Universidade do Porto
FEUP Faculdade de Engenharia

Conceção e análise de algoritmos

Sistema de evacuação de turistas numa montanha

Fábio Filipe Jesus Silva – ei11107

Fernando Manuel Rocha Magalhães – ei07046

José Pedro Lobo Marinho Trocado Moreira – ei12002

Porto, 26 de abril de 2013

Resumo

Este trabalho contextualiza-se num sistema de evacuação de turistas perdidos numa montanha. Tal evacuação é simulada com o auxílio de um sistema de grafos para determinação dos melhores percursos possíveis.

Nos grafos apresentados o peso das arestas é o tempo necessário para a percorrer.

Pela montanha estão dispersos vários pontos: pontos com turistas e pontos com veículos e pontos vazios. À data da evacuação, cada ponto turístico deve ser socorrido pelo veículo do ponto mais próximo. Não sendo o veículo capaz de socorrer todos os turistas de uma só vez, deve este fazer novas viagens ou outro mais próximo fazê-lo. Esta escolha é feita durante a execução baseando-se nas distâncias relativas de cada veículo.

Os pontos citados tanto podem ser determinados previamente (usando ficheiros de texto) como durante o programa (via linha de comandos). No caso dos ficheiros de texto, é usada uma sintaxe própria reconhecida pelo programa, sendo que cada linha deve conter uma e uma só instrução.

As instruções possiveis são:

1. Adicionar o ponto de nome x ao gráfo: AP x
2. Adicionar y turistas ao ponto de nome x: AT x y
3. Adicionar uma ligação entre o ponto de nome x e o ponto de nome y com um peso z: AR x y z
4. Defenir o ponto de nome x como a saida: PR x

Espera-se que, com este *software*, seja realizada uma evacuação o mais rápida e eficazmente possível.

Universidade do Porto
FEUP Faculdade de Engenharia

Índice

Principais algoritmos usados

O funcionamento do programa baseia-se em grande parte no algoritmo de Dijkstra para o calculo de caminho mais curto em grafos pesados.

Este algoritmo apresenta uma complexidade temporal de O(E +V*log(V)). A complexidade espacial é linear no número de vertices (V).

Este algoritmo não é no entanto o unico a operar no nosso programa. Um ciclo tipico de utilização do programa começa pela verificação de quais são os pontos do gráfo mais próximos em média de todos os outros.

Isto é alcançado com o uso do algoritmo de Dijkstra repeditas vezes (1 vez por cada vértice do grafo) .A complexidade desta operação é portanto O(V*O(Dijkstra))= =O(V*(E +V*log(V))). Enquanto estes calculos são efectuados os seus resultados (distancias entre os pontos, e caminhos) são guardados em duas matrizes VxV que permitiram mais tarde durante utilizar estes mesmos cálculos em outras operações sem os ter que repetir, sendo assim de notar o uso da programação dinâmica (complexidade espacial V^2).

Todos estes cálculos supra mencionados são usados para encontrar o melhor local para colocar os veículos de resgate (veículos serão colocados nos pontos cuja soma das distâncias a todos os outros for menor, no máximo de 1 por ponto).

Segue-se a execução da evaquação propriamente dita: fazendo uso das matrizes previamente calculadas. A operação de resgate consiste em:

1. Selecionar o ponto onde se deve resgatar no imediato os turistas
    - Os pontos com turistas são guardados numa heap, cuja organização tem complexidade O(P) onde P é o numero de pontos com turistas. Os pontos são organizados nesta heap de forma a que no topo esteja o ponto com mais gente por resgatar. O acesso ao elemento é feito portanto em tempo constante O(P).
    - Pensamos ser de destacar a euristica “gananciosa” utilizada por nós aqui
2. Selecionar o veículo mais proximo.
    - Complexidade linear no número de veículos, uma vez que se deve percorrer todo o vector com os veículos e ir ver qual a sua distancia ao ponto onde se deve ir resgatar os turistas, uma vez que calcular a distancia corresponde simplesmente a aceder a um elemento numa matrix temos para o calculo da distância complexidade constante, dando assim uma complexidade total na seleção do veículo O(Ve) onde Ve é o numero de veículos.
3. Determinar o caminho de evaquação e mover os turistas

- Complexidade linear no numero de vértices, uma vez que a determinação do caminho consiste em recursivamente ir seguindo o caminho indicado nos campos de uma matrix. O(V)

Nenhuma das operações mencionadas tem qualquer complexidade espacial assinalavel.

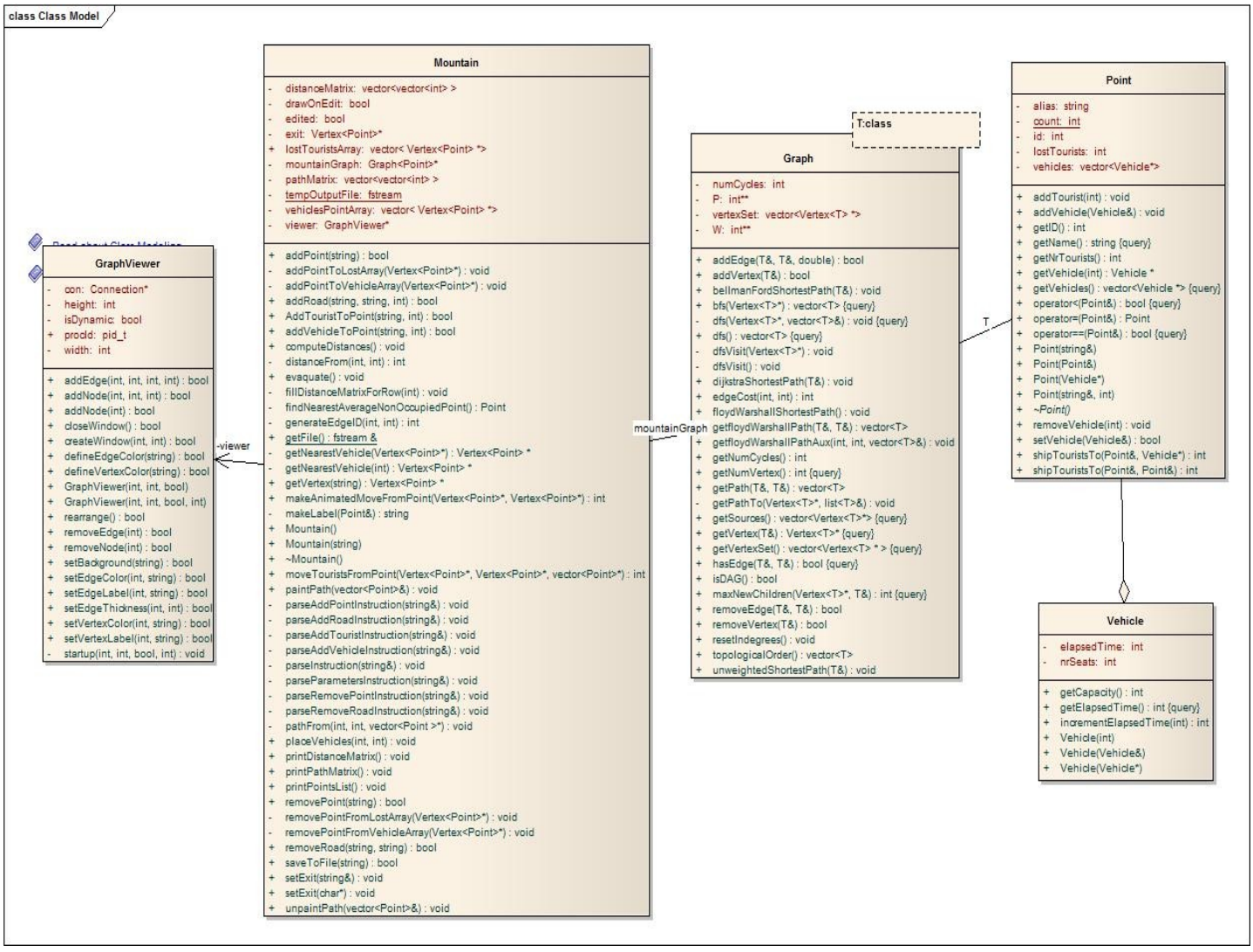
O processo assima descrito é efectuado no máximo T vezes, onde T é o numero de turistas, mas será tipicamente efectuado muito menos vezes (só assim seria se os turistas estivessem 1 em cada ponto ou os veículos tivessem todos capacidade unitária), concluimos então que esta segunda parte do programa tem complexidade O(T*(V+Ve+P)).

O algoritmo no seu todo terá então uma complexidade de execução de:

O(T*(V+Ve+P) + V*(E +V*log(V))) onde tipicamente a parte mais pesada será a segunda parcelo fazendo com que a complexidade temporal se possa aproximar bastanted te O(V*(E +V*log(V)))) que por sua vez na maior parte dos casos deve representativos da realidade modelada onde E<<V*log(V),tipicamente poderá ser mesmo aproximadamente: $O(V^2 log(V))$.

Quanto a complexidade espacial total, esta é como já referido V^2, uma vez que são necessárias duas matrizes VxV para guardar os dados dos calculos que são reutilizados.

UML


class Class Model
Mountain
- distanceMatrix: vector<vector<int> >
- drawOnEdit: bool
- edited: bool
- exit: Vertex<Point>*
+ lostTouristsArray: vector< Vertex<Point> *>
- mountainGraph: Graph<Point>*
- pathMatrix: vector<vector<int> >
- tempOutputFile: fstream
- vehiclesPointArray: vector< Vertex<Point> *>
- viewer: GraphViewer*
+ addPoint(string) : bool
- addPointToLostArray(Vertex<Point>*) : void
- addPointToVehicleArray(Vertex<Point>*) : void
+ addRoad(string, string, int) : bool
+ AddTouristToPoint(string, int) : bool
+ addVehicleToPoint(string, int) : bool
+ computeDistances() : void
- distanceFrom(int, int) : int
+ evaquate() : void
- fillDistanceMatrixForRow(int) : void
- findNearestAverageNonOccupiedPoint() : Point
- generateEdgeID(int, int) : int
+ getFile() : fstream &
- getNearestVehicle(Vertex<Point>*) : Vertex<Point> *
- getNearestVehicle(int) : Vertex<Point> *
+ getVertex(string) : Vertex<Point> *
+ makeAnimatedMoveFromPoint(Vertex<Point>*, Vertex<Point>*) : int
- makeLabel(Point&) : string
+ Mountain()
+ Mountain(string)
+ ~Mountain()
+ moveTouristsFromPoint(Vertex<Point>*, Vertex<Point>*, vector<Point>*) : int
+ paintPath(vector<Point>&) : void
- parseAddPointInstruction(string&) : void
- parseAddRoadInstruction(string&) : void
- parseAddTouristInstruction(string&) : void
- parseAddVehicleInstruction(string&) : void
- parseInstruction(string&) : void
- parseParametersInstruction(string&) : void
- parseRemovePointInstruction(string&) : void
- parseRemoveRoadInstruction(string&) : void
- pathFrom(int, int, vector<Point >*) : void
+ placeVehicles(int, int) : void
+ printDistanceMatrix() : void
+ printPathMatrix() : void
+ printPointsList() : void
+ removePoint(string) : bool
- removePointFromLostArray(Vertex<Point>*) : void
- removePointFromVehicleArray(Vertex<Point>*) : void
+ removeRoad(string, string) : bool
+ saveToFile(string) : bool
+ setExit(string&) : void
+ setExit(char*) : void
+ unpaintPath(vector<Point>&) : void
GraphViewer
- con: Connection*
- height: int
- isDynamic: bool
+ procId: pid_t
- width: int
+ addEdge(int, int, int, int) : bool
+ addNode(int, int, int) : bool
+ addNode(int) : bool
+ closeWindow() : bool
+ createWindow(int, int) : bool
+ defineEdgeColor(string) : bool
+ defineVertexColor(string) : bool
+ GraphViewer(int, int, bool)
+ GraphViewer(int, int, bool, int)
+ rearrange() : bool
+ removeEdge(int) : bool
+ removeNode(int) : bool
+ setBackground(string) : bool
+ setEdgeColor(int, string) : bool
+ setEdgeLabel(int, string) : bool
+ setEdgeThickness(int, int) : bool
+ setVertexColor(int, string) : bool
+ setVertexLabel(int, string) : bool
- startup(int, int, bool, int) : void
-viewer
mountainGraph
T:class
Graph
- numCycles: int
- P: int**
- vertexSet: vector<Vertex<T> *>
- W: int**
+ addEdge(T&, T&, double) : bool
+ addVertex(T&) : bool
+ bellmanFordShortestPath(T&) : void
+ bfs(Vertex<T>*) : vector<T> {query}
- dfs(Vertex<T>*, vector<T>&) : void {query}
+ dfs() : vector<T> {query}
- dfsVisit(Vertex<T>*) : void
- dfsVisit() : void
+ dijkstraShortestPath(T&) : void
+ edgeCost(int, int) : int
+ floydWarshallShortestPath() : void
+ getfloydWarshallPath(T&, T&) : vector<T>
+ getfloydWarshallPathAux(int, int, vector<T>&) : void
+ getNumCycles() : int
+ getNumVertex() : int {query}
+ getPath(T&, T&) : vector<T>
- getPathTo(Vertex<T>*, list<T>&) : void
+ getSources() : vector<Vertex<T>*> {query}
+ getVertex(T&) : Vertex<T>* {query}
+ getVertexSet() : vector<Vertex<T> * > {query}
+ hasEdge(T&, T&) : bool {query}
+ isDAG() : bool
+ maxNewChildren(Vertex<T>*, T&) : int {query}
+ removeEdge(T&, T&) : bool
+ removeVertex(T&) : bool
+ resetIndegrees() : void
+ topologicalOrder() : vector<T>
+ unweightedShortestPath(T&) : void
T
Point
- alias: string
- count: int
- id: int
- lostTourists: int
- vehicles: vector<Vehicle*>
+ addTourist(int) : void
+ addVehicle(Vehicle&) : void
+ getID() : int
+ getName() : string {query}
+ getNrTourists() : int
+ getVehicle(int) : Vehicle *
+ getVehicles() : vector<Vehicle *> {query}
+ operator<(Point&) : bool {query}
+ operator=(Point&) : Point
+ operator==(Point&) : bool {query}
+ Point(string&)
+ Point(Point&)
+ Point(Vehicle*)
+ Point(string&, int)
+ ~Point()
+ removeVehicle(int) : void
+ setVehicle(Vehicle&) : bool
+ shipTouristsTo(Point&, Vehicle*) : int
+ shipTouristsTo(Point&, Point&) : int
Vehicle
- elapsedTime: int
- nrSeats: int
+ getCapacity() : int
+ getElapsedTime() : int {query}
+ incrementElapsedTime(int) : int
+ Vehicle(int)
+ Vehicle(Vehicle&)
+ Vehicle(Vehicle*)

Casos de uso

- O utilizador tem a possibilidade de adicionar e remover todo o tipo de componentes (ponto turísticos, ponto de veículos de socorro, caminhos entre pontos), assim como despoletar a evacuação.
- O programa faz a gestão dos pontos, computando os caminhos mais curtos e “mobilizando” os veículos adequados.
- O utilizador recebe informação gráfica acerca do mapeamento. Se ativar a evacuação, recebe informação textual sobre quais veículos socorrem quais pontos turísticos.
- A evaquação é feita passo-a-passo com feedback para o utilizador por forma a facilitar a visualização do que está a acontecer.

Principais dificuldades e componente individual

Durante a elaboração do trabalho, as dificuldades surgiram principalmente devido ao fator tempo, pois haviam (e continuam a haver) trabalhos de outras unidades curriculares em curso, o que de certa maneira não nos permitiu dedicar ao trabalho todo o tempo que desejavamos, enriquecendo-o eventualmente com mais funcionalidades e/ou melhoramentos na sua eficiência.

No entanto o programa executa as funcionalidades desejadas e demonstra a aplicação dos conhecimentos adquiridos, que é o objetivo pretendido. Não só do algoritmo principal do programa mas também de outros algoritmos acessórios recorrendo frequentemente a euristicas quer gananciosas que de programação dinâmica.

O trabalho pensamos nós, revela ainda um dominio bastante satisfatório da linguagem e uso adequado de estroturas de dados e boa práticas de programação.

Em termos algoritmicos resultou um pouco da junção de conhecimentos adquiridos ao longo da componente teórica e prática da disciplina.