Trabalho Prático 3

Documentação

Algoritmos I - DCC206

Luiz Gustavo Almeida de Oliveira - 2019083781

1 Descrição do Problema

Durante o processo global de vacinação contra a COVID-19, um dos maiores desafios que se mostrou foi relacionado à logística. Algumas das vacinas, que foram desenvolvidas em tempo recorde, tem requisitos extremamente específicos para o armazenamento e preservação da qualidade das mesmas, o que dificulta a distribuição para áreas mais remotas que não tem a infraestrutura necessária.

Tendo isso em vista, a secretaria de saúde de um estado brasileiro que contém uma grande área florestal, composto também por pequenas vilas, precisa construir depósitos de vacinas para as suas vilas, tal que se duas vilas estão ligadas, ao menos uma delas contém um depósito.

O problema foi tratado de duas maneiras, primeiro considerando um mapa de vilas que não tem ciclos, ou seja, não é possível retornar à uma vila por um caminho diferente do que usar para sair, e segundo considerando que podem exister ciclos nos caminhos entre as vilas.

Por fim, foi feita uma modelagem computacional do problema fazendo uso de um grafo para representar as vilas e suas conexões, e o problema foi resolvido usando paradigmas de programação dinâmica no caso em que é possível sabermos a solução exata e um algoritmo guloso no caso em que é encontrada uma solução aproximada.

2 Execução do Código

2.1 Compilação e Execução

Pré-requisitos:

- Compilador moderno de C++(e.g: g++ 10.x.x)
- Programa *GNU Make* instalado (altamente recomendado)

Na pasta raiz do projeto, onde se localiza o *Makefile*, execute o comando *make* e então um arquivo chamado ***tp3*** será gerado no diretório */bin*, que também está na raiz do projeto.

```
$ make
$ cd bin
$ ./tp3 tarefa1 dados.txt
```

3 Modelagem Computacional

O problema proposto pode ser enxergado da seguinte forma: Temos um conjunto de nós e arestas, e precisamos selecionar um subconjunto mínimo de nós tal que todo nó que não está no conjunto é adjacente à um deles, sendo que nas instâncias da *tarefa1* tratamos grafos acíclicos e nas instâncias da *tarefa2* tratamos grafos que podem conter ciclos.

3.1 tarefa1

O problema da mínima cobertura de vértices é um clássico problema NP-Completo, ou seja, não pode ser resolvido em tempo polinomial se $P \neq NP$. Ainda sim, embora seja NP-Completo, existem algumas instâncias do problema que podem ser resolvidas em tempo polinomial, tais como as da *tarefa1*, que podem ser enxergadas como árvores. Nesse caso, sabendo que teremos $n-1$ nós e n arestas.

Existem algumas abordagens conhecidas para solução da *tarefa1* em tempo polinomial e de maneira exata, como a que utiliza a equivalência desse problema com o problema da máxima correspondência em grafos bipartidos, descrita pelo teorema de König. Todavia, implementei uma solução baseada em programação dinâmica que faz uso de propriedades da árvore e memoização para chegar no resultado em tempo polinomial.

A abordagem utilizada na *tarefa1*, consiste, então, de uma função recursiva que testa se seria melhor incluir o nó atual na solução ou não, utilizando de duas estratégias cruciais para o algoritmo:

- Se um nó não está na solução, todos os nós conectados à ele precisariam estar, ou então os "caminhos"não estariam cobertos. Ou seja, se dois nós estão conectados, ao menos um deles precisa estar na solução.
- Se um nó já está na solução, não necessariamente os nós conectados à ele estarão também, embora possa ser benéfico. Neste caso, utilizaremos a estratégia com memoização para verificar se é melhor incluir um certo nó ou não.

Tendo isso em vista, o algoritmo seleciona o valor ótimo(mínimo) checando se é melhor incluir ou não cada um dos nós na solução. Foi utilizada uma lista de adjacência para representar o grafo, e um vetor auxiliar para representar o pai de cada um dos nós, o que nos permite saber quais nós já estão possivelmente cobertos por um outro que está na solução ou não em tempo constante.

O código foi organizado de maneira que a execução da tarefa1 é baseada na chamada para o método *coberturaMinimaDeClinicasEmGrafoAciclico*, que receberá um nó para usar como raiz e se ele está incluso na solução ou não, nesse caso então começamos com um nó arbitrário e selecionamos o mínimo entre incluí-lo ou não na solução, e recursivamente verificamos para o resto do grafo quais nós devem ser incluídos.

```
tarefa1 = min(coberturaMinimaDeClinicasEmGrafoAciclico(raiz, incluirNaSolucao),
              coberturaMinimaDeClinicasEmGrafoAciclico(raiz, naoIncluirNaSolucao))
```

Para o pseudo-código à seguir, vamos considerar a existência de uma lista de adjacência **grafoAdj**, representando o nosso grafo, de um outro array **pai** de tamanho n que servirá para ter as relações de "pai e filho"entre os nós, e de um outro array **memo** de tamanho *[n][2]*, inicializado com -1, que é o que é usado para memoização e salvará os valores calculados.

Algorithm 1

```
coberturaMinimaDeClinicasEmGrafoAciclico (indice, naSolucao):
    Se grafoAdj[indice].tamanho == 0 ⟶ retorne naSolucao; // sem arestas
    Se memo[indice][naSolucao] != -1 ⟶ retorne memo[indice][naSolucao]; // ja calculado
    soma = 0;
    Para cada nó em grafoAdj[indice]:
        Se o nó atual não for igual à pai[indice]:
          pai[noAtual] = indice
          Se naSolucao == 0:
            // precisa estar na solucao
            soma += coberturaMinimaDeClinicasEmGrafoAciclico(noAtual, 1);
          Senão
            // checando se é melhor incluir ou não
            soma += min(coberturaMinimaDeClinicasEmGrafoAciclico(noAtual, 1),
                        coberturaMinimaDeClinicasEmGrafoAciclico(noAtual, 0);
    memo[indice][naSolucao] ⟵ soma + naSolucao;
    retorne memo[indice][naSolucao];
```

Novamente, concluindo, para *tarefa1*, começamos testando se é melhor incluir o nó raiz na solução ou não e, então, recursivamente, fazemos o mesmo para os outros nós da árvore, armazenando resultados já computados, para chegar na solução ótima, que é o menor conjunto de nós que cobre todos os vértices.

3.2 tarefa2

Para a tarefa dois, considero o mesmo problema da *tarefa1*, uma mínima cobertura de vértices, entretanto, aqui tenho instâncias do problema que contém ciclos no grafo. Se fosse possível resolver uma desses instâncias que contém ciclos de maneira exata e com um algoritmo com complexidade assintótica temporal polinomial, estaria dizendo que $P = NP$.

Aqui, optei por utilizar uma estratégia gulosa, que navega pelas arestas e caso encontre dois nós conectados que não estão cobertos, adiciona ambos na solução e remove os nós adjacentes à ambos, já que agora todos estão cobertos por esses que acabamos de adicionar.

Além disso, seguindo essa estratégia, é possível notar que o conjunto de arestas selecionadas pelo algoritmo é um conjunto disjunto de arestas, pois retiramos todos os nós adjacentes aqueles adicionados à solução e, portanto, as arestas restantes não poderão ter um nó em comum com aquelas que já estão na solução(esse possível nó foi removido pois já está coberto). Não somente isso, esse conjunto disjunto de arestas também é um emparelhamento maximal. Então podemos descrever o algoritmo de duas formas:

Algorithm 2

```
tarefa2 (G):
conjuntoSolucao = {};
while Ainda houver arestas em G do
        Selecione qualquer aresta {x, y};
        conjuntoSolucao ← {x, y};
        Remova do grafo todas as arestas incidentes em x ou y;
end while
retorne conjuntoSolucao;
```

ou

Algorithm 3

tarefa2 (G):
    Encontre um emparelhamento maximal A em G;
    retorne conjunto de nós conectados pelas arestas em A;

Evidentemente, como a versão de decisão do problema tratado é NP-Completo, também não existe solução com tempo polinomial para a versão de decisão do problema para encontrar um emparelhamento maximal mínimo, ou seja, o emparelhamento maximal com o menor número de nós possível.

Para a modelagem computacional desse algoritmo de forma eficiente, foi necessário criar um novo vetor, com tamanho igual ao número de nós, para permitir a "retirada"das arestas do grafo, pois caso a retirada das arestas fosse ser feita de maneira direta na lista de adjacência, a operação poderia se tornar custosa pela necessidade buscar e deletar os nós e todos aqueles adjacentes à ele. O vetor auxiliar funciona com booleanos e permite verificar em tempo constante se um nó já está na solução ou se é adjacente à um nó que já está na solução e, portanto, está coberto.

4 Análise de Complexidade Assintótica

4.1 tarefa1

A complexidade assintótica temporal da *tarefa1* é relativamente trivial, pois iteramos nas arestas do grafo, utilizando a lista de adjacência, e verificamos se o nó atual deveria ser incluído ou não, de maneira recursiva, sendo que ainda sim estamos limitados no número de arestas do grafo, pois utilizando memoização foi possível verificar antes de começar a iterar na lista de adjacência de um nó se o valor adequado para ele já foi calculado, ou seja, evitando repetição e, portanto, permitindo alcançar a complexidader assintótica temporal $O(|E|)$, onde E é o conjunto de arestas no grafo.

A complexidade assintótica espacial é proporcional tanto ao número de nós quanto ao de arestas, podendo ser dominada diferentemente dependendo da quantidade de nós e do quão conectado o grafo da entrada é. Conforme mencionado anteriormente, é necessária uma lista de adjacência para armazenar as arestas e representar o grafo, proporcional em tamanho com $V+E$. Além disso, dois vetores auxiliares, proporcionais ao número de nós V do grafo, são necessários para a memoização e para guardar a informação de relação pai-filho entre os nós. Portanto, a complexidade assintótica espacial é $O(3V + E)$.

4.2 tarefa2

Na *tarefa2*, existiu um desafio na modelagem eficiente devido ao fato do algoritmo implementado precisar "retirar"as arestas do grafo que está sendo avaliado. Pensando nisso, optei por aumentar o gasto com espaço mas evitar o drástico curso de busca e remoção de arestas no grafo, sendo assim, foi necessário criar um vetor auxiliar proporcional ao tamanho do número de nós, que é usado para informar se um certo nó já está "coberto"por um de seus adjacentes estar no conjunto solução ou não, sem ser necessário alterá-lo de fato na instância do grafo e permitindo consulta em tempo constante. Além disso, obviamente, é necessário armazenar o grafo na lista de adjacência e, portanto, alcançamos complexidade assintótica espacial $O(2V + E)$.

A complexidade assintótica temporal da execução dessa tarefa também é facilmente avaliada, pois o que é feito é uma iteração pelo nós e pelas suas arestas, e caso essa aresta venha a

ser incluída na solução, executamos operações com custo constante. Portanto, a complexidade assintótica temporal é $O(V + E)$.

5 Prova de Corretude do Algoritmo

5.1 tarefa1

5.1.1 O algoritmo dá como resultado uma mínima cobertura de vértices

Considerando S o conjunto solução dado pelo algoritmo proposto, vamos supor que existe S', um conjunto solução com cardinalidade menor que S. Para isso, S' precisa ter encontrado um conjunto que cobre todas as arestas com menos nós, porém na montagem de S o algoritmo seleciona sempre o mínimo entre as opções disponíveis, então se S' tem cardinalidade menor que S, ao menos uma aresta não está sendo coberta, devido ao nó que foi removido, o que é uma contradição.

5.2 tarefa2

5.2.1 O algoritmo dá como resultado uma cobertura de vértices

Sabendo que buscamos um emparelhamento maximal A no grafo G, é evidente que todas as arestas em A estão cobertas, e se existe uma aresta $b \notin A$ que não está sendo coberta, então $A \cup \{b\}$ é um emparelhamento, o que quebraria a maximalidade de A, uma contradição.

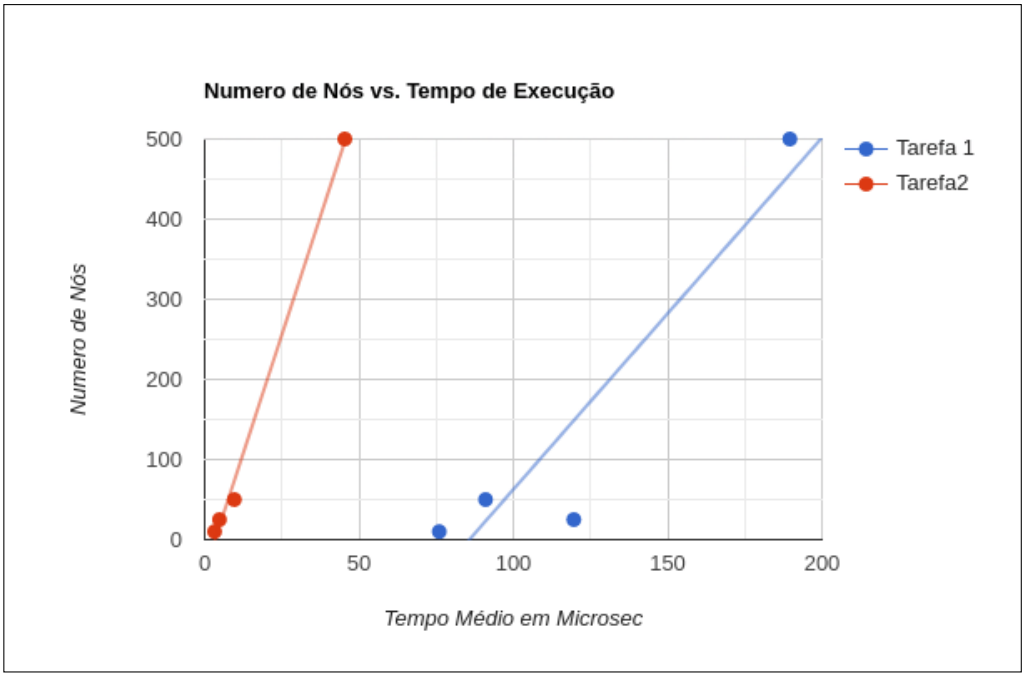
5.2.2 Prova que o algoritmo dá um resultado no máximo 2x igual ao ótimo

Sabendo que o conjunto solução tem que cobrir toda aresta no grafo G, ou seja, ele precisa conter *ao menos* um dos nós de cada aresta, tal que essas arestas não tem nós em comum em G. Assim, é possível afirmar que, no mínimo, o conjunto mínimo para cobertura dos vértices tem tamanho $|A|$, sendo que A é o conjunto do emparelhamento maximal. Porém, o algoritmo utilizado adiciona gulosamente os dois nós de cada aresta encontrada no conjunto solução, ou seja, ele resulta em um conjunto de tamanho até $2|A|$.

6 Avaliação Experimental

	Número do Teste	Tempo Médio μs	Variância $(\mu s)^2$	Nós em G	Arestas em G
Tarefa 1	0	76	515.5	10	9
Tarefa 2	0	3.2	0.2	10	9
Tarefa 1	1	119.6	1339.8	25	24
Tarefa 2	1	4.8	1.7	25	24
Tarefa 1	2	91	1174.5	50	49
Tarefa 2	2	9.6	30.8	50	49
Tarefa 1	3	189.6	1008.3	500	499
Tarefa 2	3	45.4	68.8	500	499

Na tabela acima, a *tarefa1* e a *tarefa2* foram testadas utilizando grafos acíclicos para avaliar o crescimento do tempo conforme o aumento do número de nós e arestas no grafo. É possível notar que a *tarefa1*, que encontra sempre a solução exata nessas instâncias do problema, teve tempo de execução consideravelmente maior que a *tarefa2*, que encontra soluções aproximadas. É importante notar também que a variância na *tarefa1* foi bastante alta.



No gráfico acima é possível notar que a *tarefa2* teve um crescimento mais próximo de linear do que a *tarefa1*, que, novamente, apresenta uma variância bem maior. Alguns fatores podem estar influenciando a execução da primeira tarefa, como sua implementação recursiva, que faz com que diversas chamadas sejam feitas à pilha e pode estar afetando o seu tempo de execução, o que contrasta fortemente com a implementação da *tarefa2*, que consiste em dois loops iterando pelos nós e arestas e poucas operações com custo constante.

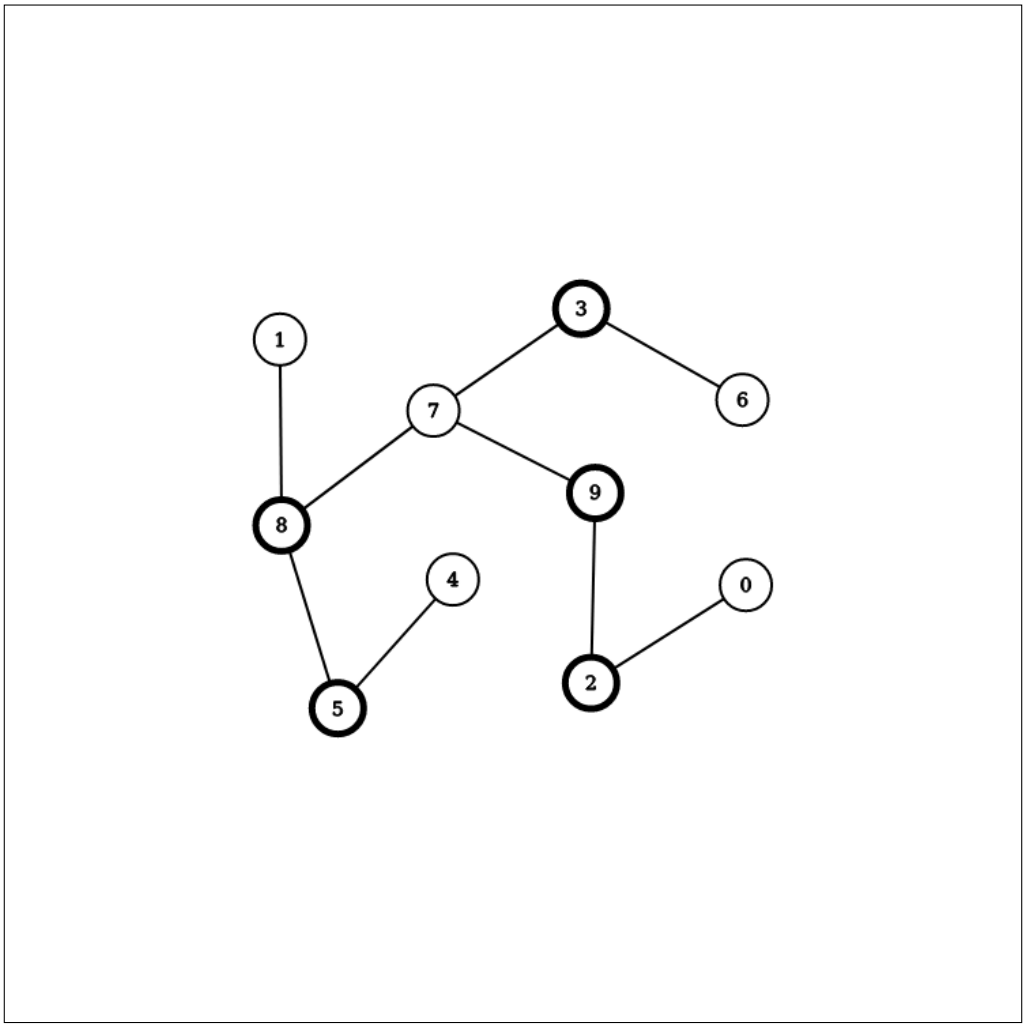


Figura 1: Grafo de exemplo com 10 nós e solução

Na figura 1, é possível visualizar um grafo sem ciclos, onde os nós circulados em destaque são o conjunto C, o conjunto solução com menor cardinalidade para a cobertura de vértices. Nesse caso, o algoritmo utilizado na *tarefa2*, que funciona com aproximação do resultado, seleciona todos os nós como resultado da solução, devido às limitações lógicas do algoritmo utilizado.

É possível concluir que, embora o algoritmo um traga soluções exatas, ele está restrito às instâncias específicas do problema da mínima cobertura de vértices. Além disso, devido a forma como foi implementado, um possível overhead de memórai foi gerado pela recursividade, o que prejudicou seu desempenho. Tratando do algoritmo utilizado na tarefa2, ele se mostrou bastante eficiente pela sua simples implementação porém o seu fator 2 de aproximação não é tão forte e pode dar resultados pouco satisfatórios em muitos casos, entretando esse é um limite conhecido na computação para o problema e altamente não trivial de ser superado.

Por fim, neste trabalho prático foi possível aplicar diversos conceitos aprendidos nos últimos módulos da disciplina, como os relacionados à programação dinâmica, algoritmos gulosos, NP-Completude, estruturas de dados e análise de complexidade assintótica do que foi implementado.