Relatório IA

Paulo Ferreira 201804977 João Ribeiro 201503706

April 6, 2021

Introdução

Este Trabalho tem como objetivo explorar diferentes estratégias, (**construtivas** e **perturbativas**) para encontrar candidatos à solução do problema do gerador aleatório de polígonos simples (sem arestas que se intersetem), que cria um polígonos simples a partir de um conjunto de pontos no plano. Que é uma aplicação estratégia para a resolução do problema Travelling Salesman.

Classes implementadas

- **Ponto** - Usada para guardar cada um dos pontos, tem como atributos as coordenadas de um ponto no espaço (x,y), e implementa métodos para verificar igualdade e gerar um hash.
- **Aresta** - Usada para guardar cada uma das arestas, tem como atributos o índice dos dois pontos que a definem, e implementa métodos para verificar a igualdade e gerar um hash.
- **Node** - "Estado" do problema, tem como atributos a permutação de pontos na forma de array de arestas, perimetro e número de cruzamentos, de forma a auxiliar as várias consultas destes valores ao longo do programa.

1.Geração Aleatória de Pontos

Métodos Usados

- **void generatePoints()** - Gera n objetos Ponto, de coordenadas aleatórias obtidas pela função *getRandomNumber*, certificando-se que não existem 3 pontos colineares, recorrendo à função *collinear*. Guarda os pontos no array *arrayPontos*, que é mantido para consulta durante toda a execução do programa.
- **int getRandomNumber(int m)** - Retorna um número aleatório no range entre -m e m.
- **boolean collinear(int x1, int y1, int x2,int y2, int x3, int y3)** - Recebe três pontos e retorna se são ou não colineares, recorrendo à propriedade da área do triângulo formado por estes, se esta for nula, são colineares.
- **void printPontosGerados()** - Imprime o índice e as coordenadas de cada ponto guardado no array *arrayPontos*.

Descrição

É pedido ao utilizador que introduza o range de coordenadas m, pretendido para os pontos, e o número de pontos n. Para gerar pontos aleatoriamente com coordenadas entre o range [-m,m], é usada a função *generatePoints*, que recorre à função *getRandomNumber*, criando objetos da classe Ponto e guardando-os no array *arrayPontos*, que é mantido inalterado e usado como referência durante todo o programa, servindo o índice de cada ponto no array como o seu identificador, estes n pontos e suas coordenadas são apresentados no início do output.

2.

a) Gerar uma permutação qualquer dos pontos

Métodos Usados

- **void generateRandomPermutation()** - Gera uma permutação aleatória de pontos, na forma de índices, guardados em caminhoIndex, que ref-

erenciam os pontos guardados em *arrayPontos* e imprime os índices desta permutação.

- **void arestasConversion()** - Converte a permutação de pontos contida em *caminhoIndex* para um conjunto de arestas, guardadas no array de Arestas *arestas*.

- **void printArestas()** - Imprime as arestas contidas no array *arestas* na forma *índice da aresta: (índice ponto 1, índice ponto2)*. É usada na função "arestasConversion".

Descrição

A função *generateRandomPermutation*, gera uma permutação aleatória de índices que referenciam os pontos guardados em *arrayPontos* e transforma-a num array de Arestas, esta permutação é apresentada no output.

b) Aplicar a heurística "nearest-neighbor first"

Métodos Usados

- **void generateNNFpermutation()** - Gera a permutação de pontos obtida pela heurística *"nearest neighbor first"* aplicada ao conjunto de pontos contido em *arrayPontos* na forma de um arrayList *caminhoIndex*, que contem por ordem de visita o índice de cada ponto. Gera um índice aleatório a partir do qual começa a pesquisa, e até esgotar todos os pontos do conjunto, adiciona ao caminho o ponto mais próximo do anterior. Recorre ainda a um HashSet *pontosUsados*, que guarda os pontos já usados no caminho, de forma a evitar que se visite um ponto já visitado. Termina com uma chamada à função *printNNFpermutation*.

- **void printNNFpermutation()** - Imprime a permutação gerada pela heurística *"nearest neighbor first"*, contida no arrayList *caminhoIndex* na forma de indices dos pontos, pela ordem em que são visitados.

- **void arestasConversion()** - Converte a permutação de pontos contida em *caminhoIndex* para um conjunto de arestas, guardadas no array de Arestas *arestas*.

- **void printArestas()** - Imprime as arestas contidas no array *arestas* na forma *índice da aresta: (índice ponto 1, índice ponto2)*. É usada na função "arestasConversion".

Descrição

A função usada para obter a permutação de pontos gerada pela heurística "nearest-neighbor first" é *generateNNFpermutation*. É escolhido um ponto aleatório (*randomIndex*) para primeiro ponto da permutação. De seguida o algoritmo escolhe o ponto mais próximo do último introduzido no "caminho". No final a função *printNNFpermutation* é chamada para imprimir o resultado para o terminal. Esta permutação de pontos é convertida para arestas com a função *arestasConversion* e guardada no array de Arestas *arestas*, e mostrada também no terminal.

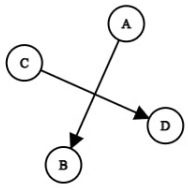
3. Determinar a vizinhança obtida por "two-exchange"

Métodos Usados

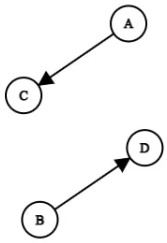
- **void calculateInterceptions()** - Calcula todas as interseções de arestas para o candidato inicial, guardando-as no ArrayList *interceptions*. Percorre todos os pares de arestas, verifica se são arestas distintas, e se não são adjacentes (de modo a evitar alguns casos particulares), caso reunam estas condições, é verificado se se intersetam através da função *line_intersection_old*, imprime o índice das arestas em que seja detetada interseção, e mantem ainda um HashSet de arestas já marcadas para evitar que a mesma interseção seja detetada mais do que uma vez.

- **Aresta[] twoExchange (Aresta[] arestas, ArrayList⟨ArrayList ⟨Integer⟩⟩interceptions)** - Aplica two exchange para resolução de interseção de arestas à última interseção contida no Arraylist de Arraylisits *interceptions*, que recebe. Para uma interseção das arestas (A,B) e (C,D), susbtitui por (A,C) e (B,D), e inverte as arestas entre estes, verifica de seguida se esta troca é válida, ou seja, se as arestas novas já existem ou se criam dois ciclos separados no caminho. Se concluir que a troca não é válida, faz a troca para (A,D) e (C,B), in-

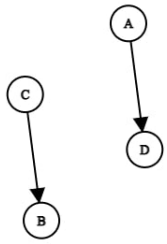
vertendo também as arestas entre elas. Devolve um array de arestas com as trocas efetuadas, correspondente a cada filho.



Interseção (A,B) , (C,D).



Aretas (A,C) , (B,D) obtidas por two-exchange.



Aretas (A,D) , (C,B) obtidas por two-exchange.

Descrição

É usada a função "calculateInterceptions" que calcula as interseções da permutação. Aplicam-se o "two-exchange" para cada interseção da permutação, gerando assim um filho por cada interseção "resolvida", adicionando-o à linkedList de filhos da permutação original.

4. Aplicar o *hill climbing*

a) candidato com menor perímetro ("best improvement first")

Métodos Usados

- **void HCperimetro (Node pai)** - Aplica o algoritmo de hill Climbing recursivamente a um nó inicial, até encontrar uma solução, a escolha greedy que faz escolhe o candidato com menor perímetro. A cada chamada, gera os filhos resultantes de two-exchange a partir do nó pai, e determina qual deles tem menor perimentro, chamando-se a ela própria para este nó. Imprime a solução na forma de arestas quando a encontra.

- **ArrayList⟨ArrayList ⟨Integer⟩⟩calculateInterceptionsFilhosList (Aresta[] arestasf)** - Semelhante à função *calculateInterceptions*, calcula as interseções a partir de uma lista de arestas *arestasf*, é usada para o calculo das interseções de nós, devolve uma lista de interseções.

- **int calculateInterceptionsFilhosSize(Aresta[] arestas)** - Semelhante à função *calculateInterceptions*, calcula o número de interseções a partir de um array de arestas, usado para calcular as interseções dos nós filhos.

Descrição

Para cada candidato, usando a função *HCperimentro* percorre todos os filhos calculando o perímetro de cada um e escolhendo aquele com com o menor perímetro("bestPerimetroIndex") para se tornar o próximo nó candidato na continuação do hill climbing. Imprime a solução na forma de arestas quando a encontra.

b) "first-improvement"

Métodos Usados

- **void HCfi (Node pai)**- Aplica o algoritmo de Hill Climbing recursivamente a um nó inicial, até encontrar uma solução, a escolha greedy que faz escolhe o primeiro candidato da vizinhança. A cada chamada, gera os filhos resultantes de two-exchange a partir do nó pai e escolhe o primeiro, chamando-se a ela própria para este nó. Imprime a solução na forma de arestas quando a encontra.

- **int calculateInterceptionsFilhosSize(Aresta[] arestas)** - Semelhante à função *calculateInterceptions*, calcula o número de interseções a partir de um array de arestas, usado para calcular as interseções dos nós filhos.

- **ArrayList⟨ArrayList ⟨Integer⟩⟩calculateInterceptionsFilhosList (Aresta[] arestasf)** - Semelhante à função *calculateInterceptions*, calcula as interseções a partir de uma lista de arestas *arestasf*, é usada para o calculo das interseções de nós, devolve uma lista de interseções.

  Descrição

  Para cada candidato, usando a função *HCfi* seleciona o index do primeiro filho que corresponde ao primeiro candidato, para se tornar o próximo candidato na continuação do hill climbing. Imprime a solução na forma de arestas quando a encontra.

c) candidato com menos conflitos de arestas

Métodos Usados

- **void HCcruzamentos (Node pai)**- Aplica o algoritmo de Hill Climbing recursivamente a um nó inicial, até encontrar uma solução, a escolha greedy que faz escolhe o candidato com menor número de interseções. A cada chamada, gera os filhos resultantes de two-exchange a partir do nó pai, e determina qual deles tem menos interseções, chamando-se a ela própria para este nó. Imprime a solução na forma de arestas quando a encontra.

- **int calculateInterceptionsFilhosSize(Aresta[] arestas)** - Semelhante à função *calculateInterceptions*, calcula o número de interseções a partir de um array de arestas, usado para calcular as interseções dos nós filhos.

- **ArrayList⟨ArrayList ⟨Integer⟩⟩calculateInterceptionsFilhosList (Aresta[] arestasf)** - Semelhante à função *calculateInterceptions*, calcula as interseções a partir de uma lista de arestas *arestasf*, é usada para o calculo das interseções de nós, devolve uma lista de interseções.

Descrição

Para cada candiato, usando a função *HCcruzamentos*, percorre todos os filhos, calculando o número de interseções de cada um e escolhendo aquele com o menos interseções("bestInterceptionIndex") para se tornar o próximo candidato na continuação do hill climbing. Imprime a solução na forma de arestas quando a encontra.

d) qualquer candidato

Métodos Usados

- **HCrandom (Node pai)**- Aplica o algoritmo de Hill Climbing recursivamente a um nó inicial, até encontrar uma solução, a escolha greedy que faz escolhe o candidato um qualquer candidato na vizinhança. A cada chamada, gera os filhos resultantes de two-exchange a partir do nó pai e escolhe um qualquer, chamando-se a ela própria para este nó. Imprime a solução na forma de arestas quando a encontra.

- **int calculateInterceptionsFilhosSize(Aresta[] arestas)** - Semelhante à função *calculateInterceptions*, calcula o número de interseções a partir de um array de arestas, usado para calcular as interseções dos nós filhos.

- **ArrayList⟨ArrayList ⟨Integer⟩⟩calculateInterceptionsFilhosList (Aresta[] arestasf)** - Semelhante à função *calculateInterceptions*, calcula as interseções a partir de uma lista de arestas *arestasf*, é usada para o calculo das interseções de nós, devolve uma lista de interseções.

Descrição

Para cada candidato, usando a função *HCrandom*, seleciona um filho aleatório para se tornar o próximo candidato na continuação do hill climbing. Imprime a solução na forma de arestas quando a encontra.

5. *Simulated annealing*

void SA(Node og) - O lgoritmo de AS começa por dar um valor à temperatura que é decrementado segundo uma redução geométrica (t=t*a). A cada itereção a diferença de energia (Edif) é atualizada para a diferença das interseções do próximo filho e do corrente. O nó corrente passa a ser o filho se a diferença for positiva ou segundo uma probabilidade determinada pelo exponencial da diferença a dividir pela temperatura.

$$P(difE) = e^{\frac{Edif}{T}} \tag{1}$$

[1]

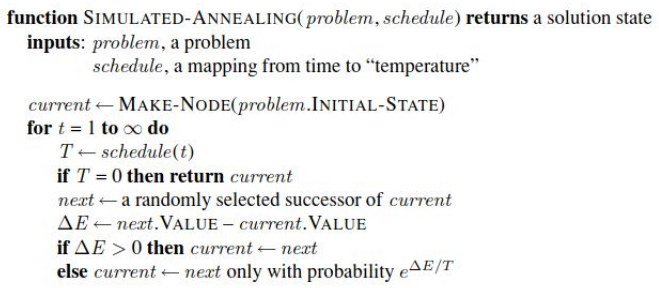


Figure 1: .
Pseudo Código Simulated Annealing.

Assim quanto maior for a temperatura e menor for a diferença mais provável é ele optar por uma solução pior.

Resultados Experimentais

De modo a comparar a performance das diferentes heurísticas para cada tipo de cadidato inicial, fez-se um conjunto de 1000 testes para cada combinação

candidato/heurística com *n=50* e *m=50*, e usou-se como medida de performance a média de iterações hill climbing. Os ficheiros de texto com os resultados completos encontram-se no arquivo.

	"best-improvement first"	"first-improvement"	menos cruzamentos	aleatório
Random	40.3	85.739	81.503	74.926
"NNF"	6.334	6.412	6.118	6.673

Table 1: Média de iterações para cada combinação candidato inicial/heurística hill climbing.

Permutação inicial

Verificou-se que gerar o candidato inicial através da técnica "nearest neighbour first", resulta de forma geral em significativamente menos interseções face a um candidato gerado aleatoriamente. O que se traduz numa redução dramática do número de iterações hill climbing necessárias para chegar a uma solução, independentemente da heurística utilizada.

Heurísticas hill climbing

Para candidatos aleatórios, verificou-se que a heuristica "best-improvement first", é significatiamente melhor do que as restantes, que apresentam performance idêntica.

Já em candidatos gerados por "nearest neighbour first", não se verificou uma diferença significativa entre as heurísticas. É possível que isto se deva a uma homogeneidade da vizinhança, minimizando os ganhos obtidos em escolher um melhor candidato ao passo que a partir de um candidato aleatório, é gerada uma vizinhança mais diversa em que o ganho em escolher o melhor candidato, é maior.

Simulated annealing

Segundo a fórmula dada podemos deduzir que a probabilidade de o algoritmo tomar uma decisão mais uma arriscada aumenta quando a temperatura ainda tem valores elevados ou quando a diferença de energia(neste caso diferença de número de cruzamentos) é menor. O esquema de arrefecimento usado pode ainda mudar o ritmo a que a temperatura decresce, sendo que diferentes

regras são melhor optimizadas para certos modelos. Um exemplo de outra fórmula possível de se usar é a de redução linear(t=t-a).

Erros e Bugs

Se for introduzida uma combinação m, n de tal forma que não seja possível criar n pontos distintos, ou certificar que não há 3 pontos colineares, o programa entrará em ciclo e deve ser parado. Apesar de entendermos o algoritmo de Simulated Annealing não o conseguimos implementar corretamente, deixando-o incompleto. Pelo que decidimos mantê-lo no programa mas não analisar os resultados obtidos. O exercício 6 também não foi resolvido.

Distribuição e Colaboração

O trabalho foi realizado na sua totatidade em conjunto pelos dois elementos, e sem colaboração com outros grupos.

Conclusões

Através da análise dos resultados experimentais, pode-se concluir que gerar o candidato inicial por "nearest neighbour first" é extremamente vantajoso, traduzindo-se numa pesquisa muito mais sucinta pela solução.

References

[1] S.Russell and P.Norvig. *AIMA: Artificial Intelligence - A Modern Approach* https://cs.calvin.edu/courses/cs/344/kvlinden/resources/AIMA-3rd-edition.pdf