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

Estudo sobre o problema do Caixeiro Viajante*

Bernardo Ama Raquel[1]
Túlio Nunes Polido Lopes[2]

Resumo

O Problema do Caixeiro Viajante, ou como é conhecido em inglês, Travelling Salesman Problem(TSP) é um problema computacional classificado como NP-Difícil. Nesse trabalho exploramos algumas formas de solucionar esse problema e demonstramos seus resultados, comparando-os.

*Trabalho Teórico Prático apresentado na disciplina de Projeto e Análise de Algoritmos do curso de Ciência da Computação da Pontifícia Universidade Católica de Minas Gerais sobre Travelling Salesman Problem(TSP).

[1]Aluno do Programa de Graduação em Ciência da Computação, Brasil – bernardo.victor@sga.pucminas.br.
[2]Aluno do Programa de Graduação em Ciência da Computação, Brasil – tulio.polido@sga.pucminas.br.

1 INTRODUÇÃO

O problema do caixeiro viajante, também conhecido como Travelling Salesman Problem(TSP) em inglês é um problema que consiste em achar o menor caminho existente entre N cidades, partindo de uma cidade inicial A, percorrendo todas as outras N-1 cidades e retornando à cidade inicial. Este artigo tem como objetivo analisar e discutir sobre diferentes soluções propostas para esse problema.

2 IMPLEMENTAÇÃO

Nesta seção serão abordadas quatro implementações diferentes para solucionar o caixeiro viajante.Além disso forneceremos informações técnicas sobre o ambiente computacional utilizado. Todos os algoritmos utilizados recebem a mesma entrada. Ela consiste em um valor inteiro N que se refere a quantidade de cidades do problema. Depois as seguintes N linhas são pares X e Y inteiros que representam as coordenadas das cidades. Esses pares são utilizadas como base para calcular as distâncias entre todas as cidades e salvar na matriz de adjacência referente àquela entrada.

Estrutura de dados

```
typedef pair<int,int> ponto; //par de inteiros
int matriz[100][100];   //Matriz de Adj.
int cidades[100];   //max 100 cidades
vector<vector<int> > caminhos; //Caminhos calculados
int n;  //Num. de cidades
```

O preenchimento da matriz de adjacência é feito pelo seguinte bloco:

Calculo das distâncias

```
for(int i = 0; i< n; i++) {
    for( int j = 0; j < n; j++) {
        //calcula a distancia entre duas cidades
        double distance = dist2pontos(coordenadas[i],
                                      coordenadas[j]);
        matriz[i][j] = distance;
        matriz[j][i] = distance;
    }
}
```

2.1 Ambiente Computacional

O sistema operacional utilizado foi o Windows 7 e o compilador usado foi o gcc versão 7.4.0. O desenvolvimento foi feito com o editor de texto Atom, que conta com uma extensão que permite edição simultânea do mesmo arquivo em computadores diferentes.

2.2 Força Bruta

O algoritmo de Força Bruta consiste basicamente em testar uma a uma, todas as possibilidades de soluções. Ou seja, dado N cidades, o programa irá calcular N-1 permutações entre as cidades, somar o custo total de cada caminho e escolher a solução com menor custo. O algoritmo utilizado nesse artigo, usa um vetor de cidades para calcular as permutações e uma matriz de adjacência com as distâncias entre cada cidade. Após ler a entrada e calcular as distâncias entre as cidades, um método faz as permutações entre as cidades e salva todas as possibilidades no vetor "caminhos".

Permutação

```
void findPermutations(int a[], int n) {
    do {
        vector<int> c; //vetor auxiliar
        for (int i = 0; i < n; i++) {
            c.push_back(a[i]);
        }
        c.push_back(a[0]);
        caminhos.push_back(c);
    } while (next_permutation(a+1, a+n));
}
```

Por fim o algoritmo percorre o vetor de caminhos, e para cada caminho no vetor é calculada a distância total utilizando uma função. Uma variável auxiliar se encarrega de salvar o menor valor até o momento, e ao fim ela terá o resultado final.

Cálculo da menor distância

```
double calcDist(vector<int> caminho) {
    double dist = 0;
    //A cada iteracao, eh salvo a
    //distancia entre a cidade i e a anterior
    for(int i = 1; i < caminho.size(); i++) {
        dist += matriz[caminho[i-1]][caminho[i]]
    }
    return dist;
}
```

Ao final o programa printa o melhor caminho encontrado bem como a distância que será percorrida naquele caminho.

2.3 Branch and Bound

Funciona de maneira similar ao força bruta mas não gera todas as possibilidades de caminho.(RAI, 2016) Durante a geração de uma nova permutação, ele também calcula a distância e compara com a menor distância armazenada. Se a permutação atual ultrapassar o valor da menor distância, ele aborta o cálculo dessa permutação e testa a próxima possibilidade, assim economizando tempo para achar a solução. O algoritmo chama o método recursivo "branchAndBound", que recebe três parâmetros: parcial, um inteiro que representa a soma da distância parcial até a cidade atual; level, um int que representa a que nível de recursão a instância atual se encontra; caminho-parcial, um vetor de inteiros que representa o caminho percorrido até o momento.

No caso base, se o nível atual de recursão for igual ao número de cidades, significa que um caminho está completo, portanto deve-se calcular a distância percorrida por ele e comparar ao melhor resultado atual. Caso seja uma melhor solução, ela deve substituir a melhor resposta.

Caso Base

```
//Se chegar na ultima posicao e a distancia for diferente de 0
if (level==N) {
    if (matriz[caminho_parcial[level-1]][caminho_parcial[0]] != 0) {

        int curr_res = parcial
        + matriz[caminho_parcial[level-1]][caminho_parcial[0]];

        //se a resposta parcial for menor que a final
        if (curr_res < menorDist) {
            //atualiza o melhor caminho
            for (int i=0; i<N; i++)
                        caminho_final[i] = caminho_parcial[i];
                caminho_final[N] = caminho_parcial[0];
            menorDist = curr_res; //atualiza a melhor resposta
        }
    }
}
```

No passo recursivo, o algoritmo utiliza um vetor auxiliar de booleanos, que salva as cidades que já foram visitadas por aquela recursão. Ele possui um ciclo que percorre o vetor de cidades e se a cidade da posição i não tiver sido visitada, o resultado parcial soma a distância entre a cidade atual e a cidade i, então é verificado se esse resultado parcial é menor que o melhor caminho atual, caso seja, a recursão é chamada, caso contrário, todas as derivações daquele caminho são ignoradas.

Passo Recursivo

```
else {
    //itera pelas cidades do problema
    for (int i=0; i<N; i++) {
        //Se a cidade nao foi visitada e
        //se a distancia eh diferente de 0
        if (matriz[caminho_parcial[level-1]][i] != 0
            && visited[i] == false) {

            //atualiza a parcial
            parcial += matriz[caminho_parcial[level-1]][i];

            //Se a parcial for maior que a melhor resposta ja obtida
            //Ignora-se esse caminho e seus derivados
            if (parcial < menorDist) {
                caminho_parcial[level] = i;
                visited[i] = true;
                branchAndBound(parcial, level+1, caminho_parcial);
            }

            //Reseta os valores
            parcial -= matriz[caminho_parcial[level-1]][i];
            memset(visited, false, sizeof(visited));
            for (int j=0; j<=level-1; j++)
                visited[caminho_parcial[j]] = true;
        }
    }
}
```

Ao final da recursão do método, duas variáveis globais terão o resultado do problema salvos. Sendo elas menorDist, um inteiro que representa o valor da menor distância. E a outra caminho-final, um vetor de inteiros que salva a ordem das cidades a serem percorridas no melhor caminho.

2.4 Programação Dinâmica

O algoritmo da programação dinâmica salva valores já calculados para uso posterior, evitando assim que cálculos sejam refeitos. Nele é usado uma matriz auxiliar, chamada dp, onde serão salvas as distâncias já calculadas. O método TSPDynamic é quem calcula a resposta de forma recursiva, recebendo 4 parâmetros e retornando o inteiro que representa a menor distância calculada.

TSPDynamic

```
int TSPDynamic(int mask, int pos,
                int VISITOU_TUDO, string caminho_parcial)
```

Os parâmetros recebidos são: mask, um inteiro representando quais cidades já foram visitadas(em bits); pos, um inteiro representando qual a cidade atual; VISITOU-TUDO, um inteiro que representa o caso em que todas as cidades foram visitadas(em bits), ou seja, para 4 cidades esse inteiro seria 1111; caminho-parcial, uma string que salva o caminho percorrido até o momento.

Caso Base

```
//Se todas cidades ja foram visitadas,
//retorna a distancia entra a ultima cidade e a primeira
if(mask==VISITOU_TUDO){

    //salva o caminho percorrido numa variavel global
    fim_do_caminho = caminho_parcial;
    return matriz[pos][0];
}

//Se dp for diferente de -1, o valor ja foi calculado
if(dp[mask][pos] != -1) {
    return dp[mask][pos];
}
```

No caso base, caso na recursão atual todas as cidades tenham sido visitadas, o método retorna o valor da distância entre a cidade atual e a cidade inicial, a ser somado no resultado final. Caso o valor de dp na mask e pos atual seja diferente de -1, significa que aquela distância já foi calculada, então o algoritmo retorna o valor salvo na matriz auxiliar.

Passo Recursivo

```
int resp = INT_MAX;
for(int cidade = 0; cidade < n; cidade++) {

    //Se a cidade ainda nao foi visitada, visite a cidade
    if((mask&(1<<cidade)) == 0) {

    int newResp = matriz[pos][cidade]
        + TSPDynamic(mask|(1<<cidade), cidade,
                    VISITOU_TUDO,caminho_parcial);

        if(newResp < resp) {
            caminho_final = fim_do_caminho + '1';
            reverse(caminho_final.rbegin(),caminho_final.rend());
            caminho_final = caminho_final.substr(0,n*2-1);
            resp = newResp;
        }
    }
}
return resp;
```

No passo recursivo, o algoritmo itera pelo vetor de cidades, e caso a cidade ainda não tenha sido visitada, ele atribui à resposta parcial a distância entre a cidade atual e a cidade a ser visitada e soma com o resultado a ser retornado da recursão da próxima cidade.

Na volta da recursão, é verificado se a distância percorrida é menor que a melhor resposta atual, caso seja o caminho percorrido será salvo numa variável global.

2.5 Algoritmo Genético

O algoritmo genético utilizado nesse artigo foi desenvolvido por Marcos Castro (CASTRO, 2014). Devido à sua alta complexidade, será explicado somente o básico da teoria de códigos genéticos.

Um algoritmo genético é baseado na seleção natural.(LARRANAGA et al., 1999) Então, o algoritmo cria indivíduos que são possíveis soluções para o problema e simula a evolução entre eles. Baseado nisso, cada iteração do código, dois indivíduos são selecionados para ser o pais da próxima solução. Após serem selecionados, as duas soluções são mescladas e mutações são aplicadas.

Em seguida, cada uma das novas soluções é avaliada. Elas passam por um processo que verifica sua validade e de qualidade. No exemplo do TSP, soluções não válidas seriam aquelas em que cidades sejam repetidas, ou que haja conexão entre duas cidades que não possuem liga-

ção direta. Na avaliação de qualidade, soluções piores que a geração anterior são descartadas, e as que seguem são as soluções que melhoraram os resultados já obtidos.

Um conceito importante do algoritmo genético é a ideia de geração. A proposta é que cada geração seja melhor que a anterior, assim quanto mais gerações o código rodar, mais confiável será seu resultado final.

Pseudocódigo

```
void Procedimento AG {
    int t = 0;
    inicia_populacao(P, t);
    avaliacao(P, t);

    //d representa a quantidade de geracoes
    while (t != d) {
        t = t + 1;
        selecao_dos_pais(P,t);
        recombinacao(P, t);
        mutacao(P, t);
        avaliacao(P, t);
        sobrevivem(P, t);
    }
}
return resp;
```

No pseudocódigo acima os passos necessários são representados. Primeiro deve-se estabelecer a quantidade, d, de gerações a serem calculadas. Então randomicamente a população inicial é criada, e há uma avaliação se ela é possível ou não.

Posteriormente, o código inicia seu ciclo de gerações, e, a cada iteração são concluidas diversas etapas. Primeiro, seleciona-se os pais dessa geração, então seus genes são combinados. Após isso, será aplicada uma mutação aleatória no novo código genético gerado, e essa mutação será avaliada a fim de excluir as soluções impossíveis e as ruins. Por fim o ciclo termina avaliando quais indivíduos da nova geração serão os sobreviventes, reiniciando o ciclo.

Ao final de todos os ciclos de evolução, os indivíduos sobreviventes serão aqueles que portarão o resultado final para o problema. No caso do TSP eles representarão o caminho com menor distância que percorra todas as cidades e volte à cidade inicial.

3 ANÁLISE DE COMPLEXIDADE

Nesta seção será feita a análise de complexidade de cada algoritmo no pior e no melhor caso. Além disso será explicada a operação mais relevante e as configurações de entrada que levam ao pior e ao melhor caso.

3.1 Força Bruta

Como o força bruta sempre gera todas as possibilidades de caminho a serem testados, o melhor e o pior caso são iguais para quaisquer entradas. Neste algoritmo temos duas operações relevantes :
as movimentações no vetor de permutações para gerar todos os possíveis caminhos e as somas dos elementos da matriz de adjacência para calcular a distância de cada caminho.

A função de permutação obviamente faz n-1! movimentações pois sempre começamos da cidade 1 e o cálculo de permutação é feito em fatorial.

A função de somar as distâncias é feita em tempo n para cada caminho pois deve-se calcular a distância das n-1 arestas do caminho + a aresta da última cidade para a primeira.

Como deve-se somar a distância de cada caminho, temos a complexidade n*n-1! somada a complexidade de achar todos os caminhos n-1!. Logo temos: n! + n-1! = $O(n!)$

3.2 Branch and Bound

O Branch and Bound descarta um possível caminho quando a soma da distância parcial é maior do que a menor distância. Mas em uma entrada em que as distâncias são muito parecidas, ele só consegue descartar quando já somou a distância total daquele caminho. Portanto no pior caso o Branch and Bound tem complexidade igual ao do Força Bruta que é $O(n!)$. Caso seja utilizada a técnica de estabelecer limites inferiores, o Branch and Bound terá complexidade de $O(n^2 * 2^n)$.(RANJANA, 2018)

Já no melhor caso as distâncias devem ser discrepantes, então você pode eliminar elas antes. Supondo que o primeiro caminho calculado fosse a menor distância e as distâncias parciais dos outros caminhos fossem maiores do que a menor distância, temos :

-custo para achar a menor distância = n (custo de somar distância total do primeiro caminho) * 1 (custo para achar a primeira permutação).

-custo para achar cada distância parcial = 1(custo para achar a primeira distância parcial) * n-2! (custo para achar as permutações restantes).
Logo:

n + n-2! = $O(n!)$

3.3 Programação Dinâmica

Nesse problema há apenas um vértice inicial, que representa a primeira cidade lida. A partir dele existirão 2^n subgrafos que representam os possíveis caminhos que saem da primeira cidade, percorrem todas as outras e voltam à posição inicial. Cada uma dessas chamadas recursivas faz no máximo n-1 outras chamadas. Assim a complexidade pode ser dada como $O(n * 2^n)$. Caso o problema não especificasse o vértice inicial, teríamos mais n permutações de caminhos possíveis, pois o vértice inicial seria adicionado a essas permutações. Isso tornaria a complexidade $O(n^2 * 2^n)$

3.4 Algoritmo Genético

Este algoritmo é uma metaheurística pois ele gera um resultado aproximado do problema. M representa o número de gerações possíveis. Quanto maior esse valor, maiores são as chances de se atingir o melhor resultado. No nosso problema colocamos M como 1000.

Devido à complexidade do algoritmo, será feita uma análise bruta de complexidade. O algoritmo roda por M gerações. Em cada uma das iterações são feitas seleções($O(N)$), recombinações($O(N)$), mutações($O(1)$)) e avaliações($O(N)$). Assim a cada iteração seriam realizadas 3n + 1 procedimentos. Como isso se repete por M gerações, serão feitas M * (3N+1) procedimentos. Então a complexidade pode ser dita como $O(M * N)$ para quaisquer entradas.

4 TESTES

4.1 Força Bruta

Tabela 1 – Execução Força Bruta

N Cidades	Tempo de execução (hh:mm:ss)	Saída(caminho / distância)
2	00:00:00.037	1 2 / 1600
3	00:00:00.038	1 2 3 / 2400
4	00:00:00.040	1 2 3 4 / 3200
5	00:00:00.037	1 2 3 5 4 / 3530
6	00:00:00.050	1 2 3 6 5 4 / 3530
7	00:00:00.062	1 2 3 7 6 5 4 / 3529
8	00:00:00.235	1 2 3 8 7 6 5 4 / 3529
9	00:00:02.176	1 2 3 9 8 7 6 5 4 / 3529
10	00:00:21.734	2 1 4 5 6 7 8 3 9 10 / 3670
11	00:04:07.269	2 1 4 5 6 7 8 3 9 10 11 / 3811
12	00:25:16.435+	-
13	-	-
14	-	-
15	-	-

Como era esperado o Força Bruta encontra a melhor solução mas em um tempo de execução muito extenso. Resolvendo o problema do Caixeiro Viajante para 12 cidades a máquina travou depois de 25 min e nenhum resultado foi gerado.

4.2 Branch and Bound

Tabela 2 – Execução Branch and Bound

N Cidades	Tempo de execução (hh:mm:ss)	Saída(caminho / distância)
2	00:00:00.038	1 2 / 1600
3	00:00:00.038	1 2 3 / 2731
4	00:00:00.037	1 2 3 4 / 3200
5	00:00:00.037	1 2 3 4 5 / 3530
6	00:00:00.038	1 2 3 6 5 4 / 3530
7	00:00:00.037	1 2 3 7 6 5 4 / 3529
8	00:00:00.040	1 2 3 8 7 6 5 4 / 3529
9	00:00:00.043	1 2 3 8 9 7 6 5 4 / 3811
10	00:00:00.070	1 2 3 10 9 8 7 6 5 4 / 3811
11	00:00:00.249	1 2 3 10 11 9 8 7 6 5 4 / 4093
12	00:00:01.279	1 2 3 10 12 11 9 8 7 6 5 4 / 4375
13	00:00:06.738	1 2 3 10 12 13 11 9 8 7 6 5 4 / 4657
14	00:00:34.125	1 2 3 10 12 14 13 11 9 8 7 6 5 4 / 4939
15	00:02:48.088	1 2 3 10 12 14 15 13 11 9 8 7 6 5 4 / 5221

O método Branch and Bound geralmente encontra a melhor solução mas em alguns momentos encontra uma solução aproximada quando o custo seria muito alto para gerar uma solução mais otimizada. Portanto seus tempos de execução são muito melhores do que do Força Bruta e ele conseguiu resolver o problema para 15 cidades facilmente.

4.3 Programação Dinâmica

Tabela 3 – Execução Programação Dinâmica

N Cidades	Tempo de execução (hh:mm:ss)	Saída(caminho / distância)
2	00:00:00.037	1 2 / 1600
3	00:00:00.036	1 2 3 / 2731
4	00:00:00.036	1 2 3 4 / 3200
5	00:00:00.036	1 2 3 4 5 / 3530
6	00:00:00.037	1 2 3 4 5 6 / 3530
7	00:00:00.038	1 2 3 4 5 6 7 / 3529
8	00:00:00.054	1 2 3 4 5 6 7 8 / 3529
9	00:00:00.186	1 2 3 4 5 6 7 8 9 / 3529
10	00:00:01.384	1 2 3 4 5 6 7 8 9 10 / 3775
11	00:00:13.606	1 2 3 4 5 6 7 8 9 10 11 / 4030
12	00:02:38.368	1 2 3 4 5 6 7 8 9 10 11 12 / 4292
13	00:31:27.636	1 2 3 4 5 6 7 8 9 10 11 12 13 / 4557
14	04:30:52.209+	-
15	-	-

A programação dinâmica se provou melhor que o Força Bruta (em termos de tempo de execução) mas não conseguiu superar o Branch and Bound. Resolvendo o problema para 14 cidades, haviam se passado 4 horas e 30 min, e a execução foi interrompida pois nenhum resultado havia sido gerado.

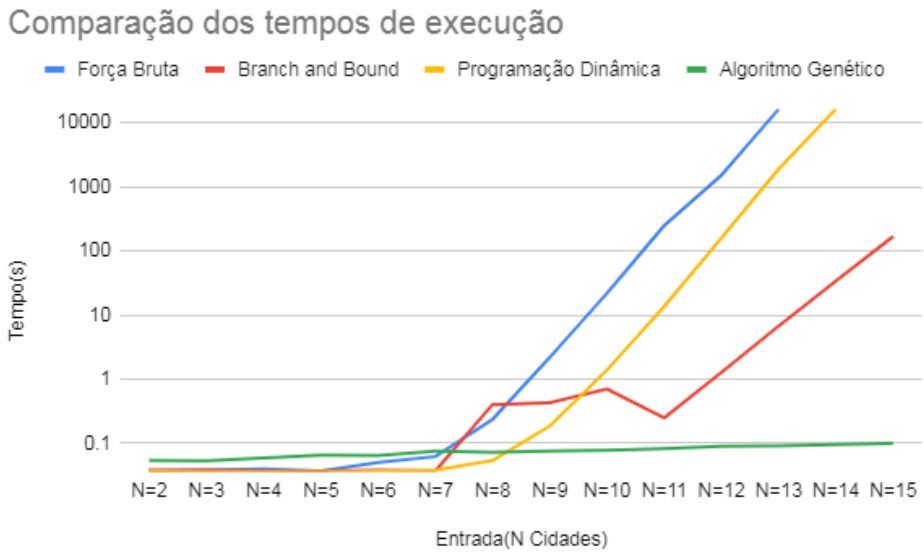
4.4 Algoritmo Genético

Tabela 4 – Execução Algoritmo Genético

N Cidades	Tempo de execução (hh:mm:ss)	Saída(caminho / distância)
2	00:00:00.054	1 2 / 1600
3	00:00:00.053	1 2 3 / 2731
4	00:00:00.059	1 2 3 4 / 3200
5	00:00:00.065	1 5 4 3 2 / 3530
6	00:00:00.064	1 4 3 6 5 2 / 3530
7	00:00:00.076	1 2 5 6 7 3 4 / 3529
8	00:00:00.072	1 2 3 8 7 6 5 4 / 3529
9	00:00:00.076	1 2 9 3 8 7 6 5 4 / 3529
10	00:00:00.078	1 2 10 3 9 8 7 6 5 4 / 3775
11	00:00:00.083	1 4 11 10 9 3 8 7 6 5 2 / 4030
12	00:00:00.090	1 4 5 6 7 8 3 9 10 11 12 2 / 4292
13	00:00:00.091	1 2 5 6 7 8 9 3 10 11 12 13 4 / 4557
14	00:00:00.095	1 2 14 13 12 11 10 9 3 8 7 6 5 4 / 4826
15	00:00:00.100	2 2 15 14 13 12 11 10 3 9 8 7 6 5 4 / 5098

O algoritmo Genético se provou mais eficiente (em termos de tempo de execução) que até mesmo o próprio Branch and Bound. Isso se deve ao fato dele constantemente estar procurando soluções aproximadas, que em alguns casos (principalmente para entradas menores) tem grandes chances de ser a melhor solução.

4.5 Gráfico Comparativo



5 CONCLUSÃO

Esta foi uma pesquisa interessante a ser feita pois ela mostra como diferentes paradigmas de programação podem melhorar a eficiência de um algoritmo. O Força Bruta embora lento, é a forma mais intuitiva de se resolver inicialmente um problema. A partir dele é possível explorar o Branch and Bound, que é trivial de ser implementado para resolver o Caixeiro Viajante(ele é apenas uma edição do código do Força bruta). Porém implementamos uma versão mais elaborada (lógica feita pelo GeeksforGeeks) que calcula o limite inferior para gerar a solução ótima. Já a Programação Dinâmica e o Algoritmo Genético são mais difíceis de ser implementados. Por isso utilizamos códigos de outros Autores para esses dois.

Após analisar os tempos de execução e as saídas dos programas, concluímos o que já era esperado : não existe ainda um algoritmo eficiente para resolver o Caixeiro Viajante. Apesar da abordagem Genética ter um tempo de execução excepcional, ela gera apenas uma solução aproximada e não a melhor solução, que é a desejada.

6 ANEXOS

Listagem dos Programas :
CaixeiroBruto.cpp - Força Bruta
BranchAndBound.cpp - Branch and Bound
Dynamic.cpp - Programação Dinâmica
main.cpp -Algoritmo genético
tsp.cpp -Algoritmo genético
tsp.h -Algoritmo genético
tester.bat - Script para testar os algoritmos
timethis.exe - Programa para calcular tempo de execução

REFERÊNCIAS

CASTRO, MARCOS. **Genetic TSP**. [S.l.]: GitHub, 2014. <https://github.com/marcoscastro/tsp_genetic>.

LARRANAGA, Pedro et al. Genetic algorithms for the travelling salesman problem: A review of representations and operators. **Artificial Intelligence Review**, Springer, v. 13, n. 2, p. 129–170, 1999.

RAI, Anurag. **Traveling Salesman Problem using Branch And Bound**. [S.l.]: Geeks For Geeks, 2016. <https://www.geeksforgeeks.org/traveling-salesman-problem-using-branch-and-bound-2/>.

RANJANA, P. Travelling salesman problem using reduced algorithmic branch and bound approach. **International Journal of Pure and Applied Mathematics**, v. 118, n. 20, p. 419–424, 2018.