# Problema do Mapa
# (Grafo de cidades)

Rafael de Souza Terra Mateus Coutinho Marim

8 de dezembro de 2017

## 1 Descrição do problema

O problema do mapa ou grafo de cidades consiste em um grafo em que as cidades são representadas por tuplas com valores X e Y que representam as coordenadas da cidade em um mapa.
O objetivo do problema é encontrar o caminho de menor custo entre uma cidade A e outra B.
As operações aplicáveis nesse problema consistem em sair de uma cidade e ir até a cidade vizinha (ou conectada) a ela.

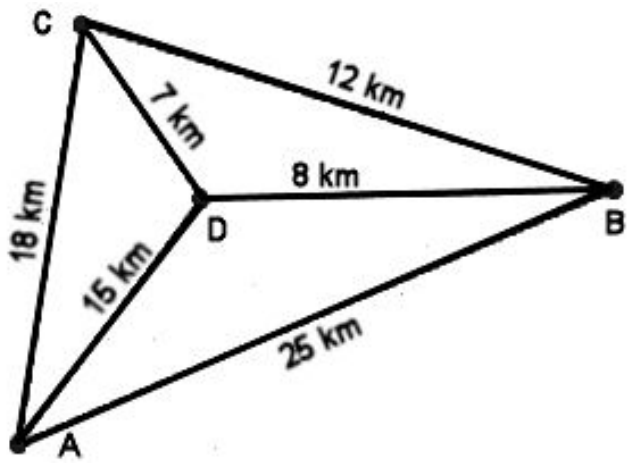


Figura 1: Exemplo de grafo de cidades.

# 2 Implementação

## 2.1 Ambiente utilizado

- Linguagem: Python 3 e C++
- Ferramentas: Bibliotecas time, math e Queue
- Windows 10 e Ubuntu 17.04

## 2.2 Classes principais

Para facilitar a implementação e utilização dos algoritmos foi criada uma superclasse chamada Solver do qual as classes dos algoritmos herdam os métodos que lhe são comuns e mais duas classes auxiliares chamadas State e Solution para representar os estados do problema e sua solução respectivamente.

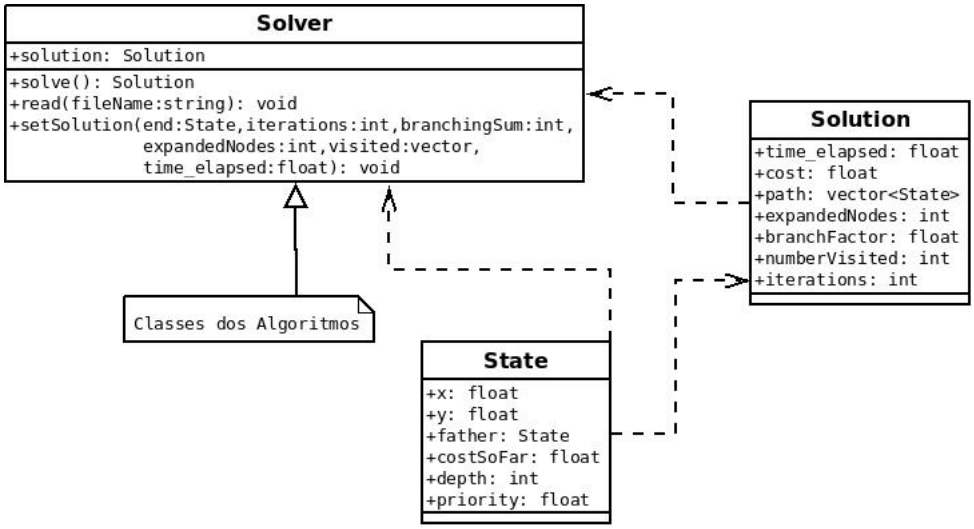


Figura 2: Diagrama de classes do sistema.

## 2.3 Representação Computacional do Grafo de Cidades

Para a representação do grafo de cidades foi utilizada uma tabela hash mapeando cada estado ou cidade para uma lista de adjacências representando as cidades vizinhas com os custos para se chegar a cada uma delas a partir da cidade representada por aquela posição na tabela.
Logo, em cada posição da lista de adjacências, temos uma tupla em que a primeira posição é uma cidade vizinha e a segunda é o custo para se chegar até ela.
A função de hashing utilizada para mapear a cidade as suas vizinhas, é o valor obtido da multiplicação das coordenadas x e y da cidade.

# 3 Execuções

Métodos disponíveis:

- BTCK - Backtracking;
- BFS - Breadth First Search;
- DFSL - Depth First Search (Limited);
- OS - Ordered Search;
- GS - Greedy Search (Euclidean Distance);
- AS - A* Search (Euclidean Distance);
- IDAS - IDA* Search (Euclidean Distance).

```
python3 main.py --instance Instancias/589.inst --method AS --x 9 --y 102 --x1 125 --y1 140
```

Figura 3: Exemplo de comando para execução do A* na instância 589.ins da cidade (9, 102) até a (125, 140).

Nos algoritmos informados, foram implementadas as heurísticas das distâncias euclidianas e de manhattan, mas como a diferença nos resultados não foram expressivas, as execuções das tabelas se referem apenas a euclidiana.

## 3.1 Comparações

Tabela 1: 91 elementos ponto (417, 285) até (339, 101)

| **Algoritmo** | **Tempo de execução** | **Custo** | **Profundidade** |
|---|---|---|---|
| Backtracking | Stack Overflow | - | - |
| Busca em largura | 0.0003716945648 | 1249.894 | 3 |
| Busca em profundidade (maxDepth = 5) | 0.000239 | 1082.925 | 4 |
| Busca ordenada | 0.00111 | 1249.894 | 3 |
| Busca Gulosa | 0.0002694 | 1266.518 | 4 |
| A* | 0.000534 | 1082.925 | 4 |
| IDA* | - | - | - |

Tabela 2: 91 elementos ponto (417, 285) até (339, 101)

| **Algoritmo** | **Estados visitados** | **Fator de ramificação** | **Iterações** |
|---|---|---|---|
| Backtracking | - | - | - |
| Busca em largura | 20 | 2.315789474 | 19 |
| Busca em profundidade (maxDepth = 5) | 14 | 1.3076923076923077 | 13 |
| Busca ordenada | 29 | 1.851851852 | 27 |
| Busca Gulosa | 8 | 2.571428571 | 7 |
| A* | 14 | 2.53846 | 13 |
| IDA* | - | - | - |

Tabela 3: 964 elementos ponto (94, 448) até (74, 445)

| **Algoritmo** | **Tempo de execução** | **Custo** | **Profundidade** |
|---|---|---|---|
| Backtracking | Stack Overflow | - | - |
| Busca em largura | 0.01586699486 | 1336.236 | 3 |
| Busca em profundidade (maxDepth = 4) | 0.0194272995 | 1353.25 | 3 |
| Busca ordenada | 0.1704976559 | 990.938 | 3 |
| Busca Gulosa | 0.005874633789 | 1602.5208 | 7 |
| A* | 0.01584124565 | 990.938 | 3 |
| IDA* | - | - | - |

Tabela 4: 964 elementos ponto (94, 448) até (74, 445)

| **Algoritmo** | **Estados visitados** | **Fator de ramificação** | **Iterações** |
|---|---|---|---|
| Backtracking | - | - | - |
| Busca em largura | 78 | 6.987012987 | 77 |
| Busca em profundidade (maxDepth = 4) | 88 | 1.045977011 | 87 |
| Busca ordenada | 334 | 2.557228916 | 332 |
| Busca Gulosa | 28 | 8.333333333 | 27 |
| A* | 62 | 7.426229508 | 61 |
| IDA* | - | - | - |

# 4 Geração de Instâncias

## 4.1 Lógica da Geração de Instâncias Aleatórias

Para gerar uma instância o usuário passa como parâmetro do programa o número de pontos (cidades) desejados, o mínimo e máximo dos eixos X e Y, o número de arestas que vão estar presentes no grafo e o intervalo em que vai estar o custo adicional de cada aresta.
Com esses parâmetros, o algoritmo vai gerar a quantidade de pontos pedida aleatóriamente e depois vai remover os duplicados, com esses pontos vão ser criadas as arestas com os custos sendo a distância euclidiana mais um custo adicional aleatório dentro do intervalo passado, como último passo, as arestas repitidas são removidas da instância.
O arquivo gerado estará no formato .inst e cada linha terá a seguinte estrutura:
$x$ $y$ $x1$ $y1$ $c$
Sendo a cidade de coordenadas $(x, y)$ conectada a cidade de coordenadas $(x1, y1)$ com custo $c$.

```
1 349 244 46 108 335.131
2 410 234 459 217 58.0519
3 452 154 66 489 519.289
4 57 263 67 401 147.189
5 215 96 369 220 199.327
6 446 103 82 103 371.591
```

Figura 4: Exemplo de conteúdo do arquivo de instância do problema.

```
./randomInstanceGenerator 367 0 200 0 200 600 0 200
```

Figura 5: Comando para gerar instância com 367 pontos, coordenadas variando de 0 a 200, 600 arestas e custo adicional de 0 a 200.

# 5 Dificuldades encontradas

## 5.1 Linguagem

O programa foi desenvolvido usando python 3, porém nenhum dos integrantes possuía domínio nesta linguagem. Portanto muito tempo foi gasto com o aprendizado da linguagem e correção de erros relacionados à mesma.

## 5.2 Instâncias

Não conseguimos encontrar instâncias para o problema então tivemos que gerar as nossas próprias.

### 5.3 Backtracking

Devido a geração aleatória das nossas instâncias, elas possuem alguns ciclos fazendo o Backtracking estourar a pilha de recursão em algumas instâncias.

## 6 Distribuição do trabalho

### 6.1 Mateus

- Implementação dos algoritmos de busca em largura, busca ordenada, A* e busca gulosa.
- Criação do algoritmo de geração de instâncias aleatórias.
- Implementação da classe solver.
- Execução dos métodos informados.

### 6.2 Rafael

- Implementação dos algoritmos de busca em profundidade, backtracking e IDA*
- Implementação das classes solution e state.
- Execução dos métodos não informados.