DEEC
DEPARTAMENTO DE ENGENHARIA ELETROTÉCNICA E DE COMPUTADORES
TÉCNICO LISBOA

Mestrado Integrado em Engenharia Eletrotécnica e de Computadores (MEEC)

Algoritmos e Estrutura de Dados

AEDMAPS

Relatório do Projeto 2º Semestre 20/21

Grupo nº38

Pedro Miguel Lopes Jacinto nº97352 Mail: pedro.m.l.jacinto@tecnico.ulisboa.pt

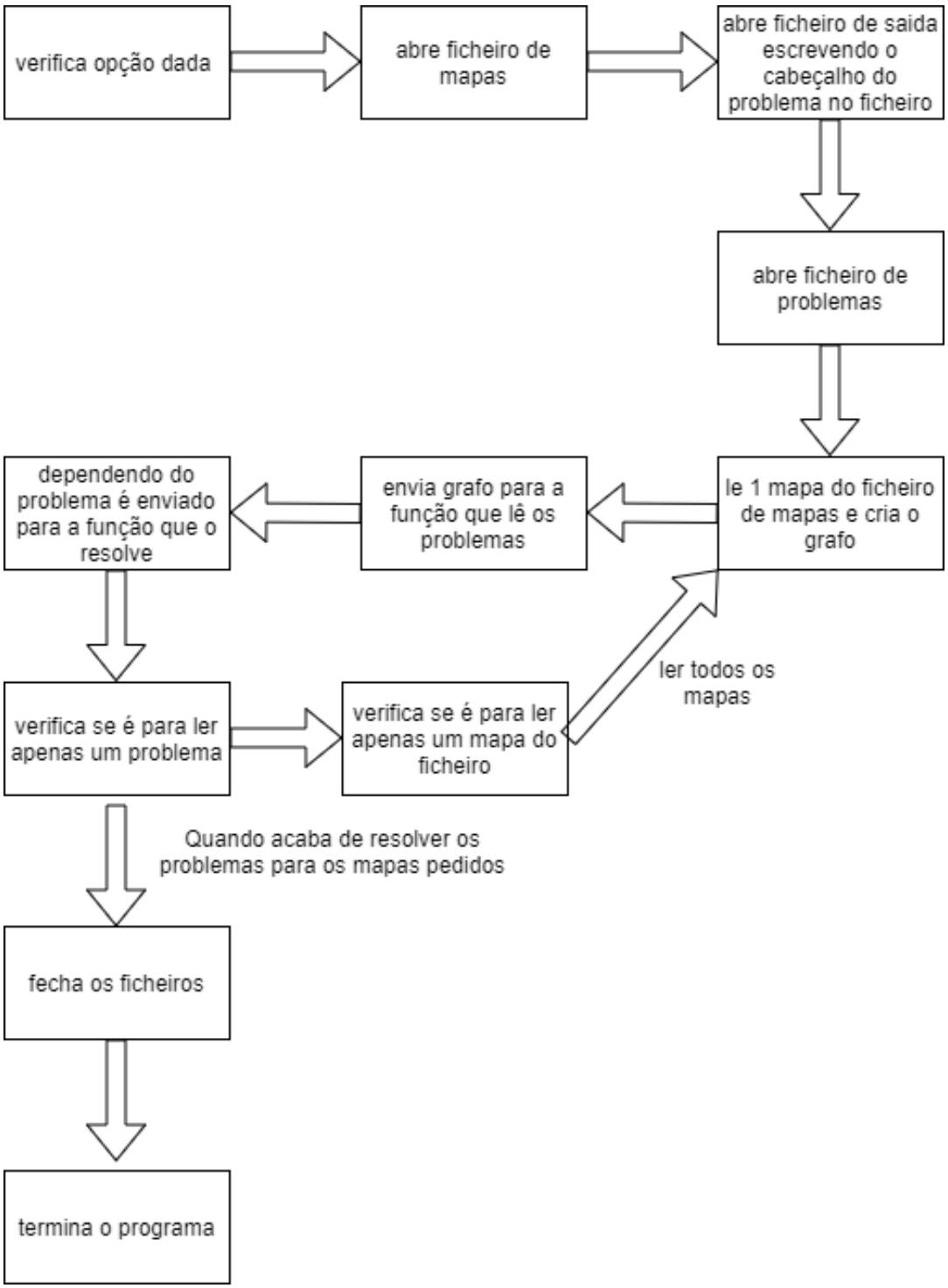
Docente responsável: Carlos Bispo

Índice:

1. Descrição do problema

Neste projeto pretendeu-se desenvolver um programa capaz de produzir caminhos entre povoações de um mapa e que esses caminhos obedeçam a um determinado conjunto de restrições predefinidas.

2. Arquitetura do programa



O fluxograma anterior mostra de maneira sucinta o funcionamento do programa. Na função que lê os problemas, estes são encaminhados para as funções modoA1,modoB1, modoC1 ou modoD1 que resolvem os problemas lidos recebendo os argumentos lidos e o grafo criado na função que lê os mapas.

3. **Descrição da estrutura de dados**

Struct edge:

Double **weight** -> custo da aresta;

Int **VertexId** -> vértice ao qual está ligado;

Pointer para a próxima aresta;

Struct vertex:

Lista de arestas do vértice;

Int **degree** -> grau do vértice;

Int **state** -> estado do vértice (utilizado nos modos C0 e D0);

Char **settlement** -> ponto de interesse do vértice;

Int **parent** -> pai do vértice (utilizado nos modos A1, C1 e D1);

Double **parentcost** -> custo da aresta para o parente do vértice;

Struct GraphAdjLst:

Int **NrVertices** -> número total de vértices do grafo;

Int **NrEdges** -> número total de arestas do grafo;

Array de vértices;

Lista de arestas;

4. Descrição dos modos e algoritmos utilizados

Na primeira parte do projeto para os modos A0 e B0 não foi utilizado nenhum algoritmo pois a maneira escolhida para guardar os mapas lidos continha em cada vértice o seu grau (pedido em A0) e, para cada vértice, um array com todos os seus vértices adjacentes (pedido em B0 para verificar se dois vértices tinham um via direta que os unia. Para os modos C0 e D0 foi utilizado o algoritmo breadth first search, com uma complexidade temporal de O(V+E), sendo V o numero total de vértices e E o numero total de arestas. O algoritmo foi alterado para obter o que os modos pediam. Foi criada uma estrutura ITEM que permite guardar o vértice e a quantas etapas ele está do vértice pedido. No modo C0 a função bfsC0 retorna o numero de passos pedidos no problema se o vértice dado tiver vizinhos a k etapas dele. Na função modoC0 verifica-se isso, caso o numero retornado seja diferente do numero de etapas pedido significa que o vértice não tem vizinhos a k etapas. No modo D0 pretende-se contar o numero de vizinhos de um dado vértice a k etapas. A função bfsD0 encarrega-se disso mesmo, comparando o numero de passos de cada vértice com o numero pedido.

Na segunda parte do projeto foi necessário implementar o algoritmo de Dijkstra, que tem uma complexidade temporal de O(ElogV), sendo E numero de arestas e V numero de vértices. Este algoritmo permite encontrar a menor distância entre dois vértices pertencentes ao grafo. Alterei o algoritmo de maneira a guardar na estrutura vertex (definida na secção 3) o pai do vértice e o custo desta ligação, assim correndo o Dijkstra começando no destino dado até encontrar o vértice inicial dado ficava com uma arvore em que o o vértice final estava no topo. Bastando correr os vértices, começando no vértice inicial dado e passando sucessivamente para o pai do vértice até encontrar o vértice final dado para conseguir imprimir o caminho entre os dois com os seus custos.

A função modoA1 faz exatamente isso, começa por verificar se os vértices pertencem ao grafo e se não são o mesmo vértice, caso isto se verifique envia os dois vértices dados para a função Dijkstra que retorna -1 se não houver caminho entre os vértices ou retorna o custo do caminho caso exista.

A função modoB1 tem como objetivo encontrar o caminho mais curto entre dois vértices passando num ponto de interesse dado no enunciado do problema e com uma restrição de custo. Para isto comecei por fazer a verificação que os vértices dados pertencem ao grafo e que não são o mesmo vértice. Depois corri o Dijkstra normal para saber se existe caminho entre os vértices e saber o custo desse caminho, depois verifiquei se o ponto de interesse dado já faz parte do caminho de menor custo entre os dois vértices. Caso isso não se verifique o programa vai procurar os vértices que contêm o ponto de interesse pedido no enunciado do problema e, utilizando a função dijkstraB1, vemos o caminho do vértice inicial para o ponto de interesse e o caminho do ponto de interesse para o vértice final. Faz-se isto para todos os vértices que contenham o ponto de interesse e guarda-se o que tenha menor custo, guardando o custo em minpreco e guardando os caminhos em minpathStart e minpathEnd. Quando forem corridas todas as hipóteses de caminho entre os dois vértices passando nos vértices que contenham o ponto de interesse verifica-se se existe caminho ou se não existe caminho. Se não existir caminho escreve-se a resposta no ficheiro de saída. Calcula-se o preço em excesso e compara-se com a restrição de custo dado, caso exceda a restrição dada o programa escreve a resposta no ficheiro de saída. Depois, verifica-se se algum dos caminhos é nulo, caso seja não existe caminho e o programa escreve no ficheiro de saída a reposta. Se nada disto se verificou é porque correu tudo bem e o programa encontrou caminho entre os dois vértices passando no ponto de interesse pedido e cumprindo a restrição de custo dada e assim sendo escreve a reposta no ficheiro de saída tal como o caminho que se deve adotar.

A função modoC1 tem como objetivo encontrar o caminho de menor custo entre dois vértices no caso de um vértice do caminho original esteja interdito. Comecei por verificar

se os vértices dados pertencem ao grafo e se não são o mesmo vértice, caso isto se verifique o programa corre a função dijkstra e caso esta tenha retornado -1 significa que não existe caminho entre os dois vértices e assim sendo escreve e a reposta no ficheiro de saída. Caso exista, o programa vai contar o número de passos total deste caminho e guarda o vértice interdito. O programa volta a correr a função dijkstra mas agora com este vértice interdito como 3º argumento. O que o dijktra vai fazer é quando recebe um vértice como 3º argumento não altera na estrutura vertex o parente e o parentcost e quando encontra o vértice dado como 3ºargumento na lista de edges de um vértice ele ignora-o o e salta para o próximo vértice na lista de edges do vértice. Assim o dijkstra vai retornar o custo do novo caminho, se existir, ou -1 caso não exista um novo caminho entre os dois vértices. Depois o programa vai escrever no ficheiro de saída a resposta ao problema.

A função modoD1 tem como objetivo encontrar o caminho de menor custo entre dois vértices se uma via do caminho original estiver interdita. Comecei por verificar se os vértices dados pertencem ao grafo e se não são o mesmo vértice. Seguindo o mesmo raciocínio da função modoC1, da mesma maneira que retiro o vértice interdito em C1, retiro a via interdita guardando os índices dos vértices que fazem a aresta. Enviando estes dois vértices para a função dijkstra, esta quando está a correr a lista de arestas de um dado vértice vai verificas se esse vértice é um dos proibidos e se o identificador do vértice na estrutura edge é um dos vértices proibidos e saltar essa ligação quando isto se verifica. Se o dijsktra retornar -1 é porque não existe novo caminho sem aquela aresta. Caso contrário existe novo caminho e o programa escreve as repostas no ficheiro de saída.

5. Análise dos requisitos computacionais

Na parte inicial do programa, quando está a ler o mapa do ficheiro, o programa vai fazer a alocação de memória para o array de vértices, com um tamanho de V*sizeof(VertexGraphAdjLst) e para a lista de arestas, de tamanho 2*E*sizeof(EdgeGraphAdjLst), multiplica-se por dois pois, utilizando listas de adjacência, cada ligação aparece na lista de edges de dois vértices.

As duas funções bfsC0 e bfsD0 utilizam o algoritmo de breadth first search com uma completixade temporal de O(V+E), sendo V número de vértices e E número de arestas.

As funções modo da segunda parte utilizam todas o algoritmo de Dikstra quem tem uma complexidade temporal de O(E + VlogV), sendo V numero de vértices e E numero de arestas.

6. Exemplo

Decidi utilizar como argumentos de entrada "–ao GuppyB1.prbs e mapa04.maps"

- O programa cria o nome do ficheiro de saída -> mapa04.queries;
- Verifica se a opção dada é valida através da função ReadOption que retorna 1 pois a opção "-ao" é válida;
- Abre o ficheiro de mapas fornecido, neste caso mapa01.maps;
- Cria o ficheiro de saída com o nome anteriormente gravado -> mapa04.queries;
- Abre o ficheiro de problemas;
- Lê o número de vértices e de arestas do ficheiro de mapas;
- Aloca memoria para o array de vértices e para a lista de arestas;
- Começa por colocar em cada vértice o settlement lido do ficheiro;
- Cria as ligações dadas no ficheiro de mapas, guardando na lista de cada um vértice o vértice a que esta ligado e o custo da ligação;
- Envia este grafo criado, o ficheiro de problemas, o ficheiro de saída e o argv(para saber se é suposto ler apenas um problema ou todos) para a função ReadProblem;
- Esta função retira do ficheiro de problemas os enuciados e consoante o modo que peçam envia para a função que o resolve;
- Quando sair da função ReadProblem faz rewind no pointer do ficheiro, para o caso de existirem mais mapas no ficheiro de mapas e se terem de resolver outra vez os problemas;
- Liberta a memória alocada para o grafo;
- Verifica se a opção é para ler todos os mapas do ficheiro de mapas ou só um;
- Neste caso sendo a opção –ao vai ler so o primeiro mapa;
- O programa liberta a memoria alocada para o nome do ficheiro de saída;
- O programa fecha os ficheiros abertos;