COMP2123 SELF-LEARNING REPORT
PRIM'S ALGORITHM

MA Rutian 3035234791
MAO Wenxu 3035243235
YAO Qingning 3035238644

SOLVING THE MINIMUM-SPANNING-TREE PROBLEM WITH PRIM'S ALGORITHM

Objectives

Today there's gonna be something exciting to learn, and at the end of day you shall be able to:

- Understand what is the *Minimum-Spanning-Tree Problem*
- Know what is *Prim's Algorithm* and how to implement it
- Understand how Prim's Algorithm could be used to solve the minimum-spanning-tree problem

Motivations

"So what is a minimum-spanning-tree problem anyways?" Great question! In simple words, it is about finding a set of shortest possible edges of a *connected*, *edge-weighted* and *undirected* graph that connects all the vertices together without cycles.

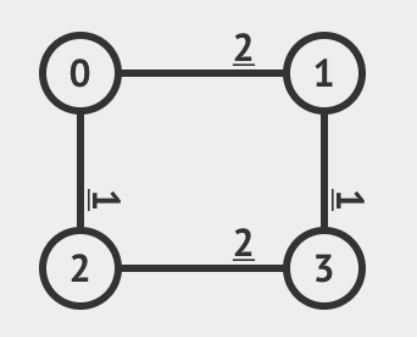
Wow that's quite a lot of information and really not that simple I admit, but if you take a closer look it's actually pretty easy to understand and logical:

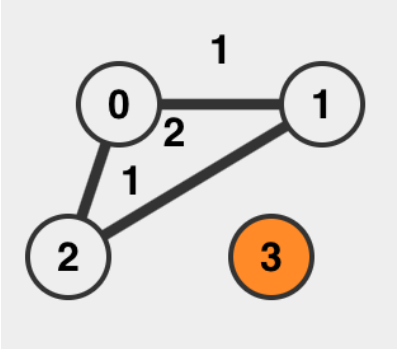
First, What Does These Fancy Words Mean?

1. "Connected"

Recall from what you have learnt in COMP2121😀, a connected graph is a graph in which you can find a path between **any** two vertices.

EXAMPLES





On the left is an example of connected graph since you can find a path between any two vertices.** is an example of connected graph since you can find a path between any two vertices.

On the right is **not** a connected graph as *3* is not connected to other parts of the graph. For instance, there is no path between *3* and *2*.

2. "Edge-Weighted"

Still, from COMP2121, edge-weighted means that there is a numerical value for the weight of each edge, you can think about this as the length or the cost of the edge.

3. "Undirected"

Once again from COMP2121, all edges in the graph indicate a two-way relationship which means they can be traversed in two directions.

Great! Now What's a Minimum-Spanning-Tree?

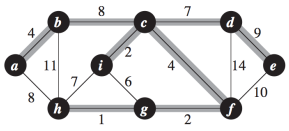
What is a tree?

Also a discrete math concept, basically, a tree is a connected graph with no circle.

What is a Spanning-Tree in a graph?

We call a tree a Spanning-Tree of graph `g` if and only if the tree spans (in other words includes) all the vertices in that graph.

What is a Minimum-Spanning-Tree in a graph?



A Minimum-Spanning-Tree is a spanning tree with ***minimum*** total weight among all spanning tree. **This is what we are looking for!**😃

And that's it!

"But why do we need want to find that though?" Minimum-Spanning-Tree problem has many applications in the design of *networks*, and a real life example would be laying cable to a new neighborhood and we would want to minimize our cost.

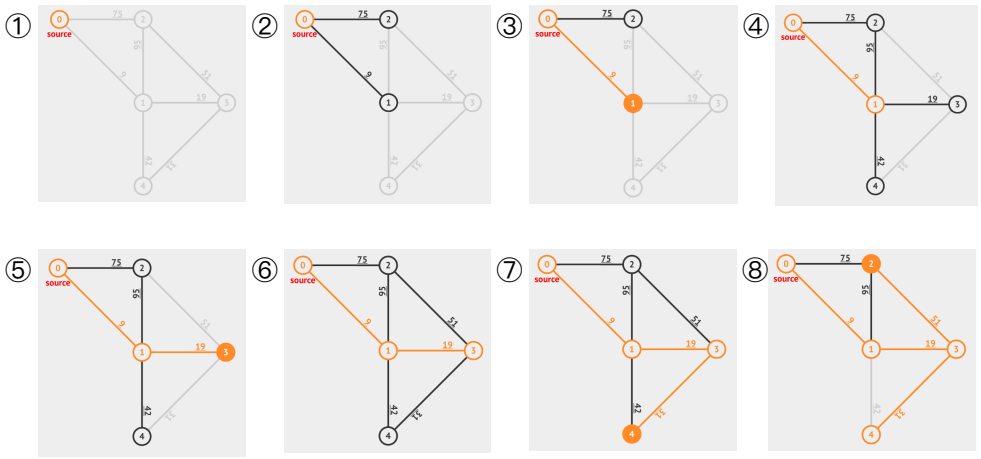
"Hmm... That sounds interesting, so how do we find it?" That what we are going to learn today! Let welcome our special guest today - **Prim's Algorithm**!

What is Prim's Algorithm About?

In simple words, Prim's Algorithm can help us find the Minimum-Spanning-Tree we mentioned earlier.

The algorithm may be described in following steps:

1. Initialize a tree with an arbitrary node in the graph (①)
2. Find a node not yet in the tree that can be connected to existing tree with the minimum weight edge and add it to the tree. (②③)
3. Repeat step 2 until all the nodes are in the tree



Your Trusty Data Structure - Graph

How can we do this in C++? First we need to have a data structure for modeling a *graph*. As this is not the main focus of today's self learning, we have provided a `Graph.h` file which contains a basic implementation of the a Graph data structure! *It uses* `template` *in implementation for the purpose of a more generic usage, which could make some function calling a little bit different from things you already know.*

In today's session, we will be using `double` vertices, which means each vertex contains a `double` value, and this is for not letting you confuse vertex value with its storage index (which is type `int`) in later stage. We have provided you with a handy `Graph.h` to help you on your journey. You do not need to know the actual implementation details (if you are curious enough you can have a look), all you need to know about is these functions below:

`Graph<double>(false)`

Construct a **undirected graph object** with vertices of type `double`.

`double getVertex(int index)`

Return the **vertex** corresponds to a index.

`int getIndex(double vertex)`

Return the **index** corresponds to a vertex.

`int getNumOfVertex()`

Return the **number** of vertices.

`void addVertex(double vertex)`

Add a **vertex** to the graph.

`void addEdge(double v1, double v2, int w)`

Add an **edge** connecting `v1` to `v2` with weight `w`.

`int getEdge(double v1, double v2)`

Return the **weight** of edge connecting `v1` to `v2`.

`vector<double> getAllAdjacentVertex(double vertex)`

Return a vector of all the vertices adjacent to a vertex.

`bool contain(vertex)`

Return whether a graph contains a vertex.

You are good to go!🙂

And now you are equipped with our weapon and shield, we can finally go on our conquest to find our *Minimum-Spanning-Tree*!

Let's Do It!🤗

- First, you need to open the existing file `Prim.cpp`.

```
gedit Prim.cpp
```

Or whatever text editor that you are comfortable with😜.

Let's implement this algorithm!🤗

- First, let's see the skeleton of our algorithm

```
Graph<double> prim(Graph<double> g, int root){
    //All your code in this implementation find their way here😊
}
```

The return type should be a graph with double as vertices, same as the original graph.

Two arguments here are: 1. the graph we want to find our Minimum-Spanning Tree in and, 2. the arbitrary root that we would want to begin with.

- Now, let's create an empty graph `minimum_spanning_tree` for storing what we have added.

```
Graph<double> minimum_spanning_tree(false);
```

Note: the `false` here is for creating an undirected graph (you can refer to the Graph.h provided for implementation details)

Three Important Arrays / Vectors for Your Quality of Life

- **First**: We need an array (with size of number of vertices) to store each index's **cheapest connection** (the connection that uses lowest weighted edge possible) to the existing tree.
  - *The `i-th` entry in this array stores the lowest cost vertex `i` can be connected to the existing tree.*
  - Think 😆 What value should they be initialized to when all of them are not in the tree?🤔
  - Yes, they should be initialized to the largest `int` possible to indicate we have not been there yet.
  - Now, let's include them to our code.

```
int* cost_of_cheapest_connection_to = new int[g.getNumOfVertex()];

for (int i = 0; i < g.getNumOfVertex(); ++i){
    cost_of_cheapest_connection_to[i] = 0x7fffffff;
}
```

- **Second**: We also need an array to record which vertex this cheapest connection connect to keep track of edges we want to add to the tree.
  - The `i-th` entry of this array will store an `int`, indicating the **index of the vertex** of `i-th` vertex's cheapest connection target.
  - Think about it 😆 How could we denote that we have not explored a vertex in terms of index?🤔
  - Correct! Let's use `-1` (or basically any negative integer) to represent it.
  - Now let's implement this.

```
int* source_of_cheapest_connection_to = new
int[g.getNumOfVertex()];

for (int i = 0; i < g.getNumOfVertex(); ++i){
    source_of_cheapest_connection_to[i] = NOT_CONNECTED;
}
```

- **Third:** we need yet another array to record whether this node is already in our tree (we don't want to add the same vertex over and over again right?)
  - A vector of `bool` can do the job right?
  - And what should their initial value be when no node is in the tree? 🙈
  - The answer is - `false`!
  - Cool, let's do that

```
vector<bool> visited;
visited.resize(g.getNumOfVertex(), false);
```

Or an simple array would do the job (remember to initialize it though).

Keep these three arrays in mind cause they are the keys to this problem!😉

Let the Iteration Begin!

Now, we could start the iteration to build up our tree!

- A while loop would be a good choice in this case 😀.

```
while(minimum_spanning_tree.getNumOfVertex() !=
g.getNumOfVertex()){
    //All the code in the iteration go to here
}
```

Can you tell why the condition we set termination condition as `minimum_spanning_tree.getNumOfVertex() != g.getNumOfVertex()`? 🤔😜

Yeah, I know you can recall that the iteration should not stop until all nodes are added😊

Now, we what we need to do in the iteration can be divided into 3 steps 😃

Step 1 in Iteration: Finding Nearest Unvisited Vertex

We could use two `int` variables to record the cheapest cost and the corresponding index respectively.😀

Now we can add the following code to iterate through the array `cost_of_cheapest_connection_to` to find the cheapest connection.😆

```
int cheapest_vertex_index = -1;
int cheapest_vertex_cost = 0x7fffffff;
for (int i = 0; i < g.getNumOfVertex(); ++i) {
    //a comparison and resulting action need to be added
}
```

In side the for loop, what comparison we need to make?🤔

Recall that we need to find the cheapest yet unvisited vertex, so that's the two criteria in the comparison.💡

```
if (cost_of_cheapest_connection_to[i] <= cheapest_vertex_cost &&
visited[i] ==  false) {
    cheapest_vertex_cost = cost_of_cheapest_connection_to[i];
    cheapest_vertex_index = i;
}
```

After finding that vertex, Guess what comes next?😜

Bingo!

Step 2: Add the Vertex to Our Tree and Set it To be Visited

Now it is time to scroll back to where functions in `Graph.h` is introduced cause we need to use it now. Take your time~ I will be waiting here🤗

Simple, isn't it? Ready for the code?😀

```
minimum_spanning_tree.addVertex(g.getVertex(cheapest_vertex_index)
);
if (root != cheapest_vertex_index) {

minimum_spanning_tree.addEdge(g.getVertex(cheapest_vertex_index),g.
getVertex(source_of_cheapest_connection_to[cheapest_vertex_index]),
cheapest_vertex_cost);
}
```

Note: we have to perform a checking before adding an edge because the first node we add to the tree does not have anything to connect to.

Then, just set the corresponding entry in the visited vector to be `true` (to represent we have visited this vertex already).

```
visited[cheapest_vertex_index] = true;
```

Hang in there! You are almost there!💪🏽

Step 3: Update `cost_of_cheapest_connection_to` and `source_of_cheapest_connection_to` after a New Vertex is Added to the Tree

This step is a little bit more complicated, but I believe we can make it together🙂

In this step, basically what we need to do is to update two arrays. Recall that: `cost_of_cheapest_connection_to` stores the cheapest cost to connect to the tree and `source_of_cheapest_connection_to` stores the corresponding vertex to connect to.

Then, after we add a vertex to the tree, some of these values might change as the newly added vertex could provide a **cheaper** connection (or make it possible for some vertex to connection).

How do we update these values you might ask? Great question! Let's find out together!😀

As vertices that could be affected are those **connected to the newly added vertex**, we can examine them one by one and update if necessary.

LET'S GIVE IT A TRY!

First, we can get all the adjacent vertices using method from Graph.h

```
vector<double> all_adjacencies =
g.getAllAdjacentVertex(g.getVertex(cheapest_vertex_index));
```

This gives a vector of all vertices adjacent to the newly added one.

Second, let's iterate through these vertices to see if update is necessary.

```
for (vector<double>::iterator it = all_adjacencies.begin(); it !=
all_adjacencies.end(); it++) {

    //code for checking and updating need to be added

}
```

- ① Remember we should only check unvisited vertices

```
if (visited[g.getIndex(*it)] == false && g.getIndex(*it) !=
cheapest_vertex_index) {
    //code in following step should be added here
}
```

- ② If the vertex satisfies condition above, we can move on to check if it **does** have a cheaper connection and update the two arrays if it's really the case.

Get the weight of edge connecting this vertex and the newly added one.

```
int cost = g.getEdge(g.getVertex(cheapest_vertex_index), *it);
```

If `cost` is less than its original cheapest cost, update two corresponding values in the arrays.

```
if (cost <= cost_of_cheapest_connection_to[g.getIndex(*it)]) {
    cost_of_cheapest_connection_to[g.getIndex(*it)] = cost;
    source_of_cheapest_connection_to[g.getIndex(*it)] =
cheapest_vertex_index;
}
```

Well, so many things happen in this step! I know right? Don't worry, let's check if you got everything correct. Here are what the code in **Step 3** should look like🙂

```
vector<double> all_adjacencies =
g.getAllAdjacentVertex(g.getVertex(cheapest_vertex_index));
for (vector<double>::iterator it = all_adjacencies.begin(); it !=
all_adjacencies.end(); it++) {
    if (visited[g.getIndex(*it)] == false && g.getIndex(*it) !=
cheapest_vertex_index)
    {
        int cost = g.getEdge(g.getVertex(cheapest_vertex_index),
*it);
        if (cost <=
cost_of_cheapest_connection_to[g.getIndex(*it)])
        {
            cost_of_cheapest_connection_to[g.getIndex(*it)] = cost;
            source_of_cheapest_connection_to[g.getIndex(*it)] =
cheapest_vertex_index;
        }
    }
}
```

CONGRATULATIONS ! THAT'S EVERYTHING ABOUT THE WHILE LOOP

We can return `minimum_spanning_tree` after the while loop and the implementation is done!

Buuuuuut, 😌 one more thing to consider... How could we determine the first node to be added to be `root` as we promised in the declaration? This is a tricky question 🤔

Think about the how we choose the vertex to add in the first step in while loop. Do you find something?😊

💡 Cool, the answer is to set value corresponds to `root` in `cost_of_cheapest_connection_to` to be `0`. Then in the first iteration of while loop, it will be recognized as the cheapest connection and be added to the tree!

What you need to do is simply to add

```
cost_of_cheapest_connection_to[root] = 0;
```

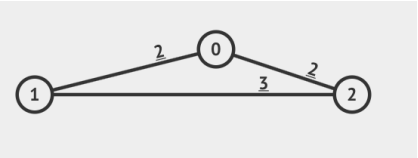
Before the while loop.

GREAT! CONGRATULATIONS! YOU HAVE FINISHED IMPLEMENTING THE ALGORITHM!

Can't wait to try if it works? We have provided the main method for you to test your code!☺️ Now you should be able to use it if everything goes smoothly 🙂

We have prepared some test cases for you!

1.case0



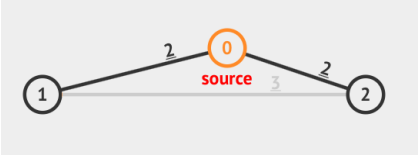
😊You can run this test case by type in:

```
make run_case0
```

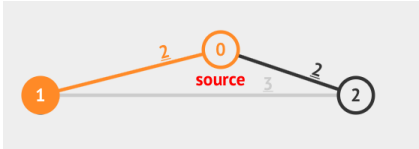
The process of your algorithm should look like this if correct🙂

①

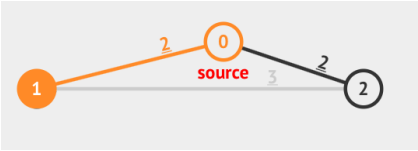
②



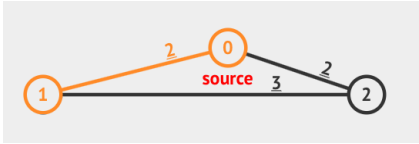
③



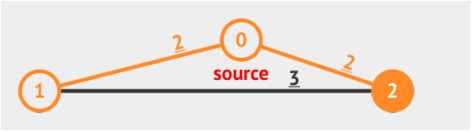
④



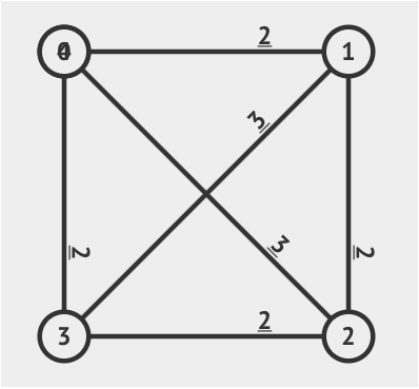
⑤



⑥



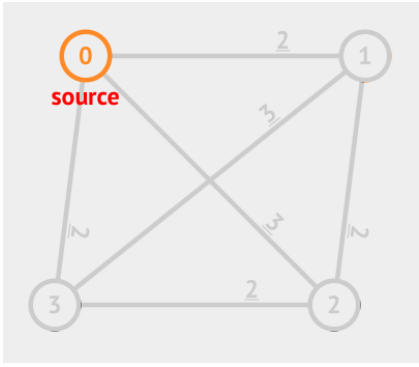
2.case1



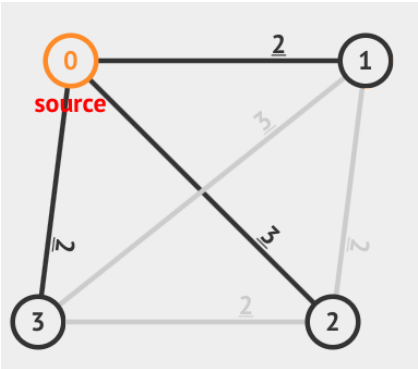
😊You can run this test case by type in:

```
make run_case1
```

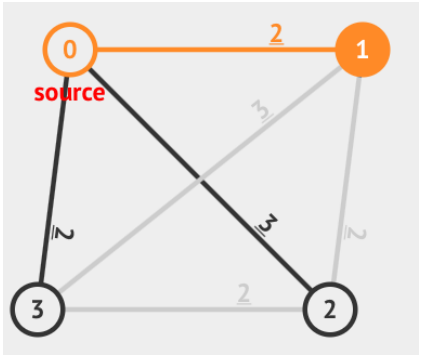
①



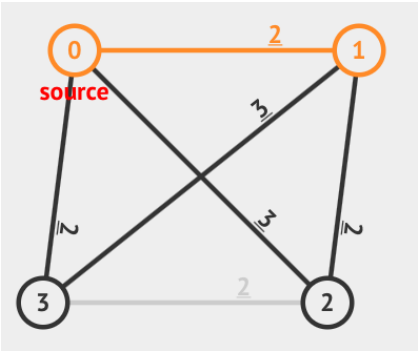
②



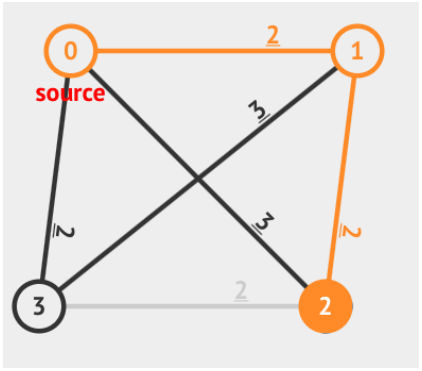
③



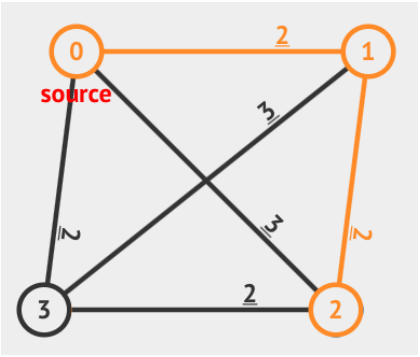
④



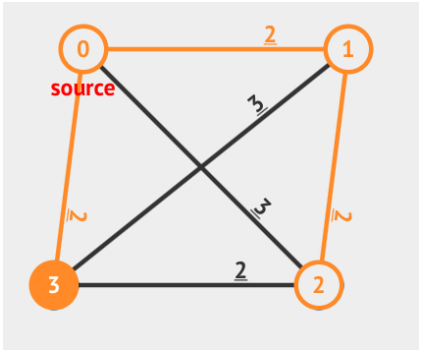
⑤



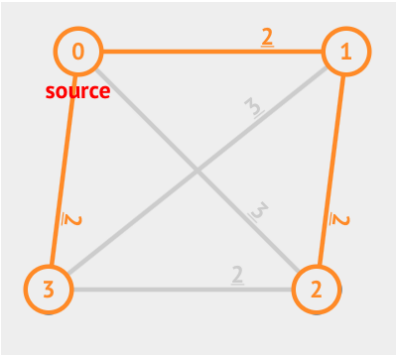
⑥



⑦



⑧

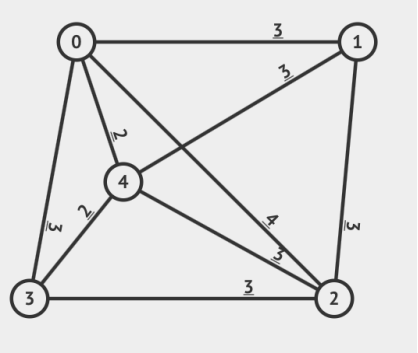


The process of your algorithm **could** look like this if correct🙂

Note: Because this problem has more than one possible solution, the final result could be slightly different. Just make sure your total equals the one in picture.😊

We have also provide more test cases for you to play with, and the process is highly similar and after going through the above content, you should already have a good idea about what is going on there 😊.

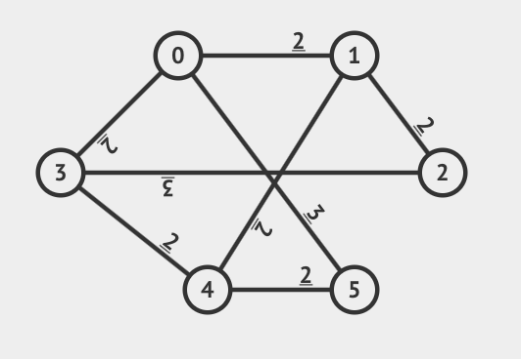
3.case2



😊You can run this test case by type in:

```
make run_case2
```

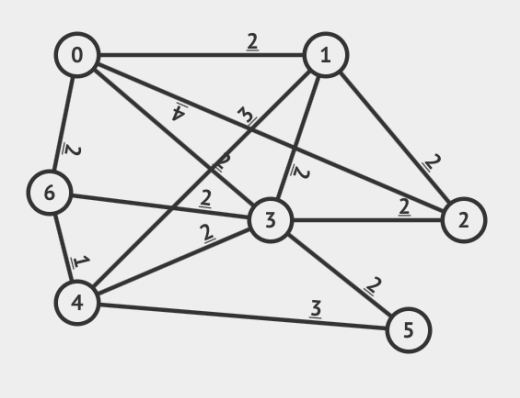
4.case3



😊You can run this test case by type in:

```
make run_case2
```

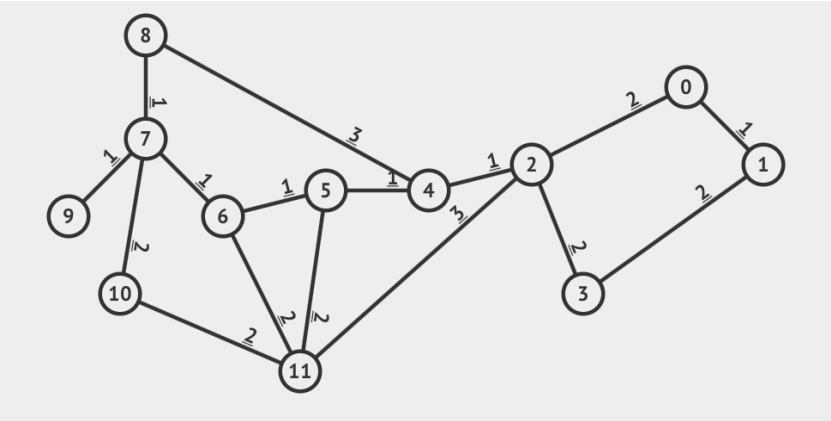
5.case4



😊You can run this test case by type in:

```
make run_case4
```

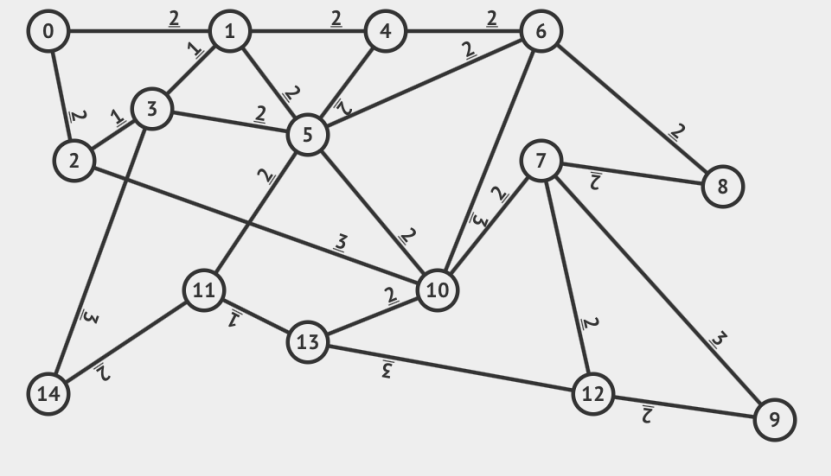
6.case5



😊You can run this test case by type in:

```
make run_case5
```

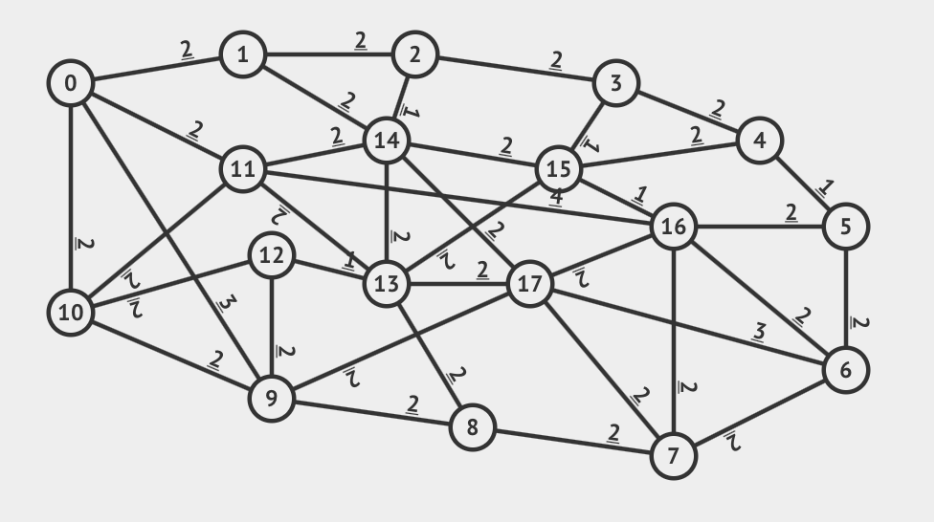
7.case6



😊You can run this test case by type in:

```
make run_case6
```

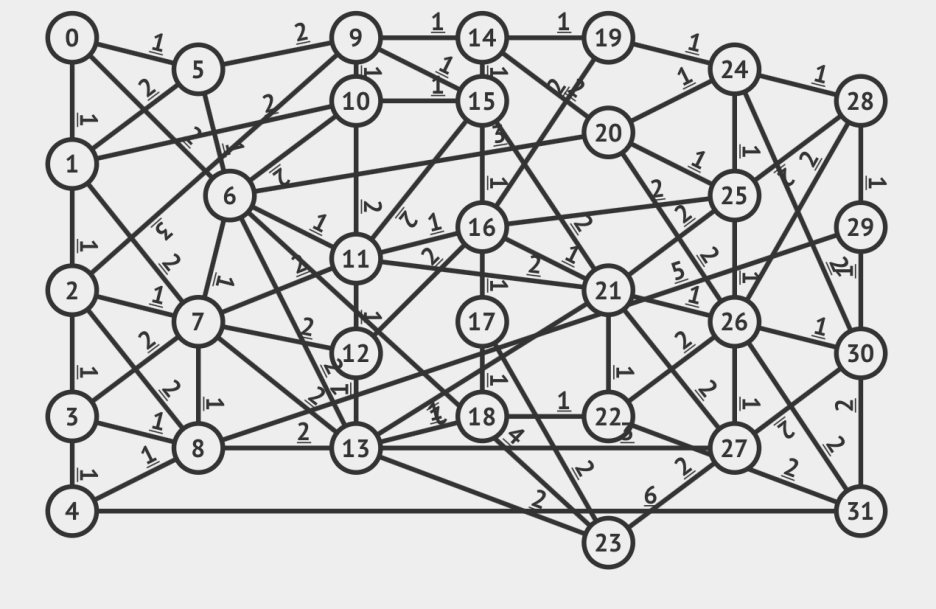
8.case7



😊You can run this test case by type in:

```
make run_case7
```

9.case8



😊You can run this test case by type in:

```
make run_case8
```

10.case9

This one is a little different as we did not use integer vertex value in this case (remember we are using `double` vertices right?).

😊You can run this test case by type in:

```
make run_case9
```

Final Remarks from the Team

This marks the end of our COMP2123 Self-Learning Report and thank you reading all the way through the end. Our team devoted great effort into this to make it as excellent as possible, and our effort are largely invisible. For instance, we implemented a universal `Graph.h` that could be used for any type of vertex using templates, and our own sorting function ("QuickSort.h") to support the data structure (as we are not allowed to use `<algorithm>`). Initially we implemented the

`Graph.h` using adjacency list however due to restriction on the usage of `<map>` (and `<list>` ... and `<pair>`) we finally redid it using adjacency matrix (and thus using `<vector>` only).

Our teammates all learned a lot in the process of making this, and our whole project is version controlled using GitHub, here is link to the repo https://github.com/GarfieldMa/COMP2123_Selflearning_Report (which will become public after 23:59 Dec 7, 2016) and this report is written in markdown and thank to our excellent Markdown editor iA Writer. Thank you again for reading!

MA Rutian, MAO Wenxu and YAO Qingning