



PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS
Instituto de Ciências Exatas e de Informática

DOCUMENTAÇÃO TRABALHO PRÁTICO

PROBLEMA DO CAIXEIRO VIAJANTE

Bruno Ribeiro Gonzaga Silva
Lucas Felipe Silva Carvalho
Pedro Henrique Silva Egg

Belo Horizonte
2020

Bruno Ribeiro Gonzaga Silva
Lucas Felipe Silva Carvalho
Pedro Henrique Silva Egg

DOCUMENTAÇÃO TRABALHO PRÁTICO

PROBLEMA DO CAIXEIRO VIAJANTE

Trabalho apresentado à
disciplina de Projeto e Análise de
Algoritmos do departamento de Ciência
da Computação da Pontifícia
Universidade Católica de Minas Gerais,
ministrada pela professora Raquel
Aparecida de Freitas Mini.

Belo Horizonte
2020

Sumário

Introdução	4
Implementação	5
Algoritmo de Força Bruta	6
Algoritmo Branch and Bound	6
Algoritmo de Programação Dinâmica	7
Algoritmo Genético	7
Análise de Complexidade	9
Algoritmo de Força Bruta	9
Melhor e Pior Caso	9
Algoritmo Branch and Bound	9
Melhor Caso	10
Pior Caso	10
Algoritmo de Programação Dinâmica	10
Melhor Caso	10
Pior Caso	11
Algoritmo Genético	11
Melhor Caso	11
Pior Caso	11
Testes	12
Conclusão	17
Bibliografia	18
Anexos	19

1. Introdução

Na ciência da computação, o problema do caixeiro viajante é um problema que tenta determinar a menor rota para percorrer uma série de cidades e retornar à cidade de origem. Para um número pequeno de cidades, qualquer implementação é válida e nos entrega uma resposta em um tempo satisfatório, porém ao aumentar minimamente o tamanho da entrada, encontramos uma complexidade exponencial que inviabiliza qualquer tipo de solução em um tempo humanamente possível. Por se tratar de um problema exponencial, criar algoritmos mais eficientes pode ajudar, mesmo que pouco, a execução em alguns casos.

Esse trabalho tem o objetivo de implementar e discutir sobre algoritmos para a resolução do problema do caixeiro viajante seguindo os paradigmas de Força Bruta, Branch and Bound, Programação Dinâmica e Algoritmo Genético. Realizaremos testes e iremos comparar os tempos de execução e as complexidades dos algoritmos, a fim de analisar as diferentes abordagens para o problema e presumir qual o melhor modo de enfrentá-lo.

Além disso, testaremos nossas capacidades para enfrentar, projetar a analisar problemas usando diferentes abordagens, a fim de aprimorar nossos conhecimentos em projeto e análise de algoritmos.

2. Implementação

Os algoritmos implementados foram feitos usando a linguagem de programação C++ utilizando como IDE o Visual Studio Code.

Como parte do desenvolvimento foi usado o repositório GitHub para versionamento e controle de código entre os integrantes do grupo.

Todos os testes foram realizados no seguinte ambiente:

- Sistema Operacional: Manjaro Linux 64bits
- Processador: 4 x Intel Core i5 7200U 2.50GHz
- Memória: 8GB
- Compilador: g++

A execução do programa deve seguir os seguintes comandos:

- g++ caixeiroViajante.cpp cidade.cpp main.cpp -o main
- ./main < inputs/test1.txt > outputs/test1.txt

Podemos escolher 1 dentre os 5 arquivos de teste, gerados aleatoriamente, para realizar as nossas execuções.

O programa executa todos os 4 algoritmos (Força Bruta, Branch and Bound, Programação Dinâmica e Algoritmo Genético), calculando individualmente o tempo de cada um e nos retorna uma saída como a seguinte:

```
[pedroegg@Manjaro-Egg TRABALHO_PRATICO_PAA]$ ./main < inputs/test4.txt
Força Bruta
Distancia: 2493.17
Reposta:  1 6 7 3 2 8 9 5 10 4
Tempo: 1.31672

Dinamico
Distancia: 2493.17
Reposta:  1 6 7 3 2 8 9 5 10 4
Tempo: 0.901091

Branch And Bound
Distancia: 2493.17
Reposta:  1 6 7 3 2 8 9 5 10 4
Tempo: 1.08166

Genetico
Distancia: 2619.33
Reposta:  1 7 3 2 8 9 5 10 4 6
Tempo: 0.129637
```

(Imagem 1: Exemplo de saída)

2.1. Algoritmo de Força Bruta

O algoritmo de força bruta implementado consiste na geração de todas as combinações possíveis de caminhos que, saindo de uma cidade, passa por todas as outras cidades e retorna à cidade inicial. A geração dessas permutações respeita as regras de não repetir nenhum caminho nem cidade já visitada e do caminho sempre começar e terminar na cidade 1.

Após a geração de todos os caminhos e o resultado salvo em uma lista, percorremos essa lista, e para cada item calculamos a distância de se percorrer aquele caminho. Comparamos a distância encontrada em cada item com a menor já registrada anteriormente. Ao final do algoritmo, teremos como resposta a distância e a ordem que as cidades devem ser visitadas para que o menor caminho seja feito.

Por se contar de um algoritmo de força bruta, o resultado encontrado, é de fato, o melhor caminho possível que podemos encontrar, pois testamos todos os caminhos possíveis para isso. Porém isso traz um custo enorme, de tempo e memória, que discutiremos mais a fundo na seção *Análise de Complexidade*.

2.2. Algoritmo Branch and Bound

O algoritmo de branch and bound consiste em testar todos os possíveis caminhos, saindo de uma cidade, passando nas outras cidades e voltando para a cidade inicial sem repetir nenhuma cidade.

Porém a diferença dele para o algoritmo de força bruta é que a medida que ele vai calculando a distância entre as cidades, o algoritmo verifica se a soma das distâncias das cidades já passou a última distância mínima calculada. Caso a soma já tenha passado, ele passa para a próxima sequência. Caso contrário ele continua o cálculo das distâncias mas sempre verificando se já não passou da última distância mínima encontrada.

Ao final o algoritmo vai ter encontrado o melhor caminho possível, pois vai ter testado todas as combinações. Porém ele vai se sair um pouco melhor que o

algoritmo de força bruta pois não vai calcular todas as combinações mas ainda sim tendo um alto custo de tempo e de memória.

2.3. Algoritmo de Programação Dinâmica

O algoritmo feito utilizando programação dinâmica se assemelha com o de força bruta, porém com uma pequena porém considerável mudança. Sua implementação consiste em testar todos os caminhos, porém utilizando-se de uma estrutura de dados auxiliar, armazenar valores de caminhos já calculados anteriormente, que não necessitam de serem calculados novamente ao serem encontrados.

A lista de caminhos possíveis é feita usando o mesmo método de permutação que o algoritmo de força bruta utiliza. Após a geração da lista realizamos o cálculo da distância dos pares de cidades que representam o caminho, e salvamos esse valor em um mapa, que possui o ID do caminho como chave e a distância como valor. Nas primeiras vezes, o algoritmo se assemelha ao força bruta, porém ao decorrer das iterações, iremos encontrar pares de cidades que já tem suas distâncias calculadas no mapa gerado, e portanto evitaremos de fazer novamente os cálculos.

De qualquer forma, ainda testamos todos os caminhos, portanto, como resposta temos o menor caminho possível para o conjunto de cidades, mas diferente do algoritmo de força bruta, temos uma otimização em operações custosas, acelerando levemente a execução do algoritmo em troca de maior uso de memória.

2.4. Algoritmo Genético

O algoritmo genético criado para a resolução do problema se difere dos demais por se contar de uma heurística, portanto o resultado encontrado pode não ser o melhor possível, mas o seu ganho em performance justifica uma resposta boa o suficiente.

O algoritmo consiste em gerar uma quantidade específica de caminhos aleatórios, chamado de população, calculamos a distância que cada indivíduo (caminho) dessa população tem, a partir dessa distância geramos um vetor de 'pontuação' dos indivíduos, quanto menor a distância desse caminho, maior a sua pontuação. Armazenamos também o menor caminho e a menor distância encontrados até então.

A partir do vetor de pontuação, escolhemos aleatoriamente um dos elementos, tendo maiores chances de ser escolhido um elemento que possui uma pontuação maior. Ao escolher um elemento fazemos uma mutação nele; que consiste em trocar 2 elementos de posição entre si (Ex: [1,2,3,4,5] após uma mutação pode se transformar em [2,1,3,4,5]).

Com isso, geramos uma nova população de caminhos aleatórios, porém um desses caminhos é o caminho escolhido na geração anterior. Para essa nova geração repetimos o processo anterior até que o número de gerações tenha chegado ao fim.

Ao final do algoritmo teremos o melhor resultado dentre todas os indivíduos de todas as gerações.

Esse algoritmo possui algumas variáveis de controle, são elas: Tamanho da População, Quantidade de Randomização e Quantidade de Gerações.

O Tamanho da População representa a quantidade de indivíduos que as populações possuem. A Quantidade de randomização representa a quantidade de randomizações que uma nova população terá na sua geração. A Quantidade de gerações representa quantas ciclos serão feitos e quantas populações serão criadas antes do fim do algoritmo.

O custo desse algoritmo é extremamente baixo, comparado aos demais citados acima, discutiremos mais dele na seção *Análise de Complexidade*.

3. Análise de Complexidade

Os algoritmos desenvolvidos neste trabalho possuem abordagens diferentes para um mesmo problema, portanto temos implementações diferentes que utilizam estruturas de dados diferentes. Para conseguirmos prever o custo de um deles dado uma entrada, devemos fazer a análise de complexidade computacional.

Para o problema do caixeiro viajante, tomamos como operação relevante a chamada da função *calcularDistancia*, que calcula a distância entre 2 cidades.

3.1. Algoritmo de Força Bruta

O algoritmo de força bruta possui um custo fatorial $\Theta(n!)$ para ser executado, sendo n o número de cidades da entrada, pois devemos passar em todas as permutações de n , gerando $n!$ permutações possíveis e diferentes. Para cada uma das permutações calculamos a distância entre todos os pares das n cidades do caminho, resultando em um algoritmo com complexidade $\Theta(n * n!)$.

3.1.1. Melhor e Pior Caso

O melhor e o pior caso deste algoritmo são os mesmos, pois independente da configuração da entrada, teremos as mesmas operações sendo executadas.

- Melhor Caso: $\Theta(n * n!)$
- Pior Caso: $\Theta(n * n!)$

3.2. Algoritmo Branch and Bound

O algoritmo de branch and bound possui um custo parecido com o custo do algoritmo de força bruta. Pois ele monta e testa todas as combinações possíveis, com a diferença de que ele faz uma verificação se a soma das distâncias das cidades já não passou da distância mínima atual. Com isso o algoritmo tem uma complexidade de $\Theta(n * n!)$.

3.2.1. Melhor Caso

O melhor caso do algoritmo branch and bound é quando a primeira combinação testada tem o melhor caminho. Com isso ele irá chamar menos vezes a função de calcular a distância entre as cidades.

3.2.2. Pior Caso

O pior caso do algoritmo é o mesmo caso do algoritmo de força bruta. Pois pode ter uma lista de combinações de rotas onde o melhor caminho somente será encontrado na última combinação.

3.3. Algoritmo de Programação Dinâmica

O algoritmo feito com programação dinâmica possui o custo de complexidade aproximado do algoritmo de força bruta. Realizamos, para cada uma das $n!$ permutações e para n cidades do caminho de uma permutação, o cálculo de distância, porém se o par entre as duas cidades já foi calculado, não necessitamos de realizar o cálculo novamente.

Em compensação, essa implementação utiliza uma lista que armazena todos os pares ($2 * n$) de cidades, aumentando o custo de alocação de memória.

Complexidade: $\Theta(n * n!)$.

3.3.1. Melhor Caso

O melhor caso deste algoritmo acontece quando as entradas são maiores, pois acontece mais repetições entre os pares, reduzindo a quantidade de cálculos de distâncias realizados.

3.3.2. Pior Caso

O pior caso acontece quando a entrada diminui, pois teremos menos pares repetidos, em relação ao total de combinações, portanto teremos um número maior de cálculos sendo executado.

3.4. Algoritmo Genético

A análise de complexidade do algoritmo genético leva em consideração algumas variáveis. São elas: Tamanho da população (TP), Quantidade de randomizações (QR), Quantidade de gerações (QG) e o Número de cidades (n).

A variável QG nos informa a quantidade de vezes que o algoritmo será executado, e em cada iteração será criado uma população de tamanho TP. Para cada elemento da população calcularemos a distância entre os seus pares de n cidades.

Ao final, chegamos que o algoritmo possui como complexidade: $\Theta(QG * TP * n)$.

3.4.1. Melhor Caso

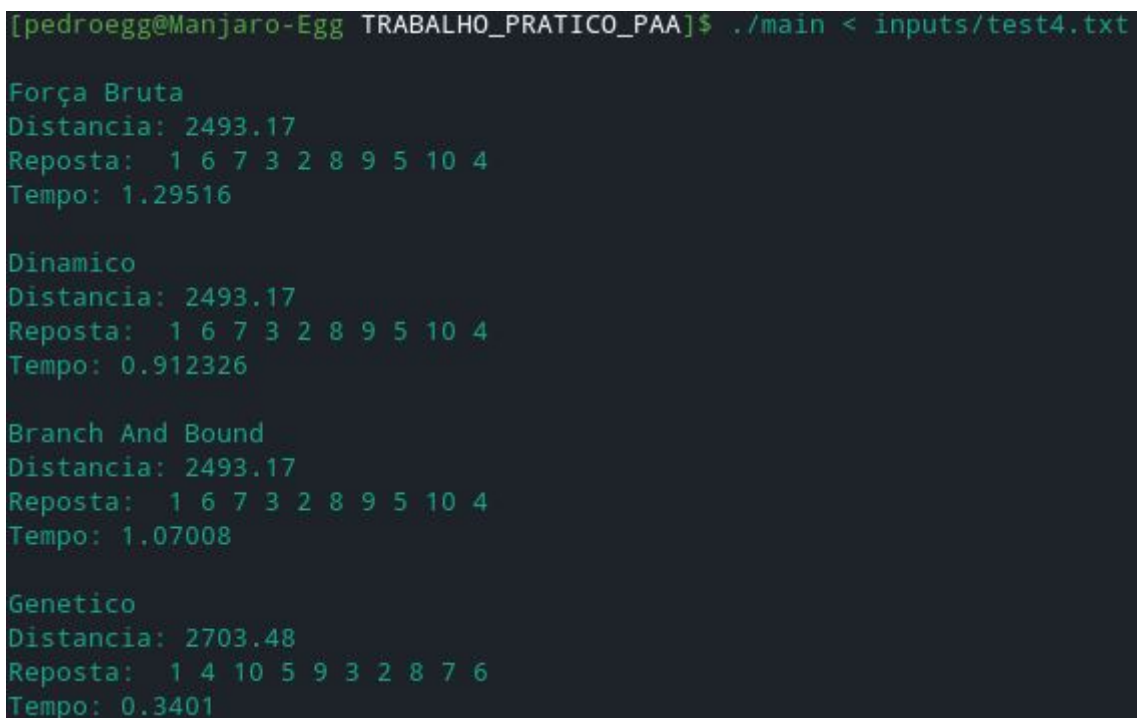
O melhor caso do algoritmo genético é quando tem um equilíbrio entre o tamanho da população e o número de cidades. Com isso o algoritmo irá mostrar um resultado mais satisfatório e não terá feito tantas comparações como o algoritmo de força bruta.

3.4.2. Pior Caso

O pior caso do algoritmo genético seria quando o número da população é o resultado de do fatorial do número de cidades. Com isso ele passa a ter o mesmo desempenho do algoritmo de força bruta.

4. Testes

Os testes foram realizados usando arquivos de testes com um número variável de cidades e valores entre eles. Ao executar o programa dando como entrada algum dos arquivos de testes, ele faz os cálculos necessários e retorna a resposta através do terminal/prompt de comando da seguinte forma:



```
[pedroegg@Manjaro-Egg TRABALHO_PRATICO_PAA]$ ./main < inputs/test4.txt

Força Bruta
Distancia: 2493.17
Reposta:  1 6 7 3 2 8 9 5 10 4
Tempo: 1.29516

Dinamico
Distancia: 2493.17
Reposta:  1 6 7 3 2 8 9 5 10 4
Tempo: 0.912326

Branch And Bound
Distancia: 2493.17
Reposta:  1 6 7 3 2 8 9 5 10 4
Tempo: 1.07008

Genetico
Distancia: 2703.48
Reposta:  1 4 10 5 9 3 2 8 7 6
Tempo: 0.3401
```

(Imagem 2: Exemplo de teste)

O programa retorna a distância do menor caminho total encontrado, o caminho encontrado e o tempo total levado para executar a função e os processos.

Os arquivos de teste são escritos no formato seguinte, mostrando como por exemplo o arquivo de teste “test4.txt” usado na imagem anterior:

```

10
909 775
71 176
139 206
802 345
414 252
755 770
329 540
301 10
357 159
478 251

```

(Imagem 3: Exemplo de arquivo de entrada)

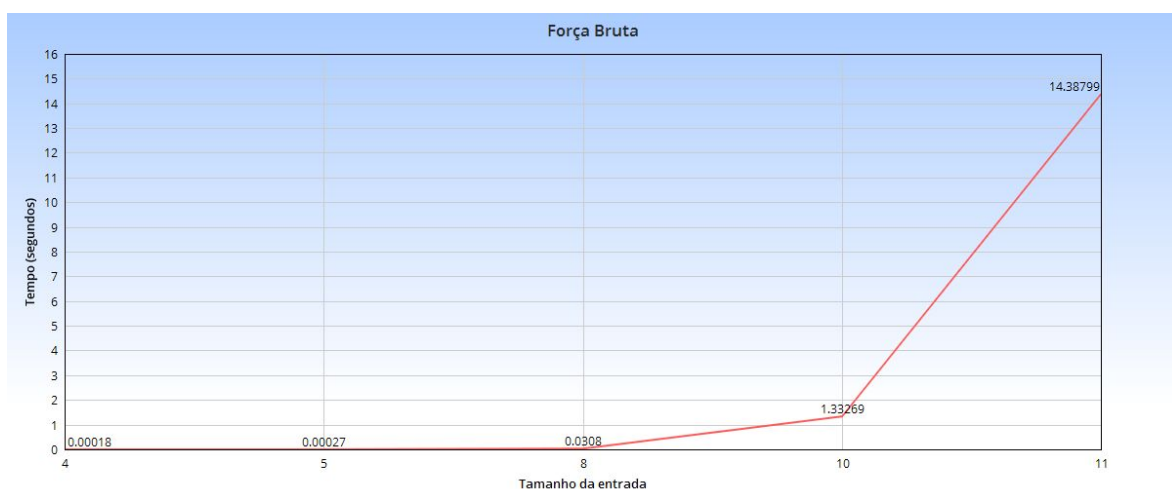
Em que 10 representa o número de cidades e as linhas seguintes são as coordenadas X e Y das respectivas cidades.

Para a montagem do gráfico, foi executado 10 testes para cada arquivo de teste com diferentes números de cidades e anotado esses valores para o cálculo da média e geração de gráficos.

Os arquivos usados como teste, possuem respectivamente 4, 5, 8, 10 e 11 cidades, e quanto maior o número de cidades, maior o tempo levado para calcular o resultado, e consequentemente, maior a média.

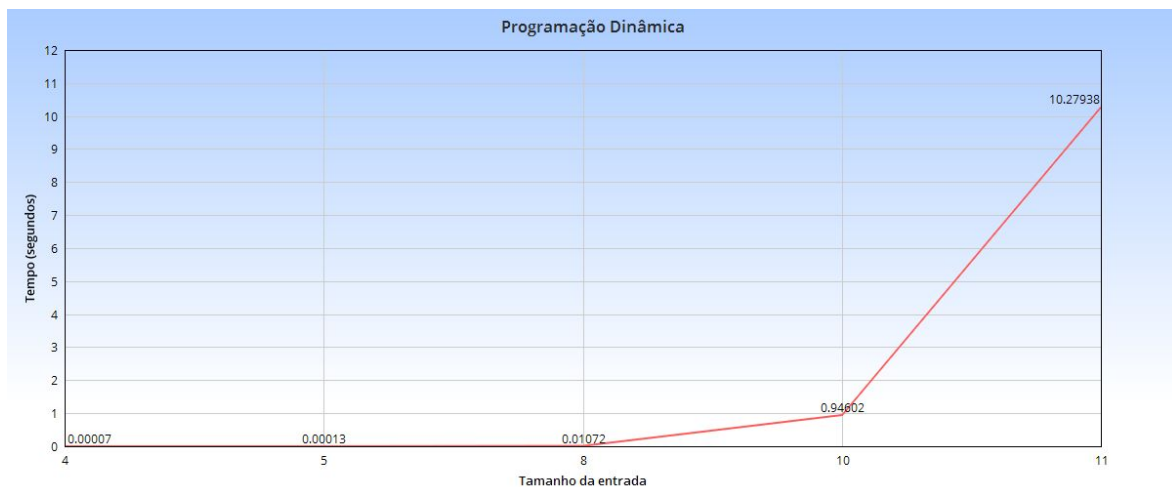
Realizando comparações e análises sobre os resultados obtidos, pode-se afirmar uma série de informações sobre os algoritmos e suas utilizações:

1. O algoritmo de Força Bruta, pelo fato de testar todos caminhos gerados pela permutação, sempre sai com um custo maior do que todos os outros, uma vez que é necessário comparar caminho por caminho, o que é bem custoso, ou seja, quanto maior a entrada, maior o tempo que vai ser levado.



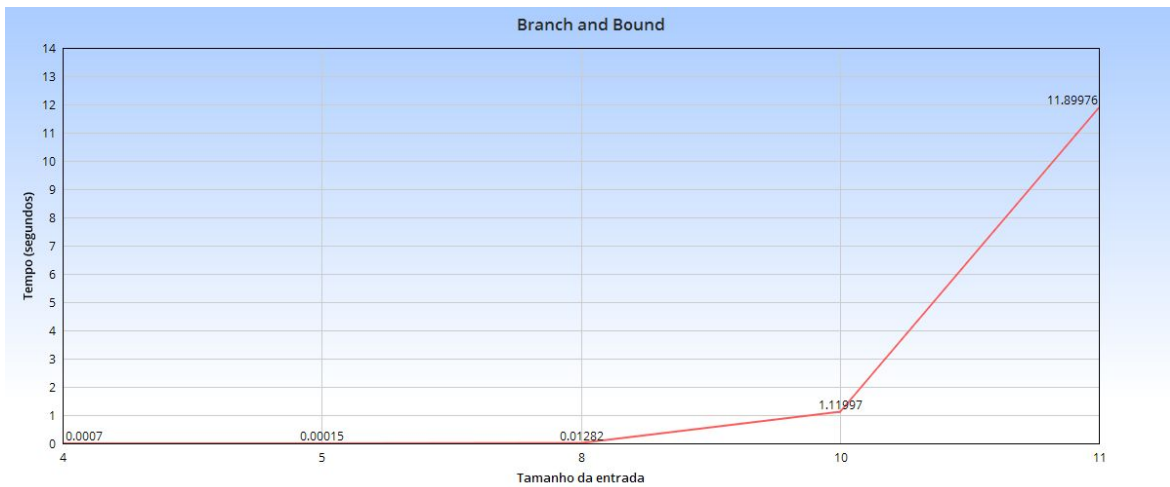
(Imagem 4: Crescimento de tempo no algoritmo Força Bruta)

2. O algoritmo Dinâmico, em geral, se sai melhor na comparação entre todos os outros (com exceção do genérico), pois ele utiliza da estratégia de **não** calcular a distância entre duas cidades se essa distância já foi calculada anteriormente. Com isso, é possível deixar de calcular vários caminhos, pois quanto maior a entrada, mais repetições daquela mesma fração de caminho irá ocorrer, o que vai poupar bastante tempo, ou seja, quanto maior a entrada, melhor ele se sai em comparação com os outros algoritmos (com exceção do genético). Porém, o algoritmo dinâmico utiliza uma maior quantidade de memória, visto que ele armazena as distâncias já calculadas dentro de uma matriz.



(Imagem 5: Crescimento de tempo no algoritmo Programação Dinâmica)

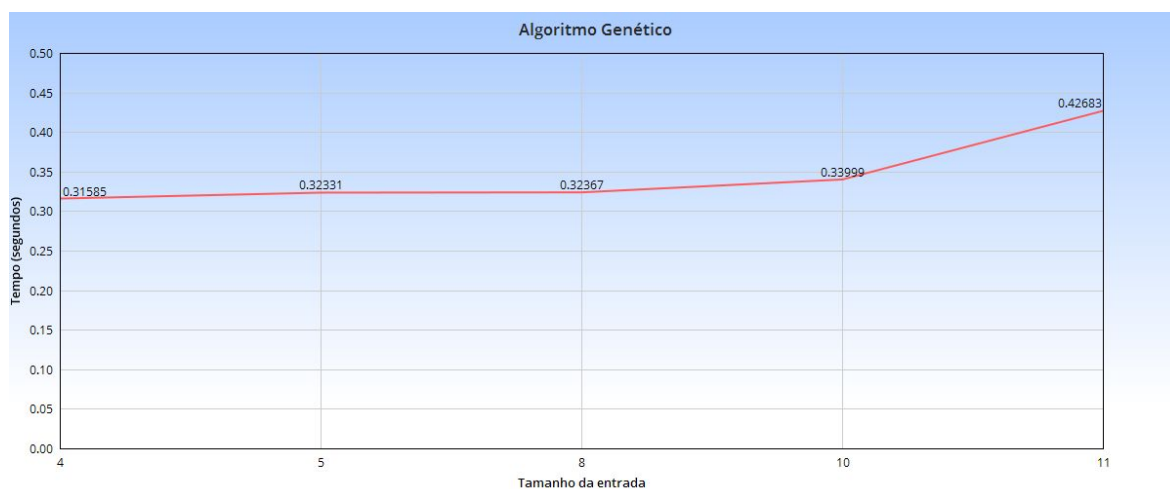
3. O algoritmo Branch and Bound em geral se sai bem e somente um pouco pior que o Dinâmico, pois ele utiliza a estratégia de ir calculando e somando as distâncias e comparando o resultado com uma variável que armazena a menor distância já encontrada das sequências que ele já percorreu até o momento, se a distância for maior, ele já pula para a próxima sequência de caminhos. Isso faz com que ele poupe tempo pulando cálculos e evitando de fazer todas as operações da iteração atual, pois se a distância já foi maior que a menor, pode-se ignorar a opção.



(Imagem 5: Crescimento de tempo no algoritmo Branch and Bound)

4. O algoritmo Genético se sai bem em todas as vezes pois ele é um algoritmo que usa heurística para calcular sua resposta em troca de um aumento na velocidade. Devido a heurística, o seu resultado pode não ser sempre o melhor possível, porém possui uma boa aproximação considerando a diferença de tempo. Sendo assim, o algoritmo consegue executar entradas que, caso forem executadas utilizando os outros algoritmos acima seriam praticamente impossíveis de serem calculados em tempo hábil. Nos nossos casos de testes do algoritmo genético, utilizamos as seguintes variáveis de controle:

- tamanhoPopulacao = 1000;
- qtdRandomizacoes = 10;
- geracoes = 50;



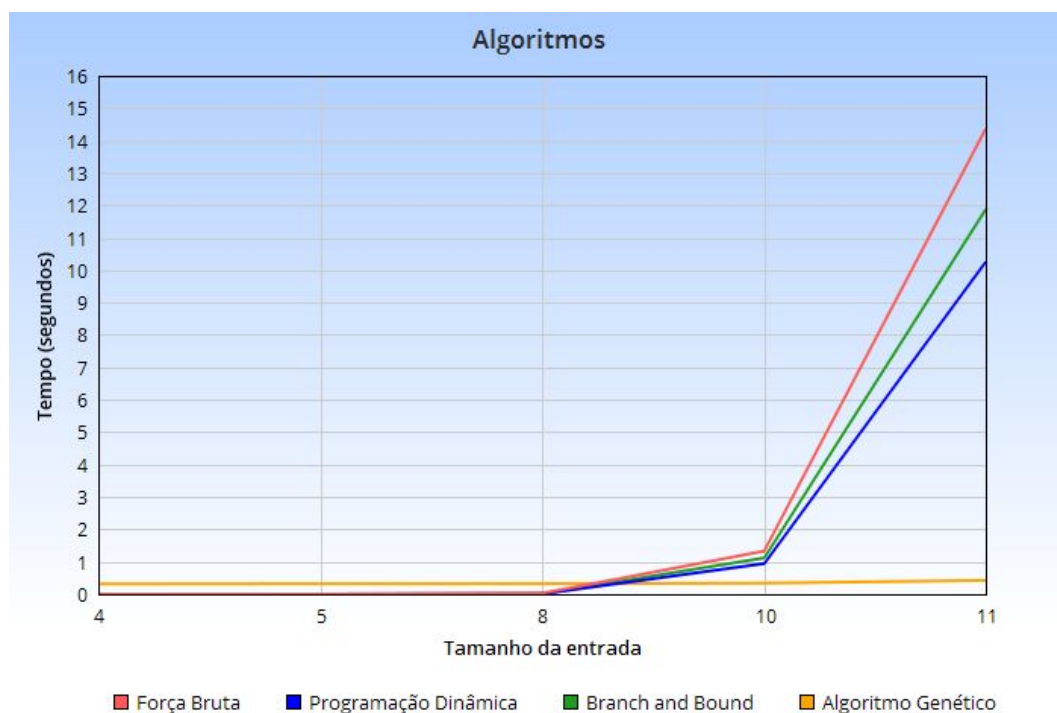
(Imagem 6: Crescimento de tempo no algoritmo Programação Genético)

No gráfico seguinte, podemos observar os resultados de todos os algoritmos sobrepostos, utilizando as mesmas entradas.

A partir da observação e análise desse gráfico, podemos concluir que, para esse problema, o algoritmo de força bruta é o mais simples de ser implementado, porém traz um tempo superior aos demais.

Com isso, concluímos que, outros tipos de paradigmas de implementações, podem ser mais difíceis de serem implementadas, porém podem trazer resultados interessantes em relação ao tempo de execução, como é o exemplo do programação dinâmica e branch and bound, mas que ainda assim possui uma alta complexidade.

A solução mais interessante foi utilizando o algoritmo genético, que possui um custo quase constante comparado aos demais. Para entradas muito pequenas ele realiza muitas operações repetidas e desnecessárias, mas aumentando a entrada, mal conseguimos observar um aumento considerável no tempo de execução. Ele pode não trazer a resposta mais correta para alguns casos, porém sua aproximação é muito mais vantajosa do que o custo de se esperar alguns “milhares de séculos” para termos a resposta correta.



(Imagem 7: Crescimento de tempo nos algoritmos desenvolvidos)

5. Conclusão

Durante a confecção deste trabalho, pudemos treinar e aprimorar os nossos conhecimentos em análise e projeto de algoritmos com diferentes abordagens a um mesmo problema.

Conseguimos observar, durante os testes, que um problema pode ser resolvido de várias maneiras, e que às vezes um resultado aproximado pode ser mais vantajoso do que o melhor possível. Alterar um paradigma de desenvolvimento pode mudar consideravelmente a complexidade sem alterar significativamente o resultado obtido.

Durante o início do desenvolvimento surgiram algumas dificuldades relacionadas à linguagem de programação C++, porém com o tempo e estudo da linguagem já não enfrentávamos mais problemas triviais, nos permitindo focar no desenvolvimento dos algoritmos, assim como o estudo e entendimento dos mesmos.

6. Bibliografia

STANDARD C++ Library reference. [S. /], 2020. Disponível em: <http://www.cplusplus.com/reference>. Acesso em: 23 maio 2020.

THE CODING Train. *In*: Traveling Salesperson with Genetic Algorithm. [S. /], 2020. Disponível em: https://www.youtube.com/watch?v=M3KTWnTrU_c. Acesso em: 23 maio 2020.

TRUE Random Number Service. [S. /], 2020. Disponível em: <https://www.random.org/>. Acesso em: 23 maio 2020.

TRAVELING Salesman Problem using Genetic Algorithm. [S. /], 2020. Disponível em: <https://www.geeksforgeeks.org/traveling-salesman-problem-using-genetic-algorithm/>. Acesso em: 19 maio 2020.

TRAVELING salesman problem using branch and bound. [S. /], 2020. Disponível em: <https://www.geeksforgeeks.org/traveling-salesman-problem-using-branch-and-bound-2/>. Acesso em: 19 maio 2020.

Travelling Salesman Problem | Set 1 (Naive and Dynamic Programming) [S. /], 2020. Disponível em: <https://www.geeksforgeeks.org/travelling-salesman-problem-set-1/>. Acesso em: 19 maio 2020.

7. Anexos

Repositório contendo todos os códigos e arquivos:

https://github.com/LucasFSCAR/TRABALHO_PRATICO_PAA