UNIVERSIDADE DO ESTADO DA BAHIA

SISTEMAS DE INFORMAÇÃO

BARTIRA DE OLIVEIRA SENA MACHADO

UM ALGORITMO GENÉTICO PARA OTIMIZAÇÃO DO PROCESSO DE PERFURAÇÃO DE PLACAS DE CIRCUITO IMPRESSO

Salvador

2011

BARTIRA DE OLIVEIRA SENA MACHADO

UM ALGORITMO GENÉTICO PARA OTIMIZAÇÃO DO PROCESSO DE PERFURAÇÃO DE PLACAS DE CIRCUITO IMPRESSO

Monografia apresentada ao curso de Sistemas de Informação da Universidade do Estado da Bahia – UNEB, como pré-requisito para obtenção do grau de bacharel, sob orientação do Prof. Cláudio Alves de Amorim.

Salvador

2011

BARTIRA DE OLIVEIRA SENA MACHADO

UM ALGORITMO GENÉTICO PARA OTIMIZAÇÃO DO PROCESSO DE PERFURAÇÃO DE PLACAS DE CIRCUITO IMPRESSO

Monografia apresentada ao curso de Sistemas de Informação da Universidade do Estado da Bahia – UNEB, como pré-requisito para obtenção do grau de bacharel.

COMISSÃO EXAMINADORA

_____

Prof. Dr. Cláudio Alves de Amorim

_____

Prof. MSc. Tricia Souto Santos

_____

Prof. MSc. Julian Hermogenes Quezada Celedon

Salvador, 30 de setembro de 2011

AGRADECIMENTOS

À minha família, em especial a minha mãe, pelo apoio e carinho em todos os momentos da minha vida.

Ao meu namorado Anderson Barros pelo companheirismo e compreensão sempre que solicitados.

Aos meus colegas José Grimaldo e Caio Nascimento pela amizade, pelas horas de estudo e pelo incentivo incondicional.

À minha colega Sheila Matos pelos incontáveis momentos de descontração durante estes cinco anos de graduação.

Agradeço ainda aos colegas Allan Ariel, Felipe Piñeiro, Raul Abreu, Kelly Bregonci, Shankar Cabus, Juliana Fajardini, Bruno Vinicius, Adailton Junior, Marcos Cezar, André Araújo, Igor Eloi, Rafael Linsmar e Arnaldo Correia, que estiveram ao meu lado durante o curso, pela camaradagem.

Agradeço também ao professor Cláudio Amorim pela valiosa orientação e por toda a atenção que recebi durante a concepção deste trabalho, à professora Tricia Santos pelos conselhos ao longo desta caminhada e ao professor Diego Frias por se apresentar sempre prestativo quando requisitado.

Por fim, agradeço a todos os demais professores e funcionários do colegiado de Sistemas de Informação da UNEB, em especial os professores Maria Inês, Antonio Atta, Grinaldo Oliveira, Jorge Farias e Marcos Cerqueira, pela indiscutível contribuição na minha formação.

RESUMO

Este trabalho tem por fim apresentar uma solução aproximada do problema do caixeiro viajante com o intuito de otimizar o processo de perfuração de placas de circuito impresso pela redução do comprimento da rota a ser percorrida pela broca. Para tal, foi elaborado um algoritmo genético híbrido combinando heurísticas de construção (Heurística de *Savings*) e otimização de rotas (*RemoveSharp* e *LocalOpt*) a uma variação do operador de *crossover Edge Recombination* e um operador de mutação (*Shuffling*). Este trabalho inclui, ainda, as análises da importância da variabilidade genética na população de um algoritmo genético e dos efeitos da preservação das "subrotas comuns" a dois cromossomos no operador *Edge Recombination*. Para efeito de teste e validação do algoritmo elaborado, foram utilizadas as instâncias d198, a228 e pcb442 da TSPLIB que representam modelos de placas de circuito impresso. Os resultados obtidos foram comparados com outros algoritmos presentes na literatura comprovando a eficiência do algoritmo implementado, com uma variação de até 3,4% da solução ótima para cada uma das instâncias.

Palavras-chave: Placas de circuito impresso. Otimização. Problema do caixeiro viajante. Algoritmos Genéticos.

ABSTRACT

This work aims to present an approximate solution of the traveling salesman problem with the intention to optimize the drilling process of printed circuit boards by reducing the path length to be traveled by the drill. For such was prepared a hybrid genetic algorithm combining construction (Savings Heuristic) and optimization (RemoveSharp and LocalOpt) heuristics with a variation of a crossover operator (Edge Recombination) and a mutation operator (Shuffling). This work also includes the analysis of importance of genetic variability in the genetic algorithm population and the preservation effects of the 'common subsequences' to two chromosomes in the Edge Recombination crossover operator. For testing and validation purposes of the elaborate algorithm were used the instances d198, a228 and pcb442 of the TSPLIB that represent printed circuit boards designs. The results were compared with other algorithms proving the efficiency of the implemented algorithm with a range up to 3,4% of the optimal solution for each instance.

Keywords: Printed circuit boards. Optimization. Traveling salesman problem. Genetic algorithms.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

1. INTRODUÇÃO

Controle Numérico Computadorizado (CNC do inglês *Computer Numeric Control*) é um conjunto de técnicas de automação com diversas aplicações em processos industriais. A CNC é uma evolução do Controle Numérico (NC do inglês *Numeric Control*), desenvolvido pelo MIT (*Massachusetts Institute of Technology*) na década de 40, que permitiu a um operador se comunicar com esta através de números e símbolos.

As CNCs são, atualmente, fatores chave na produção de milhares de produtos em todo o mundo e, desde sua criação, trouxeram inúmeras mudanças na indústria metalúrgica. As máquinas CNC possibilitaram, por exemplo, que a indústria produzisse, rotineiramente, peças com precisão antes impensável, além de diminuir o tempo de produção. (Krar e Gill, 1999)

As máquinas CNCs são utilizadas em diversos processos industriais de acordo com a programação, dentre eles os processos de perfuração em lote de placas de circuito impresso (PCB do inglês *Printed Circuit Board*). As placas de circuito impresso são placas de suporte, produzidas em material isolante, onde componentes eletrônicos (capacitores, diodos, resistores, transistores, etc.) são montados e soldados e conectados através de trilhas de cobre gravadas na placa (Braga, 2001).

O tempo gasto na perfuração de uma placa de circuito impresso divide-se entre o tempo gasto para perfurar a placa e o tempo gasto pela broca para se movimentar entre os furos. Este último, não é pequeno se comparado à simples perfuração das placas e, por isso, é um importante elemento para o aumento da produtividade. (Ayoama et al., 2004).

O processo de perfuração durante a produção de PCBs representa um gargalo nos esforços empreendidos pelas indústrias para acompanhar o rápido crescimento da sua produção, devido a limitações tecnológicas presentes nas máquinas convencionais de perfuração e ao aumento da densidade destas placas, representada pela quantidade de furos presentes na mesma.

Para diminuir o tempo de perfuração e o gasto de energia durante esse processo, é necessário encontrar o caminho ótimo entre os furos da placa levando em consideração, além da distância cartesiana total, características particulares da CNC envolvida no processo como, por exemplo, velocidade nos eixos X e Y e o tempo perdido com acelerações e desacelerações a cada movimento. Deve-se, ainda, atentar ao tamanho e a espessura da placa.

O problema de otimização do comprimento do caminho a ser percorrido pela broca é uma variação do problema do caixeiro viajante (TSP do inglês T*raveling Salesman Problem*) (Ancău, 2008), de extrema importância dada sua grande aplicação prática e sua enorme relação com outros modelos (Goldbarg e Luna, 2000).

O TSP está na classe dos problemas NP-Completos considerados insolúveis por algoritmos polinomiais. Entretanto, uma solução aproximada do problema do caixeiro viajante pode ser determinada através da aplicação de algoritmos genéticos (GA do inglês *Genetic Algorithms*), que é "uma técnica de busca baseada numa metáfora do processo biológico de evolução natural" (Linden, 2008).

O objetivo deste trabalho é propor um GA para resolução do TSP que poderá ser aplicado no problema de otimização do processo de perfuração de placas de circuito impresso, através da aplicação desta solução em modelos de PCBs, comparando os resultados com os obtidos por outras técnicas.

Esta monografia está organizada em sete capítulos. O capítulo 1 apresenta uma breve introdução ao problema de otimização de PCBs.

No capítulo 2 encontra-se a revisão bibliográfica do TSP bem como as estratégias utilizadas para solucioná-lo.

O capítulo 3 apresenta uma breve introdução aos GA, seus componentes e estrutura e como esta técnica é utilizada na resolução do TSP.

No capítulo 4 são apresentados os três trabalhos já realizados na tentativa de solucionar o problema de otimização da perfuração de PCBs bem como o TSP.

Os detalhes da configuração do GA proposto por este trabalho para otimizar o processo de perfuração de PCBs encontram-se no capítulo 5.

No capítulo 6 encontram-se os resultados obtidos com o algoritmo proposto, comparando com os resultados obtidos por outras técnicas.

E, por fim, no capítulo 7, são apresentadas as conclusões finais deste trabalho.

2. O PROBLEMA DO CAIXEIRO VIAJANTE

O problema do caixeiro viajante pode ser descrito da seguinte forma, segundo Michalewicz (1996): um caixeiro viajante deve percorrer todas as cidades dentro de um território qualquer de modo que cada cidade seja percorrida apenas uma vez e o caixeiro viajante retorne ao seu ponto de partida ao final do percurso. Dado o custo da viagem entre uma cidade e outra, se deve determinar seu itinerário para que o custo total da viagem seja o mínimo possível.

Christofides (1979) descreve o problema do caixeiro viajante da seguinte forma: Considere um grafo $G = \{N, A\}$ completo, orientado ou não, onde N é o conjunto de n vértices e A o conjunto de m arestas. $C_{i,j}$ é o custo da aresta (i,j). Um circuito Hamiltoniano de G é um circuito passando por todos os vértices de G uma única vez. O custo de cada circuito é a soma dos custos das suas arestas. Logo, o problema do caixeiro viajante consiste em encontrar no grafo G, com uma dada matriz $[c_{i,j}]$, o circuito Hamiltoniano que apresente o menor custo.

O problema do caixeiro viajante pertence à classe dos problemas de otimização combinatória, caracterizados pelo objetivo de se maximizar, ou minimizar, uma função definida sobre um domínio finito. No caso do problema do caixeiro viajante, o objetivo é encontrar a solução cujo custo da viagem seja mínimo dentro do conjunto de todas as rotas possíveis de serem realizadas pelo caixeiro viajante.

O problema do caixeiro viajante surge em inúmeras aplicações como, conforme destacado por Goldbarg e Luna (2000): programação de operações de máquinas em manufatura, programação de transporte entre células de manufatura, otimização do movimento de ferramentas de corte, otimização de perfuração de furos em placas de circuito impressos, na maioria dos problemas de roteamento de veículos, na solução de problemas de seqüenciamento, na solução de problemas de programação e distribuição de tarefas em plantas e em trabalhos administrativos.

Como explanado anteriormente, o problema do caixeiro viajante está na classe dos problemas NP-completos. Duas propriedades definem os problemas NP-completos (Papadimitriou e Steiglitz, 1998):

- Problemas NP-completos não podem ser exatamente solucionados por nenhum algoritmo polinomial [1] conhecido, dado um problema cuja população inicial é realisticamente grande.
- Se existe uma solução para um problema NP-completo, então existem algoritmos polinomiais para todos os problemas NP-completo.

A quantidade de rotas possíveis de serem percorridas no problema do caixeiro viajante é *n!* onde *n* é a quantidade de cidades existentes na rota (Bryant, 2000). Logo, para encontrar a solução para uma rota contendo 25 cidades ($\approx 1{,}5 \times 10^{25}$ rotas) em uma máquina que identifique e calcule uma rota por nano segundo, ou seja um bilhão de rotas por segundo, levar-se-ia mais de 400 milhões de anos.

Nas próximas sessões serão apresentadas algumas soluções aproximadas para a resolução do problema do caixeiro viajante.

2.1. HEURÍSTICAS PARA O PROBLEMA DO CAIXEIRO VIAJANTE

Apesar de não possuírem uma solução passível de ser executada em tempo polinomial, existe uma série de métodos para obtenção de boas, porém não ótimas, soluções para problemas NP-completos. Estes métodos são denominados *algoritmos de aproximação* ou *algoritmos heurísticos*.

Algoritmos de aproximação não garantem a solução ótima para o problema, mas são capazes de encontrar soluções eficientes que se aproximam da solução ótima. Algoritmos de aproximação são utilizados, de acordo com Michalewicz (1996), para solucionar problemas como o caixeiro viajante, modelagem cognitiva, otimização de consultas em bancos de dados, etc.

Ganhoto (apud CHONG, 2004) classifica as heurísticas para solução do problema do caixeiro viajante em três grupos: heurísticas construtivas, heurísticas de melhoria e heurísticas compostas, detalhadas a seguir.

[1] Algoritmos cujo tempo de execução é da ordem de uma função exponencial ou fatorial. Algoritmos polinomiais são considerados rápidos enquanto algoritmos não polinomiais são considerados inaceitavelmente lentos.

2.1.2. HEURÍSTICAS CONSTRUTIVAS

As heurísticas construtivas, também conhecidas como heurísticas de construção de rotas, têm por fim construir a rota de forma gradativa, através de adições seqüenciais de cidades (Ganhoto, 2004). As mais conhecidas são o *Nearest Neighbor*, o *Greedy* e os algoritmos de Inserção Econômica, que serão descritas adiante (Ancău, 2008).

O algoritmo *Greedy*, ou "Guloso", $O(n^2 log_2(n))$, seleciona repetidamente o menor caminho entre dois pontos e o adiciona a rota até que um ciclo contendo *n* caminhos seja formado ou até que um ponto não possua mais do que dois caminhos partindo do mesmo (Nilsson, 2003). Os passos do algoritmo são descritos a seguir (Nilsson, 2003):

1. Ordene todos os caminhos do de menor custo para o de maior custo.
2. Selecione o menor caminho e adicione-o a rota se ele não violar nenhuma das restrições citadas anteriormente.
3. Verifique se a rota já possui *n* caminhos. Em caso negativo, volte para o passo 2.

O algoritmo *Nearest Neighbor*, ou "Vizinho mais próximo", $O(n^2)$, é similar ao algoritmo *greedy* na sua simplicidade. Ele constrói uma rota a partir dos seguintes passos (Rosenkrantz et al., 1977):

1. Escolha uma cidade arbitrária.
2. Encontre a cidade mais próxima à última cidade adicionada, e que ainda não esteja na rota, e adicione-a a rota conectando-as.
3. Quando todas as cidades tiverem sido adicionadas à rota, conecte a primeira cidade adicionada à última cidade adicionada.

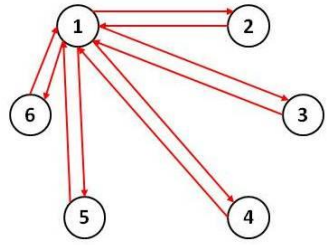
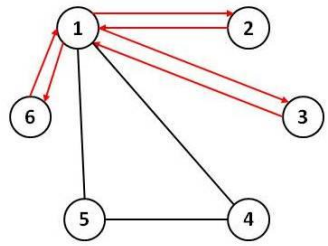
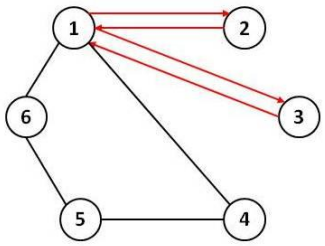
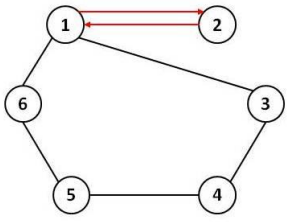
Este algoritmo é, talvez, a mais simples das heurísticas para resolução do problema do caixeiro viajante, porém nem sempre apresenta uma boa solução, pois o último percurso adicionado à rota pode ser muito grande (Nilsson, 2003) (Bryant, 2000).

A heurística de *Savings*, ou heurística das Economias ou, ainda, heurística de Clarke e Wright, $O(n^2 log(n))$, aplica ao problema do caixeiro viajante uma estratégia originalmente concebida para a resolução de problemas de roteamento (Buchberger et. al, 2009) (Goldbarg e Luna, 2000). O algoritmo é descrito por Goldbarg e Luna (2000) como segue:

1. Selecione um vértice *k*, selecionado por algum critério ou aleatoriamente.

2. Para os demais vértices do grafo, crie subrotas saindo de k_i, onde $i = 1, ..., n$, passando pelo vértice k e retornando ao vértice k_i. Ou seja, um circuito não hamiltoniano que passa n vezes pelo nó k.
3. Obtenha a lista das economias da seguinte forma: $S_{ij} = c_{jk} - c_{ij}$, onde $i,j = 1, ..., n$, onde S é a economia feita se o vértice i for ligado diretamente ao vértice j sem passar por k.
4. Ordene a lista de economias em ordem decrescente (da maior para a menor economia).
5. Percorra a lista de economias iniciando pela primeira posição.
6. Se a inserção da aresta (i,j) e a retirada da aresta (k,j) e (j,k) resultar em uma rota iniciando em k e passando pelos demais vértices, eliminar da lista.
   Caso contrário, tentar a ligação seguinte da lista.
7. Repetir o processo até que um ciclo hamiltoniano tenha sido formado.

As etapas da heurística de *Savings* podem ser visualizadas na figura 1, onde foram utilizados os seguintes custos para o cálculo das economias: $C_{2,1} = 2$, $C_{2,3} = 5$, $C_{2,4} = 9$, $C_{2,5} = 7$, $C_{2,6} = 2$, $C_{3,1} = 1$, $C_{3,4} = 3$, $C_{3,5} = 8$, $C_{3,6} = 6$, $C_{4,1} = 4$, $C_{4,5} = 2$, $C_{4,6} = 1$, $C_{5,1} = 9$, $C_{5,6} = 2$, $C_{6,1} = 1$.

<table>
<tr><td colspan="2"> 
</td><td>⇒ $S_{45} = 11$
$S_{56} = 8$
$S_{46} = 4$
$S_{25} = 4$
$S_{35} = 2$
$S_{34} = 2$
$S_{26} = 1$</td><td colspan="2">$S_{23} = -2$
$S_{36} = -4b$</td></tr>
<tr><td colspan="2">Inicialização</td><td colspan="3">Lista das Economias</td></tr>
<tr><td colspan="2"> 
</td><td>$S_{45} = 11$
⇒ $S_{56} = 8$
$S_{46} = 4$
$S_{25} = 4$
$S_{35} = 2$
$S_{34} = 2$
$S_{26} = 1$</td><td> 
</td><td>$S_{56} = 8$
⇒ $S_{46} = 4$
$S_{25} = 4$
$S_{35} = 2$
$S_{34} = 2$
$S_{26} = 1$
$S_{23} = -2$</td></tr>
<tr><td colspan="2">Ciclo decorrente de S_{45}</td><td>Economias</td><td>Ciclo decorrente de S_{56}</td><td>Economias</td></tr>
<tr><td>(Inviável)
$S_{46} = 4$
⇒ $S_{25} = 4$
$S_{35} = 2$
$S_{34} = 2$
$S_{26} = 1$
$S_{23} = -2$</td><td>(Inviável)
$S_{25} = 4$
⇒ $S_{35} = 2$
$S_{34} = 2$
$S_{26} = 1$
$S_{23} = -2$
$S_{36} = -4$</td><td>(Inviável)
$S_{35} = 2$
⇒ $S_{34} = 2$
$S_{26} = 1$
$S_{23} = -2$
$S_{36} = -4$</td><td colspan="2"> 
</td></tr>
<tr><td>Economias</td><td>Economias</td><td>Economias</td><td colspan="2">Ciclo decorrente de S_{34}</td></tr>
</table>

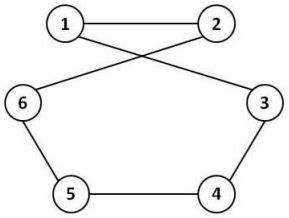
$S_{34} = 2$ $\Rightarrow S_{26} = 1$ $S_{23} = -2$ $S_{36} = -4$	
Economias	Ciclo decorrente de S_{26} C = 12 (Solução Ótima)

Figura 1 – Etapas da heurística de *Savings* (Adaptado de Goldbarg e Luna, 2000)

Heurísticas de inserção (*Initialization Heuristics*) são heurísticas simples e que apresentam uma série de variações (Nilsson, 2003). De acordo com Ganhoto (2004 apud. GOLDBARG E LUNA, 2000, p. 20), estas heurísticas "partem de um circuito inicial (normalmente utilizando três vértices) e vão selecionando e inserindo vértices, ainda não pertencentes à solução, até completar o circuito Halmitoniano". Ainda segundo Ganhoto (2004), deve-se tomar três tipos de decisão ao executar os procedimentos anteriormente citados:

- Escolher o circuito inicial.
- Escolher o vértice a ser inserido no grafo.
- Escolher a posição onde o novo vértice será inserido.

Quanto à segunda decisão, a escolha do vértice a ser inserido, deve-se, ainda, levar em consideração os seguintes critérios:

- Inserir o novo vértice o mais próximo a um dos vértices já presentes no circuito.
- Inserir o novo vértice o mais distante de um dos vértices já presentes no circuito.
- Inserir o novo vértice de modo que o circuito gerado seja o de menor custo.
- Inserir o vértice aleatoriamente.

No caso de uma inserção baseada na proximidade ou distância entre o novo vértice e o vértice já presente no circuito deve-se decidir como ele será inserido no circuito. Um novo vértice, para ser inserido no grafo, necessita de duas novas arestas. Se a inclusão do novo vértice V_j for efetuada entre os vértices V_i e V_{i+1}, pois V_j encontra-se mais próximo a V_i, será necessário remover a aresta existente entre esses dois vértices para inclusão do vértice

intermediário, de modo que a gerar as arestas $V_{i,j}$ e $V_{j,i+1}$. Neste caso, V_{i+1} deve ser escolhido de modo a minimizar $C_{i,j} + C_{j,i+1} - C_{i,i+1}$. (Ganhoto, 2004).

Os procedimentos para construção de rotas seguindo a heurística de inserção é descrita, passo a passo, a seguir:

1. Selecione um sub-grafo contendo *x* vértices selecionados a partir de algum critério.
2. Encontre um vértice V_j, não pertencente ao circuito, tal que *C(i,j)*, onde *C* é o custo/distância entre os dois pontos, seja mínimo.
3. Ache a aresta $V_{i,i+1}$ no sub-grafo tal que $C_{i,j} + C_{j,i+1} - C_{i,i+1}$ seja mínimo e, então, insira *V*j entre V_i e V_{i+1}.
4. Volte ao passo 3 até que um circuito Hamiltoniano tenha sido formado.

Uma variação da heurística de inserção denominada *Convex Hull*, ou Invólucro Convexo, $O(n^2 log(n))$, caracteriza-se pela obtenção de um invólucro convexo (figura 2) do conjunto de nós do grafo, fazendo dele o sub-grafo do passo 1 do procedimento descrito no parágrafo anterior.

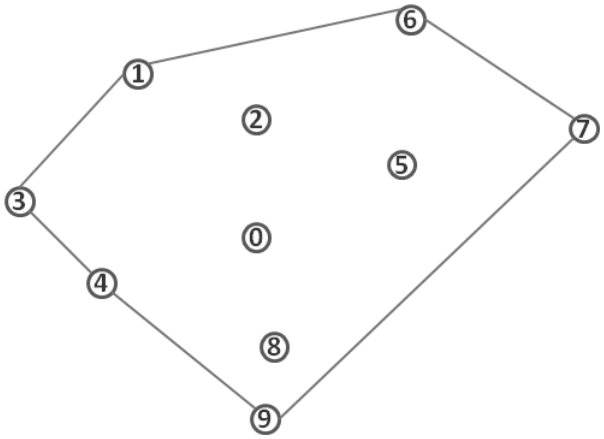


Figura 2 - Invólucro Convexo

Splinder (2010) aponta como desvantagem de heurísticas construtivas a baixa diversidade de suas soluções finais e o fato de que uma decisão equivocada tomada no início do processo de construção não pode ser corrigida.

Heurísticas construtivas, de acordo com Chong (apud Golden e Stewart, (2001)), produzem soluções dentro de 5% a 7% do ótimo e são utilizadas como ponto de partida para heurísticas de melhoria e na composição de heurísticas híbridas.

2.1.3. HEURÍSTICAS DE MELHORIA

As heurísticas de melhoria buscam, a partir de uma rota dada, melhorar o custo da rota através da permutação de cidades (Ganhoto, 2004). Incluem-se nesta categoria os algoritmos *2-Opt* e *3-Opt, kOpt* e algoritmos genéticos, dentre outros.

Os algoritmos *2-Opt*, *3-Opt* e *k-Opt* são os procedimentos de melhoria de rotas mais utilizados. Fazem parte de um conjunto de algoritmos de busca local simples, proposto inicialmente por Croes (1958), que elaborou as técnicas *2-Opt* e *3-Opt* para modificações simples de rotas, e melhorada por Lin e Kernigham (1973) que propuseram a generalização da técnica anterior através do algoritmo *k-Opt* (Ancău, 2008).

Os algoritmos de busca local do tipo *2-Opt* excluem duas arestas de uma rota R gerando dois caminhos desconectados de R, conforme figura 3. Estes caminhos são, então, reconectados de modo que uma nova rota R' seja gerada e a distância percorrida em R' seja menor do que a distância percorrida em R (Toscano Filho, 2006). As trocas são efetuadas até que mais nenhuma melhoria possa ser efetuada. Quando isto ocorre, a rota é denominada 2-ótima. O procedimento de troca é apresentado na figura abaixo.

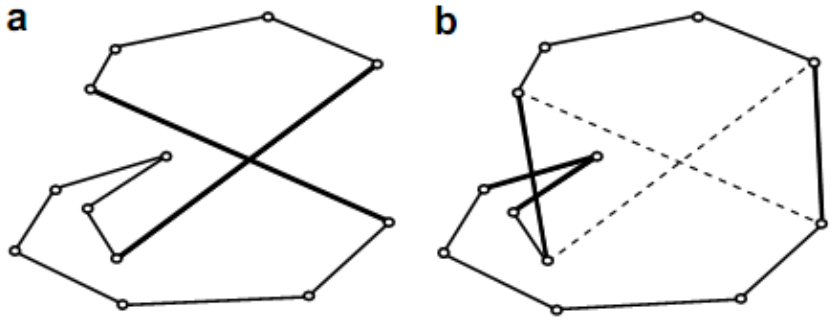


Figura 3 - Duas arestas são removidas (a) e reconectadas (b) (Ancău, 2008)

O algoritmo *3-Opt* funciona de forma semelhante ao algoritmo *2-Opt*, porém, ao invés de duas arestas, são removidas três. Se uma solução é 3-ótima então ela é uma solução 2-ótima (Nilsson, 2003).

O algoritmo *k-Opt* é mais eficiente que os algoritmos supracitados, pois ele decide a cada iteração o valor de *k* adequado para efetuar as permutas, o que o torna mais complexo, e poucos têm sido capazes de fazer melhorias utilizando-o (Nilsson, 2003).

O algoritmo *k-Opt* é de complexidade, aproximada, $O(n^{2.2})$ logo mais lento que uma simples melhoria *2-Opt* de complexidade $O(n^2)$.

Segundo Papadimitriou e Steiglitz (1998), citando estudo de Lin, o *3-Opt* gera soluções muito melhores que o *2-Opt*, mas soluções *4-Opt* não são suficientemente melhores do que as *3-Opt* de modo a justificar o tempo de computação adicional e que a probabilidade de uma rota gerada por um *3-Opt* ser ótima é de $2^{-n/10}$, onde n é o número de cidades.

2.1.4. HEURÍSTICAS COMPOSTAS

As heurísticas compostas, ou heurísticas híbridas, são compostas por elementos das duas heurísticas anteriores (Ganhoto, 2004). Neste caso, uma rota é elaborada utilizando-se uma heurística construtiva e, em seguida, a rota gerada é otimizada fazendo uso de uma heurística de melhoria. Na categoria das heurísticas compostas incluem-se, dentre outros, o algoritmo GRASP, o algoritmo CCAO e o algoritmo GENIUS. (Chong, 2001).

O algoritmo GRASP (do inglês *Greedy Randomized Adaptive Search Procedure*) é um processo interativo, onde cada iteração apresenta duas fases: uma fase de construção, onde uma solução viável é construída, e outra fase de busca local, onde as vizinhanças são investigadas em busca de uma solução ótima local (Chaves, 2005) (Mestria et al., 2010). A figura 7 apresenta um pseudo-código para o GRASP.

```
procedimento GRASP( )
1   para (Critério de parada GRASP não satisfeito) faça
2        Construção(Solução);
3        BuscaLocal(Solução);
4        AtualizarSolução(Solução, MelhorSoluçãoEncontrada);
5   fim-para
6   retorne(MelhorSoluçãoEncontrada)
fim-GRASP
```

Figura 4 - Pseudo-código para o algoritmo GRASP (Chaves, 2005)

Uma das vantagens do algoritmo GRASP é a diversificação das soluções encontradas (Goldbarg e Luna, 2000).

A heurística CCAO é uma combinação dos métodos Invólucro convexo (*Convex Hull*), Inserção mais barata (*Cheapest Insertion*), Escolha por ângulo (*Angle Selection*) e *Or-Opt*. Os três primeiros métodos são heurísticas construtivas enquanto o último é uma

heurística de melhoria. Estes métodos serão rapidamente detalhados a seguir de acordo com as definições de Chong (2001).

- **Invólucro convexo:** Escolha uma cidade k fora da rota tal que $c_{ik} + c_{kj} - c_{ij}$ é mínimo e insira k^* entre i e j de modo que $(c_{ik^*}+c_{k^*j})/c_{ij}$ é mínimo.
- **Inserção mais barata:** Escolha a cidade k que possua o menor custo dentre todas as outras cidades que ainda não estão na rota, isto é, $min(c_{ik})$ para todo k não pertencente a rota.
- **Escolha por ângulo:** Escolha a cidade k não pertencente à rota tal que o ângulo formado pelas arestas (i,k) e (k, j), onde i e j são cidades na rota, seja o maior.
- **Or-Opt:** É uma melhoria do algoritmo *3-Opt* discutido anteriormente. Consiste em remover r (r = 3, então 2) vértices na rota e tentar inserir estes vértices entre duas outras cidades tal que o tamanho da rota seja reduzido.

O algoritmo inicia com a definição de um circuito parcial formado por um invólucro convexo de vértices. Para cada vértice k não pertencente ao circuito parcial são identificadas duas cidades adjacentes i e j tal que o custo calculado por $c_{ik} + c_{kj} - c_{ij}$ seja mínimo. Selecione o vértice k de modo que o ângulo entre as arestas (i,k) e (k, j) é máximo e insira k entre i e j. A inserção mais barata e a escolha por ângulo são aplicadas até que um circuito Hamiltoniano tenha se formado. O algoritmo *Or-Opt* é, então, aplicado para otimizar a rota (Chong, 2001).

O algoritmo GENIUS foi desenvolvido em 1992 por Gendreau, Hertz e Laporte. O algoritmo é divido em duas partes denominadas GENI (construção) e US (melhoria).

A fase GENI (*Insertion Generalized*) inicia-se com uma rota parcial formada por três cidades e a cada iteração uma nova cidade, selecionada randomicamente, é adicionada a rota parcial até que um ciclo Hamiltoniano tenha se formado (Chong, 2001).

Após a fase GENI, um procedimento de melhoria é aplicado à rota construída. Esta fase é denominada US (*Unstringing and Stringing*) e consiste em remover um vértice da rota e inseri-lo de volta em outra posição (Chong, 2001). De acordo com Chong (2001), o procedimento US pode ser aplicado a qualquer rota completa. Substituindo o procedimento *Or-Opt* do algoritmo CCAO, por exemplo, pelo procedimento US, temos um novo algoritmo denominado CCAUS.

3. ALGORITMOS GENÉTICOS

Algoritmos genéticos (GA do inglês *Genetic Algorithms*) fazem parte de um conjunto de sistemas baseado em princípios da evolução e da hereditariedade denominados Programas Evolutivos (EP do inglês *Evolution Programs*). EP conservam uma população de soluções potenciais, com algum processo de seleção baseada na aptidão destes indivíduos, e alguns operadores "genéticos". (Michalewicz, 1996).

GA consistem em uma técnica de busca heurística baseada no processo biológico de seleção natural proposto por Charles Darwin em 1859 em seu livro "A Origem das Espécies" e são largamente utilizados para a resolução de problemas de otimização como roteamento, escalonamento, controle adaptativo, modelagem cognitiva, problemas de transporte, problemas do caixeiro viajante, otimização de consultas em bancos de dados, etc. (Michalewicz, 1996)

Ao contrário de outros métodos de busca, os GA mantêm um conjunto de soluções potenciais enquanto os demais métodos processam um único ponto no espaço de buscas (Michalewicz, 1996).

De acordo com Linden (2008), os GA não são métodos *hillclimbing*, ou seja, as buscas não são interrompidas quando uma boa solução é encontrada, ao contrário, os GA buscam sempre soluções cada vez melhores.

GAs utilizam termos oriundos da genética como cromossomos e genes que são as unidades que formam um cromossomo. Cada cromossomo representa uma possível solução para o problema. Um processo evolutivo executado sobre uma população de cromossomos corresponde a efetuar uma busca em um espaço de possíveis soluções (Michalewicz, 1996).

A estrutura de um algoritmo genético é descrito por Michalewicz (1996):

- Durante um iteração *t* é mantida uma população de soluções potenciais (conjunto de cromossomos), representada por P(t) = { , ..., }, onde representa uma solução.
- Cada solução é submetida a uma avaliação da sua aptidão (*fitness*) que determina o quão próxima ela é da solução ótima.
- Uma nova população será gerada para a iteração *t+1* formada pelos indivíduos mais aptos.

Alguns membros desta nova população sofrerão modificações, através de *crossover* e mutações, para formar novas soluções. *Crossover* consiste em combinar as características de dois cromossomos pais para formar dois descendentes similares através da troca de genes correspondentes entre seus pais (Michalewicz, 1996). O processo de *crossover* é demonstrado na figura 5.

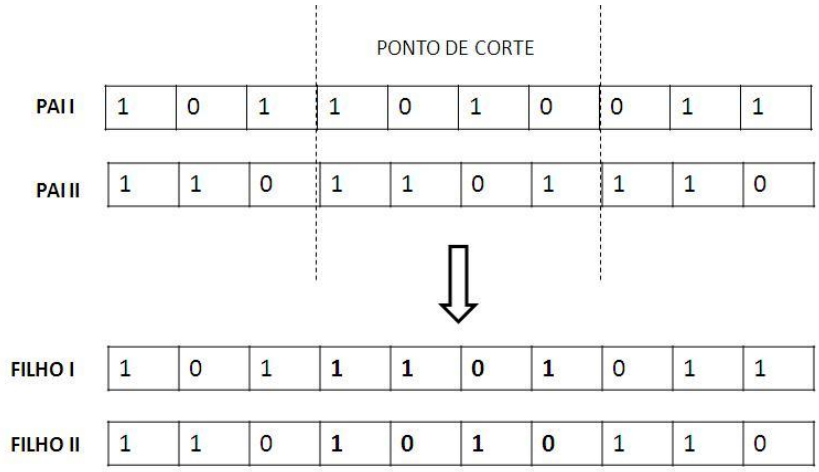


Figura 5 - *Crossover*

Já o processo de mutação altera, arbitrariamente, um ou mais genes de um determinado cromossomo com o objetivo de introduzir alguma diversidade na população.

Para Michalewicz (1996) um algoritmo genético para solucionar um problema particular deve apresentar os cinco componentes abaixo:

- Uma representação genética das possíveis soluções para o problema.
- Uma forma de criar uma população inicial de possíveis soluções.
- Uma função para avaliar o grau de aptidão de uma solução.
- Operadores genéticos que alterem a composição dos filhos (*crossover* e mutação).
- Valores para os parâmetros utilizados em algoritmos genéticos, como tamanho da população, probabilidade de *crossover*, probabilidade de mutação.

Estes componentes serão detalhados nas seções que seguem.

3.1. REPRESENTAÇÃO GENÉTICA

A representação genética ou representação cromossomial de um problema consiste em uma tradução da informação do problema de modo que a mesma possa ser tratada pelo

computador (Linden, 2008). Esta representação é denominada cromossomo e cada pedaço desta representação é chamado de gene.

Linden (2008) sugere que algumas regras sejam seguidas durante a definição de uma representação genética, apesar de deixar claro que a mesma é completamente arbitrária, ou seja, fica a critério do programador:

- A representação deve ser o mais simples possível.
- Soluções proibidas ao problema, quando houver, devem apresentar uma representação.
- Se o problema impuser quaisquer condições, estas devem estar implícitas dentro da representação.

3.2. ESCOLHA DA POPULAÇÃO INICIAL

De acordo com Linden (2008), na maioria dos trabalhos feitos na área de GA, a população inicial é selecionada aleatoriamente de modo que esta operação seja o mais simples possível. Ainda segundo Linden (2008, p. 64), "a inicialização aleatória, de forma geral, gera uma boa distribuição das soluções no espaço de busca e o uso da operação de mutação de forma eficaz garante uma boa exploração de todo o espaço de busca".

3.3. FUNÇÃO APTIDÃO

A função aptidão, ou função *fitness* ou função de avaliação, julga a qualidade da solução apresentada, ou seja, o quanto a solução se aproxima de uma solução considerada ótima. Ela faz o papel da natureza em um GA, avaliando cada solução potencial de acordo com sua aptidão para a resolução do problema. A função aptidão utiliza todos os valores armazenados em um cromossomo e retorna um valor numérico que será utilizado como métrica da qualidade da solução (Linden, 2008).

A função aptidão é o componente mais importante de um GA. Segundo Linden (2008, p. 66) ela deve embutir "todo o conhecimento que se possui sobre o problema [...], tanto suas restrições quanto seus objetivos de qualidade" devendo, portanto, refletir "os objetivos a serem alcançados na resolução de um problema".

3.4. OPERADORES GENÉTICOS

Os operadores genéticos são utilizados durante a fase de alteração do GA. Os mais conhecidos são o operador de *crossover* e o operador de mutação.

3.4.1. CROSSOVER

O operador de *crossover*, ou operador de cruzamento, tem por fim originar um ou mais cromossomos a partir de dois cromossomos pertencentes à população. Para Fonseca (2005), a idéia do *crossover* é, em linhas gerais, gerar uma nova geração de indivíduos aproveitando o que há de melhor na geração anterior. Sendo assim, as novas gerações serão formadas pelos descendentes mais aptos, ou seja, aqueles que apresentam maiores chances de ser a solução ótima para o problema.

Existe uma série de técnicas aplicadas para o operador de *crossover*, e cada uma delas se aplica melhor a uma determinada classe de problemas. A mais simples é técnica conhecida como *crossover* de um ponto. A técnica inicia-se com a seleção de dois cromossomos na população. Estes cromossomos são chamados de pais.

A seleção dos pais deve simular o mecanismo de seleção natural onde pais mais aptos tendem a gerar mais filhos, ao mesmo tempo em que pais menos aptos também geram seus descendentes. Ou seja, devemos priorizar para o cruzamento os indivíduos mais aptos sem descartar os indivíduos menos aptos. Isto porque, mesmo indivíduos com baixa aptidão podem ser portadores de um gene que possibilite a criação de um indivíduo que seja a melhor solução para o problema, sendo que este gene pode não estar presente em nenhum outro indivíduo considerado apto. Além do mais, populações formadas apenas pelos indivíduos mais aptos tendem a gerar descendentes cada vez mais semelhantes e faltará diversidade às novas populações, um fenômeno conhecido por convergência genética. (Linden, 2008).

Um método utilizado pelos pesquisadores de GA para a seleção de pais é denominado método da roleta viciada. Neste método uma roleta é criada e cada cromossomo recebe um pedaço da roleta proporcional à sua aptidão. Depois a roleta é rodada aleatoriamente e o indivíduo correspondente ao pedaço no qual ela parar será o indivíduo selecionado. A tabela abaixo apresenta o exemplo de uma roleta e foi adaptada de Linden (2008).

Indivíduo	***Aptidão***	***Pedaço da roleta (%)***

0001	1	1.61
0011	9	14.51
0100	16	25.81
0110	36	58.07
Total	*62*	*100.00*

Podemos notar pelo exemplo que o indivíduo 0110 apresenta a maior aptidão dentre os outros indivíduos e recebe 58,07% (36/52) da roleta. Sendo assim, a possibilidade deste indivíduo ser selecionado é de três em cada cinco sorteios. Em contrapartida, o indivíduo 0001, por possuir uma aptidão baixa, recebe apenas 1,61% da roleta e sua chance de ser selecionado é bem menor. Ou seja, os indivíduos mais aptos têm maior possibilidade de gerar descendentes, mas os indivíduos menos aptos também têm sua vez no processo de reprodução.

Voltando a operação de *crossover*, selecionados os pais, aplicando um método de seleção como o método da roleta viciada, uma posição denominada ponto de corte é selecionada entre dois genes de um cromossomo. Um cromossomo com *n* genes apresenta *n-1* pontos de corte.

Selecionado o ponto de corte em cada pai, os pais são separados em duas partes, uma a esquerda do ponto de corte e outra a direita do ponto de corte. O primeiro filho será formado pela concatenação da parte esquerda do primeiro pai com a parte direita do segundo pai, enquanto o segundo filho será formado pela concatenação da parte esquerda do segundo pai com a parte direita do primeiro pai.

Para exemplificar, iremos aplicar o operador de *crossover* nos cromossomos v_1 e v_2, apresentados abaixo, escolhendo o ponto de corte após o quinto gene (o ponto de corte pode ser escolhido randomicamente).

$$v_1 = (00000|0011100000000010000)$$

$$v_2 = (11100|0000011111000101)$$

Deste modo, os indivíduos v_1' e v_2' gerados a partir deste cruzamento serão:

$$v_1' = (00000|0000011111000101)$$

$$v_2' = (11100|0011100000000100000)$$

3.4.2. MUTAÇÃO

O operador de mutação altera um ou mais genes (posições no cromossomo) de um determinado cromossomo. É fundamental para um algoritmo genético, pois garante, de acordo com Linden (2008), a existência de diversidade genética na população. O operador de mutação insere novos cromossomos na população possibilitando ao GA buscar soluções fora dos limites pré-determinados pela população inicial (Linden apud VOSSE, 2008).

O operador de mutação age da seguinte forma: dada uma taxa de mutação, que determina a probabilidade deste operador atuar sobre um gene durante a execução do GA, é sorteado um número entre 0 e 1. Se o número sorteado for menor ou igual à taxa de mutação, então o operador atua sobre o gene selecionado, alterado aleatoriamente o seu valor. O processo é então repetido para cada gene.

A taxa de mutação varia de problema para problema. Em geral, ela não deve ser muito baixa, ou a população não se diversificará o que levará a uma estagnação rápida do GA, no entanto não deve ser muito alta ou o GA se assemelhará a técnica *random walk*[2]. Em geral, a taxa de mutação deve variar de acordo com a execução do GA (Linden, 2008).

3.5. VALORES PARA OS PARÂMETROS

A efetividade de um GA está intimamente ligada aos valores dos parâmetros utilizados durante sua execução. Segundo Linden (2008, p. 134), os GAs constituem "um processo dinâmico, que evolui no tempo e no espaço". Desta forma, definir os valores de cada parâmetro antes do início da execução de um GA, e mantê-lo durante todas as fases de execução, não é uma decisão adequada, posto que as características da população mudam em cada fase dentro do espaço de soluções.

Uma idéia razoável para determinar estes valores para Linden (2008) é usar técnicas de adaptação destes valores de modos que eles mudem de acordo com o progresso do GA.

[2] A técnica *random walk* determina a solução para um problema de forma aleatória sem levar em consideração qualquer informação passada.

Existem três técnicas principais de adaptação dos valores dos parâmetros de acordo com Linden (apud Hinterding, (2008)), são elas:

- **Determinística**: Os valores dos parâmetros mudam seguindo uma regra determinística, ou seja, sem nenhum *feedback* do GA. Por exemplo, aumentar a taxa de mutação em 0,01% a cada nova geração.
- **Adaptativa**: Os valores dos parâmetros mudam a partir de um *feedback* dado pela geração anterior. Um exemplo de técnica adaptativa é a regra de 1/5 de Rechenberg, que diz que se mais de 1/5 das mutações gerarem indivíduos mais aptos do que seus pais, a taxa de mutação deve aumentar. Em contraponto, se menos de 1/5 das mutações gerarem indivíduos mais aptos do que seus pais, a taxa de mutação deve ser diminuída.
- **Auto-Adaptativa**: Os valores dos parâmetros são definidos dentro do cromossomo e evoluem de acordo com a evolução da solução do problema.

3.6. ALGORITMOS GENÉTICOS E O PROBLEMA DO CAIXEIRO VIAJANTE

Michalewicz (1996) assume que o problema do caixeiro viajante apresenta uma função aptidão extremamente simples e natural que ele descreve como: para cada solução potencial, que consiste em uma permutação de *n* cidades, pode-se localizar, sobre uma tabela contendo as distâncias entre todas as cidades, o tamanho total da rota. Portanto, em uma população contendo inúmeras rotas é simples comparar duas delas. Apesar da simplicidade da função aptidão, escolher uma representação genética para o problema do caixeiro viajante está longe der uma solução tão simples.

Segundo Michalewicz (1996) há um consenso entre os pesquisadores de GA de que uma representação genética binária das rotas não é a melhor escolha para o problema do caixeiro viajante. Uma representação binária exigiria algoritmos especiais de reparação visto que a alteração de um único *bit* poderia resultar em uma rota inviável.

Existe uma série de formas possíveis de representação genética do problema do caixeiro viajante, sendo as mais conhecidas, de acordo com Goldbarg e Luna (2000): seqüência de inteiros (*path*), lista de adjacências (*adjacency*) e lista ordinal (*ordinal*). Estas representações serão discutidas com maiores detalhes adiante. Cada uma dessas operações, com exceção da lista ordinal, possui seus próprios operadores genéticos.

A mais simples das representações genéticas para o problema do caixeiro viajante é a

sequência de inteiros, também conhecida como representação por caminhos. Nesta representação um cromossomo é definido com um vetor contendo a sequência de cidades em uma rota, como, por exemplo, o cromossomo $c_1 = (1, 4, 2, 3, 6, 5)$ corresponde à rota da figura 6.

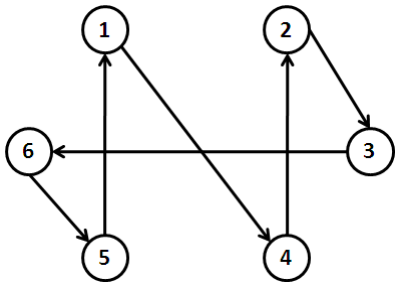


Figura 6 - Solução para PCV (adaptado de Goldbarg e Luna (2000))

O problema deste tipo de representação, de acordo com Goldbarg e Luna (2000) é a inviabilidade de alguns ciclos produzidos pela operação de *crossover*. Como, por exemplo, para os pais p_1 e p_2 definidos abaixo, os filhos f_1 e f_2 gerados refletem rotas inviáveis:

$$\left.\begin{array}{l} p_1 = (1, 2, 3, \mid 4, 5, 6) \\ p_2 = (4, 3, 2, \mid 5, 1, 6) \end{array}\right\} \Rightarrow f_1 = (1, 2, 3, 5, 1, 6) \text{ e } f_2 = (4, 3, 2, 4, 5, 6)$$

Pode-se notar que na rota representada pelo cromossomo f_1, o caixeiro passaria pela cidade 1 duas vezes e não passaria pela cidade 4, enquanto que na rota representada pelo cromossomo f_2, o caixeiro passaria pela cidade 5 duas vezes e não passaria pela cidade 1, logo, as rotas geradas descaracterizam o problema do caixeiro viajante.

Para evitar este problema, foram criados operadores especiais de *crossover* para este tipo de representação, são eles: *Partially-Mapped (PMX)*, *Order (OX)* e o *Cycle (CX)* (Michalewicz, 1996) (Goldbarg e Luna, 2000).

Outra representação genética para o problema do caixeiro viajante é chamada lista de adjacências. Esta representação consiste em um vetor representando uma rota contendo *n* cidades. Uma cidade *j* ocupa uma posição *i* se e somente se, a rota conduza da cidade *i* para a cidade *j*. Por exemplo, para o cromossomo representado por:

$$c = (2\ 4\ 8\ 3\ 9\ 7\ 1\ 5\ 6)$$

Temos a rota:

$1 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 8 \rightarrow 5 \rightarrow 9 \rightarrow 6 \rightarrow 7$

A lista de adjacências também não suporta a operação de *crossover* clássica discutida na seção 3.4.1. Três operadores de *crossover* para representações da lista de adjacências são: *alternating edges*, *subtour chunks* e *heuristics* (Michalewicz, 1996).

Por fim, tem-se a representação denominada lista ordinal, onde a rota "é representada por sua posição em uma lista ordenada de cidades" (Goldbarg e Luna, 2000, p. 427). A lista *l* varia de 1 a *n* cidades, enquanto o cromossomo é um vetor que denota a posição da cidade em *l*. Por exemplo, assuma *l = (1 2 3 4 5 6 7 8 9)*, ou seja, a rota possui 9 cidades, e o cromossomo *c = (1 1 2 1 4 1 3 1 1)*, que representa a seguinte rota *r*:

$c_1 = 1 \Rightarrow$ primeira posição na lista *l = (1 2 3 4 5 6 7 8 9)*, ou seja, $r_1 = 1$, *r = {1}*;

$c_2 = 1 \Rightarrow$ primeira posição na lista atualizada *l = (* 2 3 4 5 6 7 8 9)*, onde os valores anulados não são computados, ou seja, *l = (2 3 4 5 6 7 8 9)*, e $r_2 = 1$, o que leva a *r = {1, 2}*;

$c_3 = 2 \Rightarrow$ segunda posição na lista atualizada *l = (* * 3 4 5 6 7 8 9)*, onde os valores anulados não são computados, ou seja, *l = (3 4 5 6 7 8 9)*, e $r_3 = 4$, o que leva a *r = {1, 2, 4}*;

$c_4 = 1 \Rightarrow$ primeira posição na lista atualizada *l = (* * 3 * 5 6 7 8 9)*, onde os valores anulados não são computados, ou seja, *l = (3 5 6 7 8 9)*, e $r_4 = 3$, o que leva a *r = {1, 2, 4, 3}*;

$c_5 = 4 \Rightarrow$ quarta posição na lista atualizada *l = (* * * * 5 6 7 8 9)*, onde os valores anulados não são computados, ou seja, *l = (5 6 7 8 9)*, e $r_5 = 8$, o que leva a *r = {1, 2, 4, 3, 8}*;

$c_6 = 1 \Rightarrow$ primeira posição na lista atualizada *l = (* * * * 5 6 7 * 9)*, onde os valores anulados não são computados, ou seja, *l = (5 6 7 8 9)*, e $r_6 = 5$, o que leva a *r = {1, 2, 4, 3, 8, 5}*;

$c_7 = 3 \Rightarrow$ terceira posição na lista atualizada *l = (* * * * * 6 7 * 9)*, onde os valores anulados não são computados, ou seja, *l = (6 7 9)*, e $r_7 = 9$, o que leva a *r = {1, 2, 4, 3, 8, 5, 9}*;

$c_8 = 1 \Rightarrow$ primeira posição na lista atualizada *l* = (* * * * * 6 7 * *), onde os valores anulados não são computados, ou seja, *l = (6 7)*, e $r_8 = 6$, o que leva a *r = {1, 2, 4, 3, 8, 5, 9, 6}*;

$c_9 = 1 \Rightarrow$ primeira posição na lista atualizada *l = (* * * * * * 7 * *)*, onde os valores anulados não são computados, ou seja, *l = (7)*, e $r_9 = 7$, o que leva a *r = {1, 2, 4, 3, 8, 5, 9, 6,*

7};

Ao contrário das demais representações citadas, a representação ordinal é compatível com os operadores clássicos de *crossover*. Por exemplo:

$l = (1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9)$

$p_1 = (1\ 1\ 2\ 1 \mid 4\ 1\ 3\ 1\ 1)$ \

$\Rightarrow f_1 = (1\ 1\ 2\ 1\ 5\ 3\ 3\ 2\ 1)$ *e* $f_2 = (5\ 1\ 5\ 5\ 4\ 1\ 3\ 1\ 1)$

$p_2 = (5\ 1\ 5\ 5 \mid 5\ 3\ 3\ 2\ 1)$ /

No exemplo acima, os pais p1 e p2, que representam, respectivamente, as rotas *{1, 2, 4, 3, 8, 5, 9, 6, 7}* e *{5, 1, 7, 8, 9, 4, 6, 3, 2}*, geram os descendentes *f1* e *f2* que, por sua vez, correspondem, respectivamente, aos ciclos, *{1, 2, 4, 3, 9, 7, 8, 6, 5}* e *{5, 1, 7, 8, 6, 2, 9, 3, 4}*.

4. TRABALHOS RELACIONADOS

A seguir serão apresentados alguns trabalhos já realizados com o intuito de solucionar o problema de otimização da perfuração de PCBs, bem como para obter uma solução aproximada problema do caixeiro viajante através de um algoritmo genético híbrido.

4.1. A HEURÍSTICA HÍBRIDA DE ANCAU

Ancău (2008) propôs uma solução heurística híbrida para o problema do caixeiro viajante para otimizar a perfuração de placas de circuito impresso baseada no algoritmo de Inicialização Heurística de construção de rota e no algoritmo de otimização de rotas *kOpt*.

A estratégia de Ancău (2008) consiste, basicamente, nas seguintes etapas:

a) Gerar, aleatoriamente, uma solução S viável.
b) Tentar encontrar uma solução S*n* viável a partir de transformações sob a solução S.
c) Se uma solução melhor tive sido encontrada, ou seja, distância (S*n*) < distância (S), substitui-se S por S*n* e retomam-se as buscas a partir da etapa *b*.
d) Se nenhuma solução melhor for encontrada então S é a solução ótima. A busca pode ser interrompida até a capacidade máxima de processamento ser atingida.

Durante a iteração são construídas as rotas. O processo de construção de rotas consiste na escolha aleatória de três pontos para compor inicialmente uma sub-rota enquanto os demais são incluídos na rota, um por um, em uma posição baseada no critério de aumento mínimo do custo, ou seja, baseada na distância mínima entre o ponto e a sub-rota (Ancău, 2008). O processo segue até que o caminho Hamiltoniano tenha sido formado.

Dando prosseguimento ao procedimento supracitado, Ancău (2008) aplica os algoritmos *2Opt*, *3Opt* e *4Opt* para melhoria da rota construída.

4.2. TSP NA PRODUÇÃO DE PLACAS DE CIRCUITO IMPRESSO POR REINELT

Reinelt (1994) empregou, satisfatoriamente, uma solução aproximada para o problema do caixeiro viajante na perfuração de placas de circuito impresso.

Em geral, placas de circuito impresso apresentam furos de diversos tamanhos, logo, a broca deve ser trocada. Na aplicação de Reinelt (1994) as brocas não podem ser trocadas em qualquer posição, ela deve ser movida para a origem para que a troca seja efetuada. O tempo de troca das brocas deve ser considerado a menos que sejam perfurados todos os furos de mesmo diâmetro antes de cada troca.

O problema a ser resolvido, segundo Reinelt (1994), é encontrar a sequência de furos a serem perfurados de modo que o tempo total da produção de placas de circuito impresso seja mínimo. Seguindo a observação supracitada, o problema foi decomposto em um conjunto de subproblemas para cada diâmetro. Desta forma, após todos os furos do diâmetro selecionado terem sido perfurados, a máquina retorna a origem, efetua a troca da broca, e inicia a próxima sequência de perfuração. O tempo de perfuração não é levado em consideração. De acordo com Reinelt (1994), o tempo de produção só sofrerá decréscimo se o tempo de posicionamento, isto é, o tempo necessário para a máquina alcançar a posição dos furos, for reduzido. Isto equivale a resolver o problema do caixeiro viajante no grafo completo de $n + 1$ nós, onde n furos devem ser perfurados (o nó $n + 1$ representa a origem).

A distância entre dois nós i e j corresponde ao tempo que a máquina leva para sair da posição i e alcançar a posição j. Na prática, não é possível definir exatamente o tempo de posicionamento da máquina, pois o posicionamento da máquina depende de três fases (Reinelt, 1994): acelerar a máquina, acelerar na velocidade máxima, desacelerar até parar completamente. Para pequenas distâncias, a velocidade máxima não pode ser alcançada o que leva a sensação que posições mais distantes podem ser alcançadas mais rapidamente do que posições próximas. Mesmo que uma função de tempo seja elaborada ela pode não ser tão precisa, ou tão complicada, que pode-se levar muito tempo para executar problemas muitos grandes. Sendo assim, devem-se determinar aproximações razoáveis do tempo de movimentação real.

Para solucionar o problema, Reinelt (1994) executou heurísticas elaboradas, a saber, uma implementação do algoritmo *k-Opt* aplicado a uma rota obtida através do algoritmo *nearest neighbor*. Os parâmetros foram escolhidos a partir de experiências computacionais obtidas anteriormente pelo autor e do tamanho do problema.

4.3. UMA HEURÍSTICA GENÉTICA HÍBRIDA APLICADA AO TSP POR JAYALAKSHMI ET AL.

Jayalakshmi et al. (2001) propôs três novas heurísticas para o problema do caixeiro viajante: *Initialization Heuristics* (IH), *RemoveSharp* e *LocalOpt*. Uma heurística genética híbrida (HGA do inglês *Hybrid Genetic Algorithm*) foi então elaborada através da combinação de um operador de *crossover* e estas três heurísticas.

O HGA funciona através dos seguintes passos:

PASSO 1:

- Inicialize parte da população utilizado o algoritmo IH.
- Inicialize parte da população aleatoriamente.

PASSO 2:

- Aplique o algoritmo *RemoveSharp* em toda a população inicial.
- Aplique o algoritmo *LocalOpt* em toda a população inicial.

PASSO 3:

- Selecione randomicamente dois pais.
- Aplique o *crossover* entre os pais e gere um descendente.
- Aplique o algoritmo *RemoveSharp* no descendente.
- Aplique o algoritmo *LocalOpt* no descendente.
- Se CustoRota(descendente) < CustoRota(qualquer um dos pais) substitua o pai de maior custo pelo descendente, caso contrário descarte o descendente.

PASSO 4:

- Embaralhe uma das rotas da população selecionada randomicamente.

PASSO 5:

- Repita os passos 3 e 4 até um determinado número de iterações.

O operador de *crossover* escolhido por Jayalakshmi et al. (2001) é uma adaptação do operador conhecido como recombinação de arestas (ER do inglês *Edge Recombination*). Este

operador constrói um descendente baseando-se exclusivamente nas arestas entre os nós, não levando em consideração a ordenação relativa entre eles (Linden, 2006). O funcionamento do algoritmo é descrito por Linden (2006), de forma geral, pelo seguinte pseudo-código:

1. Monte a lista de arestas existentes em cada um dos pais. Este estrutura é denominada mapa de arestas (*edge map*).
2. Escolha randomicamente, ou de acordo com algum critério do passo 3, o nó inicial de um dos pais. O nó escolhido é denominado nó corrente.
3. Escolha uma dentre as arestas válidas para o nó corrente, observando as seguintes recomendações:
    - Escolha o nó com menor número de arestas válidas.
    - Se houver um empate, escolha um dos nós aleatoriamente.
    - Se não houver arestas válidas para o nó escolhido, escolha qualquer aresta aleatoriamente.
4. Repita o processo até que não haja mais nós a escolher.

Um exemplo, extraído de Linden (2006): considere os cromossomos P_1 = (1, 2, 4, 5, 6, 7, 3) e P_2 = (4, 5, 7, 2, 6, 1, 3). O primeiro passo é a montagem do mapa de arestas, formado pelas arestas adjacentes a cada nó. Logo, se em P_1, o nó 1 é seguido pelo nó 2 e antecedido pelo nó 3 (visto que a rota é circular) e em P_2, o nó 1 é precedido pelo nó 6 e sucedido pelo nó 3 (que já foi listado), então as arestas adjacentes ao nó 1 são os nós 2, 3 e 6. Partindo deste princípio, teremos o seguinte mapa de arestas.

1 têm arestas para os nós 2, 3 e 6.
2 têm arestas para os nós 1, 4, 6 e 7.
3 têm arestas para os nós 1, 4 e 7.
4 têm arestas para os nós 2, 3 e 5.
5 têm arestas para os nós 4, 6 e 7.
6 têm arestas para os nós 1, 2, 5 e 7.
7 têm arestas para os nós 2, 3, 5 e 6.

O cromossomo filho será inicializado com um dos dois nós iniciais dos cromossomos pais. Como os nós 1 e 4, respectivamente nó inicial de P_1 e P_2, possuem a mesma quantidade de arestas (três arestas), o nó 1 é escolhido randomicamente. O mapa de arestas do nó 1 indica que 2, 3 e 6 são os nós candidatos à próximo nó do cromossomo filho. Como os nós 2, 3 e 6 possuem, respectivamente 3, 2 e 3 arestas (eliminamos o 1 que já foi selecionado do mapa de arestas), escolhemos como próxima cidade do cromossomo filho o nó 3, pois este possui a menor quantidade de arestas. O mapa de arestas é então atualizado, eliminando os nós 1 e 3, pois estes não podem mais serem escolhidos, e se torna então:

Nó 2: 4, 6 e 7.
Nó 3: *4 e 7. (nó corrente)*
Nó 4: 2 e 5.
Nó 5: 4, 6 e 7.
Nó 6: 2, 5 e 7.
Nó 7: 2, 5 e 6.

Agora, a escolha para o próximo nó recai entre os nós 4 e 7, que pertencem a lista de arestas adjacentes de 3, que é o nó corrente. Eles possuem respectivamente 2 e 3 nós, logo o nó 4 é escolhido. O cromossomo filho é dado, até o momento, por (1, 3, 4, x, x, x, x), onde x representa as posições ainda não definidas. Tem-se, então, o seguinte mapa de arestas:

Nó 2: 6 e 7.
Nó 4: *2 e 5. (nó corrente)*
Nó 5: 6 e 7.
Nó 6: 2, 5 e 7.
Nó 7: 2, 5 e 6.

O próximo nó deve ser escolhido entre os nós 2 e 5, que são os nós adjacentes ao nós corrente 4. Como ambos possuem a mesma quantidade de arestas, o nó 2 é, então, escolhido randomicamente e tem-se o cromossomo filho (1, 3, 4, 2, x, x, x) e o mapa de arestas:

Nó 2: *6 e 7. (nó corrente)*
Nó 5: 6 e 7.
Nó 6: 5 e 7.
Nó 7: 5 e 6.

Novamente a escolha é randômica entre os nós 6 e 7. Escolhendo o nó 6, tem-se o cromossomo (1, 3, 4, 2, 6, x, x) e o seguinte mapa de arestas:

Nó 5: 7.
Nó 6: *5 e 7. (nó corrente)*
Nó 7: 5.

Escolhendo novamente de forma randômica o nó 5, restará apenas o nó 7 que finalizará o cromossomo filho, dado por (1, 3, 4, 2, 6, 5, 7).

A única diferença entre o operador de Jalayakshmi et al. (2001) e o ER original é que no passo 3, enquanto o ER seleciona o nó com a menor quantidade de arestas, de acordo com o mapa de arestas, o operador de Jalayakshmi et al (2001) seleciona o nó mais próximo do nó corrente.

Para inicializar parte da população, Jalayakshmi et al (2001) utilizou um algoritmo guloso de construção de rotas, denominado *Initialization Heuristcs*, que nada mais é do que uma variação do algoritmo de Inicialização Heurística descrito no capítulo 2.1.2, e descrito a seguir:

1. Crie duas listas para representar a rota. A primeira, denominada *Input List*, deve conter todas as cidades da rota, enquanto a segunda, denominada *Output List*, será preenchida no decorrer da execução do algoritmo.
2. Selecione quatro cidades, a primeira com a maior coordenada *x*, a segunda com a menor coordenada *x*, a terceira com a maior coordenada *y* e a quarta com a menor coordenada *y*. Mova-as da *Input List* para a *Output List*.
3. Dentre todas as sequências possíveis formadas pelas quatro cidades, encontre a sequência com o menor custo e troque a sequência contida na *Output List* pela sequência de menor custo.
4. Embaralhe os elementos da *Input List*.
5. Remova o primeiro elemento da *Input List* e insira-o na *Output List* na posição que represente o menor incremento de custo na rota.
6. Repita o passo 5 até que todos os elementos da *Input List* sejam movidos para a *Output List*.

Conforme explanado anteriormente, à população inicial e a cada cromossomo filho gerado são aplicados, seguidamente, os algoritmo de melhoria de rotas *RemoveSharp* e *LocalOpt*.

O algoritmo *RemoveSharp* funciona da seguinte forma:

1. Uma lista (NEARLIST) contendo as *m* cidades mais próxima da cidade selecionada é criada.
2. A cidade selecionada é removida da rota e uma nova rota com *N-1* cidades é formada.
3. A cidade selecionada é então reinserida na rota antes ou depois de qualquer uma das cidades presentes na NEARLIST e o custo da nova rota é calculado para cada caso.
4. A rota de menor custo é então selecionada.
5. Os passos são repetidos para cada cidade presente na rota.

A figura 7 apresenta um exemplo de uma rota que apresenta uma cidade mal posicionada, a cidade 5, enquanto a figura 8 apresenta a rota otimizada pelo algoritmo *RemoveSharp* com a realocação da cidade 5 entre as cidades 0 e 1 mais próximas.

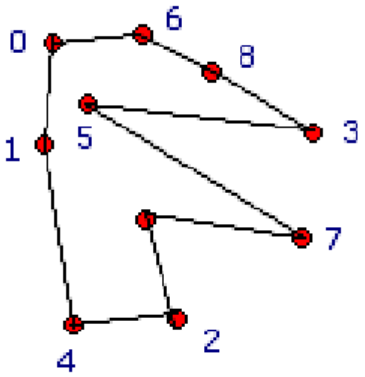


Figura 7 - Rota com cidade mal posicionada (Jayalakshmi et al., 2001).

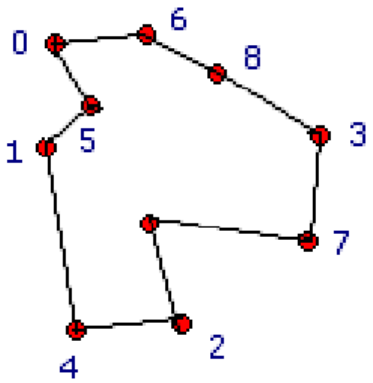


Figura 8 - Rota otimizada pelo algoritmo *RemoveSharp* (Jayalakshmi et al., 2001).

O algoritmo *LocalOpt* consiste em selecionar q cidades consecutivas (S_{p+0}, S_{p+1}, ..., S_{p+q-1}) da rota e reorganizar as cidades S_{p+1}, S_{p+2}, ..., S_{p+q-2}, de modo que a distância entre as cidades S_{p+0} e S_{p+q-1} seja mínimo através da busca por todos os arranjos possíveis. O valor de p varia de 0 para $n-q$, onde n é o total de cidades presentes na rota. Abaixo, as figuras 9 e 10 apresentam o exemplo da aplicação do algoritmo em uma rota, cuja distância entre as cidades 1 e 6 é diminuída.

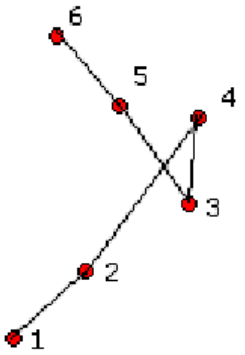


Figura 9 - Uma rota ruim (Jayalakshmi et al., 2001).

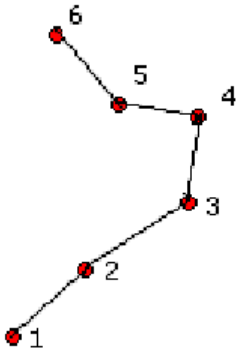


Figura 10 - Uma rota otimizada pelo algoritmo *LocalOpt* (Jayalakshmi et al., 2001).

5. HEURÍSTICA GENÉTICA HÍBRIDA PARA RESOLUÇÃO DO PROBLEMA DA PERFURAÇÃO DE PLACAS DE CIRCUITO IMPRESSO

Ancău (2008) e Reinelt (1994) demonstraram que o problema de perfuração de placas de circuito impresso (PCBs) nada mais é do que uma aproximação do problema do caixeiro viajante. Logo, eliminando-se algumas particularidades do problema de perfuração de PCBs, é possível solucioná-lo satisfatoriamente obtendo-se uma solução aproximada do problema do caixeiro viajante.

Sendo assim, este trabalho se propõe a otimizar parte do processo de perfuração de placas de circuito impresso obtendo uma solução aproximada do problema do caixeiro viajante e, para tal, se baseará na heurística genética híbrida proposta por Jayalakshmi et al.(2001), devido a eficácia demonstrada pela mesma e a simplicidade da sua implementação. A heurística apresentada na seção 4.3 foram efetuadas às seguintes alterações:

- Inclusão da heurística de *Savings* na inicialização da população, em detrimento da *Initialization Heuristics*.
- Preservação das "subrotas comuns" no operador de *crossover Edge Recombination*.
- Controle da população pela similaridade das rotas, além do custo das mesmas.

Para reduzir o problema de perfuração de placas de circuito impresso a um problema do caixeiro viajante faz-se necessário, inicialmente, remover as especificidades do primeiro problema. A primeira especificidade do problema de perfuração de PCBs é a diferença entre os diâmetros dos furos presentes em uma placa e, conseqüentemente, o tempo gasto para efetuar a troca da broca. Esta especificidade será eliminada decompondo o problema em um conjunto de subproblemas para cada diâmetro, como proposto por Reinelt (1994). Logo, sugere-se a geração de uma rota para cada subproblema, sendo o custo total do problema a soma dos custos de cada subproblema.

A segunda especificidade envolve a mecânica das máquinas de perfuração: as velocidades obtidas pela máquina nos eixos X e Y e o tempo entre as acelerações e desacelerações da máquina. Esta especificidade poderá ser encapsulada na função aptidão do algoritmo genético e será detalhado mais a seguir. A terceira e última especificidade consiste no tempo levado para efetuar um furo na placa. Entretanto, este tempo é tão pequeno se

comparado ao tempo que a broca gasta para percorrer a placa que o mesmo não será levado em consideração.

Eliminadas as particularidades do problema de perfuração de PCBs, os esforços ficam voltados à resolução do problema do caixeiro viajante. A solução adotada consiste em uma heurística híbrida, utilizando heurísticas de construção e melhoria combinadas a um algoritmo genético, assumindo como representação genética a lista ordinal descrita no capítulo 3.6, por ser a forma mais natural e simples de representar o problema do caixeiro viajante. A implementação da heurística será detalhada a seguir.

5.5. ALGORITMO FINAL

O HGA proposto por este trabalho configurar-se-á da seguinte forma:

PASSO 1:

- Inicialize 50% da população através da Heurística de *Savings*.
- Inicialize 50% da população aleatoriamente.

PASSO 2:

- Aplique o algoritmo *RemoveSharp* em toda a população inicial.
- Aplique o algoritmo *LocalOpt* em toda a população inicial.

PASSO 3:

- Selecione randomicamente dois pais.
- Aplique o operador de *crossover edge recombination* entre os pais e gere um descendente.
- Aplique o algoritmo *RemoveSharp* no descendente.
- Aplique o algoritmo *LocalOpt* no descendente.
- Se CustoRota(descendente) < CustoRota(qualquer um dos pais):
    - Se o descendente possuir a melhor rota dentre todas as rotas existentes na população, substitua pelo descendente o pai que mais se assemelhe ao mesmo.
    - Senão substitua pelo descendente o pai com o pior custo.

PASSO 4:

- Selecione randomicamente um número entre 0 e 1. Se o número selecionado for menor do que a taxa de mutação, aplique o operador de mutação a um cromossomo da população selecionado randomicamente.

PASSO 5:

- Repita os passos 3 e 4 até um determinado número de iterações.

A seguir, serão discutidas as escolhas efetuadas para esta configuração.

5.5.1. POPULAÇÃO INICIAL

Segundo Barcellos (2000), uma das principais vantagens de um GA consiste em sua capacidade de processar simultaneamente vários elementos hiper-planos no espaço de busca, chamado "Paralelismo Implícito". Isto se torna possível devido à capacidade do GA de trabalhar com pontos distintos.

Logo, quanto maior a variabilidade genética dos cromossomos, ou seja, quanto maior a diversidade de possíveis soluções, dentre boas e ruins, maior a capacidade do GA em obter soluções ótimas. Caso a população seja idêntica ou muito parecida, se o ponto de mínimo global estiver distante desta população, o tempo de convergência do GA será muito grande (Barcellos, 2000).

Normalmente, a variabilidade máxima é alcançada no início da execução do GA, quando há uma geração aleatória dos cromossomos que integrarão a população inicial. No entanto, gerando a população inicial aleatoriamente, obtêm-se uma população heterogênea, porém muito distante do ponto mínimo global, o que afetará negativamente o tempo de convergência do GA.

Logo, para aumentar a variabilidade da população de modo a não afetar negativamente o tempo de convergência, apenas parte da população foi gerada aleatoriamente, 50%. O restante da população foi inicializado por intermédio da heurística construtiva de *Savings*, descrita no capítulo 2.1.2, cujo objetivo é aproximar parte da população inicial do ponto mínimo global, diminuindo assim o tempo de convergência.

5.5.2. CROSSOVER

O operador de *crossover* escolhido foi o *Edge Recombination* (ER), efetuando-se a mesma adaptação proposta por Jayalakshmi et al.(2001), conforme descrito no capítulo anterior.

Entretanto, mais uma modificação, proposta por Michalewicz (1996 apud. STARKWEATHER ET AL., 1991), foi adicionada ao operador. O autor propõe a preservação das subrotas comuns aos dois pais no descendente. Considere o seguinte exemplo: para os cromossomos P_1 = (1, 2, 3, 4, 5, 6, 7, 8, 9) e P_2 = (4, 1, 2, 8, 7, 6, 9, 3, 5), o mapa de arestas terá a entrada abaixo:

Nó 4: 3, 5 e 1.

Esta entrada no mapa indica que existe uma subrota que está presente tanto em P1 quanto em P2, a subrota (4,5), enquanto as subrotas (4,3) e (4,1), estão presentes, apenas nos pais P1 e P2, respectivamente. A proposta de Michalewicz(1996) é adicionar a estrutura do mapa de adjacências uma *flag* nos nós para indicar a ocorrência de um nó adjacente ao nó corrente em ambas as rotas representadas pelos respectivos pais. No exemplo anterior, teríamos no mapa de arestas a seguinte entrada:

Nó 4: 3, - 5 e 1.

Neste caso o caractere '-', que antecede o nó 5, indica que este nó é adjacente ao nó 4 em ambas as rotas representadas por P1 e P2. Este nó será, então, priorizado durante a geração do descendente, em detrimento do nó mais próximo do nó corrente.

5.5.3. MUTAÇÃO

Assim como Fonseca (2005), foram implementados três operadores de mutação: inversão, troca recíproca e heurística.

A inversão, ou *invertion mutation*, consiste em selecionar duas posições aleatórias de um cromossomo e inverter as posições dos genes presentes na partição entre elas, conforme mostrado na figura 15.

Já a troca recíproca, ou *reciprocal exchange mutation*, consiste em efetuar uma troca de posições entre dois genes selecionados aleatoriamente, como pode ser visto na figura 16.

Por fim, para a mutação heurística, ou *heuristic mutation*, um cromossomo é submetido a uma heurística qualquer de melhoria. A heurística selecionada neste trabalho foi à heurística *kOpt*, descrita anteriormente no capítulo 2.1.3.

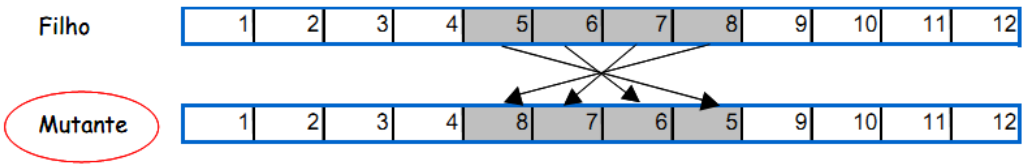


Figura 11 - Inversão (Fonseca, 2005).

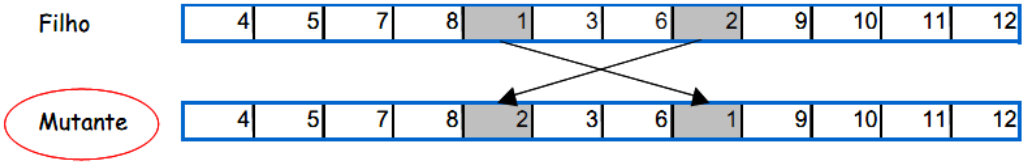


Figura 12 - Troca recíproca (Fonseca, 2005)

Entretanto, como nenhuma das heurísticas anteriormente citadas apresentou resultados relevantes quando comparadas à técnica de *shuffling*, proposta por Jayalakshmi et al.(2001), que consiste em embaralhar uma rota, esta última foi mantida no algoritmo.

5.5.4. CONTROLE DA POPULAÇÃO

Para simplificar o algoritmo, o tamanho da população se manterá constante durante cada iteração do GA. Jayalakshmi et al.(2001) propôs para isto que os descendentes mais aptos do que qualquer um dos seus respectivos pais fosse adicionado à população em detrimento do pai menos apto, que seria então eliminado. Caso os descendentes fossem menos aptos do que qualquer um dos seus pais, o mesmo seria descartado.

Esta técnica visaria, de acordo com Jayalakshmi et al.(2001), além de simplificar o algoritmo e diminuir o seu tempo de convergência, preservar as melhores rotas, visto que as melhores rotas da população em uma iteração só seriam substituídas se do seu cruzamento

resultasse um descendente mais apto, garantindo que o desempenho do GA não decrescesse com o decorrer das gerações.

No entanto esta técnica não contempla um cenário que pode diminuir o desempenho do GA ao possibilitar a perda da diversidade da população, pois leva em consideração apenas a função aptidão de cada cromossomo. Sendo assim, suponha que tenhamos os cromossomos P1 e P2, onde *fitness*(P2) > *fitness*(P1), e ambos gerem o cromossomo F, sendo *fitness*(F) < *fitness*(P1) e *fitness*(F) < *fitness*(P2). Neste caso, pelo método descrito no parágrafo anterior, o cromossomo P2 será substituído pelo cromossomo F resultante do seu cruzamento com P1. Porém, suponha que a rota representada por F seja muito semelhante à rota representada por P1, enquanto a rota representada pelo cromossomo P2, além de não apresentar grandes semelhanças com a rota do cromossomo F, apresente, ainda, características imprescindíveis à solução ótima. Neste caso, descartar o cromossomo P1 não representará uma perda significativa para o GA, visto que o mesmo gerou, praticamente, uma cópia no cruzamento, garantindo a preservação das suas características. Por outro lado, descartar o cromossomo P2 pode significar uma perda irreparável para o GA, pois o mesmo pode ser o único detentor de alguma subrota da solução ótima, por exemplo.

Deste modo, a seguinte alteração foi adicionada a técnica citada no primeiro parágrafo com o intuito de minimizar este problema: os descendentes mais aptos do que qualquer um dos seus respectivos pais substituirá o pai que menos se assemelhe ao mesmo, caso ele represente a melhor rota da população. A semelhança entre dois cromossomos é dada pela quantidade de nós na mesma posição em suas rotas.

5.5.5. FUNÇÃO APTIDÃO

Nos testes efetuados, para garantir uma comparação fidedigna aos resultados obtidos por Jayalakshmi et al.(2001), dentre outros autores, a função aptidão implementada ateve-se apenas em calcular a distância Euclidiana entre cada nó do grafo, considerando que o movimento da broca é livre. Caso os movimentos da broca se restringissem apenas à vertical e horizontal, o cálculo da distância de Manhattam seria mais adequado.

Não foram implementadas neste trabalho funções de tempo, considerando as velocidades dos eixos X e Y da broca, bem como seus respectivos tempos de aceleração e desaceleração, para não aumentar a complexidade do trabalho. Entretanto, uma modelagem

matemática do tempo de perfuração de uma placa de circuito impresso pode ser vista em Ăncau (2008, p. 286).

5.5.6. SELEÇÃO DOS PAIS

Os pais que participarão do *crossover* serão selecionados aleatoriamente, pois não foram visualizadas alterações positivas no tempo de convergência do GA e na solução final obtida ao implementar os métodos de seleção por torneio e roleta viciada.

A não eficácia de métodos de seleção de pais nesta configuração do GA pode ser justificada pelo fato de metade da população inicial ser formada por super indivíduos, ou seja, indivíduos cuja aptidão chega a ser até 90% maior do que a outra metade restante.

No caso do método de seleção por torneio, por exemplo, onde k indivíduos são selecionados aleatoriamente, e o indivíduo cuja aptidão for maior do que os outros k-1 indivíduos é selecionado para operar um *crossover*, a chance de um super indivíduo ser selecionado é de 75%, o que porventura afetará a diversidade da população do GA.

5.5.7. PARÂMETROS

Os parâmetros escolhidos para a execução do GA se baseiam nas experiências relatadas por Jayalakshmi et al.(2001). Deste modo teremos a seguinte configuração:

Tabela 1 - Parâmetros utilizados pelo GA para três modelos de PCBs com dimensões distintas.

Dimensão	**Tamanho da População**	**Taxa de Mutação**
198	50	0.02
280	50	0.02
442	50	0.02

6. RESULTADOS E TESTES

Os testes descritos neste capítulo com o intuito de avaliar o HGA proposto foram executados em uma máquina Dell, modelo Inspiron N4010, processador Intel(R) Core(TM) i5 2,67 GHz operando em sistema Windows 7 Home Basic (64 bits). O algoritmo foi escrito em Java, na sua versão 1.6. As instâncias do problema do caixeiro viajante bem como o *benchmark* utilizado nos experimentos foram extraídas da biblioteca TSPLIB (<http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/>).

A seguir serão demonstradas os efeitos de cada uma das modificações propostas por este trabalho no algoritmo original de Jayalakshmi et al.(2001), utilizando os parâmetros descritos no capítulo 5.5.7. A instância utilizada para os testes comparativos foi a d198 da TSPLIB, que dentre as instâncias retratas por Jayalakshmi et al.(2001), é a única que representa um modelo de placa de circuito impresso, contendo esta 198 furos de mesmo diâmetro. Ademais, para comprovar a eficácia da heurística, foram utilizadas as instâncias a280 e pcb442 que também representam modelos de placas de circuito impresso.

6.1. EFEITO DA HEURÍSTICA DE SAVINGS NA INICIALIZAÇÃO DA POPULAÇÃO

Para inicialização da população, manteve-se a proposta do autor de inicializar apenas metade da população randomicamente, inicializando a outra metade através de uma heurística construtiva. A heurística selecionada pelo autor foi a *Initialization Heuristic*, descrita nos capítulo 4.3, porém, devido à baixa diversidade de soluções geradas por este algoritmo, outras heurísticas construtivas foram analisadas para avaliar o comportamento do HGA quando submetido a outros métodos e os resultados são os apresentados abaixo.

As heurísticas selecionadas foram a *Nearest Neighbor* (NN), a *Savings Heuristic* (SH) e a Heurística de Inicialização em duas variações: a apresentada pelo autor (IH) e a que inicializa a rota com o invólucro convexo formado pelos nós do grafo, conhecida como *Convex Hull* (CH). Todas as heurísticas citadas foram apresentadas nos capítulos 2.1.2 e 4.3.

Os parâmetros avaliados foram: a média do *fitness* da população inicial (ou seja, a soma das aptidões de todos os cromossomos integrantes da população inicial dividido pelo tamanho da população), a diversidade da população (DP, ou seja, o quão parecido os

cromossomos da população são em relação aos demais, calculada através da quantidade de nós iguais presentes na mesma posição entre uma rota e todas as demais rotas da população) e a diferença entre a melhor solução obtida e a solução ótima global, sendo esta representada pelo *benchmark*. A execução do HGA é finalizada após 20000 iterações, e os melhores resultados, apresentados na tabela abaixo, foram obtidos após 50 execuções do HGA para cada configuração.

Tabela 2- Resultados obtidos através da inicialização de metade da população por diversos métodos heurísticos.

Condição	Problema	*Fitness* da População	DP da População	Melhor Solução	Nº de Iterações	Diferença para o *Benchmark*
Com IH	d198	111161,8	5456,9	16256	16878	+3,0%
Com HS	d198	111070,2	1445,8	16165	9272	+2,4%
Com NN	d198	111752,2	46473,7	16188	19191	+2,6%
Com CH	d198	111184,3	4967,5	16089	16860	+2,0%

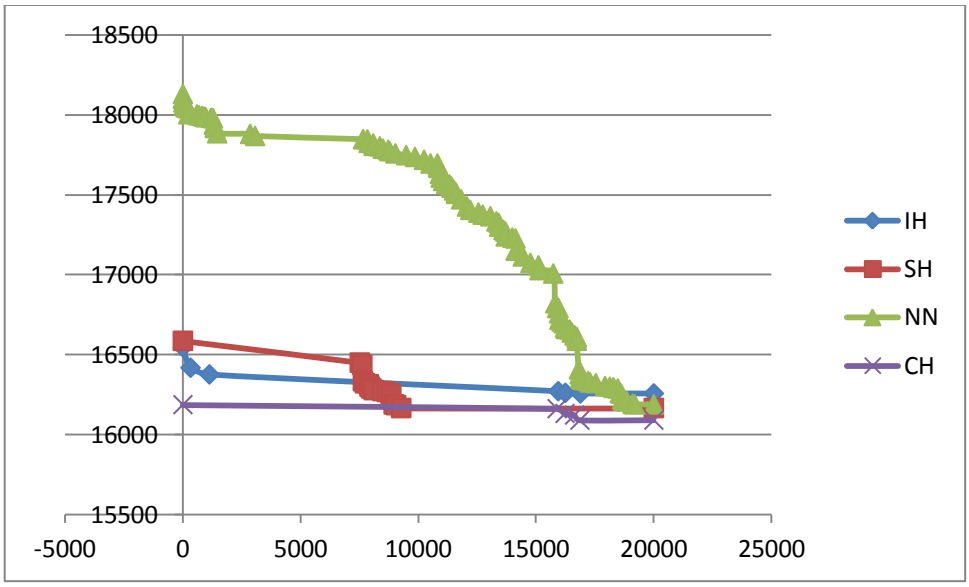


Figura 13 – Gráfico FITNESS X ITERAÇÃO de convergência da melhor solução após 20000 iterações.

Dentre as heurísticas avaliadas, a que apresentou o melhor *fitness* médio na população inicial foi à *Savings Heuristic*, entretanto este parâmetro se mostrou irrelevante quando confrontado com os demais. A *Savings Heuristic* apresentou, ainda, a população com maior variabilidade de soluções, o que afeta diretamente o desempenho do GA conforme discutido no capítulo 5.5.1 e o menor tempo de convergência, obtendo a melhor solução duas vezes mais rapidamente do que a segunda colocada.

Entretanto, a heurística de *Savings* não apresentou a melhor solução final, que ficou a cargo da heurística denominada *Convex Hull* que, dentre as demais heurísticas, é a que gera

soluções mais próximas da melhor solução definida na TSPLIB. Isto pode ser visualizado na figura 12, onde o HGA inicializado por intermédio desta heurística apresenta o melhor *fitness* inicial.

Apesar da heurística do *Convex Hull* ter apresentado a melhor solução final, em uma quantidade razoável de iterações, ela não pode ser caracterizada com a melhor heurística de construção, pois a mesma só consegue obter boas avaliações por iniciar sua população com valores muito próximos ao ponto mínimo global, sendo assim muito pouco afetada pelas operações efetuadas pelo HGA. Além disso, em mais de 50% das tentativas de obter a solução através do *Convex Hull*, o HGA sequer conseguiu obter uma solução melhor do que a melhor solução gerada pela heurística.

A *Nearest Neighbor*, apesar de apresentar uma baixa variabilidade populacional e o maior *fitness* médio se saiu bem em sua avaliação final, se mostrando sensível à aplicação dos operadores do HGA, além de ter sido a única que não apresentou sinais significativos de convergência após as 20000 iterações, ou seja, se o HGA continuasse sua execução a heurística poderia obter resultados ainda melhores se comparado aos demais.

A *Initialization Heuristics* por sua vez apresentou o pior resultado dentre todas as heurísticas avaliadas, demonstrando a fragilidade da baixa diversidade de soluções que produz.

Por fim, podemos concluir que a *Savings Heuristic* se mostrou a melhor heurística para inicialização do HGA, apresentando uma boa solução final, apesar de não ter sido a melhor. Além disto, esta heurística apresentou uma taxa de convergência extremamente baixa quando comparada aos demais e uma alta variabilidade populacional, o que a torna mais sensível aos operados do HGA e as demais configurações que serão descritas a seguir.

6.2. EFEITOS DA MODIFICAÇÃO NO OPERADOR DE *CROSSOVER EDGE RECOMBINATION*

Definido o método de inicialização da população, partimos para avaliação da modificação do operador de *crossover Edge Recombination*(ER) proposta no capítulo 5.5.2.

Tabela 3 - Resultados obtidos a partir da modificação no operador de crossover *Edge Recombination*.

Condição	Problema	Qtd. de filhos mais aptos	Melhor Solução	Nº de Iterações	Diferença para o *Benchmark*

ER Original	d198	15	16165	9272	+2,4%
ER Modificado	d198	12	15872	19913	+0,6%

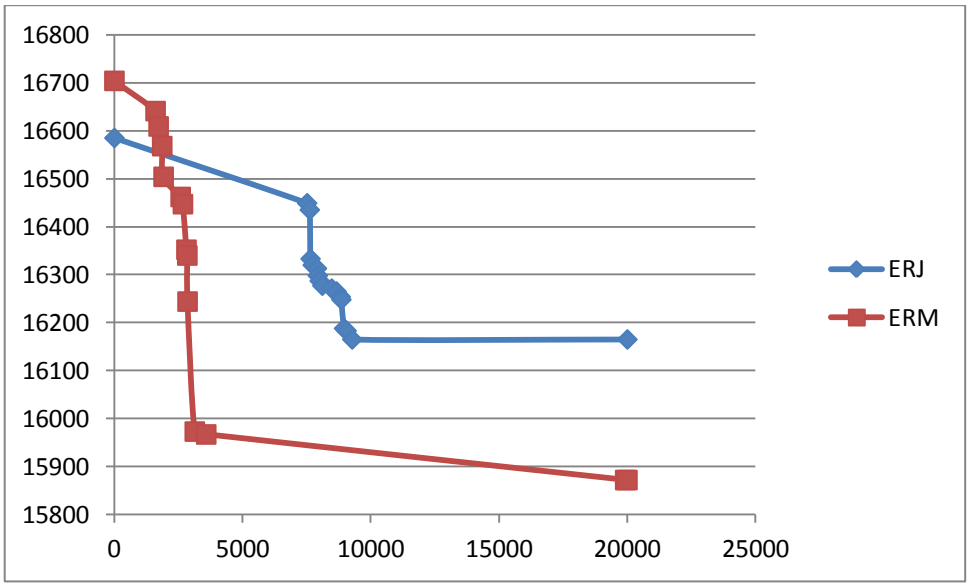


Figura 14- Gráfico FITNESS X ITERAÇÃO de convergência da melhor solução após 20000 iterações.

Apesar do tempo de convergência do algoritmo ter aumentado em cerca de 50%, observem que o HGA com o novo operador de *crossover* consegue alcançar a melhor solução obtida pelo HGA com o operador de *crossover* original em menos de 5000 iterações. Apesar do operador de *crossover* original ter gerado mais filhos aptos do que seus respectivos pais em relação ao operador de *crossover* modificado, conforme coluna 3 da tabela 2, a qualidade dos filhos gerados por este último se mostrou superior ao primeiro conforme demonstrado no gráfico da figura 13.

Os experimentos demonstram, portanto, que o operador de *crossover ER* modificado resulta em melhores rotas do que o operador de *crossover* ER originalmente proposto por Jayalakshmi et al.(2001).

6.3. EFEITOS DA SELEÇÃO POR SIMILARIDADE

Conforme demonstrado no capítulo anterior, a adição do operador de *crossover* ER modificado aumentou o tempo de convergência do HGA. No entanto, os novos experimentos efetuados a partir da adição da seleção de descendentes por similaridade, descrita na seção 5.5.4, além de apresentar resultados superiores à configuração anterior, diminuíram o tempo de convergência da solução em cerca de 50%.

Tabela 4- Resultados obtidos a partir da seleção por similaridade.

Condição	Problema	Melhor Solução	Nº de Iterações	Diferença para o *Benchmark*
Sem Similaridade	d198	15872	19913	+0,6%
Com Similaridade	d198	15820	9289	+0,3%

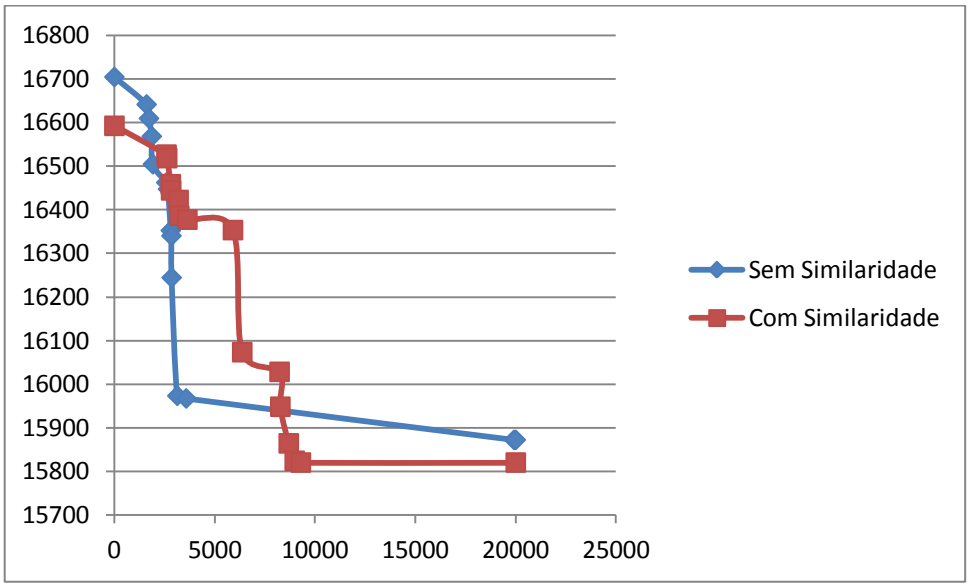


Figura 15 - Gráfico FITNESS X ITERAÇÃO de convergência da melhor solução após 20000 iterações.

6.4. OUTROS RESULTADOS

Na tabela 4 são apresentados os resultados obtidos a partir da aplicação do algoritmo proposto por este trabalho em outros modelos de placa de circuito impresso dispostos na TSPLIB. A execução do algoritmo limitou-se a 40000 iterações para os modelos abaixo e a melhor solução foi extraída entre as melhores soluções obtidas em 15 execuções do algoritmo para cada modelo.

Tabela 5 - Resultados obtidos a partir da aplicação na nova configuração do HGA

Problema	Melhor Solução	Nº de Iterações	Diferença para o *Benchmark*
a280	2579	24643	0%
pcb442	52486	39686	+3,4%

Para o problema a280, a nova configuração conseguiu atingir a solução ótima global. Já para o problema pcb442, a melhor solução obtida mostrou-se 3,4% mais "cara" do que a solução ótima conhecida para o problema. No entanto, pelo gráfico de convergência apresentado na figura 16, é possível notar que o algoritmo ainda não convergiu após a execução das 40000 iterações delimitadas. Logo, aumentando o tempo de execução do algoritmo é possível diminuir a diferença da solução obtida para a solução ótima global. Ainda assim, 40000 iterações mostraram-se suficientes para comprovar a eficácia do método, principalmente quando comparada a outro método, como a rede neural proposta por Siqueira (2005). Siqueira (2005) apresentou como melhor solução obtida a partir da aplicação da sua rede neural ao problema pcb442 da TSPLIB uma solução 9,2% mais cara do que a solução original.

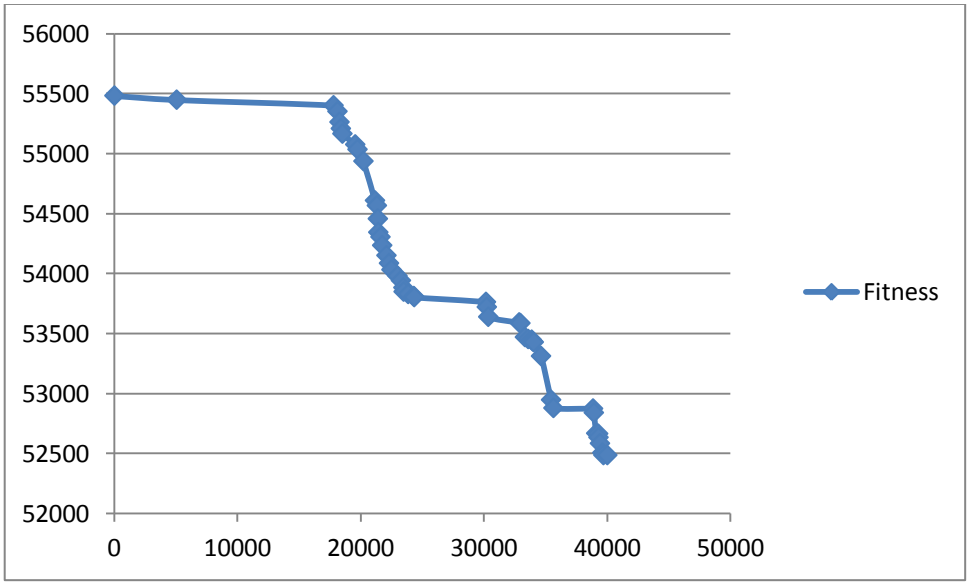


Figura 16 - Gráfico FITNESS X ITERAÇÃO de convergência da melhor solução após 40000 iterações para o problema pcb442.

7. CONCLUSÃO E TRABALHOS FUTUROS

Neste trabalho foi implementado um método baseado no algoritmo genético híbrido proposto por Jalayakshmi et al. (2001) para resolução do problema do caixeiro viajante com o intuito de otimizar o processo de perfuração de placas de circuito impresso, propondo-se às seguintes modificações:

- Inclusão da heurística de *Savings* na inicialização da população, em detrimento da *Initialization Heuristics*.
- Preservação das "subrotas comuns" no operador de *crossover Edge Recombination*.
- Controle da população pela similaridade das rotas, além do custo das mesmas.

Com os resultados obtidos no capítulo 6, comprova-se eficácia do método proposto quando comparada ao método original para o modelo de placa de circuito impresso d198 do TSPLIB, onde a configuração descrita no capítulo 5 alcançou uma média até 0,3% acima da solução ótima global e 0,2% abaixo da solução obtida por Jalayakshmi et al. (2001) em, aproximadamente, ¼ das iterações deste último.

O algoritmo se mostrou eficaz, ainda, em outras instâncias do TSPLIB, a208 e pcb442, que, assim como a instância d198, também representam um modelo de placa de circuito impresso. O algoritmo alcançou a solução ótima global para o primeiro problema, enquanto que para o segundo obteve uma solução apenas 3,4% acima da solução dita ótima, obtendo assim uma solução três vezes melhor à alcançada por Siqueira (2005) através da aplicação de uma rede neural.

Além disso, para o problema a208, diferentemente das heurísticas propostas por Ăncau(2008) e Siqueira (2005), o algoritmo proposto consegue alcançar a solução ótima, enquanto os demais apresentam, respectivamente, soluções 4,8% e 12,1% acima da solução ótima.

Vale ressaltar, ainda, que a solução obtida para o problema pcb442 esteve limitada a um escopo de apenas 40000 iterações e, a partir do gráfico apresentado na figura 16 do capítulo 6, pode-se concluir que o algoritmo teria grandes chances de obter melhores resultados em caso de aumento do escopo.

Entretanto, é importante frisar, que, por limitações técnicas, o algoritmo proposto por este trabalho não foi executado para instâncias cujo modelo da placa apresenta alta densidade. Para uma placa com 2000 furos, por exemplo, são necessários em média, para a configuração convencional da máquina de testes apresentada no capítulo 6, dois dias para se obter valores próximos da solução ótima global. Logo, para obter um resultado dentro de 30 execuções do algoritmo, seriam necessários aproximadamente dois meses de testes ininterruptos.

Logo, como sugestão para futuros trabalhos, tem-se a paralelização do algoritmo genético híbrido proposto por este trabalho com o objetivo de analisar sua eficiência para instâncias maiores.

Além disso, novas modificações podem ser feitas no HGA com o intuito de aperfeiçoá-lo, como:

- Elaborar um HGA cujos parâmetros, tamanho da população e taxa de mutação, sejam variáveis.
- Testar novas heurísticas de otimização em substituição aos algoritmos *RemoveSharp e LocalOpt* utilizados por este trabalho.
- Elaborar uma função aptidão fidedigna ao problema de perfuração de placas de circuito impresso.

REFERÊNCIAS BIBLIOGRÁFICAS

ANCĂU, M. **The optimization of printed circuit board manufacturing by improving the drilling process productivity**. *Computer & Industrial Engineering*. p. 279-294, 2008.

AYOAMA, E.; HIROGAKI T.; KATAYAMA T.; HASHIMOTO N. **Optimizing drilling conditions in printed circuit board by considering hole quality optimization from viewpoint of drill movement time**. *Journal of Materials Processing Technology*. p. 1544-1550, 2004.

BARCELLOS, J. C. H. **Algoritmos Genéticos Adaptativos: Um estudo comparativo**. 2000. 131 f. Tese (Mestrado em Engenharia) – Escola Politécnica da Universidade de São Paulo, São Paulo. 2000.

BRAGA, N. C. **Eletrônica Básica para Mecatrônica**. *Mecatrônica Fácil*, Tatuapé, n. 1, p. 9-17, 2001.

BRYANT, K. **Genetic Algorithms and the Traveling Salesman Problem**. Department of Mathematics, Harvey Mudd College. 2000.

BUCHBERGER, B. et al. **Hagenberger Research**. 1 ed. Springer, 2009. 496 p.

CHAVES, A. A. **Heurística Híbrida com Busca através de Agrupamento para o Problema do Caixeiro Viajante com Coleta de Prêmios**. 2005. 68 f. Tese (Mestrado em Computação Aplicada) – Instituto Nacional de Pesquisas Espaciais, São José dos Campos. 2005.

CHRISTOFIDES, N. **The Travelling Salesman Problem**. In: CHRISTOFIDES, N.; MINGOZZI, A.; TOTH P.; SANDI C. *Combinatorial Optimization*. Cichester: John Wiley & Sons, 1979. p. 131-149.

CHONG, Y. N. **Heuristic Algorithms For Routing Problems**. 2001. 279 f. Tese (PhD em Filosofia) – School of Mathematics and Statistics, Curtin University of Technology, Perth. 2001.

FONSECA, C. H. C. **Otimização do posicionamento e da montagem automática de componentes eletrônicos através de um algoritmo genético**. 2005. 80 f. Tese (Mestrado em Ciências em Engenharia Elétrica) – COPPE, Universidade Federal do Rio de Janeiro, Rio de Janeiro. 2005.

GANHOTO, M. A. **Abordagens para problemas de roteamento**. 2004. 112 f. Tese (Mestrado em Computação) – Instituto de Computação, Universidade Estadual de Campinas, Campinas. 2004.

GOLDBARG M. C.; LUNA H. P. L. **Otimização combinatória e programação linear**: modelos e algoritmos. 1 ed. Rio de Janeiro: Campus, 2000. 649 p.

JAYALAKSHMI, G. A.; SATHIAMOORTHY S.; RAJARAM R. A Hybrid Genetic Algorithm – A New Approach to Solve Traveling Salesman Problem. **International Journal of Computational Engineering Science**. 2001.

KRAR, S.; GILL A. **Computer Numeric Control Programming Basics**. 1 ed. New York: Industrial Press Inc, 1999. 51 p.

LINDEN, R. **Algoritmos Genéticos**. 2 ed. Rio de Janeiro: Brasport, 2008. 428 p.

MESTRIA, M; OCHI, L. S.; MARTINS, S. L. Heurísticas Híbridas para o Problema do Caixeiro Viajante com Grupamentos. In: ENCONTRO NACIONAL DE ENGENHARIA DE PRODUÇÃO, 30., 2010. **Anais do XXX Encontro Nacional de Engenharia de Produção**. São Carlos: ENEGEP, 2010.

MICHALEWICZ, Z. **Genetic Algorithms + Data Structure = Evolution Programs**. 2 ed. New York: Springer, 1996. 387 p.

NILSSON, C. **Heuristics for the Traveling Salesman Problem**. Linköping University, 2003.

PAPADIMITRIOU, C. H.; STEIGLITZ K. **Combinatorial Optimization**: Algorithms and Complexity. 1 ed. Dover Science, 1998. 512 p.

REINELT, G. **The Traveling Salesman: Computational Solutions for TSP Applications**. *Lectures Notes in Computer Science*, v. 840. Berlim: Spring-Verlag, 1994.

ROSENKRANTZ, D. J.; STEARNS, R. E.; LEWIS II, P. M.. **Analysis of several heuristics for the traveling salesman problem**. *SIAM Journal on Computing*, v. 6, n. 3, p. 563–581, 1977.

SIQUEIRA, P. H. **Uma nova abordagem na resolução do problema do caixeiro viajante**. 2005. 116 f. Tese (Doutorado em Ciências) – Universidade Federal do Paraná, Curitiba. 2005.

SPLINDER, M. **Uma proposta de solução para problemas de horário educacional utilizando busca dispersa e reconexão por caminhos**. 2010. 84 f. Dissertação (Mestrado em Computação Aplicada) – Universidade do Vale do Rio dos Sinos, São Leopoldo. 2010.

TOSCANO FILHO, A. B. A. **Um ambiente na *Web* para Otimização do Transporte de Passageiros sob Regime de Fretamento**. 2006. 138 f. Dissertação (Mestrado em Informática Aplicada) – Fundação Edson Queiroz, Universidade de Fortaleza, Fortaleza. 2006.

APÊNDICE A – Trecho da implementação da heurística de *Savings*

```
/**
 * Método que retorna uma rota construida a partir da heurística de Savings
 *
 * @param tsp - Instância do TSP
 * @return Rota construida
 * @throws Exception
 */
public static int[] savingHeuristic(TSP tsp) throws Exception {

        SavingHeuristic sh = new SavingHeuristic();

        //Seleciona um nó aleatoriamente.
        int noBase = (int)(Math.random()*tsp.getDimensao());

        //Obtém a Lista de Economias.
        List<Economia> listEconomia = obterListaEconomias();

        //Ordena a Lista de Economias
        Collections.sort(listEconomia);

        //Cria a lista de rotas
        List<List<Integer>> listRotas = new ArrayList<List<Integer>>();

        //Obtém cada aresta da lista de economias.
        for (Economia e : listEconomia) {
                //Se a lista de rotas está vazia cria uma rota.
                if (listRotas.isEmpty()) {
                        List<Integer> rota = new ArrayList<Integer>();
                        rota.add(noBase);
                        rota.add(e.no1);
                        rota.add(e.no2);
                        rota.add(noBase);
```

```
                        listRotas.add(rota);
                }
                else {
                        //Variáveis que indicam se os nós da aresta atual da lista de economias
já estão em alguma rota e se elas são nós internos.
                        boolean possuiNo1 = false, possuiNo2 = false, isNo1Interno = false,
isNo2Interno = false;
                        //Variáveis que indicam a rota e a posição nesta rota onde estão
presentes os nós da aresta atual da lista de economias.
                        int rotaNo1 = -1, posNo1 = -1, rotaNo2 = -1, posNo2 = -1;
                        //Verifica se a aresta atual da lista de economias já pertence a alguma
rota.
                        for (int i = 0; i < listRotas.size(); i++) {
                                List<Integer> rota = listRotas.get(i);
                                //Verifica em cada rota da lista de rotas se os nós da aresta já
foram inseridos.
                                for (int j = 1; j < rota.size() - 1; j++) {
                                        Integer no = rota.get(j);
                                        //Verifica se o nó 1 da aresta está presente na rota.
                                        if (e.no1 == no.intValue()) {
                                                possuiNo1 = true;
                                                //Verifica se o nó 1 é um nó externo ou interno.
                                                if (rota.get(j+1).intValue() == noBase) {
                                                        rotaNo1 = i;
                                                        posNo1 = j+1;
                                                }
                                                else if (rota.get(j-1).intValue() == noBase) {
                                                        rotaNo1 = i;
                                                        posNo1 = j;
                                                }
                                                else {
                                                        isNo1Interno = true;
                                                }
                                        }
                                        //Verifica se o nó 2 da aresta está presente na rota.
                                        if (e.no2 == no.intValue()) {
                                                possuiNo2 = true;
```

```
                                        //Verifica se o nó 1 é um nó externo ou interno.
                                        if (rota.get(j+1).intValue() == noBase) {
                                                rotaNo2 = i;
                                                posNo2 = j+1;
                                        }
                                        else if (rota.get(j-1).intValue() == noBase) {
                                                rotaNo2 = i;
                                                posNo2 = j;
                                        }
                                        else {
                                                isNo2Interno = true;
                                        }
                                }
                                //Como um nó não pode estar presente em mais de uma rota,
caso os nós 1 e 2 sejam encontrados sai do loop.
                                if (possuiNo1 && possuiNo2) {
                                        break;
                                }
                        }
                        //Como um nó não pode estar presente em mais de uma rota, caso
os nós 1 e 2 sejam encontrados sai do loop.
                        if (possuiNo1 && possuiNo2) {
                                break;
                        }
                }


                //Checa as condições para inserção do nó na rota ou para fundir duas
rotas.


                //Se possuir apenas o nó 1 e este nó não for interno, adiciona na rota
onde o nó 1 está presente o nó 2 adjacente ao mesmo.
                if (possuiNo1 && !possuiNo2) {
                        if (!isNo1Interno) {
                                listRotas.get(rotaNo1).add(posNo1, e.no2);
                        }
                }
```

```
                        //Se possuir apenas o nó 2 e este nó não for interno, adiciona na rota
onde o nó 2 está presente o nó 1 adjacente ao mesmo.
                        else if (!possuiNo1 && possuiNo2) {
                                if (!isNo2Interno) {
                                        listRotas.get(rotaNo2).add(posNo2, e.no1);
                                }
                        }
                        //Se possui os dois nós checa se é possível fundir as duas rotas onde
estes nós estão presentes.
                        else if (possuiNo1 && possuiNo2) {
                                //Se os nós não forem internos e eles não estiverem na mesma
rota, funde as duas notas.
                                if (!isNo1Interno && !isNo2Interno && rotaNo1 != rotaNo2) {
                                        fundirRotasNosExternos(posNo1, posNo2, rotaNo1, rotaNo2,
listRotas);
                                        //Remove as rotas que foram eliminadas devido a fundição
das rotas.
                                        Iterator<List<Integer>> iterator = listRotas.iterator();
                                        while (iterator.hasNext()) {
                                                List<Integer> rota = iterator.next();
                                                if (rota == null || rota.isEmpty() || rota.size()
== 2) {
                                                        iterator.remove();
                                                }
                                        }
                                }
                        }
                        //Se não possuir nenhum dos dois nós cria uma nova rota com estes nós.
                        else if (!possuiNo1 && !possuiNo2) {
                                List<Integer> rota = new ArrayList<Integer>();
                                rota.add(noBase);
                                rota.add(e.no1);
                                rota.add(e.no2);
                                rota.add(noBase);
                                listRotas.add(rota);
                        }
```

```
                }


        }


        //Obtém a rota final restante da fundição das rotas.

        List<Integer> circuitoHamiltoniano = listRotas.get(0);

        //Remove o último elemento da rota pois a rota é circular não sendo necessário incluir
na rota o nó final (nó inicial = nó final).

        circuitoHamiltoniano.remove(circuitoHamiltoniano.size()-1);



        //Retorna o vetor contendo o circuito hamiltoniano formado.

        return obterVetor(circuitoHamiltoniano);

}
```

APÊNDICE B – Trecho da implementação do módulo de inicialização do HGA

```
/**
 * Método que cria a população incial.
 *
 * @param tamanho - Tamanho da população inicial.
 */
public void inicializaPopulacao(int tamanhoPopulacao) throws Exception {


        Cromossomo novoCromossomo;
        RemoveSharp removeSharp;
        LocalOpt localOpt;


        try {
                //Inicializa a população.
                this.populacao = new ArrayList<Cromossomo>(tamanhoPopulacao);


                //Gera metade da população através do algoritmo IH.
                for (int i = 0; i < tamanhoPopulacao/2; i++) {
                        novoCromossomo = new
Cromossomo(SavingHeuristic.savingHeuristic(this.tsp));
                        //Otimiza a rota representada pelo cromossomo gerado através do
algoritmo RemoveSharp.
                        removeSharp = new RemoveSharp(this.tsp, novoCromossomo.getRota());
                        removeSharp.otimizar();
                        //Otimiza a rota representada pelo cromossomo gerado através do
algoritmo LocalOpt.
                        localOpt = new LocalOpt(this.tsp, novoCromossomo.getRota());
                        localOpt.otimizar();
                        //Calcula o fitness do cromossomo.
                        novoCromossomo.funcaoAptidao(this.tsp);
                        //Adiciona o cromossomo a população
                        this.populacao.add(novoCromossomo);
                }
                for (; i < tamanhoPopulacao; i++) {
                        //Gera a outra metada da população randomicamente.
```

```
                        novoCromossomo = new Cromossomo();

                        novoCromossomo.inicializaCromossomo(this.tsp);

                        //Otimiza a rota representada pelo cromossomo gerado através do
algoritmo RemoveSharp.

                        removeSharp = new RemoveSharp(this.tsp, novoCromossomo.getRota());

                        removeSharp.otimizar();

                        //Otimiza a rota representada pelo cromossomo gerado através do
algoritmo LocalOpt.

                        localOpt = new LocalOpt(this.tsp, novoCromossomo.getRota());

                        localOpt.otimizar();

                        //Calcula o fitness do cromossomo.

                        novoCromossomo.funcaoAptidao(this.tsp);

                        //Adiciona o cromossomo a população

                        this.populacao.add(novoCromossomo);

                }


        } catch (Exception e) {

                throw new Exception ("Erro durante a inicialização da população: " +
e.getMessage());

        }

}
```

APÊNDICE C – Trecho da implementação do operador de *crossover Edge Recombination* do HGA

```
/**
 * Método que executa a operação de crossover (Edge Recombination).
 *
 * @param pai1 - Um dos pais que participará da operação de crossover.
 * @param pai2 - Um dos pais que participará da operação de crossover.
 * @return Cromossomo gerado a partir da operação de crossover.
 */
public Cromossomo edgeRecombination(Cromossomo pai1, Cromossomo pai2) throws Exception {
	try {

		Cromossomo filho = new Cromossomo();
		filho.setRota(new int[this.tsp.getDimensao()]);

		int i, aux;
		int noCorrente;
		int proximoNo = 0;

		//Inicializa o mapa de adjacências
		Map<Integer, List<Integer[]>> mapaAdjacencias = new HashMap<Integer,
List<Integer[]>>();
		for (i = 0; i < this.tsp.getDimensao(); i++) {
			mapaAdjacencias.put(i, new ArrayList<Integer[]>());
		}

		//Cria o mapa de adjacências
		criarMapaAdjacencias(mapaAdjacencias, pai1, pai2);

		//Seleciona o nó inicial do pai 1
		noCorrente = pai1.getRota()[0];

		aux = 0;
```

```
                //Cria o filho.
                while (aux < this.tsp.getDimensao()) {

                        //Adiciona o nó corrente a rota.
                        filho.getRota()[aux++] = noCorrente;

                        //Elimina o nó corrente das listas de arestas do mapa de adjacencias.
                        eliminaNoCorrenteListaArestas(noCorrente, mapaAdjacencias);

                        //Verifica se a lista de arestas do nó corrente está vazia
                        if (mapaAdjacencias.get(noCorrente) == null
                                        || mapaAdjacencias.get(noCorrente).isEmpty()) {
                                //Seleciona primeiro nó ainda não utilizado.
                                for (int k = 0; k < tsp.getDimensao(); k++) {
                                        if (k != noCorrente && mapaAdjacencias.containsKey(k)) {
                                                proximoNo = k;
                                        }
                                }
                        }
                        else {
                                //Obtém o nó mais próximo do nó corrente.
                                proximoNo = obterNoMaisProximo(noCorrente, mapaAdjacencias);
                        }

                        //Remove o nó do mapa de adjacencias.
                        mapaAdjacencias.remove(noCorrente);

                        noCorrente = proximoNo;

                }

        } catch (Exception e) {
                throw new Exception ("Erro durante a execução do crossover: " +
e.getMessage());
```

```
        }

        return filho;

}
```

APÊNDICE D – Trecho da implementação da estrutura do HGA

```
/**
 * Executa um HGA.
 *
 * @param tsp - Instância do TSP.
 * @param txShuffling - Probabilidade do operador de shuffling ser executado.
 * @param tamanhoPopulacao - Tamanho da população.
 * @param numIteracoes - Quantidade de Interações que o HGA executará (Tempo de Vida do HGA).
 */
public HGA(TSP tsp, double txShuffling, int tamanhoPopulacao, int numIteracoes) {
	try {

		//Inicializa a população (PASSOS 1 E 2).
		this.inicializaPopulacao(tamanhoPopulacao);

		//Variável que armazenará o melhor filho obtido em cada iteração.
		this.melhorFilho = new Cromossomo();

		//Define o valor máximo de uma rota.
		this.melhorRota = Integer.MAX_VALUE;

		//Inicializa iterações.
		int i = 0;
		while (i < numIteracoes) {

			int idxPai1, idxPai2;
			Cromossomo pai1, pai2;

			//INICIA PASSO 3

			//Seleciona aleatorimente dois pais para a operação de crossover.
			idxPai1 = (int)(Math.random()*tamanhoPopulacao);
			do {
```

```
                        idxPai2 = (int)(Math.random()*tamanhoPopulacao);

                    } while(idxPai1 == idxPai2); //Verifica se um pai foi selecionado mais
de uma vez para participar do crossover.
                    pai1 = this.populacao.get(idxPai1);
                    pai2 = this.populacao.get(idxPai2);

                    //Gera um filho a partir da combinação dos pais selecionados.
                    Cromossomo filho = this.edgeRecombination(pai1, pai2);

                    //Aplica as heurísticas de otimização ao filho gerado.
                    RemoveSharp removeSharp = new RemoveSharp(tsp, filho.getRota());
                    removeSharp.otimizar();
                    LocalOpt localOpt = new LocalOpt(tsp, filho.getRota());
                    localOpt.otimizar();

                    //Calcula o fitness do filho.
                    filho.funcaoAptidao(tsp);

                    //Verifica se a aptidão do filho é menor que a aptidão de qualquer um
dos pais.
                    if (filho.getAptidao() < pai1.getAptidao() || filho.getAptidao() <
pai2.getAptidao()) {

                        //Verifica se o filho possui a melhor rota.
                        if (filho.getAptidao < melhorRota) {
                            //Verifica qual o pai mais similar ao filho e efetua
substituição.
                            if (obterDP(filho.getRota(), pai1.getRota(),
pai2.getRota()) > 0) {
                                this.populacao.remove(idxPai2);
                                this.populacao.add(idxPai2, filho);
                            }
                            else {
                                this.populacao.remove(idxPai1);
                                this.populacao.add(idxPai1, filho);
                            }
```

```
                                        //Define nova melhor rota.
                                        this.melhorRota = filho.getAptidao();
                                        //Guarda o filho.
                                        this.melhorFilho.setAptidao(filho.getAptidao());
                                        this.melhorFilho.setRota(Arrays.copyOf(filho.getRota(),
tsp.dimensao));
                                }
                                else {
                                        //Substitui o filho pelo pai com o pior fitness.
                                        if (pai1.getAptidao() < pai2.getAptidao()) {
                                                ga.populacao.remove(idxPai2);
                                                ga.populacao.add(idxPai2, filho);
                                        }
                                        else {
                                                ga.populacao.remove(idxPai1);
                                                ga.populacao.add(idxPai1, filho);
                                        }
                                }

                        }

                        //PASSO 4

                        //Sorteia um número entre 0 e 1.
                        double shufflingRate = Math.random();
                        //Se o número sorteado for menor do que a taxa de mutação executa o
Shuffling.
                        if (shufflingRate < txShuffling) {
                                //Seleciona aleatoriamente um cromossomo na população.
                                int c = (int)(Math.random()*tamanhoPopulacao);
                                //Embaralha a rota do cromosso selecionado.
                                Tool.embaralhar(this.populacao.get(c).getRota());
                                //Calcula o novo fitness do cromossomo mutado.
                                this.populacao.get(c).funcaoAptidao(tsp);
                        }
```

```
                    i++;
                }

        } catch (Exception e) {
                throw new Exception("Erro durante a execução do HGA: " + e.getMessage());
        }
}
```