CSCI-2270 Data Structures Recitation 10// Instructors: Hoenigman/Zagrodzki/Zietz

Learning Objectives:

- Graphs
- Graphs representation
- Vectors

1 Graph

1.1 Definition

A graph G consists of two sets, V and E.

- V is non-empty set of vertices.
- E is a set of pairs of vertices. These pairs are called edges.

$V(G)$ and $E(G)$ will represent set of vertices and set of edges respectively of graph G.

1.2 Directed Graph and Undirected Graph

Depending on a particular aspect we can classify graph into classes and there are many such classes. Here we will discuss two basic types of the graph.

- **Undirected Graph**- In an undirected graph G for an edge $e \in E(G)$ the ordering of vertices does not matter. Remember an edge is defined mainly by a pair of vertices. So in an undirected graph if there is an edge between u and v $(u, v \in V(G))$ we can represent it as (u, v) or (v, u). In other words an edge between u and v in an undirected graph implies $(u, v) \in E(G)$ and $(v, u) \in E(G)$. Refer to Fig 1 of figure 1.
- **Directed Graph**- In directed graph each edge represented by a directed pair $< u, v >$ where u is the source and v is the destination. Note presence of $< u, v >$ does not imply presence of $< v, u >$. Refer to Fig 2 of figure 1.

Edges can be weighted as well as unweighted. Edges in both graphs in figure 1 are unweighted. Figure 2 depicted an image of undirected graph with weighted edges.

2 Graph Representation

To store a graph in memory we can use following representation-

2.1 Adjacency Matrix

In this representation, a graph is stored as a matrix (say A). $A[i][j]$ denotes the presence of edge between vertex i and j. Adjacency matrix for undirected graph(Fig 1) in figure 1 is given in table 1. Similarly, adjacency matrix for directed graph(Fig 2) of figure 1 is given in table 2.

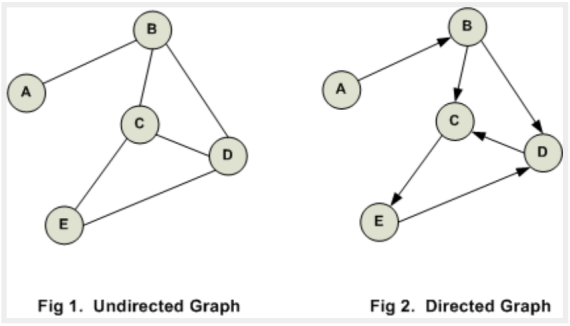

Figure 1: Undirected graph and directed graph

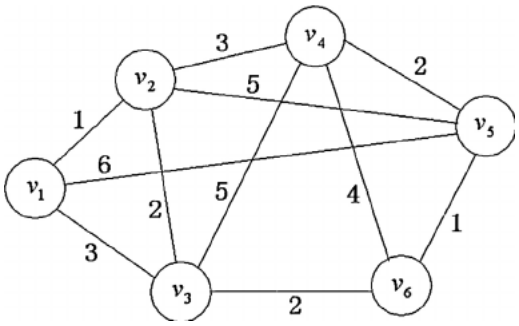

Figure 2: Undriected graph with weighted edges

Table 1: Adjacency matrix for undriected graph of fig 1

	A	**B**	**C**	**D**	**E**
A	0	1	0	0	0
B	1	0	1	1	0
C	0	1	0	1	1
D	0	1	1	0	1
E	0	0	1	1	0

Table 2: Adjacency matrix for directed graph of fig 1

	A	**B**	**C**	**D**	**E**
A	0	1	0	0	0
B	0	0	1	1	0
C	0	0	0	0	1
D	0	0	1	0	0
E	0	0	0	1	0

2.2 Adjacency List

One problem with adjacency matrix is if the graph has many vertices but few edges the matrix will be sparse. There will be many zeros but very few ones. Hence it is helpful to use adjacency list. In this representation each vertex has a list of adjacent vertices. For example figure 3 shows a graph and its adjacency list representation.

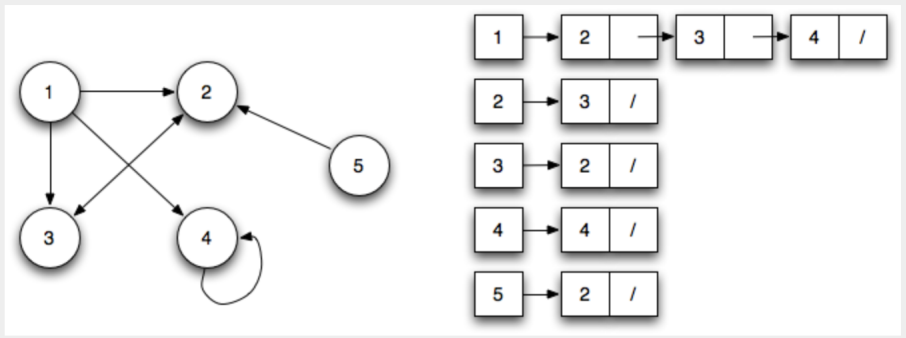

Figure 3: Adjacency List

3 Vectors

c++ standard template library (STL) has implementations for many useful data-structures like vectors, lists, stacks and queues.

As per c++ documentation (http://www.cplusplus.com/reference/vector/vector/) Vectors are sequence containers representing arrays that can change size. So just like arrays element of vectors can be accesses using offset index.

Vectors internally use a dynamically allocated array. However this array may need re-sizing. Vectors handle those memory operation efficiently. Actually vectors may allocate some extra storage to accommodate for possible growth, and thus the container may have an actual capacity greater than the storage strictly needed to contain its elements (i.e., its size).

Therefore, compared to arrays, vectors consume more memory in exchange for the ability to manage storage and grow dynamically in an efficient way.

Here is an example code using vector. To see other functions of vectors refer to c++ documentation (http://www.cplusplus.com/reference/vector/vector/)

```
#include <iostream>
#include <vector>
using namespace std;

int main() {

   // create a vector to store int
   vector<int> vec;
   int i;

   // display the original size of vec
   cout << "vector size = " << vec.size() << endl;

   // push 5 values into the vector
   for(i = 0; i < 5; i++) {
      vec.push_back(i);
   }

   // display extended size of vec
   cout << "extended vector size = " << vec.size() << endl;

   // access 5 values from the vector
   for(i = 0; i < 5; i++) {
      cout << "value of vec [" << i << "] = " << vec[i] << endl;
   }

   // use iterator to access the values
   vector<int>::iterator v = vec.begin();
   while( v != vec.end()) {
      cout << "value of v = " << *v << endl;
```

```
31          v++;
32      }
33      return 0;
34  }
```

Output is given as

```
vector size = 0
extended vector size = 5
value of vec [0] = 0
value of vec [1] = 1
value of vec [2] = 2
value of vec [3] = 3
value of vec [4] = 4
value of v = 0
value of v = 1
value of v = 2
value of v = 3
value of v = 4
```

4 Exercise

Download the starter code and the text file. The text file contains an adjacency matrix as given in table 3. The first column represents the source city names and the first rows represent the destination city names. Content in the a cell represent the weighted edge between a source to a destination. For example there is an edge from Denver to New Mexico with edge weight 170 (refer to A[Denver][NewMexico]).

destination/ source	Boulder	Denver	NewMexico	Texas
Boulder	0	30	200	500
Denver	30	0	170	400
NewMexico	200	170	0	50
Texas	500	400	50	0

Table 3: Adjacency table for assignment

This table is given in the mat.txt.

- each line is a row in the text file.
- columns are separated by single space only.

Starter code contains class implementation of a graph. It has functionalities such as addEdge and addVertex. You need to write a driver method (main function) that will read this file and call those functions to create a graph.