

Homework assignment 3

Author: Vendramin Nicolò

The task has been fulfilled with a python script made up of a class containing a class Triest that includes the following methods:

A. Function - "__init__":

```
01. def __init__(self, M, input_file, dynamic=False):
02.     self.M = M
03.     self.graph = self.import_graph(input_file, dynamic)
04.     self.t_counter = 0
05.     self.t_dictionary = {}
06.     self.s = []
    self.di = 0
    self.do = 0
    self.avg_term = 0
```

This function simply sets the parameter of the class and the variables used in the computation of the algorithm. Just a standard __init__ function.

B. Function - "import_graph":

This function just iterates over the dataset to import the edges as a sequence in order to be able to stream while the algorithm is running. In case the dynamic parameter is set to True the function make_graph_dynamic is called to add also edge deletions to the stream.

```
01. def import_graph(self, filename, dynamic=False):
02.     file = open(filename, "rU")
03.     line = file.readline()
04.     graph = []
05.     while len(line)>0:
06.         edge = line.split(" ")
07.         graph.append((edge[0], edge[1]))
08.         line = file.readline()
09.     file.close()
10.     number_of_edges = len(graph)
11.     print(number_of_edges)
    if dynamic:
        return self.make_graph_dynamic(graph)
    return graph
```

C. Function - "make_graph_dynamic":

```
01. def make_graph_dynamic(self, graph):
02.     graph = [(l, edge) for edge in graph]
03.     return graph
04.     graph_ = list(graph)
05.
06.     for i in range(0, len(graph)):
07.         if self.flip_biased_coin(DELETION_PROPORTION, 100):
08.             new_ = (-1, graph[i][1])
09.             pos = np.random.randint(i, len(graph)+1)
10.             graph_.insert(pos, new_)
    return graph_
```

This function randomly selects edges from the streamed graph with a certain probability and adds to the stream a tuple representing the deletion of the same edge in a random step after the one in which it was inserted. In this

way a stream containing only edge insertions becomes a fully dynamic graph stream. This function is called by import graph in case the fully dynamic version of the triest algorithm is used.

D. Function - "flip_biased_coin":

D

```
def flip_biased_coin(self, favorable, possible):
    return np.random.randint(0, possible) < favorable
```

This function just represents the flip of a biased coin with favorable/possible probability of head.

E. Function - "get_neighbors":

```
01. def get_neighbors(self, edge):
02.     n1 = []
03.     n2 = []
04.     for edge_ in self.s:
05.         if edge_[0] == edge[0]:
06.             n1.append(edge_[1])
07.         if edge_[1] == edge[0]:
08.             n1.append(edge_[0])
09.         if edge_[0] == edge[1]:
10.             n2.append(edge_[1])
11.         if edge_[1] == edge[1]:
12.             n2.append(edge_[0])
13.     return set(n1) & set(n2)
```

E

This function retrieves the neighbours of an edge intended as the intersection of the neighbourhoods of the two vertices composing the edge.

The following are the functions actually explained and illustrated in the paper referenced above. I am going to explain them one by one and later to illustrate how to combine them to get the different versions of the Triest algorithm.

F. Functions - "sample_edges" & "sample_edges_improved":

```
01. def sample_edges(self, edge, time_step):
02.     if time_step <= self.M:
03.         return True
04.     elif self.flip_biased_coin(self.M, time_step):
05.         pos = np.random.randint(len(self.s))
06.         random_edge = self.s.pop(pos)
07.         self.update_counters(random_edge, operation=-1)
08.         return True
09.     return False
10.
11.
12. def sample_edges_improved(self, edge, time_step):
13.     if time_step <= self.M:
14.         return True
15.     elif self.flip_biased_coin(self.M, time_step):
16.         pos = np.random.randint(len(self.s))
17.         random_edge = self.s.pop(pos)
18.         return True
19.     return False
```

F

These two functions represent the reservoir sampling algorithm for the two cases of the triest base and improved. In both cases the edge is sampled if there is still space in the memory, otherwise is kept with a probability equal to the ratio between the time step and the total available size of memory. If an edge is sampled it replaces a random one from the set of

saved edges and the counters are updated negatively in the base version while no negative update is done in the improved version.

G. Function - "sample_edges_fully_dynamic":

This is the version of the reservoir sampling for the general case in which the streamed graph contains both insertion of edges and deletions. The general functioning is the same with the condition that now also exists the possibility of finding a deletion.

```

01. def sample_edges_fully_dynamic(self, edge, time_step):
02.     if self.do + self.di == 0:
03.         if len(self.s) < self.M:
04.             self.s.append(edge)
05.             return True
06.         elif self.flip_biased_coin(self.M, time_step):
07.             pos = np.random.randint(len(self.s))
08.             random_edge = self.s.pop(pos)
09.             self.update_counters(random_edge, operation=-1)
10.             self.s.append(edge)
11.             return True
12.         elif self.flip_biased_coin(self.di, self.di + self.do):
13.             self.s.append(edge)
14.             self.di -= 1
15.             return True
16.         else:
17.             self.do -= 1
18.             return False

```

In case the d_0 and d_i variables sums to zero the procedure is the same of the normal reservoir sampling of the base case. Otherwise with a random probability equal to the ratio between d_i and the sum of d_i and d_0 the edge is sampled and the d_i counter decreased and in the remaining cases d_0 is decremented and the edge is not sampled.

H. Function - "update_counters":

```

01. def update_counters(self, edge, operation=1):
02.     neighbors = self.get_neighbors(edge)
03.
04.     if edge[0] not in self.t_dictionary.keys():
05.         self.t_dictionary[edge[0]] = 0
06.     if edge[1] not in self.t_dictionary.keys():
07.         self.t_dictionary[edge[1]] = 0
08.
09.     for neighbor in neighbors:
10.         self.t_counter += 1 * operation
11.         self.t_dictionary[edge[0]] += 1 * operation
12.         self.t_dictionary[edge[1]] += 1 * operation

```

This function is used in the base and fully dynamic version of the algorithm. It receives as input an edge and an operation encoded as 1 = + and -1 = -.

The shared neighbours of the edge are obtained through the `get_neighbors` function and for each

common neighbour the counters are updated with the operation specified as input parameter of the function.

I. Function - "update_counters_improved":

```

01. def update_counters_improved(self, edge, time_step, operation=1):
02.     neighbors = self.get_neighbors(edge)
03.
04.     if edge[0] not in self.t_dictionary.keys():
05.         self.t_dictionary[edge[0]] = 0
06.     if edge[1] not in self.t_dictionary.keys():
07.         self.t_dictionary[edge[1]] = 0
08.
09.     n = max(1, ((time_step - 1) * (time_step - 2) / (self.M * (self.M - 1))))
10.
11.     for neighbor in neighbors:
12.         self.t_counter += n * operation
13.         self.t_dictionary[edge[0]] += n * operation
14.         self.t_dictionary[edge[1]] += n * operation

```

The update counters function for the improved version of the algorithm is slightly modified. As a matter of fact the counters at each step are updated not by adding (or subtracting 1, but a quantity n that is the maximum between one and the quantity $(t-1) *$

$(t-2)$ divided by $M * (M-1)$ where t is the current time step and M the total available memory space.

Here follow the descriptions of the three function to be used to call the three different versions of the algorithm, in which it appears which combination of the above mentioned functions it's used.

J. Function - "triest_base":

This function implements the base version of the Triest algorithm as specified in the pseudocode included in the paper. It goes through the edges of the streamed graph, incrementing at each passing edge the the time step counter is increased and the base version of the reservoir sampling is included to decide wether to keep or not the edge. If the edge is kept the counters are updated by calling the `update_counters` function.

```

01. def triest_base(self):
02.     t = 0
03.     pbar = ProgressBar()
04.     for edge in pbar(self.graph):
05.         t += 1
06.         if self.sample_edges(edge, t):
07.             self.s.append(edge)
08.             self.update_counters(edge)
09.             epsilon = max(1, (t * (t - 1) * (t - 2)) / (self.M * (self.M - 1) * (self.M - 2)))
10.             self.avg_term = (float((t - 1) * self.avg_term + 1 * self.t_counter * epsilon)) / t
11.         return self.t_counter

```

K. Function - "triest_improved":

```

01. def triest_improved(self):
02.     t = 0
03.     pbar = ProgressBar()
04.     for edge in pbar(self.graph):
05.         t += 1
06.         self.update_counters_improved(edge, t)
07.         if self.sample_edges_improved(edge, t):
08.             self.s.append(edge)
09.         return self.t_counter

```

The improved version is equal to the base one but for the fact that the update counters function that is used is the improved version and that the update is called unconditionally even if the edge is not sampled.

L. Function - "triest_improved":

```

01. def triest_fully_dynamic(self):
02.     t = 0
03.     s = 0
04.     pbar = ProgressBar()
05.     for opt, edge in pbar(self.graph):
06.         t += 1
07.         s += 1 * opt
08.         if opt == +1:
09.             if self.sample_edges_fully_dynamic(edge, t):
10.                 self.update_counters(edge)
11.             elif edge in self.s:
12.                 self.update_counters(edge, -1)
13.                 self.s.remove(edge)
14.                 self.di += 1
15.             else:
16.                 self.do += 1
17.         return self.t_counter

```

This third version implementing the fully dynamic Triest algorithm is slightly modified because now the stream is made up of tuples operation (+, -) and edge. The function goes through the stream and if it's an insertion, it calls the sample_edges_fully_dynamic function to apply the reservoir sampling and in case the result it's true it updates the counters.

In case the operation is a deletion, if the edge is in the set of saved edges the counters are negatively updated, the edge is removed from the saved edges and the di counter is increased, otherwise the do counter is increased and nothing else is done.

M. Imports and usage:


To run the code is just necessary to set up an environment with Python 3.5, install the progressbar, argparse and numpy external libraries, open a console in the folder of the project and run the command:

"python triest.py"

With the following argument options:

- version: is a string value expressing the desired version of the algorithm to be run. Accepted values are "base", "improved" and "fd". Default value is "base".

- —bound: is an integer indicating the maximum number of edges that the algorithm is allowed to save in memory. The variable M referenced by the paper. The default value is 10000.
- —input_file: is a string indicating the relative path to the input file. Default is "data.txt".



```

01. import numpy as np
02. from progressbar import ProgressBar
03. import time
04. import argparse
05.
06. parser = argparse.ArgumentParser()
07. parser.add_argument('--version', type=str, default="base")
    parser.add_argument('--bound', type=int, default=10000)
    parser.add_argument('--input_file', type=str, default="data.txt")
    args = parser.parse_args()
  
```

N. Bonus task - questions:

- 1 What were the challenges you have faced when implementing the algorithm?

I found the implementation of the algorithm pretty straight forward thanks to the good explanation present in the paper and the availability of pseudocode in the same. The only design choice were limited to how to represent the different types of data to be stored but none of them was actually hard to figure out.

- 2 Can the algorithm be easily parallelized? If yes, how? If not, why? Explain.

The algorithm could be parallelized even if an eventual parallelisation would require either the graph stream to be including the timestamp in order to be able to know the t for each edge that is processed in parallel. Nevertheless the parallel version would require the different processes to share many variables adding a relevant overhead for access synchronisation to the shared data such as the list of sampled edges. Due to this fact is likely that the parallel version of the algorithm doesn't show a sensible gain in performances.

- 3 Does the algorithm work for unbounded graph streams? Explain.

The algorithm works also for unbounded graph streams. Of course the more is the size of the graph compared to the memory bound and the more the triangle count would be a worse approximation of the real number of triangles present in the graph.

- 4 Does the algorithm support edge deletions? If not, what modification would it need? Explain.

The algorithm supports edge deletions only in the fully dynamic version. Thus, no modification is required and in case we need to process a stream containing both insertions and deletions we only need to apply the fully dynamic Triest algorithm.