MAC0328 GRAPH ALGORITHMS: PROGRAMMING ASSIGNMENT 3

LOOPHOLE

MARCEL K. DE CARLI SILVA AND HELOISA DE LAZARI BENTO

Due Date: 17/Nov/2024 at 11:59pm

1. Introduction

In this programming assignment, to be graded out of 100 marks, your tasks[1] are to (i) model the problem described in Section 2 as negative cycle detection, and (ii) to write a program to solve it using a variation of the Bellman-Ford algorithm.

Your code **must** be written in `C++17` and use the BGL library. As before, you must use the BGL **only** for the data structures (and corresponding accessor functions) that store a graph and the attributes for its vertices and edges. The use of any **BGL algorithm** is forbidden.

2. The Problem

Consider the following scenario. There is a finite set V of currencies. You may think of the Brazilian Real, US dollar, Euro, as well as other usual suspects. We play the role of some shady character whose aim is to game the system, by exploiting the exchange rates among currencies to become richer without doing any work whatsoever.

There is a bunch of financial services available to perform currency exchanges. Some of these services tax the player some extra fee on top of the "official"/market exchange rate, or provide us with a discount[2] if they are desperate to attract new customers.

Assume that, if $u, v \in V$ are distinct currencies such that some service allows the exchange of currency u to currency v, we have already shopped for the best exchange rate you can get on the market from u to v, and it allows you to exchange 1 unit of currency u for $c(uv) > 0$ units of currency v. Suppose, for instance, that we have 1 unit of currency $r \in V$ and we want to exchange it to some other currency $s \in V$, and that $v \in V$ is a currency distinct from r and s so that the exchanges from r to v and from v to s are possible. The player might perform the desired exchange by making 1 unit of currency r into $c(rv)$ units of the intermediate currency v, which can then be exchanged for $c(rv) \cdot c(vs)$ units of the target currency s.

A *loophole* is a sequence $\langle v_0, v_1, \ldots, v_\ell \rangle$ of currencies such that (i) $\ell \geq 1$, (ii) the currencies $v_1, \ldots, v_\ell$ are pairwise distinct, (iii) $v_0 = v_\ell$, and (iv) $\prod_{i=1}^{\ell} c(v_{i-1}v_i) > 1$. Hence, if our player can find such a loophole, the player might buy an initial position of 1 unit of currency v_0 and then perform the exchanges in the loophole to become infinitely/arbitrarily rich.

In this problem, you are given the set $V = [n]$ of currencies (for some positive integer n), the exchange rates between some ordered pairs of currencies, and your goal is to find a loophole, or to

Instituto de Matemática e Estatística, Universidade de São Paulo, R. do Matão 1010, 05508-090, São Paulo, SP

E-mail address: `{mksilva,heloisa}@ime.usp.br`.

[1]These two tasks are actually independent and treating them independently is an integral part of this assignment.

[2]You are welcome to join us in laughing out loud about this possibility. After you solve this problem, you will find one of the reasons why this never happens.

determine that no loophole exists. In the latter case, you must find a function[3] $z\colon V \to \mathbb{R}_{>0}$, where $\mathbb{R}_{>0} := \{\, x \in \mathbb{R} : x > 0 \}$, such that $z_v \geq z_u \cdot c(uv)$ for each pair uv of currencies for which $c(uv)$ is defined[4].

3. Test Cases

Your program should solve each test case in $O(nm)$ time, where n is the number of currencies and m is the number of ordered pairs of currencies for which an exchange is possible.

Each test case should be read from `stdin` (`std::cin`) and has following format:

- The first line has two integers, n and m, the numbers of currencies and possible exchanges, respectively, such that $1 \leq n \leq 500$ and $1 \leq m \leq n \cdot (n-1)$. You may assume that, for each pair of currencies u and v, there is at most one way of exchanging u for v.
- The next m lines have the description of the possible exchanges. Each exchange is represented by two distinct integers, $u, v \in [n]$, followed by a floating-point number $c(uv)$, meaning that 1 unit of currency u may be exchanged for $c(uv)$ units of currency v.

The existing driver/template code already handles reading the input digraph, called the `market`, and calling certain functions with a strict interface, as explained in Section 4.

4. Submission and Implementation Details

In the public repository for the course, folder `https://gitlab.uspdigital.usp.br/mksilva/mac0328-2024-public/-/tree/master/assignments/asgt3`, you will find 11 template files, which you must use in this assignment. We now describe how the files interact and which ones you should modify.

The only files you shall need to modify are: (i) `asgt.cpp`, and (ii) `digraph.h`. You are free to modify the other files for debugging purposes. However, you must **submit only** those files through the Moodle system, in the following format. The submission must be a compressed archive `NUSP.tar.gz`, obviously with `NUSP` replaced by your university ID number. The compressed archive must have precisely two files, inside no directory, namely `asgt.cpp` and `digraph.h`.

Failure to follow these instructions exactly will be **severely** penalized.

As before, you may modify auxiliary fields in bundled properties for vertices and edges/arcs in `digraph.h`. The prototypes of the functions you shall implement in `asgt.cpp` are listed in `asgt.h`:

[3] Such a function will be called a *multiplier*.

[4] A multiplier which satisfies these inequalities is called *feasible*.

[git] • 4cd63ef • Helo • 2024-10-29 23:22:01 -0300

```
Digraph build_digraph(const Digraph& market);

std::tuple<bool,
   boost::optional<NegativeCycle>,
   boost::optional<FeasiblePotential>> has_negative_cycle(Digraph& digraph);

FeasibleMultiplier build_feasmult(const FeasiblePotential& feaspot,
                                  const Digraph& aux_digraph,
                                  const Digraph& market);

Loophole build_loophole(const NegativeCycle& negcycle,
                        const Digraph& aux_digraph,
                        const Digraph& market);
```

To understand the types, pre- and post-conditions for these functions, we shall go over the driver code in `main.cpp`.

4.1. **Building an Auxiliary Digraph.** The driver starts by reading the input digraph, called `market`, and feeding it as a parameter to the function `build_digraph`, which you must code:

```
Digraph market{read_market(std::cin)};
Digraph digraph{build_digraph(market)};
```

The return value of this `build_digraph` call should be an auxiliary digraph on which we shall run the function `has_negative_cycle`, described next. The call to `build_digraph` should take time $O(n + m)$.

4.2. **Detecting Negative Cycles or Feasible Potentials.** The auxiliary `digraph` is fed as input to the `has_negative_cycle` function:

```
auto ret = has_negative_cycle(digraph);
```

Here is the prototype of that function again:

```
std::tuple<bool,
   boost::optional<NegativeCycle>,
   boost::optional<FeasiblePotential>> has_negative_cycle(Digraph& digraph);
```

The return value should be a triple (of type `std::tuple`) with the following conventions. The first element of the triple is a `bool`ean answer to the query “does `digraph` have a negative cycle?”. Here, the definition of negative cycle is the same as seen in the lectures, and the cost to be considered is a field for the arcs, defined in `digraph.h`:

```
struct BundledVertex {};

struct BundledArc { double cost; };

typedef boost::adjacency_list<boost::vecS,
                              boost::vecS,
                              boost::directedS,
                              BundledVertex,
                              BundledArc> Digraph;
```

Naturally, filling the `cost` field correctly for each arc is part of your modeling task from Section 4.1.

If `digraph` has a negative cycle, the call to `has_negative_cycle` should ultimately[5] return it in the second element of the return triple, which is an optional object of type `NegativeCycle`. Otherwise, the call to `has_negative_cycle` should build and return a feasible potential through the third element of the return triple, which is an optional object of type `FeasiblePotential`. As you may expect, this task can be done via an adaptation of the Bellman–Ford algorithm, and it should take time $O(nm)$.

Note that the second and third elements of the return tuple use `boost::optional`, described here. The template files already illustrate how to use them, and you will likely not need to consult the documentation though.

4.3. **Building a Feasible Potential.** To return a `FeasiblePotential` object from inside the function `has_negative_cycle`, you shall need to build one. The only way to do so is to use the constructor in `potential.h`:

```
class FeasiblePotential
{
public:
  FeasiblePotential(const Digraph& digraph,
                    const std::vector<double> y);
   [...]
};
```

The parameter `std::vector<double> y`, as usual, should be indexed by the vertices of `digraph`.

You can see from the implementation of the constructor in `potential.cpp` that the construction will be completed **only if you provide a feasible potential**; that is, if you try to build `FeasiblePotential` by passing a potential that is not feasible, the constructor will throw an exception[6]:

[5] As you start coding, you may first aim at obtaining the right answer for the query, **without** actually building a negative cycle. The driver will work correctly, but it will “complain” that you reported a negative cycle but did not provide one.

[6] You can find the definitions of the exceptions related to potentials in the file `potential-errors.h`.

```
FeasiblePotential::FeasiblePotential(const Digraph& digraph,
                                     const std::vector<double> y)
[...]
{
  [...]
  const auto& arcs = boost::make_iterator_range(edges(digraph));
  const auto& viol = find_if_not(arcs.begin(), arcs.end(),
                                 [&](const Arc& arc) -> bool {
                                   const Vertex& u = source(arc, digraph);
                                   const Vertex& v = target(arc, digraph);
                                   return y[v] <= y[u] + digraph[arc].cost;
                                 });
  if (viol != arcs.end()) {
    throw InfeasiblePotential_error(digraph, *viol, y);
  }
}
```

4.4. **Building a Negative Cycle.** The alternative to building a feasible potential in the call to `has_negative_cycle` is to build a negative cycle. However, similarly to what was described in Section 4.3, one can only build `NegativeCycle` object from a true negative cycle; that is, if you try to build a `NegativeCycle` from a walk that is not a cycle, or from a cycle that is not negative, the constructor will throw an exception.

Here is how the process of building a `NegativeCycle` works. The relevant files now are `cycle.h` and `cycle.cpp`, with corresponding exceptions in `cycle-errors.h`.

First, one must build a `Walk` object, with part of the public interface as follows:

```
class Walk {
public:
  Walk(const Digraph& digraph, const Vertex& start);
  bool extend(const Arc& arc);
  [...]
};
```

One must start by calling the constructor while providing the host/ambient `digraph` where the walk lives, and the `starting` vertex of the walk. You can then `extend` the walk one arc at a time; the `bool` return value of a call to `extend` tells whether the call succeeded[7]. The `Walk` object thus maintains a sequence of vertices and arcs that is guaranteed to be a true walk. Once the walk has been “grown” into a cycle (this can be checked by calling `Walk`’s `is_cycle` method), one may build a cycle:

[7]The call will fail if the `const Arc& arc` argument does not start at the last vertex of the current walk.

```
class Cycle
{
public:
  Cycle(const Walk& cycle) [...] {
    if (!cycle.is_cycle()) {
      throw NotCycle_error();
    }
  }
  [...]
}
```

Of course, we only care about negative cycles. A `NegativeCycle` object may be built similarly:

```
class NegativeCycle : public Cycle
{
public:
  NegativeCycle(const Walk& cycle) [...] {
    [...]
    _cost = accumulate(costs.cbegin(), costs.cend(), 0.0);
    if (_cost >= 0.0) {
      throw NonnegativeCycle_error(static_cast<Cycle>(*this), _cost);
    }
  }

  [...]
};
```

4.5. **Back to the Driver.** Upon regaining control after the call to `has_negative_cycle`, the driver branches according to the answer to the query. If the first element of the triple is `false`, then the third tuple element must hold a `FeasiblePotential` object, which is then passed as a parameter in a call to the following function, which you must provide:

```
FeasibleMultiplier build_feasmult(const FeasiblePotential& feaspot,
                                  const Digraph& aux_digraph,
                                  const Digraph& market);
```

This function must somehow transform the feasible potential `feaspot` in the auxiliary digraph `aux_digraph` into a `FeasibleMultiplier` object (whose construction is analogous to the `FeasiblePotential` object, as described in Section 4.3) in the `market` input digraph. This must run in time $O(n)$.

On the other hand, if the first element of the triple returned by `has_negative_cycle` is `true`, the driver feeds the `NegativeCycle` object from the second tuple element as a parameter to the following function, which you also must provide:

```
Loophole build_loophole(const NegativeCycle& negcycle,
                        const Digraph& aux_digraph,
                        const Digraph& market);
```


[git] • 4cd63ef • Helo • 2024-10-29 23:22:01 -0300

This function must somehow transform the negative cycle `negcycle` in the auxiliary digraph `aux_digraph` into a `Loophole` object (whose construction is analogous to the `NegativeCycle` object, as described in Section 4.4) in the `market` input digraph. This must run in time $O(n+m)$.

4.6. **Numerical Issues.** Since the computations involve floating-point numbers, your code may bump into numerical issues. The driver code has some tolerance for errors, as you may easily check. However, it may be the case that the tolerance is still a bit too stringent.

If the code bumps into numerical issues with the driver (e.g., a feasible potential is rejected due to rounding) and you feel confident that the issue lies with a "faulty" tolerance, please report back.

5. Workflow and Hints

You are welcome to ask questions about the mechanics of the template/driver code, though bear in mind that understanding how to solve the problem in this strict framework is an important component of the assignment.

Here is a possibility on how you may use the driver code. First, explore it with the bogus code already provided in `asgt.cpp`, to exercise raising various exceptions. Then, have the `build_digraph` function just make a copy of the input digraph, while you code a working version of the Bellman–Ford algorithm and test it (with your own handmade inputs). In this preliminary phase, you may just have the function `has_negative_cycle` just return a triple with the first `bool`ean element filled.

As you become confident that you can build a feasible potential correctly, work on getting the constructor of `FeasiblePotential` to complete and return that kind of object. Finally, as you figure out how to build a negative cycle when Bellman–Ford detects one, you may start the slightly more complicated building process described in Section 4.4.

Independently from these tasks, you can work on modeling the desired problem to fit the framework. Once you figure out a correct way of modeling the problem, implementing the three remaining functions in `asgt.h` should be possible.

6. Fallback Grading

As in previous assignments, in case your code does not provide correct answers, the driver code will fall back to testing only the `has_negative_cycle` implementation, for partial credit. This is done first by checking whether it provided a `NegativeCycle` object in the appropriate cases, and later accepting just the correct `bool`ean return value **and**[8] a `FeasiblePotential` object when that is the case.

Note that your code need not do anything different about this fallback grading.

[8] That is, the least stringent fallback option will only be accepted if the `bool`ean answer is correct, **and** a feasible potential is provided whenever it should be.