Lista #7

Partição Dinâmica e Grafos

Data de entrega: 20 de junho de 2017

Modifique os arquivos que acompanham este enunciado, a fim de implementar as questões pedidas, e envie-os de volta zipados com nome no padrão <numero_matricula>.zip por email para **profs-eda@tecgraf.puc-rio.br**, com o assunto [**EDA**] **Lista 7**. Atenção: Crie um arquivo contendo a função main do seu programa para testar suas implementações, mas envie SOMENTE os arquivos e as classes solicitadas.

---

1. A estrutura Union-find (ou Disjoint-set), vista em aula, armazena conjuntos de elementos, divididos em subconjuntos disjuntos. Com a operação de union é possível unir dois conjuntos em um só. Com a operação de find, é possível identificar a qual conjunto pertence um determinado elemento. Utilizando essa estrutura, fica fácil construir um labirinto: a idéia é partir de uma grade 2d de tamanho $m \times n$, onde cada célula é inicialmente um subconjunto de um único elemento. A partir daí, grupos vizinhos vão sendo unidos (paredes vão sendo derrubadas) até que reste somente um único grupo, quando então se torna possível ir de um ponto a qualquer outro do labirinto.

   (a) Implemente uma classe UnionFind na estrutura definida no código abaixo. Os elementos armazenados são implicitamente definidos por índices.

   A classe armazena um vetor de inteiros **parent**, que armazena o pai atual de cada elemento. Assim, por exemplo, se parent[2] = 9, significa que o pai do elemento 2 é o elemento 9.

   No vetor **size**, deve ser armazenado o tamanho atual de cada conjunto. Assim, size[a] contém o tamanho do conjunto cuja raiz é a. Se a não é raiz, size[a] deve conter zero.

   A variável **numSets** armazena a quantidade de conjuntos existentes. Ela é inicializada com o número de elementos inicial, e então decrementada sempre que dois conjuntos são unidos.

   IMPLEMENTE PATH-COMPRESSION e UNION-BY-SIZE.

```
#ifndef UNIONFIND_H
#define UNIONFIND_H
```

```
#include <vector>

class UnionFind
{
public:
    //Inicializa a UnionFind com o numero de elementos
    UnionFind(int n);

    //Destrutor
    ~UnionFind();

    //Retorna a raiz do conjunto a que u pertence
    int find(int u);

    //Une os dois conjuntos que contem u e v
    void makeUnion(int u, int v);

    //Retorna o numero de conjuntos diferentes
    int getNumSets();

private:
    //Armazena o pai de cada elemento
    std::vector<int> parent;

    //Armazena o tamanho de cada conjunto
    std::vector<int> size;

    //Quantidade de conjuntos atual
    int numSets;
};

#endif
```

(b) Utilize a classe UnionFind para gerar um labirinto. A estrutura do labirinto pode ser armazenada num vetor de "paredes", conforme mostrado na Figura 1. As paredes da direita e de baixo de todas as células são armazenadas sequencialmente no vetor, contendo valor true quando existem, e false, quando não existem mais. Inicialmente, todas as paredes existem, como na Figura 1.

O algoritmo consiste em manter as células representadas em uma UnionFind, e sucessivamente sortear paredes para serem derrubadas, até que só reste um conjunto na UnionFind. Duas regras no entanto, devem ser respeitadas:

- As paredes de borda não podem ser excluídas

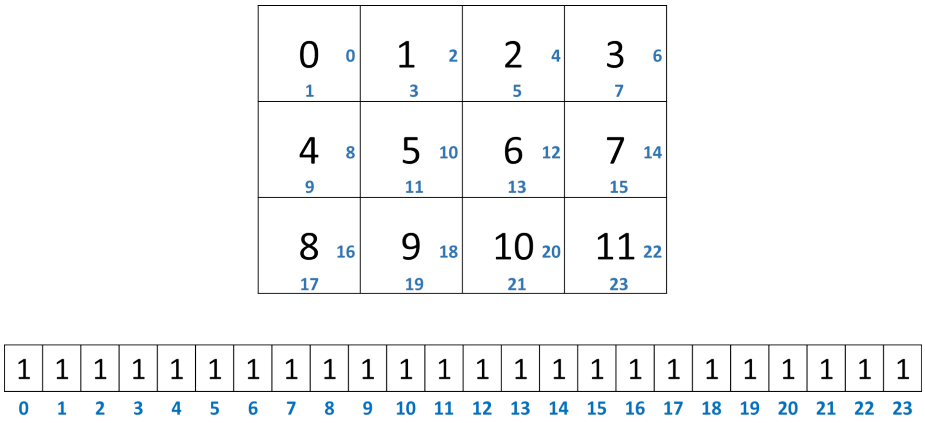

Figure 1: Labirinto inicialmente completo e sua representação em vetor.

- Uma parede sorteada só pode ser excluída quando suas células vizinhas pertencerem a grupos diferentes.

No fim, um resultado possível é mostrado na Figura 2. As paredes excluídas contém 0 (false) nas suas respectivas posições, e as bordas foram respeitadas.

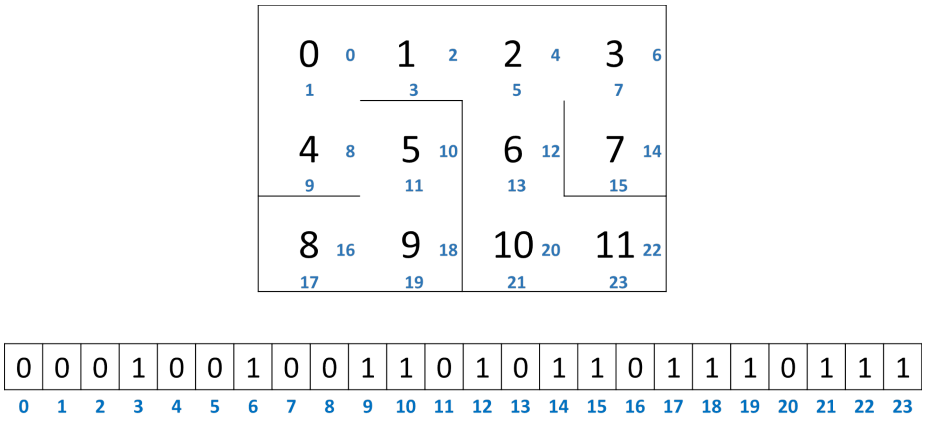

Figure 2: Possível labirinto gerado e sua representação em vetor.

Implemente a função createMaze, com a assinatura especificada abaixo. Esta função recebe como parâmetro de entrada a largura (m) e a altura (n) do labirinto, e também o vetor em que deve armazenar o resultado.

```
void createMaze(int m, int n,
                std::vector< bool >& maze);
```

Três outras funções já estão implementadas e podem ser utilizadas:

```
int randomInt(int from, int to);
void drawMaze(const std::vector< bool >& maze,
              int m, int n);
void printMaze(const std::vector< bool >& maze);
```

A função randomInt gera um número inteiro aleatório dentro do range especificado (inclusivo).

A função drawMaze desenha o labirinto no terminal. O labirinto da Figura 2 gera a saída:

```
+---+---+---+---+
|               |
+   +---+   +   +
|       |   |   |
+---+   +   +---+
|       |       |
+---+---+---+---+
```

E a função printMaze imprime o labirinto como texto, no formato "célula parede_direita parede_inferior". O mesmo labirinto gera a saída:

```
0 0 0
1 0 1
2 0 0
3 1 0
4 0 1
5 1 0
6 1 0
7 1 1
8 0 1
9 1 1
10 0 1
11 1 1
```

2. Implemente uma classe Graph, que armazena a configuração do grafo utilizando uma lista de adjacências. Uma vez definido, o número de nós do grafo não é alterado, ou seja, nós não são inseridos ou excluídos. A função para inserir arestas no grafo e a função de imprimir o grafo já estão implementadas.

   (a) Implemente as funções de grafo a seguir, respeitando as exigências definidas:

   void bfs(int s)
   Saída: Imprimir (std::cout) a ordem de visita dos nós e **preencher** o vetor distances, com as distâncias (em quantidade de arestas, SEM considerar os pesos) até o nó de partida. A forma de controlar os nós visitados fica ao seu critério (vetor de cores, vetor de booleanos...). O grafo da Figura 3, quando executado a partir do vertice 0, deve gerar a saída (separada por espaços como abaixo):

   ```
   0 1 5 2 6 4 3
   ```

   E o vetor distances deve conter os valores:

   ```
   0 1 2 3 2 1 2
   ```

   Dica: Usar a classe queue ou a classe vector, para a fila de próximos nós.

   void dfs(int s)
   Saída: Imprimir (std::cout) a ordem de visita dos nós. A forma de controlar os nós visitados fica ao seu critério (vetor de cores, vetor de booleanos...).O grafo da Figura 3, quando executado a partir do vertice 0, deve gerar a saída:

   ```
   0 1 2 6 3 4 5
   ```

   Dica: Definir uma função dfs-visit para ser chamada recursivamente.

   void djikstra(int s)
   Saída: Preencher o vetor distances com as distâncias mínimas até o nó de partida, agora sim considerando os pesos das arestas, conforme o algoritmo de Djikstra.
   Dicas: Armazenar os próximos nós e suas distâncias num Min-Heap. Como são duas informações, você pode por exemplo, usar um std::pair, ou definir uma struct (structs em c++11 podem ser definidas dentro de funções). Quanto ao heap, você pode utilizar a classe priority_queue, da stl, ou a sua classe Heap, implementada no T5. No primeiro caso, lembre-se de que por default a priority_queue é um Max-Heap, então verifique como utilizá-la como um Min-Heap.

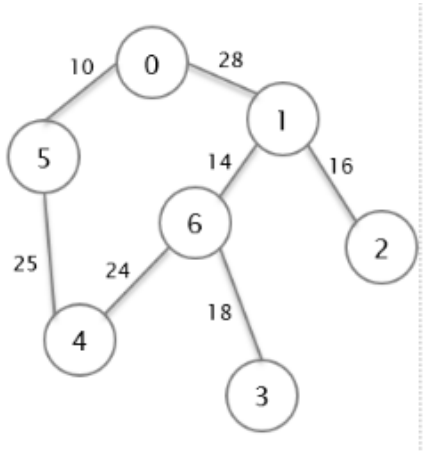

Figure 3: Exemplo de grafo.

No segundo caso, não esqueça de enviar sua classe heap quando enviar o trabalho.

Graph kruskal()
Saída: **Retornar** um grafo composto dos mesmos nós do grafo de entrada, mas contendo somente as arestas pertencentes à árvore geradora mínima.
Dicas: Utilize a UnionFind implementada neste trabalho. E para ordenar as arestas e inseri-las uma por uma no grafo resultante, fica mais fácil representá-las explicitamente: utilizando alguma estrutura que guarde os dois nós de extremidade e o seu peso. Como são três informações, você pode usar a classe tuple, da stl, ou definir uma struct dentro da função.

(b) Implemente as seguintes aplicações:

i. Um grafo **bicolorido** (ou **bipartido**) é um grafo cujos vértices podem ser divididos em dois grupos U e V, tais que toda e qualquer aresta une um vértice pertencente ao grupo U a um vértice pertencente ao grupo V. Em outras palavras, se atribuirmos as cores preta e branca aos vértices, um vértice branco sempre terá somente vizinhos pretos, e um vértice preto sempre terá somente vizinhos brancos.
Implemente a função **bool Graph::isBicolored()**, que retorna true caso o grafo seja bicolorido e false, caso contrário.
Dicas: O algoritmo para verificar se um grafo é bicolorido é muito semelhante a um dos algoritmos implementados na questão

anterior. Você pode utilizar as cores definidas no enum Color, da classe.

ii. Utilize a sua classe Graph para calcular onde devem ser as extremidades (entrada e saída) do labirinto da questão 1, de forma que estas estejam o mais distante possível uma da outra; e que uma delas (a saída) esteja na borda do labirinto.
Cada célula do labirinto é representada por um nó do grafo. Sempre que é possível passar de uma célula para outra, ou seja, não há parede entre as duas, existe uma aresta entre elas, de peso 1. A função para converter um labirinto expresso no vetor de paredes em um grafo já está implementada:

```
Graph createGraph(
    const std::vector<bool>& maze,
    int m, int n );
```

Implemente a função findStartEnd abaixo, em **maze.cpp**:

```
void findStartEnd(
    const std::vector<bool>& maze,
    int m, int n,
    int& start, int& end );
```

A função recebe como entrada o labirinto e as suas dimensões. E deve escrever nos parâmetros start e end o resultado encontrado.
Dicas: Note que, apesar de estar buscando os vértices que produzem o MAIOR caminho no labirinto, o caminho entre eles deve ser o MENOR possível, sem voltas desnecessárias.

Definição da classe Graph:

```
#ifndef GRAPH_H
#define GRAPH_H

#include <vector>

class Graph
{
public:
  //Construtor. N = numero de vertices.
  Graph( int N );

  /* Insere uma aresta no grafo, do no 'from' ao no
     'to', com peso 'weight'. Caso undirected esteja
     marcado como true, insere a aresta tambem no
     sentido to->from
   */
  void insertEdge( int from, int to,
```

```
                    int weight = 1,
                    bool undirected = true );

  //Imprime os vertices e seus vizinhos
  void print();

  //Imprime os nos do grafo na ordem de uma busca
  //em largura
  void bfs( int s );

  //Imprime os nos do grafo na ordem de uma busca
  //em profundidade
  void dfs(int s);

  //Retorna a arvore geradora minima do grafo
  Graph kruskal();

  //Calcula a menor distancia de s ate cada vertice
  void djikstra( int s );

  //Verifica se o grafo e bicolorido/bipartido
  bool isBicolored();

  //Menor distancia de cada no ate um vertice especifico
  std::vector<int> distances;

private:
  struct Edge
  {
    int v; //Vertice destino
    int w; //Peso da aresta
  };

  enum Color
  {
    WHITE,
    GRAY,
    BLACK
  };

  //Lista de adjacencias
  std::vector< std::vector< Edge > > G;
};

#endif // GRAPH_H
```