

# CMPUT 275 Wi18 - INTRO TO TANGIBLE COMPUT II

## Combined LBL Wi18

### Exercise 3: Bellman Ford and Graph Potentials

The purpose of this exercise is to:

- Learn and implement yet another (very simple) algorithm for finding least-cost paths in graphs. This algorithm can handle negative-cost edges, unlike Dijkstra's (without extra work). It just runs slower, which is why we cannot use it for the driving route finder.
- Understand that we do not always need to build and use a big graph class to process graph data.
- Learn how to build the vertex function from Dijkstra's worksheet so we can use Dijkstra's algorithm in graphs that might have negative-cost edges, as long as there are no negative-cost cycles.

**Note:** you have two implementation tasks mentioned below. Pay close attention to the code submission instructions below.

## 1) The Bellman-Ford Algorithm

Given a graph  $G = (V; E)$  and a **start vertex**  $s$  where each edge  $uv$  has a **cost**, say  $cost(u, v)$ , we often want to find a least-cost path from  $s$  to other reachable vertices. In some cases costs may be negative!

Such an instance may occur when, say, edges represent gains or losses in energy. For example, edges could represent gains or losses in energy when driving some electric cars: braking while going downhill may gain energy.

If the graph has cycles whose total edge cost is negative, then this may not be well-defined. We could just run around such cycles forever to make the path arbitrarily cheap. Of course, in real life this doesn't really happen: I expect most if not all "natural" graphs we actually want to find paths in will not have negative cost cycles. In this part, **you may assume the graphs to not have negative-cost cycles**.

The **Bellman-Ford algorithm** finds shortest paths from a vertex  $s$  to all other reachable vertices in  $O(|V| * |E|)$  time. It maintains two dictionaries:

```
dist - Keys are vertices. dist[v] stores a value, perhaps infinity
reached - Same as with all other searches. Maps vertices to predecessors on the search tree
```

At each step of the algorithm, the reached dictionary is a search tree to some reachable vertices. But it may not represent shortest paths until the end of the algorithm, and may even change its edges over time to vertices that are already reached.

The pseudocode is very simple:

```
dist <-- dictionary mapping s to 0 and every other vertex to infinity
reached <-- dictionary mapping s to s, and nothing else

for |V|-1 iterations do
    for each edge (u,v) do
        if dist[v] > dist[u] + cost(u,v)
            dist[v] = dist[u] + cost(u,v)
            reached[v] = u
```

At the end, reached will be a search tree representing minimum-cost paths and `dist[v]` will be the cost of this minimum-cost path from `s` to `v` (still infinity if `v` was not reached).

Why does this work? You can prove that after  $k$  iterations that `dist[v]` is at most the cost of the cheapest path to `v` using at most  $k$  edges. This is why  $|V|-1$  iterations suffice.

### Your First Job

Implement the Bellman-Ford algorithm as the following function. Note you **should not** use the Graph class we developed. Adhere exactly to the specification.

```
def bellman_ford(vertices, edges, start):
    '''
    Computes shortest paths to every reachable vertex from the vertex "start"
    in the given directed graph.

    vertices: the set of vertices in the graph.
    edges: maps pairs of vertices to values representing edge costs
           example - {'A', 'B'}: -3} means the edge from vertex
                       'A' to vertex 'B' has cost -3
    start: the start vertex to search from

    Assumes the graph does not have negative cost cycles,
    that all edges have endpoints in "vertices", and that
    "start" is also in "vertices".

    returns dist, reached

    Here reached is the search tree to all reachable vertices along
    minimum-cost paths and dist[v] is the cost to v along
    this path. If v is not reachable, it should not be in the
    search tree nor an index in dist.
    '''
```

### Helpful tips:

1. The floating point type actually supports the value infinity: `float('inf')`. It works as you expect: infinity plus any finite number is still infinity (though infinity - infinity is not defined). Use it at your discretion (i.e. you are not required to use it).
2. If you define entries in `dist` for vertices that are not reachable, remember to remove them before returning from the `bellman_ford` function.

Recall from the Dijkstra's worksheet that if we can find values  $p[v]$  for each vertex  $v$  such that  $p[v] + \text{cost}(u,v) \geq p[u]$  then we can run Dijkstra's with modified edge costs  $\text{cost}(u,v) + p[v] - p[u]$  and find shortest paths even if some values  $c(u,v)$  are negative. But it did not specify how to find these values  $p[v]$ , which are called **potential values**.

We find them using a slight modification of the Bellman-Ford algorithm. The modifications are as follows:

- there is no reached dictionary
- there is no start vertex
- initially  $\text{dist}[v] = 0$  for every vertex  $v$

Then run the main loop of the Bellman-Ford algorithm for  $|V| - 1$  steps. The claim is that afterward the values  $-\text{dist}[v]$  for each vertex  $v$  are potential values if and only if the graph does not have a negative-cost cycle.

### Your Second Job

Implement the function to either find a potential or determine the graph has a negative-cost cycle. That is, run the above algorithm and then check that the  $-\text{dist}[v]$  values are indeed potentials. If not, return None. Otherwise, return these potential values as a dictionary.

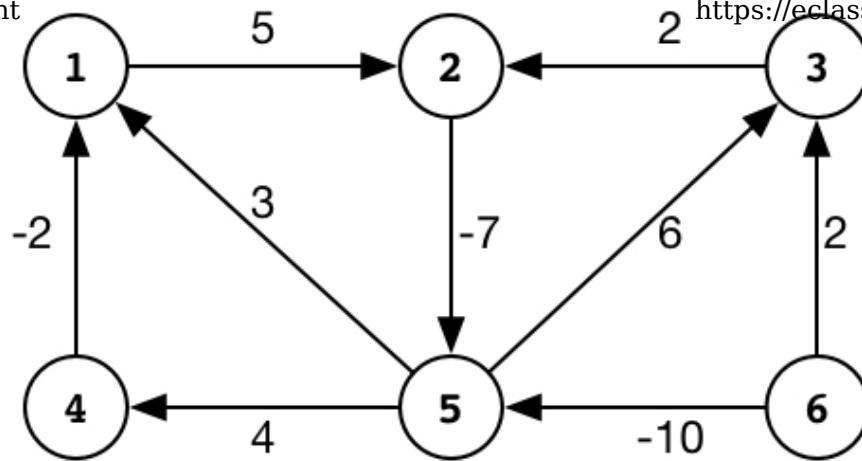
```
def find_potential(vertices, edges):
    '''
    Finds a potential for the graph or determines the graph has
    a negative-cost cycle.

    vertices: the set of vertices in the graph.
    edges: maps pairs of vertices to values representing edge costs
           example - {'A', 'B': -3} means the edge from vertex
                    'A' to vertex 'B' has cost -3
    start: the start vertex to search from

    If the graph has a negative-cost cycle, this simply returns None.
    Otherwise, it returns a dictionary mapping each vertex to its value
    in a potential function.
    '''
```

### Helpful Tests

The following graph may be helpful to test on. Examples of some function calls with this graph are given below.



Here is an example of using this graph in the functions.

```

>>> vertices = {1, 2, 3, 4, 5, 6}
>>> edges = {(1,2):5, (2,5):-7, (3,2):2, (4,1):-2, (5,1):3,
              (5,3):6, (5,4):4, (6,3):2, (6,5):-10}
>>> dist, reached = bellman_ford(vertices, edges, 1)
>>> print(dist)
{1: 0, 2: 5, 3: 4, 4: 2, 5: -2}
>>> print(reached)
{1: 1, 2: 1, 3: 5, 4: 5, 5: 2}
>>> print(find_potential(vertices, edges))
{1: 8, 2: 3, 3: 4, 4: 6, 5: 10, 6: 0}
>>> edges[(5,4)] = 3 # creates a negative-cost cycle
>>> print(find_potential(vertices, edges))
None

```

### Code Submission

Submit a single file called `bellman_ford.py` that includes the implementation of these two functions and any functions you think might help. You **must** add some docstring tests to test these two functions. We will run these tests via

```
python3 -m doctest -v bellman_ford.py
```

It is fine if you include these tests in the function documentation itself. Alternatively, you can just test both functions in the docstring tests for the entire module (because typing out the graph description more than once can be a pain). See this page for an example of using docstring tests for a whole module.

Submit only the file `bellman_ford.py`, do not zip it. The above graph is ok to have as a test, but you must create at least two more graphs to test on



### Tips for Testing

Recall that we do not generally have control over which path is found so it would be sufficient to ensure the reached dictionary has its keys being the reached vertices. But you can still test that the


dist dictionary has the correct distances. Remember that doctest tests only check that the output matches the expected output exactly (character for character), so it is tricky with dictionaries and sets where we don't have control over the order things will be printed. I like to test things as follows:

```
>>> dist[5]
-2
>>> reached.keys == {1, 2, 3, 4, 5}
True
>>> potential[5] + edges[(2,5)] >= potential[2] # you don't have to do this for all edge
s if you don't want, just a few is ok
True
```

### Submission status

Attempt number	This is attempt 1 ( 1 attempts allowed ).
Submission status	Submitted for grading
Grading status	Graded
Due date	Monday, 5 February 2018, 11:55 PM
Time remaining	Assignment was submitted 8 hours 12 mins early
Last modified	Monday, 5 February 2018, 3:42 PM
File submissions	<div> bellman_ford.py </div> <div>Export to portfolio</div>
Submission comments	► Comments (0)

### Feedback

Grade	100.00 / 100.00
Graded on	Monday, 12 February 2018, 8:09 PM
Graded by	 Jason Cannon
Feedback comments	<b>Test Cases Passed: (50/50)</b> Bellman Ford - (25/25 passed) Graph Potentials - (25/25 passed) <b>Final Mark: 100.0%</b>