Documentação ALG

Estrutura de Dados

A principal estrutura de dados utilizada foram os vectores da biblioteca std do C++. Esta estrutura foi escolhida porque além de ser simples e já estar implementada na biblioteca padrão do C++, permite que seja acessado qualquer elemento com complexidade O(1).

Algoritmo

O algoritmo utilizado foi inspirado no Kruskal's Algorithm:

- Vou criando a árvore geradora mínima inserindo sucessivamente as arestas com menor custo, o que significa que começo do **ponto de interesse "p1"** tal que **p1** tem o menor valor de atratividade, e escolho das **n arestas que chegam a p1** aquela com menor custo e maior atratividade agregada.
- O próximo ponto de interesse "**p2**" será o segundo ponto com menor valor de atratividade.
- Caso exista mais de um ponto com o mesmo valor de interesse, o primeiro na ordem será selecionado. Por exemplo, caso **p2** e **p3** tenham atratividade igual a 40, **p2** será o selecionado. E assim sucessivamente até chegarmos no último ponto.

Pseudo Código

- Achar o ponto de interesse (nós) com menor atratividade.
- Achar trechos (arestas) para o nó com menor atratividade atual.
- Obter desses trechos aquele com menor custo e maior atratividade agregada (em caso de existirem dois trechos com o mesmo custo).
- Repetir esse loop n -1 vezes, onde n é o número de pontos de interesse.

Análise de complexidade assintótica

Temos um while que roda **n - 1** vezes, **onde n é o número de pontos de interesse**, e engloba todo o resto da nossa solução. Dentro desse while são chamadas 3 funções que executam **n -1** vezes: acharPontoDeInteresseComMenorAtratividade, obterTrechosParaPonto e obterTrechoComMenorCustoEMaiorAtratividade.

- *acharPontoDeInteresseComMenorAtratividade* -> executa um for que roda **n vezes**, onde **n é o número de pontos de interesse**.
- *obterTrechosParaPonto* -> executa um for que roda **m vezes**, onde **m é o número de trechos (arestas).**
- *obterTrechoComMenorCustoEMaiorAtratividade* -> executa um for que roda **m vezes**, onde **m é o número de trechos (arestas).**

Também temos uma função que executa apenas 1 vez:

- *acharUltimoPonto* -> executa um for que roda **n vezes**, onde **n é o número de pontos de interesse**.

Ou seja, temos ((n - 1) * (n + 2m)) + n. Quando n e m assumem valores muito grandes, podemos considerar que nossa complexidade fica da seguinte forma: n * (n + m). Onde n é o número de nós do nosso problema e m o número de arestas.
Observação: Para a analise da complexidade assindética a função que imprime o resultado foi desconsiderada.