Relatório de projeto computacional

Computação Bio-inspirada em Otimização Contínua e Discreta

Thiago A. Lechuga
RA079699
Instituto de Computação, Universidade Estadual de Campinas
Campinas, São Paulo, Brasil
thiago.lechuga@students.ic.unicamp.br

RESUMO

O objetivo desse trabalho computacional é utilizar conceitos e ferramentas vistos em sala para tratar problemas de interesse prático.

O trabalho é composto por um sucinto referencial teórico seguido por aplicações praticas de ACO min-max, algorítimo Genético, simulated annealing, PSO, opt-aiNet,aiNet, ARIA e comparações entre os três primeiros algoritmos e o Concorde. Para cada aplicação o trabalho demostra as técnicas, ferramentas e dados utilizados, seguidos pelos resultados.

1. IMPLEMENTAÇÃO

Nessa sessão os algoritmos desenvolvidos serão apresentados. Para cada algoritmo temos uma subseção com a descrição do problema, detalhes de implementação e resultados. Na primeira parte, o problema de TSP será resolvido com diversos algoritmos diferentes - ACO, AG e Simulated annealing -, em seguida será apresentado um breve resumo de características do Concorde, e após uma comparação de todos os algoritmos. Por fim serão apresentados os três métodos imuno-inspirados implementados: opt-aiNet,aiNet e ARIA.

Todas as instâncias de TSP foram retiradas do TSPLib [7] ; a base ulysses22 para uma instância fácil, a eil51 como média e a pr439 como difícil.

1.1 ACO min-max

Para resolver o problema do TSP com ACO utilizei a biblioteca para ACO do grupo IRIDIA [1] com o intuito de abstrair a implementação de partes da modelagem, como o "grafo" que as formigas devem percorrer, e me preocupar apenas com com a implementação específica do algoritmo.

Trabalhar com essa biblioteca foi bem desapontador. Ela é toda desenvolvida em C em com programação estruturada, não possui ferramentas de auxílio gráfico e não foi planejada para ser extensível. O código é bem comentado (E mantive o mesmo padrão nas atualizações), mas a estrutura procedural tornou o ato de fazer modificações um trabalho árduo e a legibilidade do código ficou prejudicada. Ainda assim a biblioteca ofereceu diversas ferramentas que colaboraram bastante com o desenvolvimento do projeto como a leitura de arquivos, parse das entradas da linha de comando e outros. O uso de algumas ferramentas dessa biblioteca também colaborou com os bons resultados apresentados pelo ACO.

Como o código foi todo em C, tive dificuldade em implementar uma interface gráfica, então o programa é todo acessado por linha de comando e para plotar os gráficos com os resultados utilizei a saída do ACO como entrada para a parte gráfica dos outros algoritmos, desenvolvidos em Java.

Na implementação foi utilizado um modelo de grafo completo com os feromônios sendo aplicados nas arestas. Inicialmente o algoritmo calcula as distâncias de todas as arestas para montar a matriz do grafo. O limites min e max podem ser vistos na função "checkPheromoneTrailLimits" no arquivo "ants.c" e foi usado um fator de evaporação de "0.5". Como termo heurístico foi utilizada a distância para a próxima cidade com uma influencia duas vezes maior que a do feromônio (numero em que foram obtidos melhores resultados).

O programa ficou dividido em 4 arquivos principais:

- **acotsp.c:** Principais rotinas do protocolo.
- **ants.c:** Comportamento das formigas.
- **TSP.c:** Funções específicas de TSP.
- **parse.c:** Funções para efetuar o parse das instruções enviadas pela linha de comando.

Para usar o software basta descompactar o arquivo, entrar na pasta pelo console de comando e digitar "./acotsp -i arquivo.tsp -m qtdFormigas". A quantidade de formigas padrão usadas nos testes foi 25.

1.1.1 Avaliação dos Resultados

Para a instância fácil foi usado o arquivo "DadosTsp/facil-ulysses22-TSPLIB.tsp". Foi observado que para esse caso os limites de min-max não tiveram grande influência; com ou sem os limites o algoritmo encontrou o caminho ótimo

(Também encontrado pelo concorde) em menos de 1 segundo e na primeira iteração. Esse comportamento foi bastante influenciado pelo termo heurístico. A distância final foi de 72 e os resultados estão no arquivo "resultados/facil-ACO".A figura 1 exibe o caminho encontrado.

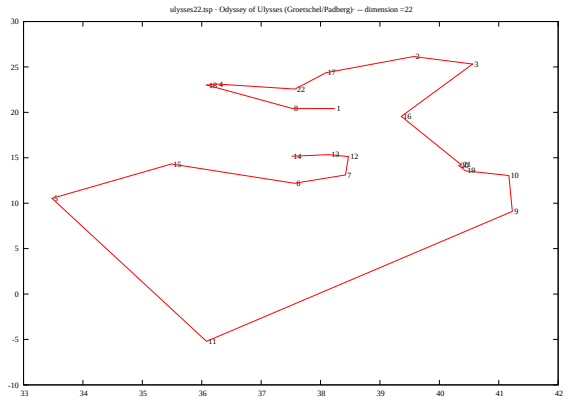

Figura 1: Resultado ótimo para a instância fácil de TSP

Para a instância mediana foi usado o arquivo "DadosTsp/medio-eil51-TSPLIB.tsp". Foi observado um comportamento semelhante a instância fácil pois, apesar de ser um pouco mais complexa, escolher a cidade mais próxima era uma boa escolha na maioria das vezes e por isso o termo heurístico teve grande peso nos resultados. Para evitar isso a instância difícil foi escolhida cuidadosamente. Mas já foi possível verificar uma boa diferença, a quantidade média de iterações para encontrar a melhor resposta subiu para 1.7 e o tempo sem os limites min e max foi dez vezes maior. A distância final foi de 426 e os resultados estão no arquivo "resultados/medio-ACO".A figura 2 exibe o caminho encontrado.

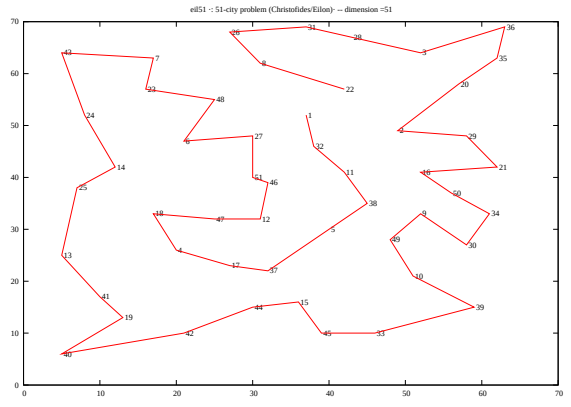

Figura 2: Resultado ótimo para a instância média de TSP

Para a instância difícil o arquivo usado foi o "DadosTsp/dificil-pr439-TSPLIB.tsp". Para instâncias desse porte o ACO se mostrou com os melhores resultados, mais uma vez só perdendo para o concorde. O melhor resultado conhecido é o visto na figura 2 com tamanho 107217. Com aproximadamente 25 segundos e pouco mais de 150 iterações o algoritmo foi capaz de encontrar caminhos muito próximos do Concorde com pequenas diferenças no trajeto, quase que imperceptíveis no gráfico, e com tamanho 107881 em média. Para a execução sem os limites max e min os resultados foram bem inferiores.

1.2 Algoritmo Genético

Nessa etapa do trabalho tinha-se como objetivo implementar um algoritmo genético para resolver problemas de TSP com propostas de busca local. Para evitar os problemas da etapa anterior tive maior cuidado na busca por ferramentas que poderiam auxiliar a implementação do projeto. Utilizei então duas ferramentas de apoio: A biblioteca JGAP [4] para facilitar a estruturação e extensibilidade da aplicação e a biblioteca JFreeChart[3] para plotar os gráficos com os caminhos do TSP, ambas para a linguagem Java.

A implementação foi toda orientada a objetos. A biblioteca JGAP facilitou bastante a implementação e permitiu que o código ficasse melhor estruturado do que se eu mesmo fosse modelar a aplicação em um curto período de tempo. O código ficou bem legível, facil de alterar e extensível, contudo as vantagens ficaram por aí. Apesar se já existir a estrutura de um gene por exemplo, é necessário especificar o formato do gene e do cromossomo específico para a aplicação, além dos algoritmos de fitness, crossover, mutação, seleção e busca local. Essa abordagem também prejudicou a velocidade da aplicação. A biblioteca JFreeChart facilitou a plotagem dos gráficos, porém se mostrou muito lenta e inviabilizou constantes atualizações dos gráficos.

A função de fitness usada pode ser vista na classe SalesmanFitnessFunction.java. Ela é bem simples, o maior inteiro possível de se representar em java foi definido como o pior resultado possível (Integer.MAXVALUE) e zero o melhor, então simplesmente calcula-se a distancia total do percurso (incluindo a volta) e o fitness resultante é dado por: (Integer.MAXVALUE / 2) - distanciaTotal;

Os algoritmos de crossover e mutação implementados foram os vistos em sala de aula e se encontram respectivamente nas classes Crossover.java e SwappingMutationOperator.java. O algoritmo de seleção pode ser escolhido entre roleta e torneiro mas o segundo foi utilizado para todos os testes por gerar melhores resultados (testado experimentalmente).

Para a implementação das buscas locais foi utilizado o padrão de projeto AbstractFactory, permitindo configurar o TSP com a busca desejada (Configurável pela tela inicial). Duas buscas foram implementadas A primeira consistia em apenas trocar aleatoriamente a ordem de duas cidades adjacentes e testar se o caminho tinha melhorado até acertar alguma vez. Esse método melhorou os resultados, mas verifiquei que o custo computacional era muito alto e vislumbrei uma abordagem potencialmente melhor. Não cheguei a verificar na literatura por alternativas mas a minha idéia para implementar a segunda busca local foi baseada na figura 3, onde

podemos ver que um cruzamento ”sempre” é uma escolha ruim e busca-los e eliminá-los melhora o algoritmo além de não ser necessário testar continuamente se ouve melhora, já que sempre há melhora. Então o algoritmo segue buscando cruzamentos e retira o primeiro que encontra. Na figura 3 tento mostrar que não importa a configuração do grafo, eliminar um cruzamento sempre irá diminuir a rota.

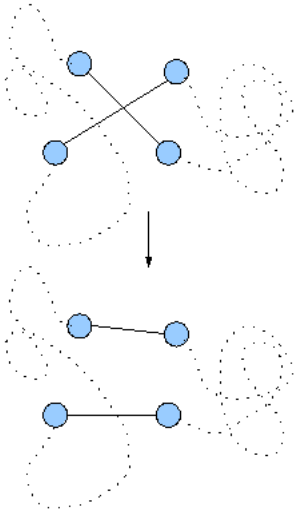

Figura 3: Exemplo de um cruzamento retirado de um grafo genérico

O programa ficou então dividido em 8 classes principais:

- **controle.GA.TSP:** Define o comportamento geral do algoritmo e garante a estrutura dos cromossomos
- **controle.GA.Alelo:** Definindo um alelo, basicamente cada alelo representa uma cidade.
- **controle.GA.GeneTSP:** Define um gene para o TSP composto por uma alelo especifico para TSP.
- **buscaLocal.BuscaLocalCruzamento:** Executa a busca local retirando cruzamento de um dado GeneTSP.
- **buscaLocal.BuscaLocalTroca:** Executa a busca local trocando alelos de um dado chromosome composto por GeneTSP.
- **Crossover.java:** Algoritmo de Crossover.
- **SwappingMutationOperator.java:** Algoritmo de mutação.
- **SalesmanFitnessFunction.java:** Algoritmo da função de Fitness.

Para executar o programa a maquina virtual Java é necessária. Em seguida basta digitar ”java -jar programa-ga.jar”. Na figura 4 vemos a tela principal do programa. Para utiliza-lo basta configurar os parâmetros: População, Gerações, Passo (De quantas em quantas gerações deve-se plotar o gráfico com o resultado atual.), Algoritmo de seleção, algoritmo de busca local e arquivo de dados. Em seguida basta executar. O programa ficou com um pequeno problema de memória que não tive tempo de arrumar, só é possível executar uma vez o GA, depois é necessário fechar o programa e abrir novamente.

Figura 4: Tela principal do programa de Algoritmo Genético

1.2.1 Avaliação dos Resultados

Para a instância fácil foi usado o arquivo ”DadosTsp/facil-ulysses22-simples.tsp”. Mesmo para a instância simples o sistema não foi capaz de encontrar o resultado ótimo (tamanho 72) mas conseguiu melhorar bastante do caminho inicial (tamanho 138), visto na figura 5, para bem próximo do resultado ótimo (tamanho 85), visto na figura 6. Os melhores resultados foram obtidos usando o algoritmo de busca local ”BuscaLocalCruzamento”, esses resultados foram na média 18%melhores do que sem busca local. Foram utilizadas 200 indivíduos, o ganho adicionando mais indivíduos era muito pequeno; e 100 gerações, mais que isso a diversidade não era suficiente para progredir. Tempo médio para encontrar o melhor resultado foi de 2.4 minutos.

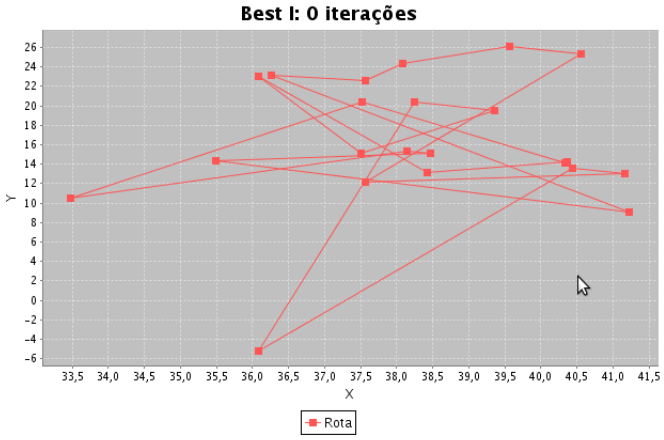

Figura 5: Solução para TSP gerada randomicamente na primeira geração do GA (instância fácil)

Para a instância de dificuldade média foi usado o arquivo ”DadosTsp/medio-eil51-simples.tsp”. O algoritmo foi configurado com 1000 indivíduos e 1000 gerações. Após aproximadamente 5 minutos de execução o melhor resultado obtido

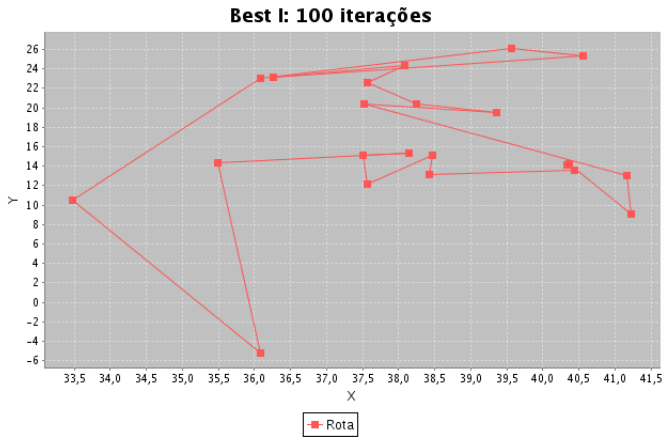

Figura 6: Melhor solução para TSP gerada após 100 gerações do GA (instância fácil)

foi o exibido na figura 8. Esse resultado teve um tamanho de 700, mais de 60%menor do que o gerado aleatoriamente (figura 7) e 22%melhor do que os gerados sem busca local (em média), mas ainda distante do ótimo (tamanho 426).

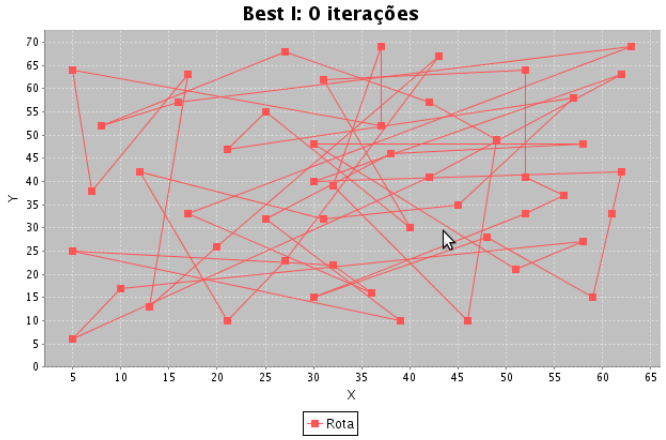

Figura 7: Solução para TSP gerada randomicamente na primeira geração do GA (instância média)

Para a instância difícil o arquivo usado foi o "DadosTsp/dificil-pr439-simples.tsp". Para instâncias desse porte a estrutura escolhida para o GA não foi capaz de resolver os problemas. Mesmo usando populações pequenas, com pouco mais de 2 minutos de execução ocorreram problemas por falta de memória (figura 9).

Foi observado que um dos motivos para os resultados ruins é que o algoritmo de crossover não está gerando bons descendentes, esse poderia ser melhorado. Outro problema é que o algoritmo de seleção perde muitas vezes indivíduos bons. Ele mantém bem a diversidade, mas poderia ser melhorado para sempre manter alguns dos melhores resultados por exemplo.

1.3 Simulated Annealing

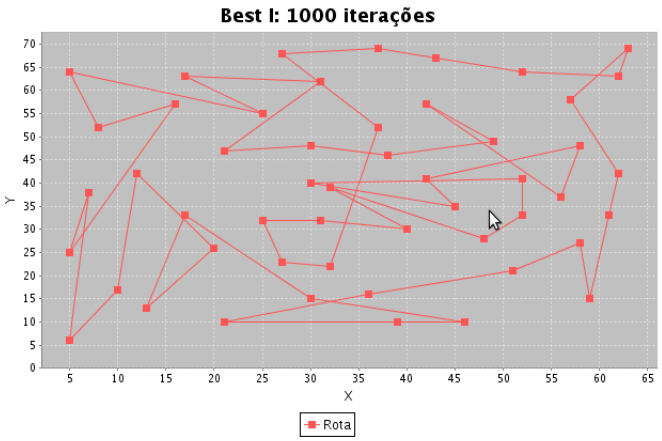

Figura 8: Melhor solução para TSP gerada após 100 gerações do GA (instância média)

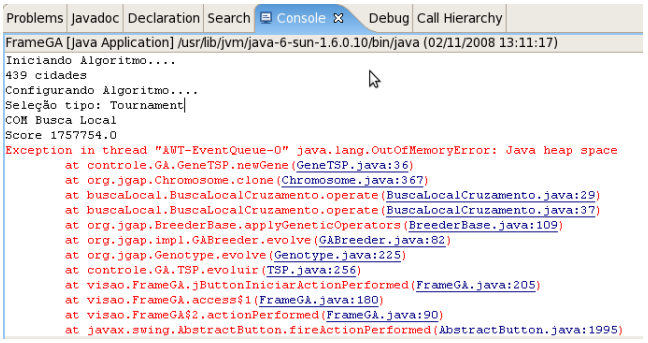

Figura 9: Erro por falta de memória do GA executando a instância difícil de TSP

Para comparar com os outros algoritmos optei por também resolver TSP com Simulated Annealing.

Por causa da simplicidade do algoritmo e das experiências não muito produtivas com as bibliotecas anteriores, optei por implementar esse algoritmo totalmente em Java e sem nenhuma biblioteca.

O algoritmo ficou mais leve e mais rápido. Contudo não ficou tão bem estruturado e reusável quando o genético. A plotagem dos gráficos também ficou muito mais rápida e permitiu uma flexibilidade maior, contudo ficou simples de mais e sem várias funcionalidades, por exemplo zoom.

O algoritmo foi todo implementado em uma única classe, TSP.java. A execução do algoritmo pode ser vista no método "comp-run-action". Primeiramente a temperatura é iniciada com a distância média entre os pontos do caminho inicial, em seguida roda repetitivamente até atingir a quantidade de steps configurada executando os seguintes passos:

- Executa o annealing: repete até o número de tentativas sem melhora ou o número de trocas com melhora (ambos configuráveis) serem atingidos:

  - Escolhe aleatoriamente dois pontos para efetuar uma troca.
  - Se não melhorou volta o anterior.
  - Se melhorou mantém a troca e aumenta o contador de trocas.
- Verifica se obteve melhora. Se obteve reduz a temperatura segundo o fator configurado e repete.
- Compara com a melhor rota obtida e substitui se foi melhor.

Lembrando que sempre a melhor solução é mantida. Portanto, executar o algoritmo repetidamente tende a ir melhorando a resposta.

Para executar o programa a maquina virtual Java é necessária. Em seguida basta digitar "java -jar programa-simann.jar". Na figura 10 vemos a tela principal do programa. Para utiliza-lo basta clicar em executar/propriedades, configurar os parâmetros (mostrados na figura), clicar em arquivo/abrir, selecionar o arquivo desejado e finalmente clicar em executar/run. O melhor resultado obtido é exibido na tela e o tamanho pode ser visto na parte inferior do gráfico.

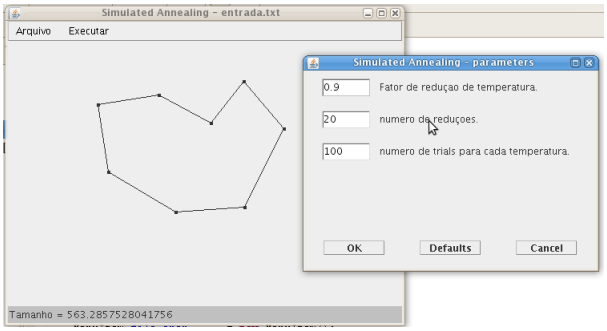

Figura 10: Tela principal do programa de Simulated Annealing

1.3.1 Avaliação dos Resultados

Como configuração padrão nos testes foram utilizados: Fator de redução de temperatura = 0.9; Número de reduções = 20 e Numero de trials por temperatura = 100. Foi possível verificar que esse foi um dos algoritmos com a melhor relação custo X benefício dos três implementados. Ele foi bem simples de implementar e com pouco tempo de execução conseguiu resultados melhores do o GA, só não mais mais rápido ou eficiente que o ACO.

Para os teste com a instância fácil o arquivo "DadosTsp/medio-eil51-simples.tsp" foi utilizado. Com uma execução já chegou a resultados próximos dos do Genético (com tamanho = 85). Pela ausência de ferramentas de Zoom tive problemas para plotar os gráficos desse algoritmo Essa instância de TSP possuía pontos muito próximos e ficou impossível de visualizar, por isso não coloquei imagens. Essas poderão ser vistas nas próximas instâncias.

Para o TSP de dificuldade média com poucas execuções já foi possível obter bons resultados , na faixa de 574 de tamanho, com aproximadamente 10 execuções já foi possível obter resultados bem superiores que o do GA, na figura 11 vemos o melhor resultado gerado (tamanho 482)(o ótimo é de 426). A visualização ficou prejudicada pela falta de uma ferramenta de zoom, como a biblioteca JFreeChart (utilizada no algoritmo de GA) possuía

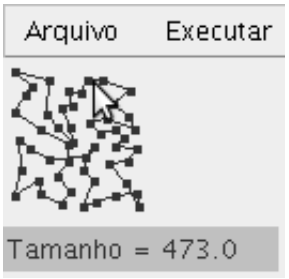

Figura 11: Melhor solução para TSP instância média gerada por Simulated Annealing

Para a instância difícil o arquivo usado foi o "DadosTsp/dificil-pr439-simples.tsp". Para instâncias desse porte o Simulated Annealing se mostrou muito ineficaz. O algoritmo foi sempre muito rápido mas conseguiu melhorar pouco a partir de um caminho aleatório. Os melhores resultados obtidos após diversas execuções consecutivas foram próximos de 328764, sendo que obtido pelo concorde era de 107217. É fácil entender o porquê desse resultado, quando o número de cidades é muito grande, a chance de obter melhora em uma troca é cada vez menor, dessa forma a temperatura acaba caindo muito rapidamente. Para contornar isso deve-se diminuir o fator de redução da temperatura e/ou aumentar o número de tentativas de troca. Contudo, dessa forma o algoritmo fica cada vez mais lento e mesmo assim não obtém bons resultados. Mesmo com configurações bem demoradas os melhores resultados obtidos foram próximos de 200000.

1.4 Concorde

Para facilitar e permitir a realização de testes de qualquer local, utilizei a execução do concorde disponível na web pelo servidor NEOS [5]. Essa abordagem poupou o tempo de instalação e permitiu que os testes fossem realizados em máquinas onde eu não possuía privilégios de administrador.

O artigo [8] fala sobre soluções eurísticas e exatas para TSP; os autores chamam o concorde de "implementação estado da arte" e dizem que ele é um dos melhores solucionadores de TSP atualmente disponíveis. [9] adiciona que o concorde é o solucionador existente mais rápido para instancias grandes. O concorde inclusive ganhou prêmios em 2001 por solucionar problemas de roteamento de veículos propostos em 1996.

O Concorde suporta diversos tipos de cálculo de distância entre pontos, como distância Euclidiana e Geométrica. Essa opção pode ser configurada no arquivo TSPLIB. Para melhorar a precisão na comparação dos resultados e simplificar a implementação dos algoritmos em todos os testes o cálculo de distância Euclidiana foi utilizado.

1.4.1 Avaliação dos Resultados

Para a instância fácil foi utilizado o arquivo "DadosTsp/facil-ulysses22-TSPLIB.tsp" e os resultados são os mesmos visto na imagem 1 (tamanho=72) que encontram no arquivo "resultados/facil-concorde". Para a instância média foi utilizado o arquivo "DadosTsp/medio-eil51-TSPLIB.tsp", os resultados são os mesmos vistos na imagem 2(tamanho=426) e se encontram no arquivo "resultados/medio-concorde". Para a instância difícil, arquivo "DadosTsp/dificil-pr439-TSPLIB.tsp", nenhum outro algorítimo implementado conseguiu resultados tão bons quando o Concorde; esses resultados podem ser vistos na figura 2(tamanho=107217) e no arquivo "resultados/dificil-concorde" com mais detalhes. É importante ressaltar que o concorde obteve resultados muito rápidos para todas as instâncias de TSP.

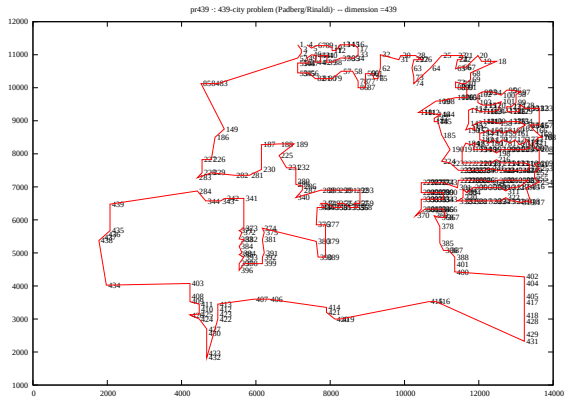

Figura 12: Resultado do Concorde para a instância difícil de TSP

Na próxima sessão iremos comparar os resultados dos algoritmos implementados nesse trabalho entre eles e com o concorde.

1.5 Comparação entre os algoritmos para TSP

A tabela mostrada na figura 13 mostra, de forma resumida, os dados apresentados nas seções anteriores.

	ACO		GA		Sim. Ann.		Concorde	
	Tempo	Tamanho	Tempo	Tamanho	Tempo	Tamanho	Tempo	Tamanho
Fácil	0,1	72	144	85	0,5	85	0.07	72
Médio	1	426	300	700	5	574	0.09	426
Difícil	25	107881	*	*	60	200000	20	107217

Figura 13: Tabela resumida com os resultados dos algoritmos de TSP

1.6 PSO

Nessa etapa do projeto o objetivo era otimizar a função "$F(x, y) = x * sen(4\pi x) - y * sen(4\pi y + \pi) + 1$" com PSO. O algorítimo de PSO foi desenvolvido em Java e os gráficos foram plotados usando a ferramenta Octave [6] (Equivalente ao Matlab para linux), já que não encontrei maneira simples de plotar gráficos 3D em java.

O algoritmo de PSO foi implementado na linguagem Java e completamente orientado a objetos. Foi desenvolvido com vizinhança global e limites de velocidade. Os principais arquivos são:

- **MyPSO.java:** Classe com as funções gerais do PSO. Configurações de velocidade, quantidade de partículas e iterações.
- **MyParticle.java:** Classe com informações das partículas.
- **MyFitnessFunction.java:** Classe que implementa a função a ser maximizada/minimizada. É a responsável por avaliar as partículas.
- **Swarm.java:** Classe que implementa efetivamente o algoritmo de Swarm.

Para executar o programa a maquina virtual Java é necessária. Em seguida basta digitar "java -jar programa-pso.jar". Para plotar os gráficos basta executar o arquivo "plotar.octave" no Octave ou no Matlab, substituindo o vetor X pela saída do programa em Java.

1.6.1 Avaliação dos Resultados

Com aproximadamente 25 partículas e 100 iterações o algoritmo já foi capaz de encontrar um resultado ótimo na função. Na figura 14 vemos a saída do programa em java e na 15 vemos esse resultado plotado no gráfico. Os pontos verdes representam as partículas. O algoritmo foi capaz de encontrar um dos picos ótimos sempre que existia um número satisfatório de partículas espalhadas por uma boa área do gráfico inicialmente. Dependendo da organização inicial da partículas o algoritmo convergia para um pico diferente. Quando os limite de velocidade são retirados as partículas até passam pelo ótimo, mas acabam passando pela velocidade muito alta e acabam se concentrando nas extremidades do gráfico.

```
Iniciando execução
Best fitness: 2.258832300649708
Best position:  [0.6325984332743503, -0.6307767702403388]
Number of evaluations: 2500

Posição das partículas
0.5388621261020519   -0.5239210066496041
0.6289836916689535   -0.7378984452599402
0.695402791688588   -0.598767068678111
0.6288538400247207   -0.7372641903440801
0.6288596356941938   -0.7373920486671754
0.7300243697996843   -0.5420795162665352
0.6930398819129134   -0.5350551476502093
0.6941332859038912   -0.5604472245017542
0.6288672297837908   -0.7376458437489903
0.6288764383409043   -0.736613786344036
0.7172808434460985   -0.640925740244132
0.72129125145166   -0.6193310157437718
0.721232149341126   -0.6014542588822827
0.7212284239627397   -0.6013190436405785
0.7248686960571802   -0.6617273611085994
0.6951505166919849   -0.5950667636855808
0.6288815157327702   -0.7375701186933533
0.6534881141605313   -0.6082060445366113
0.6288673704560457   -0.7376303454645295
```

Figura 14: Saida do algoritmo de PSO

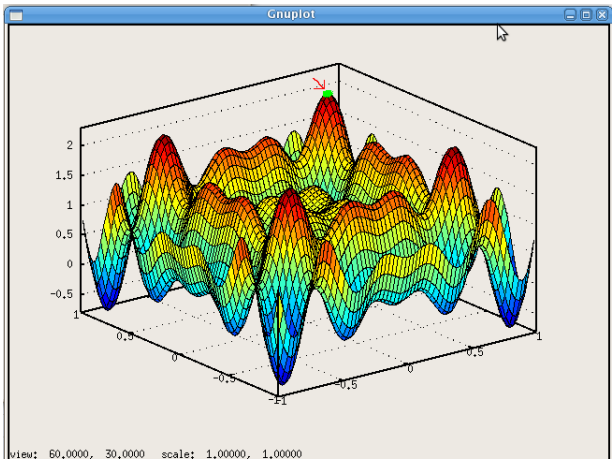

Figura 15: Resultados do PSO plotados com o Octave

Como o algoritmo foi implementado com vizinhança global, todas as partículas convergem para o mesmo pico. Não tive tempo hábil de implementar vizinhança local, mas provavelmente com esse tipo de implementação seria possível identificar vários picos.

1.7 opt-aiNet

Nessa etapa do projeto o objetivo era otimizar a função "$F(x, y) = x * sen(4\pi x) - y * sen(4\pi y + \pi) + 1$" com opt-aiNet. De forma similar ao PSO, o algorítimo de opt-aiNet foi desenvolvido em Java e os gráficos foram plotados usando a ferramenta Octave [6].

O algoritmo de opt-aiNet foi implementado na linguagem Java e orientado a objetos, baseado nos códigos explicados em [2].Os principais arquivos são:

- **Main.java:** Classe com as funções gerais do algoritmo. Inicialização e parâmetros de configuração.
- **OptFunction.java:** Classe que implementa a função a ser maximizada/minimizada. É a responsável por avaliar.
- **NetworkCell.java:** Classe que implementa o comportamento das network cells.
- **OptAinet.java:** Classe que implementa efetivamente o Opt-aiNet.

Para executar o programa a maquina virtual Java é necessária. Em seguida basta digitar "java -jar programa-optainet.jar". Para plotar os gráficos basta executar o arquivo "plotar.octave" no Octave ou no Matlab, substituindo o vetor X pela saída do programa em Java.

1.7.1 Avaliação dos Resultados

Para os testes os seguintes parâmetros foram utilizados:

- **numCells**=20.
- **numClones**=10.
- **maxIter**=500.
- **suppThres**=0,2.
- **errorThres**=0,001.
- **divRatio**=0,4.
- **mutnParam**=100.

Nos testes foi possível verificar diversas propriedades interessantes do opt-aiNet como a manutenção da diversidade e a auto-adaptação na quantidade de cells. Graças a essas características o sistema foi capaz de identificar os picos quase que na sua totalidade. Na figura 16 vemos o resultado, onde cada ponto verde representa uma cell.

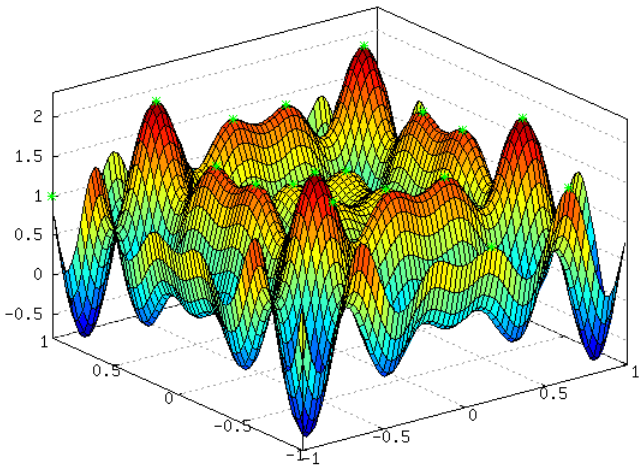

Figura 16: Resultados do opt-aiNet plotados com o Octave

1.8 ARIA

O objetivo nessa etapa do projeto era implementar um algoritmo de agrupamento de dados utilizando ARIA. Efetuei tentativas com o Octave e com Java mas mas consegui obter resultados em tempo hábil. O algoritmo para o Octave está com problemas e não consegui concluir o em java. Ambos estão na pasta "ARIA".

2. CONCLUSÕES

Este trabalho permitiu verificar de maneira prática, os conceitos e características de diversos algoritmos estudados em sala de aula. TODO: Relembrar os principais comentários dos resultados

3. REFERENCES

[1] Aco - iridia. http://iridia.ulb.ac.be/ mdorigo/ACO/aco-code/public-software.html.

[2] Aisweb. http://www.artificial-immune-systems.org/.

[3] Jfreechart. http://www.jfree.org/jfreechart/.
[4] Jgap. http://jgap.sourceforge.net/.
[5] Neos server. http://www-neos.mcs.anl.gov/neos/solvers/co:concorde/TSP.html.
[6] Octave. http://www.gnu.org/software/octave/.
[7] Tsplib. http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/.
[8] M. Hahsler and K. Hornik. Tsp – infrastructure for the traveling salesperson problem. *Journal of Statistical Software*, 2007.
[9] S. A. Mulder and I. Donald C. Wunsch. Million city traveling salesman problem solution by divide and conquer clustering with adaptive resonance neural networks. *Neural Netw.*, 16(5-6):827–832, 2003.