
Faculdade de Ciências da Universidade do Porto

---

**Inteligência Artificial (CC2006) - 2020/2021**

# Geração de Polígonos Simples com Métodos de Pesquisa Local

**Relatório do 1º Projeto de Inteligência Artificial**

---

Fábio Silva up201707021

Mateus Almeida up201805265

# Índice

**Introdução** **2**

**Lista de Ficheiros** **2**

**Geração de Pontos** **4**

**Geração de Candidato à Solução** **4**

**Vizinhança por 2-Exchange** **5**

**Hill Climbing** **5**

Implementação 5

**Simulated Annealing** **6**

Implementação 6

**Ant Colony Optimization** **7**

Implementação 7

**Análise de Resultados** **7**

Hill Climbing 12

Random 12

Less Conflicts First 12

First Improvement 12

Best Improvement First 12

Simulated Annealing 13

Ant Colony Optimization 13

**Conclusão** **14**

# Introdução

Este relatório foi realizado no âmbito do primeiro projecto da unidade curricular de Inteligência Artificial. O seu propósito é estudar diferentes algoritmos de pesquisa local como métodos de procura de polígonos simples, elaborando as vantagens e dificuldades encontrados quanto à sua aplicação.

Define-se como um polígono simples aquele que não possui arestas que se intersetem em qualquer ponto senão nos seus vértices. Na procura desses polígonos, calcula-se os cruzamentos de arestas existentes, utilizando testes baseados no produto vetorial das arestas, e resolve-se-os aplicando a heurística 2-exchange nas arestas que se interseccionam até chegar ao resultado desejado. Cada método de procura irá, depois, providenciar um diferente caminho na chegada à solução, o polígono simples.

A implementação dos algoritmos e classes auxiliares foi feita inteiramente em Java.

# Lista de Ficheiros

O ficheiro Point.java define a classe Point, constituída pelos inteiros x e y, representando as coordenadas de um vértice, o inteiro index, representando o índice com o qual o ponto será referenciado e métodos como equals(), compareTo() e toString().

O ficheiro Edge.java define a classe Edge, constituída pelos inteiros origin e dest, representando os índices dos dois vértices constituintes da aresta, e o inteiro distance, o quadrado da distância euclidiana entre eles, tal como os métodos equals() e toString().

O ficheiro Memory.java define a classe Memory, constituída por um array de pontos points[], que guarda as coordenadas de cada ponto dado o seu índice, e os inteiros memSize, o número total de pontos, e nPoints, o número atual de pontos, tal como os métodos add() e contains()..

O ficheiro Solution.java define a classe Solution que representa qualquer candidato à solução. É constituída pelo array de inteiros sol[], que contém os índices dos pontos do candidato à solução, o array de Edges edges[], que contém as arestas do candidato à solução, e os inteiros edgeSol, solSize, edgeMaxSize e solMaxSize, o número atual de arestas e vértices e o número máximo de arestas e vértices, respetivamente. Contém também os inteiros totalConflicts e validConflicts, que representam o número de interseções encontradas no candidato e o número de interseções válidas, respetivamente, tal como as ArrayLists conflicts, que contém todos os cruzamentos de arestas, e neighbours, que representa a vizinhança gerada do candidato. Por fim, possui uma cópia do conjunto de pontos Memory. Estão também implementados métodos para a implementação dos diferentes algoritmos.

O ficheiro HillClimbing.java contém a implementação do algoritmo Hill Climbing tal como os métodos auxiliares para a resolução do mesmo. Tais são:

- hillClimbing - a implementação do algoritmo Hill Climbing. Dado um candidato, gera a sua vizinhança e encontra um novo candidato segundo o critério escolhido, melhorando o candidato ao comparar os perímetros dos dois.

- chooseFunction - chama o método auxiliar pedido segundo o input.
- getNewRandom - retorna uma solução aleatória dentro da vizinhança gerada.
- firstImprovement - retorna a primeira solução encontrada cujo perímetro é melhor do que o do candidato a ser analisado.
- lesserConflicts - retorna a solução com o menor número de conflitos encontrado na vizinhança.
- bestImprovementFirst - retorna a solução encontrada que contém o melhor perímetro dentre aquelas na vizinhança.

O ficheiro SimulatedAnnealing.java contém a implementação de Simulated Annealing, tal como os métodos auxiliares para a sua resolução. Estão definidos como constantes os valores numIterations, o número de iterações em cada temperatura; Tmin, a temperatura mínima a atingir como condição de paragem, e alpha, o cooling factor. Os métodos implementados são:

- getNewRandom - o mesmo método que em HillClimbing, retorna uma solução aleatória dentro da vizinhança gerada.
- simulatedAnnealing - a implementação do algoritmo Simulated Annealing. Dado um candidato inicial, escolhe-se um vizinho aleatório que o substituirá como candidato à solução caso seja melhor, sendo o custo melhor aquele com menos interseções de arestas; caso não o seja, substitui-lo-à segundo uma probabilidade calculada. A procura pára ao encontrar um polígono simples ou quando a temperatura desce para além da temperatura mínima permitida.

O ficheiro AntColonyOptimization.java contém a implementação de Ant Colony Optimization, tal como os métodos auxiliares para a sua resolução. A instanciação da classe inicializa as variáveis alpha, beta, evaporation, a taxa de evaporação das feromonas, e pheromone, a matriz de ramos que contém a feromona atual de cada, em que cada entrada [i,j] representa a aresta (i,j). Os métodos implementados são:

- buildPheromone - inicializa a matriz de feromonas dos ramos como 1.0.
- updatePheromone - atualiza as feromonas de cada ramo como a soma do total de feromona que a formiga k encontrou, calculado com getTotalPheromone(), e o produto da feromona atual do ramo com $(1-p)$ sendo $p$ a taxa de evaporação.
- getTotalPheromone - calcula o total de feromona a adicionar num segmento, que depende do número total de formigas que passam nesse mesmo segmento e da qualidade do caminho encontrado por cada formiga.
- euclidean - calcula a distância euclidiana de dois pontos.
- buildNewSolution - constrói todas as formigas, que são representantes de cada candidato à solução, utilizando a probabilidade de utilização de cada ramo com probability() durante a sua construção.
- probability - calcula a probabilidade de uma dada formiga k usar um ramo dado a quantidade de feromona em cada ramo e o peso de cada.
- findBestPath - constrói todas as formigas e atualiza o valor das feromonas de seguida. Para

cada formiga construída verifica-se o número de conflitos, se for zero então a função é terminada, se não encontrar nenhuma formiga com zero conflitos continua a executar.No fim de todas as iterações, se não encontrar nenhuma formiga que representa um polígono simples então procura a formiga com o melhor perímetro e menor número de conflitos e a percentagem de formigas que encontraram essa solução.

O ficheiro Visualizer.java é responsável pela representação gráfica de um polígono dado por um conjunto de coordenadas, neste caso vai desenhar o polígono encontrado como sendo a solução do problema. O programa começa por desenhar um plano cartesiano de coordenadas, e de seguida desenha os vários segmentos que constituem o polígono.

O ficheiro Main.java contém o main do programa onde será selecionado o método de input, sendo ele uma geração de pontos dado um número de pontos e o alcance das suas coordenadas ou um ficheiro já com o número e coordenadas de cada ponto definido. É depois selecionado o método de geração do primeiro candidato, permutação ou aplicação da heurística Nearest Neighbour, seguindo-se da escolha do algoritmo de procura.

Os ficheiros input1.txt, input2.txt, input3.txt, input4.txt, input5.txt são ficheiros exemplo que foram utilizados para o desenvolvimento das implementações e testagem dos seus resultados.

# Geração de Pontos

A geração do espaço pode ser dado pela leitura de um ficheiro que conterá o número total de pontos como primeiro input e as coordenadas de todos esses pontos de seguida. Como alternativa, pode-se gerar também os pontos após a leitura do número total de pontos como primeiro input e do alcance máximo das coordenadas como segundo input.

A geração de pontos é dada no método generatePoints() que utiliza a classe Random para a escolha aleatória de dois inteiros que definem as coordenadas de um ponto, tal que, sendo i e j os inteiros aleatórios gerados e m o alcance máximo das coordenadas, dá-se (i,j) como as coordenadas de um ponto, $-m < i < m$ e $-m < j < m$, que são depois guardados na estrutura Memory. O método garante ainda que não são criados coincidentes, garantido que qualquer ponto gerado não está contido na Memory antes de ser adicionado.

# Geração de Candidato à Solução

A geração do candidato à solução pode ser dada pela geração de uma qualquer permutação dos pontos que compõem o espaço ou pela aplicação da heurística “nearest-neighbour first” a um ponto inicial gerado aleatoriamente.

A permutação de pontos é dada em cada instância de solução Solution, usando os métodos implementados nessa mesma classe. Chama-se o método permutation que cria uma permutação de todos os vértices com auxílio do método shuffle(), que gera cada índice da permutação aleatoriamente com apoio da classe Random,. e gera as arestas correspondentes com auxílio do método edgeGenerate(), que cria as arestas do candidato, segundo a ordem dos vértices no array sol[], por exemplo, dado o array sol do candidato $[x_0,...,x_n]$ criará $\sum_i^n [x_i, x_{i+1}]$.

A segunda abordagem cria um candidato à solução inicial utilizando uma estratégia greedy aplicando a heurística "Nearest Neighbour First", isto é, com auxílio do método nearestNeighbour() cria o candidato vértice a vértice. O método nearestNeighbour() em si retorna o índice mais próximo de um dado índice inicial, calculando a distância euclidiana desse vértice a outros e guardando o mínimo.

# Vizinhança por 2-Exchange

A vizinhança de cada candidato é mantida numa ArrayList neighbours em cada instância de solução Solution. A sua determinação dá-se com o método resolveConflicts() que cria um novo candidato à solução a partir da variável global conflicts, que mantém todas as arestas interseccionadas, e utilizando o método twoExchange para realizar o "descruzamento", para depois adicionar tal candidato à ArrayList da vizinhança, neighbours.

O cálculo das interseções é dado pelo método edgeIntersect() que

Cada cruzamento forma um possível candidato a solução que compõe a vizinhança. O "descruzamento" que leva a essa geração é dado pelo método twoExchange() que dados duas arestas, Edge *a* e Edge *b* ,que formavam uma interseção, cria dois novos ramos, Edge *newA* e Edge *newB* que serão compostos pelo ponto origem de *a* e *b* e pelos pontos destino de a e b, respetivamente, e substitui o Edge *a* por *newA* e o Edge *b* por *newB* em cada índice correspondente, invertendo a ordem ramos entre os dois. Assim, dado, por exemplo, um polígono representado pelos índices de cada ponto ABCDEFGH, em que cada letra representa um ponto e *AB* representa o ramo {A,B}, uma interseção de {B,C} e {F,G} leva à sua substituição pelas arestas {B,F} e {C,G}, respetivamente, e à reordenação das arestas entre eles, resultando no polígono ABFEDCGH.

De seguida, o método sortPath() corrige qualquer desordenamento que pode existir resultante da inversão de posição das arestas. Por fim, atualiza-se a lista de índices sol[] para corresponder à lista de arestas atualizada edges[] com o método changeSolFromEdges().

O método conflicts() calcula o conjunto de todos as arestas que estão cruzadas, isto é, todos os conflitos, e formarão futuros candidatos, mantendo-as na variável global conflicts, iterando todos os pares de segmentos que se podem intersectar, confirmando a interseção com edgeIntersect() e a registando a validade de cada conflito, tal como o total número de conflitos encontrados, nas variáveis validConflicts e totalConflicts, respetivamente.

# Hill Climbing

O algoritmo de pesquisa local Hill Climbing é um algoritmo que converge gradualmente para um máximo, terminando ao encontrar uma solução ou um máximo local, isto é, um candidato à solução que não possui vizinhos que o consigam melhorar.

No contexto do problema, o Hill Climbing converge, a partir de um polígono com *n* cruzamentos, para um polígono simples.

## Implementação

Implementou-se o Hill Climbing dando uso ao candidato inicial gerado por métodos selecionados anteriormente, ou por permutação de todos os vértices ou por procura segundo a heurística greedy,

encontrando os cruzamentos existentes nesse polígono, conflicts(), e gerando a sua vizinhança, resolveConflicts(), que fica guardada numa ArrayList de candidatos solução Solution. Após a geração, e até o número de cruzamentos do dado polígono serem zero ou o número de cruzamentos encontrados, e consequentemente a vizinhança gerada, ser nula, itera-se a solução candidata.

Em cada iteração é selecionado um novo candidato segundo o critério de seleção providenciado pelo utilizador e caso este novo candidato seja melhor que o atual, sendo o atual considerado o melhor candidato até ao momento da comparação e um "melhor candidato" aquele cujo perímetro seja menor, guarda-se o novo candidato para a próximo iteração.

Se o novo candidato não for melhor que o anterior então, devido às características das heurísticas de seleção do novo candidato, presume-se que esse fosse o melhor entre toda a vizinhança, tal estão assim implementados os métodos e, não melhorando o candidato atual, então não existem vizinhos que possam melhorar o candidato atual, isto é, chegou-se a um máximo, e termina-se a procura.

# Simulated Annealing

O algoritmo Simulated Annealing é um método de otimização estocástico que simula o arrefecimento lento de metais, caracterizado pela redução progressiva de movimento atómico, de modo a chegar a um mínimo global de energia interna. Tal como no processo de formação de cristais, o algoritmo reduz a "temperatura" lentamente, demorando tempo suficiente em cada temperatura, de modo a não ficar bloqueado num mínimo local e convergir ao mínimo global.

No contexto do problema, a partir de um polígono com *n* cruzamentos, o Simulated Annealing segue uma probabilidade de aceitação que decresce com a temperatura, convergindo assim para mínimos globais.

## Implementação

Implementou-se o Simulated Annealing seguindo o mesmo padrão da implementação do Hill Climbing, isto é, criando a vizinhança do primeiro candidato, formado a partir de uma permutação dos pontos ou da aplicação da heurística Nearest Neighbour, como a resolução dos conflitos deste candidato obtidos a partir da aplicação da heurística 2-exchange aos cruzamentos de arestas existentes.

Definiu-se a temperatura inicial como o número de conflitos do candidato inicial, a temperatura mínima a alcançar como 0,01, o critério de annealing como 0,95 e o número de iterações em cada temperatura como 5.

Para cada iteração, procura-se um candidato aleatório na vizinhança dando uso ao método getNewRandom() e calcula-se o seu número de interseções. Se a diferença do número de cruzamentos de arestas do novo candidato e do candidato à solução atual, designado num double delta, for negativa, então o novo candidato contém menos interseções que o candidato atual mantido - logo, o candidato novo é melhor e substitui o anterior; se delta for maior ou igual a zero, então calcula-se a função probabilidade densidade que é dada como o exponencial de delta com a temperatura atual. A probabilidade resultante definirá a aceitação do candidato novo.

Caso o candidato não seja aceite então procura-se um próximo candidato na mesma vizinhança. Quando são dadas todas as iterações da temperatura atual, confirma-se se se chegou à condição de paragem - o candidato atual ser um polígono simples - e termina-se a procura seja esse o caso. Caso contrário, a temperatura é reduzida como o produto da temperatura atual com o critério de annealing.

# Ant Colony Optimization

Ant Colony Optimization é uma metaheurística baseada em populações aplicável em problemas de procura em grafos pesados. Agentes chamados de formigas artificiais atravessam o grafo, construindo gradualmente a solução num processo de construção estocástico baseado no modelo das feromonas.

No contexto do problema, cada formiga representa um candidato à solução inicial e ao fim de cada iteração cada formiga construirá um polígono diferente, sendo influenciados pela feromona deixada pelas outras formigas em cada aresta. No final, escolher-se-à o polígono construído mais frequentemente pelas formigas.

## Implementação

Implementou-se o Ant Colony Optimization tomando desde início o número de iterações a realizar e o número de formigas de modo a proceder com a construção dos candidatos à solução.

A construção dos candidatos à solução é realizado no método buildNewSolution() que, para iniciar, escolhe um índice aleatório com auxílio da classe Random para ser o primeiro ponto da solução e constrói os próximos ramos dada a probabilidade da formiga usar esse ramo, isto é, a solução adiciona o ramo com a maior probabilidade de ser usado.

O cálculo de probabilidade realiza-se no método probability() em que, dado um ponto inicial, calcula-se o peso de cada aresta possível como o produto da feromona elevada ao valor de alpha e a distância dos pontos elevada a beta, guardando o somatório de todos os produtos num inteiro, *total*, para o cálculo das probabilidades. As probabilidades são depois calculadas como a divisão do peso pelo *total* e guardadas numa ArrayList total, *prob*. Por fim, calcula-se o total das probabilidades e gera-se um double aleatório, escolhendo o índice da probabilidade mais próxima a esse.

A procura do polígono simples dá-se com o método findBestPath() que, após construir as formigas, em que cada formiga representa um possível caminho, atualiza os valores da feromona com a função updatePheromone(). Nesta função alterámos a função dada no enunciado para atualização da feromona de forma a dar mais peso aos resultados com menos conflitos. Ou seja, $\Delta\boldsymbol{\mathcal{T}}_{ij}^{k} = \frac{Q}{L_k + C}$, em que C é o número de conflitos desse percurso, e Q é uma constante a que demos o valor 1.

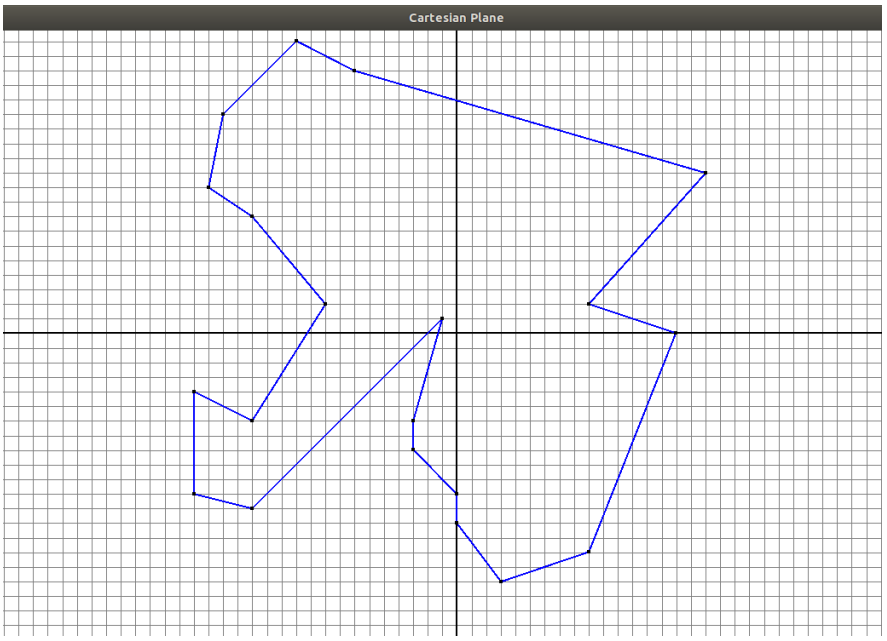
Cada vez que se gera um novo caminho, a solução é testada para ver se chegámos a uma solução sem conflitos, se chegarmos a uma situação sem conflitos imprimimos a solução encontrada e terminamos a procura, caso contrário continuamos a fazer iterações até ao limite máximo definido no input.

Caso a solução não seja encontrada, quando chegamos ao fim imprimimos a melhor solução que foi encontrada

# Análise de Resultados

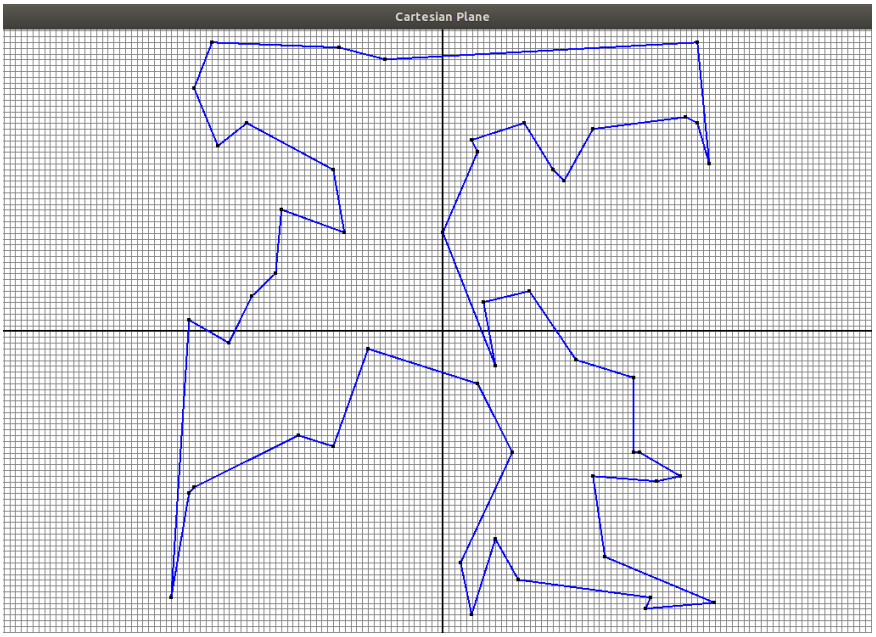
Para propósitos de testagem e análise de resultados, criou-se cinco diferentes ficheiros de input com as respectivas características:

- Input1.txt - 20 pontos com coordenadas (x,y) tal que x,y ∈ [-20,20]



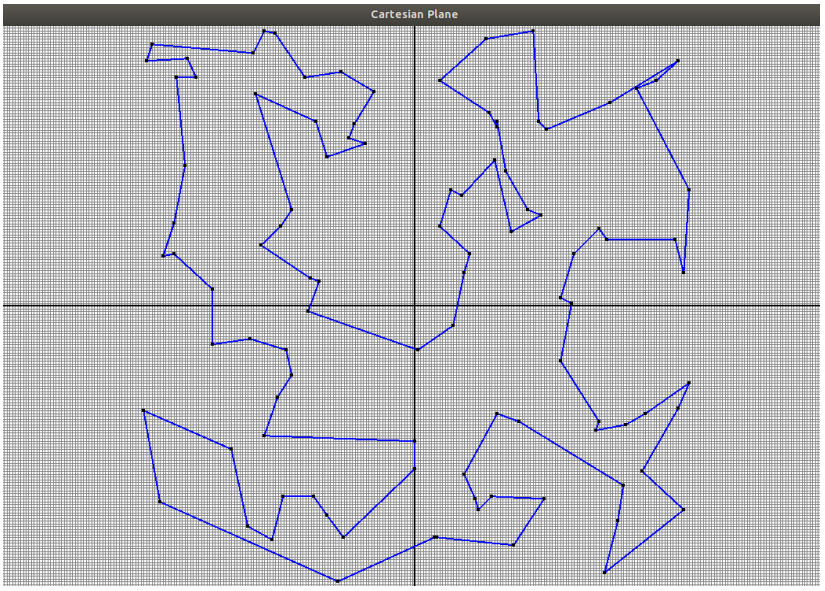

Polígono gerardo com a heurística Best Improvement de Hill Climbing.

- Input2.txt - 50 pontos com coordenadas (x,y) tal que x,y ∈ [-50,50]



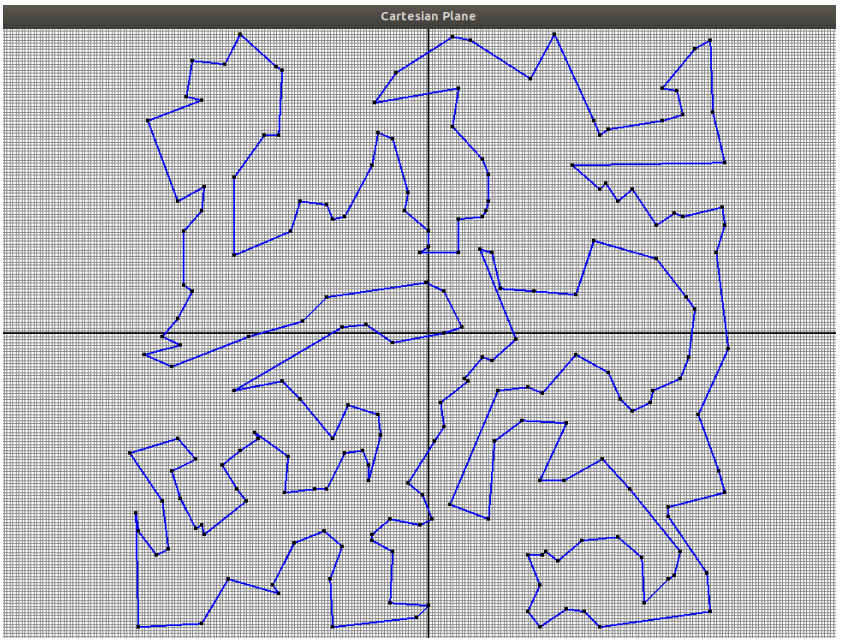

Polígono gerardo com a heurística Best Improvement de Hill Climbing.

- Input3.txt - 100 pontos com coordenadas (x,y) tal que x,y ∈ [-100,100]



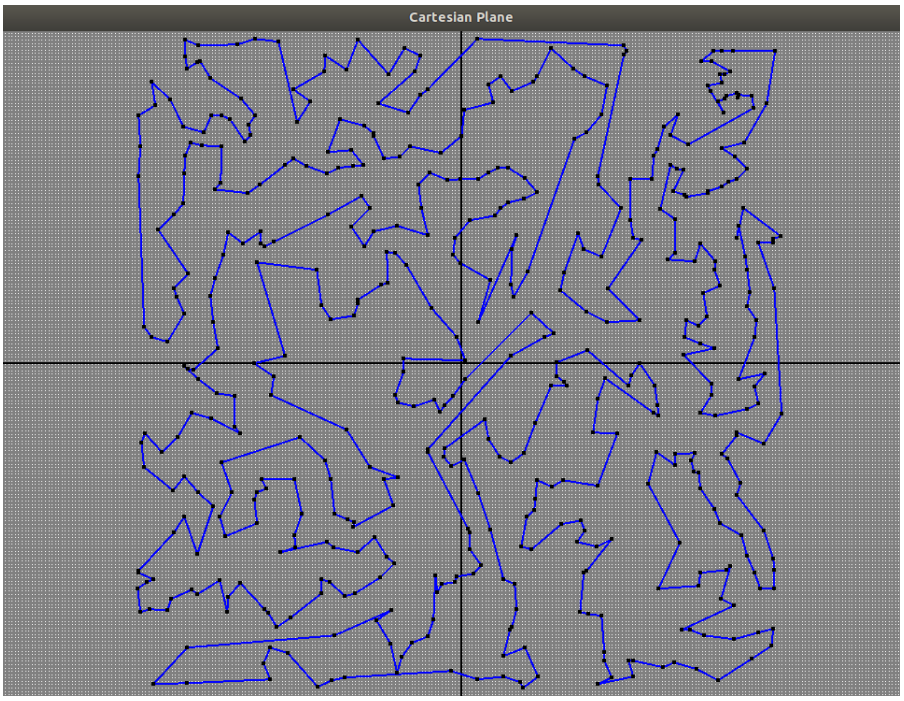

Polígono gerardo com a heurística Best Improvement de Hill Climbing.

- Input4.txt - 200 pontos com coordenadas (x,y) tal que x,y ∈ [-100,100]




Polígono gerardo com a heurística Best Improvement de Hill Climbing.

- Input5.txt - 500 pontos com coordenadas (x,y) tal que x,y ∈ [-250,250]




Polígono gerardo com a heurística Best Improvement de Hill Climbing.

Para consistência entre os polígonos gerados utilizando o algoritmo de geração de candidato Nearest-Neighbour definiu-se o seu ponto inicial como 0, onde normalmente seria gerado aleatoriamente.

De seguida, analisou-se o comportamento de cada algoritmo para com cada input. Visto haver um elemento de aleatoriedade na geração de cada candidato à solução, decidiu-se analisar os resultados dos candidatos gerados por permutação separadamente daqueles gerados pela aplicação da heurística Nearest-Neighbour first.

A avaliação de performance foi realizada sem a chamada do painel Visualizar para a representação gráfica do polígono.

| | Hill Climbing | | | | Simulated Annealing |
|---|---|---|---|---|---|
| | BFI | FI | LC | Random | |
| Input 1 | 0,139s<br>0 Iter | 0,134s<br>0 Iter | 0,145s<br>0 Iter | 0,138s<br>0 Iter | 0,150s<br>0 Iter |
| Input 2 | 0,155s<br>5 Iter | 0,173s<br>5 Iter | 0,165s<br>5 Iter | 0,161s<br>5 Iter | 0,149s<br>5 Iter |
| Input 3 | 0,143s<br>10 Iter | 0,175s<br>9 Iter | 0,235s<br>10 Iter | 0,177s<br>9 Iter | 0,191s<br>9 Iter |

| | | | | | |
|---|---|---|---|---|---|
| Input 4 | 0,273s<br>23 Iter | 0,271s<br>32 Iter | 0,642s<br>23 Iter | 0,341<br>31 Iter | 0,313s<br>31 Iter |
| Input 5 | 0,586s<br>39 Iter | 0,594s<br>45 Iter | 6,081s<br>44 Iter | 0,671<br>47 Iter | 0,945s<br>48 Iter |

Resultados de Nearest-Neighbour First. BFI - Best Improvement First; FI - First Improvement; LC - Less Conflicts; Iter - Iterações

Para eliminar o fator de aleatoriedade do Nearest-Neighbour, definiu-se um nó inicial fixo (nó de índice 0) e realizou-se 5 iterações para cada algoritmo, fazendo a média absoluta dos tempos de execução de cada, selecionando-se como o valor para a tabela aquele que mais se aproximava da média absoluta e o seu número de iterações.

| | Hill Climbing | | | | Simulated Annealing |
|---|---|---|---|---|---|
| | BFI | FI | LC | Random | |
| Input 1 | 0,141s<br>12 Iter | 0,154s<br>21 Iter | 0,182s<br>13 Iter | 0,158s<br>21 Iter | 0,159s<br>13 Iter |
| Input 2 | 0,227s<br>41 Iter | 0,286s<br>77 Iter | 0,603s<br>34 Iter | 0,264s<br>81 Iter | 0,252s<br>64 Iter |
| Input 3 | 0,407s<br>86 Iter | 0,945s<br>213 Iter | 1m12s<br>95 Iter | 0,759s<br>199 Iter | 1,111s<br>205 Iter |
| Input 4 | 3,056s<br>207 Iter | 12,933s<br>494 Iter | -- | 8,848s<br>482 Iter | 13,450s<br>490 Iter |
| Input 5 | 2m44s<br>591 Iter | -- | -- | -- | -- |

Resultados de Permutation. BFI - Best Improvement First; FI - First Improvement; LC - Less Conflicts; Iter - Iterações

A avaliação da performance com permutação dos casos iniciais não mexeu com o fator de aleatoriedade da permutação e realizou 10 iterações, seguidos da média absoluta dos mesmos, para obter resultados mais precisos. As células com "--" foram terminadas ao fim de 5 minutos, devido ao número de conflitos ser demasiado grande.

| Ant Colony optimization | | | | |
|---|---|---|---|---|
| Input 1 | Input 2 | Input 3 | Input 4 | Input 5 |
| 0,126s | 2,179s | -- | -- | -- |

Tal como nos casos anteriores, realizou-se 5 testes com cada input e calculou-se a média absoluta do total dos tempos de execução. Aqueles com "--" foram interrompidos ao fim de 5 minutos. Os dados de teste usados foram 10000 iterações, n formigas (sendo n o número de pontos do caso de teste), valor de peso da feromona 1, valor de peso da distância 5 e taxa de evaporação da feromona 0,1.

## Hill Climbing

A implementação do Hill Climbing foi feita o mais standard possível para avaliar apenas as diferentes medidas de seleção do novo candidato na vizinhança.

### Random

A performance do critério Random é, como o nome diz, mais aleatória em comparação com todos os outros, dependendo se os vizinhos selecionados nas iterações fazem melhorias consideráveis ao candidato considerado o melhor na dada iteração. Assim, foi o critério com mais divergências entre os seus resultados, e talvez a média absoluta não demonstre o mais objetivamente a sua performance. Demonstrou uma performance mais fraca com a geração do candidato como uma permutação, os dois fatores de aleatorização podendo resultar em candidatos iniciais com muitos conflitos que não são melhorados rapidamente em cada iteração.

### Less Conflicts First

A performance do critério Less Conflicts foi a que demonstrou ser mais fraca, tendo os tempos mais lentos para os candidatos gerados por Nearest-Neighbour First. Tal é devido a ser necessário calcular os conflitos de todos os elementos da vizinhança para encontrar aquele com o menor número de conflitos, operações que envolvem a chamada de muitos métodos auxiliares. Nos casos de teste que retornou resultado, retornou com menos iterações realizadas que outros critérios.

### First Improvement

A performance do critério First Improvement é, de certa forma, dependente da ordem da vizinhança, podendo retornar uma primeira melhoria que não é aquela com melhor perímetro da vizinhança, resultando em mais iterações. Conseguiu resolver todas as instâncias, excepto o caso de teste de 500 pontos permutado, cuja resolução conseguiria atingir mas para lá dos 10 minutos de tempo de execução.

### Best Improvement First

A performance do critério Best Improvement First melhora onde o First Improvement falha, demorando mais iterações locais a procurar o vizinho com melhor perímetro mas retornando sempre o melhor novo candidato, resultando em menos iterações totais do algoritmo Hill Climbing. Encontra os mesmos problemas que o First Improvement quando confrontado com o caso de teste input5.txt, tendo a capacidade de resolvê-lo mas demorando demasiado a fazê-lo.

## Simulated Annealing

Na escolha dos valores das constantes para o Simulated Annealing, escolheu-se inicialmente definir a temperatura inicial como o número de conflitos, realizar 10 iterações para cada temperatura e ter como temperatura mínima 0,01. O número de iterações não parecia afetar muito os resultados então estabilizou-se como 5 enquanto se realizava testes aos outros critérios.

Ao seguir uma implementação exemplo, experimentou-se colocar o valor da temperatura inicial como 1, mas isso levou a resultados com conflitos. Decidiu-se, então, escolher o valor da temperatura inicial entre o número de conflitos inicial e o número de pontos do teste, realizando-se testes de performance como o seguinte:

| Temperatura inicial | | Valor do Cooling Factor | | |
|---|---|---|---|---|
| | | 0,95 | 0,97 | 0,99 |
| Nº de Conflitos Inicial | | 0,982s // 51 Iter | 1,035s // 47 Iter | 0,940 // 50 Iter |
| Nº de Pontos | | 0,926s // 48 Iter | 1,013s // 51 Iter | 0,899 // 50 Iter |

Tabela exemplo de teste de performance

A tabela acima utilizou o conjunto de pontos do ficheiro input5.txt e o método de geração de candidato a solução Nearest-Neighbour First com nó inicial o de índice 0. Realizou-se também testes semelhantes para a geração de candidato com permutação e com Nearest-Neighbour First nó aleatório que levou à seleção das constantes temperatura inicial como o número de conflitos inicial e o valor do cooling factor como 0,95, visto serem aqueles que levavam a resultados melhor, sendo que na comparação de resultados dava-se prioridade ao tempo de execução e para tempos semelhantes comparava-se o número de iterações executados.

A performance do Simulated Annealing foi gradualmente pior com o escalar do número de pontos, tendo a performance mais fraca com candidatos gerados por permutation, sendo uma das razões para tal o valor da temperatura inicial maior, que leva a mais iterações até se chegar à temperatura mínima. O único resultado que não conseguia computar foi o input5.txt permutado, que continha um número de conflitos iniciais demasiado grande.

## Ant Colony Optimization

Neste algoritmo é importante permitir a exploração de vários caminhos diferentes, de forma a encontrar mais rapidamente uma solução, mas também é importante usar a feromona de forma a aumentar a probabilidade de escolher um caminho que já se descobriu que é bom. O peso de cada um destes aspectos são controlados por duas variáveis, alpha e beta, em que quanto maior for o peso da variável mais influência esse parâmetro tem na probabilidade de escolher esse caminho.

Ao testar vários valores para ambas as variáveis chegámos à conclusão de que um valor para alpha de 1 e um valor para beta de 5 seria uma boa escolha. Desta forma conseguimos ter variedade de escolha de caminhos e ao longo do tempo o peso da feromona vai influenciando mais a escolha.

Outra variável que temos de ter em conta é a taxa de evaporação da feromona de forma a que um caminho que não é usado, por não ser bom, tenha cada vez menos probabilidade de ser usado. Como não queremos que a feromona desapareça demasiado rápido escolhemos 0,1 para esta variável.

Outras duas variáveis importantes são o máximo de iterações que permitimos, ou seja, quanto tempo permitimos que o algoritmo corra até encontrar uma solução, e o número de formigas que vamos usar. Neste caso usámos como valor máximo de iterações 10000, e usámos tantas formigas quanto pontos.

# Conclusão

A geração de polígonos simples, dados como grafos completos que são ciclos de Hamilton, pode ser realizada utilizando diferentes abordagens e algoritmos diferentes. Os métodos de pesquisa local e pesquisa local estocástica implementados demonstraram comportamentos diferentes e claras diferenças de performance.

O algoritmo Hill Climbing demonstrou melhor performance com a heurística Best Improvement First que, embora ocasionalmente demorando mais iterações na chegada à solução, utilizava os benefícios da heurística Less Conflicts de encontrar um vizinho melhorativo sendo a procura baseada no perímetro dos polígonos, uma operação muito mais rápida e computacionalmente menos dispendiosa.

O algoritmo Simulated Annealing em geral realizou mais iterações que o Hill Climbing, não abdicando de encontrar resultados ótimos. O valor das constantes foi encontrado por trial and error com diferentes casos de teste, tendo podido ser encontrado um método mais eficiente de o fazer. Tal como o algoritmo Hill Climbing, Simulated Annealing não conseguiu encontrar uma solução em tempo útil para casos com 500 ou mais pontos.

O algoritmo Ant Colony Optimization é o algoritmo computacionalmente dispendioso e, devido a ser testado com grandes número de iterações, encontrou limitações com casos de teste de 100 ou mais pontos.

Encontraram-se dificuldades em optimizar os algoritmos para instâncias maiores que 500 pontos permutados, sendo que no caso de teste input5.txt o número de conflitos poderia chegar na ordem $>$ 10000. Isso seria atribuído para o alcance mais pequeno das coordenadas dos pontos, isto é, um alcance mais baixo do que o número total de pontos, que causaria um maior número de conflitos e, consequentemente, uma maior dificuldade de geração de pontos e de encontro e resolução destes conflitos.

No entanto, independentemente do número de vértices de polígono, um dos fatores que se revelou mais importante na performance dos algoritmos foi o candidato inicial de procura. A heurística Nearest-Neighbour First, embora computacionalmente mais lenta e custosa que a permutação dos pontos, melhorou a procura substancialmente.