Resolução de Problema com Grafos

Ediana da Silva de Souza[1], Êrica Peters do Carmo[1]

[1]Departamento de Ciência da Computação – Universidade do Estado de Santa Catarina(UDESC)
Joinville – SC – Brasil

edianadasilvadesouza@gmail.com, ericapetersc@gmail.com

Resumo. *Este artigo foi escrito como requisito para a conclusão da disciplina de Teoria dos Grafos do curso de Bacharelado em Ciência da Computação da Universidade do Estado de Santa Catarina. A proposta é a resolução de um problema envolvendo plano cartesiano, distância entre pontos, busca em profundidade, busca em largura e o algoritmo de Dijkstra, através de modelagem de grafos.*

1. Problema

O problema consiste - inicialmente - na construção de grafos, tendo como base o plano cartesiano, distância entre pontos e logaritmo na base dois em cada vértice (para obter a quantidade de vértices adjacentes a um vértice). Com isso, deveria-se construir grafos de tamanho (G50, G100, G500 e G1000). Posteriormente, pretendiam-se executar os algoritmos de busca em profundidade, busca em largura e dijkstra, para conseguir atender as exigências do projeto proposto.

2. Proposta de Resolução

2.1 Modelagem do Grafo

Optou-se por utilizar a linguagem de programação C++, por afinidade das autoras com esta, e utilizar o compilador g++ no Linux. Como forma de modelagem, resolveu-se representar o grafo por um vetor de vértices, onde cada vértice *u* é representado por uma struct formada pelos valores *x* e *y* do plano cartesiano, o seu índice e um *vector* que armazena os vértices adjacentes a *u* e os pesos das arestas que vão de *u* para *v*. Dessa forma, cada elemento do vector é um *pair* formado pelo vértice vizinho *v* e peso da aresta *uv*.

Observação: É importante ressaltar que, para fins matemáticos, optou-se por considerar os índices dos vértices a partir do número 1 até n. Dessa forma, o *index* de um vértice é sempre sua posição no vetor de vértices (grafo) + 1.

```
struct vertice{
    int index;
    float x;
    float y;
    vector<pair<vertice, float>> adj;
    /*lista de adjacência adj
    O primeiro valor é o vizinho e o segundo valor é o peso da aresta*/
    vertice(){}
};
```

Figura 1. Estrutura do vértice

```
class Grafo{

private:
    int V; // número de vértices
    vertice *vertices;//vetor de vértices que é alocado dinâmicamente
```

Figura 2. Estrutura do Grafo

De acordo com o enunciado, os vértices adjacentes a um vértice *u* foram obtidos da seguinte forma: primeiro, foi calculado a parte inteira da função $log_2 index\_de\_u$ cujo resultado era o número *k* de vértices adjacentes a *u*; posteriormente, procurou-se os *k* vértices mais próximos de u (pela distância euclidiana) e foram criadas arestas saindo de *u* até cada um deles.

A seguir apresenta-se o exemplo de um grafo g15 (grafo formado por 15 vértices) gerado em uma execução do projeto.

```
Vertices
Vertice 1 (0.749666,0.955521);
Vertice 2 (0.8458,0.0398184);
Vertice 3 (0.0549874,0.421769);
Vertice 4 (0.86262,0.944575);
Vertice 5 (0.303173,0.716218);
Vertice 6 (0.071144,0.463436);
Vertice 7 (0.433841,0.715137);
Vertice 8 (0.440835,0.442052);
Vertice 9 (0.774784,0.043565);
Vertice 10 (0.968141,0.55847);
Vertice 11 (0.213645,0.966369);
Vertice 12 (0.346576,0.226325);
Vertice 13 (0.724671,0.218994);
Vertice 14 (0.372537,0.891341);
Vertice 15 (0.283184,0.986464);
```

Figura 3. Vértices gerados para um grafo g15.

```
Lista Adjacencia
Vertice 1:
Vertice 2: (9,0.0711156);
Vertice 3: (6,0.0446901);
Vertice 4: (1,0.113483); (10,0.400265);
Vertice 5: (7,0.130672); (14,0.18836);
Vertice 6: (3,0.0446901); (5,0.343127);
Vertice 7: (5,0.130672); (14,0.186564);
Vertice 8: (12,0.23542); (7,0.273175); (5,0.306786);
Vertice 9: (2,0.0711156); (13,0.182446); (12,0.465578);
Vertice 10: (4,0.400265); (13,0.417758); (1,0.45319);
Vertice 11: (15,0.0723843); (14,0.175715); (5,0.265689);
Vertice 12: (8,0.23542); (3,0.351031); (6,0.363435);
Vertice 13: (9,0.182446); (2,0.216278); (8,0.360995);
Vertice 14: (15,0.130508); (11,0.175715); (7,0.186564);
Vertice 15: (11,0.0723843); (14,0.130508); (5,0.270985);
```

Figura 4. Lista de adjacência - cada par é do tipo (index do vértice vizinho, peso da aresta) - de acordo com o grafo g15 gerado.

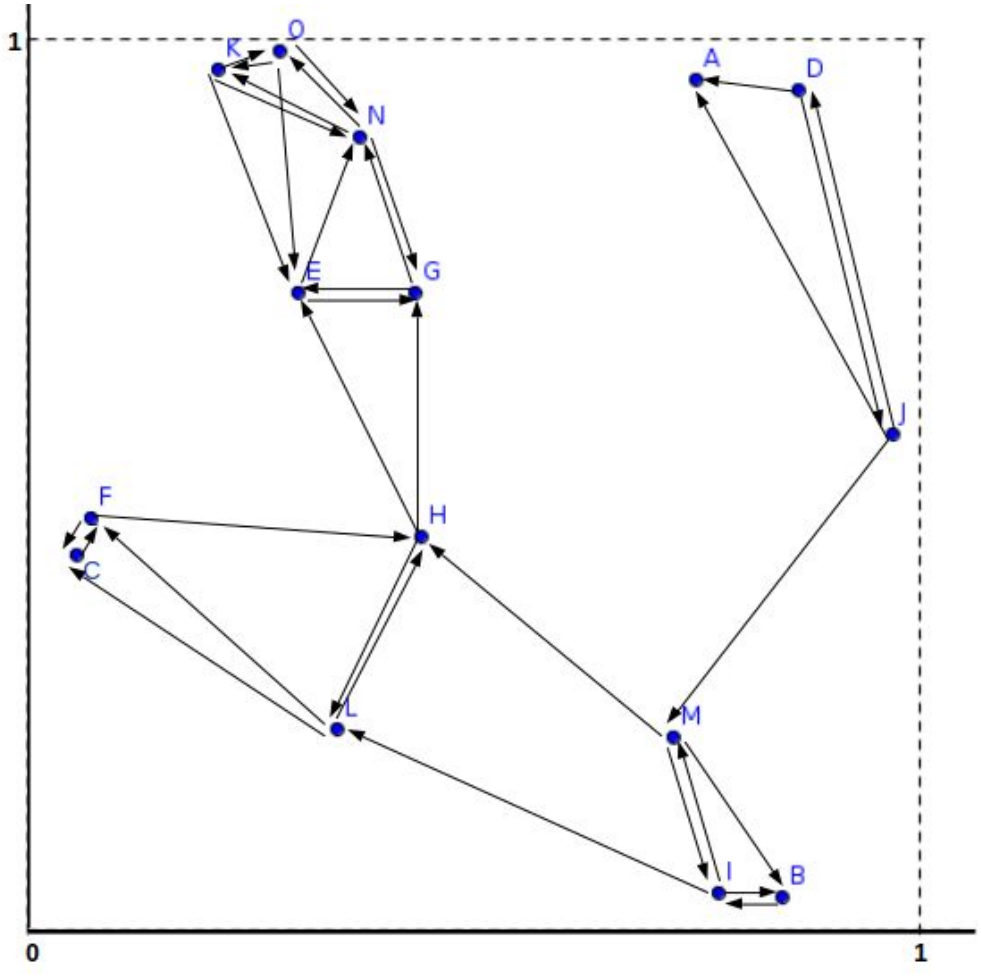


Figura 5. Representação visual do grafo g15 gerado.

Observação: Nota-se que para a representação visual, os índices numéricos foram substituídos pelas letras correspondentes do alfabeto.

2.1 Métodos

Utilizando C++, também foi possível utilizar o padrão de programação orientada à objetos. Assim, cada grafo (G50, G100, G500 e G1000) é um objeto da classe grafo.

Para a resolução do problema, como já comentado anteriormente, é necessária a modelagem e construção dos grafos, além da implementação de algoritmos de manipulação de grafos que permitam o cálculo do menor caminho considerando as distâncias entre os vértices. Também é necessário percorrer os grafos construídos para conseguir explorar todos os seus vértices e arestas. Abaixo, iremos explicar os métodos utilizados:

- Construtor do grafo: utilizado para gerar um grafo, atribuindo seu tamanho, seus vértices e suas arestas. O parâmetro de entrada deste método é um inteiro *V* que representa o número de vértices do grafo a ser gerado. Como especificado no enunciado do problema, os vértices do grafo são criados com valores de *x* e *y* entre 0 e 1, e suas arestas são criadas ao invocar os métodos *log* e *criar arestas*;
- Imprime vértices: esse método percorre o vetor de vértices, imprimindo na tela o valor do index (índice do vértice), e os valores referente ao *x* e *y* no plano cartesiano. Um exemplo da saída desse método é a Figura 3;
- Adicionar arestas: esse método adiciona o vértice de destino e o seu custo ao *vector* de adjacentes do vértice de origem. Por exemplo, sendo *u* um vértice e *v* seu adjacente com distância 0.8, considera-se o código abaixo onde origem=*u*, destino=*v* e custo=0.8:

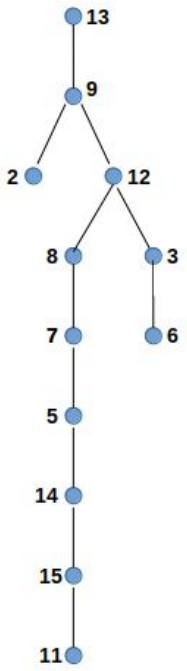
```
void adicionarAresta(vertice origem, vertice destino, float custo)
{
    vertices[origem.index-1].adj.push_back(make_pair(destino, custo));
}
```

Figura 6. Método Adicionar arestas

- Criar arestas: calcula a distância de um vértice para todos os outros vértices, utilizando a fórmula euclidiana da distância entre dois pontos. Como no construtor adicionamos *k* (onde $k=log_2 index\_de\_u$) vértices auxiliares com custo infinito no vector de adjacentes de cada vértice, comparamos o valor resultante do uso da fórmula da distância com o último valor que está no vector de adjacentes, e se o primeiro for menor, o último será substituído. Após cada substituição realizamos a ordenação do vector de acordo com os custos das arestas;
- Log: é utilizado para saber quantos vértices adjacentes um vértice terá, será considerado somente a parte inteira do log na base dois;
- Imprime adjacentes: percorre os vetor de vértices e vértices adjacentes de um

vértice, imprimindo estes - como mostrado na Figura 4.

- Chama dfs: dentres os vértices que o objeto possui, escolhe-se uma raiz aleatoriamente - utilizando o método *random* - e chama o método dfs;
- Dfs: o conceito do algoritmo de depth-first search é começar a percorrer um grafo pela raiz informada e explorar seus ramos tanto quanto possível antes de retroceder (backtracking). Para isso, foi utilizado a estrutura de dados pilha (o primeiro a chegar é o último a sair) - para colocar os vértices que estão sendo visitados - e a estrutura de dados booleana vector - para colocar uma etiquetagem nos vértices: se já foram visitados ou não. O algoritmo é feito da seguinte maneira:
    - No primeiro momento, marca-se todos os vértices como não visitados.
    - Faz uma interação enquanto for verdade:
        - Visita a raiz (passada no método chama dfs);
        - Marca a raiz como visitado;
        - Insere na pilha;
        - Busca os vértices adjacentes à raiz, percorrendo o vector de adjacentes;
        - Quando encontra o vértice adjacente à raiz, verifica se ele não foi visitado.
            - Caso positivo, marca como visitado e este passa a ser a nova raiz, voltando para o início do algoritmo;
            - Se todos os vértices adjacentes já tiverem com a etiquetagem de visitado, a raiz será removida da pilha.
                - Se a pilha ficar vazia, o dfs acabou;
                - Se não, o elemento do topo da pilha se torna a nova raiz e voltamos para o início do algoritmo;



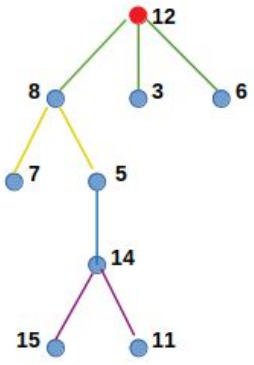

```
DFS
Visitando vertice 13
Visitando vertice 9
Visitando vertice 2
Voltando para o vertice 9
Visitando vertice 12
Visitando vertice 8
Visitando vertice 7
Visitando vertice 5
Visitando vertice 14
Visitando vertice 15
Visitando vertice 11
Voltando para o vertice 15
Voltando para o vertice 14
Voltando para o vertice 5
Voltando para o vertice 7
Voltando para o vertice 8
Voltando para o vertice 12
Visitando vertice 3
Visitando vertice 6
Voltando para o vertice 3
Voltando para o vertice 12
Voltando para o vertice 9
Voltando para o vertice 13
```

Figura 7. Árvore gerada pelo DFS aplicado no grafo da Figura 3.

Figura 8. Saída do programa na execução do DFS aplicado no grafo da FIgura 3.

Observação: Na execução mostrada acima, o vértice 13 foi escolhido aleatoriamente como raiz pelo método chama dfs.

- Chama bfs: dentres os vértices que o objeto possui, escolhe-se uma raiz aleatoriamente - utilizando *random* - e chama o método bfs;
- Bfs: o conceito do algoritmo de breadth-first search é, assim como o dfs, começar a percorrer o grafo por uma raiz, porém, o bfs irá explorar todos os vértices adjacentes primeiro. Para isso, foi utilizado a estrutura de dados booleana vector - para colocar uma etiquetagem nos vértices: se já foram visitados ou não - e a estrutura de dados fila (primeiro à chegar é o primeiro a sair) - para colocar os vértices visitados, bem como seu nível na árvore (ou seja, uma fila com pair). O algoritmo é feito da seguinte maneira:
    - Inicia a variável nível como zero (ela irá indicar o nível de cada vértice na árvore e auxiliará no momento da impressão colorida - como requisitado no projeto). Ela é um contador que incrementa sempre que um vértice adiciona os vértices adjacentes na fila (identificado por * nessa explicação);
    - No primeiro momento, coloca a etiquetagem de não visitados em todos os vértices.
    - Visita a raiz (passada no método chama bfs);
    - Marca a raiz como visitada;
    - Insere na fila o vértice e o seu nível;
    - Faz uma interação enquanto a fila não estiver vazia:
        - Retira da fila o vértice visitado;
        - Verifica se a raiz possui vizinhos:
            - Se negativo e a fila não estiver vazia, a raiz vai passar a ser o primeiro elemento da fila, vai percorrer esta e remover da fila;
            - * Se positivo, busca os vértices adjacentes, marca todos os vértices não visitados como visitados e insere-os na fila;
    - Repete a interação até que a fila fique vazia.



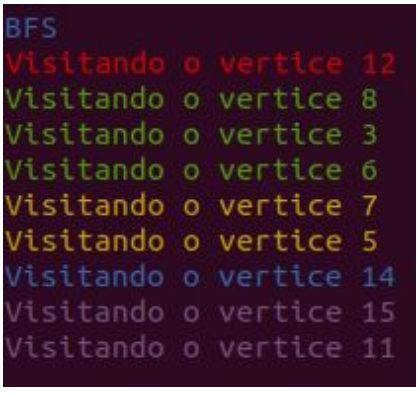
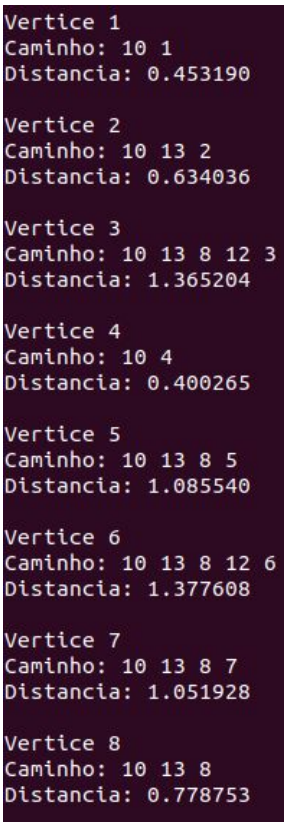


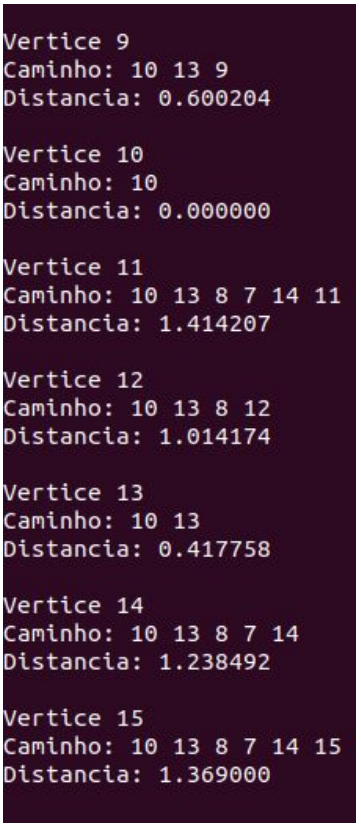
Figura 9. Árvore gerada pelo BFS aplicado no grafo da Figura 3.

Figura 8. Saída do programa na execução do BFS aplicado no grafo da FIgura 3.

Observação: Na execução mostrada acima, o vértice 12 foi escolhido aleatoriamente como raiz pelo método chama bfs.

- Chama dijkstra: dentres os vértices que o grafo possui, escolhe-se uma raiz aleatoriamente - utilizando random - e chama o método dijkstra;
- Dijkstra: o conceito do algoritmo é conseguir obter o caminho mais curto de uma origem até um destino. Para isso, foi utilizado a estrutura de dados vector - para representar as distância, os vértices visitados e os vértices antecessores. Também precisa-se garantir que, dentre os vértices adjacentes, sempre se escolherá o vértice com a menor distância (para isso utilizamos uma fila de prioridade). O algoritmo é feito da seguinte maneira:
    - No primeiro momento, coloca a etiquetagem de não visitados em todos os vértices e a distância infinita;
    - Indica que a distância de um vértice de origem para ele mesmo é zero;
    - Insere o vértice na fila de prioridade;
    - Faz uma interação enquanto a fila não estiver vazia:
        - Extrair do topo da fila um vértice que será expandido e armazena-o em uma variável;
        - Remove o vértice da fila;
        - Verifica se o vértice que será expandido já foi visitado:
            - Se negativo, marca como visitado e percorre a lista de adjacentes, obtendo o vértice adjacente e o custo da aresta. Após isso, relaxa a aresta e verifica se a distância que já existe no vetor de distâncias é maior que o novo custo da aresta. Se for maior, atualiza a distância, insere na fila e insere o vértice expandido como antecessor do vértice cuja distância foi atualizada.
    - Repete até que a fila esteja vazia;
    - Imprime a distância mínima da raiz para cada vértice e o caminho percorrido (obtido através dos vértices antecessores marcados para cada vértice).

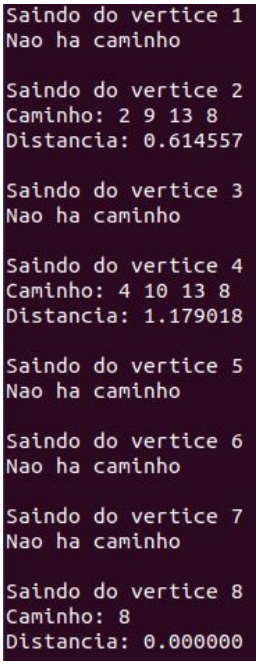


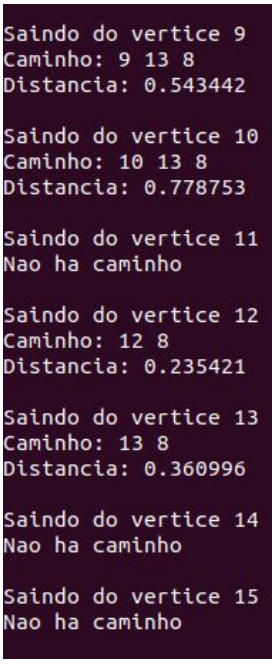


Figuras 9 e 10. Saída do programa na execução do Dijkstra (caminhos) aplicado no grafo da Figura 3.

Observação: Na execução acima, o vértice 10 foi escolhido aleatoriamente como raiz.

- Chama dijkstra modificado: dentres os vértices que o objeto possui, escolhe-se uma raiz aleatoriamente - utilizando random - e chama o método dijkstra, passando também um vértice fixo;
- Dijkstra modificado: é semelhante ao algoritmo do método dijkstra, porém, neste será apresentado a distância mínima de todos os vértices até um outro vértice como ponto fixo. Para isso, realiza o mesmo processo do dijkstra anterior, porém esse método é repetido para cada vértice do grafo e a cada repetição é armazenada a distância do vértice que está sendo analisado como raiz do Dijkstra até o ponto fixo.
- Imprime-se as distância de todos os vértices analisados até o vértice considerado o ponto fixo.





Figuras 11 e 12. Saída do programa na execução do Dijkstra Modificado (sai de todos os vértices e chega em um vértice fixo) aplicado no grafo da Figura 3.

Observação: Na execução acima, o vértice 8 foi escolhido aleatoriamente como ponto fixo.

Sobre os métodos, nota-se ainda que para chamar cada um deles, optou-se por apresentar um menu ao usuário em cada execução, de maneira que ele possa escolher com maior facilidade o tipo de operação que deseja visualizar no grafo.

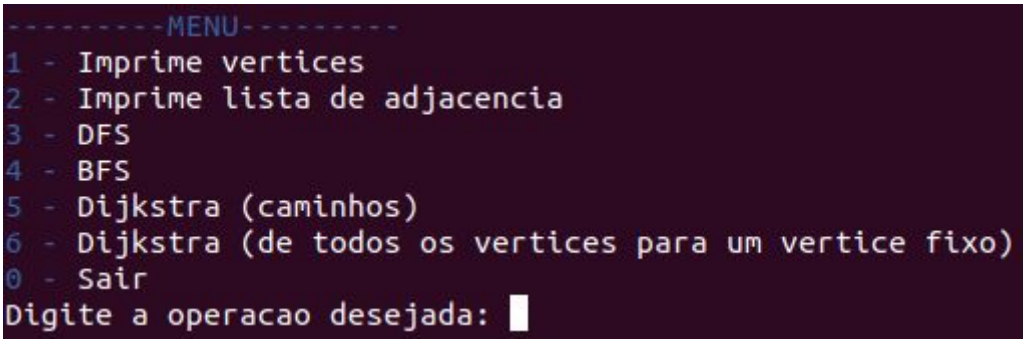


Figura 13. Menu do programa para interação com o usuário.

3. Resultados

Para rodar o algoritmo de maneira que todos os objetos possam ser criados, foi necessário abrir quatro janelas do terminal linux, pois, a memória não suportava criar a quantidade de vértices necessária em um mesmo processo. Apesar disso, é possível

executar os quatro processos paralelamente.

A partir dos resultados obtidos durantes as execuções do programa, concluímos que conseguimos resolver com êxito o problema proposto. Assim, concluímos que o objetivo do trabalho foi alcançado e que com este adquirimos uma visão prática dos conceitos estudados na disciplina de Teoria dos Grafos.