MAC0328 GRAPH ALGORITHMS: PROGRAMMING ASSIGNMENT 3

**1. INTRODUÇÃO**

Tarefas (independentes):

1. modelar o problema descrito na Seção 2 como detecção de ciclo negativo;
2. escrever um programa para resolvê-lo usando uma variação do algoritmo de Bellman-Ford.

**2. O PROBLEMA**

Considere o seguinte cenário. Existe um conjunto finito V de moedas. Desempenhamos o papel de algum personagem obscuro cujo objetivo é burlar o sistema, explorando as taxas de câmbio entre as moedas para enriquecer sem fazer nenhum trabalho.

Há um monte de serviços financeiros disponíveis para realizar trocas de moeda. Alguns destes serviços cobram uma taxa extra ao jogador sobre a taxa de câmbio “oficial”/do mercado, ou dão-nos um desconto se estiverem desesperados para atrair novos clientes.

Suponha que, se u, v ∈ V são moedas distintas de tal forma que algum serviço permite a troca da moeda u para a moeda v, já compramos a melhor taxa de câmbio que você pode obter no mercado de u para v, e ela permite que você troque 1 unidade de moeda u por c(uv) > 0 unidades de moeda v.

Suponha, por exemplo, que temos 1 unidade de moeda r ∈ V e queremos trocá-la por alguma outra moeda s ∈ V , e que v ∈ V é uma moeda distinta de r e s, de modo que as trocas de r para v e de v para s são possíveis. O jogador pode realizar a troca desejada transformando 1 unidade da moeda r em unidades c(rv) da moeda intermediária v, que pode então ser trocada por unidades c(rv) · c(vs) da moeda alvo s.

Uma brecha é uma sequência (v0, v1, ..., vl) de moedas tais que:

1. l ≥ 1,
2. as moedas v1, ..., vl são distintas aos pares,
3. v0 = vl e
4. somatório (i=1 até l) c(vi−1vi) > 1.

Portanto, se nosso jogador puder encontrar tal brecha, o jogador poderá comprar uma inicial posição de 1 unidade de moeda v0 e então realizar as trocas na brecha para ficar infinitamente/arbitrariamente rico.

Neste problema, você recebe o conjunto V = [n] de moedas (para algum inteiro positivo n), as taxas de câmbio entre alguns pares ordenados de moedas e seu objetivo é encontrar uma brecha ou determinar que não existe brecha. Neste último caso, você deve encontrar uma função z: V -> R > 0, onde R>0 := {x e R : x > 0}, tal que zv >= zu · c(uv) para cada par uv de moedas para o qual c(uv) é definido.

**3. CASOS DE TESTE**

Seu programa deve resolver cada caso de teste em tempo O(nm), onde n é o número de moedas e m é o número de pares ordenados de moedas para os quais uma troca é possível.

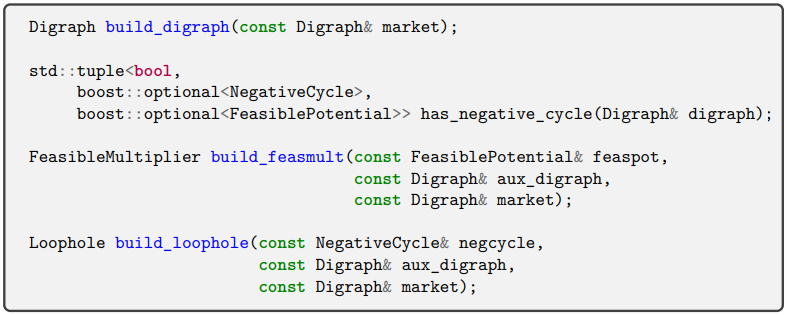
Cada caso de teste tem o seguinte formato:

* A primeira linha tem dois inteiros, **n** e **m**, os números de **moedas** e **trocas** possíveis, respectivamente.
* As próximas m linhas têm a descrição das possíveis trocas. Cada troca é representada por dois inteiros distintos, u, v ∈ [n], seguidos por um número de ponto flutuante c(uv), significando que 1 unidade de moeda u pode ser trocada por c(uv) unidades de moeda v.

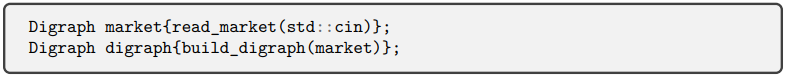
O código de driver/modelo existente já lida com a leitura do dígrafo de entrada, chamado de mercado, e a chamada de certas funções com uma interface estrita, conforme explicado na Seção 4.

**4. SUBMISSÃO E DETALHES DE IMPLEMENTAÇÃO**

Os únicos arquivos que você precisará modificar são: **asgt.cpp** e **digraph.h**.

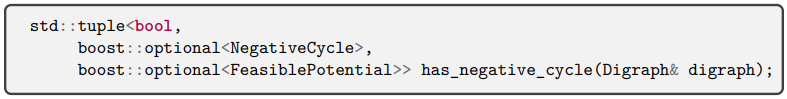
Os protótipos das funções que você deve implementar em asgt.cpp estão listados em asgt.h:

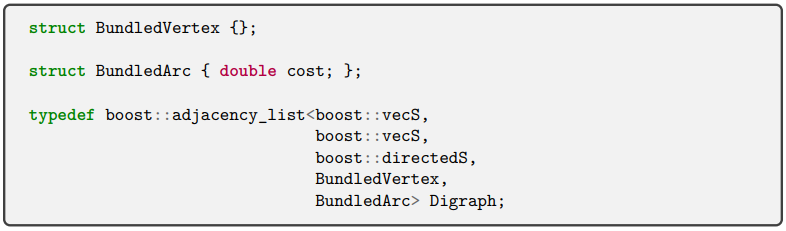
Para entender os tipos, pré e pós-condições para essas funções, vamos examinar o código do driver em main.cpp.

**4.1. Construindo um Dígrafo Auxiliar.** O driver começa lendo o dígrafo de entrada, chamado market, e alimentando-o como parâmetro para a função *build\_digraph*, que você deve codificar:

O valor de retorno dessa chamada *build\_digraph* deve ser um dígrafo auxiliar no qual executaremos a função *has\_negative\_cycle*, descrita a seguir. A chamada para build\_digraph deve levar tempo O(n + m).

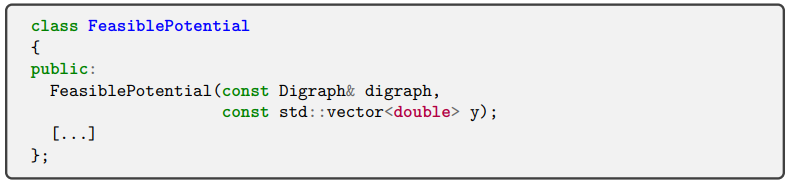
**4.2. Detectando Ciclos Negativos ou Potenciais Viáveis.** O dígrafo auxiliar é alimentado como entrada para a função *has\_negative\_cycle*:

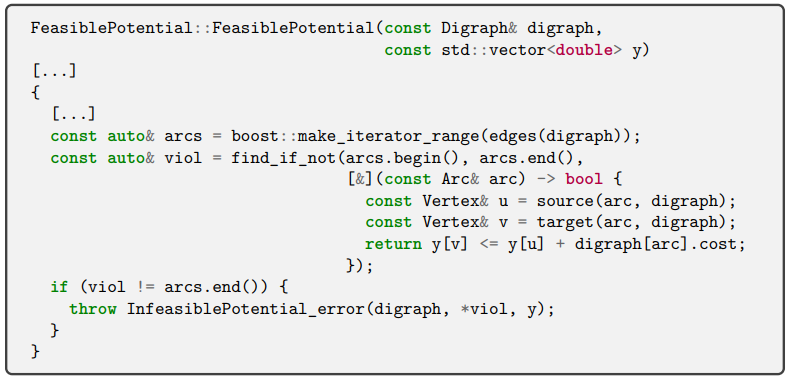
Aqui está o protótipo dessa função novamente:

O valor de retorno deve ser um triplo (do tipo std::tuple) com as seguintes convenções. O primeiro elemento da tripla é uma resposta booleana à pergunta “o dígrafo tem um ciclo negativo?”. Aqui, a definição de ciclo negativo é a mesma vista nas aulas, e o custo a ser considerado é um campo para os arcos, definido em digraph.h:

Naturalmente, preencher o campo de custo corretamente para cada arco faz parte de sua tarefa de modelagem da Seção 4.1. Se digraph tem um ciclo negativo, a chamada para *has\_negative\_cycle* deve finalmente retorná-lo no segundo elemento da tripla de retorno, que é um objeto opcional do tipo NegativeCycle. Caso contrário, a chamada para *has\_negative\_cycle* deve construir e retornar um potencial viável através do terceiro elemento da tripla de retorno, que é um objeto opcional do tipo FeasiblePotential. Como você pode esperar, essa tarefa pode ser feita por meio de uma adaptação do algoritmo de **Bellman-Ford** e deve levar tempo O(nm).

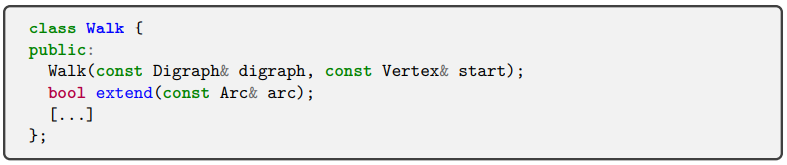
Observe que o segundo e o terceiro elementos da tupla de retorno usam boost::opcional, descrito aqui. Os arquivos de modelo já ilustram como usá-los, e você provavelmente não precisará consultar a documentação.

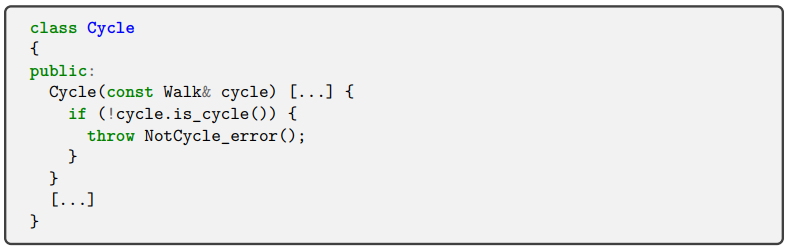
**4.3. Construindo um Potencial Viável.** Para retornar um objeto FeasiblePotential de dentro da função *has\_negative\_cycle*, você precisará construir um. A única maneira de fazer isso é usar o construtor em potential.h:

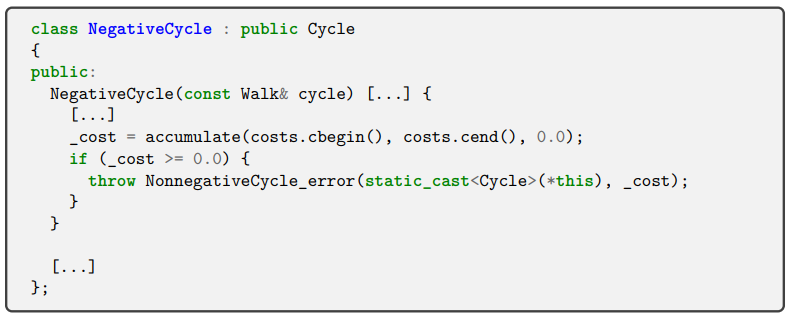
O parâmetro std::vector<double> y, como de costume, deve ser indexado pelos vértices do dígrafo. Você pode ver pela implementação do construtor em potential.cpp que a construção será concluída somente se você fornecer um potencial viável; ou seja, se você tentar construir o FeasiblePotential passando um potencial que não é viável, o construtor lançará uma exceção:

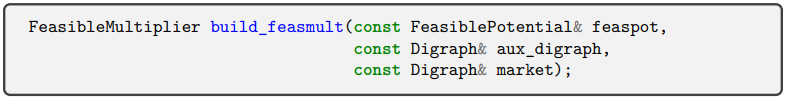
**4.4. Construindo um Ciclo Negativo.** A alternativa para construir um potencial viável na chamada para *has\_negative\_cycle* é construir um ciclo negativo. No entanto, da mesma forma que foi descrito na Seção 4.3, só se pode construir o objeto NegativeCycle a partir de um verdadeiro ciclo negativo; isto é, se você tentar construir um NegativeCycle a partir de um passeio que não seja um ciclo, ou de um ciclo que não seja negativo, o construtor lançará uma exceção.

Aqui está como funciona o processo de construção de um NegativeCycle. Os arquivos relevantes agora são cycle.he cycle.cpp, com exceções correspondentes em cycle-errors.h.

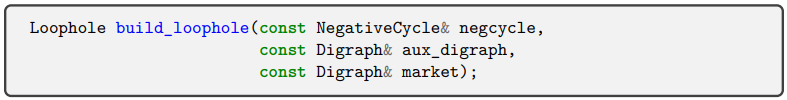
Primeiro, deve-se construir um objeto Walk, com parte da interface pública da seguinte forma:

Deve-se começar chamando o construtor enquanto fornece o dígrafo host/ambiente onde o passeio vive e o vértice inicial do passeio. Você pode então estender a caminhada um arco de cada vez; o valor de retorno bool de uma chamada para estender informa se a chamada foi bem-sucedida. O objeto Walk mantém, assim, uma sequência de vértices e arcos que garante ser um verdadeiro passeio. Uma vez que a caminhada tenha se tornado um ciclo (isso pode ser verificado chamando o método is\_cycle de Walk), pode-se construir um ciclo:

Claro, só nos preocupamos com ciclos negativos. Um objeto NegativeCycle pode ser construído de forma semelhante:

**4.5. De volta ao motorista.** Ao recuperar o controle após a chamada para has\_negative\_cycle, o driver ramifica de acordo com a resposta à consulta. Se o primeiro elemento da tripla for false, então o terceiro elemento da tupla deve conter um objeto FeasiblePotential, que é então passado como parâmetro em uma chamada para a seguinte função, que você deve fornecer:

Esta função deve, de alguma forma, transformar o feaspot potencial viável no dígrafo auxiliar aux\_digraph em um objeto FeasibleMultiplier (cuja construção é análoga ao objeto FeasiblePotential, conforme descrito na Seção 4.3) no dígrafo de entrada de mercado. Isso deve ser executado no tempo O(n).

Por outro lado, se o primeiro elemento da tripla retornado por has\_negative\_cycle for true, o driver alimentará o objeto NegativeCycle do segundo elemento da tupla como parâmetro para a seguinte função, que você também deve fornecer:

Esta função deve de alguma forma transformar o ciclo negativo negcycle no dígrafo auxiliar aux\_digraph em um objeto Loophole (cuja construção é análoga ao objeto NegativeCycle, conforme descrito na Seção 4.4) no dígrafo de entrada de mercado. Isso deve ser executado no tempo O(n + m).

**4.6. Problemas Numéricos.** Como os cálculos envolvem números de ponto flutuante, seu código pode esbarrar em problemas numéricos. O código do driver tem alguma tolerância a erros, como você pode verificar facilmente. No entanto, pode ser que a tolerância ainda seja um pouco rigorosa demais. Se o código esbarrar em problemas numéricos com o driver (por exemplo, um potencial viável é rejeitado devido ao arredondamento) e você se sentir confiante de que o problema está com uma tolerância "defeituosa", informe novamente.

**5. FLUXO DE TRABALHO E DICAS**

Você pode fazer perguntas sobre a mecânica do código do modelo/driver, embora tenha em mente que entender como resolver o problema nessa estrutura rígida é um componente importante da tarefa.

Aqui está uma possibilidade de como você pode usar o código do driver. Primeiro, explore-o com o código falso já fornecido em asgt.cpp, para exercitar a geração de várias exceções. Então, faça com que a função build\_digraph faça uma cópia do dígrafo de entrada, enquanto você codifica uma versão funcional do algoritmo Bellman–Ford e a testa (com suas próprias entradas feitas à mão). Nesta fase preliminar, você pode apenas ter a função has\_negative\_cycle apenas retornar um triplo com o primeiro elemento booleano preenchido.

À medida que você se torna confiante de que pode construir um potencial viável corretamente, trabalhe para fazer com que o construtor de FeasiblePotential complete e retorne esse tipo de objeto. Finalmente, ao descobrir como construir um ciclo negativo quando Bellman–Ford detecta um, você pode iniciar o processo de construção um pouco mais complicado descrito na Seção 4.4.

Independentemente dessas tarefas, você pode trabalhar na modelagem do problema desejado para se adequar ao framework. Depois de descobrir uma maneira correta de modelar o problema, a implementação das três funções restantes em asgt.h deve ser possível.

**6. FALLBACK GRADING**

Como nas atribuições anteriores, caso seu código não forneça respostas corretas, o código do driver voltará a testar apenas a implementação has\_negative\_cycle, para crédito parcial. Isso é feito primeiro verificando se ele forneceu um objeto NegativeCycle nos casos apropriados e, posteriormente, aceitando apenas o valor de retorno booleano correto e um objeto FeasiblePotential quando isso

é o caso. Observe que seu código não precisa fazer nada diferente sobre essa classificação de fallback.