Teoria dos Grafos

Lista 1

Profª Luciana Lee

André Thiago Borghi Couto

Questão 1)

> *a) Sabemos que para T ser uma árvore geradora de G, deve conter os n vértices de G e conter apenas n-1 arestas de G. Portanto cada vértice de tem apenas um caminho para chegar em outro vértice, assim se v tem grau > 1, ele possui ligação com dois vértices diferentes, e ao removê-lo irá resultar em um grafo desconexo, ou seja, um grafo com duas ou mais componentes conexas.*

> *b) Para v ser um vértice de corte, ao removê-lo deveremos obter pelo menos duas componentes conexas. Sabendo que existe apenas um caminho entre qualquer vértice em uma árvore geradora, se v é um vértice de corte ele deve ser adjacente a pelo menos outros dois vértices, sendo assim o grau de v é maior que 1.*

> *c) Não. Como o vértice é "folha" ele contém apenas um vértice adjacente, e ao removê-lo a árvore gerada continuara com apenas uma componente conexa, logo, as folhas não podem ser vértices de corte.*

Questão 2)

> *Para um grafo ser conexo deve-se encontrar pelo menos um caminho ligando quaisquer dois vértices pertencente ao grafo, se existem n vértices e n-1 arestas, então para ser conexo deve ocorrer ciclos entre os vértices, assim cumprimos as propriedades de uma árvore.*

Questão 3)

> *Chamando m de número de arestas e n de número de vértices, temos que para T ser árvore geradora de G,* $nt = ng$*, ou seja, T possui todos os vértices de G. Sabemos também que para T ser árvore não existe ciclos entre seus vértices, então,* $mt = nt - 1 = ng - 1$*, e para continuar com a propriedade de árvore, T é conexo, onde todas as arestas são pontes. Daí T é subgrafo conexo minimal de G.*

Questão 4)

> *Se T é arvore, então é conexa e sem ciclos, tendo em mente essas duas definições, o grau de um vértice indica quantos "galhos" diferentes existem independentes dos outros. Logo o maior grau de T, representa o número mínimo de folhas que existe em T, pois cada "galho" possui de 1 ate* $(\Delta(T) - 1)$ *folhas.*

Questão 5)

> *Cada aresta contribui exatamente para um grau de entrada e um grau de saída. Portanto, a soma das entradas = soma das saídas = número total de arestas do grafo.*

Questão 6)

> *a) Entendemos que o nosso grafo contem n vértices e assim é representado por uma Matriz de adjacência* M_{nxn} *. O enunciado nos diz que vamos ter valores de vértices* $0...N(n) - 1$ *que é devido a implementação na linguagem C para se aproveitar do índice de posição 0. Analisando uma matriz:*

$$\mathbf{M}_{nxn} = \begin{bmatrix} 0 & u0j1 & u0j2 & u0j3 & ... & u0jn-1 \\ & & u0j1 & u0j1 & ... & ... \\ & & & u1j3 & ... & un-4jn-1 \\ & & & & ... & un-3jn-1 \\ & & & & & un-2jn-1 \end{bmatrix}$$

> *Fica fácil perceber que para representar os valores dessa matriz com os índices variando de 0…N-1 o vetor para armazenar essa matriz precisara ter tamanho:*
>
> $N = 1 + 2 + 3 + \cdots + (n-3) + (n-2) + (n-1)$
>
> *Essa equação pode ser representada pelo somatório:*
>
> $N = \sum_{p=1}^{n-1} p = 1 + 2 + 3 + \cdots + (n-3) + (n-2) + (n-1)$

> *b) Essa equação pode ser encontrada analisando a relação entre as duas variáveis utilizadas para percorrer a matriz u e v. Vamos primeiro ter um caso inicial onde o valor se encontra na primeira linha da matriz:*
>
> $f(u, v) = v - 1$
>
> *Notamos que o valor é diminuído por 1 pois não devido a forma que a linguagem C trabalho com vetores é interessante utilizar a primeira posição do vetor logo diminuir o valor de v por 1 para utilizar o vetor completo. Para o caso que u ≠ 0:*
>
> $f(u, v) = pos(max(u-1)) + v - u$
>
> *Onde a posição do máximo de* $u - 1$ *é a posição em que se encontrava o ultimo valor de* $u - 1$. *O comportamento da matriz vai eliminado os valores de u conforme descemos na matriz então armazenar o valor da ultima posição combate esse comportamento.*

> *c)Anexo*

> *d)Anexo*

Questão 7)

> *a) Sabemos que para um subgrafo H ser induzido de G, para quaisquer dois vértices u e v pertencentes a G que estão presentes em H, então existe a aresta (u,v) se a mesma pertencer a G. Daí, vemos que a expressão dos grafos induzidos dado o número de n é:*
>
> $N = \sum_{p=1} \frac{n!}{p!(n-p)!}$
>
> *Que é o somatório de todas as combinações realizadas escolhendo de p a p o número de vértices do subgrafo induzido, onde a ordem dos vértices não importa.*

> *b) Como o subgrafo é gerador temos a certeza de que todos os vértices de G estão presentes em H, portanto o número de subgrafos é dado por combinação das arestas m, tomadas de p a p:*
>
> $N = \sum_{p=1} \frac{m!}{p!(m-p)!}$
>
> *Onde* $p = 1$ *é quando existe apenas uma aresta e* $p = m$ *é o número máximo de arestas.*

> *c) Definição: Um subgrafo gerador de um grafo* $G = (V, E)$ *é um grafo* $G1 = (V, E1)$ *que possui os mesmos vértices de G. Quando o subgrafo gerador é uma árvore ele recebe o nome de árvore geradora.*

Questão 8) Anexo

Questão 9)

> *Se T é uma árvore, não possui ciclos e é conexo, portanto cada vértice v de T possui pelo menos um vértice adjacente, e neste caso, v é uma folha. O grau de uma folha é 1, e a sua remoção resulta em apenas uma componente conexa, se o* $\delta(v) > 1$, *v é chamado de vértice de corte, pois a sua remoção irá gerar pelo menos duas componentes conexas e esse número depende do grau de v, pois como não existe ciclos não há como traçar um caminho para chegas nos vértices ligados a v após a sua remoção, logo* $|C(T - v)| = \delta(v)$.

Questão 10)

> *No percurso em largura os vértices alcançados são colocados em uma fila, ou seja, o que está a mais tempo na fila é o primeiro a ser utilizado. Com esse esquema garantimos que primeiro encontramos todos vértices adjacentes a raiz, e descendo os níveis de acordo com o número de vértices necessários para chegar a raiz. Se um vértice já foi encontrado não fazemos nada, evitando a criação de ciclos. No final o resultado é uma arvore onde o caminho da raiz r até qualquer vértice v, possui a menor quantidade de arestas possível.*

Questão 11)

> *a) As arvores em largura para um grafo completo adquirem uma forma interessante. Entende-se que é escolhido uma raiz aleatória. A principal característica do percurso em largura é que ele visita todos os nós irmãos, primeiro, para depois visitar os filhos, os irmãos de seus filhos e assim por diante. Considerando um grafo completo de 4 vértices considerando que os vértices são numerados de* $0, \ldots, 3$ *e selecionando um vértice qualquer como raiz, entendemos que como o grafo é completo e estamos falando sobre um grafo não orientado então qualquer vértice com raiz terá uma arvore com forma idêntica, mudando apenas olocal dos vértices na arvore. Escolhendo 0 como raiz temos que um irmão qualquer será visitado, marcado como visitado, e assim por diante até o ultimo irmão. Quando esse processo acabar seria quando a arvore então visitaria o primeiro filho da raiz 0, porem esse filho já foi visitado, pois ele também é um irmão. Parando para analisar, todos os vértices já foram visitados, pois todos são irmãos logo assim encerra o percurso.*
> *O resultado é uma arvore balanceada de altura 1 onde todos os nós folhas são interligados por arestas para trás. Ela é balanceada pois a raiz é o único nó não folha e sua estrutura lembra uma arvore n-aria após o processo de cisão.*

> *b) O processo de busca em profundidade é caracterizado pelo início em uma raiz, e uma chamada recursiva de visitas até não existe mais predecessores. Neste caso a chamada recursiva então retorna um passo para cima e analisa então para todos os nós que ainda não foram visitados até não existirem mais predecessores. Esse processo recursivo ocorre até toda a arvore for visitada.*
> *A forma que esse processo ocorre gera uma arvore de altura* V *onde* V *é o número de vértices para nosso grafo completo. Iniciando na nossa raiz 0, ocorre uma escolha aleatória de visita em filho, mas como todos os vértices são interligados por arestas, acontece um fenômeno que o processo recursivo encontrar* $V - 1$ *filhos para a Raiz, logo a nossa arvore lembrará a estrutura de uma lista encadeada com alguns vértices tendo arestas para trás.*

Questão 12)

> *a) Para a arvore encontrada no arquivo anexo referente a questão 12:*
> *Percurso Pré-Ordem:* $AHEFBGDCJI$
> *Percurso Pós Ordem:* $CDJGBFEIHA$

> *b) Não existe vértice de corte na arvore. Pela definição um vértice de corte é encontrado quando se for o removido o vértice então é criado uma nova componente conexa no grafo. Também sabemos que não existe vértice de corte pois esse grafo é fortemente conexo. Neste caso é garantido que remover um vértice não irá criar uma nova componente conexa no grafo.*

Questão 13)

> *a) No percurso em profundidade, a estrutura em que colocamos os vértices alcançados pela primeira vez é uma pilha, logo, ao inserir o primeiro vértice* v *adjacente a raiz* r*, na próxima iteração a busca será feita a partir de* v*, e segue o processo para os descendentes de* v*, até que o programa comece a desempilhar até voltar a* r*, se ainda existir vértices adjacentes a* r*, o procedimento continua até não existir. Caso exista vértices adjacentes a raiz após o primeiro empilhamento e desempilhamento, significa que nenhum dos descendentes de* w *possui ligação com os mesmos, portanto concluímos que* r *é vértice do corte em* G*.*

> *b) Considerando que* v *é um vértice de corte, para que isso ocorra seus descendentes não podem ter "ligações" com os seus ancestrais, pois implicaria em um ciclo em* G*, fazendo com que exista outro caminho para sair dos ancestrais e chegar nos descendentes. Portanto se em* G*,* v *liga os ancestrais até os descendentes com um único caminho, significa que* v *é um vértice de corte.*

> *c) De maneira análoga ao que ocorre no exercício anterior, se não existir caminhos entre os ancestrais de u com um descendente de v, temos que em G, (u, v) é uma ponte, ou seja, uma aresta de corte que ao ser removida gera duas componentes conexas.*

Questão 14)

```
void dfsRcConexa(grafo *graph, int *cc, int v, int id){
    cc[v] = id;
    for (int w = 0; w < graph->vertice; ++w){
        if (graph->adj[v][w] != 0 && cc[w] == -1){
            dfsRcConexa(graph, cc, w, id);
        }
    }
}

int cConexa(grafo *graph, int *cc){
    int id = 0;
    for (int v = 0; v < graph->vertice; ++v)
        cc[v] = -1;
    for (int v = 0; v < graph->vertice; ++v){
        if (cc[v] == -1) dfsRcConexa(graph, cc, v, id++);
    }
    return id;
}
```

Questão 15)

```
int dfsBipart(Grafo *graph, int v, int *color, int c){
    color[v] = c;
    for (int w = 0; w < graph->vertice; ++w) {
            printf("color %d v %d w %d\n", color[w], v, w);
        if (graph->adj[v][w] != 0 && color[v] == -1) {
                printf("color %d v %d w %d\n", color[v], v, w);
            if (dfsBipart(graph, w, color, 1 - c) == 0) {
                return 0;
            }
        } else if (color[w] == c){
            printf("color %d v %d w %d c %d\n", color[w], v, w, c);
            return 0;
        }
    }
    return 1;
}

int biparty(Grafo *graph, int *color){
    for (int v = 0; v < graph->vertice; ++v)
        color[v] = -1;
    for (int v = 0; v < graph->vertice; ++v)
        if (color[v] == -1){
```

```
            if (dfsBipart(graph, v, color, 0) == 0){
                return 0;
            }
        }
    return 1;
}
```

Questão 16)

```
int *pre, *low;
int *stack, *pa;
int N, k, cnt;
```

```
void rPonte(Grafo *graph, int v, int *sc) {
    pre[v] = cnt++;
    int min = pre[v];
    stack[N++] = v;
    Vertice *aux = &(graph->adj[v]);
    while(aux != NULL){
        int w = aux->no;
        if (pre[w] == -1) {
            rPonte(graph, w, sc);
            if (low[w] < min){
            min = low[w];
            }                           //A
        } else if (pre[w] < pre[v] && sc[w] == -1) {
            if (pre[w] < min) {
                min = pre[w];
            }                           //B
        }
        aux = aux->prox;
    }
    low[v] = min;
    if (low[v] == pre[v]) {             //C
        int u;
        do {
            u = stack[--N];
            sc[u] = k;
        } while (u != v);
        k++;
    }
}
```

```
int isPonte(Grafo *graph, int *sc) {
```

```
    pre = (int *) malloc(graph->vertice * sizeof (int));
    low = (int *) malloc(graph->vertice * sizeof (int));
```

```
    stack = (int *) malloc(graph->vertice * sizeof (int));
```

```
    for (int v = 1; v < graph->vertice; ++v) pre[v] = sc[v] = -1;

    k = N = cnt = 0;
    for (int v = 0; v < graph->vertice; ++v){
        if (pre[v] == -1){
            rPonte(graph, v, sc);
        }
    }
    free( pre); free( low); free( stack);
    return k;
```

```
}
```