UNIVERSIDADE FEDERAL DA PARAÍBA

CENTRO DE INFORMÁTICA

Metaheurísticas aplicadas ao Symmetric Traveling Salesman Problem

Grupo:

Aellison Cassimiro Teixeira dos Santos - 11311469
Diego Filipe Souza de Lima - 11408104
Marcelo Aguiar Rodrigues - 11311862

JOÃO PESSOA - PB
NOVEMBRO/ 2016

Sumário

1 Descrição do problema **3**

2 Metaheurísticas e implementação **3**

2.1 Memetic Algorithm 3

2.2 Iterated Greedy (Gulosa Iterada) 4

3 Resultados **5**

4 Considerações finais **9**

5 Referências **10**

1 Descrição do problema

Para o exercício desta unidade, nosso grupo foi sorteado com o problema do caixeiro viajante de custo simétrico (STSP). Neste problema, temos como entrada um grafo não direcionado, pesado e simétrico, e precisamos achar um caminho de uma forma que seja possível percorrer todos os pontos do gráfico sem passar pelo mesmo ponto duas vezes percorrendo a menor distância.

O nome do problema é dado pelo seu caso de exemplo mais popular: O caixeiro viajante. Neste problema, um vendedor viaja de cidade em cidade para vender sua mercadoria, e para evitar maiores custos de transporte, ele precisa pensar cuidadosamente a respeito de qual caminho vai seguir para continuar com seus negócios. Passar por uma cidade duas vezes lhe custará tempo, além de provavelmente não conseguir nova clientela em um lugar que já tentou fazer seus negócios. Também é necessário levar em consideração a distância percorrida, pois isso frequentemente significa uma perda maior de tempo e dinheiro.

O problema possui diversas variações e aplicações além da construção de rotas, como a construção de ciclos hamiltonianos para o design de circuitos, escalonamento do Staff de voo nas linhas aéreas, roteamento de veículos, distribuição e avanço de forças militares e muitos outros casos.

Tendo em mente este problema, neste trabalho faremos uso de algumas metaheurísticas para tentar achar as melhores rotas para determinados casos de testes, extraídos do site recomendado pelo professor. Utilizando a biblioteca da linguagem java TSPLIB4J[1], fornecida pelo usuário do GitHub David Hadka (dhadka), conseguimos fazer o parsing das entradas, recebendo as informações em formatos uniformizados pela biblioteca.

2 Metaheurísticas e implementação

2.1 Memetic Algorithm

Algoritmos meméticos se baseiam na ideia de algoritmos evolucionários de criação de uma população de possíveis soluções com a ideia de refinamentos locais e mutação das soluções. Durante a execução as soluções com melhor *fitness* vão competir para passar suas características adiante fazendo com que a população evolua para uma resposta melhor. No entanto isso não quer dizer que as características irão ser repassadas sem alteração, uma parte importante dos algoritmos meméticos é o *crossover* entre os indivíduos que irão gerar novos membros da população.

Para o algoritmo funcionar eu sabia que deveria criar uma população e depois recombinar, melhorar e gerar mutações nessa população. A primeira dúvida que eu tive foi saber em que ordem eu deveria fazer cada uma dessas coisas, porém após a leitura do capítulo *MEMETIC ALGORITHMS* [3] o esquema apresentado pelos autores na figura 12-1 do referido capítulo foi suficiente para elucidar essa dúvida. O referido capítulo em conjunto com o artigo *A Gentle Introduction to Memetic Algorithms* [4] foram essenciais para o entendimento da metaheurística.

A metaheurística se mostrou bastante eficiente para as duas menores instâncias que escolhemos trabalhar, chegando à solução ótima na maioria das suas execuções. Quando comparei o número de iterações das instâncias maiores com o número de iterações do Iterated Greedy foi que percebi o quão cara é cada iteração do algoritmo memético.

Apesar do alto custo de cada iteração as soluções apresentadas sempre foram foram bastante superiores à metaheurística gulosa iterada.

2.2 Iterated Greedy (Gulosa Iterada)

A heurística gulosa iterada se baseia na ideia de uma sequência de iterações que alteram trechos de um caminho em específico. Cada uma dessas iterações altera uma área de forma que a melhor distância local será encontrada para substituir o caminho que estava ali. Dito isto, podemos ver que este algoritmo se baseia em ótimos locais numa tentativa de alcançar ótimos globais.

Durante a modelagem do algoritmo que seria implementado, foram lidos diversos artigos que especificavam mais precisamente aspectos que poderiam influenciar na eficiência do algoritmo. Para casos de uso diferentes, os pontos de destruição do caminho poderiam seguir um padrão ou uma combinação de estocacidade com uma heurística, cada um trazendo um benefício diferente para a situação.

A fim de aprimorar os conhecimentos na disciplina, foi decidido que a implementação não seguiria nenhum artigo em específico, sendo a única orientação seguida seria a dada pelo professor, que nos informou que a aplicação do processo de quebra e reconstrução é efetiva quando o ponto de execução desse processo é escolhido aleatoriamente.

No algoritmo que resolve o problema do caixeiro viajante implementado com a metaheurística gulosa iterada, se obtém primeiramente um caminho aleatório baseado na entrada fornecida. Este caminho, por sua vez, será dado como parâmetro de uma função dentro de um loop que irá modificar o mesmo, destruindo e reconstruindo partes deste caminho. Cada vez que a função de modificação é

chamada, um ponto é escolhido aleatoriamente para ser o “ponto de destruição” delimitando o início da área de modificação, e o índice desse ponto somado a um determinado número que é uma porcentagem do tamanho total do grafo (chamado de “fator de destruição”) delimita o fim do intervalo. O intervalo obtido será modificado e o novo caminho será retornado, sendo este usado pela função de modificação novamente para ser melhorado. O processo é repetido até alguma condição que termine o loop seja atingida.

Na lógica do algoritmo, é possível notar que o aumento do fator de destruição resulta numa queda maior da distância total a cada iteração, mas um fator de destruição extremamente alto iria se assimilar a execução de um algoritmo guloso sobre todo o grafo, tornando-o sujeito aos problemas de prisão em máximos locais que são comuns aos algoritmos gulosos. Então, após alguns testes, foi decidido que o valor do fator de destruição seria de 40% do tamanho total do caminho.

Além disso, foi notado um comportamento no algoritmo onde o algoritmo simplesmente chegava a resultados semelhantes depois de certa quantidade de iterações, não obtendo nenhuma melhora depois deste ponto. Para contornar esta situação, a solução usada originalmente é substituída após 1000 iterações, recomeçando todo o processo.

Em questões de dificuldades enfrentadas na implementação do algoritmo, as mais envolveram lógica de programação, pois o coração da teoria da metaheurística gulosa iterada é muito simples de se compreender. Uma vez entendida a lógica que seria implementada, não houveram mais dificuldades com relação ao design do algoritmo.

O estado primitivo da algoritmo era extremamente ineficiente, facilmente retornando os melhores resultados com um valor quadruplicado do resultado ótimo. A cada correção de bugs, os resultados foram se tornando melhores, até que chegamos aos resultados que podem ser observados na seção seguinte.

3 Resultados

Os resultados foram obtidos pela média de dez execuções de cada uma das metaheurísticas em cada uma das cinco entradas escolhidas da TSPLib. Todas as execuções foram limitadas em 60 segundos ou até que a solução ótima fosse alcançada. Após isso foi calculado o RPD de cada instância.

As figuras 3.1 e 3.2 mostram cada metaheurística individualmente para cada uma das entradas.

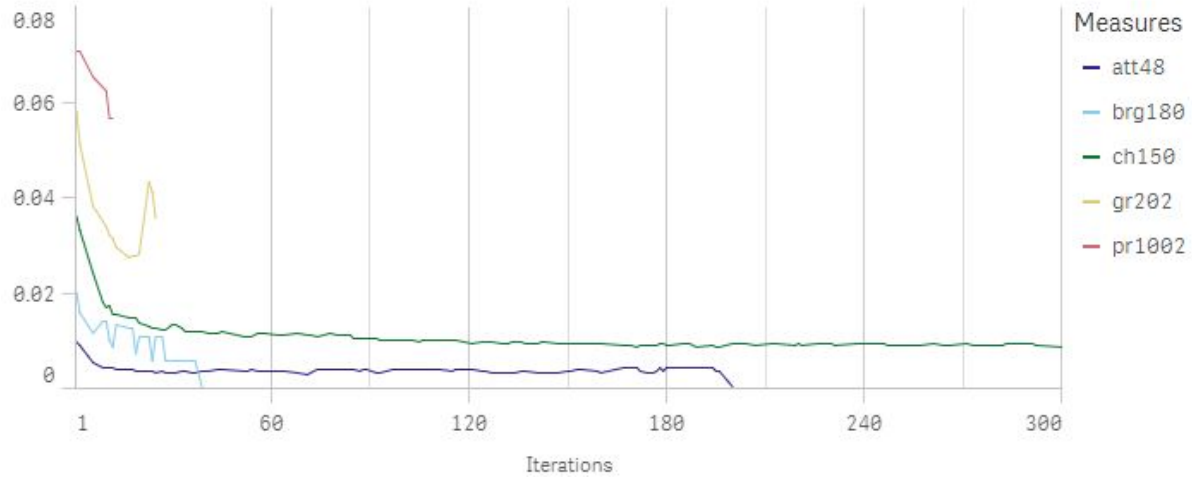

Figura 3.1 - Memetic Algorithm para todas as instâncias

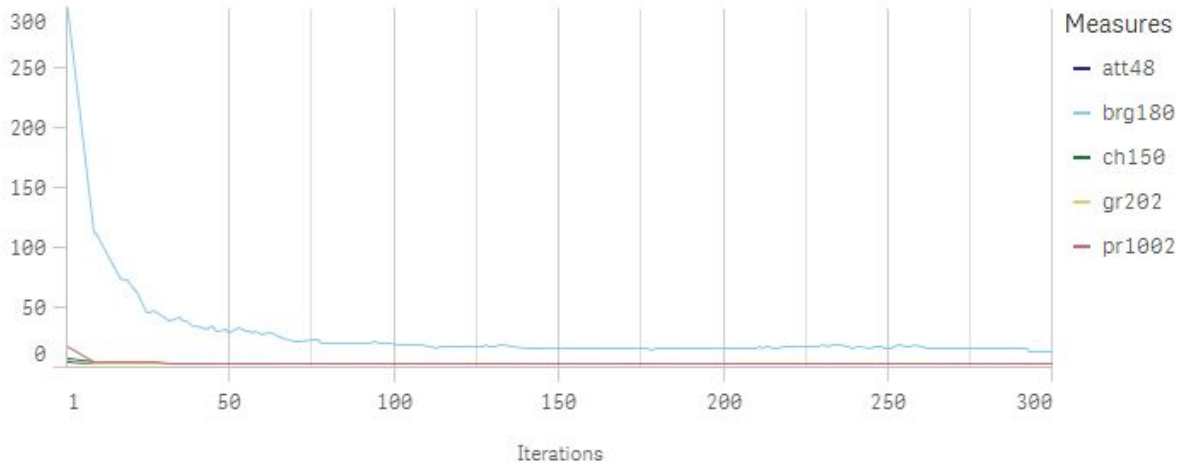

Figura 3.2 - Iterated Greedy para todas as instâncias

Como a instância *brg180* gerou um resultado muito ruim para a metaheurística Iterated Greedy a figura 3.3 mostra os resultados com essa instância removida.

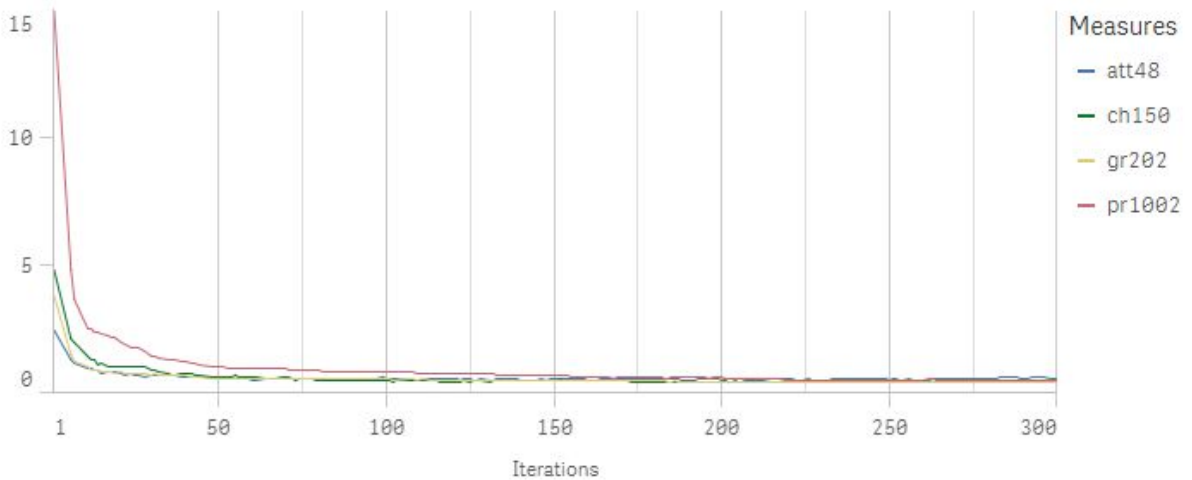

Figura 3.3 - Iterated Greedy removida a instância brg180.

Das figuras 3.4 à 3.8 mostram uma comparação entre as metaheurísticas para cada uma das instâncias testadas. Para as figuras 3.4 à 3.8 a metaheurística Iterated Greedy é mostrada em vermelho enquanto a Memetic Algorithm é mostrada em azul.

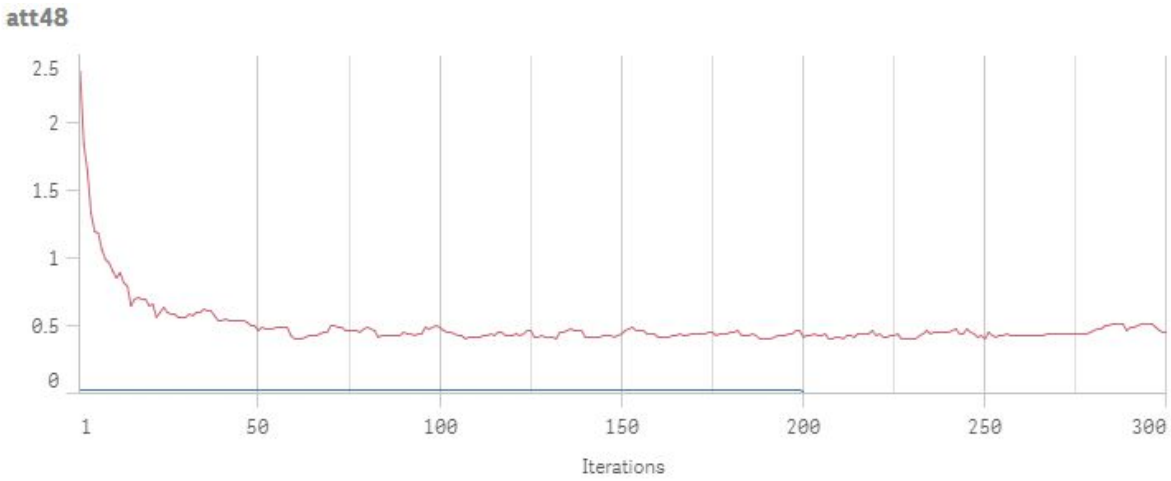

Figura 3.4 - Comparação entre Iterated Greedy e Memetic Algorithm para a instância att48.

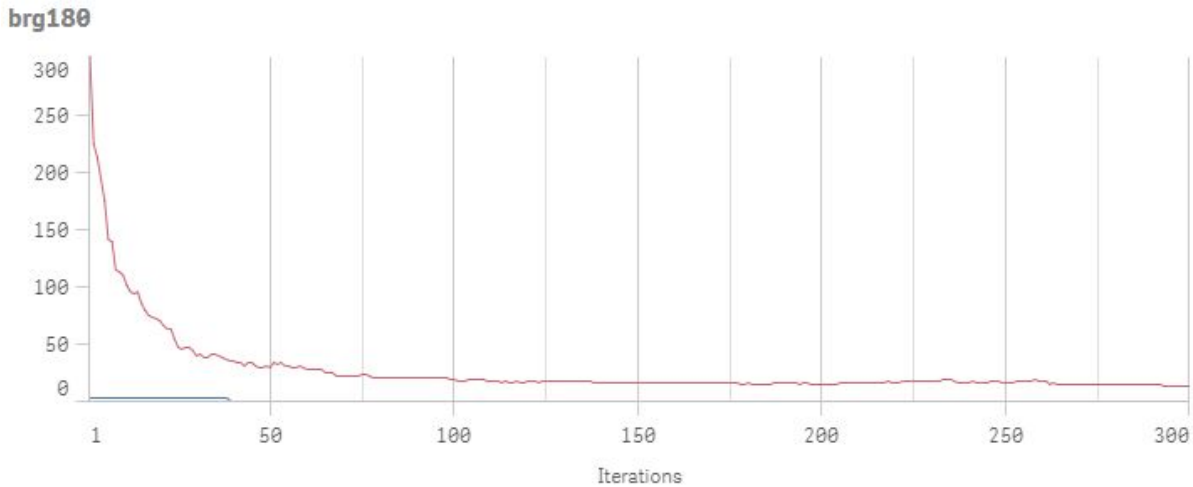

Figura 3.5 - Comparação entre Iterated Greedy e Memetic Algorithm para a instância brg180.

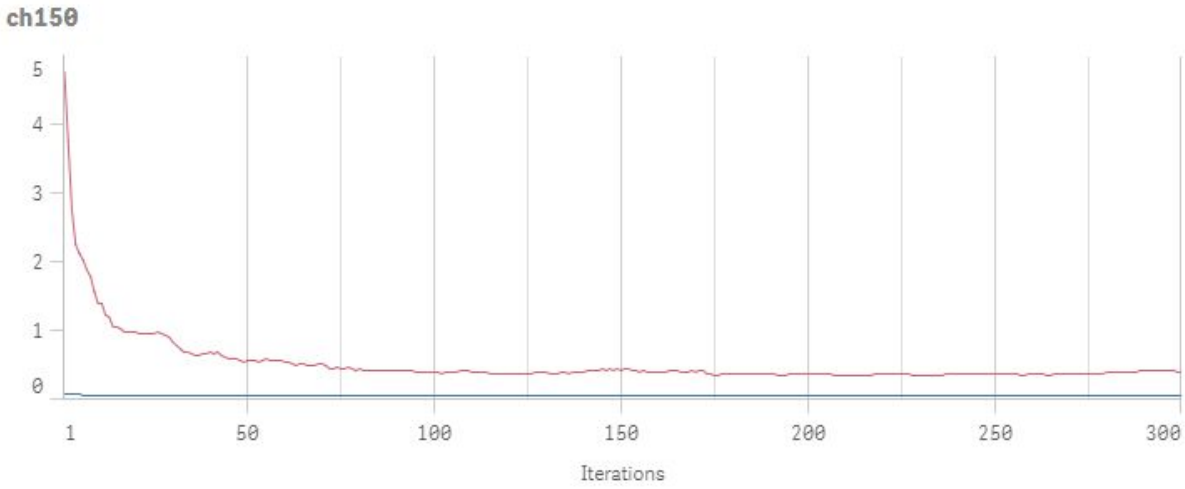

Figura 3.6 - Comparação entre Iterated Greedy e Memetic Algorithm para a instância ch150.

.

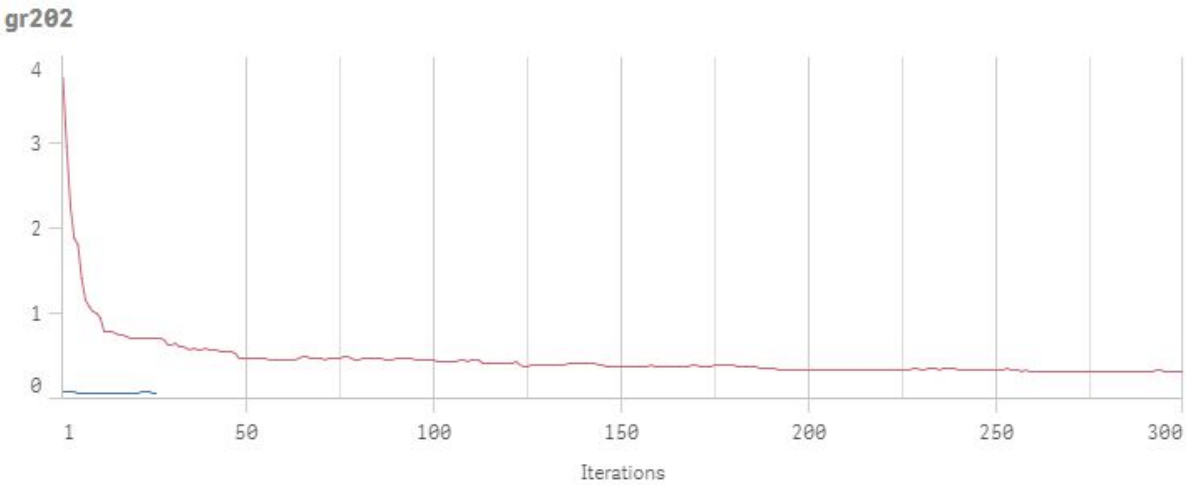

Figura 3.7 - Comparação entre Iterated Greedy e Memetic Algorithm para a instância gr202.

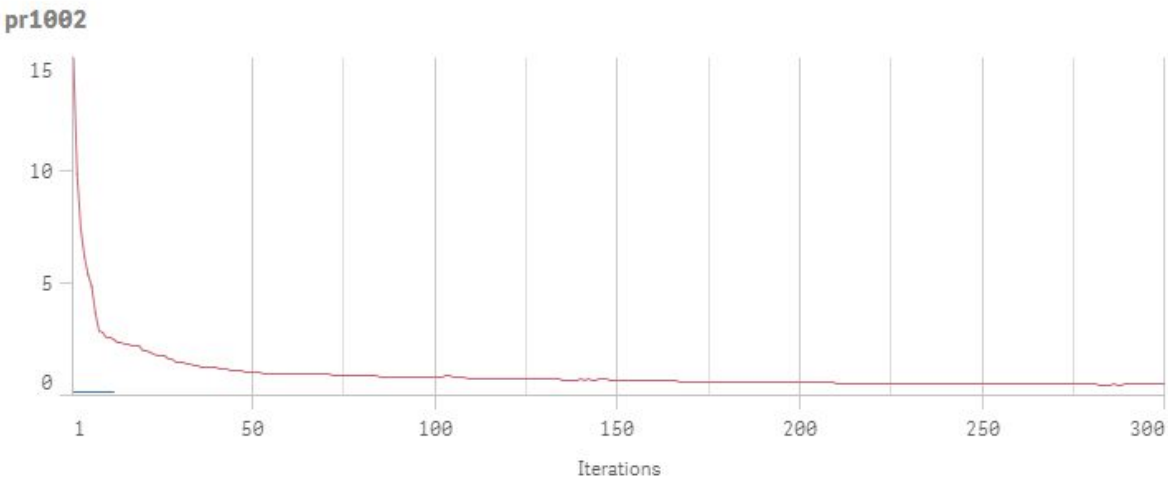

Figura 3.8 - Comparação entre Iterated Greedy e Memetic Algorithm para a instância pr1002.

4 Considerações finais

Como vimos neste trabalho, para um determinado problema, podemos aplicar uma gama de métodos para solucioná-los. Na busca destes métodos, os cientistas da computação tentaram generalizar suas soluções, e depois de muitos desafios, as metaheurísticas que podemos estudar hoje foram alcançadas. Mesmo assim, nossos exemplos nos confirmaram que uma metaheurística pode não ser adequada para fornecer o melhor caso para um problema, cabendo à equipe de desenvolvimento tomar as decisões a respeito das metaheurísticas a serem usadas para resolver diversos desafios.

No nosso caso, foi possível visualizar a diferença entre duas metodologias com complexidades diferentes. O algoritmo memético, enquanto efetivo, possui iterações lentas, e em um caso extremamente grande irá representar uma ameaça ao tempo de execução do programa. O algoritmo implementado sob a metaheurística gulosa iterada irá ter iterações rápidas, mas a reconstrução gulosa dos caminhos fará a busca do melhor caso ser mais problemática, às vezes sendo até impossível.

No fim das contas, cabe ao cientista da computação conhecer as metaheurísticas oferecidas para selecionar uma opção de solução para o problema. Tendo em mente que sua escolha poderá ter graves consequências na aplicação desenvolvida.

5 Referências

[1] Biblioteca TSPLIB4J, no GitHub - https://github.com/dhadka/TSPLIB4J

[2] Pisinger and Ropke (2010) - Chapter 13: Large Neighborhood Search.

[3] KRASNOGOR, N.; ARAGÓN, A; PACHECO, J. Chapter 12: MEMETIC ALGORITHMS In: ALBA, E.; MARTÍ, R. *Metaheuristic Procedures for Training Neural Networks*. 2006.

[4] MOSCATO, Pablo; COTTA, Carlos; *A Gentle Introduction to Memetic Algorithms*, 2003