Problema do Caixeiro Viajante: Implementação de um algoritmo de otimização baseado nas técnicas de pesquisa operacionais para análise de complexidade

Fábio Costa
8200306@estg.ipp.pt

[a] Instituto Politécnico do Porto, Escola Superior de Tecnologia e Gestão, Rua do Curral, Casa do Curral–Margaride, 4610-156 Felgueiras, Portugal.

Resumo. Este trabalho trata o estudo e análise do Problema do Caixeiro Viajante - PCV de forma a poder implementar uma solução adequada para o mesmo. O PCV consiste em encontrar um trajeto fechado de custo mínimo que percorre todos os vértices de um grafo e finaliza no vértice inicial. Podemos com esta definição concluir que este problema trata um circuito Hamiltoniano cujo conceito tem várias aplicações na área da logística, como por exemplo, planeamento de transportes ou fabricação de placas de circuitos. Com estes conceitos em mente e através da análise e pesquisa efetuada no contexto do problema acabei por desenvolver um script na linguagem Java de forma a automatizar o processo de geração de trajetos ideais, ao mesmo tempo foi realizada uma série de testes.

Palavras-Chave: PCV; Script; Grafo; Vértice; Aresta; Hamiltoniano; Euleriano

1 Introdução

O Problema do Caixeiro Viajante – PCV (do inglês Traveling Salesman Problem - TSP) é um dos problemas mais estudados e conhecidos em ciências computacionais.
Este problema de dificuldade NP-Hard revela grande importância e aplicabilidade em muitas áreas como redução dos custos e planeamento de rotas na área de transportes e fabricação de placas de circuitos.

Problema do Caixeiro Viajante

Consiste em encontrar o ciclo hamiltoniano mais curto de um grafo onde os vértices representam clientes (cidades, localizações…) e as arestas representam a distância entre elas (ruas), ou seja, começando no primeiro cliente definir a rota de menor custo para percorrer todos os clientes e finalizar no cliente inicial.

Grafos

Um grafo é uma representação abstrata de um conjunto de objetos e das relações existentes entre eles. É definido por um conjunto de nós ou vértices, e pelos arcos ou arestas, que ligam pares de nós. O mesmo pode servir como representação se várias estruturas no mundo real.

Grafos Hamiltonianos

Um grafo Hamiltoniano requer que todos os vértices sejam percorridos pelo menos uma vez em contraste com um **Grafo Euleriano** que requer que todas as arestas sejam percorridas pelo menos uma vez.

2 Métodos e materiais

Método de resolução do problema do caixeiro-viajante

1. **Definir o problema:**
    - O Problema do Caixeiro Viajante consiste em encontrar o caminho mais curto que visita todos os vértices de um grafo exatamente uma vez e retornar ao vértice de origem.

2. **Gerar todas as permutações dos vértices:**
    - Iniciar a partir do vértice de origem;
    - Gerar todas as possíveis permutações dos vértices restantes excluindo o vértice de origem;
    - Isso pode ser feito usando técnicas como recursão ou algoritmos de geração de permutações;

3. **Calcular a distância total para cada permutação:**
    - Para cada combinação de vértices, calcula-se a distância total percorrida ao visitar os vértices na ordem de permutação;
    - A distância total é a soma das distâncias entre os vértices consecutivos.

4. **Encontrar o caminho mais curto:**
    - Compara-se as distâncias totais calculadas para cada combinação e encontra-se a menor distância;
    - Regista-se o caminho correspondente à menor distancia como o melhor caminho encontrado até ao momento.

5. **Retornar o resultado:**
    - Após percorrer todas as combinações, retorna-se o melhor caminho encontrado com a menor distância percorrida.

Algoritmo para a resolução do Problema do Caixeiro Viajante

Para resolver o problema do caixeiro-viajante usei um algoritmo em Java capaz de automatizar o processo de cálculo do caminho de menor custo. O algoritmo utilizado para resolver este problema é uma implementação do método de "Força Bruta", que explora todas as possíveis permutações dos vértices para encontrar a solução ótima. Embora seja um método simples, a sua complexidade é exponencial e pode ser impraticável para grafos grandes, isto acontece devido ao tempo de execução exponencial para casos com muitos vértices.

O algoritmo "Força Bruta" é usado de forma a obter a solução exata, contudo falha na utilização de grafos maiores, como já havia indicado anteriormente, contudo como neste caso irei trabalhar com grafos mais pequenos, uma vez que são mais fáceis de representar e de visualizar no relatório. No caso de ser preciso utilizar grafos maiores e com mais vértices, esta solução seria inviável, contudo a exatidão do algoritmo utilizado em alternativa não seria tão exata.

Este algoritmo tem algumas vantagens:
1. **Simplicidade -** O algoritmo é relativamente simples de implementar.
2. **Garantia de solução Ótima -** O algoritmo garante a solução ótima.
3. **Disponibilidade de recurso -** Em alguns casos, o número de vértices é pequeno o suficiente para que o algoritmo seja viável, mesmo com a sua complexidade.

Contudo também apresenta algumas desvantagens:

1. **Complexidade exponencial** - A complexidade deste algoritmo torna o seu tempo de execução impraticável em grafos maiores.
2. **Uso intensivo de recursos** - O algoritmo requer uma quantidade significativa de recursos computacionais.
3. **Limitação do tamanho do problema** - Devido á sua complexidade, o algoritmo é impraticável para problemas com um número grande de vértices.

Método de cálculo da ordem de complexidade

Para calcular a ordem de complexidade irei usar a notação "big O"

Bateria de testes

A Bateria de testes é uma coleção de grafos ou testes (listados no anexo 1) manualmente criados e resolvidos em papel de forma a obter matrizes adjacentes para podermos então testar com a implementação do algoritmo e descobrir a ordem de complexidade do mesmo obtendo resultados coerentes.
A Bateria é constituída por 3 testes, sendo o próximo mais complicado que o anterior tendo o primeiro, 4 vértices, o segundo 6 e o terceiro 8 vértices correspondentes a cidades.

Porque os grafos escolhidos são bons casos de teste:

1. **Grafo 1:**
    - **Estrutura Simples** - O grafo possui uma estrutura simples com apenas 4 vértices, o que facilita a sua visualização e compreensão.
    - Distâncias Equilibradas - As distâncias entre os vértices são equilibradas, o que significa que não há grandes diferenças entre elas,
    - Melhor Caminho Único - Existe apenas um caminho ótimo, ou seja, um caminho que visita todos os vértices uma só vez e retorna só vértice de origem.

2. **Grafo 2:**
    - **Estrutura Mais Complexa** - O grafo possui 6 vértices, o que é um pouco mais complexo do que o 1º caso de teste.
    - **Distâncias Variadas** - As distâncias entre os vértices são variadas, algumas são curtas e outras são mais longas, proporcionando um desafio adicionado na busca pelo caminho mais curto, por parte do algoritmo.
    - Diferentes Caminhos Ótimos - Existem diferentes caminhos ótimos que podem ser percorridos, levando diferentes solução válidas para o Problema do Caixeiro Viajante.

3. **Grafo 3:**
    - **Maior Número de Vértices** - O grafo possui 8 vértices, o que representa um cenário mais desafiador e próximo da complexidade prática.
    - **Distâncias Curtas** - As distâncias entre os vértices são curtas, o que pode levar a soluções mais rápidas.
    - **Ciclo Longo** - O caminho ótimo percorre um ciclo longo antes de retornar ao vértice de origem.

3 Resultados Experimentais

1º Grafo

Caminho Mais Curto: **0 – 1 – 3 – 2 – 0**
Custo: **80**

```
--Grafo1--
Melhor caminho: [0, 1, 3, 2, 0]
Menor distância: 80
Número de comparações: 36
```

Figura 1- Resultado Algoritmo Grafo 1

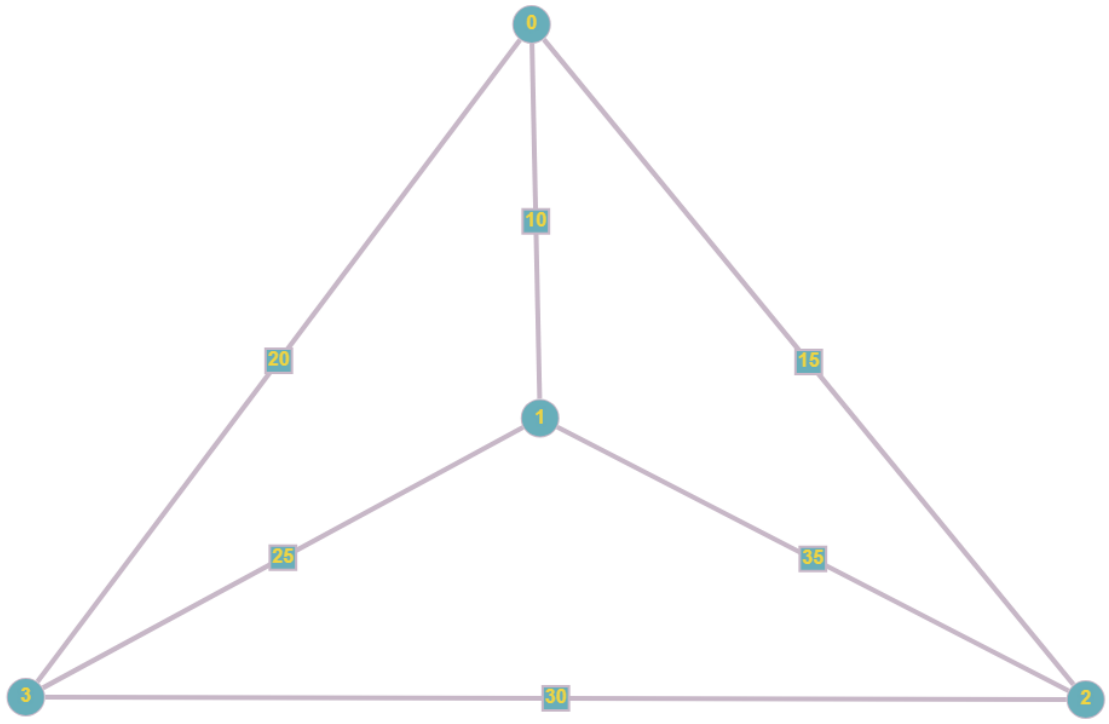

Figura 2- Imagem Grafo 1

2º Grafo

Caminho Mais Curto: **0 – 1 – 2 – 3 – 5 – 0**
Custo: 20

```
--Grafo2--
Melhor caminho: [0, 1, 2, 3, 4, 5, 0]
Menor distância: 20
Número de comparações: 1150
```

Figura 3- Resultado Algoritmo Grafo 2

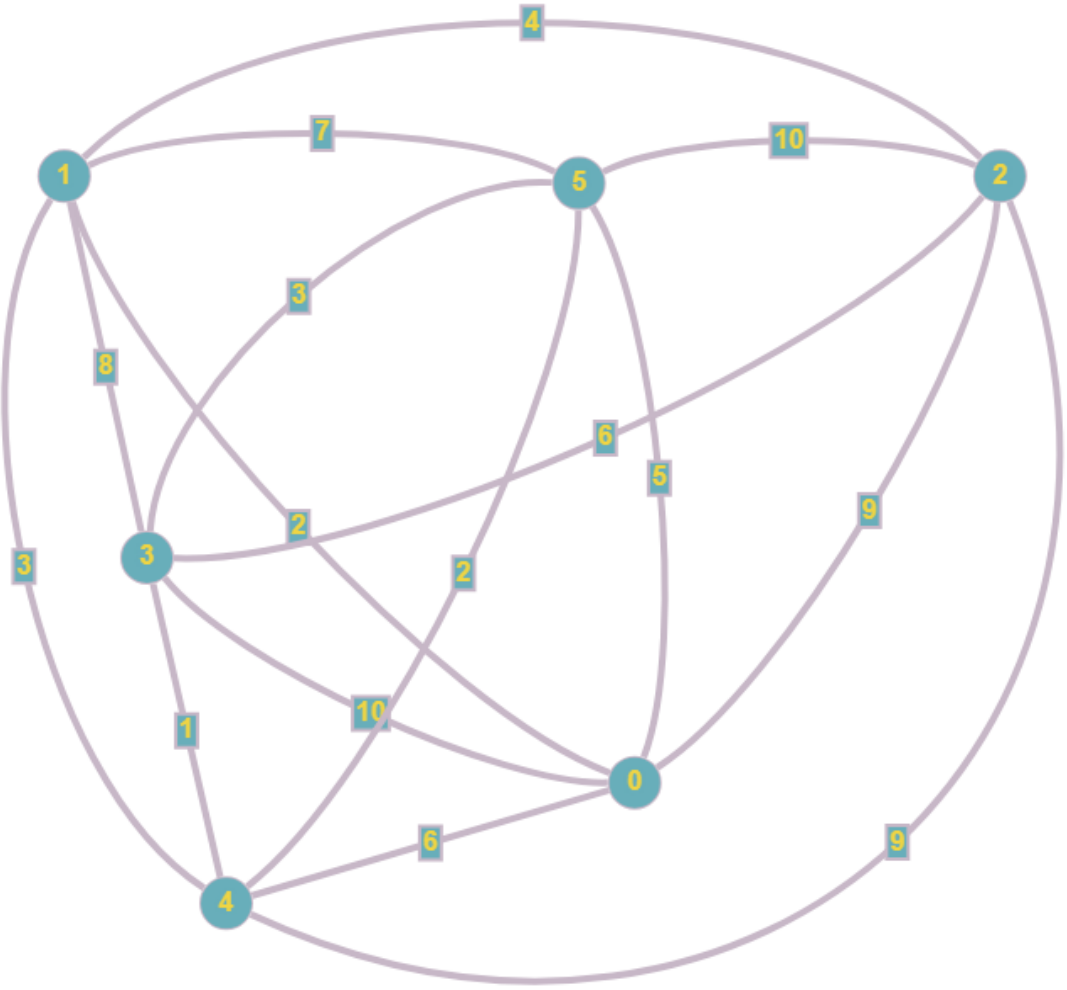

Figura 4 – Imagem Grafo 2

3º Grafo

Caminho Mais Curto: **0 – 3 – 2 – 5 – 4 – 1 – 7 – 6 - 0**
Custo: **15**

```
--Grafo3--
Melhor caminho: [0, 3, 2, 5, 4, 1, 7, 6, 0]
Menor distância: 15
Número de comparações: 65660
```

Figura 5 - Resultado Algoritmo Grafo 3

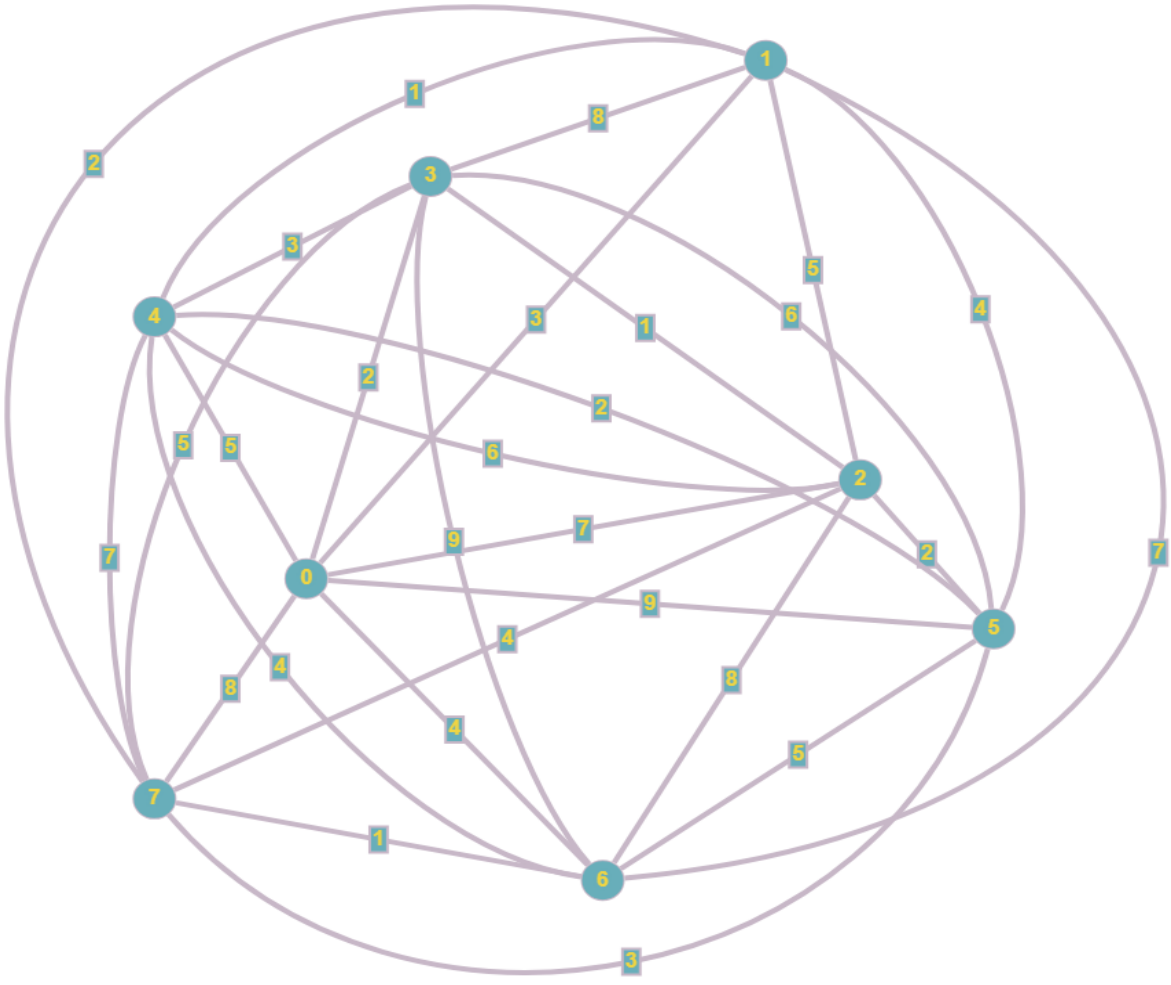

Figura 6 - Imagem Grafo 3

4 Discussão

O algoritmo utilizado para resolver o Problema do Caixeiro Viajante por meio de força bruta tem uma complexidade exponencial. Vamos analisar o desempenho em termos de tempo de execução e número de comparações.

Em relação ao tempo de execução, a complexidade do algoritmo é O(!n), onde n é o número de vértices. Isso ocorre porque o algoritmo considera todas as comparações possíveis dos vértices. Conforme o número de vértices aumenta, o tempo de execução cresce rapidamente, tornando o algoritmo impraticável para grafos grandes.

Por exemplo para um grafo com 10 vértices, o algoritmo teria que permutar 10! = 3.628.800 possibilidades. Essa complexidade exponencial limita a aplicabilidade algoritmo a problemas de pequeno porte.

É importante também destacar que a ordem de complexidade O(!n) representa o pior caso do algoritmo de "Força Bruta". Em alguns caos específicos, é possível otimizar e reduzir o número de combinações a serem validadas. No entanto, mesmo com otimizações, a ordem de complexidade continua a ser exponencial.

Pontos	Comparações	BigO
4	36	O(!n)
6	1150	
8	65660	

Tabela 1: Complexidade

5 Conclusão

Em resumo, o algoritmo de força bruta utilizado para resolver o Problema do Caixeiro viajante tem uma complexidade exponencial, o que significa que o seu desempenho deteriora rapidamente á medida que o número de vértices aumenta. Ele requer um grande número de comparações e, consequentemente, um tempo de execução impraticável para grafos grandes. Para lidar com grafos maiores, são utilizadas técnicas mais eficientes, como algoritmos heurísticos (ex.: busca em vizinhança, algoritmos do vizinho mais próximo), ou então algoritmos de programação dinâmica (ex.: algoritmo de Held-Karp), que fornece soluções aproximadas com complexidades mais razoáveis.

Entretanto foi-nos possível aplicar o método científico para a analise do Problema do Caixeiro Viajante - PCV e ainda analisar a ordem de complexidade de um algoritmo que implementamos numa linguagem de programação para resolver tal problema, ajudando-nos assim a compreender vários conceitos nas áreas de teoria de grafos e algoritmia que nos acompanharão para o futuro das nossas carreiras.

Referências

1. T.H Cormen, C. E. Leiserson, R. L. Rivest, and C.Stein. “Introduction to Algorithms” (Capítulo 35: “Traveling Selesman Problem”). The MIT Press, 2009.

2. Feofiloff, P., Kohayakawa, Y., & Wakabayashi, Y. (2011). Uma introdução sucinta à teoria dos grafos.

3. https://chat.openai.com/ (como auxílio no desenvolvimento do script do nosso algoritmo.

Anexos

Anexo 1 - Conjunto de Testes

```java
public static void main(String[] args) {
    int[][] grafo1 = {
            { 0, 10, 15, 20 },
            { 10, 0, 35, 25 },
            { 15, 35, 0, 30 },
            { 20, 25, 30, 0 }
    };

    CaixeiroViajante caixeiro1 = new CaixeiroViajante(grafo1);
    System.out.println(x:"\n--Grafo1--");
    caixeiro1.encontrarMenorCaminho();


    int[][] grafo2 = {
            { 0, 2, 9, 10, 6, 5 },
            { 2, 0, 4, 8, 3, 7 },
            { 9, 4, 0, 6, 9, 10 },
            { 10, 8, 6, 0, 1, 3 },
            { 6, 3, 9, 1, 0, 2 },
            { 5, 7, 10, 3, 2, 0 }
    };

    CaixeiroViajante caixeiro2 = new CaixeiroViajante(grafo2);
    System.out.println(x:"\n--Grafo2--");
    caixeiro2.encontrarMenorCaminho();


    int[][] grafo3 = {
            { 0, 3, 7, 2, 5, 9, 4, 8 },
            { 3, 0, 5, 8, 1, 4, 7, 2 },
            { 7, 5, 0, 1, 6, 2, 8, 4 },
            { 2, 8, 1, 0, 3, 6, 9, 5 },
            { 5, 1, 6, 3, 0, 2, 4, 7 },
            { 9, 4, 2, 6, 2, 0, 5, 3 },
            { 4, 7, 8, 9, 4, 5, 0, 1 },
            { 8, 2, 4, 5, 7, 3, 1, 0 }
    };

    CaixeiroViajante caixeiro3 = new CaixeiroViajante(grafo3);
    System.out.println(x:"\n--Grafo3--");
    caixeiro3.encontrarMenorCaminho();
    System.out.println(x:"");

}
```

Figura 7 - Conjunto de Grafos Teste

Anexo 2 – Script do algoritmo de "Força Bruta" ("CaxeiroViajante.java")

```java
public class CaixeiroViajante {
    private int[][] grafo;
    private int numVertices;
    private List<Integer> melhorCaminho;
    private int menorDistancia;
    private int numComparacoes;

    public CaixeiroViajante(int[][] grafo) {
        this.grafo = grafo;
        this.numVertices = grafo.length;
        this.melhorCaminho = new ArrayList<>();
        this.menorDistancia = Integer.MAX_VALUE;
        this.numComparacoes = 0;
    }

    public void encontrarMenorCaminho() {
        List<Integer> caminhoAtual = new ArrayList<>();
        caminhoAtual.add(e:0); // Começar no vértice 0
        boolean[] visitado = new boolean[numVertices];
        visitado[0] = true;

        encontrarMenorCaminhoAux(caminhoAtual, visitado, distanciaAtual:0, nivel:1);

        // Adicionar o vértice de origem novamente para fechar o ciclo
        melhorCaminho.add(e:0);

        System.out.println("Melhor caminho: " + melhorCaminho);
        System.out.println("Menor distância: " + menorDistancia);
        System.out.println("Número de comparações: " + numComparacoes);
    }

    private void encontrarMenorCaminhoAux(List<Integer> caminhoAtual, boolean[] visitado, int distanciaAtual,
            int nivel) {
        if (nivel == numVertices) {
            int distanciaTotal = distanciaAtual + grafo[caminhoAtual.get(nivel - 1)][0];
            if (distanciaTotal < menorDistancia) {
                menorDistancia = distanciaTotal;
                melhorCaminho = new ArrayList<>(caminhoAtual);
            }
            numComparacoes++;
            return;
        }

        for (int i = 1; i < numVertices; i++) {
            if (!visitado[i]) {
                caminhoAtual.add(i);
                visitado[i] = true;
                int novaDistancia = distanciaAtual + grafo[caminhoAtual.get(nivel - 1)][i];

                encontrarMenorCaminhoAux(caminhoAtual, visitado, novaDistancia, nivel + 1);

                caminhoAtual.remove(caminhoAtual.size() - 1);
                visitado[i] = false;
            }
            numComparacoes++;
        }
    }
```

Figura 8 - Algoritmo Java Força Bruta

Anexo 3 – Link GitHub com Algoritmo, imagens e relatorio

https://github.com/FabioCosta0011/AlgoritmoAAO