Faculdade de Ciências da Universidade do Porto

Trabalho Prático de Inteligência Artificial

Grupo 34

Joana Pinto (up201905100)

Valentina Wu (up201907483)

Abril 2021

Índice

Introdução

Este projeto teve como objetivo o estudo de métodos de pesquisa local e pesquisa local estocástica usando o problema **RPG**, ou seja, gerador de polígonos aleatórios, que gera um polígono simples dados um conjunto de n pontos no plano. A implementação, experimental, deste problema nos vários exercícios foi feita com a linguagem Java.

Usamos o método de **pesquisa local heurística**, que foi desenhado com o intuito de resolver problemas de maneira rápida, mas não tem garantia que seja a mais eficaz. Este método é usado para questões, em que é preciso encontrar uma solução que otimize uma regra entre vários candidatos de solução. Também, utilizamos o algoritmo **2-exchange**, que desconecta 2 arestas e, depois, após estarem desligados, junta os com pontos dessas arestas que permitam cria um polígono com só uma componente convexa.

Através da pesquisa local e do algoritmo 2-exchange foi possível gerar um polígono simples.

Para este trabalho criamos duas classe, Coordenadas (), para guardar dois pontos (x,y) e uma classe Grafo, que tem como atributo um inteiro, para guardar o número de nós, um array de Coordenadas, com nome nodes, para sabermos num determinado índice qual a coordenada, uma matriz de adjacências para guardar as ligações que se fazem nos índices das coordenadas e um array de boolean que marca se o nó já foi visitado.

Exercício 1

Criamos uma classe de Coordenadas para guardar os pontos.

Também foi criada uma função *contem()* com o objetivo de fazer a comparação entre os dois pontos, se as coordenadas fossem iguais daria true caso contrário daria false.

Para geramos coordenadas aleatórias, utilizamos uma função do java, o Random (), para obter os pontos e guardá-las num array de Coordenadas, verificando se esta já existe ou não no conjunto com a função *contem()*.

Exercício 2

Os candidatos da solução são gerados por pesquisa local.

a) Para criar o candidato com uma permutação de qualquer dos pontos, geramos os pontos aleatoriamente, utilizamos o *Random*(), para dar um índice de uma coordenada, e se essa coordenada ainda não foi visitada, isto é, ainda não houve nenhuma ligação dessa coordenada com outra e for diferente daquele que se liga então podemos ter uma ligação, isto é, acrescentar essa aresta no grafo.

b) Para gerar uma solução aplicando a heurística “nearest-neighbour first”, isto é, a partir de um ponto chegamos ao primeiro mais perto.

Para isso, tivemos de implementar dois métodos o NNFirst() e a distância(), o primeiro retorna um nó mais próximo daquele que se passa como argumento, comparando assim com todos os nós que ainda não foram visitados e o que tiver menor distância é retornado e o segundo retorna a distância euclidiana por dois nós, isto é, duas coordenadas.

Exercício 3

Determinar a vizinhança por “2-exchange”, isto é, dando duas arestas {i,j},{k,l} que intersetam passa para {i,k},{j,l}, esta mudança faz com que as arestas não fiquem mais intersetadas.

Para implementar este exercício, com ajuda do algoritmo de interseção de um par de segmentos [1], podemos verificar se duas arestas composta por 4 coordenadas, se intersetam e guardá-las numa lista para depois poder fazer a operação “2-exchange”, e um método *two_exchange()*, que remove primeiro as ligações entre as coordenada, isto é, por exemplo, duas arestas {i,j},{k,l} temos de fazer a remoção entre a ligação i,j e k,l e de seguida fazer uma ligação com i,k e j,l verificando se o grafo ainda é um polígono.

Essa verificação é feita pela verificação de componentes conexos, se houver mais que uma então temos que remover a ligação feita e ligar pela ordem contraria.

Exercício 4

A ideia é com o algoritmo de *hill-climbing* gerar um polígono simples de um candidato inicial.

Com a ideia de gerar candidatos na alínea 3, podemos a partir desses candidatos optar por um deles, segundo um critério de cada alínea, até chegar a um polígono simples.

Neste exercício, implementou-se um método para verificar se um polígono é simples, utilizou-se como auxiliar a função para verificar se duas arestas se intersetam, se houver ainda interseções, retornamos um true senão um false.

A ideia principal foi enquanto o grafo original não for um polígono simples, então geramos candidatos e a escolha de um candidato segundo um critério torna esse candidato no grafo original, e continua o processo de novo até chegar a uma solução.

a) A escolha do candidato é aquela que reduzirá mais o perímetro, como sabemos que duas arestas se intersetam a sua remoção de interseção faz com que o novo grafo fique com menos perímetro, logo temos a certeza de que não ficamos preso do máximo local, se as arestas não ficarem intersetadas.

O cálculo do perímetro foi a soma das arestas, segundo a distância euclidiana, e comparando todos os candidatos e escolher aquele que tiver menor perímetro.

b) A escolha do primeiro candidato encontrado.

c) A escolha do candidato com menos conflitos de aresta, todos os candidatos são comparados e aquele que tiver menos interseções, passa a ser o original.

d) Com a função *Random* () do java, gere se aleatoriamente um candidato nessa vizinhança.

Resultados:

4a) Com o critério, NB (propriedade geométrica), do menor perímetro temos a certeza que o número de interações no candidato escolhido nunca é maior que o original, isto é, nunca aumenta, pois sabemos que duas aresta {i,j} e {k,l} se intersetam o perímetro é sempre maior do que {i,k}{j,l}, a não intersetarem.

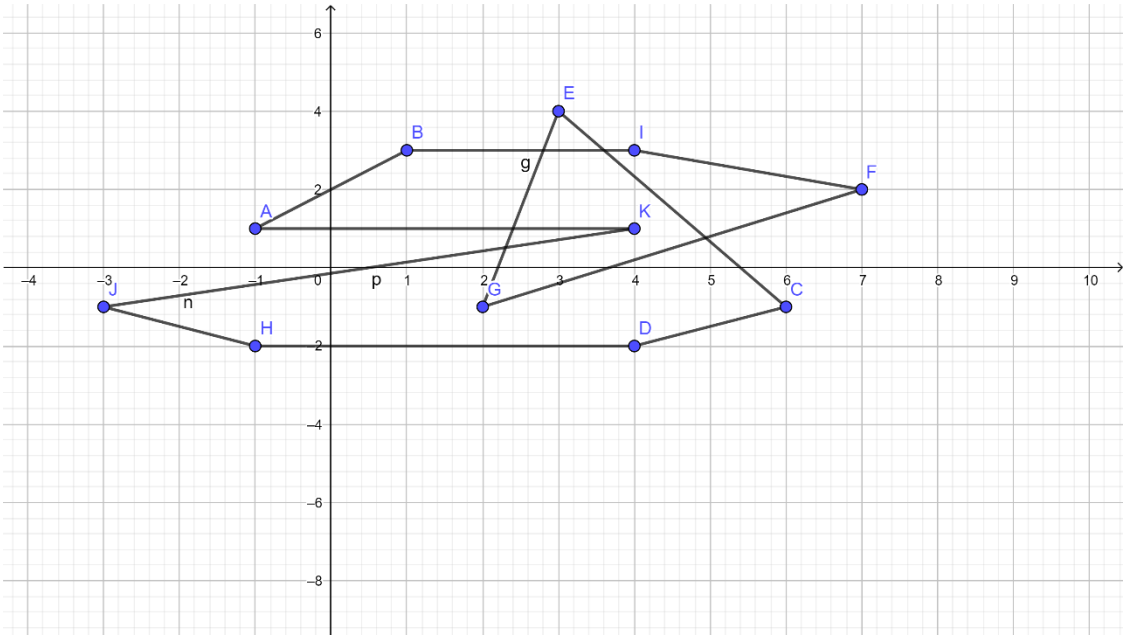

Fig.1: Gráfico 1 – com 5 interseções {A,K} {E,G}, {B,I}{C,E}, {B,I}{E,G}, {C,E}{F,G} e {E,G}{J,K}.

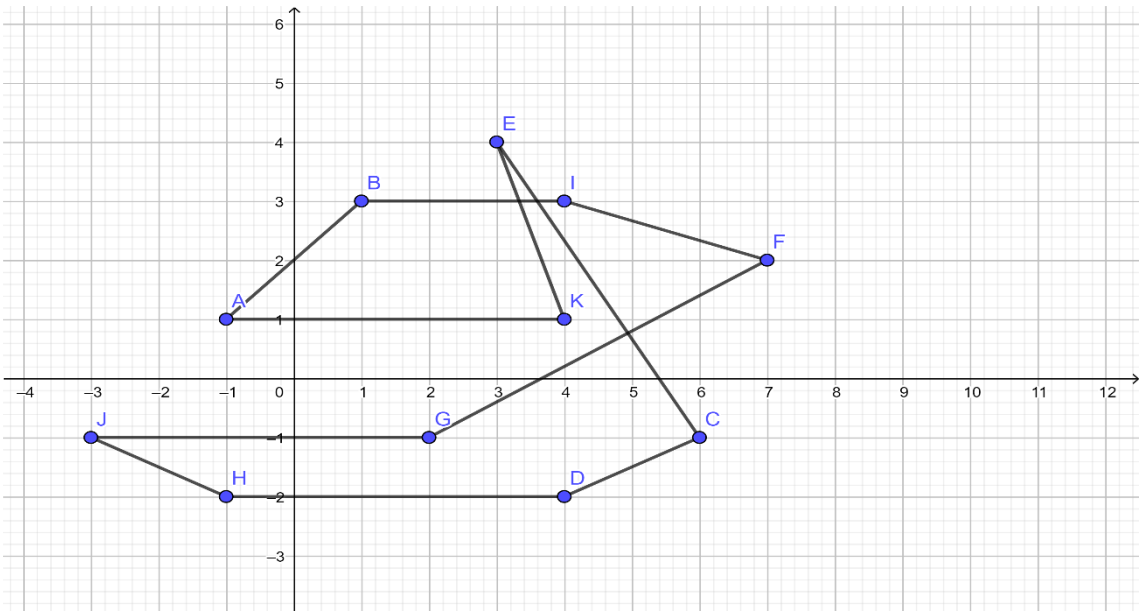

Fig.2: 1.ª iteração do gráfico 1 após escolha do 5.º candidato com passa a ser tem 3 interseções {B, I} {C,E}, {B,I}{E,K} e {C,E}{F,G}.

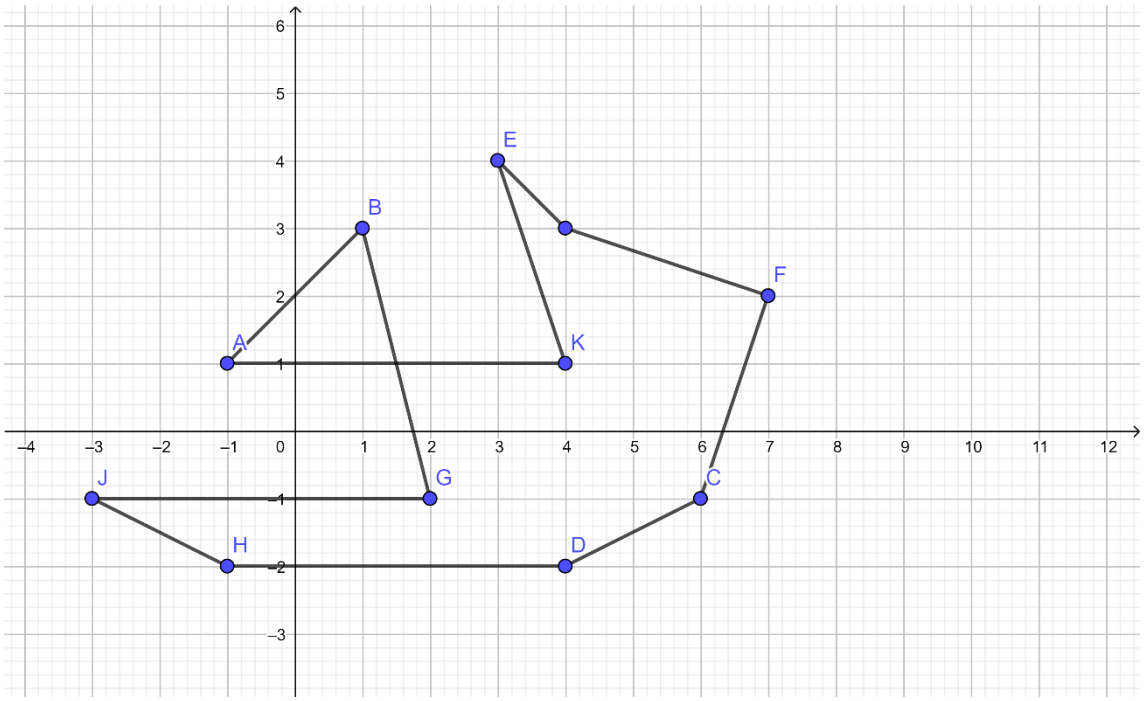

Fig.3: Penúltima iteração do gráfico 1 – passa a ter uma interseção {A,K}{B,G}.

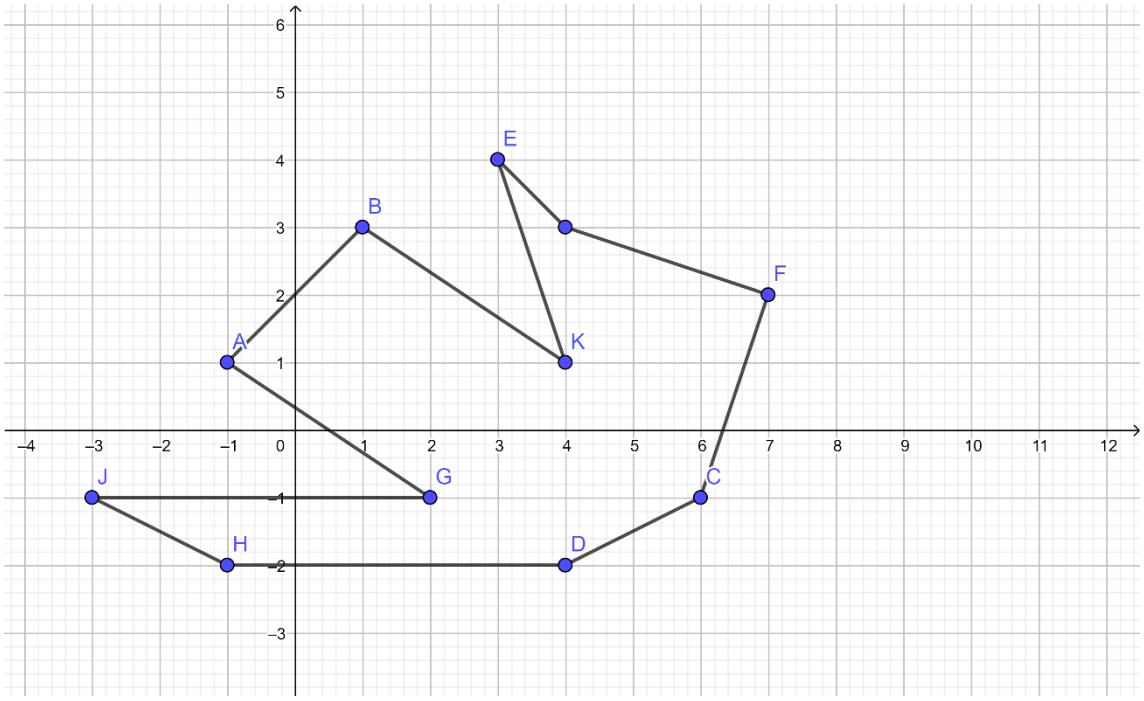

Fig.4: Solução do gráfico 1.

4b) Podemos ter uma desvantagem neste critério, pelo que se a quantidade de interseções dos pontos não mudarem, ele pode ficar preso num máximo local. Como por exemplo, num polígono convexo:

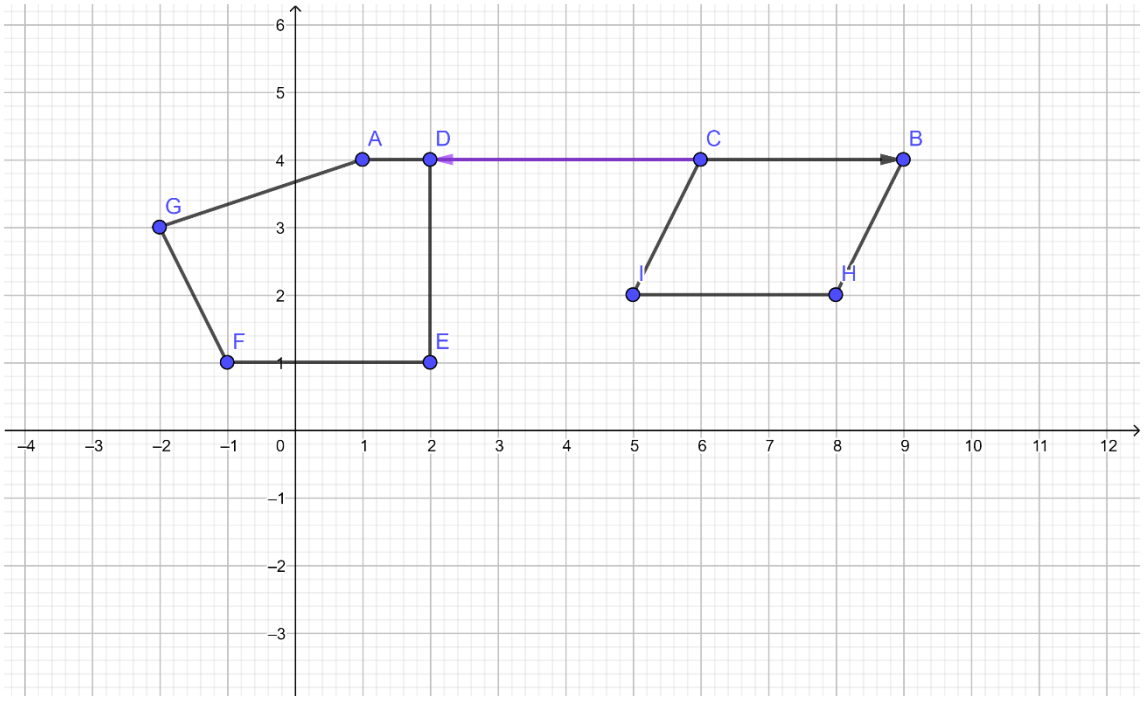

Fig.5: Gráfico 2 – Polígono Convexo

O primeiro candidato é {A, B}, {C, D}, após a operação de 2-exchange ficaria {A, C} {B, D}, e com essa escolha regressaria ao início.

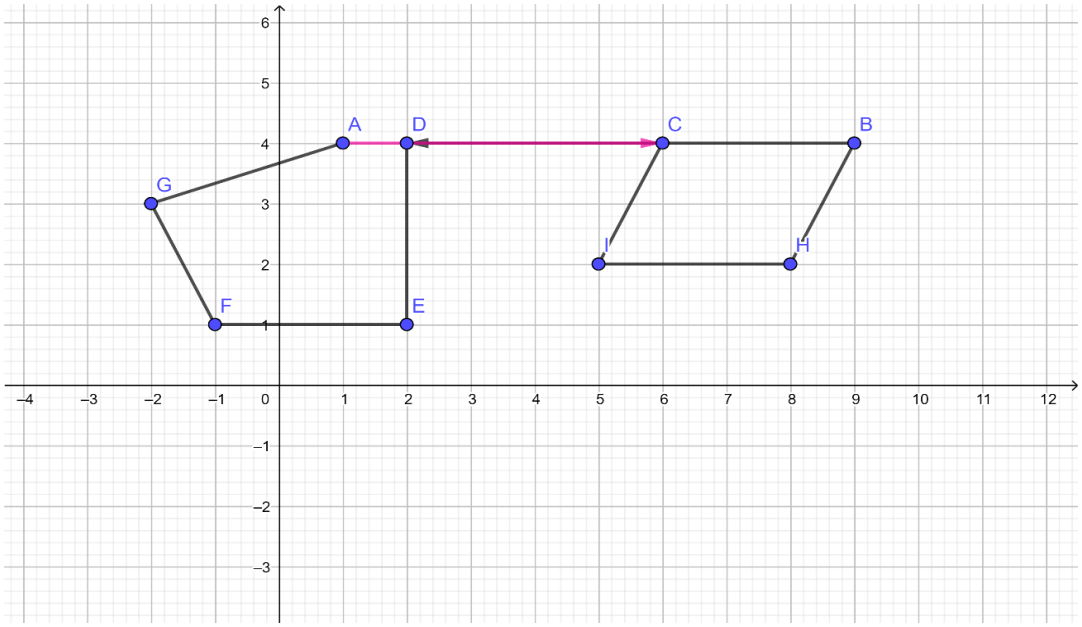

Fig.6: Gráfico 2

4c) Este critério é vantajoso, se o nosso polígono fizer uma mudança “ampla”, pois ao escolher sempre o que tiver menos conflitos e se houver um candidato que dê logo para a solução essa é a opção.

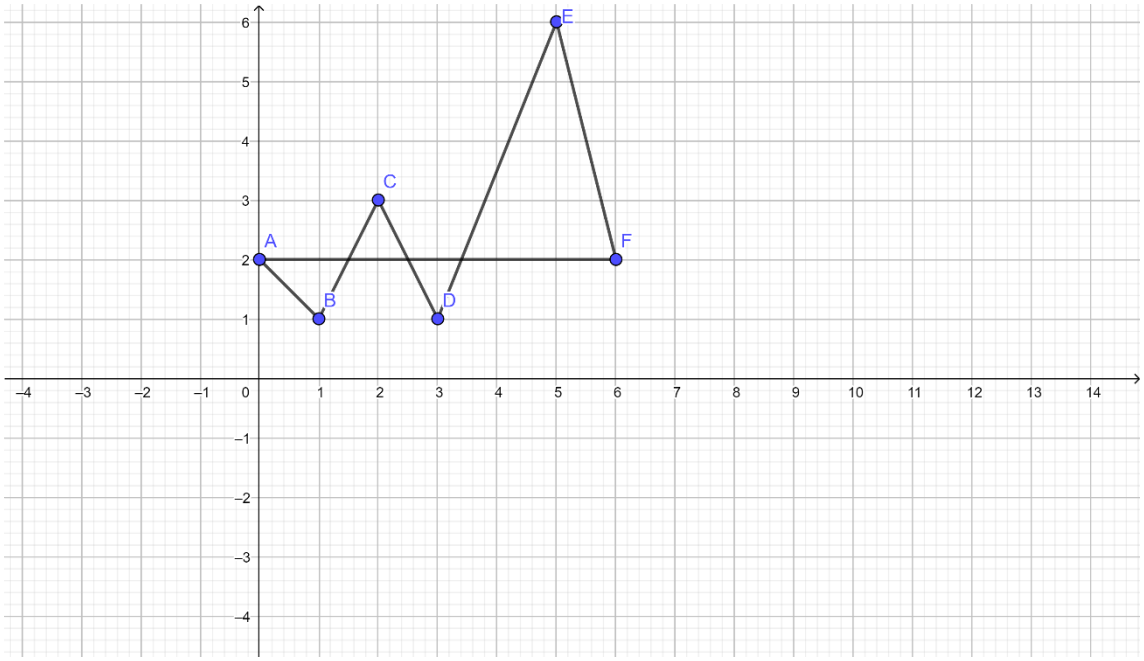

Fig.5: Gráfico 2 – três interseções {A,F}{B,C}, {A,F}{C,D} e {A,F}{D,E}

A escolha do 3.º candidato implica logo um polígono simples:

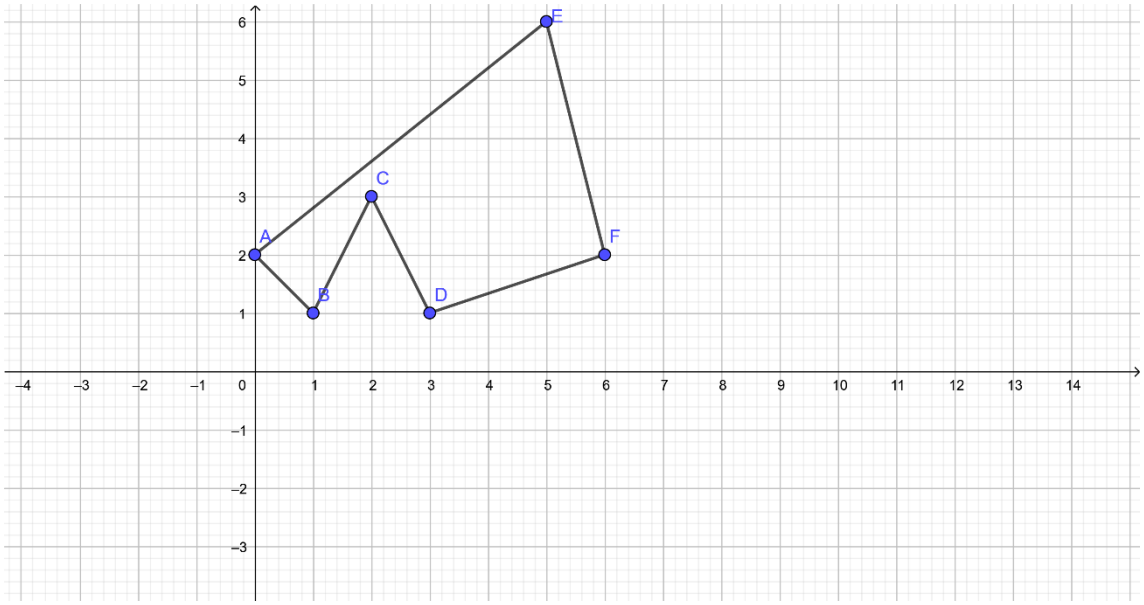

Fig.6: Solução do gráfico 2.

4d) Com o critério aleatório, podemos entrar em mínimos locais ou sair dos máximos locais, isto é, como por exemplo nunca iríamos ficar preso no exemplo 2b), mas a solução podia não ser tão rápido a encontrar-se.

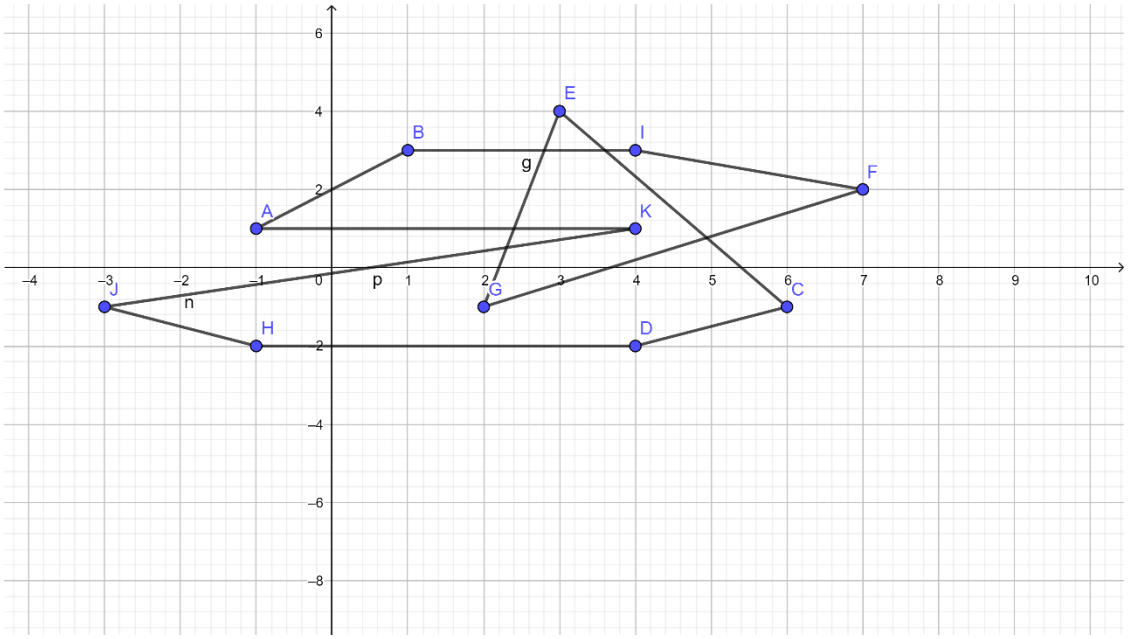

Fig.7: Gráfico 3 – tem 5 interseções {A,K} {E,G}, {B,I}{C,E}, {B,I}{E,G}, {C,E}{F,G} e {E,G}{J,K}.

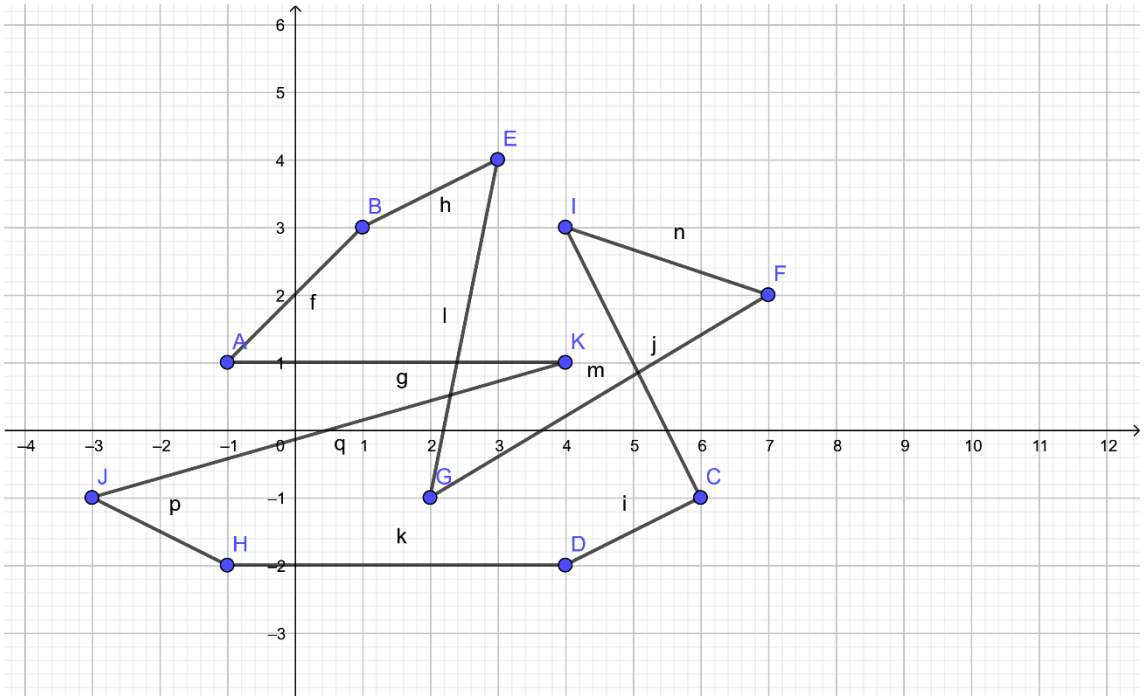

Fig.8: 1.ª iteração do gráfico 3 após escolha aleatória do 2.º candidato – 3 interseções {A,K}{E,G}, {C,I}{F,G} e {E,G}{J,K}

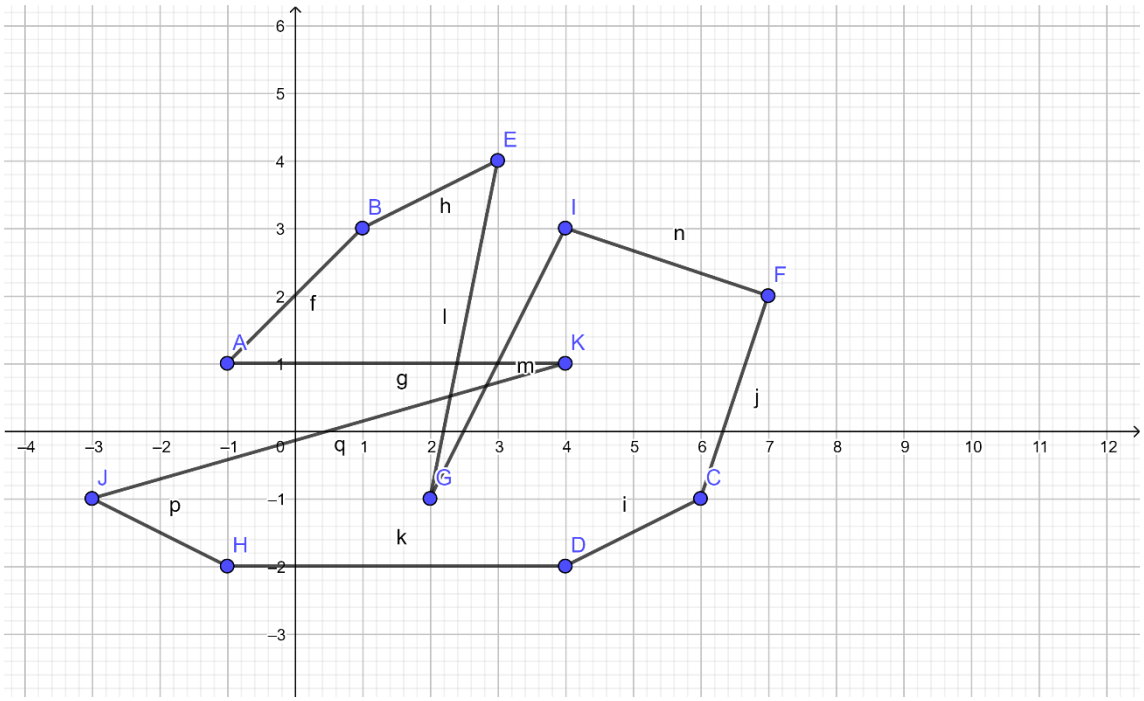

Fig.9: 2.ª iteração do gráfico 3 passa a ter 4 interseções.

Exercício 5

Usamos o método de *simutaled annealing* que usa a pesquisa local probabilística que tem como base uma analogia com a termodinâmica. Com este algoritmo havendo uma nova iteração é gerado um novo ponto aleatório, a sua distância ao ponto atual, é apoiada numa partilha de probabilidade com escala proporcional à temperatura. Implementamos o algoritmo de simutaled annealing, que minimiza o tempo de execução. E faz a escolha de um candidato aleatório, mas temos de confirmar algumas condições para continuar o processo. ΔE=f(candidato)-f(original), f é o custo de número de cruzamentos, se ΔE for maior que 0, estamos a aceitar numa primeira faze os mínimos locais, e senão podemos aceitar um determinado candidato com uma probabilidade e^(ΔE/T), T a temperatura de acordo com o *annealing Schedule*.

Resultados:

Este método é vantajoso uma vez que, este aceita mínimos locais, para não ficarmos preso nos máximos locais, o que ocorre bastantes vezes no início. E como este aceita candidatos com uma determinada probabilidade, o que importa sendo, é que essa probabilidade diminua, ficamos na escolha do candidato razoavelmente correto.

Exercício 6

ACO (*Ant Colony Optimization*), diz que cada formiga tem uma atividade interativa, uma formiga a cada iteração, quando faz um percurso, percorrendo tudo o grafo e não visita o mesmo nó duas vezes e a iteração seguinte é probabilística, tendo em conta o nível de concentração de feromonas [3].

Numa primeira fase, inicializamos todas as variáveis com valores 1, a feromona e a probabilidade.

Utilizamos como tmax, o máximo de iterações, o valor de 30, e 30 formigas, também, temos que encontrar um percurso para cada formiga, daí a função *tour_construct(),* começando num ponto aleatório, a sua escolha do ponto que vá fazer uma ligação tem haver com uma dada probabilidade, e essa probabilidade depende da feromona deixada pelas outras formigas e a distância entre os dois nós. Sendo assim, fazemos um percurso para cada formiga, e atualizamos a feromona conforme os percursos feitos pelas formigas, e continuamos o processo até tmax.

A solução é feita a partir de um ponto inicial aleatório e a partir desse ponto ver a feromona nos restantes pontos e escolher aquela que tiver mais feromona, pois isto significa que por aquele caminho passaram mais formigas.

Resultados:

A solução devia, em princípio, dar sempre um polígono simples, pois sabemos que um polígono com interseção tem sempre o perímetro maior que o polígono sem nenhuma interseção, contudo uma vez que a diferença não é significativa, por essa razão, que acontece a interseção apresentada no gráfico do exemplo 1.

O valor que colocámos às constantes alfa, beta e Q concebem, também uma diferença no resultado final.

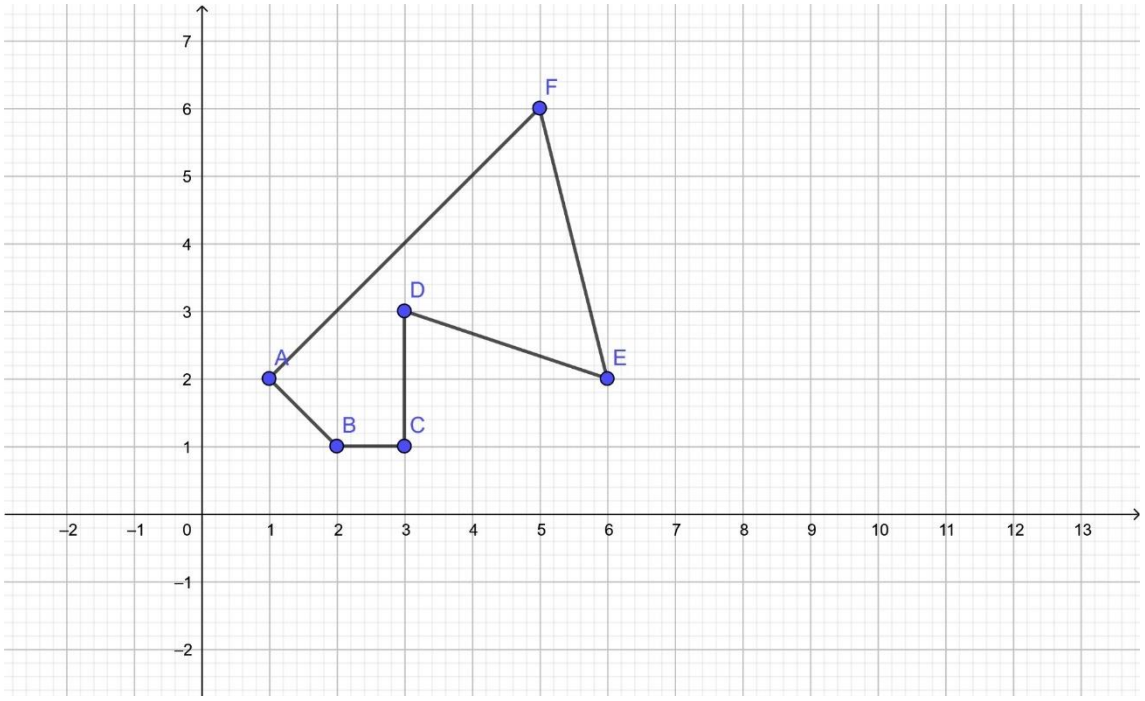

Fig.10: Gráfico – Exemplo 1.

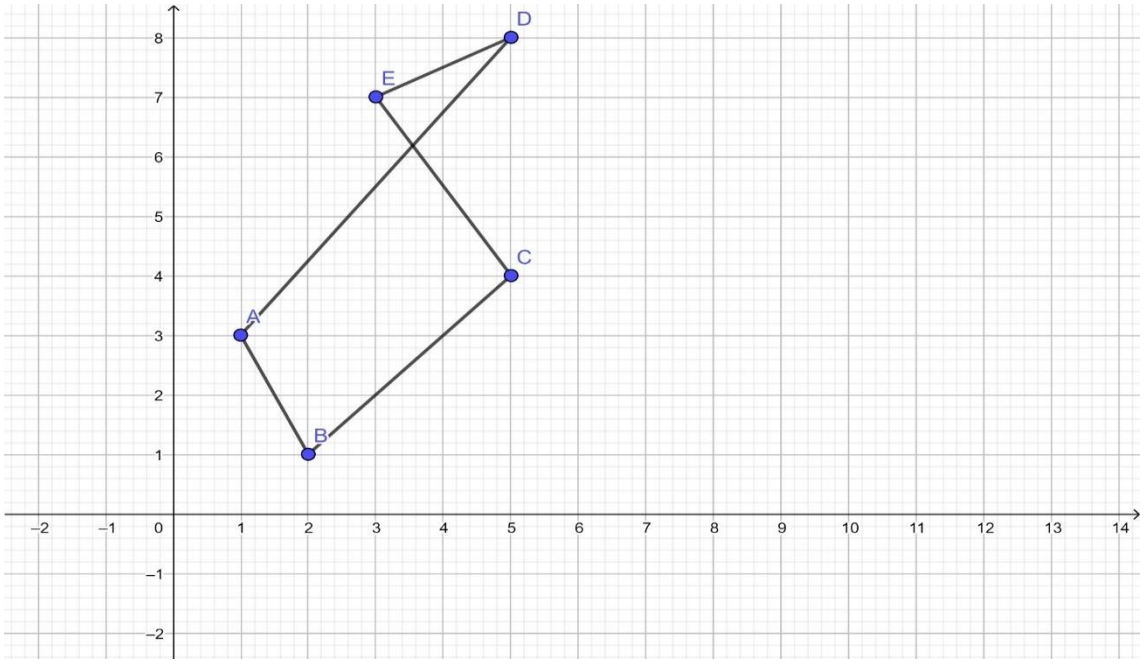

Fig.11: Gráfico – Exemplo 2.

Conclusão

Utilizamos 3 diferentes algoritmos para resolver o problema RPG, *hill climbing*, *simullated annealing* e ACO.

Há vantagens e desvantagens destes três métodos, *hill climbing*, embora seja de uma complexidade baixa comparando com as outras duas e ser base para outros métodos heurísticos, tem a desvantagem de podemos ficar presos num máximo local.

Sobre o *simulated annealing*, são aceites os mínimos locais e nunca fica preso num máximo local, apesar da convergência ser garantida, a implementação do programa é mais complexa, logo mais lenta.

O ACO é um algoritmo que dá para aplicar a vários problemas parecidos a este problema, ou seja, é um algoritmo bastante flexível, mas na nossa implementação não deu uma solução ótima e comparando com os outros dois são efetuados demasiados cálculos, o que causa um elevado tempo de execução computacional.

Bibliografia

[1] http://www.inf.ed.ac.uk/teaching/courses/ads/Lects/lecture1516.pdf

[2] Artificial Intelligence - A Modern Approach (3rd Edition).pdf

[3]https://staff.fmi.uvt.ro/~daniela.zaharie/ma2019/lectures/metaheuristics2019_slides7.pdf

[4] https://en.wikipedia.org/wiki/Local_search_(optimization)

[5]https://paginas.fe.up.pt/~mac/ensino/docs/OR/CombinatorialOptimizationHeuristicsLocalSearch.pdf

[6] https://www.researchgate.net/publication/226000523_TOPOS_-_A_new_constructive_algorithm_for_nesting_problems

[7] https://www.mathworks.com/help/gads/what-is-simulated-annealing.html

[8] http://www.lac.inpe.br/~lorena/cap/Aula_C01.pdf