Instituto Superior Técnico

Engenharia Eletrotécnica e de Computadores

Programação Orientada por Objectos

SIMULAÇÃO DE EVENTOS

Autores:
Mariana Galrinho 81669
Paulo Eusébio 81607
Renato Henriques 81588
Grupo 6

Corpo Docente:
Prof. Alexandra Carvalho

11 de Maio de 2018

1 Introdução:

O objetivo deste projeto é desenvolver um programa em Java tendo em conta as caraterísticas de uma linguagem orientada a objetos e o *open-closed principle*, que dita que as entidades (classes, métodos, etc) devem ser abertos para extensão mas fechados para alterações.

O programa desenvolvido consiste numa simulação do ciclo de vida de indivíduos que se encontram num mapa retangular com obstáculos. Estes indivíduos podem realizar vários tipos de eventos, como moverem-se pelo mapa, reproduzirem-se ou morrerem. Os eventos são guardados numa PEC, *pending event container*.

2 Estruturas de dados escolhidas

2.1 Representação do Mapa:

O mapa retangular pode ser interpretado como um grafo, por ter vértices (os pontos) e arestas (onde se representa as adjacências e o custo entre pontos). Para além disso, este é um grafo esparso, ou seja, o número de arestas não é muito superior ao número de vértices, já que para n e m maiores que 2 o número máximo de arestas que um ponto pode ter é 4 e o mínimo é 2. Isto significa que para ver o número de conexões de um ponto, o número máximo de operações que se tem que fazer é 4. Por este motivo, a vantagem que se teria em usar uma matriz de adjacências (O(1) para descobrir arestas) é pouco significativa e não compensa o maior uso de memória ($O(n^2)$, em que n é o número de pontos) face ao da lista de adjacências. Por estas razões decidiu-se utilizar uma lista de adjacências para representar o mapa.

A lista de pontos é representada com um *ArrayList* porque garante uma boa eficiência no acesso aos pontos e porque não existe o problema da capacidade desta estrutura de dados ser excedida já que o número de pontos do mapa é fixo. Assim sendo, basta inicializar o *ArrayList* com a área (altura×largura) do mapa (que corresponde ao número de pontos) para não haver mais necessidade de alterações na capacidade do array.

Pelo mesmo motivo, a lista de adjacências (ou conexões) de cada ponto da lista de pontos foi representada com um *ArrayList*. Dado que no máximo se pode ter 4 conexões, inicializando a capacidade da lista com este valor não haverá necessidade em ter outras alocações de memória que consomem tempo desnecessário no decorrer do programa.

2.2 Representação da PEC, Pending Event Container:

A PEC é representada utilizando a estrutura de dados *PriorityQueue* que existe na *java.util*. A escolha desta estrutura de dados é óbvia porque existe a necessidade de obter a cada ciclo da simulação o evento seguinte, ou seja, o evento com o menor tempo. Utilizando uma *priority queue*, sempre que um novo evento é adicionado à fila, este será inserido no seu devido lugar tendo em conta uma ordenação crescente dos tempos (eventos com menor tempo no início da fila e maior tempo no fim).

A ordenação é realizada a partir do *TimeEventComparator* fornecido na construção da *priority queue*. Escolheu-se ordenar a PEC com uma classe externa que implementasse um *Comparator* ao invés de implementar a interface *Comparable* na classe *Event*. Considerou-se que o tempo dos eventos não definia a sua ordem natural já que dois eventos diferentes podem ter o mesmo tempo, ainda que isto seja muito pouco provável no caso de uma simulação estocástica.

Neste caso ordenar a lista de eventos a cada ciclo do simulação é uma solução menos eficiente que a inserção ordenada da *priority queue*, porque seria necessário fazer O(NlogN) operações (caso geral do *mergesort*), sendo N o nº de eventos na PEC, todos os ciclos da simulação, enquanto que a inserção na *PriorityQueue* precisa apenas, no pior caso, de O(N) operações.

A inicialização da PriorityQueue é feita com uma capacidade igual a 3 vezes o número máximo de indivíduos estabelecido no ficheiro XML, porque se considerou que é um valor suficientemente grande para não haver alocação de memória da Queue com frequência mas que ao mesmo tempo faz sentido para os dados do problema.

2.3 Representação da Lista de Individuals:

A estrutura de dados utilizada para a lista de Individuals é uma LinkedList. Considerou-se este tipo de lista mais adequado do que uma ArrayList porque não é possível prever o número máximo de indivíduos que pode estar na lista, o que significa que existe a desvantagem de poder ser necessário alocar mais memória para aumentar a capacidade da ArrayList. Para além disso, a maior vantagem em usar uma ArrayList é o acesso O(1) aos elementos da lista. Contudo, neste programa nunca é necessário fazer acessos a elementos específicos, mas sim apenas fazer o varrimento completo da lista, cuja complexidade é igual em ambos os tipos de lista. Por fim, a nível de gestão da memória do programa a LinkedList é mais eficiente que utilizar uma ArrayList.

Para conseguir determinar os 5 melhores indivíduos durante a epidemia decidiu-se ordenar a lista de indivíduos utilizando o método sort da classe List do java.util que, segundo a documentação, tem aproximadamente complexidade O(N) para listas praticamente ordenadas e O(N*Log(N)) para casos gerais. O comparador utilizado para efetuar a ordenação pode ser encontrado na classe IndividualComfortComparator e segue uma ordem decrescente do conforto de cada Individual.

Optou-se por ordenar esta lista apenas quando há epidemias do que utilizar uma estrutura de dados ordenada como uma PriorityQueue porque seria necessário fazer uma inserção ordenada de cada indivíduo sempre que este altera-se o seu conforto. Como se espera que na grande maioria dos simulações existam muitas mais vezes alterações dos parâmetros dos indivíduos do que ocorrências de epidemias concluiu-se que esta seria uma maneira preferível de encontrar os 5 melhores indivíduos.

2.4 Representação do Path do Individual:

O caminho do Individual é representado utilizando uma LinkedList da java.util cujos nós são objetos do tipo Point. A escolha desta estrutura de dados deve-se ao facto de não haver necessidade de aceder a pontos específicos da lista, mas sim varrê-la até encontrar um certo ponto. Para além disso, quando ocorrem ciclos é necessário remover pontos da lista desde o ponto que criou ao ciclo até ao fim da lista. Por estes motivos considerou-se que uma LinkedList seria ideal e mais adequada do que outras estruturas de dados, como por exemplo uma ArrayList.

2.5 Escolha do XML Parser para analisar o ficheiro XML:

Optou-se por utilizar o SAX como XML Parser porque consome pouca memória e também porque para este projeto não há necessidade de caraterísticas como acesso a elementos já processados e alteração do ficheiro XML que apenas o DOM possui.

Para validar a estrutura do XML utilizou-se um DTD cujo nome é simulation.dtd e que impossibilita, entre outras condições mais óbvias, que haja mais do que um ponto inicial, um ponto final

e mais do que um elemento do tipo *events*. Para além disso, o DTD permite que haja zero ou um elemento do tipo *specialcostzones* e do tipo *obstacles*.

O parser está implementado na classe MyHandler onde também é feito algum controlo dos valores recebidos do XML. O programa termina, por exemplo, quando:

- Os valores recebidos do XML são de qualquer tipo diferente de um inteiro;
- As coordenadas dos pontos recebidas são negativas, zero ou excedem a largura ou comprimento do mapa;
- Os parâmetros dos eventos são negativos;
- O *finalinst*, a *initpop* ou a *maxpop* é negativa;
- O *comfortsens* é negativo ou zero;
- O nº de colunas ou nº de colunas são negativos ou zero.

3 Abordagem orientada a objetos

3.1 Eventos

3.1.1 Interface EventI

Optou-se por fornecer uma interface para eventos de qualquer tipo, isto é, que não necessitam de ser obrigatoriamente associados a um tempo, podendo ser movidos por outro tipo de atributos. Assim, o método *simulateEvent* não precisa de ser limitado a simulações baseadas em tempo, sendo aplicável a uma maior gama de casos a partir da extensão desta interface, bem como das restantes interfaces oferecidas no package *general*.

O método *simulateEvent* retorna uma lista de eventos. Posto isto, caso não se queira retornar nenhum evento pode-se retornar uma lista vazia ou uma referência *null*. No entanto, caso se queira adicionar novos eventos ao *Pending Event Container* é possível retorná-los e adicioná-los posteriormente na simulação, o que torna mais fácil este processo já que a adição de novos eventos é tratada exclusivamente na classe do simulador.

3.1.2 Classe Event - polimorfismo

Criou-se a classe *Event* como uma implementação da interface anterior, estando já associada a um tempo de simulação. Esta classe deixa, deste modo, em aberto a implementação do método *simulateEvent*, podendo ser extensível por qualquer evento temporal. A utilização de uma PEC com eventos do tipo *Event* e a declaração na simulação do *current event* como *Event*, permite fazer uso de polimorfismo. O polimorfismo encontra-se presente quando o método *simulateEvent* é chamado na simulação já que o método invocado é determinado apenas em tempo de execução dependendo do tipo dinâmico do objeto pelo qual foi invocado, isto é, do tipo de evento (subclasse de *Event*) retirado da PEC.

3.1.3 Classes específicas da simulação

Forneceu-se a classe abstracta *IndividualEvent* que estende a classe *Event*, como uma classe que irá ser especificada pelos eventos relacionados com a evolução dos indíviduos na simulação (*Death*, *Move* and *Reproduction*). Esta providencia já alguns atributos comuns como o indivíduo do evento e um método estático que verifica se o tempo de um evento é superior ao tempo da morte de um dado indivíduo. Assim, existe já uma classe com aspetos comuns a diferentes eventos, passível de ser extensível numa outra simulação semelhante a esta mas em que sejam utilizados eventos diferentes.

Optou-se por implementar as observações com informação do estado da simulação como eventos uma vez que estas possuem um tempo definido no qual vão ocorrer. As observações são inseridas na PEC uma de cada vez, isto é, sempre que uma observação é simulada, adiciona a seguinte. Tendo em conta que a simulação pode terminar antes do instante final estabelecido, poderão não ser apresentadas 20 observações pelo que se preferiu não adicionar eventos à PEC desnecessariamente.

3.2 Interface para PEC

O programa possui uma interface (PECI) para o *Pending Event Container*. Nesta, cedem-se todos os métodos necessários para realizar operações a uma qualquer *collection* que contenha eventos que implementem a interface *EventI*, sendo por isso uma interface genérica com um tipo: *E extends EventI*. Esta inteface torna o programa mais extensível na medida em que oferece a possibilidade de criar novas classes com *event containers* distintas da pretendida para a simulação específica, mas cujas operações se encontram fixas. Assim torna possível a criação de containers com diferentes estruturas de dados (caso se pretenda, por exemplo, comparar as suas eficiências), ou de *containers* para outros eventos que implementem a interface *EventI* mas que não descendam classe de evento utilizada no nosso caso específico, oferecendo a oportunidade do *event container* ser ordenado por outro atributo.

3.3 Simulação

3.3.1 Interface e classe Abstracta

A existência de uma interface para a simulação, *SimulationI*, permite atingir uma maior abstracção e promove a extensibilidade da aplicação desenvolvida na medida em que fixa obrigatoriamente a existência dos métodos essenciais à simulação (*simulate* e *init*) nas classes que o implementam (ou obrigando as classes a serem abstractas se não os implementarem), permitindo também a definição posterior de métodos adicionais.

Fornece-se adicionalmente uma classe abstracta, *SimulationA*, já com uma PEC de eventos associados a um tempo e com alguns métodos auxiliares, de forma a facilitar a implementação da interface anterior para diferentes simulações que sejam conduzidas por eventos temporais, como o caso da simulação da grelha ou da simulação da autoestrada.

3.3.2 Polimorfismo

A existência da interface e da classe abstracta apresentadas permitem a utilização de polimorfismo em diversas aplicações. Por exemplo, na definição de um programa que corra várias simulações, não é necessário alterá-lo posteriormente caso se pretenda alterar a simulação a efetuar. Assim, é possível

ignorar os aspetos específicos das simulações, deixando estes para a parte de desenvolvimento da própria simulação, e restringindo a interacção aos tipos (interfaces ou classes) superiores.

3.3.3 Simulação específica

Adicionalmente, desenvolveu-se a classe do simulador específico ao projeto de forma a ser possível instanciar um objeto de simulação a partir de um ficheiro e a correr a simulação diversas vezes (quer sequencialmente, quer paralelamente). Dado não existirem atributos estáticos e tendo em conta que todas as variáveis dinâmicas são limpas no início da simulação.

3.4 Gerador de números

3.4.1 No package *general*

Fornece-se, no package *general*, a interface *INumberGenerator* com um método que retorna um número em formato double, de forma a ser possível aceder de igual a forma a diferentes tipos de geradores de números que se pretenda utilizar na simulação.

Uma vez que a evolução de uma simulação possui, à partida, uma dependência em mecanismos de gerações de números (de forma estocástica ou determinística) para determinar o tempo de um novo evento a simular ou a forma como a simulação dos eventos se desenrola, desenvolveu-se a classe *SimulationCommands* que é associada às simulações (associação com a classe *SimulationA*). Esta classe encapsula os diferentes geradores de números necessários a uma simulação, guardado-os num *array*, fazendo assim corresponder, por exemplo, um tipo de evento ao índice do gerador de números que o rege. O acesso a estes é feito através do método *getCommand*. Uma vez que cada comando da simulação (gerador de números) é declarado como o tipo da interface (*INumberGenerator*), neste método recorre-se a polimorfismo para aceder de igual forma a qualquer um dos comandos, recebendo apenas a posição do gerador no array e chamando o método da interface determinado em tempo de execução pela posição fornecida.

3.4.2 No package *specific*

Para o caso da simulação da grid, considerou-se que eram necessários 3 geradores que calculassem o tempo entre cada um dos diferentes tipos de eventos e um gerador que retornasse um número entre 0 e 1. Apesar da simulação em questão ser conduzida por mecanismos estocásticos, sendo apenas necessária a criação de geradores de números aleatórios segundo uma distribuição exponencial e segundo uma distribuição uniforme, pretendia-se o desenvolvimento de uma aplicação de tal forma a que fosse possível a reutilização de todo o código desenvolvido para o caso determinístico, sendo apenas necessária a alteração dos geradores pretendidos nessa situação.

No entanto, foi necessário ter em conta que no caso estocástico a geração dos tempos entre eventos é dependente do indivíduo aos quais se encontram associados, sendo a expressão e os parâmetros das distribuições variáveis consoante o tipo de evento. Para solucionar este problema, criou-se a classe abstracta *IndividualTimeGenerator*, que não implementa ainda o método getNumber mas que associa o gerador a um determinado indivíduo. No caso da simulação específica esta classe foi estendida por outras 3, criando a geração dos tempos entre cada evento a partir do conforto do respetivo indivíduo.

De forma a possibilitar o uso destes geradores pela classe do simulador (*GridSimulation*), estendeu-se a classe *SimulationCommands* com a classe *GridCommands* que fornece um método para retornar o número gerado, alterando contudo o indivíduo antes disso para o indivíduo com o qual se vai

determinar o valor do tempo. Na classe de simulação específica optou-se por receber cada gerador como argumento do construtor, sendo a criação do objeto *GridCommands* apenas realizada e acedida dentro da própria classe.

Assim, é possível utilizar polimorfismo para tratar de igual forma diferentes tipo de geradores sem ser necessário modificar a simulação e reutilizar a simulação estocástica desenvolvida para o caso determinístico. Caso se pretendesse implementar uma simulação regida por mecanismos determinísticos, em que os tempos dos eventos se encontram à partida fixos, seria apenas necessário estender a classe IndividualTimeGenerator ou INumberGenerator para o que se pretendesse e fornecê-los no construtor da simulação ao invés dos fornecidos no caso específico.

4 Outras escolhas feitas

- Cada Individual possui uma a referência para cada evento (a sua morte, a próxima reprodução e move), de forma a tornar a epidemia mais eficiente. Uma alternativa seria, caso o individuo morre-se percorrer o pec de forma a encontrar cada evento pertencente a esse indivíduo e retirá-lo, o que seria menos eficiente;
- Só se adicionam eventos à PEC se forem anteriores ao tempo final da simulação e ao tempo de morte do respetivo indíviduo, de forma a não a sobrecarregar;
- O Individual possui uma referência para a População da simulação, que possui parâmetros informativos da simulação necessários aos indivíduos, de forma a evitar a passagem de muitos parâmetros entre eventos e indivíduos, guardando-se uma única referência.

5 Visibilidades

Quanto às visibilidades utilizadas na aplicação, teve-se em consideração a possibilidade de extensão das classes já desenvolvidas, por exemplo para uma simulação num mapa 3D ou para a inclusão de um atributo idade no indivíduo, estando o seu evento morte relacionado com esse parâmetro.

Posto isto, definiram-se todas as classes como públicas e os respetivos atributos com visibilidade *protected* na maior parte dos casos, de forma a facilitar a extensão por subclasses fora do *package*. Para os métodos, tomou-se a visibilidade pública se fazia sentido a utilização destes fora do *package* em questão, *private* se eram métodos apenas utilizados apenas pela própria classe e que se considerou que não seriam necessários numa outra simulação, ou *protected* se apesar de no caso em questão serem utilizados exclusivamente pela classe se considerou que podiam ser necessários para a extensão por subclasses.

6 Avaliação da performance do programa

Começou-se por testar a aplicação desenvolvida com o ficheiro XML fornecido, tendo-se verificado que o melhor caminho (com custo 12) era descoberto na maior parte das vezes, cerca de 80%, e que se chegava apenas ao caminho com custo 13 em cerca de 20% das vezes. Verificou-se que a grande maioria dos casos em que não se obtinha o melhor caminho, eram casos em que menos eventos eram simulados, com um número inferior a cerca de 15 mil eventos.

Aplicou-se o programa desenvolvido a diferentes tipos de mapas e populações, tendo-se verificado que para mapas pequenos o ponto final era quase sempre atingido, enquanto que para mapas maiores, era necessário estender mais o tempo de simulação bem como a população para que se chegasse ao ponto final, como seria de esperar já que o conforto diminui com a distância ao ponto final e passa a existir um número muito mais elevado de possíveis caminhos. Observou-se que o escalamento do tempo de execução do programa com o tamanho do mapa, mudando os parâmetros deste de tal forma a que o ponto final fosse atingido (ficheiro de teste test_1.xml), segue uma curva semelhante à apresentada na figura 2 (esquerda) para o aumento da população.

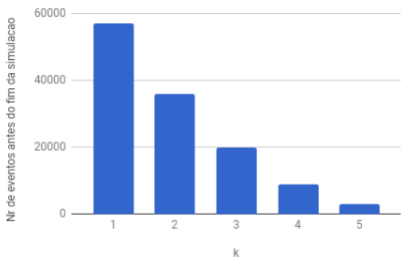

Figura 1: Número de eventos obtidos com a variação da sensibilidade do conforto a pequenas variações, utilizando os restantes parâmetros fornecidos no projectexample.xml

Efetuaram-se também testes variando os outros parâmetros da população. Observou-se, por exemplo, que com o mapa fornecido um aumento no parâmetro de sensibilidade do conforto, resultava num menor conforto e consequentemente num maior tempo entre eventos, pelo que o número de eventos obtidos no final da simulação era menor (figura 1). Para além disso, nas epidemias, e caso a população máxima fosse reduzida, havia um maior número de indivíduos a morrer. A fixação de um instante final grande, bem como um parâmetro *death* elevado e uma população final pequena conduzia à existência de um elevado número de epidemias. Verificou-se também que uma simulação com *maxpop* e *death* elevados e com o parâmetro *reproduction* pequeno conduzia a uma população muito elevada, e consequentemente, à simulação de um número mais elevado de eventos, o que resulta um maior tempo de execução da simulação.

Apresentam-se também alguns testes efetuados à performance do programa nas figuras 2, sendo cada resultado apresentado aproximadamente a média resultante de correr o programa 10 vezes.

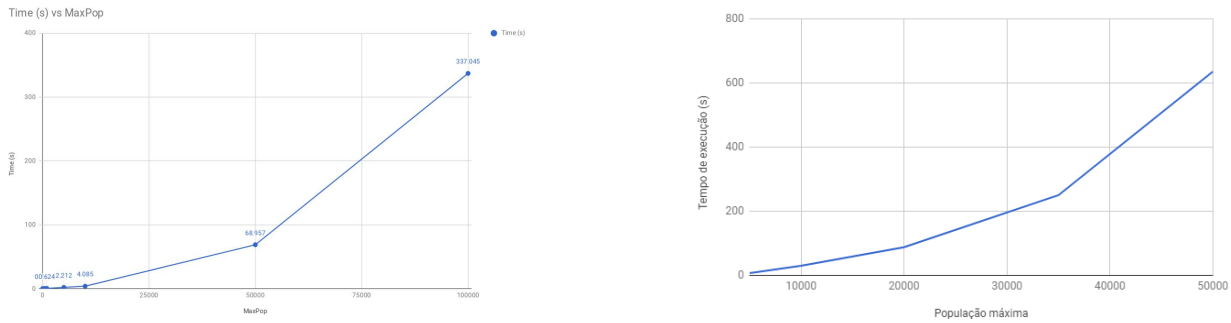

Figura 2: Representação do escalamento do tempo de execução do programa com o aumento do número máximo de indivíduos (esquerda) e aumento do número inicial provocando epidemias (direita) de acordo com os parâmetros do test_5.xml .

Conforme se pode observar pela Figura 2 (esquerda), foi testada a execução do programa com o ficheiro XML do enunciado com valores de *maxpop* cada vez superiores. Verificou-se que o tempo que

demorava a executar o programa escala como esperado já que quanto maior o número de indivíduos, maior o número de eventos que são simulados até ao fim da simulação.

Na figura da direita fez-se variar o número máximo e o número inicial da população com *maxpop=initPop*. Utilizaram-se os parâmetros fornecidos no *test_5.xml*, de forma a que ocorresse um elevado número de epidemias. Em comparação com a figura da esquerda é possível verificar que o tempo de execução se torna mais elevado com uma menor população devido à perda de eficiência causada pelas epidemias.

Por fim, as análises que foram feitas levam a acreditar que o bottleneck do programa a nível do desempenho é a ordenação da lista de indivíduos, fator que é facilmente visível em testes onde ocorram epidemias com frequência.

7 Outras funcionalidades implementadas

Com o objetivo de correr várias simulações em simultâneo decidiu-se tornar o projeto compatível com *multithreading*. Assim sendo, criou-se uma classe chamada *SimulationThread* que herda a classe *Thread*, que tem uma referência para uma simulação e dá override do método *run*. Este método apenas chama o método *simulate* da simulação recebida no constructor. O multithreading foi testado criando várias simulações e várias *threads* para as correr. Verificou-se que funcionam da maneira esperada em termos de resultados apesar das impressões para a consola das várias simulações serem feitas de forma não-determinística.

8 Conclusões

O simulador foi desenvolvido com o objetivo de resolver o problema proposto, providenciando ainda liberdade para a extensão da solução e facilitar a adição de novas funcionalidades ao programa, cumprindo assim com os requisitos de uma solução numa linguagem orientada a objetos. Todos os requisitos são resolvidos de forma eficaz, mantendo um bom equilíbrio entre memória consumida e tempo de execução.

Contudo, sempre com o objetivo de facilitar a extensibilidade da aplicação, algumas soluções simples sofreram um aumento de complexidade que pode dificultar a compreensão do programa, daí a necessidade de algumas descrições extensas neste relatório.

Referências

[1] Alexandra Carvalho, *Object Oriented Programming Project 2017/18* , em `https://fenix.tecnico.ulisboa.pt/downloadFile/1970943312318729/projecto-POO1718.pdf`