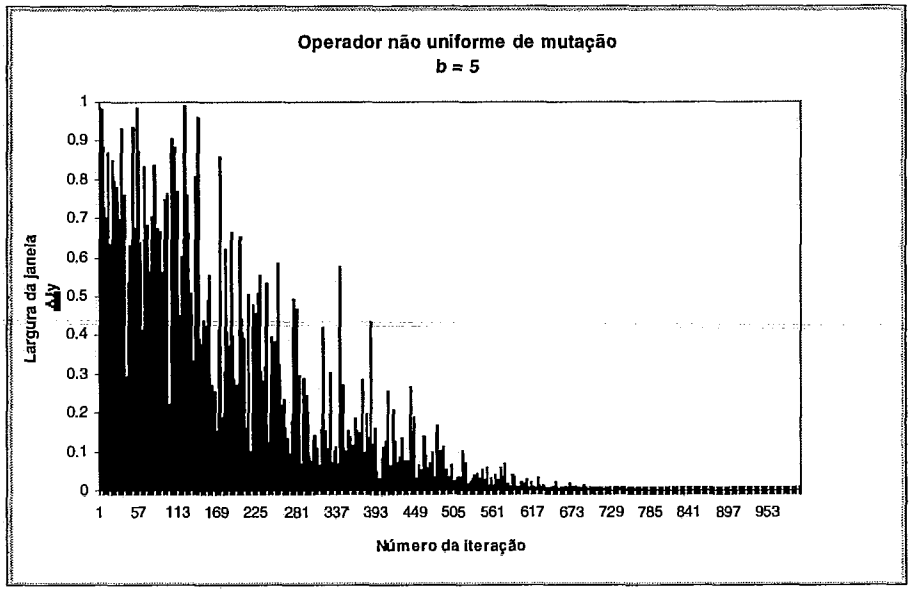


Figura 4.11 - Operador não uniforme de mutação. $b = 5$.

O operador não uniforme de mutação não apresentou bons resultados. Este fato foi atribuído à constatação de que, em regiões próximas à convergência, a operação de mutação se torna rara o que reduz a diversidade de material genético na população, condição essencial para o sucesso do *crossover*

b) Mutação por *swap* de células e *displacements* de coordenadas.

Inspirado nos operadores de vizinhança do algoritmo *Simulated Annealing*, foram implementados operadores de mutação baseado em *swaps* de células e *displacements* de coordenadas. O operador é implementado da seguinte forma: escolhe-se aleatoriamente um módulo no cromossoma e uma posição destino no plano de alocação. Se a posição escolhida estiver ocupada procede-se a um *swap* entre as células ocupando as posições origem e destino do deslocamento. Caso contrário, isto é, se a posição destino estiver livre, a célula é deslocada e passa a ocupar a nova posição no plano.

A vantagem destes procedimentos é a de que as operações elementares de mutação são próximas ao espaço do problema e, portanto, de fácil interpretação. Assim, ao invés de tomados aleatoriamente, os módulos envolvidos nas operações de *swap* e *displacements* poderiam ser determinados por algum algoritmo do tipo *min-cut* ou método das forças. Uma vantagem adicional dos operadores é a de que eles não provocam sobreposições de células.

Foram efetuados ainda testes com número crescente ou decrescente de mutações sem qualquer resultado notável.

4.2.7) Resultados.

Nesta seção são apresentados os resultados para o algoritmo genético, modelo de população única. Foram utilizados os seguintes operadores e parâmetros:

a) Os genes dos cromossomas representam a localização no plano dos módulos do circuito.

b) As sobreposições de módulos são removidas antes do cálculo da função custo.

c) Função aptidão calculada por uma transformação bilinear.

d) Seleção implementada pelo método da roleta. Apenas os indivíduos mais aptos participam da seleção.

e) *Crossover* de 2 pontos.

f) Operador de mutação implementado pelo *swap* de módulos e *displacement* de coordenadas.

A tabela a seguir mostra os resultados obtidos para o algoritmo genético aplicado ao *placement* de um circuito composto por 80 módulos e 30 redes equipotenciais (CIRCUITO_1). Os resultados foram medidos para três tamanhos de população: POPSIZE = 256, POPSIZE = 1024 e POPSIZE = 10000 indivíduos. Para fins de comparação, os resultados obtidos com o *Simulated Annealing* para o mesmo circuito são incluídos na Tabela. A Figura 4.12 mostra a evolução da função custo para os diferentes tamanhos de população.

	Custo	
	Média	Desvio Padrão
POPSIZE = 256	127,33	15,10
POPSIZE = 1024	106,73	8,55
POPSIZE = 10000	94,91	3,05
Annealing	83,67	3,34

Tabela 4.1 - Resultados obtidos para o algoritmo genético - modelo de população única - para 3 tamanhos de população.

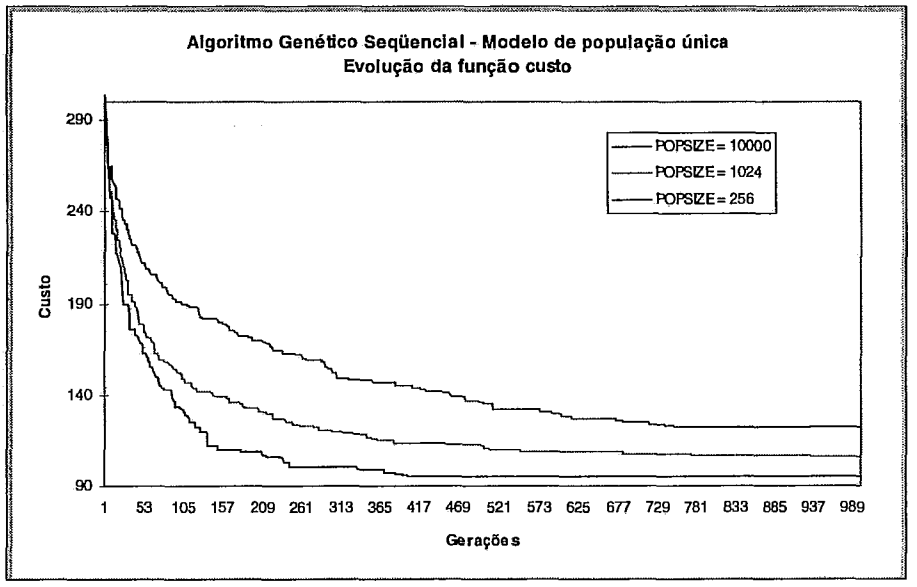


Figura 4.12 - Evolução da função custo para diferentes tamanhos de população.

Alguns fatos tornam-se evidentes pela observação da Tabela 4.1 e da Figura 4.12:

- O algoritmo genético seqüencial falha em atingir os mesmos resultados obtidos pelo *Simulated Annealing*.
- O tamanho da população tem marcada influência sobre o comportamento do algoritmo, mas, mesmo para uma população de 10000 indivíduos, os resultados obtidos são significativamente inferiores àqueles obtidos pelo *annealing*. Deve-se ressaltar que o tempo gasto pelo algoritmo genético para processar 1000 gerações, com POPSIZE = 10000, foi de aproximadamente 12 horas contra pouco mais de 1 minuto gasto pelo *annealing* para o *placement* do mesmo circuito!!!

Um outro parâmetro importante, ainda não mencionado, para o ajuste do algoritmo genético ao problema de *placement* é o tamanho do plano de alocação. Na construção do algoritmo *Simulated Annealing* vista nos Capítulos anteriores, o tamanho do plano de alocação é várias vezes maior do que a área necessária ao *placement* do circuito de teste. O algoritmo de *annealing* é capaz de formar a solução do problema em algum ponto deste "super plano" independente do tamanho do mesmo. Para o algoritmo genético, no entanto, pela própria natureza dos operadores utilizados (o operador de *crossover*, por exemplo, tem muita dificuldade em tomar dois indivíduos com genes

relacionados a regiões distintas do plano e, a partir deles, gerar *offsprings* superiores a seus pais), é necessário trabalhar com um plano de dimensões ligeiramente superiores à área mínima necessária à alocação do circuito. Para comprovar este fato, o algoritmo genético foi aplicado ao circuito anterior com POPSIZE = 1024, 1000 gerações e um plano de dimensões 64 x 64, muito maior do que a área necessária à alocação dos 80 módulos. Os resultados obtidos para o comprimento aproximado da fiação foram: Custo Médio = 406,40 e Desvio Padrão = 30,83 o que comprova a importância de trabalhar-se com planos de dimensões reduzidas.

4.2.8) Conclusões.

O algoritmo genético, na sua forma clássica, falhou em produzir resultados de qualidade satisfatória em tempos de processamento aceitáveis. Por este motivo, no desenvolvimento que se seguiu, foram realizados testes com algumas técnicas não convencionais em algoritmos genéticos. Além disso, com o objetivo de tentar mover a solução para regiões mais promissoras do espaço de estados, a pesquisa foi orientada no sentido de introduzir no algoritmo proposto conhecimentos específicos sobre o problema sendo atacado (o problema de *placement*). Estes tópicos serão vistos nas seções seguintes.

4.3) Organização da população: O modelo paralelo.

Muitas das idéias aqui discutidas foram inicialmente apresentadas em [Balu92].

O algoritmo descrito nesta seção, ainda que serial, busca compartilhar os benefícios introduzidos por algoritmos genéticos paralelos. Para isso, foram utilizadas as premissas básicas extraídas da teoria do "equilíbrio pontual" (*punctuated equilibria*) mas buscou-se, através de um maior grau de distribuição e da interseção de populações, evitar os problemas associados com a introdução súbita de material genético novo.

4.3.1) O modelo paralelo convencional.

Algoritmos genéticos paralelos (pGA) representam mais do que um meio de acelerar o processamento da versão serial. Ainda que existam muitas formas de acelerar o processamento usando uma máquina paralela (por exemplo, as operações de *crossover*, mutação e avaliação poderiam ser realizadas em paralelo) um pGA pode ser executado em uma máquina serial. Isto se deve ao fato de que a paralelização do algoritmo - entendida aqui como a existência de várias subpopulações, avaliadas independentemente - proporciona uma solução de melhor qualidade do que a versão serial. Um pGA se baseia na teoria do "equilíbrio pontual" descrita em [Coho88]:

> "Punctuated Equilibria is based upon two principles: allopatric speciation and stasis. Allopatric speciation involves the rapid evolution of new species after a small set of members of species, peripheral isolates, becomes segregated into a new environment. Stasis, or stability, of a species, is simply the notion of lack of change. It implies that after equilibria is reached in an environment, there is very little drift away from the genetic composition of

> species. (...) Punctuated Equilibria stresses that a powerful method for generating new species is to thrust on old species into a new environment, where change is beneficial and rewarded. For this reason we should expect a genetic algorithm approach based upon punctuated equilibria to perform better than the typical single environment scheme."

A implicação desta teoria na estrutura de um algoritmo genético é que, dada uma população única, esta convergirá para uma situação de equilíbrio (não necessariamente o mínimo global). Os cromossomas filhos gerados a partir deste ponto serão todos muito parecidos entre si e parecidos também com os pais, tornando o operador de *crossover* muito ineficiente. Uma forma de resolver este problema é criar subpopulações distintas. Cada subpopulação trabalha sobre os seus cromossomas independentemente de todas as outras subpopulações. A função aptidão, usada para determinar a probabilidade de seleção de um cromossoma, é relativa apenas aos membros de uma mesma subpopulação. O processamento independente das subpopulações deve resultar, possivelmente, em indivíduos diferentes nas subpopulações. A fim de continuar a evolução depois que as subpopulações tenham convergido, alguns membros podem ser trocados entre as subpopulações na esperança de que, ao longo das gerações, o novo esquema seja incorporado à população local.

Ainda que a súbita introdução de um "novo" material genético seja um aspecto importante da teoria do equilíbrio pontual, ele nem sempre é efetivo. Os cromossomas podem ser trocados entre populações que estejam em diferentes estágios do processamento. Assim, uma população que já tenha convergido para um mínimo local, pode exportar indivíduos para uma outra população onde a função custo médio ainda é alta, mas cuja evolução vinha sendo promissora no sentido do mínimo global. O material genético introduzido pode, rapidamente, dominar a população "hospedeira" e atraí-la para o mesmo mínimo local. O caminho inverso também é verdadeiro: um indivíduo de função custo mais alta pode, rapidamente, ser dominado pela população hospedeira sem ter tempo de espalhar o seu, possivelmente benéfico, material genético.

4.3.2) Populações superpostas. O algoritmo paralelo modificado (mpGA).

Deve-se novamente ressaltar que paralelismo aqui é entendido não como a presença de vários processadores mas sim como a existência de populações separadas que são avaliadas independentemente.

De modo a lidar com os problemas associados com a troca de indivíduos entre subpopulações, Baluja, em seu artigo *"A Massively Distributed Parallel Genetic Algorithm"* [Balu92], propôs uma modificação na arquitetura básica do algoritmo paralelo. O algoritmo proposto se baseia na premissa de que, reduzindo a rigidez das fronteiras entre as subpopulações, deve ser possível superar os problemas associados com a súbita introdução de material genético novo. Uma forma de visualizar esta modificação é conceituar as populações como superpostas, isto é, alguns indivíduos tomam parte, simultaneamente, em mais de uma população (Fig. 4.13). Esta estrutura permite a transferência gradual de material genético entre as populações.

A motivação desta organização de populações é a esperança de que subpopulações separadas por uma grande distância (relativa a N) evoluam para cadeias genéticas

significativamente diferentes, de maneira similar ao algoritmo genético paralelo convencional. Alem disso, é de se esperar que toda a população se beneficie pelo espalhamento suave de material genético obtido em uma das subpopulações e que cause um refinamento da função custo. É claro que, subpopulações próximas terão maior influência umas sobre as outras do que aquelas separadas por uma grande distância.

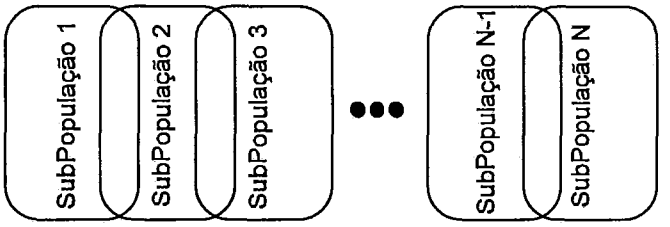


Figura 4.13 - Sobreposição de populações em um mpGA.

Em princípio, o risco do mpGA evoluir para um mínimo local é maior do que nos algoritmos paralelos convencionais. Isto se deve, em primeiro lugar, ao maior grau de troca de material genético entre as subpopulações. Além disso, no algoritmo implementado, existem significativamente menos indivíduos por população no mpGA do que nos algoritmos paralelos convencionais. De forma a resolver este problema o mpGA baseia-se no tamanho da população global, muito maior do que a usada nos algoritmos convencionais.

4.3.3) Distribuição da população e processo de reprodução.

a) Estratégias de Seleção e *Tournament.*

Chamamos de *Tournament* ao procedimento usado para determinar onde inserir os filhos, provenientes dos cruzamentos nas subpopulações.

A população global de indivíduos assume a forma de um toróide, representado no plano por um vetor de duas dimensões (Fig. 4.14). Cada posição da grade armazena dois indivíduos. As subpopulações (ou *demes*) são formadas por 10 cromossomas: os dois cromossomas no centro do *deme* e 1 cromossoma de cada um dos vizinhos ao Norte, Nordeste, Leste, Sudeste, Sul, Sudoeste, Oeste e Noroeste. Cada um dos cromossomas provenientes dos vizinhos é escolhido aleatoriamente entre os dois disponíveis.

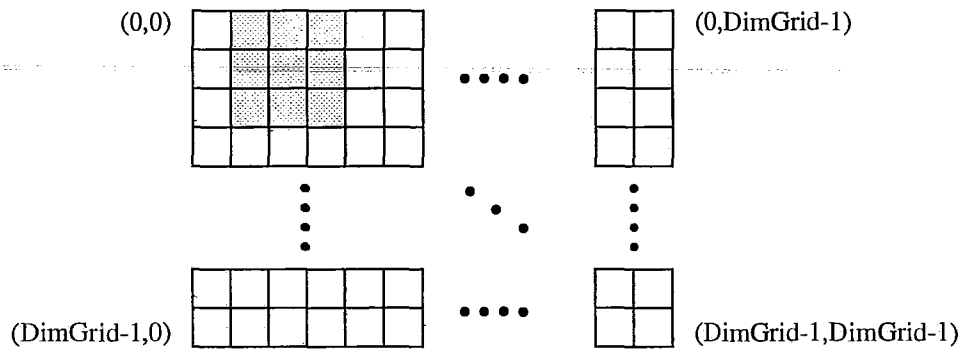


Figura 4.14 - Arquitetura das subpopulações e um *deme.*

Em cada *deme* dois indivíduos são selecionados para reprodução, a cada geração. Os dois cromossomas selecionados geram dois "filhos" (por *crossover* e mutação) que vão substituir os indivíduos no centro do *deme*. A função aptidão é relativa somente aos 10 cromossomas de cada *deme*.

A estratégia empregada tem a vantagem de reduzir a comunicação entre processadores em uma implementação paralela. Cada posição da grade exporta indivíduos para completar a população dos *demes* centrados nas posições vizinhas, mas não existe o fluxo de dados em sentido inverso. Teoricamente, em uma máquina SIMD maciçamente paralela, seria possível atribuir um processador a cada posição da grade e substituir toda a população em um único passo, efetuado de forma síncrona por todos os processadores. A estratégia é igualmente apropriada para máquinas paralelas de memória compartilhada uma vez que, devido ao procedimento de *tournament* utilizado, não existem problemas de coerência de dados nas regiões de fronteira entre os processadores.

Ainda que de adaptação não tão evidente, a estratégia deu margem também a uma proposta de implementação paralela, utilizando trocas de mensagens. Esta proposta será vista no Capítulo 5.

b) Elitismo.

Elitismo é uma técnica comumente empregada em algoritmos genéticos para assegurar que o progresso obtido em uma geração não seja perdido devido a escolhas aleatórias. Uma vez que os indivíduos gerados em cada subpopulação substituem incondicionalmente os indivíduos no centro do *deme*, não existe a garantia de que os melhores indivíduos em uma geração sobrevivam até a geração seguinte.

Para resolver este problema implementou-se então uma forma de seleção elitista. A estrutura da grade foi "aumentada" e cada posição passa a armazenar também o melhor indivíduo presente à subpopulação na última vez em que o *deme* foi selecionado. Isto não implica em que o melhor cromossoma na geração *k-1* será selecionado para reprodução na geração *k*, mas ele estará presente na população de 10 indivíduos os quais são candidatos à recombinação. Uma vez que o tamanho da população é limitado em 10, o melhor cromossoma na geração anterior substitui o pior da geração atual, antes do procedimento de seleção.

O inconveniente óbvio desta estratégia é o de que ela implica em maiores requisitos de memória necessários à implementação do algoritmo.

c) Mutação.

Todo indivíduo recém gerado passa, probabilisticamente, por uma operação de mutação.

Deve-se recordar que o operador de mutação provoca *swaps* (2 cromossomas envolvidos) ou deslocamentos (1 cromossoma envolvido) de módulos. Uma vez que o tamanho do plano de alocação é cerca de 25% maior do que o número de módulos, o número médio de cromossomas (n_m) afetados a cada operação de mutação é dado por:

$$n_m = (0.2*1 + 0.8*2) \text{ cromossomas} \qquad (4.19)$$

$$n_m = 1{,}8 \text{ cromossomas} \tag{4.20}$$

A probabilidade de mutação (*Pmutation*) de um único cromossoma é então dada por:

$$Pmutation = \frac{n_m}{nmodulos} \tag{4.21}$$

Para o caso do circuito de teste (*nmodulos* = 80), a expressão acima resultaria em *Pmutation* = 2,25%, um valor (segundo senso comum na comunidade de algoritmos genéticos) muito alto. Utilizamos então uma probabilidade de ajuste (*Pajuste*) obtida através de uma função randômica *flip*( ) a qual retorna 1 com probabilidade *Pajuste*. Se o valor retornado por *flip*( ) for 1, procede-se a mutação. Caso contrário o cromossoma permanece inalterado. O valor de *Pajuste* tem de ser determinado, em função do número de módulos, para cada circuito otimizado. Para o circuito de teste foi utilizado *Pajuste* = 0,444. Com estes valores *Pmutation* é então dada por:

$$Pmutation = Pajuste * \frac{n_m}{nmodulos} \tag{4.22}$$

$$Pmutation = 0.444 * 1{,}8/80 \tag{4.23}$$

$$Pmutation \cong 1\% \tag{4.24}$$

4.3.4) Pseudo código:

```
/* Melhor_k   ⇒  Indivíduo mais fitted na geração atual                    */
/* Melhor_k-1 ⇒  Indivíduo mais fitted na geração anterior                 */
/* Pior_k     ⇒  Indivíduo menos fitted na geração atual                   */

Genese ( );                          /* Produz uma população inicial de POPSIZE indivíduos.*/
                                     /* As células nos indivíduos não se sobrepõem         */
Repete por T gerações          {
    EscolheDeme( );                  /* Aleatoriamente escolhe um deme na população        */
    Pior_k = Melhor_k-1;             /* Esquema elitista. Substitui o pior indivíduo desta */
                                     /* geração pelo melhor da geração anterior            */
    Evaluation( );                   /* Avalia o fitness dos indivíduos no deme.           */
                                     /* Transformação bilinear                             */
    Pai1 = Seleção( );               /* Escolhe o primeiro pai. Método da roleta.          */
    Pai2 = Seleção( );               /* Escolhe o segundo pai                              */
    Gera 2 filhos segundo o esquema: {
        Crossover(Pai1, Pai2);       /* Gera um filho.                                     */
        EliminaOverlapping( );       /* Elimina sobreposições.                             */
        Mutação( )                   /* Probabilisticamente aplica mutação ao              */
                                     /* indivíduo recém gerado.                            */
    }
    Tournament( );                   /* Os 2 indivíduos recém gerados substituem os        */
                                     /* indivíduos no centro do deme.                      */
}
Imprime resultados.
```

4.3.5) Resultados:

Os resultados são apresentados nas Tabelas 4.2 e 4.3 para um problema pequeno (CIRCUITO_1) e um problema médio (CIRCUITO_2, composto por 100 módulos e 300 redes equipotenciais) respectivamente. A fim de facilitar a comparação, os resultados obtidos com o *annealing* para os mesmos circuitos foram também incluídos nas tabelas. Para cada tamanho de população foram efetuadas um mínimo de 15 execuções do algoritmo. Em cada um dos testes permitiu-se que o algoritmo executasse por um número arbitrariamente grande de iterações de modo a que a convergência fosse atingida.

	POPSIZE 256	POPSIZE 625	POPSIZE 1024	POPSIZE 2500	POPSIZE 4096	POPSIZE 10000	ANNEAL
Média	97,50	95,00	92,00	88,90	85,90	83,30	83,67
Desvio Padrão	7,53	2,97	2,76	2,74	2,88	2,33	3,34
Máximo	113	101	97	93	91	86	93
Mínimo	88	90	89	84	81	78	77
Tempo (s)	47,48	115,33	189,15	562,67	805,03	2187,90	26,12

Tabela 4.2 - Resultados obtidos para o CIRCUITO_1.

	POPSIZE 256	POPSIZE 625	POPSIZE 1024	POPSIZE 2500	POPSIZE 4096	POPSIZE 10000	ANNEAL
Média (C_x)	2117,40	2083,00	2056,40	2046,40	2035,30	2010,50	1915,00
Desvio Padrão	51,65	37,72	25,41	26,85	17,95	15,62	28,00
Máximo	2243	2155	2102	2086	2063	2039	1884
Mínimo	2048	1996	2020	1983	1997	1987	1963
Tempo (s)	267,95	646,86	1074,10	2717,47	4225,85	10391,55	89,94

Tabela 4.3 - Resultados obtidos para o CIRCUITO_2.

Os resultados são apresentados na forma de gráficos nas Figuras 4.15 e 4.16. Os valores assinalados referem-se à função custo (C_x) e aos valores de ($C_x + \sigma$) e ($C_x - \sigma$). A reta horizontal corresponde ao valor obtido pelo algoritmo *Simulated Annealing*.

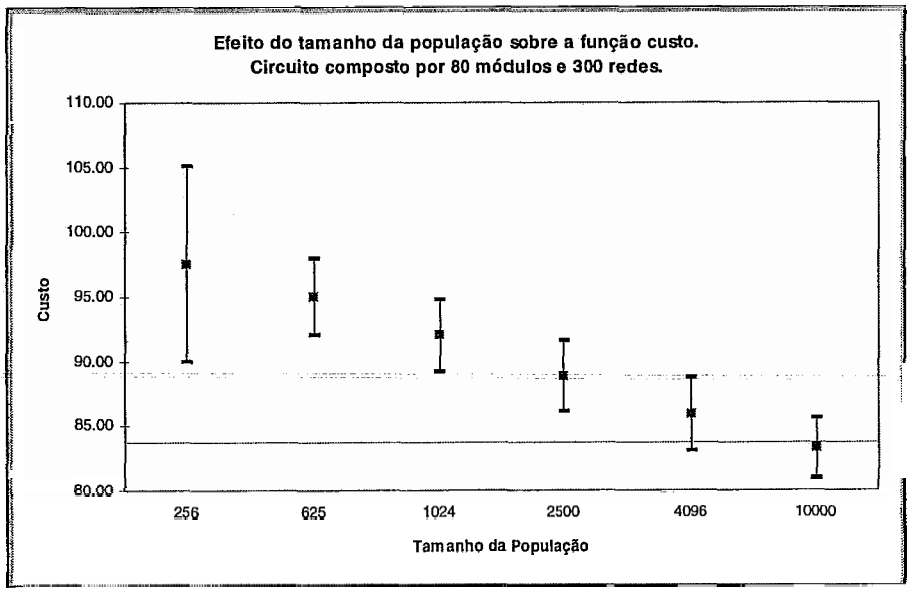


Figura 4.15 - O modelo paralelo. Resultados obtidos para o CIRCUITO_1.

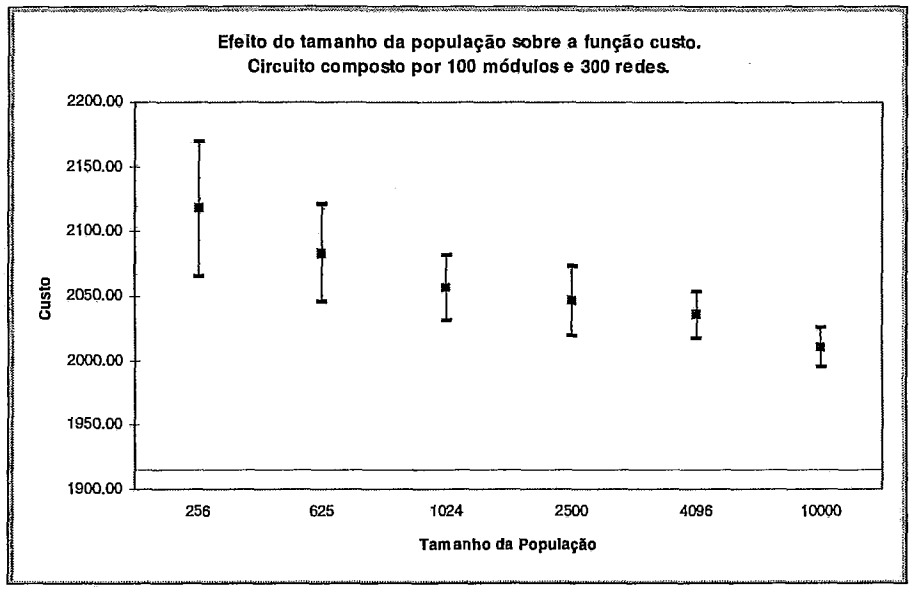


Figura 4.16 - O modelo paralelo. Resultados obtidos para o CIRCUITO_2.

A partir da observação dos gráficos e tabelas acima conclui-se pela efetividade do modelo paralelo quando comparado ao modelo de população única (Vide Tabela 4.1). Para tamanhos de populações iguais, os resultados obtidos foram marcadamente superiores e obtidos em tempos de processamento significativamente menores. Como era de se esperar, devido ao tamanho reduzido das subpopulações (10 indivíduos), os melhores resultados foram alcançados com um número de indivíduos muito superior àqueles usados por algoritmos genéticos tradicionais.

Observa-se no entanto que, em comparação com o algoritmo de *annealing*, os resultados foram inferiores ou obtidos em tempos de processamentos proibitivamente longos. Desta forma era preciso continuar buscando técnicas e operadores que permitissem obter resultados de melhor qualidade e/ou em tempos de processamento menores.

É interessante estudar o efeito da técnica de elitismo (implementado na forma de um terceiro indivíduo por posição da grade) sobre o desempenho do algoritmo. Considere a Tabela 4.4 onde os resultados para o CIRCUITO_2 e uma população de 1024 indivíduos são comparados com e sem a utilização de elitismo.

	Com Elitismo	Sem Elitismo
Média (C_i)	2056,40	2253,20
Desvio Padrão	25,41	35,22
Máximo	2102	2307
Mínimo	2020	2200

Tabela 4.4 - Efeito da técnica de Elitismo.

A observação da tabela comprova de forma inequívoca o benefício da utilização de elitismo.

Deve-se ressaltar que, uma vez que as subpopulações são sempre formadas por 10 indivíduos, o tempo de processamento de uma única geração não é afetado pelo tamanho da população. Este parâmetro tem impacto apenas sobre os requisitos de

memória e sobre o número de iterações necessárias a atingir-se a convergência. Estes fatos, aliados à independência do processamento das subpopulações, apontam para a conveniência de uma solução paralela. Esta alternativa é discutida no Capítulo 5.

4.4) Algumas técnicas não convencionais.

Esta seção descreve a implementação de algumas técnicas não convencionais objetivando a obtenção de resultados de melhor qualidade e/ou em tempos de processamento mais curtos.

4.4.1) População de Tamanho Variável:

A influência do tamanho da população sobre o desempenho dos algoritmos genéticos tem sido estudada por muitos pesquisadores, segundo diferentes perspectivas. Grefenstette [Gref86] utilizou um "meta" algoritmo genético para controlar os parâmetros de outro GA (incluindo o tamanho da população e o método de seleção). Goldberg [Gold85, Gold89] desenvolveu um estudo teórico sobre o tamanho ótimo da população. Experimentos adicionais com o tamanho da população foram reportados em [Guch89] e [Mott91].

Nesta seção é apresentado um algoritmo genético com tamanho de população variável. Este algoritmo não usa qualquer dos métodos de seleção apresentados anteriormente. Ao invés disto, ele trabalha com o conceito de "idade" de um cromossoma, que é equivalente ao número de gerações que um cromossoma permanece "vivo". Assim, a idade de um cromossoma substitui o conceito de seleção e, uma vez que ela depende da aptidão do indivíduo, influencia o tamanho da população nas diferentes fases do processo. A esperança aqui é desenvolver um esquema adaptativo em que o tamanho da população se ajuste ao valor ótimo necessário às diferentes etapas do algoritmo.

Michalewicz, em [Mich94], descreve a implementação de um GA com população variável. Ali, é dito pelo autor:

> "It seems also that such approach is more "natural" than any selection mechanism considered earlier: after all, the aging process is well-known in all natural environments. (...) It seems reasonable to assume that at different stages of the evolution process different operators would have different significance and the system should be allowed to self-tune their frequencies and scope. The same should be true for population sizes: at different stages of the evolution process different sizes of the population may be "optimal", thus it is important to experiment with some heuristic rules to tune the size of the population to the current stage of the search."

Devido à dificuldade em organizar-se uma população de tamanho variável sob a forma de uma grade, foi utilizado aqui o modelo de população centralizada ou única.

- Pseudo-Código:

```
/* t          ⇒   Geração corrente                                          */
/* ρ(t)       ⇒   Taxa de reprodução                                        */
/* PopSize(t) ⇒   Tamanho corrente da população                             */
/* Nm(t)      ⇒   Número de mutações por geração                            */

Genese( );              /* Gera a população inicial com POPINI indivíduos.  */
                        /* As células nos indivíduos não se sobrepõem.      */
Evaluation( );          /* Atribui os tempos de vida dos elementos da população */

Repete por T gerações   {
        Gera [ρ(t) * PopSize(t)] novos indivíduos por crossover;
                        /* Os indivíduos têm chances iguais de serem selecionados. */
        Retira da população os indivíduos que já viveram suas vidas;
        Mutação( );     /* Swap de coordenadas                              */
        Evaluation( );  /* Atribui os tempos de vida aos novos indivíduos da população. */
}
Imprime resultados.
```

- Descrição do algoritmo:

O algoritmo, na geração t, processa uma população composta por $PopSize(t)$ cromossomas. Durante a fase de reprodução $[\rho(t) * PopSize(t)]$ novos indivíduos são gerados. Cada cromossoma na população pode ser escolhido para reprodução com igual probabilidade (independente do valor de sua função aptidão). Uma vez que não existe o procedimento de seleção, introduz-se o conceito de "idade" e "tempo de vida" de um cromossoma. Aos indivíduos mais aptos é atribuído um tempo de vida maior e, portanto, tais cromossomas terão maior chance de, durante a sua vida (número de gerações durante as quais o cromossoma permanece na população), serem escolhidos para reprodução.

Um indivíduo "morre" (ou é retirado da população) quando a sua "idade" ultrapassa o seu "tempo de vida". Assim, após uma única iteração, o tamanho da população é dado por:

$$PopSize(t+1) = PopSize(t) + \rho(t)*PopSize(t) - D(t) \qquad (4.25)$$

onde $D(t)$ é o número de indivíduos retirados da população na geração t.

- Detalhes de Implementação e Resultados:

a) Atribuição dos tempos de vida.

Indivíduos com aptidão acima da média devem ser distinguidos com um tempo de vida maior. De modo a poder controlar o espalhamento da função "tempo de vida" em torno de um valor médio, foi utilizada uma transformação bilinear (Fig. 4.17) com tempo de vida mínimo ($MinT$) igual a 1 geração e tempo de vida máximo ($MaxT$) igual a 7 gerações. Ao indivíduo com aptidão igual à média da população foi atribuído um tempo de vida de 4 gerações. Estes valores foram adotados a partir da implementação descrita em [Mich94].

b) Taxa de reprodução.

Inicialmente, a exemplo do trabalho descrito em [Mich94], foi utilizada uma taxa de reprodução (ρ) fixa, igual a 0,4. Ali, Michalewicz descreve a implementação de um algoritmo genético com população variável aplicado à maximização de 4 funções:

$$\text{G1:} \quad -x\,\text{sen}(10\pi x)+1 \qquad -2.0 \le x \le 1.0 \qquad (4.26)$$

$$\text{G2:} \quad \textbf{integer}(8x)/8 \qquad 0.0 \le x \le 1.0 \qquad (4.27)$$

$$\text{G3:} \quad x \cdot \text{sgn}(x) \qquad -1.0 \le x \le 2.0 \qquad (4.28)$$

$$\text{G4:} \quad 0.5+\frac{\text{sen}^2\sqrt{x^2+y^2}-0.5}{\left(1+0.001\left(x^2+y^2\right)\right)^2} \qquad -100 \le x,y \le 100 \qquad (4.29)$$

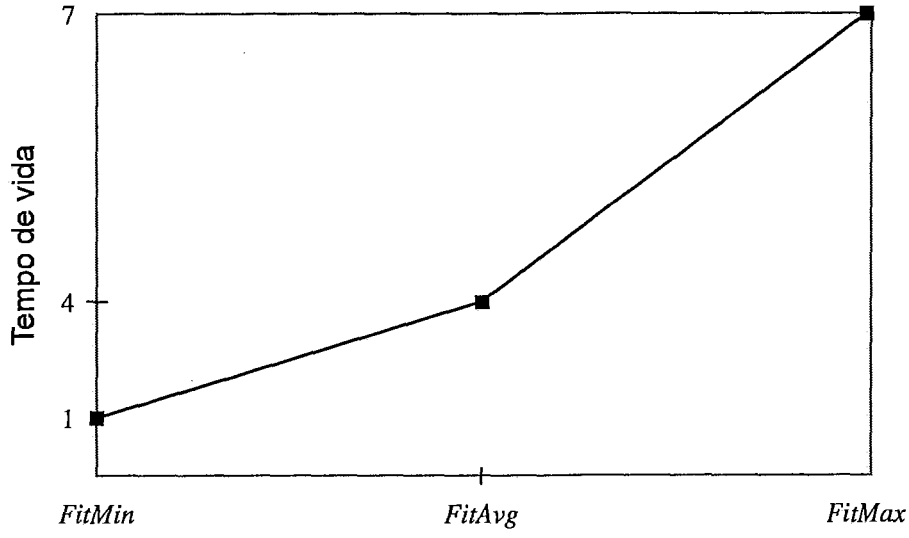


Fig. 4.17 - Transformação bilinear utilizada para atribuir os tempos de vida aos indivíduos da população.

Os resultados reportados são excelentes. O algoritmo genético implementado converge para o máximo das funções e o tamanho da população, efetivamente, parece ajustar-se às diferentes fases do algoritmo. A Figura 4.18 mostra os resultados obtidos por aquele autor para a otimização da função G4. A curva pontilhada representa o valor da função e a curva sólida o tamanho da população.

Infelizmente, os resultados aqui obtidos não foram tão animadores. Para uma taxa de reprodução fixa o tamanho da população parece variar aleatoriamente, sem guardar qualquer relação com a fase do algoritmo e, ao final do processo, parece sempre "explodir". Além disso, o resultado final jamais aproximou-se daquele obtido com o *Simulated Annealing*.

As Figuras 4.19, 4.20 e 4.21 mostram o comportamento do algoritmo, em 3 execuções diferentes, aplicado ao *placement* do CIRCUITO_1. A fim de não exceder a capacidade instalada de memória da máquina de teste, o tamanho máximo da população foi limitado em 460 indivíduos. O tamanho inicial da população é de 20 indivíduos.

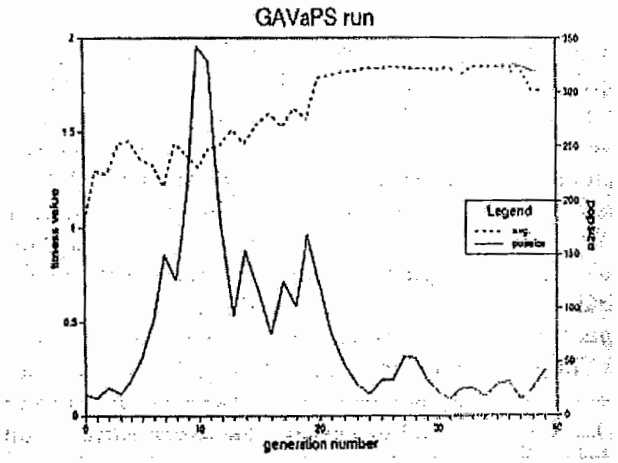


Figura 4.18 - Resultados reportados em [Mich94] para a otimização da função G4

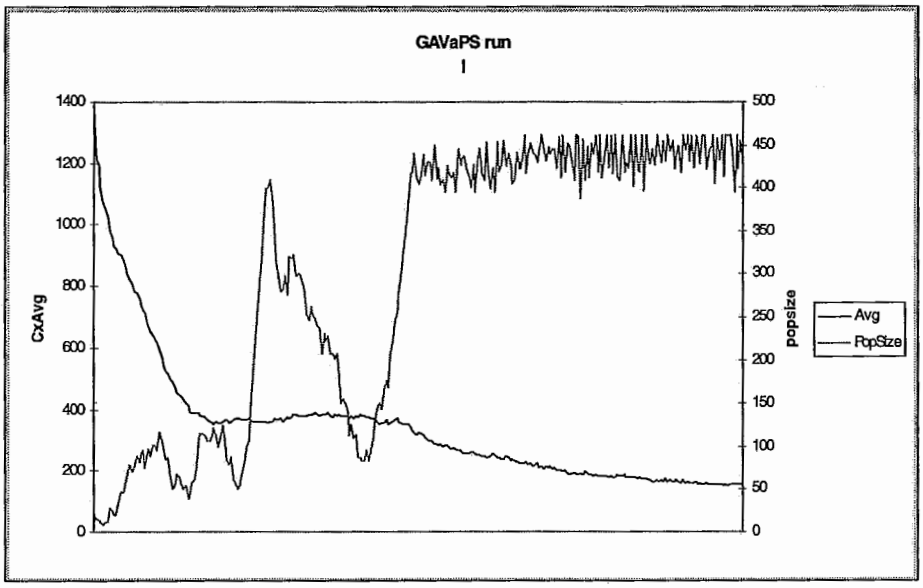


Figura 4.19 - Tamanho da população e custo da fiação para uma rodada do GA população variável.

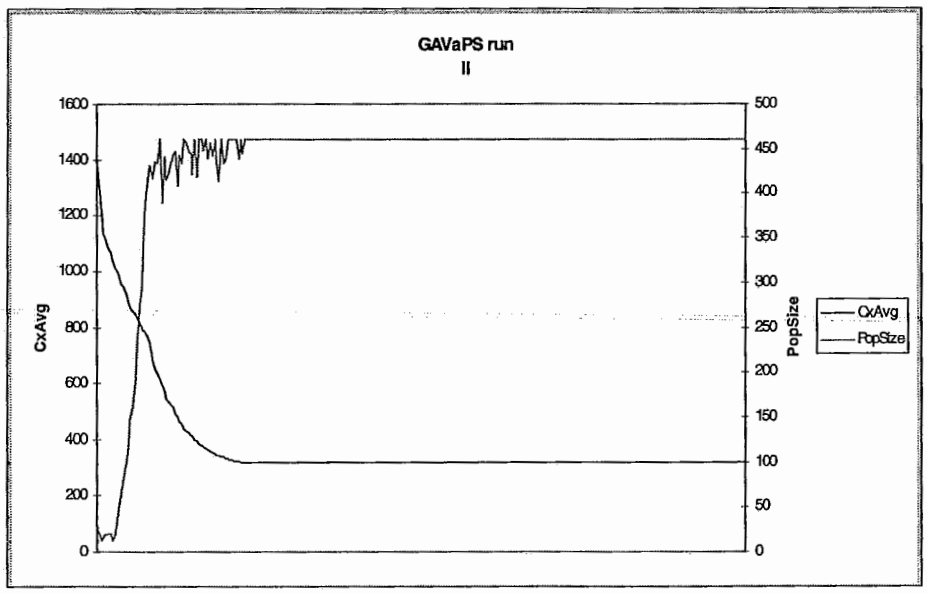


Figura 4.20 - Tamanho da população e custo da fiação para uma rodada do GA população variável.

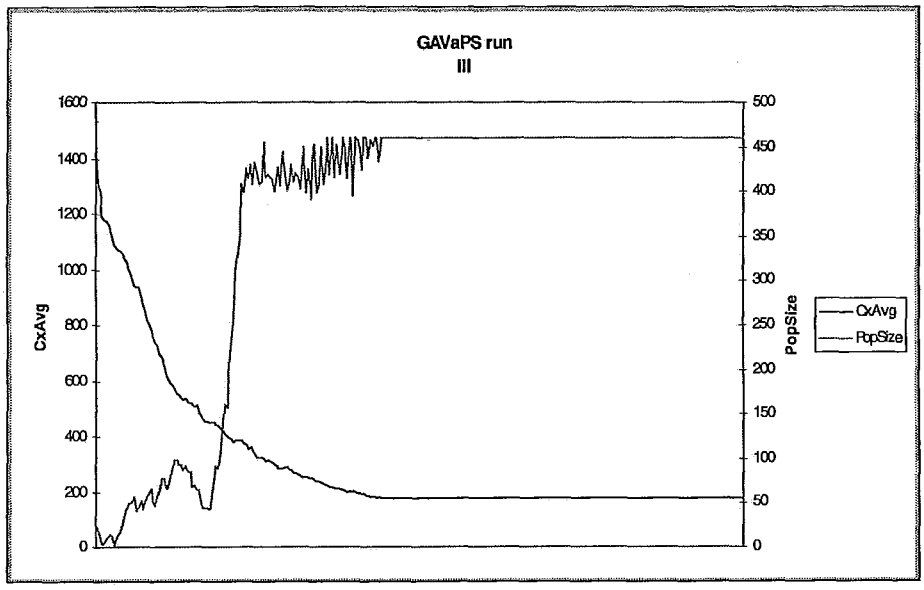


Figura 4.21 - Tamanho da população e custo da fiação para uma rodada do GA população variável.

A partir da observação de que, em algum instante da execução do algoritmo, a população parece sempre passar por um ponto de forte derivada positiva (quando então ela converge para o tamanho máximo permitido), tentou-se introduzir algum controle externo sobre o tamanho da população. Este controle é baseado na derivada do tamanho da população. Desta forma, ao verificar-se a aceleração na taxa de crescimento da população, diminui-se a taxa de reprodução (ρ) na esperança de evitar-se a "explosão" da população. De maneira similar, ao verificar-se a desaceleração na taxa de crescimento, aumenta-se ρ visando com isso manter o tamanho da população dentro de limites razoáveis.

A fim de calcular a derivada da população foi utilizado um polinômio de interpolação $f(t)$ passando pelo ponto atual e pelos 5 pontos imediatamente anteriores na curva. O valor da função derivada é então dado por:

$$f'(t) = 2.283333 * f(t) - 5 * f(t-1) + 5 * f(t-2) - 3.333333 * f(t-3) + \\ +1.25 * f(t-4) - 0.2 * f(t-5) \qquad (4.30)$$

Visando verificar a correção da fórmula, a mesma foi aplicada ao cálculo da derivada da função seno. O resultado pode ser visto na Figura 4.22.

A função $f'(t)$ foi então aplicada ao cálculo da derivada do número de indivíduos na população (Fig. 4.23). Pela observação do gráfico, percebe-se que a função tamanho da população é por demais "nervosa" para que a sua derivada forneça alguma informação útil. Tentou-se então suavizar o comportamento da função derivada pela aplicação de $f'(t)$ não mais ao tamanho da população, mas a uma função amortecida dada por:

$$g[0] = PopSize[0]; \qquad (4.31)$$

$$g[1] = 0.01 * PopSize[1] + 0.99 * g[0]; \qquad (4.32)$$

.
.
.

$$g[t] = 0.01 * PopSize[t] + 0.99 * g[t\text{-}1]; \qquad (4.33)$$

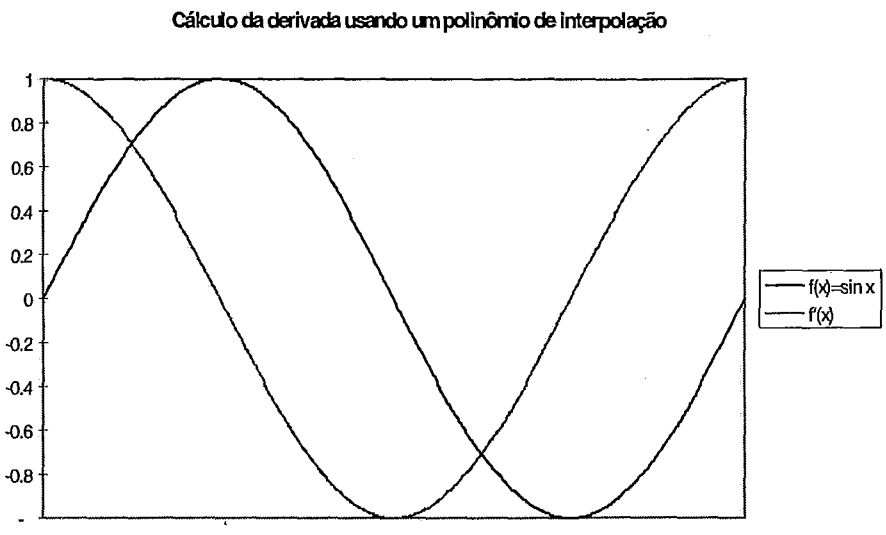


Figura 4.22 - Cálculo da derivada da função seno usando um polinômio de interpolação de ordem 5.

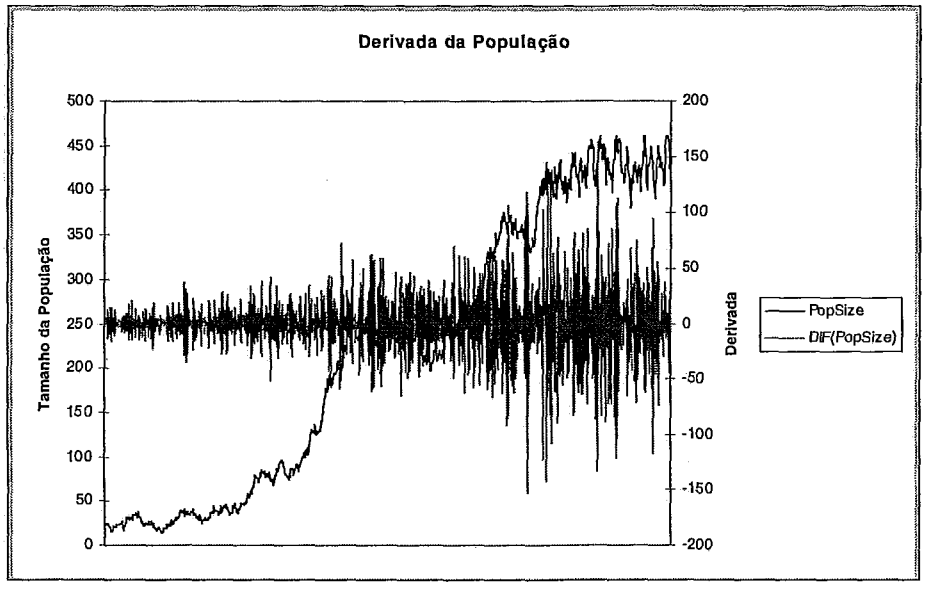


Figura 4.23 - Curvas do tamanho da população e sua derivada

O tamanho da população e a função $f'[g(t)]$ (derivada da função amortecida $g[t]$) para uma execução do algoritmo são vistos na Figura 4.24. Diante do comportamento mais "suave" da função $f'[g(t)]$, a mesma foi usada para o controle da taxa de reprodução fazendo-se:

Se $(f'[g(t)] > 0.0)$

$$\rho(t) = 0.95*\rho(t); \tag{4.34}$$

Senão

$$\rho(t) = \rho(t)/0.95; \tag{4.35}$$

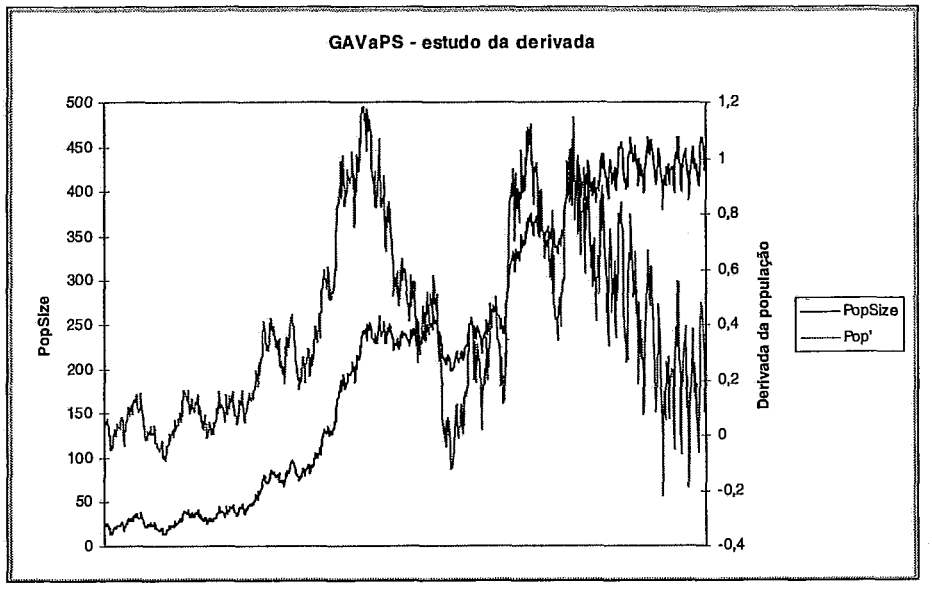


Figura 4.24 - Tamanho da população e derivada da função amortecida $g(t)$.

A nova estratégia foi testada em duas execuções do algoritmo [Figuras 4.25 e 4.26]. Apesar de o tamanho da população manter-se mais ou menos sob controle, o resultado final da otimização é muito ruim: O melhor valor do custo da fiação obtido é cerca de duas vezes e meia maior do que aquele obtido pelo *Simulated Annealing*, para o mesmo circuito de teste. Estes resultados levaram ao abandono do esquema da população variável mas é intrigante a disparidade dos resultados obtidos com aqueles reportados em [Mich94]. A explicação que parece mais convincente tem como base o fato de que as funções ali testadas são funções muito simples (funções de uma ou duas variáveis), que tornaram impossível uma avaliação efetiva do algoritmo.

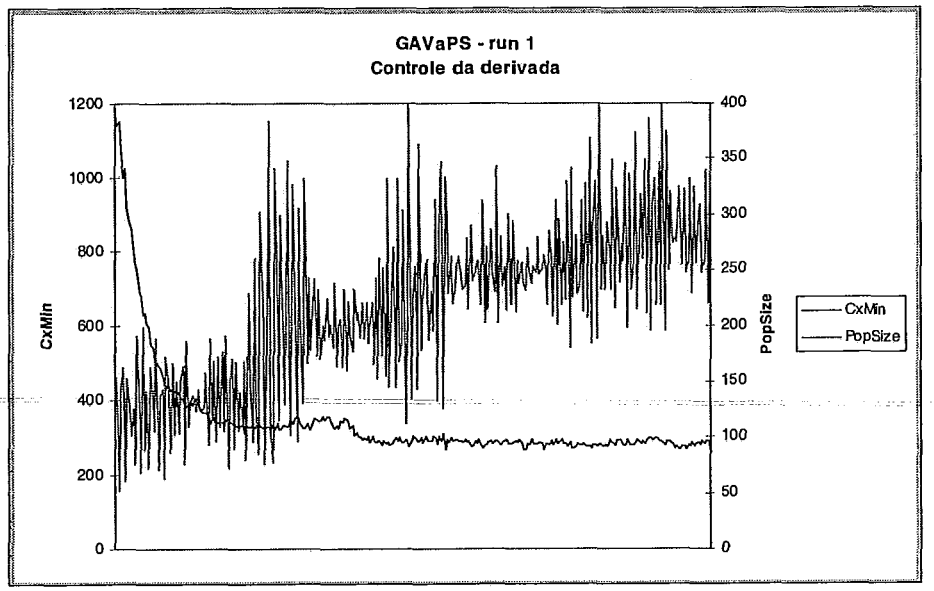


Figura 4.25 - Função custo e tamanho da população. Controle da taxa de reprodução baseado na derivada de $g[t]$. Execução 1.

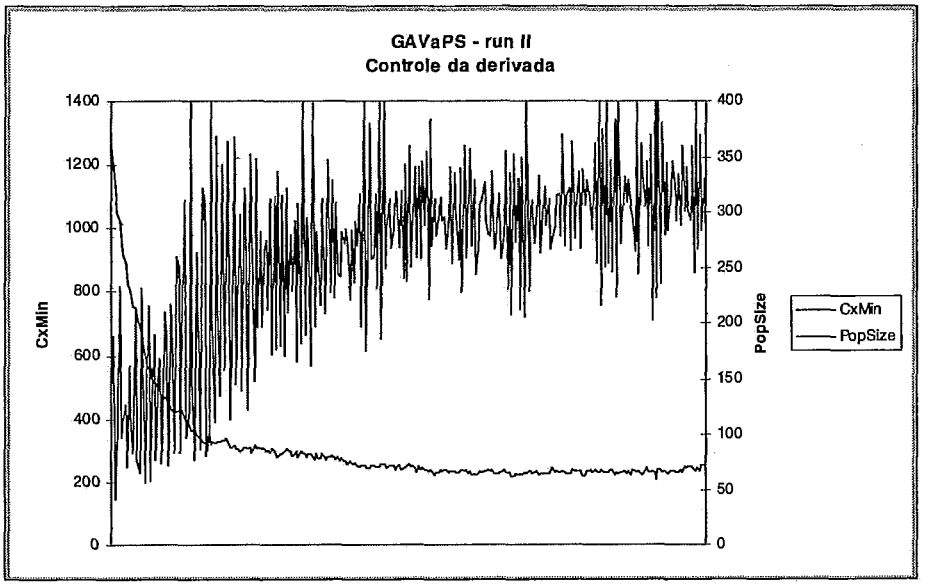


Figura 4.26 - Função custo e tamanho da população. Controle da taxa de reprodução baseado na derivada de $g[t]$. Execução 2.

4.4.2) O algoritmo híbrido:

Este seção descreve a implementação de um algoritmo híbrido. Este algoritmo visa melhorar as propriedades de convergência do algoritmo genético através da introdução do conceito de resfriamento e da aceitação probabilística dos novos indivíduos gerados em função da temperatura. O objetivo aqui é tentar reproduzir os bons resultados descritos em [Gold93] onde semelhante técnica foi utilizada.

O algoritmo de *Simulated Annealing* (SA) e os algoritmos genéticos (GA) são técnicas similares, baseadas na natureza, para a otimização de problemas genéricos. Ambos os algoritmos apresentam resultados satisfatórios em uma variedade de problemas e requerem, em princípio, conhecimento específico sobre o problema apenas para o cálculo de uma função custo apropriada. Ambas as técnicas geram novos pontos no espaço de busca, aplicando operadores aos pontos correntes e movendo-se, probabilisticamente, para regiões mais promissoras do espaço de soluções.

SA e GAs possuem, no entanto, algumas diferenças cruciais. Ainda que, na prática, ambos possam convergir prematuramente para pontos de mínimos locais, somente o SA possui uma prova formal de convergência [Aart87]. Através da manipulação da estratégia de resfriamento o projetista pode exercer controle sobre a convergência do algoritmo: resfriamento mais lento e mais iterações por temperatura levam a resultados finais melhores.

Regular a convergência de algoritmos genéticos, por sua vez, não é tarefa trivial. O ajuste de parâmetros globais tais como tamanho da população, probabilidade de mutação e probabilidade de *crossover* tem sido a técnica mais recomendada para controlar a convergência prematura de GAs. Ainda que alguns parâmetros sejam efetivos para problemas específicos [Jong75, Gref86], uma forma geral, efetiva, de ajustar os parâmetros ainda não foi demonstrada. Os valores ideais dos parâmetros são, provavelmente, dependentes do problema a ser otimizado.

Outras diferenças importantes entre os algoritmos dizem respeito à perda de configurações e à facilidade de paralelização.

Devido ao fato de que o SA mantém uma única solução, sempre que uma solução é aceita a antiga deve ser descartada. Não existe redundância ou memória do passado. A conseqüência disto é que boas configurações podem ser descartadas (no caso extremo, até mesmo o ótimo global) e, se o resfriamento for muito rápido, podem nunca voltar a ocorrer. Devido à natureza destrutiva dos operadores genéticos, os GAs também são sujeitos à perda de soluções. No entanto, devido ao fato de os algoritmos genéticos carregarem uma população de indivíduos, as soluções perdidas podem se repetir em outros elementos da população ou mesmo, através de uma política de seleção *elitista*, pode-se preservar, de uma geração para outra, os melhores indivíduos da população.

Outra importante diferença entre os algoritmos é a facilidade com que eles podem ser paralelizados. Os algoritmos genéticos são naturalmente paralelos. Eles operam sobre uma população de indivíduos usando operadores binários (*crossover*) e unários (mutação). O algoritmo *Simulated Annealing*, por outro lado, opera sobre um único indivíduo e, como vimos anteriormente, não é facilmente paralelizável.

O objetivo então desta implementação é tentar juntar as características favoráveis de ambos os algoritmos: as propriedades de convergência do *Simulated Annealing* aliadas às vantagens de manutenção de uma população de indivíduos e à facilidade de paralelização dos algoritmos genéticos.

- Pseudo código:

```
/* Melhor_{k-1}  ⇒  Indivíduo mais fitted na geração anterior                    */
/* Pior_k        ⇒  Indivíduo menos fitted na geração atual                     */
/* T             ⇒  Temperatura                                                 */
/* α             ⇒  Fator de decaimento da temperatura                          */

Genese ( );                    /* Gera uma população inicial de POPSIZE indivíduos.  */
                               /* As células nos indivíduos não se sobrepõem.        */
Repete por T gerações          {
    Se atingiu o equilíbrio em uma temperatura T
        T = α * T;

    EscolheDeme( );            /* Aleatoriamente escolhe um deme na população.       */
    Pior_k = Melhor_{k-1};     /* Esquema elitista.                                  */
    Evaluation( );             /* Avalia a aptidão dos indivíduos no deme.           */

    Pai1 = Seleção( );         /* Escolhe o primeiro pai. Método da roleta.          */
    Pai2 = Seleção( );         /* Escolhe o segundo pai.                             */

    Gera 2 filhos segundo o esquema:  {
        Crossover(Pai1, Pai2);
        EliminaOverlapping( );
        Mutação( );
    }
    Tournament( );             /* Procede a um julgamento de Boltzmann entre os 2 indivíduos*/
                               /* recém gerados e os indivíduos no centro do deme.   */
}
Imprime resultados.
```

- Detalhes de implementação:

a) Estratégia de resfriamento.

A temperatura é inicializada com o valor *TINI* = 1000.0 e reduzida a cada *Ngerações* gerações. *Ngerações* é determinada no início do processamento em função do número de gerações e da temperatura final a qual deseja-se atingir. É adotado um esquema simples de resfriamento: $T_{n+1} = \alpha_n * T_n$, onde T_n é o enésimo valor na seqüência de temperaturas e $\alpha_n = 0.98$ é a constante de resfriamento.

b) Teste de Boltzmann.

O teste de Boltzmann refere-se à competição entre as soluções *i* e *j*, onde o elemento *i* vence com probabilidade $1/\left(1+e^{(Ei-Ej)/T}\right)$.

Existem muitas formas possíveis de competição entre os dois filhos gerados e os indivíduos no centro do *deme*. A primeira possibilidade, aceitação/rejeição dupla, permite a competição em conjunto dos indivíduos no centro do *deme* contra ambos os filhos. A soma dos custos dos indivíduos no centro do *deme* substitui E_i na equação acima e a soma dos custos dos filhos substitui E_j. A segunda possibilidade (a qual foi utilizada nos testes), aceitação/rejeição simples, promove duas competições. Em cada uma delas um dos filhos compete contra um dos indivíduos no centro do *deme*.

- Resultados e Conclusões:

Conforme dito anteriormente, o objetivo do algoritmo híbrido é tentar conciliar as vantagens dos algoritmos genéticos (facilidade de paralelização, manutenção de uma população de indivíduos) com aquelas do *Simulated Annealing* (controle sobre a convergência do algoritmo). Os resultados obtidos são apresentados na Tabela 4.5 e na Figura 4.27. A linha horizontal na Figura 4.27 corresponde ao resultado obtido pelo algoritmo *Simulated Annealing*.

	POPSIZE 256	POPSIZE 625	POPSIZE 1024	POPSIZE 2500	POPSIZE 4096	POPSIZE 10000	ANNEAL
Média (C_i)	2030,91	2024,00	2008,30	1968,55	1966,10	1947,60	1915,00
Desvio Padrão	39,76	37,25	27,04	31,04	26,02	13,71	28,00
Máximo	2136	2072	2070	2027	2013	1966	1884
Mínimo	1996	1944	1977	1920	1931	1921	1963

Tabela 4.5 - Resultados obtidos pelo algoritmo híbrido para o CIRCUITO_2.

Ainda que, consistentemente, o algoritmo híbrido tenha obtido melhores resultados do que o algoritmo padrão descrito na seção 4.3, ele falha em atingir os resultados obtidos pelo *annealing*. Além disso, os tempos de processamento são proibitivamente longos. A partir da observação dos resultados, é possível tirar-se algumas conclusões interessantes a respeito do algoritmo híbrido:

a) Em relação ao *Simulated Annealing* simples, o algoritmo misto têm a desvantagem evidente dos requisitos de memória necessários à manutenção de uma população.

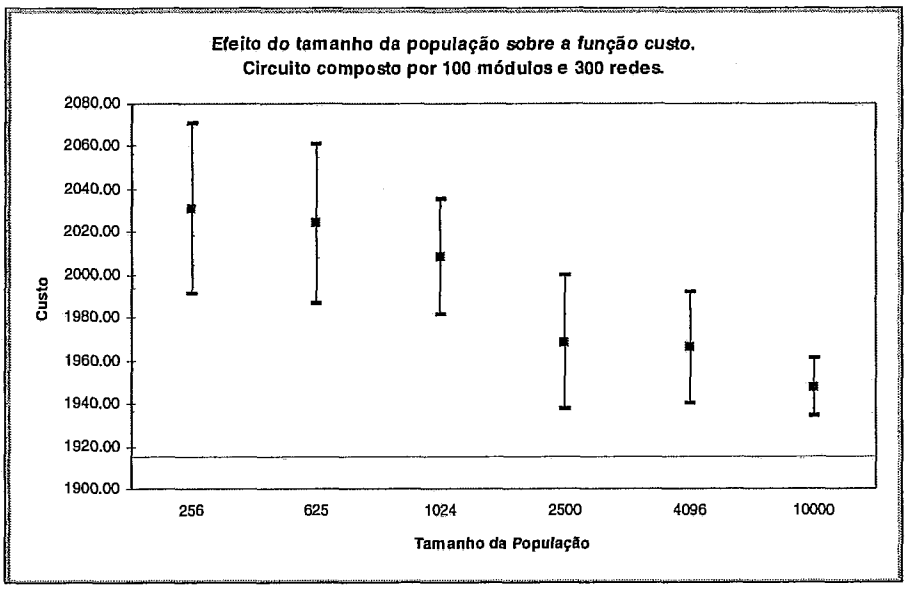


Figura 4.27 - Algoritmo híbrido. Efeito do tamanho da população sobre a função custo.

b) A aceitação probabilística de um novo indivíduo, baseada em uma função exponencial, é uma operação lenta e computacionalmente dispendiosa.

c) As operações elementares de *swaps* e *displacements*, ao invés de exaustivamente aplicadas a um único indivíduo como no *Simulated Annealing*, são agora distribuídas por toda a população. Assim, para preservar as propriedades de convergência do SA (baseadas em um número suficiente de estados gerados por temperatura), o número de gerações (ou iterações) do algoritmo genético teria de ser extremamente elevado.

d) Para minorar o problema descrito no item anterior poder-se-ia pensar em trabalhar com populações de poucos indivíduos mas, populações reduzidas não parecem combinar com o esquema do equilíbrio pontual usado no modelo paralelo de distribuição da população.

e) O parâmetro temperatura acaba por ter um efeito negativo uma vez que, em baixas temperaturas, é muito pequena a probabilidade de uma operação de mutação ser efetivada. Isto resulta em populações com pouca diversidade genética, condição essencial para que o operador de *crossover* tenha sucesso.

Pelos motivos acima, o modelo híbrido não foi utilizado como base para a paralelização do algoritmo. No entanto, é interessante notar que nos algoritmos genéticos as grandes variações na função custo são obtidas durante a fase inicial do algoritmo. Desta forma, é possível que uma estratégia híbrida no tempo trouxesse algum resultado. Uma população de indivíduos seria inicializada e o algoritmo genético prosseguiria por um certo número de gerações quando então o melhor indivíduo da população serviria de ponto de partida para o *annealing*. De modo a não perder uma configuração promissora, o *annealing* seria inicializado com uma temperatura mais baixa, já próxima ao resfriamento. Estratégias parecidas foram já testadas por outros pesquisadores. Rose, em [Rose86], relata uma implementação bem sucedida em que uma estratégia de *placement* baseada no algoritmo Min-Cut é combinada com o *annealing* em baixas temperaturas.

4.4.3) O Operador de Inversão e o Crossover Cíclico:

Este algoritmo corresponde à primeira tentativa de projetar operadores genéticos que levassem em conta características do problema de *placement*. Os operadores introduzidos: inversão e *crossover* cíclico, são baseados no trabalho de Mazumder et al. [Mazu93], [Mazu90] e [Mazu91].

A necessidade de novos operadores é proveniente do fato de que os genes são dispostos nos cromossomas sem nenhuma preocupação com a conectividade entre os módulos representados pelos mesmos. A simples aplicação do *crossover* tradicional (1-ponto ou 2-pontos) a estes cromossomas parece não ser satisfatória. Para um perfeito entendimento do problema é necessário introduzir-se o conceito de epistasia.

O termo epistasia foi definido por geneticistas para enfatizar o fato de que a influência de um gene na aptidão de um indivíduo depende dos "valores" assumidos pelos outros genes no cromossoma. O termo foi adotado no contexto de algoritmos genéticos para indicar forte interação entre genes, ou, segundo definição de Beasley *et al.* [Beas93]:

> "Epistasis is the interaction between different genes in a chromosome. It is the extent to which the "expression" (i.e. contribution to fitness) of one gene depends on the values of other genes. The degree of interaction will, in general, be different for each gene in a chromosome. If a small change is made to one gene we expect a resultant change in chromosome fitness. This resultant change may vary according to the values of other genes.
> (...)Unfortunately, according to the *building block hypothesis*, one of the basic requirements for a GA to be successful is that there is low epistasis. This suggests that GAs will not be effective on precisely those type of problems in which they are most needed. Clearly, understanding epistasis is a key issue for GA research. We need to know whether we can either avoid it, or develop a GA which will work even with high epistasis."

Percebe-se então que a epistasia entre os genes é uma forte característica do problema de *placement*. Isto deve-se ao fato de que a contribuição de um gene (representando a localização de um único módulo n do circuito) no cálculo da função custo depende da localização de todos os outros módulos pertencentes a todas as redes das quais o módulo n faz parte.

Até aqui, a alocação dos módulos nos genes do cromossoma se dava na ordem em que os mesmos apareciam no arquivo de descrição do circuito. O *crossover* tradicional, se aplicado a este cromossoma, pode ter um efeito bastante destrutivo. Examinemos o seguinte exemplo: Seja um circuito composto por N módulos e a rede S_1, integrante deste circuito. A rede S_1 é formada pelos módulos n_1, n_2, n_3 e n_4. Suponha que a descrição do circuito seja tal que o módulo n_1 é o primeiro a ser alocado no cromossoma e o módulo n_4 o último. Este cromossoma é representado na Figura 4.28.

Figura 4.28 - Ilustração do problema da epistasia entre os módulos

Percebe-se claramente que, se utilizado o *crossover* de 1 ponto, a rede S_1 será necessariamente seccionada pelo operador, qualquer que seja o ponto de quebra escolhido. O operador de *crossover* tem então um efeito negativo na medida em que ele, potencialmente, destrói sub *placements* promissores provenientes de um dos pais. Ainda que de visualização não tão óbvia o problema se repete para o *crossover* de 2 pontos.

O problema consiste então em projetar operadores genéticos que reduzam a epistasia a níveis aceitáveis ou, conforme citação de Beasley *et al.* [Beas93]:

> "Davidor [Davi90] also points out that present-day GAs are only suitable for problems of medium epistasis. If the epistasis is very high, the GA will not be effective. If it is very low, the GA will be outperformed by simpler techniques, such as hill climbing. Until such a time as we have GAs which are effective on problems of high epistasis, we must devise representation schemes (or crossover/mutation operators) which reduce epistasis to an acceptable level."

- O Operador de Inversão:

O operador de inversão altera a ordem dos genes em um cromossoma representando um *placement.* Por exemplo, dado um cromossoma AB.CDE.F onde os pontos de inversão (randomicamente determinados) são assinalados, o resultado da operação de inversão é o cromossoma AB.EDC.F. Esta operação deve ser realizada de forma a que ela não modifique a solução codificada em um cromossoma, mas apenas a representação desta solução. Desta forma, as estruturas de dados nos genes devem ter interpretação independente da posição que elas ocupam no vetor. Isto é obtido associando-se um número inteiro (correspondente à identificação do módulo) a cada gene no cromossoma. A interpretação do conteúdo dos genes se dá em relação à identificação do módulo e não à posição no vetor. Quando um gene é movido no cromossoma, a identificação do módulo correspondente é movida com ele, de modo que a interpretação do gene permanece inalterada.

O *placement* é codificado em um vetor de estruturas representando módulos do circuito (o cromossoma). Cada estrutura contém as seguintes informações:

- Identificação do módulo (*id*).
- Coordenada *x* no plano.
- Coordenada *y* no plano.
- Acesso indireto (*xlat*).

O campo *xlat* permite um rápido acesso aos módulos do circuito. Seja o exemplo da Figura 4.29. O campo *xlat*, na posição *i* do vetor, contém o índice *j* onde está representado o módulo *i*. Este campo permite um rápido acesso aos módulos no cálculo da função custo e na implementação do *crossover* cíclico, a ser discutida mais adiante.

O operador de inversão, ao embaralhar a localização dos módulos no vetor, permite que conjuntos de módulos, já bem posicionados relativamente uns aos outros, permaneçam próximos também no cromossoma. Isto aumenta a probabilidade de que o sub *placement* composto por estes módulos seja passado intacto de um pai ou de outro

para o filho. Este processo, segundo Mazumder *et al.*, permitiria a formação de sub *placements* altamente otimizados muito antes do processo de otimização como um todo estivesse terminar.

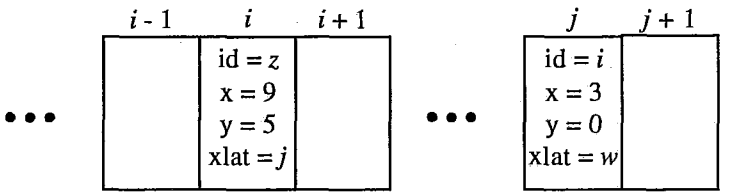


Figura 4.29 - Estrutura de dados de um cromossoma para a implementação do *crossover* cíclico e do operador de inversão.

- Crossover cíclico:

O operador de *crossover*, na sua forma mais simples, envolve a troca de pedaços entre os cromossomas de dois indivíduos. Este *crossover* simples não pode, no entanto, ser aplicado a cromossomas embaralhados pelo operador de inversão na medida em que os indivíduos gerados podem não ter representação física. Considere, por exemplo, um *crossover* de 1 ponto entre os cromossomas ABCD e BDCA. Os indivíduos gerados poderiam ter a forma: ABCA e BDCD. Aqui, cada indivíduo tem um módulo duplicado e um módulo ausente. São necessárias, portanto, estratégias mais complexas de *crossover*, que preservem a coerência do *placement*. Mazumder *et al.* propuseram então um operador modificado de *crossover* chamado de *crossover* cíclico.

A Figura 4.30 ilustra o funcionamento do *crossover* cíclico. Comece com o módulo na primeira posição de *Pai1* e copie-o para a primeira posição de *offspring*. Considere então o módulo na primeira posição de *Pai2*. Este módulo não pode ser herdado de *Pai2* uma vez que a sua posição já foi ocupada em *offspring*. Este módulo é então também herdado de *Pai1* e o processo se repete até que o módulo a ser posicionado já se encontre em *offspring* (isto é, quando completar-se o ciclo). Escolha então o primeiro módulo ainda não colocado em *Pai2* e repita o procedimento anterior. Os números abaixo das posições do vetor em *offspring* mostram a ordem em que os módulos foram alocados.

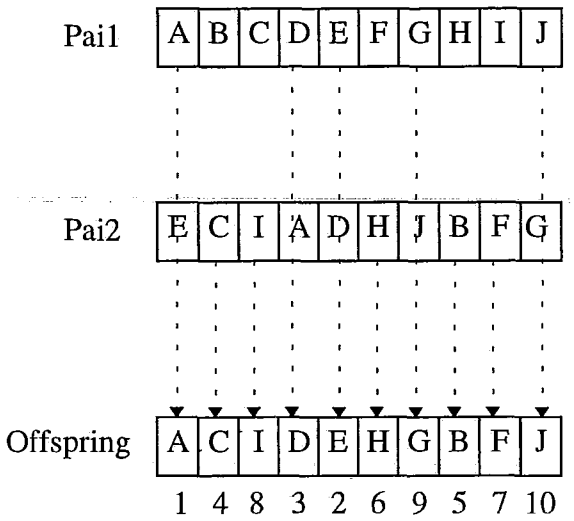


Figura 4.30 - Crossover cíclico

- Pseudo Código:

Em cada geração um *deme* é escolhido aleatoriamente. O operador de inversão é aplicado a cada indivíduo do *deme* com probabilidade *Pinversion* (nesta implementação foi utilizado *Pinversion* = 0.33).

```
/* Melhor_k      ⇒   Indivíduo mais fitted na geração atual                    */
/* Melhor_k-1    ⇒   Indivíduo mais fitted na geração anterior                 */
/* Pior_k        ⇒   Indivíduo menos fitted na geração atual                   */

Genese ( );                          /* Gera uma população inicial de POPSIZE indivíduos. */
                                     /* As células nos indivíduos não se sobrepõem        */
Repete por T gerações          {
    EscolheDeme( );                  /* Aleatoriamente escolhe um deme na população       */
    Inversão( );                     /* Com probabilidade Pinversion, inverte cada indivíduo */
                                     /* do deme.                                          */
    Pior_k = Melhor_k-1;             /* Esquema elitista. Substitui o pior indivíduo desta */
                                     /* geração pelo melhor da geração anterior           */
    Evaluation( );                   /* Avalia o fitness dos indivíduos no deme.          */
                                     /* Transformação bilinear                            */
    Pai1 = Seleção( );               /* Escolhe o primeiro pai. Método da roleta.         */
    Pai2 = Seleção( );               /* Escolhe o segundo pai                             */
    Gera 2 filhos segundo o esquema:  {
        Crossover(Pai1, Pai2);       /* Crossover cíclico.                                */
        EliminaOverlapping( )        /* Elimina sobreposições.                            */
        Mutação( )                   /* Probabilisticamente aplica mutação ao             */
                                     /* indivíduo recém gerado.                           */
    }
    Tournament( );                   /* Os 2 indivíduos recém gerados substituem os       */
                                     /* indivíduos no centro do deme.                     */
}
Imprime resultados.
```

- Resultados:

O operador de inversão e o *crossover* cíclico foram implementados, mas o algoritmo não apresentou bons resultados no que se refere à qualidade do *placement* gerado. Além disso, o *crossover* cíclico é computacionalmente muito dispendioso, o que tornou extremamente lento o processamento do algoritmo.

A inconsistência do algoritmo parece residir no *crossover* cíclico. De fato, a utilização do operador de inversão parece consistente na medida em que (conforme será constatado posteriormente) existem benefícios em que grupos de genes densamente conectados (por exemplo, módulos pertencentes a uma mesma rede) sejam herdados inteiros de um dos pais. Segundo Mazumder *et al.*, o operador de inversão facilitaria a aproximação destes genes de modo a que eles tivessem probabilidade menor de serem separados pelo operador de *crossover*. A inconsistência parece vir do fato de que o *crossover* simples, se aplicado, gera configurações incoerentes e o operador de *crossover* projetado para resolver este problema: o *crossover* cíclico, não oferece probabilidade maior de que módulos próximos uns dos outros no cromossoma permaneçam juntos após o *crossover*.

Considere novamente a Figura 4.30. Suponha que os módulos *A* e *B* em *Pail* sejam densamente conectados e que eles tenham sido aproximados com sucesso pelo operador de inversão. Observe que, no *offspring* resultante, tais módulos foram separados pelo *crossover* cíclico, invalidando assim o benefício advindo da utilização do operador de inversão.

Uma possibilidade a se investigar para a solução deste problema, seria utilizar-se o operador de inversão de forma síncrona, isto é, de tempos em tempos seria efetuada a inversão, e o ordenamento resultante seria comunicado a toda a população. Com isso seria possível usar o *crossover* tradicional, de 1-ponto ou 2-pontos. Esta solução tem a desvantagem óbvia da necessidade de sincronização, ainda mais crítica quando se estiver paralelizando o algoritmo.

4.4.4) Codificação Expandida:

Esta seção descreve os experimentos com uma outra alternativa para lidar com a epistasia entre genes, característica marcante do problema de *placement*. Ela consiste de um esquema especial de representação projetado de forma a tentar reduzir a epistasia a níveis aceitáveis.

A idéia básica consiste em substituir uma representação composta por poucos genes, fortemente acoplados, por outra consistindo de um número maior de genes com acoplamento mais fraco. Produz-se assim um espaço de soluções maior e, ao mesmo tempo, mais simples e de cálculo mais fácil. Os princípios desta técnica foram enumerados por Beasley, Bull e Martin em [Beas93c].

O algoritmo é implementado como se segue. Se um nó pertence a mais de uma rede, são criadas várias instâncias deste nó, cada uma com uma numeração diferente. Durante a evolução do processo de otimização as sobreposições de módulos são permitidas mas penalizadas, quando os módulos que se sobrepõe não correspondem a instâncias diferentes de um mesmo módulo. A esperança é a de que, ao final do processamento, todas as instâncias de um mesmo nó convirjam para a mesma posição no plano.

- Resultados:

Os resultados obtidos mostraram-se decepcionantes. Em uma série de execuções do algoritmo, o comprimento de fiação obtido foi sempre muito superior àquele obtido pelo *Simulated Annealing*, para o mesmo circuito de teste. Além disso, o algoritmo mostrou-se incapaz de eliminar completamente as sobreposições de módulos na configuração final gerada. Assim, a técnica da codificação expandida veio somar-se ao rol das experiências mal sucedidas empreendidas na busca por um algoritmo seqüencial confiável.

4.4.5) Crossover por Redes:

Este algoritmo corresponde a uma nova tentativa de projetar operadores que, levando em conta particularidades do problema de *placement*, contribuam para a redução do problema da epistasia entre os genes.

A idéia aqui é, ao invés dos filhos herdarem pedaços aleatórios dos cromossomas pais, fazer com que a herança de material genético se dê por módulos componentes de redes equipotenciais. O objetivo é preservar redes potencialmente já bem posicionadas em um dos pais.

A rotina toma como entrada dois cromossomas: *Pai1* e *Pai2*. A cada iteração, o algoritmo constrói o conjunto S formado por um máximo de (*nnets*/2) redes do circuito, escolhidas aleatoriamente. O cromossoma filho herda as redes contidas em S (isto é, todos os módulos que as compõem) de *Pai1* e as demais de *Pai2*.

Observe-se que a estratégia de *crossover* adotada tem o efeito de que $|S|$ redes são tomadas inteiras de *Pai1*, algumas outras são tomadas inteiras de *Pai2* (redes que não têm nenhum módulo em comum com as redes em S) e algumas redes são seccionadas, isto é, alguns módulos são provenientes de *Pai1* e outros de *Pai2*. O pseudocódigo do *crossover* por redes é apresentado a seguir.

```
Crossover(Pai1, Pai2)          {
/*
Variáveis:
        nnets      ⇒    número de redes no circuito
Funções:
        rnd(n)     ⇒    número inteiro no intervalo [0..n)
Saída:
        filho      ⇒    cromossoma contendo genes de Pai1 e Pai2.
*/
        Seleciona aleatoriamente rnd(nnets/2) redes do circuito;
        O vetor filho herda de Pai1 todos os módulos que pertencem a alguma das
                redes selecionadas;
        Os demais módulos são herdados de Pai2;
}
```

Numa primeira observação é possível apontar, já neste ponto, uma primeira deficiência do *crossover* por redes: não existe a preocupação em minimizar o número de redes seccionadas. Consideremos, por exemplo, o caso do módulo mais densamente conectado do circuito. É bastante improvável que todas as redes das quais o módulo participa sejam selecionadas. Este fato pode dificultar o *placement* da região onde este módulo esteja sendo alocado.

- Resultados:

Os resultados obtidos não foram satisfatórios. A Tabela 4.6 mostra os valores medidos para o CIRCUITO_2 e uma população de 10000 indivíduos. Para fins de comparação são também apresentados os resultados para o algoritmo básico (modelo paralelo de distribuição de população, *crossover* de 2 pontos) e para o *Simulated Annealing*.

	Crossover por redes	Algoritmo básico	*Simulated Annealing*
Média	1983,82	2010,50	1915,00
Desvio Padrão	17,72	15,62	28,00
Máximo	2003	2039	1963
Mínimo	1949	1987	1884
Tempo (s)	21875,82	10391,55	89,94

Tabela 4.6 - Resultados para o *crossover* por redes.

Os resultados obtidos, ainda que ligeiramente superiores aos obtidos pelo algoritmo básico, são ainda inferiores aos do *annealing*. Observe ainda o tempo proibitivo necessário à execução do algoritmo: a cada iteração é preciso determinar um conjunto diferente de redes que serão herdadas pelo filho. É preciso cuidar para que a seleção não contenha redes duplicadas. Uma vez escolhidas as redes é preciso percorrê-las uma a uma e copiar as coordenadas dos módulos que as constituem para o filho.

Desta forma, outras alternativas para lidar com o problema da epistasia continuaram a ser buscadas.

4.4.6) Crossover por Regiões do Plano:

Como foi visto na seção anterior, o *crossover* por redes buscava preservar sub *placements* promissores potencialmente presentes em um dos pais, fazendo com que o filhos herdassem redes inteiras dos cromossomas pais. Uma deficiência do algoritmo é a de que não existe a preocupação em minimizar o número de redes seccionadas. No caso de módulos densamente conectados, dificilmente todas as redes das quais estes módulos participam seriam selecionadas ao mesmo tempo. Este fato pode dificultar o *placement* devido ao efeito destrutivo sobre as regiões onde estes módulos estejam sendo alocados.

Uma alternativa que pareceu então viável, foi o *crossover* por regiões: regiões inteiras do plano são herdadas dos pais pelo cromossoma recém criado. Ou, em outras palavras, o filho passa a herdar os genes dos pais não mais segundo a sua localização no cromossoma (*crossover* tradicional), mas sim segundo a localização no plano de alocação. O objetivo aqui é o de tentar preservar regiões do plano nos cromossomas pais, já com uma boa alocação de módulos. O *crossover* por regiões foi primeiro descrito por Cohoon e Paris em [Coho86]. Os detalhes de implementação serão vistos a seguir.

O operador seleciona uma região do plano formada por um quadrado de dimensões L x L. Esta região é então copiada de *Pai1* para *Pai2*. Seja o exemplo da Figura 4.31 onde são marcadas as regiões R_1 e R_2. Seja n a identificação de um módulo qualquer do circuito. É possível definir então 3 conjuntos de interesse:

$$S_1 = \{ n \mid n \in R_1, n \in R_2 \} \tag{4.36}$$

$$S_2 = \{ n \mid n \in R_1, n \notin R_2 \} \tag{4.37}$$

$$S_3 = \{ n \mid n \notin R_1, n \in R_2 \} \tag{4.38}$$

Os módulos em S_1 (módulos *B*, *G*, *H* e *I* no exemplo) apenas trocam de posição em R_2. Os módulos em S_2 (módulos *A*, *C*, *D*, *E* e *F*) são deslocados de outras posições do plano, fora de R_2, para o interior da região assinalada. Os módulos em S_3 (módulos *X*, *W*, *P*, *M* e *N*), que ocupavam anteriormente posições em R_2, passam a causar sobreposição de células e têm, portanto, de ser retirados do interior do quadrado. Usam-se para isto as posições vagas deixadas pelos módulos em S_2 quando estes foram deslocados para o interior de R_2. (Vide Figura 4.31)

Uma desvantagem clara do operador é o potencial efeito destrutivo sobre as redes das quais os módulos em S_3 fazem parte. Idealmente, ao final do processo de otimização, quando a população é fortemente homogênea (isto é, os indivíduos diferem pouco entre si), não deveria haver grandes deslocamentos de módulos causados pelo operador de

crossover. No entanto, o deslocamento dos módulos em S_3 para posições aleatórias no plano pode terminar por anular os benefícios advindos da cópia da região R_1.

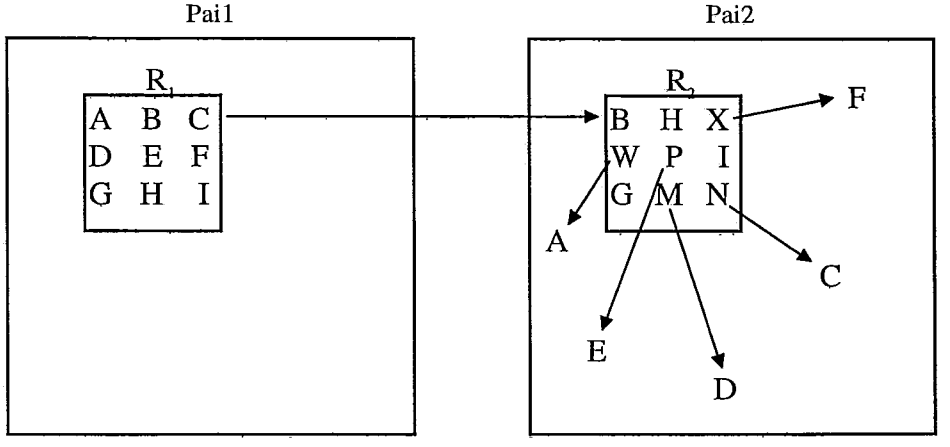


Figura 4.31 - Operador de *crossover* por regiões.

- Resultados:

A Tabela 4.7 mostra os valores medidos para o CIRCUITO_2 e uma população de 10000 indivíduos. Para fins de comparação são também apresentados os resultados para o algoritmo básico (modelo paralelo de distribuição de população, *crossover* de 2 pontos), para o *crossover* por redes e para o *Simulated Annealing*.

	Crossover por regiões	*Crossover* por redes	Algoritmo básico	*Simulated Annealing*
Média	1912,20	1983,82	2010,50	1915,00
Desvio Padrão	19,78	17,72	15,62	28,00
Máximo	1937	2003	2039	1963
Mínimo	1882	1949	1987	1884
Tempo (s)	10301,19	21875,82	10391,55	89,94

Tabela 4.7 - Resultados para o *crossover* por regiões do plano.

Do ponto de vista da qualidade da solução gerada, o *crossover* por regiões do plano apresentou resultados satisfatórios. O tempo de processamento é ainda, no entanto, extremamente elevado. Era necessário então continuar buscando por alternativas que permitissem a utilização de populações menores e/ou um menor tempo de processamento.

4.4.7) Pré alocação de módulos:

Os operadores de *crossover* descritos anteriormente (*crossover* por redes e *crossover* por regiões do plano) buscavam preservar sub *placements* já otimizados porventura presentes em um dos pais. Desta forma, ambos os operadores constituíam-se em uma tentativa de minimizar o efeito destrutivo do *crossover* tradicional sobre o problema de

placement ou, em outras palavras, os operadores buscavam uma forma efetiva de lidar com a epistasia entre genes.

Ainda com o objetivo de projetar operadores que preservassem soluções promissoras, foi desenvolvida aqui uma terceira estratégia de *crossover*: o operador utilizado é o *crossover* tradicional, onde os filhos herdam pedaços dos cromossomas dos pais, mas procede-se a uma pré alocação dos módulos nos cromossomas, antes do início do processo de otimização. O objetivo da pré alocação é fazer com que o menor número possível de redes seja seccionada pela partição do cromossoma em um ponto arbitrário ou, em outras palavras, busca-se minimizar o número de redes que tenham módulos simultaneamente em ambos os "pedaços" do cromossoma resultantes da operação de *crossover*. É conveniente ressaltar que a ordenação dos módulos no cromossoma nada tem a ver com a posição desses módulos no plano de alocação. Ela constitui-se, tão somente, em uma tentativa de manter juntas, após o *crossover*, redes já bem posicionadas presentes em um dos pais.

O problema consiste em que a alocação ótima dos módulos no cromossoma é também um problema não trivial (possivelmente não polinomial) de otimização. Desta forma, visando confirmar a hipótese de que a ordenação dos módulos tinha algum efeito sobre a qualidade do *placement* gerado, foi desenvolvido um algoritmo baseado em *Simulated Annealing* para a pré alocação dos módulos. A função custo, a ser minimizada, é obtida partindo-se o cromossoma em todos os pontos possíveis e calculando-se o somatório do número de redes seccionadas. O operador de vizinhança consiste no *swap* de módulos no cromossoma. O pseudocódigo da função custo é apresentado abaixo.

```
Função Custo( )      {
/*
Ns(i)        ⇒   Número de redes seccionadas pelo corte do cromossoma na posição i.
nmodulos     ⇒   Número de módulos no circuito.
*/
        Custo = 0;
        Para (i = 1) até (nmodulos - 1)      {
            Corta o cromossoma na posição i;   /* i.e. entre o gene (i - 1) e o gene i   */
            Determina Ns(i);
            Custo = Custo + Ns(i);
        }
        Retorna Custo;
}
```

O algoritmo de alocação baseado em *Simulated Annealing*, apesar de fornecer excelentes resultados, não mostrou-se uma alternativa viável uma vez que o custo envolvido em determinar a ordenação ótima dos módulos no cromossoma é, possivelmente, da ordem do custo necessário a resolver-se o problema de *placement*. Por este motivo, foi desenvolvida uma heurística para a alocação dos módulos. Esta heurística foi validada pelos resultados obtidos e por comparação com o algoritmo baseado no *annealing*. Estes pontos serão vistos com mais detalhes a seguir.

A fim de comprovar o efeito da ordenação dos módulos no cromossoma, o *annealing* foi configurado de forma a obter a melhor e a pior ordenação possíveis. Os resultados obtidos para o CIRCUITO_1 podem ser vistos na Figura 4.32. As curvas representam o

número de redes não seccionadas em cada ponto de corte. Dois aspectos nessas curvas merecem especial atenção:

a) A diferença dramática obtida entre as curvas de melhor e pior ordenação dos módulos. Observe-se que, para a curva de melhor ordenação, no pior caso, apenas 4 redes são seccionadas pelo operador de *crossover*. Por outro lado, para a pior ordenação possível, nenhuma rede é preservada se o ponto de corte for escolhido entre os genes de número 16 e 64 (uma extensão considerável do cromossoma).

b) A curva sob o rótulo "normal" corresponde à ordenação que vinha, até então, sendo utilizada nos testes. Pode-se observar que ela corresponde a uma ordenação já "razoável" dos módulos. Isto se deve ao fato de que os módulos eram alocados no cromossoma à medida em que eles apareciam na descrição das redes. As redes (obtidas automaticamente por um programa gerador), por sua vez, são organizadas fortemente em *clusters* no arquivo de entrada. Assim, se um módulo pertence simultaneamente a mais de uma rede, estas redes, na maioria das vezes, estão em linhas contíguas no arquivo de entrada. Isto assegura uma certa ordenação não ótima aos genes do cromossoma.

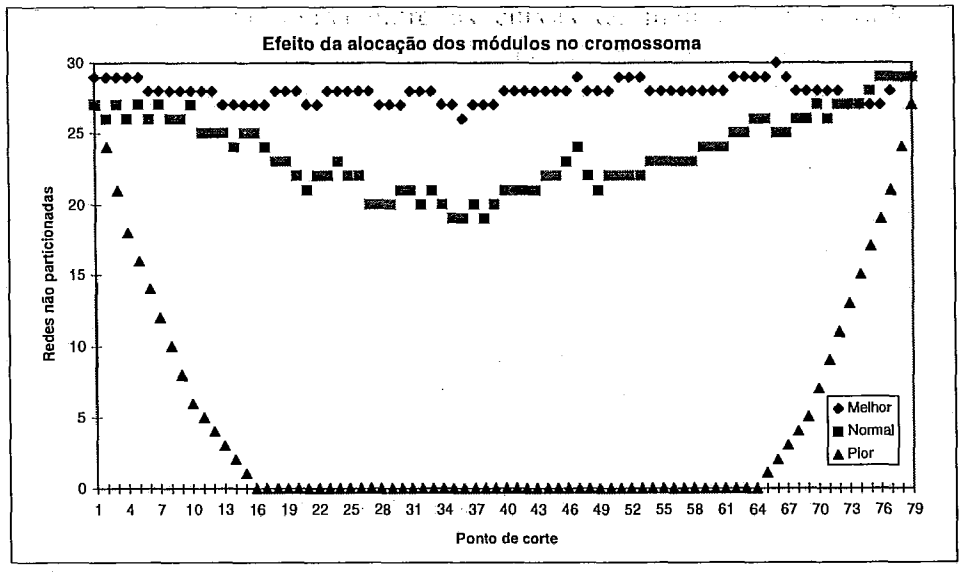


Figura 4.32 - Efeito da ordenação dos módulos no cromossoma sobre o número de redes seccionadas.

Os resultados para a melhor e pior ordenação possíveis são apresentados nas Figuras 4.33 e 4.34, respectivamente. Os pontos nas curvas referem-se à função custo (C_x) e aos valores de ($C_x + \sigma$) e ($C_x - \sigma$), onde σ é o desvio padrão da distribuição. As retas horizontais nos gráficos correspondem ao resultado obtido pelo *annealing*.

A partir da observação dos gráficos conclui-se que, para a aplicação do *crossover* tradicional ao problema de *placement*, é fundamental a ordenação dos genes no cromossoma de modo a reduzir a epistasia a níveis aceitáveis.

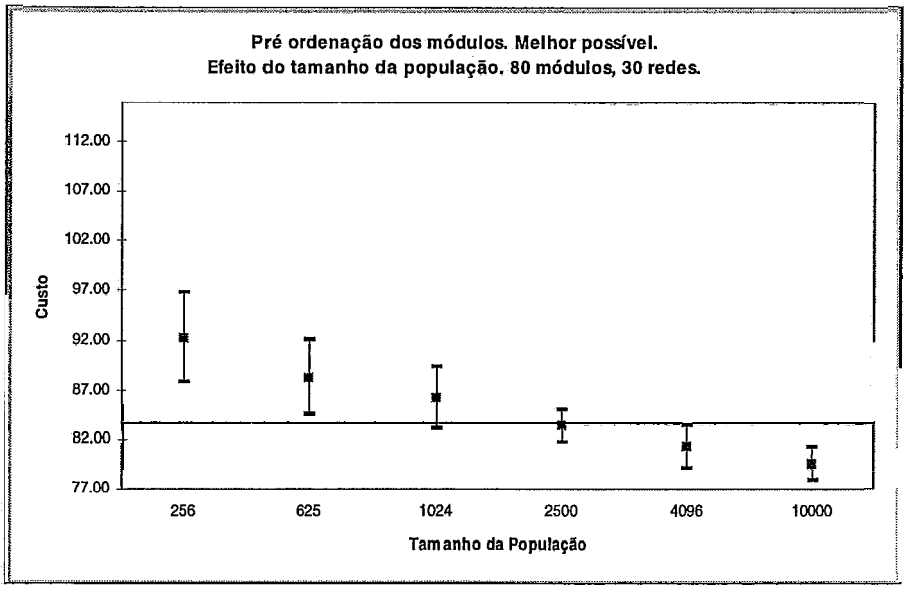


Figura 4.33 - Efeito da ordenação dos módulos no cromossoma. Melhor ordenação possível. CIRCUITO_1.

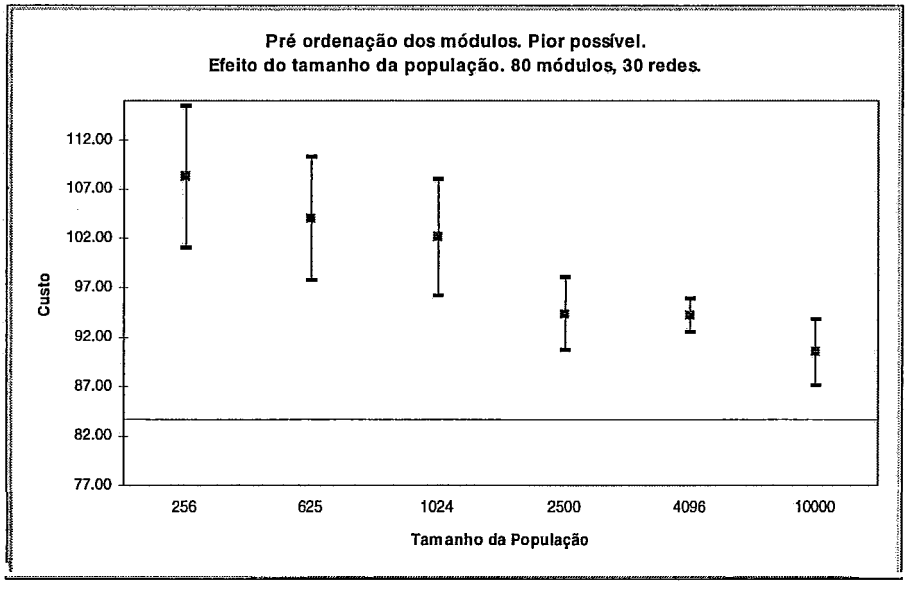


Figura 4.34 - Efeito da ordenação dos módulos no cromossoma. Pior ordenação possível. CIRCUITO_1.

A Tabela 4.8 e a Figura 4.35 mostram os resultados obtidos para o CIRCUITO_2. A curva tracejada corresponde à regressão logaritmica dos pontos medidos.

	POPSIZE 256	POPSIZE 625	POPSIZE 1024	POPSIZE 2500	POPSIZE 4096	POPSIZE 10000	ANNEAL
Média (C_x)	1955,18	1921,30	1926,80	1900,00	1894,70	1890,18	1915,00
Desvio Padrão	18,99	24,63	14,68	9,75	13,35	12,60	28,00
Máximo	1989	1972	1947	1912	1919	1908	1884
Mínimo	1925	1890	1892	1883	1873	1866	1963
Tempo (s)	239,18	585,37	960,43	2347,28	4225,85	10391,55	89,94

Tabela 4.8 - Resultados obtidos por pré alocação dos módulos. CIRCUITO_2.

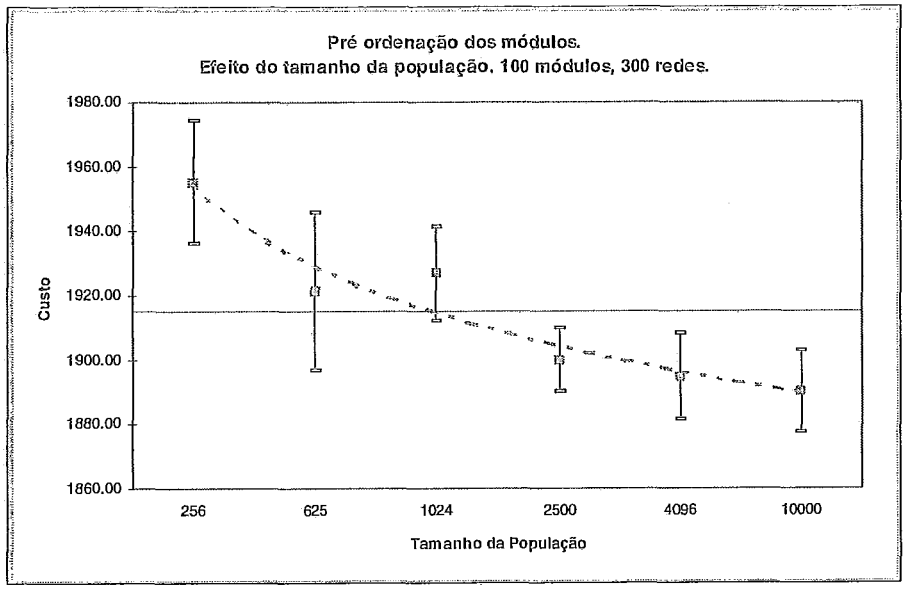


Figura 4.35 - Pré alocação dos módulos. Efeito do tamanho da população. CIRCUITO_2.

A partir dos resultados acima observa-se que, para obter-se resultados semelhantes ao *annealing*, o tamanho da população necessário (em torno de 1024 indivíduos) acarreta em um tempo de processamento ainda excessivamente longo. A esperança para a solução deste problema recai na implementação paralela, a ser vista no próximo Capítulo.

Dada a inviabilidade do cálculo da ordenação ótima usando *annealing*, foi portanto necessário desenvolver alguma heurística de processamento rápido que produzisse resultados aceitáveis. Detalhes da construção deste algoritmo serão vistos a seguir.

O algoritmo desenvolvido é um algoritmo guloso. Os módulos são alocados no cromossoma em posições contíguas, da esquerda para a direita. A cada instante busca-se, basicamente, alocar o módulo que tem mais ligações com módulos já alocados.

Sejam:

j $\Rightarrow$ Próxima posição a ser preenchida no cromossoma

S $\Rightarrow$ Conjunto dos módulos já alocados no cromossoma

G $\Rightarrow$ Conjunto dos módulos candidatos a ocuparem a posição j, $G \cap S = \varnothing$

Para cada módulo $g \in G$, calcula-se a função $f(g)$:

$$f(g) = \sum_i y(j - posic(s_i)) \tag{4.39}$$

onde:

S_i $\Rightarrow$ Conjunto dos módulos $s_i \in S$ que pertencem a pelo menos uma rede em comum com g.

$posic(s_i)$ $\Rightarrow$ Índice da posição do cromossoma ocupada pelo módulo s_i.

$y(\ )$ é uma função bilinear na forma:

$$dist = j - posic(s_i)); \tag{4.40}$$

Se $(dist \leq L)$

$$y(dist) = 1.0; \tag{4.41}$$

Senão

$$y(dist) = 1.0 + \alpha\ (dist - L); \tag{4.42}$$

A constante L corresponde ao número médio de módulos nas redes do circuito. α é o coeficiente angular da reta em (4.42) e dá a velocidade de decaimento da influência dos módulos alocados a uma distância maior do que L da posição j. Procura-se, desta forma, priorizar ligações com módulos mais recentemente alocados.

A curva do número de redes não seccionadas obtida pela aplicação da heurística ao CIRCUITO_1 pode ser vista na Figura 4.36.

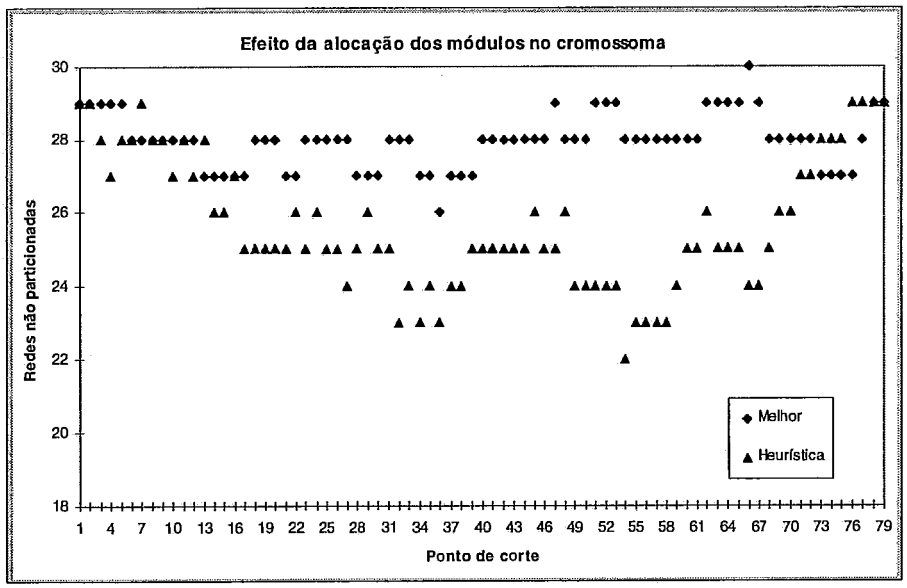


Figura 4.36 - Resultado da heurística para a pré alocação dos módulos no cromossoma.

Os valores medidos para o CIRCUITO_1, usando a heurística construída para a pré alocação dos módulos podem ser vistos na Figura 4.37. Observe-se que os resultados medidos, ainda que inferiores aos obtidos por *annealing*, são melhores do que a simples alocação dos módulos no cromossoma, na ordem em que eles aparecem no arquivo de entrada.

4.4.8) Efeito dos Operadores de *Crossover* e Mutação:

A eficiência do operador de *crossover* no processo de otimização, apesar de fato notório em literatura sobre GAs, foi durante certo tempo no desenvolvimento desta tese, alvo de desconfianças. Esta desconfiança foi motivada, principalmente, pela falta de resultados satisfatórios durante grande parte do desenvolvimento, com as soluções convergindo, invariavelmente, para pontos de mínimos locais. A suspeita então era a de que, na fase final do algoritmo, quando é menor a diversidade genética na população,

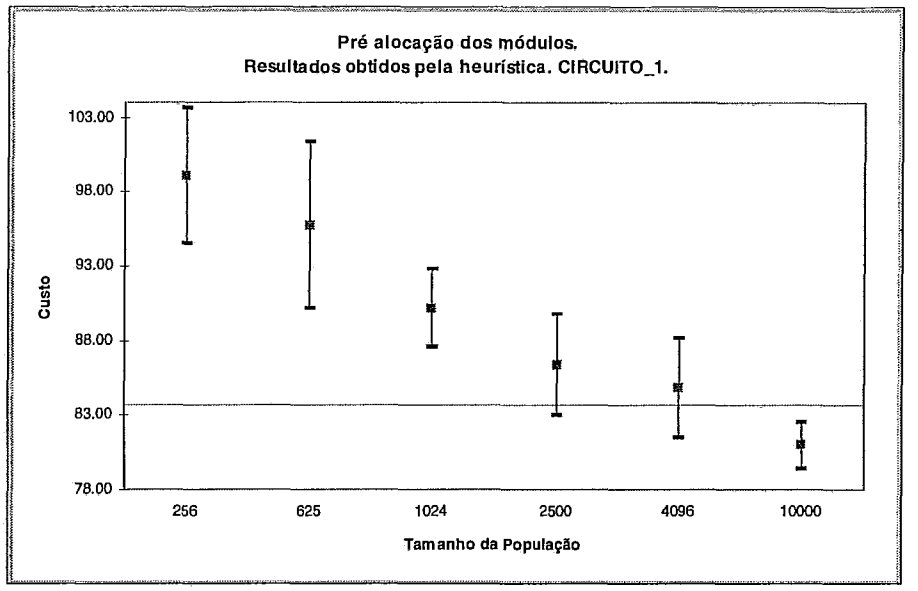


Figura 4.37 - Efeito da aplicação da heurística para a pré alocação dos módulos (CIRCUITO_1).

seria o operador de mutação o único responsável pelas diminuições na função custo. O algoritmo se comportaria então como um grande *Simulated Annealing* distribuído, com a desvantagem óbvia da necessidade de manutenção da população. Um trabalho recente de Park e Carter [Park95] corroborava nossas suspeitas sobre a ineficiência do operador de *crossover* em regiões próximas à convergência. Segundo os autores:

> "(...) In particular, we isolate the effects of cross-over, treated as the central component of genetic search. We show that for problems of nontrivial size and difficulty, the contribution of cross-over search is marginal, both synergistically when run in conjunction with mutation and selection, or when run with selection alone, the reference point being the search procedure consisting of just mutation and selection. The latter can be viewed as another manifestation of the Metropolis process. Considering the high computational cost of maintaining a population to facilitate cross-over search, its marginal benefit renders genetic search inferior to its singleton-population counterpart, the Metropolis process, and by extension, simulated annealing. This is further compounded by the fact that many problems arising in practice may inherently require a large number of state transitions for a near-optimal solution to be found, making genetic search infeasible given the high cost of computing a single iteration in the enlarged state-space."

A questão a ser respondida então era quantas vezes, durante a execução do algoritmo, cada um dos operadores obtinha sucesso em reduzir a função custo.

O algoritmo foi observado durante o terço final da execução (quando, supõe-se, a diversidade genética na população é reduzida). Os contadores são incrementados cada vez que os operadores obtém sucesso em reduzir o custo de um dos indivíduos no centro do *deme*. Os resultados são vistos na Figura 4.38. Contrário às expectativas, percebe-se que, com razoável freqüência, o operador de *crossover* obtém sucesso em melhorar os indivíduos no centro do *deme*. A primeira hipótese para explicar este fato

foi a de que, uma vez que os "filhos" gerados em um *deme* substituem incondicionalmente os indivíduos no centro, o operador de *crossover* estaria apenas "recuperando" indivíduos perdidos em gerações anteriores.

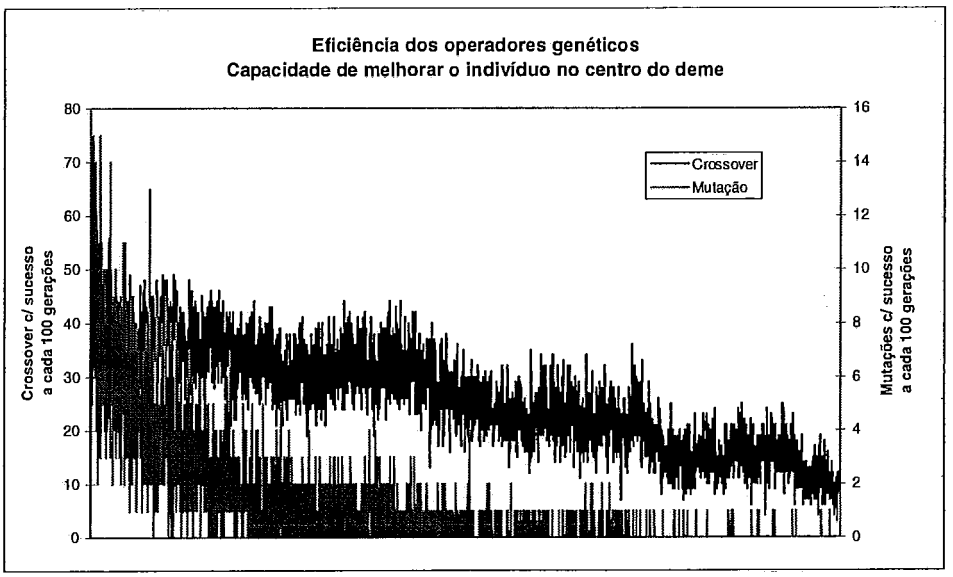


Figura 4.38 - Estudo da eficiência dos operadores genéticos: capacidade em melhorar os indivíduos no centro do *deme*.

Assim, uma vez que a estratégia elitista empregada impede que o melhor indivíduo de um *deme* seja perdido, buscou-se determinar o número de vezes que os operadores conseguem melhorar a função custo não apenas dos indivíduos do centro, mas de todo o *deme*. Os resultados são vistos na Figura 4.39

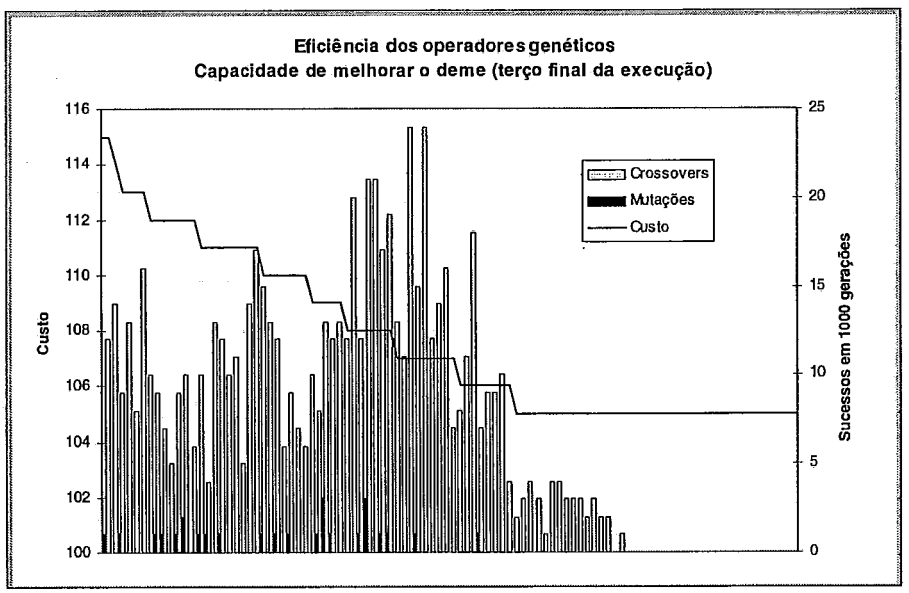


Figura 4.39 - Estudo da eficiência dos operadores genéticos: capacidade em melhorar a função custo no *deme*.

Dado ao sucesso obtido pelo operador de *crossover*, restava apenas verificar a capacidade do operador em reduzir o custo do melhor indivíduo de toda a população. Os resultados podem ser vistos na Figura 4.40.

É possível tirar algumas conclusões interessantes a partir da observação desta figura. Em primeiro lugar, o que confirma o senso comum na comunidade de algoritmos genéticos, verifica-se que é o operador de *crossover* o grande responsável pelas transformações que resultam em diminuições na função custo. Não deve-se concluir a partir daí pela inutilidade do operador de mutação. É a mutação nos genes que mantém a diversidade na população, necessária ao sucesso do operador de *crossover*.

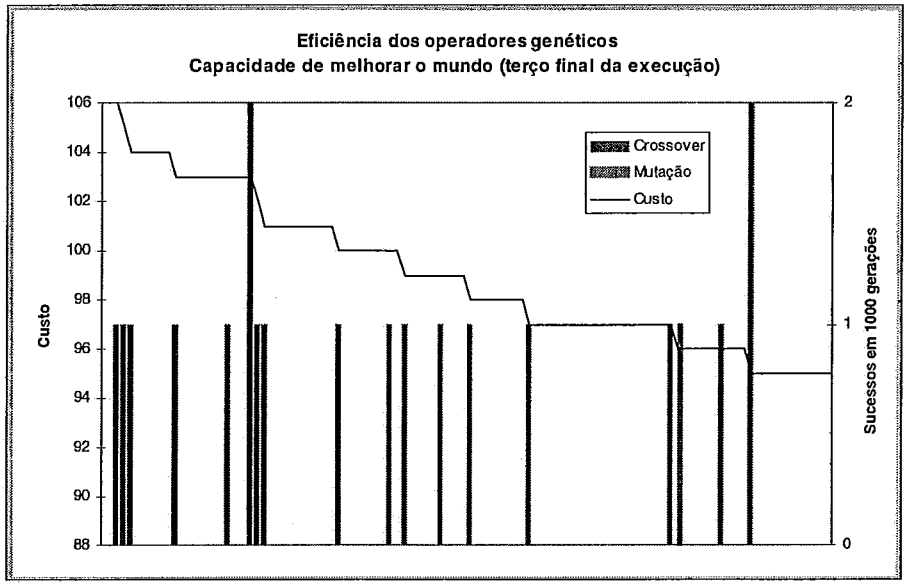


Figura 4.40 - Estudo da eficiência dos operadores genéticos: capacidade em melhorar a função custo em toda a população.

Observa-se ainda que nem toda operação de *crossover* bem sucedida resulta em diminuição da função custo. Para entender o motivo destas "reproduções perdidas" deve-se lembrar que toda operação de *crossover* é seguida por uma mutação e assim, no mesmo processo reprodutivo, o segundo operador pode "estragar" uma configuração promissora obtida por *crossover*.

A partir da observação do gráfico pode-se argumentar que, de qualquer forma, o algoritmo genético não parece ser uma alternativa "econômica" para o ajuste fino da solução: são por vezes necessárias milhares de iterações para obter-se uma diminuição na função custo de uma unidade. Isto parece estar de acordo com a observação de outros autores [Heit94, Beas93a] segundo os quais os algoritmos genéticos não deveriam ser usados em problemas onde fossem já conhecidas outras maneiras eficientes de resolvê-los. Ou, segundo palavras de Beasley *et. al.*:

> "The power of GAs comes from the fact that the technique is robust, and can deal successfully with a wide range of problem areas, including those which are difficult for other methods to solve. GAs are not guaranteed to find the global optimum solution to a problem, but they are generally good at finding "acceptably good" solutions to problems "acceptably quickly". Where specialized techniques exist for solving particular problems, they are likely to out-perform GAs in both speed and accuracy of the final result. The main ground for GAs, then, is in difficult areas where no such techniques exist. Even where existing techniques work well, improvements have been made by hybridizing them with a GA."

4.5) Estratégias Evolucionárias e outras estratégias.

Nesta seção são descritas algumas técnicas não diretamente relacionadas com algoritmos genéticos as quais foram implementadas isoladamente ou em conjunção com estes. O objetivo era o de investigar a existência de outras estratégias eficientes, baseadas na evolução de populações, que servissem de base aos esforços de paralelização do algoritmo.

4.5.1) A heurística TABU:

A heurística TABU consiste em um algoritmo guloso que visa levar a solução rapidamente a um ponto de mínimo local. A idéia consistia em, ao invés de inicializar a população do algoritmo genético com configurações aleatórias, usar a heurística desenvolvida para gerar indivíduos mais aptos. A esperança aqui era a de que o algoritmo genético, partindo já de configurações de qualidade, atingisse pontos de menor custo em um tempo de processamento menor. A heurística foi usada ainda como operador de mutação. Os detalhes de construção do algoritmo e os resultados obtidos são apresentados a seguir.

O algoritmo baseia-se no fato de que, se um módulo estabelece a fronteira de pelo menos uma rede, existe um retângulo dentro do qual o deslocamento deste módulo diminui ou, no pior caso, não altera a função custo. Chamamos a este retângulo de "retângulo de segurança".

Seja o seguinte exemplo onde um módulo *i* pertence a três redes, n_1, n_2 e n_3:

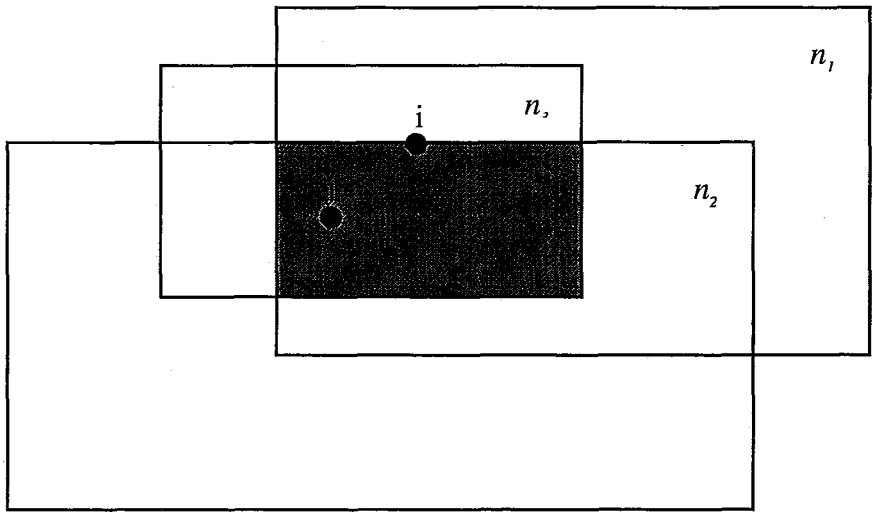


Figura 4.41 - O módulo *i* pertence simultaneamente as redes n_1, n_2 e n_3

O deslocamento do módulo *i* no interior da região sombreada (o retângulo de segurança, formado pela interseção das 3 redes) não afeta o custo das redes n_1 e n_3. No entanto, o custo da rede n_2 pode diminuir na medida em que o módulo *i* estabelece uma de suas fronteiras. Suponha, por exemplo, que o módulo *j* também pertença a rede n_2 e que ele seja o módulo mais ao norte desta rede depois de *i*. O deslocamento de *i* para o sul do retângulo de segurança teria então a seguinte conseqüência:

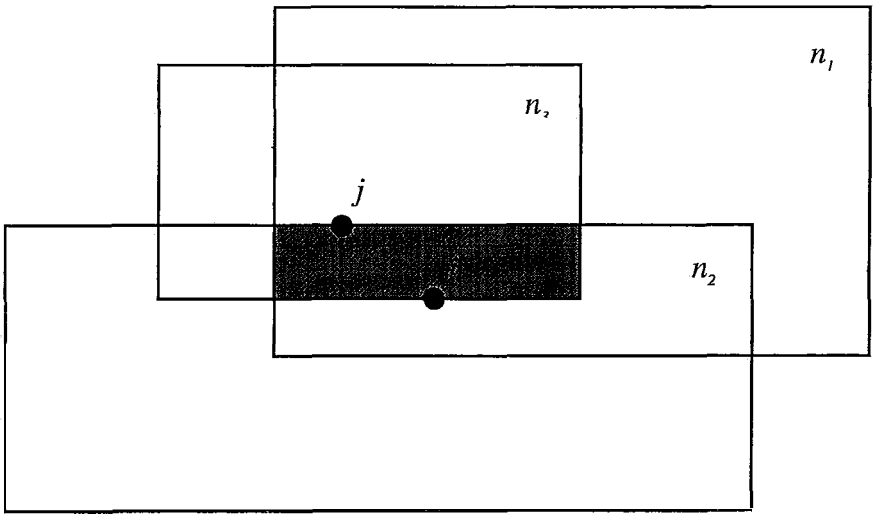


Figura 4.42 - Diminuição da função custo ocasionada pelo deslocamento do módulo *i*.

Observe-se a diminuição nas dimensões de n_2 o que, obviamente, resulta em uma redução na função custo.

De maneira genérica, todos os movimentos candidatos são avaliados. Para cada módulo que estabelece uma fronteira, é calculado o maior deslocamento permissível - o lado oposto do retângulo de segurança - e o decréscimo da função custo correspondente. O movimento que causa o maior decréscimo global na função custo é efetivado. As iterações prosseguem até que não mais seja possível diminuir a função custo (o algoritmo atinge um mínimo local). O pseudo código da heurística TABU é visto a seguir:

```
/* Cj      ⇒   Custo da configuração candidata                          */
/* MinCj   ⇒   Custo mínimo entre as configurações candidatas           */

GeraBaseDados( );          /* Gera uma solução inicial sem overlappings */
Enquanto for possível diminuir o valor da função custo  {
    Para todos os nós       {
        ClassificaNo( );    /* Verifica se o módulo estabelece fronteira de alguma rede */
                            /* Determina o retângulo de segurança                       */
        Se o módulo estabelece fronteira        {
            Move o módulo para a posição diametralmente oposta
                no retângulo de segurança;
            EliminaOverlapping( );
                            /* apenas o módulo candidato é movido                       */
            Cj = Custo da nova configuração;
                            /* apenas as redes das quais o módulo candidato faz         */
                            /* parte são recalculadas                                   */
            Se Cj < MinCj       {
                Retém a configuração candidata;
                MinCj = Cj;
            }
        }
    }
}
Imprime resultados.
```

Os primeiros resultados obtidos pela heurística não foram satisfatórios. Um estudo mais detalhado do algoritmo revelou então a situação descrita na Figura 4.43. Os módulos, mesmo que distantes uns dos outros, não são jamais candidatos a serem movidos. Isto se deve ao fato de que cada fronteira é estabelecida por mais de um módulo, resultando em que nenhuma diminuição da função custo seja obtida pelo deslocamento de qualquer um deles.

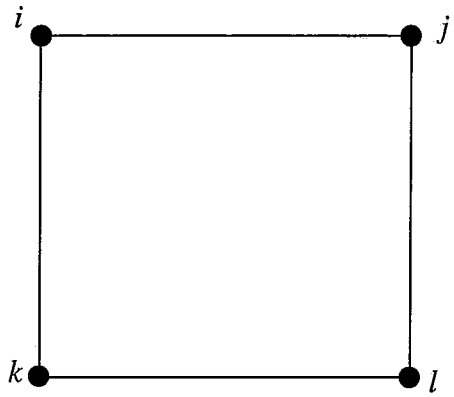


Figura 4.43 - Uma rede composta por 4 módulos

Para escapar deste "*dead lock*" implementou-se uma modificação no algoritmo original. Sempre que não for possível encontrar um candidato ao deslocamento, um módulo qualquer é tomado aleatoriamente e movido para o interior do seu retângulo de segurança. O problema em tentar escapar-se de pontos de mínimo local, mantendo-se na vizinhança da região de origem, é o risco de ciclos, isto é, a solução pode retornar ao mínimo local logo após tê-lo abandonado. De modo a evitar ciclos, o módulo movido entra em uma lista TABU de movimentos proibidos. A lista TABU é apagada se algum movimento futuro resultar em diminuição da função custo.

No exemplo da Figura 4.43, o deslocamento de algum dos módulos para o interior do retângulo de segurança (a própria rede no caso do exemplo) permitiria que, na iteração seguinte, algum dos módulos contribuísse para a diminuição da função custo (Figura 4.44).

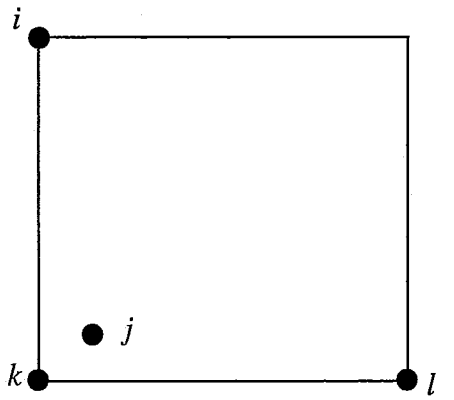


Figura 4.44 - Pelo deslocamento do módulo *j* o algoritmo tem a chance de continuar diminuindo a função custo.

A heurística foi aplicada à otimização do *placement* do CIRCUITO_1. O histograma seguinte é o resultado de 1000 execuções do algoritmo para configurações iniciais diferentes, geradas randomicamente.

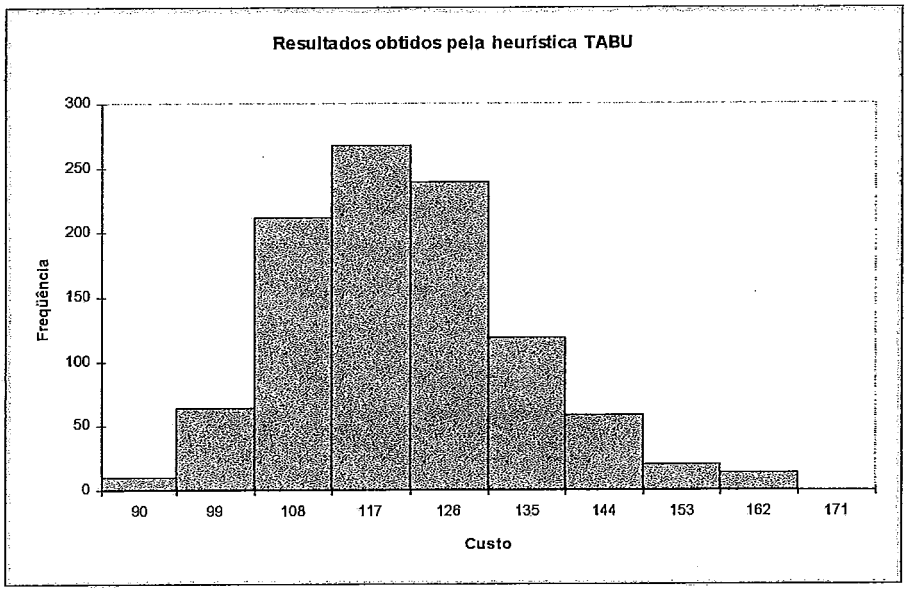


Figura 4.45 - Resultados obtidos pela heurística TABU para o *placement* de CIRCUITO_1.

A heurística obtém uma redução na função custo de em média 66% (o custo médio para as configurações geradas aleatoriamente situou-se em 356 unidades) em um tempo de processamento menor do que 1 segundo.

A heurística foi então aplicada para iniciar a população do algoritmo genético. O problema consiste em que a organização das subpopulações em *demes* precisa, para ser efetiva, de populações grandes (tipicamente alguns milhares de indivíduos). A inicialização de tais populações mostrou-se extremamente lenta (cerca de 30 minutos para uma população composta por 4096 indivíduos em uma estação IBM©25T) e terminou por anular os benefícios advindos da utilização da heurística.

A heurística foi ainda usada como operador de mutação. A cada chamada ao algoritmo, um único módulo é movido: aquele que provoca a maior diminuição na função custo segundo a heurística desenvolvida em TABU. Sempre que a heurística não trouxer resultados (o que deve ocorrer próximo à convergência do algoritmo), usa-se o operador de mutação convencional (*swaps* e *displacements*) em seu lugar. Não houve qualquer alteração significativa no comportamento do algoritmo pela utilização do novo operador de mutação. Isto se deve provavelmente ao fato de que, próximo à convergência, a heurística tem dificuldade em gerar deslocamentos que causem diminuições na função custo e, então, o algoritmo confia no operador tradicional de mutação para manter a diversidade da população.

4.5.2) Aprendizado Incremental Baseado em Populações (AIBP):

Esta implementação é baseada em um trabalho de Shumeet Baluja e Rich Caruana [Balu95].

O algoritmo de Aprendizado Incremental é uma combinação de estratégias evolucionárias e de métodos baseados na descendente máxima. O objetivo do algoritmo

é criar um vetor de probabilidades para os valores das variáveis do problema, que seja uma matriz para a geração de indivíduos de alta qualidade. Seja por exemplo um problema de 3 variáveis: x, y e z. Uma matriz geradora de soluções para este problema é dada por $\left[\bar{x}, \sigma_x, \bar{y}, \sigma_y, \bar{z}, \sigma_y\right]$, onde σ é o desvio padrão das distribuições (Figura 4.46). Se a solução ótima do problema é dada por $[x_0, y_0, z_0]$, um bom valor final para a matriz geradora seria dado por $\left[0.99x_0, 0, 1.01y_0, 0, 0.99z_0, 0\right]$.

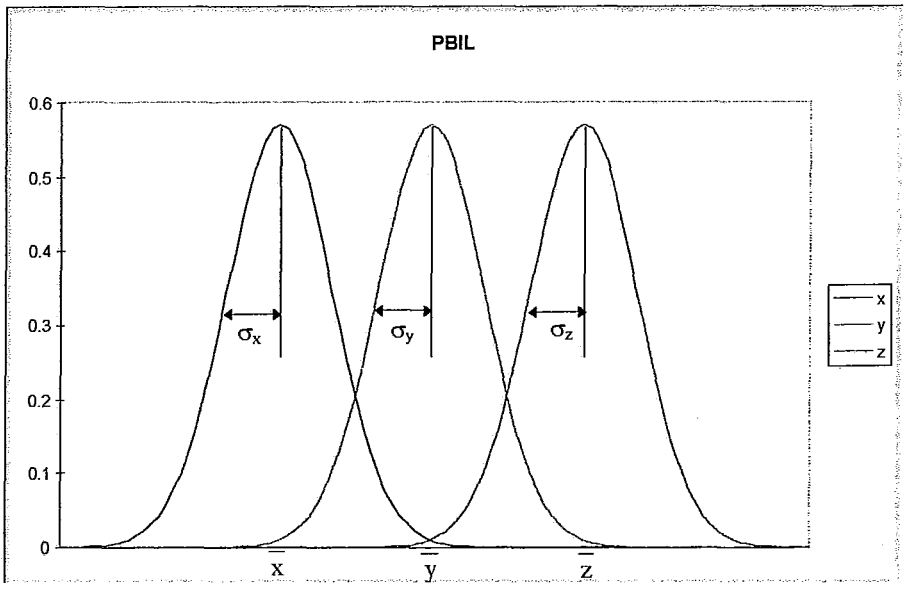


Figura 4.46 - Princípio de funcionamento do algoritmo de aprendizado incremental.

Para o problema de *placement*, as variáveis do problema são cada uma das coordenadas x e y dos módulos no circuito. Inicialmente os valores no vetor gerador são inicializados com média igual ao centro do plano de alocação e desvio padrão igual às dimensões do plano. Desta forma, no início do processamento, os módulos têm boas chances de serem gerados em qualquer lugar do plano. À medida em que o processo de busca prossegue, o desvio padrão das distribuições é reduzido e, idealmente, as médias movem-se para o entorno dos valores ótimos. A implementação do algoritmo AIBP se dá da seguinte forma: Uma população de indivíduos (soluções potenciais para o problema) é gerada usando distribuições normais com as médias e desvios padrões especificados no vetor gerador. Em cada geração, as médias das variáveis no vetor gerador são deslocadas na direção dos indivíduos mais aptos. O valor do deslocamento das médias em cada geração é dado pela taxa de aprendizado L. Depois que o vetor gerador é atualizado, uma nova população de indivíduos é gerada (usando o novo vetor gerador) e o processo continua.

O algoritmo AIBP é caracterizado por 3 parâmetros. O primeiro é o número de indivíduos em cada uma das populações. Nesta implementação, as populações foram mantidas constantes em 200 indivíduos. O segundo parâmetro é a taxa de aprendizado (L), que define o tamanho dos passos em direção às boas soluções. A taxa de aprendizado foi também mantida constante em 0,05. O terceiro parâmetro é o número de indivíduos a considerar na atualização do vetor gerador. Nesta implementação somente o melhor indivíduo da população foi usado para atualizar o vetor gerador em cada geração.

- Pseudo código:

```
/* L              ⇒   Taxa de aprendizagem                                  */
/* Melhor_k       ⇒   Indivíduo mais apto na geração atual                  */
/* Vetor          ⇒   Vetor gerador                                         */

Inicializa o vetor gerador;          /* Médias iguais ao centro do plano de alocação.   */
                                     /* Desvios padrões iguais às dimensões do plano.   */
Repete por T gerações          {
      GeraPopulação( );              /* Gera nova população com 200 indivíduos.         */

      Seleciona Melhor_k;            /* Seleciona o melhor indivíduo da geração         */

      Para cada uma das médias i no vetor gerador, faz:
            Vetor[i] = Vetor[i]*(1.0 - L) + Melhor_k[i]*L;

      σ = 0,95*σ;                    /* Reduz o desvio padrão das distribuições.        */
}
Imprime resultados.
```

- Resultados:

Os resultados obtidos com o algoritmo AIBP foram decepcionantes. O valor da função custo é várias vezes maior do que o obtido pelo *Simulated Annealing*, para o mesmo circuito. Não chegaram a ser feitos testes exaustivos para avaliar a influência do tamanho da população e da taxa de aprendizado sobre o comportamento do algoritmo.

4.6) Conclusões.

As principais conclusões e contribuições decorrentes da busca por um algoritmo genético eficiente para o problema de *placement* foram as seguintes:

- O algoritmo genético seqüencial, para produzir resultados com qualidade semelhante ao algoritmo de *Simulated Annealing*, apresenta um custo computacional significativamente maior.
- O ajuste do algoritmo genético seqüencial ao problema de *placement* foi um problema não trivial, tendo requerido experimentações diversas e o uso de estratégias pouco discutidas na literatura. Esta série de testes exaustivos com algoritmos não convencionais nos deu a oportunidade de comprovar a baixa eficiência (ao menos para o problema de *placement*) de uma série de técnicas descritas em literatura. Entre estas técnicas, podemos destacar: populações de tamanho variável [Mich94], o operador de inversão e o *crossover* cíclico [Mazu93], algoritmos híbridos: genético + *annealing* [Gold93], *Linear Ranking* [Bake85] e Codificação Expandida [Beas93c].
- Para um bom desempenho do algoritmo genético seqüencial, foi fundamental a utilização da técnica do equilíbrio pontual. A população é disposta na forma de um toróide e organizada, logicamente, em subpopulações. Os indivíduos tomam parte, simultaneamente, em mais de uma subpopulação. Cada subpopulação trabalha sobre os seus cromossomas independentemente de todas as outras subpopulações. Esta

estrutura permite a transferência gradual de material genético entre as subpopulações. A motivação desta organização é a esperança de que subpopulações separadas por uma grande distância evoluam para cadeias genéticas significativamente diferentes e que toda a população se beneficie do espalhamento suave de material genético que cause refinamentos na função custo.

- A organização utilizada da população tem o efeito de provocar trocas freqüentes de material genético entre subpopulações de tamanho reduzido (10 indivíduos). Desta forma, para evitar que o algoritmo convirja para pontos de mínimo local, é necessário trabalhar-se com tamanhos de população significativamente grandes. Para os circuitos testados, os melhores resultados foram obtidos com populações de 5000 ou 10000 indivíduos.
- O problema de *placement* caracteriza-se por uma forte epistasia entre os genes do cromossoma. Devido a este fato, o operador de *crossover* tradicional tem um efeito destrutivo na medida em que ele, potencialmente, destrói sub *placements* promissores provenientes de um dos pais. De modo a lidar com este problema alguns operadores genéticos especiais (*crossover* por herança de redes e *crossover* por herança de regiões do plano) foram implementados mas os melhores resultados foram obtidos por uma heurística de pré-alocação dos genes no cromossoma. Em essência, busca-se minimizar o número de redes seccionadas pela partição do cromossoma em um ponto arbitrário, ou, em outras palavras, busca-se minimizar o número de redes que tenham módulos simultaneamente em ambos os "pedaços" do cromossoma resultantes do *crossover.*

CAPÍTULO 5

Paralelização do Algoritmo Genético

5.1) Introdução:

O esquema das populações paralelas visto no Capítulo anterior precisa, para ser efetivo, de populações de tamanho consideravelmente maior do que os algoritmos genéticos tradicionais. Isto deve-se ao fato de que populações maiores levam a soluções melhores devido à maior diversidade existente na população. Populações grandes devem ser avaliadas por muitas gerações (deve-se recordar que geração, no nosso contexto, refere-se à criação de dois novos indivíduos em um *deme*) o que torna o algoritmo genético lento quando comparado à alternativa do *Simulated Annealing*. Uma alternativa imediata para este problema é a paralelização do algoritmo buscando encontrar soluções de melhor qualidade ou em um tempo menor de processamento.

Os algoritmos genéticos são procedimentos altamente paralelos o que, em princípio, permite uma implementação razoavelmente simples em qualquer tipo de arquitetura. No entanto, é necessário, como em qualquer aplicação, definir uma estratégia que combine a paralelização dos procedimentos e/ou dados. Uma possível estratégia seria ter os dados centralizados no mestre e os procedimentos distribuídos pelos processadores escravos, com o mestre executando a fase de seleção e mandando pares de indivíduos para os processadores escravos. Os escravos receberiam os pares, transformariam-nos usando os operadores genéticos, computando a função custo dos indivíduos gerados e, finalmente, retornando os novos indivíduos para o processador mestre. Esta estratégia básica, na maior parte das vezes, não é efetiva em arquiteturas de memória distribuída: o mestre representa um gargalo no sistema; uma fração das computação é seqüencial (a fase de seleção); e a granularidade do processo é muito fina gerando comunicação excessiva entre os processadores. O custo de comunicação é um dado ainda mais crítico em nosso ambiente de programação (um *cluster* de estações de trabalho comunicando-se por troca de mensagens usando TCP/IP sobre barramentos Ethernet) e, na prática, limita as alternativas de paralelização que podem ser empregadas. A solução aponta no sentido de populações distribuídas, onde os processadores manipulam grupos menores de indivíduos e computam por um certo número de gerações entre trocas de mensagens.

Neste Capítulo são apresentadas duas alternativas de paralelização usando esta filosofia: um modelo centralizado onde, de tempos em tempos, toda a população é centralizada em um processador mestre e um modelo distribuído, possivelmente assíncrono, onde os indivíduos, periodicamente, migram de um processador para outro.

5.2) O modelo centralizado:

Nesta implementação os processadores são organizados na forma mestre-escravos. A população é distribuída entre os processadores que computam, no seu pedaço da população, todos os passos do algoritmo genético serial consistindo de seleção, reprodução e *tournament*. Periodicamente a população é reunida no processador mestre. O mestre embaralha a população e a redistribui entre os escravos.

O número de operações de comunicação pode ser feito tão pequeno quanto se queira, aumentando-se o número de gerações entre sincronizações. Uma vez que o tamanho das subpopulações em cada um dos processadores é, em geral, muito pequeno para que o esquema do equilíbrio pontual seja efetivo, o número de sincronizações representa um compromisso entre velocidade de processamento e a qualidade da solução gerada.

- Pseudo código (processador mestre):

```
/* nhost      ⇒      número de processadores na máquina virtual                    */

Dispara os processos escravos;

Genese ( );                    /* Gera população inicial de POPSIZE/nhost indivíduos.  */

Repete por T gerações          {
        Repete até sincronização   {
              Reprodução( );
        }
/*      SINCRONIZAÇÃO:                                                              */
        Recebe as sub populações dos escravos;
        Embaralha a população resultante;
        Redistribui a população embaralhada para os escravos;
}
Imprime resultados.
```

O procedimento *Reprodução*( ) é dado por:

```
/* Melhor_{k-1}  ⇒   Indivíduo mais fitted na geração anterior.                     */
/* Pior_k        ⇒   Indivíduo menos fitted na geração atual.                       */

EscolheDeme( );              /* Aleatoriamente escolhe um deme na população         */
                             /* Forma uma sub população de 10 indivíduos.           */
Pior_k = Melhor_{k-1};       /* Elitismo. Substitui o pior indivíduo desta geração  */
                             /* pelo melhor da geração anterior.                    */
Evaluation( );               /* Avalia a aptidão dos indivíduos no deme.            */
                             /* Transformação bilinear.                             */
Pai1 = Seleção( );           /* Escolhe o primeiro pai. Método da roleta.           */
Pai2 = Seleção( );           /* Escolhe o segundo pai.                              */

Gera 2 filhos segundo o esquema: {
        Crossover(Pai1, Pai2); /* Crossover de 2 pontos.                            */
        EliminaOverlapping( ); /* Elimina sobreposições.                            */
        Mutação( );            /* Mutação probabilística.                           */
}
Tournament( );               /* Os 2 indivíduos recém gerados substituem            */
                             /* os indivíduos no centro do deme.                    */
```

- Detalhes de implementação:

1. Freqüência de sincronizações

Cada uma das posições da grade de indivíduos (Vide seção 4.3.2) tem a chance de ser selecionada K vezes entre sincronizações. Assim, o número de gerações entre sincronizações (n_g) é dado por:

$$n_g = K * POPSIZE / nhost \quad (5.1)$$

Nos experimentos realizados o valor de K foi variado de 1 até 10.

2. Embaralhamento da população.

O embaralhamento da população se dá no processador mestre. Um vetor *pool*[ ] é ordenado segundo um campo *idx* atribuído randomicamente no intervalo [0, 10**POPSIZE*]. A operação de ordenação é otimizada pelo fato de que o vetor *pool*[ ] não contém os cromossomas propriamente ditos, mas apenas ponteiros para estes. Assim, as pesadas estruturas de dados representando os cromossomas não precisam ser movidas durante a ordenação. O código da operação de embaralhamento é visto a seguir.

```
/*
Funções:
        rnd(n)      ⇒     Devolve um número inteiro no intervalo [0, n)
        qsort( )    ⇒     Quick Sort
*/

for(Paux=pool, i=0; i<POPSIZE; i++)        {
        Paux->idx=rnd(10*POPSIZE);
        Paux++;
}
qsort((void *)pool, POPSIZE, sizeof(struct Nicho), (pint)sort_pool);
```

- Resultados:

O algoritmo não apresentou bons resultados para qualquer valor do período de sincronização K. Para valores pequenos de K a execução do algoritmo é muito lenta. Pode-se entender este fato pela análise do número de *bytes* transferidos em cada operação de sincronização. Sejam:

$nhost = 4$	⇒	Número de processadores na máquina virtual;
$POPSIZE = 625$	⇒	Tamanho das sub populações;
$n_c = 3$	⇒	Número de cromossomas em cada posição da grade (Vide seção 4.3.2);
$nmodulos = 80$	⇒	número de módulos para o problema de teste;
$SizeGene = 2$	⇒	tamanho em *bytes* de cada gene;

O número de *bytes* transferidos em cada operação de sincronização é, então, dado por:

$$nbytes = 2*(nhost-1)*POPSIZE*n_c*nmodulos*SizeGene; \quad (5.2)$$

$$nbytes = 1800 \text{ Kbytes}; \quad (5.3)$$

O fator multiplicativo de 2 refere-se ao fato de que os dados devem ser enviados ao mestre e deste redistribuídos aos escravos. O fator (*nhost*-1) refere-se ao fato de que o mestre também participa do processamento e, portanto, uma parte dos dados não precisa ser transmitida pela rede. A quantidade de dados transmitida pela rede parece ser excessiva para o ambiente utilizado.

Para valores maiores de K observa-se uma redução no tempo de processamento ao custo, porém, de uma perda de qualidade na solução obtida. Pode-se entender este fato imaginando-se a situação em que os processadores não se comunicam durante o procedimento de cálculo. Cada um deles estaria trabalhando então sobre um pedaço por demais reduzido da população, acarretando a convergência para um mínimo local. A solução parece então apontar para o modelo distribuído, que será visto na próxima seção.

5.3) O modelo distribuído:

Como vimos na seção anterior, a centralização da população no processador mestre implica em uma grande quantidade de dados circulando pela rede o que ocasiona um gargalo no processamento. Na implementação distribuída elimina-se a figura do mestre. Cada processador tem um pedaço da população total e roda o algoritmo genético sobre o seu conjunto de indivíduos, do início ao fim do processamento. A comunicação acontece periodicamente quando os processadores mandam alguns dos seus indivíduos para os vizinhos. Os indivíduos recebidos são imediatamente incluídos pelo processador hospedeiro na população local. A esperança aqui é a de que o conjunto das subpopulações possa beneficiar-se pela migração de indivíduos potencialmente *fitted.*

Cada um dos processadores roda um algoritmo genético baseado no modelo paralelo descrito na seção 4.3.2. Neste modelo a população é organizada em *demes*. Em cada *deme* dois indivíduos são selecionados para reprodução, a cada geração. Estes indivíduos são escolhidos em uma população de 10 cromossomas: os dois cromossomas no centro do *deme* e 1 cromossoma de cada um dos vizinhos ao Norte, Nordeste, Leste, Sudeste, Sul, Sudoeste, Oeste e Noroeste. Cada um dos cromossomas provenientes dos vizinhos é escolhido aleatoriamente entre os dois disponíveis. Os dois cromossomas selecionados geram dois "filhos" (por *crossover* e mutação) que vão substituir os indivíduos no centro do *deme.* Cada posição da grade exporta indivíduos para completar a população dos *demes* centrados nas posições vizinhas, mas não existe o fluxo de dados em sentido inverso. Teoricamente, em uma máquina SIMD maciçamente paralela, seria possível atribuir um processador a cada posição da grade e substituir toda a população em um único passo, efetuado de forma síncrona por todos os processadores. Esta observação deu margem à nossa estratégia de paralelização. As regiões de fronteira são duplicadas em cada um dos processadores e, periodicamente, atualizadas (Figura 5.1).

Observe que as posições hachuradas no processador ao centro correspondem a uma área apenas de leitura, isto é, os *demes* centrados nestas posições não são, em tempo algum, selecionados para reprodução. A reduzida necessidade de comunicação representa uma característica muito interessante deste algoritmo: apenas os indivíduos na fronteira das áreas atribuídas aos processadores precisam ser transferidos. Além disso, uma vez que o instante de recebimento de uma fronteira não é crítico no desempenho do algoritmo, a comunicação entre os processos pode ser feita assincronamente. É óbvio que a migração de indivíduos deve produzir melhores resultados quando os processadores encontram-se em estágios semelhantes do processo

de otimização a fim de que um indivíduo exportado não domine a população hospedeira nem por esta seja dominado. Desta forma, a estratégia assíncrona deve ser empregada, preferencialmente, em arquiteturas compostas por processadores homogêneos e dedicados. Quando este não for o caso pode-se forçar a sincronização através do mecanismo de barreiras.

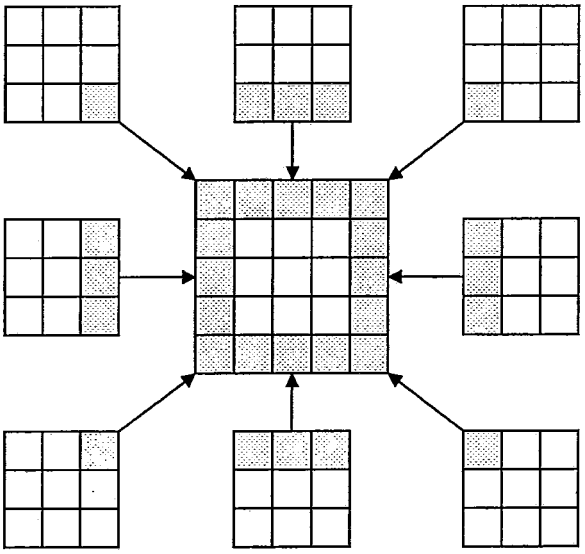


Figura 5.1 - Estratégia de migração de indivíduos no algoritmo distribuído.

De forma análoga à estratégia centralizada, o número de operações de comunicação pode ser feito tão pequeno quanto se queira, representando um compromisso entre a qualidade da solução e o tempo de processamento. Nos testes realizados, cada uma das posições da grade tem a chance de ser selecionada uma vez entre sincronizações. Assim, o número de gerações entre sincronizações (n_g) é dado por:

$$n_g = POPSIZE \,/\, nhost \tag{5.4}$$

- Pseudo código (todos os processadores):

```
/* nhost    ⇒    número de processadores na máquina virtual                    */

Genese ( );      /* Gera população inicial de POPSIZE/nhost indivíduos.        */

Repete por T gerações        {
        Repete até envio de dados      {
                Recepção( );
                Reprodução( );
        }
/*      ENVIO DE DADOS:                                                        */
        EnviaFronteira(NW);
        Envia Fronteira(N);
        EnviaFronteira(NE);
        EnviaFronteira(E);
        EnviaFronteira(SE);
        EnviaFronteira(S);
        EnviaFronteira(SW);
        EnviaFronteira(W);
}
```

O procedimento *Reprodução*( ) é idêntico àquele apresentado na seção anterior. O procedimento *Recepção*( ) verifica a existência de dados a serem recebidos e sua procedência a fim de determinar a posição onde eles devem ser inseridos.

- Resultados:

Nos testes foi utilizado um *cluster* de estações IBM®25T dedicadas, isto é, nenhum outro processo encontrava-se em execução nas máquinas. Oito processadores interligados através de uma rede Ethernet estavam disponíveis para os testes. O algoritmo foi testado para um problema pequeno (CIRCUITO_1: 80 módulos, 30 redes equipotenciais), um problema médio (CIRCUITO_2: 100 módulos, 300 redes) e para um problema maior (CIRCUITO_3: 1024 módulos; 3000 redes). Os resultados podem ser vistos nas tabelas a seguir. Os resultados do *annealing* serial foram incluídos para fins de comparação. *N* é o número de processadores na máquina paralela virtual.

CIRCUITO_1, POPSIZE = 1024 indivíduos.				
	Genético Serial	Paralelo, $N = 4$	Paralelo, $N = 8$	*Annealing* Serial
Custo Médio	86,27	83,20	82,10	83,67
Desvio Padrão	3,14	2,91	1,98	3,34
tempo(s)	170,49	68,13	59,57	26,12
Speedup	--	2,50	2,86	--

Tabela 5.1 - Resultados para o CIRCUITO_1. População = 1024 indivíduos.

CIRCUITO_1, POPSIZE = 2500 indivíduos.				
	Genético Serial	Paralelo, $N = 4$	Paralelo, $N = 8$	*Annealing* Serial
Custo Médio	83,36	83,70	83,50	83,67
Desvio Padrão	1,67	2,45	2,15	3,34
tempo(s)	416,14	144,67	104,04	26,12
Speedup	--	2,88	4,00	--

Tabela 5.2 - Resultados para o CIRCUITO_1. População = 2500 indivíduos.

CIRCUITO_2. POPSIZE = 1024 indivíduos.				
	Genético Serial	Paralelo. $N = 4$	Paralelo. $N = 8$	*Annealing* Serial
Custo Médio	1926,80	1931,27	1916,82	1915,00
Desvio Padrão	14,68	19,94	13,68	28,00
tempo(s)	960,43	277,92	163,77	89,94
Speedup	--	3,46	5,86	--

Tabela 5.3 - Resultados para o CIRCUITO_2. População = 1024 indivíduos.

CIRCUITO_2. POPSIZE = 2500 indivíduos.				
	Genético Serial	Paralelo. $N = 4$	Paralelo. $N = 8$	*Annealing* Serial
Custo Médio	1900,00	1903,10	1921,90	1915,00
Desvio Padrão	9,75	15,40	11,70	28,00
tempo(s)	2347,28	662,44	388,05	89,94
Speedup	--	3,54	6,05	--

Tabela 5.4 - Resultados para o CIRCUITO_2. População = 2500 indivíduos.

CIRCUITO_3.				
	Genético Serial	Paralelo. $N = 4$	Paralelo. $N = 8$	*Annealing* Serial
Custo Médio	116076,00	--	--	61929,00
Desvio Padrão	977,83	--	--	1193,00
tempo(s)	52546,02	--	--	12673,11
Speedup	--	--	--	--

Tabela 5.5 - Resultados para o CIRCUITO_3.

Alguns resultados tornam-se evidentes a partir da observação das Tabelas acima:

- Para circuitos pequenos, o ganho em tempo de processamento advindo do paralelismo é limitado pelo *overhead* devido à troca de mensagens. Observe-se na Tabela 5.1 o reduzido benefício em alocar-se 8 processadores à máquina virtual.

- Observa-se o efetivo sucesso da estratégia distribuída. A solução paralela mantém a qualidade da versão serial proporcionando ainda um *speedup* significativo no tempo de processamento. Estes bons resultados devem-se, principalmente, aos seguintes fatores:

  1. A reduzida necessidade de comunicação entre os processadores.
  2. A solução ótima é buscada em paralelo.
  3. O tamanho da população alocada a cada processador é significativamente menor do que o mínimo necessário a se obter soluções próximas da ótima no algoritmo serial.

- Mesmo as versões paralelas do algoritmo genético foram incapazes de igualar os tempos de processamento obtidos pelo *annealing* serial. Este é um resultado notável e parece indicar o problema de *placement* como um nicho não apropriado aos algoritmos genéticos.

- O tempo de processamento da algoritmo genético paralelo poderia ser reduzido pela adição de processadores à máquina virtual. Deve-se considerar no entanto, que o esquema do equilíbrio pontual não é efetivo para subpopulações muito pequenas. Este fato forçaria o aumento do número de migrações de indivíduos (com o

conseqüente aumento no número de mensagens trocadas) o que poderia anular os benefícios advindos do maior número de processadores.

- O algoritmo genético (versão serial) foi incapaz de reproduzir os resultados obtidos pelo *annealing* para o CIRCUITO_3. Este resultado ainda precisa ser melhor investigado. Por ser um circuito muito grande (1024 módulos, 3000 redes equipotenciais) pode ser necessário utilizar um número muito maior de indivíduos na população e/ou de gerações. O problema consiste em que tal solução, ainda que trouxesse bons resultados para a função custo, dificilmente seria efetiva em comparação com o *annealing*. Seriam necessários, provavelmente, vários dias de CPU para completar o cálculo.

CAPÍTULO 6

Conclusões

6.1) Contribuições:

Este trabalho de tese compreendeu um estudo experimental detalhado dos algoritmos Genético e de *Simulated Annealing*, tendo em vista a aplicação ao problema de *placement* de macrocélulas em circuitos VLSI. Tendo em vista que ambos os algoritmos são capazes de encontrar a solução ótima global do problema a um custo computacional elevado, este trabalho de tese estudou também diferentes técnicas de paralelização destes algoritmos, considerando uma plataforma formada por um *cluster* de estações de trabalho e o uso do ambiente de programação paralela PVM. As principais conclusões e contribuições decorrentes deste estudo experimental foram as seguintes:

- O algoritmo genético seqüencial, para produzir resultados com qualidade semelhante ao algoritmo de *simulated annealing*, apresenta um custo computacional significativamente maior. De fato, para um circuito composto por 100 módulos e 300 redes equipotenciais, o tempo de processamento gasto pelo algoritmo genético paralelo é maior do que o tempo correspondente na versão serial do *annealing*. Este é um resultado notável e parece confirmar a observação de outros autores [Beas93a, Heit94] segundo os quais os algoritmos genéticos não deveriam ser usados em problemas onde fossem já conhecidas outras maneiras eficientes de resolvê-los.

- O ajuste do algoritmo genético seqüencial ao problema de *placement* foi um problema não trivial, tendo requerido experimentações diversas e o uso de estratégias pouco discutidas na literatura. Esta série de testes exaustivos com algoritmos não convencionais nos deu a oportunidade de comprovar a baixa eficiência (ao menos para o problema de *placement*) de uma série de técnicas descritas em literatura. Entre estas técnicas, podemos destacar: populações de tamanho variável [Mich94], o operador de inversão e o *crossover* cíclico [Mazu93], algoritmos híbridos: genético + *annealing* [Gold93], *Linear Ranking* [Bake85] e Codificação Expandida [Beas93c].

- Para um bom desempenho do algoritmo genético seqüencial, foi fundamental a utilização da técnica do equilíbrio pontual. A população é disposta na forma de um toróide e organizada, logicamente, em subpopulações. Os indivíduos tomam parte, simultaneamente, em mais de uma subpopulação. Cada subpopulação trabalha sobre os seus cromossomas independentemente de todas as outras subpopulações. Esta estrutura permite a transferência gradual de material genético entre as populações. A motivação desta organização é a esperança de que subpopulações separadas por uma grande distância evoluam para cadeias genéticas significativamente diferentes e que toda a população se beneficie do espalhamento suave de material genético que cause refinamentos na função custo.

- A organização utilizada da população tem o efeito de provocar trocas freqüentes de material genético entre subpopulações de tamanho reduzido (10 indivíduos). Desta forma, para evitar que o algoritmo convirja para pontos de mínimo local, é necessário trabalhar-se com tamanhos de população significativamente grandes. Para

os circuitos testados, os melhores resultados foram obtidos com populações de 5000 ou 10000 indivíduos.

- O problema de *placement* caracteriza-se por uma forte epistasia entre os genes do cromossoma. Devido a este fato, o operador de *crossover* tradicional tem um efeito destrutivo na medida em que ele, potencialmente, destrói sub *placements* promissores provenientes de um dos pais. De modo a lidar com este problema alguns operadores genéticos especiais (*crossover* por herança de redes e *crossover* por herança de regiões do plano) foram implementados mas os melhores resultados foram obtidos por uma heurística de pré-alocação dos genes no cromossoma. Em essência, busca-se minimizar o número de redes seccionadas pela partição do cromossoma em um ponto arbitrário, ou, em outras palavras, busca-se minimizar o número de redes que tenham módulos simultaneamente em ambos os "pedaços" do cromossoma resultantes do *crossover*.

- O algoritmo genético foi facilmente paralelizável e produziu, para problemas de *placement* de tamanho médio, um *speedup* aproximadamente linear com o número de estações de trabalho incorporadas ao *cluster*. Este sucesso deve-se, principalmente, aos seguintes fatores:

    1. A reduzida necessidade de comunicação entre os processadores.
    2. A solução ótima é buscada em paralelo.
    3. O tamanho da população alocada a cada processador é significativamente menor do que o mínimo necessário a se obter soluções próximas da ótima no algoritmo serial.

- A paralelização do algoritmo de *simulated annealing* não é trivial e, no ambiente de teste utilizado, produz *speedup* significativo apenas para problemas de tamanho muito grande (*speedup* de aproximadamente 3 para um circuito composto por 1024 nós e 3000 redes equipotencias, utilizando 10 processadores na máquina paralela).

- O algoritmo de Kravitz e Rutenbar [Krav87], freqüentemente citado em literatura como uma alternativa viável para a paralelização do *annealing*, é equivocado. O algoritmo baseia-se na idéia do conjunto serializável mínimo onde, em cada passo, todos os estados rejeitados e um único estado aceito são contabilizados no sentido do equilíbrio. Verificou-se que esta estratégia não preserva a proporção entre estados aceitos/rejeitados verificada no algoritmo serial e, por este motivo, as soluções geradas para o algoritmo paralelo são de baixa qualidade.

- O parâmetro temperatura, que controla os movimentos de *hill-climbing* no *Simulated Annealing*, altera profundamente o comportamento do algoritmo durante a sua execução. Este comportamento dinâmico fornece novas oportunidades de explorar o paralelismo inerente ao processo. A estratégia de paralelização empregada consiste em mudar parâmetros de um algoritmo, ou mesmo o algoritmo utilizado de acordo com a fase do processo.

- A estratégia de paralelização empregada para o *annealing* em baixas temperaturas consiste em uma variante do algoritmo especulativo proposto na literatura [Sohn95]. O algoritmo tem um comportamento inteiramente dinâmico onde o número de processadores integrantes da arquitetura paralela e o número de estados avaliados por processador entre sincronizações variam com a temperatura. Durante o domínio do algoritmo (probabilidade de aceitação de novos estados menor do que 20%), para um

máximo de 10 processadores na máquina paralela, e um circuito composto por 1024 nós e 3000 redes, foi possível obter um *speedup* de 3,87 unidades.

- Para o algoritmo em temperaturas altas, partiu-se da constatação de que, para cadeias de mesmo comprimento, o algoritmo paralelo atinge o ponto de troca de estratégias com uma configuração de mais baixo custo e, possivelmente, em uma temperatura mais alta. A partir daí divisou-se uma estratégia onde é possível obter ganho em velocidade de processamento a partir da redução do tamanho das cadeias. Durante o domínio do algoritmo (probabilidade de aceitação de novos estados maior do que 20%), 10 processadores na máquina paralela, e um circuito composto por 1024 nós e 3000 redes, foi possível obter um *speedup* de 2,88 unidades.

6.2) Trabalhos Futuros:

Uma limitação importante da presente versão, que pretende-se remover no futuro, é o fato de que o *placement* a ser otimizado é composto por módulos de mesmas dimensões. Além disso, o programa não provê o *placement* dos pinos no interior dos módulos. Adicionalmente, a função custo deveria incorporar alguma medida da facilidade de roteamento advinda do *placement* gerado.

Como continuação deste trabalho pretende-se investigar outras classes de algoritmos aplicados ao problema de *placement.* Entre essas classes podemos citar a lógica TABU [Glov90] e os Times Assíncronos [Souz93].

Uma outra evolução prevista para este trabalho é a implementação e análise de desempenho dos algoritmos genéticos e *simulated annealing* nos ambientes MPVM [Sant96], PVM Padrão e MULPLIX [Aude96] nativo a serem disponibilizados em breve no multiprocessador MULTIPLUS [Aude96] de memória distribuída compartilhada em desenvolvimento no NCE/UFRJ.

O ambiente MPVM é uma implementação do ambiente PVM no multiprocessador MULTIPLUS que mapeia *tasks* concorrentes do PVM em *threads* do sistema operacional MULPLIX e utiliza o compartilhamento de memória entre *threads* para implementar as funções de troca de mensagens. O ambiente PVM Padrão será implementado com o mapeamento de *tasks* PVM em "processos" MULPLIX e utilizará o compartilhamento de memória entre processos para a implementação das primitivas de troca de mensagens. Este ambiente será totalmente compatível com o ambiente PVM de domínio público enquanto que o ambiente MPVM apresentará pequenas diferenças para o usuário. No ambiente MULPLIX nativo, o usuário trabalha diretamente com o paradigma de memória compartilhada e possui maiores facilidades para explorar de forma conveniente a arquitetura do tipo NUMA (*Non-Uniform Memory Access*) do multiprocessador MULTIPLUS.

Espera-se que deste exercício de paralelização em três ambientes distintos, não só novas propostas de implementação dos algoritmos resultem como decorrência da busca de maior desempenho, como também alguns gargalos da plataforma MULTIPLUS/MULPLIX sejam detectados e eventualmente removidos.

Bibliografia
CAPÍTULO 7

[Aart87] Aarts, E.H.L. and Van Laarhoven, P.J.M., *Simulated Annealing: Theory and Applications*, D.Riedel, Dordrecht-Holland, 1987.

[Aude96] Aude J.S. *et al.*, *MULTIPLUS/MULPLIX Parallel Processing Environment*, Proceedings of the International Symposium on Parallel Architectures, Algorithms and Networks, Pequim, China, Jun. 1996.

[Bäck91] Bäck, T. and Hoffmeister F., *Extended selection mechanisms in genetic algorithms*, In Richard K. Belew, editor, Proceedings of the Fourth International Conference on Genetic Algorithms and their Applications, San Diego, California, USA, 1991. Morgan Kaufmann Publishers.

[Bake85] Baker J.E., *Adaptive selection methods for genetic algorithms*, In J.J. Grefenstette, editor, Proceedings of the First International Conference on Genetic Algorithms and Their Applications, pages 101-111, Hillsdale, New Jersey, 1985. Lawrence Erlbaum Associates.

[Balu92] Baluja, S., *A Massively Distributed Parallel Genetic Algorithm (mdpGA)*, Technical Report, School of Computer Science, Carnegie Mellon University, 1992.

[Balu95] Baluja S. e Caruana R., *Removing the Genetics from the Standard Genetic Algorithm*, Proceedings of the Twelfth International Conference on Machine Learning, Lake Tahoe, CA. July, 1995.

[Batt92] Battiti, R. and Tecchiolli, G., *Parallel biased search for combinatorial optimization: genetic algorithms and TABU*, Microprocessors and Microsystems, Vol 16, No 7, 1992.

[Beas93a] Beasley, D., Bull, D.R. and Martin, R.R., *An Overview of Genetic Algorithms: Part 1, Fundamentals*, University Computing, 15(2), pp. 58 - 69, 1993.

[Beas93b] Beasley, D., Bull, D.R. and Martin, R.R., *An Overview of Genetic Algorithms: Part 2, Research Topics*, University Computing, 15(4), pp. 170 - 181, 1993.

[Beas93c] Beasley, D., Bull, D.R. and Martin, R.R., *Reducing epistasis in combinatorial problems by expansive coding*, In Forrest S., editor, Proceedings of the Fifth International Conference on Genetic Algorithms, pages 400-407, Morgan Kaufmann, 1993.

[Brin81] Brindle, A., *Genetic Algorithms for Function Optimization*, Doctoral Dissertation, University of Alberta, Edmonton, 1981.

[Caso87] Casotto A., Romeo F., Vincentelli A.S., *A Parallel Simulated Annealing Algorithm for the Placement of Macro-Cells*, IEEE Transactions on Computer Aided Design, Vol. CAD-6, No. 5, Sep 1987.

[Chen84] Cheng, C. e Kuh, E., *Module Placement Based on Resistive Network Optimization*, IEEE Trans. Computer-Aided Design, CAD-3 (July), 218-225, 1984.

[Coho86] Cohoon, J.P. and Paris, W.D., *Genetic Placement*, in Proceedings of the IEEE International Conference on Computer-Aided Design, pp. 422-425, 1986.

[Coho88] Cohoon, J.P., Hedge S.U., Martin W.N. and Richards D., *Distributed Genetic Algorithms for the Floor Plan Design Problem*, Technical Report TR-88-12, School of Enginnering and Applied Science, Computer Science Department, University of Virginia, 1988.

[Davi87] Davis, L. and Steenstrup, M., *Genetic Algorithms and Simulated Annealing: An Overview*, in Genetic Algorithms and Simulated Annealing, Morgan Kaufmann Publishers, Los Altos, CA, 1987.

[Davi90] Davidor, Y., *Epistasis variance: Suitability of a representation to genetic algorithms*, Complex Systems, 4:369-383, 1990.

[Diek92] Diekmann R., Lüling R., Simon J., *Problem Independent Distributed Simulated Annealing and its Applications*, Proc. of the 4th IEEE Symposium on Parallel and Distributed Processing (SPDP '92), 1992.

[Dona80] Donath, W.E., *Complexity Theory and Design Automation*, In Proceedings of the 17th Design Automation Conference, pp. 412-419, 1980.

[Dunl85] Dunlop, A.E. e Kernigham, B.W., *A Procedure for Placement of Standard Cell VLSI Circuits*, IEEE Trans. Computer-Aided Design, CAD-4, 1 (Jan.), 92-98, 1985.

[Dura89] Durand, M.D., *Parallel Simulated Annealing accuracy vs. speed in placement*, IEEE Design and Test of Computers, pp. 8-34, June 1989.

[Eshe89] Eshelman, L.J., Caruana, R.A., and Schaffer, J.D., *Biases in the Crossover Landscape*, in Proceedings of the Third International Conference on Genetic Algorithms, Morgan Kaufmann Publishers, Los Altos, CA, 1989.

[Fidu82] Fiduccia, C.M. e Mattheyses, R.M., *A Linear Time Heuristic for Improving Network Partitions*, In Proceedings of the 19th Design Automation Conference, pp. 175-181, 1982.

[Geis94] Geist A. et al., *PVM3 Users's Guide and Reference Manual*, Oak Ridge National Laboratory, 1994.

[Glov89] Glover, F., *Tabu Search - Part I*, ORSA Journal on Computing 1, 1989.

[Glov90a] Glover, F., *Tabu Search - Part II*, ORSA Journal on Computing 2, 1990.

[Glov90b] Glover, F., *Tabu Search: A Tutorial*, Interfaces 20, 1990.

[Gold85] Goldberg, D.E., *Optimal Initial Population Size for Binary-Coded Genetic Algorithms*, TCGA Report No. 85001m Tuscaloosa, University of Alabama, 1985.

[Gold89] Goldberg, D.E., *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley, Reading, MA, 1989.

[Gold89] Goldberg, D.E., *Sizing Populations for Serial and Parallel Genetic Algorithms*, Proceedings of the Third International Conference on Genetic Algorithms, Morgan Kaufmann Publishers, Los Altos, CA, 1989.

[Gold90] Goldberg, D.E., *Real-Coded Genetic Algorithms, Virtual Alphabets, and Blocking*, University of Illinois at Urbana-Champaign, Technical Report No. 90001, September 1990.

[Gold93] Goldberg, D.E. and Mahfoud, S.W., *Parallel Recombinative Simulated Annealing: A Genetic Algorithm*, IlliGAL Report No. 93006, July, 1993.

[Goto86] Goto S. e Matsuda, T., *Partitioning, Assignment and Placement*, In Layout Design an Verification, T. Ohtsuki, Ed. Elsevier North-Holland, New York, Chap. 2., pp. 55-97, 1986.

[Gref86] Grefenstette, J.J, *Optimization of control parameters for genetic algorithms*, IEEE Transactions on Systems, Man & Cybernetics 16(1), pages 122-128, 1986.

[Guch89] Gucht, D.V., Suh, J.Y. and Jog, P., *The Effects of Population Size, Heuristic Crossover, and Local Improvement on a Genetic Algorithm for the Traveling Salesman Problem*, Proceedings of the Third International Conference on Genetic Algorithms, Morgan Kaufmann Publishers, Los Altos, CA, 1989.

[Hall70] Hall, K.M., *An r-dimensional Quadratic Placement Algorithm*, Manage. Sci. 17, 3 (Nov.), 219-229, 1970.

[Hana72] Hanam, M. e Kurtzberg, J.M., *Placement Techniques*, In Design Automation of Digital Systems, 1, M.A. Breuer, Ed. Prentice Hall, Englewood Cliffs, N.J., Chap. 5, pp. 213-282., 1972.

[Heit94] Heitkötter, J. and Beasley, D. (editors), *The Hitch-Hiker's Guide to Evolutionary Computation*, FAQ in comp.ai.genetic, Issue 2.4, 20 December 1994.

[Holl75] Holland, J.H., *Adaptation in Natural and Artificial Systems*, University of Michigan Press, Ann Arbor, 1975.

[Huan86] Huang M.D., Romeo F., Vincentelli A.S., *An Efficient General Cooling Schedule for Simulated Annealing*, IEEE International Conference on Computer Aided Design, 1986.

[Jone87] Jones, M. and Banerjee, P., *Performance of a parallel algorithm for standard cell placement on the Intel Hypercube*, in Proc. 24th ACM/IEEE Design Automation Conference, 1987.

[Jong75] De Jong, K.A, *An analysis of the behavior of a class of genetic adaptive systems*, Dissertation Abstracts International 36(10), 5140B (University Microfilms No. 76-9381), Ph.D. Thesis, University of Michigan, Ann Arbor, 1975.

[Kern70] Kernighan, B.W. and Lin, S., *An Efficient Heuristic Procedure for Partitioning Graphs*, Bell Syst. Tech. J. 49, 2, 291-308, 1970.

[Kirk83] Kirkpatrick S., Gelatt C.D., Vecchi M.P., *Optimization by Simulated Annealing*, SCIENCE, Volume 220, Number 4598, May 1983.

[Klin87] Kling, R.M. and Banerjee, P., *ESP: A new standard cell placement package using simulated evolution*, in Proc. 24th ACM/IEEE Design Automation Conference, 1987.

[Krav87] Kravitz S.A., Rutenbar R.A., *Placement by Simulated Annealing on a Multiprocessor*, IEEE Transactions on Computer-Aided Design, Vol. CAD-6, No. 4, July 1987.

[Laar87] Laarhoven P.J.M., Aarts E., *Simulated Annealing. Theory and Applications*, D.Reidel Publishing Company, 1987.

[Leig83] Leighton, F.T., *Complexity Issues in VLSI*, MIT Press, Cambridge, Mass., 1983.

[Leon85] Leong H.W., Wong D.F and Liu, C.L., *A simulated annealing chanel router*, In Proc. ICCAD, pp 226-228, Nov. 1985.

[Mare94] Maresky, J., *On effective Communication in Distributed Genetic Algorithms*, MSc Thesis, Technion Institute, 1994.

[Mazu90] Mazumder, P. and Shahookar, K., *A genetic approach to standard cell placement using meta-genetic parameter optimization*, IEEE Trans. Computer-Aided Design, vol. CAD-9, no. 5, pp. 500-511, May 1990.

[Mazu91] Mazumder, P. and Shahookar, K., *VLSI cell placement techniques*, ACM Computing Surveys, vol. 23, no. 2, pp. 143-220, June 1991.

[Mazu93] Mazumder P. and Mohan S., *Wolverines: Standard Cell Placement on a Network of Workstations*, IEEE Transactions on Computer-Aided Design, Vol 12, N. 9, September 1993.

[Mich92] Michalewicz, Z. and Janikow, C., *GENOCOP: A Genetic Algorithm for Numerical Optimization Problems with Linear Constraints*, Communications of the ACM, 1992.

[Mich94] Michalewicz, Z., *Genetic Algorithms + Data Structures = Evolution Programs*, Springer-Verlag, 1994.

[Mitr85] Mitra, D., Romeo, F. e Sangiovanni-Vincentelli, A., *Convergence and Finite Time Behaviour of Simulated Annealing*, In Proceedings of the 24th Conference on Decision and Control, pp. 761-767, 1985.

[Moor85] Moore T.P. and DeGeus A.J., *Simulated annealing controlled by a rule based expert system*, in Proc. ICCAD, Nov. 1985.

[Mott91] Mott, G.F. and Cartwright, H.M., *Looking Around: Using Clues from the Data Space to Guide Genetic Algorithm Searches*, Proceedings of the Fourth International Conference on Genetic Algorithms, Morgan Kaufmann Publishers, Los Altos, CA, 1991.

[Park95] Park, K. and Carter, B., *On the effectiveness of genetic search in combinatorial optimization*, Proc. 10th ACM Symposium on Applied Computing, Genetic Algorithms and Optimization Track, February, 1995.

[Port92] Porto, S.C.S. and Ribeiro, C.C., *A Tabu Search Approach to Task Scheduling on Heterogeneous Processors under Precedence Constraints*, Relatório Técnico PUCRioInf-MCC03/93, 1992.

[Rose86] Rose, J.S, Blythe, D.R, Snelgrove, W.M. and Vranesic, Z.G., *Fast, High Quality VLSI Placement on an MIMD Multiprocessor*, Technical Report, Departments of Electrical Engineering and Computer Science, University of Toronto, 1986.

[Rose88] Rose, J.S, Snelgrove, W.M. and Vranesic, Z.G., *Parallel standard cell placement algorithms with quality equivalent to simulated annealing*, IEEE Transactions on Computer Aided Design, vol CAD-7, no. 3, pp. 387-396, Mar 1988.

[Sahn80] Sahni, S. e Bhatt, A., *The Complexity of Design Automation Problems*, In Proceedings of the 17th Design Automation Conference, pp. 402-411, 1980.

[Sant96] Santos C.M.P. e Salek J.S., *Uma Implementação Eficiente do Ambiente PVM no Multiprocessador MULTIPLUS*, artigo submetido ao SBAC-PAD 96, Recife, 1996.

[Scha87] Schaffer, J.D. and Morishima, A., *An Adaptive Crossover Distribution Mechanism for Genetic Algorithms*, in Proceedings of the Second International Conference on Genetic Algorithms, Lawrence Erlbaum Associates, Hillsdale, NJ, 1987.

[Scha89] Schaffer J.D., editor. *Proceedings of the Third International Conference on Genetic Algorithms and Their Applications*, San Mateo, California, June 1989, Morgan Kaufmann Publishers.

[Sech85] Sechen C., Vincentelli A.S., *The Timberwolf Placement and Routing Package*, IEEE Journal of Solid-State Circuits, Vol SC-20, No. 2, April 1985.

[Sech88] Sechen, C., *VLSI Placement and Global Routing Using Simulated Annealing*, Kluwer, 1988.

[Sohn93] Sohn A., Wu Z. and Jin X., *Generalized Speculative Computation for Simulated Annealing*, Technical Report CIS-93-05, NJIT CIS Dept., May 1993.

[Sohn95] Sohn A., *Parallel N-ary Speculative Computation of Simulated Annealing*, IEEE Transactions on Parallel and Distributed Systems, Vol. 6, No. 10, October 1995.

[Souz93] Souza, P.S., *Assynchronous Organizations for Multi-Algorithm Problems*, Ph.D. Thesis, Department of Electrical and Computer Enginnering, Carnegie Mellon University, April 1993.

[Spea91] Spears W.M. and De Jong, K.A., *On the Virtues of Parametrized Uniform Crossover*, Proceedings of the Fourth International Conference on Genetic Algorithms, Morgan Kaufmann Publishers, Los Altos, CA, 1991.

[Srin94] Srinivas, M. and Patnaik, L.M., *Genetic Algorithms: A Survey*, IEEE Computer, p. 17-26, June 1994.

[Sysw89] Syswerda, G., *Uniform Crossover in Genetic Algorithms*, in Proceedings of the Third International Conference on Genetic Algorithms, Morgan Kaufmann Publishers, Los Altos, CA, 1989.

[Tsay88] Tsay, R., Kuh, E. e Hsu, C., *Module Placement for Large Chips Based on Sparse Linear Equations*, International J. Circuit Theory Appl. 16, 411-423, 1988.

[Vecc83] Vecchi M.P. and Kirkpatrick S.A., *Global wiring by simulated annealing*, IEEE Trans. on Computer Aided Design, vol. CAD-2, no.4, pp. 215-222, Oct.1983.

[Whit89] Whitley D., *The GENITOR algorithm and selection pressure: Why rank-based allocation of reproductive trials is best*, In Schaffer [Scha89], pages 116-121, 1989.