this mousy mouse in her housie house has peony for her name
let peony show you how to be. you'll never be the same

Just see. Don't kill yourself locating it. Best be. Just fill yourself placating it.

Graph Algos

In this quest you get to implement a simple `Graph` class and a bunch of cool algorithms on it. You will define the graph class in `Graph.h` and `.cpp`, and implement your `Graph` algorithms in a separate friend class called `Gx` in files called `Graph_Algorithms.h` and `.cpp`.

The `Graph` class is one of the simplest classes of CS2C. Those of you who met *Simplee* can simply use her code.

The others can refer to Figure 1 and hack up your own copy quickly. This is going to be the basis of all your mqs in this quest.

```
#include <vector>
#include <climits>
#include <cfloat>  // FLT_MAX

// The graph is represented as a vector of nodes in which each node has
// a vector of edges. Each edge is represented by a target node and the
// weight of that edge.
class Graph {
public:
    struct Edge {
        int dst;
        float wt;

        Edge(int tgt = -1, float w = 0) : dst(tgt), wt(w) {}
        bool operator==(const Edge &that) const { return dst == that.dst && wt == that.wt; }
        bool operator!=(const Edge &that) const { return dst != that.dst || wt != that.wt; }
    };

protected:
    static double constexpr FLOOR = 1e-6;
    std::vector<std::vector<Edge> > _nodes;

public:
    size_t get_num_nodes() const { return _nodes.size(); }
    bool is_empty() const { return _nodes.empty(); };
    void clear() { _nodes.clear(); }
    std::string to_string() const;

    Graph &add_edge(int src, int tgt, float wt, bool replace=true);
    float find_edge_weight(int src, int tgt) const;

    friend class Gx;
    friend class Tests; // Don't remove this line.
};
```

Figure 1. The Graph Class

`Graph` is not a template class. Every node in this graph is identified by a unique non-negative integer.

Store it as a vector of a vector of `Edges`, where each edge is a structure containing a destination node number and a floating point weight.

Important: The graph is NOT a 2D matrix. It is a vector of nodes, where each node contains a vector of edges. Nodes that don't have edges leaving them will be represented by the cells that hold empty vectors. Put another way, the edge from node A to node B can be found in the vector located at `_nodes[A]`. But that edge is NOT `_nodes[A][B]`. Instead you must scan this inner vector to find the edge with the target of interest.

Some silent design decisions

- Node numbering starts at 0
- Self-loops are disallowed (and method behavior unspecified for graphs with self loops)
- Interpret edge weight as distance in distance problems, and capacity in flow problems.

Before getting started on the real MQs, first make sure your green level MQs are still functional. You even get rewards for it. You have to complete the implementation of the `Graph` class by supplying the following missing implementations in your `Graph.cpp`.

add_edge(int src, int tgt, float wt, bool replace=true)

After ensuring that your `_nodes` vector is large enough to index into either given node, insert an `Edge` object in the vector of edges leaving the source (`src`) pointing to the given destination (`tgt`) with the given weight (wt). Further helpful things about my reference implementation:

- I'm not keeping the edge lists sorted
- Every incoming Edge is compared to every existing one until a match is found or none can be found.
- If a match to an existing `Edge` with the same destination is found, the boolean variable `replace` tells what to do. If it is true, the given weight replaces the existing weight. If it is false then the given weight is added to the existing weight.
- If no match can be found, then a new `Edge` object with the given destination and weight is created and it is inserted at the *end* of the vector of edges leaving the given source.

float find_edge_weight(int src, int tgt)[1]

Here is the cool thing about this method which should bring back memories of water birds.[2] If it's asked for the weight of an edge that doesn't exist, it returns `Graph::Floor`. This hints at the hidden assumption in any of our algorithms - A Graph is essentially *completely connected*. But unlike stilts which believe things they can't see, mice don't assume that invisible edges exist.

[1] If you get to choose the name of this method, would it start with `get_` or `find_`? Why?
[2] Why?

Graph methods should scan the vector of edges for the given source to find if requested edges are explicitly present with non-FLOOR weights even for simple connectivity tests.

to_string

It's optional, and for your own benefit (like DDA said 'ere). If you don't wanna do it, simply return the empty string. No complaints.

If you do wanna, I'm afraid all I have is an example. See if you can reverse-engineer the specs from Figure 2.

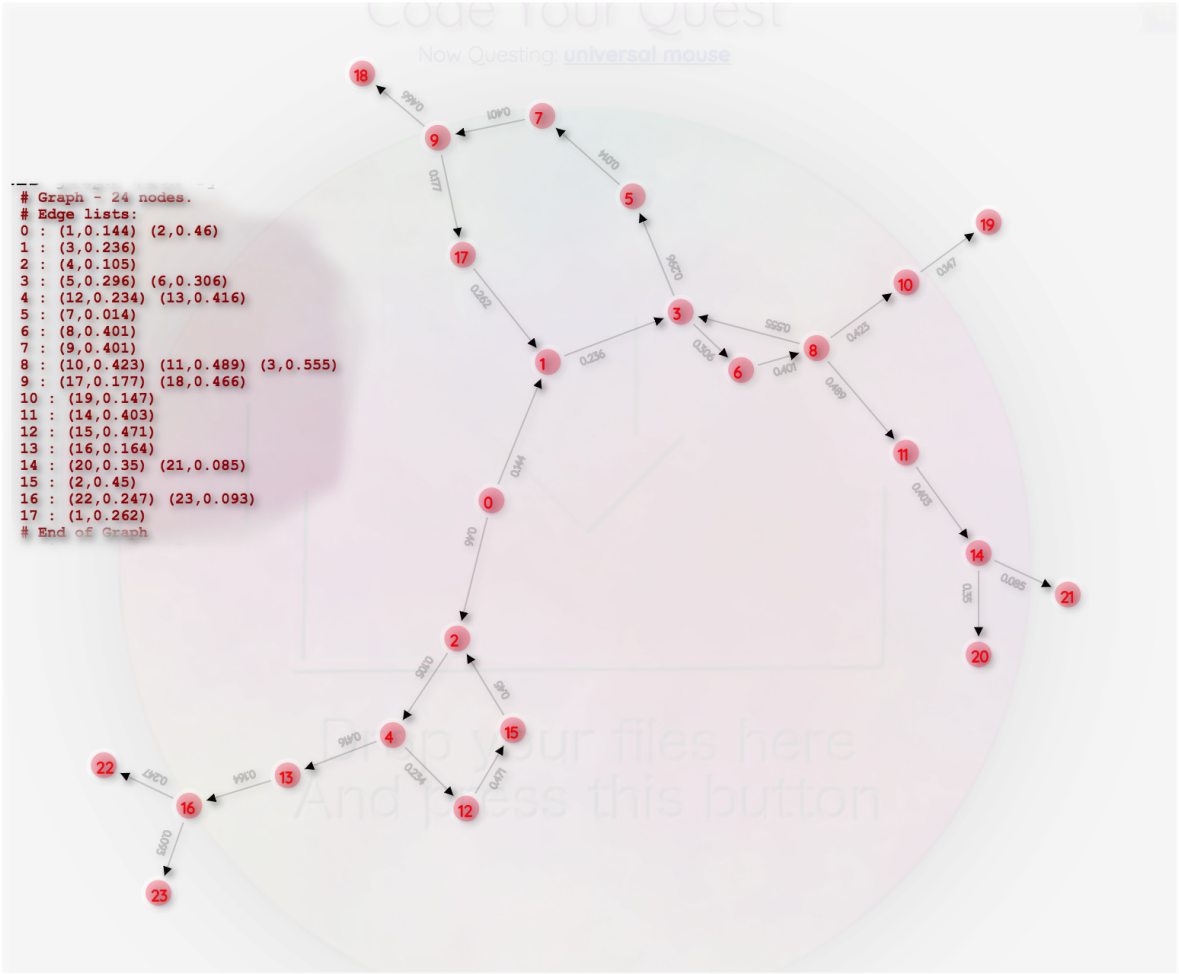


Figure 2. Example Graph::to_string() output

Now comes the interesting part. You get to implement all the cool algorithms in `Gx`. And some of them are worth a LOT of loot See Figure 3 for a picture of the `Gx` class.

```
class Gx {
private:
    struct NW { // package three co-used qtys
        int node, pred;
        float weight;
        NW(int n, int m, float wt) : node(n), pred(m), weight(wt) {}
        bool operator<(const NW &that) const { return this->weight > that.weight; }
        bool operator>(const NW &that) const { return this->weight < that.weight; }
    };

    static float _get_capacity_of_this_path(const Graph &g, const std::vector<int> &path);
    static float _get_max_capacity_path(const Graph &g, int src, int dst, std::vector<int> &path);
    static bool _is_cyclic(const Graph &g, size_t node, std::vector<bool> seen, std::vector<bool>& cycle_free);

public:
    static bool is_cyclic(const Graph &g);
    static bool prune_unreachables(Graph &g, int src);
    static size_t get_shortest_unweighted_path(const Graph &g, int src, int dst, std::vector<int> &path);
    static size_t get_shortest_weighted_path(const Graph &g, int src, int dst, std::vector<int> &path);

    static float get_max_flow(const Graph &g, int src, int dst);

    friend class Tests; // Don't remove
};
```

Figure 3. The `Gx` Class. Hmm… What does it take to get rid of the unsightly inversion of the comp ops?

Here are a few important things to know about the graphs you'll get:

In general, you can assume there are no self-loops in the graphs. You can safely ignore testing for them except where absolutely critical to make double-sure at low cost.

NW

This is like `std::pair`, except that it has 3 things in it that frequently ride together.

You'll find that you need to provide the comparison operators if you ever end up wanting to stick NWs in a heap. So it comes with two handy ones.

This is an optional (suggested) inner class. You may decide to package your variables differently, or not at all.

is_cyclic

If the given graph has one or more cycles, return true. Else return false.

There is no guarantee that the graph has only one entry point. That is, you cannot assume that node 0 is the only source without a parent.

Here is how I check for cyclicity. You don't have to do it this way: I iterate over the nodes and invoke a recursive helper method on each to tell if a graph rooted at the node is cyclic. The moment I find the

first cycle, I return true all the way back immediately. I don't waste time trying to find all the cycles.

My private recursive helper _is_cyclic

I start a depth first descent from the given node. Every node I encounter must be a brand new node. If I hit a node I have already seen, I immediately return true.

To keep track of seen nodes I need a collection that I can look up quickly. A hash table comes to mind. But if you're thinking that it's too heavy for something as simple as what we need, you be thinkin like me.

Instead of a hash table, I exploit the fact that my nodes are indexed by integers. This lets me store a set of N node markers as a vector of N bools.

In my recursive helper, I have two set-of-node-markers parameters. One parameter, called `seen`, keeps track of the seen nodes in a particular recursive descent. The other, called `cycle_free`, keeps track of nodes whose acyclicity has been established, so they don't have to be checked again.

`cycle_free` is passed by reference, so it contributes negligible space overhead to the recursive call. But `seen` is passed in by copy (why?).[3] This means that each new level in a recursive call will create a fresh copy of the vector.[4]

So suppose we do it that way, what's the worst that could happen? Always a good question, methinks.

The worst is if you end up with a degenerate graph which is simply one long linked list of many nodes. Then you'll end up recursing as deeply as the graph is *long*.

Just how bad is that? Suppose your input graph, rooted at 0, has 1K nodes, and that your graph is degenerate (a linked list of 1K nodes).

Then when you start your recursion at the root, you could end up recursing all the way to the leaf, 1000 nodes down, and thus 1000 levels deep. The call at each level would have its own copy of the `seen` vector. Since the graph has 1K nodes, each vector would take up 1K bytes (assuming 1 byte per bool). And therefore the total overhead due to this technique on the worst degenerate graph of 1000 nodes is at least 1K x 1K = 1MB.[5]

---

[3] Keven Yeh (2023) observed that it's possible to optimize further by passing `seen` by reference. How?

[4] YAWN (Yet another wacky nuance): The actual size of the vector passed by copy is not a serious consideration for running out of stack space in recursion. The vectors themselves will live on the heap, only their references will be on the program stack. How wacky cool is that?

[5] You can easily generalize this argument to show that the worst degenerate graph of N nodes will suffer a recursive space overhead of $O(N^2)$. Go for it.

To put this whole thing in perspective, you can assume that your process will get enough non-swap heap to handle graphs with degenerate filaments up to 15K nodes long (extremely unlikely).

prune_unreachables

Mark all nodes in the graph that are reachable from the given source node. Then sweep through the graph clearing out the edges of all OTHER nodes.

It's important that you don't delete the nodes themselves. Save yourself the hassle of renumbering the graph by leaving behind these edgeless nodes. Disconnected nodes can exist in a valid graph.

One minor deviation from the normal behavior of this method is that it will simply return false when passed a non-existent node.

The alternative is to clear the edges of all the nodes in the graph.

Since that's pretty drastic, we'll simply provide a better named method called `clear()` and let this method fail with a more humble false return in this situation.[6]

get_shortest_unweighted_path[7]

Essentially, this is just a breadth-first search starting at your source node, ending when you find the destination. The found path must be written into the parameter path vector passed by reference, and the length of the path must be returned.[8] The first element of `path` should be the source, and its last should be the destination.

---

[6] The three levels of Questing Humility:
Mouse level: (`return false`) "I'm sorry, m'boss. I canna do it."
Duck level: (`throw exception`) "Are you outta your quackin mind? You stupid quack! If you `try` that again I'll extinctify you! Dumb quackin dodos…"
Fangs level: (`delete *; format *; end_world();`) "There! You sorry-ass m$!%#. That'll teach you."

[7] If you get to choose the name of this method, would it start with `get_` or `find_`? Why? (Don't assume choices made in this quest are the best ones).

[8] Does it have to be BFS?

If there is no path from the source to the destination, you must return `string::npos`. If I request a path from a node to itself, you should return a path of length 1 (intuitively, it begins and ends at the source node).

Note that there may be multiple shortest unweighted paths. But only one of them is the lucky winner. No use sharing unlucky paths in the forums. 'Cuz they, like, change all of the time.

Better to figure out what is unlucky and stay away from it.

get_shortest_weighted_path

Once you have the unweighted path finder debugged, you'll find it easy to adapt it to find the shortest weighted path using Dijkstra's algorithm.

There are a few things to note here. If you work it out on paper, you'll see that this algorithm is really not all that different from its unweighted cousin. So think of it as enhancing what you already have, to account for edge weights.

The biggest difference is that instead of pushing and popping in a queue, you will push and pop in a min heap. Use the STL priority queue for this. Not the one you built in Baltimore.

Using a min heap guarantees that you process each of your neighbors in increasing order of their distance from you.

Why is it important to only process the nearest node each time? Intuitively, you can say that the nearest node is the ONLY node you can know for sure to not have a shorter path to it. How can you be sure that there is no shorter path from the source to your nearest node's nearest neighbor than the one through it (your nearest node)? Discuss.

Unsettling Settling

When you *settle* a node, you update the distances of all its neighbors. Some of these neighbors may already be in the min heap and need to be updated. How? As you probably know from Butterfly, updating particular elements in a heap is a right pain.

I know at least two less-than-ideal ways to handle this: One way is to hack the heap to let you hot-tweak element priorities.

The other way is to insert a new heap element with the now shorter distance. The old one will still be in the heap, but since it's a min heap, it won't be seen until after the new one has been seen. I think this is slightly better for our situation.

We'll go with this latter approach in this quest. But note that since your heap may collect a lot of junk as a consequence (why?), you need to have a way to remember nodes you've *settled* so you can silently skip them when they pop off the heap.

get_max_flow

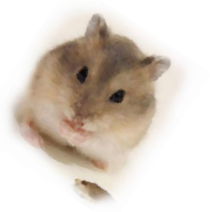
This is like the Grand Ganondorf of Red Questopia. Get this one right and you're pretty much done.

Conceptually, the max flow problem is even easier to solve than the shortest path problem. But only once you've solved the shortest path problem. It needs to return a single floating point number equal to the maximum calculated flow from src to dst.

I found that many programmers who find maxflow challenging at first get confused at a particular point: They assume that an edge that has been used up is gone, and not part of the graph any more. Thus they fear that they may end up with a graph in which some flow cannot be extracted because lots of such "taken" edges are now missing, but critically needed - Yes?

Here is a different perspective of the problem that you won't find elsewhere. I tried this on a student (in 2017) who said it helped:

Think of it this way: The moment some of the flow has been taken up in an edge, it immediately becomes capable of supporting exactly that much flow in the reverse direction. How? Just diminishing the forward flow is equivalent to increasing the reverse flow. See Figure 4.



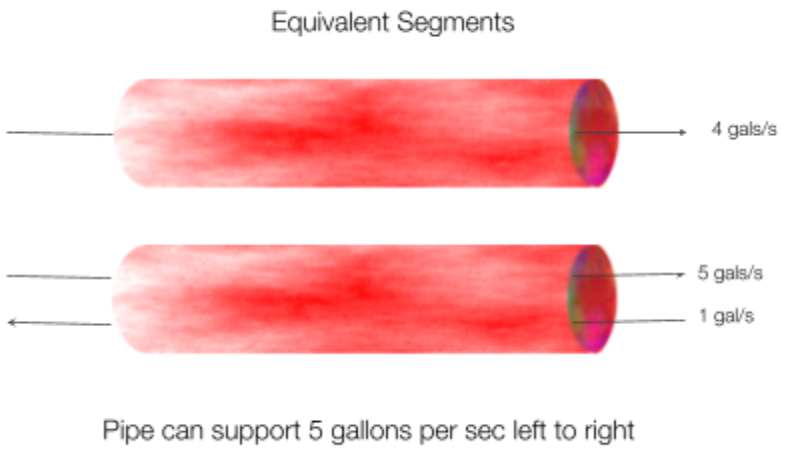


Figure 4. Another look at maxflow

So the trick to handling depleted edges (edges whose forward capacities you have used up) is not to remove them, but to add equal and opposite edges because those edges are still capable of conducting flow. Just not in the same direction.

I hope that helps you too.

As you implement max_flow, you may find you need the following two private helpers.

Private helper _get_capacity_of_this_path

Interpreting the edge weights of the input graph as flow capacities, return the capacity supported by the given path.

Yeah, something about a chain and its weakest link comes to mind here.

Private helper _get_max_capacity_path

This is almost identical to your get_shortest_weighted_path method, except for the fact that your heap compares in the opposite way.

Consider it a juicy opportunity to copy and refactor the code so that you need to rewrite as little as possible.[9]

[9] Here's an engineering question: How do you re-signaturize your get_shortest_weighted_path method to be able to select between a minheap or a maxheap at run time? (Don't try it for this quest). Is it worth it?

Submission

When you think you're happy with your code and it passes all your own tests, it is time to see if it will also pass mine.

1. Head over to https://quests.nonlinearmedia.org
2. Enter the secret password for this quest in the box.
3. Drag and drop your source files[10] into the button and press it.

Wait for me to complete my tests and report back (usually a minute or less).

And then…

It's all entirely up to you now.

If you peeps were cormorants, I guess I might say *go forth and multiply*. I wish you the very best.

&

Thank you for letting me serve you. With a pal like c++, ain't nobody no more to unnerve you[11]

[10] `Graph.{h,cpp}` & `Graph_Algorithms.{h,cpp}`
[11] Weeelll… I ate module zero for me brunch. These triple negs - they be me lunch.

Well, dat be all folks

in yer questopic adventure with rare and wondrous creatures

i bet you loved c++, and all her arcane features

With any luck, you're now ready to take on the best of 'em.