Projeto IA

Faculdade de Ciências da Universidade do Porto

Pedro Santos, up201907254

Tiago Eusébio, up201904872

Introdução

Neste relatório abordamos diferentes estratégias usadas para a construção de polígnos na sua forma mais simples, recorrendo assim a métodos que têm como base algoritmos de pesquisa local que se baseiam na procura de possíveis candidatos a soluções, como por exemplo o (TSP, "Traveling Salesman Problem") que dado um grafo não dirigido completo pesado consegue encontrar uma solução peseudo-ótima como irá ser demonstrado no decorrer do relatório.

Desenvolvimento

Com a otimização em mente o "Traveling Salesman Problem" mais conhecido como TSP,(é um problema que partindo de um conjunto de pontos inicial cria um percurso que une todos pontos de forma a que possa minimizar a distancia euclidiana/custo entre pontos estando fundamentalmente associado a problemas relacionados com viagens e transportes, contudo o seu algoritmo é associado a varias discussões que resultam do facto de o mesmo nao ser capaz de chegar a uma solução ótima sem que resulte num grande custo tanto de memoria como de tempo assim sendo têm sido criadas muitas variações do seu algoritmo que procuram encontrar uma resposta para o mesmo problema reduzindo ao máximo a sua complexidade temporal necessária conformando-se com uma pseudo-solução.

Exercicio 1)

Neste primeiro exercício decidimos criar uma estrutura "Point" capaz de suportar dois inteiros de forma a pudermos guardar e aceder de forma mais eficiente a todas as coordenadas de cada ponto do poligono. Criamos depois um array $polig[n]$ com um tamanho n igual ao numero de pontos que o utilizador tenha selecionado, depois a partir de um ciclo que percorre todas as posições do mesmo geramos dois numeros aleatórios dentro de um intervalo também dado pelo utilizador e guardamos os conjuntos de pontos nas diferentes posições do array.

Exercicio 2)

Este exercício consiste na criaçao de ligações entre pontos criados no exercicio anterior, para esse objetivo formamos ligações por inserção e também seguindo a heurística "nearest-neighbor first" que tem por base o algoritmo TSP.

a)

Relativamente as ligações por inserção, para a sua execução, seguimos a ordem que ficou guardada em $polig[\,]$ considerando um array de pontos do género $[A, B, C, D]$ isto significaria que iriamos obter as seguintes ligações $A- > B- > C- > D$, sendo que o array inicial iria sofrer uma permutação tal como foi pedido pelo exercício.

b)

Seguindo a heurística anteriormente mencionada decidimos criar mais dois arrays com o mesmo tamanho de $polig[\,]$ um para guardar a ordem pela qual os pontos serão ligados $order[\,]$ e outro $visited[\,]$ que foi usado para sinalizar os pontos ja guardados no array anterior, como é possível observar no excerto de código abaixo que apresenta a implementação que foi feita para concretizar a heurística em questão.

```
static int min(Point[] polig, int rand, int m, boolean[] visited) {
    double temp = Math.pow(m, 2) * 2; // distancia maxima entre
   pontos
    int fim = 0;
    for (int i = 0; i < polig.length - 1; i++) {
        if (rand != i && !visited[i]) {
            double dist = Math.sqrt((Math.pow((polig[rand].x -
   polig[i].x), 2)) + (Math.pow(polig[rand].y - polig[i].y, 2)));
            if (temp > dist) {
                temp = dist;
                fim = i;
            }
        }

    }
    visited[fim] = true;
    return fim;
}
```

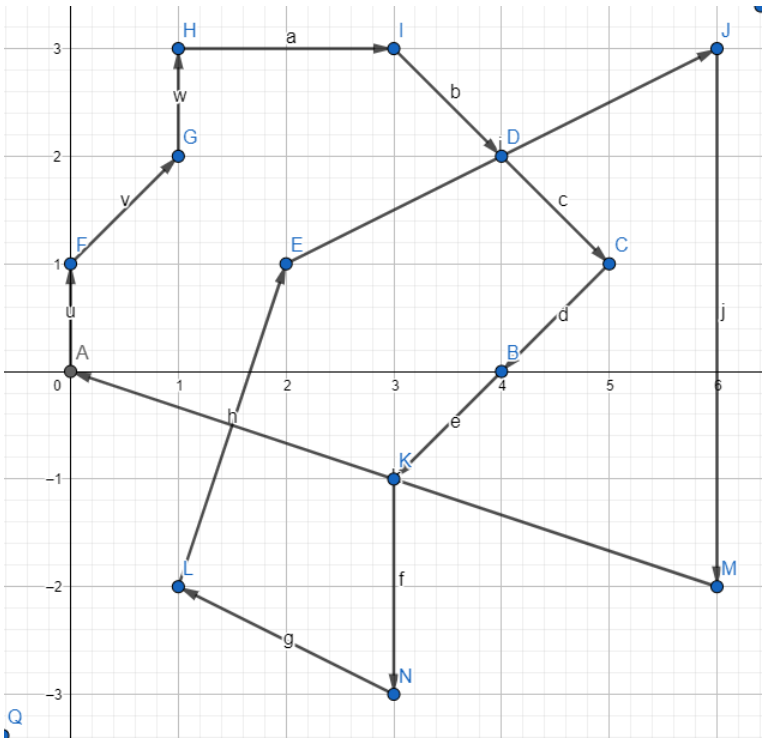
A função que criamos para resolver este exercício recebe um qualquer dos pontos e ira guardalo no próximo index de *order*[], seguido-se a procura sequencial do "vizinho" mais próximo e tendo em atenção todos os nós já visitados , calcula a distancia euclidiana desse ponto a todos os outros, este processo é repetido até termos ligado todos os pontos de forma a gerar um poligono.

Resultados:

Para testar o algoritmo anteriormente apresentado passamos como input o seguinte conjunto de pontos:

$$[(0,0),(4,0),(5,1),(4,2),(0,1),(2,1),(1,2),(1,3),(3,3),(6,3),(3,-1),(1,-2),(6,-2),(3,-3)]$$

e obtivemos o seguinte gráfico:

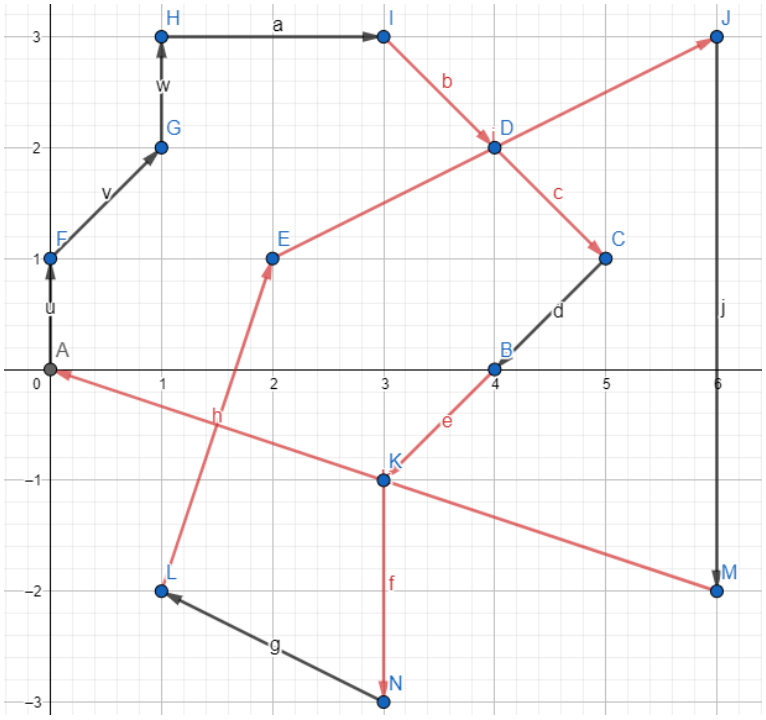


Como é possível observar através desta visualização gráfica o algoritmo fez ligação correta dos pontos dados como input inicialmente resultando na seguinte ordem final: $(0,0) -> (0,1) -> (1,2) -> (1,3) -> (3,3) -> (4,2) -> (5,1) -> (4,0) -> (3,-1) -> (3,-3) -> (1,-2) -> (2,1) -> (6,3) -> (6,-2)$

Exercicio 3)

Neste exercício foi usado o poligono resultante do exercício anterior para tentar encontrar/identificar possíveis interseções com recurso ao "2-exchage" que procura desfazer intreseções trocando duas arestas, ligações essas que são formadas por qualquer um dos métodos a "nearest-neighbor first" ou ligação por inserção acima referidos.

Resultados:



Para isso testamos se a partir de dois segmentos que nao contenham pontos repetidos, pois nesta implementação uma sequencia de segmentos como $A->B$ e $B->C$ são consideradas interseções que neste problema em questão são ignoráveis, se relativamente á direção de ambos os segmentos os dois pontos do outro segmento se encontram em lados opostos

Exercicio 4)

Agora que graças ao método enunciado no exercicio 3 já somos capazes de encontrar interseções agora falta escolher o metodo para escolher a ordem pela qual as vamos resolver para isso usamos 4 métodos distintos, tendo apenas em comum o facto de todos aplicarem o melhoramento interativo (hill climbing), que consiste em a partir de um "ponto", neste caso um polígono procura sempre um vizinho que melhor até não ser capaz de encontrar qualquer tipo de vizinho com maior qualidade que o corrente, ficando esclarecido que nem sempre é certo que seja capaz de no final obter

um resultado ótimo.

a)

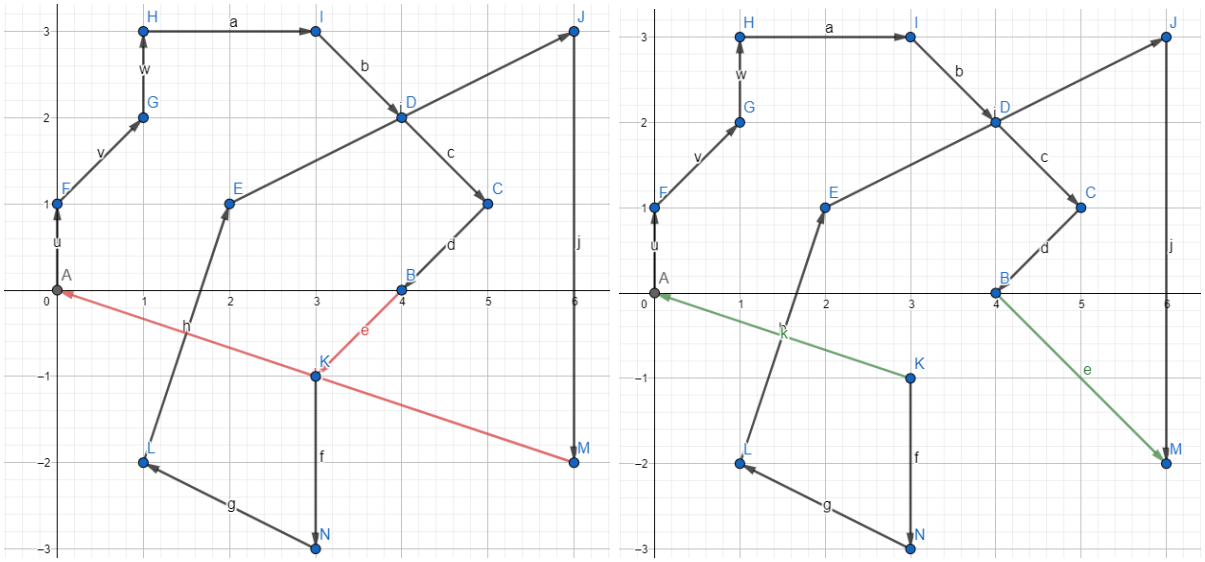
O primeiro método implementado busca resolver a cada iteração entre todas as possíveis interseções que o polígono possa conter aquela que depois da sua resolução resulte num maior melhoramento do perímetro "best-improvement first" na sua próxima fase como se pode ver no seguinte excerto de código:

```
Point k = new Point(0, 0);
Perimetro v = new Perimetro(k,k,k,k);
while(flag){
v.variacao=0;
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 2; j < n; j++) {
            if(i==0 && j==n-1) break;
            if(check_intersection(order[i], order[i + 1], order[j],
   order[j + 1])){
                Perimetro p = new Perimetro(order[i],order[i+1],order[
   j],order[j+1]);
                if(p.variacao>v.variacao){
                    v.variacao = p.variacao;
                    pi = i;
                    pj = j;
                }
                    flag++;
            }
        }
    }
    if(flag > 1){
        espelhar(pi+1,pj,order);
        flag = 1;
    }
    else flag = 0;
}
```

Com esse objetivo em mente percorremos todas as possíveis combinações de segmentos intersetáveis calculando qual seria a variação do perímetro do polígono resultante fase ao original e caso fosse uma alteração que nos fizesse chegar a um estado melhor do que qualquer outro encontrado guardávamos os pontos referentes a essa interseção para mais tarde a resolver.

Resultados:

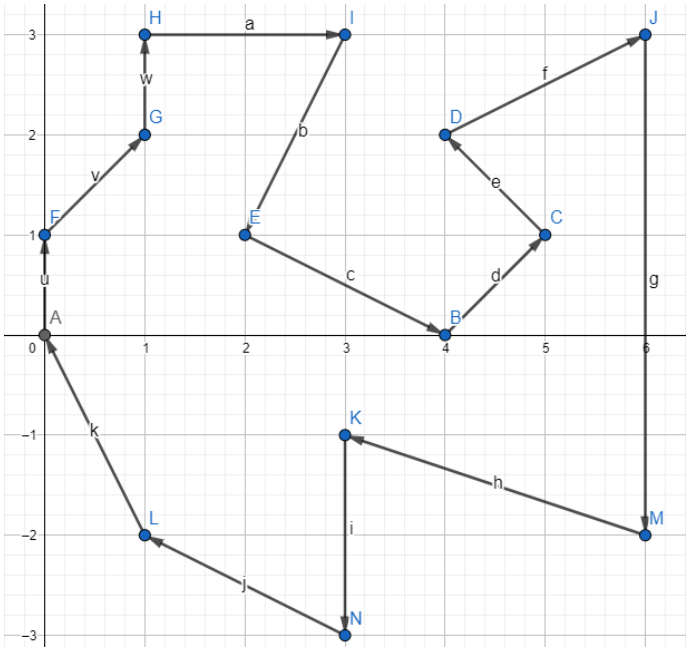
Em seguida é possível a partir das imagens abaixo visualizar qual das interseções encontradas pelo exercício 3, este método escolhe para resolver numa primeira instância juntamente com a sua resolução.



b)

Nesta alínea obtámos por uma implementação greedy da anterior, uma vez que agora nao procuramos qual das interseções é que ao ser resolvida formara o maior melhoramento mais sim aceitar sempre o primeiro melhoramento encontrado "first-improvement" por muito pequeno que seja.

Resultados:



É possível através da imagem acima visualizar o polígono final obtido através deste método que ao contrario do exercício anterior resolve as intersões no momento em que é comprovado que a sua "destruição" ira resultar num melhoramento do perímetro do polígono

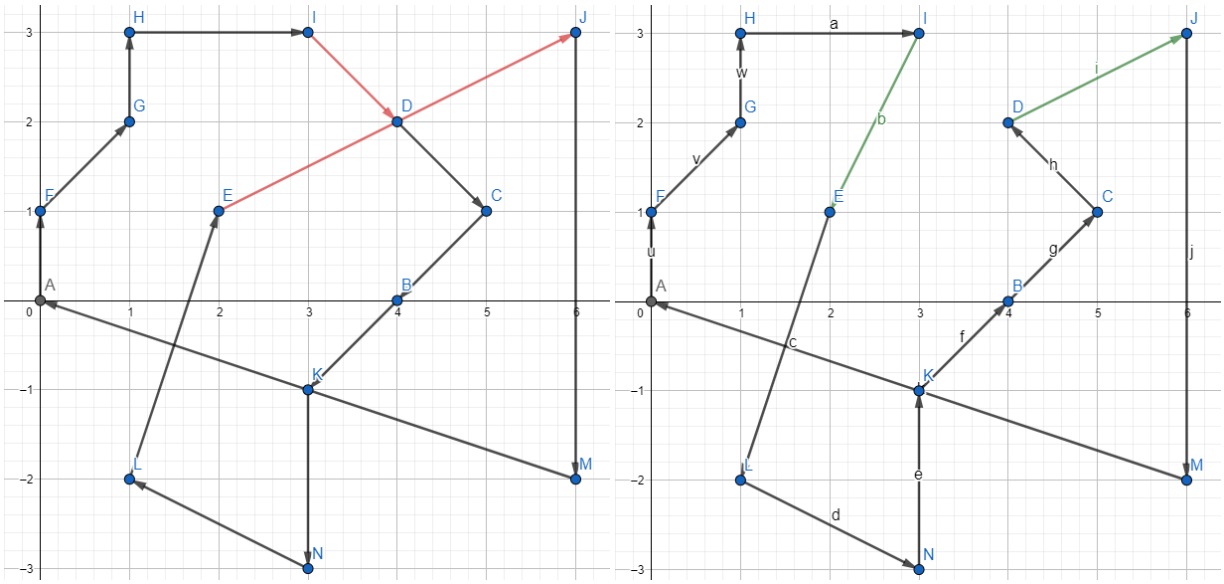
c)

Desta vez escolhemos uma estratégia que ao contrário das anteriores não depende de forma alguma do perímetro mas sim da quantidade de interseções que seriam geradas pelo poligono seguinte. No código deste exercício é usada a mesma estrutura do exercício 4a) á exceção da condição apresentada abaixo que resolve especificamente o problema anunciado neste exercício.

```
1 if (check_intersection(order[i], order[i + 1], order[j], order[j + 1])
    ) {
2     espelhar(i+1,j,order); // funcao que resolve a intresecao p(i) ->
    p(i+1) com p(j) -> p(j+1)
```

```
3     int y= colisions(order,n); // numero de colisoes depois de
      quebrada a intresesao
4     espelhar(i+1,j,order); // retoma ao estado inicial
5     if(y<maxcol){ // obter o menor numero de colisoes e guardar sobre
      que retas aconteceu
6          maxcol = y;
7          pi = i;
8          pj = j;
9     }
10 }
```

Resultados:



Tal como na alínea a) nesta percorremos o polígono na totalidade de forma a obter o número de interseções com que ficaria o polígono resultante da resolução de cada uma das existentes e guardar os pontos que fazem parte daquela que diminuiria mais esse numero. Como podemos comprovar nas imagens existem bastantes interseções que seriam consideradas a melhor escolha para a resolver neste caso o nosso programa seleciona a primeira melhor interseçao que encontrar.

d)

Criamos ainda para finalizar mais uma implementação que nao tem qualquer tipo de critério de escolha da ordem pela qual vai resolver as interseções. Para isso percorremos novamente todos os conjuntos de pontos intersetáveis e guardamos numa lista todos os conjuntos de 4 pontos que criavam interseções, a seguir escolhemos aleatoriamente um desses conjuntos para resolver a interseção por eles criada e repetimos este processo até eliminar todas as interseções.

Exercicio 5)

Desta vez iremos por em pratica melhoramentos a partir da aplicação de "simulated annealing" um algoritmo inspirado na industria metalúrgica que caracteriza-se por um processo controlado em que um metal é aquecido acima da sua temperatura de recristalização e arrefecido lentamente, este processo reduz a dureza do metal e modifica a sua estrutura interna de forma a que se torne muito mais resistente a deformações. Este processo da origem a um metal com maior elasticidade e consequentemente mais fácil de trabalhar.

Concetualmente "simulated annealing" esta associado a estados de energia, e como existe uma tendência ao prosseguir com o algoritmo que o resultado que seja obtido com um menor estado de energia contrastante com o elevado estado de energia obtido no inicio, a variação entre dois estados de energia representada por ΔE, quando obtemos uma variação $E_{Proximo} - E_{Atual} < 0$ significa que iremos transitar para um estado melhor que permitiria chegar a uma solução ótima, caso contrario $\Delta E > 0$ significa que estaremos a analisar uma solução pior que só será aceite conforme a formula de probabilidade que se segue:

$$P(\Delta E) = e^{\frac{\Delta E}{T}}$$

onde T representa e temperatura atual.

Repare-se que ao diminuir a temperatura com o decorrer do algoritmo, a probabilidade de aceitar vizinhos piores reduz proporcionalmente, por isso é necessário numa

fase inicial começar com uma temperatura alta de forma a começar por procurar num maior espaço por soluções ótimas e tender mais tarde para não se afastar dessas soluções ideais ao se concentrar numa menor área onde esteja o nosso objetivo.

Como dito anteriormente, este processo foi usado como inspiração para formular um algoritmo de otimização que implementamos neste exercício, desta forma criamos um método que procura um um extremo global para resolver este problema.

```
static int r_colisions(Point[] order,Point[] temp,int n){
int col= 0;
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 2; j < n; j++) {
            if(i==0 && j==n-1) break;
            if(check_intersection(order[i], order[i + 1], order[j],
   order[j + 1])){
                col++;
            }
            if(check_intersection(temp[i], temp[i + 1], temp[j], temp[
   j + 1])){
                col--;
            }
        }
    }
    return col;
}
```

Aqui neste excerto de código temos uma função que recebe tanto o polígono original como um polígono "vizinho" resultante da resolução de uma interseção do original calculando assim a diferença do numero de interseções de ambos para mais tarde esse valor fazer parte da ordem de testes para aceitar ou não esse vizinho.

```
private static double temperature = 1000000000;
private static double coolingFactor = 0.995;
public static double probability(int v_E, double temp) {
    if (v_E > 0) {
        return 1;
```

```
6       } else{
7           return Math.exp(v_E / temp);
8       }
9   }
```

A função acima foi usada para fazer o calculo da probabilidade com recurso a variação do numero de colisões antes e depois da remoção da interseção representada por "v_E", uma vez que é calculado a partir de Nº de Interseções Originais - Nº Interseções Após a Mudança, caso esta variação for maior que zero significa que a interseção vai sempre passar na condição de verificação que temos depois:

$$Math.random() < probability(r\_colisions(order, temp, n), t)$$

uma vez que $Math.random()$ vai gerar um valor aleatório num intervalo $[0, 1[$, caso contrario significa que vamos considerar a passagem para um estado pior de acordo com a formula mencionada anteriormente.

Conclusão

A partir de todos os resultados obtidos ao longo dos exercícios anteriores é nos possivél tirar algumas conclusões tanto face a noção de melhor resposta final quanto ao tempo de resolução do problema. Por exemplo se observarmos com maior atenção o exercício 2 a sua primeira implementação obterá o melhor resultado quando a questão for tempo de execução porém a segunda ira resultar num caso muito mais próximo do final pois será capaz de evitar um grande numero de interseções e já encontrar alguma interseções "perfeitas" para um polígono de menor perímetro possível. Da mesma forma são bastante comparáveis ás alineas a) e b) do execício 4 pois a primeira terá um maior custo tanto de tempo quanto de memoria, pois para cada iteração terá de guardar e conferir diversas interseções ao contrário da alínea seguinte porém irá incentivar a uma melhor resposta final. Da mesma forma qualquer das alíneas do exercício 4 poderiam da mesma forma ser comparadas á implementação do exercício 5, que graças ao facto de aceitar a transição para um vizinho pior com intuito a descobrir se reorganizando

completamente o polígono nao sera possivel encontrar algum polígono melhor mesmo demorando mais tempo.

Referências

[1] Line Segment Intersection, https://www.youtube.com/watch?v=R08OY6yDNy0