

# **Projeto Final**

## **Problema do Caixeiro Viajante**

**Pontifícia Universidade Católica**  
**Ciência da Computação - 5º Período**  
**Projeto e Análise de Algoritmos**

**Alunos:**

***Raul Formiga Mansur***  
***Luiz Junio Veloso Dos Santos***  
***Matheus Luiz Oliveira Spindula***

### **Descrição do problema**

O que vai ser tratado em nosso projeto é o problema do caixeiro viajante, um dos problemas mais estudados na área da computação e matemática. Esta problema tenta determinar a menor rota para percorrer um número  $n$  de cidades visitando-as uma única vez e retornando à cidade de origem.

Ele foi baseado na necessidade que os vendedores têm de realizar entregas em diferentes cidades, percorrendo o menor caminho possível.

### **O que foi realizado**

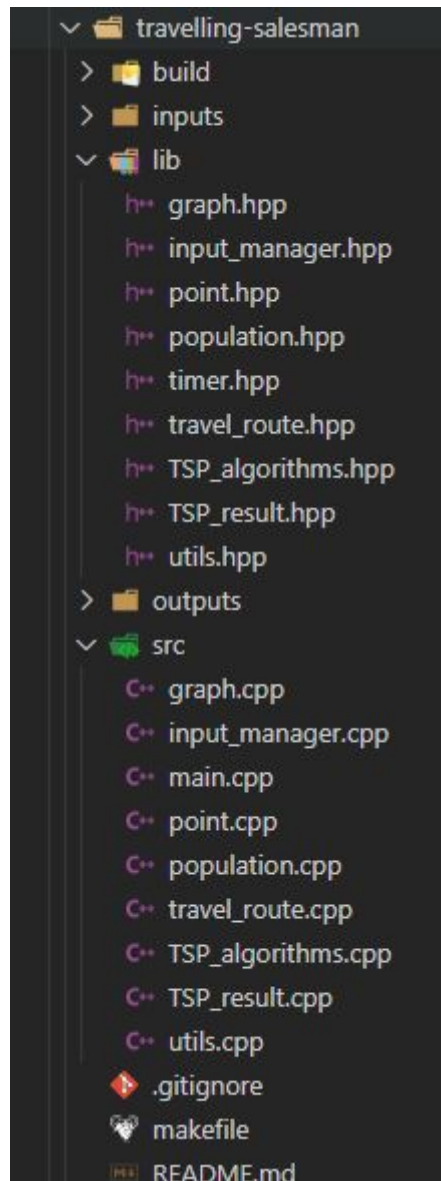
Em nosso trabalho utilizamos 4 algoritmos centrais onde cada um representa um tipo de técnica a ser utilizada para resolver o problema do caixeiro viajante.

- Força Bruta;
- Branch and Bound;
- Programação Dinâmica;
- Algoritmo Genético;

O objetivo deste trabalho é implementar esses algoritmos, e realizar uma sequência de testes para observar o comportamento de cada um deles com relação ao tamanho da entrada, tempo de execução, quantidade de recursos computacionais necessários e eficiência.

## Implementação

O programa foi implementado em C++, e suas várias funcionalidades foram subdivididas em classes, arquivos de cabeçalho, arquivos de código, arquivos de entrada/saída e arquivos de compilação.



A pasta build contém os arquivos de compilação temporários usados pelo makefile, e também o executável do programa.

A pasta inputs contém os arquivos de entrada que geramos em nosso programa no formato .in. Os arquivos de entrada são gerados automaticamente(e de forma aleatória) pela classe InputManager.

A pasta outputs contém as saídas do programa em formato .csv para cada algoritmo(tempo em seg), e saídas no formato .out com o custo mínimo e a rota mínima.

A pasta lib contém os cabeçalhos de todos os códigos fontes respectivos as classes usadas no programa.

A pasta src contém os códigos fontes das classes utilizadas no programa.

O arquivo main.cpp é o principal.

O projeto contempla de um makefile, por possuir uma grande quantidade de arquivos para serem compilados, e regras específicas de compilação.

O makefile basicamente compila cada arquivo .cpp (código fonte), incluindo seu respectivo .hpp(cabeçalho), em um arquivo .o (object). Ao final da compilação, ele "une" todos os *objects* em um executável final.

### **Arquivos principais:**

**Graph** é a estrutura de dados de grafo que foi utilizada. Ela possui uma matriz de adjacências com tipo de dados *double*, que armazena o peso da aresta entre os vértices e possui a quantidade de vértices, um vetor de pontos (x,y).

**Point** é a estrutura de dados que contém o produto cartesiano (x, y). Ela é responsável por fazer o cálculo da distância euclidiana em relação a outro ponto.

**Timer** é uma classe que é responsável por fazer o cálculo de tempo gasto na execução de trechos de código.

**Utils** é uma classe que contém métodos de utilidades gerais para reaproveitamento de código no projeto.

**InputManager** é uma classe que manipula as entradas do programa. Ela é responsável por gerar os arquivos de entrada com pontos(x,y) aleatórios, baseado em um N máximo, e em valores (X,Y) máximos, e também em ler os arquivos no formato da classe *Graph*.

**TSPAlgorithms** é o arquivo que une a implementação das classes que contém os algoritmos que resolvem o problema do caixeiro viajante (Travelling Salesman Problem - TSP). Ele possui em cada classe métodos públicos para a execução do algoritmo para uma entrada n, e possui métodos privados relacionados à execução dos algoritmos em si.

**TSPResult** é uma classe responsável por mostrar o usuário o menor caminho percorrido, sua distância e o tempo levado para executar o algoritmo.

**Main** é o arquivo principal, e possui um menu com itens selecionáveis, para interação do usuário com a execução dos algoritmos.

## Entradas e saídas

As entradas estão na pasta inputs, e tem extensão .in.

O primeiro número, n, é a quantidade de cidades (indexadas com os números de 1 a n). Os números seguintes vêm em pares e representam as coordenadas das cidades (X e Y). O primeiro par de números são as coordenadas da cidade 1, o segundo par, da cidade 2 e assim sucessivamente até a cidade n.

Exemplo:

```
4
100 100
900 100
900 900
100 900
```

As saídas estão na pasta outputs, e tem extensão .csv e .out.

As saídas .csv tem o nome do algoritmo como prefixo, e possui em cada linha a média do tempo de execução em segundos para cada entrada n.

Exemplo:

```
0.0009970
0.0110190
0.1077110
```

As saídas .out possuem etiquetas que indicam o que cada valor representa na execução de uma entrada para algum algoritmo.

Exemplo:

```
brute_force
Input size: 9
Min dist: 2689.11
Min path: 0 7 2 8 1 4 3 6 5
Time taken: 997 microseconds

brute_force
Input size: 10
Min dist: 2745.85
Min path: 0 7 2 8 9 1 4 3 6 5
Time taken: 11.0190 milliseconds
```

Todo o código está disponível em um [repositório](#) no GitHub, e também em anexo com este trabalho.

## Como executar o projeto

O projeto foi desenvolvido e testado em 3 sistemas operacionais distintos: Windows 10, Linux e MacOS.

Para executá-lo é necessário ter instalado na máquina o make e o g++ (ambos em sua última versão, de preferência).

Guia rápido de instalação:

### Windows:

Instale através do cygwin o g++ e o make

<https://cygwin.com/install.html>

### Linux:

Ubuntu e derivados: `sudo apt install build-essential`

Arch e derivados: `sudo pacman -S gcc make`

Gentoo/Funtoo\*: `sudo emerge -a make`

\*Essa distro já vem com o gcc por padrão, instale apenas o make caso não esteja instalado.

### Mac:

Instale o [Developer Tools](#) da Apple, ou utilize o gerenciador de pacotes [homebrew](#).

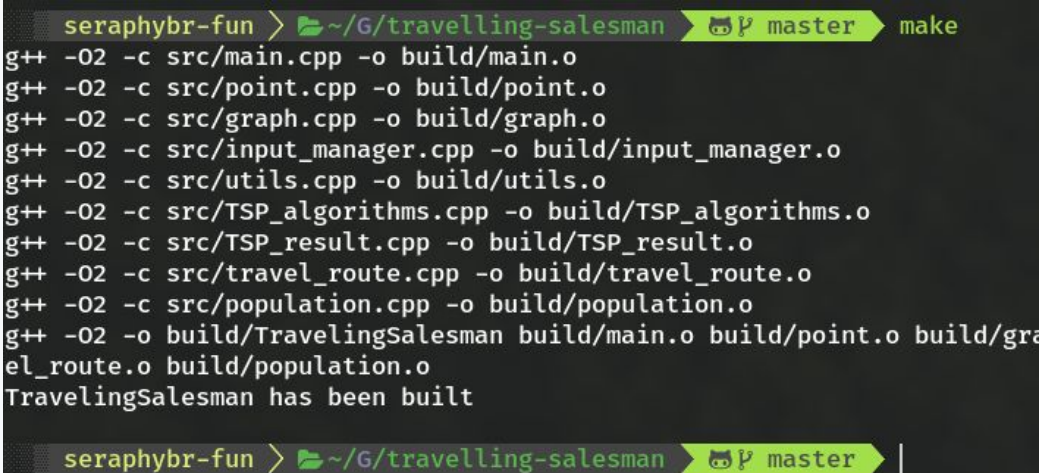
Utilizando o brew, basta instalá-lo e executar:

```
brew install gcc  
brew install make
```

## Execução

Para compilar o projeto, vá na pasta raiz do projeto, e execute:

```
make
```

A terminal window with a dark background and green accents. The prompt is 'seraphybr-fun > ~/G/travelling-salesman' with a 'master' branch indicator. The command 'make' has been executed, resulting in a series of g++ compilation commands for various source files in the 'src' directory, each outputting an object file in the 'build' directory. The final line of the output is 'TravelingSalesman has been built'.

```
seraphybr-fun > ~/G/travelling-salesman master make
g++ -O2 -c src/main.cpp -o build/main.o
g++ -O2 -c src/point.cpp -o build/point.o
g++ -O2 -c src/graph.cpp -o build/graph.o
g++ -O2 -c src/input_manager.cpp -o build/input_manager.o
g++ -O2 -c src/utils.cpp -o build/utils.o
g++ -O2 -c src/TSP_algorithms.cpp -o build/TSP_algorithms.o
g++ -O2 -c src/TSP_result.cpp -o build/TSP_result.o
g++ -O2 -c src/travel_route.cpp -o build/travel_route.o
g++ -O2 -c src/population.cpp -o build/population.o
g++ -O2 -o build/TravelingSalesman build/main.o build/point.o build/graph.o build/travel_route.o build/population.o
TravelingSalesman has been built
seraphybr-fun > ~/G/travelling-salesman master |
```

Se tudo ocorrer certo, deverá obter a mensagem *“TravelingSalesman has been built”*.

Para limpar os arquivos temporários de compilação, entradas e saídas, execute:

```
make clean
```

Para rodar o executável, entre na pasta *build*, e execute o arquivo *TravelingSalesman*.

```
cd build
./TravelingSalesman
```

O menu é bastante simples. Basta escolher qual opção deseja executar, digitá-la, e pressionar enter. Para fechar o programa no meio de sua execução, digite `Ctrl + c` ou `Cmd + c`.

```
Traveling Salesman Solver:
1. Gen random inputs [1..100]
2. Brute Force through inputs[1..100]
3. Branch and Bound through inputs[1..100]
4. Dynamic through inputs[1..100]
5. Genetic through inputs[1..100]
6. Brute Force statistics[1..100]
7. Branch and Bound statistics[1..100]
8. Dynamic statistics[1..100]
9. Genetic statistics[1..100]
0. Exit

OP: |
```

Os arquivos de entrada e saída são criados no tempo de execução do programa, e se encontram na pasta inputs e outputs.

## Informações Técnicas:

Projeto desenvolvido em C++, e compilado com o gcc v9.3.0

Utilizamos o editor de texto Visual Studio Code, o sistema de versionamento git, a plataforma de repositórios remotos de código GitHub.

O projeto foi testado em 3 ambientes diferentes:

*Windows 10 - Intel Core i5 7ª geração de 3.5 GHz, 8 GB de RAM DDR4*

*Funtoo Linux - Intel Core i5 8ª geração de 4.0 Ghz 8GB de RAM DDR4*

*MacOS Catalina - Intel core i7 8ª geração de 3.2GHz, 16 GB de RAM DDR4*

## Algoritmos:

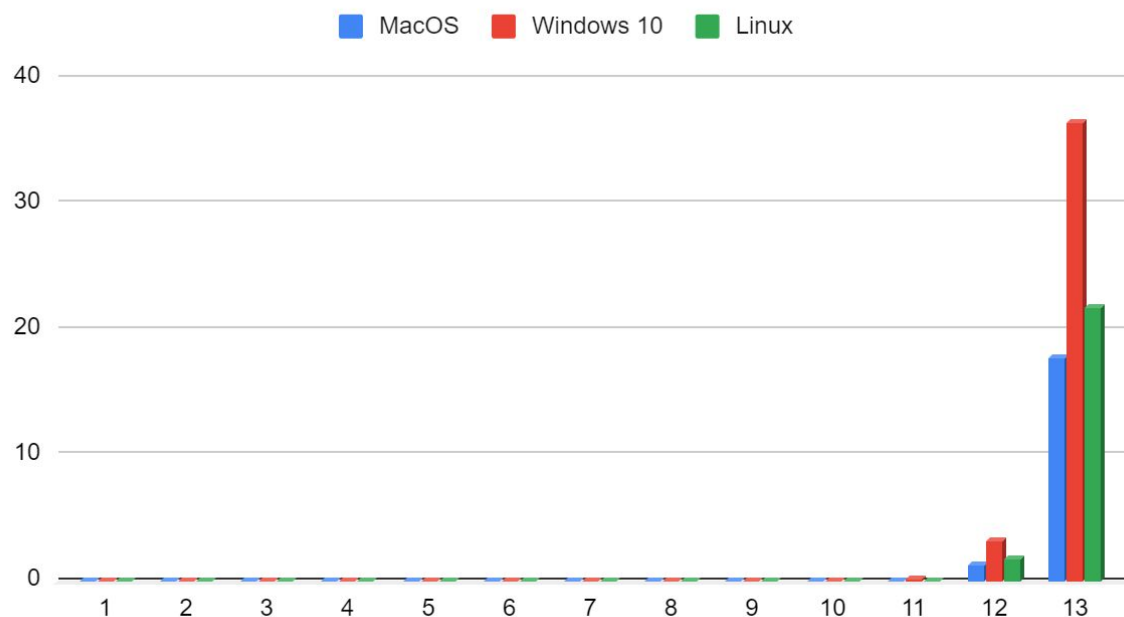
### Força Bruta

O algoritmo de força bruta é bem simples, e encontra a solução ótima para o problema. Ele resolve o problema fazendo todas as permutações de todos os caminhos possíveis, e buscar qual o menor caminho entre todas as permutações.

A operação relevante neste algoritmo, é a de soma de distâncias para verificação do menor caminho atual. Para cada permutação, verifica se o caminho atual é o menor até o momento, e para obter o valor, são necessárias  $n-2$  somas. São feitas no total  $(n-1)!$  verificações para uma entrada de tamanho  $n$ , em todos os casos, então fazemos no total  $(n-1)! * (n-2)$  operações.

Logo a complexidade deste algoritmo é  $O((n-1)! * (n-2)) = O(n! * n)$ , e portanto  $O(n!)$ .

### Força Bruta



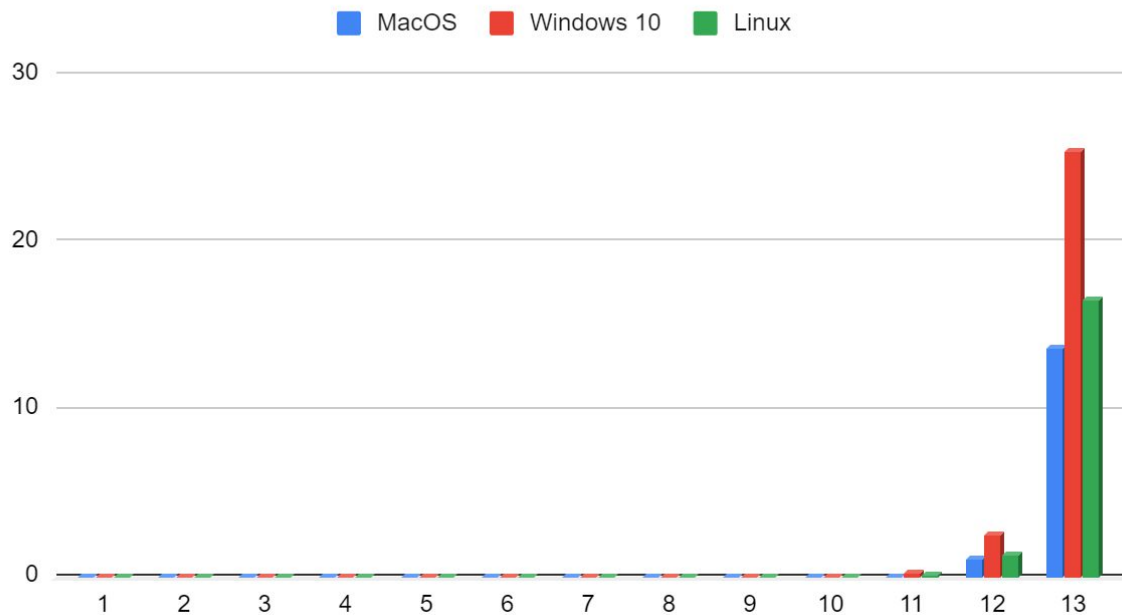
### Branch and Bound

O algoritmo branch and bound funciona como o de força bruta, gerando todas as permutações de caminhos, mas ele é mais sofisticado. Ele deixa de somar as arestas caso a soma tenha ultrapassado o *lower bound*, e portanto deixa de computar somas irrelevantes para o resultado ótimo. O *lower bound* é o somatório de todas as 2 menores arestas de todos os vértices, dividido por 2. O menor caminho sempre



será menor ou igual a esse limite. A complexidade no pior caso se mantém igual a complexidade do Força Bruta,  $O(n!)$ , pois ainda seriam testadas todas as possibilidades, e feitas todas operações sem redução.

## Branch and Bound



## Programação Dinâmica

Foram utilizados grafo para calcular a distância e matriz de adjacência, para representar a distância de cada cidade .

Consideramos 0 como ponto inicial e final da saída. Para todos os outros vértices  $i$  (exceto 0), encontramos o caminho de custo mínimo com 0 como ponto inicial,  $i$  como ponto final e todos os vértices aparecendo exatamente uma vez. Seja o custo desse caminho  $(i)$ , o custo do ciclo correspondente seria o custo  $(i) + \text{dist}(i,1)$  onde  $\text{dist}(i,1)$  é a distância de  $i$  a 1. Com isso, retornamos o mínimo de todos  $[\text{custo}(i) + \text{dist}(i,1)]$  valores.

Para calcular o custo  $(i)$  usando a Programação Dinâmica, para isso precisamos ter alguma relação recursiva em termos de subproblemas, então vamos definir um termo  $C(S, i)$  como o custo do caminho de custo mínimo visitando cada vértice no conjunto  $S$  exatamente uma vez, começando em 1 e terminando em  $i$ . Começamos com todos os subconjuntos de tamanho 2 e calculamos  $C(S, i)$  para todos os subconjuntos onde  $S$  é o subconjunto, depois calculamos  $C(S, i)$  para todos os subconjuntos  $S$  de tamanho 3 e assim por diante.

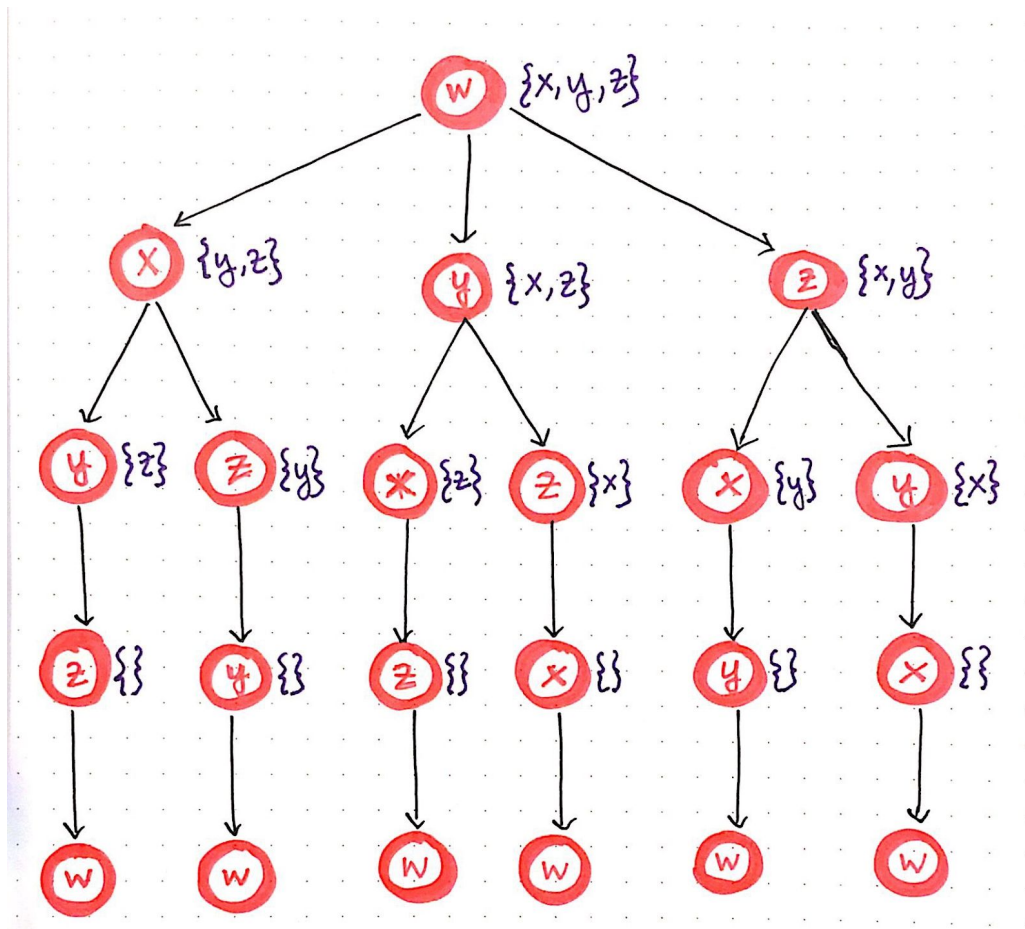
Baseamos a criação do algoritmo na seguinte fórmula :

Se o tamanho de S for 2, S deverá ser:

$\{1, i\}$ ,  $C(S, i) = \text{dist}(1, i)$

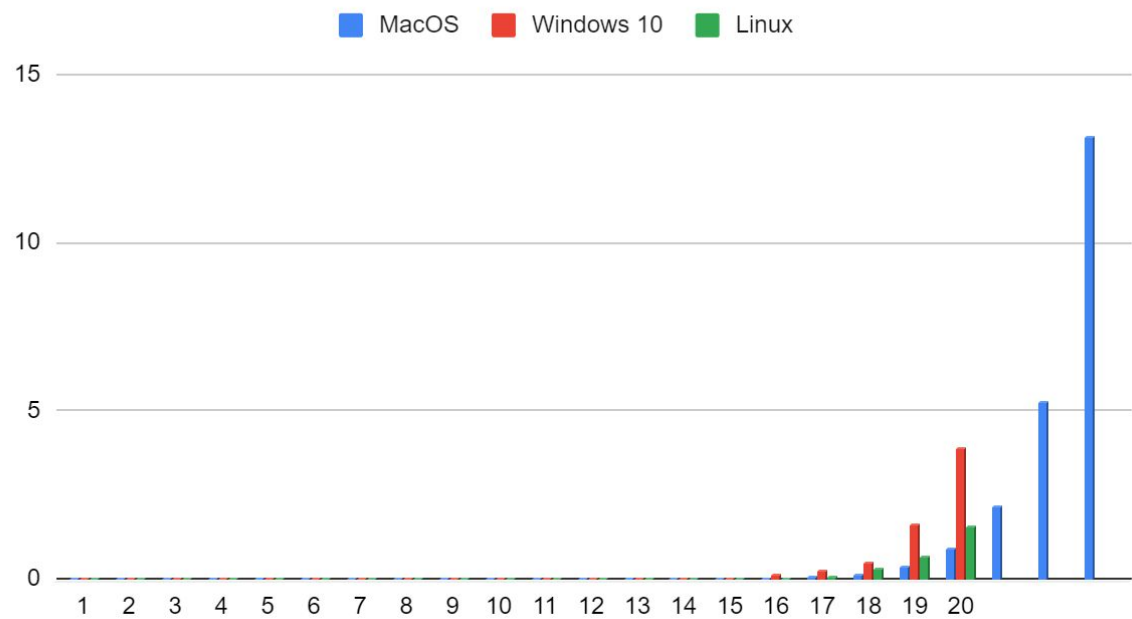
Caso contrário, se o tamanho de S for maior que 2:

$C(S, i) = \min \{C(S - \{i\}, j) + \text{dis}(j, i)\}$  onde  $j$  pertence a S,  $j! = I$  e  $j! = 1$



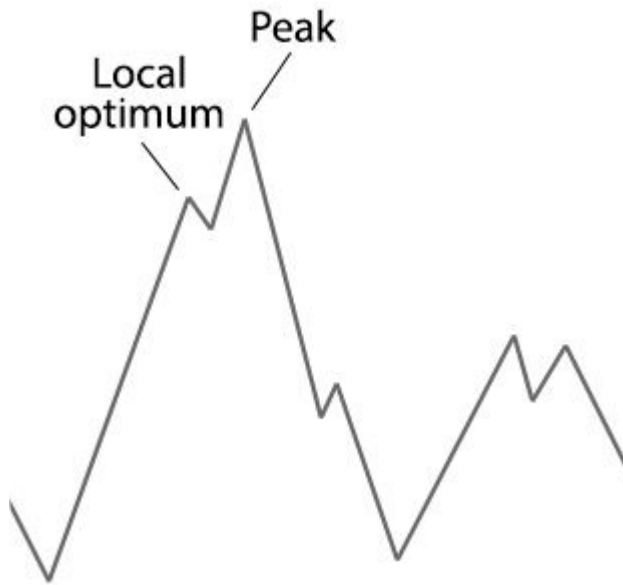
Com isso para calcular a complexidade do algoritmo temos no máximo subproblemas  $O(n * 2n)$ , e cada um leva um tempo linear para resolver. O tempo total de execução é, portanto,  $O(n^2 * 2n)$ . Sendo que o uso do espaço de memória também é exponencial. Portanto, essa abordagem também é inviável mesmo para um número um pouco maior de vértices. No caso dos testes a partir de  $n = 23$  as máquinas travam pelo excesso de uso da memória, mas o tempo do algoritmo se mantém baixo. Ou seja a execução do algoritmo depende primariamente do tamanho da nossa memória, podendo ser executado para valores maiores de  $n$ .

# Programação Dinâmica



## Algoritmo Genético

O algoritmo genético é simples de entender pois é baseado no processo da seleção natural, isso significa que ele utiliza as propriedades fundamentais do processo de seleção natural e aplica ao problema.



Contudo, diferente dos demais algoritmos vistos acima, o algoritmo genético não foi feito para encontrar a solução ótima do problema, e sim uma solução local ótima, uma aproximação do ótimo.

O processo do algoritmo genético é dividido em:

1. **Inicialização:** Criamos uma população inicial de possíveis rotas, essas rotas são geradas de forma randômica.
2. **Avaliação:** Cada rota da população é avaliada e calculamos o seu "fitness". Esse valor é calculado com base no requerimento de "quanto menor a distância, melhor é a rota", logo esse valor aumenta conforme a distância diminui.
3. **Seleção:** Queremos constantemente melhorar o "fitness" global da nossa população. A seleção serve para descartar as rotas com pior "fitness" e apenas os melhores serão selecionados para compor a nova geração da população.
4. **Cruzamento:** Durante o cruzamento criamos novas rotas ao combinar aspectos das rotas selecionados, na chance de que combinando características de duas rotas geramos uma com fitness ainda melhor.
5. **Mutação:** Precisamos adicionar um pouco de aleatoriedade na "genética" de nossa população, senão cada combinação de soluções que poderíamos criar já estaria em nossa população inicial.
6. **Repetição:** Agora que temos nossa próxima geração da população, podemos começar novamente a partir do segundo passo até que chegamos em uma condição de parada, nesse caso é o número de gerações que queremos produzir até gerar um resultado final.

As estruturas de dados utilizadas para compor este algoritmo foram:

1. **TravelRoute** que contém:

- a. um vetor de inteiros que armazena a rota a ser percorrida. A Partir dela indexamos a posição no **Grafo** e no vetor de cidades.
- b. Um grafo com uma matriz de adjacência armazenando a distância para cada cidade.
- c. Um vetor de cidades (a.k.a **Point**) que armazena as cidades correspondentes a rota.
- d. Variáveis que armazenam o fitness e a distância da rota.
- e. Métodos para operarem sobre essa estrutura, como calcular o fitness, a distância com base no grafo e rota, trocar uma cidade com outra na rota, gerar uma rota completamente aleatória...

2. **Population** que contém:

- a. um vetor de **TravelRoute**, que armazena as rotas da população.
- b. Métodos para adicionar uma rota em uma posição no vetor, obter uma rota e obter a rota com melhor fitness no vetor.

A complexidade do algoritmo genético depende de 5 variáveis:

1. O número de cidades de entrada = A;
2. O tamanho da população = B;
3. O tamanho da população a ser criada no método de seleção por torneio = C.
4. O número de gerações de população a ser gerado = D;
5. A taxa de mutação = E.

Partindo da chamada principal `Genetic::run`, consideramos como operação relevante o método `Genetic::evolve_population`, logo temos  $(1+D)*E_p$ , sendo  $E_p$  = a complexidade de `Genetic::evolve_population`.

Agora partindo de `Genetic::evolve_population`, consideramos como operação relevante o número de chamadas aos métodos `Genetic::tournament_selection`, `Genetic::crossover` e `Genetic::mutate`, portanto temos  $E_p = (B - 1) \times (2*Ts + Cr + Mt)$ .

Agora partindo de `Genetic::tournament_selection` tomamos como operação relevante o número de vezes que obtemos uma rota da população p, nesse caso é igual ao número de repetições do loop for, que é C.

Agora partindo de `Genetic::crossover` consideramos como operação relevante o número de atribuições a posição do vetor `child_cities`.

Partindo do primeiro loop, considerando o caso médio em que a diferença do valor entre start\_pos e end\_pos (variáveis cujo valor é randômico) corresponda a metade do tamanho do TravelRoute a, temos que no primeiro *for* A/2 atribuições, uma vez que o tamanho de qualquer rota é igual ao número de cidades.

Como foi atribuído metade das cidades em child\_cities, consideramos a outra metade como sendo null, então no segundo *for*, responsável por adicionar o restante das cidades da rota b que child\_cities, o *if* só será verdadeiro metade das vezes e o *for* interno só vai realizar apenas uma atribuição a cada iteração do *for* externo, logo a complexidade é de A/2 atribuições. O terceiro *for* não realiza atribuição ao child\_cities.

Por fim concluímos que a complexidade de Genetic::crossover = A.

Agora partindo de Genetic::mutate, consideramos como operação relevante o número de swaps feitos na rota. Nesse caso a complexidade é de (A - 1) \* E.

Temos até então que:

Principal: (1 + D) \* Ep

Ep = (B - 1) \* (2 \* Ts + Cr + Mt)

Ts = C

Cr = A

Mt = (A - 1) \* E

Fazendo as substituições temos:

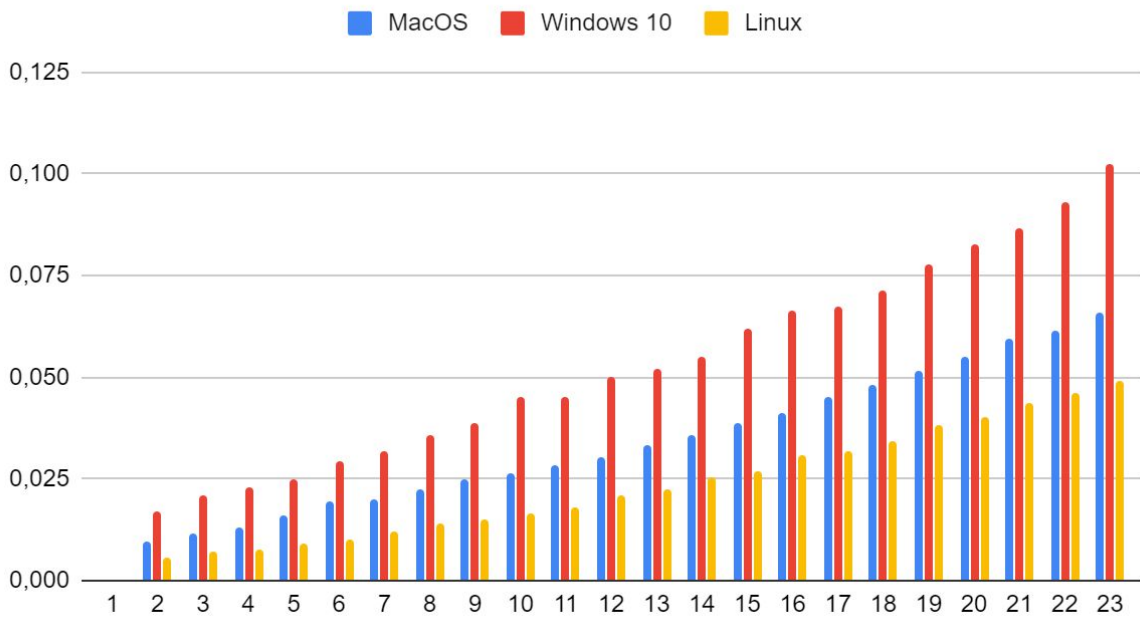
Considerando B = 50, C = 5, D = 100, E = 0,015

Complexidade Geral = (1+D) \* (B - 1) \* ( 2\*C + A + ((A - 1) \* E))  
= (101) \* (49) \* (10 + A + ((A - 1) \* 0,015))  
= 4949 \* (10 + A + ((A - 1) \* 0,015))  
= 4949 \* (10 + A + ( 0,015A - 0,015))  
= 4949 \* (9,985 + 1,015A)  
= 49415765 + 5023235A  
ou seja O(n).

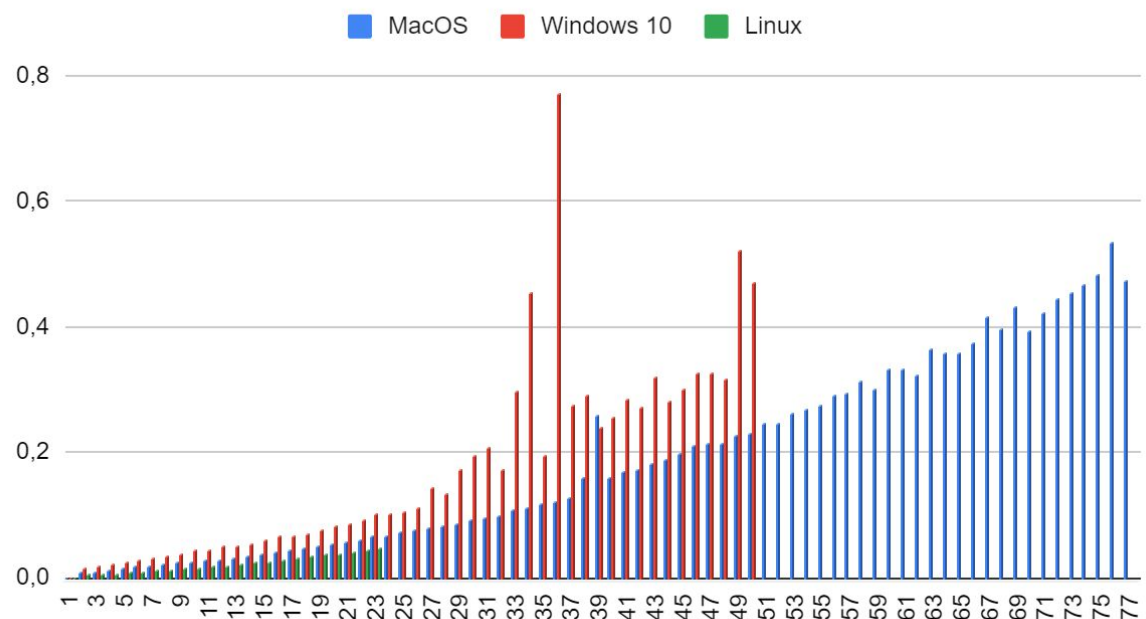
Tomando A = 20, Resultado = 149.880.465 operações relevantes.

Considerando os valores das variáveis B, C, D e E constantes, podemos definir uma função de complexidade baseada em A, para A > 1:  
Genetic(A) -> = 49415765 + 5023235A

## Algoritmo Genético



## Algoritmo Genético



## Conclusão

Com as implementações dos 4 algoritmos, podemos notar que o força bruta apesar de ser facilmente implementado não é nem um pouco eficiente, consumindo muito tempo, deixando-o inviável de ser usado.

O algoritmo de branch and bound ainda nos fornece a solução ótima, e age melhor que o força bruta a partir das otimizações de corte de caminhos irrelevantes para o cálculo da solução. Nosso algoritmo apresentou uma melhoria em comparação ao força bruta, mas ainda sim se manteve inviável para entradas maiores, e pode ser melhorado com mais otimizações de corte sofisticadas.

Com o algoritmo dinâmico foi possível resolver problemas de entradas um pouco maiores estando limitados ao tamanho da memória que a máquina possui, lembrando que o custo da memória também é exponencial neste caso.

Com o algoritmo genético foi possível resolver problemas de entradas bem maiores em menos tempo, comparado aos demais algoritmos, com consumo de recursos menor comparado ao dinâmico, embora seu resultado não seja ótimo e sim aproximado.

Um das principais dificuldades foi a falta de conhecimento na parte de criação de algoritmos genéticos, e o pouco conhecimento na parte de algoritmos dinâmicos, o estudo para realizar esses algoritmos foi maior que os demais. Houveram dificuldades para nos aproximarmos mais da linguagem C++, e utilizar melhor os recursos que ela nos fornece, além de criar um makefile e toda uma estrutura de projeto do zero.

## Bibliografia

- [Creating a genetic algorithm for beginners](#)
- [Applying a genetic algorithm to the traveling salesman problem](#)
- [Análise formal da complexidade de algoritmos genéticos](#)
- [Travelling Salesman Dynamic](#)
- [Dynamic programming](#)
- [cplusplus.com - The C++ Resources Network](#)