Graph Algorithms (PR on 04-2-2016 and 10-02-2016)

Note that **each student** must be able to present his/her own working version of Bellmann-Ford and Dijkstra's algorithm for shortest path computations in digraphs on 10-02-2016. That is Exercises 1 and 2 below. This will be individually checked in the PR session on 10-02-2016.

In this practical session we will implement graph algorithms for computing *shortest paths*. For this, you should use the *tools* that have been introduced in the previous session:

- The `basicgraphs.py` module for creating and working with graph objects.
- The `graphIO.py` module for reading and writing graphs, and for visualizing graphs.

See the previous exercise sheet for detailed information on how to use these.

For this practical session, a new module `spst.py` *(Shortest Paths and Spanning Trees)* is given, which imports the above modules, contains four functions that should be filled in, contains some options and test code, and takes care of input and output.
In addition, a set of test instances is given on blackboard (edge weighted graphs, which can be viewed as undirected or directed graphs). These instances should be copied into your *working directory* (the one where also all your python modules are stored, including `basicgraphs.py` and `graphIO.py`).

The methods of the module `basicgraphs.py` are mainly intended for working with *undirected* graphs, but internally all edges are directed, so you can easily use it to implement *directed* graph algorithms.
For example: calling the `inclist` method for a vertex `v` gives all edges that are incident with `v`, without regard for their direction. However, by only using those edges `e` in this list that satisfy

```
e.tail()==v
```

you can consider only the *outgoing* edges of v (which is what you want to do for the directed version of Dijkstra's algorithm).

Exercise 1: *(Bellman-Ford)* In this exercise, we will implement the Bellman-Ford algorithm for computing distances and shortest paths from a given vertex in the graph. By default, use vertex 0 to start.

(a) Implement the *directed* version of the Bellman-Ford algorithm, in the given function `BellmanFordDirected`.

Loop over all edges `e` of the graph, and consider their end vertices by calling `e.tail()` and `e.head()`. The *weight* of an edge `e` is stored in its attribute `e.weight`.

Store the *distance* of a vertex `v` in its attribute `v.dist`. In addition, you may keep track of the edges that are part of a shortest path by letting the vertex attribute

`v.inedge` point to the incoming shortest path edge. (This attribute is used for visualization of the result.) Note that distance attributes are initialized with the value `None`, which should be considered 'infinity'.

Test your implementation using the graph `weightedexample.gr` (by setting the options `TestInstances=["weightedexample.gr"]` and `TestBellmanFordDirected=True`).

If you use the option `WriteDOTFiles=True`, the program will write a `.dot` file called `BellmanFordDirected.dot`, which can be opened in an appropriate viewer to visualize and verify the results. (See last week's sheet.)

(b) (Optional) Implement the *undirected* version of this algorithm, in the given function `BellmanFordUndirected`. Edges should now be viewed as undirected (or equivalently, 'bidirected') edges, so distances should be updated in both directions. Similar to before, test and view the results using the graph `weightedexample.gr`. (Set `TestBellmanFordUndirected=True`. The output file will be called `BellmanFordUndirected.dot`.)

Remark: Note that, if implemented cleverly, the undirected version is the same as the directed version, only that you do not need to bother with `e.tail()` and `e.head()`. So you might want to introduce a boolean `directed`, as an argument of your function, which you can use to decide if you consider only the directed edge, or both directions.

Exercise 2: *(Dijkstra)* In this exercise, you implement Dijkstra's algorithm for computing distances and shortest paths from a given start vertex in the graph. By default, use vertex 0 to start.

(a) Implement the *directed* version of the algorithm, in the given function `DijkstraDirected`.

Use the `inclist` method to obtain all edges that are incident with a given vertex, for example, by typing `for e in v.inclist(): ....` You can get the other vertex of tt e via `w = e.otherend(v)`, and like before, you get access to the tail and head of a directed edge `e` by using `e.tail()` and `e.head()`. For instance, you can check if an edge `e={v,w}` is indeed directed from `v` to `w` by typing `if (w == e.tail()): ....` The *weight* of an edge `e` is stored in its attribute `e.weight`. Again, use `dist` and (optionally) `inedge` attributes of vertices to store their distance and incoming shortest path edge.

The main issue here is how to efficiently find the next vertex for which the distance label can be 'frozen'. For this, you may use the slow (yet simple) solution of linearly searching through a list of all candidates.

Test your implementation using the graph `weightedexample.gr` (by setting the option `TestDijkstraDirected=True`), and view the results in an appropriate viewer.

(b) (Optional) Also implement the *undirected* version of this algorithm, in the given function `DijkstraUndirected`. Similar to before, test and view the results using the graph `weightedexample.gr` (set `TestDijkstraUndirected=True`).

Again, if implemented cleverly, the undirected version is the same as the directed version, only that you do not need to bother with `e.tail()` and `e.head()`. So you

might want to introduce a boolean `directed`, as an argument of your function, which you can use to decide if you consider only the directed edge, or both directions.

(Optional) Also test your shortest path algorithms on the graph `randomplanar.gr`, and view the results.

Exercise 3: *(Negative edge weights)* Dijkstra's algorithm will not give the correct result when *negative edge weights* are present in the graph, but the Bellman-Ford algorithm will, provided that there are no *negative length cycles* (in particular, this only works for directed graphs, since undirected/bidirected edges can be seen as length two cycles). This is no weakness of the Bellman-Ford algorithm, since distances are simply not well-defined in the presence of negative cycles.

(a) Test your two shortest path algorithms for directed graphs on the example `negativeweightexample.gr` (use the `TestInstances` option), and visually verify this behavior. Try to understand why Dijkstra's algorithm fails.

(b) Bellman-Ford can in fact be used to detect the presence of negative weight cycles in directed graphs: a negative cycle is present iff after $n-1$ iterations, an update to the distances is still possible for some edge. Extend your algorithm to include such a test, and verify it using `negativeweightcycleexample.gr`.

Exercise 4: *(Optional / Advanced Material on Time Complexity)*

(a) Test the speed of your (directed/undirected) shortest path implementations on the larger examples `WDE100.gr` – `WDE2000.gr`, and estimate the complexity of your implementations. Does the algorithm behavior match the theoretical worst case complexity?

(The number in the filename indicates the number of vertices n, and the number of edges m is always $10n$.)

Important: Set `WriteDOTFiles=False`, since trying to view such large graphs may crash your computer!

(b) The *worst case* complexity of the Bellman-Ford algorithm cannot be improved by a smart implementation trick, but it is possible to speed it up significantly on many instances: change your implementation such that the main for loop terminates when in one iteration, no distance has been updated. (Why is it safe to do this?)

Test out the effect of this improvement on the same examples again.

In addition, test on the example `bbf2000.gr`, to see that the worst case complexity does indeed not improve.

(c) The implementation of Dijkstra's algorithm can be improved significantly. In the module `basicgraphs.py`, the `inclist` method of vertices simply iterates over all edges (in time $O(m)$), and returns those that are incident with the given vertex. So with this implementation, the worst case complexity of Dijkstra's algorithm is also $O(nm)$.

Improve this by starting the algorithm with *one for loop* over all edges, in which the incidence lists of *all* vertices are constructed. Use these lists in the main algorithm[1].

Test out this improvement on the same examples again.

(If you did it correctly, the complexity is now $O(m + n^2)$ or better – why?)

(d) *Discussion:* Probably you have not implemented the selection of the next vertex to be 'frozen' optimally yet, and this still requires time $O(n)$ in the worst case, which explains the n^2 term in the complexity.

This can be improved by storing all candidate vertices in a *(binary) heap* or *priority queue*. See the ADS lectures for more information. This way, the complexity of Dijkstra's algorithm can be improved to $O(m + n \log n)$, where the $\log n$ factor results from the heap implementation.

However, implementing such a heap structure would be too much effort for now.

[1] Alternatively, you may take care of this in your *graph class* implementation – see the discussion on `fastgraphs.py` on last week's exercise sheet; exercise 6(b).