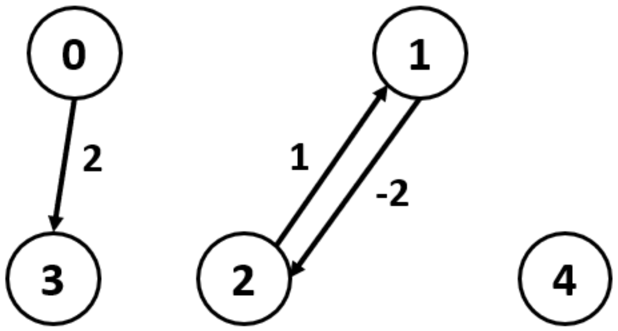
CS2040S: Data Structures and Algorithms

Exercises/Optional Problems for Week 10

For: March 24, 2020

Problem 1. Bellman Ford

You are given a directed graph where the edges can have negative weights. The nodes are labelled as $0, 1, 2, ..., V-1$, assuming there are V nodes. Below is a sample graph of 5 nodes:



To represent the weighted edges in the graph, you have been provided with `IntPair` class (in `IntPair.java`) and `IntPairComparator` class (in `IntPairComparator.java`). Do not modify these 2 classes and do not include `IntPair.java` and `IntPairComparator.java` in your submission.

This graph will be stored in an adjacency list as `ArrayList<ArrayList<IntPair>>` in the following manner:

- 0: [(3, 2)] // There is an outgoing edge of weight 2 from 0 to 3
- 1: [(2, -2)] // There is an outgoing edge of weight -2 from 1 to 2
- 2: [(1, 1] // There is an outgoing edge of weight 1 from 2 to 1
- 4: [] // There is no outgoing edge from 4

Given this graph, suppose we fix the source node to be node 0, then the shortest distance to node 1, 2 and 4 will all be `INF` (more details on this below) because they are unreachable; the shortest distance to 3 will be 2.

Your job is to implement the following 3 methods in `BellmanFord.java` to find the shortest distance from the fixed source node to any other node:

- `BellmanFord(ArrayList<ArrayList<IntPair>> adjList)` Given an adjacency list, initialize the necessary attributes needed for your implementation.
- `void computeShortestPaths(int source)` Given a source node, compute the distance of the shortest path from this source node to any other node (including source itself) in the graph. If a node is unreachable from the source node, the distance should be `INF` as defined in `BellmanFord.java`. If the path contains a negative weight cycle, the distance should be `NEGINF` as defined in `BellmanFord.java`. The computation should be done in $O(VE)$ time, where V is the number of nodes and E is the number of edges.

- `int getDistance(int node)` Given a target node, return the distance of the shortest path from the source node as described above to this target node. It is guaranteed that `computeShortestPaths` is called before the first call of `getDistance`. This method can be called multiple times for one graph. It should run in $O(1)$ for each call.