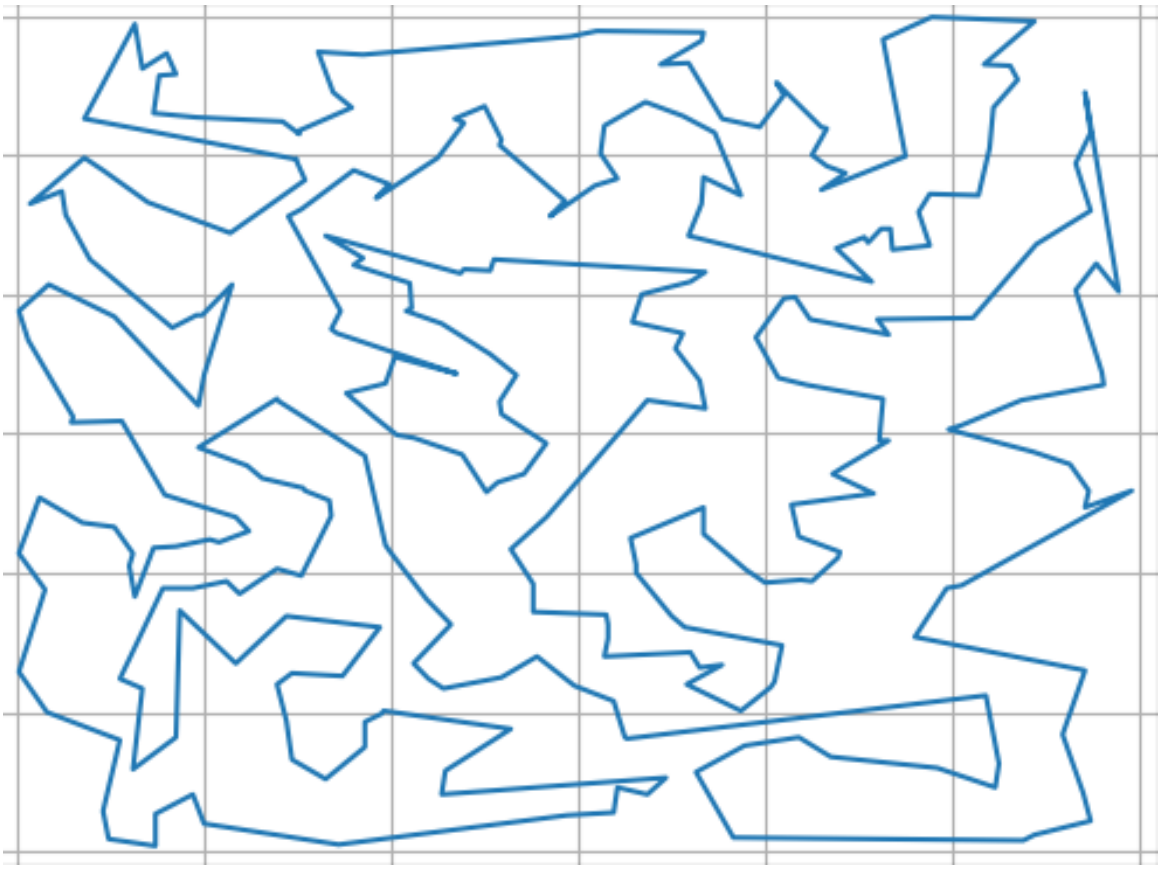
U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Criação de um polígono simples a partir de um conjunto de n pontos no plano

Unidade curricular: Inteligência Artificial CC2006

Regente da cadeira: Ana Paula Tomás



Grupo 31

Catarina Ferreira Teixeira up201805042 – L:CC

Patrícia Daniela Tavares Vieira up201805238 – MI:ERSI

ABRIL 2020

Introdução ao problema

Com base num problema bem conhecido no mundo da algoritmia, *"Traveling Salesman Problem"* - TSP, foi-nos proposto criar um polígono simples e fechado. Como o ciclo de Hamilton o nosso grafo tem que ser um percurso fechado e que não repete nós e passe em todos os nós.

O nosso programa, para a tentativa de resolução deste problema, foi desenvolvido em Java e a visualização (gráfica) de polígonos construídos foi desenvolvido em **Python**.

O objetivo do programa é criar um polígono simples com **n** pontos (**n** é dado pelo utilizador), sendo que cada ponto contém coordenadas inteiras entre o limite de **–m** e **m** (**m** é o range dado pelo utilizador). Para criar as arestas/ligações entre pontos, temos 2 alternativas: combinar as coordenadas aleatoriamente ou usando a heurística *"nearest-neighbour first"* a partir de um ponto inicial random.

Depois da criação do polígono vamos estudar métodos de pesquisa local e de pesquisa estocástica onde um polígono simples não poderá ter arestas que se intersetem sem ser nos seus vértices. Para estudar as interseções usamos o método **2-exchange** para o descruzamento de arestas de 1ºgrau, gerando uma lista de polígonos possíveis.

De seguida vamos analisar várias alternativas usando o melhoramento iterativo, *"hill climbing"*, para escolher o melhor candidato para 4 critérios diferentes: *"best-improvement first"*, *"first-improvement"*, menos conflitos de arestas e um candidato aleatório. Estes critérios têm como meio de comparação o perímetro do polígono.

Também foi usado o algoritmo *"simulating annealing"* que otimiza essa procura da solução ótima a partir do perímetro mínimo da busca local probabilística, tendo como analogia a termodinâmica.

Para visualização dos gráficos foi usada um programa em **python** que dado o range e as coordenadas, vai representar graficamente o nosso polígono.

Como Compilar o Programa

Para compilar o código tem que utilizar um terminal e estar do local do ficheiro e executar o comando: **javac RPG.java && java RPG**. Depois de executar o comando vai aparecer o menu principal do programa.

Para conseguir visualizar o gráfico, temos de compilar um programa **Python** chamada **Pontos.py,** dando o range que queremos e as coordenadas e compila-se da seguinte forma:

python3 Pontos.py

Para conseguir executar tem de ter instalado o **matplotlib** no **python** para não haver problemas, já que o programa utiliza esta biblioteca para fazer os gráficos.

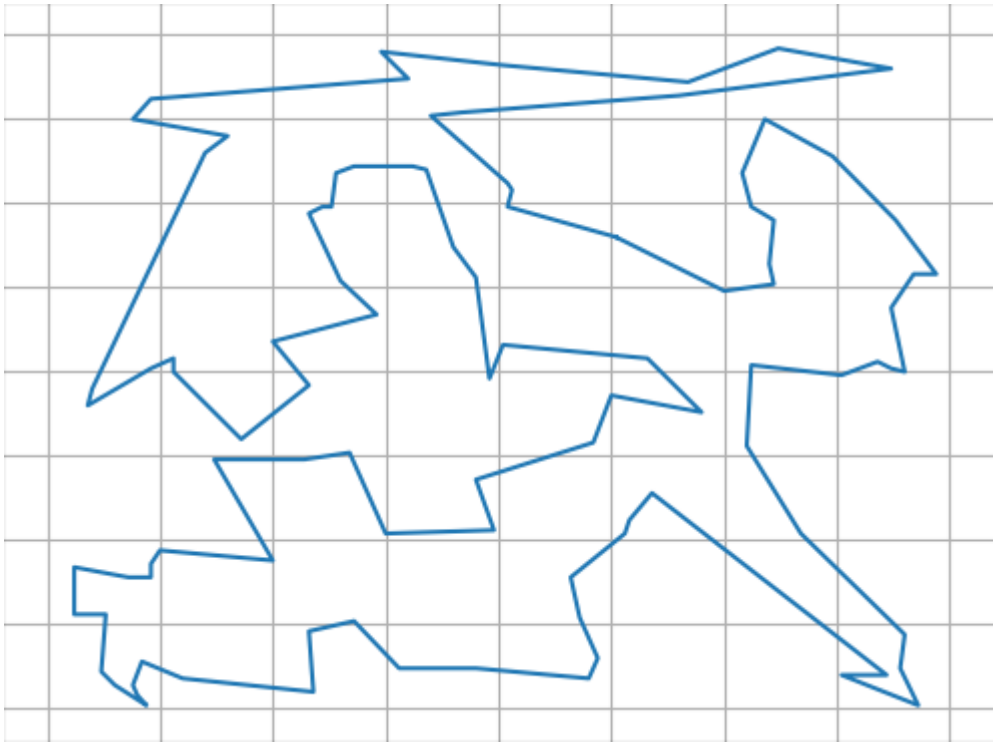


Figura 1-Exemplo de um polinómio sem conflitos

Visualização do Programa

Este programa inicia com um menu informativo, cada valor corresponde ao número do exercício de avaliação. Existem também menus secundários: no exercício 2 para escolher qual a permutação a escolher e no exercício 4 para optar pelo critério de escolha.

Caso inicie a execução do programa e não escolher a opção 1 (corresponde a criação dos pontos), o programa automaticamente pede para inserir os valores pedidos (nº pontos e o range).

No final de executar cada exercício tem um menu secundário em que dá a opção de continuar a executar o programa ou sair.

```
Trabalho 1- IA

1 - Gerar aleatoriamente pontos no plano
2 - Determinar um candidato a solução
3 - Determinar a vizinhança obtida por (2-exchange)
4 - Aplicar melhoramento iterativo (hill climbing)
5 - Aplicar simulated annealing
NOTA: Caso pretenda criar um novo array, volte a escolher o ex.1!
      Caso pretenda alterar as ligações do array, volte a escolher o ex. 2!
Escolha o exercicio:
```

Figura 2- Menu Principal

```
Ex2:

Escolha uma das seguintes alternativas para criar ligações:
1-Gerar uma permutação qualquer dos pontos.
2-Heuristica 'nearest-neighbour first'
```

Figura 3- Menu do exercício 2

```
Pretende visualizar a lista de candidatos?
0-Sim, Outro-Não
```

Figura 4- Menu do exercício 3 (Mostrar ou não a lista)

```
Ex4:

Escolha uma das seguintes alternativas para escolher o candidato na vizinhança "2-exchange":
1-Minimo Perímetro - 'best-improvement first'
2-Primeiro candidato nessa vizinhança - 'first-improvement'
3-Menos Conflitos de arestas - menos cruzamentos de arestas
4-Qualquer candidato nessa vizinhaça
```

Figura 5 - Menu do exercício 4

Estrutura do Código do Programa

No ficheiro RPG.java, o programa está dividido em 3 classes: a classe que corresponde ao main, a classe **Grafo** e a classe **Reta**.

A classe **Grafo** foi criada com o intuito de armazenar todos os dados relativo ao polígono. Tem dados/argumentos como: o tamanho do **array**, o array de pontos correspondentes ao polígono (**arrayC**), o array auxiliar para guardar pontos nos exercícios 4 e 5 (**bestSoFar**) e uma lista de Retas (classe **Reta**) para utilizar lista auxiliar para os métodos de pesquisa estocástica. Para além das variáveis e o método construtor, contem também funções de manipulação ou impressão do Grafo.

A classe **Reta** vai ser utilizada para a lista de array de pontos, onde tem um método em que transforma o conteúdo da lista em arrays de pontos. Esta classe tem como função principal auxiliar o armazenamentos de dados importantes para a resolução de alguns exercícios.

```
class Grafo{
    int tamanho; // numero de nos no grafo
    Point2D[] arrayC; //array aonde vao ficar as coordenadas
    Point2D[] best_so_far; //usado no hill climbing para determinar o melhor
    LinkedList<Reta> lista;

    Grafo(int tamanho){
        this.tamanho=0;
        this.arrayC = new Point2D[tamanho];
        this.best_so_far= new Point2D[tamanho];
        this.lista= new LinkedList<>();

    }
```

Figura 6 - Declaração das variáveis e construtor da classe Grafo

```
class Reta{
    public Point2D[] x;

    Reta(Point2D[] a){
        x=a.clone();
    }

    //pega na nossa reta e transforma-a para um array de pontos
    Point2D[] toarray(){
        return x;
    }
}
```

Figura 7 - Código relativo a class Reta

Usamos o **Poitn2D** para facilitar o uso de coordenadas que já existiam numa biblioteca java e também para usufruir das suas funções já implementadas (**distance, clone, etc)**. Nesta classe objeto as coordenadas do ponto são do tipo **double**, mas para solução convertemos para valores inteiros.

Usamos como forma de armazenar os dados, arrays para facilitar a troca de posições (ligação dos pontos/criação dos pontos).

Exercício 1

Neste exercício a partir de **n** pontos e de um range de **–m** a **m**, valores dados pelo utilizador, geramos n coordenadas aleatórias no plano.

Depois de ler os valores dados, com base neles, criamos um grafo e de seguida usando o método **criacaoPontos(n,m)** da classe **Grafo**. Neste método criamos um **random seed** (a partir da biblioteca **Random** do java) que vai gerar os números aleatórios inteiros que convertermos para **doubles,** para as coordenadas dos pontos **(x e y)**. Para que os pontos respeitassem os limite/range imposto pelo utilizador (**[-m,m]**), usamos a seguintes formula **(max - min) + min**, fazendo a mudança para os nossos valores a fórmula fica **(2*m) -m**.

Como não podemos ter duas coordenadas iguais na geração, utilizamos uma função que verifica se o ponto gerado nos valores **random** já existe ou não, retornando um booleano, **verificarPontos(double x, double y)**. Ou seja, se o ponto ainda não existir adicionamos ao array de pontos como um **Point2D.Double**, e aumentamos o tamanho, e repetimos até termos o número de nós desejado e pedido pelo utilizador.

```
//Ex1-Funçao geradora de pontos random
void criacaoPontos(int n,int m){
    long startTime = System.currentTimeMillis();

    double x,y;
    int indice=0;
    Random seed= new Random();
    while(indice<n){
        //int number = random.nextInt(max - min) + min;
        x=(double)seed.nextInt(2*m)-m;
        y=(double)seed.nextInt(2*m)-m;
        if(!verificarPontos(x,y)){ //Vai verificar se existe pontos repetidos
            arrayC[tamanho++]= new Point2D.Double(x,y);
            indice++;
        }
    }
    System.out.println("Novo Array de pontos: ");
    printArrayPontos();
    long endTime = System.currentTimeMillis();
    System.out.println("\nDemorou " + (endTime - startTime) + " milisegundos");
}

//Ex1-Verifica se os pontos já existem no array
boolean verificarPontos(double x, double y){
    if(this.tamanho==0) return false;
    for(int i=0;i<this.tamanho;i++){
        if(x==arrayC[i].getX() && y==arrayC[i].getY())
            return true;
    }
    return false;
}
```

Figura 8 - Código da função criacaoPontos

```
Quantidade de pontos no plano: 10
Insira o range desejado: 20
Novo Array de pontos:
(-6,-17)(-20,-6)(-17,3)(15,-2)(4,-7)(-10,-9)(-7,14)(5,3)(-15,-8)(3,-13)

Demorou 11 milisegundos
```

Figura 9 - Exemplo 1 de criação de pontos

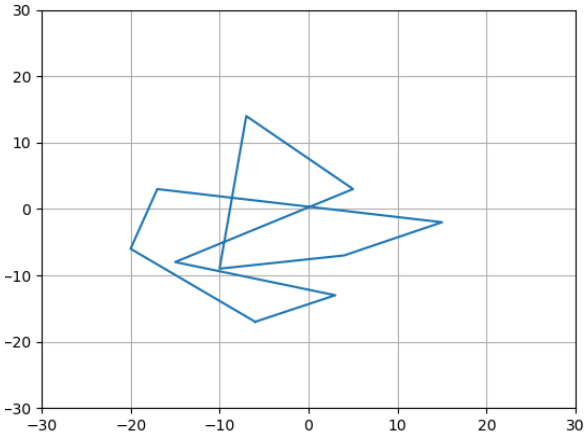


Figura 10 - Representação gráfica do exemplo 1

```
Quantidade de pontos no plano: 30
Insira o range desejado: 20
Novo Array de pontos:
(-19,14)(-8,18)(3,-1)(-9,-7)(2,-7)(11,7)(-12,-13)(15,-1)(-5,-10)(14,4)(-4,-14)(-8,-11)(4,11)(13,-13)(2,16)(1,-1)(
-7,-4)(-12,0)(0,-11)(-11,2)(15,18)(10,3)(-16,-14)(-17,-8)(8,7)(-16,-12)(-7,-6)(5,-8)(-3,-16)(17,13)

Demorou 1 milisegundos
```

Figura 12 - Exemplo 2 de criação de pontos

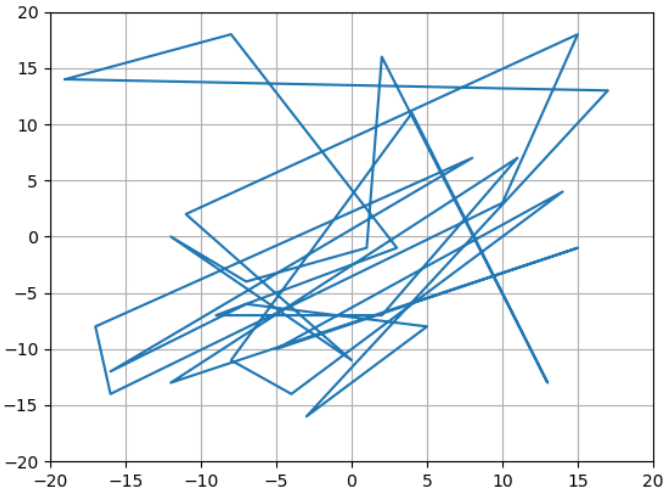


Figura 11 - Representação gráfica do exemplo 2

Exercício 2

Usando o array de pontos que criamos no exercício anterior, neste exercício determinamos um candidato a solução de duas formas diferentes (criando uma aresta/ligação entre pontos):

1) Gerar uma permutação qualquer dos pontos

Para fazer uma permutação dos pontos, como a escolha é aleatória, criamos um número random em que enquanto vamos percorrendo o nosso array de pontos, geramos sempre um número random entre 0 e o tamanho do nosso array e trocávamos o ponto que estava no índice **i** com o índice do nosso número gerado.

```
//Ex2.1-Permutação de pontos
void permutation(){
    long startTime = System.currentTimeMillis();
    System.out.println("Array Original: ");
    printArrayPontos();

    System.out.println("\nPermutação de pontos: ");
    Random number= new Random();
    for(int i=1;i<this.tamanho;i++){
        int swap= number.nextInt(this.tamanho);
        Point2D aux= arrayC[swap];
        arrayC[swap]=arrayC[i];
        arrayC[i]=aux;
    }
    printArrayPontos(); //NOVA ORDEM
    long endTime = System.currentTimeMillis();
    System.out.println("\nDemorou " + (endTime - startTime) + " milisegundos");
}
```

Figura 13 - Código da função permutaion

2) Usando heurística “nearest-neighbour first”

Para a nossa heurística *“nearest-neighbour first”* queremos gerar um polígono em que verifica a partir de um ponto inicial, escolhido aleatoriamente, o ponto com a distância mais próxima.

Então inicialmente escolhemos um ponto para a posição inicial, gerado aleatoriamente. Depois de realizar a troca da posição 0, vamos percorrer o nosso array original, através de um ciclo com o índice **j**, e vamos procurando a distância mínima. Depois de comparar com todos os valores do array, colocamos o ponto que encontramos com distancia mínima, no lado esquerdo do array (adaptação do algoritmo **selection sort**).

```
//Ex2.2-Nearest-neighbour first
void nnf(){
    long startTime = System.currentTimeMillis();
    System.out.println("Array Original: ");
    printArrayPontos();
    System.out.println("\nNearest-neighbour first: ");

    int noInicial;
    Random number= new Random();
    noInicial=number.nextInt(this.tamanho); //Escolhe o nó inicial
    Point2D aux=new Point2D.Double(); //Para ajudar na troca de posiçoes
    double  minDist; //Guarda a distancia minima
    int indicemin=0,cont;

    //Fazer a troca da posição inicial
    if(noInicial!=0){
        aux=arrayC[0];
        arrayC[0]=arrayC[noInicial];
        arrayC[noInicial]=aux;
    }

    for(int j=0; j<this.tamanho-1;j++){
        cont=j+1;
        minDist=arrayC[cont].distance(arrayC[j]); //distancia minima inicial
        indicemin=cont; //guarda o indice do valor minimo

        cont++;
        while(cont<this.tamanho){
            if(arrayC[j].distance(arrayC[cont]) < minDist){
                minDist=arrayC[j].distance(arrayC[cont]);
                indicemin=cont;
            }
            cont++;
        }
        //organizar o array, minimo passa para o lado esquerdo (i+1)
        Point2D a = arrayC[indicemin];
        arrayC[indicemin]=arrayC[j+1];
        arrayC[j+1]=a;
    }
    printArrayPontos(); //NOVA ORDEM
    long endTime = System.currentTimeMillis();
    System.out.println("\nDemorou " + (endTime - startTime) + " milisegundos");
}
```

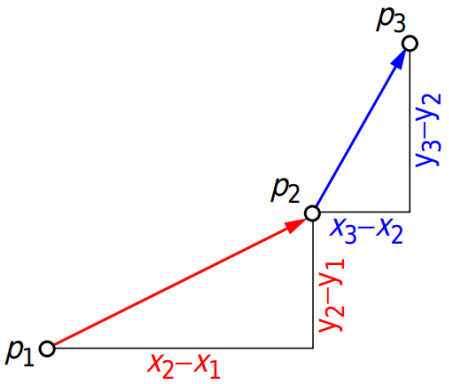
Figura 14 - Código da função nnf

Exercício 3

Neste exercício para um candidato **s** (array), temos que determinar a vizinhança obtida por *"2-exchange"*. Para isso, temos de ter em conta o cruzamento de segmentos, que vai corresponder ao número de elementos na lista que vai ser criada para guardar todos os novos candidatos.

A vizinhança obtida vai ser uma lista de filhos com descruzamento de arestas de grau 1, isto é, quando só é feita um só descruzamento de arestas.

Nós primeiramente temos de ter em caso a orientação do polígono, ou seja, se nós tivermos dois segmentos em que sejam os dois com direções diferentes, então aí sabemos que poderá haver interseção. Isso vamos verificar com a função de declive.



Declive do segmento **(p1,p2)**: **σ = (y2−y1) / (x2−x1)**
Declive do segmento **(p2,p3)**: **τ = (y3−y2) / (x3−x2)**
Ou seja, a fórmula para descobrir a orientação disto seria:

(y2−y1) (x3−x2) − (y3−y2) (x2−x1)

Se o resultado for 0 quer dizer que vai ser colinear;
Se o resultado for >0 quer dizer que vai ser no sentido horário;
Se o resultado for <0 quer dizer que vai no sentido contra-horário.

```
//Ex3 e 4-Encontrar a orientação do trio ordenado (p, q, r).
static double orientation(Point2D p, Point2D q, Point2D r){
    double val = ((q.getY()-p.getY()) * (r.getX()-q.getX()) - (q.getX()-p.getX()) * (r.getY()-q.getY()));

    if (val == 0) return 0; //p, q e r são colineares
    return (val > 0)? 1: 2; //1-sentido horário ou 2-sentido anti-horário
}
```

Figura 15 - Código da função orientation

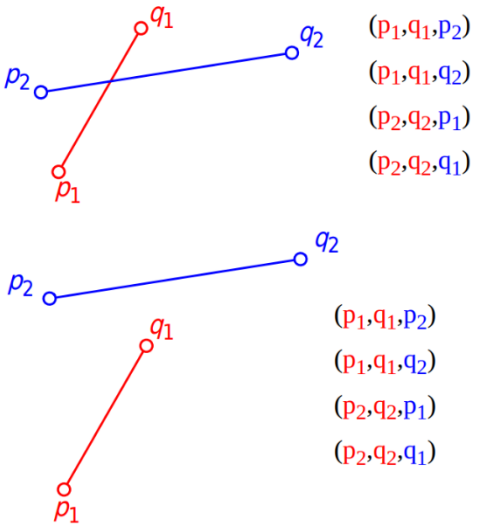


Figura 16 – Caso geral de orientações

Caso geral:

- **(p1,q1,p2)** e **(p1,q1,q2)** têm orientações diferentes.
- **(p2,q2,p1)** e **(p2,q2,q1)** têm orientações diferentes.

OU

- **(p1,q1,p2)** e **(p1,q1,q2)** têm orientações diferentes.
- **(p2,q2,p1)** e **(p2,q2,q1)** têm orientações diferentes.

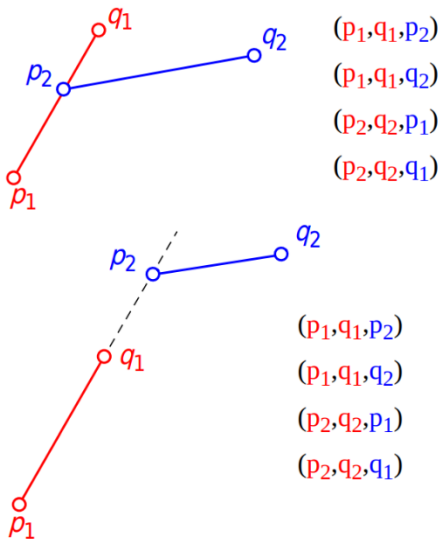


Figura 17 – Casos especiais de colineares e interseção

Casos especiais:

Os casos especiais incidem sobre quando os pontos são colineares.

- **(p1,q1,p2), (p1,q1,q2), (p2,q2,p1)**
- **(p2,q2,q1)** são todos colineares.

E

- As produções x de **(p1,q1)** e **(p2,q2)** intersetam.
- As produções y de **(p1,q1)** e **(p2,q2)** intersetam.

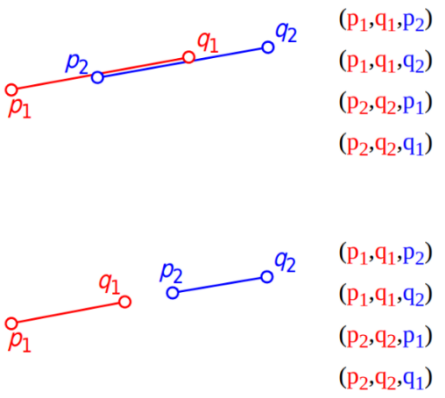


Figura 18 – Casos especiais de colineares

Para descobrir se existe pontos colineares temos uma função **onSegment** que dado 3 pontos, vai verificar se um ponto está "em cima" de um segmento:

```
//Ex3 e 4-Verifica se um ponto esta contido numa segmentos
static boolean onSegment(Point2D p, Point2D q, Point2D r){
    if (q.getX() <= Math.max(p.getX(), r.getX()) &&
        q.getX() >= Math.min(p.getX(), r.getX()) &&
        q.getY() <= Math.max(p.getY(), r.getY()) &&
        q.getY() >= Math.min(p.getY(), r.getY()))
        return true;
    return false;
}
```

Figura 19 - Código da função onSegment

Quando temos pontos colineares desta forma só podemos aceitar a remoção se o seu produto escalar for maior que 0. Para isso definimos uma função que vai calcular o produto escalar e retornar o resultado desse cálculo e depois na nossa função principal fazemos os casos relativamente a isso.

```
//Ex3 e 4-Produto Vetorial
double produtoVet(Point2D p1, Point2D q1, Point2D p2, Point2D q2){
    Point2D x= new Point2D.Double(q1.getX()-p1.getX(),q1.getY()-p1.getY());
    Point2D y= new Point2D.Double(q2.getX()-p2.getX(),q2.getY()-p2.getY());
    return (x.getX()*y.getX()) + (x.getY()*y.getY());
}
```

Figura 20 - Código da função produtoVet

Estando esta parte esclarecida, passamos ao código onde juntamos as condições todas para verificar todas as interseções que nos interessa:

```
//Ex3 e 4-Verifica se os segmentos se intersetam
boolean intersecao(Point2D p1, Point2D q1, Point2D p2, Point2D q2) {
    //indica as orientações dos segmentos
    double o1 = orientation(p1, q1, p2);
    double o2 = orientation(p1, q1, q2);
    double o3 = orientation(p2, q2, p1);
    double o4 = orientation(p2, q2, q1);

    if(o1!=o2 && o3!=o4) return true;

    if((o1==0 &&o2==0 && o3==0 &&o4==0) && produtoVet(p1,q1,p2,q2)>0)   return true;

    // p1, q1 e p2 sao colineares e p2 é colinear com p1q1
    if (o1 == 0 && onSegment(p1, p2, q1)) return true;

    // p1, q1 e q2 sao colineares e q2 é colinear com p1q1
    if (o2 == 0 && onSegment(p1, q2, q1)) return true;

    // p2, q2 e p1 sao colineares e p1 é colinear com p2q2
    if (o3 == 0 && onSegment(p2, p1, q2)) return true;

    // p2, q2 e q1 sao colineares e q1 é colinear com p2q2
    if (o4 == 0 && onSegment(p2, q1, q2)) return true;

    return false;
}
```

Figura 21 - Código da função intersecao

Explicado o contexto das interseções, passamos agora para a explicação da troca dos pontos e das novas ligações.

A função **exchange** recebe um argumento, este argumento serve para indicar ao programa qual o array vamos usar para fazer esta implementação. No caso do exercício 3, vamos usar o array original (opção **1**) para procurar os seus candidatos, posteriormente no exercício 4 e 5 vamos usar o array **bestSoFar** (opção **2**). Decidimos fazer desta maneira, pois nos exercícios seguintes iriamos usar o mesmo algoritmo, só que o array inicial (array de comparação) é diferente.

Depois de termos o array inicial (“array pai”), e vamos percorrê-lo através de 2 ciclos para procurar qualquer tipo de interseção entre 2 retas (4 pontos). A inicialização do segundo ciclo **(int j)** é lhe sempre atribuído mais 1 à variável do primeiro ciclo **(j=i+1)**, assim não temos posições do array/arestas como 0<->1 e 1<->2, pois estas retas não se podem intersetar.

Como a ordem do array, determina as suas ligações, a última posição tem ligação com o primeiro elemento (aresta 0 <-> tamanho-1), como os ciclos são relacionados, seria impossível chegar a estes valores, logo criamos uma verificação, em que alteramos os valores iniciais.

Depois fazemos uma verificação dos índices, para não termos duas retas com os mesmos pontos entre si, verificamos se existe ou não interseção dessas duas retas.

Se houver interseção vamos trocar os índices **(b e i)** e inverte os valores desde o índice mínimo até ao índice máximo (**b<i ou i>b**), através da função reverse e depois adicionamos o novo candidato a lista que armazena os candidatos. Por exemplo, se tivermos 0<->1<->2<->3<->4<->5 e tivermos interseção entre (0,1) e (4,5), vamos trocar o 1 até 4, então no final iríamos ficar com 0<->4<->3<->2<->1<->5.

```
//Ex3 e 4-Reverte o restante array depois do exchange
Point2D[] reverse(int i,int a, Point2D[] novoarray){
    for(int j=i;j<a;j++){
        Point2D aux= novoarray[j];
        novoarray[j]=novoarray[a];
        novoarray[a]=aux;
        a--;
    }
    return novoarray;
}
```

Figura 22 - Código da função reverse

Ao chamar a função **reverse** temos de mandar o clone do nosso array de pontos, porque nós queremos preservar o array original para continuar a avaliar os cruzamentos de arestas para depois armazenar todos os possíveis filhos.

```
//Ex3 e 4-Determinar a vizinhança obtida por (2-exchange)
void exchange(int op){
    lista.clear();
    Point2D[] nvarry;

    //De acordo com as opçoes copia os valores do array a escolher
    if(op==1)
        nvarry=arrayC.clone();
    else
        nvarry=bestSoFar.clone();

    //pontos da reta 1: (i-1)<->(i)
    //pontos da reta 1: (b)<->(a)
    Point2D[] novoarray;
    int a=0,b=0;
    for(int i=1;i<this.tamanho-2;i++){
        for(int j=i+1;j<this.tamanho;j++){
            //Caso chegue a ultima reta, ultimo elemento com o primeiro (tamanho-1 -> 0)
            if(j+1==this.tamanho){
                a=0;
                b=this.tamanho-1;
            }
            else{
                a=j+1;
                b=j;
            }

            //pontos da reta1 e da reta2 têm que ser diferentes
            if(b!=(i-1) && b!=i && a!=i && a!=(i-1)){
                if(intersecao(nvarry[i-1],nvarry[i],nvarry[b],nvarry[a])){
                    //Imprime as trocas
                    //System.out.print("\n \n("+(i-1)+","+i+") -> (" +b+","+ a +")\n");
                    //System.out.print(lista.size()+": ");
                    //System.out.print("("+(int)nvarry[i].getX()+","+(int)nvarry[i].getY()+")");
                    //System.out.println("->("+(int) nvarry[b].getX()+","+(int) nvarry[b].getY()+")\n");

                    if(b<i)
                        novoarray=reverse(b,i,nvarry.clone());
                    else
                        novoarray=reverse(i,b,nvarry.clone());

                    lista.addLast(new Reta(novoarray));
                }
            }
        }
    }

}
```

Figura 23 - Código da função exchange

```
Vizinhança 2-exchange:
0: (-6,-17)  (-20,-6)  (5,3)  (3,-13)  (-17,3)  (15,-2)  (-10,-9)  (4,-7)  (-7,14)  (-15,-8)
1: (-6,-17)  (3,-13)  (-20,-6)  (5,3)  (-17,3)  (15,-2)  (-10,-9)  (4,-7)  (-7,14)  (-15,-8)
2: (-6,-17)  (-17,3)  (3,-13)  (-20,-6)  (5,3)  (15,-2)  (-10,-9)  (4,-7)  (-7,14)  (-15,-8)
3: (-6,-17)  (15,-2)  (-17,3)  (3,-13)  (-20,-6)  (5,3)  (-10,-9)  (4,-7)  (-7,14)  (-15,-8)
4: (-6,-17)  (-10,-9)  (15,-2)  (-17,3)  (3,-13)  (-20,-6)  (5,3)  (4,-7)  (-7,14)  (-15,-8)
5: (-6,-17)  (4,-7)  (-10,-9)  (15,-2)  (-17,3)  (3,-13)  (-20,-6)  (5,3)  (-7,14)  (-15,-8)
6: (-6,-17)  (5,3)  (3,-13)  (-20,-6)  (-17,3)  (15,-2)  (-10,-9)  (4,-7)  (-7,14)  (-15,-8)
7: (-6,-17)  (5,3)  (-17,3)  (3,-13)  (-20,-6)  (15,-2)  (-10,-9)  (4,-7)  (-7,14)  (-15,-8)
8: (-6,-17)  (5,3)  (4,-7)  (-10,-9)  (15,-2)  (-17,3)  (3,-13)  (-20,-6)  (-7,14)  (-15,-8)
9: (-6,-17)  (5,3)  (-7,14)  (4,-7)  (-10,-9)  (15,-2)  (-17,3)  (3,-13)  (-20,-6)  (-15,-8)
10: (-6,-17)  (5,3)  (-20,-6)  (-7,14)  (4,-7)  (-10,-9)  (15,-2)  (-17,3)  (3,-13)  (-15,-8)
11: (-6,-17)  (5,3)  (-20,-6)  (3,-13)  (15,-2)  (-17,3)  (-10,-9)  (4,-7)  (-7,14)  (-15,-8)
12: (-6,-17)  (5,3)  (-20,-6)  (3,-13)  (-10,-9)  (15,-2)  (-17,3)  (4,-7)  (-7,14)  (-15,-8)
13: (-6,-17)  (5,3)  (-20,-6)  (3,-13)  (-7,14)  (4,-7)  (-10,-9)  (15,-2)  (-17,3)  (-15,-8)
14: (-6,-17)  (5,3)  (-20,-6)  (3,-13)  (-17,3)  (4,-7)  (-10,-9)  (15,-2)  (-7,14)  (-15,-8)
15: (-6,-17)  (5,3)  (-20,-6)  (3,-13)  (-17,3)  (-7,14)  (4,-7)  (-10,-9)  (15,-2)  (-15,-8)
16: (-6,-17)  (5,3)  (-20,-6)  (3,-13)  (-17,3)  (15,-2)  (4,-7)  (-10,-9)  (-7,14)  (-15,-8)

Números de candidatos da lista: 17

Demorou 4 milisegundos
```

Figura 24 - Exemplo 1 do exercício 3

```
99: (-19,14)  (-8,18)  (3,-1)  (-9,-7)  (2,-7)  (11,7)  (-12,-13)  (15,-1)  (-5,-10)  (14,4)  (-4,-14)  (-8,-11)
  (4,11)  (13,-13)  (2,16)  (1,-1)  (-7,-4)  (-12,0)  (0,-11)  (-11,2)  (15,18)  (17,13)  (-3,-16)  (5,-8)  (-7,-6
)  (-16,-12)  (8,7)  (-17,-8)  (-16,-14)  (10,3)
100: (-19,14)  (-8,18)  (3,-1)  (-9,-7)  (2,-7)  (11,7)  (-12,-13)  (15,-1)  (-5,-10)  (14,4)  (-4,-14)  (-8,-11)
  (4,11)  (13,-13)  (2,16)  (1,-1)  (-7,-4)  (-12,0)  (0,-11)  (-11,2)  (15,18)  (10,3)  (-7,-6)  (-16,-12)  (8,7
)  (-17,-8)  (-16,-14)  (5,-8)  (-3,-16)  (17,13)
101: (-19,14)  (-8,18)  (3,-1)  (-9,-7)  (2,-7)  (11,7)  (-12,-13)  (15,-1)  (-5,-10)  (14,4)  (-4,-14)  (-8,-11)
  (4,11)  (13,-13)  (2,16)  (1,-1)  (-7,-4)  (-12,0)  (0,-11)  (-11,2)  (15,18)  (10,3)  (-16,-14)  (-17,-8)  (8,
7)  (-16,-12)  (-7,-6)  (-3,-16)  (5,-8)  (17,13)

Números de candidatos da lista: 102

Demorou 17 milisegundos
```

Figura 25 - Exemplo 2 do exercício 3

Exercício 4

O **hill climbling** é uma técnica de otimização matemática que é um tipo de pesquisa local. É um algoritmo iterativo que começa com uma solução arbitrária do problema e depois tenta encontrar uma solução melhor mudando incrementalmente o elemento único da solução. Como vamos ter 4 opções de candidato fizemos uma função opção em que damos como argumento um inteiro sendo que esse inteiro vai representar uma das nossas 4 opções:

1. best-improvement first;
2. first-improvement;
3. menos cruzamentos de arestas
4. qualquer candidato;

```
//Ex4 e 5-Opçoes para o hill climbing
Point2D[] opcao(int op){
    int indiceMin;
    switch(op){
        case 1: //minimo perimetro
                indiceMin=0;
                double pBest=Double.MAX_VALUE;
                int pos=0; //guarda posiçao

                for(Reta cand : lista){ //para cada posiçao da lista
                    double pCand=perimetro(cand.toarray());//calculamos o perimetro
                    if(pCand<pBest){//se for menor, troca e muda a posiçao para computar a metrica
                        pBest=pCand;
                        pos=indiceMin;//computa a metrica
                    }
                    indiceMin++;//senao avança
                }
                return lista.remove(pos).toarray(); //retiramos da lista para ser avaliado

        case 2: return lista.removeFirst().toarray(); //retiramos o primeiro elemento da lista

        case 3: //retira o elemento da lista com menos conflitos
                int cont=0 , iBest= Integer.MAX_VALUE, iCand=0;
                indiceMin=0;

                for(Reta cand : lista){
                    Point2D[] b= cand.toarray();
                    iCand=inter(b);
                    if(iCand<iBest){
                        iBest=iCand;
                        indiceMin=cont;
                    }
                    cont++;
                }
                return lista.remove(indiceMin).toarray();

        case 4: //retiramos um random da lista para ser avaliado
                Random s= new Random();
                return lista.remove(s.nextInt(lista.size())).toarray();
        default: return null;

    }

}
```

Figura 26 - Código da função opcao

1) Best-improvement first

Para esta alínea, chamamos a função **hillClimbing** com a opção **1** (**hillClimbing(1)**).

Temos de aplicar o **best-improvement first** (candidato que reduz mais o perímetro). Para isso na função **opcao** com o argumento **1**, percorremos a lista através de um ciclo for each que vai percorrer cada elemento da lista e vai calcular o respetivo perímetro (**pCand**) e verificar com o perímetro mínimo guardado (**pBest**).

Quando acaba de percorrer a lista, retornamos o elemento que foi retirado da lista na posição (**pos**) que foi descoberto o perímetro menor.

```
Original:
(-6,-17)(5,3)(-20,-6)(3,-13)(-17,3)(15,-2)(-10,-9)(4,-7)(-7,14)(-15,-8)

Array Solução:
(-6,-17)(3,-13)(4,-7)(15,-2)(5,3)(-7,14)(-17,3)(-10,-9)(-20,-6)(-15,-8)
Perimetro: 112.78573290345877


Demorou 19 milisegundos
Iterações do programa: 5
```

Figura 27 - Exemplo 1 com o exercício 4.1

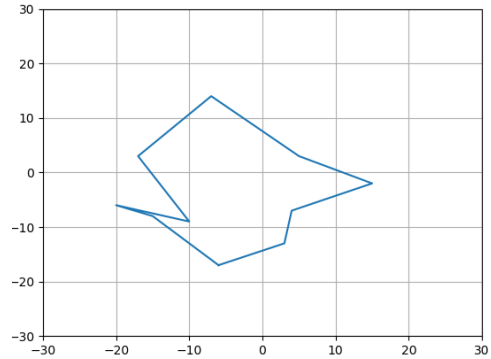


Figura 28 - Representação gráfico do exemplo 1 no exercicio 4.1

```
Original:
(-19,14)(-8,18)(3,-1)(-9,-7)(2,-7)(11,7)(-12,-13)(15,-1)(-5,-10)(14,4)(-4,-14)(-8,-11)(4,11)(13,-13)(2,16)(1,-1)(-7
,-4)(-12,0)(0,-11)(-11,2)(15,18)(10,3)(-16,-14)(-17,-8)(8,7)(-16,-12)(-7,-6)(5,-8)(-3,-16)(17,13)

Array Solução:
(-19,14)(-8,18)(2,16)(11,7)(10,3)(15,18)(17,13)(14,4)(15,-1)(13,-13)(5,-8)(2,-7)(0,-11)(-3,-16)(-4,-14)(-12,-13)(-1
6,-14)(-17,-8)(-16,-12)(-9,-7)(-12,0)(-7,-4)(-7,-6)(-8,-11)(-5,-10)(3,-1)(1,-1)(8,7)(4,11)(-11,2)
Perimetro: 229.35451864846488


Demorou 5 milisegundos
Iterações do programa: 19
```

Figura 30 - Exemplo 2 com o exercício 4.1

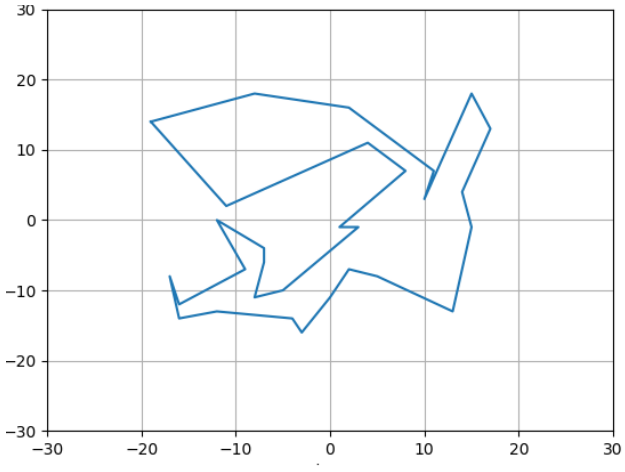


Figura 29 Representação gráfico do exemplo 2 no exercício 4.1

2) First-improvement

Para esta alínea, chamamos a função **hillClimbing** com a opção **2** (**hillClimbing(2)**).

Queremos o primeiro candidato da lista, basta retornar o primeiro elemento que foi retirado da lista.

```
Original:
(-19,14)(-8,18)(3,-1)(-9,-7)(2,-7)(11,7)(-12,-13)(15,-1)(-5,-10)(14,4)(-4,-14)(-8,-11)(4,11)(13,-13)(2,16)(1,-1)(-7
,-4)(-12,0)(0,-11)(-11,2)(15,18)(10,3)(-16,-14)(-17,-8)(8,7)(-16,-12)(-7,-6)(5,-8)(-3,-16)(17,13)

Array Solução:
(-19,14)(-8,18)(2,16)(-7,-4)(-16,-12)(-12,-13)(-4,-14)(5,-8)(10,3)(2,-7)(3,-1)(0,-11)(-5,-10)(-8,-11)(-9,-7)(-7,-6)
(1,-1)(4,11)(8,7)(11,7)(15,18)(17,13)(15,-1)(14,4)(13,-13)(-3,-16)(-16,-14)(-17,-8)(-12,0)(-11,2)
Perimetro: 280.3382394362971


Demorou 6 milisegundos
Iterações do programa: 34
```

Figura 33 - Exemplo 1 no exercício 4.2

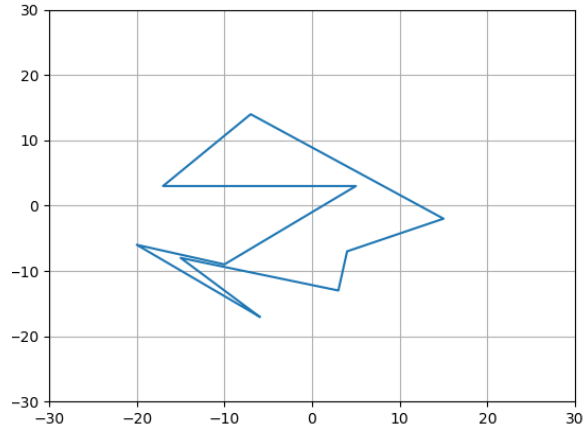


Figura 32- Representação gráfica do exemplo 1 no exercício 4.2

```
Original:
(-6,-17)(5,3)(-20,-6)(3,-13)(-17,3)(15,-2)(-10,-9)(4,-7)(-7,14)(-15,-8)

Array Solução:
(-6,-17)(-20,-6)(-10,-9)(5,3)(-17,3)(-7,14)(15,-2)(4,-7)(3,-13)(-15,-8)
Perimetro: 161.0984550582834


Demorou 1 milisegundos
Iterações do programa: 7
```

Figura 34 -Exemplo 2 no exercício 4.2

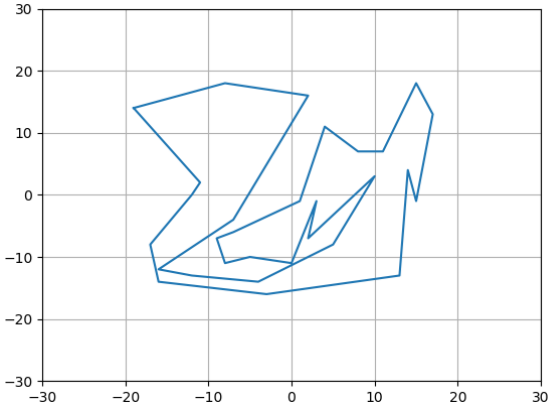


Figura 31 - Representação gráfica do exemplo 2 no exercício 4.2

3) Menos cruzamentos de arestas

Para esta alínea, chamamos a função **hillClimbing** com a opção **3** (**hillClimbing(3)**).

Precisamos devolver o elemento da lista com menos conflitos, ou seja, com menos interseções das retas. Para calcular o número de interseções criamos uma função **inter** que recebe como argumento um array de pontos, que percorrer esse array e incrementar o contador e no final retorna esse contador, usamos também uma função que já nos referimos acima **(interseções)** para auxiliar-nos. Sabendo isto, o resto do procedimento é muito parecido ao da opção **1** mas o processo de comparação na função no **hill climbing** vai passar a ser pelo número conflitos em vez do perímetro (Figura 35).

```
if(op==3){
    iBest=inter(this.bestSoFar);
    iCand=inter(candidate);
    if(iCand<iBest)
        op3=true;
}
```

Figura 35 - Código na função hillClimbling com a opção 3 (conflitos)

```
//Ex4.3-Funçao para saber o número de interseçoes
int inter(Point2D[] array){
    int cont=0;
    for(int i=1;i<this.tamanho;i++){
        for(int j=i+1;j<this.tamanho;j++){
            if(j!=(i-1) && j!=i && j-1!=i && j-1!=(i-1)){
                if(intersecao(array[i-1],array[i],array[j-1],array[j]))
                cont++;
            }
        }
    }
    return cont;
}
```

Figura 36 - Código da função inter

```
Original:
(-6,-17)(5,3)(-20,-6)(3,-13)(-17,3)(15,-2)(-10,-9)(4,-7)(-7,14)(-15,-8)

Array Solução:
(-6,-17)(3,-13)(4,-7)(15,-2)(5,3)(-7,14)(-10,-9)(-17,3)(-20,-6)(-15,-8)
Perimetro: 120.16101763722126


Demorou 1 milisegundos
Iterações do programa: 6
```

Figura 37 – Exemplo 1 do exercício 4.3

```
Original:
(-19,14)(-8,18)(3,-1)(-9,-7)(2,-7)(11,7)(-12,-13)(15,-1)(-5,-10)(14,4)(-4,-14)(-8,-11)(4,11)(13,-13)(2,16)(1,-1)(-7
,-4)(-12,0)(0,-11)(-11,2)(15,18)(10,3)(-16,-14)(-17,-8)(8,7)(-16,-12)(-7,-6)(5,-8)(-3,-16)(17,13)

Array Solução:
(-19,14)(-8,18)(-16,-12)(-7,-6)(-9,-7)(-12,0)(-7,-4)(-11,2)(2,16)(4,11)(11,7)(10,3)(15,18)(17,13)(15,-1)(13,-13)(5,
-8)(-3,-16)(-12,-13)(-4,-14)(-8,-11)(0,-11)(2,-7)(-5,-10)(14,4)(3,-1)(8,7)(1,-1)(-16,-14)(-17,-8)
Perimetro: 329.92261505446515


Demorou 15 milisegundos
Iterações do programa: 19
```

Figura 38 – Exemplo 2 do exercício 4.3

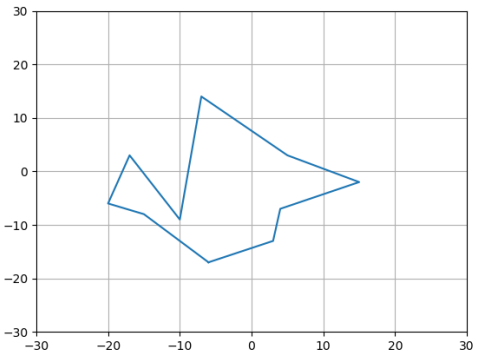


Figura 40 - Representação gráfica do exemplo 1 no exercício 4.3

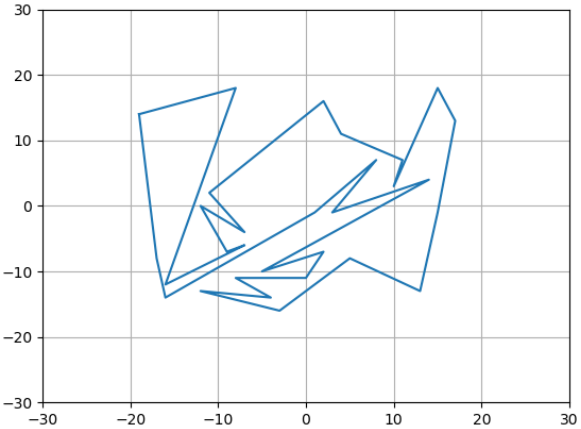


Figura 41 - Representação gráfica do exemplo 2 no exercício 4.3

4) Qualquer candidato

Para esta alínea, chamamos a função **hillClimbing** com a opção **4** (**hillClimbing(4)**).

Queremos um candidato qualquer da lista, então criamos um **Random s** e no final retornamos o elemento retirado da lista na posição random entre 0 e o tamanho da lista.

Como default retorna **null** porque não vai mais nenhuma opção sem ser de **1 a 4**.

Em código, o que foi dito em cima é traduzido desta forma:

```
Original:
(-6,-17)(5,3)(-20,-6)(3,-13)(-17,3)(15,-2)(-10,-9)(4,-7)(-7,14)(-15,-8)

Array Solução:
(-6,-17)(3,-13)(4,-7)(15,-2)(-7,14)(-17,3)(5,3)(-10,-9)(-20,-6)(-15,-8)
Perimetro: 139.84644216017975


Demorou 1 milisegundos
Iterações do programa: 10
```

Figura 42 - Exemplo 1 do exercício 4.4

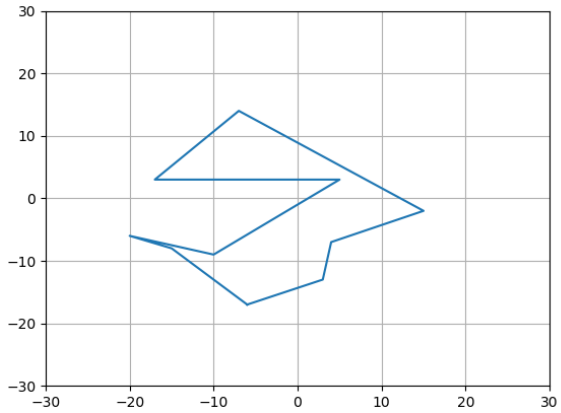


Figura 43 - Representação gráfica do exemplo 1 no exercício 4.4

```
Original:
(-19,14)(-8,18)(3,-1)(-9,-7)(2,-7)(11,7)(-12,-13)(15,-1)(-5,-10)(14,4)(-4,-14)(-8,-11)(4,11)(13,-13)(2,16)(1,-1)(-7
,-4)(-12,0)(0,-11)(-11,2)(15,18)(10,3)(-16,-14)(-17,-8)(8,7)(-16,-12)(-7,-6)(5,-8)(-3,-16)(17,13)

Array Solução:
(-19,14)(-8,18)(15,18)(17,13)(2,16)(4,11)(8,7)(1,-1)(10,3)(11,7)(14,4)(3,-1)(-7,-6)(-9,-7)(-7,-4)(-12,0)(-16,-12)(-
12,-13)(-8,-11)(-5,-10)(2,-7)(15,-1)(13,-13)(-3,-16)(5,-8)(0,-11)(-4,-14)(-16,-14)(-17,-8)(-11,2)
Perimetro: 271.87792311698763


Demorou 5 milisegundos
Iterações do programa: 28
```

Figura 44 - Exemplo 2 do exercício 4.4

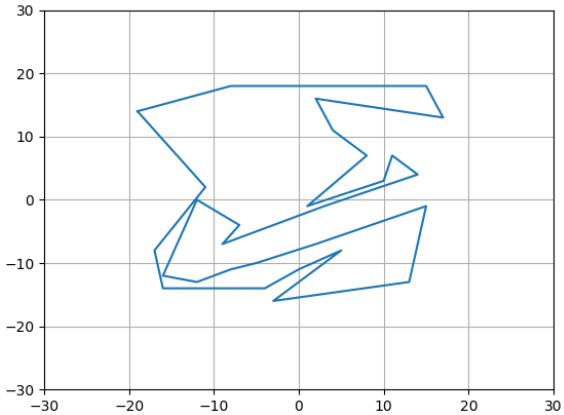


Figura 45 - Representação gráfica do exemplo 2 no exercício 4.4

Depois da explicação de cada opção, podemos finalmente fazer da aplicação do **Hill Climbing.**

Esta função recebe o argumento **op**, que vai ser a opção que o utilizador escolheu.

Antes de mais, iniciamos o nosso caso base, o array de pontos **bestSoSar** que vai ser inicializada com o nosso array inicial e criamos a lista dos candidatos respetivos do array **bestSoFar** (usando **exchange(2)**).

Enquanto a nossa lista de candidatos não for vazia, vamos pedir o array a partir da opção selecionada. De seguida calculamos o perímetro do candidato (**pCand**) e do **bestSoFar** (**pBest**) e depois verificamos qual o perímetro menor. Se o perímetro do candidato for menor, então o candidato passa a ser o nosso **bestSoFar**, e voltamos a criar a lista de candidatos. E voltamos a pedir o melhor candidato de acordo com a opção e a compararam o perímetro, até a lista do **bestSoFar** for vazia.

Caso a **op** for **3**, quer dizer que o utilizador escolher fazer a opção **3,** então em vez do meio de comparação for o perímetro, passa a ser o número de conflitos (Figura 35).

No final imprimimos o array de pontos resultante desse ciclo.

```
//Ex 4-Algoritmo HiilClimbing com menor perimentro ou menor interseções
void hillClimbing(int op){
    long startTime = System.currentTimeMillis();
    System.out.println("Original: ");
    printArrayPontos();

    this.bestSoFar=arrayC; //estado inicial
    double perimetroMin=perimetro(this.bestSoFar);
    double pBest=0.0, pCand=0.0; //Perimentros
    int iBest=0, iCand=0; //Interseções
    int cont=0; //contador de iterações
    Point2D[] candidate;

    exchange(2); //cria a lista de candidatos de acordo com o bestSoFar
    while(!lista.isEmpty()){
        candidate= opcao(op); //melhor candidato de acordo com a opçao escolhida
        pBest=perimetro(this.bestSoFar);
        pCand=perimetro(candidate);

        boolean op3=false; //validar a opção 3
        if(op==3){
            iBest=inter(this.bestSoFar);
            iCand=inter(candidate);
            if(iCand<iBest)
                op3=true;
        }
        if(pCand<pBest || op3){
            bestSoFar=candidate;
            perimetroMin=pCand;
            exchange(2);
        }
        cont++;
    }
    arrayFinal(bestSoFar,perimetroMin);

    long endTime = System.currentTimeMillis();
    System.out.println("\nDemorou " + (endTime - startTime) + " milisegundos");
    System.out.println("Iterações do programa: " +cont);
}
```

Figura 46 – Código da função hillClimbing

Exercício 5

Aplicar **simulated annealing**. Usar como medida de custo o número de cruzamentos de arestas.

Neste exercício o objetivo é usar o algoritmo **simulated annealing**.

O pré-código correspondente ao **Simulated annealing** é:

Simulated Annealing (SA):

determine initial candidate solution s
set initial temperature T according to *annealing schedule*
While termination condition is not satisfied:
  probabilistically choose a neighbour s' of s
    using *proposal mechanism*
  If s' satisfies probabilistic *acceptance criterion* (depending on T):
    $s := s'$
  update T according to *annealing schedule*

Figura 47 – Pseudocódigo do algoritmo do simulated anneling

Nesta adaptação do algoritmo para a temperatura inicial vamos considerar o número de cruzamentos do array inicial. A escolha do candidato vai ser feita a partir do elemento da lista que tem menos cruzamentos. Neste caso podemos reutilizar a função opção com o argumento nº 3 (**opcao(3)**).

Para o critério de aceitação, temos uma função auxiliar **acceptanceProbability** que vai comparar os perímetros e se o perímetro do candidato for menos que o do **besSoFar** então retorna 1 ou seja é aceite, senão retorna **e^((min - max) / temp)** que é a rejeição.

Então a nossa função de acceptibilidade vai ser:

```
//Ex5-Função se aceita o candidato
double acceptanceProbability(double min, double max, double aux) {
    //se for aceite o candidato
    if (max < min)
        return 1;

    //caso nao seja aceite
    return Math.exp((min - max) / aux);
}
```

Figura 48 – Código da função acceptanceProbability

Para caso base vamos ter um array de pontos **bestSoFar** inicializado com o array original e efetuamos o **exchange(2)** para termos a lista de filhos relativamente ao **bestSoFar**. Depois fazemos uma condição em que enquanto a lista não for vazia e a temperatura for maior que zero, vamos buscar o nosso candidato chamando a função **opcao(3)** e calculamos o perímetro do candidato **(pCand)** e do **bestSoFar (pBest)**.

Agora verificamos a função explicada acima onde vimos se a função de aceitação for igual a 1 ou seja se for aceite, o nosso **bestSoFar** vai passar a ser o candidato, limpamos a lista e voltamos a fazer o **exchange** para termos a lista de filhos do candidato. Se for rejeitado então passamos o nosso candidato como **bestSoFar**, limpamos a lista e fazemos o **exchange**. No final vamos atualizar a temperatura e é nesta atualização em que nós decidimos a rapidez que nós queremos que ela converge, neste caso nós metemos a convergência para **0.95xtemperatura**. No final, imprimimos o array resultante.

```java
//Ex5-Simulated annealing, medida de custo cruzamentos de arestas
void simA(){
    long startTime = System.currentTimeMillis();
    System.out.println("Original:  ");
    printArrayPontos();

    this.bestSoFar=arrayC;
    double auxTemp=(double)inter(arrayC); //temperatura
    double perimetro=0.0;
    int cont=1;
    Point2D[] candidate;
    exchange(2);

    while(!lista.isEmpty() && auxTemp>0){
        candidate=opcao(3);
        double pBest=perimetro(this.bestSoFar);
        double pCand=perimetro(candidate);

        if(acceptanceProbability(pBest, pCand, auxTemp)==1){
            bestSoFar=candidate;
            perimetro=pCand;
            exchange(2);
        }
        //atualizar a temperatura
        auxTemp=(double) 0.95*auxTemp;
        cont++;
    }
    arrayFinal(bestSoFar,perimetro);
    long endTime = System.currentTimeMillis();
    System.out.println("\nDemorou " + (endTime - startTime) + " milisegundos");
    System.out.println("Iterações do programa: " +cont);
}
```

Figura 49 – Código da função simA

```
Original:
(-6,-17)(5,3)(-20,-6)(3,-13)(-17,3)(15,-2)(-10,-9)(4,-7)(-7,14)(-15,-8)

Array Solução:
(-6,-17)(3,-13)(4,-7)(15,-2)(5,3)(-7,14)(-10,-9)(-17,3)(-20,-6)(-15,-8)
Perimetro: 120.16101763722126


Demorou 1 milisegundos
Iterações do programa: 7
```

Figura 53 – Exemplo 1 do exercício 5

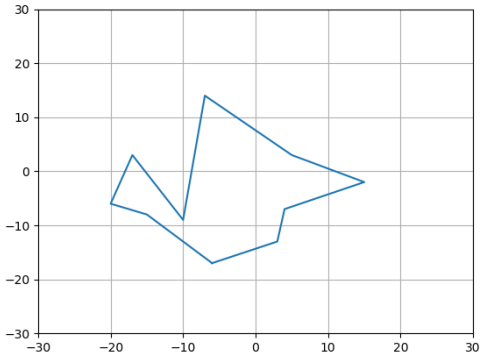


Figura 51 - Representação gráfica do exemplo 2 no exercício 5

```
Original:
(-19,14)(-8,18)(3,-1)(-9,-7)(2,-7)(11,7)(-12,-13)(15,-1)(-5,-10)(14,4)(-4,-14)(-8,-11)(4,11)(13,-13)(2,16)(1,-1)(-7,
-4)(-12,0)(0,-11)(-11,2)(15,18)(10,3)(-16,-14)(-17,-8)(8,7)(-16,-12)(-7,-6)(5,-8)(-3,-16)(17,13)

Array Solução:
(-19,14)(-8,18)(-16,-12)(-7,-6)(-9,-7)(-12,0)(-7,-4)(-11,2)(2,16)(4,11)(11,7)(10,3)(15,18)(17,13)(15,-1)(13,-13)(5,-
8)(-3,-16)(-12,-13)(-4,-14)(-8,-11)(0,-11)(2,-7)(-5,-10)(14,4)(3,-1)(8,7)(1,-1)(-16,-14)(-17,-8)
Perimetro: 329.92261505446515


Demorou 10 milisegundos
Iterações do programa: 20
```

Figura 52- Exemplo 2 do exercício 5

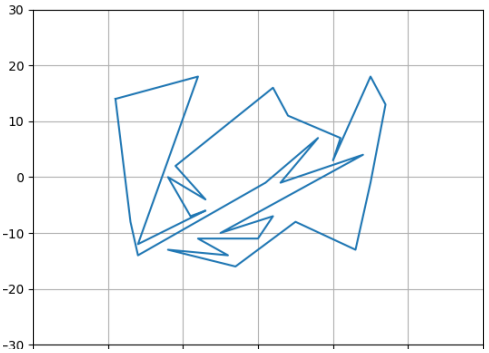


Figura 50 - Representação gráfica do exemplo 2 no exercício 5

Conclusões finais

Depois de executar o programa com vários valores, diferentes valores n e m, conseguimos retirar algumas conclusões.

Todos os nossos métodos, independentemente do número de coordenadas e do range, vai formar sempre polígonos simples.

Há diferenças significativas relativamente à lista de candidatos quando partimos do **random** ou do *"candidato nearest-neighbour first"*, sendo que iremos ter muito mais candidatos na lista se o fizermos pelo **random** do que pelo *"nearest-neighbour first"*.

Se tivermos um range pequeno a probabilidade de ter mais interseções é mais alto do que se tivermos um range muito grande.

No **Simulated Annealing** a temperatura e o esquema de arrefecimento vão influenciar o resultado, porque como a temperatura inicial é igual ao número de cruzamentos do nosso array inicial, quanto maior for a sua temperatura mais pode influenciar no tempo, podendo demorar mais se a temperatura for muito alta. O esquema de arrefecimento também pode influenciar o resultado, porque vai determinar com que rapidez queremos com que o nosso programa converge, sendo que se tiver um valor alto de convergência pode demorar muito pouco a concluir ou até dar erro e, se tive um valor baixo de convergência pode influenciar no seu tempo de execução.

Não conseguimos resolver o **exercício 6** devido a um bug encontrado perto do prazo da entrega, tendo sido descoberto na parte de troca de índices na função **exchange**, o que nos fez demorar mais que o previsto a corrigir. Este bug fazia com que os candidatos pudessem ter um perímetro maior que o array pai.

Referências

https://www.geeksforgeeks.org/check-if-two-given-line-segments-intersect/ -> verificação se 2 segmentos se intersetam

https://www.geeksforgeeks.org/orientation-3-ordered-points/ -> verifica a orientação dos segmentos

https://stackabuse.com/simulated-annealing-optimization-algorithm-in-java/ -> ajudar a perceber como funciona o simulated annealing

Piazza CC2006.

Distribuição do trabalho

Devido a situação que nos encontramos agora, o trabalho foi realizado em conjunto (ao mesmo tempo). Para isso usamos muito a partilha de ecrã, para partilhar código (vermos o que cada uma ia escrevendo) e desenhos para tentarmos perceber as diferentes interpretações. Isto só foi possível pois comunicamos muito e tentávamos ao máximo explicar o nosso ponto de vista à outra.

Sendo a comunicação através das redes socias muito mais fácil, para partilha de ideias entre colegas era muito mais simples. Ajudamos alguns colegas a perceber certos algoritmos através de simples desenhos e também fomos ajudadas na implementação da interface gráfica de python, pelo nosso colega Duarte Alves que nos ajudou a perceber python já que era algo que nenhuma de nós tinha tido contacto com a linguagem.

Em relação a implementação gráfica, tentamos realizá-la em Java, porém não estava a dar o gráfico correto, e depois da explicação do nosso colega, achamos mais simples realizar em python.