Shortest path

This problem consists of three subtasks: the Bellman-Ford algorithm, the Dijkstra's algorithm, and the difference constraints problem.

Please go to Piazza Resources to download the related materials first.

The `Graph` class

`graph.hpp` is a header file defining a `Graph` class that represents a **directed, edge-weighted** graph $G=(V,E)$ using adjacency lists. For simplicity, we assume that $V=\{0,1,\cdots,n-1\}$ and that the edge weights are integers. Considering that you might have not written any graph-related code before, we have implemented two basic functions for you as demonstration:

- The Breadth-First Search algorithm, which has the following signature

  ```
  void bfs(VertexID start, std::invocable<VertexID> auto callback) const;
  ```

  `start` is the id of the vertex where the traversal begins. `callback` is a callable object that is called every time a vertex is visited. For example, the following code (see `examples/graph_example.cpp`) performs BFS on the graph `graph` and prints the id of the visited vertices in order.

  ```
  graph.bfs(0, [](auto x) { std::cout << x << ' '; });
  ```

- The naive $O\left(V^2\right)$ Dijkstra's algorithm, which has the following signature

  ```
  std::vector<Weight> dijkstra(VertexID source) const;
  ```

  `source` is the id of the source vertex. This function returns a `std::vector<Weight>`, in which the element indexed `i` is the length of the shortest path from `source` to the vertex `i`, or `Graph::infinity` if no path exists.

  The behavior is undefined if there exists an edge with negative weight.

  > By saying "the behavior is undefined", we mean that **you don't need to care about this case**. Do not waste your effort detecting it.

  The implementation of this function is in `graph.cpp`, which you need to overwrite (See below).

Read through the code we have provided in `graph.cpp`, `graph.hpp` and `examples/graph_example.cpp` and make sure you have understood the design. To compile and run `graph_example.cpp`, refer to compile and run.

Note: Your submission will not contain the contents in `graph.hpp`. Any modification to that file (e.g. adding a member to the class `Graph`) will have no effect.

Subtask 1: The Bellman-Ford algorithm

First, you need to implement the Bellman-Ford algorithm that calculates the length of the shortest path from a given source vertex to each vertex, or reports that there is a negative cycle reachable from `source`. It is a member function of `Graph` that has the following signature

```
std::optional<std::vector<Weight>> bellmanFord(VertexID source) const;
```

> `std::optional<T>` is a special standard library type that represents **either an object of type `T`, or nothing (represented by `std::nullopt`)**. You can refer to `examples/equation.cpp` to see an example of using it.

This function should return `std::nullopt` if the graph contains a negative cycle reachable from `source`. Otherwise, it should return a vector of numbers, in which the element indexed `i` is the length of the shortest path from `source` to the vertex `i`, or `Graph::infinity` if no path exists. This function should run in $O(VE)$ time with $O(V)$ extra space.

Write your implementation in `graph.cpp`.

Subtask 2: The Dijkstra's algorithm

In `graph.cpp`, there is an implementation of the naive Dijkstra's algorithm whose time complexity is $O\left(V^2\right)$. Suppose now the graph is not so dense, or more specifically, $E = o\left(V^2/\log V\right)$. Overwrite this function to make it run within $O(E\log V)$ time, with $O(V)$ extra space.

You may need `std::priority_queue`. We have provided two examples of using it: `examples/priqueue.cpp` and `examples/huffman.cpp`. The latter is a program that computes the Huffman codes given the frequencies of characters.

> By default, `std::priority_queue<T>` uses `std::less<T>` (which is `operator<`) to compare elements and models a **max-heap**. If you need a min-heap, just pass `std::greater<>` as the third template argument (see details in the examples).
>
> We have seen some inexperienced students overload `operator<` to do the work that should have been done by `operator>` (e.g. `return lhs.something > rhs.something;`), or pass negated values to the max-heap to *fake* a min-heap. **You should never do that.**

Subtask 3: System of difference constraints

In a ***system of difference constraints*** (or, ***difference constraints problem***), there are n variables $x_0, x_1, \cdots, x_{n-1}$ and m constraints on these variables, where the k-th constraint ($0 \leqslant k < m$) is of the form

$$x_{u_k} - x_{v_k} \leqslant c_k$$

where $0 \leqslant u_k, v_k < n$, and $u_k \neq v_k$. We want to find a solution to the given difference constraints problem, that is, to assign the variables with values so that all the constraints are satisfied.

For example, the following is a system of difference constraints with 5 variables and 8 constraints:

$$\begin{aligned}
x_0 - x_1 &\leqslant 0,\\
x_0 - x_4 &\leqslant -1,\\
x_1 - x_4 &\leqslant 1,\\
x_2 - x_0 &\leqslant 5,\\
x_3 - x_0 &\leqslant 4,\\
x_3 - x_2 &\leqslant -1,\\
x_4 - x_2 &\leqslant -3,\\
x_4 - x_3 &\leqslant -3.
\end{aligned}$$

One solution to this problem is $(x_0, x_1, x_2, x_3, x_4) = (-5, -3, 0, -1, -4)$, which you can verify directly by checking each inequality. Note that a system of difference constraints usually have infinitely many solutions, but some systems may have no solutions at all. The following is a system with 3 variables and 3 constraints that have no solutions:

$$\begin{aligned}
x_0 - x_1 &\leqslant -1,\\
x_1 - x_2 &\leqslant -1,\\
x_2 - x_0 &\leqslant -1.
\end{aligned}$$

A system of difference constraints can be modeled as a graph, and then solved by solving a single-source shortest-path problem on the graph. **Read Section 24.4 of *Introduction to Algorithms, Third Edition*** which is available in Piazza Resources.

The difference constraints problem that you need to solve will be presented as the class `Problem` in `problem.hpp`. Read through the code in `problem.hpp` and understand the structure of that class. Note that we have implemented some useful functions for you, such as `hasNegativeConstant()`, `getNumVars()` and `getConstraints()`.

Your task is to implement the `solve` function in `main.cpp`, which has the signature

```
std::optional<Problem::Solution> solve(const Problem &problem);
```

It solves the given difference constraints problem `problem`, and returns the solution to it or `std::nullopt` if it has none. A solution is of type `Problem::Solution` i.e. `std::vector<Problem::Value>` in which the element indexed `i` is the value assigned to the variable x_i.

For testcase 1~5, c_k may be negative, and your algorithm should run within $O\left(n^2 + nm\right)$ time. For testcase 6~10, c_k are non-negative, and your algorithm should run within $O((n+m)\log n)$ time.

Compile and run

Basic knowledge

Read CompileBasics.md if you need.

Compile multiple files

The functions `Graph::dijkstra` and `Graph::bellmanFord` are defined in `graph.cpp`. Therefore, every time you compile a program that calls those functions, `graph.cpp` should be compiled and linked as well. You will see an error like

```
/usr/bin/ld: /tmp/ccxICQ7U.o: in function `main':
graph_example.cpp:(.text+0x15e): undefined reference to `Graph::dijkstra(unsigned long) const'
collect2: error: ld returned 1 exit status
```

if you did not link them correctly. The simplest way is to also pass the path to `graph.cpp` as an argument to the compiler. For example, to compile `solve.cpp`, `cd` to the directory `attachments` first, and then

```
g++ solve.cpp graph.cpp -o solve -std=c++20
```

To compile `examples/graph_example.cpp`: If the current working directory is `attachments`,

```
g++ examples/graph_example.cpp graph.cpp -o examples/graph_example -std=c++20
```

This will generate an executable named `graph_example` (`graph_example.exe` on Windows) in the directory `attachments/examples`. If the current working directory is `attachments/examples`,

```
g++ graph_example.cpp ../graph.cpp -o graph_example -std=c++20
```

will do the same thing.

> It is so annoying to enter such a long command every time! You can press `↑` to obtain your command history.

VSCode configurations

Read VSCodeConfig.md if you need.

Submission

Create a zip file `submission.zip` containing `graph.cpp` and `solve.cpp`, and then submit `submission.zip` to the OJ. You can `cd` to `attachments`, and then run the following command to create it:

```
zip -r submission.zip graph.cpp solve.cpp
```

Note that the files `graph.cpp` and `solve.cpp` should be placed directly in the zip file.