GraphsImplementations

August 7, 2020

1 Graph Implementation

https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/GraphImpl.html

1.0.1 Table of Contents

1.1 Graph Representations

- two traditional approaches to representing graphs:
  1. adjacency matrix
  - adjacency list
- this notebook demonstrates both approaches

Representing Directed Graph using Adjacency Matrix and List

Representing Undirected Graph using Adjacency Matrix and List

1.2 Graph ADT using Adjacency Matrix

- graph implementation presented here do not address the issue of how the graph is actually created
- graph can be built e.g., by reading the graph description from a file
- the graph can be built-up by using the addEdge method provided by the ADT

[11]:
```
#include <iostream>
#include <algorithm>
#include <vector>
#include <unordered_set>
#include <list>

using namespace std;
```

[12]:
```
class GraphM {
    private:
        vector<vector<int> > matrix;
        size_t numEdge;
        unordered_set<int> nodes;

    public:
        GraphM(size_t n) {
            this->numEdge = 0;
            // initializes matrix with 0s
            this->matrix.assign(n, vector<int>(n, 0));
            /*
            for (int i=0; i<n; i++) {
                vector<int> v(n, 0); // create an array of n elements␣
↪initialized with 0s
                matrix.push_back(v);
            }
            */
        }
        // returns the number of vertices/nodes
        size_t nodeCount() {
            return nodes.size();
        }

        // returns the number of edges
        size_t edgeCount() {
            return numEdge;
        }

        // adds a new edge from node u to node v, with weight w
        void addEdge(int u, int v, int w=1) {
            if (w == 0) return;
            matrix[u][v] = w;
            ++numEdge;
            nodes.insert(u);
            nodes.insert(v);
        }

        // get the weight value for an edge, u->v
        int weight(int u, int v) {
            return matrix[u][v];
        }

        // removes an edge from the graph
        void removeEdge(int u, int v) {
            matrix[u][v] = 0;
            --numEdge;
```

```
        }

        // checks if there's an edge between u and v
        bool hasEdge(int u, int v) {
            return matrix[u][v] != 0;
        }

        // returns vector containing neighbors of u
        vector<int> neighbors(int u) {
            vector<int> neighs;
            for(int v = 0; v<matrix[u].size(); v++) {
                if (matrix[u][v] != 0) neighs.push_back(v);
            }
            return neighs;
        }

        // prints graph
        void print() {
            for(auto u: matrix) {
                for (auto v: u)
                    cout << v << " ";
                cout << endl;
            }
        }

};
```

1.2.1 Let's test GraphM ADT using the following directed graph

[3]:
```
// test GraphM with some data
GraphM graph(5); // graph with 5 nodes; see directed graph above
```

[4]:
```
graph.addEdge(0, 1, 1); // let's say all weights are 1
graph.addEdge(0, 4, 1);
graph.addEdge(1, 3); // default weight is 1
graph.addEdge(2, 4);
graph.addEdge(3, 2);
graph.addEdge(4, 1);
```

[5]:
```
cout << "# of nodes = " << graph.nodeCount() << endl;
cout << "# of edges = " << graph.edgeCount() << endl;
```

```
# of nodes = 5
# of edges = 6
```

[6]:
```
graph.print();
```

```
0 1 0 0 1
0 0 0 1 0
0 0 0 0 1
0 0 1 0 0
0 1 0 0 0
```

[7]:
```
graph.removeEdge(0, 4);
cout << "# of nodes = " << graph.nodeCount() << endl;
cout << "# of edges = " << graph.edgeCount() << endl;
graph.print();
```

```
# of nodes = 5
# of edges = 5
0 1 0 0 0
0 0 0 1 0
0 0 0 0 1
0 0 1 0 0
0 1 0 0 0
```

[8]:
```
cout << "neighbors of 0 are: ";
for (auto n: graph.neighbors(0))
    cout << n << ", ";
```

```
neighbors of 0 are: 1,
```

[9]:
```
cout << boolalpha << " has edge 0->4? " << graph.hasEdge(0, 4) << endl;
```

```
 has edge 0->4? false
```

1.3 Graph ADT using Adjacency List

[13]:
```
#include <iostream>
#include <algorithm>
#include <vector>
#include <unordered_set>
#include <list>

using namespace std;
```

[14]:
```
class GraphL {
    private:
        vector<list<pair<int, int> > > graph; // list stores pair of neighbor␣
 ↪id and weight
        // can also use unorderd_map<int, vector<int> >
        size_t numEdge;
        unordered_set<int> nodes;

    public:
```

```
        GraphL(size_t n) {
            this->numEdge = 0;
            // initialize vector with empty list
            for (int i=0; i<n; i++) {
                list<pair<int, int> > v; // create an empty list of <int, int> pair type
                graph.push_back(v);
            }
        }
        // returns number of vertices/nodes
        size_t nodeCount() {
            return nodes.size();
        }

        // returns the current number of edges
        size_t edgeCount() {
            return numEdge;
        }

        // adds a new edge from node u to node v, with weight w
        void addEdge(int u, int v, int w=1) {
            if (w == 0) return;
            graph[u].push_back({v, w});
            ++numEdge;
            nodes.insert(u);
            nodes.insert(v);
        }

        // returns the weight of an edge, u->v
        int weight(int u, int v) {
            for(auto p: graph[u]) {
                if (p.first == v)
                    return p.second;
            }
            return 0;
        }

        // removes an edge from the graph
        void removeEdge(int u, int v) {
            // p.first is neibhbor id and p.second is weight
            for(auto p: graph[u]) {
                if (p.first == v) {
                    graph[u].remove(p);
                    --numEdge;
                    break;
                }
            }
```

```
        }

        // checks if there's an edge between u and v
        bool hasEdge(int u, int v) {
            for(auto p: graph[u]) {
                if (p.first == v)
                    return true;
            }
            return false;
        }

        // returns vector containing neighbors of u
        vector<int> neighbors(int u) {
            vector<int> neighs;
            for(auto p: graph[u]) {
                neighs.push_back(p.first);
            }
            return neighs;
        }

        // print graph
        void print() {
            for(int i=0; i<graph.size(); i++) {
                cout << i << " -> ";
                for (auto p: graph[i])
                    cout << p.first << " ";
                cout << endl;
            }
        }
};
```

```
input_line_24:1:7: error: redefinition of 'GraphL'
class GraphL {
      ^

input_line_9:1:7: note: previous definition is here
class GraphL {
      ^
```

```
Interpreter Error:
```

1.3.1 Let's test GraphL ADT using the following undirected graph

[3]:
```
// represent undirected graph shown in above diagram
GraphL graph1(5);
```

[4]:
```
graph1.addEdge(0, 1);
graph1.addEdge(0, 4);
graph1.addEdge(1, 0);
graph1.addEdge(1, 4);
graph1.addEdge(1, 3);
graph1.addEdge(2, 4);
graph1.addEdge(2, 3);
graph1.addEdge(3, 1);
graph1.addEdge(3, 2);
graph1.addEdge(4, 0);
graph1.addEdge(4, 1);
graph1.addEdge(4, 2);
```

[5]:
```
graph1.print();
```

```
0 -> 1 4
1 -> 0 4 3
2 -> 4 3
3 -> 1 2
4 -> 0 1 2
```

[6]:
```
cout << "neighbors of 4 are: ";
for(auto n : graph1.neighbors(4))
    cout << n << " ";
```

```
neighbors of 4 are: 0 1 2
```

[7]:
```
cout << boolalpha << "is 1 connected to 3? " << graph1.hasEdge(1, 3);
```

```
is 1 connected to 3? true
```

[8]:
```
cout << " weight of edge between 1 and 3 = " << graph1.weight(1, 3);
```

```
 weight of edge between 1 and 3 = 1
```

[9]:
```
// remove edge 1->3
graph1.removeEdge(1, 3);
cout << boolalpha << "is 1 connected to 3? " << graph1.hasEdge(1, 3);
```

```
is 1 connected to 3? false
```

1.4 Graph Traversals

https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/GraphTraversal.html - many graph applications need to visit the vertices of a graph in some specific order - graph traversal algorithms begin with a start vertex and attempt to visit the remaining vertices from there at least (at most) once - must deal with two main troublesome cases: 1. all the vertices may not be reachable from given start vertex - occurs when graph is not fully connected 2. graph might contain cycles (avoid infinite loop) - a simple solution to avoid these cases is using a VISITED flag for each vertex that is visited

1.5 Depth-First Search (DFS)

- whenever a vertex, v is visited, DFS will recursively visit all of v's unvisited neighbors
- use stack to push all the neighbors (leading out of v) (iterative version)
  - effectively follow one branch through the graph to its conclusion
  - back up and follow another branch, and so on…
- can be applied to both directed and undirected graphs

1.5.1 visualize DFS: https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/GraphTraversal.ht first-search

1.6 DFS Implementation (iterative version using stack)

- implemented as a function
- can also be a method in Graph ADT

[10]:
```
#include <iostream>
#include <stack>
#include <vector>
#include <algorithm>

using namespace std;
```

[15]:
```
// iterative version using stack
template<class T>
void DFS(T &G, int start) {
    vector<bool> visited(G.nodeCount(), false); // boolean vector to keep track␣
 ↪of visited nodes
    stack<int> vertices;
    vertices.push(start);
    while(!vertices.empty()) {
        int u = vertices.top(); // get the top of the stack
        // pop the stack
        vertices.pop();
        // mark node as visited
        // do something with the data if not already visited
        // stack may contain same vertex twice; print the item if it's not␣
 ↪already visited
        if (!visited[u]) {
```

```
            cout << " " << u;
            visited[u] = true;
        }
        // add all the unvisited neighbors (adjacent vertices) of u to the stack
        for(auto v: G.neighbors(u)) {
            if (!visited[v])
                vertices.push(v);
        }
    }
}
```

[20]:
```
// recusive version
template<class T>
void DFSRec(T &G, vector<bool> & visited, int start) {
    visited[start] = true;
    cout << start << " ";
    vector<int> neighbors = G.neighbors(start);
    for (int i=0; i<neighbors.size(); i++) {
        int v = neighbors[i];
        if (!visited[v])
            DFSRec(G, visited, v);
    }
    // post visit
}
```

1.6.1 Let's test DFS with the following directed graph

[16]:
```
// Work with directed graph shown in above figure
GraphM graph2(5);
```

[17]:
```
graph2.addEdge(0, 4);
graph2.addEdge(0, 1);
graph2.addEdge(1, 3);
graph2.addEdge(2, 4);
graph2.addEdge(3, 2);
graph2.addEdge(4, 1);
```

[19]:
```
// depth first search on graph2
DFS<GraphM>(graph2, 4);
```

```
 4 1 3 2
```

1.6.2 Let's test DFS and DFSRec functions with the following undirected graph

[21]:
```
// represent undirected graph shown in above diagram
GraphL graph3(5);
```

[22]:
```
graph3.addEdge(0, 1);
graph3.addEdge(0, 4);
graph3.addEdge(1, 0);
graph3.addEdge(1, 4);
graph3.addEdge(1, 3);
graph3.addEdge(2, 4);
graph3.addEdge(2, 3);
graph3.addEdge(3, 1);
graph3.addEdge(3, 2);
graph3.addEdge(4, 0);
graph3.addEdge(4, 1);
graph3.addEdge(4, 2);
```

[25]:
```
// Depth first search on graph3
DFS<GraphL>(graph3, 0);
```

```
 0 4 2 3 1
```

[28]:
```
// previsit if necessary
// boolean vector to keep track of visited nodes
vector<bool> visited1(graph3.nodeCount(), false);
// Depth-first search recursive
```

[ ]:
```
vistited.
```

[29]:
```
DFSRec<GraphL>(graph3, visited1, 4);
```

```
4 0 1 3 2
```

1.7 Breadth-First Search (BFS)

- BFS visits all the neighbors connected to current (start) vertex before visiting vertices further away
- similar to DFS implementation wise, but uses **queue** instead of stack
- NOTE: if the graph is a tree and the start vertex is the root, BFS is equivalent to visiting vertices level by level from top to bottom

1.7.1 visualize BFS: https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/GraphTraversal.ht first-search

2 BFS Implementation using queue

- implemented as a function

- can also be a method in Graph ADT

[31]:
```
#include <iostream>
#include <queue>
#include <vector>
#include <algorithm>

using namespace std;
```

[32]:
```
// BFS can also be a method in Graph ADT
// Also, keep track of parent node of each node to print the edges in BFS Tree
template<class T>
void BFS(T &G, int start, vector<int> &parent) {
    vector<bool> visited(G.nodeCount(), false);
    queue<int> q;
    q.push(start); // push start vertex into the queue

    while(!q.empty()) {
        // get the front of the queue
        int u = q.front();
        // remove the front element
        q.pop();
        // visit and mark the node visited
        cout << " " << u;
        visited[u] = true;

        // add neighbors of u to the queue
        for(auto v: G.neighbors(u)) {
            if (!visited[v]) {
                q.push(v);
                // prevent from duplicate nodes being pushed to the queue
                visited[v] = true;
                // v is visited from u
                parent[v] = u;
            }
        }
    }
}
```

2.0.1 Let's test BFS on the following directed graph

[33]:
```
// Work with directed graph shown in above figure
// parent vector to keep track of parent for each node
// initialized with -1
vector<int> parent(graph2.nodeCount(), -1);
BFS<GraphM>(graph2, 0, parent); // prints the node as they're visited using BFS
```

```
0 1 4 3 2
```

[34]:
```
// let's check the content of parent vector
parent
```

[34]:
```
{ -1, 0, 3, 1, 0 }
```

[35]:
```
// print the edges in BFS tree using parent vector
for (int v=0; v<parent.size(); v++) {
    if (parent[v] == -1) continue;
    cout << parent[v] << "->" << v << endl;
}
```

```
0->1
3->2
1->3
0->4
```

2.0.2 Let's test BFS on the following undirected graph

[36]:
```
// Work with undirected graph shown in above figure
vector<int> parent_g3(graph3.nodeCount(), -1);
BFS(graph3, 0, parent_g3);
```

```
0 1 4 3 2
```

[37]:
```
// print the eddges in BFS tree using parent vector
for (int v=0; v<parent_g3.size(); v++) {
    if (parent_g3[v] == -1) continue;
    cout << parent_g3[v] << "->" << v << endl;
}
```

```
0->1
4->2
1->3
0->4
```

2.1 Exercises

1. Where's My Internet?? - https://open.kattis.com/problems/wheresmyinternet
    - DFS on undirected graph
    - check graph connectedness

- Erdos Numbers - https://open.kattis.com/problems/erdosnumbers
    - BFS on undirected graph
- Running MoM - https://open.kattis.com/problems/runningmom
    - finding cycle in a DAG

[ ]:
```

```