Trabalho Prático 2

Vitor Rodarte Ricoy - 2019007112

Universidade Federal de Minas Gerais (UFMG)

Belo Horizonte – MG – Brasil

vitorricoy@ufmg.br

1. Introdução

O problema proposto foi implementar um programa que dados os possíveis trechos de ciclovia entre pontos turísticos de uma cidade e a atratividade desses pontos determine a construção de uma ciclovia entre todos esses locais, de forma que o custo dessa construção seja o menor possível e a atratividade dos trechos construídos seja a maior dentre as possíveis construções de custo mínimo.

Algumas conclusões triviais dessa definição do problema são que a ciclovia construída não pode conter ciclos e que ela deve conectar todos os pontos, ou seja, nessa ciclovia deve ser possível alcançar todos os pontos a partir de cada ponto.

Por fim, o programa deve imprimir o custo mínimo e a atratividade agregada de um ciclovia que cumpre as condições apresentadas. Além disso, também deve ser exibida a quantidade de trechos que cada ponto de interesse está contido na configuração escolhida da ciclovia e a relação dos trechos selecionados, da mesma forma que são lidos pelo programa.

2. Modelagem Computacional do Problema

Para modelar o problema foi utilizado grafos, mais especificamente, um grafo não direcionado ponderado. Os vértices do grafo são os pontos turísticos e as arestas representam um possível trecho de ciclovia entre dois pontos. O peso das arestas é definido como um par formado pelo custo da construção do trecho e pela atratividade desse trecho, que é calculada como a soma das atratividades dos dois vértices conectados por ele.

Com a modelagem do grafo descrito acima, a solução para o problema se torna encontrar a árvore geradora mínima do grafo modelado, sendo que uma aresta é considerada de menor peso do que outra caso o custo de construção dela seja menor ou caso esse custo seja igual, mas ela possua uma atratividade maior. Essa definição de comparação das arestas permite que ao construir a árvore geradora mínima seja encontrado o menor custo de construção com a maior atratividade possível.

Após construir a árvore geradora mínima, resta calcular as saídas do problema. Para obter o custo total da construção e a atratividade agregada basta somar ambos os valores de cada aresta da árvore gerada. Já o número de trechos que contém cada ponto de interesse é o mesmo que o grau de cada vértice na árvore geradora mínima, portanto basta calcular esse valor. Por fim, a impressão das arestas escolhidas é feita facilmente imprimindo as arestas da árvore.

3. Estrutura de Dados e Algoritmos

3.1. Estrutura de Dados

Para a solução foram utilizadas três estruturas de dados: um grafo não-direcionado ponderado, um union-find usado para implementar o algoritmo de Kruskal e um tipo `ValorAresta` para representar o valor de uma aresta do grafo. Vale notar que o grafo foi representado como uma lista de arestas, implementada com um `vector` da STL de valores do tipo `ValorAresta`.

3.1.1. Union-find

O union-find foi implementado em uma classe `UnionFind`. A implementação da estrutura

foi feita com base na explicação do union-find mais eficiente do livro-texto da disciplina. A representação escolhida para a estrutura de dados é a de ponteiros, onde cada vértice aponta diretamente ou transitivamente para o vértice-pai do conjunto a que ele pertence. Inicialmente todo vértice aponta para ele mesmo, já que ele é o vértice-pai do conjunto que contém apenas ele.

A classe criada possui duas variáveis internas: um vetor que guarda o vértice para o qual cada vértice aponta e um vetor que guarda o número de vértices que pertencem a cada conjunto. Já o construtor da classe aponta o ponteiro de todos os vértices para si mesmo e atribui 1 ao número de elementos de todos os subconjuntos. A classe também define duas operações: buscar o conjunto a que um determinado vértice pertence e unir os conjuntos de dois vértices.

A operação de buscar o conjunto de um determinado vértice é responsável por percorrer os ponteiros recursivamente até chegar ao elemento que aponta a si mesmo, que é o vértice-pai que identifica um conjunto. Vale notar que foi implementada a otimização de atualizar os ponteiros dos vértices visitados pela busca para que apontem diretamente ao vértice-pai que representa o conjunto, o que evita que seja necessário percorrer essa cadeia de ponteiros novamente em outra busca.

Por fim, a operação de unir dois vértices primeiramente obtém o vértice-pai que identifica o conjunto de cada vértice que será unido. Após isso, caso os vértices não pertençam ao mesmo conjunto, a raíz do conjunto que possui menos vértices deixa de ser uma raíz e passa a apontar para a raíz do conjunto que possui mais vértices.

3.1.2. Tipo ValorAresta

O tipo *`ValorAresta`* foi definido para facilitar a manipulação dos dados da aresta, já que cada aresta contém dois vértices, um custo de construção e um valor de atratividade. Além disso, a classe *`ValorAresta`* implementa o operador `<`, que é usado pela função *`sort`* da STL no algoritmo de Kruskal para ordenar corretamente as arestas em ordem crescente de custo e decrescente de atratividade, nessa ordem.

O operador `<` foi implementado para determinar se uma aresta é considerada menor do que outra de acordo com a definição de comparação das arestas descrita na modelagem do problema. Ou seja, esse operador retorna verdadeiro somente se a aresta da esquerda possui um custo menor ou se ambas as arestas possuem um custo igual e a da esquerda possui uma atratividade maior.

3.2. Algoritmos

Foram implementados dois algoritmos: o algoritmo de Kruskal para construir uma árvore geradora mínima e um algoritmo para gerar as saídas esperadas pelo problema a partir da árvore geradora mínima obtida.

3.2.1. Kruskal

O algoritmo de Kruskal implementado recebe o número de vértices do grafo e o grafo na forma de uma lista de arestas. O algoritmo inicialmente constrói um union-find em que cada vértice está em um conjunto apenas com ele mesmo e ordena as arestas do grafo de acordo com a comparação definida em *`ValorAresta`*.

Após isso, para cada aresta é analisado se ela forma um ciclo no grafo sendo construído, o que é feito por meio da verificação dos conjuntos, que devem ser diferentes, dos vértices da aresta no union-find. Caso a aresta não forme um ciclo, ela é adicionada no grafo e os conjuntos dos dois vértices da aresta são unidos. Após percorrer todas as arestas, a árvore geradora mínima foi construída. O pseudocódigo do algoritmo de Kruskal implementado é o seguinte:

```
Kruskal(n, grafo)
    inicializa unionFind como um union-find com n conjuntos de um elemento
```

```
inicializa vetor_retorno como um vetor de arestas vazio
ordena o grafo na forma de lista de arestas na ordem crescente de acordo com o custo de construção e em ordem decrescente de atratividade, nessa ordem
para cada aresta na lista de arestas
    salva em endpoint1 e endpoint2 os dois vértices da aresta
    se endpoint1 e endpoint2 pertencem a conjuntos diferentes no union-find
        une os conjuntos de endpoint1 e endpoint2
        insere a aresta em vetor_retorno
retorne vetor_retorno
```

O algoritmo de Kruskal foi escolhido para resolver esse problema de construir a árvore geradora mínima pois sua implementação é mais simples do que o algoritmo de Prim, desde que já esteja familiar com a implementação do union-find, e também porque ele possui a mesma complexidade assintótica do algoritmo de Prim, logo pode ser usado eficientemente.

3.2.2. Cálculo dos Valores da Saída

Após calculada a árvore geradora mínima, foram calculados os valores pedidos na saída do programa. Tais valores consistem na soma dos dois parâmetros de cada aresta da árvore, no grau de cada vértice e nos vértices seguidos do custo de construção de cada aresta da MST. O pseudocódigo deste processamento é o seguinte:

```
mst = kruskal(n, grafo)
inicializa custoMST e atratividadeMST como 0
inicializa grau_vertice com 0 para todo vértice do grafo
para cada aresta de mst
    custoMST += aresta.custo
    atratividadeMST += aresta.atratividade
    grau_vertice[aresta.endpoint1] += 1
    grau_vertice[aresta.endpoint2] += 1
imprime custoMST e atratividadeMST
imprime o grau de cada vértice
imprime os vértices e o custo de construção das arestas de mst
```

3.2.3. Prova do Funcionamento do Algoritmo Proposto

O algoritmo proposto deve gerar um grafo que cumpre duas condições. A primeira condição é que o grafo gerado deve ser uma árvore, portanto ser não-direcionado, acíclico e conectado, o que é verdade, já que ele é gerado pelo algoritmo de Kruskal, que sabidamente constrói uma árvore geradora, que possui essas características.

Outra característica necessária é a árvore gerada ser ótima para o problema, ou seja, ser mínima com relação ao custo de construção e possuir a maior atratividade agregada das possíveis árvores com o custo de construção mínimo. Para provar que o algoritmo gera uma árvore com essa característica, devemos provar que dada a ordenação das arestas utilizada pelo algoritmo, a árvore gerada é ótima. Segue a prova desta propriedade:

Prova. Primeiramente, definimos A como um subgrafo qualquer construído pelo algoritmo proposto. Sabemos que A é uma árvore geradora do grafo, pela corretude do algoritmo de Kruskal. Vamos provar que A é também uma árvore ótima para o problema proposto.

Primeiro supomos que as arestas de A são $e_1, ..., e_k$, e que elas estão ordenadas em ordem de

escolha, portanto e_1 foi escolhido antes de e_2 , e_2 antes de e_3 e assim por diante. Também supomos que existe uma árvore ótima A*, tal que A* tem o maior número possível de arestas em comum com A. Agora, queremos provar que A = A*.

Para isso, suponhamos que A* é diferente de A e definimos e_j como a primeira aresta escolhida em A que não é uma aresta de A*. Vale notar que as arestas $e_1, ..., e_{j-1}$ pertencem tanto a A quanto a A*. Sejam u e v os vértices dos extremos de e_j e C o caminho em A* que conecta u a v, temos que o caminho C deve ter pelo menos uma aresta a que não pertence a A, já que A* não possui a aresta e_j e A não possui ciclos, logo não pode conter todas as arestas do caminho C.

Como a aresta e_j foi escolhida na árvore gerada pelo algoritmo e a não foi, concluímos que: o custo de construção de e_j é menor que o de a; ou o custo de construção de e_j e a são iguais e a atratividade de e_j é maior que a de a; ou o custo e a atratividade das duas arestas são iguais. Caso isso não fosse verdade, a teria sido escolhida antes de e_j, já que a não forma um ciclo com as arestas $e_1, ..., e_{j-1}$ pois ela pertence a A*, e o algoritmo prioriza a escolha de arestas com menor custo de construção ou maior atratividade caso o custo de construção seja igual.

Definimos a árvore A' = A* + e_j - a. É possível perceber que A' é uma árvore geradora por sua construção e que ela ou tem um custo menor do que A*, ou um custo igual e uma atratividade maior do que A*, ou custo e atratividade iguais a A*. Isso acontece porque e_j tem um custo menor do que a ou uma atratividade maior com o mesmo custo ou ambos valores iguais. Como A* é uma árvore ótima, o custo e a atratividade de A* e de A' devem ser os mesmos, portanto A' também é uma árvore ótima. Mas podemos ver que A' tem mais arestas em comum com A do que A*, já que A* difere de A ao menos pela arestas a e e_j, enquanto A' possui a aresta e_j em comum com A.

O fato de A' ter mais arestas em comum é uma contradição à escolha de A*, portanto devemos ter que A* = A, o que prova que A é uma árvore ótima. ■

4. Análise de Complexidade de Tempo

Para a análise da complexidade de tempo do programa, primeiramente vamos analisar a complexidade de algumas partes que compõem esse programa, para facilitar a conclusão final. Também para auxiliar a análise, iremos definir `m` como o número de possíveis trechos da ciclovia e `n` como o número de pontos de interesse. Notamos que o grafo modelado possui `m` arestas e `n` vértices. As análises feitas são referentes ao custo assintótico de tempo no pior caso. Também vale notar que todas as operações do tipo `ValorAresta` são O(`1`), portanto podem ser desconsideradas ao analisar a complexidade dos trechos de código.

4.1. Entrada dos dados

A entrada dos dados é dada por um comando de leitura do número de pontos turísticos e de trechos possíveis da ciclovia, por um loop para ler a atratividade de cada ponto e outro loop para ler os pontos que são conectados por cada trecho e seu custo de construção.

A leitura das primeiras duas variáveis tem custo O(`1`), já que é feita apenas com um `cin`. Já o primeiro loop tem custo O(`n`), já que é executado um `cin` para cada ponto de interesse. Por fim, o segundo loop tem custo O(`m`), já que executa apenas comandos que custam O(`1`), para cada possível trecho da ciclovia. Portanto, toda a leitura dos dados tem complexidade O(`1+n+m`) = O(`n+m`).

4.2. Union-find

O union-find foi implementado baseado na implementação mais eficiente do livro. Primeiramente, analisando o construtor da estrutura, podemos afirmar que ele custa O(`n`), já que itera pelos pontos de interesse, realizando uma operação que custa O(`1`), que é a inicialização dos valores dos dois vetores utilizados pelo union-find.

Já a busca do conjunto de um elemento da estrutura custa O(`logn`). A prova desse custo é

apresentada após o teorema 4.24 do livro-texto, portanto será omitida para concisão dessa análise.

Por fim, a união dos conjuntos de dois elementos custa O($Logn$), diferente do custo de O(1) da estrutura descrita no livro, isso acontece pois na implementação desse trabalho a operação de união une os conjuntos dos dois vértices a partir da raíz de cada conjunto, ou seja, executa uma busca para cada vértice que será unido. Isso cria uma árvore menos profunda e uma execução mais próxima de O(1) da operação de busca, mesmo que ela continue O($Logn$) no pior caso. Além da busca, a união executa operações O(1) como atribuições, comparações e somas.

Assim, concluímos que o construtor da estrutura custa O(n) e tanto a busca quanto a união custam O($Logn$).

4.3. Algoritmo de Kruskal

O algoritmo de Kruskal primeiramente inicializa o union-find com custo O(n) e inicializa o vetor de retorno com custo O(1). Após isso, são ordenadas as arestas do grafo com custo O($mLogm$), já que a comparação entre duas arestas tem custo O(1) e a ordenação é feita por meio do $sort$ da STL, que garantidamente tem o pior caso O($NLogN$), sendo N o número de elementos do vetor.

Em seguida é feita uma iteração pelas arestas do grafo e para cada aresta são executadas operações O(1), além de um find no union-set que custa O($Logn$) e uma união que também custa O($Logn$). Assim o custo da iteração pelas arestas é de O($mLogn$). Portanto, concluímos que o algoritmo de Kruskal tem a complexidade O($n+1+mLogm+mLogn$) = O($n+mLogm+mLogn$). Como sabemos que m = O(n^2), podemos reescrever essa complexidade como O($mLogm$).

4.4. Cálculo dos Valores da Saída

O cálculo dos valores da saída consiste em comandos O(1), na criação de um vetor para guardar os graus dos vértices com custo O(n) e em uma iteração pelas arestas da árvore geradora mínima, que executa comandos O(1), custando assim O(n), já que a árvore possui $n-1$ arestas. Portanto, o cálculo dos valores de saída tem custo O($n+1+n$) = O(n).

4.5. Saída dos Dados

A saída dos dados executa um comando $cout$, que custa O(1), itera pelos vértices da MST executando comandos que custam O(1), o que custa O(n), e itera pelas arestas da MST executando também comandos que custam O(1), também com custo O(n). Portanto, essa seção do código tem o custo O($1+n+n$) = O(n).

4.6. Análise do Programa Completo

Por fim, temos que a leitura de dados é O($n+m$), a execução do algoritmo de Kruskal é O($mLogm$), o cálculo dos valores da saída é O(n) e a saída dos dados também é O(n). Também temos alguns trechos que são trivialmente menos complexos do que a execução dessas seções, como a inicialização do vetor de atratividade que custa O(n). Como o programa completo consiste em, basicamente, executar tais trechos de código, sua complexidade é igual a O($n+m+mLogm+n+n$) = O($n+mLogm$). Novamente, como temos que m = O(n^2), podemos escrever a complexidade como O($mLogm$).

Assim, concluímos que a complexidade de tempo da solução é igual a O($mLogm$).

Referências

Kleinberg, Jon (2006). Algorithm Design: Chapter 4: Greedy Algorithms. Pearson Education India.

Almeida, Jussara. Slides da Disciplina Algoritmos I. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. 2021.