Análise de algoritmos de ordenação para a disciplina de Projeto e Análise de Algoritmos

Amarildo Campelo Costa Junior1, Victor Emmanuel Vieira Soares2

Departamento de Computação – Universidade Federal do Piauí

¹ajunior283@ufpi.edu.br, ²victoremmanuel@ufpi.edu.br

**Abstract.** This work aims to seek logistics solutions for commercial establishments (supermarkets, stores) seeking an optimization in the number of security cameras, in order to reduce costs, using algorithms through an eleuristic and optimal solution of the Vertex Coverage problem.

**Resumo.** Este trabalho tem o objetivo de monitorar corredores de estabelecimentos comerciais (supermercados, lojas) buscando uma otimização na quantidade de câmeras de monitoramento por meio de uma logística nas suas posições, a fim de reduzir os custos, utilizando algoritmos em busca de uma solução heurística e ótima do problema de Cobertura de Vértices.

# 1. Introdução

Nos dias atuais, a segurança está cada vez mais ameaçada devido ao aumento da criminalidade e falta de medidas preventivas. Devido a isso, faz-se necessário a implementação de recursos que inibem esse tipo de ação e contribuem para uma maior confiabilidade para os proprietários de estabelecimentos comerciais e seus clientes. Pensando nesses fatores, uma forma de reduzir essas ações seria facilitar o acesso de medidas de segurança a quem gerencia estabelecimentos, por exemplo, adicionar câmeras em locais estratégicos de modo a ter o controle de eventos que podem ocorrer.

Nesse viés, além da adição de câmeras, este trabalho tem como objetivo principal encontrar uma maneira de dispor a quantidade mínima de câmeras que possam cobrir todos os corredores do local a ser estudado buscando a redução de custos. Em um problema como este podem ser aplicados conhecimentos matemáticos como os de teoria dos grafos. Através disso, é possível representar os corredores do estabelecimento como arestas de um grafo e os possíveis locais das câmeras de segurança como vértices. Entretanto, esse tipo de problema é classificado como NP-Completo, o que consiste que para encontrar uma solução ótima é necessário um tempo não-hábil, nesse caso em específico se classifica como um problema de Cobertura de Vértices Mínima.

O problema de vértice mínimo visa buscar a menor quantidade de vértices necessários para que todas as arestas do grafo estejam cobertas, ou seja, o objetivo é encontrar um conjunto de vértices de modo que cada aresta é incidente a pelo menos um vértice do conjunto. Uma cobertura de um grafo G é mínima se não existe outra menor. Em outras palavras, uma cobertura F de G é mínima se não existe cobertura H de G tal que |H| < |F|. Percebe-se que o problema é explicitamente custoso. Um algoritmo de força bruta, que verifica todos os subconjuntos do conjunto de vértices do grafo, soluciona o problema. Entretanto não foi descoberto um algoritmo capaz de solucionar em tempo hábil. Trazendo essa noção para este trabalho, uma aresta do grafo estar coberta significa que uma câmera está tendo visão de um corredor.

Uma imagem contendo Gráfico

Descrição gerada automaticamente Dado o grafo acima, temos como vértices que compõem o conjunto que satisfaz o problema de cobertura mínima os vértices: 2, 3, 4, assim como 1, 2, 4. Nestes casos, o grafo estaria coberto porque os vértices contidos no subconjunto que soluciona o problema estão incididos por todas as arestas do grafo.

# 2. Implementação de uma instância do problema

Para a implementar uma instância do problema foi utilizada a linguagem Python e seus recursos de orientação a objetos para ser possível representar os componentes do problema (localização das câmeras, corredores), de uma forma que o usuário possa criar e modificar os parâmetros da entrada dentro da aplicação.

A partir do momento em que as partes necessárias para desenvolver a solução do problema foram definidas, elas devem ser representadas no algoritmo na forma de uma estrutura que seja possível manipular para chegar no objetivo final. Neste caso, utilizamos uma classe Grafo que vai representar o grafo correspondente ao estabelecimento comercial, além de uma classe Vértice que está relacionada a um possível local onde a câmera será colocada para observar os corredores, representados por uma classe Aresta.

A classe Vértice foi implementada com um construtor, recebendo como parâmetro um id para identificação do vértice no grafo e possui métodos para acessar e modificar seus atributos, como o getId() que retorna o id do objeto vértice. Não é possível criar vértices com o mesmo id, uma vez que para a instância do problema não faria sentido ter mais de uma câmera no mesmo local, pois o objetivo é encontrar o menor número de câmeras que consiga cobrir a área total do estabelecimento, ou seja, cobrir todos os corredores.

De modo similar, a classe Aresta foi implementada também possuindo um construtor, tendo como parâmetro o vértice de origem, o vértice destino, que representam os vértices ligados, mesmo que o grafo não seja direcionado, além do peso da aresta, apesar que para a maioria dos casos que serão tratados isso será desconsiderado ou seja, peso = 0. Ademais, a classe possui métodos, como o getOrigem() que retorna o vértice de origem dessa aresta. Não é possível criar arestas que incidem no mesmo vértice, pois no contexto da aplicação não faz sentido representar duas vezes o mesmo corredor.

Por fim, a classe Grafo foi implementada a partir de um construtor que recebe como parâmetro se o grafo é direcionado e cria atributos como uma lista de vértices e uma lista de arestas. Para preencher e interagir com o grafo é necessário utilizar os métodos criados, como o novo\_Vertice() que recebe o identificador do vértice e o adiciona na lista de vértices ou o nova\_Aresta() que recebe origem, destino e peso da aresta a ser adicionada, além de métodos para buscar um vértice ou aresta ou até remover. Com essa classe implementada o estabelecimento é representado de uma forma simples e prática para que se possa manipular e chegar ao resultado requerido para o problema.

# 3. Implementação do método de força bruta

Embora o problema de Cobertura mínima de vértices seja considerado como NP-Completo, é possível encontrar uma solução ótima para uma instância do problema utilizando um algoritmo de força bruta, mesmo que este não retorne a resposta em um tempo hábil.

Primeiramente, é disposto um menu para a interação do usuário com os métodos disponíveis, onde a cada ação que ele deseja fazer existe uma opção com número correspondente. Exemplo: Para adicionar um possível ponto de localização para uma câmera, basta digitar 1 e inserir o número correspondente à localização, uma vez que esta é uma abstração para o vértice do grafo.

Para a parte do método que encontra a cobertura mínima de vértices é utilizado um método baseado em força bruta. Em outras palavras, o algoritmo considera e verifica todas as possibilidades a fim de encontrar aquelas que gerem uma solução ótima. O método é implementado dentro da classe Grafo e executa outros métodos, além de criar variáveis para chegar na resposta final.

O método bruteForce(self) recebe como parâmetro apenas o objeto que o chama, sendo esta uma exigência da linguagem de programação escolhida (Python). Inicialmente, as listas de vértices e arestas são guardadas nas variáveis vertices e edges, respectivamente, para facilitar o acesso a esses valores quando necessário. Guarda-se também a quantidade de vértices em uma variável vertices\_quantity, que será útil logo depois para o laço de repetição. É criada uma lista res, incialmente vazia, responsável por guardar os subconjuntos que compõem a solução final. Ademais, cria-se a variável min\_vertices\_lenght que armazena a quantidade mínima de vértices para cobrir todas as arestas do grafo e inicialmente recebe o valor correspondente à quantidade de vértices. A partir disso, é feito um laço de repetição que começa em 1 e vai até a quantidade de vértices do grafo, realizando duas operações: Gera todos os subconjuntos de tamanho i (valor correspondente à iteração do laço) e percorre em cada subconjunto gerado com o intuito de verificar se é uma solução para o problema, ou seja, se os vértices desse subconjunto cobrem todas as arestas do grafo.

Para a primeira parte do laço, onde o objetivo é gerar todos os subconjuntos de tamanho i, utiliza-se o método gen\_subsets (self, set\_, k), que recebe como parâmetros o próprio objeto, a lista de vértices do grafo e k, sendo este o tamanho dos subconjuntos a serem criados. Porém, gen\_subsets() é apenas o método responsável por chamar o principal, que será utilizado para gerar realmente os subconjuntos e chamado recursivamente para gerar todas as combinações. O método principal é generate\_subsets() que recebe como parâmetro o próprio objeto que o chamou(self), um subconjunto atual(curr\_subset), todos os subconjuntos formados até aquela parte da execução(subsets\_), o tamanho dos subconjuntos a serem formados(k) e também o próximo índice(next\_index). Então, a função é definida em duas condições: A primeira é para verificar se o tamanho do subconjunto atual, referente ao momento da execução, é igual ao tamanho do subconjunto recebido como parâmetro, caso positivo este subconjunto é adicionado à lista de subconjuntos formados e a função utiliza um return para terminar sua execução; A segunda condição verifica se o próximo índice somado a 1 é menor ou igual ao conjunto principal, formado por todos os vértices do grafo. Dentro dessa condição o valor do subconjunto atual é salvo numa variável curr\_subset\_exclude para que seja possível resultar em todas as possibilidades de combinações com os vértices. Após isso, o próximo vértice do conjunto é adicionado ao subconjunto atual e então são feitas duas chamadas recursivas à generate\_subsets(), mas passando como parâmetros curr\_subset e curr\_subset\_exclude que equivalem ao subconjunto atual e o mesmo no estado anterior à adição do próximo vértice, respectivamente. Por fim, é feita a ligação com gen\_subsets() e são retornados todos os subconjuntos formados por uma combinação dos n vértices em grupos de k elementos.

Considerando a parte do laço do BruteForce() responsável por percorrer cada subconjunto formado e analisar se as condições necessárias para ser uma solução do problema são atendidas, é feita através de um foreach dentro da lista de subconjuntos formados e a partir de cada subconjunto de vértices analisar se cobrem as arestas do grafo. No laço que passa por cada subconjunto, há duas condições a serem satisfeitas para que o subconjunto seja adicionado como solução. A primeira corresponde ao valor de retorno da função verify\_vertex\_cover, que recebe como parâmetros o subconjunto e a lista de arestas e tem como objetivo verificar se o subconjunto de vértices cobre todas as arestas do grafo, retornando o valor booleano True. Caso alguma aresta não seja coberta por um dos vértices do subconjunto, a função retorna False. Sendo o retorno da função citada igual a True, então a segunda parte deve resultar em True para que o subconjunto possa ser adicionado como solução. Esta condição consiste em verificar se a quantidade de vértices que formam o subconjunto é igual ou menor à quantidade mínima de vértices para a cobertura do grafo, representado pela variável min\_vertices\_lenght, caso seja atendida, o valor min\_vertices\_lenght é atualizado para o tamanho do subconjunto e este é adicionado a res como parte da resposta do problema para o grafo dado.

# 4. Complexidade

Devido ao problema de Cobertura mínima ser NP-Completo seu algoritmo para encontrar a solução ótima, tornando assim o algoritmo inviável de se utilizar a partir de um certo tamanho da entrada.

Portanto o algoritmo de força bruta que soluciona o problema de Cobertura mínima possui complexidade O(2^n), que provém da quantidade de vértices para cada tamanho possível de subconjuntos. Além disso, ele verifica se o subconjunto de vértices atualmente selecionado é uma cobertura de vértices válida, iterando sobre todas as arestas e verificando se existe alguma aresta que não esteja coberta pelos vértices no subconjunto atualmente selecionado