Geração De Polígonos Simples

Inteligência Artificial 2021/2022
Relatório Trabalho 1

André Cerqueira - 201804991
Sara Sá - 201804838
6 de abril de 2021

Conteúdo

Introdução

Este trabalho está inserido no problema do caixeiro viajante(TSP,"Traveling Salesman Problem") que faz a seguinte pergunta:"Dada uma lista de cidades e as distâncias entre cada par de cidades, qual é a rota mais curta possível que visita cada cidade exatamente uma vez e retorna à cidade de origem?".

O problema foi formulado pela primeira vez em 1930 e é um dos problemas de otimização mais intensamente estudados, sendo usado como referência para muitos problemas. Possui diversas aplicações até mesmo na sua formulação mais pura,como planeamento,logística e fabricação de microchips.

O TSP pode ser representado como um grafo pesado e não dirigido e completo, de forma a que as cidades sejam os vértices do grafo, os caminhos as arestas e a distância do caminho seja o peso da aresta. É um problema de minimização que começa e termina num vértice especifico após ter visitado todos os nós exatamente uma vez.

De forma a encontrar candidatos a soluções através de pesquisa local, podemos escolher entre 2 tipos de métodos: métodos construtivos ou perturbativos.

Os métodos perturbativos geram novos candidatos na vizinhança do candidato completo inicial, podendo esta ser definida, por exemplo,pela heurística "2-exchange"que tem como objetivo remover as duas arestas que se intersetem no interior do polígono.

Já os métodos construtivos constroem os candidatos a soluções, passo a passo a partir de uma estratégia , como por exemplo a nearest neighbour first, que consiste em inserir no percurso o nó ainda não visitado, e que esteja mais próximo do último visitado, sendo que o nó inicial pode ser escolhido aleatoriamente (baseado numa estratégia Greedy).

Nos anos 1990s, Thomas Auer e Martin Held propuseram um gerador de polígonos aleatórios (RPG) que utiliza como base ideias semelhantes ao problema do TSP e que tem por objetivo criar um polígono simples(não tem arestas que se intersetem a não ser em vértices) a partir de um conjunto de n pontos no plano, ou seja, os seus vértices.

Conseguimos implementar os 5 primeiros exercícios apesar de termos alguns erros.

Estruturação do Código

O código , está divido em 3 ficheiros

- **Main.java** → neste ficheiro encontra-se a *main* , a declaração da classe *Graph* que representa o polígono e o menu com as opções a selecionar.
- **search.java** → Neste ficheiro encontram-se implementados os algoritmos de pesquisa. como o *Hill Climbing* e o *Nearest Neighbour First*.
- **gui.java** → implementação da parte gráfica que é chamada quando a opção imprimir gráfico é selecionada .

Implementação dos exercícios propostos

1) Gerar aleatoriamente n pontos no plano

Neste exercício é necessário implementar um algoritmo que gera aleatoriamente n pontos no plano x, y com coordenadas inteiras de $[-m, m]$ para valores de n e m dados.

De forma a resolver o exercício, criamos uma classe Graph que vai representar o polígono, usando como estrutura para guardar os nós do grafo de forma ordenada ,uma List do java. Para guardar os pontos usamos a classe Point2D do java que tal como o nome indica representa o (x, y) num espaço de coordenadas de um referencial. Criamos um método da classe Graph que gera pontos de forma aleatória (randomPointGenerator) a partir da classe *Random* do java e também um método para adicionar pontos ao grafo em si.

```
1  import java.util.ArrayList;
2  import java.util.List;
3  import java.util.Random;
4  import java.awt.geom.Point2D;
5
6  class Graph {
7      List<Point2D> nodes; // list of nodes (order sorts)
8      List<Graph> sons; // list of possible sons
9
10     Graph() {
11         this.nodes = new ArrayList<Point2D>();
12         this.sons = new ArrayList<Graph>();
13     }
14
15     boolean addNewPoint(double x, double y) {
16         Point2D tmp = new Point2D.Double(x, y);
```

```
17
18          if (nodes.contains(tmp))
19          return false;
20
21          nodes.add(tmp);
22          return true;
23      }
24
25      boolean randomPointGenerator(int range) {
26          Random random = new Random();
27          // (upperbound-lowerbound) + lowerbound;
28          if (addNewPoint((double) random.nextInt(range * 2) -
    range, (double) random.nextInt(range * 2) - range))
29              return true;
30          else
31              return false;
32      }
33  }
```

2a) Gerar uma permutação qualquer dos pontos

Para resolver este exercício,implementamos um método de Java que através da classe de java *Collections*permite baralhar os elementos da lista de nós.

```
1  import java.util.Collections;
2
3  class Graph {
4      void randomPermutation() {
5          Collections.shuffle(this.nodes);
6      }
7  }
```

2b) Aplicar a heurística *nearest-neighbour-first*

O nearest-neighbour-first foi um dos primeiros algoritmos usados para resolver aproximadamente o problema do caixeiro viajante. Neste problema o vendedor começa numa cidade aleatória e visita repetidamente a cidade mais próxima da atual até que todas tenham sido visitadas. O algoritmo produz rapidamente uma solução, mas geralmente esta não é a ideal. Este algoritmo pode não encontrar um passeio viável,mesmo quando existe um.

Na implementação deste algoritmo seguimos as seguintes etapas:

1. Inicializamos todos os vértices como não visitados.
2. Selecionamos o primeiro vértice da lista como sendo inicial.

3. Adicionamos o cur à lista de visitados e calculamos a distancia para todos os outros nós ainda não visitados, escolhendo assim como ligação o que tem a menor distancia.

4. Fazemos o passo 3 enquanto houver nós não visitados.

A ordem pelo qual os pontos são visitados é a solução do algoritmo.

```
public static void nearestNeighbourFirst(Graph g) {
    List<Point2D> visited = new ArrayList<Point2D>(); // list
    of visited nodes (visited)
    List<Point2D> path = new ArrayList<Point2D>(); // list of
    visited nodes (visited)

    boolean goalFound = false;
    int index = 0;
    double minDistance;

    while (!goalFound) {
        minDistance = Double.MAX_VALUE;
        Point2D cur = g.nodes.get(index);
        path.add(cur);
        visited.add(cur);

        if (visited.size() == g.nodes.size()) {
            goalFound = true;
            continue;
        }

        for (Point2D K : g.nodes) {
            if (!visited.contains(K)) {
                if (cur.distanceSq(K) < minDistance) {
                    index = g.nodes.indexOf(K);
                    minDistance = cur.distanceSq(K);
                }
            }
        }
    }
    g.nodes = path;
}
```

3) Determinar a vizinhança por Two-exchange a partir de um candidato

Para este exercício era necessário através do método Two-Exchange , encontrar as interseções e assim gerar os possíveis filhos do Polígono atual.

De forma a implementar este algoritmo, seguimos as seguintes etapas:

1. Para todos os pares de vetores possíveis verificamos se estes se intersetam e/ou são colineares através da função *segmentsIntersect*.

2. No caso de uma interseção entre dois vetores ainda não ter sido adicionada,adicionamos à lista de interseções.

Para a implementação da função *segmentsIntersect* , utilizamos o algoritmo fornecido no enunciado do projeto:

- `https://www.inf.ed.ac.uk/teaching/courses/ads/Lects/lecture1516.pdf`
- `https://www.youtube.com/watch?v=R08OY6yDNy0`

No final,retornamos uma lista com as interseções existentes no polígono.

```
/*
* Calculates the dot product
*/
private static double dotProduct(Point2D p1, Point2D p2,
    Point2D p3, Point2D p4) {
    Point2D x = new Point2D.Double(p2.getX() - p1.getX(), p2.
    getY() - p1.getY());
    Point2D y = new Point2D.Double(p4.getX() - p3.getX(), p4.
    getY() - p3.getY());
    return (x.getX() * y.getX()) + (x.getY() * y.getY());
}

/*
* Check if a , lies on BC
*/
static boolean inBox(Point2D a, Point2D b, Point2D c) {
    if (b.getX() <= Math.max(a.getX(), c.getX()) && b.getX() >=
     Math.min(a.getX(), c.getX())
    && b.getY() <= Math.max(a.getY(), c.getY()) && b.getY() >=
    Math.min(a.getY(), c.getY()))
    return true;

    return false;
}

/*
* To find orientation Returns orientation, 0) colinear , 1)
    Clockwise 2)
* CounterClockwise
*/
static int dir(Point2D a, Point2D b, Point2D c) {
    double val = (b.getY() - a.getY()) * (c.getX() - b.getX())
    - (b.getX() - a.getX()) * (c.getY() - b.getY());
```

```
27
28      if (val == 0.0)
29      return 0;
30
31      return (val > 0) ? 1 : 2;
32  }
33
34  /*
35  * Checks if intersects https://www.youtube.com/watch?v=
        R08OY6yDNy0
36  */
37  private static boolean segmentsIntersect(Point2D p1, Point2D p2
        , Point2D p3, Point2D p4) {
38      int dir1 = dir(p1, p2, p3);
39      int dir2 = dir(p1, p2, p4);
40      int dir3 = dir(p3, p4, p1);
41      int dir4 = dir(p3, p4, p2);
42
43      if (dir1 != dir2 && dir3 != dir4)
44      return true;
45      else if ((dir1 == 0 && dir2 == 0 && dir3 == 0 && dir4 == 0)
         && dotProduct(p1, p3, p3, p4) > 0)
46      return true;
47      else if (dir1 == 0 && inBox(p1, p3, p2))
48      return true;
49      else if (dir2 == 0 && inBox(p1, p4, p2))
50      return true;
51      else if (dir3 == 0 && inBox(p3, p1, p4))
52      return true;
53      else if (dir4 == 0 && inBox(p3, p2, p4))
54      return true;
55
56      return false;
57  }
58
59  /*
60  * Checks if intersection isn't already on intersected List
61  */
62  private static boolean intersectionAlreadyExists(List<Point2D>
        list, Point2D a, Point2D b, Point2D c, Point2D d) {
63      if (list.size() > 0 && list.size() % 4 == 0) {
64          for (int i = 3; i < list.size(); i += 4) {
65
66              Point2D q = list.get(i - 3);
67              Point2D w = list.get(i - 2);
68              Point2D e = list.get(i - 1);
69              Point2D r = list.get(i);
70
71              if (q == a && w == b && e == c && r == d) {
```

```
                return true;
            }

        }
    }
    return false;
}

/*
* 2-Exchange
*/
public static List<Point2D> twoExchange(Graph g) {
    List<Point2D> intersections = new ArrayList<Point2D>(); //
    list of nodes that intersect

    Point2D a, b, c, d;

    for (int i = 1; i < g.nodes.size(); i++) {
        a = g.nodes.get(i - 1);
        b = g.nodes.get(i);

        for (int j = 1; j < g.nodes.size(); j++) {
            c = g.nodes.get(j - 1);
            d = g.nodes.get(j);

            if (a != c && a != d && b != c && b != d) {
                if (segmentsIntersect(a, b, c, d)) {
                    if (!intersectionAlreadyExists(
    intersections, c, d, a, b)) {
                        intersections.add(a);
                        intersections.add(b);
                        intersections.add(c);
                        intersections.add(d);
                    }
                }
            }
        }
    }

    // test last connection

    a = g.nodes.get(0);
    b = g.nodes.get(g.nodes.size() - 1);

    for (int i = 2; i < g.nodes.size() - 1; i++) {
        c = g.nodes.get(i - 1);
        d = g.nodes.get(i);

        if (segmentsIntersect(a, b, c, d)) {
```

```
119             intersections.add(a);
120             intersections.add(b);
121             intersections.add(c);
122             intersections.add(d);
123         }
124 
125     }
126 
127     return intersections;
128 }
```

4) Aplicar *Hill Climbing* através do *Two-exchange*

O melhoramento iterativo(hill climbing) é uma técnica de otimização matemática pertencente à família da pesquisa local.

É um algoritmo iterativo que começa com uma solução arbitrária para um problema e,em seguida, tenta encontrar uma solução melhor através de uma mudança incremental na solução.

Se esta mudança produzir uma solução melhor, outra mudança incremental será feita na nova solução e assim por diante até que nenhuma possa ser encontrada.

A relativa simplicidade do algoritmo torna-o uma primeira escolha entre algoritmos de otimização sendo que muitas vezes pode produzir um resultado melhor do que outros algoritmos quando a quantidade de tempo disponível para realizar a pesquisa é limitada. Pode ainda retornar uma solução válida mesmo se for interrompido a qualquer momento antes de terminar.

Para a implementação do *Hill Climbing* em si , implementamos um ciclo *while* que continua enquanto houver filhos que possa optar (que ainda não tenham sido visitados), através da escolha fornecida pelo utilizador (*options*).

```
1  /*
2  * Generate the candidates
3  */
4  private static void generateSons(Graph g) {
5      // list of intersections
6      List<Point2D> intersection = twoExchange(g);
7  
8      // Create possible sons
9      while (intersection.size() > 0) {
10         Point2D c = intersection.remove(3);
11         c = intersection.remove(2);
12         Point2D b = intersection.remove(1);
13         b = intersection.remove(0);
14 
```

```
15          Graph tmp = new Graph(g);
16
17          swap(tmp, b, c);
18          g.sons.add(tmp);
19      }
20  }
21
22  /*
23  * Implements hill Climbing
24  */
25  public static Graph hillClimbing(Graph cur, int option) {
26      List<Graph> visited = new ArrayList<Graph>(); // list of
        visited permutations
27      boolean existNext = false; // check if hill climb can
        continue
28      int indexNext = -1; // index of son
29      int count = 0;
30
31      do {
32          visited.add(cur);
33          generateSons(cur);
34
35          if (option == 1) {
36              indexNext = selectSonPerimeter(cur,visited);
37          } else if (option == 2) {
38              indexNext = selectFirstSon(cur,visited);
39          } else if (option == 3) {
40              indexNext = selectSonLessConflits(cur,visited);
41          } else if (option == 4) {
42              indexNext = selectSonRandom(cur,visited);
43          }
44
45          if (indexNext >= 0) {
46              cur = cur.sons.get(indexNext);
47              existNext = true;
48              indexNext = -1;
49              count++;
50          } else {
51              existNext = false;
52          }
53      } while (existNext == true);
54
55      System.out.println("-------- Acabei ---------");
56      System.out.println("Ao fim de " + count + " iteracoes");
57      return cur;
58  }
```

4a) Optar pelo candidato com o menor perímetro

Neste caso, é necessário escolher o próximo filho que terá o menor perímetro. Assim, implementamos a função *selectSonPerimeter* que vai percorrer todos os possíveis filhos e seleccionar como sendo o próximo, o que tem o menor perímetro.

Para calcular o perímetro do Polígono atual , criamos um método na classe *Graph* que percorre os Pontos todos fazendo a soma e retornando assim o perímetro.

```
class Graph {
    /*
    * Calculates the perimeter of the graph
    */
    double perimeter() {
        double answer = 0;
        Point2D a,b;
        for (int i = 1; i < this.nodes.size(); i++) {
            a = this.nodes.get(i - 1);
            b = this.nodes.get(i);
            answer += a.distanceSq(b);
        }
        a = this.nodes.get(this.nodes.size() - 1);
        b = this.nodes.get(0);
        answer += a.distanceSq(b);

        return answer;
    }
    //...
}

/*
****************
* search.java
****************
* Returns index of son with less perimeter
*/
private static int selectSonPerimeter(Graph g,List<Graph>
    visited) {
    // Calculates perimeter
    double minPerimeter = Integer.MAX_VALUE;
    double curPerimeter = 0;
    int indexNext = -1;
    for (Graph a : g.sons) {
        if (!visited.contains(a)) {
            curPerimeter = a.perimeter();
            if (curPerimeter < minPerimeter) {
                minPerimeter = curPerimeter;
```

```
                indexNext = g.sons.indexOf(a);
            }
        }
    }
    if (indexNext == -1) {
        return -1;
    } else {
        return indexNext;
    }
}
```

4b) Optar pelo primeiro candidato

Basicamente verificamos se existe algum filho que possamos optar e se sim, escolhemos o primeiro.

```
/*
* Returns index of first son (only if available)
*/
private static int selectFirstSon(Graph g,List<Graph> visited)
    {
    if (g.sons.size() > 0 && !visited.contains(g.sons.get(0)))
        return 0;
    else
        return -1;
}
```

4c) Optar pelo candidato com menor conflito de arestas

Para esta alínea , chamamos o *Two-exchange* para todos os candidatos (os nós filhos), e retornamos o índex do que tiver o menor número de interseções.

```
/*
* Returns number of conflicts
*/
private static int numberOfConficts(Graph g){
    List<Point2D> tmp = twoExchange(g); // list of
    intersections of the possible son
    return tmp.size();
}

/*
* Returns index of son with less conflicts (by two exchange)
*/
private static int selectSonLessConflits(Graph g,List<Graph>
    visited) {
```

```
    int minConflit = Integer.MAX_VALUE;
    int curConflict = 0;
    int indexNext = -1;
    for (Graph a : g.sons) {
        if (!visited.contains(a)) {
            curConflict = numberOfConficts(a);
            if (curConflict < minConflit) {
                minConflit = curConflict;
                indexNext = g.sons.indexOf(a);
            }
        }
    }
    return indexNext;
}
```

4d) Optar por um candidato qualquer

Nesta alternativa, escolhemos um candidato,de forma aleatória e selecionamos o *index* do próximo filho.

```
/*
* Returns index of a random son
*/
private static int selectSonRandom(Graph g,List<Graph> visited)
    {
    int indexNext = -1;
    if (g.sons.size() > 0) {
        Random random = new Random();
        while (true) {
            indexNext = random.nextInt(g.sons.size());
            if (!visited.contains(g.sons.get(indexNext))) {
                return indexNext;
            }
        }
    }
    return indexNext;
}
```

5) *Simulated Annealing*

O simullated Anneling é uma heurística para otmização que consiste numa técnica de pesquisa local. O algoritmo substitui a solução atual por uma solução próxima escolhida de acordo com uma função e uma variável T (dita Temperatura,por analogia). À medida que o algoritmo avança, o valor de T é decrementado,começando a convergir para uma solução ótima.

Uma das vantagens deste algoritmo é permitir testar soluções distantes da solução atual.

Para a implementação deste algoritmo de pesquisa, baseamos-nos no pseudo-código do *AIMA*.

Definimos uma temperatura inicial de 1000 e um *cooling factor* de 0.95 que diminui a temperatura a cada iteração. De forma a escolher qual seria o próximo candidato, escolhemos um index aleatório e calculamos a variação do número de cruzamentos entre o candidato atual e o possível candidato a escolher a seguir (ΔE).

A cada geração de um novo vizinho, a variação do valor da função é testada, havendo as seguintes situações:

- $\Delta E > 0 \rightarrow$ Implica que a nova solução é melhor do que a anterior, ficando esta como nova solução corrente.
- Nos restantes casos $\rightarrow$ Apenas aceitamos o próximo candidato com uma probabilidade menor que 1.

```
/*
* Implements the Simulated Annealing
*/
public static Graph simulatedAnnealing(Graph cur){
    List<Graph> visited = new ArrayList<Graph>(); // list of
    visited permutations
    int temperature = 1000;
    double coolingFactor = 0.95;
    int deltaE;
    int indexNext = -1;
    double probability = 0;
    for (int i = 0 ;  i < Integer.MAX_VALUE ; i++){
        probability = Math.random();
        visited.add(cur);
        generateSons(cur);
        temperature*= coolingFactor;
        if (temperature == 0 || cur.sons.size() == 0) {
            return cur;
        }
        indexNext = selectSonRandom(cur,visited);

        deltaE = numberOfConficts(cur.sons.get(indexNext))  -
    numberOfConficts(cur);

        if (deltaE > 0) {
            cur = cur.sons.get(indexNext);
        }
        else if(Math.exp(deltaE / temperature) > probability) {
```

```
27                cur = cur.sons.get(indexNext);
28            }
29        }
30        return cur;
31    }
```

Erros Conhecidos

- Na maioria das vezes o nosso *hillClimbing* entra em loop infinito o que implica que o nosso two-Exchange não está perfeito.

Casos experimentais

Para o seguinte polígono, com os pontos (Figura 1 na página 16):

$(1,1)(-1,-3)(-3,0)(1,0)(2,-3)(1,-2)(1,2)(0,-2)(-1,2)(2,1)(1,1)$

- **Hill Climbing** - *Candidato com menor perímetro*
  - Foram efetuadas 30 iterações
  - Tempo de execução: 1023182 nano-segundos
  - $(1,2)(-1,2)(1,1)(1,0)(-3,0)(-1,-3)(0,-2)(1,-2)(2,-3)(2,1)(1,2)$
  - Figura 2 na página 16
- **Hill Climbing** - *Primeiro Candidato*
  - Foram efetuadas 34 iterações
  - Tempo de execução: 476434 nanossegundos
  - $(2,-3)(-1,-3)(0,-2)(-3,0)(1,-2)(1,0)(1,1)(-1,2)(1,2)(2,1)(2,-3)$
  - Figura 3 na página 17
- **Hill Climbing** - *Candidato com menor número de Interseções*
  - Foram efetuadas 5 iterações
  - Tempo de execução: 1268185 nanossegundos
  - $(1,-2)(2,-3)(-1,-3)(-3,0)(-1,2)(0,-2)(1,0)(1,1)(1,2)(2,1)(1,-2)$
  - Figura 4 na página 17
- **Hill Climbing** - *Candidato aleatório*

- Foram efetuadas 21 iterações
- Tempo de execução: 1065889 nanossegundos
- $(2,-3)(-1,-3)(0,-2)(1,-2)(1,0)(-3,0)(1,1)(-1,2)(1,2)(2,1)(2,-3)$
- Figura 5 na página 17

- **Simulated Annealing**
  - Tempo de execução: 844323 nanossegundos
  - $(2,-3)(-1,-3)(0,-2)(1,-2)(1,0)(-3,0)(1,1)(-1,2)(1,2)(2,1)(2,-3)$
  - Figura 6 na página 17

Analisando os resultados obtidos nestes casos experimentais, podemos verificar pelas imagens que obtemos um polígono simples. Pelos nossos testes,podemos verificar que executando a heurística "nearest-neighbour-first"primeiro, o número de iterações efetuadas diminui pois alguns dos cruzamentos ficam resolvidos ao contrário de fazer permutação aleatória que pode piorar o número de cruzamentos. Ao testar o simulated annelling verificamos que quanto maior a temperatura inicial, a aceitação de movimentos que pioram a situação atual é maior.O contrário se verifica que quanto mais perto de zero se encontra o arrefecimento, mais depressa o algoritmo vai recusar movimentos que pioram a situação atual.

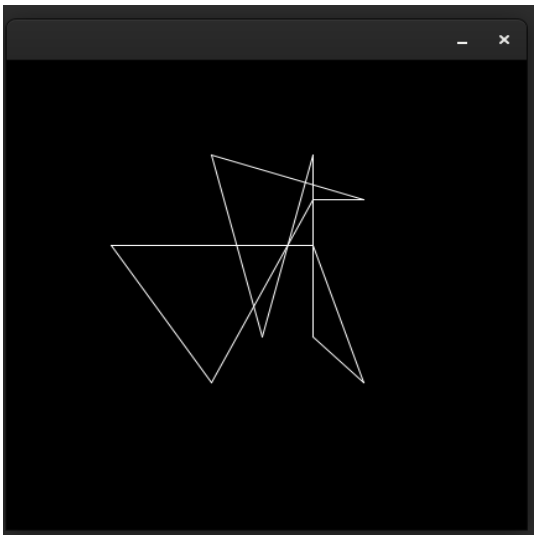

Figura 1: Polígono Original

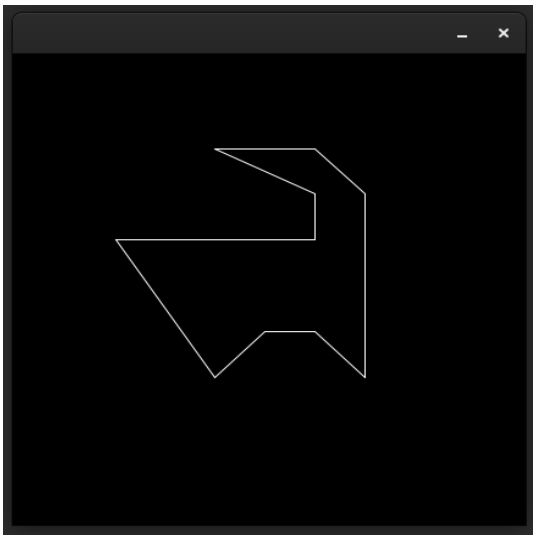

Figura 2: Hill Climbing - Perímetro

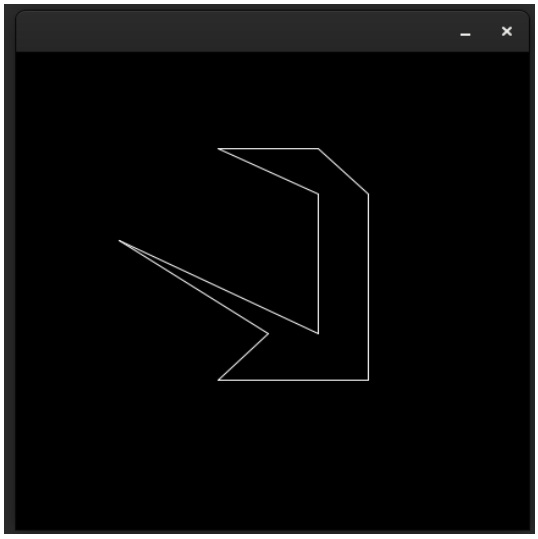

Figura 3: Hill Climbing - Primeiro Candidato

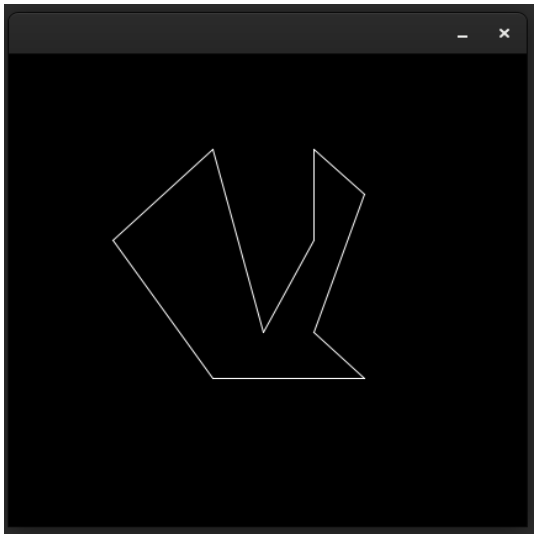

Figura 4: Hill Climbing - Menor Interseções

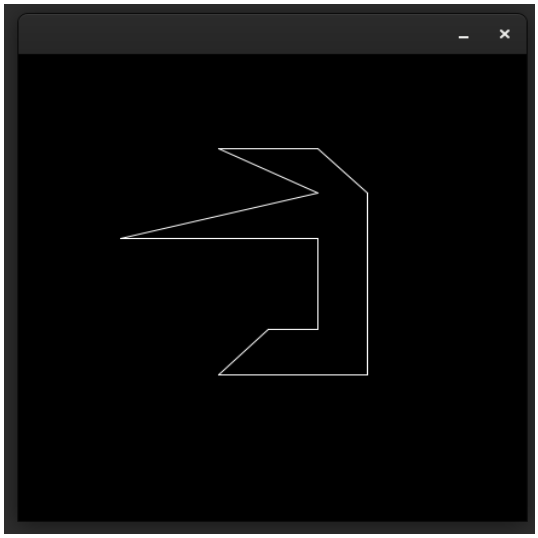

Figura 5: Hill Climbing - Aleatório

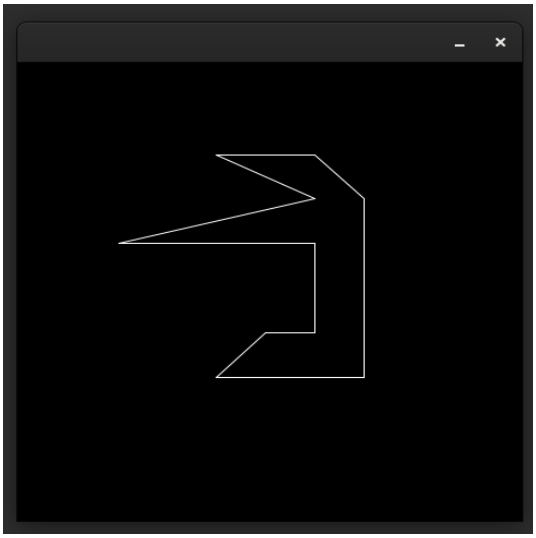

Figura 6: Simulated Annealing

Conclusão

Neste trabalho,como objetivos não atingidos destacamos o facto de não termos conseguido implementar o último exercício proposto como também o facto do nosso algoritmo de *two-Exchange* não parecer estar a funcionar direito para todos os casos provocando a que entre em *loop*.Em contrapartida,podemos considerar como objetivos atingidos o facto de termos conseguido implementar a parte gráfica(ainda que longe de perfeita) assim como os 5 primeiros exercícios propostos.

Contudo,com a realização deste trabalho conseguimos enriquecer o nosso conhecimento sobre o problema do caixeiro viajante (TSP), utilizando um gerador de polígonos simples(RPG) que aplica ideias semelhantes.

A implementação dos diferentes algoritmos de pesquisa propostos(*nearest-neighbour first*, *2-exhange* e *hill climbing* assim como a *gui*(parte visual) foi muito produtivo para nós pois nunca tínhamos tido contacto com nenhuma delas contribuindo deste modo para o nosso crescimento académico.

Para a execução deste trabalho não foi necessária a divisão de tarefas pois conseguimos organizar-nos de forma a resolver os problemas propostos em conjunto assim como a execução do relatório.

Bibliografia

- `https://www.cosy.sbg.ac.at/~held/projects/rpd/rpd.html`
- `https://sbgdb.cs.sbg.ac.at/`
- `https://github.com/aimacode/aima-pseudocode/blob/master/md/Simulated-Annealing.md`
- `https://en.wikipedia.org/wiki/Travelling_salesman_problem#Description`
  5 de Abril de 2021
- `https://en.wikipedia.org/wiki/Nearest_neighbour_algorithm`
  5 de Abril de 2021
- `https://en.wikipedia.org/wiki/Hill_climbing`
  5 de Abril de 2021
- `https://pt.wikipedia.org/wiki/Simulated_annealing`
  5 de Abril de 2021