Algoritmos e Estruturas de Dados III
Trabalho Prático 02 - Festa na Piscina

Lucas Pereira Monteiro[1]

[1]Departamento de Ciência da Computação – DCC
Universidade Federal de Minas Gerais – UFMG

lucasmonteiro@dcc.ufmg.br

Resumo. *O objetivo deste trabalho é a resolução de um problema relacionado à Cobertura de Vértices*[1]*. Para tal, o aluno deve propor uma solução heurística*[2] *e uma solução ótima para o mesmo.*

1. Introdução

Zambis quer dar uma festa para inaugurar sua nova piscina e quer chamar todos os seus amigos. Porém ele não quer perder tempo chamando cada um deles. Para resolver isso, ele sabe que existem pessoas de sua rede de amizade que se conhecem. Então ele quer pedir para o menor número de amigos chamarem quem eles conhecem até que todos os amigos do Zambis sejam convidados. Vale lembrar que os amigos do Zambis só irão chamar quem eles conhecem. Quantas pessoas o Zambis vai precisar chamar para que todos seus amigos sejam convidados?

1.1. Modelagem

O problema supracitado pode ser modelado em um grafo *G* da seguinte forma:

Seja *G(V,A)* o grafo que representa as amizades entre Zambis e seus amigos.

Seja *V* os vértices do grafo, os quais serão representados pelos amigos de Zambis.

Seja *A* as arestas do grafo, representando as relações de amizades (conexão) entre os vértices.

1.2. Descrição do problema

A partir da modelagem em grafos, nos deparamos com um problema muito difícil de resolver. Precisamos achar o menor número de amigos (vértices) que consigam chamar (conectar) todos os outros amigos. Esse problema é conhecido em Ciência da Computação como Cobertura de Vértices. A única forma de se saber qual o menor número exato de amigos que conectam todos os outros é testando todas as possibilidades possíveis. Exemplo:

Seja o conjunto $C = \{1, 2, 3\}$, quais são todas as possibilidades de combinação entre esses elementos?

$\{\emptyset\}, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\} \longrightarrow 2^n$

Percebemos que é necessário testar todas as possibilidades de combinação para achar a cobertura de vértices mínima (menor número de vértices que cobre um grafo). Pela resolução desse problema ser exponencial, sabemos que o problema da Cobertura de Vértices é NP-Completo[3].

[1]http://en.wikipedia.org/wiki/Vertex_cover
[2]http://en.wikipedia.org/wiki/Heuristic_(computer_science)
[3]http://en.wikipedia.org/wiki/NP-complete

Dado que o problema é NP-Completo (sua solução tem um custo computacional altíssimo), podemos propor uma heurística para achar uma cobertura de vértices mínima próxima à ótima. Nas duas próximas seções, as soluções ótima e heurística são explicadas detalhadamente.

2. Implementação Heurística

Para a implementação da solução heurística, foi escolhida uma implementação de grafo por meio de Listas de Adjacência. Usando essa implementação, o tempo de execução é $O(V + A)$, onde V é o número de vértices e A é o número de arestas.

2.1. Estruturas e tipos utilizados

```
1  typedef struct TipoItem {
2      int Vertice; // Identificacao do amigo (vertice)
3  } TipoItem;
4
5  typedef struct TipoCelula *TipoApontador;
6
7  typedef struct TipoCelula {
8      TipoItem Item;       // Item que vai conter um vertice
9      TipoApontador Prox; // Apontador para um proximo elemento
10 } TipoCelula;
11
12 typedef struct TipoLista {
13     TipoApontador Primeiro; // Aponta para o primeiro elem. da lista
14     TipoApontador Ultimo;   // Aponta para o ultimo elem. da lista
15     int flagAmigoUnico;     // Indica se o amigo e amigo so de Zambis
16 } TipoLista;
17
18 typedef struct TipoGrafo {
19     int numAmigos;        // Quantidade de amigos que Zambi possui
20     int Inicio;           // Posicao inicial da lista de adjacencia
21     int Fim;              // Ultima posicao da lista de adjacencia
22     TipoLista *ListaAdj; // Representa as relacoes de amizade
23 } TipoGrafo;
```

Estruturas_e_Tipos

2.2. Resolução do problema

A solução do problema é baseada na seguinte idéia: escolha dois vértices *(u,v)* conectados no grafo, coloque-os no menor caminho e exclua todos os vertices com arestas adjacentes a u e v. Para esse processo, como já dito acima, será necessário percorrer todos os vértices e todas as suas arestas, resultando em complexidade de tempo de $O(V + A)$.

Algoritmo 1: Heurística para Cobertura Mínima de Vértices

Entrada: Grafo contendo todos os amigos de Zambis
Resultado: Número mínimo de amigos que Zambis precisa chamar para que eles chamem todos os outros amigos ainda não chamados

```
início
    C = ∅
    enquanto Grafo ainda tiver vértices faça
        Escolha duas arestas u, v
        C ⟵ C ∪ (u, v)
        enquanto Aresta u ainda tiver vértices faça
            Remova todos os vértices adjacentes a aresta u
        enquanto Aresta v ainda tiver vértices faça
            Remova todos os vértices adjacentes a aresta v
    retorna C
```

2.3. Prova que a solução é um algoritmo aproximativo

O conjunto de vértices retornado por C é uma cobertura de vértices, pois o algoritmo entra em loop até que não haja mais nenhum vértice no grafo de amigos de Zambis.

Seja A o conjunto de todas as arestas pertencente aos vértices (u, v). Para cobrir as arestas em A qualquer que seja uma cobertura de vértices - em particular uma cobertura ótima C^* - deve incluir pelo menos uma extremidade de cada aresta em A. Duas arestas em A não compartilham uma extremidade, pois todas as arestas que são incidentes em suas extremidades são removidas, uma vez que são adjacentes. Logo, não há duas arestas em A cobertas pelo mesmo vértice de C^*, assim temos o limite inferior: $|C^*| \geq |A|$ sobre uma cobertura de vértices ótima. Cada escolha das arestas (u, v) escolhe uma aresta para a qual nenhuma das suas extremidades já está em C, produzindo um limite superior sobre o tamanho da cobertura de vértices retornada: $|C| \geq 2|A|$. Combinando as equações acima, obtemos que $|C| \leq 2|C^*|$

2.4. Funções principais e análise de complexidade temporal e espacial

FGrafoVazio() - grafo_heuristica.c: Seja V o número de vértices (amigos) que Zambis possui. Logo a complexidade temporal e espacial da função é $O(V)$, pois ela possui um loop que percorre todos os vértices e aloca uma lista vazia para cada um deles.

SetaParametros() - grafo_heuristica.c: Em $O(1)$ define os parâmetros do grafo e faz o uso da função FGrafoVazio(). Logo a complexidade é $O(max(1 \ldots 1, V))$, onde V é o número de vértices do grafo, resultando em complexidade espacial e temporal de $O(V)$.

SalvaAmizades() - grafo_heuristica.c: Seja N o número de amigos (vértices adjacentes) de um vértice qualquer. Essa função tem complexidade $O(N)$, pois salva todos os vértices lidos em uma linha na lista de adjacentes do primeiro vértice lido.

Heuristica() - grafo_heuristica.c: É a função principal do programa e já teve seu pseudo-código descrito acima. Possui complexidade de tempo igual a $O(V + A)$, onde V é o número de vértices e A o número de arestas do grafo.

2.5. Função *main_h*

É a principal função do programa que faz a análise heurística. Seja I o número de instâncias que são necessárias para a execução do programa. A maior complexidade dentro do loop de instâncias é a função *Heuristica()*, a qual possui complexidade $O(V + A)$. Portanto, a complexidade dessa função é $O(I(V + A))$.

3. Implementação Ótima

Para a implementação da solução ótima foi escolhida uma implementação de grafo por meio de Matriz de Adjacência. A escolha pela matriz se dá pelo fato que é necessário representar todos os conjuntos possíveis. Exemplo:

> Seja o conjunto $C = \{1, 2, 3\}$, quais são todas as possibilidades de combinação entre esses elementos?
> $\{\emptyset\}, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\} \longrightarrow 2^n$

Se a solução ótima tivesse sido implementada por meio de listas de adjacência, o programa ficaria muito lento e custoso, visto que seria necessário acessar todos os elementos da lista várias vezes, o que não acontece com a matriz de adjacência, uma vez que acessamos os elementos em $O(1)$.

3.1. Estruturas e tipos utilizados

```
1  /*
2   * Definicicao do tipo da matriz de adjacencia que representa o grafo.
3   */
4  typedef short **TipoMatrizAdj;
5
6  /*
7   * TipoGrafo que e usado para a definicao do grafo que vai ser
        representado por lista de adjacencia e
8   * usado para a solucao otima.
9   */
10 typedef struct TipoGrafoO {
11     TipoMatrizAdj mAdj;
12     short* vetorVertices;
13     int inicio;     // Inicio do grafo, que sera a posicao 1
14     int fim;        // Fim do grafo, que sera a posicao[tamanho do grafo
           + 1]
15     int numAmigos; // Numero de amigos (vertices)
16 } TipoGrafoO;
```

Estruturas_e_Tipos2

3.2. Resolução do problema

A solução do problema é baseada na seguinte idéia: gere um conjunto C com todas as possibilidades de combinação entre os elementos do grafo G. Após gerado o conjunto, faça uma interação, indo para cada subconjunto $c \subset C$ e verifique se ele cobre todos os vértices de G. Ao final, pegue o subconjunto c com menor número de elementos que conecte todos os vértices e retorne.

Algoritmo 2: Solução Ótima para Cobertura Mínima de Vértices

Entrada: Grafo contendo todos os amigos de Zambis
Resultado: Número mínimo de amigos que Zambis precisa chamar para que eles chamem todos os outros amigos ainda não chamados

início
- Seja V o número de vértices do grafo
- $C \longleftarrow$ Todos os subconjuntos gerados pela combinação dos elementos de V
- $minimo$ = INFINITO
- **enquanto** *Não tiver percorrido todos os subconjuntos de* C **faça**
  - $c \longleftarrow$ primeiro subconjunto de C
  - Remova o primeiro subconjunto de C
  - **se** c *cobre todos os vértices* V **então**
    - **se** *Número de elementos de* c *é menor que* $minimo$ **então**
      - $minimo$ = numero de elementos de c
- **retorna** *minimo*

3.3. Funções principais e análise de complexidade temporal e espacial

CriaMatrizAdjQuadrada() - grafo_otimo.c: Essa função cria uma matriz quadrada, com o número de (vértices + 1) do grafo e inicializa-os com 0. Portanto, sua complexidade temporal e espacial é $O((V+1)^2)$ onde V é o número de vértices do grafo.

SalvaAmizadesO() - grafo_otimo.c: Cria as conexões entre os grafos. Essa função é executada para cada linha do arquivo de entrada, portanto, sua complexidade é temporal e espacial é $O(N)$, onde N representa o número de vértices que se encontram em uma linha.

CriaVetorVertices() - grafo_otimo.c: Essa função tem complexidade temporal $O(V)$, onde V é o número de vértices, uma vez que ela inicializa todas as posições do vetor com 0.

VerificaCobertura() - grafo_otimo.c: Verifica o vetor de vértices. Portanto possui complexidade $O(V)$, onde V é o número de vértices do grafo.

DescobreMenorCobertura() - grafo_otimo.c: Faz N testes, onde N é o número de vértices de um subconjunto. Além desses testes, ela realiza uma verificação com custo $O(V)$, onde V é o número de vértices do grafo. Portanto, sua complexidade é $O(N * V)$.

3.4. Função *main_h*

É a principal função do programa que faz a análise ótima. Seja I o número de instâncias que são necessárias para a execução do programa. A maior complexidade dentro do loop de instâncias é a função *DescobreMenorCobertura()*, a qual possui complexidade $O(N * V)$. Entretanto, a função *DescobreMenorCobertura()* é chamada 2^V vezes. Portanto, a complexidade dessa função é $O(I * 2^V(N * V))$.

4. Organização do código, detalhes técnicos e execução

4.1. Organização

o código foi divido em vários arquivos .c e .h para a modularização do código, são eles:

grafo_heuristica.c/h Implementa as funções relacionadas à solução heurística da Cobertura de Vértices para Listas de Adjacência.
grafo_otimo.c/h Implementa as funções relacionadas à solução ótima da Cobertura de Vértices para Matriz de Adjacência.
io.c/h Implementa as funções para leitura, escrita e fechamento de arquivos.
main_e.c Executa o programa principal para a solução ótima.
main_h.c Executa o programa principal para a solução heurística.

Um arquivo adicional (**combinacoes.txt**) é criado, para armazenar todos os subconjuntos de um conjunto de vértices.

4.2. Detalhes técnicos

- IDE de desenvolvimento: Netbeans 7.3.1;
- Processador: Core i5;
- Memória RAM: 4GB;
- Sistema Operacional: Linux Mint 15.

4.3. Compilação e execução

O código fonte deve ser compilado e os executáveis gerados através do compilador GCC através do *Makefile*.

make

Serão gerados dois executáveis, são eles: *tp2e*, o qual dá a solução ótima, e *tp2h*, o qual dá a solução heurística. Os executáveis necessitam de dois parâmtros, são eles:

1. Arquivo de entrada *input.txt* com as instâncias do problema.
2. Arquivo de saída *output.txt* que conterá o resultado gerado pelo programa.

Para rodar os executáveis, é necessário digitar o seguinte comando no terminal.

./tp2e *input.txt output.txt*
./tp2h *input.txt output.txt*

5. Testes

Testes foram realizados para comprovar os testes de complexidade do programa, tanto para a solução ótima[3] quanto para a solução heurística[2]. Como a solução heurística foi implementada por meio de listas de adjacência, a complexidade ficou baixa ($O(V + A)$). Logo, o número de vértices pode ser muito grande que o desempenho do programa não ficará comprometido. Para os testes realizados, a solução do problema se dava instantaneamente.

Por outro lado, para a solução ótima, era necessário testar todos os subconjuntos de um conjunto de vértices. Então, foi escolhida a representação por matriz de adjacência. Como a função principal era chamada várias vezes, o desempenho do programa ficou comprometido para um número baixo de vértices. Como mostrado no gráfico abaixo, a partir de número de vértices $V \geq 20$ o programa começou a demorar mais de $1 segundo$ para executar.

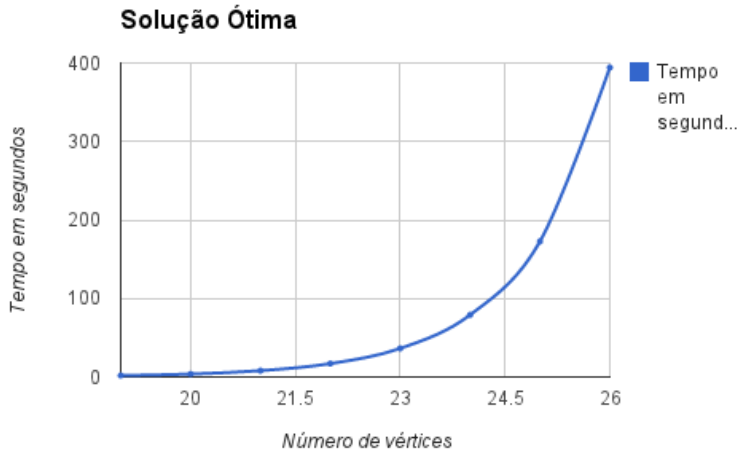

Figura 1. Testes de tempo de execução para a solução ótima

A tabela a seguir mostra os valores para os quais o gráfico acima foi gerado.

Número de Vértices	Tempo de exec. em segundos
19	1.87
20	3.75
21	8.04
22	17.04
23	36.4
24	79.09
25	172.68
26	394.28

Tabela 1. Tempos de execução para a solução ótima

6. Considerações

A resolução do problema da Cobertura de Vértices foi muito interessante. Uma vez que propiciou ao aluno o trabalho com um problema da classe NP-Completo, a qual é muito importante em Ciência da Computação.

Ao longo do trabalho pude perceber o quanto ter uma solução heurística é importante, pois se não a tivermos, nos deparamos com um problema que só pode ser resolvido com custo exponencial, o que na maioria das vezes não é viável.

Infelizmente, demorei muito tempo para implementar a solução heurística, visto que precisava fazer muitas manipulações de ponteiro, alocação e desalocação, o que causou muitas falhas de segmentanção.

A partir dos testes realizados pude perceber o quão custosa uma solução ótima pode ser, logo, sempre que possível, é melhor achar uma solução heurística para um problema sabidamente da classe NP-Completo. Para que essa solução seja interessante, é importante que faça a prova que ela é um algoritmo aproximativo. Se a solução for um

algoritmo aproximativo, teremos um limite superior para a solução, o que é importante para saber a acurácia da solução proposta.

Por fim, concluo que o trabalho foi extremamente importante para me mostrar uma visão diferente dos problemas em Ciência da Computação, os quais podem ser muito difíceis de resolver. Também pude praticar bastante os conceitos de manipulação de ponteiros e técnicas para se fazer a abordagem de um problema. Portanto, tenho que certeza que para os próximos problemas que me deparar, conseguirei ter uma visão diferente sobre eles e possivelmente propor uma solução melhor.

Referências

[1] CORMEN, T. H. *Algoritmos - Teoria e Prática*. Elsevier, 2001.

[2] ZIVIANI, N. *Projeto de Algoritmos*, 4 ed. Pioneira, 1999.