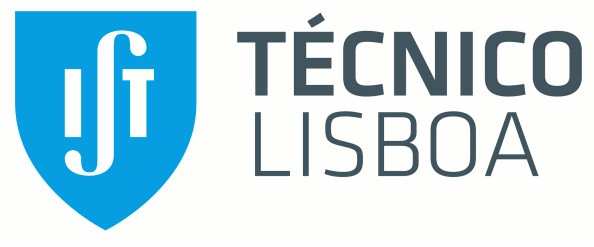

Programação Orientada a Objetos

Travelling salesmen problem by ant colony optimization

Grupo N.º 20

Autores:

André Meneses 84005
Gonçalo Cunha 84062
Leandro Almeida 84112

Professor:

Alexandra Carvalho

9 de Maio de 2019

1 Introdução

O programa a desenvolver em `Java` consiste numa simulação de uma colónia de formigas, durante um intervalo de tempo estabelecido no ficheiro de entrada, de modo a encontrar o menor ciclo que passe por todos os nós apenas uma vez num grafo pesado (ciclo de *hamilton*). Cada vez que uma formiga faz um ciclo de *hamilton* aumenta o valor das feromonas de cada aresta do caminho deste ciclo, que posteriormente vão sendo decrementadas.

O modelo da simulação utilizado será discreto e estocástico, o que significa que será simulado a partir de uma sequência de eventos simulados, em que os tempo entre certos eventos seguirão leis aleatórias.

Pretende-se que todo o código apresente a maior extensibilidade possível, pelo que toda a comunicação entre diferentes identidades foi feita através de uma interface bem definida, pelo que, além de uma camada não saber a implementação da outra, ao se mudar a implementação de uma das camadas, a outra não sabe que se mudou, pois só conhece a interface.

2 Estruturas de Dados

2.1 Grafo

O grafo em análise tanto pode ser um grafo esparso como um grafo denso. Decidiu-se realizar o grafo através de uma matriz de adjacências pois existem imensas procuras no grafo por uma certa aresta ou por um certo vértice, pelo que com esta estrutura de dados a procura tem uma complexidade de *O(1)*, ao passo que com lista de adjacências ter-se-ia uma complexidade média de *O(N)*. Para o grafo com a matriz M, a entrada M_{ij} representa a aresta associada ao nó i e ao nó j. Caso não exista ligação, encontra-se a *null*. A classe *Graph* contém a matriz de adjacências. A classe *Edge* contém a toda a informação da ligação de dois nós (peso da ligação, feromonas). Uma vez que no início do programa se sabe logo o número de nós que o grafo contém, a matriz de adjacências é criada logo ao início e inicializada a *null*.

2.2 Formiga

Como se sabe o tamanho da colónia de formigas, é criado um *Array* de tamanho estático para a colónia. Cada formiga (classe *Ant*) contém os métodos necessários para escolher seu caminho.

2.3 Caminho

O caminho é representado através de uma *LinkedList* de inteiros (identificador de cada nó). Optou-se por esta estrutura de dados por ser mais rápido para as operações que se pretendem realizar: Na remoção de elementos da lista (utilizado por exemplo quando se deteta um ciclo que não tem todos os nós). A razão da *LinkedList* ter uma manipulação mais rápida, aliada ao facto de não haver necessidade de escolher um elemento da lista específico (em que aí haveria vantagem de um *ArrayList*, levou-nos a optar por uma *LinkedList*.

2.4 PEC

Visto que a simulação é baseada na simulação de eventos consecutivos, é importante existir uma estrutura de dados que contenha os eventos a simular por ordem cronológica. Devido à elevada frequência de operações de inserção e remoção, é necessário que essa estrutura tenha o mínimo de complexidade possível para essas duas operações. Para esse efeito, foi escolhida uma *PriorityQueue*, pois tem complexidade $O(log(n))$ nos dois casos mencionados.

2.5 XML Parser

Para a leitura e interpretação do ficheiro de entrada, utilizou-se o *SAX* para realizar o *XML Parse*, uma vez que não é necessário alterar o ficheiro *XML*(mas apenas ler dele) nem ler elementos já processados, e também porque o *SAX* apresenta um menor consumo de memória que o *DOM*. Verifica-se a estrutura do ficheiro de entrada através de um *DTD* de modo a que todos os parâmetros necessários apareçam no ficheiro de entrada no formato correto e na quantidade pretendida. Toda a execução do *Parse* é implementado na classe *SaxHandler* e *Parse* e é feita a verificação de cada parâmetro, garantindo-se casos como: o número de nós, o instante final,α, β, .. são positivos; não há nós duplicados, identificador do nó ser maior que o número máximo de nós. Outras verificações que são feitas são se não se tenta fazer uma ligação de um nó assim mesmo, e se um nó tem menos que duas adjacências. Neste último caso nunca vai haver um ciclo pelo que se decidiu saír do programa.

3 Extensibilidade e Abstração de dados

Pretende-se que todo o código apresente a maior extensibilidade possível, pelo que toda a comunicação entre diferentes identidades foi feita através de uma interface bem definida, pelo que, além de uma camada não saber a implementação da outra, ao se mudar a implementação de uma das camadas, a outra não sabe que se mudou, pois só conhece a interface.

3.1 Interface SimulationInt e Classe abstrata *SimulationAbs*

Apesar de neste projeto só haver simulação de colónia de formigas, criou-se uma interface para a simulação, SimulationInt de modo a ser extensível para outras simulações diferentes, que necessitam de ter os métodos definidos pela interface como os métodos de uma simulação. Associada a esta interface está também a classe abstrata que a implementa, *SimulationAbs*, que contém variáveis como o tempo atual e o instante final e métodos associados a estas variáveis. Esta classe implementa a interface para simulações que necessitem de indicações temporais. Realça-se que estas variáveis são do tipo *protected* para as classes que as herdarem terem acesso. Assim, qualquer classe (associada a uma simulação que pretender) que herde esta classe abstrata contém já estes parâmetros temporais e tem de implementar os métodos declarados na interface de acordo com o tipo de simulação.

3.2 Interface SimulationAntsInt

Objetivo de fornecer uma interface para a simulação de formigas. Desta forma, pode-se implementar esta interface e desenvolver a simulação de formigas de várias formas.

3.3 Interface GraphInt e Classe Abstrata GraphAbs

A interface do Grafo (GraphInt) contém métodos para aceder a uma determinada aresta (com os parâmetros de entrada que são os dois nós que a formam), método para adicionar uma nova ligação, reduzir/aumentar as feromonas de uma dada arestas, saber o peso, ... A classe *GraphAbs* implementa a interface e, contém também variáveis como o número de nós, o ninho, e métodos associados a estas variáveis. Realça-se que estas variáveis são do tipo *protected* para tornar extensível às classes que as herdarem.

Assim, se se quiser fazer um grafo com uma lista de adjacências e outro com matriz de adjacências com classes diferentes sem outras identidades saberem-se herda-se a classe abstrata e implementa-se os métodos definidos pela interface, aumentando-se assim a abstração. Assim uma formiga ao interagir com o Grafo, como comunica com a interface do Grafo, não sabe qual é a sua implementação, pelo que conhece apenas os seus métodos definidos. A formiga não tem como saber se o grafo está implementado com uma lista de adjacências ou com uma matriz, nem se para a constituição do grafo existe ou não a classe *Edge* ou *Node* (que neste caso não existe). Para que esta abstração fosse bem feita, nos métodos declarados na interface do grafo não se retorna nem se tem como parâmetro de entrada nada do tipo *Edge* para não se ter conhecimento do seu "interior". Por exemplo, o método *getPheromone* ou *getEdgeWeight* são para uma dada aresta, mas o parâmetro de entrada são dois inteiros identificadores dos dois nós que compõe a aresta, uma vez que para ser extensível, não se sabe se na implementação existe ou não uma classe aresta, ou nó. O mesmo se aplica quando se pretende obter todas as arestas de um determinado nó: neste caso recebe-se como argumento de entrada um inteiro identificador do nó, e retorna-se um vetor de inteiros que representa os nós a que este se encontra ligado. Ao se pretender saber informações desta ligação (peso, feromonas,...) acede-se aos métodos definidos na interface, com os parâmetros de entrada a serem os identificadores dos nós. Quando se criou uma instância do grafo declarou-se a variável como do tipo GraphInt, tendo-se assim uma maior abstração.

3.4 Interface PathInt e Classe Abstrata PathAbs

O PathInt é a interface de comunicação da formiga para guardar, e manusear todo o seu caminho. A interface do Path tem métodos que comunicam com o grafo (através dos métodos da interface do grafo), no entanto a formiga não tem conhecimento desses métodos pelo que não sabe como comunica. A formiga tem apenas acesso aos métodos disponibilizados pela interface do Path. A classe abstrata *PathAbs* tem as variáveis de custo do caminho assim como uma que indica se é um ciclo. A interface assim como a classe abstrata permitem ter uma maior abstração, pelo que se pode herdar a classe abstrata obrigando-se a implementar os métodos da interface da meneira que o utilizador pretender para o seu caminho.

3.5 Eventos

Numa simulação discreta, é essencial identificar os eventos relevantes para modelar o sistema. Foram identificados os eventos

- Movimento de uma formiga
- Evaporação das feromonas de uma aresta do grafo
- Observação do sistema

3.5.1 EventInterface e EventAbstract

Para interagir com os eventos evidenciados, definiu-se uma interface *EventInterface*, que permite simular o evento com o método *simulateEvent*, obter o tempo em que o evento será simulado com o método *getTimeStamp*, obter o número de vezes que um certo tipo de evento ocorreu com o método *getEventNr*. Esta interface dá para qualquer tipo de eventos.

A classe abstrata *EventAbstract* implementa a interface definida acima e possui o atributo *timeStamp* com visibilidade *protected*, pois é um atributo comum a todos os eventos, bem como uma referência ao PEC, com a mesma visibilidade, que será usado para organizar os eventos. As classes que representam os eventos identificados em cima herdam esta classe abstrata.

3.5.2 PecInterface

Para organizar todos os eventos da simulação num Pending Event Container, definiu-se a interface *PecInterface*, que disponibiliza um método para adicionar eventos ao PEC, *addEvPEC*, e um método para obter uma referência ao próximo evento a simular, *nextEvPEC*. Desta forma o PEC pode ser implementado de diversas formas, dependendo da estrutura do programa ou da quantidade de inserções e remoções de eventos.

4 Polimorfismo e Visibilidade

O uso de uma interface e uma classe abstrata que implementa a interface permite fornecer polimorfismo pois várias classes podem herdar a classe abstrata e implementam os métodos abstratos de acordo com as especificações da classe, e assim diferentes objetos invocam o mesmo método que apresenta implementações diferentes.

Por exemplo, a interface EventInterface é implementada pela classe abstrata *EventAbstract* (que facilita a implementação da interface adicionando o PEC e o instante de tempo), e contém o método abstrato *simulateEvent* que é implementado por cada classe que herda a classe abstrata (neste caso é o EvAntMove, EvPheromoneEvaporation e EvPrintCycle). Quando do PEC é escolhido um destes três tipos de eventos invoca-se o método *simulateEvent* que é determinado no momento, em *runtime*, pelo tipo de objeto (de um daqueles três) que o invocou. O mesmo raciocínio é usado para o caso da simulação, do Grafo e do Path. A simulação está extensível a diferentes tipos de simulações. A interface da simulação e a classe abstrata apresentam um método *simulation*. Cada classe que herde esta classe abstrata, implementa este método de acordo com o tipo de simulação pretendida, e sabe-se qual dos métodos a correr em *runtime* de acordo com

o objeto que invocou o método. O grafo está extensível a diferentes tipos de grafo, em que uma classe grafo que herde a classe abstrata *GraphAbs* implementa à sua maneira métodos abstratos como *addAdjacent* ou *getPheromone*.

Concluí-se assim que se encontra polimorfismo em todos os métodos declarados numa interface sempre que esta é implementada por uma classe abstrata e os métodos sejam abstratos pois várias classes podem herdar esta classe abstrata e implementar os métodos de acordo com o tipo de classe. Assim, de acordo com o tipo de objeto a invocar o método, a execução do programa muda.

Também é usado polimorfismo como no exemplo da classe Path. Uma vez que esta classe necessita de saber a que grafo se associa, existe um construtor que recebe diretamente a interface do grafo e um outro que recebe a simulação e através desta consegue obter o grafo.

Quanto à visibilidade, todas as variáveis definidas numa classe abstrata são do tipo *protected* para que todas as subclasses desta classe fora do *package* consigam ter acesso e modificar a variável, garantindo-se assim maior extensibilidade. A maioria das outras variáveis de classes que não sejam abstratas são do tipo *private* sendo que se acede a estas através de *getter* e *setter*. Os métodos utilizados nas classes são do tipo *public*, à excepção dos métodos que são utilizados apenas dentro da classe, que são do tipo *private* ou *protected* caso faça sentido serem usados por uma classe que herde esta classe (por exemplo na classe *Ant* existem estes dois casos). A classe *Edge* é do tipo *package* pois é uma classe que só comunica com a classe *Graph* pelo que só faz sentido ser usada dentro do *package*. Os seus métodos são do tipo *package*(à excepção do construtor) e as suas variáveis também são do tipo *package*.

5 Excepções

Foi criado um *package* para as excepções, em que foram introduzidas quatro diferentes excepções: *AttributeLowerThanZero*, *IllegalAttributeChangeTwiceException*, *UnexistingEdgeException* e *UnexistingNodeException*. As duas primeiras, como o nome indica, são excepções lançadas caso alguns dos parâmetros de entrada sejam menores que zero (tempos por exemplo) ou alguns parâmetros de entrada de construtores (por exemplo o número de nós de um grafo), ao passo que a segunda é quando se tenta trocar um atributo duas vezes e este só pode ser definido uma vez. As outras duas são utilizadas para as classes associadas à interface do grafo: Graph e Edge, nos métodos em que se acede a um nó ou a uma aresta, sendo que o programa lança uma. excepção caso se aceda a um método e o parâmetro de entrada seja um nó inválido, ou dois nós sem qualquer ligação.

6 Resultados e performance

Primeiro que tudo refere-se que se testou colocar parâmetros errados, assim como não colocar certo parâmetro obrigatório e colocar outro não especificado, e confirma-se que o programa lança uma excepção. Fizeram-se também ficheiros de teste em que um nó só tem uma adjacência e em que o identificador do nó é superior ao número de nós, e o programa saí logo de início, como previsto. Realizaram-se alguns ficheiros de teste sem um ciclo de *hamilton* e como esperado não encontra nenhum resultado.

6.1 data1.xml, data2.xml, data3.xml e data5.xml

O primeiro ficheiro foi o fornecido pela docente. O data3 apresenta um caminho com um menor custo que todos os outros por alguma margem e o nosso programa alcança-o quase sempre. O data5 só contém dois caminhos (um inverso do outro).

6.2 data4.xml

O ficheiro de teste *data4.xml* apresenta um grafo de 15 nós e só tem dois ciclos de *hamilton* possíveis (em que um é o inverso do outro). Devido ao número de ligações, este grafo leva a que ocorram muitos ciclos que não contenham todos os nós do grafo. Assim, correndo este ficheiro para um *finalinst* inferior a 300.0 e *antcolsize* inferior a 4, e todos os outros parâmetros *default*, a maioria das vezes não é encontrado sequer um ciclo de *hamilton*. Aumentando o tempo final ou o número de formigas da colónia, já consegue encontrar um ciclo de *hamilton* que é o que permanece em todas as outras observações até ao final, visto que este grafo só tem dois caminhos e são inversos.

6.3 denso.xml

Realizou-se um grafo denso com 10 nós todos ligados uns aos outros com peso 10, à excepção de uma ligação de cada nó que tem peso 9 (peso não pouco afastado do valor de todos os outros para ser mais complicada a escolha do próximo nó). Com este ficheiro fez-se vários testes. A primeira observação a fazer é que nem sempre encontra o melhor caminho (só existem dois melhores caminhos, sendo estes os dois caminhos iguais, mas inversos). Para encontrar o melhor caminho tem de se aumentar o *finalinst* consideravelmente e/ou o *antcolsize*.

Foi para este grafo que se analisou a variação dos parâmetros de entrada. Fixando *antcolsize* a 1 e os restantes parâmetros a *default*, obtiveram-se os diferentes custos (valor médio de várias vezes a simulação corrida) para os diferentes valores do tempo de simulação:

finalinst	cycleCost
150	99
300	97
500	97
1000	97
3000	95
5000	95
10000	94/95
100000	93/94
1000000	93
5000000	92
10000000	91/92
100000000	91

Tabela 1: Performance do algoritmo em função de $finalinst$.

Realça-se que o melhor custo é 91, pelo que só a partir de um *finalinst* consideravelmente grande é que consegue obter o melhor custo em quase todos os casos e que quando o custo que começa a obter é cada vez menor, é necessário um instante de tempo muito maior para conseguir reduzir mais.

Fixando agora o *finalinst* a 10000, a variação do custo com o *antcolsize* é:

antcolsize	cycleCost	mevents	eevents
1	94	5100	900
10	93/94	51000	9000
50	92/93	255000	38500
100	92/93	512000	65000
1000	92/93	5100000	80000
5000	91	25800000	223000
10000	91	51500000	225000

Tabela 2: Performance do algoritmo em função de $antcolsize$.

É importante també frizar, além do óbvio aumento do número de eventos de evaporação e de movimento das formigas como aumento do *finalinst*, que estes valores aumentam também bastante com o aumento das formigas, como se observa na tabela. O *mevents* aumenta proporcionalmente como esperado

De acordo com a formula das probabilidades de movimento da próxima aresta, quanto mais pequeno for o valor de α, maior é a sensibilidade das probabilidades aos níveis de feromonas! Assim, estudou-se também a variação do número de eventos de evaporação de feromonas bem como o valor do custo do caminho obtido para diferentes valores de *plevel*, fixando-se um α desprezável ($\alpha = 0.00001$). Fixou-se também *finalinst*=100000 *antcolsize* = 100.

Uma vez que o valor de *plevel* influencia o valor de feromonas que se coloca nas arestas sempre que se faz um ciclo de *hamilton* nesse caminho, um aumento do valor de *plevel*, leva a que o valor de feromonas colocadas seja maior, pelo que as respetivas ligações tenham um aumento de "peso"(probabilidade de o próximo ciclo passe por aí). Assim, pode não conseguir encontrar outro caminho além do primeiro a encontrar ou simplesmente atrasa o processo de encontrar o melhor caminho, como se conseguiu observar.

δ	**cycleCost**	**mevents**	**eevents**
0.2	91.0	5146243	2249202
2	92.0	514003	2248365
20	93.0	51935	1612978
200	94.0	5148	359830
2000	-	596	0

Tabela 3: Performance do algoritmo em função de δ.

Como se pode observar na tabela 3, o aumento do valor de δ diminui a probabilidade de encontrar o melhor ciclo de Hamilton no grafo. Este é um resultado bastante intuitivo, visto que δ é diretamente proporcional ao tempo médio que uma formiga demora a percorrer uma determinada aresta do grafo. Com um valor mais elevado de δ, as formigas

são mais lentas e, por consequência, existe uma menor "exploração"do grafo, o que explica também o menor número de *mevents*.

Os parâmetros η e ρ não têm um impacto tão linear na descoberta de caminhos com menor custo como os parâmetros anteriores. O nível de feromonas numa aresta é reduzido por ρ unidades, com uma distribuição exponencial de tempo médio η entre evaporações.

Um aumento de ρ leva a que o nível de feromonas tenda para zero mais rapidamente, havendo por isso menos eventos de evaporação. Tem de se ter em atenção também que o aumento de ρ pode levar a que não encontre o caminho mais curto, uma vez que há uma evaporação de feromonas mais acentuada, acabando assim por reduzir a importância destas no cálculo da probabilidade de se mover para uma certa aresta. Uma diminuição acentuada de ρ pode também ser prejudicial para a descoberta do melhor ciclo de hamilton, pois as feromonas permanecem mais tempo nos ciclos encontrados e isto será uma desvantagem quando um ciclo sub-ótimo é encontrado.

Um aumento do valor de η aumenta o tempo médio entre evaporações, o que resulta em menos evaporações na simulação (caso semelhante à diminuição do valor de ρ). Uma diminuição do valor de η aumenta o número de evaporações na simulação (caso semelhante ao aumento do valor de ρ).

O β quanto maior for, normaliza mais as probabilidades, não tendo o peso das ligações tanto efeito. Para um valor muito grande de beta, os pesos das ligações deixam de ter efeito para o valor da probabilidade.

Analisou-se também o tempo de execução do programa, que aumenta com o aumento de *finalinstant* e *antcolsize*, uma vez que quanto maior o número de formigas ou maior o instante final maior é o número de eventos simulados.

6.4 big.xml

Este ficheiro contém um grafo com 1000 nós e muito denso. O nosso programa aproxima-se do caminho mais curto várias vezes.

7 Conclusão

Desenvolveu-se um simulador para resolver o problema de encontrar um ciclo de *hamilton*, através de uma sequência de eventos simulados, com o tempo entre os eventos a seguirem leis aleatórias. Todo o projeto foi projetado de modo a ter a maior extensibilidade e abstração de dados possível, e de modo a que a que houvesse facilidade em adicionar novas funcionalidades sem a necessidade de se trocar as outras.

Toda a comunicação entre diferentes identidades (formiga, simulação, grafo, evento) foi realizada através de interfaces bem definidas, pelo que uma camada não sabe a implementação de outra, e uma alteração numa camada não é reconhecida por outra.

Por vezes, de modo a conseguir-se fornecer a maior extensibilidade e abstração possível, houve decisões menos diretas para resolver certo problemas.

Em todo o projeto teve-se também em atenção as estruturas de dados utilizadas de modo a se ter um balanço positivo entre memória utilizada e tempo de execução do programa, tentando ter sempre a menor complexidade possível.