Cuda Louvain HPC project

foreword

I first implemented a sequential algorithm and then tried to parallelize phase 1 with cuda, much lik
that the approach to transfer all data once to the device and then work with it would be much bette
I would say that now is the perfect moment to sit down and rewrite all of this code in a neat way, b
this semester. That being said, it works! unfortunately I wasn't able to really dig into it and play witl
limit, which I strongly regret.

Implementation

graph

the graph is kept as list of edges and list of indexes that are the last exclusive index of an vertice's
lastExclusiveEdgeIndex[vertice-1] to lastExclusiveEdgeIndex[vertice].

phase I

the graph is moved to the device and then we enter a loop. In the loop dQ is calculated in parallel f
moves are applied to see if the modularity will increase above the specified min-gain, if not then a
the best move you still dont achieve the specified minimal gain in modularity (aka. Q) then loop fin

phase II

we want to aggregate the graph, so we take all the edges and in every community the minimal vert
of the edges are changed so that they go in and out of this "super vertice" representing this comm
they are sorted, and merged and sorted again.

This is another great place for parallelization, however due to lack of time...

quality tests

why do you think your implementation is correct?

Well... I have tested it by hand with debugger. Also I have made the sequential implementation res
calculate all the values in batches, so I was able to check all the values every step of the way.

Then I tested it with the code below. I have run numerous examples to check if I achievie the same

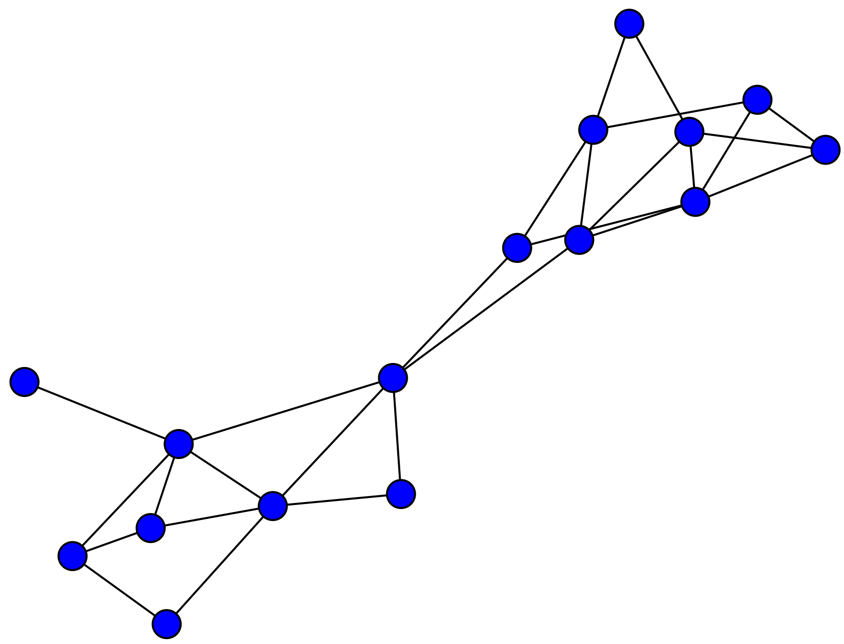
```
from IPython.display import SVG, display
import numpy as np
from sknetwork.data import karate_club, painters, miserables
from sknetwork.clustering import Louvain, BiLouvain, modularity, bimodularity
from sknetwork.linalg import normalize
from sknetwork.utils import bipartite2undirected, membership_matrix, edgelist2adja
from sknetwork.visualization import svg_graph, svg_digraph, svg_bigraph
from matplotlib import pyplot as plt
```

```
from matplotlib import pyplot as plt

edgelist=[(0,2,1.0), (0,3,1.0), (0,4,1.0), (0,5,1.0), (1,2,1.0), (1,4,1.0), (1,7,1
adjacency = edgelist2adjacency(edgelist)


louvain = Louvain()
labels = louvain.fit_transform(adjacency)
image = svg_graph(adjacency, labels=labels)
mylabels = np.array([0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1])
image2 = svg_graph(adjacency, labels=(mylabels != labels) )
print("theirs",modularity(adjacency, labels))
print("mine", modularity(adjacency, mylabels))
SVG(image2)
```

```
theirs 0.42798353909465026
mine 0.42798353909465026
```



performance tests:

how fast is your program? what is the influence of the optimizations you did?

again, time... I tested it only on very few examples and I was able to see "with the naked eye" that t ~10k vertices and 100k edges. There is one flaw however that I did not yet fixed. The algorithm as dQ in phase one, so if it exceeds that at eny point it throws an error.