FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Trabalho Prático 1

Inteligência Artificial CC2006

João Pedro Neves Diniz 201503876
José Alberto Martins Fernandes 201707227

2 Abril 2021

1. Introdução

Neste projeto iremos rever um dos exercícios mais conhecidos baseado no melhoramento combinatório, o *Traveling Salesman Problem (TSP)*.

Este problema procura, sabendo a localização de uma certa quantidade de pontos num qualquer plano, produzir o ciclo de Hamilton mais curto possível. Um ciclo de Hamilton é um percurso fechado que visita cada ponto apenas uma vez (exceto o ponto inicial que é visitado duas vezes: na partida e na chegada).

Tivemos então como objetivo deste trabalho a implementação deste mesmo problema num programa de Java, incluindo a geração de um conjunto *random* de pontos no plano e de uma ordem que procura a produção de um polígono simples usando esses mesmos pontos, e a implementação também de vários métodos de pesquisa local e de pesquisa local estocástica que visam procurar melhores soluções consoante diferentes objetivos. Um polígono simples é uma figura plana limitada por segmentos de reta que não se intersetam (exceto nos extremos que ligam dois quaisquer segmentos).

2. Estruturas de Dados

O desenvolvimento do programa teve como base uma única classe de onde resultou a seguinte estrutura:

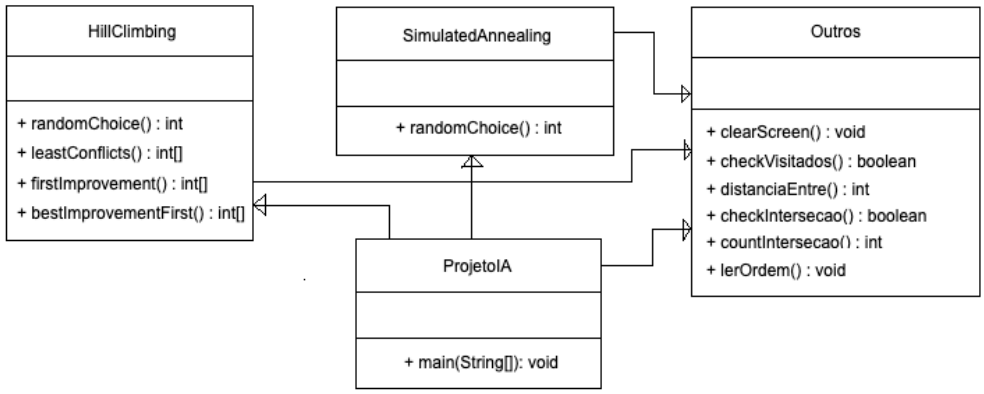


Figura 1. Diagrama de métodos

2.1. ProjetoIA

2.1.1. main()

Na classe ProjetoIA encontra-se unicamente a função main que serve como interface gráfica para interação entre o programa e o utilizador para efeitos de input/output.

2.2. Outros

A classe Outros alberga funções auxiliares necessárias para os diversos cálculos, para leitura de elementos e para uma melhor organização do output.

2.2.1. clearScreen()

A função *clearScreen* não retorna nada nem recebe argumentos e serve única e exclusivamente para limpar o ecrã do terminal sendo que a sua relevância no programa em si é mínima.

```
public static void clearScreen () {
    System.out.print("\033[H\033[2J");
    System.out.flush();
}
```

2.2.2. checkVisitados()

A função *checkVisitados* retorna um inteiro e recebe como argumentos um array de inteiros e um inteiro. O array *visitados* é um argumento que representa os pontos existentes e *n* o número de pontos definido. Se existe um qualquer ponto ainda não visitado, a função retorna 0. Caso contrário, a função irá então retornar 1.

```
public static int checkVisitados (int[] visitados, int n) {
    for (int i=0; i<n; i++){
      if (visitados[i] == 0) return 0;
    }
    return 1;
 }
```

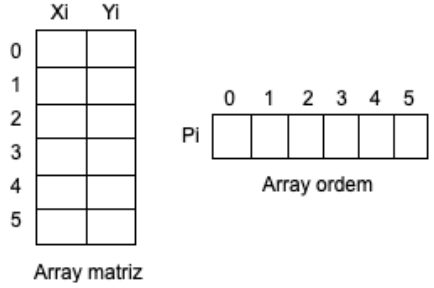
2.2.3. distanciaEntre()

A função retorna um inteiro e recebe quatro inteiros como argumentos. Esse input corresponde às coordenadas (x,y) de dois pontos do plano. Começa-se por calcular o tamanho dos catetos recorrendo à libraria *Math* da API do Java e através dos mesmos calcula-se a hipotenusa que corresponde à distância absoluta entre os pontos fornecidos.

```
public static int distanciaEntre (int xA, int xB, int yA, int yB) {
    int cateto1 = Math.abs(xA-xB);
    int cateto2 = Math.abs(yA-yB);
    int hipotenusa = cateto1*cateto1 + cateto2*cateto2;
    return hipotenusa;
}
```

2.2.4. checkIntersecao()

A função, tal como o nome indica, calcula se existem interseções entre arestas. Recebe como argumentos um array denominado *ordem,* que contém a ordem dos pontos do polígono gerado, e *matriz*, que contém as coordenadas dos pontos existentes no seguinte formato:



Recebe também um inteiro *n* que indica o número de pontos. Através da função *linesIntersect* da libraria *Line2D* da API do Java é possível calcular a existência de interseções entre dois segmentos de reta. Se existir uma ou mais interseções a função retorna o valor booleano *true*, caso contrário devolve *false*.

```
public static boolean checkIntersecao (int[] ordem, int[][] matriz, int n){
    for (int i=0; i<n-1; i++) {
      int aux=0;
      if (i==0) aux = n-1;
      else aux = n;
      for (int j=i+2; j<aux; j++)  {
```

```
            boolean temp = Line2D.linesIntersect(matriz[ordem[i]][0], matriz[ordem[i]][1],
matriz[ordem[i+1]][0], matriz[ordem[i+1]][1], matriz[ordem[j]][0], matriz[ordem[j]][1],
matriz[ordem[j+1]][0], matriz[ordem[j+1]][1]);
            if (temp == true) return true;
          }
        }
        return false;
}
```

2.2.5. countIntersecao()

A função *countIntersecao* à semelhança da função *checkIntersecao* partilham dos mesmos argumentos sendo que a única diferença é o tipo de variável retornada que no caso é um inteiro contendo o número de interseções determinadas.

```
public static int countIntersecao (int[] ordem, int[][] matriz, int n){
      int contagem=0;
      for (int i=0; i<n-1; i++) {
         int aux=0;
         if (i==0) aux = n-1;
         else aux = n;
         for (int j=i+2; j<aux; j++)
             boolean temp = Line2D.linesIntersect(matriz[ordem[i]][0], matriz[ordem[i]][1],
matriz[ordem[i+1]][0], matriz[ordem[i+1]][1], matriz[ordem[j]][0], matriz[ordem[j]][1],
matriz[ordem[j+1]][0], matriz[ordem[j+1]][1]);
            if (temp == true) contagem++;
         }
      }
      return contagem;
 }
```

2.2.6. lerOrdem()

A função tem como objetivo imprimir no terminal a atual sequência de pontos que formam o polígono. Tem como argumentos o array *ordem* e o inteiro *n* que determina o tamanho do array (exceto a última posição, pois será sempre igual à primeira).

```
public static void lerOrdem (int[] ordem, int n) {
      for (int i=0; i<n; i++) {
         System.out.print(ordem[i] + ",");
      }
      System.out.println(ordem[n]);
}
```

2.3. SimulatedAnnealing

2.3.1. randomChoice()

A função *randomChoice* recebe os mesmos argumentos que as funções *checkIntersecao* e *countIntersecao* e começa primeiramente por calcular exatamente isso, o número de interseções existentes. Posteriormente determina um número aleatório no intervalo [1, numInter]. Esse mesmo número carateriza a interseção a tratar. Quando o número da iteração é equivalente ao número

determinado anteriormente significa que estamos na interseção pretendida, ao qual ele aplica o seguinte procedimento:

- Assume-se AB (um segmento de reta composto pelos pontos A e B), CD (um outro segmento de reta composto por C e D) e a interseção dos mesmos, ABxCD. Efetua-se então a troca de um dos vértices de cada segmento, resultado em AC e BD.
- No final, a função retorna um novo array de inteiros com a nova ordem de pontos.

```
public static int[] randomChoice (int[] ordem, int[][] matriz, int n) {
    int numInter = Outros.checkIntersecao(ordem, matriz, n);
    int contagem = 0;
    int numRandom = (int)((Math.random()*(numInter))+1);
    for (int i=0; i<n-1; i++) {
      int aux = 0;
      if (i==0) aux = n - 1;
      else aux = n;
      for (int j=i+2; j<aux; j++) {
        boolean temp = Line2D.linesIntersect(matriz[ordem[i]][0], matriz[ordem[i]][1],
matriz[ordem[i+1]][0], matriz[ordem[i+1]][1], matriz[ordem[j]][0], matriz[ordem[j]][1],
matriz[ordem[j+1]][0], matriz[ordem[j+1]][1]);
        if (temp == true) {
          ++contagem;
          if (contagem==numRandom) {
            int a = ordem[i+1];
            ordem[i+1] = ordem[j];
            ordem[j] = a;
            return ordem;
          }
        }
      }
    }
    return ordem;
}
```

2.4. HillClimbing

2.4.1. randomChoice()

A função *randomChoice* presente na classe *HillClimbing* é semelhante à função *randomChoice* presente na classe SimulatedAnnealing sendo que a única diferença está na impressão para o terminal de todas as interseções existentes e posterior escolha da interseção a resolver.

```
public static int[] randomChoice (int[] ordem, int[][] matriz, int n) {
    int numInter = 0, contagem = 0;
    System.out.println("Possibilidades:");
    for (int i=0; i<n-1; i++) {
      int aux = 0;
      if (i==0) aux = n - 1;
      else aux = n;
      for (int j=i+2; j<aux; j++) {
        boolean temp = Line2D.linesIntersect(matriz[ordem[i]][0], matriz[ordem[i]][1],
matriz[ordem[i+1]][0], matriz[ordem[i+1]][1], matriz[ordem[j]][0], matriz[ordem[j]][1],
matriz[ordem[j+1]][0], matriz[ordem[j+1]][1]);
```

```
            if (temp == true) {
               ++numInter;
               System.out.println(numInter +" => [" + ordem[i] + "-" + ordem[i+1] + "]-[" + ordem[j]
+ "-" + ordem[j+1] + "]");
            }
         }
      }
      int numRandom = (int)((Math.random()*(numInter))+1);
      System.out.println("Número escolhido: " + numRandom);
      for (int i=0; i<n-1; i++) {
         int aux = 0;
         if (i==0) aux = n - 1;
         else aux = n;
         for (int j=i+2; j<aux; j++){
            boolean temp = Line2D.linesIntersect(matriz[ordem[i]][0], matriz[ordem[i]][1],
matriz[ordem[i+1]][0], matriz[ordem[i+1]][1], matriz[ordem[j]][0], matriz[ordem[j]][1],
matriz[ordem[j+1]][0], matriz[ordem[j+1]][1]);
            if (temp == true) {
               ++contagem;
               if (contagem == numRandom) {
                  System.out.println("A aplicar o melhoramento");
                  int a = ordem[i+1];
                  ordem[i+1] = ordem[j];
                  ordem[j] = a;
                  return ordem;
               }
            }
         }
      }
      return ordem;
}
```

2.4.2. leastConflicts()

A função começa por calcular o número de interseções existentes consoante o array *ordem* fornecido. É criado um array bidimensional denominado *solucoes* de *contagem* linhas e *n+1* colunas onde inicialmente são escritas em todas as linhas do array a ordem atual dos pontos que formam o polígono.

São então solucionadas por ordem as interseções encontradas e escrita a nova ordem de pontos na respetiva linha do array *solucoes*. Isto é, na linha 0 encontrar-se-á a ordem posterior à resolução a primeira interseção, na linha 1 teremos a ordem posterior à resoluçao da segunda interseção, etc.

O array *contagem3* é criado com o tamanho *contagem* e em cada posição desse mesmo array é guardado a quantidade de conflitos que surgem consoante cada possível solução calculada anteriormente.

Seguidamente é escolhida a interseção que gera menos conflitos com a sua resolução. A posição do menor valor em *contagem3* irá corresponder à posição da ordem correta guardada em *solucoes.*

Este processo é repetido até não se encontrar mais interseções.

```
public static int[] leastConflicts (int[] ordem, int[][] matriz, int n) {
```

```
      int contagem = Outros.countIntersecao(ordem, matriz, n);
      System.out.println("Contagem de conflitos: " + contagem);
      int[][] solucoes = new int[contagem][n+1];
      for (int i=0; i<contagem; i++) {
         for (int j=0; j<n+1; j++) {
            solucoes[i][j] = ordem[j];
         }
      }
      int contagem2=0;
      for (int i=0; i<n-1; i++) {
         int aux = 0;
         if (i==0) aux = n - 1;
         else aux = n;
         for (int j=i+2; j<aux; j++) {
            boolean temp = Line2D.linesIntersect(matriz[ordem[i]][0], matriz[ordem[i]][1],
matriz[ordem[i+1]][0], matriz[ordem[i+1]][1], matriz[ordem[j]][0], matriz[ordem[j]][1],
matriz[ordem[j+1]][0], matriz[ordem[j+1]][1]);
            if (temp == true) {
               int a = solucoes[contagem2][i+1];
               solucoes[contagem2][i+1] = solucoes[contagem2][j];
               solucoes[contagem2][j] = a;
               contagem2++;
            }
         }
      }
      int[] contagem3 = new int[contagem];
      for (int i=0; i<contagem; i++) {
         contagem3[i] = 0;
      }
      for (int i=0; i<contagem; i++) {
         for (int j=0; j<n-1; j++) {
            int aux = 0;
            if (j==0) aux = n - 1;
            else aux = n;
            for (int k=j+2; k<aux; k++) {
               boolean temp = Line2D.linesIntersect(matriz[solucoes[i][j]][0],
matriz[solucoes[i][j]][1], matriz[solucoes[i][j+1]][0], matriz[solucoes[i][j+1]][1],
matriz[solucoes[i][k]][0], matriz[solucoes[i][k]][1], matriz[solucoes[i][k+1]][0],
matriz[solucoes[i][k+1]][1]);
               if(temp == true){
                  contagem3[i]++;
               }
            }
         }
      }
      int minConflitos = Integer.MAX_VALUE;
      int minSolucao = 0;
      for (int i=0; i<contagem; i++) {
         if (contagem3[i] < minConflitos) {
```

```
            minConflitos = contagem3[i];
            minSolucao=i;
          }
        }
        System.out.println("A aplicar o melhoramento");
        System.out.println("Número minimo de conflitos encontrado foi " + minConflitos);
        for (int i=0; i<n+1; i++) {
          ordem[i] = solucoes[minSolucao][i];
        }
        return ordem;
}
```

2.4.3. firstImprovement()

Um método iterativo que vai solucionando a primeira interseção que encontra a cada iteração. Pode não ser um método tão eficiente como o *leastConflicts* pois não escolhe a interseção ótima atual. Retorna um array de inteiros com a nova ordem de pontos.

```
public static int[] firstImprovement (int[] ordem, int[][] matriz, int n) {
      for (int i=0; i<n-1; i++) {
        int aux = 0;
        if (i==0) aux = n - 1;
        else aux = n;
        for (int j=i+2; j<aux; j++){
              boolean temp = Line2D.linesIntersect(matriz[ordem[i]][0], matriz[ordem[i]][1],
         matriz[ordem[i+1]][0], matriz[ordem[i+1]][1], matriz[ordem[j]][0], matriz[ordem[j]][1],
         matriz[ordem[j+1]][0], matriz[ordem[j+1]][1]);
            if(temp == true){
              System.out.println("A aplicar o melhoramento");
              int a = ordem[i+1];
              ordem[i+1] = ordem[j];
              ordem[j] = a;
              return ordem;
            }
         }
      }
      return ordem;
}
```

2.4.4. bestImprovementFirst()

O método *bestImprovementFirst* é bastante semelhante ao método *leastConflicts* sendo que diferem no critério de escolha da interseção a resolver. O primeiro escolhia a interseção que ao ser resolvida criasse menos conflitos. Neste caso, a função escolhe a interseção que ao ser resolvida gera o menor perímetro possível do polígono. Retorna um array de inteiros com a nova ordem de pontos.

```
public static int[] bestImprovementFirst (int[] ordem,int[][] matriz, int n) {
      int contagem = Outros.countIntersecao(ordem, matriz, n);
      System.out.println("Contagem: " + contagem);
      int[][] solucoes = new int[contagem][n+1];
      for (int i=0; i<contagem; i++) {
        for (int j=0; j<n+1; j++) {
```

```
            solucoes[i][j]=ordem[j];
        }
    }
    int contagem2 = 0;
    for (int i=0; i<n-1; i++) {
        int aux = 0;
        if (i==0) aux = n - 1;
        else aux = n;
        for (int j=i+2; j<aux; j++) {
            boolean temp = Line2D.linesIntersect(matriz[ordem[i]][0], matriz[ordem[i]][1],
matriz[ordem[i+1]][0], matriz[ordem[i+1]][1], matriz[ordem[j]][0], matriz[ordem[j]][1],
matriz[ordem[j+1]][0], matriz[ordem[j+1]][1]);
            if (temp == true) {
                int a = solucoes[contagem2][i+1];
                solucoes[contagem2][i+1] = solucoes[contagem2][j];
                solucoes[contagem2][j] = a;
                contagem2++;
            }
        }
    }
    int minPerimetro = Integer.MAX_VALUE;
    int minSolucao=0;
    for (int i=0; i<contagem; i++) {
        int tempPerimetro=0;
        for (int j=0; j<n; j++) {
            int aux = Outros.distanciaEntre(matriz[solucoes[i][j]][0], matriz[solucoes[i][j+1]][0],
matriz[solucoes[i][j]][1], matriz[solucoes[i][j+1]][1]);
            tempPerimetro += aux;
        }
        if (tempPerimetro < minPerimetro) {
            minPerimetro = tempPerimetro;
            minSolucao = i;
        }
    }
    System.out.println("A aplicar o melhoramento");
    System.out.println("Perimetro mínimo encontrado foi " + minPerimetro);
    for (int i=0; i<n+1; i++) {
        ordem[i] = solucoes[minSolucao][i];
    }
    return ordem;
}
```

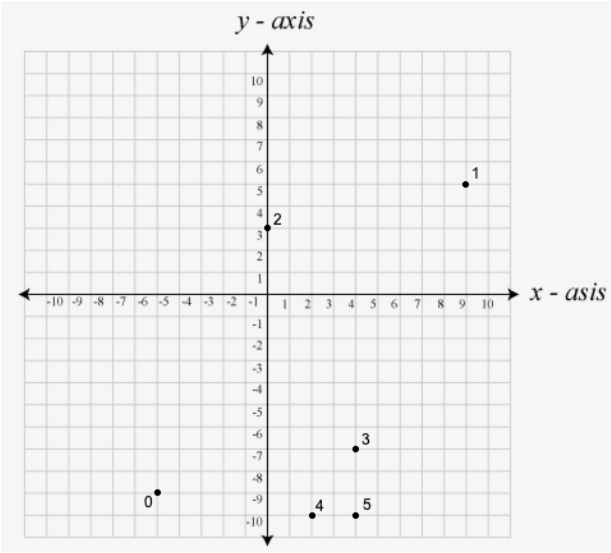
3. Exemplos

Compilado o programa, é indicado para introduzirmos o valor *m* que indica a coordenada máxima e o negativo da coordenada mínima de cada eixo do plano. Neste exemplo vamos escolher 10, ou seja, as coordenadas dos pontos gerados podem variar entre -10 e 10 no eixo das abcissas e no eixo das ordenadas.

Depois de definido o valor *m*, é pedido para introduzir o valor para a variável *n* que indica o número de pontos que se pretende gerar.

Seguidamente irá aparecer uma opção para escolher entre uma Permutação *Random* dos Pontos ou um *Nearest-Neighbour First*.

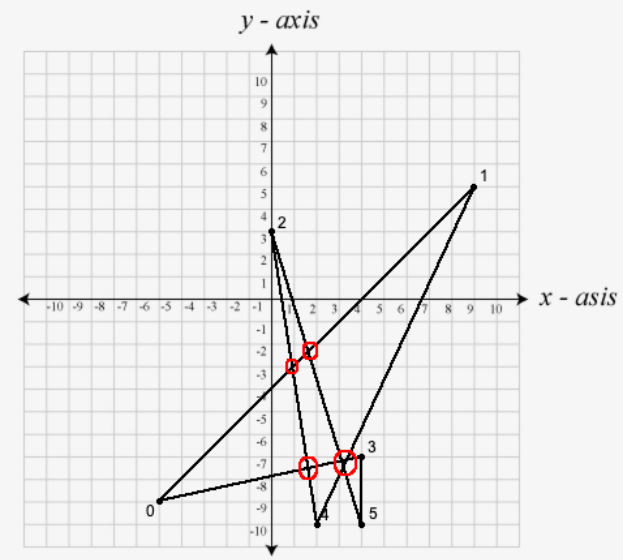
```
Insira o m:
   => 10
Insira o n:
   => 6

Ponto 0 => [-5,-9]
Ponto 1 => [9,5]
Ponto 2 => [0,3]
Ponto 3 => [4,-7]
Ponto 4 => [2,-10]
Ponto 5 => [4,-10]

Selecione o tipo de solução desejado:
1) Permutação Random dos Pontos
2) Nearest-Neighbour First
   =>
```



3.1. Permutação Random dos Pontos

A Permutação *Random* dos Pontos não define objetivo específico para a escolha da ordem dos pontos, apenas efetua uma escolha iterativa aleatória dos pontos gerados.

```
Selecione o tipo de solução desejado:
1) Permutação Random dos Pontos
2) Nearest-Neighbour First
   => 1

Ordem: 2,5,3,0,1,4,2
```



3.2. *Nearest-Neighbour First*

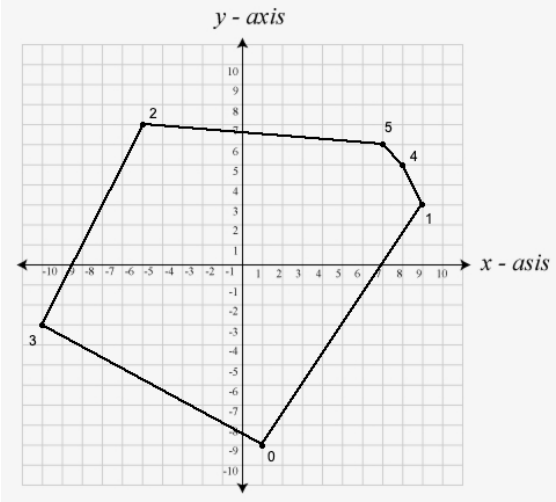
O *Nearest-Neighbour First* procura a partir de um ponto inicial escolhido de forma aleatória o ponto sucessor na vizinhança, valorizando a proximidade dos mesmos.
Neste exemplo obtivemos os seguintes pontos (com m=10 e n=6):

```
Insira o m:
   => 10
Insira o n:
   => 6

Ponto 0 => [1,-9]
Ponto 1 => [9,3]
Ponto 2 => [-5,7]
Ponto 3 => [-10,-3]
Ponto 4 => [8,5]
Ponto 5 => [7,6]
```

Escolhendo o método *Nearest Neighbour First*:

```
Selecione o tipo de solução desejado:
1) Permutação Random dos Pontos
2) Nearest-Neighbour First
   => 2

Ordem: 1,4,5,2,3,0,1

Não há interseções
```

Obtemos o seguinte polígono simples:



De reparar que dado os pontos gerados e o método utilizado não foi criada uma única interseção sendo que desde o início se verifica um polígono simples.

3.3. Métodos de melhoramento

3.3.1. *Hill Climbing*

Começamos, como em qualquer um destes métodos, por inserir o m e n desejados:

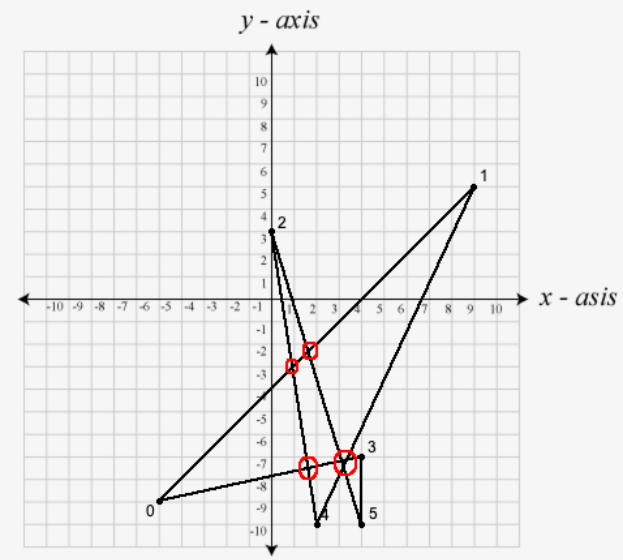
```
Insira o m:
   => 10
Insira o n:
   => 6

Ponto 0 => [-5,-9]
Ponto 1 => [9,5]
Ponto 2 => [0,3]
Ponto 3 => [4,-7]
Ponto 4 => [2,-10]
Ponto 5 => [4,-10]
```

Depois de gerados os pontos, podemos gerar um polígono através de qualquer uma das duas formas distintas. Neste exemplo vamos escolher a permutação dos pontos:

```
Selecione o tipo de solução desejado:
1) Permutação Random dos Pontos
2) Nearest-Neighbour First
   => 1

Ordem: 2,5,3,0,1,4,2

Existem 6 interseções
```

A ordem dos pontos indicada representa o polígono gerado. O ponto 2 liga ao ponto 5, o 5 ao 3 e assim sucessivamente até ligar novamente ao ponto inicial. É indicado também o número de interseções encontradas.

As interseções são facilmente identificáveis.




```
Escolha o melhoramento a aplicar:
1) Hill-Climbing
2) Simulated Annealing
   => 1
```

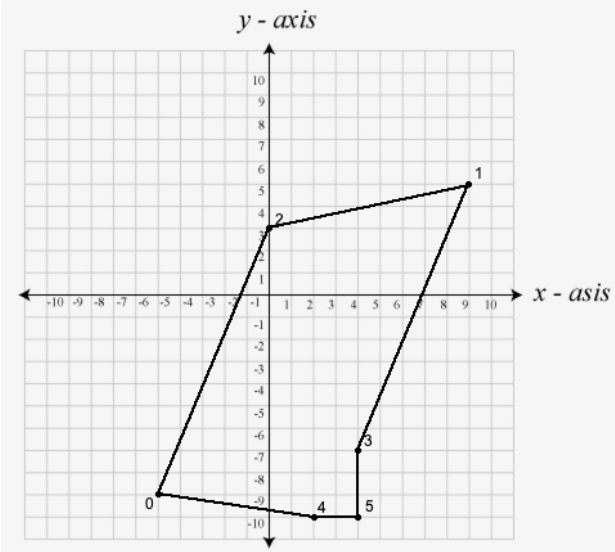
Depois de selecionarmos *Hill Climbing* escolhemos o objetivo que pretendemos.

3.3.1.1. *Best-Improvement First*

Este método específico tem como objetivo procurar sempre o candidato com menor perímetro. Caso nenhum dos candidatos na vizinhança seja melhor do que a ordem atual, a ordem manter-se-á e não são efetuados mais melhoramentos.
Neste exemplo iremos usar os pontos do exemplo anterior (fornecidos no 3.3.1):

```
Escolha o objetivo que pretende aplicar ao hill-climbing:
1) Escolher sempre o candidato que reduz mais o perímetro do polígono
2) Escolher sempre o primeiro candidato da vizinhança
3) Escolher sempre o candidato que produza menos conflitos de arestas
4) Escolher sempre um candidato random
   => 1

Ordem Inicial: 2,5,3,0,1,4,2

Perimetro Inicial: 1118

Contagem: 6
A aplicar o melhoramento
Perimetro mínimo encontrado foi 598
novaOrdem: 2,1,3,0,5,4,2

Contagem: 2
A aplicar o melhoramento
Perimetro mínimo encontrado foi 522
novaOrdem: 2,1,3,4,5,0,2

Contagem: 1
A aplicar o melhoramento
Perimetro mínimo encontrado foi 486
novaOrdem: 2,1,3,5,4,0,2

Melhorias aplicadas
Resultado Final: 2,1,3,5,4,0,2
```

O resultado final apresenta o seguinte polígono:



Neste caso, após o melhoramento obtivemos como resultado um polígono simples com o perímetro mínimo possível para os pontos fornecidos.

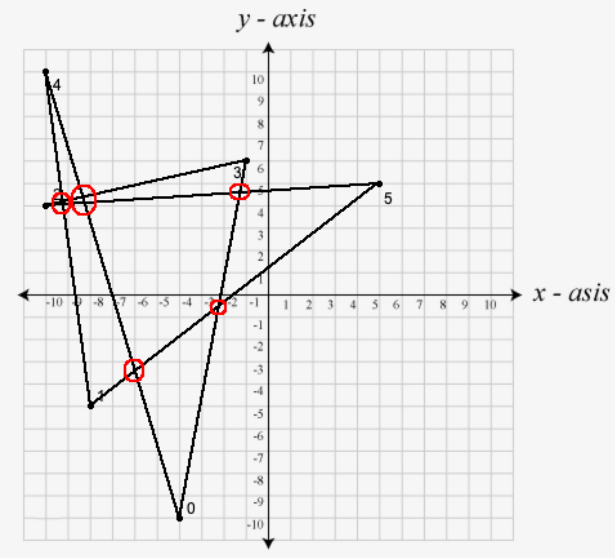
3.3.1.2. *First-Improvement*

O *First-Improvement* percorre a ordem dos pontos e escolhe sempre o primeiro conflito que encontra.

Neste exemplo vamos repetir os inputs do 3.3.1.1. (m = 10, n = 6 e a Permutação *Random* dos Pontos) mas iremos obter pontos diferentes:

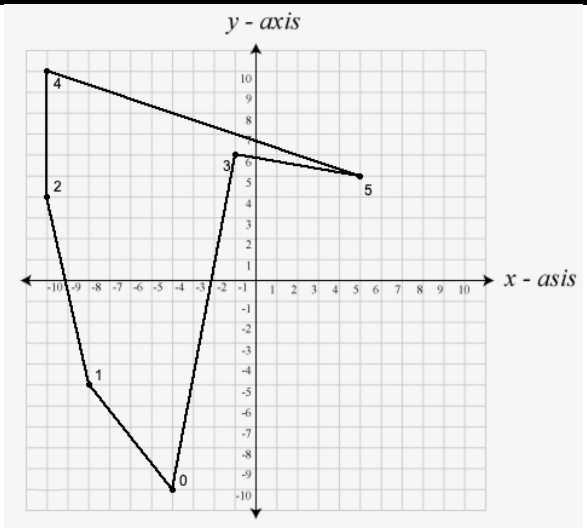
```
Insira o m:
   => 10
Insira o n:
   => 6

Ponto 0 => [-4,-10]
Ponto 1 => [-8,-5]
Ponto 2 => [-10,4]
Ponto 3 => [-1,6]
Ponto 4 => [-10,10]
Ponto 5 => [5,5]

Selecione o tipo de solução desejado:
1) Permutação Random dos Pontos
2) Nearest-Neighbour First
   => 1

Ordem: 3,2,5,1,4,0,3

Existem 7 interseções
Escolha o melhoramento a aplicar:
1) Hill-Climbing
2) Simulated Annealing
   =>
```

A seguinte ordem de pontos gerada forma um polígono com a seguinte forma geométrica onde foram detetadas 7 interseções:



Após aplicar o melhoramento:

```
Escolha o objetivo que pretende aplicar ao hill-climbing:
1) Escolher sempre o candidato que reduz mais o perímetro do polígono
2) Escolher sempre o primeiro candidato da vizinhança
3) Escolher sempre o candidato que produza menos conflitos de arestas
4) Escolher sempre um candidato random
   => 2

Ordem Inicial: 3,2,5,1,4,0,3

A aplicar o melhoramento
novaOrdem: 3,1,5,2,4,0,3

A aplicar o melhoramento
novaOrdem: 3,5,1,2,4,0,3

A aplicar o melhoramento
novaOrdem: 3,5,4,2,1,0,3

Melhorias aplicadas
Resultado Final: 3,5,4,2,1,0,3
```

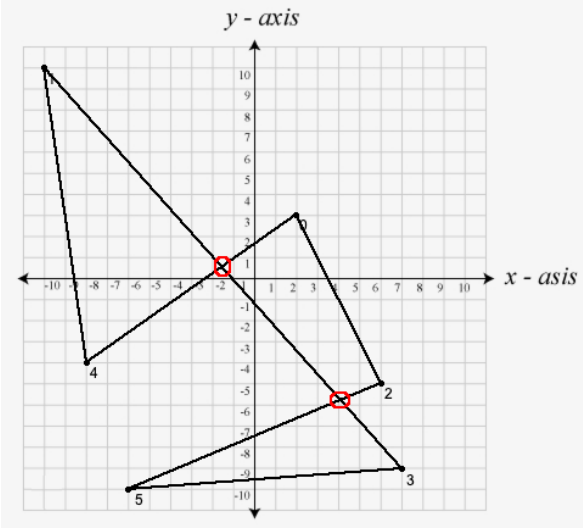


3.3.1.3. *Least-Conflicts*

O *Least-Conflicts* verifica quantos conflitos vão ser produzidos em cada 2-exchange possível. Se algum dos candidatos a solução produza menos conflitos, essa será a nova ordem. Caso contrário, o programa não melhora mais e aceita a ordem atual como o melhor que consegue melhorar.
Repetimos novamente o input do 3.3.1.1.:

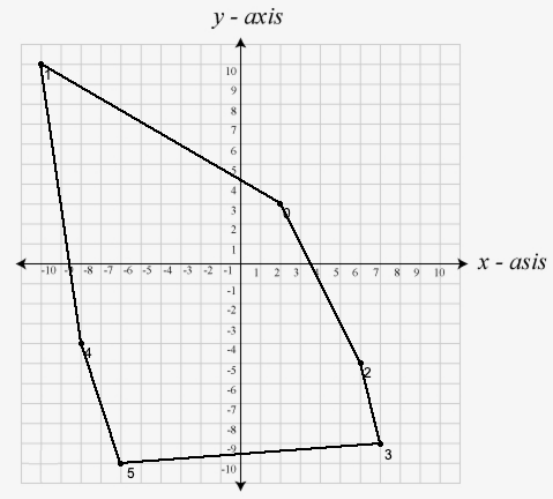
```
Insira o m:
   => 10
Insira o n:
   => 6

Ponto 0 => [2,3]
Ponto 1 => [-10,10]
Ponto 2 => [6,-5]
Ponto 3 => [7,-9]
Ponto 4 => [-8,-4]
Ponto 5 => [-6,-10]

Selecione o tipo de solução desejado:
1) Permutação Random dos Pontos
2) Nearest-Neighbour First
   => 1

Ordem: 5,2,0,4,1,3,5

Existem 2 interseções
Escolha o melhoramento a aplicar:
1) Hill-Climbing
2) Simulated Annealing
```

A seguinte ordem de pontos gerada forma um polígono com a seguinte forma geométrica onde foram detetadas 2 interseções:



Após aplicar o objetivo desejado:

```
Escolha o objetivo que pretende aplicar ao hill-climbing:
1) Escolher sempre o candidato que reduz mais o perímetro do polígono
2) Escolher sempre o primeiro candidato da vizinhança
3) Escolher sempre o candidato que produza menos conflitos de arestas
4) Escolher sempre um candidato random
   => 3

Ordem Inicial: 5,2,0,4,1,3,5

Contagem de conflitos: 2
A aplicar o melhoramento
Número minimo de conflitos encontrado foi 1
novaOrdem: 5,2,0,1,4,3,5

Contagem de conflitos: 1
A aplicar o melhoramento
Número minimo de conflitos encontrado foi 1
novaOrdem: 5,4,0,1,2,3,5

Contagem de conflitos: 1
A aplicar o melhoramento
Número minimo de conflitos encontrado foi 0
novaOrdem: 5,4,1,0,2,3,5

Melhorias aplicadas
Resultado Final: 5,4,1,0,2,3,5
```

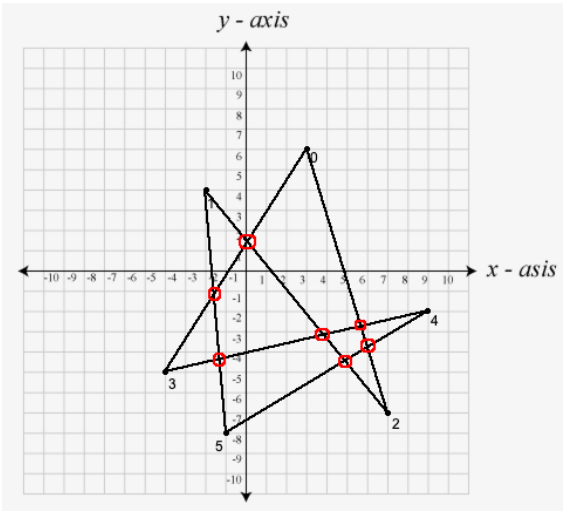


3.3.1.4. *Random Choice*

Este método escolhe a cada iteração um conflito aleatório para resolver até encontrar a solução. Repetimos novamente o input do 3.3.1.1.:

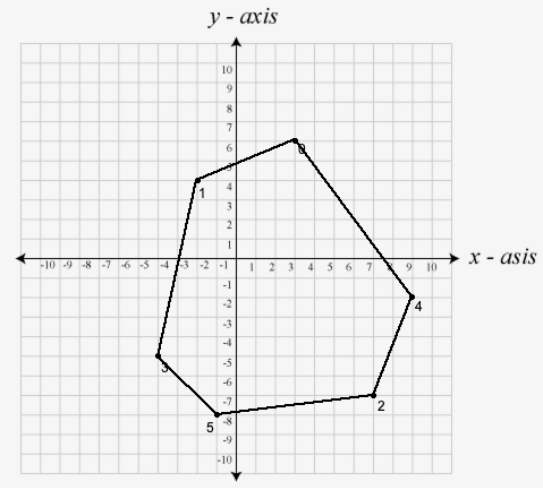
```
Insira o m:
   => 10
Insira o n:
   => 6

Ponto 0 => [3,6]
Ponto 1 => [-2,4]
Ponto 2 => [7,-7]
Ponto 3 => [-4,-5]
Ponto 4 => [9,-2]
Ponto 5 => [-1,-8]

Selecione o tipo de solução desejado:
1) Permutação Random dos Pontos
2) Nearest-Neighbour First
   => 1

Ordem: 2,0,3,4,5,1,2

Existem 7 interseções
Escolha o melhoramento a aplicar:
1) Hill-Climbing
2) Simulated Annealing
```

A seguinte ordem de pontos gerada forma um polígono com a seguinte forma geométrica onde foram detetadas 7 interseções:



Após as melhorias:

```
Escolha o objetivo que pretende aplicar ao hill-climbing:
1) Escolher sempre o candidato que reduz mais o perímetro do polígono
2) Escolher sempre o primeiro candidato da vizinhança
3) Escolher sempre o candidato que produza menos conflitos de arestas
4) Escolher sempre um candidato random
   => 4

Ordem Inicial: 2,0,3,4,5,1,2

Possibilidades:
1 => [2-0]-[3-4]
2 => [2-0]-[4-5]
3 => [0-3]-[5-1]
4 => [0-3]-[1-2]
5 => [3-4]-[5-1]
6 => [3-4]-[1-2]
7 => [4-5]-[1-2]
Número escolhido: 6
A aplicar o melhoramento
novaOrdem: 2,0,3,1,5,4,2

Possibilidades:
1 => [2-0]-[5-4]
2 => [0-3]-[1-5]
Número escolhido: 2
A aplicar o melhoramento
novaOrdem: 2,0,1,3,5,4,2

Possibilidades:
1 => [2-0]-[5-4]
Número escolhido: 1
A aplicar o melhoramento
novaOrdem: 2,5,1,3,0,4,2

Possibilidades:
1 => [5-1]-[3-0]
Número escolhido: 1
A aplicar o melhoramento
novaOrdem: 2,5,3,1,0,4,2

Melhorias aplicadas
Resultado Final: 2,5,3,1,0,4,2
```



3.3.2. *Simulated Annealing*

O Simulated Annealing é uma técnica que procura o melhor possível o máximo global de uma função (normalmente em espaços amostrais de grandes dimensões).

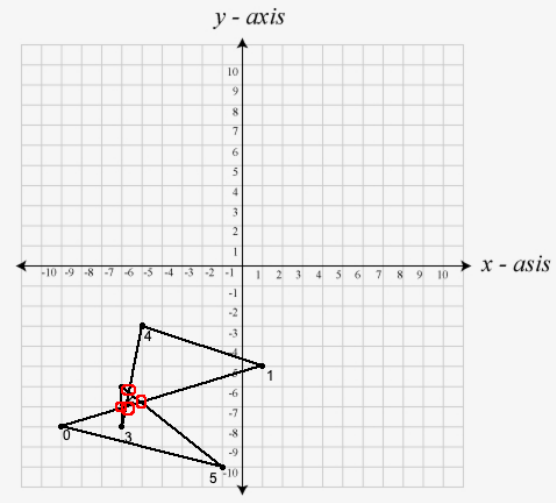
Começamos com uma solução inicial, um candidato vizinho aleatório e uma temperatura escolhida previamente. Calcula-se a variação de energia (Δ E) entre ambos. Como o nosso objetivo é minimizar, procuramos uma variação de energia negativa. Caso seja menor que zero, podemos considerar a nova ordem como solução. Caso não seja menor que zero, compara-se o fator de Boltzmann atual com um número aleatório do intervalo [0-1]. Se o fator for igual ou superior ao número aleatório, a nova ordem será aceite. Senão será rejeitada. Este processo é repetido até à temperatura atingir o valor zero ou até a solução ser encontrada. Durante estas iterações a temperatura vai arrefecendo, isto é, vai-se aproximando de zero gradualmente. Como o fator de Boltzmann se baseia na temperatura atual, significa que quanto mais tempo a pesquisa demore mais provável será a rejeição de ordens com energia superior a zero.

3.3.2.1. Exemplo

Neste exemplo usamos novamente o input do 3.3.1.1.:

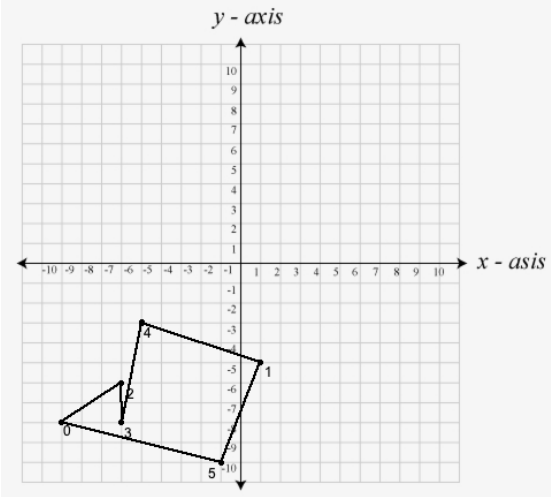
```
Insira o m:
   => 10
Insira o n:
   => 6

Ponto 0 => [-9,-8]
Ponto 1 => [1,-5]
Ponto 2 => [-6,-6]
Ponto 3 => [-6,-8]
Ponto 4 => [-5,-3]
Ponto 5 => [-1,-10]

Selecione o tipo de solução desejado:
1) Permutação Random dos Pontos
2) Nearest-Neighbour First
   => 1

Ordem: 4,3,2,5,0,1,4

Existem 4 interseções
```

Obtemos a ordem de pontos que resulta no seguinte polígono com 4 interseções detetadas:



Após a escolha do método de melhoria Simulated Annealing obtemos o polígono seguinte:

```
Escolha o melhoramento a aplicar:
1) Hill-Climbing
2) Simulated Annealing
   => 2

T is 3.8
novaOrdem: 4,3,2,0,5,1,4
DeltaE >= 0
Menor que random, novaOrdem recusada


Terminado
Nº de Iteraçoes: 1
Melhorias aplicadas
Resultado Final: 4,3,2,0,5,1,4
```



4. Conclusão:

Ao longo deste trabalho efetuamos bastantes testes para compreender se os métodos estavam efetivamente a funcionar. A partir desses testes chegamos a estas conclusões:

- Enquanto que a Permutação Random de Pontos fornece uma qualquer ordem sem um objetivo definido, muito provavelmente gerando também bastantes conflitos, a aplicação do Neirest Neighbour First apresenta normalmente uma quantidade mínima de interseções graças à valorização da proximidade do candidato.
- A diferença mais acentuada entre os métodos Hill Climbing (HC) e Simulated Annealing (SA) é o facto do primeiro regularmente procurar um máximo local enquanto que o segundo procura um máximo global. Isso acontece porque, ao contrário do SA, o HC não permite a regressão caso não encontre candidatos vizinhos melhores (porque é um método Greedy). Podemos então concluir que o SA é o método mais eficaz para encontrar o polígono simples.
- Dentro dos vários objetivos que usamos no método Hill Climbing, os mais eficazes pareceram ser

Continuamos com o mesmo bug que tinhamos na primeira submissão (24 de março) que possibilita o crash do programa consoante a escolha elevada de pontos para um certo plano, pois como decidimos evitar a geração de três pontos consecutivos, consoante a escolha random dos pontos pode chegar a um estado em que nenhum dos espaços abertos gera um ponto não-consecutivo aos já guardados.

5. Distribuição do trabalho:

Este trabalho foi efetuado em partes equivalentes entre ambos os elementos. Procuramos dividir o melhor possível os vários métodos pedidos e fomos juntando o progresso de cada um ao longo do tempo corrigindo qualquer erro de conexão de ambos os códigos (normalmente problemas de input). O relatório foi também dividido igualmente consoante a escolha de cada membro.

Durante o desenvolvimento deste projeto não recorremos a mais nenhum código (seja de colegas ou recursos online) exceto a pseudocódigos presente nas plataformas da cadeira e de material de anos anteriores.