Programming Assignment 10

Minimum Spanning Tree (Prim’s Algorithm)

[Approved Includes](#_rdjdg7wwpb9y)

[Code Coverage](#_776g0839yxxo)

[Starter Code](#_jmrk4oe3zs75)

[Files to Submit](#_van3nv2hgzl6)

[Task 1](#_aihti315r2yk)

[Requirements](#_aq2yfnijzs82)

[Files](#_6ux7s0e0v2mo)

[Class](#_1s6hgm3drobu)

[Functions](#_ck5c10n975l3)

[Constructors](#_z4rylwnvbt7)

[Capacity](#_7166shqekiew)

[Element Access](#_c71r54fnwtkp)

[Modifiers](#_d63i5m986ypz)

[Optional](#_4jbjrvyvf9md)

[Task 2](#_hutw5taun68d)

[Requirements](#_s8k1975o5yba)

[Files](#_gqavpjci9cp2)

[Functions](#_hir84phig07p)

[Visualization (optional)](#_fdldi9n4q447)

[Example (for Tasks 1 and 2)](#_191rz7rtzwe2)

[Example Output](#_aopmh91186r2)

# 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: Undirected Graph

Implement a data structure to store an **undirected** graph. You should modify your digraph code.

## 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 would encourage you to use one of those.

### 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 u, size\_t v) const** - return true if the graph contains an edge with the specified members (as identifiers), false otherwise.

**double cost(size\_t u, size\_t v) const** - returns the weight of the edge between u and v, 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 u, size\_t v, double weight=1)** - add an **undirected** edge between u and v with the specified weight if there is not one already, return true on success, false otherwise. If you use adjacency lists, make sure to update both u’s and v’s lists.

**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 u, size\_t v)** - 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: Prim’s Algorithm

Implement Prim’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

**std::list<std::pair<size\_t,size\_t>> prim()** - compute a minimum spanning tree using Prim’s algorithm. Return a list of edges. There may be more than one possible spanning tree depending on the starting vertex. Any correct (minimum weight) tree will be recognized as such. If a MST does not exist, then the return value should be an empty list.

**double distance(size\_t id) const** - assumes Prim has been run, returns the cost of the edge that connects this vertex to the minimum spanning tree, or 0 is the vertex is the root of the tree, or INFINITY if the vertex is not part of the tree (i.e. the graph is not connected).

#### Visualization (optional)

**void print\_minimum\_spanning\_tree(std::ostream& os=std::cout) const** - assumes Prim has been run, pretty prints the minimum spanning tree as a sequences of lines with the format <vertex> --{<edge weight>} <vertex>

## 

## Example (for Tasks 1 and 2)

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

Graph G;

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

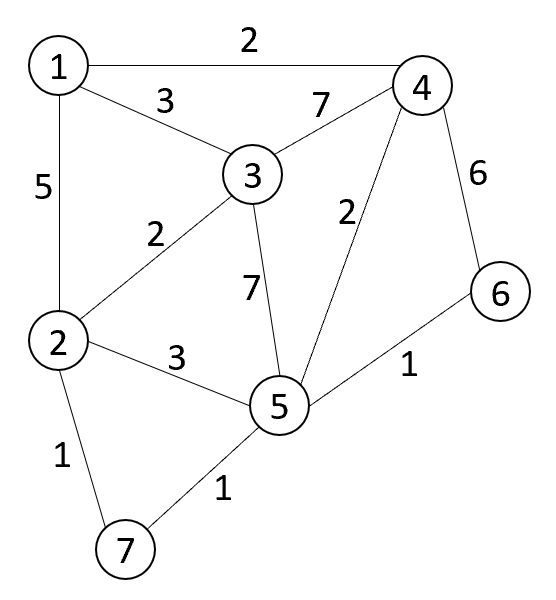
for (size\_t n = 1; n <= 7; n++) {

G.add\_vertex(n);

}

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

G.add\_edge(1,2,5); // 1 --{5} 2; (edge between 1 and 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 and ";

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

std::cout << "compute a minimum spanning tree" <<std::endl;

std::list<std::pair<size\_t,size\_t>> = G.prim();

std::cout << "print minimum spanning tree" <<std::endl;

double tree\_cost = 0;

for (const std::pair<size\_t,size\_t>& edge : mst) {

std::cout << edge.first << " --{"<<G.cost(edge.first,edge.second)<<"} " << edge.second << ";" << std::endl;

tree\_cost += G.cost(edge.first,edge.second);

}

std::cout << "tree cost = " << tree\_cost <<std::endl;

### 

### Example Output

make an empty graph

add vertices

add undirected edges

G has 7 vertices and 12 edges

compute a minimum spanning tree

print minimum spanning tree

2 --{1} 7;

5 --{1} 7;

6 --{1} 5;

3 --{2} 2;

4 --{2} 5;

1 --{2} 4;

tree cost = 9

