Instituto Superior Técnico

MEEC

Programação Orientada por Objectos

Problema do Caixeiro-Viajante

Alunos:	Número
Alexandre Filipe	83991
José Rocha	94304
Sofia Salgueiro	84185

Grupo 27

9 de Maio de 2019

Conteúdo

1 Introdução

O problema do caixeiro-viajante, que consiste na determinação da menor rota que passe por todos os pontos de interesse uma única vez e comece e termine no mesmo ponto (Ciclo Hamiltoniano), é um dos problemas de otimização NP-completo, ou seja, não foi possível até agora desenvolve um algoritmo que o solucione em tempo polinomial. Apesar de, e tal como o nome sugere, este problema poder ser descrito como um vendedor que quer fazer as suas entregas, voltando ao ponto de origem, através do menor caminho possível, pode também ser aplicado em problemas tais como máquinas de perfuração de circuitos, agendamento, entre outros.

Neste projeto pretende-se obter uma obter uma aproximação da solução óptima através de uma técnica probabilística, neste caso, o algoritmo de optimização de colónias de formigas. Inicialmente gera-se um número x de formigas que, começando na origem, irão percorrer o grafo, cujos nós representam os pontos de interesse e as arestas as ligações entre eles, tentando encontrar o ciclo Hamiltoniano com menor custo. Sendo assim, sempre que encontram um ciclo, deverão deixar feromonas nas arestas deste. Devido à escolha aleatória de cada aresta, mas com probabilidades diretamente proporcionais às feromonas e inversamente proporcionais ao custo da mesma, as formigas começam, depois de andarem um tempo "sem rumo", a convergir para o caminho óptimo.

Sendo assim, tirando partido de Programação Orientada por Objetos, tem-se como objetivo criar uma aplicação extensível que realize a simulação de uma colónia de formigas a executar o algoritmo referido, num grafo dado como parâmetro e definido através de um ficheiro XML, utilizando para isto uma lista de eventos que executará todos os passos da colónia sequencialmente.

2 Funcionamento do programa

Em primeiro lugar, para iniciar o desenvolvimento do programa, desenhou-se o UML do mesmo. Posteriormente, ao longo do tempo, este foi alterado e resultou no UML final, figura (1).

Em seguida, descreve-se o funcionamento geral do programa, que pode também ser observado na figura (2)(a).

Inicialmente, o programa começa por receber a localização do ficheiro XML que contém os dados da estrutura do sistema. O processo de leitura dos dados provenientes do XML é feito através da desserialização dos nós do ficheiro. Implementado o *parser* - extração de informação - do documento, é feita a instanciação de um objecto do tipo *Initializer*. Este conterá todas as características do sistema de simulação, desde atributos como o tamanho da colónia de formigas (*AntColSize*) até à lista de arestas do grafo (*edges*) em causa. Existem uma série de regras que validam os dados do ficheiro, inicialmente através do ficheiro *simulation.dtd* e posterior verificação ponto a ponto, assegurando que o sistema criado será válido - não é possível ter um número negativo de formigas, o instante final da simulação deve ser maior que zero e não podem existir índices não-positivos para os nós do grafo.

Após lida a informação esta é registada na classe de *AntSimulator* onde, com base nesta, cria o grafo e este em si cria a lista de adjacências entre os nós dados.

Depois de se ter todos os dados para realizar a simulação em si, cria-se então um evento de movimento por formiga e um de observações que depois se adicionam à pilha de eventos PEC. Esta pilha reduz-se a uma *Priority Queue* que recebe objetos do tipo *Event* e que é ordenada pelo *timestamp* em que estes ocorrem, fazendo assim que, ao retirar um evento do topo da pilha, este

seja sempre o próximo/mais recente. Sendo que todos os eventos possibilitam a criação de eventos, o programa executa um ciclo onde se vai retirando e executando os eventos no topo da PEC até que esta esteja vazia, algo que ocorre quando se atinge o instante final da simulação, visto não ser possível criar eventos com *timestamp* superior ao instante final.

Referindo agora aos eventos em si, estes podem de ser de 3 tipos: de observação, de evaporação e de movimento de formiga.

O de observação é adicionado no início para acontecer em 1/20 do instante final, imprimindo na consola as informações de qual é o instante atual, a quantidade de eventos de evaporação e de movimento e, finalmente, o melhor ciclo hamiltoniano encontrado, caso já exista um. No fim da impressão é colocada a próxima observação na PEC com *timestamp* de mais 1/20 do instante final caso esta não seja a última, garantindo assim que estas são realizadas 20 vezes, igualmente espaçadas.

O evento de evaporação, sendo este criado pelas formigas após descoberto um ciclo, resume-se a decrementar o nível de feromonas da correspondente aresta, de acordo com os parâmetros lidos. Caso que, com a redução o nível de feromonas se mantenha positivo, este evento em si agenda uma nova evaporação para essa aresta.

Finalmente, o evento de movimento da formiga simula, tal como o próprio nome indica, a chegada da formiga a um nó e a tomada de decisão sobre que nó visitar em seguida. Dada a complexidade deste evento, fez-se um *flowchart* do mesmo, observável na figura (2)(b).

Então, em primeiro lugar, quando chega a um nó, a formiga verifica se o nó a que chegou é a origem e, consequentemente, se completou um ciclo Hamiltoniano. Caso isto se verifique, calcula o peso total do caminho e compara com o peso do menor caminho encontrado até agora na simulação. Se o novo caminho for menor, atualiza o melhor caminho na simulação, bem como o peso do mesmo.

Posteriormente, obtêm-se todos os nós adjacentes ao nó atual e verifica-se se há nós por visitar. Se houver nós por visitar, caso seja apenas um, a formiga segue diretamente para esse nó. Caso contrário, havendo mais do que um nó por visitar, a formiga calcula o custo C_{ijk} de cada aresta que liga ao nó adjacente e o custo total. Então, a formiga escolhe aleatoriamente um destes nós com uma probabilidade diretamente proporcional a C_{ijk} e inversamente proporcional ao custo total.

Caso não haja nós por visitar, a formiga verifica se já visitou todos os nós do grafo e pode ir para a origem. Caso isto se verifique, vai diretamente para a origem. Caso contrário, escolhe aleatoriamente um dos nós adjacentes, podendo ser a origem, com distribuição uniforme.

Escolhido o nó seguinte, a formiga calcula o tempo que demorará a percorrer a aresta e, caso este instante seja inferior ao instante final da simulação, coloca o seu próximo evento na PEC.

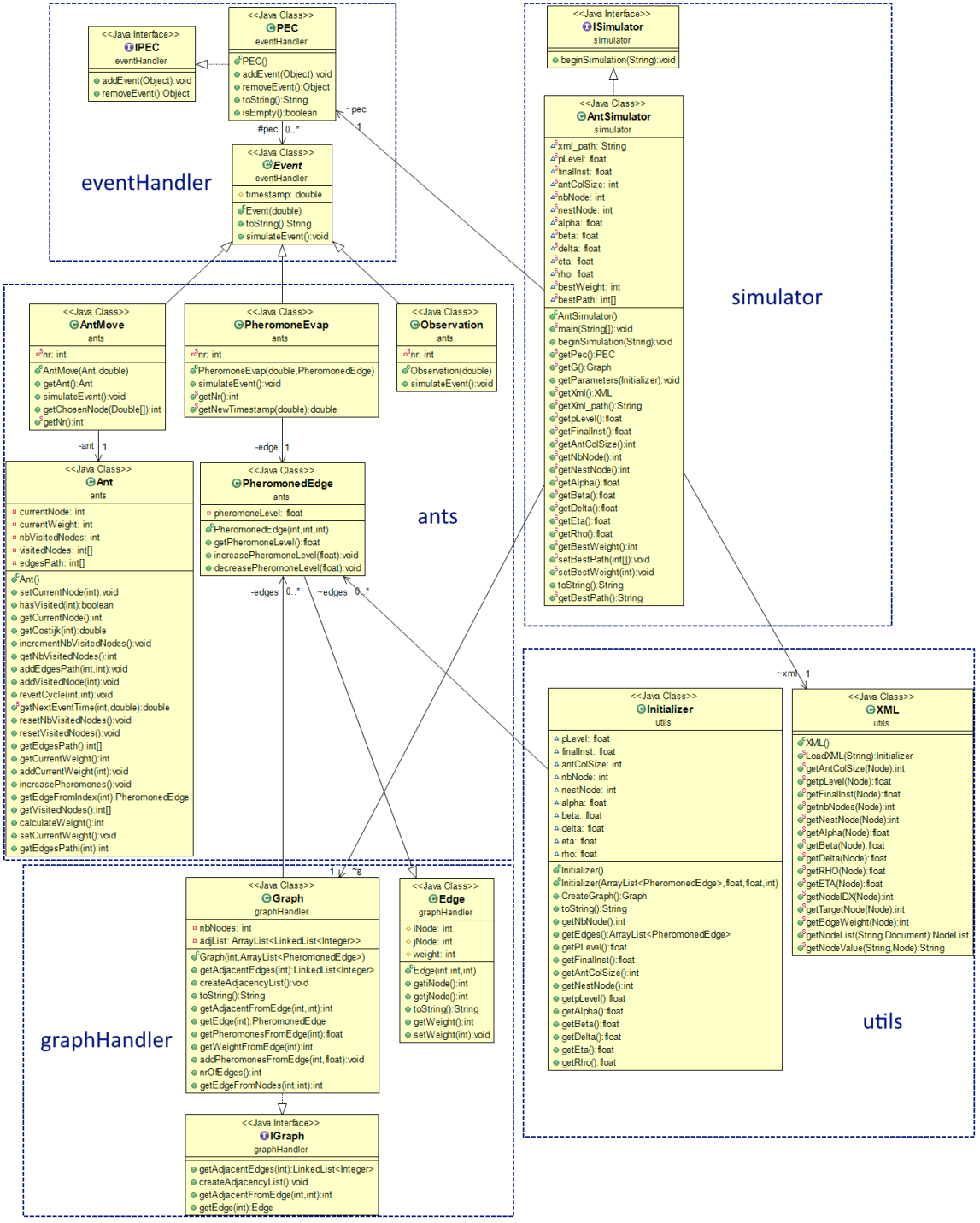
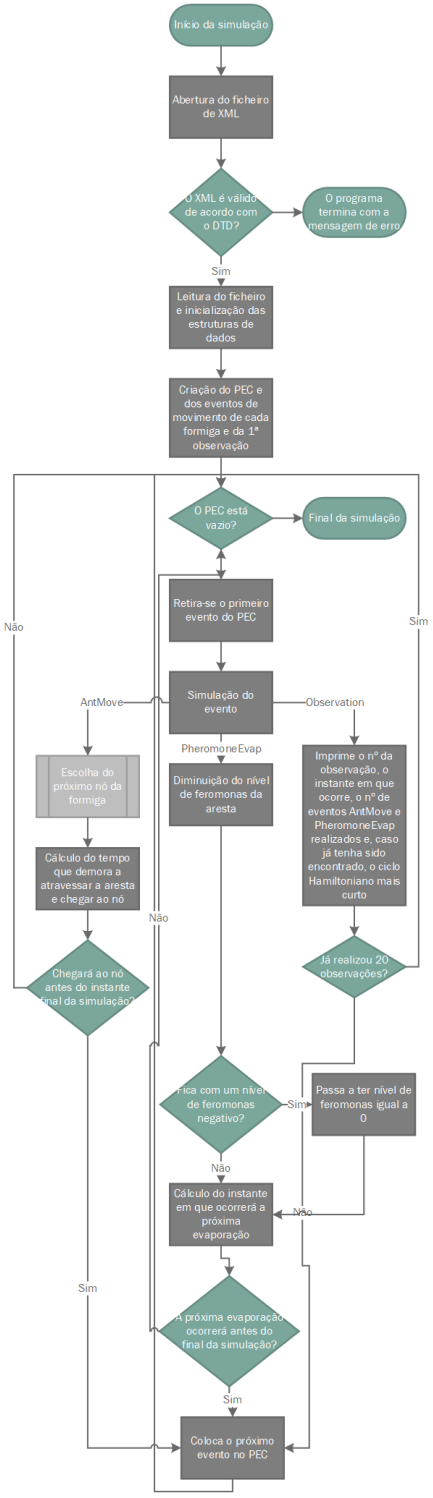
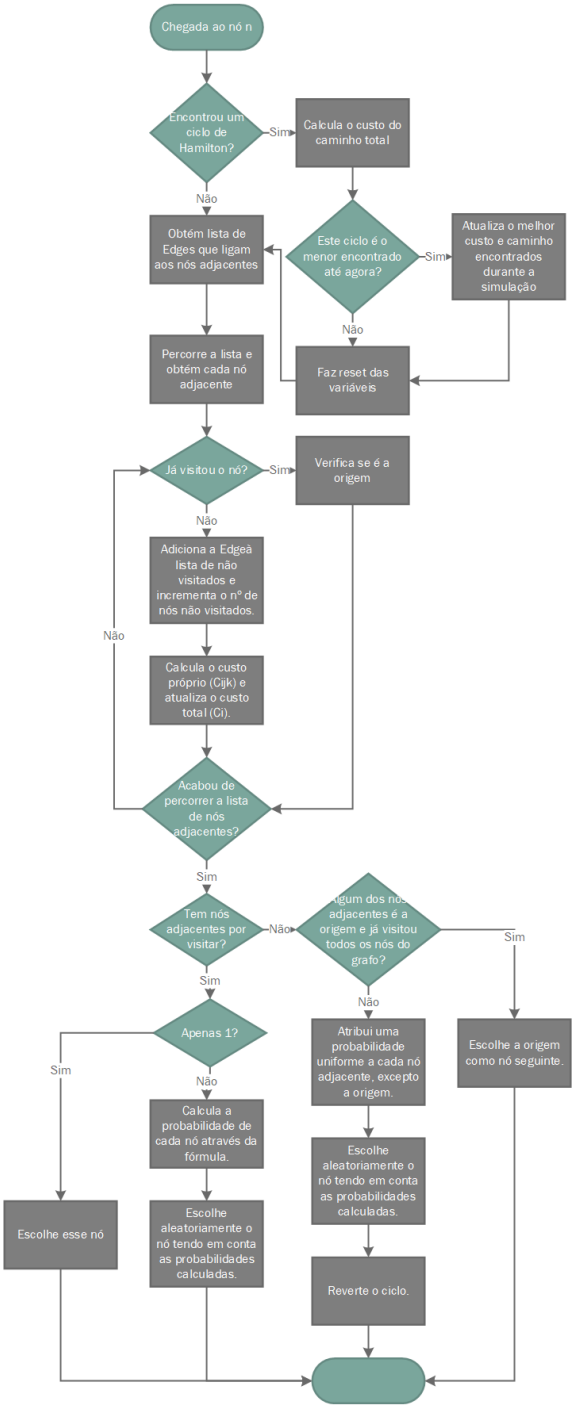

eventHandler
simulator
ants
graphHandler
utils
<<Java Interface>> IPEC eventHandler
addEvent(Object):void
removeEvent():Object
<<Java Class>> PEC eventHandler
PEC()
addEvent(Object):void
removeEvent():Object
toString():String
isEmpty():boolean
~pec 1
#pec 0..*
<<Java Class>> Event eventHandler
timestamp: double
Event(double)
toString():String
simulateEvent():void
<<Java Interface>> ISimulator simulator
beginSimulation(String):void
<<Java Class>> AntSimulator simulator
xml_path: String
pLevel: float
finalInst: float
antColSize: int
nbNode: int
nestNode: int
alpha: float
beta: float
delta: float
eta: float
rho: float
bestWeight: int
bestPath: int[]
AntSimulator()
main(String[]):void
beginSimulation(String):void
getPec():PEC
getG():Graph
getParameters(Initializer):void
getXml():XML
getXml_path():String
getpLevel():float
getFinalInst():float
getAntColSize():int
getNbNode():int
getNestNode():int
getAlpha():float
getBeta():float
getDelta():float
getEta():float
getRho():float
getBestWeight():int
setBestPath(int[]):void
setBestWeight(int):void
toString():String
getBestPath():String
<<Java Class>> AntMove ants
nr: int
AntMove(Ant,double)
getAnt():Ant
simulateEvent():void
getChosenNode(Double[]):int
getNr():int
<<Java Class>> PheromoneEvap ants
nr: int
PheromoneEvap(double,PheromonedEdge)
simulateEvent():void
getNr():int
getNewTimestamp(double):double
<<Java Class>> Observation ants
nr: int
Observation(double)
simulateEvent():void
-ant 1
-edge 1
<<Java Class>> Ant ants
currentNode: int
currentWeight: int
nbVisitedNodes: int
visitedNodes: int[]
edgesPath: int[]
Ant()
setCurrentNode(int):void
hasVisited(int):boolean
getCurrentNode():int
getCostijk(int):double
incrementNbVisitedNodes():void
getNbVisitedNodes():int
addEdgesPath(int,int):void
addVisitedNode(int):void
revertCycle(int,int):void
getNextEventTime(int,double):double
resetNbVisitedNodes():void
resetVisitedNodes():void
getEdgesPath():int[]
getCurrentWeight():int
addCurrentWeight(int):void
increasePheromones():void
getEdgeFromIndex(int):PheromonedEdge
getVisitedNodes():int[]
calculateWeight():int
setCurrentWeight():void
getEdgesPathi(int):int
<<Java Class>> PheromonedEdge ants
pheromoneLevel: float
PheromonedEdge(int,int,int)
getPheromoneLevel():float
increasePheromoneLevel(float):void
decreasePheromoneLevel(float):void
-edges 0..*
~edges 0..*
~xml 1
<<Java Class>> Initializer utils
pLevel: float
finalInst: float
antColSize: int
nbNode: int
nestNode: int
alpha: float
beta: float
delta: float
eta: float
rho: float
Initializer()
Initializer(ArrayList<PheromonedEdge>,float,float,int)
CreateGraph():Graph
toString():String
getNbNode():int
getEdges():ArrayList<PheromonedEdge>
getPLevel():float
getFinalInst():float
getAntColSize():int
getNestNode():int
getpLevel():float
getAlpha():float
getBeta():float
getDelta():float
getEta():float
getRho():float
<<Java Class>> XML utils
XML()
LoadXML(String):Initializer
getAntColSize(Node):int
getpLevel(Node):float
getFinalInst(Node):float
getnbNodes(Node):int
getNestNode(Node):int
getAlpha(Node):float
getBeta(Node):float
getDelta(Node):float
getRHO(Node):float
getETA(Node):float
getNodeIDX(Node):int
getTargetNode(Node):int
getEdgeWeight(Node):int
getNodeList(String,Document):NodeList
getNodeValue(String,Node):String
1 ~g
<<Java Class>> Graph graphHandler
nbNodes: int
adjList: ArrayList<LinkedList<Integer>>
Graph(int,ArrayList<PheromonedEdge>)
getAdjacentEdges(int):LinkedList<Integer>
createAdjacencyList():void
toString():String
getAdjacentFromEdge(int,int):int
getEdge(int):PheromonedEdge
getPheromonesFromEdge(int):float
getWeightFromEdge(int):int
addPheromonesFromEdge(int,float):void
nrOfEdges():int
getEdgeFromNodes(int,int):int
<<Java Class>> Edge graphHandler
iNode: int
jNode: int
weight: int
Edge(int,int,int)
getiNode():int
getjNode():int
toString():String
getWeight():int
setWeight(int):void
<<Java Interface>> IGraph graphHandler
getAdjacentEdges(int):LinkedList<Integer>
createAdjacencyList():void
getAdjacentFromEdge(int,int):int
getEdge(int):Edge

Figura 1: UML



(a) Execução global do programa



(b) Simulação do evento de movimento de uma formiga

Figura 2

3 Estruturas de dados

Relativamente a estruturas de dados exclusivas de linguagens orientadas a objetos, o projeto contém 3 interfaces, de modo a que, caso seja necessária alguma mudança posteriormente para um problema diferente, mantenha as bases em comum. Estas são a do simulador, sendo no caso presente implementado um simulador de formigas, a da PEC que neste utiliza como objeto em particular objetos do tipo Evento e, finalmente, a do grafo que para o problema foi utilizado um grafo particular que usa arestas com níveis de feromonas.

Para além das interfaces, também de modo a tratar do problema de uma forma genérica, as arestas utilizadas resultam da herança da classe de arestas, tendo as utilizadas a particularidade das feromonas, e os eventos que também podendo ser de vários tipos, resultam com polimorfismo da classe abstrata nos eventos de formiga mover-se, as feromonas decrementarem ou serem retiradas observações, tendo todos estes em comum da classe evento o instante em que ocorrem e um método que as executa.

4 Críticas à implementação e *performance* do programa

Finalizado o programa, toda a estrutura foi revista e foi possível identificar alguns pontos que poderiam ter sido melhorados e outros que se pensa terem contribuído favoravelmente para a performance do programa:

- **Lista de Adjacências & Vetor de Arestas -** a lista de adjacências permite, principalmente em grafos esparsos, aceder da forma mais eficiente aos nós adjacentes. No entanto, na implementação desenvolvida, foi criado um *ArrayList* de uma *LinkedList* de inteiros, que identificavam o índice da aresta no vetor de arestas do grafo. Sendo assim, após uma análise mais cuidadosa, foi possível concluir que teria sido melhor criar, na mesma, um *ArrayList*, mas, desta vez, de uma *LinkedLists* de arestas, *PheromonedEdges*. Seria mais eficiente aceder diretamente à aresta em si do que ter de aceder ao vetor que a contém.
- **Eventos de Observação -** na PEC existe, no máximo, um evento de observação. Isto significa que, no início da simulação, é colocado na PEC o 1º evento de observação e, posteriormente, apenas no final da simulação de cada evento de observação é que é colocado o evento seguinte. Esta solução é mais eficiente que colocar de início todos os eventos na PEC, pois implicaria um maior número de comparações ao adicionar um evento de movimento de formiga e de evaporação de feromonas, que ocorrem em número bastante mais elevado que o de observações.
- **Movimento da formiga -** Na escolha do nó da formiga o código foi desenvolvido para ser o mais eficiente possível, principalmente porque a maior parte do tempo de execução do programa decorre neste evento. Então, o uso da lista de adjacências permitiu uma eficiente "descoberta"dos nós adjacentes e, por exemplo, teve-se o cuidado de melhorar pequenos pormenores, tal como a formiga ir diretamente para o nó adjacente, se apenas tiver um por visitar, e não perder tempo no cálculo das probabilidades, neste caso.

5 Ficheiros XML

Foram criados 5 ficheiros XML para verificar a correta implementação do programa. Em seguida, descreve-se sumariamente cada ficheiro e correspondente objetivo:

- **1 -** Exemplo fornecido na página da cadeira, com apenas 1 formiga
- **2 -** Pequeno grafo, 4 nós, com um nº elevado de formigas, 500
- **3 -** Grafo médio, 10 nós, com um pequeno nº de formigas, 10
- **4 -** Grafo grande e completamente interligado, 20 nós, com um nº médio de formigas, 200
- **5 -** XML com 2 parâmetros inválidos, nº de formigas negativo e nó destino negativo

6 Conclusão

Sumariando, é possível identificar as vantagens que uma implementação através de uma linguagem de programação orientada a objetos trouxe à solução.

Em primeiro lugar, o nível de *Abstração* que este paradigma oferece é claramente um ponto positivo visto que, apesar de o problema em questão ser particular para a otimização da colónia de formigas, este programa pode facilmente ser alterado para qualquer outro que utilize grafos, pilhas de eventos e/ou um simulador.

Adicionalmente, a utilização de polimorfismo nos eventos, e herança nas arestas, possibilitou uma generalização do programa.

Considera-se também ter ido ao encontro de uma das principais técnicas da programação orientada a objetos que é o *Encapsulamento*. Através desta técnica, consegue-se certificar da segurança das entidades da aplicação, garantindo que elementos exteriores não tenham acesso direto às propriedades dos objetos, pondo em causa a segurança e coerência do sistema.