Universidade Estadual de Maringá
Departamento de Informática
Curso: Informática
Disciplina: 5199 – Modelagem e Otimização Algorítmica
Professor: Ademir Constantino

Relatório do Trabalho

Aluno: Henrique Possidonio Silverio **RA:** 82077
Vitor Hugo E. do Nascimento **RA:** 85922

Maringá
2016

Sumário

Introdução

O presente trabalho consistiu em desenvolver um **Algoritmo Genético** na linguagem *Python*, o que consiste em um **Algoritmo Construtivo e Melhorativo** e em especial nesse trabalho, adaptado para o problema do Caixeiro Viajante. A implementação deverá permitir entrar com uma população inicial e o objetivo é melhorar a população em cada iteração afim de que se chegue a um valor aproximado do melhor conhecido até então. O programa deve ler várias linhas, a primeira sendo a quantidade de vértices que terá a população e o restante sendo os vértices, o primeiro número é o identificador do vértice e os outros dois a coordenada X e coordenada Y do mesmo no gráfico, após isso serão geradas N soluções (caminhos) que serão melhoradas ao longo da execução.

Resolução do Problema

O **Algoritmo Genético** implementado constitui-se de várias etapas para chegar ao resultado final, se baseando em uma solução (caminho completo no grafo) representada por um vetor, onde cada elemento do mesmo corresponde a um vértice informado na entrada do programa, sendo elas:

1. **Geração**: Etapa responsável por gerar a população inicial, de maneira aleatória e conforme as informações de entrada do programa.

2. **Avaliação**: Nessa etapa é quantificado a qualidade dos cromossomos, conhecida como aptidão, corresponde a uma função de custo conforme a coordenada X e coordenada Y de cada vértice presente na solução, representada pelo cálculo:

$$\sqrt{(\mathbf{coordenada\ Xa} - \mathbf{coordenada\ Xb})^2 + (\mathbf{coordenada\ Ya} - \mathbf{coordenada\ Yb})^2}$$

3. **Seleção**: Nessa etapa é escolhido quais indivíduos participarão da criação do cromossomo (cruzamento) conforme o resultado da função apresentada no item anterior. Em específico na nossa implementação é escolhido dois indivíduos, e entre ambos, quem possuir o menor custo é o selecionado, método conhecido como *Seleção por Torneio*.

4. **Cruzamento**: Com os dois indivíduos selecionados na etapa anterior é feito o cruzamento dos mesmos, gerando então um novo cromossomo, com a tendência de possuir características melhores que a de seus geradores. Em nossa implementação foi utilizado o *método OX*, que constrói filhos escolhendo subsequência de um pai e preservando a ordem relativa do segundo.

5. **Mutação**: Essa operação tem a responsabilidade de inverter aleatoriamente alguma característica do cruzamento gerado no passo anterior, afim de diversificar o mesmo diante dos outros indivíduos da população. A ideia utilizada no nosso trabalho foi de gerar dois valores randômicos e alterar os elementos entre as posições, porém essa mutação é feita de maneira aleatória, conforme um valor gerado estar dentro de uma taxa pré-definida.

6. **Busca Local**: Etapa responsável por gerar soluções vizinhas (aproximadas) do indivíduo em questão (sendo o cruzamento), afim de se obter um outro com menor custo. Em nossa implementação foi utilizada a lógica do *algoritmo 2-opt*, alterando dois elementos adjacentes entre si para gerar a vizinhança.

7. **Atualização**: Essa etapa é responsável por atualizar a população com o cruzamento gerado e mutado nos passos 3 e 4, é comparado o pior indivíduo da população, e caso o mesmo possua um custo

maior do que o do cruzamento gerado, ele é substituído pelo cruzamento. Dessa forma a população se mantem atualizada com os melhores indivíduos. Em nossa implementação foi utilizada a estratégia de *Elitismo*.

A cada iteração do algoritmo é gerado uma nova parte da população, substituindo os N piores os piores indivíduos existentes na população, dando então a característica de melhoramento ao algoritmo.

Esse processo é feito até que se chegue a um critério de parada definido, podendo ser:

- Tempo de Execução;
- Número fixo de iterações;
- Avaliação melhorativa dos cromossomos gerados.

Após isso é esperado que o custo da solução (indivíduo) seja a mais aproximada possível da melhor solução conhecida (apresentada no documento “Terceiro Trabalho”, postado no moodle).

Implementação

Explicado o problema, nos baseamos nos exemplos realizados durante as aulas para desenvolver uma solução em *Python*.

A construção foi dividida, basicamente em 3 partes:

1. **BLOCO CLASSE E VARIÁVEIS**: Bloco onde está definido as classes dos objetos Vertice e Solucao e as variáveis que serão utilizadas ao longo do programa.
    a. Explicação dos atributos da classe Vertice:
        i. **Número:** Representado pelo atributo numero, onde irá armazenar o número do vértice, conforme será passado na entrada do programa.
        ii. **Coordenada X:** Representado pelo atributo coordenadaX, utilizado para armazenar o valor da coordenada X do vértice no gráfico.
        iii. **Coordenada Y:** Representado pelo atributo coordenadaY, utilizado para armazenar o valor da coordenada Y do vértice no gráfico.

    b. Explicação dos atributos da classe Solucao:
        i. **Caminho**: Representado pelo atributo caminho, utilizado para guardar a sequência de vértice que compõe o caminho, que pode ser entendido como uma solução.
        ii. **Aptidão**: Representado pelo atributo aptidão, utilizado para guardar o custo de cada solução, para não ter retrabalho de calcular o custo de uma solução a todo o momento.

    c. Explicação das variáveis que serão utilizadas ao longo do programa:
        i. **Conjunto dos Vértices:** Representado pela variável listaVertice, irá armazenar os vértices que foram informados na entrada do programa.
        ii. **Tamanho do Conjunto**: Representado pela variável tamanhoConjunto, utilizado para armazenar o primeiro valor da entrada, sendo ele o tamanho do conjunto de vértices.
        iii. **Informações do vértice**: Representado pela variável entradaDeDados, irá armazenar as informações dos vértices dadas na entrada do programa, sendo o número, coordenada X e coordenada Y do mesmo.
        iv. **Sequência de passos**: Representado pela variável sequenciaDePassos, irá armazenar uma sequência de 1 até o tamanho do conjunto e será utilizada para geração da população inicial.
        v. **População**: Representado pela variável populacao, irá armazenar a população que será gerada de forma aleatória conforme o tamanho do conjunto.
        vi. **Tamanho da População**: Representado pela variável tamanhoPopulacao, valor fixo para definir o tamanho da população que será gerada.

vii. **Controle da melhora da População**: Representado pelas variáveis numeroTentativas e melhorouSolucao, o valor é incrementado a cada vez que o cruzamento gerado não possuir um custo menor que o pior elemento da população, ou seja, significa que não melhorou a população, sendo assim ao atingir um valor fixo definido a variável melhorouSolucao é setada para False fazendo com que o algoritmo se encerrado.

2. **BLOCO DEFINIÇÃO DAS FUNÇOES**: Bloco onde está definido a implementação das funções que serão utilizadas no programa principal.
   a. **Geração da População:** Representado pelo método geraPopulacao(), é feita uma iteração da posição 0 até o tamanho da população e embaralhado o vetor dos caminhos, sendo assim a cada iteração é gerado um novo indivíduo (solução).
   b. **Seleção das Rotas**: Representado pelo método selecionaRotas(), passado como parâmetro a População e a partir da mesma é selecionado dois indivíduos, e dentre os dois é selecionado o que possui menor custo.
   c. **Geração do Cruzamento**: Representado pelo método geraCruzamento(), recebendo como parâmetro dois indivíduos “pais”, a partir dos mesmos é feito a lógica do algoritmo OX e gerado um novo filho, conhecido como crossover.
   d. **Mutação do Cruzamento**: Representado pelo método mutacaoCaminho(), recebendo como parâmetro o caminho do Cruzamento gerado no passo anterior e alterado o elemento de duas posições aleatórias, dando então mais diferenciação ao cruzamento gerado, diminuindo o risco de ser igual a um outro cruzamento ou a seus pais.
   e. **Busca Local**: Representado pelo método buscaLocal(), recebendo como parâmetro o cruzamento que foi gerado e a partir do mesmo são geradas soluções vizinhas afim de se obter um indivíduo melhor, consequentemente melhorando o cruzamento que foi gerado.
   f. **Geração dos Vizinhos**: Representado pelo método geraVizinho(), recebendo como parâmetro o cruzamento e um contador sequencial, o mesmo irá utilizar o contador sequencial para alterar elemento de duas posições do cruzamento, realizando a lógica do algoritmo 2-opt, gerando novos vizinhos. O intuito do mesmo é encontrar uma solução melhor na vizinhança comparado com a solução em questão.
   g. **Atualização da População**: Representado pelo método atualizaPopulacao(), recebe como parâmetro a população e o cruzamento, nessa etapa é substituído o pior indivíduo da população pelo cruzamento que foi gerado, melhorando a população a cada iteração.

3. **BLOCO PROGRAMA PRINCIPAL**: Bloco onde está definido a lógica do programa principal que chama as funções apresentadas anteriormente.
   As funções são chamadas na seguinte sequência:
   I. geraPopulacao()
   II. selecionaRotas()
   III. geraCruzamento()
   IV. mutacaoCruzamento()
   V. buscaLocal()
   VI. atualizaPopulacao()

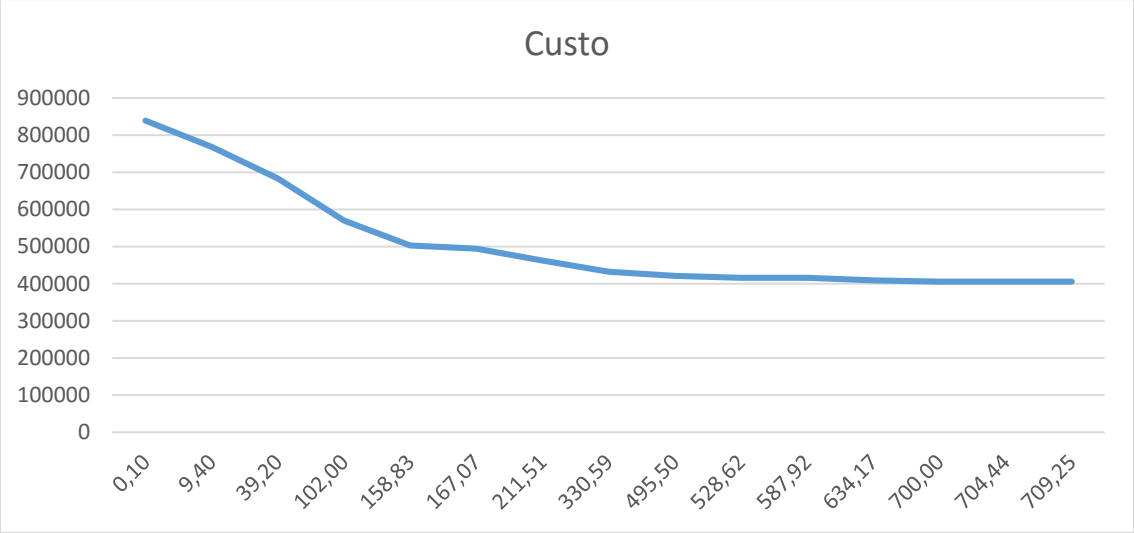
Análise dos Algoritmos

As análises descritas abaixo são das funções principais do programa.

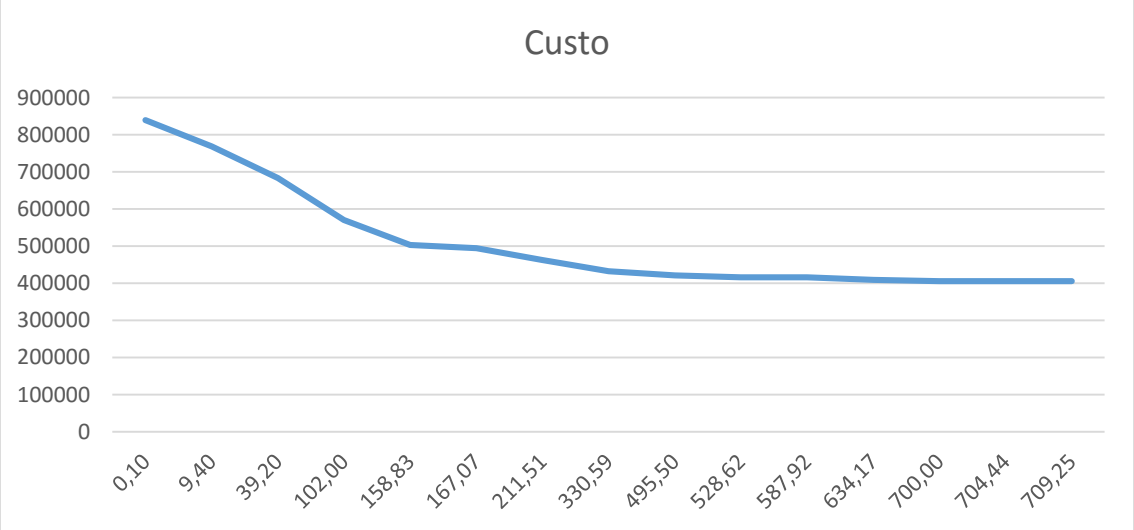
- **geraPopulacao()**: O método é chamado apenas uma vez na execução do programa e executado internamente a quantidade de vezes sendo o tamanho da população, logo possui um custo O(tamanho da população).
- **selecionaRotas():** O método é executado duas vezes a cada iteração do algoritmo e internamente calcula o custo de dois indivíduos selecionados da população, a função de cálculo do custo possui um custo O(tamanho da população).
- **geraCruzamento()***:* O método é executado uma vez a cada iteração do algoritmo e internamente possui um custo constante, podendo ser representado por O(1).
- **mutacaoCruzamento():** Não é controlado a quantidade de vezes que o método será executado, pois a cada iteração é gerado um número aleatório e verificado se está dentro de uma faixa pré-definida, caso esteja executa, caso não esteja não executa, e internamente o método possui um custo de O(1).
- **geraVizinho():** O método é chamado pelo buscaLocal() que é executado a cada iteração do algoritmo, e o mesmo executa a quantidade de vezes sendo o tamanho da população, e seu custo interno é O(1).
- **buscaLocal():** Assim como descrito anteriormente o método é executado a cada iteração do algoritmo e composto pela quantidade de chamadas sendo o tamanho da população do método geraVizinho().
- **atualizaPopulacao():** O método é executado a cada iteração do algoritmo e possui um custo de O(tamanho da população).
- **obtemMenor():** O método é executado apenas uma vez na execução do programado e internamente chama o método custo() da classe Vertice, logo é composto pelo custo desse método sendo O(tamanho da população) e seu custo interno sendo O(1), por conta do acesso direto ao vetor.

Resultado dos testes

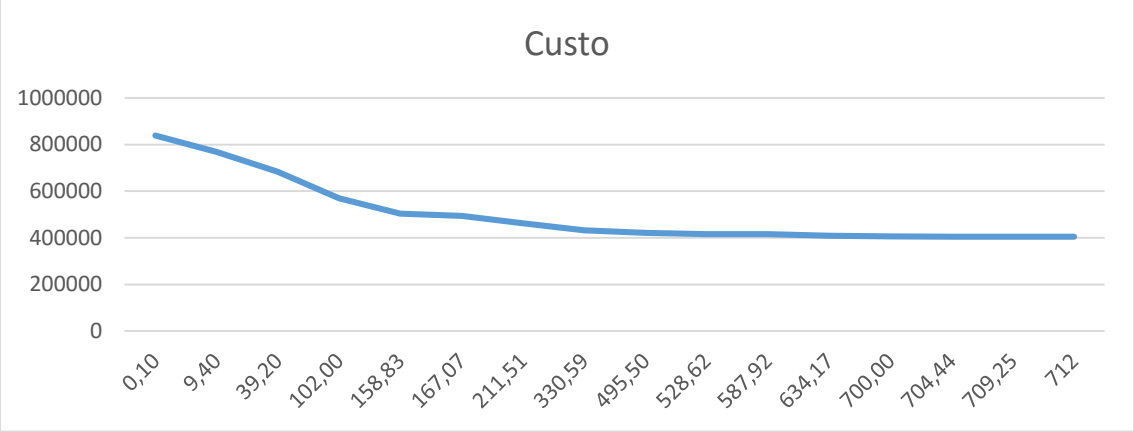
att48



att 532



d657



Caso	MS	Alg	Alb_bl	GAP%
att48	10.628	47.887	40.521	281.27
att532	27.686	446.713	435.189	1471.88
d657	48.912	287.865	237.529	385.63

Legenda:

- Caso: colocar o nome do caso, exemplo:
- *MS*: Melhor solução conhecida da literatura.
- *Alg*: Melhor solução obtida com o seu algoritmo.
- *Alg_bl*: Melhor solução obtida com a combinação do seu algoritmo com busca local.
- *GAP*%: é um parâmetro de desvio relativo calculado da seguinte forma:

$$GAP\% = \frac{\text{Alg\_}bl - MS}{MS} 100$$

Conclusão

Primeiramente, vale ressaltar a importância de se realizar um estudo para a escolha de uma estrutura de dados mais adequada para o problema.

Fundamentalmente nesse trabalho, essa situação fica evidente ao se utilizar estratégias de implementação como estrutura de heap, randomização e também as estratégias oferecidas pelo algoritmo genético, como por exemplo, a seleção dos indivíduos para geração da nova solução, em nosso caso foi utilizado a Seleção por Torneio, ambas as estratégias visam obter um melhor resultado no menor tempo possível.

Vale ressaltar a importância de utilizar a melhor combinação dos parâmetros, por exemplo, quanto maior a população maior o tempo de execução, porém a probabilidade de repetição de soluções é menor; quanto maior a taxa de mutação o tempo é menor, porém o custo da solução obtida é maior; quanto maior a quantidade de tentativas para melhorar a solução mínima da população melhores resultados são obtidos, porém o tempo de execução acaba sendo maior.

A utilização da busca local exige mais processamento, com isso não é adequada para todas as situações, mesmo obtendo uma solução melhor a partir dos vizinhos, pois quanto maior a população, maior o tempo de processamento. Dentre as estratégias de First Improvement e Best Improvement, em nosso programa a que trouxe um melhor resultado foi a First Improvement combinando com a lógica do algoritmo 2-opt para geração da vizinhança.

Por fim, podemos colocar as maneiras de atualização da população, podendo ser Populacional, Elitismo ou Steady Stated, em nosso caso a situação que mais se adequou foi o Elitismo.

Referências

1. Notas de Aula – Modelagem e Otimização de Algoritmos. Moodle DIN UEM. Disponível em <http://moodle.din.uem.br/course/view.php?id=236>. Acesso em Julho de 2016.
2. < https://pt.wikipedia.org/wiki/Algoritmo_gen%C3%A9tico>. Acesso em Julho de 2016.
3. < http://www.icmc.usp.br/~andre/research/genetic/>. Acesso em Julho de 2016.
4. <https://docs.python.org/2/library/index.html>. Acesso em Julho de 2016.
5. Dúvidas de implementação consultadas em <http://stackoverflow.com/>. Acesso em Julho de 2016.