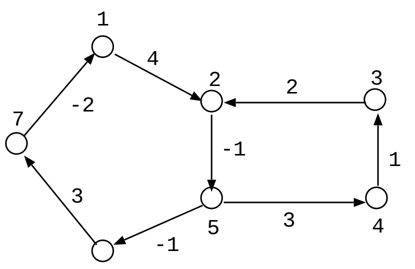
ECE 51220 Programming Assignment 3

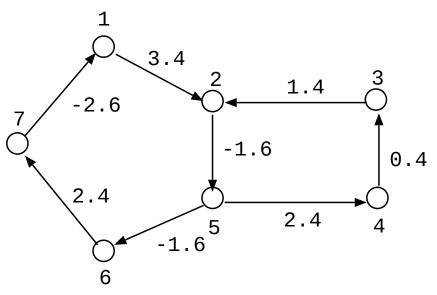
Description

You will implement a program to compute the minimum cycle mean of a graph. The minimum cycle mean problem relates to the shortest path problem. It also plays an important role in some minimum-cost flow algorithms.

Given a strongly connected graph $G(V,E,w)$, where V is the vertex set, E is the edge set, and $w : E \to \mathbb{R}$ is the edge weight function. For every cycle in the graph, the total weight of the cycle is obtained by summing the weights of all edges in the cycle. Its mean is obtained by dividing the cycle weight by the total number of edges in the cycle. Consider the following graph in (a) with two simple cycles $C_1 = 1 \to 2 \to 5 \to 6 \to 7 \to 1$ and $C_2 = 2 \to 5 \to 4 \to 3 \to 2$. A cycle is simple if it does not contain other cycles. C_1 has a cycle mean of $\frac{3}{5}$ and C_2 has a cycle mean of $\frac{5}{4}$.



(a) Original graph G(V,E,w)



(b) Modified graph G(V,E,w-λ*)

There are non-simple cycles in this graph. For example, you may loop around C_1 (C_2) many times. However, that would give you a cycle mean that is the same as that of C_1 (C_2). You may also loop around C_1 and C_2. That would give you a cycle mean that lies between $\frac{3}{5}$ and $\frac{5}{4}$.

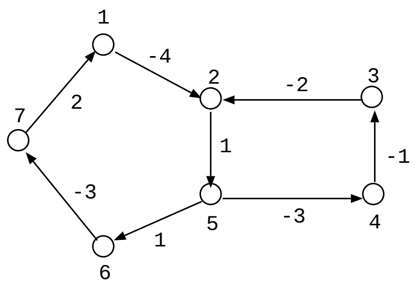
The minimum cycle mean, denoted as λ^*, is the minimum among all cycle means in a graph. For this graph, the minimum cycle mean is $\frac{3}{5}$. A cycle whose cycle mean is the minimum cycle mean is called a minimum-mean cycle. There may be several minimum-mean cycles in a graph.

When we subtract the minimum cycle mean λ^* from all edge weights, we obtain a modified graph denoted as $G(V,E,w-\lambda^*)$ (see (b) in the preceding figure). In this modified graph, a minimum-mean cycle becomes a zero-weight cycle. Moreover, the modified graph does not contain a negative cycle. Therefore, the shortest path (based on the weights of edges along the path) from any arbitrary source to a node is well-defined.

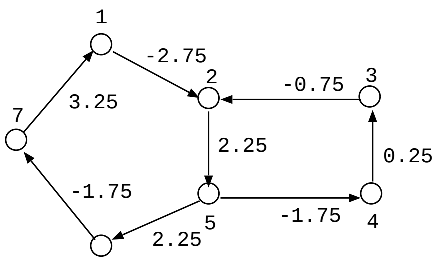
On the other hand, when we subtract a value strictly greater than the minimum cycle mean from all edge weights, there are negative cycles in the modified graph.

The minimum cycle mean is also well-defined when there are negative cycles in the graph. For the following graph in (a), which has a similar structure as the preceding graph but with different edge weights, both cycle weights and therefore, cycle means, are negative. Here, the minimum cycle mean is $-\frac{5}{4}$.

Again, when we construct the modified graph (see (b)) by subtracting the minimum cycle mean from all edge weights, the modified graph contains only non-negative cycles and all shortest paths from an arbitrary source are well defined.



(a) Original graph G(V,E,w)



(b) Modified graph G(V,E,w-λ*)

Minimum cycle mean algorithms

For solving the minimum cycle mean problem, several algorithms are available:

- [Karp] R. M. Karp. A Characterization Of The Minimum Cycle Mean In A Digraph. Discrete Mathematics 23, pages 309311, 1978.
- [Hartmann-Orlin] M. Hartmann and J. B. Orlin. Finding Minimum Cost to Time Ratio Cycles with Small Integral Transit Times. Networks, 23:567574, 1993.
- [Karp-Orlin] R. M. Karp and J. B. Orlin. Parametric shortest path algorithms with an application to cyclic staffing. Discrete Applied Mathematics, 3:3745, 1981.
- [Young-Tarjan-Orlin] N. E. Young, R. E. Tarjan, and J. B. Orlin. Faster Parametric Shortest Path and Minimum-Balance Algorithms. Networks, 21(2), 1991.

For this assignment, we focus on Karp's algorithm and the Hartmann-Orlin algorithm. Karp used dynamic programming to calculate the minimum cycle mean (MCM) in $\Theta(|V||E|)$ time complexity [Karp]. With an arbitrary source, say node 1 in both examples, Karp's algorithm starts with an 0-edge shortest path from the source, which contains only the source node 1. Then, Karp's algorithm computes all 1-edge shortest paths from the source. In general, Karp's algorithm computes all $(k+1)$-edge shortest paths from all k-edge shortest paths (from the source). Let $D_k(v)$ be the weight of the k-edge shortest path from the arbitrary source to node v,

$$D_{k+1}(v) = \min_{(u,v)\in E}[D_k(u) + w(u,v)].$$

Clearly, an iteration similar to that in Bellman-Ford algorithm extends k-edge paths to $(k+1)$-edge paths in $\theta(|E|)$ time complexity. Note that if there are no k-edge paths to v (from the arbitrary source), $D_k(v) = \infty$.

Note that these paths may already contain cycles. Karp's algorithm terminates when all $|V|$-edge shortest paths have been identified. At this point, it is guaranteed that a minimum-mean cycle is contained in one of these paths. Therefore, it takes $\theta(|V||E|)$ time complexity to find all $|V|$-edge shortest paths.

Let λ^* denote the minimum cycle mean. Karp characterized the minimum cycle mean as follows:

$$\lambda^* = \min_{v\in V} \max_{0\le k<|V|} \frac{D_{|V|}(v) - D_k(v)}{|V| - k}.$$

The early termination technique introduced by Hartmann and Orlin [Hartmann-Orlin] allows the Karp's algorithm to terminate before we construct all $|V|$-edge shortest paths. In other words, it allows Karp's algorithm to compute MCM in $O(|V||E|)$ time complexity.

Supposed we have computed all k-edge shortest paths, we have to find the smallest cycle mean among all the cycles in these k-edge shortest paths. Let λ_k denote the smallest cycle mean among all the cycles in these k-edge shortest paths. Of course, if no cycles are detected, we cannot terminate the algorithm.

With λ_k, we compute

$$\pi(v) = \min_{0 \le j \le k} (D_j(v) - j\lambda_k),\ \forall\ v \in V.$$

To better understand what $\pi(v)$ means, we examine the expression $D_j(v) - j\lambda_k$. Given $G(V,E,w)$, we reduce all edge weights by λ_k to obtain a modified graph $G(V,E,w-\lambda_k)$. The expression $D_j(v) - j\lambda_k$ corresponds to the j-edge shortest path weight to node v in the modified graph. Among these 0-edge, 1-edge, ..., k-edge shortest paths in the modified graph, we pick the smallest path weight and assign it to $\pi(v)$.

If the true shortest path to v in the modified graph is in one of 0-edge, 1-edge, ..., k-edge shortest paths, $\pi(v)$ is the real shortest path weight from the arbitrary source to v. If we have found all the true shortest paths to all nodes in the modified graph, the following constraint would hold for all edges:

$$\pi(v) \le \pi(u) + w(u,v) - \lambda_k,\ \forall\ (u,v) \in E.$$

Recall that when we have the minimum cycle mean λ^*, the modified graph $G(V,E,w-\lambda^*)$ has well-defined shortest paths because there are no negative cycles. When the preceding inequality holds for all edges, it means that we have found all the shortest paths in $G(V,E,w-\lambda_k)$. Since $\lambda_k \ge \lambda^*$, $G(V,E,w-\lambda_k)$ does not have well-defined shortest-paths when λ_k is strictly greater than λ^* (because of the presence of negative cycles). When we have well-defined shortest paths in the modified graph, it implies $\lambda_k = \lambda^*$. In other words, we have found the minimum cycle mean and we can terminate the process.

However, early termination has overhead. It takes $O(k)$ time to find cycles in a k-edge shortest path. As there may be $|V|$ such paths, it takes $O(k|V|)$ time complexity to find λ_k. It takes another $O(k|V|)$ time complexity to find $\pi(v)$ for all $v \in V$. It takes $O(|E|)$ time complexity to determine whether all shortest paths in the modified graph have been found by checking the validity of the inequality for every edge. If we perform a check for early termination for every k, $0 < k < |V|$, the complexity of the algorithm in the worst case would be $O(|V|^3 + |V||E|)$, which is worse than $\Theta(|V||E|)$ when the graph is sparse. To maintain $O(|V||E|)$ time complexity, it is recommended that the check for early termination is performed only when k is a power of 2.

When properly implemented, the Hartmann-Orlin algorithm has better run-time performance than Karp's algorithm. Moreover, if you allocate memory only when it is needed to extend k-edge paths to $(k+1)$-edge paths, the algorithm does not require as much memory as Karp's algorithm.

A good implementation of the Hartmann-Orlin algorithm is typically sufficient for this assignment. If you aim for even higher efficiency, you may want to consider Karp-Orlin [Karp-Orlin] or the Young-Tarjan-Orlin [Young-Tarjan-Orlin] algorithms. While the Young-Tarjan-Orlin algorithm suggests a Fibonacci heap in order to have good (amortized) time complexity, a binary heap implementation has been shown to be more efficient in practice.

For this programming assignment, you are given a graph that is strongly connected. In other words, all nodes can reach each other. You have to compute the minimum cycle mean of the graph.

You also have to produce a minimum-mean cycle as an output. There may be several minimum-mean cycles in the graph. You only have to produce one of them. These requirements imply that you have to keep track of the weights of the *k*-edge shortest paths as well as topology of the *k*-edge shortest paths.

Deliverables

In this assignment, you should develop your own include file and source files, which can be complied with the following command:

Your program should be compiled with the following command:

```
gcc -std=c99 -pedantic -Wvla -Wall -Wshadow -O3 *.c -o pa3
```

Again, if you supply a makefile, we will use your makefile and the command `‘‘make pa3’’` to generate the executable `pa3`. If we invoke the executable `pa3` as follows:

```
./pa3 in_file out_file1 out_file2
```

the executable loads the graph from `in_file` and saves the results into two output files: `out_file1` and `out_file2`. The first output file should contain the minimum cycle mean of the input graph (in binary) and the second output file should contain a minimum-mean cycle (in text).

Input file

`argv[1] in_file` is the name of the file that contains the input graph. Suppose there are *n* nodes and *m* edges in the graph. The first line is of the format `‘‘V %d\n’’`, where the `int` specifies that there are *n* nodes in the graph. The nodes are labeled 1 through *n*. The second line is of the format `‘‘E %d\n’’`, where the `int` specifies that there are *m* edges in the graph. There are subsequently *m* lines in the file, where each line specifies an edge with the format `‘‘%d %d %f\n’’`, where the first `int` is the destination node of the edge and the second `int` is the source node of the edge, and the `float` is the weight of the edge. The file `7_8.gr` contains the first example graph and the file `neg7_8.gr` contains the second example graph. We show the contents of `7_8.gr` below.

```
V 7
E 8
1 7 -2.000000
7 6 3.000000
6 5 -1.000000
5 2 -1.000000
2 1 4.000000
2 3 2.000000
3 4 1.000000
4 5 3.000000
```

First output file

`argv[2] out_file1` contains the name of the file that `pa3` would use to store the minimum cycle mean of the input graph. This should be a binary file that simply contains a `float` that corresponds to the minimum cycle mean that you have computed. You should use the function `fwrite` to produce the file. The respective first output files for `7_8.gr` and `neg7_8.gr` are `7_8.mcm` and `neg7_8.mcm`

Second output file

`argv[3] out_file2` is the name of the file that `pa3` would use to store a minimum-mean cycle. We will illustrate how you should store the labels of the nodes in the cycle in the second output file using the minimum-mean cycle for the first example graph. The minimum-mean cycle is $C_1 = 1 \to 2 \to 5 \to 6 \to 7 \to 1$. Let 1 be the starting (and ending node) in the cycle, we print 1 first followed by a space. The edge that leads to 1 has a source of 7. We print 7 followed by a space. In a sense, we follow the same convention as the input format, an edge is specified by the destination and then the source. Subsequently, we print 6, followed by a space, 5 followed by a space, and then 2. The next node is 1, which is the starting node. We do not print that. Instead, we print a `'\n'` after 2. Note that there is no space after 2, just a `'\n'`. The files `7_8.cycle` and `neg7_8.cycle` stored the minimum-mean cycles for the two example graphs, respectively. We show the contents of `7_8.cycle` below.

```
1 7 6 5 2
```

Note that the following shows the same cycle C_1. In fact, any cyclic left or right shift of the sequence describes the same cycle.

```
7 6 5 2 1
```

Submission

The assignment requires the submission (through Brightspace) of a zip file called `pa3.zip` that contains the source code (`.c` and `.h` files). For example, if you have only the necessary source and include files in the current directory, you can create the zip file as follows:

```
zip pa3.zip *.[ch]
```

A zip file created in this fashion does not contain a folder. You can add a `Makefile` to the zip file:

```
zip pa3.zip Makefile
```

If your `pa3.zip` contains a `Makefile`, we will use your `Makefile` to create `pa3` using the command:

```
make pa3
```

You should always copy the created zip file to a different directory, unzip the file in that directory, compile the program in that directory, and run some test cases in that directory. Submit the zip file only after you are certain that you have the correct zip file.

Your zip file should not contain a folder or directory (that contains the source files, include files, and Makefile). The zip file should contain only the source files, include files, and Makefile.

Grading

The grade depends on the correctness of your program and the efficiency of your program. The first and second output files account for 50 points each.

It is important that your program can accept any legitimate filenames as input or output files. Even if you cannot produce all output files correctly, you should still write the main function such that it produces as many correct output files as possible. Any output files that you cannot produce, you should leave them as empty file or not create them at all. Your program should return `EXIT_SUCCESS` only if you managed to produce all output files. It should return `EXIT_FAILURE` if it does not produce any of the output files.

It is important all the files that have been opened are closed and all the memory that have been allocated are freed before the program exits. Any memory issues reported by `valgrind` will result in a 50-point penalty.

What you are given

We provide in `pa3_examples.zip` three sample input files (`7_8.gr`, `neg7_8.gr`, and `10_40.gr`) in the folder `pa3_examples`. The corresponding first and second output files for `7_8.gr` are `7_8.mcm` and `7_8.cycle`, those for `neg7_8.gr` are `neg7_8.mcm` and `neg7_8.cycle`, and those for `10_40.gr` are `10_40.mcm` and `10_40.cycle`.

As the computation involves floating point numbers, you may not be able to match the minimum cycle means that we provided in these files.

In addition, we also provide in `pa3_examples.zip` three more sample input files without any associated output files: `100_400.gr`, `200_800.txt`, and `500_2000.txt`. Some of these files may be used for the evaluation of your submission.

About floating point operations

As in PA1, this assignment involves floating point operations as the edge weights are of type `float`. I suggest that you store them as `double` and use `double` variables in your implementation. However, when you produce the first output file, it is important that you copy the minimum cycle mean from a `double` variable to a `float` variable and use the `float` variable for your `fwrite` function.

With `math.h` included, you can use `INFINITY` or `-INFINITY` to initialize a `double` variable. You can also use `isinf()` to check whether a variable is infinite (positive or negative); the function takes in a `double` variable and return `1` if the variable is infinite or `0` if the variable is finite.

For early termination, your program has to check that the following condition holds:

$$\pi(v) \leq \pi(u) + w(u,v) - \lambda_k,\ \forall\ (u,v) \in E.$$

When the condition holds, you have $\lambda_k = \lambda^*$. However, with truncation errors associated with floating point operations, your computation of λ_k (or λ^*) will probably produce $\lambda'_k \neq \lambda_k$ (or $\lambda'^* \neq \lambda^*$) instead.

As in PA1, we allow for some discrepancies in your computed minimum cycle means (denoted by λ'^*) and our computed minimum cycle means (denoted by λ^*). For example, if our computed minimum cycle mean is 0, and if the absolute value of your computed minimum cycle mean is $\leq 1.0 \times 10^{-15}$, your result is considered to be acceptable. If our computed minimum cycle mean (λ^*) is non-zero, and $|\lambda^* - \lambda'^*|/|\lambda^*| \leq 1.0 \times 10^{-6}$, your computed minimum cycle mean (λ'^*) is considered to be acceptable.

While we allow for some discrepancies in λ^* and λ'^*, the inaccuracy in your computed λ'^* may not allow your implementation to terminate early. Even though $\pi(v) \leq \pi(u) + w(u,v) - \lambda_k$ for the edge (u,v), your computed λ'_k and the use of λ'_k for the computation of $\pi(v)$ and $\pi(u)$ may lead to $\pi(v) > \pi(u) + w(u,v) - \lambda'_k$, causing the early termination condition to fail.

You may want to consider a less accurate early termination condition:

$$\pi(v) \leq \pi(u) + w(u,v) - \lambda'_k + \text{TOLERANCE}, \ \forall\ (u,v) \in E,$$

where TOLERANCE is a non-negative parameter. I suggest that your implementation determines a suitable TOLERANCE based on λ'_k so that you do not terminate too early and obtain an unacceptable λ'_k.

Additional information

Check out the Brightspace website for any updates to these instructions.