

ANALYSIS OF TRAVELLING SALESPERSON PROBLEM (TSP)

DA Projeto 2

2023/2024

Turma 4

Elementos do Grupo:

1. Ângelo Oliveira(202207798)
2. Bruno Fortes(202209730)
3. José Costa(202207871)



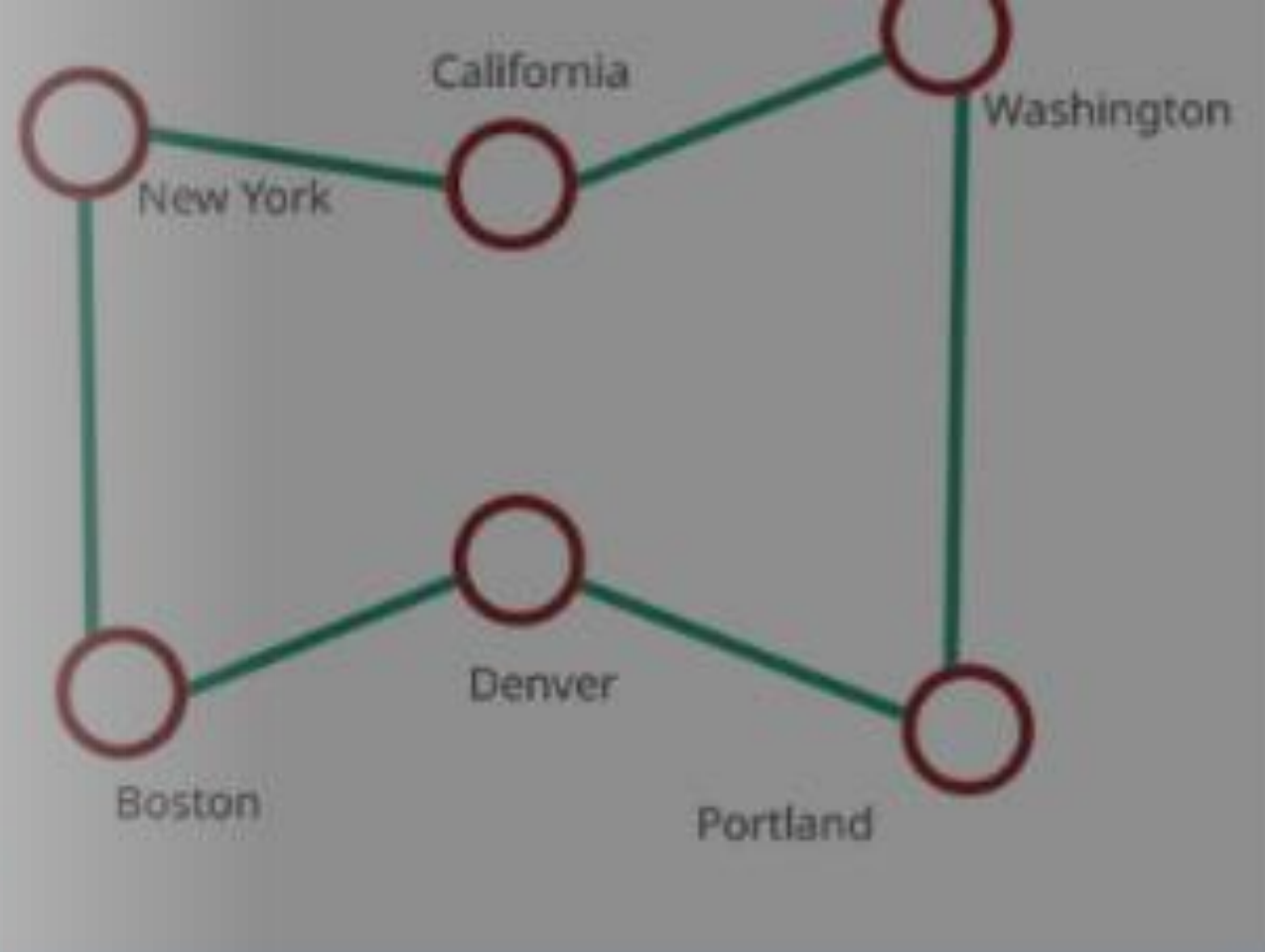


Objetivo

- O objetivo do trabalho é explorar e encontrar soluções para um problema de otimização de larga escala , através de heurísticas e algoritmos de otimização.

Classes

- Algorithm: Este módulo contém os algoritmos utilizados para realizar operações específicas no grafo, como buscas em profundidade, heurísticas para melhoria de caminho, criação de MST, cálculo de distâncias com o método de haversine e os algoritmos de aproximação para resolver o TSP.





Classes

- LoadingFunctions: Ponto de partida da nossa aplicação, trata do processamento dos dados fornecidos nos ficheiros .csv lendo-os e organizando-os nas estruturas por nós escolhidas
- UI: Interface do utilizador para a navegação do programa.

O Grafo

A classe Vertex é uma implementação de um vértice de grafo, projetada para suportar várias operações comuns em algoritmos de grafos, incluindo travessias, busca de caminhos mínimos e manipulação de estruturas de dados associadas a grafos. Ela encapsula tanto os dados quanto os métodos necessários para interagir com um grafo de forma eficiente e eficaz.

```
1 class Vertex {
2 public:
3     Vertex(int id, double intitude, double latitude);
4     Vertex(int id);
5     int getId() const;
6     int getIndegree() const;
7     int getOutdegree() const;
8     int getDegree() const;
9     double getLongitude() const;
10    double getLatitude() const;
11    double getDist() const;
12    int getNum() const;
13    int getLow() const;
14    int getTentativas() const;
15    bool isVisited() const;
16    bool isProcessing() const;
17    Edge* getPath() const;
18    std::vector<Edge*> getAdj() const;
19    std::vector<Edge*> getInc() const;
20    void addAdjEdge(int edgeId, Edge* edge);
21    void addIncEdge(int edgeId, Edge* edge);
22    void deleteIncEdge(int edgeId);
23    void deleteAdjEdge(int edgeId);
24    void setDist(double newDist);
25    void setId(int newId);
26    void setIntitude(double newIntitude);
27    void setLatitude(double newLatitude);
28    void setTentativas(int newTentativas);
29    void setAdj(std::vector<Edge*> newAdj);
30    void setVisited(bool visited);
31    void setPath(Edge* e);
32    void setOutdegree(int outdegree);
33    void setIndegree(int indegree);
34    void setNum(int i);
35    void setLow(int i);
36    void setProcessing(const bool & _processing);
37    bool operator<(Vertex & vertex) const;
38    bool operator==(Vertex & vertex) const;
39    std::vector<Edge*> orderEdges();
40    void updateDegrees();
41    friend class MutablePriorityQueue<Vertex>;
42 protected:
43     int id;
44     double intitude;
45     double latitude;
46     std::vector<Edge*> incomingEdges;
47     std::vector<Edge*> adjacentEdges;
48     bool visited = false;
49     bool processing = false;
50     unsigned int indegree = 0;
51     unsigned int outdegree = 0;
52     double dist = 0;
53     Edge* path = nullptr;
54     int queueIndex = 0;
55     int tentativas = 0;
56     int num = 0;
57     int low = 0;
58 };
```

```
1 class Edge {
2 public:
3     Edge(Vertex* source, Vertex* destination, int edgeId, double weight);
4     [[nodiscard]] Vertex* getSource() const;
5     [[nodiscard]] Vertex* getDestination() const;
6     [[nodiscard]] double getWeight() const;
7     [[nodiscard]] double getPheromones() const;
8     [[nodiscard]] int getId() const;
9     bool isSelected() const;
10
11     void setSource(Vertex* newSource);
12     void setDestination(Vertex* newDestination);
13     void setWeight(double newWeight);
14     void setId(int newId);
15     void setSelected(bool selected);
16     void setPheromones(double pheromones);
17 private:
18     Vertex* source;
19     Vertex* destination;
20     double weight;
21     int id;
22     bool selected;
23     double pheromones;
24 };
```

A classe Edge é essencial para representar e manipular as conexões entre vértices em algoritmos de grafos, permitindo operações como cálculo de caminhos mínimos, construção de MSTs, e aplicação de algoritmos.

O Grafo

A classe Graph é uma implementação abrangente de um grafo, permitindo operações completas de criação, modificação e consulta. Alguns exemplos de métodos fornecidos são:

- Adicionar e remover vértices/arestas
- Calcular distâncias geográficas
- Imprimir informações detalhadas sobre vértices e/ou arestas.

```
1 class Graph {
2 public:
3     ~Graph();
4     Edge* findEdge(int sourceId, int destId) const;
5     Vertex* findVertex(int id) const;
6     bool addVertex(int id , double longitude , double latitude);
7     bool removeVertex(int id);
8     bool addEdge(int sourceId, int destId,int edgeId, double weight) ;
9     bool removeEdge(int sourceId, int destId) const;
10    bool addBidirectionalEdge(int sourceId, int destId,int edgeId, double weight) ;
11    int getNumVertex() const;
12    Clock getClock();
13    std::unordered_map<int, Vertex*> getVertexSet() const;
14    std::unordered_map<int, Edge*> getEdgeSet() const;
15
16    double Harverstein(double longitude1, double latitude1, double longitude2, double latitude2) const;
17    void clear();
18    void printNodesContente() const;
19    void printGraphInfo() const;
20    void populate_in_and_out_degree();
21    bool makeFullyConnected() ;
22
23    void orderVertexAdj(Vertex * v);
24
25 private:
26
27     Clock clock;
28
29     std::unordered_map<int, Edge*> edgeSet;
30     std::unordered_map<int, Vertex*> vertexSet;
31 };
32
```

Descrição da leitura do dataset a partir dos ficheiros dados

- A leitura e processamento dos datasets são cruciais para popular o grafo com todas as cidades em que o mercador irá passar.
- Esta operação é realizada por meio de várias classes e métodos específicos, garantindo que os dados sejam carregados de forma correta e eficientemente no sistema.

Função responsável por carregar os dados dos caminhos possíveis entre pontos de referência, a partir de um arquivo CSV.

```
1 void loadMediamGraphs(Graph * g, const std::string& path, const int& graph){
2     std::string file_name2 = "Edges.csv";
3     std::string full_path = path + file_name2;
4     std::string file_name;
5     int edgid = 0;
6     switch (graph) {
7         case 0:
8             file_name = "edges_25.csv";
9             break;
10        case 1:
11            file_name = "edges_50.csv";
12            break;
13        case 2:
14            file_name = "edges_75.csv";
15            break;
16        case 3:
17            file_name = "edges_100.csv";
18            break;
19        case 4:
20            file_name = "edges_200.csv";
21            break;
22        case 5:
23            file_name = "edges_300.csv";
24            break;
25        case 6:
26            file_name = "edges_400.csv";
27            break;
28        case 7:
29            file_name = "edges_500.csv";
30            break;
31        case 8:
32            file_name = "edges_600.csv";
33            break;
34        case 9:
35            file_name = "edges_700.csv";
36            break;
37        case 10:
38            file_name = "edges_800.csv";
39            break;
40        case 11:
41            file_name = "edges_900.csv";
42            break;
43        default:
44            std::cerr << "Choose a valid graph";
45            return;
46    }
47    std::string numbers = extractIntegers(file_name);
48    int i = std::stoi(numbers);
49    std::ifstream file(full_path);
50    if (!file.is_open()) {
51        std::cerr << "Failed to open the CSV file." << std::endl;
52    }
53    std::string line;
54    getline(file, line);
55    while (getline(file, line) && i > 0) {
56        line.erase(std::remove(line.begin(), line.end(), '\r'), line.end());
57        std::stringstream lineStream(line);
58        std::vector<std::string> tokens;
59        std::string token;
60        while (getline(lineStream, token, ',')) {
61            tokens.push_back(token);
62        }
63        std::string id = tokens[0];
64        std::string longitude = tokens[1];
65        std::string latitude = tokens[2];
66        RemoveTerminations(latitude);
67        try {
68            int id_v1 = std::stoi(id);
69            double longitude = std::stod(longitude);
70            double latitude = std::stod(latitude);
71            g->addVertex(id_v1, longitude, latitude, i);
72        } catch (const std::exception& e) {
73            continue;
74        }
75        i--;
76    }
77    file.close();
78    full_path = path + "/" + file_name;
79    std::ifstream file2(full_path);
80    if (!file2.is_open()) {
81        std::cerr << "Failed to open the CSV file." << std::endl;
82    }
83    std::string line2;
84    while (getline(file2, line2)) {
85        line2.erase(std::remove(line2.begin(), line2.end(), '\r'), line2.end());
86        std::stringstream line2Stream(line2);
87        std::string id_v1 = id_v2 = distance;
88        std::getline(line2Stream, id_v1, ',');
89        std::getline(line2Stream, id_v2, ',');
90        std::getline(line2Stream, distance, ',');
91        try {
92            int id_v1 = std::stoi(id_v1);
93            int id_v2 = std::stoi(id_v2);
94            double v_distances = std::stod(distance);
95            Vertex v1 = Vertex(id_v1, 0, 0);
96            Vertex v2 = Vertex(id_v2, 0, 0);
97            Edge edge = Edge(v1, v2, edgid, v_distances);
98            g->addBidirectionalEdge(v1.getId(), v2.getId(), edge.getId(), edge.getWeight());
99        } catch (std::exception& e) {
100            continue;
101        }
102        edgid += 2;
103    }
104    file2.close();
105 }
```

Descrição da leitura do dataset a partir dos ficheiros dados

Esta função é responsável por carregar os dados dos “ToyGraphs”(Grafos mais pequenos, puramente de teste), a partir de um arquivo CSV.

```
1 void LoadToyGraphs(Graph * g , const std::string& path , const int& graph){
2     std::string file_name;
3     int edgeId = 0;
4     switch (graph) {
5     case 0:
6         file_name = "shipping.csv";
7         break;
8     case 1:
9         file_name = "stadiums.csv";
10        break;
11    case 2:
12        file_name = "tourism.csv";
13        break;
14    case 3:
15        file_name = "myGraph.csv";
16        break;
17    default:
18        std::cerr << "Choose a valid graph";
19        return;
20    }
21    std::string full_path = path + '/' + file_name;
22    std::ifstream file(full_path);
23    if (!file.is_open()) {
24        std::cerr << "Failed to open the CSV file." << std::endl;
25    }
26    std::string line;
27    getline(file, line);
28    while (getline(file, line)) {
29        line.erase(std::remove(line.begin(), line.end(), '\r'), line.end());
30        std::stringstream lineStream(line);
31        std::string id_v1 , id_v2 , distance;
32        getline(lineStream , id_v1 , ',');
33        getline(lineStream , id_v2 , ',');
34        getline(lineStream , distance , ',');
35        int id_v1 = stoi(id_v1);
36        int id_v2 = stoi(id_v2);
37        double v_distances = stod(distance);
38        g->addVertex(id_v1 , DBL_MAX , DBL_MAX);
39        g->addVertex(id_v2 , DBL_MAX , DBL_MAX);
40        g->addEdge(id_v1 , id_v2 ,edgeId, v_distances);
41        edgeId++;
42        g->addEdge(id_v2 , id_v1 ,edgeId, v_distances);
43        edgeId++;
44    }
45    file.close();
46 }
47 }
```

Esta função é responsável por carregar os dados dos “RealWorldGraphs”(Grafos que se aplicam a situações reais, em que há bastantes mais caminhos e pontos para visitar), a partir de um arquivo CSV.

```
1 void LoadRealWorldGraphs(Graph * g , const std::string& path , const int& graph){
2     std::string file_name;
3     int edgeId = 0;
4     switch (graph) {
5     case 0:
6         file_name = "/graph1/nodes.csv";
7         break;
8     case 1:
9         file_name = "/graph2/nodes.csv";
10        break;
11    case 2:
12        file_name = "/graph3/nodes.csv";
13        break;
14    default:
15        std::cerr << "Choose a valid graph";
16        return;
17    }
18    std::string full_path = path + '/' + file_name;
19    std::ifstream file(full_path);
20    if (!file.is_open()) {
21        std::cerr << "Failed to open the CSV file." << std::endl;
22    }
23    std::string line;
24    getline(file, line);
25    while (getline(file, line)) {
26        line.erase(std::remove(line.begin(), line.end(), '\r'), line.end());
27        std::stringstream lineStream(line);
28        std::string id , intitudo , latitude;
29        getline(lineStream , id , ',');
30        getline(lineStream , intitudo , ',');
31        getline(lineStream , latitude , ',');
32        int id_v1 = stoi(id);
33        double intitudo_ = stod(intitudo);
34        double latitude_ = stod(latitude);
35        g->addVertex(id_v1 , intitudo_ , latitude_);
36    }
37    file.close();
38    switch (graph) {
39    case 0:
40        file_name = "/graph1/edges.csv";
41        break;
42    case 1:
43        file_name = "/graph2/edges.csv";
44        break;
45    case 2:
46        file_name = "/graph3/edges.csv";
47        break;
48    default:
49        std::cerr << "Choose a valid graph";
50        return;
51    }
52    full_path = path + '/' + file_name;
53    std::ifstream file2(full_path);
54    if (!file2.is_open()) {
55        std::cerr << "Failed to open the CSV file." << std::endl;
56    }
57    std::string line2;
58    getline(file2, line2);
59    while (getline(file2, line2)) {
60        line2.erase(std::remove(line2.begin(), line2.end(), '\r'), line2.end());
61        std::stringstream lineStream2(line2);
62        std::string id_v1 , id_v2 , distance;
63        getline(lineStream2 , id_v1 , ',');
64        getline(lineStream2 , id_v2 , ',');
65        getline(lineStream2 , distance , ',');
66        int id_v1 = stoi(id_v1);
67        int id_v2 = stoi(id_v2);
68        double v_distances = stod(distance);
69        Vertex v1 = Vertex(id_v1 , 0 , 0);
70        Vertex v2 = Vertex(id_v2 , 0 , 0);
71        Edge edge = Edge(&v1 , &v2 ,edgeId, v_distances);
72        g->addBidirectionalEdge(v1.getId() , v2.getId() , edge.getId() , edge.getWeight());
73        edgeId ++;
74    }
75    file2.close();
76 }
```


Ui

Menu Interativo:

- O sistema conta com um menu interativo implementado no arquivo UI.cpp. Este menu oferece uma interface de usuário amigável e fácil de usar, que contém a implementação das funções necessárias para interagir com o usuário e conduzir as operações principais do Sistema de algoritmos de routing



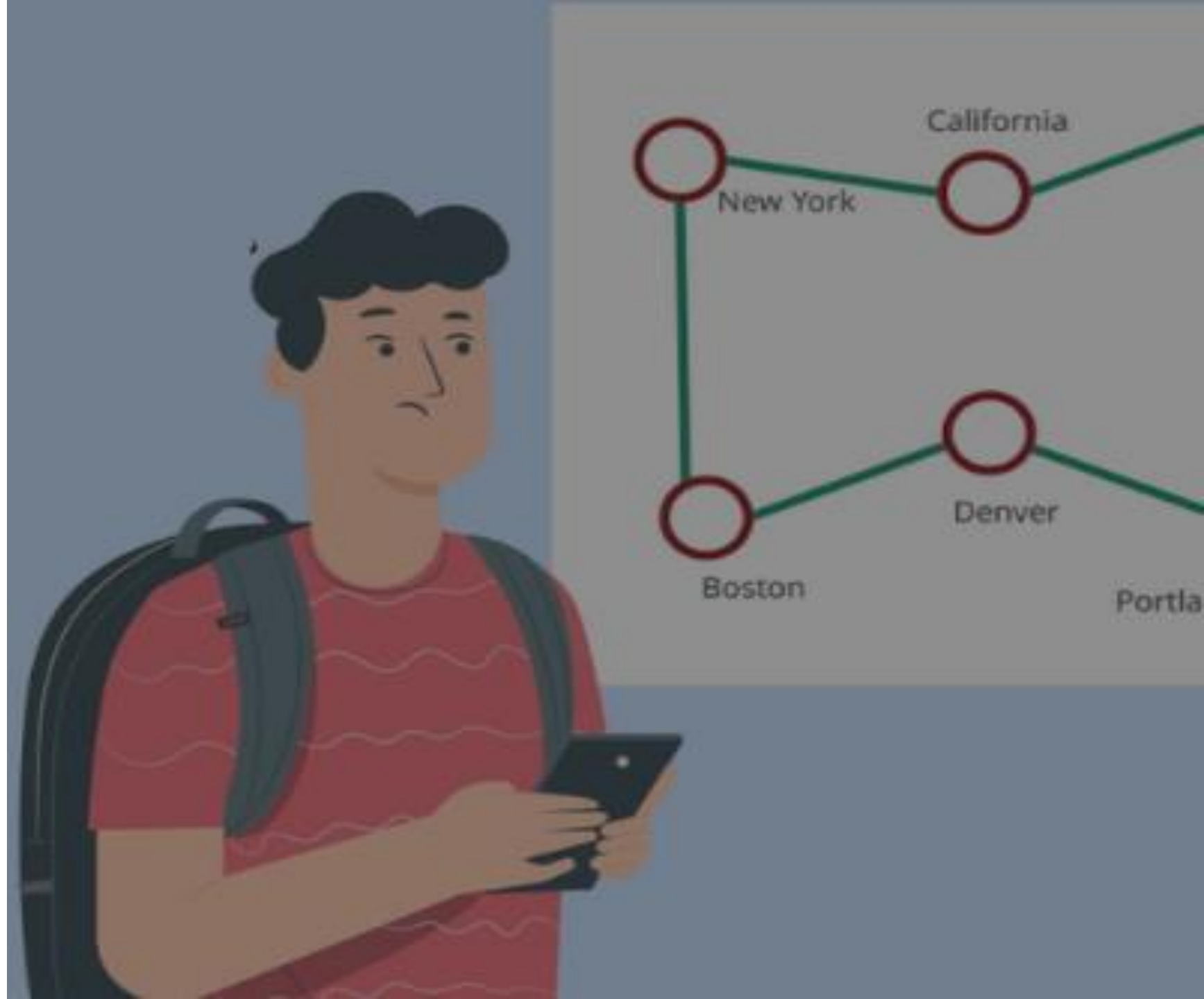


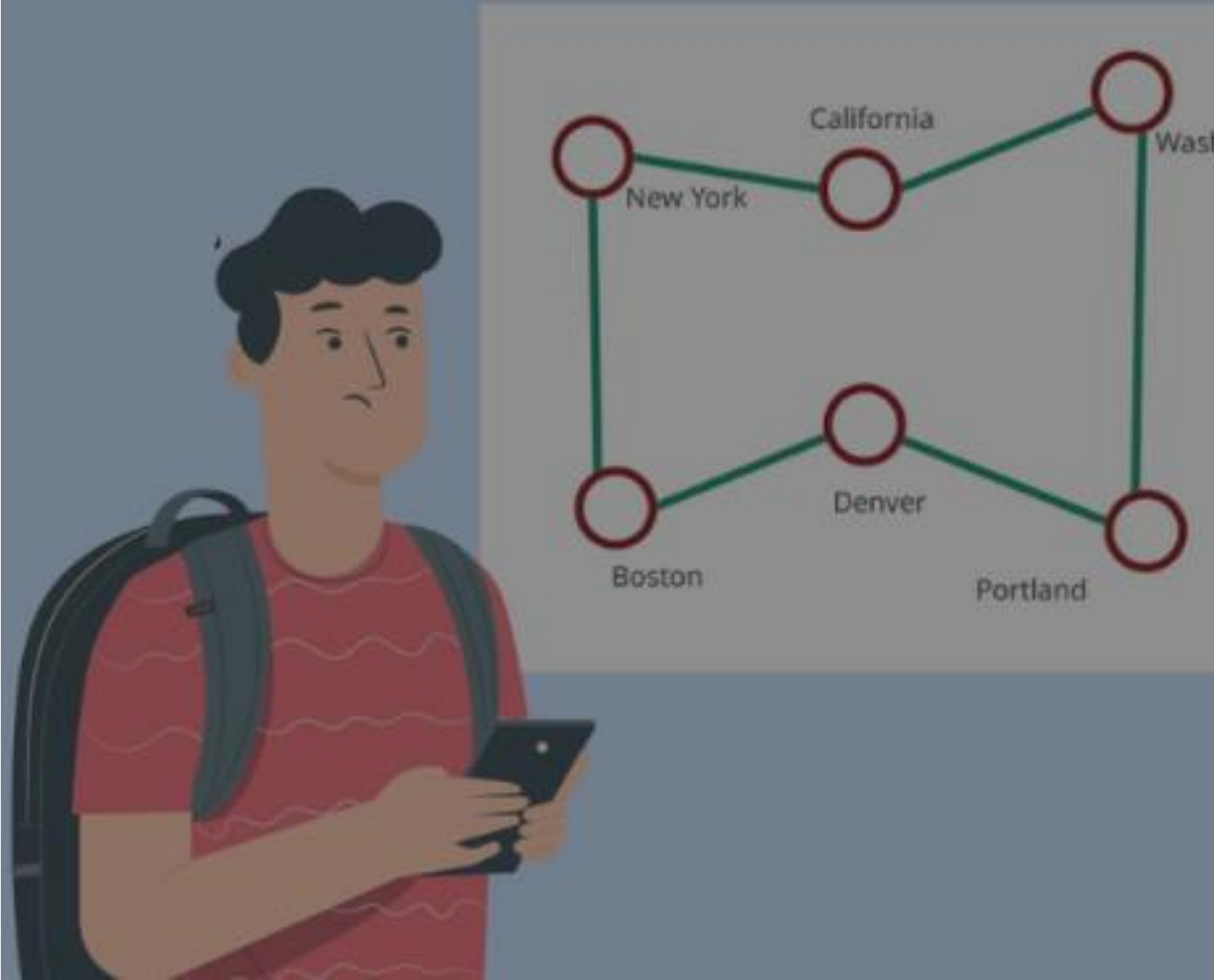
Ui



Backtracking

- O backtracking é responsável por explorar todas as permutações possíveis de vértices para encontrar o caminho mais curto que visita cada cidade exatamente uma vez e retorna à cidade inicial (Hamiltonian Path).
- Ele garante que todas as soluções viáveis sejam consideradas e que a solução ótima seja encontrada.
- O único problema deste algoritmo é que ele é bastante dispendioso em termos computacionais, o que leva a que para grafos pequenos seja uma boa solução, mas para grafos maiores não seja uma solução remotamente desejável.





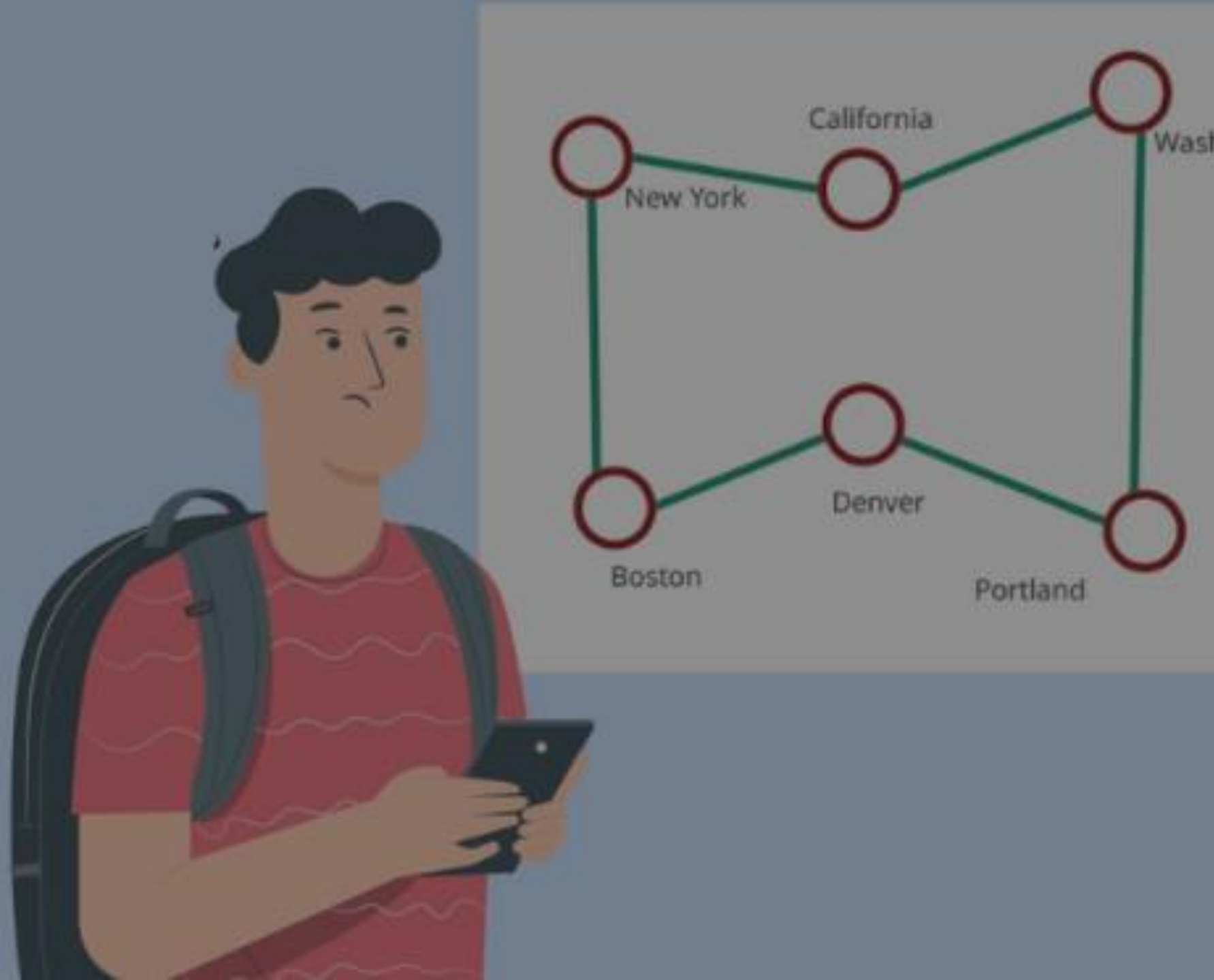
Triangular Aproximation

- Recorre ao algoritmo de Prim para construir uma MST (Minimum Spanning Tree), a partir do vértice fornecido como inicial, neste caso o vértice 0.
- Realiza uma travessia em pré-ordem na MST, gerando uma sequência de vértices que representa uma rota inicial aproximada.
- Procura uma aresta existente entre dois vértices. Se a aresta não existir, uma nova aresta é criada usando a distância calculada pela fórmula de Haversine.
- Este algoritmo garante a “2-approximation”, ou seja, o custo do caminho inteiro com esta heurística, nunca é mais de duas vezes superior ao caminho ótimo .

Nearest Neighbour

- O algoritmo Nearest Neighbour(NN) é um algoritmo greedy que funciona bem para obter uma solução rápida e aproximada para o TSP, mas não garante a solução ótima.
- A sua simplicidade e rapidez fazem dele uma boa escolha para problemas grandes onde soluções exatas são impraticáveis.
- A lógica central é sempre escolher o próximo vértice mais próximo, o “vizinho” ainda não visitado, o que pode não resultar no caminho mais curto, mas geralmente fornece uma solução razoavelmente boa.
- Segundo estatísticas comprovadas, em média atinge um resultado 25% superior ao caminho ótimo.





2-OPT

- O algoritmo 2-opt tenta otimizar um caminho anteriormente encontrado por outro algoritmo.
- O cerne do 2-opt consiste em remover 2 arestas e reconectar os dois caminhos resultantes de forma reversa, sempre que isso resulta em um caminho mais curto.
- O processo continua até que não haja mais melhorias possíveis ou até que o número de iterações máximas seja atingido, retornando assim o custo do caminho otimizado.
- O único problema oriundo deste algoritmo é que pode depender do quão “bom” é o caminho inicial.
- Este será posteriormente usado, para melhor o resultado de uma das heurísticas criadas por nós

Heurística Própria para RealWorldGraphs

(Variação de nearest neighbor com backtracking com ou sem 2-opt)

- O algoritmo desenvolvido resulta de uma junção de várias heurísticas e algoritmos que juntos resolvem o problema do tsp para grafos “grandes” em tempos praticáveis.
- O algoritmo começa por calcular uma MST para o grafo em questão, de seguida ordenas as arestas dos vertices adjacentes consoante o degree de cada destino de cada aresta, aumentando a capacidade de encontrar um caminho eficiente.
- De seguida, começa a partir do vértice inicial a procurar construir um caminho de forma recursiva, visitando os vertices, ainda não visitados, adicionando-os ao caminho e de seguida, escolhendo o próximo vértice através do “vizinho mais próximo”, ou seja, do vértice que se encontra a menor distância.
- Caso o caminho não leve a um Hamiltonian path, o algoritmo volta para trás (backtrack) e marcando o vértice como não visitado e removendo-o do caminho, tentando uma próxima opção.
- Assim que um potencial caminho é encontrado, o algoritmo verifica se os vertices foram colocados apenas uma vez e se o caminho retorna para o princípio.
- Por último, calcula o seu tamanho iterando por todas as edges
- O resultado ainda pode ser aprimorado recorrendo ao algoritmo 2-opt mencionado anteriormente

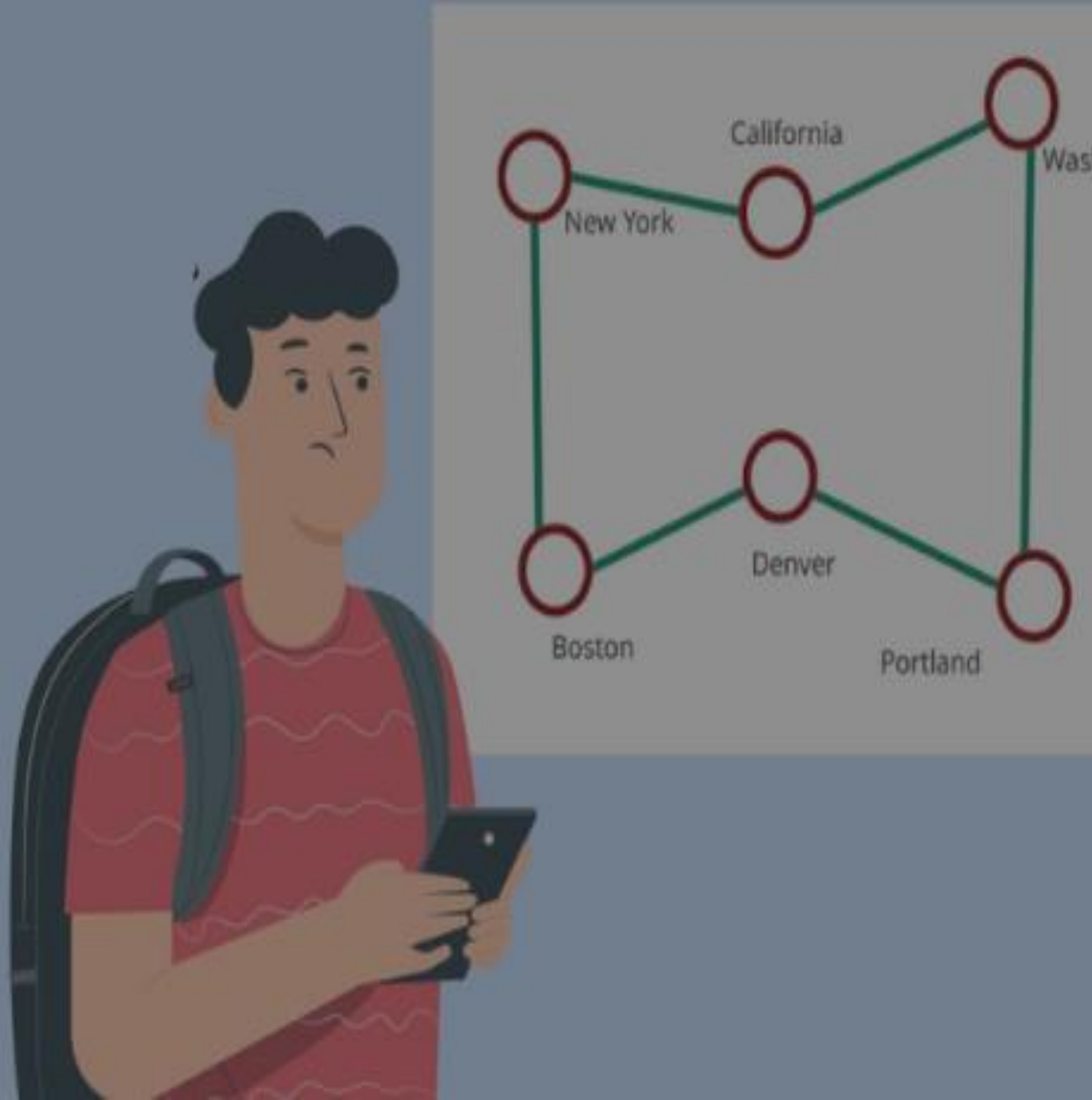


Heurística Própria para RealWorldGraphs

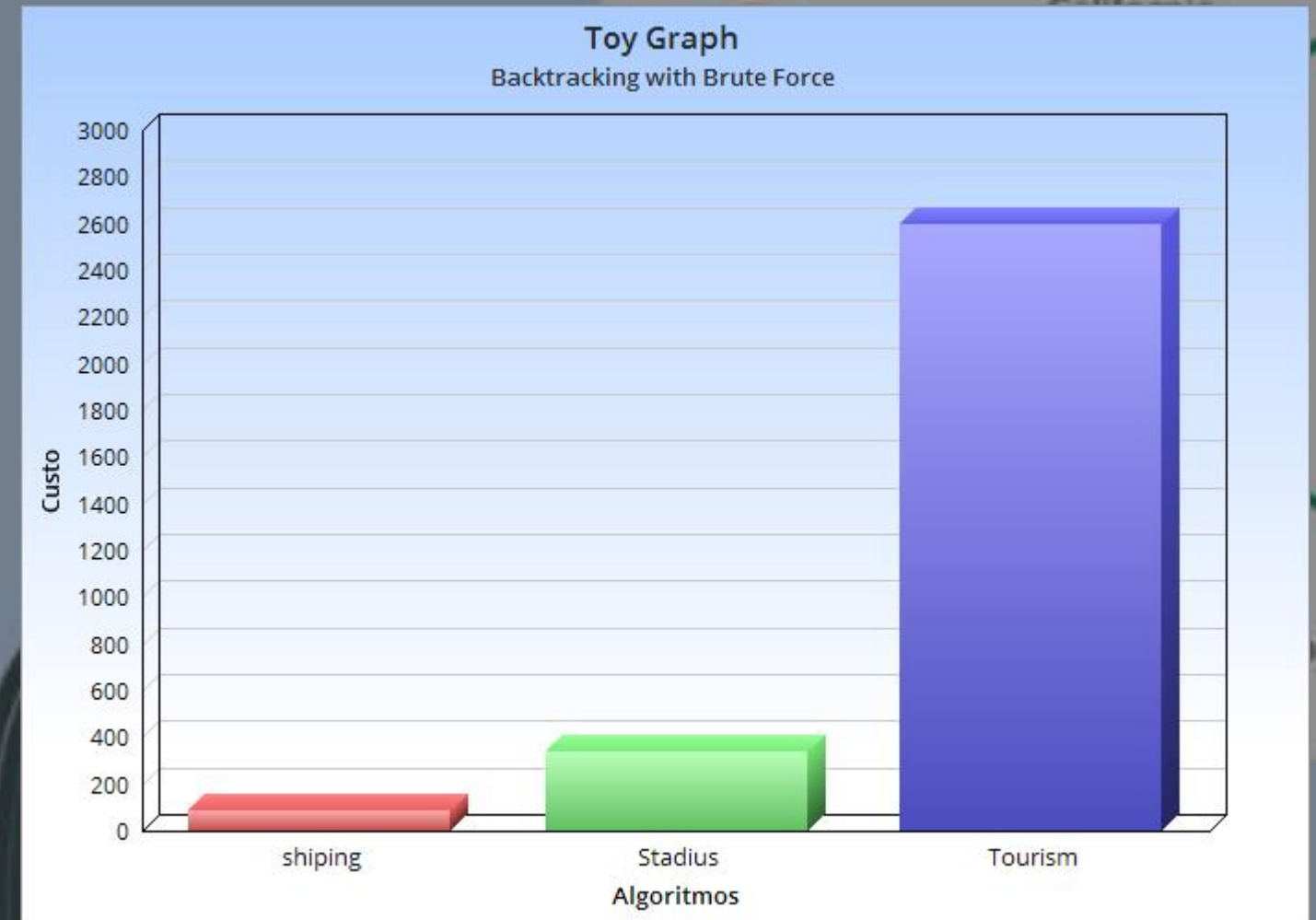
(Variação de nearest neighbor com backtracking com ou sem 2-opt)

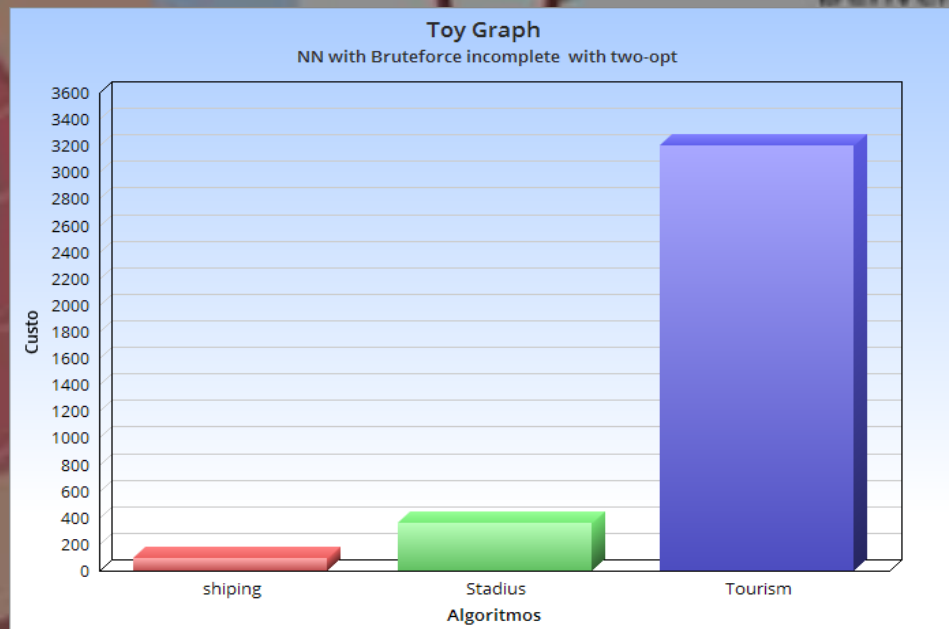
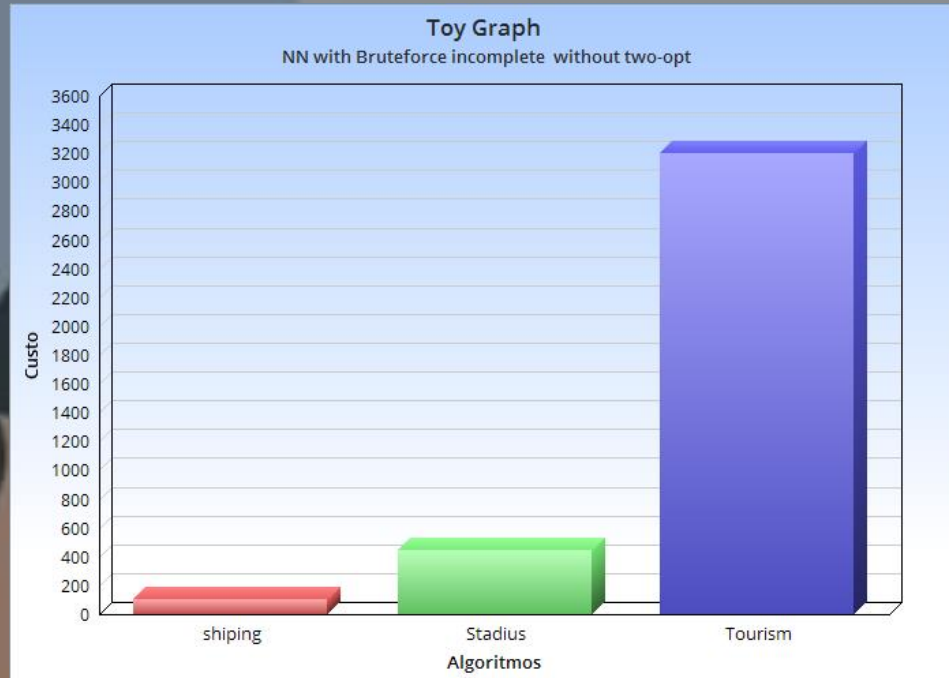
A ideia por detrás do algoritmo apresentado anteriormente resulta de uma abordagem probabilística, isto é:

- Os vértices com menor grau, estão conectados a um número menor número de vértices, logo quando visitados por último, a probabilidade de eles estarem ligados a um vértice ainda não visitado, é baixa.
- A intuição resulta da ideia de que ao visitar primeiro os vértices que podem fazer com que fique “preso”, isto é, vértices com menor grau, que por terem poucas conexões, não permitem sair da parte do grafo a ser visitada. Deixando assim os vértices mais conectados para o fim, pelo que a probabilidade de terem ligações para um dos vértices não visitado é muito maior.
- Deste modo, a probabilidade do algoritmo ficar “preso” e ter realizar $O(V!)$ backtracks diminui, algo que se denota no próprio algoritmo.



Backtracking nos toy graphs

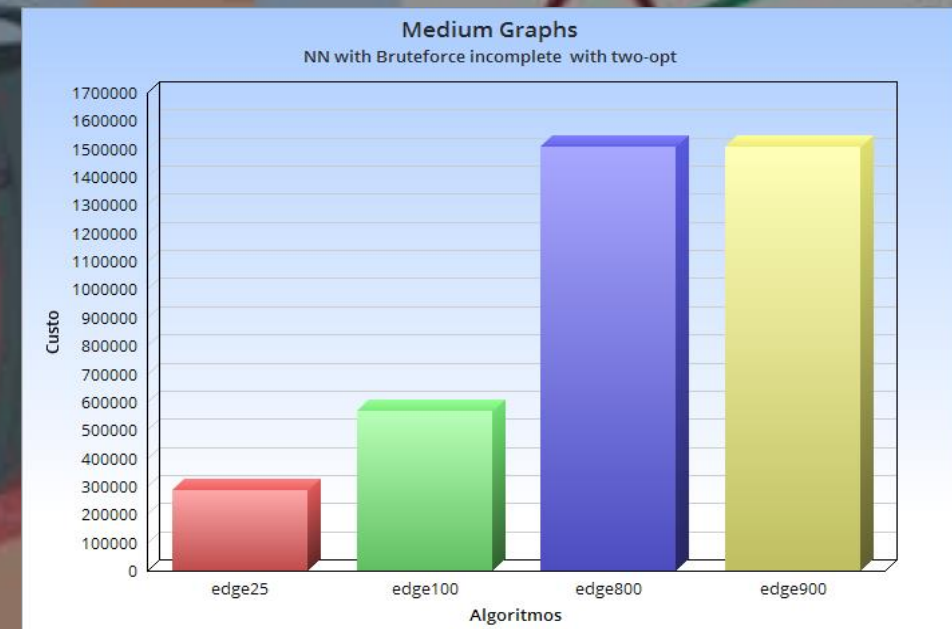
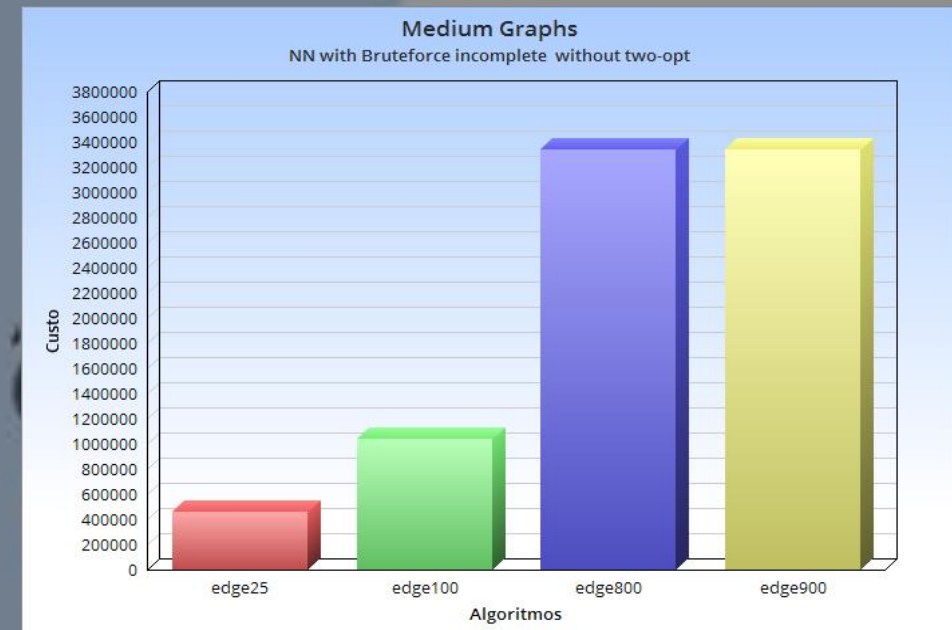


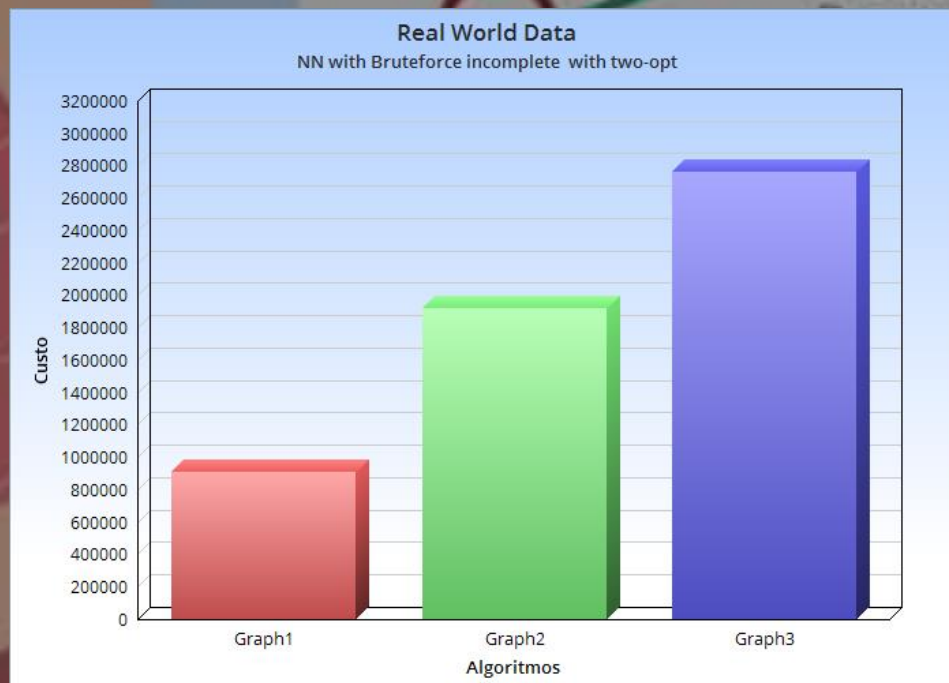
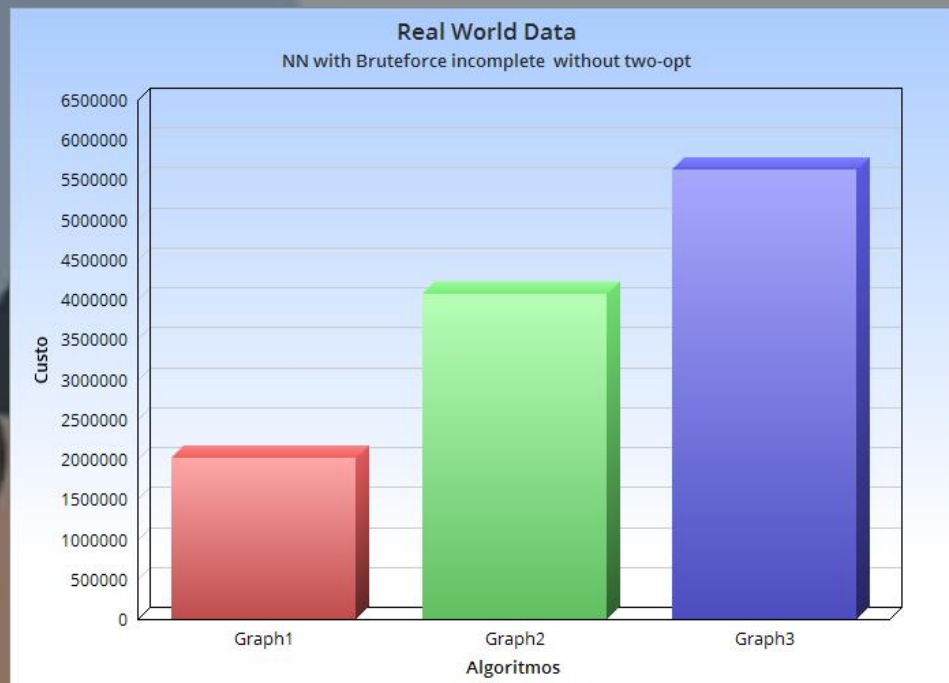


Heurística própria Toy Graphs com e sem two-opt



Heurística própria Medium Graphs com e sem two-opt





Heurística própria Real World Graphs com e sem two-opt

Destaque de Funcionalidades

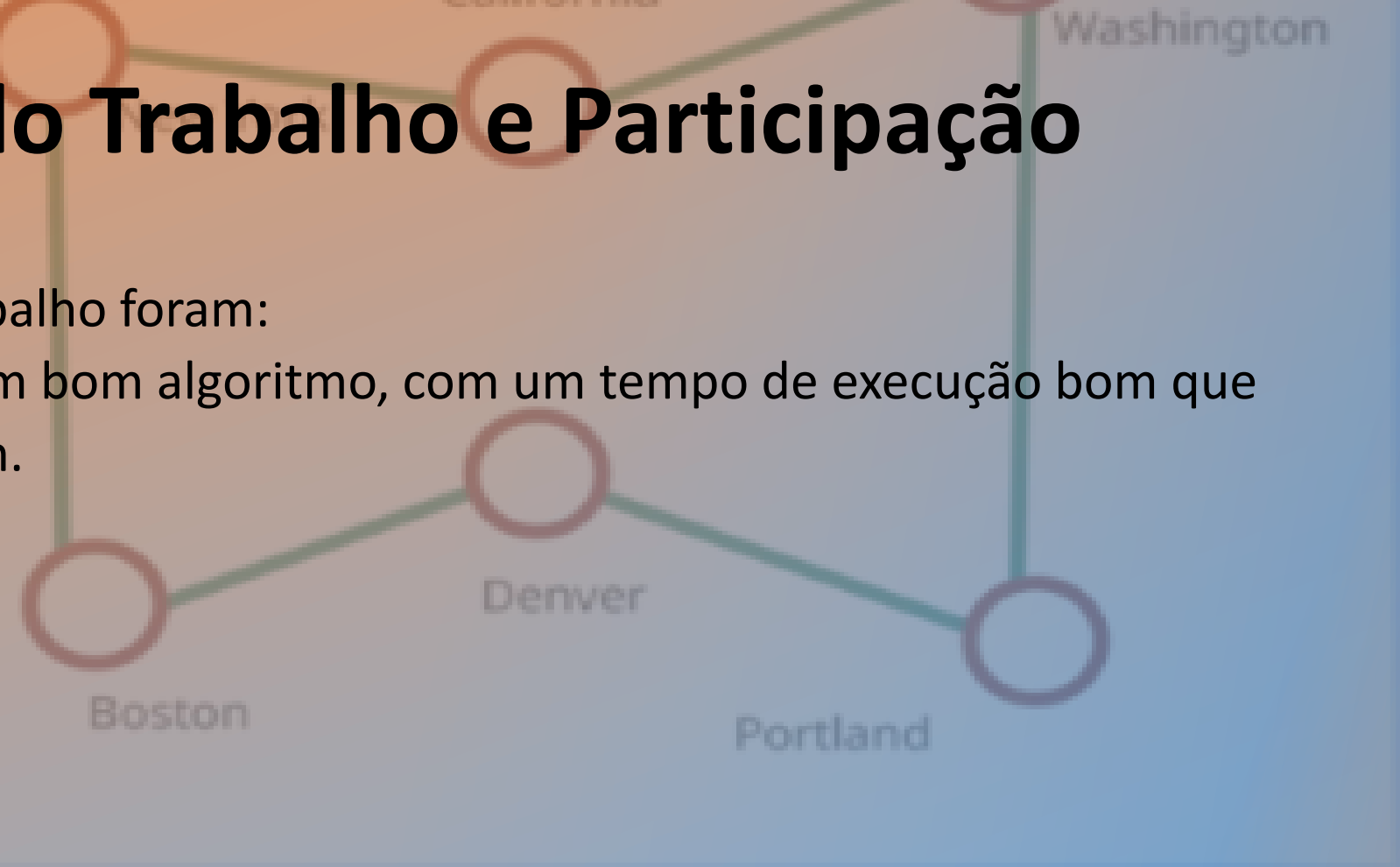
O Destaque neste Projeto foram:

- TSP in the Real World: Foram usados 3 algoritmos misturados que foram: Nearest Neighbour, Backtracking e TwoOpt, com estes algoritmos podemos inferir como a combinação inteligente de diferentes técnicas pode resultar em um algoritmo poderoso e eficiente, capaz de resolver problemas desafiadores como o TSP com alta eficácia.

Dificuldades do Trabalho e Participação

As principais dificuldades deste trabalho foram:

TSP in the Real World: Encontrar um bom algoritmo, com um tempo de execução bom que obtesse um resultado bastante bom.



up202207798 – Ângelo Oliveira – 33,3%

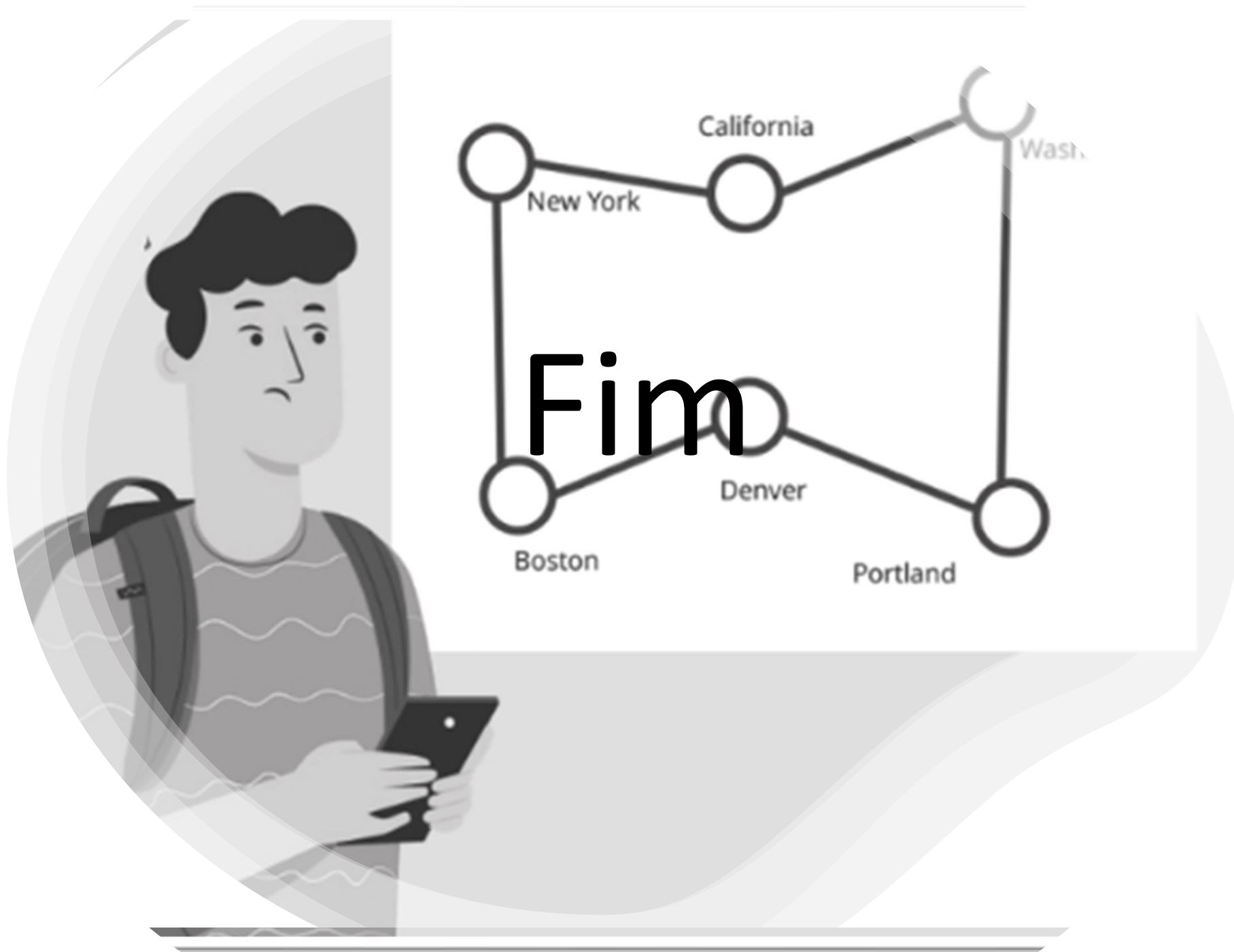
up202209730 – Bruno Fortes – 33,3%

up202207871 – José Costa – 33,3%

The background features a stylized illustration of a person with brown hair, wearing a red shirt and a grey backpack, looking down at a laptop. Overlaid on this is a network diagram with circular nodes and connecting lines. The nodes are labeled with city names: New York, California, Washington, Denver, Boston, and Portland. The diagram is rendered in a light, semi-transparent style.

Considerações finais

- Este projeto foi uma oportunidade valiosa para aplicar os algoritmos que aprendemos em sala de aula na resolução de problemas do mundo real. Foi uma prova concreta de que esses algoritmos não são apenas teoria acadêmica, mas ferramentas poderosas que têm aplicações práticas em diversos contextos. Pudemos também perceber como os algoritmos de otimização funcionam, e podem ser utilizados para resolver problemas reais.
- Essa experiência preparou-nos para enfrentar desafios semelhantes no futuro, seja na academia, em ambientes profissionais ou até mesmo em projetos pessoais. Além disso, essa vivência mostrou-nos a importância de compreender não apenas a teoria por trás dos algoritmos, mas também sua aplicação prática. Isso permite-nos não apenas resolver problemas de forma eficiente, mas também nos dá uma base sólida para propor soluções inovadoras e eficazes em diversos cenários.



Film