Trabalho Prático 2 – Algoritmos I

Projeto de ciclovia para a cidade de Belleville

Gilliard Gabriel Rodrigues
Matrícula: 2019054609

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil

gilliard2019@ufmg.br

1. Introdução

O problema proposto foi implementar um sistema que, dadas as identificações dos pontos turísticos que formam cada trecho da cidade de Belleville, assim como seus valores turísticos e o custo para a construção de cada trecho, retornasse a melhor configuração para uma ciclovia, sem ciclos, que conectasse todos os pontos de interesse, com o menor custo de construção possível e priorizando a construção de trechos mais atrativos.

Ademais, definiu-se a atratividade de um trecho da ciclovia como sendo a soma dos valores turísticos dos dois pontos de interesse conectados por esse trecho. Também foi solicitado que, para dois pontos de interesse quaisquer, haja somente uma alternativa de trecho possível.

Portanto, com base nessas informações, é possível notar que tal problema pode ser modelado através de um grafo valorado não direcionado. Para solucioná-lo, deve ser encontrada uma árvore geradora mínima que, além de obedecer ao critério de possuir o menor peso total, que aqui será o custo de construção, também deve priorizar os trechos/as arestas com maior atratividade. Baseado nisso, será apresentada nesta documentação uma solução que resolva o problema, baseada no algoritmo de Kruskal, com as devidas modificações.

2. Implementação

O programa foi desenvolvido na linguagem C++, compilada pelo compilador G++ da GNU Compiler Collection.

2.1 Estrutura de Dados e Algoritmos

A estrutura de dados utilizada foi a de um grafo valorado não direcionado, utilizando arestas com dois pesos: o custo de construção e a atratividade. O grafo foi representado como um vetor de arestas. Foram implementadas três classes: “ConjuntoDisjunto”, “Aresta” e “Grafo”, cujos atributos e métodos serão apresentadas adiante. O algoritmo escolhido para encontrar a árvore geradora mínima (AGM) foi o algoritmo de Kruskal, com algumas modificações conforme as necessidades do TP.

2.2 Classes e seus métodos

A fim de seguir as boas práticas de programação e implementar um código modularizado, o programa foi dividido em 7 arquivos: “conjuntoDisjunto.h”, “conjuntoDisjunto.cpp”, “aresta.h”, “aresta.cpp”, “grafo.h”, “grafo.cpp” e “main.cpp”.

A classe *ConjuntoDisjunto* possui 3 atributos privados, são eles: um inteiro *n* (para representar o número de conjuntos disjuntos), um ponteiro para um *array* de inteiros chamado *representante* (como o próprio nome já diz, vai conter os representantes de cada conjunto) e um ponteiro para outro array de inteiros chamado *rank* (que armazenará o rank de cada conjunto). Além disso, há também um construtor (público), que recebe como parâmetro um inteiro, inicializa o *n* com esse inteiro recebido, além de utilizar um *for* para popular o *array representante* com “i” (i = 0, 1, 2, ..., n-1) em todas as posições (inicialmente cada elemento é representante de si mesmo) e popular o *array rank* com “0” em todas as posições (já que inicialmente todos os vértices estão em conjuntos diferentes e possuem rank 0). Há também um destrutor que desaloca a memória alocada dinamicamente para os *arrays*.

Além disso, a classe *ConjuntoDisjunto* possui 2 métodos públicos, que serão explicados separadamente a seguir.

O primeiro método da classe *ConjuntoDisjunto* chama-se *buscaConjunto* e trata-se de um método *int*, que recebe como parâmetro um inteiro *u* e busca recursivamente pelo representante do conjunto que possui o elemento *u*. Existem várias implementações para esse método, foi escolhida a implementação com otimização de compressão de caminhos, a fim de que a complexidade de tempo melhorasse de O(V) para O(log V).

O segundo método da classe *ConjuntoDisjunto* chama-se *uneConjuntos* e trata-se de um método *void*, que recebe como parâmetros dois inteiros *u* e *v* e une dois subconjuntos em um único conjunto. Existem várias implementações para esse método, foi escolhida a implementação com otimização de união por ranks.

Já a classe *Aresta* possui 4 atributos privados, são eles: um inteiro *u* (representando um vértice), um inteiro *v* (representando outro vértice, que é adjacente a u), um inteiro *custo* (representando o custo de construção da aresta/do trecho) e um inteiro *atratividade* (como o próprio nome já diz, representando a atratividade da aresta/do trecho).

Além dos 4 atributos privados, a classe *Aresta* possui um construtor público, que inicializa os atributos de acordo com os parâmetros recebidos. Por fim, essa classe também possui 4 métodos *getters* (públicos), que retornam o valor de cada atributo da classe, assim como possui um método booleano (público) que sobrescreve o operador “<” para que mais à frente, quando o *std::sort* for utilizado no algoritmo de Kruskal, as arestas sejam ordenadas de acordo com os critérios exigidos pelo TP (ordem crescente de custo, e nos casos em que o custo de duas arestas comparadas for o mesmo, a prioridade será aquela com maior atratividade).

Por outro lado, a classe *Grafo* possui 4 atributos privados, sendo eles: um inteiro *V* (representando o número de vértices do grafo), um *vector<Aresta> G* (para armazenar as arestas do grafo), um *vector<Aresta> AGM* (que irá armazenar as arestas que compõem a árvore geradora mínima, ou melhor dizendo, os trechos que compõem a ciclovia) e um ponteiro para um array de inteiros chamado *valores_turisticos* (como o próprio nome sugere, armazenará os valores turísticos dos pontos de interesse). Além

disso, essa classe possui também um construtor, que inicializa o atributo *V* com o valor recebido como parâmetro e aloca memória dinamicamente para o *array valores_turisticos* (com tamanho igual ao número de vértices do grafo).

Além dos atributos privados, a classe *Grafo* possui 5 métodos públicos, que serão explicados separadamente a seguir.

O primeiro método chama-se *insereValoresTuristicos* e trata-se de um método *void*, que popula o array *valores_turisticos* com a entrada do usuário.

O segundo método chama-se *adicionaArestaPonderada* e trata-se de um método *void*, que recebe como parâmetro dois inteiros representando dois vértices/pontos de interesse e dois outros inteiros representando o custo e a atratividade, instancia um objeto do tipo *Aresta*, passando os parâmetros do método como argumentos para o construtor da classe *Aresta* e, em seguida, adiciona a aresta ponderada ao grafo, através da função *push_back*.

O terceiro método chama-se *inserePropostas* e trata-se também de um método *void*, que recebe como parâmetro um inteiro E (representando o número de arestas ou quantidade de trechos possíveis) e lê da entrada do usuário os pontos de interesse e o custo do trecho, além de calcular a atratividade do trecho somando os valores turísticos dos dois pontos e, por fim, chama o método *adicionaArestaPonderada*, passando como parâmetros os dois pontos recebidos, o custo de construção e a atratividade do trecho, que adiciona a aresta ao grafo.

O quarto método chama-se *kruskal* e trata-se de um método *void*, sem parâmetros, que nada mais é do que o algoritmo de kruskal, com a diferença de que o método que ordena as arestas do início (*std::sort*) irá ordenar de acordo com os critérios exigidos pelo TP: por ordem crescente de custo de construção e em caso de empate no custo, dará prioridade àquela aresta com maior atratividade. O método *kruskal* funciona da seguinte forma: primeiro ordena as arestas com o *std::sort*, em seguida instancia um objeto do tipo *ConjuntoDisjunto* chamado *conjunto_disjunto*, passando como parâmetro o número de vértices do grafo (*V*), logo após há um *for* que itera no tamanho do grafo e chama a função *buscaConjunto* (da classe *ConjuntoDisjunto*) para os dois vértices de cada aresta do grafo, armazenando os identificadores dos conjuntos em duas variáveis *u* e *v*, em sequência há um *if* que verifica se *u* é diferente de *v* (se forem diferentes, significa que a união de ambos os conjuntos não forma um ciclo), e se essa condição for verdadeira, adiciona a aresta da i-ésima posição de *G* na *AGM* através do método *push_back* e, em seguida, chama o método *uneConjuntos* (da classe *ConjuntoDisjunto*), passando como parâmetros *u* e *v*, unindo ambos os conjuntos em *conjunto_disjunto*. Após as iterações acabarem, o vetor de arestas AGM possuirá todas as arestas que compõem a árvore geradora mínima (ciclovia de Belleville) e o destrutor da classe *ConjuntoDisjunto* será chamado para o objeto *conjunto_disjunto*, que no final desse método já não será mais útil.

O quinto método chama-se *imprimeResultado* e trata-se também de um método *void*, sem parâmetros, que imprime as saídas esperadas, ou seja, imprime o custo de construção mínimo e a atratividade agregada da ciclovia na primeira linha, a quantidade de trechos chegando/saindo de cada ponto de interesse na segunda linha e, por fim, os trechos selecionados (um por linha) no mesmo formato da entrada a partir da terceira linha.

Por fim, há também um destrutor que, além de limpar o grafo e a AGM, também desaloca a memória alocada dinamicamente para o *array valores_turisticos*.

2.3 Estrutura do main

No *main()*, ocorrem as declarações de variáveis, além da leitura da primeira linha, composta pela quantidade de pontos turísticos (número de vértices) e quantidade de trechos possíveis (número de arestas), a instanciação de um objeto do tipo *Grafo*, seguido pelas chamadas dos métodos *insereValoresTuristicos*, *inserePropostas*, *kruskal*, *imprimeResultado* e, por fim, o destrutor da classe *Grafo* é chamado.

2.5 Instruções de compilação e execução

Para compilar o trabalho, acesse o diretório dos arquivos do TP via terminal e digite "*make"*. Em seguida, para executar o código, digite "*./tp02".*

Logo após, na primeira linha, insira a quantidade de pontos de interesse (N), seguida pela quantidade de trechos possíveis (T), já na segunda linha, insira os N valores turísticos de cada ponto de interesse separados por espaço e nas próximas T linhas insira 3 inteiros separados por espaço (representando dois pontos de interesse e o custo do trecho entre os dois). Observação: os identificadores dos pontos iniciam em 0, ou seja, na entrada e saída, P0 é identificado por 0, P1 por 1, e assim por diante.

Para ler a entrada de um arquivo, basta digitar "./tp02 < nomeDoArquivo.txt".

3. Análise de complexidade de tempo

Para chegar a uma conclusão sobre o custo de complexidade do programa, serão analisadas as chamadas da função *main(),* passo a passo. Seja "V" a quantidade de pontos de interesse e de "E" a quantidade de trechos possíveis.

A primeira chamada trata-se do método *insereValoresTuristicos*. Esse método possui apenas um *for* que itera V vezes lendo a entrada do usuário. Portanto, sua complexidade de tempo é da ordem de $O(V)$.

A segunda chamada trata-se do método *inserePropostas*. Esse método possui um *for* que itera E vezes lendo as entradas do usuário e após calcular a atratividade efetuando uma soma, ele chama o método *adicionaArestaPonderada*, que possui custo O(1). Com isso, tem-se que sua complexidade de tempo é da ordem de $O(E)$.

A terceira chamada trata-se do método *kruskal*. Dentro desse método há o *std::sort*, cuja ordem de complexidade é de O(E log E), já que ordena as arestas do grafo. Em seguida, há um *for* que itera E vezes e dentro desse *for* são chamados os métodos *buscaConjunto* e *uneConjuntos* (da classe *ConjuntoDisjunto*), cujas complexidades são da ordem de $O(\log V)$ e O(1), respectivamente. Com isso tem-se que a complexidade de tempo geral é O(E log E + E log V). Sabe-se que $|E| < |V|^2$, além disso: $\log |E| \leq \log |V|^2 \rightarrow \log |E| \leq 2 \log |V| \rightarrow \log |E| = O(\log V)$. Portanto, a complexidade desse método é da ordem de $O(E \log V)$.

A quarta chamada trata-se do método *imprimeResultado*. Esse método possui 4 *for*'s que iteram sobre o número de arestas da AGM (V-1) e um *for* que itera sobre o número de vértices (V) do grafo G. Logo, a complexidade de tempo desse método é da ordem de $O(V)$.

Portanto, analisando todos esses métodos e sabendo que, tendendo ao infinito, V e E são equivalentes (uma vez que $V\text{-}1 \leq E < V^2$ para um grafo conexo e não direcionado), conclui-se que a complexidade de tempo que domina o programa é a do método *kruskal*, da ordem de $O(E \log V)$.

4. Conclusão

Os pontos mais importantes para conseguir implementar o sistema foram, sem dúvidas, conseguir modelar o problema usando grafos não direcionados, saber que se tratava de um problema de árvore geradora mínima, além de saber modificar o algoritmo de Kruskal de acordo com os requisitos do problema.

No decorrer desse trabalho, ficou evidente a importância de fazer testes para ter certeza de que o algoritmo estava funcionando de forma correta. Por fim, foi dada uma atenção especial ao processo de refatoração e modularização do código a fim de atender às boas práticas de programação e facilitar a identificação e correção de erros.

Em suma, foi possível praticar o uso de grafos para modelar problemas e também ver na prática a aplicação de uma árvore geradora mínima, complementando o aprendizado adquirido durante as aulas.