Projeto Final - Linguagem de Programação II e Estruturas de Dados II
IMD0040 - T01 - 2022.2

Gregorio Pinheiro Cunha[1], **Rafael Gomes Dantas Gurgel Siqueira**[1]

[1]Instituto Metrópole Digital – Universidade Federal do Rio Grande do Norte (IMD/UFRN)

gregopc@hotmail.com, rafael.dantas.096@ufrn.edu.br

Abstract. *This project aims to solve an NP-hard problem in two different ways. The first solution consists of finding, by testing all possibilities, a tree-shaped network, with the lowest possible cost, of the n houses in a neighborhood, ensuring that a house is directly connected to, at most, d houses. The second solution proposes to find the same result with a more efficient strategy, making it possible to find the solution with a significantly lower number of tests than the total number of possibilities.*

Resumo. *Este projeto visa resolver um problema NP-difícil de dois modos distintos. A primeira solução consiste em encontrar, através do teste de todas as possibilidades, uma rede em formato de árvore, com menor custo possível, das n casas de um bairro, garantindo que uma casa esteja ligada diretamente a, no máximo, d casas. A segunda solução se propõe a encontrar esse mesmo resultado com uma estratégia mais eficiente, possibilitando encontrar a solução com quantidade de testes significativamente inferior ao total de possibilidades.*

1. Introdução

O problema das conexões das casas é do tipo NP-difícil que esta no grupo dos problemas mais difíceis da computação. Eles são utilizados para encriptação de informação, pois o cálculo do resultado é difícil de encontrar, fácil de se verificar e são resolvidos por Máquina Não Determinística em tempo polinomial. Desta forma criar algoritmos eficientes com a ajuda de algumas estruturas de dados é de extrema importância para este caso.

O problema consiste em um conjunto de n casas, onde é possível fazer conexões entre elas, cada qual com o seu próprio custo. A solução visa encontrar um conjunto de conexões que possuam o menor custo possível e que conectem todas as casas, considerando cada casa limitada a ter no máximo d conexões.

2. Detalhes do Projeto

Organização, simplicidade e eficiência são de extrema importância em qualquer projeto, por isso adotar um bom padrão de projeto e se preocupar com um bom diagrama de classe torna-se indispensável, este projeto seguiu o padrão de projeto MVC com pacotes separados para o Modelo (cuida dos dados e das operações nos dados), Visão (cuida de apresentar as informações de forma visual ao usuário) e o Controle (comunicação entre o Modelo e a Visão).

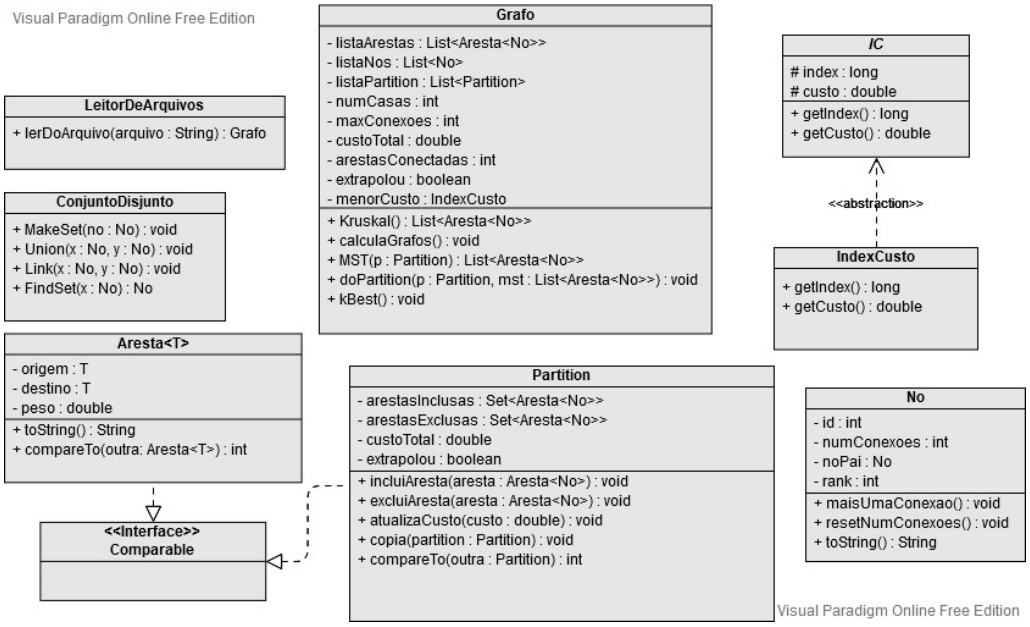

Figura 1. Diagrama de classes

2.1. Diagrama de Classes

A escolha das classes utilizadas teve como objetivo manter o projeto o mais simples possível, com uma classe *LeitorArquivo* que cuida da leitura do arquivo de entrada com tratamento de exceções, e retorna um objeto do tipo *Grafo*, que é a classe principal do projeto onde o algoritmo esta implementado.

Foi definida a classe *No* para armazenar a identificação do número da casa, o número de conexões do nó (equivalente ao grau do vértice de um grafo), uma referência para um noPai e o rank do nó. Estes dois últimos atributos foram utilizados nas operações de Conjuntos Disjuntos para representar o comportamento desta estrutura de dados.

Uma classe *Aresta* foi criada para cuidar das conexões entre as casas, armazenando dois nós para representar as casas e o respectivo custo da conexão. Nela foi utilizada a interface *Comparable* para que seja possível comparar as arestas com relação ao custo.

A classe *IC* é uma classe abstrata que possui as assinaturas dos métodos a serem implementados em uma classe descendente desta.

A classe *IndexCusto* foi feita para referenciar a arvore válida de menor custo. Ela descende da classe abstrata *IC*. Durante a execução do código ela guarda o índice que representa a árvore de menor custo encontrada até o momento e, caso seja encontrada uma árvore válida com custo menor, o índice desta passa a ser armazenado, juntamente com o novo custo mínimo.

A classe *Grafo* ficou responsável por armazenar todas as informações cruciais do programa como o numero total de casas, o máximo de conexões permitidas, a lista contendo as arestas, o processamento do algoritmo (detalhado na seção 3.2) e escrita do resultado em um arquivo.

A classe *ConjuntoDisjunto* foi desenvolvida para poder realizar as operações *MakeSet*, *Union* e *FindSet*, que permitem conectar $n - 1$ nós, cada qual representando uma

casa, sem formar ciclos, gerando árvores. Os códigos dessas operações foram desenvolvido com base no apresentado em [Cormen et al. 2009].

Por fim, a classe *Partition* representa uma partição. Ela armazena as arestas obrigatórias e as arestas proibidas da partição, juntamente com o custo associado a essa partição e uma variável booleana para indicar se a MST (Minimum Spanning Tree) referente a essa partição extrapolou o máximo de conexões permitidas para uma casa. Esta classe também implementa a interface *Comparable* para que seja possível comparar as partições com relação ao custo.

3. Descrição

3.1. Estruturas de dados utilizadas

As estruturas de dados utilizadas foram:

- **Lista.** Representada pela interface *List* e com a implementação de *ArrayList*. Esta estrutura, com essa implementação, foi escolhida, pois permite inserção de elementos mantendo ordem específica entre os elementos, também permite a ordenação dos elementos e o percorrimento da lista, busca por elemento específico e retorno de elementos em índices determinados.
- **Conjunto.** Representado pela interface *Set* e com a implementação de *HashSet*. Esta estrutura, com essa implementação, foi escolhida por realizar as operações de inserção, remoção e retorno em tempo constante ($O(1)$) e por utilizar pouca memória.
- **Conjuntos Disjuntos.** Esta estrutura de dados avançada foi desenvolvida para este projeto para realizar as operações de um *Conjunto Disjunto*: *MakeSet*, *Union* e *FindSet*. Com estas operações é possível conectar $n - 1$ nós, cada qual representando uma casa, sem formar ciclos, obtendo, assim, árvores.

3.2. Algoritmos utilizados

O funcionamento do programa para apresentar solução que teste todas as possíveis árvores se baseia na seguinte lógica:

1. Realiza a leitura do arquivo de entrada, armazenando o número de casas, o máximo de conexões permitidas por casa e as arestas representando o custo de conexão de duas casas.
2. Calcula o número de possibilidades de arestas conectadas. Por exemplo, para $n = 5$, teríamos o número de arestas igual a $\frac{n(n-1)}{2} = \frac{5 \cdot 4}{2} = 10$ arestas e o número de possibilidades igual a $2^{10} = 1024$.
3. Um laço testa todas as possibilidades, analisando apenas as que atenderem ao critério de ter apenas $n - 1$ arestas conectadas, possibilidades que podem resultar em uma árvore, e adiciona essas arestas desde que não formem ciclo entre as casas e não haja mais de d conexões nas respectivas casas conectadas.
4. Em seguida compara o custo dessa árvore com o menor custo já obtido para alguma árvore, armazenando o índice dessa árvore apenas se o custo dela for menor do que o custo mais baixo obtido até o momento.
5. Depois de rodar todas as possibilidades, obtém-se a árvore com menor custo possível e armazena-se este resultado em um arquivo de texto.

Já o funcionamento do programa para apresentar solução que teste todas as possíveis árvores em ordem de custo se baseia na seguinte lógica, que utiliza as ideias apresentadas no artigo [Sörensen and Janssens 2005]:

1. Realiza a leitura do arquivo de entrada, armazenando o número de casas, o máximo de conexões permitidas por casa e as arestas representando o custo de conexão de duas casas.
2. Roda o algoritmo *Kruskal* para obter a árvore geradora mínima (MST), verifica se essa árvore atende ao limite do máximo de conexões para cada nó. Caso atenda, já armazena essa árvore em um arquivo e encerra o processamento. Caso contrário, irá para o passo seguinte.
3. Este passo consiste em utilizar a MST encontrada pelo Kruskal para construir novas partições, calcular as MST's das partições e armazenar estas por ordem de custo crescente das MST's em uma lista de partições, desde que seja possível gerar uma árvore com todos os nós conectados a partir dessas partições. Em seguida, checa se a de menor custo atende ao critério do máximo de conexões por casa e armazena essa árvore e encerra o processamento caso atenda, ou remove essa partição de menor custo, gera novas partições com base na MST dela e armazena essas novas partições na lista de partições, mantendo a lista ordenada por custo crescente das MST's das partições e repetindo esse processo até achar uma partição que possua MST que atenda ao critério definido.

Este segundo método de teste de todas as possíveis árvores em ordem de custo crescente utiliza a lógica descrita no artigo [Sörensen and Janssens 2005], adicionando o critério de parada que, neste caso, é obter uma árvore com número especificado para o máximo de conexões por casa.

3.3. Complexidade dos Algoritmos

O algoritmo desenvolvido para apresentar solução que teste todas as possíveis árvores tem sua complexidade assintótica dominada pelo laço que roda todas as combinações possíveis, ou seja, $2^{\frac{n(n-1)}{2}} = 2^{\frac{n^2-n}{2}} = O\left(2^{n^2}\right)$, portanto a complexidade desse algoritmo é $O\left(2^{n^2}\right)$, onde n é o número de casas.

Com base no artigo [Sörensen and Janssens 2005], a complexidade assintótica do algoritmo para apresentar solução que teste todas as possíveis árvores em ordem de custo é $O(N.|E|log.|E| + N^2)$, onde N é número de árvores geradoras do grafo representando todas as conexões das casas, e $|E|$ é o número de arestas desse grafo.

4. Conclusão

Como podemos notar problemas do tipo NP-difíceis são possíveis de serem resolvidos para valores pequenos de n, porém rapidamente atingem um limite computacional onde torna-se impossível calcular todas as possíveis soluções.

A primeira parte do projeto possui tal limitação quando valores de n passam da faixa de 7-8 casas. A segunda parte do projeto visa resolver esse problema calculando as árvores por ordem de custo crescente, desse modo, existindo grande possibilidade prática de se encontrar uma árvore que atenda aos critérios estabelecidos realizando-se uma quantidade de testes significativamente menor do que a quantidade total de testes possíveis, viabilizando, assim, o encontro da solução em tempo hábil.

Referências

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms*. The MIT Press, 3th edition.

Sörensen, K. and Janssens, G. K. (2005). An algorithm to generate all spanning trees of a graph in order of increasing cost. *SciELO*.