Programming Assignment 9

Digraphs and Dijkstra

[Approved Includes](#_15zx66kj8irm)

[Code Coverage](#_776g0839yxxo)

[Starter Code](#_jmrk4oe3zs75)

[Files to Submit](#_van3nv2hgzl6)

[Task 1: Directed Graph](#_aihti315r2yk)

[Requirements](#_aq2yfnijzs82)

[Files](#_6ux7s0e0v2mo)

[Class](#_1s6hgm3drobu)

[Functions](#_ck5c10n975l3)

[Constructors](#_z4rylwnvbt7)

[Capacity](#_7166shqekiew)

[Element Access](#_c71r54fnwtkp)

[Modifiers](#_d63i5m986ypz)

[Optional](#_4jbjrvyvf9md)

[Task 2: Dijkstra](#_hutw5taun68d)

[Requirements](#_s8k1975o5yba)

[Files](#_gqavpjci9cp2)

[Functions](#_hir84phig07p)

[Visualization](#_fdldi9n4q447)

[Example (for Tasks 1 and 2)](#_6wn0fne2vc5a)

[Example Output](#_cjrt1ie7a9pj)

[Notes](#_6p2rkv48b1mg)

[Graph notation format](#_8whfqehfr3tg)

[Testing Advice](#_5ez18t76xyn3)

# Approved Includes

<cassert>

<cmath>

<cstddef>

<iostream>

<list>

<sstream>

<stack>

<queue>

<unordered\_map>

<unordered\_set>

<vector>

"graph.h"

# Code Coverage

You must submit a test suite for each task that, when run, covers at least 90% of your code. You should, at a minimum, invoke every function at least once. **Best practice is to also check the actual behavior against the expected behavior, e.g. verify that the result is correct. You should be able to do this automatically, i.e. write a program that checks the actual behavior against the expected behavior.**

Your test suite should include ALL tests that you wrote and used, including tests you used for debugging. You should have MANY tests.

# Starter Code

graph.h

graph\_compile\_test.cpp

graph\_tests.cpp

Makefile

## Files to Submit

graph.h

graph\_tests.cpp

# Task 1: Directed Graph

Implement a data structure to store a **directed** graph.

## Requirements

### Files

graph.h - contains the Graph class definition (define the methods **inside** the class)

graph\_tests.cpp - contains the test cases and test driver (main)

### Class

class Graph;

You can represent the Graph internally however you want. This could be adjacency lists, an adjacency matrix, sets of Vertex and Edge objects, linked Vertex and/or Edge objects, or even some combination of methods. In class, we learned the adjacency list and adjacency matrix representations, so I encourage you to use those.

**Performance Matters.**

Advice: You want the speed of a matrix, but the space of a list. How can you get fast access with minimal space?

### Functions

#### Constructors

**Graph()** - makes an empty graph.

**Graph(const Graph&)** - constructs a deep copy of a graph

**Graph& operator=(const Graph&)** - assigns a deep copy of a graph

**~Graph()** - destructs a graph (frees all dynamically allocated memory)

#### Capacity

**size\_t vertex\_count() const** - the number of vertices in the graph

**size\_t edge\_count() const** - the number of edges in the graph

#### Element Access

**bool contains\_vertex(size\_t id) const** - return true if the graph contains a vertex with the specified identifier, false otherwise.

**bool contains\_edge(size\_t src, size\_t dest) const** - return true if the graph contains an edge with the specified members (as identifiers), false otherwise.

**double cost(size\_t src, size\_t dest) const** - returns the weight of the edge between src and dest, or [INFINITY](https://en.cppreference.com/w/c/numeric/math/INFINITY) if none exists.

#### Modifiers

**bool add\_vertex(size\_t id)** - add a vertex with the specified identifier if it does not already exist, return true on success or false otherwise.

**bool add\_edge(size\_t src, size\_t dest, double weight=1)** - add a **directed** edge from src to dest with the specified weight if there is no edge from src to dest, return true on success, false otherwise.

**bool remove\_vertex(size\_t id)** - remove the specified vertex from the graph, including all edges of which it is a member, return true on success, false otherwise.

**bool remove\_edge(size\_t src, size\_t dest)** - remove the specified edge from the graph, but do not remove the vertices, return true on success, false otherwise.

#### Optional

**Graph(Graph&&)** - move constructs a deep copy of a graph

**Graph& operator=(Graph&&)** - move assigns a deep copy of a graph

# Task 2: Dijkstra’s Algorithm

Implement Dijkstra’s Algorithm as a method of the Graph class from Task 1.

## Requirements

### Files

graph.h - contains the Graph class definition (define the methods **inside** the class)

graph\_tests.cpp - contains the test cases and test driver (main)

### Functions

**void dijkstra(size\_t source\_id)** - compute the shortest path from the specified source vertex to all other vertices in the graph using Dijkstra’s algorithm.

**double distance(size\_t id) const** - assumes Dijkstra has been run, returns the cost of the shortest path from the Dijkstra-source vertex to the specified destination vertex, or INFINITY if the vertex or path does not exist.

#### Visualization

**void print\_shortest\_path(size\_t dest\_id, std::ostream& os=std::cout) const** - assumes Dijkstra has been run, pretty prints the shortest path from the Dijkstra source vertex to the specified destination vertex in a “ → “- separated list with “ distance: #####” at the end, where <distance> is the minimum cost of a path from source to destination, or prints “<no path>\n” if the vertex is unreachable.

## 

## Example (for Tasks 1 and 2)

std::cout << "make an empty digraph" << std::endl;

Graph G;

std::cout << "add vertices" << std::endl;

for (size\_t n = 1; n <= 7; n++) {

G.add\_vertex(n);

}

std::cout << "add directed edges" << std::endl;

G.add\_edge(1,2,5); // 1 ->{5} 2; (edge from 1 to 2 with weight 5)

G.add\_edge(1,3,3);

G.add\_edge(2,3,2);

G.add\_edge(2,5,3);

G.add\_edge(2,7,1);

G.add\_edge(3,4,7);

G.add\_edge(3,5,7);

G.add\_edge(4,1,2);

G.add\_edge(4,6,6);

G.add\_edge(5,4,2);

G.add\_edge(5,6,1);

G.add\_edge(7,5,1);

std::cout << "G has " << G.vertex\_count() << " vertices" << std::endl;

std::cout << "G has " << G.edge\_count() << " edges" << std::endl;

std::cout << "compute shortest path from 2" <<std::endl;

G.dijkstra(2);

std::cout << "print shortest paths" <<std::endl;

for (size\_t n = 1; n <= 7; n++) {

std::cout << "shortest path from 2 to " << n << std::endl;

std::cout << " ";

G.print\_shortest\_path(n);

}

### 

### Example Output

make an empty graph

add vertices

add edges

G has 7 vertices

G has 12 edges

compute shortest path from 2

print shortest paths

shortest path from 2 to 1

2 --> 7 --> 5 --> 4 --> 1 distance: 6

shortest path from 2 to 2

2 distance: 0

shortest path from 2 to 3

2 --> 3 distance: 2

shortest path from 2 to 4

2 --> 7 --> 5 --> 4 distance: 4

shortest path from 2 to 5

2 --> 7 --> 5 distance: 2

shortest path from 2 to 6

2 --> 7 --> 5 --> 6 distance: 3

shortest path from 2 to 7

2 --> 7 distance: 1

# 

# Notes

## Graph notation format

<source\_vertex\_id> ->[{<cost>}] <destination\_vertex\_id>;

Examples:

* 1 ->{1} 2
  + “Vertex 1 has an edge to vertex 2 with cost 1”
* 3 -> 4
  + “Vertex 3 has an unweighted edge to vertex 4”

## Testing Advice

1. Write tests before you write implementation.
2. Write more tests.
3. Don’t only add all the vertices all at once at the beginning.
   1. Test adding vertices in random orders and interleaved with adding edges.