Discrete Mathematics Exercises 1: Working with Graphs (6.2.2015)

This practical session introduces tools that you can use for the main project. Therefore, some extra remarks and hints are given on this sheet, which will be useful later.

Exercise 1: *A module and interface for working with graphs:* Download the module `basicgraphs.py` from blackboard. This contains three *classes* for working with *undirected* graphs (both simple graphs and multigraphs): *vertex*, *edge* and *graph*.

(a) Study the interface of this module by reading the *documentation strings (docstrings)* in the file, and try this module out at the interactive prompt by typing commands such as:

```
>>> from basicgraphs import graph
>>> G=graph(4)
>>> G
>>> G[0]
>>> u=G[0]
>>> v=G[1]
>>> G.adj(u,v)
>>> G.addedge(v,u)
>>> G.adj(u,v)
>>> G
>>> w=G.addvertex()
>>> e=G.addedge(u,w)
>>> e.tail()
>>> e.head()
>>> e.otherend(u)
>>> G.V()
>>> G.E()
>>> u.deg()
>>> u.nbs()
>>> x=G.addvertex('X')
>>> G
>>> x
```

Remark: To keep the output readable, vertices are simply represented by their *label*, which is an integer by default. You should however keep in mind that vertices and edges are always custom objects of the types *vertex* and *edge*!

If you want to confuse yourself, you can choose the same label for different vertices (but this is not recommended).

(b) Write a small module that imports `basicgraphs.py`. This new module should contain functions for creating *paths*, *cycles*, and *complete graphs* on a given number of vertices.

Test out your module either at the interactive prompt, or by adding 'test code' at the end of it.

(c) Extend your module with a function `disjointunion` that takes as argument two graphs G and H, and returns a new graph that is the disjoint union of G and H.

(Suggestion: use a dictionary for mapping vertices of G and H to vertices of the new graph. Recall that 'custom objects' such as vertices can be used as dictionary keys.)

Exercise 2: *Input/output:* Download the module `graphIO.py` and the text file `examplegraph.gr` from blackboard. The first is a module that can *read/write* graph objects from/to *text files* (and from *stdin*/to *stdout*). The format of these text files is very simple and easily readable/editable; see `examplegraph.gr`.
It is not necessary to understand how this module works, but you should understand how to use it: to this end, do the following exercise, and you may later read the *documentation strings* in the code.

(a) At the interactive prompt, use `graphIO.py` to read `examplegraph.gr`, add a few edges and vertices, and write the result to a new file `examplegraph2.gr`, and compare these text files. You can for instance type:

```
>>> from graphIO import loadgraph,savegraph
>>> G=loadgraph('examplegraph.gr')
>>> G
>>> G[0].nbs()
>>> G.addedge(G[0],G[3])
>>> savegraph(G,'examplegraph2.gr')
```

(b) Write a small module `complement.py` that loads a graph G from a text file, computes the *complement* $\overline{G}$, and then writes this to a new text file.

(The *complement* of a simple graph $G = (V, E)$ has the same vertex set V, and for every pair of vertices $u, v \in V$, has an edge uv if and only if $uv \notin E$.)

(Remark: The module `graphIO.py` also contains functions `inputgraph,printgraph` for reading and writing graphs from stdin and stdout, respectively. See the documentation in the program for more information.)

Exercise 3: *Visualization:* The module `graphIO.py` also contains a function `writeDOT` that can generate a `.dot` file from a graph object G. This function should be called as follows, using a `graph` object `G`:

```
writeDOT(G,'mygraph.dot')
```

`.dot` files are in a format that is useful for *visualizing* graphs, compatible with the *GraphViz* package.

(a) Use the above command to generate a `.dot` file from `examplegraph.gr`. Open this file in a text editor. Copy/paste this text to one of the following sites, and try the different visualization options:

```
http://sandbox.kidstrythisathome.com/erdos/
http://graphviz-dev.appspot.com/
```

(b) **(Optional/later:)** There are many programs for visualizing `.dot` files. Some such as *dotty* may already be installed in your OS (in linux - try out this command). For a complete overview/introduction to GraphViz, see:

```
http://www.graphviz.org/
http://www.linuxdevcenter.com/pub/a/linux/2004/05/06/graphviz_dot.html
```

Two other useful programs are *ZGRViewer* (Java) and *xdot* (Python 2.x), see:

```
http://zvtm.sourceforge.net/zgrviewer.html
https://github.com/jrfonseca/xdot.py
```

You can install/try out different programs. (Though it is recommended to continue first with the other exercises!)

Exercise 4: *Algorithms on graphs:* Test the following algorithms on `examplegraph.gr`, starting your search on the first vertex $G[0]$.

(a) Program a *Breadth-First Search (BFS)* algorithm. Use this to:

- test whether a given graph is connected,
- compute distances from a given vertex (unit edge weights), and
- label vertices in the order they are visited.

Use a *queue* to store the vertices that should be visited next. (A queue can be implemented using a Python list, or using a Doubly-Linked List for more efficiency.)

(b) Modify your program such that for a vertex v, the 'visiting order' label is stored in the attribute `v.label`. After assigning these labels, write the graph to a `.dot` file, and visualize the result.

(Note that the `writeDOT` function will write these label attributes to the `.dot` file - just like the attributes `v.colortext`, `v.colornum` and `edge.weight`, which should be a string (like "Blue"), an integer and an integer, respectively. For colorful pictures, you can assign `v.colornum=v._label` for every vertex v.)

(c) Program a *Depth-First Search (DFS)* algorithm, by changing the *queue* to a *stack* (and possibly making some more changes depending on your implementation). Visually compare the results, and see that the 'visiting order' labels are completely different.

(d) **(Optional:)** Program an alternative DFS algorithm, by using *recursion*, so without explicitly storing the vertices that should still be visited. Compare this with your previous algorithm.

Remark: Even if you programmed both DFS versions correctly, the visiting order is probably different! Can you explain why?

Exercise 5: *Modifying graphs:* We will now extend `basicgraphs.py`; copy this module and call it e.g. `mygraphs.py`.

(a) Add a method `deledge(self,edge)` to the `graph` class in `mygraphs.py`, that deletes an edge `edge`.

The second argument of this method should be an object of the class `edge`. A `graph` object G has an attribute `G._E` that contains a list of all its edges. This is the only attribute that needs to be changed by your new method.

(b) Add a method `delvertex(self,vertex)` to the `graph` class in `mygraphs.py`, that deletes a vertex `vertex`.

The second argument of this method should be an object of the class `vertex`. Note that a `graph` object G has an attribute `G._V` that contains a list of all its vertices. This is the only attribute that needs to be changed by your new method, *after* deleting all edges that are incident with `vertex`.

Test your code at the interactive prompt, or by visualizing the results.

Remark: `graphIO` uses the graph class from the module `basicgraphs` as default. However, even if you give your improved graph module a different name, e.g. `mygraphs.py`, you can still use `graphIO` to read graphs for you[1]: use the optional second argument of the method, as follows:

```
>>> import mygraphs
>>> from graphIO import loadgraph
>>> G=loadgraph('graph2000.gr',graphclass=mygraphs.graph)
```

Exercise 6: *Complexity and better data structures:*

(a) Try out your DFS algorithm on the graphs `graph1000.gr`, `graph2000.gr`, `graph4000.gr` and `graph8000.gr`, which can be found on blackboard.

Measure the time your algorithm takes on these graphs. (See the last slides of the Python lecture.)

Remarks:

- Measure *only* the time of your DFS algorithm, because the `inputgraph` or `loadgraph` routine from `graphIO` will probably be the bottleneck in the total complexity!
- Do not use a recursive version of DFS, since Python will return a *Maximum Recursion Depth Exceeded* error on the larger three graphs.

The size of these graphs grows by a factor 2 each time (so `graph2000.gr` has exactly twice as many vertices and edges as `graph1000.gr`, and nearly all of them will be visited by DFS). DFS should run in *linear time*, but the measured time does not grow linearly (i.e. by factors of 2) – *What is wrong here?*
The answer is that `basicgraphs.py` is not implemented very smartly: only a list of vertices and a list of edges are stored. You can improve the performance of the module by keeping track of a list of all *adjacent vertices* or all *incident edges* for every vertex v *(adjacency lists / incidence lists)*. To this end:

[1] assuming your new class still supports the methods `addedge` and `__getitem__`!

(b) **(Optional/later:)** Make your own extension `fastgraphs.py` of the module `basicgraphs.py`, that also stores *incidence lists* for each vertex:

Modify the methods `vertex.__init__` and `graph.addedge` to initialize/change these lists, and then you can improve the efficiency of the methods `vertex.inclist` and `graph.adj`.

Test your module, and verify that using the improved module, your DFS algorithm runs in linear time[2].

Conclusions/summary:

- The `basicgraphs.py` module defines a good *interface* for working with graphs.
- It does however not use very good *data structures* internally.
- If you do it correctly, you can improve the data structures, without changing the interface.
- The best choice of data structure depends on the algorithms you use. For DFS and BFS *adjacency lists* or *incidence lists* work well, but for some other algorithms *adjacency matrices* are best.
- If you keep supporting this interface in your own graph modules, you can use `graphIO.py` for input, output and visualization of graphs.
- Internally, edges are already stored as 'directed edges' with a head and a tail. It is straightforward to generalize the module such that it can also work with directed graphs.

[2]Actually, the computation time still grows by a factor slightly larger than 2. This is because of Python's internal workings...