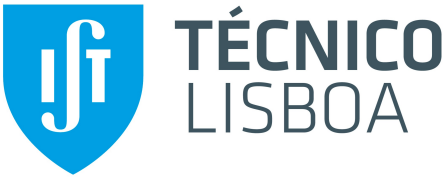

Instituto Superior Técnico

Mestrado Integrado em Engenharia Eletrotécnica e de Computadores

Programação Orientada a Objetos

Projeto Final

Travelling salesmen problem by ant colony optimization

Grupo 11:
Carolina Cunha, nº 79656
João Pedro Cardoso, nº 84096
Matilde Pereira Moreira, nº 84137

Docente:
Alexandra Sofia Martins de Carvalho

2º Semestre
2018-2019

Índice

1 O Problema **2**

2 Arquitetura da Solução e Implementação **3**
2.1 UML . . . 4
2.2 Package *main* . . . 4
2.3 Package *graph* . . . 4
2.4 Package *discreteStochaticSim* . . . 4
2.4.1 *InterfaceEvent* . . . 5
2.4.2 *Event* . . . 5
2.4.3 *EventComparator* . . . 5
2.4.4 *EventControlPrints* . . . 5
2.4.5 *Evaporation* . . . 5
2.4.6 *Move* . . . 5
2.4.7 *InterfacePec* . . . 6
2.4.8 *PriorityQueuePec* . . . 6
2.5 Package *antColony* . . . 6
2.5.1 *Parameters* . . . 6
2.5.2 *OptProblem* . . . 6
2.5.3 *StochasticOptimProb* . . . 7
2.5.4 *Ant* . . . 7
2.5.5 *HamiltonianCycle* . . . 7
2.5.6 *HCResults* . . . 7
2.5.7 *pathw* . . . 7
2.6 Algoritmo de Otimização . . . 7
2.6.1 Eventos na Lista de Prioridade . . . 7
2.6.2 Encontrando um caminho . . . 8

3 Resultados e Simulação **8**

4 Conclusão e Análise Crítica **9**

Anexo A **10**

1 O Problema

Neste projeto é abordado um *Travelling salesman problem* (TSP): um problema de optimização cujo objetivo é encontrar num grafo ponderado o caminho mais curto possível passando por todos os nós do grafo uma única vez. Um grafo ponderado, representado por $G = (N, E, \mu)$, é composto por um conjunto finito de nós N, um conjunto de arestas E (sendo que cada aresta representa a ligação entre dois nós adjacentes) e uma função $\mu : E \to \mathbb{R}^+$ que atribui um peso a cada aresta. Na figura 1 encontra-se um exemplo simples de um grafo ponderado.

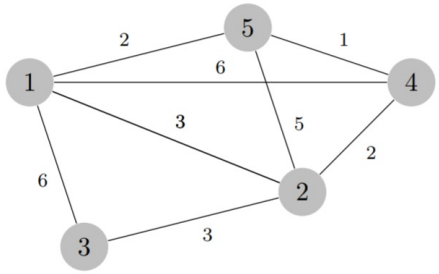

Figura 1: Exemplo de um grafo ponderado

O caminho que se pretende descobrir, dado pela sequência de nós $n_1, n_2, ..., n_k$, deverá ser um *Ciclo Hamiltoniano*: Ciclo que contém todos os nós pertencentes ao grafo e onde cada nó é visitado apenas uma vez à exceção do primeiro, uma vez que se trata de um ciclo e portanto deve ter-se $n_1 = n_k$.

Uma possível abordagem para a optimização deste problema é usar um *Ant Colony Optimization Algorithm*. Este algoritmo simula uma colónia de formigas que, uma a uma, atravessa de forma aleatória o grafo percorrendo todos os seus nós (sem visitar mais do que uma vez cada nó), e após encontrar um *Ciclo Hamiltoniano*, deixa um rasto de feromonas por todas as arestas do grafo que constituem o ciclo determinado.

Dado um nó para a posição atual da formiga e um conjunto de nós adjacentes a esse nó que ainda não tenham sido visitados, o percurso feito pela formiga está obviamente dependente do nó adjacente que é selecionado para ser visitado em seguida. Essa seleção é aleatória com uma probabilidade proporcional ao nível (quantidade) de feromonas na aresta que liga os dois nós em questão, e inversamente proporcional ao peso dessa mesma aresta.

Por outro lado, o tempo que a formiga demora a atravessar uma dada aresta é também aleatório e segue uma distribuição exponencial cuja média m é proporcional ao peso da aresta a atravessar, e tem uma função de distribuição acumulada dada por $P(T \leq t) = 1 - \mathrm{e}^{-t/m}$. Na figura 2 e 3 tem-se, a titulo exemplificativo e para melhor compreensão, o andamento de uma função de distribuição acumulada para três valores de média diferentes ($m = 1/\lambda$). Assim, no contexto do nosso problema, podemos facilmente concluir que há maior probabilidade de uma aresta ser atravessada em menos tempo, numa aresta com um menor valor para a média da distribuição do tempo relativo a essa aresta, e portanto, numa aresta com menor peso.

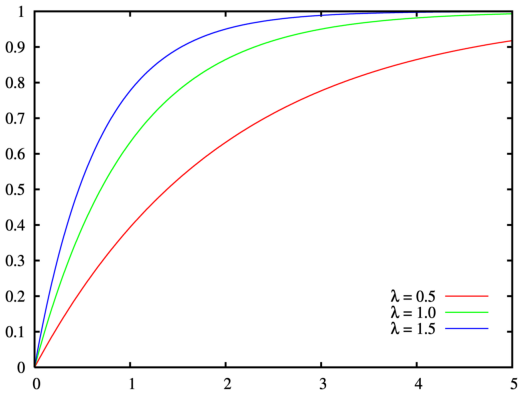

Figura 2: Função de distribuição acumulada

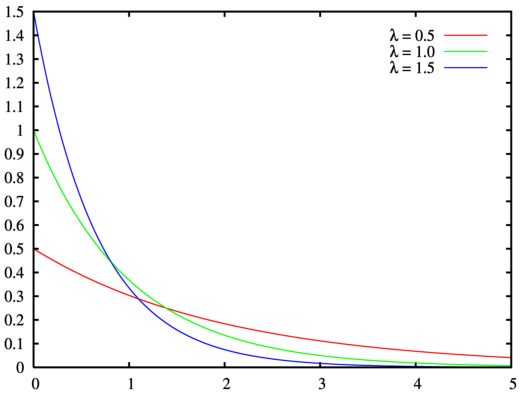

Figura 3: Função de densidade de probabilidade

Tendo em conta que o objetivo do problema é encontrar o ciclo de *Hamiltonian* mais curto, em termos temporais o ideal será que a formiga opte por traçar o seu caminho por arestas cuja probabilidade de serem atravessadas em menor tempo seja maior. Ou seja, idealmente deveriam ser escolhidas para este efeito as arestas de menor peso (como vimos anteriormente). De facto, a probabilidade de seguir determinada aresta é inversamente proporcional ao peso da aresta, mas também proporcional à quantidade de feromonas presente nessa mesma aresta. Portanto há maior probabilidade de escolher as arestas com pesos pequenos —Exatamente o desejado para minimizar o tempo! Por outro lado, também se quer que essas arestas de menor peso (como maior probabilidade de serem selecionadas) tenham elevados níveis de feromonas, que vai contribuir para uma ainda maior probabilidade de serem selecionadas, contribuindo para a minimização do tempo perdido a completar um ciclo. Ora visto que as feromonas depositadas numa aresta tendem a evaporar-se ao longo do tempo, a única forma de se ter um acumular consecutivo de feromonas nas arestas é ter mais do que uma formiga a passar por cada aresta. Ou seja, colónias maiores, aumentam a probabilidade de se determinar o menor ciclo Hamiltoniano no que toca ao tempo perdido.

2 Arquitetura da Solução e Implementação

Neste projeto foi proposta a implementação de uma aplicação em Java que permitisse encontrar o caminho de menor custo que passasse por todos os nós do grafo descrito no ficheiro XML, utilizando Programação Orientada por Objetos. Num estado inicial, um conjunto de formigas (*antColony*) é colocado no *nestnode* (ponto inicial do grafo) e durante a simulação são realizados dois tipos de eventos distintos:

- Movimento da formiga;
- Evaporação das feromonas depositadas nas arestas do grafo;

A simulação da aplicação consiste na realização dos eventos que se encontram na *pending event container* (PEC) até que seja atingido o limite de tempo (*finalinst*) ou não existam mais eventos a realizar. O projeto desenvolvido está dividido em 4 packages: *antColony*, *discreteStochaticSim*, *graph* e *main*, cuja relação está representada na figura 4, existindo na totalidade 20

classes. Destas 20 classes, 4 delas correspondem a interfaces: classes implicitamente abstratas e onde não é feita a implementação de métodos do tipo `public`, mas apenas a sua declaração.

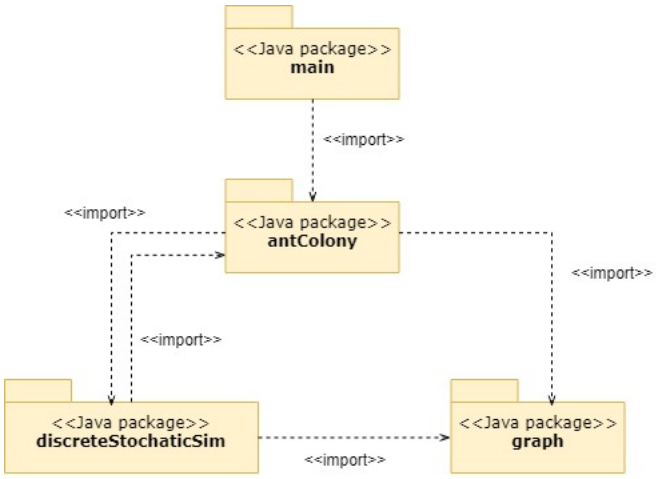

Figura 4: Organização do projeto em packages

2.1 UML

O UML com a especificação e estruturação da nossa implementação do projeto encontra-se na figura 5 no Anexo A. O esquema do UML foi obtido de forma automática através da *ObjectAid UML* que foi previamente adicionado ao *Eclipse*.

2.2 Package *main*

Nesta package existe apenas a classe *Main* com o método `public static void main` que recebe como parâmetro de entrada o ficheiro XML com todos os dados que definem o problema e cria um problema de otimização que posteriormente é executado.

2.3 Package *graph*

A package *graph* inclui a interface *GraphInterface*, as classes *Graph*, *Edge* e *Vertex*, relacionadas com a definição, construção e manipulação da estrutura do grafo.

2.4 Package *discreteStochaticSim*

Esta package está relacionada com a simulação de eventos discretos que modela a operação do sistema como uma sequência de eventos discretos no tempo (diminuição do nível de feromonas nas arestas e movimento da formiga). Cada evento ocorre num determinado instante de tempo e marca uma mudança de estado. As classes constituintes desta package serão detalhadas de seguida.

2.4.1 *InterfaceEvent*

A *InterfaceEvent* é uma classe abstrata que contém a declaração do método `public abstract void ExecutaEvent` que irá ser implementado pelas classes Evaporation e Move, sendo esta implementação dependente da classe que está a invocar a interface.

2.4.2 *Event*

Event é uma classe abstrata, definido por um campo para a formiga e outro para o tempo em que devemos executar o evento. Existem 2 campos estáticos, sendo o primeiro um objeto aleatório para gerar números aleatórios obtido por um método igualmente estático `(public static double expRandom)` e o outro, um comparador de eventos, que é usado quando queremos adicionar eventos ao PEC.

2.4.3 *EventComparator*

EventComparator é uma classe usada somente para comparar Eventos baseando-se nos seus tempos. Quando usado com o método list.sort (Comparador), ele classificará uma lista de eventos pelo menor tempo primeiro. Ele implementa a interface do Comparador substituindo o método de comparação.

2.4.4 *EventControlPrints*

Esta subclasse é uma extensão da classe *Event*, é abstrata e manipula os eventos de impressão dos resultados obtidos ao longo das várias observações no terminal . Não possui campos em si, mas herda o tempo e o individuo (formiga) da superclasse. A formiga é definida como `null` uma vez que não fazemos "target"a uma formiga em especifico, e o tempo é definido como um múltiplo do tempo total (*finalinst*) dividido por 20. Ele sobrescreve o método `public abstract void ExecutaEvent`, para imprimir os dados finais.

2.4.5 *Evaporation*

A subclasse *Evaporation* é uma extensão da classe *Event* e manipula os eventos de evaporação das feromonas. Da superclasse é herdado o tempo, no entanto, não é necessário saber o indivíduo (formiga) visto a evaporação não estar dependente do indivíduo, pelo que que a formiga é definida a null. O método `public void ExecutaEvent` nesta subclasse é reescrito de forma a atualizar a quantidade de feromonas nas arestas do grafo e adicionar à lista de eventos (PEC), o novo evento para a evaporação considerando a quantidade de feromona atualizada face ao tempo.

2.4.6 *Move*

Esta subclasse é uma extensão da classe Event abstrata e manipulará os eventos de movimento das formigas. Herda o tempo e também a formiga em específico que se irá mover. O método `public abstract void ExecutaEvent` é reescrito de forma a implementar o algoritmo de otimização (*Ciclo hamiltoniano*) para o movimento da formiga, sendo este algoritmo posteriormente abordado em maior detalhe.

2.4.7 *InterfacePec*

A *InterfacePec* é uma classe abstrata que contém a declaração de todos os métodos a serem implementados pela classe *PriorityQueuePec*.

2.4.8 *PriorityQueuePec*

Nesta classe implementa-se a PEC onde se armazena os eventos de todas as formigas que irão ser realizados. Para definir a PEC optou-se por uma *Priority Queue* (fila de prioridades) que ordena os eventos por ordem crescente de acordo com o instante de cada um. Para tal, implementou-se a classe *EventComparator* anteriormente abordada, que define o comparador entre os instantes de cada evento (método compare). Implementou-se também métodos para adicionar eventos à PEC, para obter o próximo evento da fila e inclusive para imprimir no terminal, caso seja preciso.

2.5 Package *antColony*

A package *antColony* engloba alguns dos seguintes aspetos:

- parâmetros armazenados após a leitura do ficheiro cuja extensão é ".xml (*Parameters.java*);
- criação da formiga e respetiva colónia (*Ant.java*);
- algoritmo de otimização para encontrar o caminho recorrendo ao ciclo hamiltoniano (*HamiltonianCycle.java*);
- armazenamento de todos os ciclos hamiltonianos únicos e respetivos custo totais (*HCResults*);
- node que a formiga atravessou e o seu custo associado (*pathw*);
- entre outros;

2.5.1 *Parameters*

Nesta classe é possível encontrar os parâmetros que são extraídos do ficheiro .xml para posterior resolução do problema de otimização. Os métodos presentes nesta classe são maioritariamente getters e setters dados os atributos serem privados, à exceção do grafo `Gr`.

2.5.2 *OptProblem*

OptProblem corresponde a uma interface e é usado no *main*, permitindo o fornecimento de implementações alternativas de problemas de otimização. Os métodos declarados na interface são `void simulacao()`, que é responsável por executar a simulação por si mesma e o método `public void runOptimizationProb(String filename)`, que deve abrir o arquivo especificado nos argumentos, analisa-lo adequadamente, inicializar o problema de otimização e, em seguida, chamar o método de simulação, anteriormente mencionado.

2.5.3 *StochasticOptimProb*

Esta subclasse é uma extensão da classe *OptProblem* e recebe os métodos referidos na classe *OptProblem.java*, ou seja,`public void simulacao()` e `public void runOptimizationProb` em que invocamos o método `private void readXML(String filename)` e criamos as novas variáveis para a execução da simulação do ficheiro de problema de otimização. Nesta classe para além dos dois métodos falados antes, existem outros métodos como getters e setters, devido aos atributos serem do tipo privado, mas também existe um método designado por `private void readXML(String filename)` que analisa o argumento de entrada na classe *Main*. Para a correta descodificação do XML foi desenvolvido um ficheiro DTD que tem o formato do XML que irá ser recebido. Caso os formatos não coincidam, o programa realiza `exit`. No projeto optou-se pelo DOM como *parser* a utilizar, logo através da análise de *tags* que identificaram os diferentes parâmetros necessários à realização do projeto foi possível realizar uma correta divisão do ficheiro.

2.5.4 *Ant*

A `Ant` classe é representativa de cada formiga. Por isso contém os dados que precisamos de saber sobre ela, que é o seu caminho atual que é uma `LinkedList` de `pathw`. Contém diversos getters e setters em relação a essa subclasse.

2.5.5 *HamiltonianCycle*

Esta classe tem os métodos para o calculo do próximo movimento da formiga. Tem como parâmetros o α, β e o grafo.

Contém o método `GetNextMove` que é chamado por um Evento `Move` que faz a escolha do movimento e atualiza o caminho da formiga. Os outros métodos são auxiliares ao mesmo.

2.5.6 *HCResults*

A classe `HCresults` tem como parâmetros um caminho único de Hamilton com o seu custo total associado. Quando há um evento de print, é escolhido o caminho com menor custo.

2.5.7 *pathw*

A classe *pathw* contém todos os métodos relacionados com a manipulação do nó a avaliar que fará parte do caminho percorrido pela formiga. Tem dois parâmetros como atributo: o índice do nó em questão e o custo implicado no deslocamento para esse nó. Cada formiga terá um caminho percorrido associado a si que corresponde a uma lista de *pathw*.

Para mais detalhe sobre as várias packages, classes e métodos nelas implementados pode-se consultar a pasta JDOC.

2.6 Algoritmo de Otimização

2.6.1 Eventos na Lista de Prioridade

Os eventos a executar estão armazenados numa Lista de Prioridade (PEC) em que o seu tempo de execução é o fator de prioridade.No inicio do programa, a colónia de formigas são

postas no PEC como eventos de movimento.Os eventos de movimento, criam outros eventos de movimento e de evaporação que se inserem na PEC ordenadamente.

2.6.2 Encontrando um caminho

Para poder encontrar os caminhos de Hamilton, as formigas vão mover-se pelo grafo de forma aleatória com probabilidades baseada nos critérios definidos no ficheiro XML. Para tal fez-se o método "ant GetNextMove(ant)"que retorna a formiga com o caminho atualizado e lança para a PEC um novo evento move e se aplicável um novo evento de evaporação. No máximo, na PEC existe 200 eventos de move e E eventos de evaporação, sendo E o número de Arestas. Para calcular o próximo vértice a ir, primeiro analisa-se os vértices adjacentes disponíveis a visitar, ou seja, que não tenham sido visitados ainda. Se não há vértices disponíveis, é escolhido um dos vértices com probabilidade equitativa, e apaga-se no vetor de caminhos da formiga, tudo após o vértice escolhido, ou seja, para o caminho é como se fosse "voltar para trás". Se houver disponíveis, calcula-se os coeficientes de cada aresta relevante e depois a probabilidade acumulativa. Chama-se o método `random_decision(prob array, int size)` que retorna o índice que a decisão aleatória escolheu. Por fim, atualiza-se o caminho da formiga com o novo movimento, o custo desse movimento. No "ExecutaEvent"do "Move", ele verifica se já fez um caminho de Hamilton, se o já fez, armazena o resultado se for único, atualiza as feromonas e adiciona eventos de evaporação caso se aplique (feromonas=0). Este evento, põe outro evento de move na PEC com o tempo dependente das condições dadas no enunciado.

3 Resultados e Simulação

Para uma boa e crítica avaliação da nossa implementação foram criados 5 ficheiros de teste para, juntamente com o ficheiro de teste fornecido pela professora `test_0.xml`, testar o funcionamento do programa desenvolvido.

Ficheiros de Teste:

São usados para testar o programa os seguintes ficheiros de teste:

- test_0.xml : Ficheiro fornecido pela professora
- test_1.xml : Ficheiro de teste para validar os parâmetros do xml.

  O peso na aresta (2,4) tem um valor negativo (-2) e portanto está errado. Ao correr o programa com este ficheiro de entrada, é criada uma `new Exception` e é impresso, o tipo de erro assim como a sua localização no ficheiro XML, sabendo o `nodeidx` e o node vizinho.
- test_2.xml : Ficheiro de teste para avaliar a influência da densidade do grafo para um *finalinst* pequeno com uma colónia grande.

  Após a simulação com este ficheiro de entrada, verifica-se que não é garantido que se encontre um ciclo Hamiltoniano , devido ao facto do tempo de simulação ser muito pequeno relativamente à dimensão do grafo.

- test_3.xml : Ficheiro de teste para avaliar a influência da densidade do grafo para um *finalinst* pequeno com uma colónia também pequena.

  Com este ficheiro de entrada, tem-se uma simulação idêntica à anterior, no entanto, como só existe uma formiga na colónia, a probabilidade é ainda menor.

- test_4.xml : Ficheiro de teste para avaliar um *pLevel* baixo

- test_5.xml : Ficheiro de teste para avaliar um *pLevel* elevado

4 Conclusão e Análise Crítica

A realização deste projeto permitiu o aprofundamento dos conhecimentos sobre *java* obtidos na cadeira de Programação Orientada a Objetos. O projeto permitiu a familiarização e utilização de muitos conceitos da disciplina, como por exemplo: classes abstratas, interfaces, polimorfismo, LinkedLists, arrays de objetos, entre outros.

É de salientar que a construção do UML foi essencial para a estruturação do trabalho facilitando bastante a implementação do projeto.

Por fim, no que diz respeito aos resultados obtidos, pode afirmar-se que o grupo está satisfeito tendo em consideração que todos os objetivos foram atingidos.

Anexo A

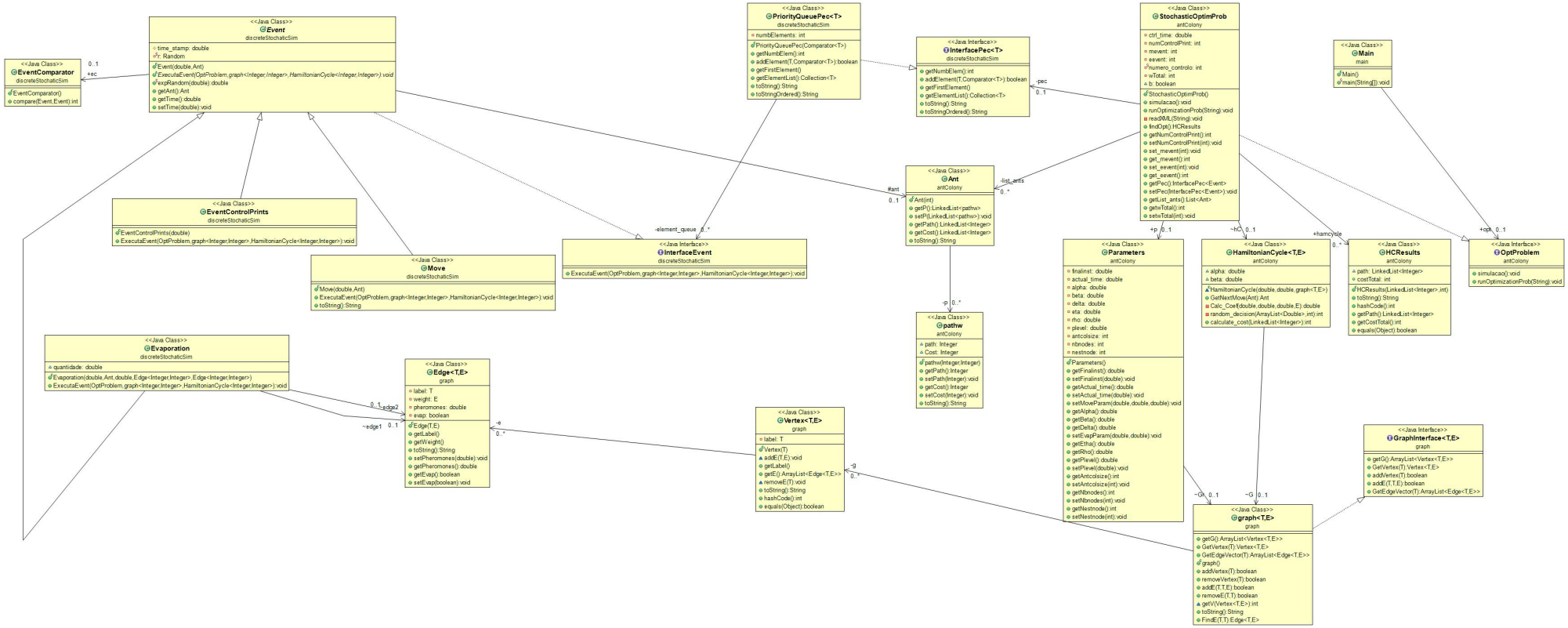

Figura 5: UML geral da implementação