UNIVERSIDADE FEDERAL DE SANTA CATARINA – UFSC
DEPARTAMENTO DE ESTATÍSTICA E INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ALGORITMOS
SIMULATED ANNEALING EM PARALELO + GENÉTICO CROSSOVER
- UMA ABORDAGEM HÍBRIDA -

EDÉLCIO AUGUSTO MAZIERO

FLORIANÓPOLIS (SC)
2003

EDÉLCIO AUGUSTO MAZIERO

ALGORITMOS
SIMULATED ANNEALING EM PARALELO + GENÉTICO GROSSOVER
- UMA ABORDAGEM HÍBRIDA -

Dissertação (Tese) submetida à Universidade Federal de Santa Catarina como parte dos requisitos para a obtenção do grau de Mestre em Ciências da Computação.
Orientador: José Mazzucco Junior, Dr.

Florianópolis (SC)
2003

FICHA

CATALOGRÁFICA

MAZIERO, Edélcio Augusto.

Algoritmos - Simulated Annealing em Paralelo + Genético Crossover
– Uma Abordagem Híbrida -, Florianópolis, 2003.
154f.

Dissertação (Mestrado) – Sistemas de Computação, Universidade Federal de Santa Catarina.

Capítulo I Introdução. Capítulo II Problemas de Otimização. Capítulo III Algoritmos Simulated Annealing e Genético. Capítulo IV Computação Paralela/Simulated Annealing Paralelo. Capítulo V Modelo Proposto. Capítulo VI Análise dos Resultados. Capítulo VII Considerações Finais. Referências Bibliográficas.

Dedico este trabalho a
minha esposa Luciana e
a minha querida filha Marina,
pelo apoio recebido.

AGRADECIMENTOS

A Deus por tudo que me é proporcionado e por ter permitido a conclusão deste trabalho.

Ao Prof. Dr. José Mazzucco Junior pela orientação, compreensão, amizade e principalmente por sua paciência, que demonstrou em todas as etapas de elaboração deste trabalho, qualidades inquestionáveis dos grandes mestres.

A todos os meus familiares que compreenderam a importância deste, e me apoiaram na realização do mesmo.

Aos professores que dedicaram seus esforços buscando repassar seus conhecimentos, e aos meus colegas em todos os momentos que passamos juntos.

À Universidade Federal de Santa Catarina que, através do Departamento de Pós-Graduação em Ciência da Computação proporcionou-me tais estudos.

SUMÁRIO

LISTA DE TABELAS

LISTA DE FIGURAS

RESUMO

Problemas combinatorias são utilizados em muitas áreas de pesquisa, devido a sua simplicidade de compreensão e a sua aplicabilidade prática em vários domínios. Porém são intratáveis devido ao elevado tempo de processamento e de armazenamento de dados, sendo assim conhecidos e classificados como problemas NP-completos. Visando resolver estes problemas, diversos algoritmos têm sido propostos ao longo de vários anos de estudo, entre eles os Algoritmos Genéticos (AG) e o Algoritmo Simulated Annealing (SA). Estes algoritmos dão um tratamento polinomial aos problemas de otimização, buscando uma boa solução próxima a ótima em um tempo de processamento aceitável. Este trabalho concentra-se no estudo do AG e do SA aplicados ao clássico "Problema do Caixeiro Viajante". Propõe-se uma abordagem híbrida baseada no desenvolvimento do algoritmo SA em ambiente distribuído acrescido do operador "crossover" dos AG. A utilização em conjunto destas abordagens busca aumentar a potencialidade de obtenção de melhores resultados quando aplicados a problemas de otimização, sendo avaliado através de testes computacionais com instâncias públicas disponíveis via internet e instâncias construídas, também com suas soluções, conhecidas a priori.

ABSTRACT

Combinatorial problems are well-known in many research areas, because they can be easily understood and are applicable in several domains, to model common situations. However, their solution can demand huge processing time and storage space. Combinatorial problems are normally classified as NP-complete problems. Several algorithms have been proposed to ease solving this kind of problems. Genetic Algorithm (GA) and Simulated Annealing Algorithm (SA) are good examples of such efforts. These algorithms use a polynomial approach to optimization problems, searching for an optimal solution in an affordable amount of processing time. This work concentrates on studying the GA and SA algorithms on the classical "Salesman" problem. It proposes a hybrid approach, in which the SA algorithm runs in a distributed environment and takes advantage on the GA's "crossover" operator. The joint usage of both algorithms looks to improve their effectiveness in finding better results when applied to optimization problems. The effectiveness of the proposal is evaluated using computational test sets publicly available and also with test sets developed by the author.

CAPÍTULO I
INTRODUÇÃO

A informática como ferramenta para auxiliar na resolução de problemas tem consolidado cada vez mais sua importância no dia a dia da sociedade. No que se refere a capacidade de processamento e de armazenamento de dados, temos nesta tecnologia um fiel aliado para as necessidades exigidas. Apesar de toda a capacidade de processamento atualmente alcançada, ainda nos deparamos com problemas simples que exigem um longo tempo de processamento, o qual ainda é inadequado para as necessidades existentes.

Nesta categoria estão os problemas combinatoriais, que são de simples resolução, porém a quantidade de combinações para localizar a solução exata cresce de forma exponencial a medida que aumentamos a população a ser analisada. Temos como exemplo a necessidade de identificar qual o melhor trajeto a ser percorrido por um caminhão de coleta de lixo para que o deslocamento seja mínimo, atendendo ainda aos requisitos de capacidade do caminhão e a quantidade de lixo em cada ponto coletor. Qual a melhor combinação (mão-de-obra, matéria-prima, máquinas, etc) para que vários produtos sejam produzidos e que seus custos sejam mínimos? Qual a rota que o movimento do braço de um robô, que realiza milhares de pontos de solda, deve fazer para que seu deslocamento seja mínimo? Esses são apenas três exemplos de problemas que podem ser encontrados no dia-a-dia das organizações e que envolvem processos de combinação e otimização ao mesmo tempo, sendo, portanto, denominados problemas de otimização combinatorial.

Áreas como ciência dos computadores, estatística, engenharia, ciência da administração, ciência biológica, projeto de circuito (VSLI), entre outras, propuseram nas últimas décadas vários problemas que são inerentemente de otimização combinatorial. De todos os problemas propostos pelas diversas áreas de estudo, o mais famoso é o “Problema do Caixeiro Viajante” (PCV). Neste problema, um vendedor partindo de uma cidade, precisa visitar todas as cidades definidas somente uma única vez e então retornar à cidade de origem. O problema consiste em definir a seqüência de visitas (rota) que corresponda ao comprimento (custo) mínimo. Vemos portanto que o PCV é um problema de fácil entendimento e estrutura simples, porém com alto grau de dificuldade de resolução à medida que o número de cidades

cresce. Esta característica é inerente à maioria dos problemas de otimização combinatorial.

Segundo ARAÚJO (2001), resolver um problema de otimização combinatorial consiste em encontrar a "melhor" solução ou a solução "ótima" dentre um número finito ou um número infinito contável de soluções possíveis que atenda a um objetivo específico. Tais problemas apresentam uma peculiaridade com relação aos outros, que é a grande dificuldade de obtenção de soluções exatas num tempo computacional aceitável.

Estudos realizados em relação aos problemas de otimização combinatorial, verificaram que o esforço computacional para a resolução do problema através de um algoritmo exato, era exponencial em relação ao tamanho do mesmo, (AARTS, 1989). Esta classe de problemas ditos de NP-completos passaram a ser de grande interesse, sendo que, ou existirá um algoritmo de complexidade polinomial capaz de resolver todos os problemas desta classe, ou então nenhum destes poderá ser resolvido em tempo polinomial. Constatou-se também que a maioria dos problemas de otimização combinatorial pertence a essa classe de problemas.

Muita ênfase vem sendo dada no desenvolvimento de heurísticas que forneçam soluções de boa qualidade em tempo de processamento compatível com as necessidades requeridas. ARAÚJO (2001) sita que as novas técnicas, especialmente as metaheurísticas, tais como *Tabu search*, *Simulated Annealing*, *Computação Evolucionária* (Algoritmos Genéticos, Programação Evolucionária, etc.) e *Redes Neurais*, são de muita importância para a solução destes problemas de otimização combinatorial, e têm se mostrado bastante eficientes em muitos problemas. As técnicas existentes buscam portanto, não a identificação da solução exata, mais sim de uma boa solução encontrada com resultados bastante eficientes.

1.1 Objetivo do Trabalho

Este trabalho concentra-se no estudo da utilização de duas técnicas para a solução de problemas de otimização, os algoritmos: Genético (AG) e S*imulated Annealing* (SA), aplicados ao clássico Problema do Caixeiro Viajante (PCV). Propõe-se também uma abordagem híbrida do algoritmo SA em paralelo, juntamente com o operador "crossover" dos AG. A utilização em conjunto destas duas abordagens pretende aumentar a potencialidade de

obtenção de melhores resultados quando aplicados a problemas de otimização, sendo avaliado através de testes computacionais com instâncias públicas disponíveis via internet e instâncias construídas, também com suas soluções, conhecidas *a priori*.

A proposta consiste portanto da transformação do algoritmo SA básico, este caracterizado pela análise de apenas uma solução de forma seqüencial, para um modelo que permita a análise de n alternativas do SA de forma simultânea, estas coordenadas por um gerente, sendo que a cada etapa de resfriamento necessária ao SA, serão aplicadas técnicas do AG entre as soluções encontradas. Entre as técnicas dos AG existentes serão utilizadas as seguintes: *elitismo, crossover e elitismo-crossover*.

Este trabalho não busca alcançar algum recorde em termos de número de cidades, reduzido tempo de processamento ou solução ótima, mesmo porque, para instâncias com um número muito grande de cidades, o tempo de processamento pode levar dias ou até meses com um parque computacional muito mais sofisticado do que o que será utilizado aqui. Esta avaliação se dará sobre o desempenho do método aplicado ao clássico Problema do Caixeiro Viajante, comparado aos resultados alcançados pela aplicação direta do algoritmo Simulated Annealing.

1.2 Estrutura do Trabalho

Este trabalho é composto por 7 capítulos organizados da seguinte forma: O Capítulo 2 apresenta uma fundamentação básica conceitual a respeito de problemas de otimização combinatória, problemas NP-Completos e alguns dos métodos utilizados na resolução dos mesmos. Apresenta também uma descrição mais detalhada sobre o Problema do Caixeiro Viajante.

No capítulo 3, é introduzida uma fundamentação básica a respeito dos algoritmos Simulated Annealing e Genético, onde os principais elementos desses métodos são revistos.
O capítulo 4 apresenta um estudo sobre sistemas distribuídos, e o estado atual dos estudos do algoritmo Simulated Annealing em Paralelo - PSA.

É apresentado no capítulo 5 o modelo desenvolvido baseado nos estudos de *Simulated Annealing*, *Algoritmos Genéticos* e *Sistemas Distribuídos*. E no capítulo 6 temos uma análise comparativa dos resultados computacionais obtidos.

Concluindo o trabalho, temos no capítulo 7 as considerações finais e sugestões para estudos futuros.

CAPÍTULO II
PROBLEMAS DE OTIMIZAÇÃO

Veremos neste capítulo, um embasamento teórico sobre a complexidade de otimização dos problemas combinatórias, e mais especificamente sobre o Problema do Caixeiro Viajante PCV.

2.1 Conceituação Básica

Segundo LEWIS & PAPADIMITRIOU (2000), em termos computacionais, os problemas podem ser classificados em duas categorias: aqueles que podem ser resolvidos por algoritmos e aqueles que não podem. Com o potencial computacional existente hoje, haveria de se supor que todos os problemas da primeira classe pudessem ser resolvidos de uma maneira satisfatória.

Porém na prática, verifica-se que muitos problemas, apesar de serem solúveis a princípio, não podem ser resolvidos devido às exigências de tempo e/ou espaço de memória, mesmo aplicando os mais modernos recursos computacionais existentes.

Os problemas que não podem ser resolvidos são aqueles para os quais a teoria atual da ciência da computação não consegue produzir um algoritmo. É desnecessário dizer que as técnicas tradicionais disponíveis são insuficientes para resolver esse tipo de problema. Como problema não computável podemos citar a própria matemática, "uma vez que já foi provado que não se pode criar um algoritmo que gere as deduções para todas as verdades da matemática" (LEWIS & PAPADIMITRIOU, 2000).

A ciência da computação tem buscado ao longo dos anos, respostas concretas aos problemas de elevado nível de complexidade, apontando alternativas que permitam aos pesquisadores dispor de fundamentação teórica suficiente para decidir se para um dado problema é possível desenvolver um algoritmo que forneça uma solução exata em tempo aceitável ou se deve contentar apenas com uma solução aproximada.

Analisar a complexidade de um problema é fundamental no processo de definição de algoritmos mais eficientes. Apesar de parecer contraditório, com o aumento da velocidade dos computadores, torna-se cada vez mais importante desenvolver algoritmos mais eficientes, devido ao aumento constante do ‘tamanho” dos problemas a serem resolvidos.

Segundo ARAUJO (2001), a complexidade computacional de um algoritmo diz respeito aos recursos computacionais – espaço de memória e tempo de máquina – necessários para se chegar a uma solução para o problema. Como o tempo de processamento de um algoritmo varia de um computador para outro, sua complexidade não é medida em termos reais, ou seja, não se analisa um algoritmo, por exemplo, em termos de tempo real de execução. Esse tipo de avaliação não é adequado, pois em geral existem muitos fatores influenciando o resultado, tais como eficiência do computador, compilador, recursos da linguagem, sistema operacional, entre outros. Embora o espaço de memória seja também passível de análise, a complexidade aqui vai ser considerada apenas em termos de tempo.

Aplica-se métodos analíticos para obter uma ordem de grandeza do tempo de execução. O objetivo desses métodos é determinar uma expressão (função) matemática que represente o comportamento do tempo do algoritmo, que ao contrário de métodos empíricos, visa aferir o tempo de execução de forma independente do ambiente utilizado.

2.1.1 Eficiência dos Algoritmos

A eficiência de algoritmo não está relacionada com o rápido avanço tecnológico experimentado pelos computadores, mas com a quantidade de operações elementares que serão realizadas, ARAUJO (2001). Essas operações logicamente dependem do tamanho da entrada do problema. Ordenar um bilhão de números é bem diferente de ordenar 10; a idéia é expressar o número de operações elementares como uma função que caracterize o volume do trabalho que será feito. Para um algoritmo de ordenação, esse número é simplesmente o número n de elementos a serem ordenados.

É comum expressar a complexidade de um algoritmo através da ordem de grandeza da função que o modela. Quando se expressa a complexidade através de sua ordem de grandeza,

podem-se desprezar constantes aditivas ou multiplicativas. Por exemplo, um valor de número de passos igual a 3n será aproximado por n, da mesma forma um valor de número de passos igual a $n^2 + n$ será aproximado para n^2 e para o valor de $6n^3 + 4n - 9$ teremos n^3.

Uma notação comum para simbolizar essa ordem de grandeza é a letra O, seguida por uma função entre parênteses que representa a ordem de grandeza. Temos na tabela 2.1, a representação de algumas ordens de grandezas de funções conhecidas.

TABELA 2.1 - ORDENS DE GRANDEZA DE ALGUMAS FUNÇÕES

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

Fonte: ARAUJO (2001).

A ordem de grandeza das funções é importante na análise de complexidade de algoritmos. Ao invés de computar-se a função exata do número de operações necessário, f(n), é mais fácil e freqüentemente válido trabalhar com a sua ordem de grandeza.

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

Segundo LEWIS & PAPADIMITRIOU (2000), algoritmos com complexidade polinomial são aceitos como de origem de problemas solúveis na prática e, portanto, algoritmos computacionalmente viáveis. Porém nem todos os algoritmos têm comportamento equivalente e de solução previsível como os polinomiais. Existe um grande número de problemas para os quais seus algoritmos não são limitados por um polinômio, suas taxas de

crescimento são explosivas à medida que n cresce, o que torna esses algoritmos inviáveis e que nem sempre contém fatores exponenciais, como é o caso da função fatorial.

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

Na tabela 2.2, observa-se as diferenças na rapidez de crescimento de várias funções típicas de complexidade. Os valores expressam os tempos de execução quando uma operação elementar no algoritmo é executável em um décimo de microssegundo.

TABELA 2.2 COMPARAÇÃO DE VÁRIAS FUNÇÕES DE COMPLEXIDADE

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

Fonte: ROUTO (1991).

Como se pode observar na tabela 2.2, o crescimento extremamente rápido das funções de complexidade exponenciais inviabiliza qualquer algoritmo de encontrar uma solução quando n cresce, independentemente da velocidade do computador que o execute. Devido ao desempenho negativo dos algoritmos de complexidade exponencial, os cientistas de computação consideram tais algoritmos inaceitavelmente ineficientes. Ao contrário, os algoritmos de complexidade polinomial são considerados eficientes.

2.1.2 Classificação dos Problemas

A grande contribuição da ciência da computação foi a de propor métodos matemáticos que ajudam a provar se um problema de interesse pertence ou não a uma determinada classe,

sendo que o esforço requerido para resolver precisamente um problema no computador pode variar tremendamente. Segundo LEWIS & PAPADIMITRIOU (2000), isso tem poupado os cientistas de tentativas inúteis e fadadas ao fracasso.

Um grande número de problemas é viável computacionalmente, ou seja, existem algoritmos que são capazes de resolvê-los; no entanto, nem para todo o problema se pode chegar a uma solução em um tempo razoável. Existem problemas com ordem de grandeza exponencial que, em termos computacionais, são intratáveis. Um problema é tratável se for possível exibir um algoritmo de complexidade polinomial que o resolva. Por outro lado, para verificar se um problema é intratável, há a necessidade de se provar que todo possível algoritmo que o resolva não possui complexidade polinomial.

A computação limitada a complexidade polinomial é um conceito desejável, já que inclui a maioria dos algoritmos viáveis e exclui a maioria dos impraticáveis. Porém a fronteira entre os algoritmos viáveis e os impraticáveis ainda é bastante nebulosa, uma vez que é possível encontrar algoritmos polinomiais que são impraticáveis em termos práticos em função de seus índices elevados, mas que nem sempre ocorrem na prática, não ficando, portanto, prejudicada a definição para a classe.

Definição: A classe P é formada pelo conjunto de todos os problemas que pode ser resolvido por um algoritmo determinístico em tempo polinomial.

Há uma classe de problemas que, apesar dos esforços despedidos para descobrir um algoritmo de tempo polinomial para cada um deles, nenhum algoritmo desse tipo foi encontrado. Esses problemas são, em sua maioria, os chamados problemas de decisão, para os quais todos os algoritmos conhecidos são de complexidade exponencial (LEWIS & PAPADIMITRIOU, 2000). Existem problemas que, apesar de pertencerem a essa classe, não são problemas de decisão, como os de otimização, mas que é possível, através de mecanismos adequados, transformá-los em tal.

Definição: A classe NP é formada pelo conjunto de todos os problemas que pode ser resolvido por um algoritmo não-determinístico em tempo polinomial.

Diz-se que um algoritmo é não determinístico se, além de todas as regras de um algoritmo determinístico, ele pode fazer escolhas de forma não determinística. Tem-se portanto que todo problema que admite um algoritmo de rapidez polinomial pertence a NP, isto é, $P \subseteq NP$, pois todo algoritmo determinístico é equivalente a um não-determinístico.

Não se sabe se P = NP, embora a maioria dos pesquisadores acredite que esta igualdade não seja válida. Esta é talvez a maior conjectura na área de ciência da computação. Uma das maiores razões do porquê se acredita que $P \neq NP$ é a existência da classe de problemas NP-completo.

Problemas NP-Completos são problemas não-polinomiais, indicando que o espaço de busca por soluções cresce exponencialmente com o tamanho do problema. Como conseqüência direta, estes problemas não podem ser resolvidos, para uma solução ótima, em uma quantidade razoável de tempo.

Segundo BENDER (1987), CORMEN et al. (1990) e PAPADIMITRIOU (1982) esta classe de problemas tem as seguintes propriedades:

- nenhum problema NP-Completo pode ser resolvido por qualquer algoritmo polinomial conhecido; e
- se existir um algoritmo polinomial para qualquer problema NP-Completo então existem algoritmos polinomiais para todos os problemas NP-Completos.

Com base nestes fatos muitas pessoas presumiram que não pode existir nenhum algoritmo polinomial para qualquer problema NP-Completo, entretanto ninguém conseguiu provar isto. De fato se acredita que esta prova nunca virá sem o desenvolvimento de técnicas matemáticas totalmente novas (PAPADIMITRIOU, 1982).

Os problemas NP-Completos aparecem nas mais diversas áreas tais como: lógica booleana, gráficos, aritmética, modelagem de redes, problemas de particionamento, problemas de armazenamento e recuperação, sequenciamento e escalonamento, programação matemática, álgebra, jogos, otimização de programas entre outras, (CORMEN et al, 1990).

A figura 2.1 ilustra este relacionamento entre as classes de problemas.

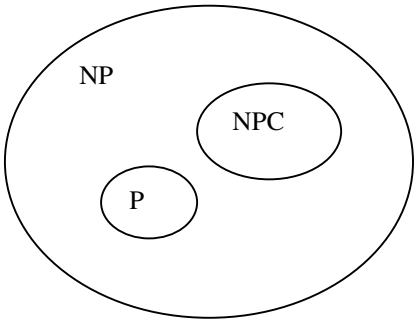

Figura 2.1 - Relacionamento entre P, NP e NPC

2.1.3 Espaço de Busca e seus Métodos

Problemas de otimização combinatorial apresentam grande dificuldade em serem solucionados, devido ao grande número de possíveis soluções que, no caso do PCV, chega a número astronômico, impossível de enumerá-las. Denominamos de *espaço de busca* o conjunto de todas as possíveis soluções, sendo que sua notação normalmente é feita por $S \subseteq \Re^n$. A solução para um determinado problema pode ser encontrar um ponto ou um conjunto de pontos onde a função $f : S \lozenge \Re$ tem valores máximos ou mínimos que, em termos de um problema de busca, pode ser escrito:

encontrar $x^* \mid f(x^*) \bullet f(x)$, para todo $x \in S$, no caso de mínimo.[1]

Segundo ARAUJO (2001), a topologia de um *espaço de busca* S pode ser contínuo ou discreto, finito ou infinito, côncavo ou convexo, que determina a aplicação desse ou daquele método, bem como as características da função f, que pode ser derivável ou não, unidimensional ou multidimensional, além de estacionária ou não-estacionária (variável com o tempo). Quando um problema apresenta somente um ponto de máximo ou de mínimo no *espaço de busca*, este é chamado *unimodal,* caso contrário *multimodal*.

TANOMARU (1995) descreve três métodos, que podem ser utilizados para pesquisar nesses espaços: *numéricos, enumerativos e métodos probabilísticos*, todos com seus

[1] Problemas de minimização em geral podem ser convertidos em problemas de maximização e vice-versa, através de artifícios bem simples, como por exemplo, multiplicar a função por -1.

respectivos derivados e ainda um número grande de métodos híbridos. Os *métodos numéricos* são bastante utilizados e apresentam soluções exatas em muitos tipos de problemas, entretanto são ineficientes para otimização de funções multimodais. Além disso, em problemas de otimização combinatorial em espaços discretos, geralmente a função a ser otimizada é não somente multimodal, mas também não diferenciável e/ou não contínua. *Métodos enumerativos* examinam cada ponto do *espaço de busca*, um por um, em busca de pontos ótimos. A idéia pode ser intuitiva, mas é obviamente impraticável quando há um número infinito ou extremamente grande de pontos a examinar. Com o advento da computação evolutiva, os *métodos probabilísticos* vêm ganhando espaço e estão presentes nas abordagens mais modernas. Este método emprega a idéia de busca probabilística, o que não quer dizer que sejam métodos totalmente baseados em sorte, como é o caso dos chamados métodos aleatórios. Dentre as abordagens mais conhecidas, podem-se citar o algoritmo *Simulated Annealing* e o algoritmo Genético.

2.1.4 Otimização Combinatorial

Problemas de otimização combinatorial refletem grande parte dos problemas científicos. Considerando um sistema onde uma série de fatores influencia seu desempenho, tais fatores podem assumir um número limitado ou ilimitado de valores, podendo ser sujeitos a certas restrições. O objetivo é encontrar a melhor combinação (*combinatorial*) dos fatores, ou seja, a combinação de fatores que proporcione o melhor desempenho (*otimização*) possível para o sistema. Um problema de Otimização Combinatorial pode ser um problema de minimização ou de maximização e é especificado por um conjunto de instâncias do problema. Uma instância é definida pelo par (S,f), onde o espaço S denota o conjunto finito de todas as soluções possíveis e f a função custo, que é um mapeamento definido por f : S ◊ $\Re$. No caso de minimização, o problema é encontrar uma solução $x_{opt} \in S$ que satisfaz $f(x_{opt}) \circ f(x)$, para todo $x \in S$. No caso de maximização, o problema é encontrar uma solução $x_{opt} \in S$ que satisfaz $f(x_{opt}) \geq f(x)$, para todo $x \in S$. A solução x_{opt} é chamada uma solução ótima global, mínima ou máxima, e $f_{opt} = f(x_{opt})$ denota o custo ótimo. Pode haver mais que uma solução com custo ótimo, definindo dessa forma S_{opt} como o conjunto de soluções ótimas, ARAUJO (2001).

Vemos que existe uma dualidade entre os conceitos de *espaço de busca e otimização combinatorial*, de tal modo que todo problema de busca pode ser considerado um problema de otimização e vice-versa.

2.1.5 Pesquisa Local e Vizinhança

Os algoritmos de pesquisa local constituem uma interessante classe dos métodos de otimização. São algoritmos baseados em melhoramento gradativo de uma solução para a outra pela exploração de sua vizinhança de acordo com algumas regras bem definidas (AARTS & KORST, 1989).

Uma estratégia de busca local inicia de uma solução arbitrária $S_i \circ S$ e a cada passo n uma nova solução S_{n+1} é escolhida a partir da vizinhança $V(S_n)$ da solução atual S_n. Isto leva a definição de uma estrutura de vizinhança em S; para cada $S_i \circ S$ é associado um sub-conjunto $V(S) \subseteq S$ chamado de vizinhança de S. A vizinhança de S é obtida a partir de S através de um movimento elementar (pequena modificação na solução corrente que resulta em uma nova solução viável do problema), (PIRLOT, 1992).

A evolução da solução atual S_n, $n = 1,2...$, desenha uma trajetória no espaço de busca S. O critério mais comum para a escolha da próxima solução S_{n+1} é escolhendo a melhor solução dentre a vizinhança de S_n, ou seja, uma solução $S_{n+1} \circ V(S_n)$ que satisfaça no caso de minimização:

$$f(S_{n+1}) \leq f(S) \quad \forall\ S \circ V(S_n)$$

e para maximização que satisfaça:

$$f(S_{n+1}) \geq f(S) \quad \forall\ S \circ V(S_n)$$

Então S_{n+1} torna-se a nova solução corrente caso seja melhor que a solução atual S_n, em caso contrário, a busca acaba. Nota-se que a escolha de uma boa estrutura de vizinhança é extremamente importante para a eficiência do processo. O ponto fraco do algoritmo é a inabilidade de escapar de um mínimo local.

A figura 2.2 demonstra um caso onde um mínimo local foi escolhido como sendo a solução para o problema. Neste caso toda as possíveis soluções em sua vizinhança $V(S_n)$ são piores do que a solução atual S_n, embora mais adiante exista uma solução melhor, que não pode ser alcançada neste caso.

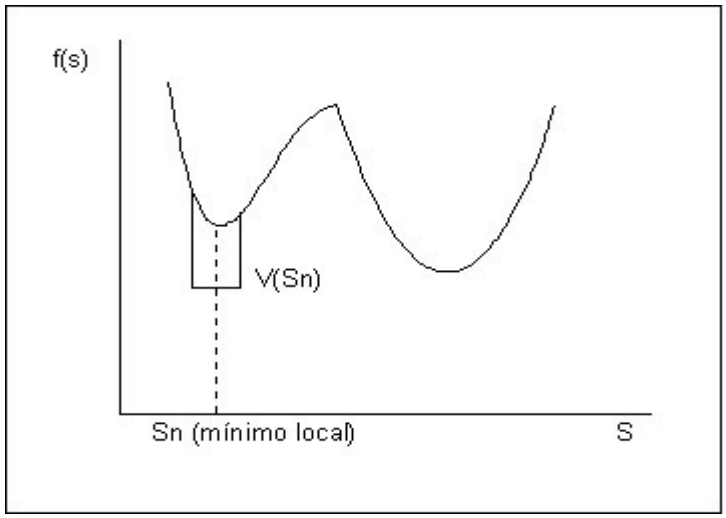

Figura 2.2 Busca local presa em um mínimo local.

Segundo AARTS & KORST (1989), para evitar as desvantagens de um algoritmo de busca local pode-se utilizar algumas técnicas alternativas como:

- executar o algoritmo várias vezes com diferentes instâncias inicias (com o custo de um aumento no tempo de processamento). Escolhem-se os melhores resultados obtidos (o fato de se executar o algoritmo com diferentes estados iniciais, embora bastante intuitivo, não traz nenhuma garantia de que uma solução ótima será encontrada, alem de criar um novo problema a respeito de quantos diferentes estados iniciais seria necessário para se obter um resultado aceitável);

- usar as informações obtidas das execuções anteriores do algoritmo para melhorar a escolha da instância inicial para a próxima execução;

- introduzir um mecanismo de geração do próximo estado mais complexo, na tentativa de saltar os mínimos locais. Quanto mais conhecimento a respeito do problema é requerido mais complexo é o mecanismo;

- aceitar transições que correspondem a um incremento na função de custo de um modo limitado.

2.2 O Problema do Caixeiro Viajante

2.2.1 Introdução

Citado pela primeira vez por DANTZIG et al. (1954, *apud* JOHNSON & MCGEOCH, 1997), o PCV relatava uma solução para 49 cidades através de métodos de programação linear, sendo que o Problema do Caixeiro Viajante ou PCV (em inglês TSP - *the Traveling Salesman Problem*) mantêm-se no topo das pesquisas dos problemas de otimização combinatorial, e que hoje pode ser considerado como sendo *o mais estudado de todos os tempos*[2].

Devido o PCV ser um problema NP-Completo, (GAREY & JOHNSON, 1979), sua importância deve-se aos seguintes fatores: simplicidade na formulação e difícil solução. Este problema serviu e continua servindo como referência no desenvolvimento, teste e demonstração de novas técnicas de otimização.

Interessante não somente sob o ponto de vista teórico, pois muitas aplicações práticas, que vão desde a fabricação de *chips* VLSI, (KORTE, 1988 *apud* JOHNSON & MCGEOCH, 1997) até cristalografia de raios-X, (BLAND & SHALLCROSS, 1989 *apud* JOHNSON & MCGEOCH, 1997), podem ser modeladas como o PCV ou como suas variações.

O PCV tem sido nos últimos anos a base para teste de numerosas técnicas. O registro de solução ótima para instância não trivial do PCV tem aumentado de 318 cidades (CROWDER & PADBERG, 1980) para 2.392 e 7.397 cidades (JOHNSON & MCGEOCH, 1997), e mais recentemente para 13.509 cidades (APPLEGATE et al., 1998).

[2] Citação feita pelos organizadores do 8th DIMACS Implementation Challenge: The Traveling Salesman Problem realizado em setembro de 2000.

2.2.2 Formulação do PCV

O PCV pode ser definido da seguinte forma: Um caixeiro deve visitar **n** cidades uma e somente uma vez. O caixeiro deve iniciar de qualquer cidade e retornar a cidade inicial. Consiste portanto em se determinar uma rota de custo mínimo, sendo o custo considerando somente a distância a ser percorrida.

Segundo ARAUJO (2001), para formular uma notação que represente o número R(n) de rotas para o caso de **n** cidades, basta seguir a seguinte lógica: Considerando que o Caixeiro parte de uma cidade determinada, portanto a primeira não afeta o cálculo, a próxima escolhida deve ser retirada do conjunto das (n – 1) cidades restantes. A seguinte, do conjunto das (n - 2) cidades restantes, e assim sucessivamente. Dessa forma, através de um raciocínio combinatorial simples e clássico, conclui-se que o conjunto de rotas possíveis, do qual o Caixeiro deve escolher a menor, possui cardinalidade dada por:

$$(n - 1) \times (n - 2) \times (n - 3) \times \ldots \times 2 \times 1$$

Ou seja, o Caixeiro deve escolher sua melhor rota de um conjunto de $(n - 1)!$ possibilidades. Considerando que não importa o sentido que se percorre o roteiro, o número total de rotas fica diminuído pela metade, obtendo-se a seguinte notação fatorial: $R(n) = (n - 1)! / 2$.

Tentar resolver o problema buscando todas as possíveis rotas conduz de um problema de otimização a um de enumeração, entretanto, essa estratégia reducionista (força bruta), aparentemente viável, não é recomendável para a maioria dos casos, principalmente quando o número de cidades é muito grande.

Suponha-se que se tem um computador capaz de fazer um bilhão de adições por segundo. Para o caso de 20 cidades, o computador precisa apenas de 19 adições para dizer qual o comprimento de uma rota; logo será capaz de calcular $10^9 / 19 = 53$ milhões de rotas por segundo. Entretanto, é necessário analisar $(19! / 2)$ rotas, totalizando $6{,}0 \times 10^{16}$ possibilidades, necessitando $6{,}0 \times 10^{16} / 53{,}0 \times 10^6 = 1{,}13 \times 10^9$ segundos para completar sua tarefa, o que equivale à cerca de 36 anos, aproximadamente.

Temos na tabela 2.3 um comparativo do esforço computacional em função do tamanho de uma rota.

TABELA 2.3 – ESFORÇO COMPUTACIONAL

N	Rotas/segundo	(n – 1)! / 2	cálculo total
5	250 milhões	12	insignificante
10	110 milhões	181.440	0,0015 seg.
15	71 milhões	$4,35 \times 10^{8}$	10 min.
20	53 milhões	$6,0 \times 10^{16}$	36 anos
25	42 milhões	$6,2 \times 10^{23}$	235×10^{6} anos

Fonte: ARAÚJO, 2001.

2.2.3 Abordagem do PCV

As abordagens para o PCV podem atuar sobre uma solução de duas maneiras, construindo passo a passo uma possível solução ou tentando sistematicamente melhorar uma solução existente. Para a primeira categoria, o conjunto de heurísticas[3] é chamado de procedimentos de construção de rotas, enquanto que, para a segunda categoria, têm-se procedimentos de melhoramento de rotas. Em geral, abordagens que combinam os dois procedimentos têm sido bastante recomendadas.

A construção de rotas utilizando heurística pode ser vista em JOHNSON & MCGEOCH (1997), sendo apresentado os métodos denominados de *Vizinho mais próximo* (mais implementada para obtenção de uma solução inicial nas abordagens de melhoramento de rotas) e de *Algoritmos de inserção*.

Os procedimentos de melhoramento de rotas, usualmente, iniciam com uma rota completa buscando reduzir o custo da mesma de tal modo que a rota se mantenha completa. A técnica mais comum é a aplicação da heurística denominada de mudança de arcos.

[3] REEVES (1995) *apud* JOHNSON & MCGEOCH (1997) define heurística (quando aplicada a problema de otimização combinatorial) como uma técnica que procura boas (próximas à ótima) soluções a um razoável custo computacional, sem ser capaz de garantir a viabilidade do ótimo, ou mesmo, em muitos casos, quão próxima do ótimo está uma particular solução viável encontrada.

A mudança de arcos consiste da troca entre duas cidades na rota; caso esta operação reduza o custo total da rota ela é mantida, sendo essa nova rota o novo ponto de partida para o próximo melhoramento. Caso contrário, tenta-se uma nova troca.

A melhor e mais efetiva heurística de mudança de arcos conhecida para melhoramento de rota aplicada ao PCV é aquela que usa o conceito *r-optimal* (ou simplesmente r-opt) abordado em LIN (1965).

Definição: Uma rota é dita ser *r-optimal* (ou *r-opt*) se for impossível obter uma outra rota com custo menor trocando qualquer conjunto de r arcos entre cidades por um outro conjunto de r arcos diferentes.

Ambas as heurísticas (*swap e r-opt*) podem ser vistas como um processo simples de pesquisa em vizinhança, onde cada rota tem uma vizinhança associada que são rotas adjacentes que podem alcançar numa única operação uma rota vizinha melhor e continuamente até que não exista nenhuma vizinha melhor.

Entre os diferentes movimentos *r-opt* existentes[4] as heurísticas *2-opt e 3-opt* estão entre as mais utilizadas. Vemos nas figuras 2.3 e 2.4 os respectivos movimentos realizados pelas heurísticas.

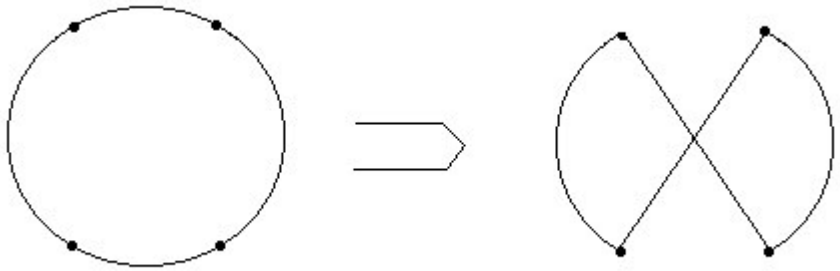

Figura 2.3 - Movimento 2-opt

[4] CHRISTOFIDES & EILON (1972) implementaram sem sucesso este método com r = 4 e 5 *apud* LAPORTE (1992).

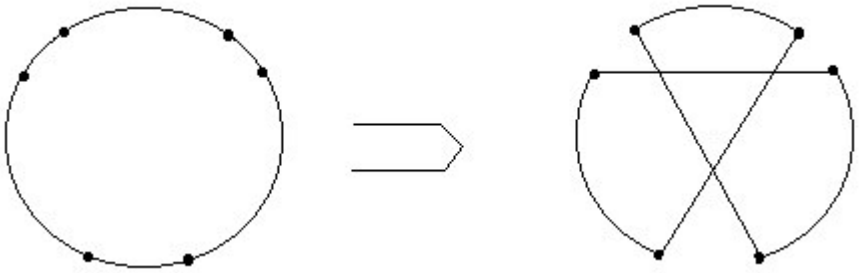

Figura 2.4 - Um possível movimento 3-opt

2.3 Outros Problemas Conhecidos

Muitos problemas relevantes do mundo real podem ser formulados com base no PCV. Mesmo alguns problemas que aparentemente não tem nenhuma relação com o PCV podem ser resolvidos através de formulações com instâncias do mesmo. Apresenta-se abaixo algumas aplicações que se encaixam nesta situação.

Roteamento de Veículos: consiste em determinar para uma frota de veículos, o ordem de atendimento a clientes;

Projeto de Circuitos Integrados: consiste em interconectar pinos com posições pré determinadas, através de fios, tendo o comprimento total dos fios minimizados;

Planejamento de Tarefas: Também chamado de "*job-shop scheduling*", consiste em determinar um programa que realize todas as operações, no menor tempo possível (MAZZUCCO, 1999);

Ligações mais rápidas de computadores a um custo menor; também pode ser de telefones ou de usuários de eletricidade;

Projeto de fibra óptica em rede de comunicação;

Projetar planos de vôos para helicópteros que fiscalizam torres de perfuração de petróleo ou de alta tensão; e programar a coleta das moedas das cabines de telefones.

CAPÍTULO III
ALGORITMOS SIMULATED ANNEALING E GENÉTICO

Será abordado neste capítulo dois tópicos, o primeiro trata do algoritmo *Simulated Annealing* (SA), sendo apresentado sua origem, estrutura e elementos básicos, em seguida será abordado o *Algoritmo Genético* (AG), seguindo os mesmos tópicos apresentados do algoritmo SA. O capítulo é finalizado com considerações sobre os algoritmos abordados.

3.1 Simulated Annealing

3.1.1 Introdução

No início dos anos 80, KIRKPATRICK, GELATT & VECCHI (1983) e posteriormente CERNY (1985) introduziram os conceitos de recozimento em problemas de otimização combinatorial. Estes conceitos estão baseados na forte analogia entre o processo de recozimento físico dos sólidos[5] proposto inicialmente por METROPOLIS et al. (1953) e a resolução de problemas de otimização combinatorial.

O algoritmo *Simulated Annealing* pode ser considerado como uma extensão do método de busca local (seção 2.1.5). A busca local requer somente a definição de um esquema de vizinhança e um método de avaliação do custo de uma solução em particular, sempre apresenta uma solução ótima, podendo esta ser pobre de acordo com a solução inicial.

O método de busca local é ineficiente quanto a armadilha do ótimo local, fazendo deste método uma heurística pobre para muitos problemas de otimização combinatorial. A figura 3.1 ilustra uma estratégia de aproximação do algoritmo de busca local. Uma propriedade desejável de qualquer algoritmo é a habilidade de achar uma boa solução, independente do ponto de partida.

[5] Por recozimento (annealing) considera-se o processo de aquecimento de um sólido até o seu ponto de fusão, seguido de um resfriamento gradual e vagaroso até atingir o seu enrijecimento, MAZZUCCO (1999).

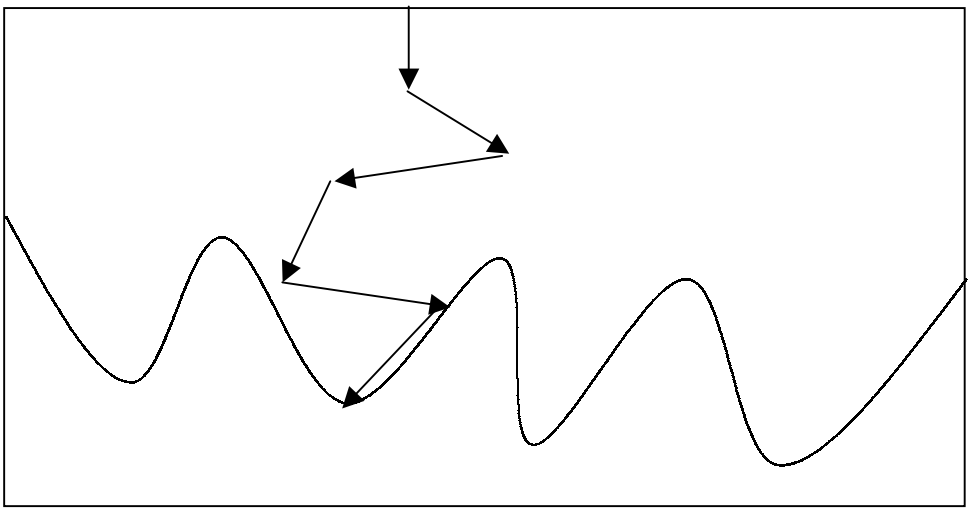

Figura 3.1 - Ilustração da estratégia da pesquisa local

Um estratégia para escapar do ótimo local é executar diversas vezes o algoritmo com diferentes soluções iniciais, sendo adotado como solução ótima a melhor solução encontrada, porém isto nos leva a outro problema que é determinar quando parar o algoritmo, bem como isto não é viável em grandes problemas, ARAUJO (2000).

3.1.2 Comportamento

Toda motivação do *Simulated Annealing* teve sua origem na mecânica estatística. Computacionalmente, o processo de recozimento (annealing) pode ser visto como um processo estocástico de determinação de uma organização dos átomos de um sólido, de modo a apresentar energia mínima.

Em alta temperatura, os átomos se movem livremente e, com grande probabilidade, podem mover-se para posições que incrementarão a energia total do sistema. Quando se baixa a temperatura, os átomos gradualmente se movem em direção à uma estrutura regular e, somente com pequena probabilidade, incrementarão suas energias (MAZZUCCO, 1999).

Segundo METROPOLIS et al. (1953), quando os átomos se encontram em equilíbrio, em uma temperatura T, a probabilidade de que a energia do sistema seja E, é proporcional a $e^{-E/kT}$, onde k é conhecida como constante de Bolzmann.

Desta forma, a probabilidade de que a energia de um sistema seja ($E + dE$) pode ser expressa por:

$$prob\,(E + dE) = prob\,(E)\;prob\,(dE) = prob\,(E)\;e^{-dE/kT}$$

Em outras palavras, a probabilidade de que a energia de um sistema passe de E para $(E + dE)$ é dada por $e^{-dE/kT}$. Na expressão $e^{-dE/kT}$, como k é uma constante, observa-se que a medida que T diminui, a probabilidade da energia do sistema se alterar é cada vez menor. Nota-se, também, que, nessas condições, quanto menor o valor de dE, maior a probabilidade da mudança ocorrer.

A pesquisa de METROPOLIS et al. (1953), proposta como método *Monte Carlo* gera uma seqüência de estados do sólido como será descrito a seguir. Dado um estado corrente i do sólido com energia E_i, então o estado subseqüente j é gerado aplicando um mecanismo de perturbação que transforma o estado corrente no próximo estado por uma pequena distorção. A energia do próximo estado é E_j. Se a diferença de energia E_j - E_i é menor ou igual a 0, o estado j é aceito como estado corrente. Se a diferença de energia é maior que 0, o estado j é aceito com uma certa probabilidade que é dada por

$$\exp\left(\frac{E_i - E_j}{k_B T}\right),$$

onde T denota a temperatura e k_B uma constante física conhecida como constante de *Boltzmann.* A regra de aceite descrita acima é conhecida como critério de *Metropolis* e o algoritmo que a usa é conhecido como algoritmo de *Metropolis*.

Se a redução da temperatura é feita suficientemente lenta, o sólido pode encontrar o equilíbrio térmico em cada temperatura. No algoritmo de Metropolis, isto é obtido gerando um grande número de transições em uma determinada temperatura. O equilíbrio térmico é caracterizado pela distribuição de *Boltzmann*.

Simulated Annealing é o método computacional que imita esse processo de recozimento de um sólido.

3.1.3 O Algoritmo

Simulated Annealing é considerado um tipo de algoritmo que tem seu princípio baseado nos algoritmos de busca local. Ele se constitui em um método de obtenção de boas soluções para problemas de otimização de difíceis soluções, e ainda oferece uma forma de escapar do ótimo local analisando a vizinhança da solução corrente e aceitando soluções que não só melhorem os resultados, como também aceitando as que piorem a solução corrente baseados em uma determinada probabilidade controlada.

Segundo ARAUJO (2000), a probabilidade de aceitar um movimento que aumente a função de custo é chamada de *função de aceite* e é normalmente representada por $exp(-\Delta / T)$, onde Δ é a diferença entre as soluções e T é um parâmetro de controle que corresponde à temperatura, fazendo uma analogia com *annealing* físico. Esta função de aceite implica que os pequenos aumentos da função de custo são provavelmente mais aceitos que os grandes e que, quando a temperatura T é alta, mais movimentos são aceitos mas, quando a temperatura T se aproxima de zero, muitos movimentos que aumentam a função de custo serão rejeitados.

Assim, o algoritmo *Simulated Annealing* é iniciado com um valor de temperatura T relativamente alto para evitar que prematuramente fique preso a um mínimo local . O algoritmo então procede tentando um certo número de movimentos na vizinhança em cada temperatura, enquanto o parâmetro temperatura é gradualmente reduzido. A figura 3.2 é uma ilustração de como o algoritmo *Simulated Annealing* caminha para fugir de um mínimo local.

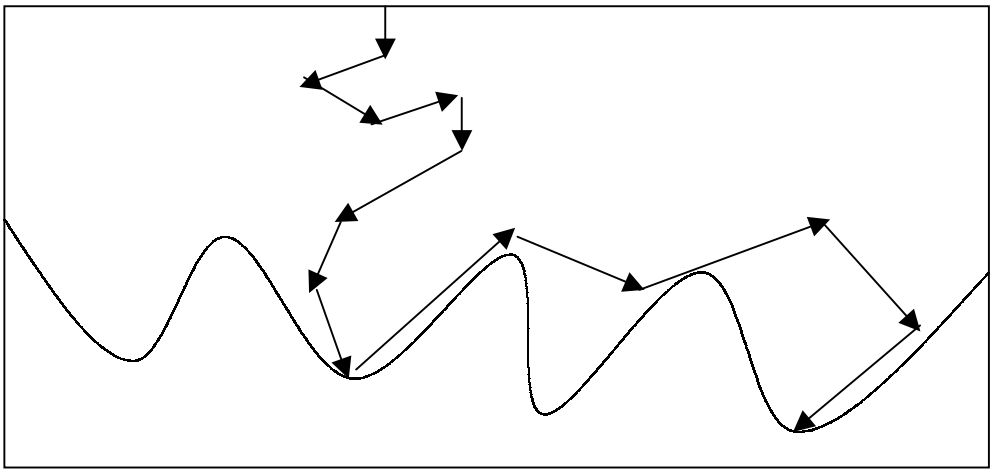

Figura 3.2 - Ilustração da estratégia do Simulated Annealing

Segundo AARTS & KORST (1989), a semelhança do algoritmo s*imulated annealing* com o método original baseado no recozimento de sólidos é muito grande, sendo que no caso dos problemas de otimização o algoritmo segue a seguinte analogia:

Soluções de problema de otimização são equivalentes a estados do sistema físico.

O custo de uma solução é equivalente à energia de um estado.

A seleção de uma solução vizinha em um problema de otimização é equivalente à perturbação de um estado físico.

O ótimo global de um problema combinatório é equivalente ao estado fundamental de um sistema de partículas.

Um ótimo local de um problema combinatório é equivalente à resfriamento rápido no sistema físico.

Define-se um problema de otimização da seguinte forma: Considerando S como o espaço total de soluções de um problema combinatorial, ou seja, S é o conjunto finito que contém todas as combinações possíveis que representam as soluções viáveis para o problema. Seja f uma função de valores reais definida sobre S, ou seja, $f : S \rightarrow R$. O problema se constitui em encontrar uma solução (ou estado) $i \in S$, tal que $f(i)$ seja mínimo.

O método de busca local localizaria uma solução denominada de mínimo local que pode estar longe de ser um mínimo global, sendo que o processo inicia com uma solução, normalmente tomada de forma aleatória, e um nova solução j é então gerada na vizinhança desta. Caso uma redução do custo dessa nova solução seja verificada, ou seja, $f(j) < f(i)$, a mesma passa a ser considerada a solução corrente e o processo se repete, caso contrário a nova solução é rejeitada e uma outra gerada. Esse processo se repete até que nenhum melhoramento possa ser obtido na vizinhança da solução corrente, após um número determinado de insistências.

Simulated annealing não utiliza essa estratégia. Esse método tenta evitar a convergência para um mínimo local, aceitando, às vezes, uma nova solução gerada, mesmo que essa incremente o valor de f (MAZZUCCO, 1999). O aceite ou a rejeição de uma nova solução, que causará um incremento de δ em f, em uma temperatura T, é determinado por um critério probabilístico, através de uma função g conhecida como função de aceite. Normalmente expressa por

$$g(\delta, T) = e^{-\delta / T}$$

Caso $\delta = f(j) - f(i)$ for menor que zero, a solução j será aceita como a nova solução corrente. Caso contrário, a nova solução somente será aceita se

$$g(\delta, T) > random(0, 1)$$

A semelhança com o método original de simulação de recozimento na termodinâmica, como já aludido, é grande, pois o parâmetro δ corresponde à variação da energia de um estado para outro (dE) e o parâmetro de controle T corresponde à temperatura. Uma vez que agora T é imaginário, a constante K, que aparecia na expressão original multiplicando T, é considerada igual a 1.

Da mesma forma que no processo físico, a função $g(\delta, T)$ implica:

i) a probabilidade de aceite de uma nova solução é inversamente proporcional ao incremento de δ;e

ii) quando T é alto, a maioria dos movimentos (de um estado para outro ou de uma solução para outra) é aceita. Entretanto, à medida que T se aproxima de zero, a grande maioria das soluções são rejeitadas.

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

vizinhanças da solução corrente, como pode ser observado no pseudo código na figura 3.3 (ARAUJO, 2000), e no fluxograma na figura 3.4 (LEE, 1995).

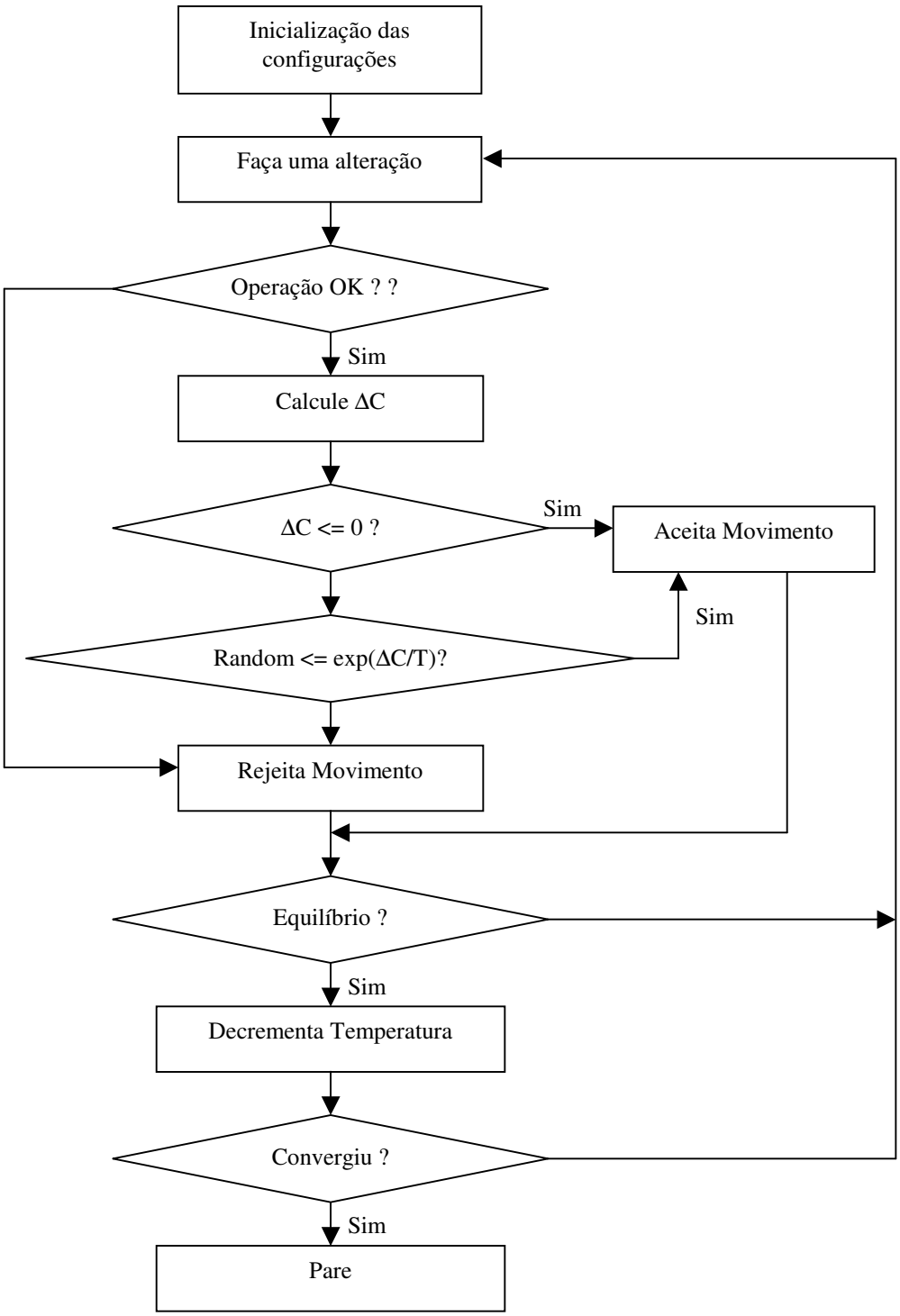

Figura 3.3 - Fluxograma do algoritmo Simulated Annealing

```
Procedimento Simulated Annealing
início
        S ⇓ S0;
        T ⇓ T0;
        enquanto temperatura elevada faça
                para interações para equilíbrio faça
                        gerar uma solução S' de N(S);
                        avaliar a variação de energia – ΔE = f(S') – f(S);
                        se ΔE < 0 então
                                S ⇓ S';
                        senão
                                gerar u ∈ random[0,1];
                                se u < exp(-ΔE/KB . T) então
                                        S ⇓ S';
                                fimse
                        fimse
                fimpara
                reduzir a temperatura T;
        fimenquanto
fimprocedimento
```

Figura 3.4 - Pseudocódigo do algoritmo Simulated Annealing

Em termos puramente teóricos, tem sido mostrado que o algoritmo converge para um conjunto de soluções ótimas globais quando o tempo tende para o infinito. Esta propriedade de convergência assintótica foi provada por meio de um modelo de cadeia de Markov. Entretanto, quando implementando o algoritmo, considerações práticas devem ser dadas ao programa de resfriamento para assegurar convergência para uma solução de boa qualidade em um tempo razoável (ARAUJO, 2000).

3.1.4 Programas de Resfriamento

Denominamos de *programa de resfriamento (cooling schedule*), o conjunto de parâmetros de controle aplicados ao algoritmo Simulated Annealing na resolução de um problema de otimização combinatorial. Segundo ARAUJO (2000) o programa de resfriamento é modelado a partir dos seguintes itens:

- A temperatura inicial;
- Uma função de redução da temperatura;

- Um critério de parada que especifica a temperatura final;
- Um número finito de transições em cada temperatura;
- Um tamanho finito de cada cadeia de Markov[6].

Para limitar o esforço computacional do algoritmo, precisa-se aumentar a queda da temperatura, o que implica menos iterações com diferentes temperaturas, exigindo cadeias de Markov maiores. Então há a necessidade de uma solução de compromisso entre o decremento da temperatura e cadeias de Markov pequenas. A busca por programas de resfriamento adequados tem sido assunto de estudo muito explorado.

Vários programas de resfriamento tem sido propostos, entre eles (ARAUJO, 2000):

Geométrico: proposto por KIRKPATRICK et al. (1983), permanece largamente utilizado. Caracteriza-se quando a temperatura inicial, o tamanho da cadeia, o decremento da temperatura e o critério de parada são todos pré definidos. Uma sugestão para o valor inicial de T é: $T = \ln f(x_0)$, onde $f(x_0)$ é o valor da função objetivo da solução inicial. Para a redução da temperatura realiza-se a multiplicação por um fator fixo $\alpha < 1$: $T_{k+1} = \alpha T_{k,}$ sendo para o valor de α usualmente aplicado entre 0.8 e 0.99, com tendência a valores próximos de 1(dando uma redução de temperatura mais lenta).

Tempo Polinomial: proposto inicialmente por AARTS & KORST (1989). Tem sido mostrado teoricamente que o mesmo produz tempo de execução limitado por uma função polinomial do tamanho da instância do problema. Como a maioria dos programas de resfriamento, ele não garante qualidade da solução final produzida. A preocupação básica é atingir um equilíbrio aproximado em uma dada temperatura.

3.1.5 Comentários Teóricos

Apesar de todo esforço dispensado com a publicação de vários trabalhos sobre a convergência do algoritmo *Simulated Annealing*, ainda não foi possível se ter uma prova matemática definitiva que justifique o sucesso empírico desse algoritmo. LEWIS &

[6] Uma cadeia de markov é uma seqüência de tentativas, onde a probabilidade do resultado de uma determinada tentativa só depende do resultado da tentativa anterior, AARTS & KORST (1989).

PAPADIMITRIOU (2000) argumentam sobre uma nova e desafiadora fronteira da teoria da computação de hoje, que é demonstrar teoricamente os impressionantes resultados práticos que se têm obtido com o emprego dessas novas heurísticas, incluindo o *Simulated Annealing*. Matematicamente, o algoritmo *Simulated Annealing* pode ser modelado através da teoria de cadeias de Markov. Utilizando esse modelo, vários resultados importantes, tratando de condições suficientes para a convergência, têm surgido na literatura.

O esforço requerido pelo *Simulated Annealing* pode variar muito, dependendo do problema e do tamanho da instância, de alguns segundos para um problema pequeno a muitas horas para um problema grande. Com isto em mente, há a necessidade de se investigar várias modificações no algoritmo básico para acelerar sua convergência para uma solução de boa qualidade. AARTS & KORST (1989) identificaram três categorias gerais de abordagens para acelerar o algoritmo *Simulated Annealing*, que são:

- um algoritmo seqüencial mais eficiente,

- aceleração do *hardware*,

- projeto de algoritmo paralelo.

3.2 Algoritmo Genético

3.2.1 Introdução

As pesquisas sobre modelos computacionais inteligentes têm, nos últimos anos, se caracterizado pela tendência em buscar inspiração na natureza, onde existem um sem-número de exemplos vivos de processos que podem ser ditos ‘inteligentes”. Muitas das soluções que a natureza encontrou para complexos problemas de adaptação fornecem modelos interessantíssimos. Embora não se possa afirmar que tais soluções sejam todas ótimas, verifica-se que estas são bem concebidas e adequadas a sua necessidade.

Tem sido crescente o interesse na resolução de problemas baseado em princípios de

evolução e hereditariedade, sendo que tais sistemas mantêm uma população de soluções em potencial, sendo aplicado algum processo de seleção baseado na aptidão dos indivíduos, e em alguns operadores "genéticos". Classifica -se tais sistemas como de computação evolucionária, ou seja, algoritmos que imitam os princípios da evolução natural para resolução de problemas de otimização.

TANOMARU (1995) cita que as observações de fenômenos da natureza e o processo estocástico de evolução natural, forneceram inspiração para os paradigmas de computação denominados de Anelamento Simulado, Redes Neurais e Computação Evolucionária (CE).

A Computação Evolucionária encara a teoria de evolução Darwiniana[7] como um processo adaptativo de otimização, sugerindo um modelo em que populações de estruturas computacionais evoluem de modo a melhorar, em média, o desempenho geral da população com respeito a um dado problema, ou seja, a "adequabilidade" da população com respeito ao ambiente.

Atualmente a Computação Evolucionária envolve um número crescente de paradigmas e métodos, dos quais os mais importantes são os Algoritmos Genéticos (AG), Programação Evolucionária (PE), Estratégias Evolucionárias (EE), Programação Genética (PG) e Sistemas Classificadores (SC), entre outros, (TANOMARU, 1995).

Os Algoritmos Genéticos representam as mais difundidas e estudadas técnicas da Computação Evolucionária, devido a sua flexibilidade, relativa simplicidade de implementação e eficácia na realização de busca global em ambientes adversos.

A origem do Algoritmo Genético pode ser atribuída ao professor John Holland da Universidade de Michigan (U.S.A). Ao final da década de 1970, Holland em suas explorações dos processos adaptativos de sistemas naturais e suas possíveis aplicabilidades em projetos de *softwares* de sistemas artificiais, conseguiu incorporar importantíssimas características da

[7] Apresentada por Charles Darwin em 1853, baseia-se na seleção natural que é o princípio da natureza de incorporar nos descendentes de um ser, modificações que condicionam vantagens para a sobrevivência de sua espécie (RICIERI, 1994).

evolução natural a um algoritmo, denominado este de Algoritmo Genético (HOLLAND, 1975).

Muitas experiências comprovam que a seleção natural é um fato incontestável, podendo não ser o único mecanismo evolutivo, porém sem dúvida um dos principais fatores do processo. Uma das principais constatações dessa evolução foi alcançada através de uma experiência realizada em laboratório, onde uma placa contendo cem milhões de bactérias foi posta em contato com uma dose de penicilina insuficiente para o combate total das mesmas.

Observou-se que dez bactérias haviam sobrevivido e se reproduziram na própria placa, sendo que essa nova geração sobreviveram ao meio, mesmo com a presença da penicilina. Dobrou-se então a quantidade do antibiótico e observou-se que quase todas as bactérias morreram. Novamente as poucas bactérias que sobreviveram a nova dosagem passaram a se multiplicar, sendo novamente expostas à dosagens cada vez mais altas. Esse processo se repetiu por cinco gerações e ao final da experiência, apresentou-se uma linhagem de bactérias resistentes a uma dose de penicilina duas mil e quinhentas vezes maior do que a inicial, utilizada na primeira cultura. É importante observar que as bactérias não desenvolveram individualmente resistência à penicilina, eram descendentes das poucas bactérias que herdaram uma característica favorável à sua sobrevivência naquele meio, havendo, portanto, um processo biológico de adaptação ao meio (MAZZUCCO, 1999).

Vista de uma forma global, a evolução natural implementa mecanismos adaptativos de otimização que embora estejam longe de serem uma forma de busca aleatória, com certeza evoluem aleatoriamente. Na evolução natural o problema que cada espécie enfrenta é a busca por adaptações benéficas para um ambiente complicado e de constantes mudanças. O conhecimento que cada espécie adquiri fica embutido nos cromossomos (dispositivos orgânicos onde a estrutura destes seres são codificadas) de seus membros (MICHALEWICS, 1999).

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

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

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

- é no processo de reprodução que a evolução se realiza. Através da mutação (*mutation*) o cromossomo de um ser descendente pode ser diferente do cromossomo de seu gerador. Através do processo de cruzamento (c*rossover*) dos cromossomos de dois seres geradores é também possível que os cromossomos do ser descendente se tornem muito diferentes daqueles dos seus geradores, e

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

O algoritmo desenvolvido por HOLLAND (1975), manipulava cadeias de dígitos binários, os quais chamou de cromossomos, sendo que realizava evoluções simuladas nas populações de tais cromossomos. Tal como nos processos biológicos, os algoritmos nada conheciam a respeito do problema que estava sendo resolvido. A única informação que lhes era fornecida consistia da avaliação de cada cromossomo que os mesmos produziam. A utilidade dessa informação era somente com respeito à condução do processo de seleção dos cromossomos que reproduziriam. Os cromossomos que melhor evoluíam apresentavam uma tendência a se reproduzirem mais freqüentemente do que aqueles cuja evolução era ruim.

Segundo MAZZUCCO (1999), esse tipo de algoritmo que, através de um simples mecanismo de reprodução sobre um conjunto de soluções codificadas de um determinado problema, consegue produzir solução de boa qualidade, constitui hoje uma importante técnica de busca, empregada na resolução de uma grande variedade de problemas, nas mais diversificadas áreas. O algoritmo genético é bem mais robusto quando comparado aos principais métodos de busca tradicionais existentes. Sabe-se que esta característica para sistemas computacionais é um fator extremamente importante. Quanto mais robusto for um sistema computacional, maior será o seu período de vida útil e mais reduzido será o seu custo para reprojetar.

Uma característica importante dos algoritmos genéticos é o seu paralelismo intrínseco. No método baseado em cálculo, assim como em outros existentes, a busca pela melhor solução se processa sempre de um único ponto para outro, no espaço de decisão, através da aplicação de alguma regra de transição. Essa característica dos métodos ponto a ponto constitui um fator de risco, uma vez que, em um espaço com vários picos, é grande a probabilidade de um falso pico ser retornado como solução. Em contraste, o algoritmo genético trabalha simultaneamente sobre um rico banco de pontos (uma população de cromossomos), subindo vários picos em paralelo e reduzindo, dessa forma, a probabilidade de ser encontrado um falso pico.

Outra importante característica do algoritmo genético é a simplificação que ele permite na formulação e solução de problemas de otimização. Apesar de serem aleatórios os algoritmos genéticos exploram informações históricas para encontrar novos pontos de busca onde são esperados melhores desempenhos. Isto ocorre através de processos iterativos, onde cada iteração é denominada de geração, constituindo assim o método generacional (CHAMBERS, 1995). Considera-se também outro método denominado este de "Steady State", onde uma parcela da nova geração é substituída por indivíduos da geração anterior.

Segundo GOLDBERG (1989) e KOZA & RICE (1994), a utilização do algoritmo genético na resolução de um determinado problema depende fortemente da realização dos seguintes passos:

- encontrar uma forma adequada de se representar soluções possíveis do problema em forma de cromossomos;
- determinar uma função de avaliação que forneça uma medida do valor (importância) de cada cromossomo gerado, no contexto do problema;
- determinação do tamanho da população e número de gerações, e
- determinação dos operadores genéticos e suas probabilidades associadas.

Segundo TANOMARU (1995), os Algoritmos Genéticos pertencem a classe dos métodos probabilísticos de busca e otimização, embora não sejam aleatórios. Usa-se o conceito de probabilidade, mas os Algoritmos Genéticos não são simples buscas aleatórias,

pelo contrário, tenta-se dirigir a busca para regiões do espaço onde é "provável" que os pontos ótimos estejam.

3.2.2 Fundamentos dos Algoritmos Genéticos

TANOMARU (1995) apresenta para os Algoritmos Genéticos a seguinte definição:

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

Como o algoritmo genético se constitui em um método de busca de propósito geral, baseado em um modelo de evolução biológica natural, tem por objetivo explorar o espaço de busca na determinação de melhores soluções, permitindo que os indivíduos na população evoluam com o tempo. Cada passo do algoritmo no tempo, na escala evolutiva, é chamado de geração. Partindo de uma geração inicial, normalmente criada aleatoriamente, o algoritmo executa continuamente o seguinte laço principal (DAVIS, 1985; GOLDBERG, 1989; TANESE, 1989; GAUTHIER, 1993; e MAZZUCCO, 1999):

1) avaliar cada cromossomo da população através do cálculo de sua aptidão, utilizando a função de avaliação (etapa de avaliação);
2) selecionar os indivíduos que produzirão descendentes para a próxima geração, em função dos seus desempenhos apurados no passo anterior (etapa de seleção);
3) gerar descendentes através da aplicação das operações de cruzamento (*crossover*), e mutação sobre os cromossomos selecionados no passo anterior, criando a próxima geração (etapa de reprodução) e
4) se o critério de parada for satisfeito, termina e retorna o melhor cromossomo, senão volta ao passo 1.

Para os passos apresentados, deve-se considerar os seguintes parâmetros na utilização do algoritmo:

- tamanho da população;
- número máximo de gerações;
- probabilidade de ocorrência de *crossover* por par de geradores e
- probabilidade de ocorrência de mutação por posição de gene.

3.2.3 Representação Cromossômica

A representação cromossômica constitui o primeiro passo para a aplicação de um algoritmo genético em um problema qualquer. Caracteriza-se por encontrar uma maneira de se representar cada possível solução *S* no espaço de busca como uma seqüência de símbolos *s* gerados a partir de um alfabeto finito *A*. Nos casos mais simples, usa-se o alfabeto binário $A = \{0,1\}$ (muitos consideram estes como algoritmo genético puro, embora na prática a utilização de uma representação binária nem sempre seja possível). No caso geral, tanto o método de avaliação quanto o alfabeto genético dependem de cada problema. Porém uma vez definida a estrutura mais apropriada para representar a solução do problema, esta deve conter todos os nós de busca e ser única, não sendo permitido a alteração da forma de representação no decorrer do processo (MICHALEWICS, 1999).

Segundo GOLDBARG (2000), usando algumas das metáforas empregadas pelos teóricos e praticantes de AGs, os termos mais utilizados são:

- população: conjunto de indivíduos (conjunto de soluções do problema);
- cromossomo: representa um indivíduo na população (uma configuração ou solução);
- gene: representa um componente do cromossomo (uma variável do problema);
- *allele* ou alelo: descreve os possíveis estados de um atributo do indivíduo (possíveis valores de uma variável do problema);
- *lócus*: representa a posição do atributo no cromossomo;
- fenótipo: denota o cromossomo decodificado, e
- genótipo: representa a estrutura do cromossomo codificado;

Na maioria dos algoritmos genéticos assume-se que cada indivíduo seja constituído por um único cromossomo (fato que não ocorre na genética natural), razão pela qual é comum utilizar os termos cromossomos e indivíduos indistintamente. As maiores parte dos

algoritmos genéticos, propostos na literatura, utilizam uma população de tamanho fixo, com cromossomos também de tamanho constantes (TANOMARU, 1995).

3.2.4 Fluxo de Execução

Definida a representação cromossômica para o problema, gera-se um conjunto de soluções iniciais denominadas de "soluções candidatas", correspondendo estas a uma população de indivíduos *P(0)*. A cada interação do algoritmo a população é modificada, sendo criada uma nova geração, embora nem todos os indivíduos de uma população sejam necessariamente "filhos" de indivíduos da população na interação anterior. Determinando a cada geração por um índice *t*, o fluxo básico de uma algoritmo genético pode ser visto na figura 3.5.

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

Figura 3.5 - Fluxo básico do Algoritmo Genético, TANOMARU (1995)

3.2.5 Etapas do Algoritmo Genético

O funcionamento do Algoritmo Genético durante o seu fluxo de execução atravessa diversas etapas, sendo elas: (TANOMARU, 1995)

Etapa de Inicialização:

Normalmente a população de indivíduos é gerada aleatoriamente ou através de algum processo heurístico, sendo importante que a população inicial cubra a maior área possível do espaço de busca, representando assim diferentes graus de adaptação ao ambiente em que vivem;

Etapa de Avaliação e Adequabilidade:

Corresponde a informação do valor de uma função objetivo para cada membro da população, devendo esta retornar um valor não negativo (caso aplicando-se o método de seleção da roleta). Obtêm-se desta forma uma medida (fitness / adequabilidade) de quão adaptado ao ambiente o indivíduo está, sendo que quanto maior o valor, maiores são as chances do indivíduo sobreviver ao meio e reproduzir-se, passando o seu material genético para as próximas gerações;

Etapa de Seleção:

Nesta etapa o Algoritmo Genético emula o processo de reprodução e de seleção natural. A seleção é efetuada com base nas aptidões individuais (calculadas estas através da função de avaliação) dos indivíduos, sendo que os indivíduos selecionados procriarão para formarem a próxima geração. Esta seleção é probabilística, assim, um indivíduo com aptidão alta tem maior probabilidade de se tornar um gerador do que um indivíduo com aptidão baixa. No caso de um algoritmo genético com a população fixa, são escolhidos tantos indivíduos quanto for o tamanho da população (MAZZUCCO, 1999).

A seleção inicia com a conversão de cada aptidão individual em uma expectativa que é o número esperado de descendentes que cada indivíduo poderá gerar. No algoritmo original proposto por HOLLAND (1993), essa expectativa individual era calculada através da divisão da aptidão individual pela média das aptidões de toda a população da geração corrente. Desta forma a expectativa *e* de um indivíduo *i*, é calculada por:

$$e_i = apt_i / mapt_t$$

Onde apt_i é a aptidão do indivíduo *i* e $mapt_t$ é a aptidão média da população na geração *t*.

As expectativas assim produzidas têm as seguintes características: um indivíduo com aptidão acima da média terá expectativa maior do que *1*, enquanto que um indivíduo com aptidão abaixo da média terá uma expectativa menor do que *1*; a soma dos

valores de todas as expectativas individuais será igual ao tamanho da população.

MAZZUCCO (1999) apresenta outra forma de conversão de valores de aptidão em números esperados de descendentes, referenciado em GOLDBERG (1989) como o método do truncamento sigma. Esta forma é baseado na aptidão média de toda a população e no desvio padrão das aptidões sobre a população, baseado na idéia de que um indivíduo, cuja aptidão seja igual à média das aptidões da população, terá uma expectativa de produzir um descendente; um indivíduo, cuja aptidão seja um desvio padrão acima da média, terá uma expectativa de produzir um descendente e meio; um indivíduo, cuja aptidão seja dois desvios padrões acima da média, terá uma expectativa de produzir dois descendentes, e assim por diante.

Após calculado a expectativa de cada indivíduo, esta é transformada na probabilidade do número de descendentes que cada indivíduo irá gerar. Esta última operação se faz necessária uma vez que o valor da expectativa é um número real e não pode ser utilizado como valor real de descendentes, pois supondo que um indivíduo receba a expectativa 1.5, não teria como um indivíduo gerar um descendente e meio.

Uma técnica denominada de "roleta", apresentada em GOLDBERG (1989) e MIRANDA (1999), define a quantidade de descendentes que cada indivíduo receberá após definidas as expectativas. Para visualizar este método considere um círculo dividido em *n* regiões (com *n* sendo o tamanho da população). A área de cada região é proporcional à expectativa de cada indivíduo. Coloca-se sobre este círculo uma "roleta" com *n* cursores, igualmente espaçados. Após um giro da roleta a posição dos cursores indica os indivíduos selecionados. Evidentemente, os indivíduos cujas regiões possuem maior área terão maior probabilidade de serem selecionados mais vezes. Como conseqüência, a seleção de indivíduos pode conter várias cópias de um mesmo indivíduo enquanto outros podem desaparecer. Aqui os indivíduos mais bem adaptados (com maiores aptidões) têm mais chances de serem selecionados do que os indivíduos mais fracos (com aptidões menores).

Temos como exemplo na figura 3.6 o método da roleta aplicado a uma população com 5 cromossomos. Nesta figura o cromossomo 1 tem sua aptidão muito maior que os demais

cromossomos, e após o giro da roleta o mesmo recebe dois descendentes. Já o cromossomo 4 tem sua aptidão muito inferior aos demais e após o giro da roleta ele não teve nenhum cursor apontando para sua área. Ele não irá gerar nenhum descendente e conseqüentemente, será extinto, desde que aplicado o método sem reposição.

É importante ressaltar que o processo de seleção utilizado pelo algoritmo genético é probabilístico e é um dos pontos essenciais do algoritmo. Como já aludido, o processo aloca a cada indivíduo uma chance de ser selecionado (não importando o quanto à aptidão desse indivíduo seja pobre) e participar no processo dos operadores genéticos (KOZA & RICE, 1994).

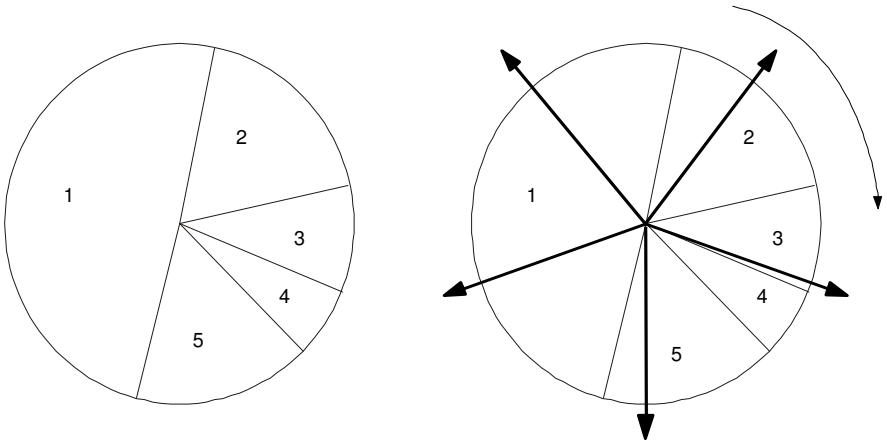

Figura 3.6 - Método de seleção "roleta".

Etapa de Reprodução:

Esta etapa se divide em dois passos, No primeiro passo são criadas cópias dos indivíduos selecionados como participantes da próxima geração na quantidade determinada pelo número de descendentes calculados na etapa anterior. No passo seguinte, as cópias são agrupadas duas a duas, de forma aleatória e sem reposição, constituindo os pares de geradores. Cada par de indivíduos, através da aplicação dos operadores genéticos, produzirá dois descendentes para a geração seguinte. Os operadores genéticos básicos, neste passo são reprodução, cruzamento (*crossover*) e mutação. Embora estes operadores sejam considerados os mais importantes, existe um outro operador que tem obtido bons resultados e merece destaque, denominado de operador de inversão (*Inversion*). Temos portanto:

Reprodução: Através da operação de reprodução, cada par de geradores, escolhido no passo anterior, fará parte da próxima população automaticamente (MAZZUCCO, 1999).

Cruzamento: O operador *crossover*, atua sobre um par de geradores, permutando alguns segmentos de suas estruturas, para formar dois novos indivíduos descendentes. Veremos adiante um maior aprofundamento deste operador que é considerado como o principal operador genético frente a outras heurísticas;

Mutação: Esta operação é considerada de retaguarda, sendo necessária para a introdução e manutenção da diversidade genética na população, alterando arbitrariamente um ou mais componentes de uma estrutura (cromossomo) escolhida, fornecendo assim meios para a introdução de novos elementos na população (CHAMBERS, 1995). A taxa, através da qual, a operação de mutação é aplicada a uma posição de um indivíduo, é determinada pelo parâmetro fornecido: probabilidade de ocorrência de mutação por posição de genes.

Inversão: A exemplo da mutação, a inversão em geral não considerado como um operador pois somente suplementa o processo de cruzamento. A inversão é um processo unário (aplicado a um único cromossomo) e por isso não há troca de informações entre os cromossomos. Este operador seleciona dois pontos de corte ao longo do cromossomo, corta o cromossomo entre estes dois pontos e inverte a ordem dos alelos (MICHALEWICS, 1999).

Considerado por muitos como o principal operador genético, o crossover é o operador responsável pelo cruzamento de características de dois pais durante a reprodução, permitindo que as próximas gerações herdem estas características (SPILLMAN, 1993). Baseia-se na troca de informações parciais entre pares de cromossomos. Embora existam outras formas de cruzamento em desenvolvimento e experimentação, veremos apenas as abordagens clássicas, com apenas dois pais geradores.

O processo de cruzamento é geralmente controlado por um parâmetro fixo que indica a probabilidade de um gene sofrer cruzamento (normalmente se utiliza uma taxa de cruzamento moderada). Este operador possui o efeito de explorar as informações contidas em seus geradores, ou seja, tendem a gerar melhores indivíduos com o passar das gerações baseados nas gerações anteriores (TANOMARU, 1995).

O operador de cruzamento pode variar conforme a representação utilizada. Segundo GOLDBERG (1989), quando a representação é binária, os dois tipos mais utilizados são: o cruzamento de um ponto e o cruzamento multiponto.

Crossover de um ponto: é realizado através da escolha aleatória de um só ponto de corte. Cada par de cromossomos escolhidos como pais gera dois descendentes através da permuta de suas partes finais, depois do ponto de corte.

Temos na figura 3.7 um exemplo de cruzamento de um ponto que é realizado depois 5° gene contando da esquerda para a direita.

Um dos problemas deste tipo de cruzamento é que os descendentes sempre irão conter as partes finais dos seus pais, não podendo fazer outra combinação possível, caracterizando uma tendência de sobrevivência das partes finais (SPILLMAN, 1993).

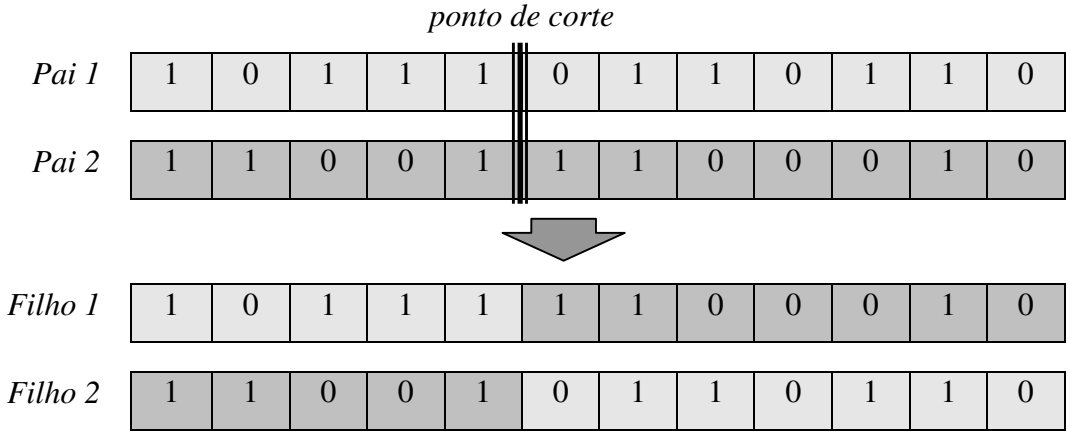

Figura 3.7 - Exemplo de cruzamento de um ponto.

Crossover multiponto: este operador busca melhorar o crossover de um ponto, sendo escolhidos mais pontos de corte para troca de material genético entre o par de geradores. A figura 3.8 exemplifica um crossover multiponto com dois pontos de cortes, sendo que o primeiro ponto de corte é após o gene 4 e o segundo após o gene 9, o material genético a ser trocado são os genes entre 5 e 9.

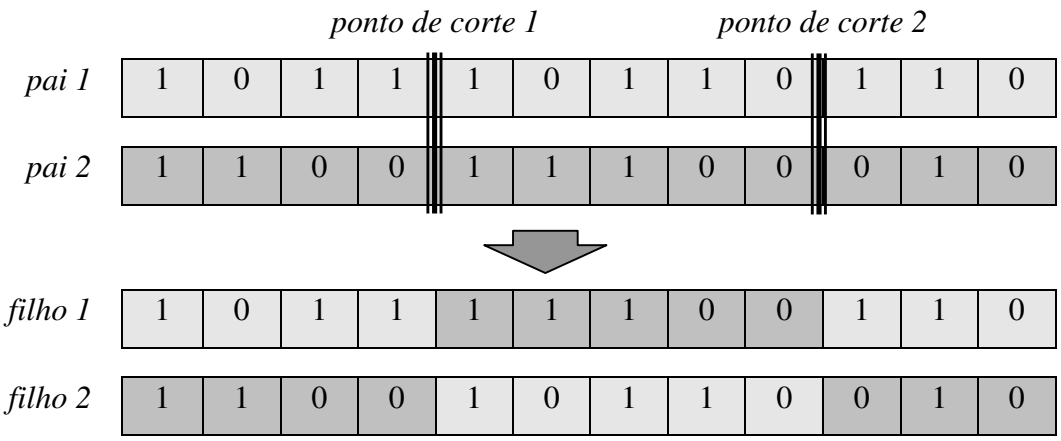

Figura 3.8 - Exemplo de cruzamento multiponto.

Nem sempre podemos utilizar a representação binária, como é o caso do problema do caixeiro viajante, onde a representação se dá através de números inteiros e pode ser denominada de "representação por caminhos" (GOLDBARG, 2000). Uma das maiores dificuldades destas representações é que o operador de cruzamento pode produzir ciclos inviáveis como no exemplo da figura 3.9. Esta figura traz um exemplo de cruzamento aplicado a cromossomos com representação através de números inteiros, neste exemplo após aplicarmos o cruzamento de um ponto de corte, o operador gerou dois cromossomos inválidos, pois os mesmos possuem alelos repetidos o que não é permitido (no caso do problema do caixeiro viajante seria o mesmo que dizer que o viajante não percorreu todas as cidades uma e somente uma vez como deveria).

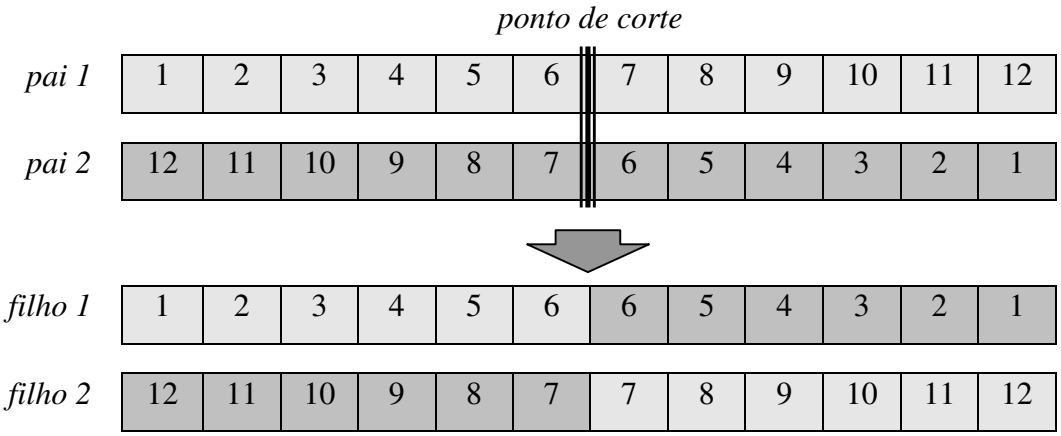

Figura 3.9 - Exemplo de cruzamento inválido.

Devido a este problema, foram desenvolvidos operadores especiais (particulares para alguns tipos de problemas) que evitam a produção de filhos inviáveis (geneticamente abortivos), dado que os pais sejam representações de soluções viáveis. Segundo GOLDBERG (1989), VACA (1995), MICHALEWICS (1999) e GOLDBARG (2000), dentre estes operadores os que mais se destacam são: o *operador OX*, o *operador PMX* e o *operador CX*.

Operador OX (*Order Crossover*): este operador começa pela escolha aleatória de dois pontos de corte em cada um dos elementos selecionados. A seção definida entre estes dois pontos é copiada integralmente no descendente. Os lugares restantes são preenchidos usando as informações não repetidas na seção de cruzamento, começando do segundo ponto de corte. A Figura 3.10 traz um exemplo de cruzamento OX, onde a seqüência j, k, c, (inicio) d, e, f, g, h, i, é o gene contido no segundo pai, começando no segundo ponto de corte. De maneira similar, obtém-se a seqüência j, k, c, (início) e, f, d, b, i, a, do segundo filho.

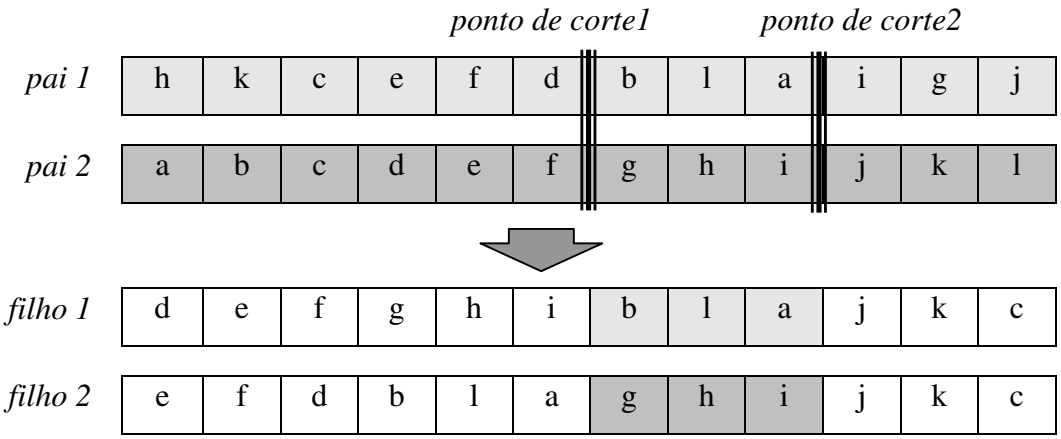

Figura 3.10 - Exemplo de cruzamento com o operador OX.

Operador PMX (*Partially Mapped Crossover*): este operador também é executado escolhendo aleatoriamente dois pontos de corte. Este processo é ilustrado na Figura 3.11. Aqui as informações contidas entre os dois pontos de corte, (b, i, a) e (g, h, i), são intercambiadas obtendo-se as representações intermediárias. Porém estas representações não são válidas, pois possuem informações repetidas e algumas faltando. Assim, o passo final é substituir estes genes repetidos mapeando (g, h, i) para (b, i, a) e vice-versa, obtendo assim as estruturas finais.

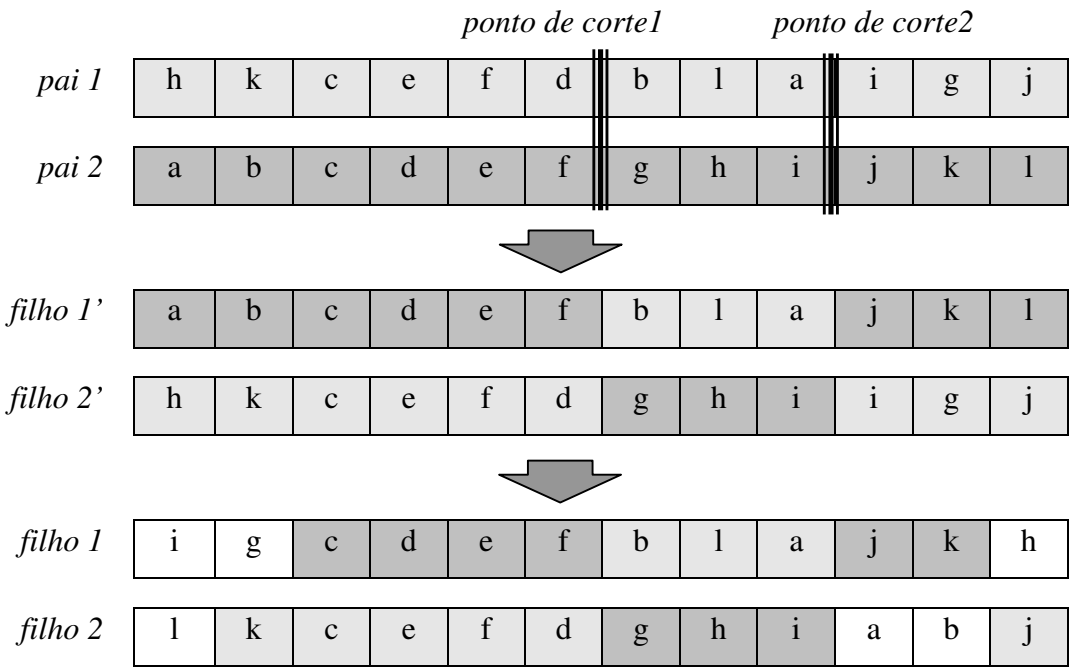

Figura 3.11 - Exemplo de cruzamento com o operador PMX.

Operador CX (*Cycle Crossover*): o esquema do operado CX é muito diferente dos demais mostrados anteriormente. O operador CX executa recombinações de forma que cada um do genes de seus descendentes venham da posição correspondente de qualquer um dos pais, não necessitando escolher pontos de cruzamento, como os usados nos operadores OX e PMX.

O procedimento se baseia em percorrer os genes dos pais, selecionando quais não podem ser cruzados, sendo cruzados os genes restantes sem que ocorra inversão de posições. Acompanhando na Figura 3.12, selecionamos o primeiro gene do *pai 1* (valor 2), este não sofrerá cruzamento, o gene equivalente no *pai 2* tem valor 3, implica portanto que não devemos cruzar o sétimo gene do *pai 1* (valor 3) senão levaremos a um estado de inconsistência, continuando o ciclo, temos para o sétimo gene do *pai 2* o valor 4, o que implica que o terceiro gene do *pai 1* (valor 4) também não poderá ser cruzado, para o terceiro gene do *pai 2* temos o valor 5, o que bloqueia o segundo gene do *pai 1* (valor 5), o segundo gene do *pai 2* tem valor 2, retornando desta forma para o primeiro gene do *pai 1*, fechando assim o ciclo. A posição destes genes é dita de formar um ciclo que, eventualmente, retorna ao primeiro gene selecionado. Para completar a operação, os espaços vazios são preenchidos com os genes do segundo pai. O segundo filho é obtido realizando o cruzamento complementar.

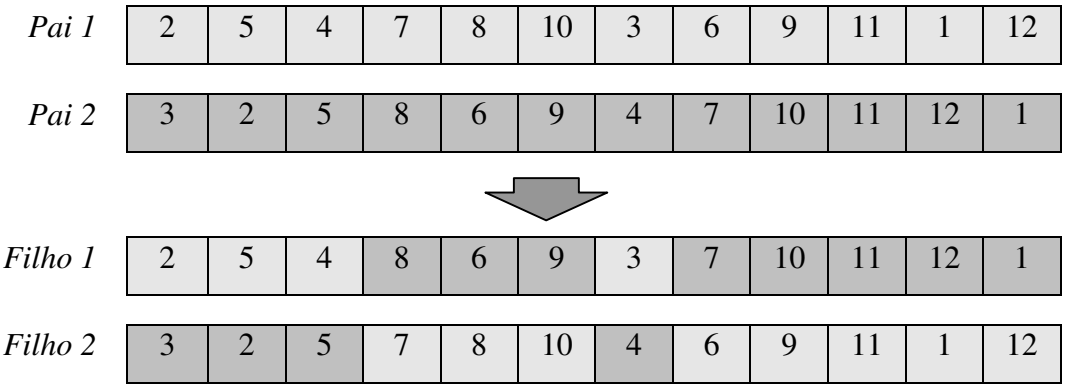

Figura 3.12 - Exemplo de cruzamento com o operador CX.

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

Outro critério que também é empregado é a idéia de "estagnação", onde o algoritmo genético termina quando nenhuma melhoria for encontrada na população depois de várias gerações consecutivas (TANOMARU, 1995 e MICHALEWICS, 1999).

3.2.6 Considerações Finais

Os Algoritmos Genéticos em muito tem se firmado na resolução de problemas de otimização como técnicas úteis e relativamente robustas de otimização de funções multivariáveis, multimodais, possivelmente descontínuas e não diferenciáveis.

Verificando as etapas apresentadas, temos que o processo de seleção ocorre ao nível de indivíduo, ao passo que os operadores genéticos ocorrem a nível cromossômico. Embora populações na natureza tenham número variável de indivíduos, na grande maioria dos Algoritmos Genéticos o tamanho da população é fixa, por questões de simplicidade e facilidade de implementação (TANOMARU, 1995).

Variações dos Algoritmos Genéticos podem ser usadas em sua implementação, visando melhorar o desempenho e os resultados obtidos pelo algoritmo, temos como exemplo:

Algoritmos Genéticos Elitistas: Tem-se a garantia de que os melhores indivíduos de uma geração sempre aparecerão na geração seguinte. A quantidade de indivíduos que será preservado depende muito do problema a ser tratado (TANOMARU, 1995);

Algoritmos Genéticos Paralelos: Neste modelo a população é particionada em sub-populações e cada sub-população de indivíduos é atribuída a um processador de um computador paralelo com memória distribuída. Cada processador executa um AG convencional em sua sub-população e periodicamente envia cópias de seus melhores indivíduos para um processador vizinho e recebe em contrapartida cópias dos melhores indivíduos do mesmo. As cópias recebidas são, em geral, usadas para substituir os piores elementos da população. O processo continua até que um critério de término seja satisfeito. Este processo de troca de indivíduos entre sub-populações é denominado de "migração" (MICHALEWICZ, 1999);

Com população de tamanho variável: Introduz o conceito de idade do cromossomo, determinando quantas gerações o cromossomo vai existir na população. Este conceito substitui o mecanismo de seleção e, desde que ele depende da aptidão do cromossomo, influencia no tamanho da população a cada geração do algoritmo;

Algoritmos Genéticos Híbridos: Utilizam os métodos de otimização tradicionais como ponto de partida para os algoritmos genéticos. A mistura de técnicas tradicionais com os algoritmos genéticos adiciona uma espécie de aprendizado no Algoritmo Genético, pois os cromossomos utilizados foram resultado de uma técnica denominada de *hill-climbing* (subida da montanha, método de busca local mais tradicional), utilizada nos métodos de otimização tradicionais (TANOMARU, 1995).

Alguns problemas podem ser encontrados na implementação de um algoritmo genético, entre eles (TANOMARU, 1995):

Determinação de Parâmetros: dificuldade em determinar o tamanho da população, além das probabilidades de cruzamento e mutação. Observa-se que quanto maior a população maior o tempo de processamento, pois muitas vezes o processo de avaliação de um cromossomo é excessivamente lento;

Erros de Amostragem: Estão relacionados ao processo de seleção, neste método, é perfeitamente aceitável que em uma população de *n* indivíduos, a maior parte dos indivíduos selecionados possuam aptidão baixa, ou mesmo que indivíduos com aptidão muito alta recebam um número desordenado de descendentes. Estes erros de amostragem podem acarretar na perda de material genético importante e, conseqüentemente, levarem a resultados de baixa qualidade.

Convergência Prematura: Em modelos de algoritmos simples, um fenômeno que se observa é que o algoritmo converge rapidamente (em umas poucas dezenas de gerações) para um ponto de alta qualidade, mas não ótimo global num fenômeno denominado de convergência prematura;

Balanço Exploração-Exploração (Exploration-Exploitation): Compreende o balanço entre "explorar" no sentido de investigar regiões desconhecidas do espaço de busca, e "explorar" no sentido de usufruir o co nhecimento genético de indivíduos da população. Basicamente, o cruzamento usa a bagagem genética de indivíduos, ao passo que mutação faz uma busca aleatória no espaço de busca. Deste modo cruzamento "explora" indivíduos realizando uma busca local, enquanto que mutação "explora" novas regiões, realizando, deste modo, uma busca global. Com base na afirmação acima se pode dizer que muito cruzamento e pouca mutação podem tornar a busca demasiadamente localizada, enquanto que muita mutação e pouca exploração podem tornar a busca desordenada e assemelhar-se a uma busca aleatória;

Tempo de Processamento: Isto normalmente é característica do problema em questão, geralmente Algoritmos Genéticos conseguem obter soluções no mínimo sub-ótimas investigando apenas uma fração do espaço de busca. Quando o espaço de busca é demasiadamente amplo, porém a busca genética pode ser exaustivamente lenta (CELKO, 1993).

Observa-se a aplicação dos Algoritmos Genéticos em muitas áreas de aplicação, temos por exemplo: Otimização combinatorial (Problema do Caixeiro Viajante), Planejamento de Tarefas, Redes Neurais, Controle de Processos e Roteamento.

Outro tópico importante relacionado aos Algoritmos Genéticos refere-se ao Teorema dos Esquemas (HOLLAND, 1993), este teorema denominado de "*Teorema Fundamental dos Algoritmo Genéticos"* trata da hipótese fundamental de que indivíduos com alto grau de aptidão são portadores de bons esquemas e que esses bons esquemas podem ser utilizados na construção de indivíduos com aptidões ainda mais altas. Sendo dado o nome de *"Blocos de Construção"* aos esquemas de grande aptidão e tamanho reduzido. O algoritmo genético constrói, portanto, uma nova geração através da exploração das informações contidas nos blocos de construção dos indivíduos da geração corrente.

CAPÍTULO IV
COMPUTAÇÃO PARALELA / SIMULATED ANNEALING EM PARALELO

Neste capítulo serão analisados conceitos referentes a Sistemas Distribuídos e a apresentação de alguns estudos referentes a implementações de SA de forma paralela.

4.1 Computação Paralela

Segundo MAZZUCCO (1999), a programação paralela pode ser considerada como sendo a atividade de se escrever programas computacionais compostos por múltiplos processos cooperantes, atuando no desempenho de uma determinada tarefa. A mesma pode ser considerada essencial em muitos sistemas computacionais, seja na minimização do tempo de processamento, ou mesmo na busca de uma estruturação melhor e/ou mais segura de um sistema. Decisões referentes a quantidade de processos e como irão interagir para um determinado problema são diretamente dependendes da natureza da aplicação e do hardware disponível.

Existem diversas correntes com relação a definição de sistemas computacionais distribuídos, sendo que as seguintes características devem ser observadas:

- A quantidade de processadores: monoprocessado ou multiprocessado;
- A memória utilizada: local privada (multicomputador) ou global compartilhada;
- Forma de comunicação (acoplamento).

Temos uma proposta para a definição de programas distribuídos em MAZZUCCO (1999), que descreve um programa distribuído como sendo um programa concorrente (ou paralelo) no qual os processos componentes comunicam-se através de passagens de mensagens. Muito embora o nome leve o adjetivo “distribuído”, decorrente do fato de que esses programas tipicamente são executados em ambientes distribuídos, um programa desse tipo, no entanto, também pode ser executado em um sistema de multiprocessadores com memória compartilhada, ou até mesmo em um monoprocessador.

A arquitetura de multicomputadores define os tipos de modelos existentes de redes, nas quais os sistemas distribuídos existem e comunicam-se. Em MAZZUCO (1999) é discutido sobre duas classes principais: as redes de interconexão de barramento e as redes de interconexão bipontuais.

De maneira geral, as razões que justificam as aplicações de sistemas distribuídos podem ser classificadas em quatro principais categorias (BAL et al., 1989):

- redução do tempo de processamento;
- aumento da confiabilidade e da disponibilidade;
- melhoria na estruturação do sistema e
- aplicações inerentemente distribuídas.

Temos para o sistema desenvolvido neste trabalho, com a aplicação de conceitos de sistemas distribuídos, a redução do tempo de processamento, bem como o aumento da disponibilidade de opções de resultados, trazendo desta forma benefícios diretos para a resolução dos problemas combinatoriais avaliados pelo método Simulated Annealing.

A implementação de uma aplicação em um sistema distribuído, requer pelo menos dois requisitos básicos essenciais, inexistentes na programação seqüencial (ANDREWS, 1991):

- distribuição de diferentes partes de um programa entre os, possivelmente, diferentes processadores; e
- coordenação dos processos componentes de um programa distribuído.

Normalmente os Sistemas Operacionais, ou as linguagens especialmente projetadas para o desenvolvimento de sistemas distribuídos, auxiliam no quesito de coordenação entre os processos, através de duas primitivas básicas (BEN, 1985):

- *SEND* (mensagem): envia uma mensagem à um processo destinatário e
- *RECEIVE* (mensagem): recebe uma mensagem de um processo origem.

Dependendo do sistema operacional, estas primitivas podem ter efeito bloqueante ou não. Um bloqueio, normalmente, se estende até que a operação seja efetivamente realizada, caracterizando assim uma forma primitiva de sincronização. Normalmente as linguagens desenvolvidas para este fim apresentam níveis mais elevados de abstração para a coordenação entre os processos de um sistema distribuído.

BAL et al. (1989) e MAZZUCCO (1999), afirmam que o modelo de programação distribuída considerada básica é aquela no qual um conjunto de processos seqüenciais rodam em paralelo, comunicam-se por meio de passagem de mensagens e a detecção de falhas é feita explicitamente. Considerando isoladamente os sistemas de *hardware* e *software*, quatro combinações viáveis diferentes podem ocorrer, gerando as classes de sistemas abaixo:

- *software* logicamente distribuído, rodando em *hardware* fisicamente distribuído;
- *software* logicamente distribuído, rodando em *hardware* fisicamente não distribuído;
- *software* logicamente não distribuído, rodando em *hardware* fisicamente distribuído;
- *software* logicamente não distribuído, rodando em *hardware* fisicamente não distribuído.

A primeira classe é a normal, uma coleção de processos, cada um rodando em um processador diferente e se comunicando através das primitivas *SEND* e *RECEIVE*, que enviam mensagens, por exemplo, utilizando uma rede. A segunda classe com a mesma estrutura lógica de multiprocessos da anterior, simula a passagem física de mensagens através de passagem lógica de mensagens, utilizando memória compartilhada. A terceira classe esconde a distribuição física do sistema de *hardware*, simulando memória compartilhada. A última também utiliza comunicação por dados compartilhados, porém nesse caso a existência de memória fisicamente compartilhada facilita a implementação.

Um grande problema existente em sistemas distribuídos é a gerência de falhas. Após uma falha ter sido detectada, o sistema deve voltar a um estado consistente. podendo ser tomadas medidas preventivas, como por exemplo, cada processador armazenar o seu conteúdo em uma memória secundária em intervalor regulares.

4.1.1 Paralelismo

Paralelismo caracteriza-se pela habilidade de rodar vários processos de um mesmo programa em vários processadores ao mesmo tempo. A relação quantidade de processos e quantidade de processadores define a granularidade do paralelismo, sendo que um sistema distribuído de baixa granularidade é considerado um sistema fortemente acoplado, quanto mais fraco é o acoplamento dos processos de um sistema, mais alta é a sua granularidade.

A maneira como o paralelismo é expresso varia de linguagem para linguagem e depende em muito da unidade do paralelismo adotado, que pode ser um processo, um objeto, um comando, uma expressão ou, ainda, uma cláusula (BAL et al., 1989).

Sendo o processo o principal elemento utilizado em paralelismo, o mesmo equivale a um procedimento especial em um programa, diferenciando em muito na sua forma de execução, pois a chamada de um processo não desvia a linha de execução do programa como num procedimento, e sim passa a existir uma linha de execução dedicada ao processo, independente do programa que o originou. Muitas linguagens permitem a criação dinâmica de processos dentro de um programa através da cláusula *create*, isto tornar a linguagem mais flexível, pois além de permitir criação dinâmica de processos, possibilita a passagem de parâmetros ao processo recém criado. Parâmetros, normalmente, são utilizados nessa situação para estabelecer canais de comunicação entre processos.

Padrões de interação entre processos na programação distribuída são apresentados em HANSEN (1995), sendo que normalmente cada um desses modelos está associado a alguma técnica de programação utilizada na resolução de problemas de programação distribuída. Temos como exemplo o modelo conhecido como cliente-servidor, no qual existem dois processos básicos, o cliente e o servidor. Os processos clientes requisitam constantemente serviços ao servidor, o servidor é desbloqueado com a requisição recebida e o cliente bloqueia-se aguardando o resultado da requisição enviada. Após o envio do resultado pelo servidor o cliente é desbloqueado e o servidor é bloqueado aguardando nova requisição. Normalmente um processo servidor é não terminante e é construído de forma que possa servir a vários clientes diferentes.

4.1.2 Comunicação e o Sincronismo

A *Comunicação* e o *Sincronismo* são os dois tipos de interação necessários na cooperação entre diferentes processadores que estão rodando em paralelo um mesmo programa. A situação considerada a seguir ilustra este aspecto. Durante a execução de um programa distribuído contendo dois processos, um processo *P1* requer, em algum ponto de sua execução, algum dado proveniente do resultado de alguma computação efetuada pelo processo *P2*, devendo existir, portanto, alguma maneira de *P1* e *P2* se comunicarem. Além disso, se *P1* atinge o ponto, na sua execução, que necessita dos dados *D* e se *P2*, por uma razão qualquer, ainda não comunicou a informação, *P1* deve estar preparado para bloquear a sua execução naquele ponto, esperando até que *P2* o faça. Devendo existir, desta forma, alguma maneira dos dois processos se sincronizarem.

MAZZUCCO (1999) descreve as várias formas existentes para controlar a interação entre processos concorrentes em um programa paralelo, sendo elas:

- Passagem de Mensagem:
  - * Envio de Mensagem ponto a ponto;
  - * Envio de Mensagem um para vários;
  - * Chamada remota de procedimento.
- Dados compartilhados:
  - * Memória compartilhada em ambiente pseudo paralelo;
  - * Memória compartilhada em ambiente paralelo real.

Necessitamos em *Passagem de Mensagem* do prévio conhecimento das seguintes informações: quem está enviando a mensagem, a quem a mensagem se destina e o que está sendo enviado. Também são requeridos controles de erros, confirmação ao emissor do recebimento da mensagem pelo receptor, modo de tratamento da mensagem pelo receptor, entre outros.

Em *Dados Compartilhados* necessitamos de alguma forma de comunicação e sincronismo a serem estabelecidos caso dois ou mais processos tenham acesso a uma mesma

variável. Por exemplo, um processo pode atribuir um certo valor a uma variável, indicando a um segundo processo, que um determinado conjunto de dados está disponível, ou que um determinado serviço está concluído. Temos para esta abordagem duas formas distintas: *Memória compartilhada em ambiente pseudo paralelo* e *Memória compartilhada em ambiente paralelo real.* A primeira forma, caracteriza-se por mais de um processo sendo executado por apenas um processador, operando sobre uma área compartilhada de dados.

Precauções devem ser tomadas para que não tenhamos resultados errados. No exemplo abaixo, temos uma situação típica, na qual o acesso simultâneo a uma variável compartilhada entre dois processos, pode conduzir a erros graves.

Processo A	*Processo B*
1. TMPA := X;	*1. TMPB := X;*
2. TMPA := TMPA + 1;	*2. TMPB := TMPB + 1;*
3. X := TMPA;	*3. X := TMPB;*
. . .	. . .

Figura 4.1 - Trechos de dois processos paralelos

Assume-se que *TMPA* e *TMPB* são variáveis locais aos processos *A* e *B*, respectivamente, que *X* é uma variável global, portanto compartilhada entre os dois processos, com valor inicial igual a 0. O valor correto da variável *X*, ao final das execuções dos fragmentos de código dos dois processos apresentados na *figura 4.1* é 2. Ambos incrementam uma unidade ao valor de *X*. Esse valor correto, no entanto, só será obtido se um dos processos executar o comando 1, após o outro processo já ter executado o comando 3. Como a ordem de execução desses comandos nos dois processos é imprevisível, o valor final da variável *X* é indeterminado, podendo variar de uma execução para outra.

Necessita-se para a resolução do problema acima, um controle de concorrência denominada de exclusão mútua, garantindo desta forma o acesso a variável global X de forma exclusiva pelos processos. Uma forma simples de implementação deste controle de concorrência, é a utilização de estruturas delimitadoras quanto ao acesso de variáveis comuns aos processos, como por exemplo as estruturas MUTEXBEGIN e MUTEXEND (do Inglês

MUTUAL EXCLUSION), temos na *figura 4.2* o código correto ao problema visto na *figura 4.1*.

Processo A	*Processo B*
MUTEXBEGIN	*MUTEXBEGIN*
TMPA := X;	*TMPB := X;*
TMPA := TMPA + 1;	*TMPB := TMPB + 1;*
X := TMPA;	*X : TMPB;*
MUTEXEND	*MUTEXEND*
. . .	. . .

Figura 4.2 - Exemplo de exclusão mútua.

O processo que primeiro alcançar o delimitador *MUTEXBEGIN*, entra na sua região, conhecida como região crítica, impedindo que o outro processo o faça. O processo perdedor deve, então, aguardar até que o vencedor alcance o delimitador *MUTEXEND*, da sua região crítica, liberando assim o acesso à variável *X*. Em uma situação em que envolva vários processos, pode haver a formação de uma fila de processos aguardando liberação de acesso.

O bloqueio imposto aos processos para que continuem suas execuções até a liberação da região crítica não é o suficiente quando uma condição mais ampla for requisitada, necessitando desta forma de um mecanismo de sincronização para a avaliação da condição geral. Sincronização, portanto, pode ser considerada como uma generalização da exclusão mútua, MAZZUCCO (1999).

Em SILBERSCHATZ & PETERSON (1994) e TANENBAUM (1992) temos vários mecanismos que podem ser utilizados na obtenção de exclusão mútua e sincronização entre processos em ambientes pseudo paralelos. Na prática, entretanto, um dos mais empregados pelas linguagens de programação é o semáforo. Essa ferramenta, introduzida por DIJKSTRA (1965), utiliza-se de operações denominadas de *P* e *V* para manipulação dos semáforos. Uma operação *P*, sobre uma variável do tipo semáforo, diminui seu valor em uma unidade, caso esse valor seja maior ou igual a um. Caso o valor da variável seja igual a zero, essa operação tem efeito bloqueante sobre o processo que a executou. Esse processo fica bloqueado até que

o valor da variável se torne maior do que zero. Esta alteração no valor da variável é conseqüência da execução de uma operação *V*, sobre essa mesma variável, realizada, eventualmente, por um outro processo. A operação *V*, incondicionalmente, aumenta, em uma unidade, o valor de uma variável semáforo.

Temos um exemplo proposto por ANDREWS (1991) conhecido por modelo produtores e consumidores. Nesse padrão de problemas, devem existir processos que continuamente produzem algum tipo de recurso, e processos que consomem esses recursos produzidos. Propondo uma situação em que vários processos continuamente produzem mensagens e as depositam em um *buffer* de tamanho igual a *n*, do outro lado, também vários processos, continuamente retiram mensagens do *buffer* e as consomem, com a utilização de semáforos para garantia da exclusão mútua e do sincronismo entre os processos, temos exemplificado esta situação na *figura 4.3*.

```
Processo Produtor                    Processo Consumidor
LOOP                                 LOOP
   Produz mensagem;                     P ( MENSAGEM );
   P ( VAGA );                             P ( LIVRE );
      P ( LIVRE );                            Retira mensagem do buffer;
         Coloca mesagem no buffer;           V ( LIVRE );
      V ( LIVRE );                        V ( VAGA );
   V ( MENSAGEM );                        consome mensagem;
ENDLOOP;                             ENDLOOP;
```

Figura 4.3 - Exemplo de utilização de semáforos

No exemplo, é assumido que *VAGA, MENSAGEM* e *LIVRE* são variáveis do tipo semáforo, cujos valores iniciais são 10, 0 e 1, respectivamente. Um processo produtor, seqüencialmente, produz uma mensagem; disputa uma vaga no *buffer* (de 10 posições), executando *P(VAGA);* disputa o acesso ao *buffer*, executando *P(LIVRE);* coloca uma mensagem no *buffer*; libera o *buffer*, executando *V(LIVRE);* aumenta em uma unidade o número de mensagens disponíveis, executando *V(MENSAGEM);* repete o ciclo. Por outro lado, um processo consumidor, seqüencialmente: disputa uma mensagem, executando

P(MENSAGEM); disputa o acesso ao *buffer*, executando *P(LIVRE);* retira uma mensagem do *buffer*; libera o *buffer*, executando *V(LIVRE);* aumenta em uma unidade o número de vagas existentes, executando *V(VAGA);* consome a mensagem; repete o ciclo.

As variáveis *VAGA* e *MENSAGEM* estão promovendo a sincronização entre as duas classes de processos, da seguinte maneira: quando o valor de *VAGA* é igual a 0, um processo produtor deve esperar por um consumidor, quando o valor de *MENSAGEM* é 0, um processo consumidor deve esperar por um produtor. A variável LIVRE, por sua vez, controla o acesso exclusivo ao *buffer*, para ambas as classes de processos.

A simplicidade e eficiência das variáveis do tipo semáforo, podem trazer um problema denominado de *deadlock* resultado da utilização descontrolada e excessiva das operações P e V. Temos para o mesmo exemplo que a simples troca da ordem das operações

```
    ...
P (VAGA);
P (LIVRE);
    ...
```

por

```
    ...
P(LIVRE);
P(VAGA);
    ...
```

No corpo do programa dos processos produtores, pode conduzir o sistema todo a uma situação de impasse irreparável, onde todos os processos se bloqueiam em um ciclo, um a espera de outro, caracterizando o *deadlock* (bloqueio fatal).

Proposto por HOARE (1974) e HANSEN (1995) uma linguagem de mais alto nível que garante uma sincronização apropriada de processos, de forma automática, conhecida por monitor, assegura exclusão mútua, isto é, apenas um processo a cada tempo pode estar ativo dentro de um monitor.

A abordagem *Memória compartilhada em ambiente paralelo real*, que permite a manipulação de uma estrutura de dados por vários processos em máquinas diferentes, simultaneamente, está fortemente baseado no conceito de tuplas (HANSEN, 1995). Um espaço de tuplas (*ET*) é conceitualmente uma memória global compartilhada por todos os processos de um programa distribuído, muito embora sua implementação não requeira memória fisicamente compartilhada. Somente três operações estão definidas sobre um *ET*: a operação *OUT*, a qual adiciona uma tupla a um espaço; a operação *READ*, que lê uma tupla de um espaço e a operação *IN*, que também lê uma tupla de um *ET*, porém remove-a do seu espaço. Essas operações são atômicas, isto é, várias operações simultaneas realizadas sobre uma mesma tupla têm o efeito final idêntico ao causado por essas operações, quando executadas em alguma ordem seqüencial (indefinida) sobre essa mesma tupla.

4.2 Simulated Annealing em Paralelo (PSA)

Vimos no capítulo 2 que o algoritmo SA utiliza-se de uma técnica eficiente para a resolução de problemas de otimização combinatorial, sendo portanto que a performance do algoritmo SA se concentra basicamente nos seguintes itens (VAN & AARTS, 1987):

- A qualidade da solução final obtida pelo algoritmo;
- O tempo de execução requerido pelo algoritmo;
- O programa de resfriamento.

Para os itens apresentados, verifica-se que o programa de resfriamento implica diretamente no tempo de execução e na qualidade da solução final obtida. Segundo AARTS & KORST (1989) o programa de resfriamento leva o algoritmo SA a um tempo de execução polinomial, mas não dá nenhuma garantia com relação a diferença entre o custo da solução encontrada e o custo ótimo do problema especificado. Quanto mais rápido o algoritmo convergir, pior é a qualidade de solução encontrada.

O esforço computacional requerido pelo algoritmo SA depende da natureza e do tamanho do problema a ser otimizado, podendo ser consumidos poucos segundos para pequenas instâncias do Problema do Caixeiro Viajante (PCV), bem como alguns dias para

grandes instâncias do Problema de Alocação (AARTS & KORST, 1989).

Embora o algoritmo SA seja conceitualmente simples, encontrar os parâmetros ideais para o SA, isto é, a temperatura inicial, a constante para o calculo da redução da temperatura, entre outros, não é simples nem direto. A configuração dos parâmetros do SA é um problema e deverá muitas vezes ser ajustados através de tentativas e erros (CHEN et al., 1998). Estudos prévios mostram que o SA é muito sensível aos parâmetros, e sua performance é altamente dependente do ajuste dos mesmos.

Como o SA é um algoritmo probabilístico, podemos ter um resultado diferente a cada execução (VAN & AARTS, 1987 e AARTS & KORST, 1989), podendo ser necessário executar repetidas vezes o algoritmo buscando desta forma melhorar ou confirmar uma determinada solução encontrada. Isto nos traz a necessidade da execução em paralelo do algoritmo SA, buscando manter o tempo computacional equivalente a execução de apenas uma instância do SA. Em AARTS & KORST (1989), afirma-se que a chave para o acréscimo de velocidade do SA é distribuir a execução de partes do algoritmo SA em vários processadores em paralelo, porém esta não é uma tarefa fácil devido a natureza seqüencial intrínseca do algoritmo SA.

A paralelização de Simulated Annealing – PSA permite manter a performance do algoritmo SA, e amplia as possibilidades de convergência pela quantidade maior de soluções encontradas num mesmo espaço de tempo, acrescido este de um overhead existente no processamento de sistemas distribuídos.

Várias abordagens tem sido propostas para implementação do algoritmo SA em paralelo. Segundo KLIEWER (2000), esquemas recentes de classificação distinguem entre "single" e "multiple -walks" (*figura 4.4*). No primeiro critério de classificação temos o número de caminhos que são avaliados no espaço de busca do problema tratado. Num algoritmo "single-walk", somente um caminho dentro do espaço de soluções é conduzido, por outro lado, em "multiple-walks" são abordados vários caminhos diferentes simultaneamente. Em "Single-walk" após o algoritmo avaliar parte da vizinhança da solução corrente somente um passo é conduzido (Single-step) ou uma seqüência de passos é executada a partir da solução

corrente (Multiple-step). Em "Multiple -walk" podem ser conduzidas soluções em paralelos de forma independente, ou estas poderão interagir de acordo com determinados padrões de comunicação. A abordagem "Independent-walk" também é conhecida como "Multiple Independent Run – MIR".

Conforme a classificação apresentada, vemos o agrupamento dos modos de execução dos PSA em dois modelos: o modelo "MIR parallelization" caracteriza -se pela execução de várias instâncias do SA em paralelo sem interações, sendo que a melhor solução é escolhida ao final do processo. Já o modelo "Clustering parallelization" caracteriza -se pela interação entre as instâncias do SA, buscando melhorar a qualidade da solução final.

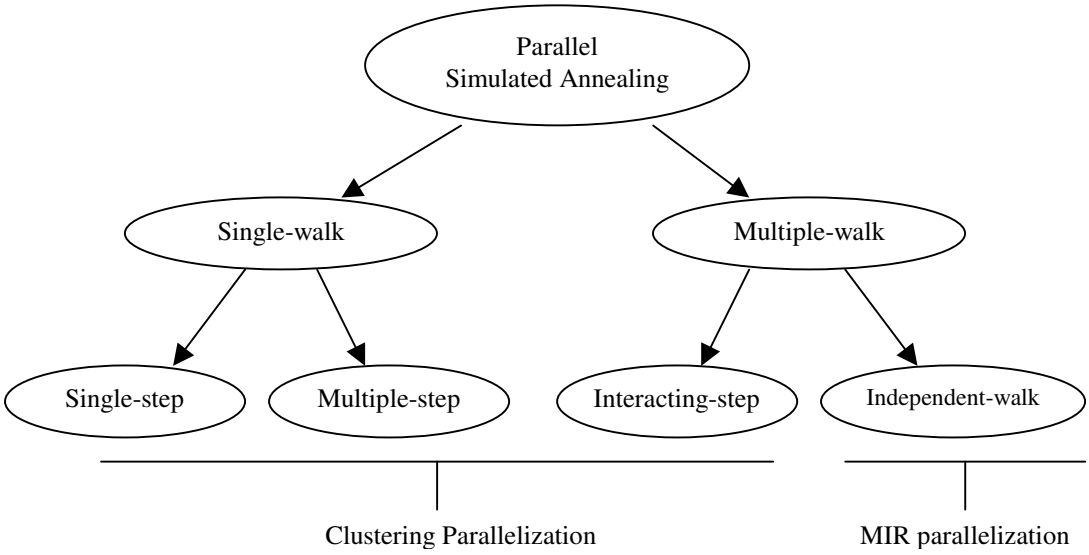

Figura 4.4 - Classificação das abordagens paralelas para SA, KLIEWER (2000).

4.2.1 Clustering Parallelization

Este modelo apresenta uma interessante abordagem na busca de soluções de ótima qualidade. Segundo KLIVER (2000). No início da otimização, a temperatura é alta e portanto o grau de aceitação também é alto (acima de 90%). Próximo do fim do processo, quando a temperatura é baixa, o grau de aceitação é baixo. (somente de 2 a 5% das soluções geradas são aceitas neste estágio do algoritmo). Isto gera uma grande ociosidade que pode ser evitada quando geradas soluções em paralelo. Clustering parallelization inicia com cada

processador avaliando caminhos individuais no espaço de busca de soluções, e à medida que o grau de aceitação diminui, os processadores são agrupados dentro de clusters. Um cluster trabalha sobre uma simples cadeia do algoritmo SA e todos os processadores do cluster têm uma solução comum a cada passo do SA.

O modelo como um todo consiste de muitos clusters que se comunicam após cada resfriamento concluído. Neste estágio o algoritmo de clustering pode ser classificado como "Interacting multiple-walk". Já na última fase, todos os processadores podem ser reunidos em um grande cluster, funcionando como um algoritmo "Single-walk" (*figura 4.5*).

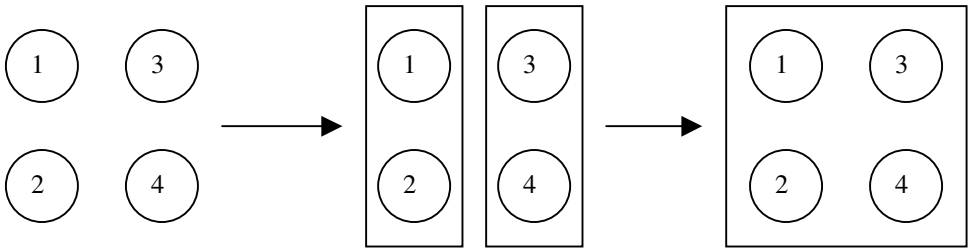

Figura 4.5 - Esboço da evolução do algoritmo Clustering

Comunicação dentro do cluster: Todos os processadores dentro do cluster procuram em paralelo por uma solução aceitável. Se uma ou mais soluções são aceitas o evento deverá ser comunicado entre os processadores e a nova solução deverá ser transmitida a todos os processadores do cluster.

Comunicação entre os clusters: Após cada resfriamento estar completo, os clusters comunicam-se e uma nova solução é escolhida para ser processada em todos os clusters. Na comunicação entre clusters é decidido quando novos clusters serão construídos.

Outros estudos discutem abordagens diferentes para os algoritmos SA em paralelo. Temos em CHEN et al. (1998) a descrição de três abordagens principais: *serial-like, altered generation* e *asynchronous*. O algoritmo Serial-like procura dividir o problema de avaliação das soluções entre os processadores, controlando as modificações independentemente em paralelo. Altered Generation muda o algoritmo de forma a aceitar várias modificações em paralelo. Asynchronous permite realizar modificações em paralelo, onde cada processador

ignora as modificações realizadas pelos outros processadores. Este método induz erros no calculo de custo da solução, mas traz a vantagem de reduzir os custos de comunicação.

Porém a mais conhecida classificação para implementações do SA em paralelo, proposta por FLYNN (1966) e LEE (1995) baseia-se segundo a arquitetura dos sistemas em quatro classes gerais, sendo elas: SISD (single instruction, single data), representa o clássico algoritmo SA seqüencial, SIMD (single instruction, multiple data), MISD (multiple instructions, single data) e MIMD (multiple instructions, multiple data), baseado portanto na utilização da memória, bem como pelo método utilizado para troca de informações, podendo ser da forma "shared memory" ou "distributed memory". LEE (1995) classifica as pesquisas de SA em paralelo em cinco grupos, sendo eles: a) "divide-and-conquer", b) "Speculative parallel annealing", c) "Parellel moves (PMSA), d) "Systolic parallel annealing" e e) "Multiple independent runs (MIR).

No próximo capítulo será apresentado o modelo a ser estudado, sendo este constituído de uma abordagem híbrida em paralelo utilizando recursos dos Algoritmos Genéticos (crossover e seleção elitista) com a estrutura de controle do fluxo de execução do algoritmo Simulated Annealing. Para a comunicação e o sincronismo dos processos será utilizado o modelo dos Sistemas Distribuídos denominado de Memória compartilhada em ambiente pseudo paralelo.

CAPÍTULO V
MODELO PROPOSTO

Neste capítulo, é apresentado o modelo implementado neste trabalho, modelo este que combina os Algoritmos Simulated Annealing (SA) e Genético (AG) visando obter melhores soluções para Problema do Caixeiro Viajante (PCV). Será abordado inicialmente o ambiente computacional utilizado e então a descrição do algoritmo desenvolvido.

5.1 Arquitetura Base

Para o desenvolvimento do modelo proposto, foi adotado o ambiente denominado de *"Memória compartilhada em ambiente pseudo paralelo"* apresentado no capítulo 4. Este modelo é constituído de somente um processador executando várias threads, onde cada thread representa uma instância do algoritmo Simulated Annealing.

A programação concorrente, através de Threads, permite simular um ambiente distribuído e o paralelismo dos sistemas multiprocessados de forma muito mais simples e prática, pois não é necessária a existência física de um cluster de máquinas, sendo que o comportamento de ambos é idêntico, permitindo desta forma a migração deste modelo para um modelo fisicamente distribuído, que trará como principal vantagem o tempo de processamento.

Temos como principal motivo para o desenvolvimento do Simulated Annealing em Paralelo, o fato de o algoritmo ser probabilístico, permitindo que para cada execução do algoritmo, um diferente resultado seja alcançado. Assim a execução de várias instâncias em paralelo do algoritmo Simulated Annealing, nos traz a vantagem de podermos analisar rapidamente vários resultados obtidos e escolher o de menor custo, como também nos permite aplicar alguns recursos dos algoritmos Genéticos buscando melhorar os resultados alcançados. Relembrando o capítulo 2, temos que o algoritmo Simulated Annealing parte de uma temperatura inicial elevada, e esta vai resfriando até o congelamento do sistema. Busca-se desta forma uma situação de equilíbrio a cada redução de temperatura realizada, podendo ser aceitos estados de maior energia conforme a condição de aceite definida, procurando assim

escapar de mínimos locais do espaço de busca trabalhado.

Considera-se que para a execução do algoritmo Simulated Annealing precisamos definir alguns parâmetros, tais como espaço de busca, temperatura inicial, função de aceite, programa de resfriamento e critério de finalização. Será adotado para o modelo distribuído proposto que os parâmetros iniciais serão iguais para todas as instâncias do algoritmo Simulated Annealing executadas, pois estas estarão otimizando o mesmo problema inicial apresentado. Poucos estudos são encontrados na literatura com tratamento de resfriamento diferenciado; vemos em CHEN et al. (1998) um estudo considerando esta abordagem.

Como em todo modelo distribuído, dispomos de uma estrutura de controle, que inicializa as instâncias SA, controla os resultados obtidos e apresenta a melhor ou as melhores soluções encontradas. Vemos na Figura 5.1 esta situação inicial.

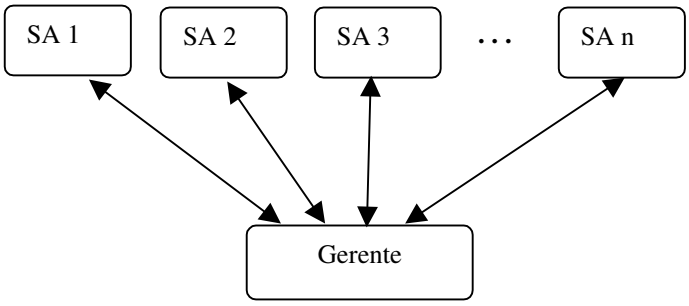

Figura 5.1 - Modelo base do Simulated Annealing em Paralelo

5.2 Modelo Desenvolvido

A combinação dos algoritmos Simulated Annealing e Genéticos em um ambiente distribuído segue os seguintes passos:

1. Definição dos parâmetros de entrada:
    a) Problema a ser solucionado;
    b) Quantidade de threads (instâncias do SA);
    c) Parâmetros Annealing;
    d) Parâmetros Genéticos.

2. Criação pelo Gerente de n instâncias SA, juntamente com os parâmetros iniciais;
3. Repete enquanto não atingir condição de término para todas as instâncias:
    a) Executa independentemente cada uma das instâncias do SA com os parâmetros fornecidos, numa determinada temperatura;
    b) Quando uma instância atingir o equilíbrio em dada temperatura, esta comunica ao gerente tal condição e aguarda novas instruções;
    c) Após todas as instâncias do SA estiverem estabilizadas e aguardando, o gerente intervêm, analisando os resultados intermediários e aplica os operadores "elitismo" e "crossover" dos algoritmos genéticos;
    d) O gerente envia para cada uma das instâncias SA uma solução selecionada e/ou transformada pelos operadores genéticos;
    e) O gerente libera as instâncias para a execução na nova temperatura, e aguarda o novo equilibro do sistema, retornando ao Passo 3;
4. Liberação das instâncias SA, avaliando os resultados obtidos e apresentando os resultados obtidos.

5.2.1 Parâmetros de Entrada

Os parâmetros de entrada definem o modelo a ser analisado e de que forma isto será realizado. Como temos um conjunto de instâncias SA que serão executadas em paralelo, este é um importante parâmetro a ser definido.

Poucas instâncias (Threads) nos trazem um conjunto menor de possibilidades de resultados a serem avaliados e combinados pelo gerente, em contrapartida o tempo de execução será mais rápido. Para muitas instâncias do SA, teremos um conjunto maior de possibilidades para serem tratadas pelo gerente, porém no ambiente computacional utilizado teremos para o tempo de processamento total, a soma dos tempos individuais, acrescidos do "overhead" ocasionado pela intervenção do gerente. Esta situação não ocorreria se o modelo fosse simulado em um ambiente distribuído formado por multicomputadores, sendo portanto somente computado o tempo de execução de uma instância do algoritmo SA acrescido do "overhead" do gerente (processamento e comunicação).

A definição dos parâmetros Annealing, determina o comportamento da condição de aceite das soluções geradas dentro do intervalo de cada temperatura, bem como da condição de término (congelamento) do sistema, visto que todas as instâncias obedecem aos mesmos parâmetros a cada interação (nova temperatura). Temos os seguintes parâmetros:

1. Parâmetros de Aceite:

a) Temperatura Inicial: indica a possibilidade de aceitar ou rejeitar soluções. Altas temperaturas são mais suscetíveis a aceitar soluções, enquanto que baixas temperaturas não.

b) Alpha - α : indica o grau de convergência do sistema, um valor alto faz com que a temperatura reduza mais lentamente, e um valor baixo reduz mais bruscamente a temperatura. Temos para alpha: $\alpha \subset (0,1)$.

2. Parâmetros de Término

a) Passos: Indica a quantidade de interações até o congelamento do modelo;

b) Tentativas: Quantidade máxima de soluções rejeitadas em determinada temperatura;

c) Mudanças: Quantidade máxima de soluções aceitas em determinada temperatura;

Os valores padrão definidos pelo sistema como parâmetros Annealing, que poderão ser modificados pelo usuário antes do início da simulação, são os seguintes (Tabela 5.1):

TABELA 5.1 – VALORES PADRÃO COMO PARÂMETROS ANNEALING

Parâmetros	Valores (n – tamanho da população)
Temperatura Inicial	$\sqrt{n}$
Alpha	0,95
Passos	20 ln (n)
Tentativas	100 n
Mudanças	10 n

Fonte: HANSEN (1995).

Já os parâmetros aplicados aos operadores do algoritmo genético, determinam a forma como serão tratadas as soluções intermediárias geradas pelas diversas instâncias de SA executadas em paralelo, quando ocorrer o equilíbrio do sistema em cada uma das temperaturas.

Os resultados intermediários gerados serão ordenados conforme o seu custo, sendo que através da seleção "elitismo", serão sel ecionados as "n" primeiras soluções de menor custo, estas não sofrerão alterações genéticas e serão repassadas integralmente a "n" instâncias SA para continuar o processamento. Para as demais soluções, serão selecionados pares de soluções de forma aleatória, sendo realizada entre elas a operação genética "crossover". Os novos indivíduos gerados serão avaliados juntamente com seus pais, podendo ser adotado um dos seguintes caminhos:

- Escolha do melhor pai e do melhor filho;
- Escolha dos dois filhos gerados;
- Escolha dos dois melhores indivíduos.

Entre os tipos de crossover existentes, o modelo permite simulações baseadas em três modelos definidos na literatura, apresentados no Capítulo 3, sendo eles:

OX : Order Crossover;
PMX : Partially Mapped Crossover;
CX : Cycle Crossover.

O sistema também gera automaticamente valores padrão para os parâmetros Genéticos, dependendo da quantidade de instâncias (Threads) a serem executadas. Estes parâmetros também poderão ser modificados antes de iniciar a execução do modelo, conforme vemos na Tabela 5.2.

TABELA 5.2 - VALORES PRÉ DETERMINADOS COMO PARÂMETROS GENÉTICOS

Parâmetros	Valores (k: tamanho do crossover)
Elitismo	0
Crossover OX, PMX	Ponto de Corte 1: k * 0,45
	Ponto de Corte 2: k * 0,50
Crossover CX	Ponto de Início: 1
Seleção Crossover	Melhor Pai e Melhor Filho

5.2.2 O Algoritmo

Conforme vimos na Figura 5.1, temos dois elementos básicos na construção do modelo proposto, o primeiro trata-se do algoritmo Simulated Annealing propriamente dito, e o segundo elemento trata do gerente. O gerente deverá controlar todas as instâncias SA, a sincronização entre estas, bem como a aplicação dos operadores genéticos definidos.

Para a implementação do modelo foi utilizada a ferramenta Delphi versão 6.0, tendo esta a linguagem Pascal como base de sua programação.

5.2.2.1 O Algoritmo Simulated Annealing Básico

A implementação do algoritmo Simulated Annealing foi baseado no modelo proposto por HANSEN (1995), sendo que seu algoritmo é relativamente simples.

Temos como as principais estruturas definidas para o algoritmo, aquelas que definem uma cidade em termos de coordenados no plano e a que representa uma rota, conforme vemos a seguir.

a) Cidade: uma cidade é definida por duas coordenadas do tipo real, que representam a localização da cidade em relação ao plano no qual as mesmas se encontram, sendo representada por uma estrutura de dados do tipo registro, com a seguinte notação:

```
type TCidade = record
                 x, y : real;
               end;
```

b) Rota: uma rota de *n* cidades é definida como uma seqüência de cidades que devem ser visitadas pelo viajante, sendo esta representada por uma estrutura de dados do tipo vetor de *n* elementos, do tipo cidade, com a seguinte notação.

```
type TRota = array [1..n] of TCidade;
```

Definidas as principais estruturas de dados necessárias a implementação do algoritmo SA, veremos a seguir os principais procedimentos que merecem destaque, pois são críticos dentro do contexto geral da implementação. São procedimentos pequenos e bastante funcionais, que os tornam de fácil entendimento.

a) Procedimento Distância: Este procedimento calcula a distância entre duas cidades quaisquer através de suas coordenadas utilizando a métrica euclidiana.

```
function Distancia (p, q : TCidade ) : real;
 var _dx, _dy : real;
 begin
   _dx := q.x - p.x;
   _dy := q.y - p.y;
   Distancia := sqrt(_dx*_dx + _dy*_dy)
 end;
```

b) Procedimento Custo: O vendedor visita as cidades em ordem numérica 1,2,…,n antes de retornar para cidade de número 1. O custo da rota é a soma das distâncias entre as cidades sucessivas.

```
function Custo (PRota : TRota; PQtdPontos : integer) : real;
 var _i : integer;
     _sum : real;
 begin
   _sum := Distancia(PRota[PQtdPontos], PRota[1]);
   for _i:= 1 to PQtdPontos-1 do
     _sum := _sum + Distancia(PRota[_i],PRota[_i+1]);
   Custo := _sum;
 end;
```

c) Procedimento Annealing: é procedimento principal do algoritmo, controla a redução da temperatura e a condição de término de otimização do modelo.

```
procedure Anneal(var Rt : TRota; Tmax, alpha : real;
                 passos, tentativas, mudanças : integer);
 var _T : real;
     _k: integer;
 begin
   _T := Tmax;
   for _k := 1 to passos do begin
     Pesquisa (Rt,_T,tentativas,mudanças);
     _T := alpha * _T
   end;
 end;
```

d) Procedimento Pesquisa: Realiza a busca por novas rotas até atingir o equilíbrio numa determinada temperatura; o equilíbrio é alcançado quando atingir o limite do numero de tentativas, ou do número de mudanças realizadas.

A pesquisa seleciona uma nova rota, verifica pela condição de aceite a rota selecionada, caso atinja um resultado favorável realiza a alteração da rota pela nova rota selecionada.

```
procedure Pesquisa(var Rt: TRota; T: real;
                   tentativas, mudanças : integer);
 var _i, _j, _nt, _nm: integer;
     _dE: real;
 begin
   _nt := 0;
   _nm := 0;
   while (_nt < tentativas) and (_nm < mudanças) do begin
     seleciona(Rt,_i,_j,_dE);
     if aceita(_dE,T) then begin
       troca(Rt,_i,_j);
       nm := nm + 1
     end;
     nt := nt + 1
   end
 end;
```

e) Procedimento Seleciona: seleciona randomicamente duas cidades e verifica a possibilidade de trocá-las, conforme a variação da energia causada pela alteração da rota da cidade de número *si* para a cidade de número *j* (movimento 2-*opt)*. A diferença de energia *dE* é calculada usando as coordenadas das duas cidades e de suas sucessoras, não havendo necessidade de calcular o custo total das rota, mas apenas verificar se teve ganho ou perda de energia com os arcos alterados.

```
procedure Seleciona (n : integer; var Rt : TRota;
                     var si, j : integer;
                     var dE : real );
 var _i, _sj : integer;
 begin
   Gera ( n, _i, j );
   si := _i mod n+1;
   _sj := j mod n+1;
   if (  _i <> j )   then
     dE := Distancia (  Rt [ _i ], Rt [   j ] ) +
           Distancia (  Rt [ si ], Rt [ _sj ] ) -
           Distancia (  Rt [ _i ], Rt [  si ] ) -
           Distancia (  Rt [  j ], Rt [ _sj ] )
   else
     dE := 0.0
 end;
```

f) Procedimento Troca: Realiza a mudança na rota entre as cidades *i* até *j*. O número de cidades no trajeto é denotado por n_{ij}. LIN (1965) appud ARAUJO (2001) determina que o critério usado é uma variante da idéia onde, de uma rota, por exemplo, c_1, c_2, c_3, ..., c_n, são selecionadas randomicamente duas cidades c_i e c_j, com suas sucessoras representadas respectivamente por c_{si} e c_{sj}. Tendo portanto a rota:

..., c_i, c_{si}, ..., c_j, c_{sj}, ..., determinará uma nova rota gerada pela reversão na ordem das cidades de c_{si} a c_j, que passa a ter a seguinte configuração: ..., c_i, c_j, ..., c_{si}, c_{sj}, ...

Este mecanismo provoca o mesmo efeito numa rota que a heurística *2-opt*.

```
procedure Troca (n : integer; var Rt : TRota; i, j : integer);
 var _k, _nij : integer;
 begin
   _nij := ( j - i + n ) mod n + 1;
   for _k := 1 to _nij div 2 do
     Swap (Rt,(i+_k-2)mod n+1,(j-_k+n)mod n+1);
 end;
```

g) Procedimento Aceita. Este pode ser determinístico ou probabilístico. Como se trata de uma abordagem probabilística, será utilizado o critério probabilístico. Uma rota de custo menor ou que não sofreu mudança será sempre aceita. Uma rota de custo maior será aceita com probabilidade $e^{-dE/T}$, comparada a um número randômico gerado entre 0 e 1. Temos portanto que para altas temperaturas maior probabilidade de aceitar rotas com maior energia (custo mais alto), enquanto que para baixas temperaturas, a probabilidade de aceitar soluções piores é reduzida.

```
function Aceita ( dE, temp : real ) : boolean;
 begin
   if ( dE > 0.0 ) then
     Aceita := exp ( - dE / temp ) > random
   else
     Aceita := true;
 end;
```

5.2.2.2 O Algoritmo Genético - Crossover

A implementação da operação Crossover do Algoritmo Genético foi baseada nos métodos apresentados em MICHALEWICZ (1999), sendo implementados os modelos OX e

PMX estes baseados em dois pontos de corte, e o modelo CX baseado na definição de um único ponto de início do processo. Detalhamentos maiores sobre estes modelos de cruzamento podem ser vistos no Capítulo 3.

Vemos a seguir os algoritmos implementados.

a) Crossover OX - (Order Crossover):

```
procedure Crossover_OX (var PR1, PR2 : TRota);
var AR, AR1, AR2, APR1, APR2 : TRota;
    AFilho, Ai, Aj : integer;
    AFim : boolean;
begin
  for AFilho := 1 to 2 do begin
   case AFilho of
     1 : begin  AR1 := PR1;  AR2 := PR2;  end;
     2 : begin  AR1 := PR2;  AR2 := PR1;  end;
   end;
   for Ai := 1 to FQtdPontos do
     if (Ai>=FCrossover.Pto1) and (Ai<=FCrossover.Pto2) then
       AR[Ai] := AR1[Ai]
     else
       AR[Ai].x := null;
   Ai := FCrossover.Pto2+1;
   Aj := Ai;
   AFim := False;
   while not AFim do begin
     while Existe_Cidade_na_Rota (AR,AR2[Aj]) do begin
       Aj := Aj + 1;
       if Aj > FQtdPontos then Aj := 1;
     end;
     AR[Ai] := AR2[Aj];
     Aj := Aj + 1;  if Aj > FQtdPontos then Aj := 1;
     Ai := Ai + 1;  if Ai > FQtdPontos then Ai := 1;
     AFim := Ai = FCrossover.Pto1;
   end;
   case AFilho of
     1 : APR1 := AR;
     2 : APR2 := AR;
   end;
  end;
  PR1 := APR1;
  PR2 := APR2;
end;
```

b) *Crossover PMX - (Partially Mapped Crossover):*

```
procedure Crossover_PMX (var PR1,PR2 : TRota);
  var AR1, AR2 : TRota;
     Ai, Aj, Ak, Ak_old : integer;
 begin
   for Ai := 1 to FQtdPontos do
     if (Ai >=FCrossover.Pto1) and
        (Ai<=FCrossover.Pto2) then begin
       AR1[Ai] := PR1[Ai];
```

```
      AR2[Ai] := PR2[Ai];
    end
    else begin
      AR1[Ai] := PR2[Ai];
      AR2[Ai] := PR1[Ai];
    end;
  for Ai := FCrossover.Pto1 to FCrossover.Pto2 do begin
    Aj := GetPosForaFaixa(AR1,AR1[Ai]);
    if Aj <> -1 then begin
      Ak := Ai;
      repeat
        Ak_old := Ak;
        Ak := GetPosFaixa (AR1,PR2[Ak]);
      until Ak = -1;
      AR1[Aj] := AR2[Ak_old];
    end;
    Aj := GetPosForaFaixa(AR2,AR2[Ai]);
    if Aj <> -1 then begin
        repeat
        Ak_old := Ak;
        Ak := GetPosFaixa (AR2,PR1[Ak]);
      until Ak = -1;
      AR2[Aj] := AR1[Ak_old];
    end;
  end;
  PR1 := AR1;
  PR2 := AR2;
end;
```

c) Crossover CX - (Cycle Crossover):

```
procedure Crossover_CX (var PR1, PR2 : TRota);
 var AR1, AR2 : TRota;
 Ai, APosInicio : integer;
 begin
   for Ai := 1 to FQtdPontos do begin
     AR1[Ai] := null;
     AR2[Ai] := null;
   end;
   APosInicio := Crossover.PosIni;
   Ai := APosInicio;
   repeat
     AR1[Ai] := PR1[Ai];
     AR2[Ai] := PR2[Ai];
     Ai := GetPosicao (PR1,PR2[Ai]);
   until Ai = APosInicio;
   for Ai := 1 to FQtdPontos do begin
       if AR2[Ai] = null then AR2[Ai] := PR1[Ai];
   end;
   PR1 := AR1;
   PR2 := AR2;
 end;
```

5.2.2.3 O Gerente

Como já foi comentado, temos no gerente o principal elemento de todo o processo, pois ele é o responsável pela inicialização e sincronização das instâncias SA, avaliação, seleção e transformação dos resultados intermediários e por fim, liberação das

instâncias e apresentação dos resultados finais. Vemos na figura 5.2 este fluxo de controle.

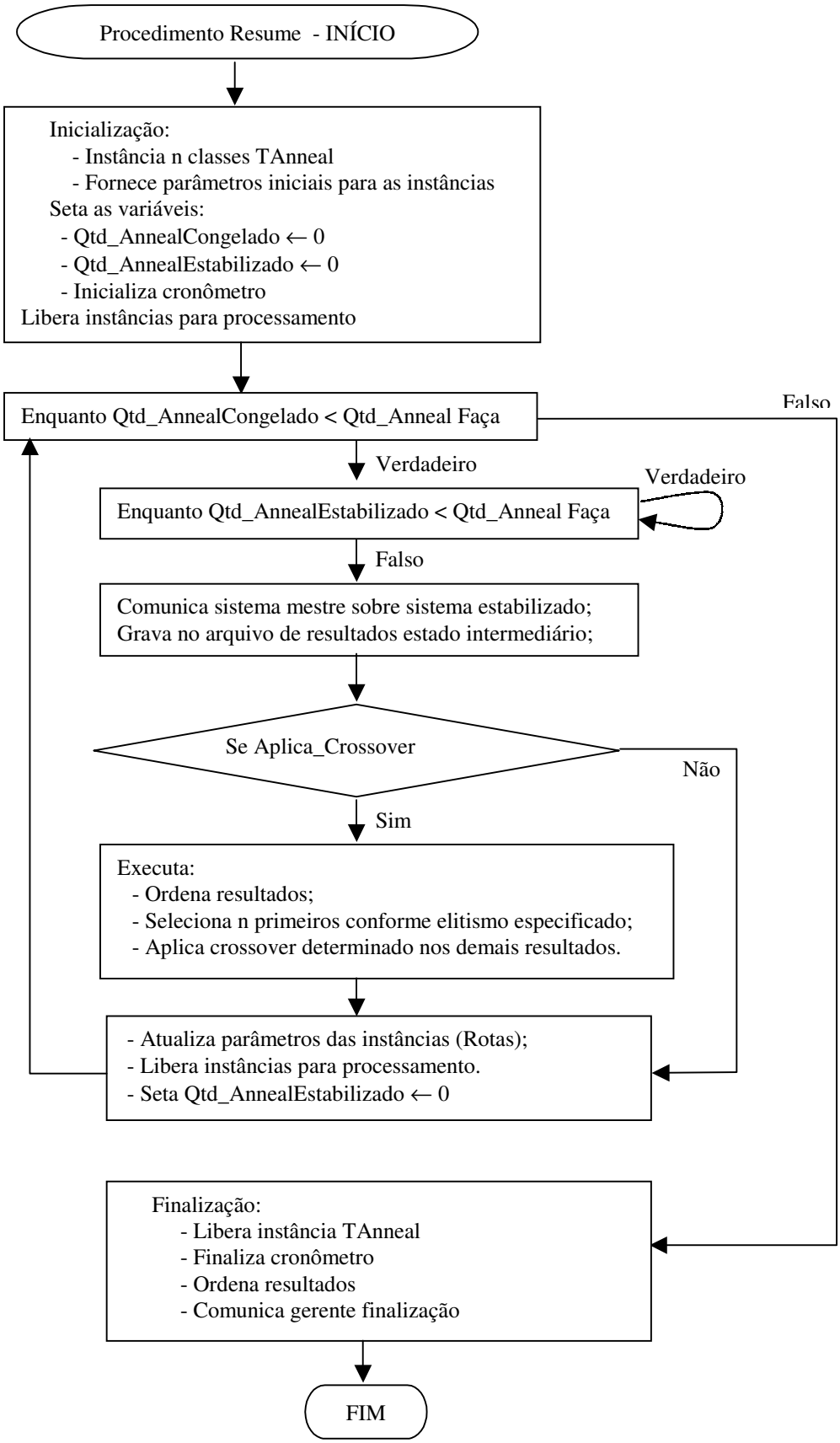

Figura 5.2 - Fluxo de controle do Gerente

Para a realização das atividades descritas, foi necessário alterar o algoritmo SA original, adicionando elementos de comunicação e sincronização. O mecanismo adotado para sincronização foi o semáforo, conforme apresentado no Capítulo 4, sendo que as instâncias do SA, quando atingem o equilíbrio numa determinada temperatura, acessam uma região exclusiva, registram o seu resultado e vão para o estado suspenso (bloqueado) aguardando liberação pelo gerente.

Para a linguagem utilizada Delphi, o acesso e liberação da região exclusiva ocorrem de acordo com os seguintes comandos:

```
...
WaitForSingleObject (hMutex,INFINITE);    = MutexBegin
  ... Região exclusiva ...
ReleaseSemaphore(hMutex,1,nil);           = MutexEnd
...
```

Como mecanismo de comunicação, foram adotadas variáveis comuns entre o gerente e cada uma das instâncias SA, sendo que estas registram os seus estados intermediários através da região exclusiva, quando atingem o equilíbrio em dada temperatura, ou quando atingem a condição de término.

Como cada uma das instâncias SA no modelo implementado é uma Thread. O algoritmo Simulated Annealing foi modelado como tal de forma a permitir a implementação da proposta. Temos no algoritmo Simulated Annealing básico, o procedimento Anneal como o procedimento inicial a ser executado, onde a partir deste, todos os demais procedimentos serão executados à medida que a execução evolui. Para o modelo em paralelo, foi necessário implementar o algoritmo Simulated Annealing através da classe Thread existente também em Delphi, conforme vemos a seguir:

```
TAnneal = class (TThread)
private
  FGerente : TAnneal_Gerente;
  FQtdPontos : integer;
  FParametros : TAnneal_Parametros;
  FEstado : ^TAnneal_Estado;
  procedure Calcula;
```

```
    procedure Set_TemperaturaEstabilizada;
    procedure Set_SistemaCongelado;
  protected
    constructor Create (PGerente : TAnneal_Gerente;
                        PQtdPontos : integer;
                        var PEstado : TAnneal_Estado;
                        PParametros : TAnneal_Parametros);
    procedure Execute; override;
end;
```

A classe *TAnneal* definida, será a base de todas as instâncias SA que serão executadas em paralelo. Quando da instanciação desta classe pelo gerente, deve o mesmo fornecer os seguintes dados:

- *PGerente:* Endereço do gerente;
- *PQtdPontos:* Quantidade de pontos da rota a ser otimizada;
- *PEstado:* Armazena estado inicial e a evolução do modelo para a instância em questão. Este parâmetro é do tipo referência, e será utilizado como canal de comunicação entre o gerente e a instância para fins de transferência dos resultados;
- *PParametros:* Parâmetros iniciais definidos: Temperatura, Alpha, Passos, Tentativas e Mudanças.

Os procedimentos básicos existentes na classe *Tanneal* são ao total de quatro. Estes procedimentos são os responsáveis pela execução e comunicação com o gerente sobre a evolução da instância. Temos:

a) Procedimento Execute: Este procedimento é obrigatório para todas as classes herdadas a partir da classe TThread, nas quais desejamos definir comportamentos específicos. É a partir deste procedimento que a instância iniciará sua execução, porém o gerente não poderá executá-lo diretamente, e sim através do comando *Resume*. O corpo deste procedimento simplesmente realiza uma chamada ao procedimento calcula.

b) Procedimento Calcula: Aqui está modelado todo o algoritmo Simulated Annealing, com algumas modificações no procedimento Anneal básico apresentado anteriormente. Foram adicionadas chamadas aos procedimentos Set_TemperaturaEstabilizada e Set_Sistema Congelado, nos locais onde estas situações ocorrem, permitindo desta forma comunicar ao gerente sobre tal situação. Temos para este procedimento:

```
procedure TAnneal.Calcula;
 begin
   while FEstado.Passo <= FParametros.Passos do begin
     Pesquisa;
     Set_TemperaturaEstabilizada;
     FEstado.Passo := FEstado.Passo + 1;
     FEstado.Temperatura := FParametros.Alpha *
                            FEstado.Temperatura;
   end;
   Set_SistemaCongelado;
 end;
```

c) Procedimento Set_TemperaturaEstabilizada: Como vemos no procedimento TAnneal.Calcula, este procedimento é chamado após ter sido realizada uma pesquisa numa dada temperatura, sendo que neste momento o sistema está estabilizado e pronto para reduzir sua temperatura para a nova etapa de pesquisa. Este procedimento então comunica o gerente sobre a situação ocorrida, e bloqueia a instância (*comando Suspend*) aguardando instruções e liberação pelo gerente para a nova etapa a ser realizada. Temos como código para este procedimento:

```
procedure TAnneal.Set_TemperaturaEstabilizada;
 begin
   MutexBegin;
     FAnneal_Estabilizado[idAnneal] := True;
   MutexEnd;
   Suspend;
 end;
```

d) Procedimento Set_SistemaCongelado: Este procedimento, semelhante ao anterior, é chamado quando a instância encontrou a condição de término *(FEstado.Passo > FParametros.Passos)* e comunica ao gerente a situação ocorrida. O que difere do procedimento anterior é o fato deste não necessitar bloquear a instância, pois esta já convergiu para alguma solução e não será mais necessária para a continuidade do sistema. Temos:

```
procedure TAnneal.Set_SistemaCongelado;
 begin
   MutexBegin;
   FAnneal_Congelado[idAnneal] := True;
   MutexEnd;
 end;
```

O gerente foi modelado através de uma classe TObject, sendo implementado procedimentos e propriedades de forma a facilitar a sua compreensão e a busca dos resultados alcançados após a execução do modelo. Temos abaixo apresentado os principais procedimentos e propriedades do objeto Gerente.

```
TGerenteAnneal = class (TObject)
 public
   procedure SetParametros (PQtdAnneal : integer;
                            PRotaInicial : TRota;
                            PQtdPontos : integer;
                            PAnneal_Par : TAnneal_Parametros;
                            PAplica_Crossover : boolean;
                            PCrossover_Par: TCrossover_Param);
   procedure Resume;
   property Anneal_RotaResult [PiProc : integer] : TRota;
   property Anneal_Custo [PiProc : integer] : real;
   property Anneal_idMin : integer;
   property HInicio : TTime;
   property HFim : TTime;
end;
```

O desenvolvimento da classe *TGerenteAnneal* permite aplicar o modelo implementado como um módulo de outro sistema maior, bastando para tanto instanciar a classe *TGerenteAnneal* , suprir os parâmetros de acordo com o problema a ser otimizado, e iniciar a execução do modelo. Por fim os resultados serão disponibilizados através das propriedades modeladas. Abaixo estão descritos os procedimentos e propriedades principais:

Procedimento Set_Parametros: Procedimento inicial, onde são fornecidas as informações sobre o modelo a ser otimizado, quantidade de instâncias a serem utilizadas, os parâmetros Anneal e Crossover. Pode-se optar pela não aplicação do operador crossover ao longo de uma simulação (parâmetro: *PAplica_Crossover : boolean*);

Procedimento Resume: Procedimento inicial do sistema, através de sua execução o gerente inicia a otimização do problema fornecido. Adiante trataremos detalhadamente sobre este procedimento, base de toda a gerência;

Property Anneal_RotaResult: fornece a rota resultante para o índice da instância fornecida;

Property Anneal_Custo: fornece o custo da rota calculada pela instância identificada pelo índice fornecido;

Property Anneal_idMin: retorna o índice da instância de menor custo calculado;

Property HInício: retorna a hora do início da otimização;

Property HFim: retorna a hora de termina da otimização.

Trataremos a seguir basicamente sobre o *procedimento Resume*, pois é através deste que todo o mecanismo de controle e de sincronização do modelo proposto está implementado.

O procedimento inicia instanciando *n* (Qtd_Anneal) classes TAnneal, conforme a quantidade fornecida no procedimento *Set_Parâmetros*. A seguir fornece para cada instância criada os valores iniciais do problema, zera os contadores de quantidade de instâncias congeladas (finalizadas) e estabilizadas, inicializa o cronômetro, libera as instâncias para processamento e aguarda o resultado de todas as instâncias em cada uma das temperaturas.

Os comandos seguintes ao processamento de inicialização são os responsáveis por todo o fluxo de controle sobre das instâncias. Temos duas estruturas "*enquanto ... faça"* aninhadas, sendo que a mais externa controla se as instâncias atingiram o estado "Congelado" e a mais interna verifica o estado "Estabilizado". A Figura 5.3 esquematiza a sincronização destas estruturas juntamente com as instâncias SA criadas.

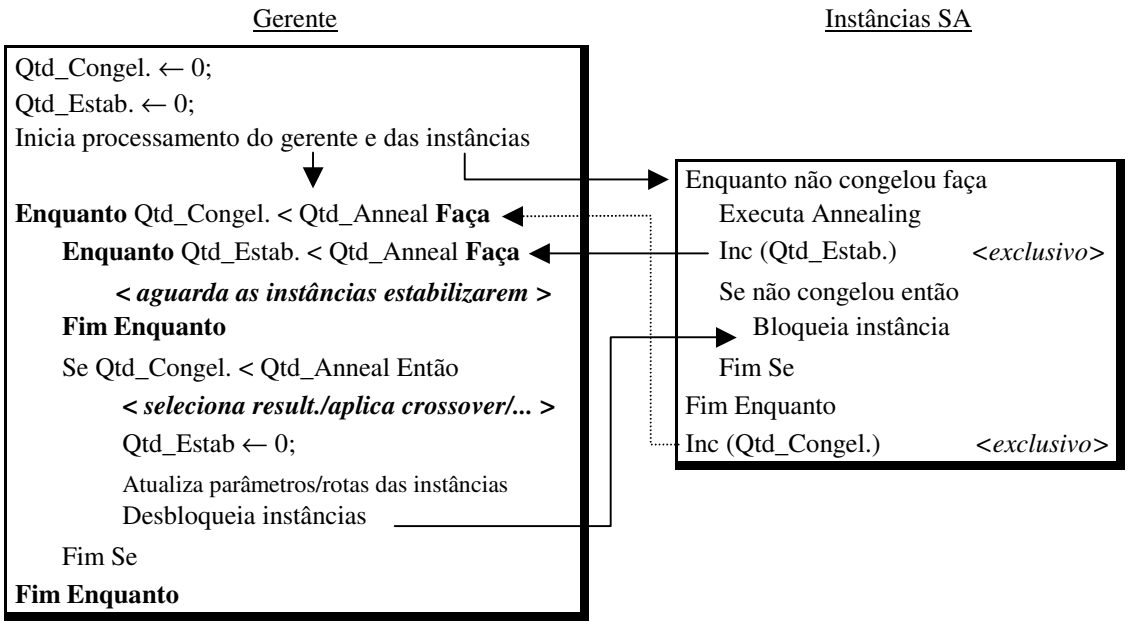

Figura 5.3 - Relacionamento do Gerente com as Instâncias SA.

Tanto o gerente quanto as instâncias podem estar em dois estados, executando ou bloqueado. Quando as instâncias estão executando, o gerente está bloqueado, e quando as instâncias estão bloqueadas, o gerente está executando.

O gerente no estado bloqueado, significa que este está aguardando as instâncias concluírem mais um passo do algoritmo Simulated Annealing. O bloqueio do gerente se faz pelo seguinte laço infinito:

Enquanto Qtd_Estabilizado < Qtd_Anneal Faça
Fim Enquanto;

À medida que as instâncias vão estabilizando na temperatura proposta, estas vão incrementando a variável *Qtd_Estabilizado* e, caso não tenham atingido o último passo (sistema congelado), executam o comando *Suspend* que as levam ao estado bloqueado (parado). Se a instância encerrou a quantidade de passos a serem executados, esta não se bloqueia e finaliza o seu processamento incrementando a variável *Qtd_Congelado*.

Após o gerente ter sido desbloqueado, este verifica se as instâncias atingiram o congelamento do algoritmo. Caso negativo, este avalia os resultados, aplica os operadores genéticos, atualiza as instâncias como os novos parâmetros e desbloqueia (reativa) as mesmas através do comando *Resume*, voltando novamente para o seu estado bloqueado (laço infinito). Com o congelamento das instâncias, o gerente avalia os resultados, paralisa o cronômetro e libera as instâncias SA alocadas, comunicando a aplicação principal sobre o término da otimização.

Temos desta forma modelado todo o mecanismo de controle e sincronização entre o Gerente e as instâncias SA, com a aplicação de Threads e semáforos.

5.2.2.4 Tratamento Aplicado às Soluções Intermediárias

Foram apresentados até o momento, o comportamento do modelo implementado com relação ao funcionamento do gerente, instâncias SA, relacionamento e sincronização entre eles. Veremos a seguir o tratamento aplicado pelo gerente com relação ao conjunto das soluções intermediárias geradas pelas diversas instâncias SA, todas as vezes que o modelo atinge o equilíbrio em determinada temperatura.

A cada interação, terá o gerente um conjunto de *n* soluções geradas pelas *n* instâncias

SA alocadas. Caso o modelo não esteja configurado para avaliar e cruzar as soluções, conforme definido no procedimento *Set_Parametros* da classe TGerenteAnneal, o gerente simplesmente repassa as soluções intermediárias recebidas às respectivas instâncias e dá seqüência a otimização do problema proposto. Vemos na Figura 5.2 representada esta situação.

A disponibilidade do modelo em realizar otimizações sem a avaliação e o cruzamento das soluções intermediárias, ou seja, executar o algoritmo Simulated Annealing na sua forma pura, porém em paralelo, é devido a necessidade de termos resultados que permitam avaliar se a aplicação das operações genéticas (elitismo e crossover) melhoram os resultados obtidos.

Temos no cruzamento das soluções intermediárias, um dos principais motivos que nos levaram ao estudo deste modelo. Este é um método híbrido, que utiliza a operação denominada de crossover aplicada em algoritmos genéticos com o algoritmo Simulated Annealing em Paralelo.

Conforme visto a respeito dos parâmetros de entrada necessários a configuração do comportamento do gerente frente às soluções intermediárias, temos na Figura 5.4 a representação deste comportamento.

Acompanhando o fluxo apresentado, assim que o gerente recebe todas as soluções intermediárias, o mesmo calcula o custo individual de cada uma, organiza as mesmas em ordem crescente de custo, preparando desta forma o conjunto de soluções a serem tratados pelas operações genéticas.

O parâmetro *Elitismo* indica a quantidade de melhores soluções que não sofrerão nenhuma alteração, ou seja, que não participarão dos cruzamentos existentes. Estas soluções serão guardadas e juntamente com os resultados dos cruzamentos das demais soluções, formarão o novo conjunto de soluções base para a próxima interação das instâncias SA.

A seleção das melhores soluções, sem alterá-las, deve-se ao fato de necessitarmos manter ao menos um conjunto de ótimas soluções a serem apresentadas como solução final

caso os cruzamentos não produzam bons resultados futuros.

O cruzamento propriamente dito será aplicado a todos os demais indivíduos que não foram incluídos no "conjunto elite". Seu processamento consiste em selecionar pares de soluções de forma aleatória HIROYASU et al. (2000), e realizar o cruzamento conforme uma das opções definidas, podendo estas ser: Crossover OX, PMX ou CX.

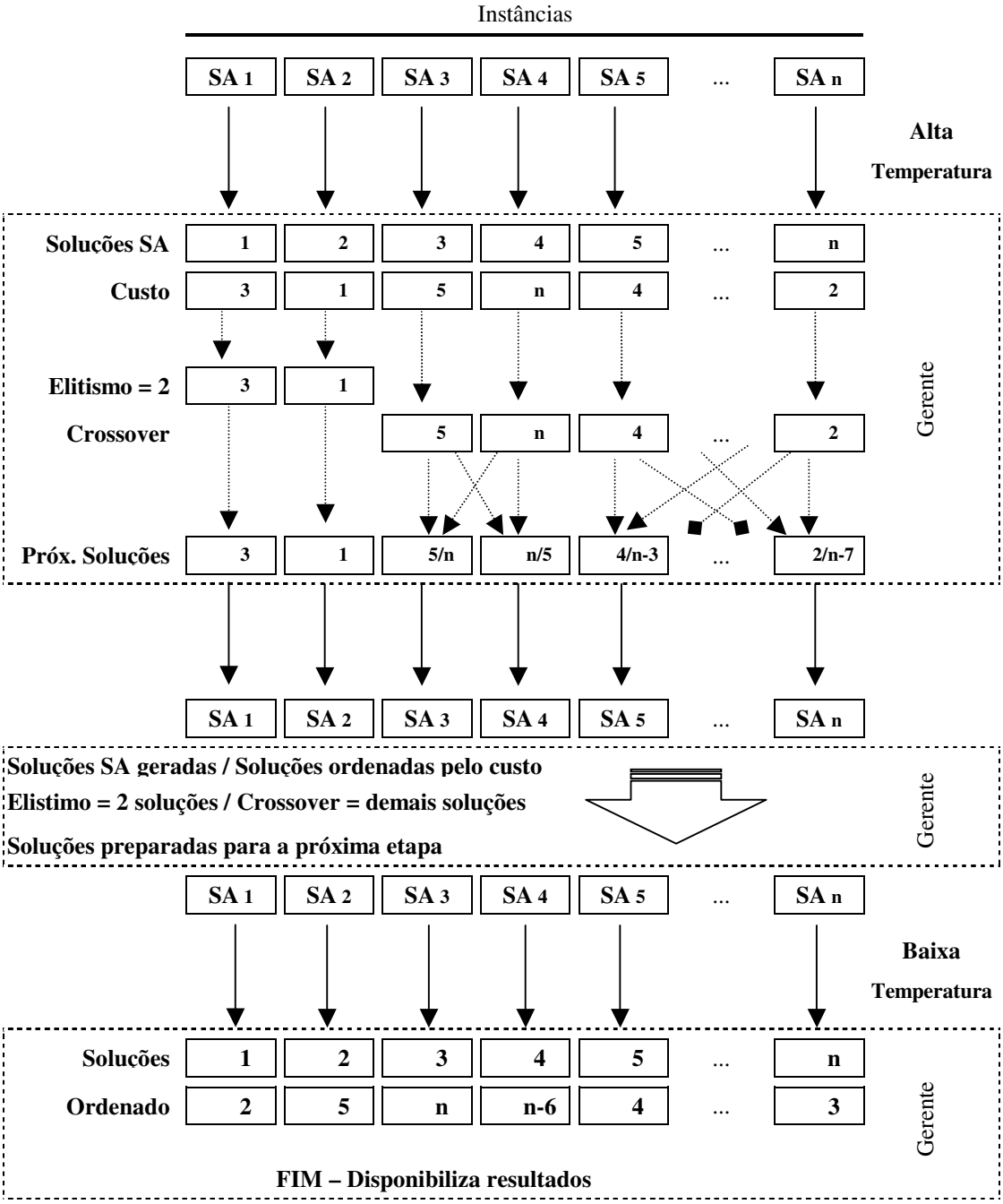

Figura 5.4 - Tratamento das soluções intermediárias encontradas.

Durante a seleção dos pares aleatórios de soluções a serem cruzados, será permitido que as soluções participem somente uma única vez dos cruzamentos. Vale ressaltar que para isto, devemos ter para as soluções restantes que participarão dos cruzamentos, uma quantidade par de soluções, sendo portanto que a quantidade de elitismo será ajustada de forma a termos esta condição verdadeira. Para a implementação da seleção dos pares de soluções, respeitando a regra de que nenhuma solução poderá participar em mais de um cruzamento, foi necessário utilizar um vetor de dimensão equivalente a quantidade total de soluções processadas, sendo este do tipo lógico (booleano). Inicialmente este vetor apresenta todos as suas posições com valor "Falso", sendo atribuído valor "Verdadeiro" as *n* primeiras posições, onde n significa a quantidade de soluções elites selecionadas. Temos desta forma que as posições com valor "Falso" significam que a solução correspondente ainda não foi selecionada para cruzamento. À medida que as soluções forem selecionadas, as posições no vetor recebem valor "Verdadeiro". Ao final o vetor terá todas as suas posições com valores "Verdadeiro".

Depois de criada a estrutura de controle de cruzamentos, será executado um laço cujo tamanho será a quantidade restante de soluções dividido por dois, sendo a cada interação selecionadas duas soluções da seguinte maneira: Gera-se um número aleatório entre 1 e o tamanho do vetor, verifica-se se a posição do vetor correspondente ao número gerado tem valor "Falso", seleciona -se esta posição e atribui-se o valor "Verdadeiro", caso contrário, vai percorrendo o vetor a partir da posição gerada até localizar uma solução que ainda não tenha sido selecionada. Caso chegue ao final do vetor e ainda não foi selecionada nenhuma solução, o processo inicia a partir da primeira posição do vetor. Repete-se a seleção da solução para obtermos desta forma duas soluções (pais) que então serão cruzadas por um dos modelos de cruzamento determinados.

Após o cruzamento, teremos quatro soluções (dois pais e dois filhos), onde o gerente deve selecionar apenas duas que serão as soluções da próxima interação das instâncias, juntamente com as soluções elite e as soluções selecionadas dos demais cruzamentos. A seleção pelo gerente das duas soluções, ocorrerá de uma das seguintes formas: Escolha do melhor pai e do melhor filho; Escolha dos dois filhos gerados e Escolha dos melhores indivíduos. A figura 5.5 apresenta o algoritmo utilizado pelo gerente para a aplicação do crossover e seleção dos resultados, gerando o novo conjunto de soluções para a próxima interação das instâncias SA.

```
var _CrossAnneal : array [1.. nMaxProcess] of boolean;
       _R1, _R2 : TRota;
   begin
     ...
     Ordena (FAnnealRotas,FQtdAnneal);
     // Inicializa vetor de controle de cruzamentos
     for _i := 1 to FQtdAnneal do
       _CrossAnneal[_i] :=_i <= FQtdElite;
     // Varia para a quantidade de pares de soluções
     for _i := 1 to (FQtdAnneal-FQtdElite)/2 do begin
       // Seleção do par de rotas a serem cruzadas
       for _CrossSelecao := 1 to 2 do begin
         _Pos := random (FqtdAnneal-1)+1;
         while _CrossAnneal[_Pos] do begin
           _P := _Pos + 1;
           if _Pos = (FQtdAnneal+1) then
             _Pos := FQtdElite+1;
         end;
         _CrossAnneal[_Pos] := True;
         case _CrossSelecao of
           1 : P1 := _Pos;
           2 : P2 := _Pos;
         end;
       end;
       _R1 := FAnnealRotas[P1].Rota;
       _R2 := FAnnealRotas[P2].Rota;
       // Cruzamento das rotas
       case FCrossover_Parametros.Modelo of
          Modelo_OX : Crossover_OX (_R1,_R2);
         Modelo_PMX : Crossover_PMX (_R1,_R2);
          Modelo_CX : Crossover_CX (_R1,_R2);
       end;
       // Seleção dos resultados
       case FCrossoverModelo of
         Cr_MelhorPaiFilho : begin
             if Custo(FAnnealRotas[P1])>Custo(FAnnealRotas[P2]) then
               FAnnealRotas[P1].Rota := FAnnealRotas[P2].Rota;
             if Custo(_R1)<Custo(_R2) then
               FAnnealRotas[P2].Rota := _R1
             else
               FAnnealRotas[P2].Rota := _R2;
           end;
         Cr_Filhos : begin
             FAnnealRotas[P1].Rota := _R1;
             FAnnealRotas[P2].Rota := _R2;
           end;
         Cr_Melhores : begin
             Ordena_Resulatados (FAnnealRotas[P1].Rota,
                                 FAnnealRotas[P2].Rota,
                                 _R1,_R2);
             FAnnealRotas[P1].Rota := Primeiro_Colocado;
             FAnnealRotas[P2].Rota := Segundo_Colocado;
           end;
       end;
     end;
     ...
   end;
```

Figura 5.5 - Algoritmo do tratamento das soluções intermediárias

5.3 Considerações Finais

Vimos neste capítulo o modelo proposto. Este modelo apresentou várias alternativas de configuração, com relação às formas de aplicação do crossover e da seleção dos resultados. Serão avaliados no próximo capítulo os resultados obtidos através de várias configurações diferentes de parâmetros, desde o tamanho da rota, quantidade de elitismo selecionado, tipo de crossover aplicado e modo de composição dos resultados.

CAPÍTULO VI
ANÁLISE DOS RESULTADOS

A análise dos resultados busca verificar em função do método proposto, o estudo de indicadores que determinem para o Problema do Caixeiro Viajante, qual é o tamanho de rota e valores de parâmetros que a aplicação deste método se torna mais apropriada. Serão realizados testes com instâncias conhecidas permitindo assim comparar os resultados alcançados.

O modelo implementado não busca atingir recorde em quantidade de cidades analisadas, nem mesmo em reduzido tempo de processamento, pois como já aludido no capítulo anterior, caso este modelo tenha a sua implementação desenvolvida em um ambiente realmente distribuído, provavelmente seriam atingidos tempos melhores de processamento que os obtidos neste trabalho. A avaliação portanto, ocorrerá principalmente sobre o a qualidade e desempenho do método estudado.

6.1 Metodologia de Avaliação

Os resultados serão gerados por meio de um protótipo que implementa o modelo proposto. O protótipo gera além dos resultados finais da simulação, ou seja, a rota otimizada pelas várias instâncias alocadas, resultados intermediários que permitem avaliar a influência dos cruzamentos durante a evolução da simulação.

A avaliação será em função de três indicadores, sendo eles:

Qualidade da Solução: esta é a medida, em porcentagem, da diferença entre a solução encontrada e a melhor solução ótima conhecida, possibilitando desta forma verificar a qualidade do resultado obtido. É calculada pela seguinte fórmula:

$$Q = 1 - \frac{f(.) - f_{opt}}{f_{opt}} * 100 \text{, onde} \qquad (6.1)$$

$f(.)$ o custo da solução encontrada e,

f_{opt} o custo da solução ótima conhecida para aquela instância.

Desempenho: mede o número de vezes que o algoritmo alcançou a solução ótima, caso venha a acontecer. Para problemas muito complexos, ou seja, com uma quantidade muito grande de pontos, será muito difícil alcançarmos a solução ótima conhecida. É calculado através da seguinte fórmula:

$$P = \frac{Num_{opt}}{Num} * 100 \text{ , onde} \qquad (6.2)$$

Num_{opt} é o número de avaliações que alcançaram o ótimo global e
Num o número total de avaliações.

Cruzamentos: mede qual influência que a operação crossover tem na solução final alcançada, É calculado através da seguinte fórmula:

$$C = \frac{N_{Crossover}}{N_{Passos\ -\ 1}} * 100 \text{ , onde} \qquad (6.3)$$

$N_{Crossover}$ é a quantidade de vezes que a operação crossover foi aplicada na melhor solução encontrada. Este valor é obtido através da operação:

$N_{Crossover}$ = $N_{Passos-1}$ - Idade da melhor geração, onde a idade da melhor geração indica a quantidade de vezes que a melhor solução encontrada foi escolhida para a próxima seqüência de resfriamento do algoritmo SA.

$N_{Passos-1}$ é a quantidade de passos aplicadas até o resfriamento do problema. Observa-se que utilizamos a quantidade de (Passos-1) pois o crossover é somente aplicado até o penúltimo passo, pois no último passo já teremos uma solução localizada que não necessitará de qualquer operação além do ordenamento dos resultados.

Temos portanto como indicador de cruzamentos, um valor que indica a influência da operação de crossover sobre o resultado final alcançado. Porém como a operação de crossover é aplicada em todos os passos da simulação, é interessante saber em que momento ela teve sua maior influência.

Este estudo se torna muito relevante principalmente nos casos em que o indivíduo pai pode ser escolhido como participante da próxima geração, podendo desta forma termos como melhor resultado um indivíduo que não teve influência nenhuma dos demais, ou somente nos passos iniciais, levando assim a concluir que o crossover não contribui em nada na convergência do modelo.

Será analisado portanto, além da influência do crossover na solução final, a sua influência ao longo da simulação, permitindo assim verificar sua verdadeira aplicabilidade.

6.2 Origem das Instâncias

As instâncias do PCV utilizadas, são oriundas de duas fontes, a primeira e mais utilizada, a TSPLIB, esta é constituída de diversos problemas cujas soluções ótimas são conhecidas para a grande maioria deles. A segunda é as instâncias em grade, pois suas soluções ótimas podem ser facilmente calculadas.

a) Instâncias da TSPLIB: Apresenta em sua maioria problemas de natureza geométrica, obedecendo à métrica euclidiana. Esta biblioteca é mantida por REINELT (1991) *apud* ARAUJO (2001), e está disponível no endereço eletrônico *http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/index.html* com problemas que variam desde 14 até 85.900 cidades, sendo muitas delas baseada em distâncias de cidades reais.

b) Instâncias em grade: Embora não muito utilizadas para testes de algoritmos, estas possuem uma interessante particularidade que é a possibilidade de se poder calcular a rota de custo ótimo. Nas instâncias em grades, as cidades são colocadas nos vértices de uma grade regular quadrada no espaço euclidiano. A figura 6.1 mostra uma grade com 16 cidades. À distância entre duas cidades é determinada pelo comprimento do lado do quadrado ou pela distância da diagonal quando for o caso.

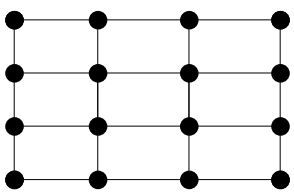

Figura 6.1 - Uma grade 4 x 4 que corresponde a 16 cidades

Para o PCV com as cidades distribuídas dessa forma, a solução ótima (a rota de menor tamanho) para uma determinada instância pode ser facilmente determinada, já que o custo f_{opt} corresponde às somas das distâncias, e pode simplesmente ser calculado em função do tamanho g do lado do quadrado, nos dois casos seguintes:

$$f_{opt} = \begin{cases} g(n) = n^2 & n = \text{número par de cidades,} \\ g(n) = n^2 - 1 + \sqrt{2} & n = \text{número ímpar de cidades.} \end{cases} \quad (5.3)$$

6.3 Simulações

As simulações serão realizadas variando diversos parâmetros, buscando desta forma aumentar o espectro de resultados e possibilitar uma melhor avaliação sobre os resultados alcançados. Serão manipulados os seguintes parâmetros:

Problema a ser solucionado: serão avaliados desde problemas mais simples, com pouca quantidade de pontos, até problemas complexos, constituídos estes por muitos pontos. Foram selecionados sete problemas, ordenados pela quantidade de pontos, sendo eles (Tabela 6.1):

TABELA 6.1 - PROBLEMAS ESTUDADOS

Ordem	Origem	Modelo	Qtd Pontos	Menor Custo Conhecido
1	TSPLIB	att48	48	33.523,708
2	TSPLIB	eil101	101	642,309
3	TSPLIB	ts225	225	126.643,000
4	Grade	21 x 21	441	441,410
5	TSPLIB	pr1002	1002	259.066,66
6	TSPLIB	fnl4461	4461	182.566,000
7	TSPLIB	d15112	15112	1.573.084,000

Quantidade de Threads: a quantidade de threads influencia diretamente na capacidade de busca, pois quanto maior a quantidade de Threads, maior é a possibilidade da localização de uma ótima solução.

Este parâmetro é de suma importância quando a simulação aplica o operador crossover do algoritmo genético, pois com o aumento da quantidade de threads, aumentamos também a quantidade de cruzamentos, elevando assim a possibilidade de que os cruzamentos conduzam a bons resultados.

Esta característica não ocorre quando a simulação for baseada na forma pura do algoritmo SA, pois o aumento de threads simplesmente resulta numa quantidade maior de execuções, já que as threads não influenciam umas às outras.

O aumento de threads neste caso, serve apenas para confrontar com a simulação equivalente do modelo com crossover.

Foram utilizadas quantidades de threads múltiplas de 2, sendo utilizado principalmente as seguintes quantidades: 16, 32, 64, 128 e 512 threads.

Parâmetros Crossover: Foram realizadas simulações com os três tipos de crossover apresentados (OX, PMX e CX), bem como a forma que os resultados serão aplicados (Melhor pai e melhor filho, Filhos, e melhores indivíduos);

Quantidade de Execuções: Indica a quantidade de repetições das simulações, reforçando desta forma os resultados obtidos.

As simulações baseiam-se portanto na combinação destes parâmetros. Com o intuito de padronizar os testes, foram definidos para os parâmetros crossover nove grupos, sendo eles (Tabela 6.2):

TABELA 6.2 - GRUPOS DE SIMULAÇÕES (ESTRATÉGIAS)

Grupo	Tipo	Elitismo	Crossover aplicado	Pontos de corte	Seleção dos resultados
G1	SA Puro	-	-	-	-
G2	Crossover	0 %	OX	45 % e 50 %	Melhor Pai e Filho
G3	Crossover	0 %	OX	30 % e 70 %	Melhor Pai e Filho
G4	Crossover	0 %	PMX	45 % e 50 %	Melhor Pai e Filho
G5	Crossover	0 %	PMX	30 % e 70 %	Melhor Pai e Filho
G6	Crossover	0 %	CX	-	Melhor Pai e Filho
G7	Crossover	0 %	OX	45 % e 50 %	Filhos
G8	Crossover	0 %	OX	45 % e 50 %	Melhores Indivíduos
G9	Crossover	40 %	OX	45 % e 50 %	Melhor Pai e Filho

Utilizando estes grupos, realizaram-se simulações em função do problema selecionado, da quantidade de threads e da quantidade de execuções. Temos a seguir um quadro que apresenta as simulações realizadas (Tabela 6.3):

TABELA 6.3 - QUANTIDADE DE SIMULAÇÕES REALIZADAS

Modelo	Grupo Aplicado	Threads	Execuções
att48	Todos	16, 32, 64 e 128	5
	G1 e G2	512	1
eil101	Todos	16, 32, 64 e 128	5
	G1 e G2	512	1
ts225	G1 e G2	16, 32, 64 e 128	5
	G1 e G2	512	1
21x21	G1 e G2	16, 32, 64 e 128	5
	G1 e G2	512	2
pr1002	G1 e G2	16, 32, 64 e 128	5
	G1 e G2	512	1
fnl4461	G1 e G2	32 e 64	2
d15112	G1 e G2	16	1

Somente nos dois primeiros modelos foram aplicados todos os grupos de crossover definidos, pois com as simulações realizadas, foi identificado um padrão de comportamento que se repete para os demais grupos, conforme será apresentado a seguir. Outro principal fator que limitou a extensão de todos os grupos a todos os modelos foi o enorme tempo requerido para a simulação dos modelos com uma grande quantidade de pontos e threads.

Com relação a quantidade de execuções, forma realizados dois tipos de análise. A primeira análise foi em função da média dos resultados alcançados, e a segunda em função do melhor resultado alcançado em todas as execuções.

6.3.1 Problema att48

Este problema apresenta um grau de dificuldade relativamente simples, sendo que os resultados alcançados foram satisfatórios, conforme vemos nos gráficos abaixo.

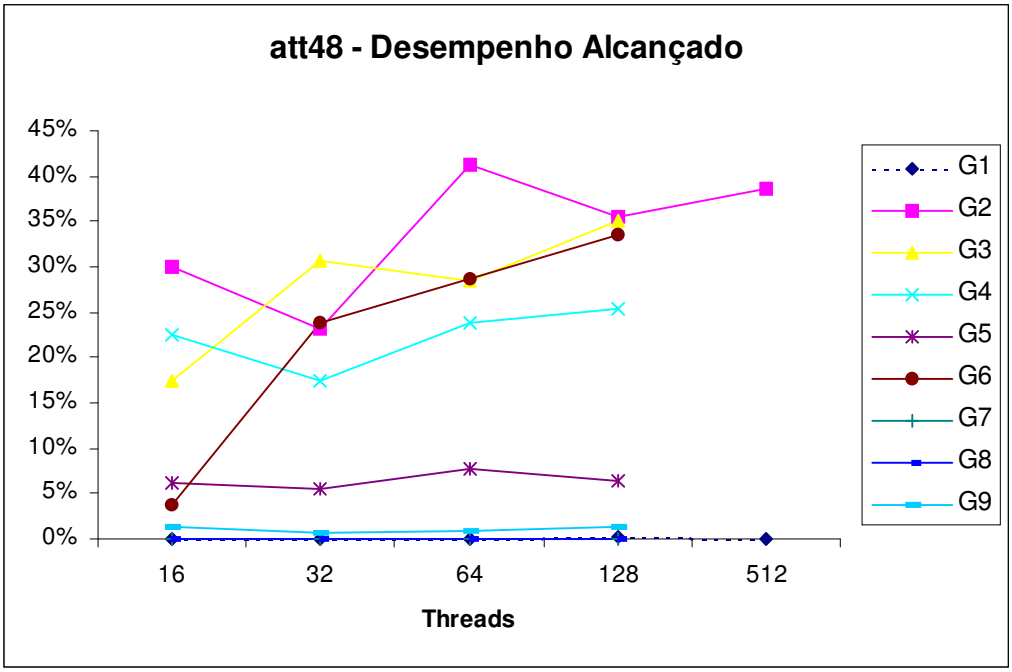

Figura 6.2 - Problema att48 – Desempenho dos grupos por Threads

As figuras 6.2 e 6.3 apresentam o desempenho do modelo e a qualidade da solução para todos os grupos testados em função da média dos resultados. Verificamos que os grupos G2 a G6 e G9 apresentam tanto um bom desempenho como uma ótima qualidade de solução,

independente da quantidade de threads selecionada, com destaque para o grupo G2. Estes grupos correspondem todos a alternativas de crossover cuja regra de aplicação dos resultados consiste na escolha do melhor pai e do melhor filho. O modelo puro do algoritmo SA não obteve sucesso em nenhuma das tentativas da identificação da melhor solução, obtendo assim um resultado ruim no quesito desempenho, porém a qualidade de sua solução evoluiu à medida que aumentamos a quantidade de threads.

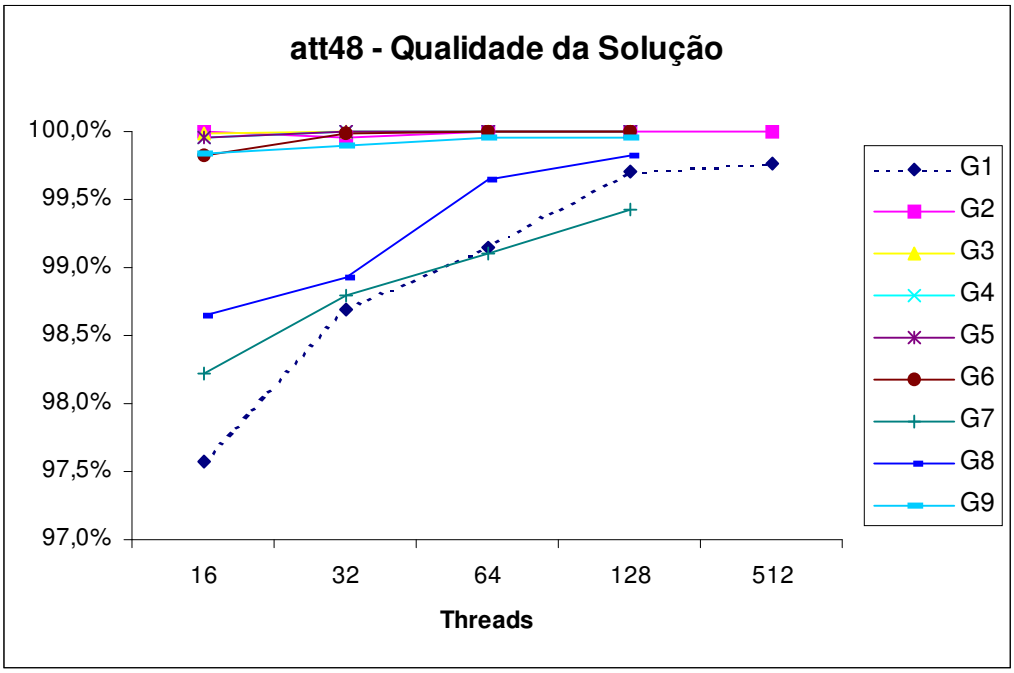

Figura 6.3 - Problema att48 – Qualidade da solução dos grupos por Threads

As estratégias de seleção dos resultados do crossover baseado na escolha dos filhos gerados ou dos melhores indivíduos, não obtiveram resultados satisfatórios comparados a estratégia de melhor pai e filho, sendo que estas estratégias tiveram um comportamento semelhante ao modelo do algoritmo SA puro.

As figuras 6.4 e 6.5 são equivalentes as figuras 6.2 e 6.3 respectivamente, porém foram agrupados as estratégias a fim de facilitar a leitura dos resultados. Temos então os novos grupos formados:

SA: Grupo G1, equivalente ao algoritmo SA puro;

OX: Grupos G2 e G3, corresponde a estratégia OX;

OXe: Grupo G9, corresponde a estratégia OX com elitismo de 40% dos indivíduos;

OX+: Grupo G8, corresponde a estratégia OX com seleção dos melhores indivíduos;

CX: Grupo G6, estratégia CX de crossover;

PMX: Grupos G4 e G5, estratégia PMX de crossover.

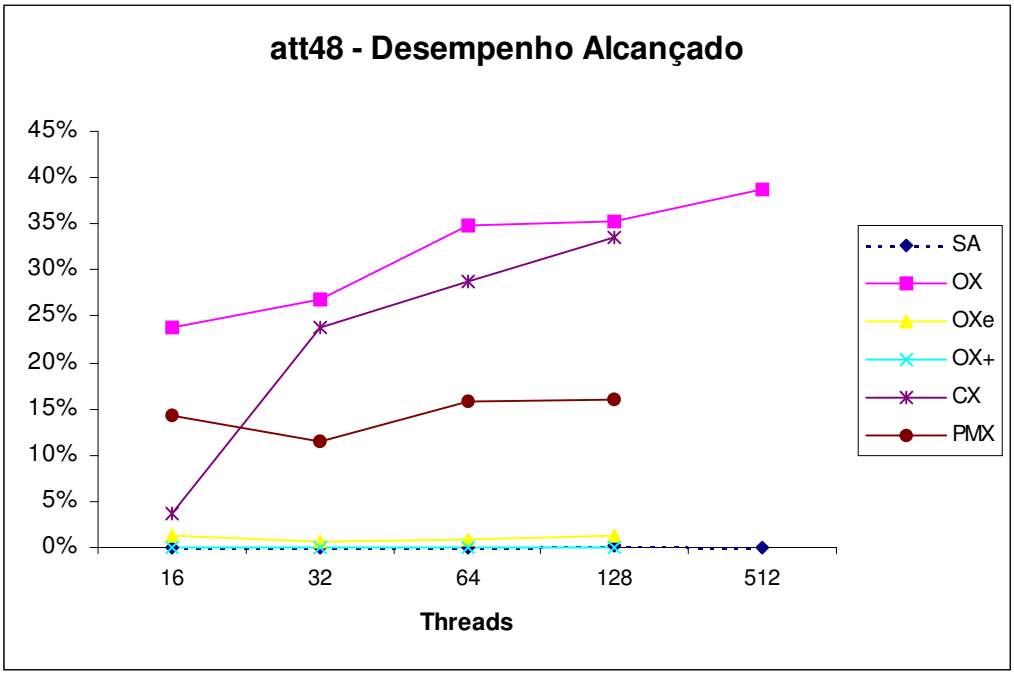

Figura 6.4 - Problema att48 – Desempenho agrupado por Threads

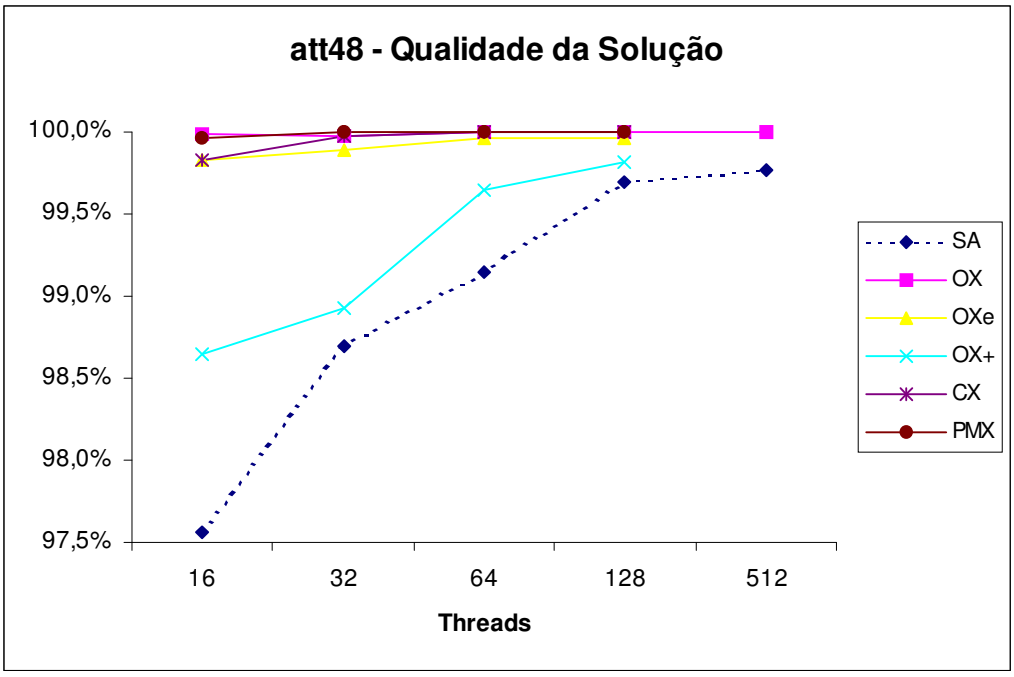

Figura 6.5 - Problema att48 – Qualidade da solução agrupada por Threads

A figura 6.6 apresenta a análise da qualidade da solução para o melhor solução encontrada de todo o conjunto de execuções realizadas, em função das estratégias SA puro e SA crossover OX (Grupo G2). Verificamos neste gráfico que para a estratégia SA puro, a qualidade da solução depende da quantidade de threads somente no sentido de termos uma maior probabilidade de localização da solução devido a uma quantidade maior de simulações, pois vemos que na simulação com 512 threads não obtivemos a mesma qualidade obtida na simulação com 128 threads.

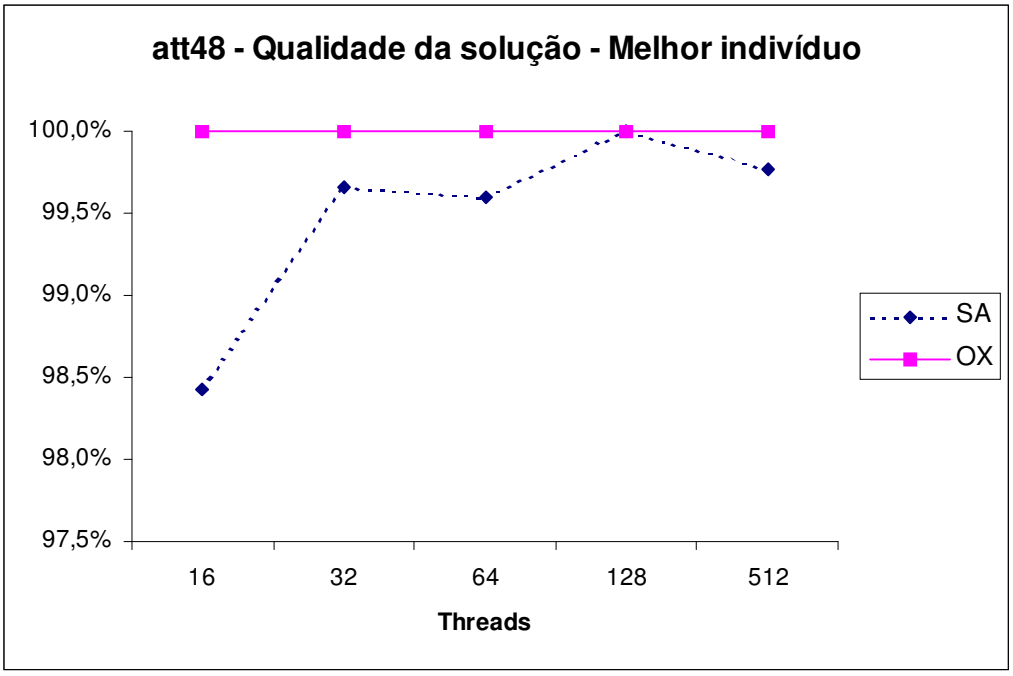

Figura 6.6 - Problema att48 – Qualidade da solução do melhor indivíduo por Threads

Temos nas figuras 6.7 a 6.11 a influência que a operação crossover teve sobre a identificação da melhor solução encontrada, em função da quantidade de threads aplicadas (16, 32, 64, 128 e 512) e dos grupos simulados (G2 a G9).

Verificando os resultados, vemos que o comportamento é semelhante para cada um dos grupos independente da quantidade de threads, bem como vemos que a operação de crossover influencia desde o início da simulação até o final da mesma, contribuindo desta forma em todo o processo de convergência na resolução do problema.

Outra constatação, ou melhor uma confirmação, trata que a influência do crossover é proporcional à estratégia aplicada e a quantidade de elitismo selecionado. Temos para a estratégia G7 (escolha dos filhos gerados) que a operação de crossover influência em 100% do resultado em todas os trechos da simulação. Isto é de se esperar pois os filhos são resultados do crossover e a seleção destes indica que a solução é totalmente resultado da operação de crossover.

A estratégia G9 (elitismo 40%) tem um comportamento semelhante as demais estratégias de seleção do melhor pai e filho (G2, G3, G4, G5 e G6), porém com um grau menor de crossover devido ao fato de selecionarmos uma quantidade de indivíduos que não sofrerão cruzamentos.

A estratégia G8 (melhores indivíduos) proposta por HIROYASU et al (2000), sofreu muito pouca influência do crossover na localização da melhor solução, pois ao final do cruzamento, quase sempre são selecionados os pais, pois os filhos geralmente apresentam um custo maior devido a terem sofrido uma alteração que visa não a redução do custo, e sim a ampliação da capacidade de busca através de nova reconfiguração do espaço de busca, procurando escapar de armadilhas como mínimos locais.

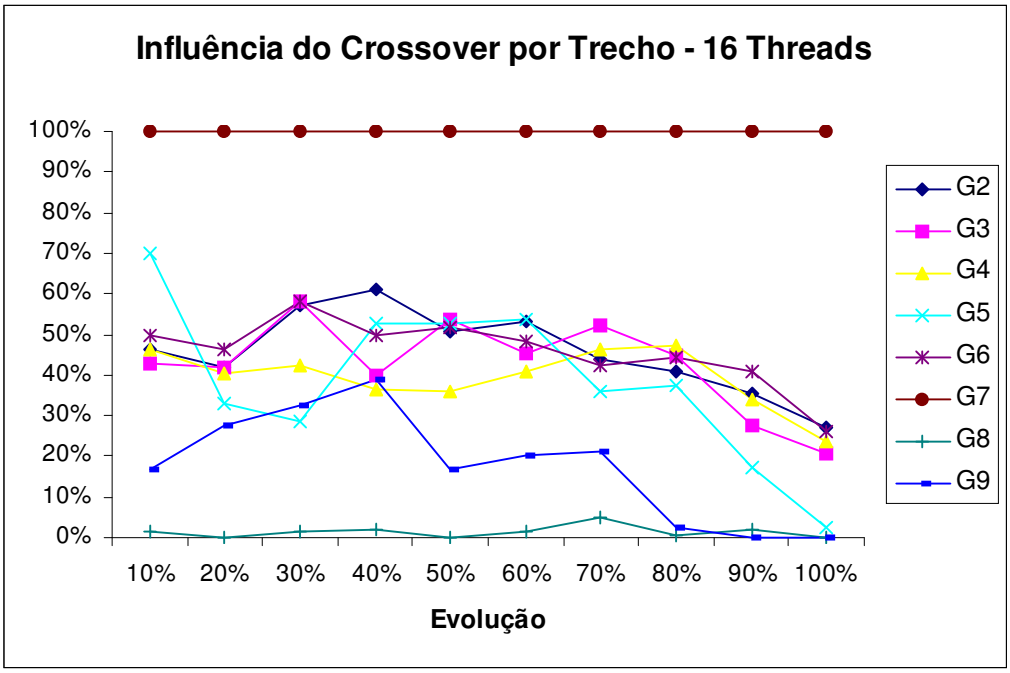

Figura 6.7 - Problema att48 – Influência do Crossover – 16 Threads

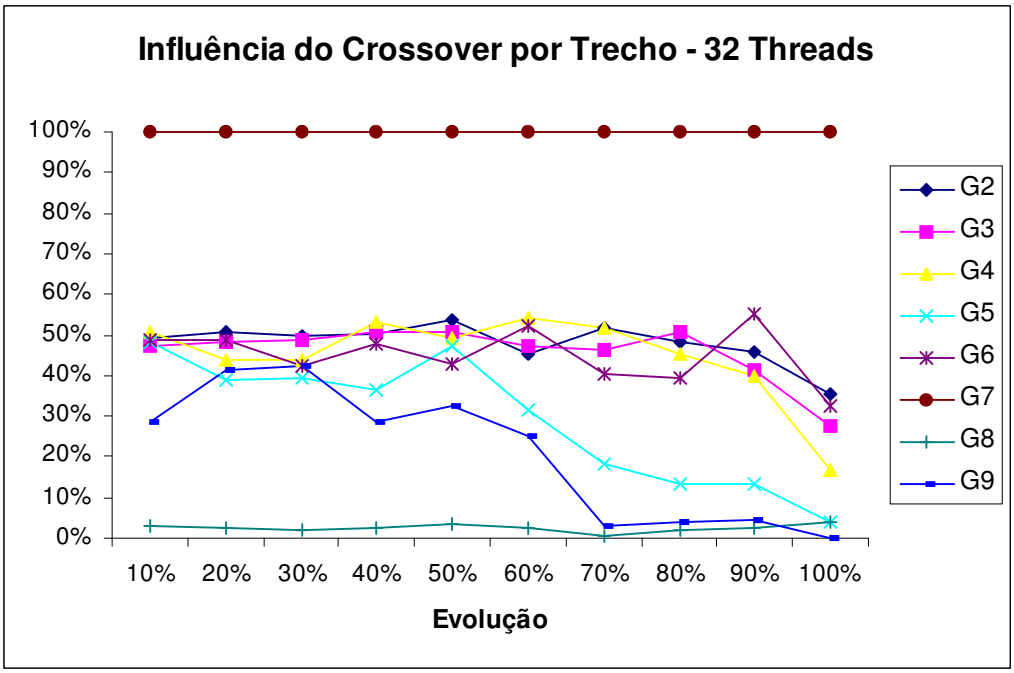

Figura 6.8 - Problema att48 – Influência do Crossover – 32 Threads

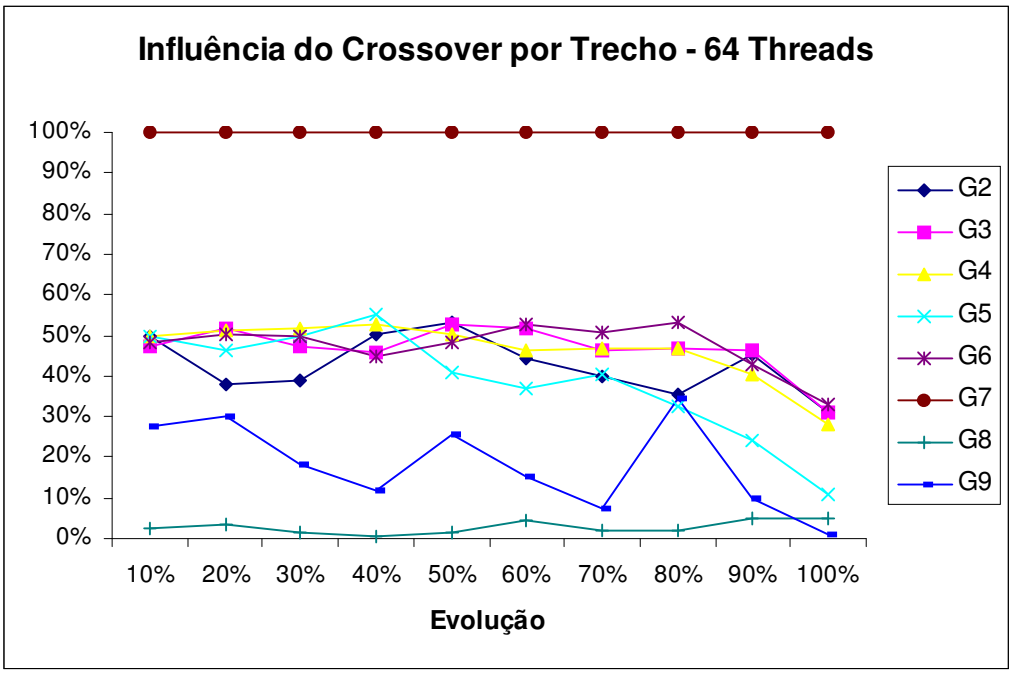

Figura 6.9 - Problema att48 – Influência do Crossover – 64 Threads

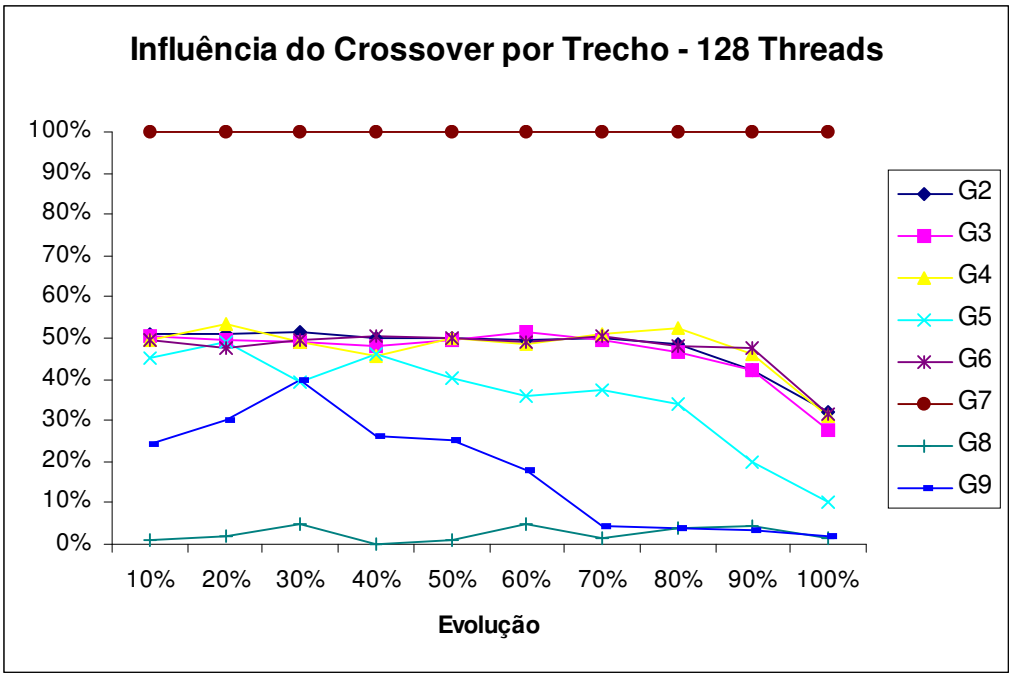

Figura 6.10 - Problema att48 – Influência do Crossover – 128 Threads

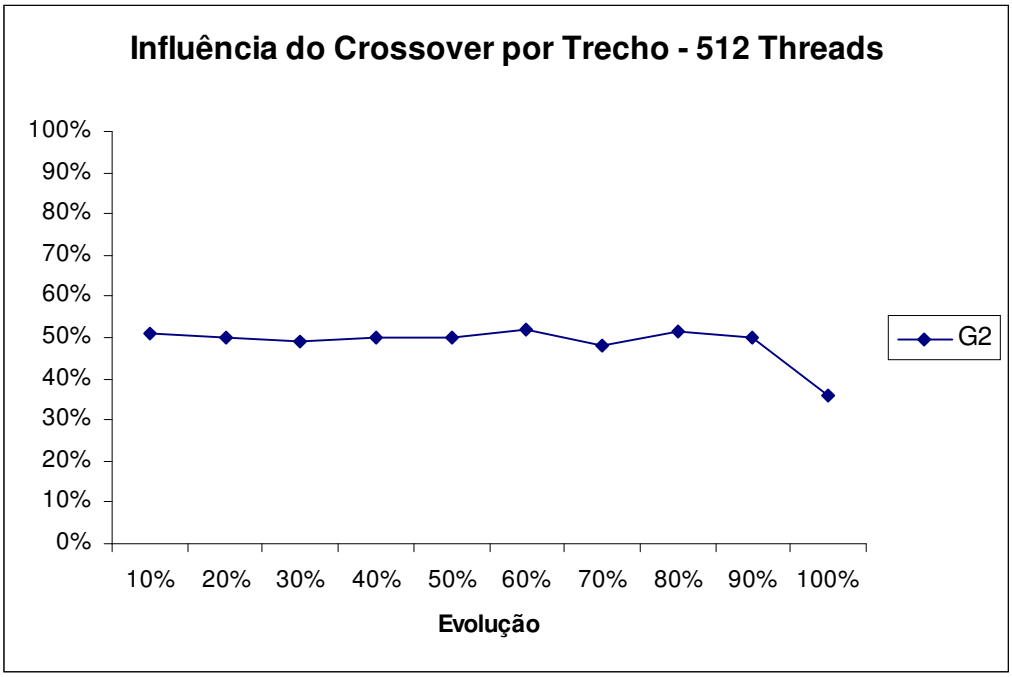

Figura 6.11 - Problema att48 – Influência do Crossover – 512 Threads

A figura 6.12 apresenta um resumo da influência do crossover utilizando os grupos compostos descrito anteriormente, bem como uma curva de média que visa salientar a influência praticamente constante ao longo de todo o processo.

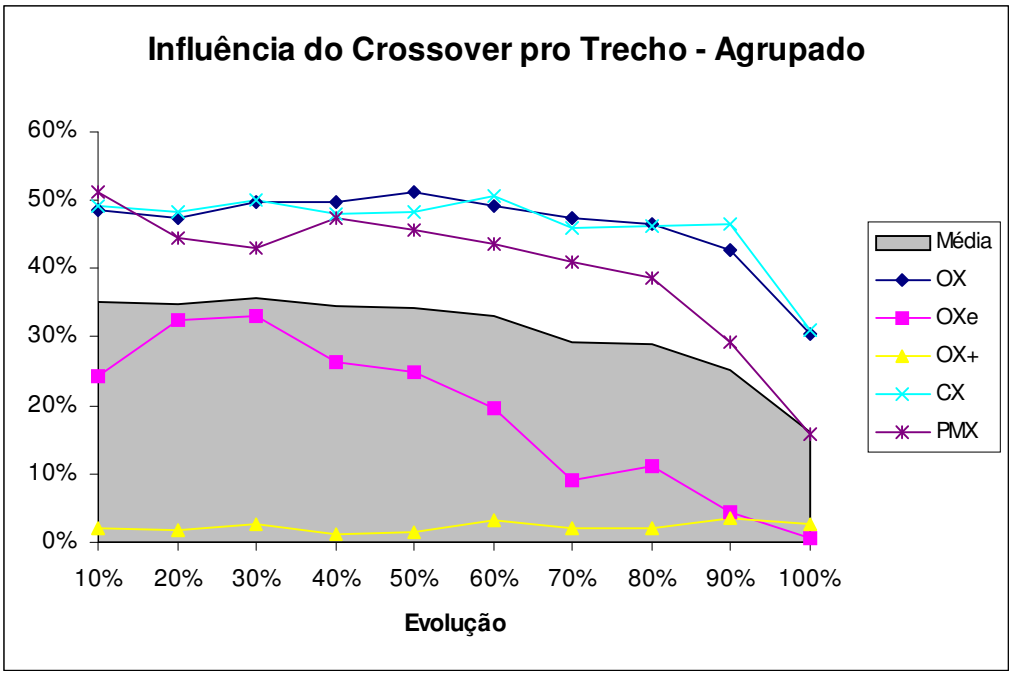

Figura 6.12 - Problema att48 – Influência do Crossover – Média

Temos na figura 6.13 a média final da influência do crossover por grupo e quantidade de threads. Nele confirma-se a afirmativa descrita anteriormente de que a quantidade de threads não influencia na participação do crossover na localização da melhor solução encontrada, sendo que somente a estratégia aplicada é que apresenta esta característica.

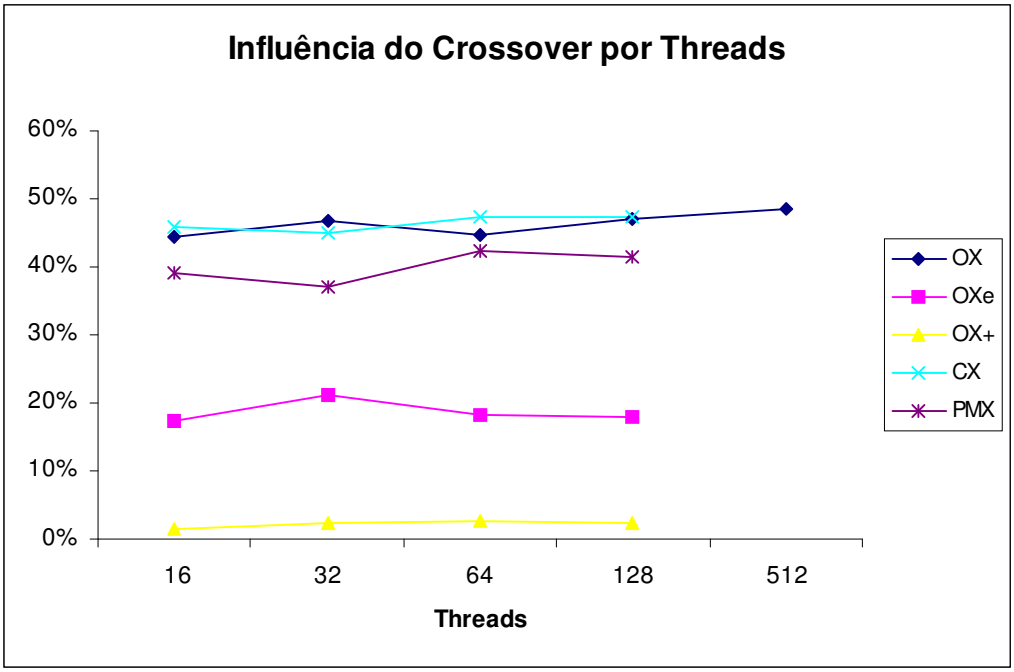

Figura 6.13 - Problema att48 – Influência do Crossover – Threads

A figura 6.14 apresenta um exemplo da convergência da solução ao longo dos passos do algoritmo SA crossover. Observa-se que no decorrer da execução ocorrem perturbações oriundas dos cruzamentos que elevam temporariamente o custo da solução, resultando ao final em uma solução de custo ótimo, equivalente este ao custo mínimo identificado na literatura.

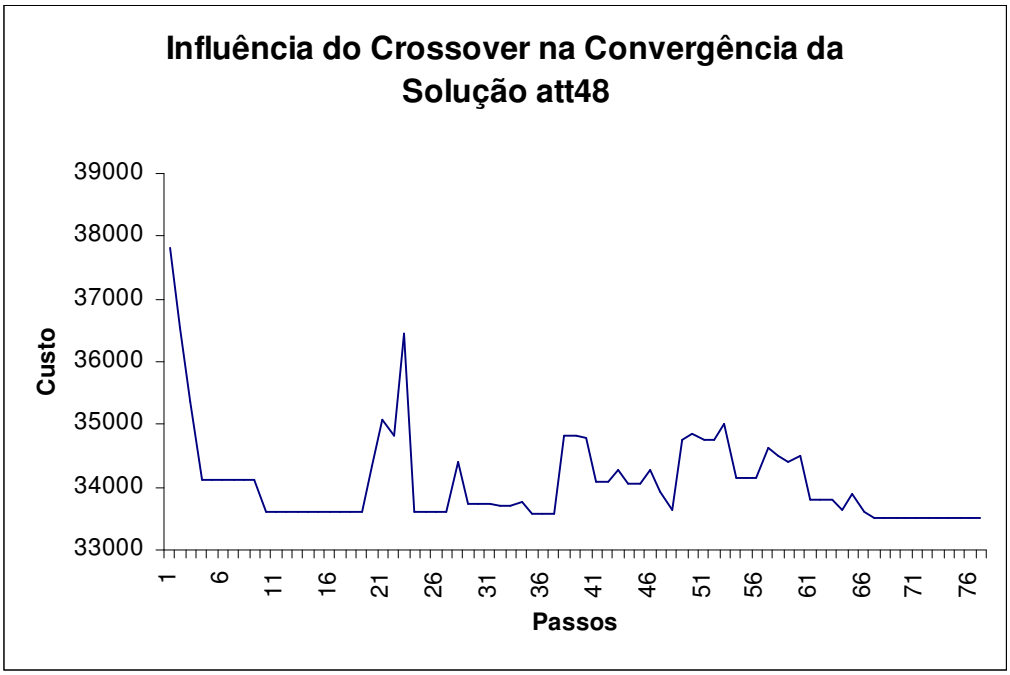

Figura 6.14 - Problema att48 – Influência do crossover na convergência da solução

Finalizando para este problema, temos na figura 6.15 os estados inicial e final do problema quando otimizados pela estratégia crossover OX. A solução apresentada corresponde à melhor solução para o problema em questão.

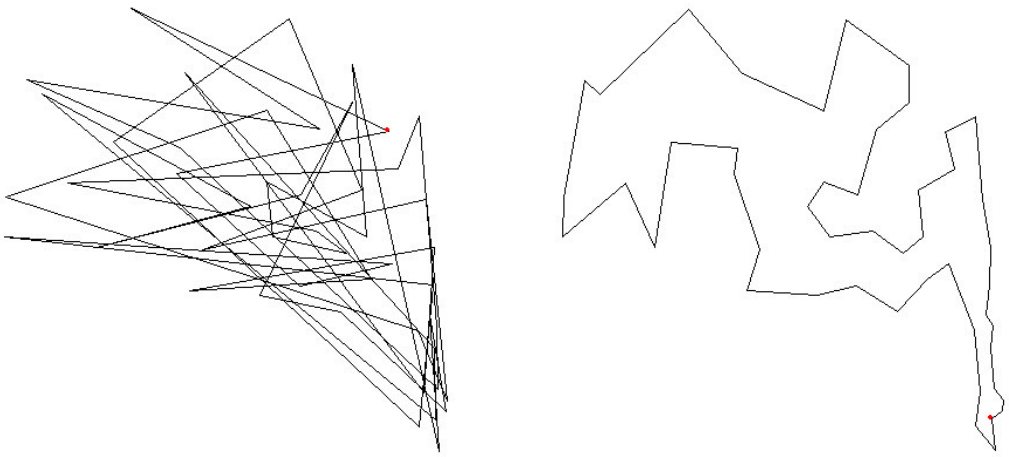

Figura 6.15 - Problema att48 – Estado inicial e final do problema após otimização pela estratégia crossover OX

6.3.2 Problema eil101

Para este problema também foi realizado um estudo completo com todos os grupos simulações definidos, buscando desta forma verificar um padrão de comportamento que poderá ser estendido aos demais problemas propostos.

Novamente tivemos bons resultados, identificando soluções melhores que a apresentada pela biblioteca TSPLIB. Observa-se porém que para este tamanho de problema, somente a partir de 64 threads alcançamos a solução ótima identificada na média de soluções, conforme vemos as figuras 6.16 e 6.17 que apresentam o desempenho e a qualidade média das soluções encontradas. Tem destaque novamente o grupo G2 (crossover OX) pelos resultados obtidos acima dos demais.

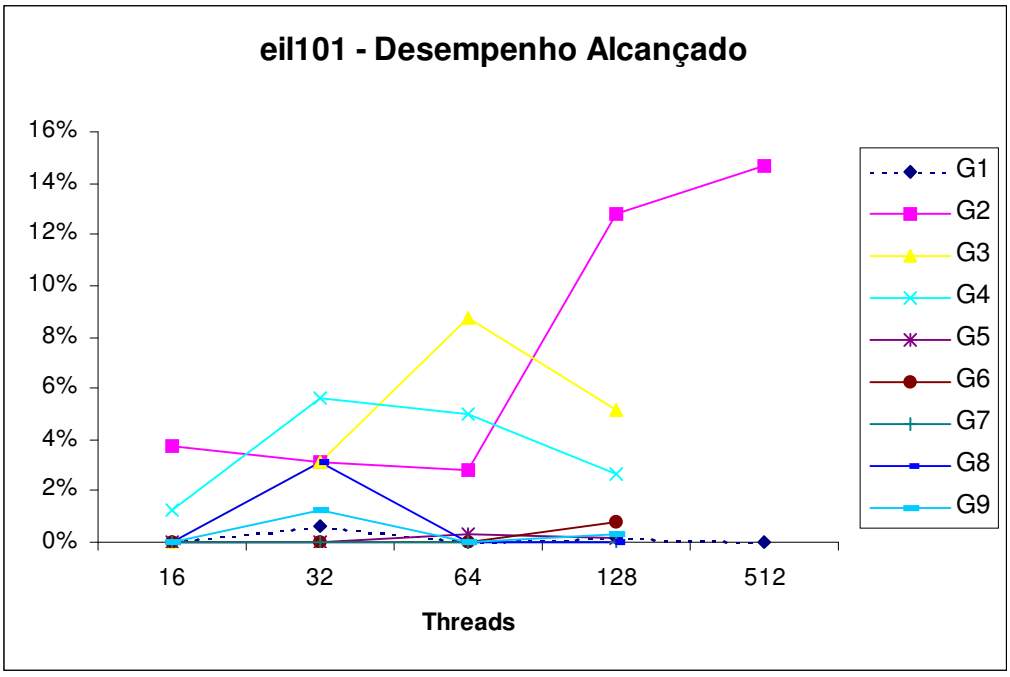

Figura 6.16 - Problema eil101 – Desempenho dos grupos por Threads

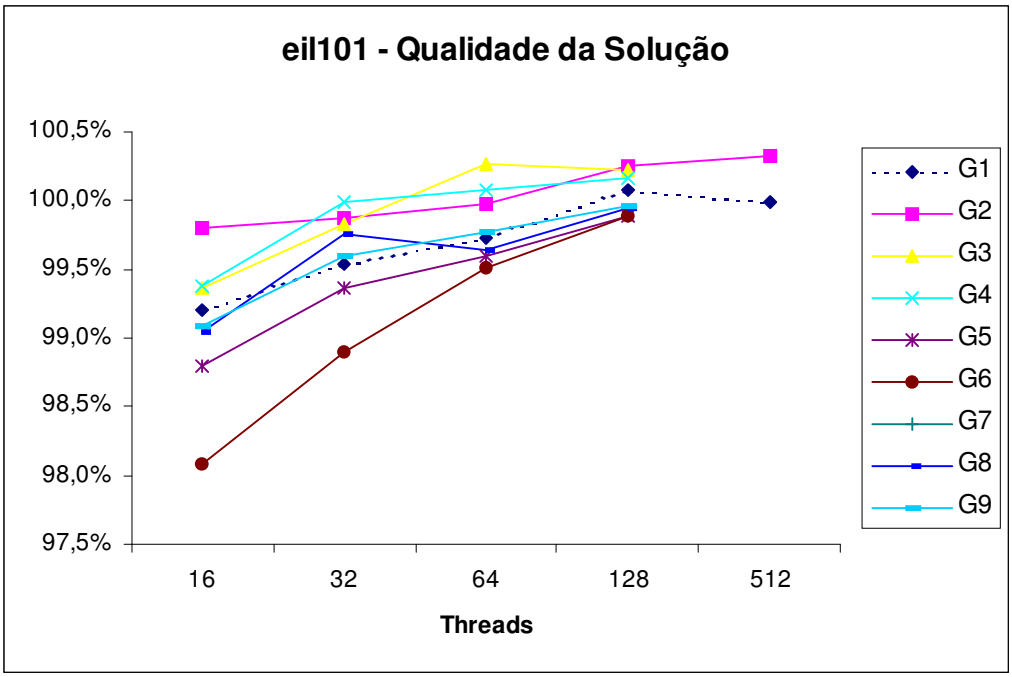

Figura 6.17 - Problema eil101 – Qualidade da solução dos grupos por Threads

As figuras 6.18 e 6.19 apresentam os mesmos resultados agrupados da mesma forma apresentada para o problema att48, conforme tipo de crossover aplicado, permitindo uma visão mais clara sobro o comportamento das estratégias aplicadas.

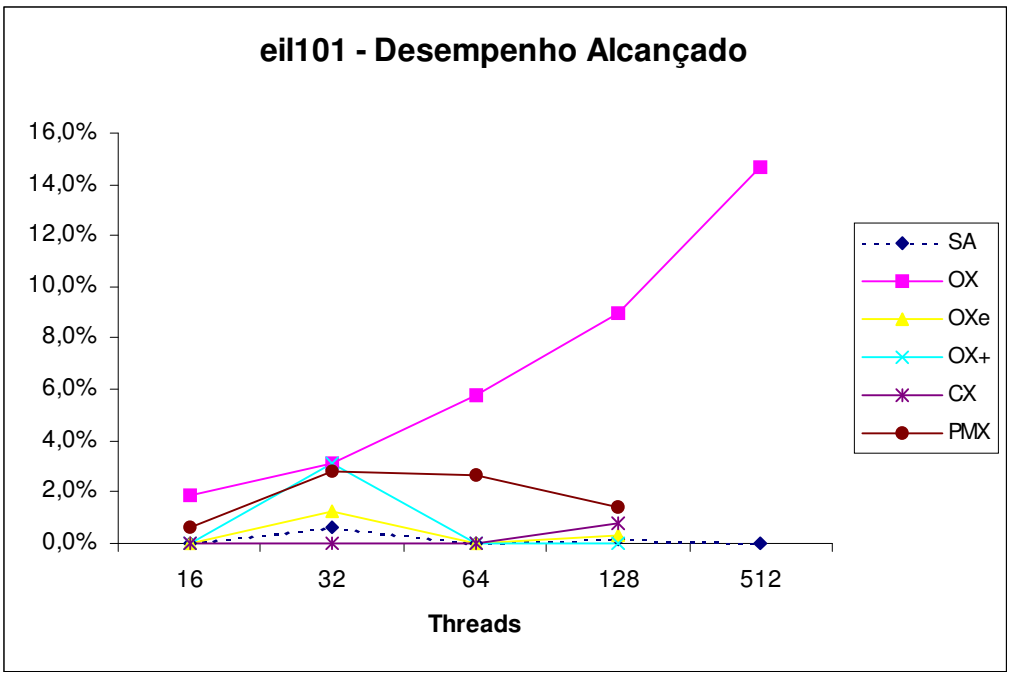

Figura 6.18 - Problema eil101 – Desempenho agrupado por Threads

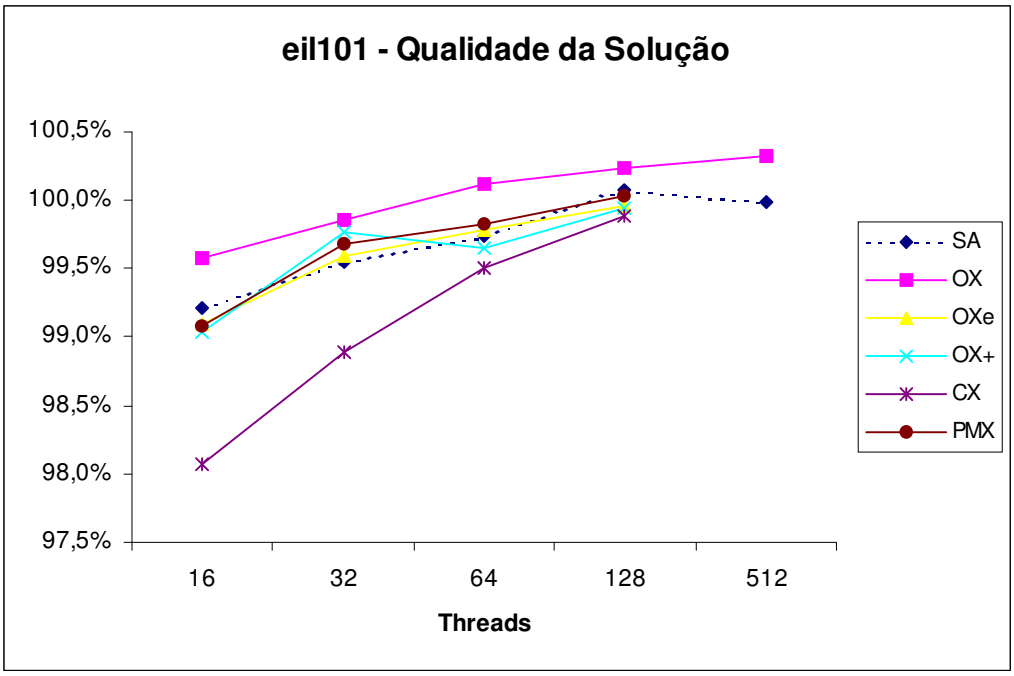

Figura 6.19 - Problema eil101 – Qualidade da solução agrupada por Threads

A figura 6.20 apresenta o gráfico de qualidade da solução para a melhor solução encontrada.

Vemos para a estratégia com crossover que dentre as execuções realizadas, sempre encontramos um ótimo resultado, que persiste à medida que aumentamos a quantidade de threads.

Já para o modelo SA puro, novamente a quantidade de threads não interfere nos resultados alcançados, pois as threads simplesmente representam um volume de alternativas de solução, não uma quantidade maior de interação entre as mesmas.

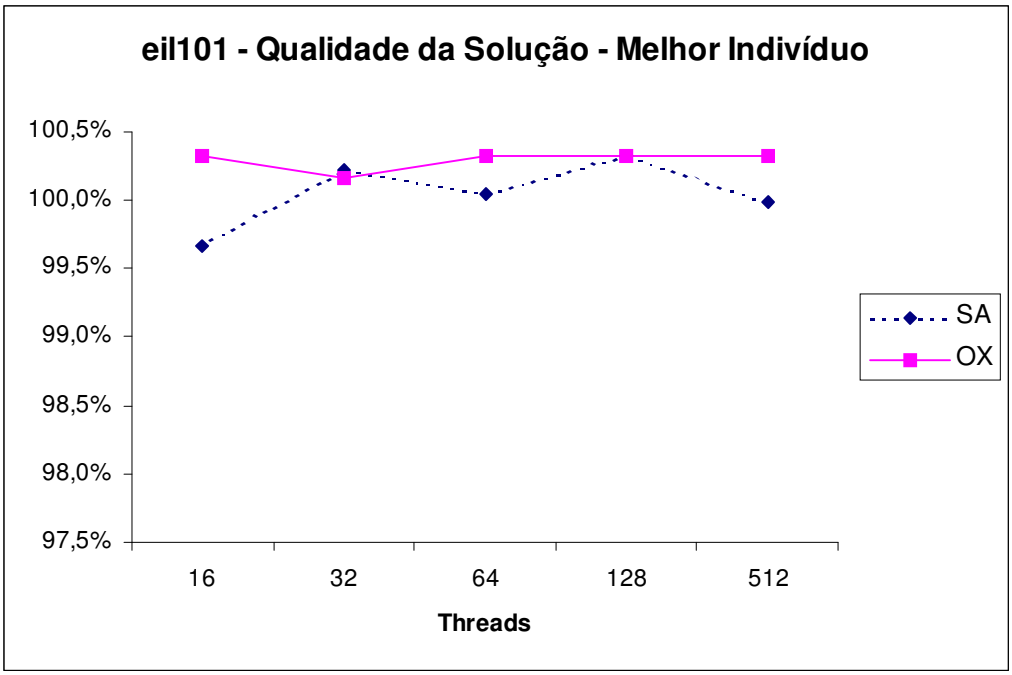

Figura 6.20 - Problema eil101 – Qualidade da solução do melhor indivíduo por Threads

Temos nas figuras 6.21 a 6.25 a influência da operação de crossover sobre a melhor solução encontrada. Vemos que o comportamento se repete se comparado ao problema att48, e este não depende da quantidade de threads aplicadas.

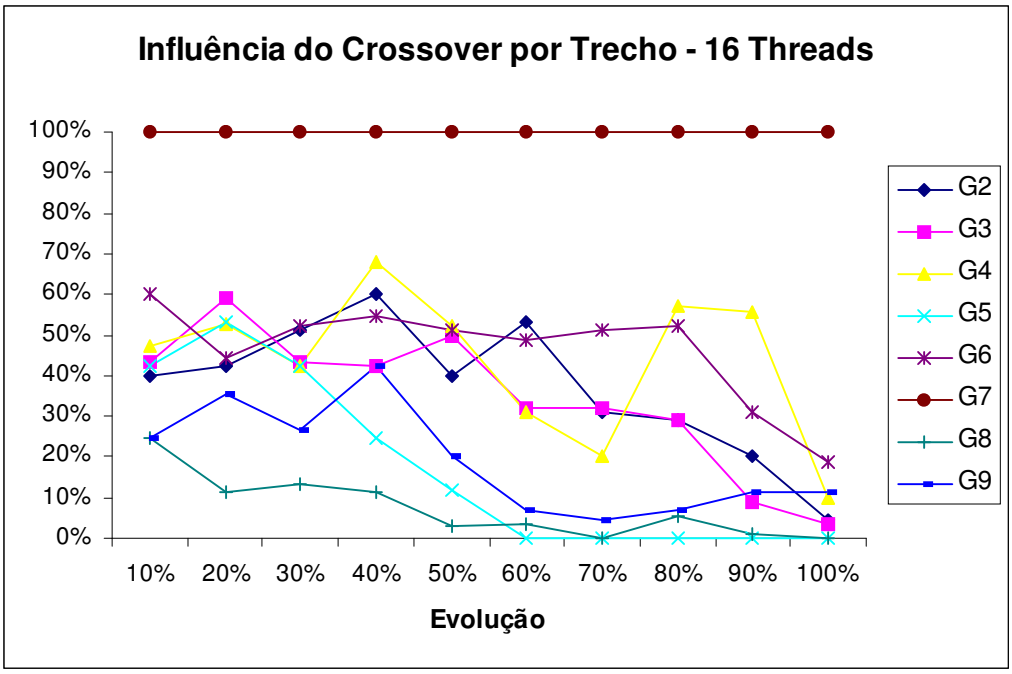

Figura 6.21 - Problema eil101 – Influência do Crossover – 16 Threads

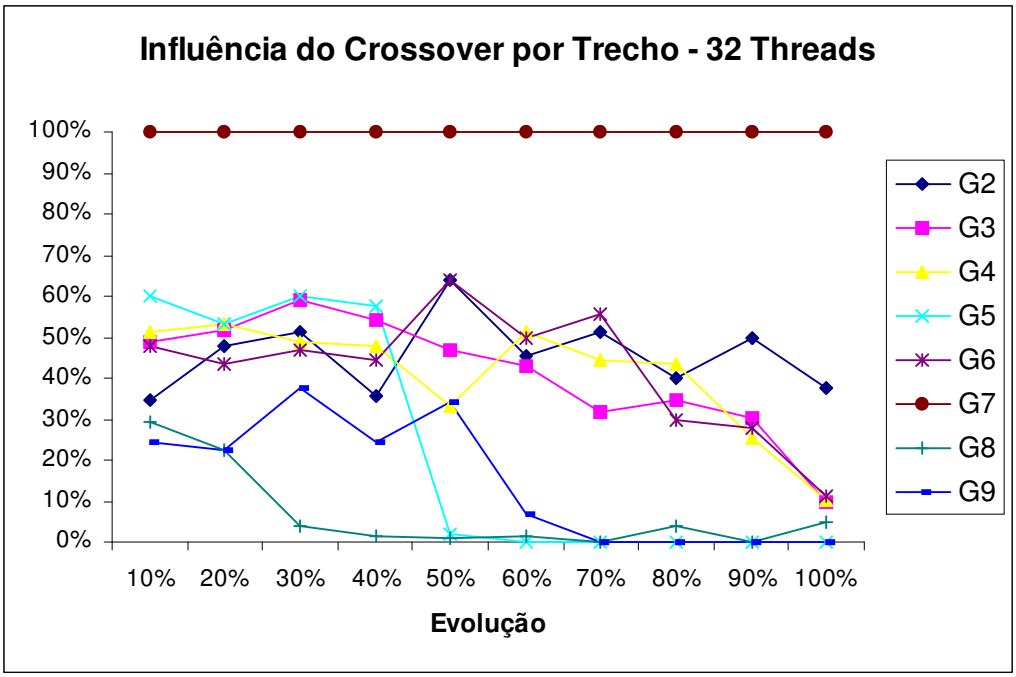

Figura 6.22 - Problema eil101 – Influência do Crossover – 32 Threads

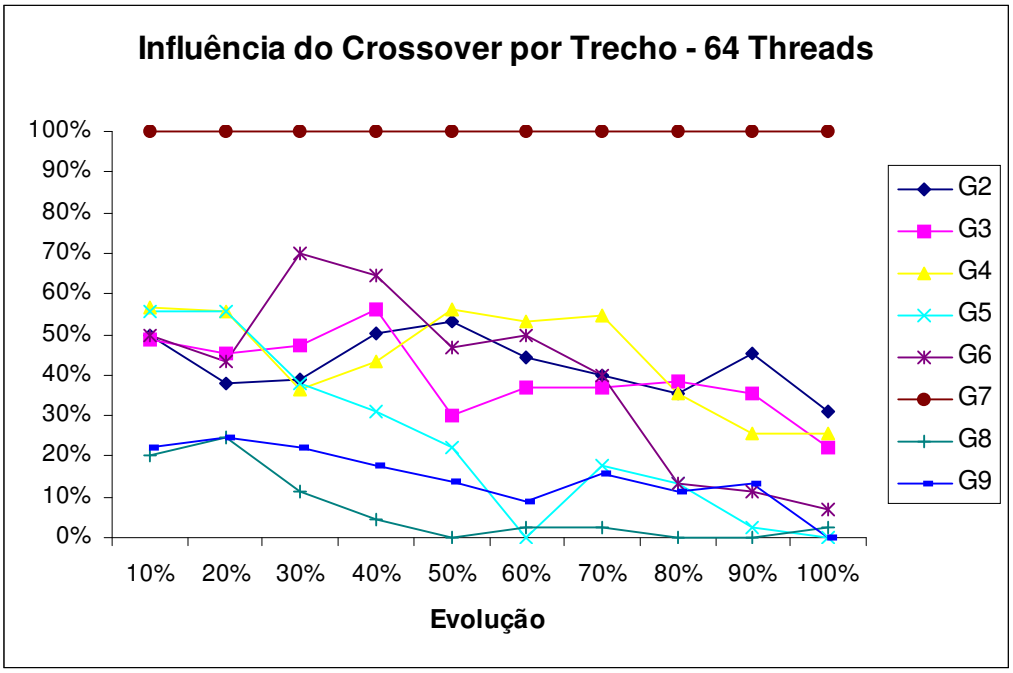

Figura 6.23 - Problema eil101 – Influência do Crossover – 64 Threads

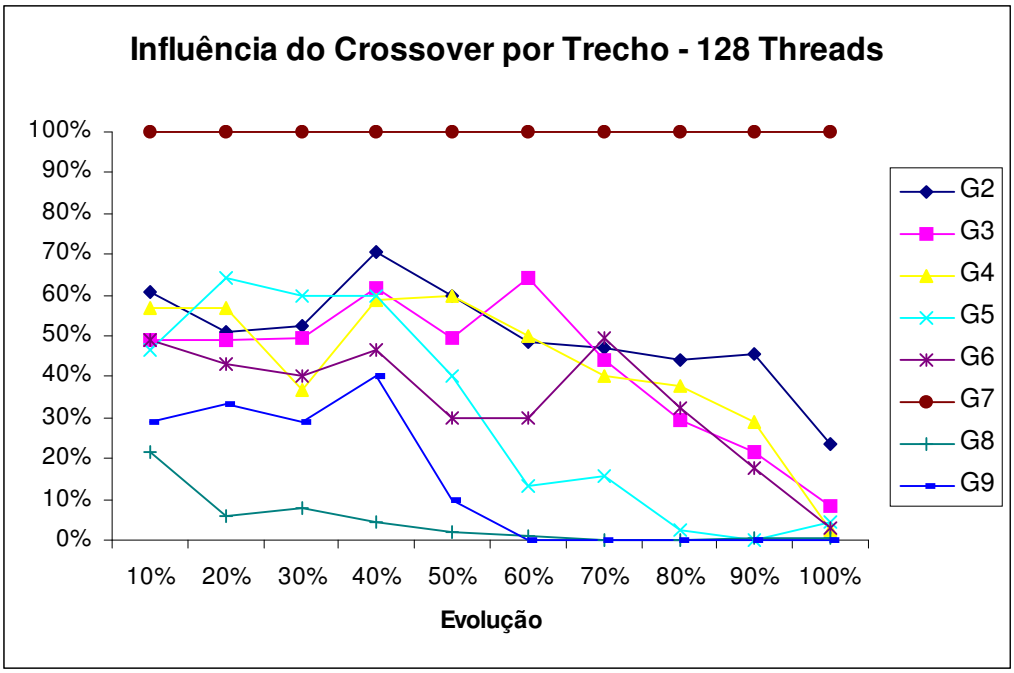

Figura 6.24 - Problema eil101 – Influência do Crossover – 128 Threads

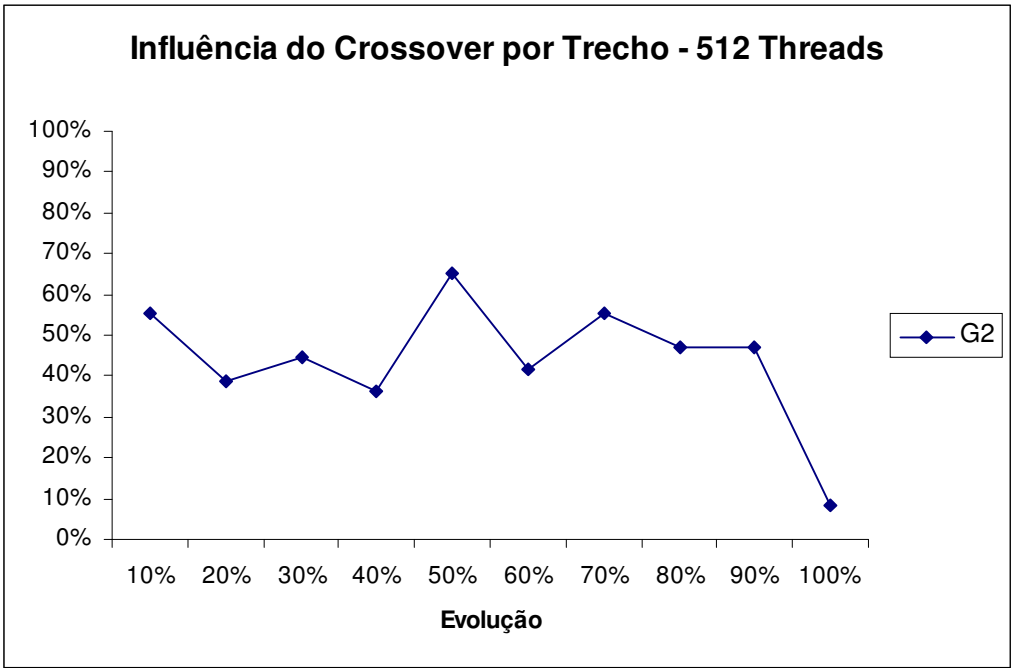

Figura 6.25 - Problema eil101 – Influência do Crossover – 512 Threads

A figura 6.26 mostra um resumo da influência do crossover utilizando os grupos compostos juntamente com uma curva de média. Já na figura 6.27 temos a média final da influência do crossover por grupo e quantidade de threads, como visto no problema anterior,

confirmando novamente que a influência do crossover não depende da quantidade de threads, e sim da estratégia aplicada.

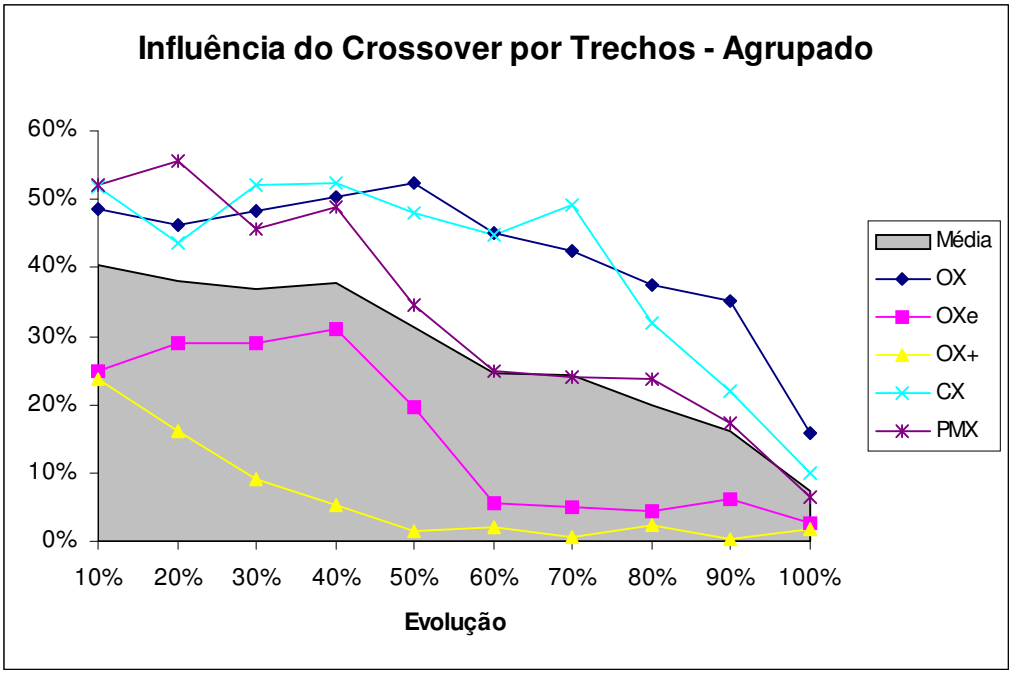

Figura 6.26 - Problema eil101 – Influência do Crossover – Média

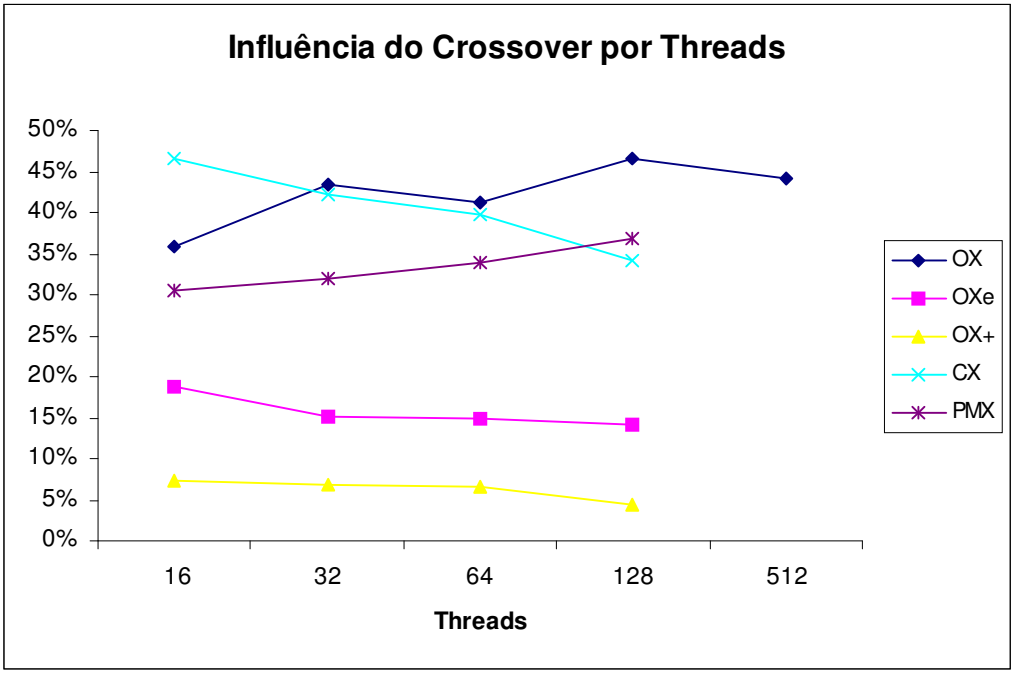

Figura 6.27 - Problema eil101 – Influência do Crossover – Threads

Finalizando a análise do problema eil101, temos na figura 6.28 um exemplo da convergência da solução ao longo dos passos do algoritmo SA crossover, podendo ser vistas as variações de custo à medida que o modelo resfria, ocasionadas estas pelas operações de crossover realizadas. E na figura 6.29 temos os estados inicial e final do problema otimizado pela estratégia crossover OX.

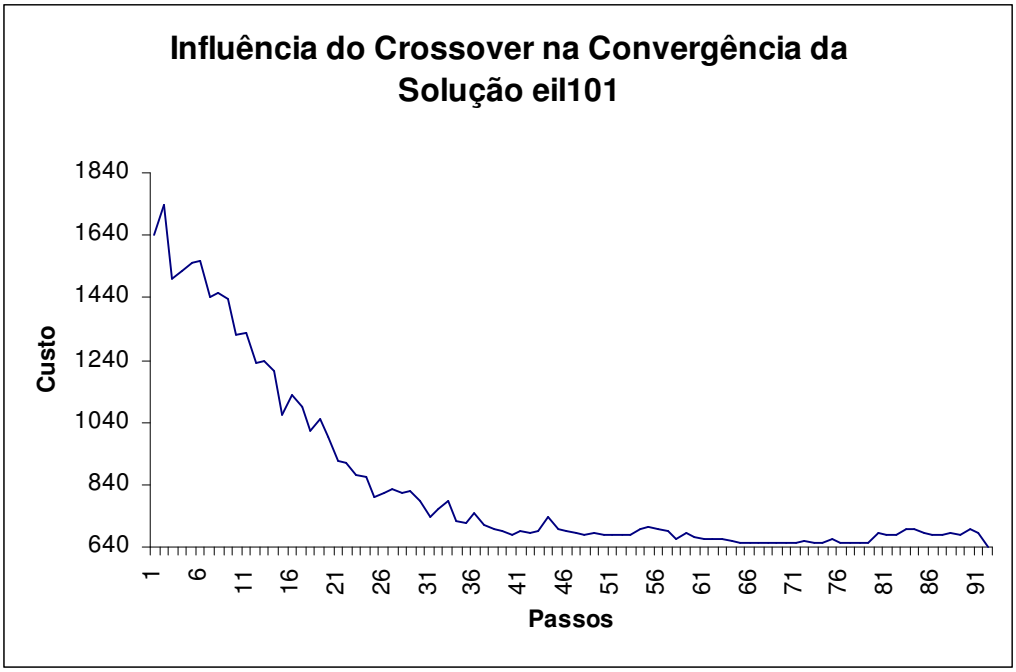

Figura 6.28 - Problema eil101 – Influência do crossover na convergência da solução

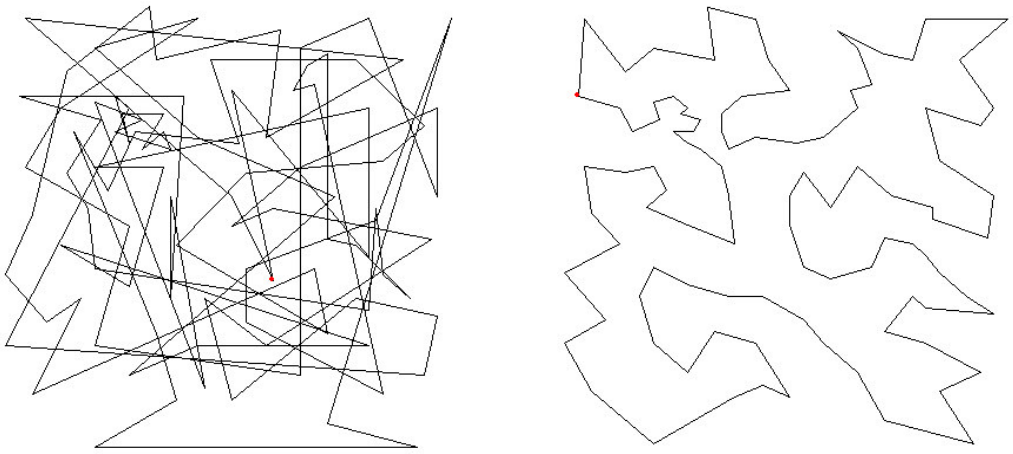

Figura 6.29 - Problema eil101 – Estado inicial e final do problema após otimização pela estratégia crossover OX

6.3.3 Problema ts225

As análises realizadas a partir deste problema, foram limitadas somente as estratégias SA-puro (G1) e SA-crossover-OX (G2), pois como pode ser verificado nos dois problemas anteriormente apresentados, todas as outras estratégias apresentam um comportamento equivalente nas duas situações estudadas, tanto em relação a influência do crossover ao longo do processo como quanto aos resultados obtidos. Praticamente, a única diferença constatada é a necessidade de uma quantidade maior de threads para a obtenção de bons resultados, o que é justificado pelo aumento de pontos do problema.

Para este problema, foram necessários 512 threads para a localização da solução ótima, porém desde a quantidade de 16 threads, obtivemos uma qualidade de solução bem superior a estratégia SA puro, sendo esta bem próxima a solução ideal, conforme vemos nas figuras 6.30, 6.31 e 6.32. Observa-se também a tendência constatada nos problemas anteriores, de que a qualidade da solução melhora a medida em que aumentamos a quantidade de threads.

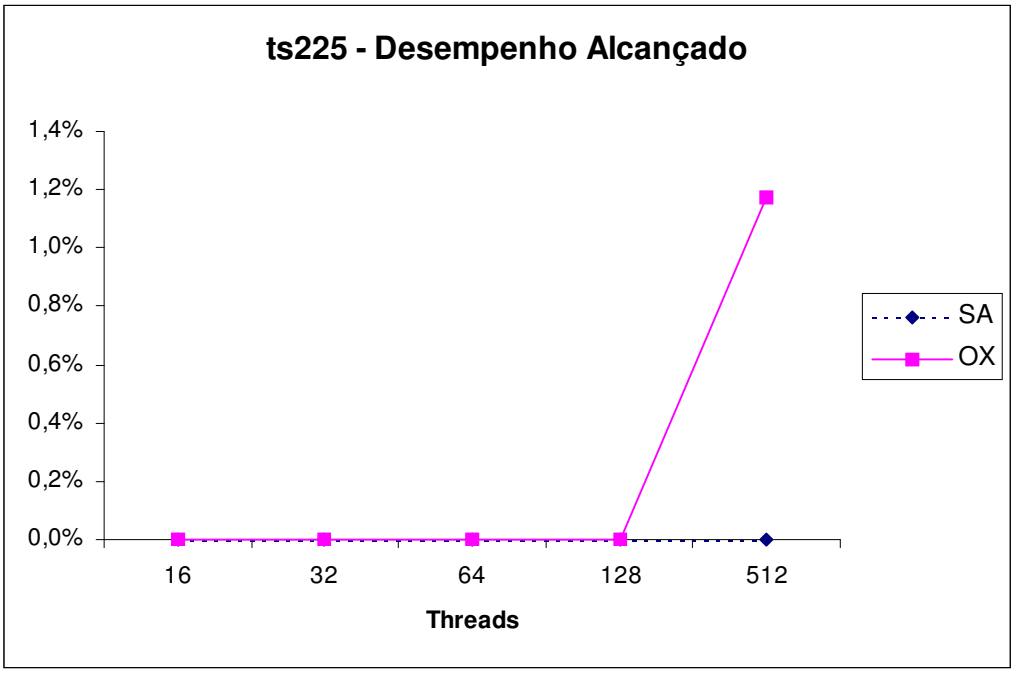

Figura 6.30 - Problema ts225 – Desempenho por Threads

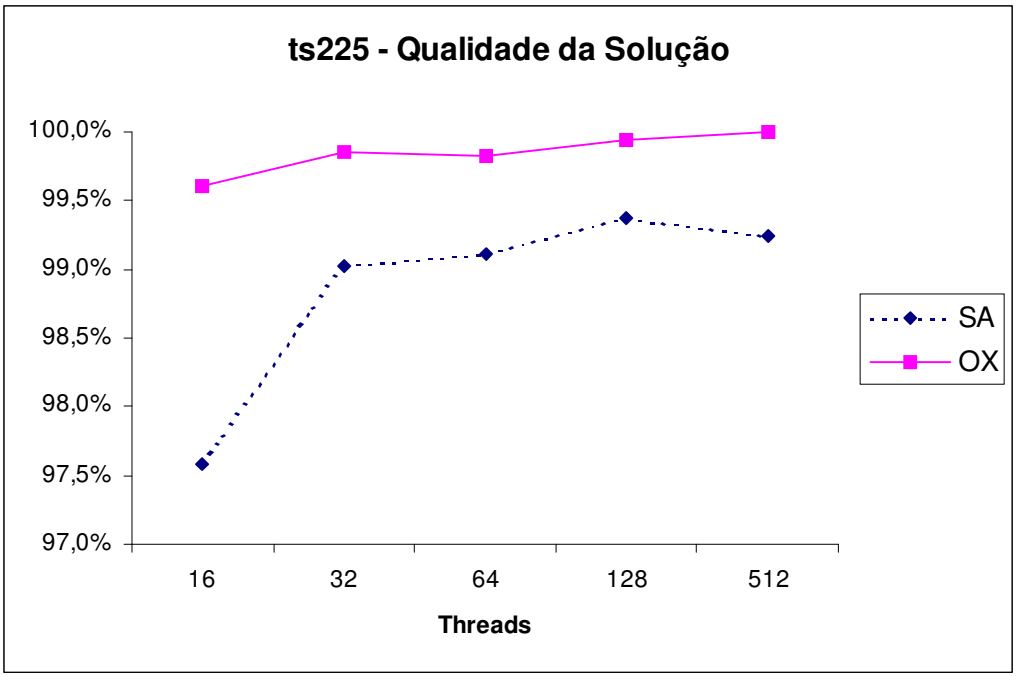

Figura 6.31 - Problema ts225 – Qualidade da solução por Threads

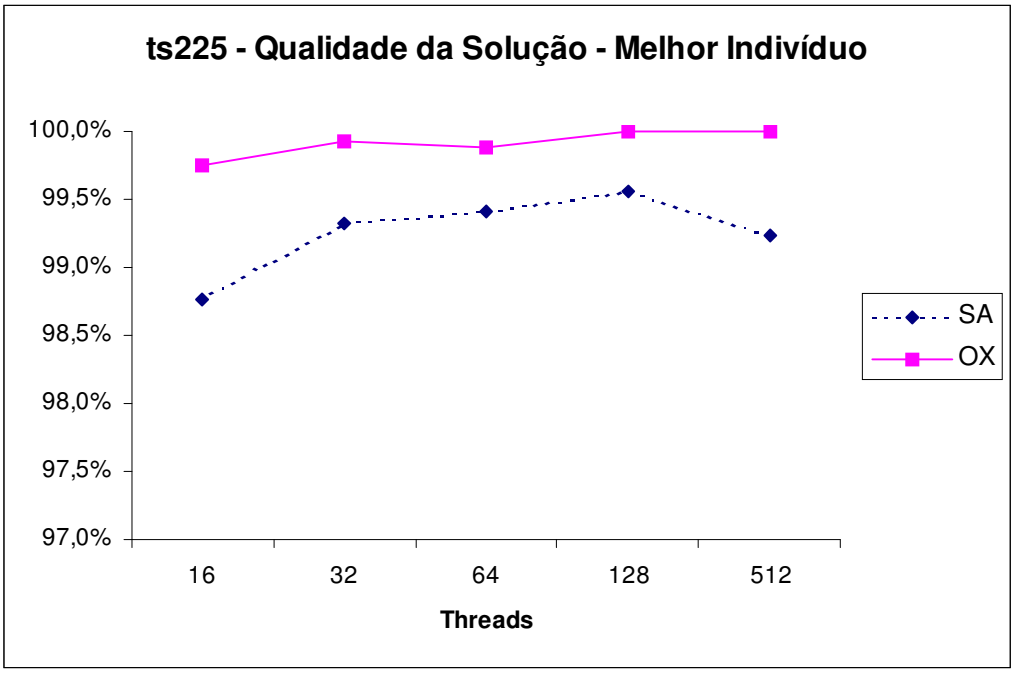

Figura 6.32 - Problema ts225 – Qualidade da solução do melhor indivíduo por Threads

A figura 6.33 apresenta a influência do crossover durante a evolução da simulação para todas as quantidades de threads aplicadas. Vemos novamente um padrão de comportamento, tendo um valor de influência média na faixa de 40% a 45% durante todo o

processo e somente ao final a influência cai por volta de 20%. Na figura 6.34 temos a influência média por quantidade de threads, sendo que esta permanece praticamente inalterada à medida que aumentamos a quantidade de threads.

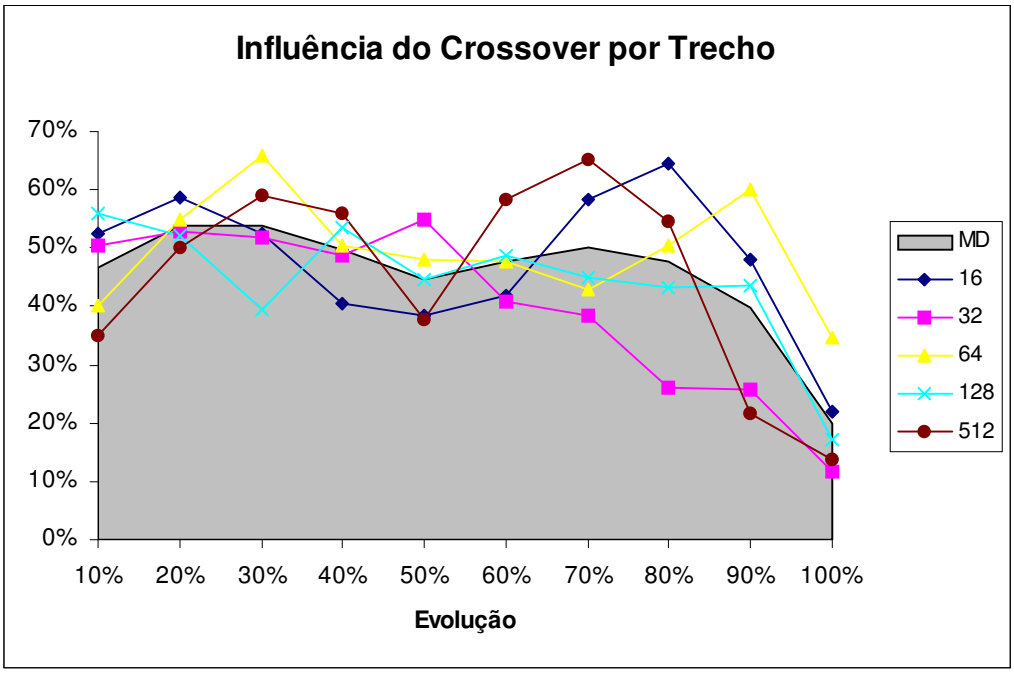

Figura 6.33 - Problema ts225 – Influência do Crossover por Trecho – Média

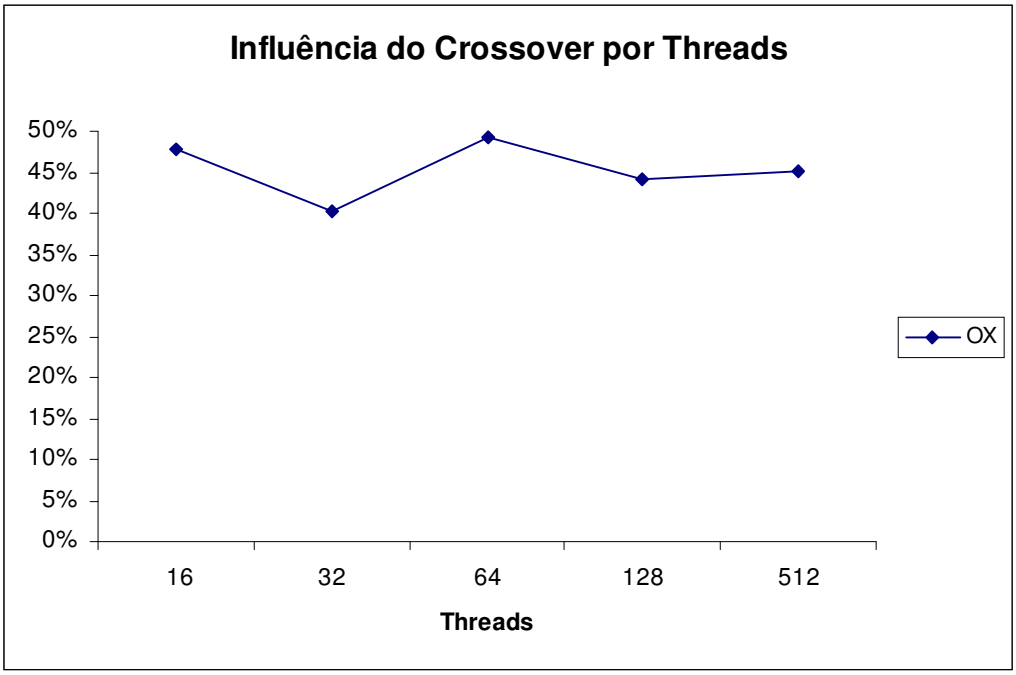

Figura 6.34 - Problema ts225 – Influência do Crossover – Threads

Finalizando este problema, temos na figura 6.35 a curva de convergência entre as estratégias SA puro e SA crossover OX. Vemos claramente uma curva sem sobresaltos na estratégia SA puro; o modelo apresenta uma convergencia rápida após poucos passos, sendo que a melhora da solução torna-se difícil de ser alcançada a medida que o sistema congela, pois situações de maior energia (pior custo) são mais difíceis de serem aceitas. Para a estratégia SA crossover OX, esta apresenta quase que até o final do processo, alternativas diferenciadas de soluções devido ao crossover, onde ao final é localizado uma solução melhor que a localizada no modelo puro.

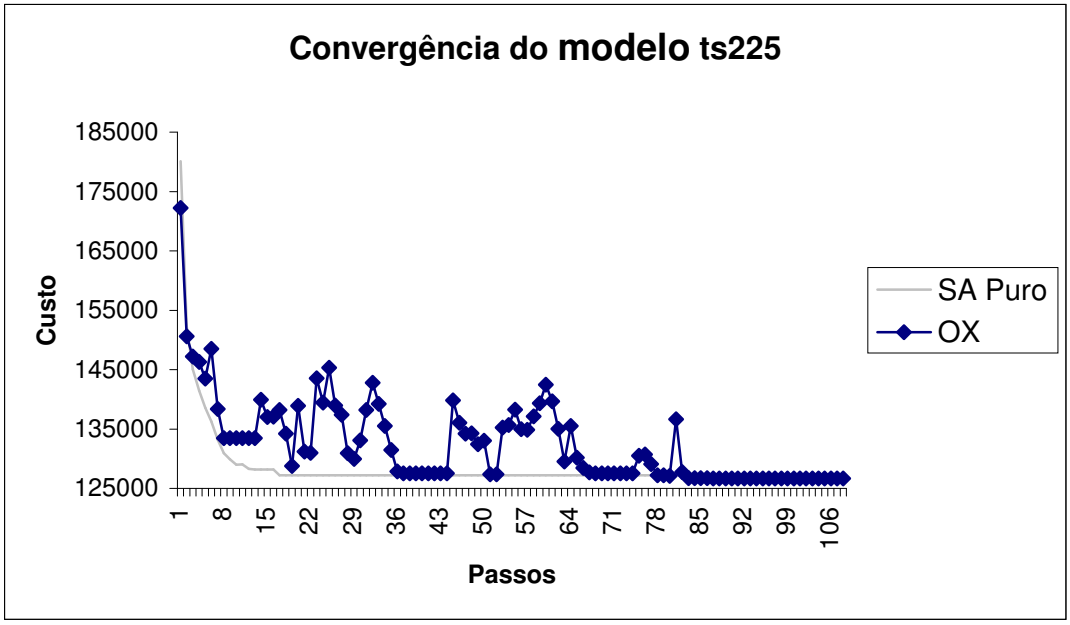

Figura 6.35 - Problema ts225 – Convergência da solução – Comparação entre modelos SA puro e SA crossover OX

A figura 6.36 demonstra os estados inicial e final do problema otimizado pela estratégia crossover OX, apresentado o resultado gerado pela execução de melhor indivíduo com 512 Threads.

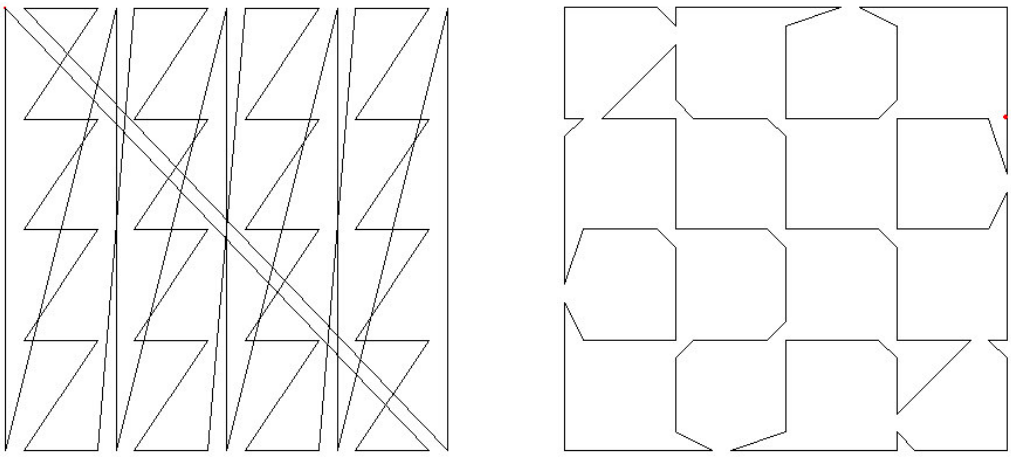

Figura 6.36 - Problema ts225 – Estado inicial e final do problema após otimização pela estratégia crossover OX

6.3.4 Problema Grade 21 x 21

Este problema apresenta um grau de dificuldade mais elevado, não somente devido a quantidade de pontos, mas principalmente pela forma de sua distribuição. Em nenhuma das tentativas foram atingidos o custo ótimo, tanto pelo algoritmo SA puro quanto pelo algoritmo SA crossover OX.

Vemos que a partir deste problema, para baixas quantidades de threads a estratégia de crossover apresenta soluções piores que a estratégia pura, e a medida em que aumentamos o número de threads participantes, elevamos a probabilidade de atingirmos resultados melhores, devido a quantidade maior de cruzamentos realizadas.

As figuras 6.37, 6.38 e 6.39 apresentam respectivamente o desempenho, a qualidade média do conjunto de execuções realizadas e a qualidade do melhor indivíduo identificado. Comparando a qualidade média com a qualidade do melhor indivíduo, verificamos que a estratégia de cruzamentos ainda apresenta resultados superiores ao método puro, porém deve-se selecionar um indivíduo de uma quantidade maior de execuções.

Devido a problemas de tempo de processamento, não foi possível tentar resolver o problema com um conjunto maior de threads, com que provavelmente teríamos um incremento na qualidade da solução, considerando os resultados até aqui obtidos.

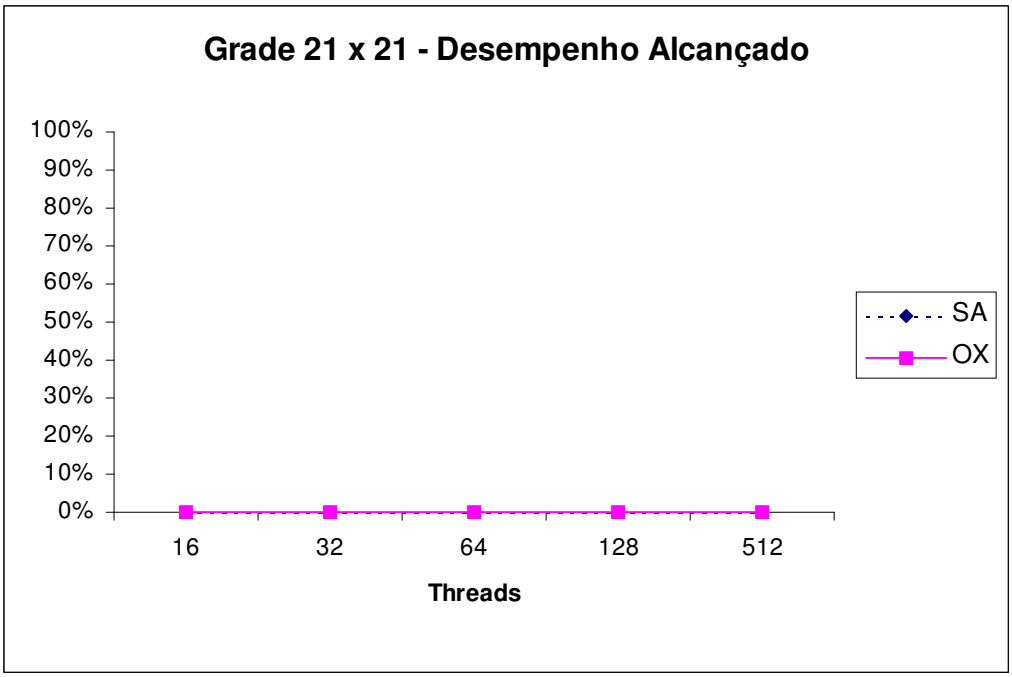

Figura 6.37 - Problema Grade 21x21 – Desempenho por Threads

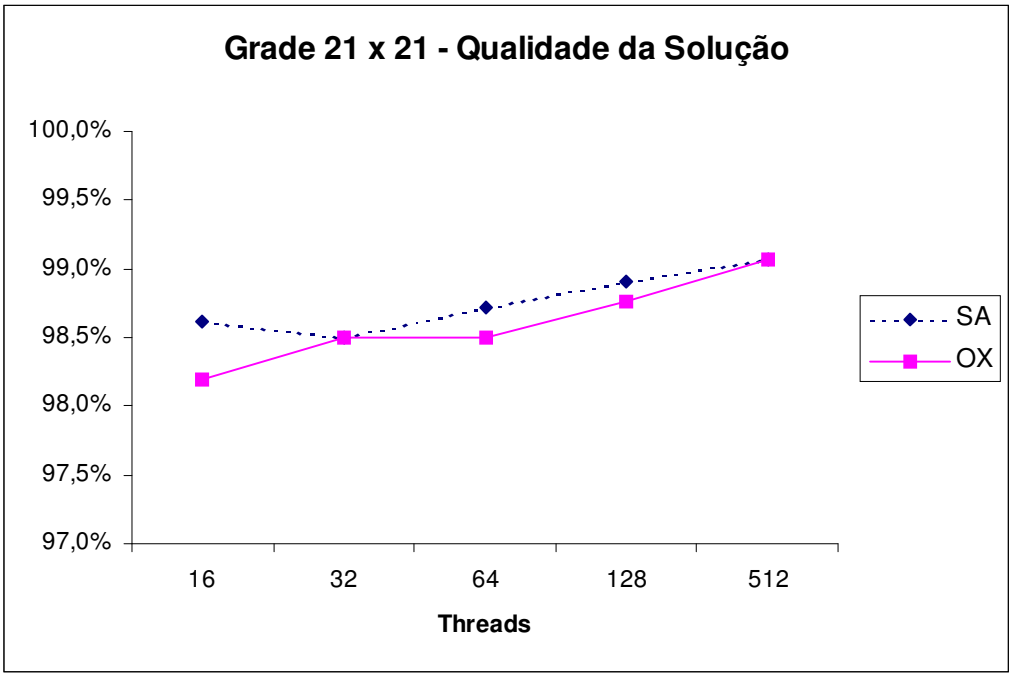

Figura 6.38 - Problema Grade 21x21 – Qualidade da solução por Threads

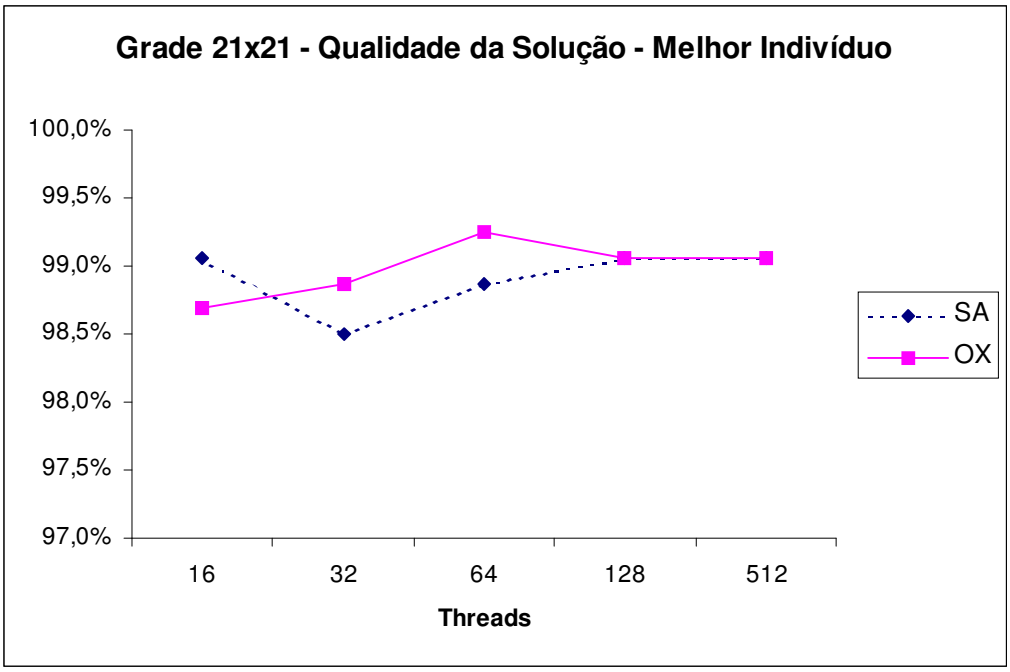

Figura 6.39 - Problema Grade 21x21 – Qualidade da solução do melhor indivíduo por Threads

Analisando os cruzamentos, verificamos novamente o mesmo comportamento obtido no estudo dos problemas anteriores.

Temos nas figuras 6.40 e 6.41 demonstrado o ocorrido, sendo que permanece até 80% da simulação uma média de cruzamentos acima de 40% e ao final esta cai em torno dos 10%. Também para este problema, verificamos que a influência dos cruzamentos na identificação da melhor solução não é afetada pela quantidade de threads.

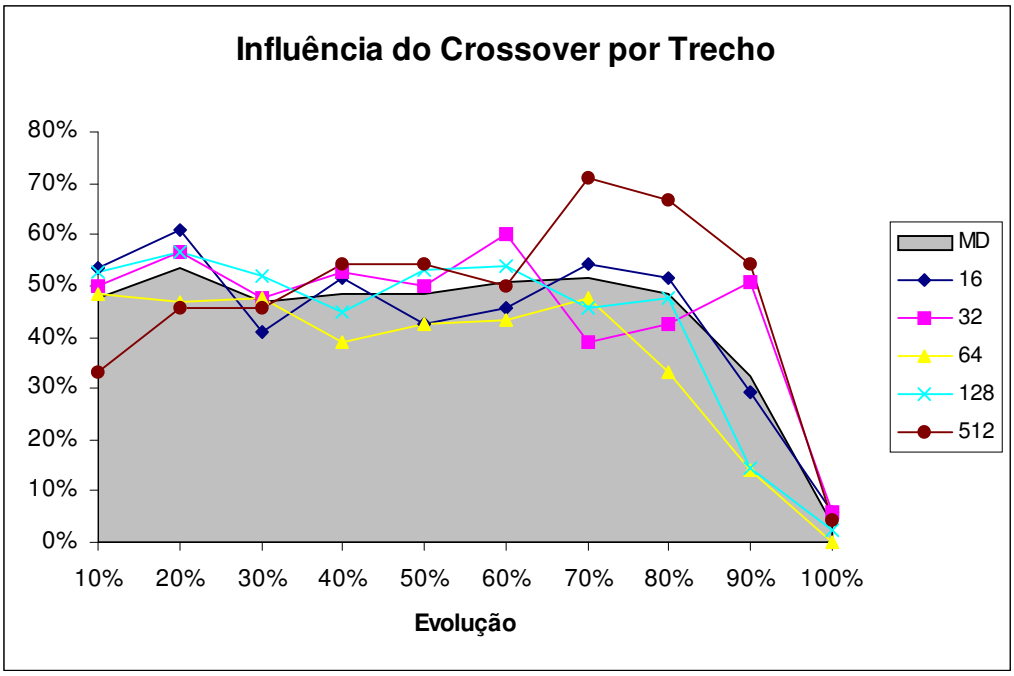

Figura 6.40 - Problema Grade 21x21 – Influência do Crossover por Trecho Média

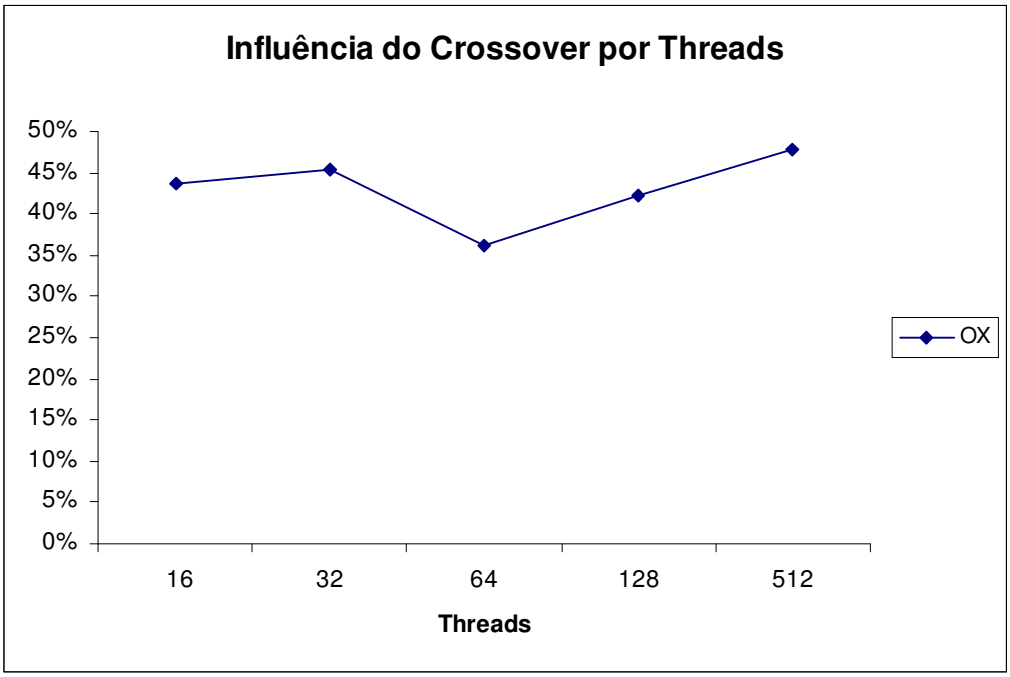

Figura 6.41 - Problema Grade 21x21 – Influência do Crossover – Threads

A curva de convergência para este problema, apesar de manter o grau de cruzamentos realizados, apresenta-se mais próxima ao do modelo puro, conforme podemos constatar na figura 6.42. Isto é devido ao fato da natureza da distribuição homogênea dos pontos no espaço de busca, sendo que os cruzamentos ocasionam pouca interferência após sua aplicação.

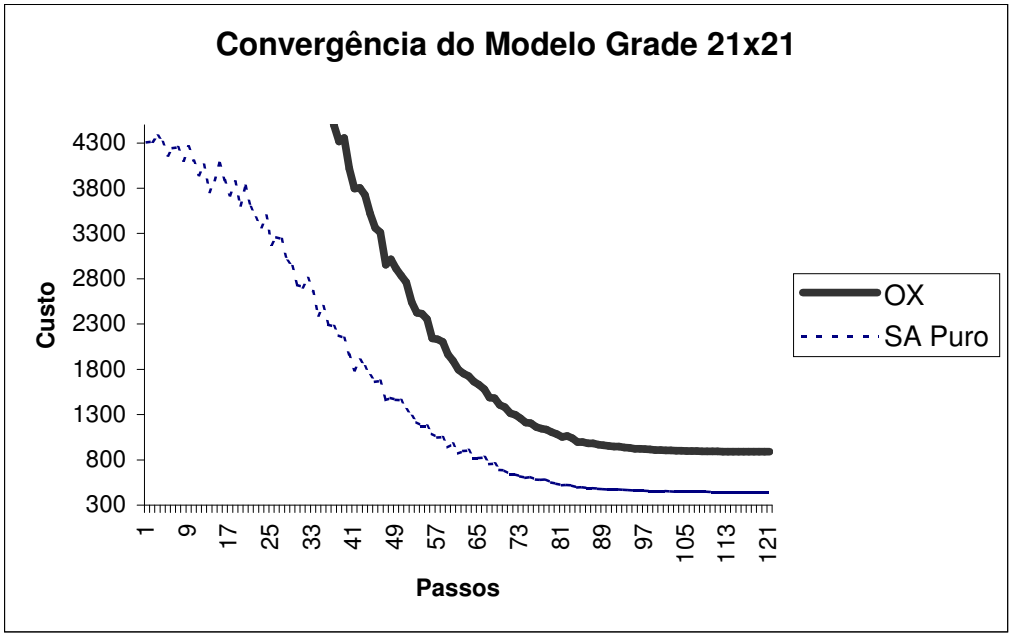

Figura 6.42 - Problema Grade 21x21 – Convergência da solução – Comparação entre modelos SA puro e SA crossover OX

A figura 6.43 apresenta os estados inicial e final da otimização pela estratégia crossover OX. O modelo otimizado apresentado corresponde à quinta execução da simulação OX com 64 Threads, sendo que o processo de número 57 foi o que obteve o melhor resultado individual.

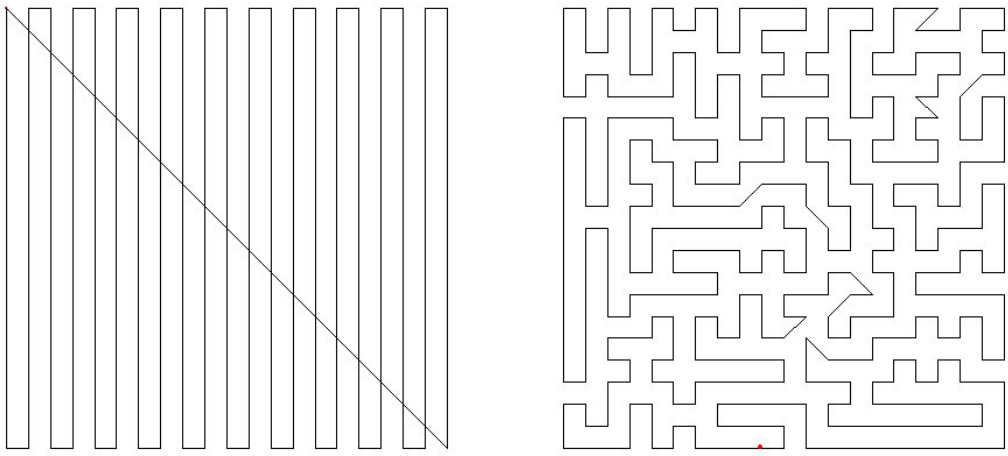

Figura 6.43 - Problema Grade 21x21 – Estado inicial e final do problema após otimização pela estratégia crossover OX

6.3.5 Problema pr1002

Este problema teve uma evolução na qualidade da solução a medida em que aumentamos a quantidade de threads, de maneira bem mais significativa que o problema anterior. Apesar do desempenho ótimo não ter sido alcançado, os resultados foram excelentes se comparados com a estratégia SA pura.

Vemos nos figuras 6.44, 6.45 e 6.46 os gráficos que demonstram os resultados obtidos, verificando o ocorrido de que a qualidade da solução com 512 threads foi relativamente inferior à qualidade obtida com 128 threads. Pode-se justificar isto pelo fato de que foram realizadas uma quantidade menor de execuções com 512 threads, e somente na quarta simulação com 128 threads é que obtivemos o melhor resultado de todo o conjunto de simulações.

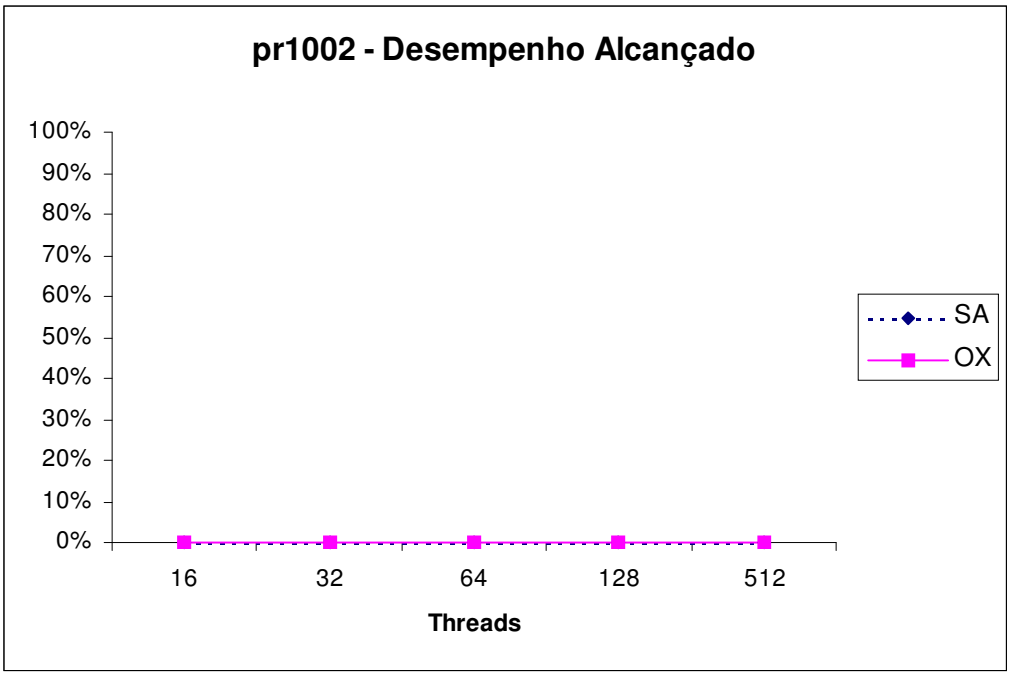

Figura 6.44 - Problema pr1002 – Desempenho por Threads

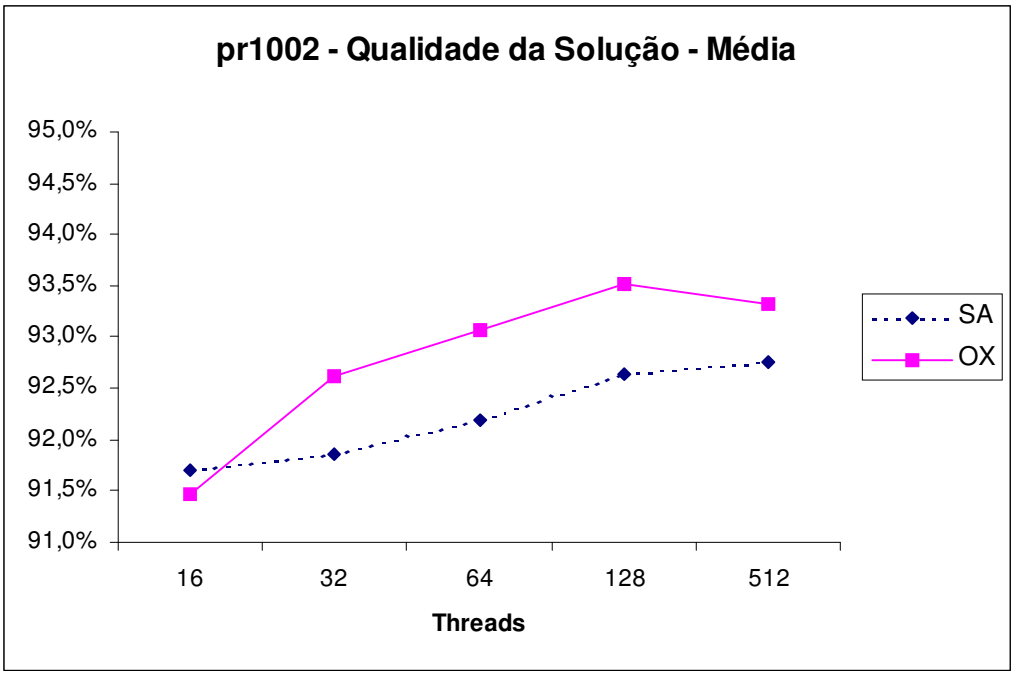

Figura 6.45 - Problema pr1002 – Qualidade da solução por Threads

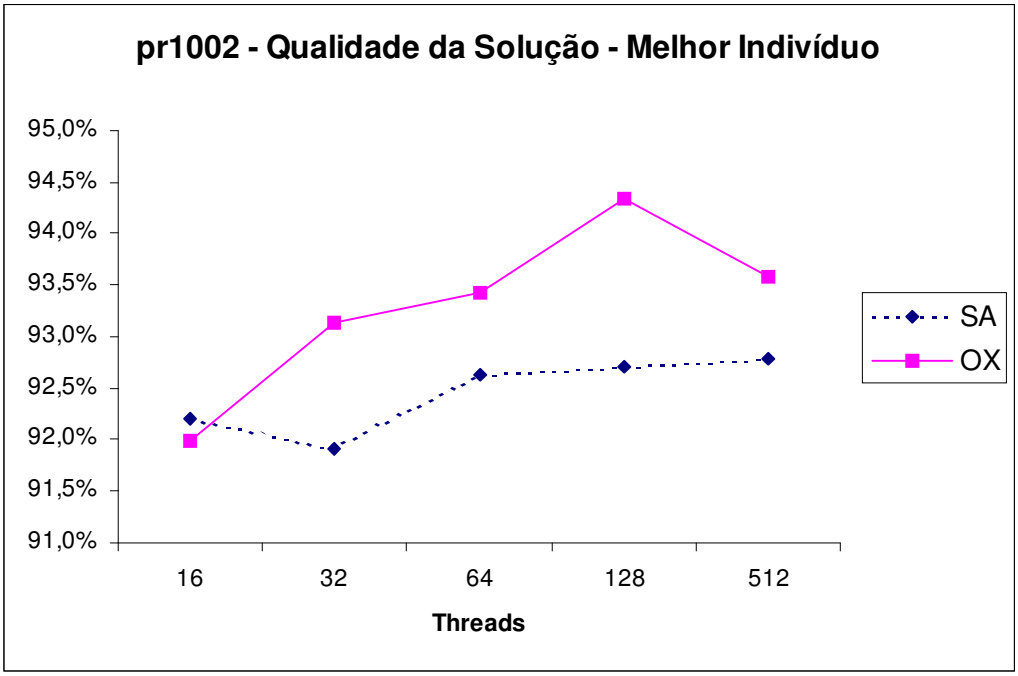

Figura 6.46 - Problema pr1002 – Qualidade da solução do melhor indivíduo por Threads

Vemos para este problema na figura 6.47 e 6.48, que a influência do crossover continua atuando sobre a melhor solução encontrada, seguindo a mesma tendência dos problemas anteriores, porém com uma leve redução dos percentuais durante todo o processo da simulação.

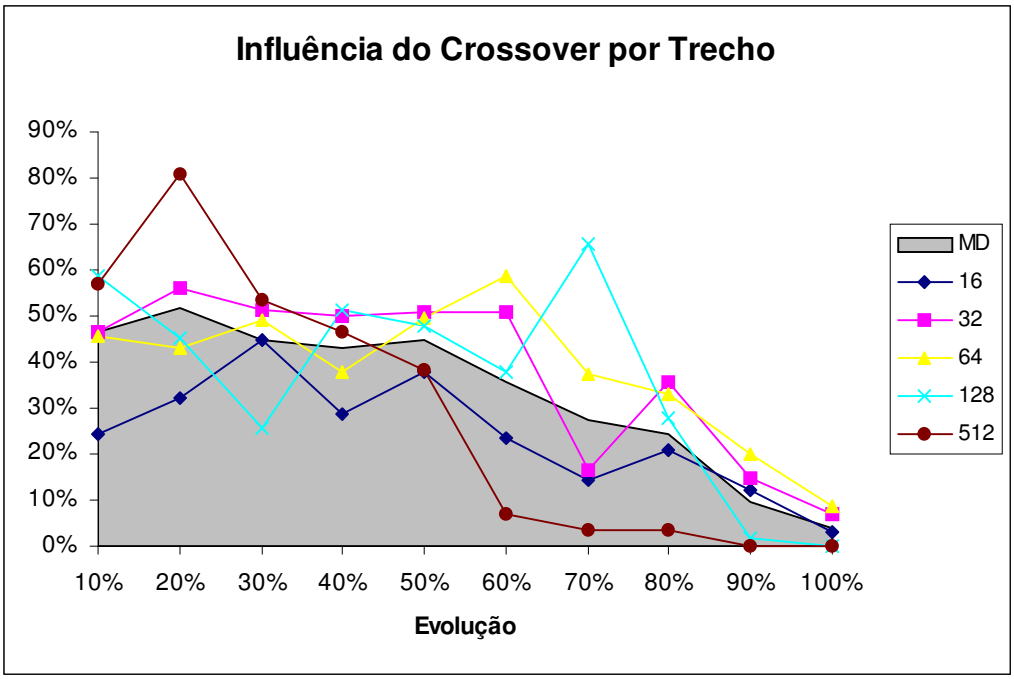

Figura 6.47 - Problema pr1002 – Influência do Crossover por Trecho – Média

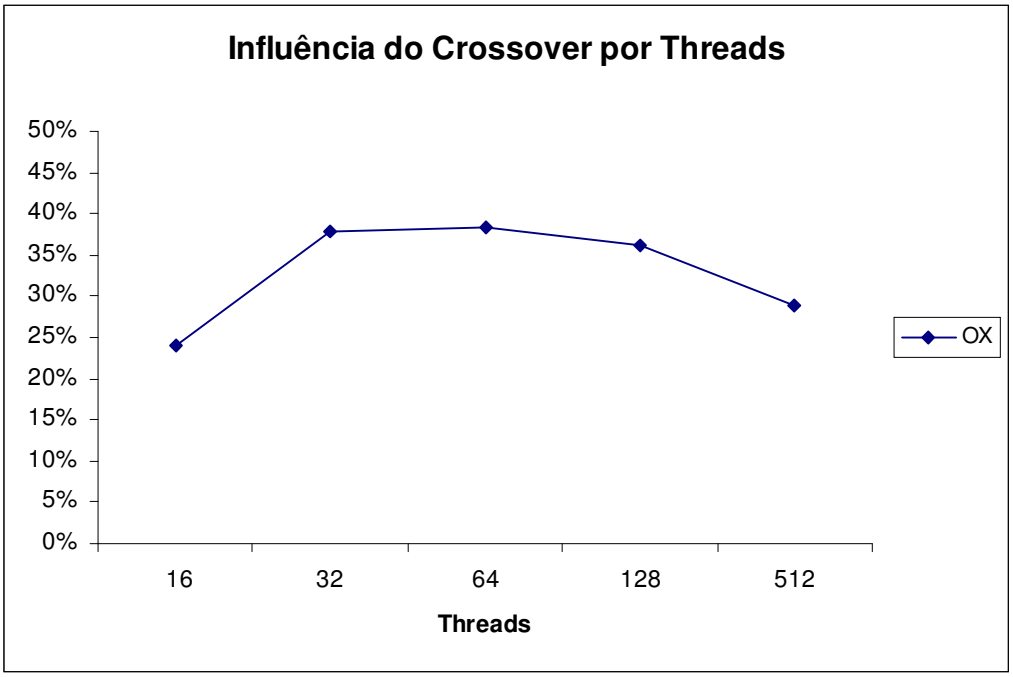

Figura 6.48 Problema pr1002 – Influência do Crossover – Threads

A figura 6.49 demonstra claramente a influência do crossover na resolução do problema, enquanto que a estratégia SA pura convergiu em no máximo 20% da simulação, a estratégia crossover OX, foi testando novos espaços de busca ao longo da otimização, sendo que ao final encontrou uma solução relativamente superior a outra estratégia.

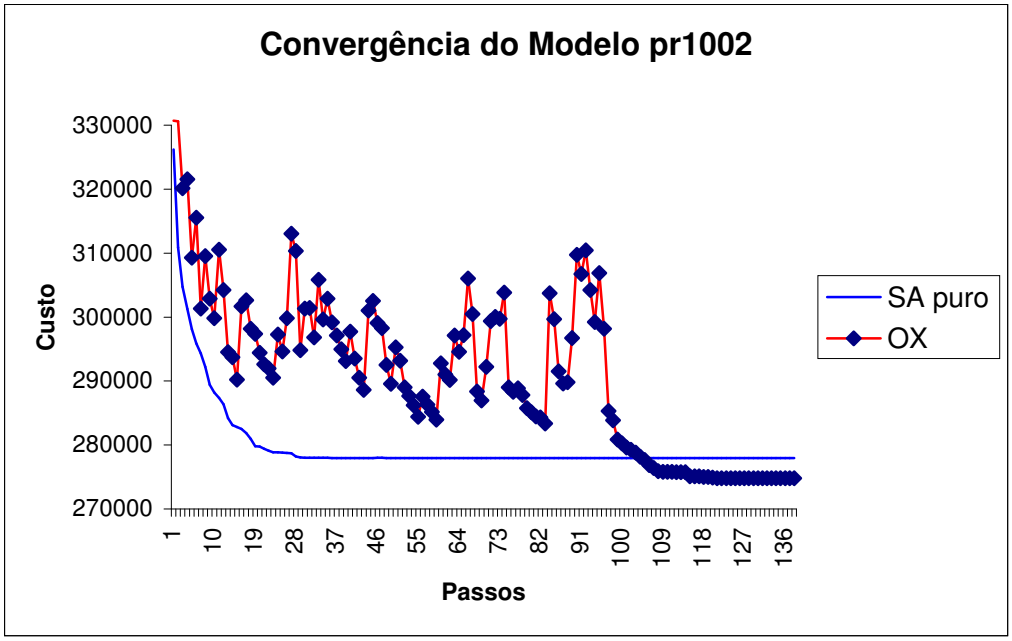

Figura 6.49 - Problema pr1002 – Convergência da solução – Comparação entre modelos SA puro e SA crossover OX

A figura 6.50 apresenta os estados inicial e final da otimização pela estratégia crossover OX. O modelo otimizado apresentado corresponde a quarta execução da simulação OX com 128 Threads, sendo que o processo de número 93 foi o que obteve o melhor resultado individual.

Figura 6.50 - Problema pr1002 – Estado inicial e final do problema após otimização pela estratégia crossover OX

6.3.6 Problema fnl4461

Este problema, apresenta um resolução bem complexa devido ao seu tamanho, com 4461 pontos. A estratégia SA pura obteve resultados levemente melhores que a estratégia com crossover. Temos nas figuras 6.51, 6.52 e 6.53 apresentados os resultados das simulações.

A principal dificuldade foi a questão de tempo de processamento, pois como visto nos outros problemas apresentados, a medida em que aumentamos a quantidade de threads, aumentamos também a quantidade de cruzamentos e a possibilidade de atingir melhores resultados. O ideal para este problema seria a simulação com a quantidade de Threads bem acima de 512, porém esta levaria alguns dias para sua finalização.

Apesar dos resultados não apresentarem melhora em relação ao modelo puro, podemos ainda afirmar que os resultados obtidos são de boa qualidade devido a complexidade do problema e à proximidade destes com os demais resultados, chegando estes a quase 89% da resultado ideal esperado.

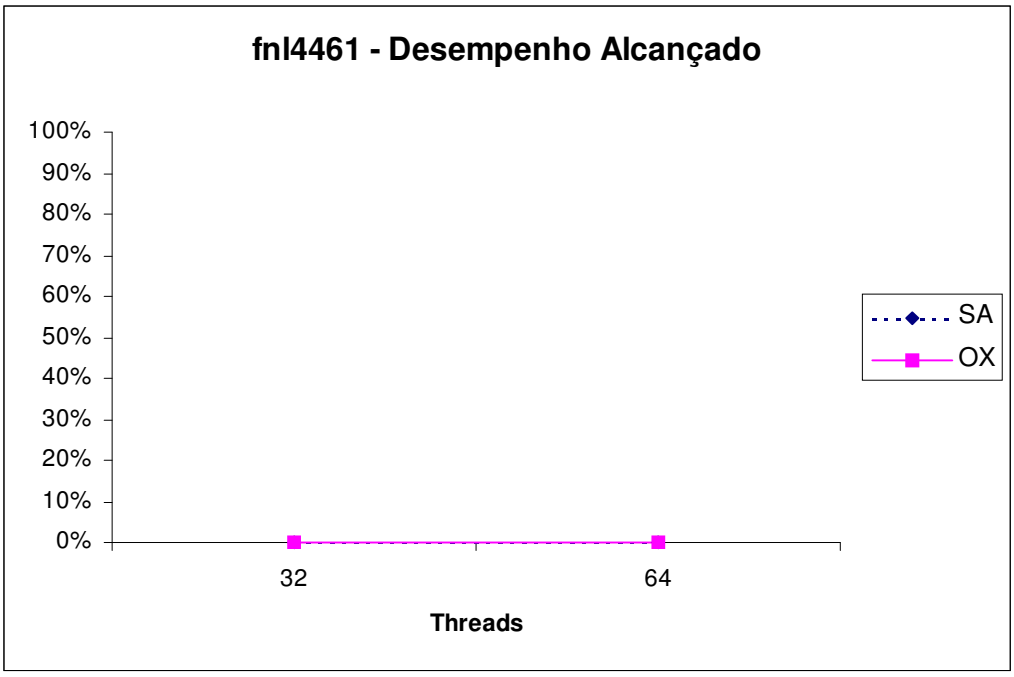

Figura 6.51 - Problema fnl4461 – Desempenho por Threads

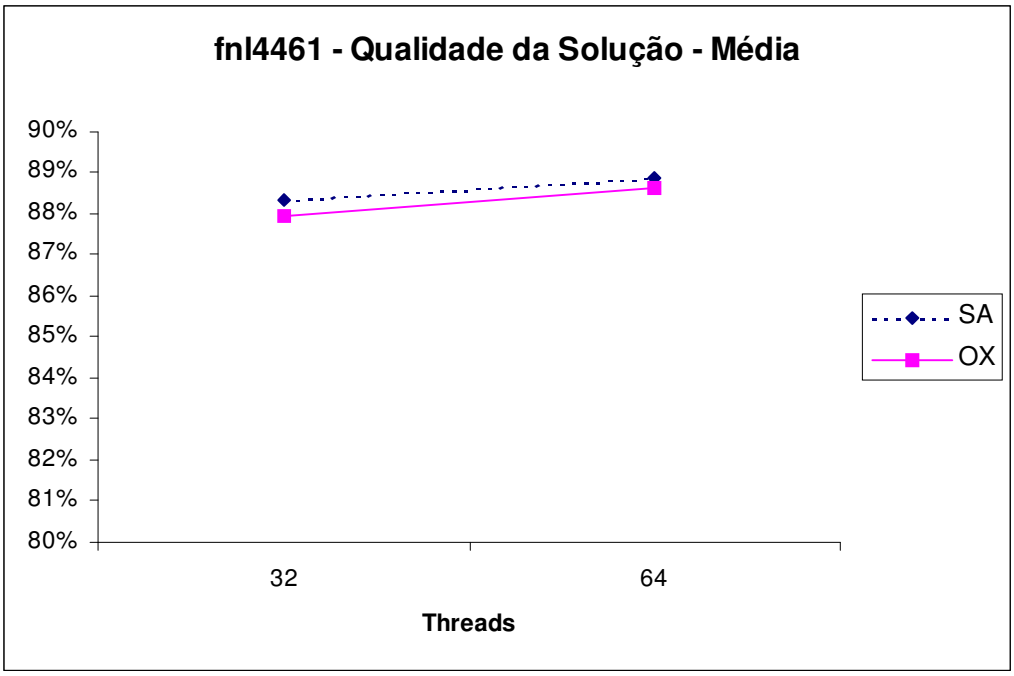

Figura 6.52 - Problema fnl4461 – Qualidade da solução por Threads

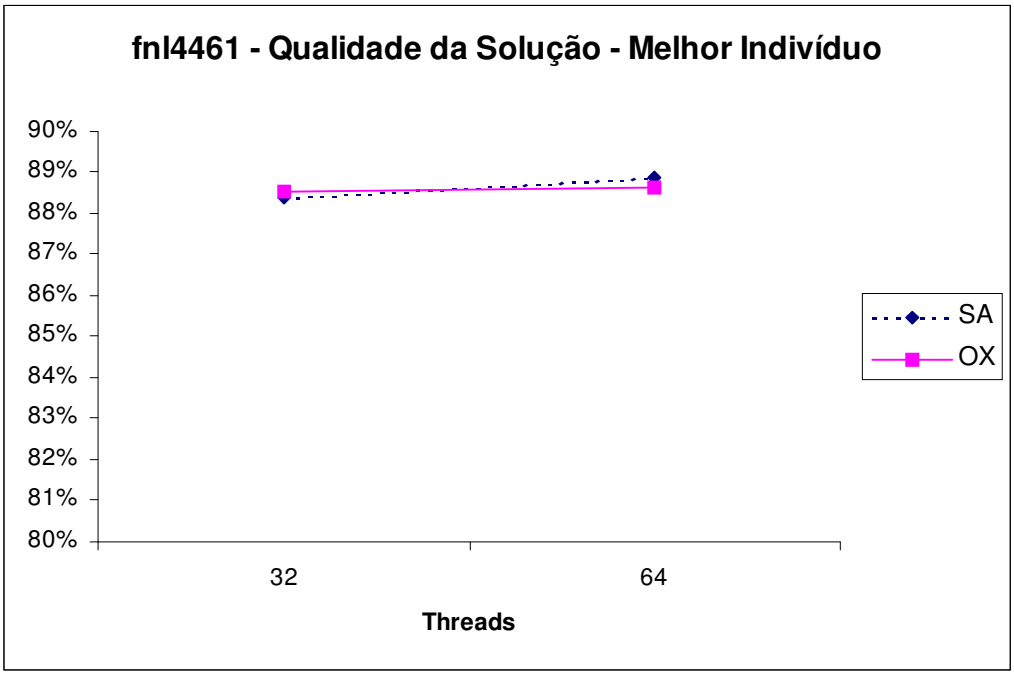

Figura 6.53 - Problema fnl4461 – Qualidade da solução do melhor indivíduo por Threads

Os gráficos que apresentam a influência do crossover na obtenção dos resultados são ligeiramente mais bruscos devido a quantidade menor de execuções (apenas 2 por quantidade de threads). Observamos na figura 6.54 o ocorrido e verificamos que após 80% da solução

encontrada, a influência do crossover cai quase que a 0% de influência sobre o resultado final obtido, o que provavelmente indica a necessidade de uma quantidade maior de threads durante a simulação. A figura 6.55 apresenta a média final por quantidade de threads da influência do crossover. Vemos que esta não apresenta grandes variações em relação aos outros problemas estudados.

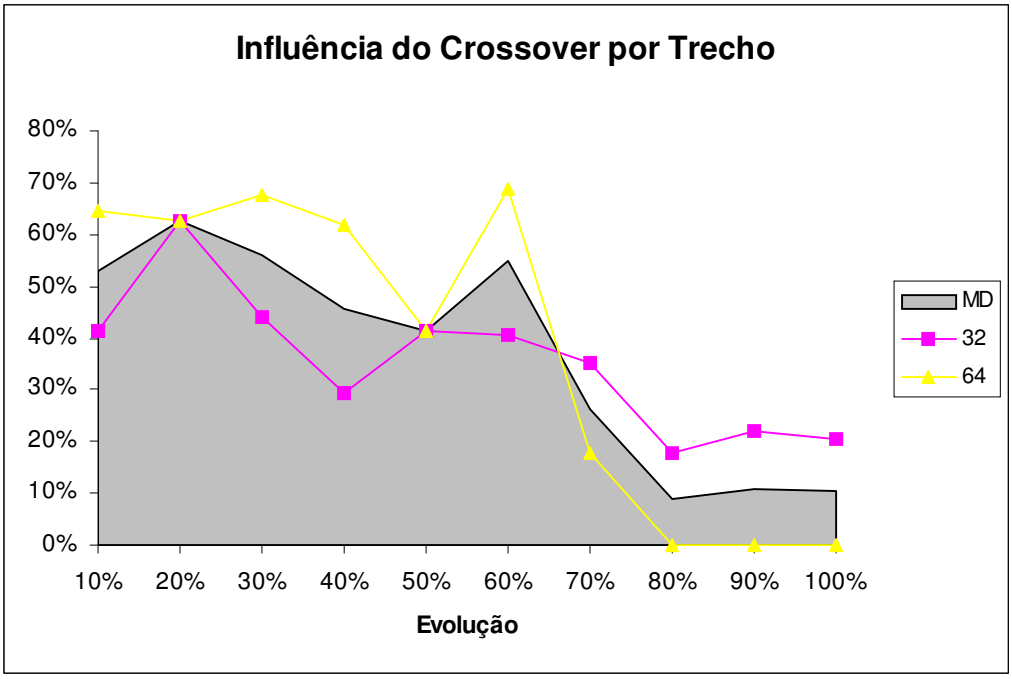

Figura 6.54 - Problema fnl4461 – Influência do Crossover por Trecho – Média

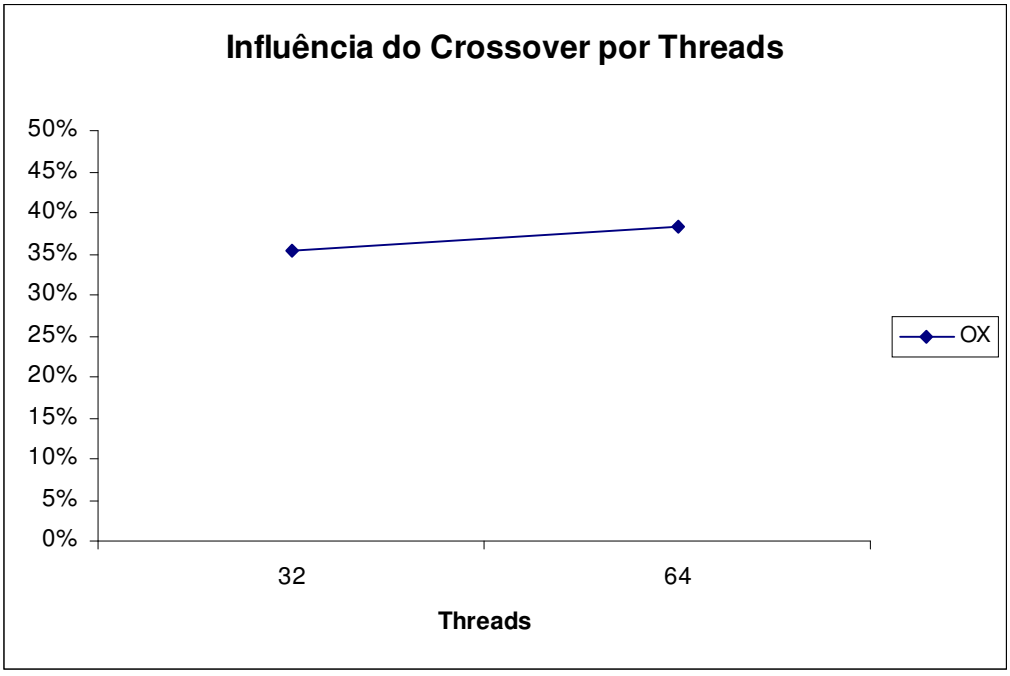

Figura 6.55 - Problema fnl4461 – Influência do Crossover – Threads

As figuras 6.56 e 6.57 exemplificam os resultados obtidos, apresentando a curva de convergência que compara a estratégia SA pura e a estratégia crossover OX e os estados inicial e final obtidos da solução otimizada, obtido este pela segunda execução da simulação com 64 threads do processo 34.

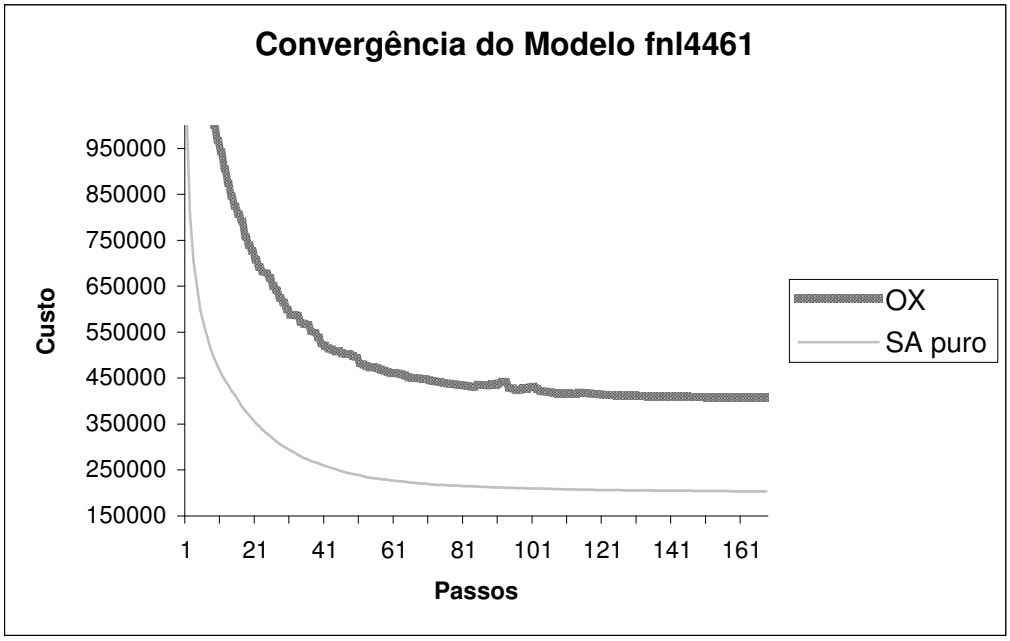

Figura 6.56 - Problema fnl4461 – Convergência da solução – Comparação entre modelos SA puro e SA crossover OX

Figura 6.57 - Problema fnl4461 – Estado inicial e final do problema após otimização pela estratégia crossover OX

6.3.7 Problema d15112

Este foi o último problema estudado. As duas estratégias avaliadas, SA puro e SA crossover OX apresentaram resultados equivalentes, sendo que o modelo puro obteve para o indicador qualidade um valor sensivelmente maior, conforme apresentado na figura 6.58.

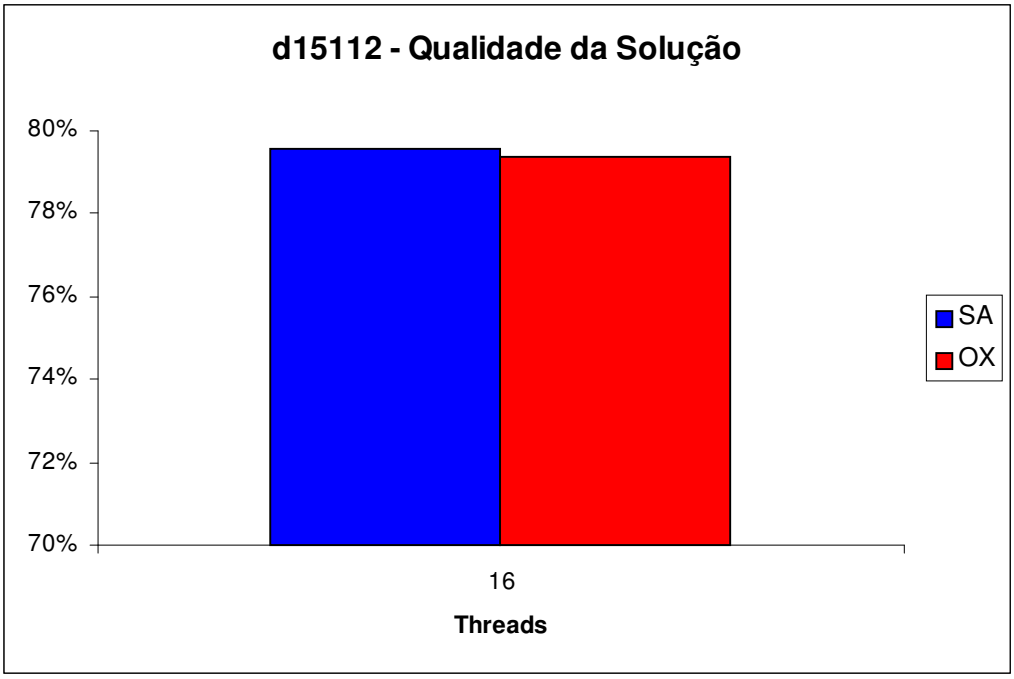

Figura 6.58 - Problema d15112 – Qualidade da solução por Threads

Os limites de processamento não foram atingidos com este problema, apesar da grande quantidade de pontos que o mesmo possui (15112 pontos), o modelo desenvolvido suporte problemas até 1024 Threads com 40.000 pontos, ou variações deste como 4096 Threads com 10.000 pontos, neste caso teremos como principal dificuldade a questão do tempo, como já comentado anteriormente.

A figura 6.59 apresenta um resultado muito interessante. Vemos que durante todo o processo de otimização do problema, a operação de crossover não influenciou em nada o resultado final obtido, ou seja, o resultado final apresentado foi exclusivamente gerado pelo esforço do próprio SA puro. Foi realizada outra simulação não apresentada aqui, sendo que os resultados obtidos foram semelhantes, com uma pequena influência do crossover somente nos 10% iniciais da simulação.

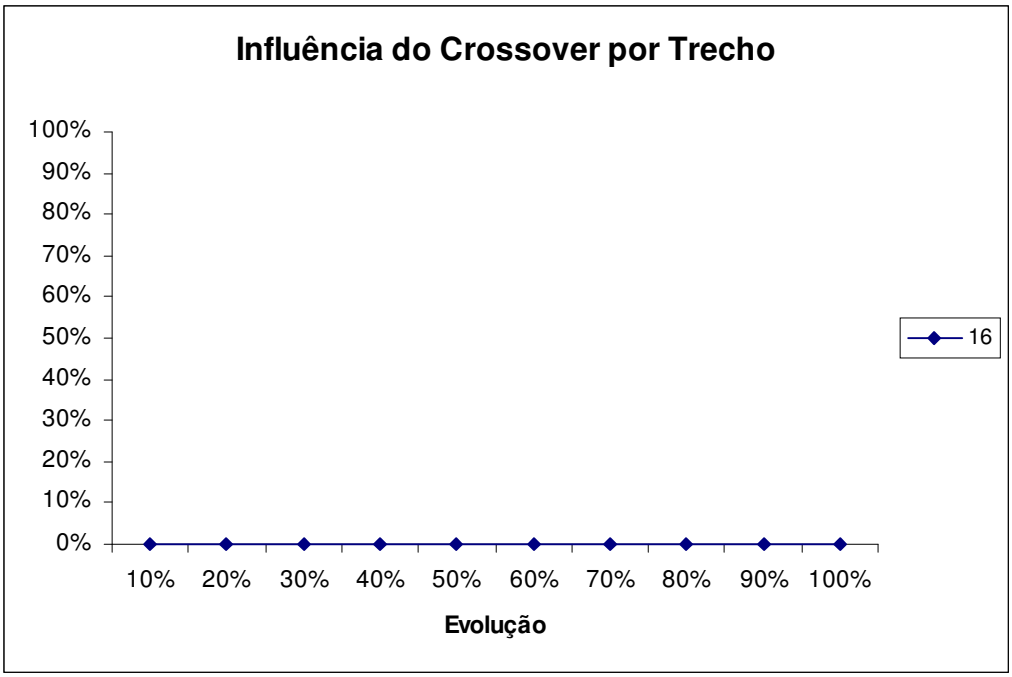

Figura 6.59 - Problema d15112 – Influência do Crossover por Trecho

Podemos concluir que o problema ocorreu devido a pequena quantidade de Threads utilizada, somente 16 Threads, fator já constatado no problema anterior nos últimos 80% da simulação.

Outra questão a ser levantada referente a este fato pode ser a influência que o aumento da quantidade de pontos gera sobre a influência do crossover no resultado final obtido, fator este que somente pode ser verificado através de simulações que utilizem grandes quantidades de Threads, necessitando para isto equipamentos mais capacitados para a realização das simulações.

Temos nas figuras 6.60 o gráfico que apresenta as curvas de convergência para as estratégias estudadas, que na prática, foram as mesmas sem a influência do crossover. Na figura 6.61 temos os estados inicial e final da solução proposta.

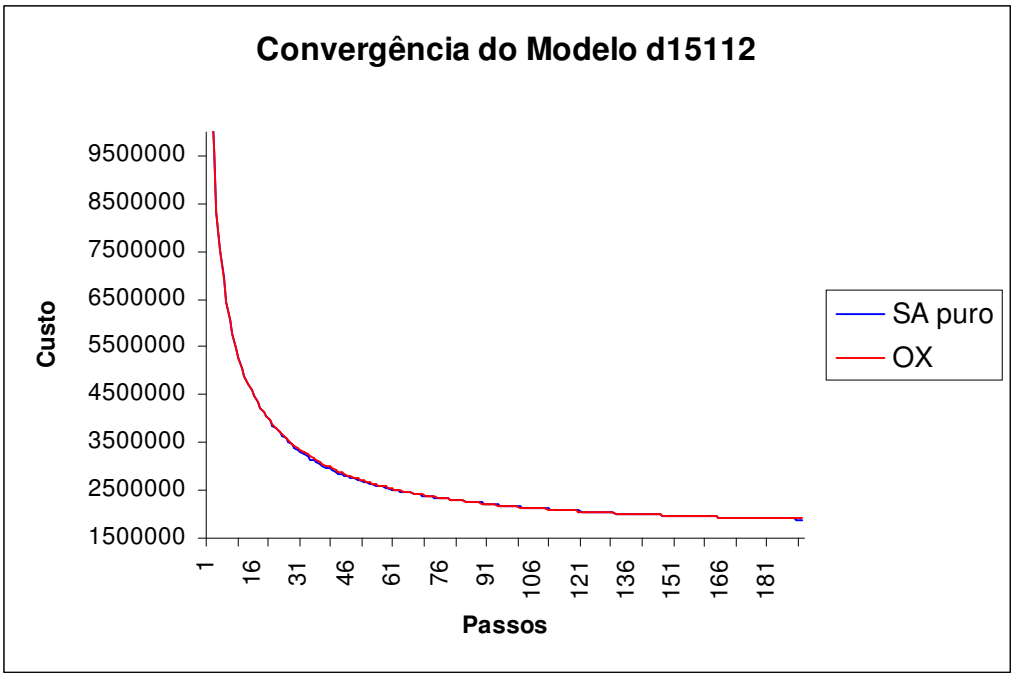

Figura 6.60 - Problema d15112 – Convergência da solução – Comparação entre modelos SA puro e SA crossover OX

Figura 6.61 - Problema d15112 – Estado inicial e final do problema após otimização pela estratégia crossover OX

6.4 Análise Geral dos Resultados

Os resultados obtidos foram satisfatórios na medida em que não obtivemos resultados inferiores aos apresentados na estratégia SA pura. Em todos os indicadores obtivemos melhorias, que foram mais expressivas para os primeiros problemas analisados.

Inicialmente estudamos várias estratégias de cruzamentos, sendo que a estratégia

crossover OX foi selecionada para ser utilizada como base para teste com os demais problemas, pois esta apresentou para os indicadores de desempenho e qualidade resultados superiores as demais estratégias.

Temos nas figuras 6.62 e 6.63 o comparativo do desempenho obtido para os problemas estudados, bem como um comparativo geral entre as duas principais estratégias avaliadas.

Vemos que a estratégia proposta obteve resultados bem superiores a estratégia pura.

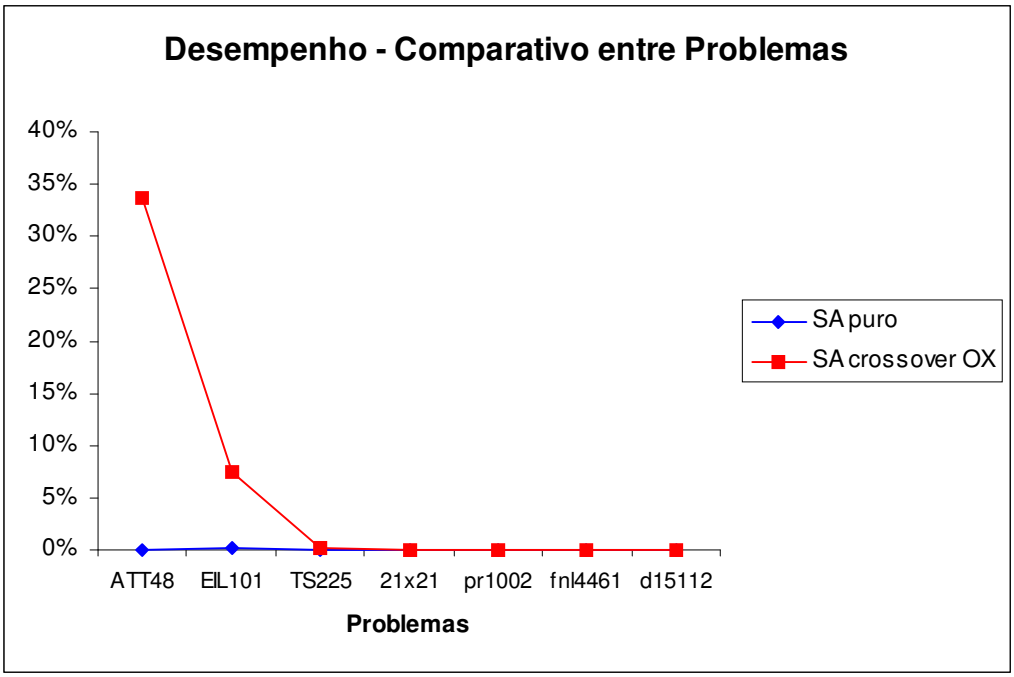

Figura 6.62 Desempenho – Comparativo entre problemas estudados

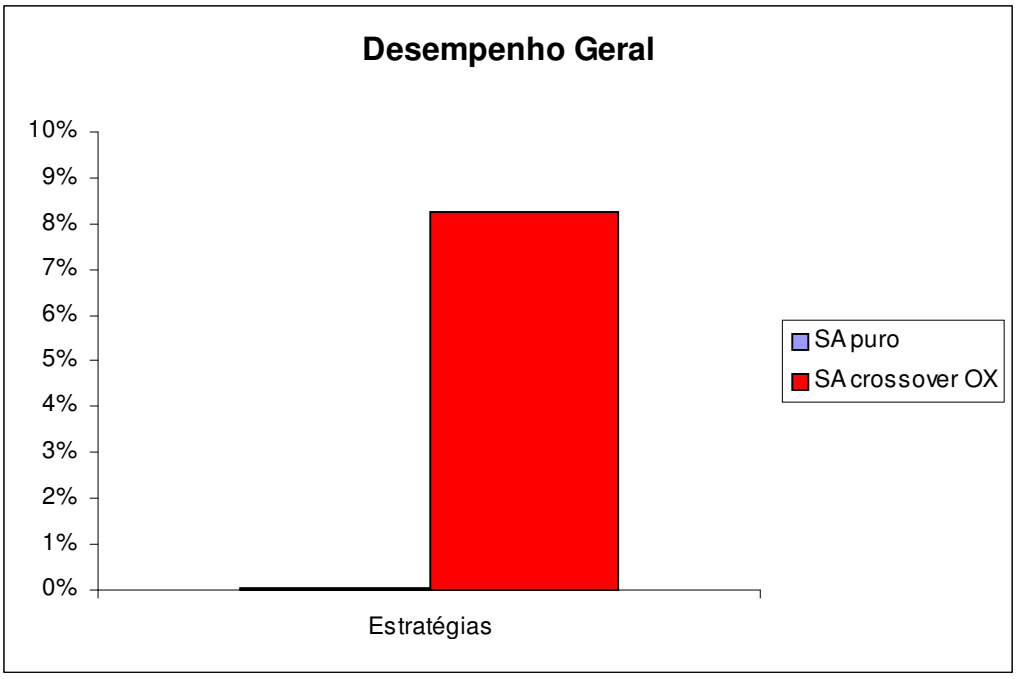

Figura 6.63 - Desempenho geral resultante

As figuras 6.64 e 6.65 da mesma forma que as figuras anteriores, apresentam o comparativo da qualidade das soluções obtidas entre os diversos problemas estudados, e o comparativo final da qualidade obtida. Observamos através destes gráficos, que o método proposto atingiu o seu objetivo pois este quando não melhorou a qualidade da solução apresentada, obteve resultados próximos aos resultados obtidos pela estratégia SA pura.

No comparativo final entre todos os problemas estudados, tivemos um acréscimo de 0,32% de melhora nos resultados encontrados pela estratégia proposta, o que é significativo em relação a qualidade que a estratégia SA pura obteve, esta em média de 94%. Avaliando em termos do erro ocorrido, temos para o estudo da estratégia SA puro o erro calculado de 6,19%, enquanto que para a estratégia SA crossover OX, tivemos um erro de 5,87%, o que significa que o modelo estudado reduziu o erro em 5,16%, ressaltando assim que foram alcançados resultados importantes comparados a estratégia SA pura.

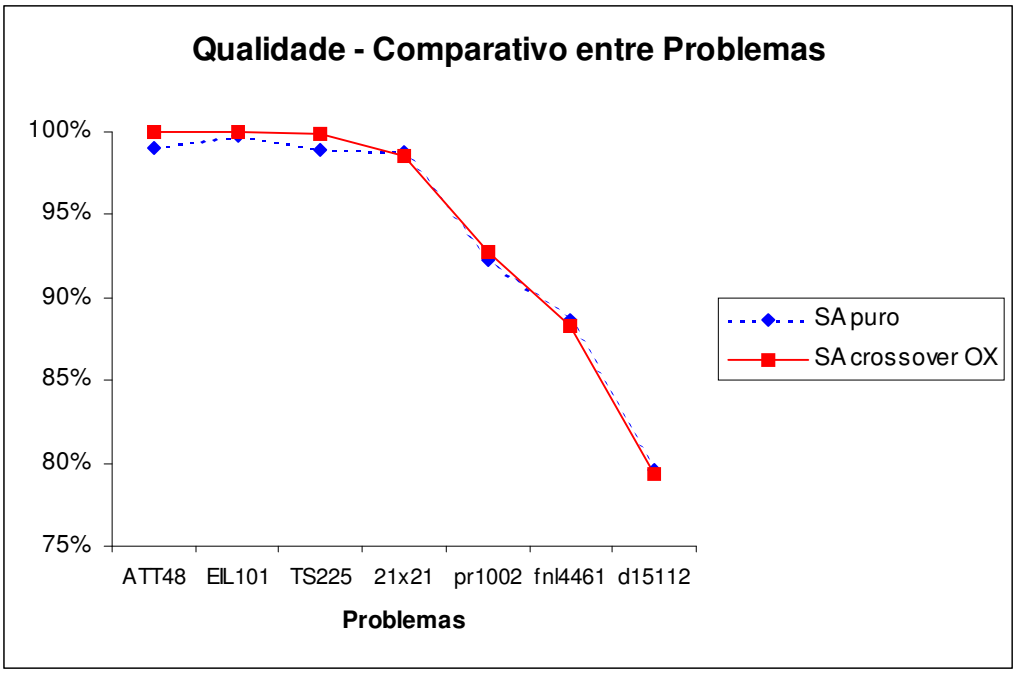

Figura 6.64 - Qualidade – Comparativo entre problemas estudados

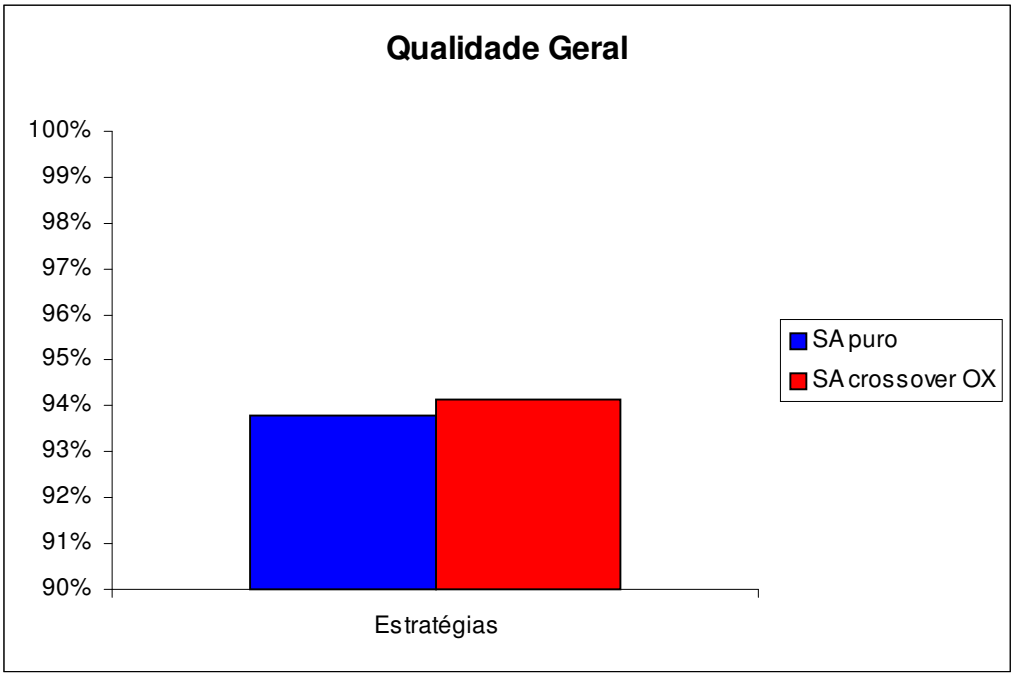

Figura 6.65 - Qualidade geral resultante

Finalmente analisando a influência que a operação de crossover gerou sobre os resultados alcançados, vemos que esta foi bastante significativa em todo o processo, conforme constatamos na figura 6.66. A operação de crossover apresentou uma média geral superior a 35%, tendo sua influência mais acentuada no início do processo e mais suave ao final.

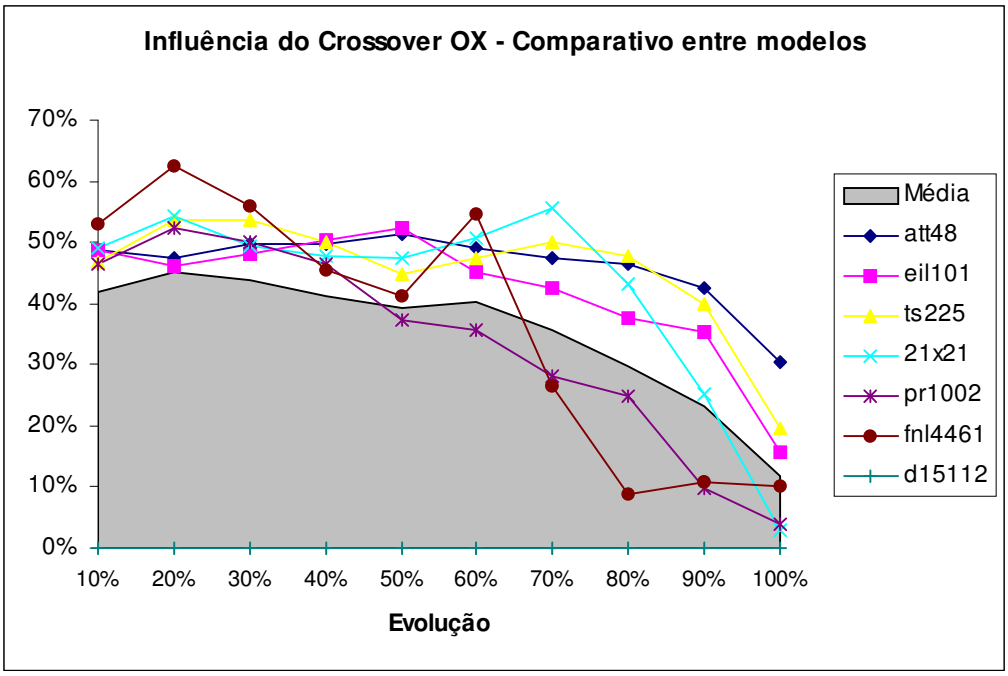

Figura 6.66 - Influência do crossover OX resultante

CAPÍTULO VII
CONSIDERAÇÕES FINAIS

Serão abordadas neste capítulo as conclusões finais para o modelo proposto, comentários sobre os resultados alcançados e propostas para pesquisas futuras.

7.1 Conclusões

O trabalho proposto apresentou um método híbrido entre os algoritmos Simulated Annealing e Genético, este fortemente concentrado na influência do operador crossover na busca de melhores soluções em problemas de otimização, principalmente na solução do clássico problema do Caixeiro Viajante.

Observa-se atualmente uma grande tendência na utilização de métodos aproximados na resolução desta classe de problemas. Uma abordagem dessa natureza, apesar de não garantir uma solução ótima para um problema, é capaz de oferecer uma solução de boa qualidade, em um tempo de processamento aceitável (MAZZUCCO, 1999).

As simulações realizadas mostraram que o método proposto representa uma boa alternativa relativa a outras técnicas heurísticas empregadas na solução dos problemas em questão que utilizam SA puro. Os resultados obtidos demonstram que o método apresenta performance variada e que vai melhorando à medida que o número de threads aumenta.

Para o método SA puro, apesar deste apresentar uma boa qualidade nos resultados alcançados, foi atingida uma redução em 5,16% do erro nas soluções dos problemas avaliados com a aplicação do método proposto. Isto foi bastante significativo, devido a grande diversidade dos problemas estudados, e dos recursos computacionais disponíveis.

Estes resultados, embora ainda não conclusivos, demonstram uma projeção otimista em relação ao potencial do método proposto, sendo este um dos primeiros testes realizados, podendo ser atingidos resultados ainda melhores com novas configurações dos parâmetros de entrada.

Entre as configurações que visam melhorar a qualidade das soluções alcançadas, deve-se realizar para os problemas acima de 1000 pontos, simulações com grandes quantidades de Threads e com outras alternativas de crossover (PMX e CX), verificando desta forma o real comportamento da proposta apresentada.

Um fato importante a ser ressaltado, é a grande influência causada pelo operador genético crossover em praticamente todas as etapas de resolução dos problemas, demonstrando desta forma que o crossover gerou perturbações positivas que auxiliaram o algoritmo Simulated Annealing a encontrar soluções ainda melhores.

7.2 Sugestões para Novos Trabalhos

Podemos descrever várias alternativas de estudo em função do método apresentado, porém a principal é a implementação da proposta em um ambiente fisicamente distribuído, ou em máquinas multiprocessadas, permitindo desta forma a simulação de problemas com um grau elevado de complexidade de forma mais rápida, permitindo assim testar outras alternativas de operadores genéticos, com grandes quantidades de indivíduos cooperantes.

Entre outras alternativas de pesquisa, temos:

Aplicação de outros operadores genéticos com o algoritmo SA. Como exemplo temos os operadores mutação ou inversão;

Controle do tempo de processamento com a adição de um time-out, necessário principalmente quando da implementação do modelo em um ambiente distribuído, para fins de garantir a finalização do processamento e apresentação dos resultados pelo gerente;

Aplicação de outras formas de seleção dos indivíduos para cruzamento. Para o modelo desenvolvido foi utilizada a abordagem de seleção aleatória, sendo que todos os indivíduos são selecionados somente uma única vez. Outras abordagens envolvem o grau de aptidão do indivíduo, podendo um indivíduo ser escolhido mais de uma única vez, em detrimento de outro.

Utilização da estratégia de clusterização proposta por KLIEWER (2000), com os operadores genéticos.

REFERÊNCIAS BIBLIOGRÁFICAS

ABNT. ASSOCIAÇÃO BRASILEIRA DE NORMAS TÉCNICAS. **Referências bibliográficas.** NBR 6023. Rio de Janeiro, 2002.

_____. **Citações**. NBR 10520. Rio de Janeiro, 2002.

_____. **Elaboração de trabalhos**. NBR 14724. Rio de Janeiro, 2002.

AARTS, E. & KORST, J. **Simulated Annealing and Boltzmann Machines**: a stochastic approach to combinatorial optimization and neural compoting, John Wiley & Sons – Interscience Series in Discrete Mathematics and Optimization, 1989.

ANDREWS, G.R. Paradigms for process Interaction in Distributed Programs. **CM Computing Surveys**, v. 23, n. 1, março de 1991.

APPLEGATE, D.; BIXBY, R.; CHVÀTAL, V. & COOK, W. **Microelectronics Technology Alert**, July 1998, Disponível em <http://www.crpc.rice.edu/CRPC/news Archive/mta_7_24_98.html>. Acesso em 20-03-2002.

ARAUJO, H.A. **Algoritmo Simulated Annealing**: Uma nova abordagem [Dissertação de Mestrado]. Florianópolis: UFSC, 2001.

BAL, H. E.; STEINER, J.G.; TANENBAUM, A.S. Programming Languages for Distributed Computing Systems, **ACM Computing Surveys**, v. 21, n. 3, 1989.

BEN-ARI. **Principles of Concurrent and Distributes Programming**, Prentice- Hall, 1985.

BENDER, E.A. **Mathematical Methods in Artificial Intelligence**. Washington : IEEE Computer Society, 1987.

BLAND, R.G. & SHALLCROSS, D.F. **Large traveling salesman problems arising from experiments in X-ray crystallography**: A preliminary report on computation, Operations Research Letters, vol. 8, 1989.

CELKO, J. Genetic Algorithms and Database Indexing : Finding the Best Set of Indexes. **Dr Dobb's Journal** – Abril 1993.

CERNY, V. Thermodynamical Approach to the Traveling Salesman Problem**:** An Efficient Simulation Algorithm, **Journal of Optimisation Theory and Applications**, n. 45, 1985.

CHAMBERS, L.. **Practical Handbook of Genetic Algorithms Aplications**. Rio de Janeiro: CRC Press LLC, 1995.

CHEN, H.; FLANN, N.S. & DANIEL W.W. **Parallel Genetic Simulated Annealing:** A Massive Parallel SIMD Algorithm., IEEE Transactions on Parallel and Distribuited Systems, Vol 9 No 2, 1998.

CHRISTOFIDES, N. & EILON, S. **Algorithms for large-scale traveling salesman problems**, Operational Research Quarterly, vol. 23, 1972.

CORMEN, T.H.; LEISERSON, C.E.; RIVEST, R.L. **Introduction to Algorithms**. Cambridge : McGraw-Hill, 1990.

CROWDER, H. & PADBERG, M. **Solving large-scale symmetric travelling salesman problems to optimality**, Mgmt Sci, vol. 26, 1980.

DANTZIG, G.; FULKERSON, R. & JOHNSON, S. **Solution of a large-scale traveling-salesman problem**, Operations Research, vol. 36, 1954.

DAVIS, L.D. **Job Shop Scheduling with Genetic Algorithms**, Proceeding of the 1st International Conference on Genetic Algorithms and their Applications, 1985.

DIJKSTRA, E. W. Solution of a Problem in Concurrent Programming Control. **Communication of ACM**, v. 8, n. 9, 1965, p. 569.

FLYNN, M. J. **Very high-speed computing systems**, Procedings IEEE, 1966

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

GAUTHIER, F.A.O. **Programação da Produção**: Uma Abordagem Utilizando Algoritmos Genético. [Tese de Doutorado] Florianópolis: PPGEP / UFSC, 1993.

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

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

GU, J. & HUANG, X. **Efficient local search with search space smoothing**: A case study of the traveling salesman problem (TSP), IEEE Trans. Systems, Man, and Cybernetics, vol. 24, 1994.

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

HIROYASU, T.; MIKI, M. & OGURA, M. **Parallel Simulated Annealingn using Genetic Crossover**, International Conference Parallel and Distribuited Computing and System, Las Vegas, Nevada, 2000.

HOARE, C.A. Monitors: An Operating System Structuring Concept. **Communication of the ACM**, v. 17, n. 10, 1974.

HOLLAND, J.H. **Adaptation in Natural and Artificial Systems.** Cambridge, MIT Press, p. 211, 1975.

JOHNSON, D.S. & MCGEOCH, L.A. The traveling salesman problem: A case study in local optimization, in (E.H.L. Aarts and J. K. Lenstra,eds), Local Search in **Combinatorial Optimization, Wiley & Sons**, New York, 1997.

KIRKPATRICK, S.; GELATT, Jr. C.D. & VECCHI, M.P. **Optimization by Simulated Annealing, Science**, n. 220, 1983.

KLIEWER, G. **A general software library for parallel simulated annealing**, University of Paderborn, Department of Mathematics and Computer Science, German, 2000.

KORTE, B. **Applications of combinatorial optimization**, talk at the 13th International Mathematical Programming Symposium, Tokyo, 1988.

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

LAPORTE, G. The traveling salesman problem**:** An overview of exact and approximate algorithms. **European Journal of Operational Research**, vol. 59, 1992.

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

LIN, S. Computer solutions of the traveling salesman problem. **Bell System Technical Journal**, n. 44, 1965.

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

LUNDY, M. & MEES, A. **Convergence of an annealing algorithm**. Mathematical Programming, vol. 34, 1986.

MAZZUCCO, J. **Uma Abordagem Híbrida do Problema da Programação da Produção através dos Algoritmos Simulated Annealing e Genético**. [Tese de Doutorado]. Florianópolis: UFSC, 1999.

METROPOLIS, W.; ROSENBLUTH, A.; ROSENBLUTH, M.; TELLER, A. & TELLER, E. Equation of State Calculations by Fast Computing Machines. **Journal of Chemical Physics**, vol. 21, 1953.

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

MIRANDA, M. N. **Algoritmos Genéticos**: Fundamentos e Aplicações, 1999. Disponível em <http://www.gta.ufrj.br/~marcio/genetic.html>.

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

PIRLOT, M. General local search methods. **European journal of operational research**, v. 1992.

REEVES, C.R. **Modern heuristic techniques for combinatorial problems**. Advanced topics in computer science, McGraw Hill, 1995.

REINELT, G. **Universität Heidelberg, Institut für Informatik.** Disponível em <http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/index.html> Acesso em 10/01/2003

RICIERI, A. P. **Otimização natural, a genética dos máximos e mínimos**. Guarulhos: Parma, 1994.

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

SILBERSCHATZ, A.; PETERSON, J.L. **Operating Systems Concepts**. Addison-Wesley, 1994.

SPILLMAN, R. Genetic Algorithms. **Dr Dobb's Journal** – February 1993.

TANOMARU, J. **Motivação, fundamentos e aplicações de algoritmos genéticos.** II

Congresso Brasileiro de Redes Neurais, UFSC - SC, 1995.

TANENBAUM, A.S. **Modern Operating Systems.** Prentice-Hall, 1992.

TANESE, R. **Distributed Genetic Algorithms for Function Optimization**. [Ph.D. Dissertation]. University of Michigan, 1989.

VACA, O.C.L. **Um algoritmo evolutivo para a programação de projetos multi-modos com nivelamento de recursos limitados**. Florianópolis : UFSC, 1995.

VAN LAARHOVEN, P.J.M. & AARTS, E.H.L. **Simulated Annealing:** Theory and Applications, Kluwer Academic Publisher, 1987.