김종윤
2023-28318
협동과정 인공지능 전공

소셜컴퓨팅

과제 5 - 6~7주차

1. Network Book Exercise 13.6 - Q1~3

1.1. Q1

Strongly Connected Component (SCC) is consisted of node {1, 3, 4, 8, 9, 13, 14, 15, 18}. The node {6, 11, 7, 12} are building "IN" section and node {5, 10, 16} are constructing "OUT" section. Therefore other nodes connected, {2, 17}, are tendrils.

1.2. Q2

1.2.1. a)

Adding the edge connecting node 9 from 10 will make SCC larger by including {5, 10, 16}. Another way to make SCC larger is adding edge from 13 to node 6 which will make node {6, 7} to be included by SCC.

1.2.2. b)

Removing the edge from node 15 to 18 will make node 18 to be included in set IN. Another way to enlarge the IN set is removing edge from node 4 to 1 which will make node 1 to be included into set IN.

1.2.3. c)

To enlarge the set OUT, the cyclic loop in SCC should be eliminated. Removing edge from node 1 to 8 and from node 3 to 8 will make node {1, 3, 4} to be included in SCC.

1.3. Q3

1.3.1. a)

The graph that the largest SCC can be shrunk by single edge elimination is constructed by all node connected from it-selves to the node on their nearest right. This graph is a huge SCC when all the nodes (more than 1000 nodes) are connected, but the graph break downs when any one single edge from the SCC is removed.

1.3.2. b)

By adding the edge from the very endpoint node from set OUT, to the any node in SCC will make a cyclic loop which will make the node in the loop to be included in SCC.

2. Network Book Exercise 14.7 - Q3, 4

2.1. Q3

2.1.1. a)

Step K	Mode	A	B	C	D	E	F
0		1.000	1.000	1.000	1.000	1.000	1.000
1	Hub → Auth	3.000	2.000	1.000	1.000	1.000	1.000
1	Hub ← Auth	3.000	2.000	3.000	3.000	5.000	2.000
1	Normalize	0.600	0.400	0.231	0.231	0.385	0.154
2	Hub → Auth	0.846	0.538	0.231	0.231	0.385	0.154
2	Hub ← Auth	0.846	0.538	0.846	0.846	1.385	0.538
2	Normalize	0.611	0.389	0.234	0.234	0.383	0.149

2.1.2. b)

Option 1: let's assume node X is on Authority side where Y is on Hub side

Step K	Mode	A	B	X	C	D	E	F	Y
0		1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
1	Hub → Auth	3.000	2.000	1.000	1.000	1.000	1.000	1.000	1.000
1	Hub ← Auth	3.000	2.000	1.000	3.000	3.000	5.000	2.000	1.000
1	Normalize	0.500	0.333	0.167	0.214	0.214	0.357	0.143	0.071
2	Hub → Auth	0.786	0.500	0.071	0.214	0.214	0.357	0.143	0.071
2	Hub ← Auth	0.786	0.500	0.071	0.786	0.786	1.286	0.500	0.071
2	Normalize	0.579	0.368	0.053	0.229	0.229	0.375	0.146	0.021

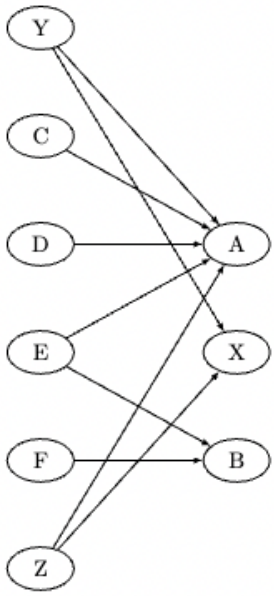
Option 2: let's assume node X is on Authority side where Y is on Hub side with given condition.

Step K	Mode	A	B	X	C	D	E	F	Y
0		1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
1	Hub → Auth	4.000	3.000	1.000	1.000	1.000	1.000	1.000	1.000
1	Hub ← Auth	4.000	3.000	1.000	4.000	4.000	7.000	3.000	8.000
1	Normalize	0.500	0.375	0.125	0.154	0.154	0.269	0.115	0.308
2	Hub → Auth	0.885	0.692	0.308	0.154	0.154	0.269	0.115	0.308
2	Hub ← Auth	0.885	0.692	0.308	0.885	0.885	1.577	0.692	1.885
2	Normalize	0.469	0.367	0.163	0.149	0.149	0.266	0.117	0.318

For the authority score, obviously option 2 will get higher score. As the option 1 only connects from Y to X with single link and there is no any other out link from X, the score will never converge and will vanish after few iteration. Unlike option 1, option 2 will converge into some value through the iteration of HITS algorithm as option 2 will make node X also being affected by other nodes via node Y.

2.1.3. c)

To make node X to be ranked on second, node Y and Z must make link to node A along with X. By connecting the nodes that connects to X to the most connected node (A), the authority score from node A will flow into node X via node Y and Z.



Step K	Mode	A	B	X	C	D	E	F	Y	Z
0		1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
1	Hub → Auth	5.000	2.000	2.000	1.000	1.000	1.000	1.000	1.000	1.000
1	Hub ← Auth	5.000	2.000	2.000	5.000	5.000	7.000	2.000	7.000	7.000
1	Normalize	0.556	0.222	0.222	0.152	0.152	0.212	0.061	0.212	0.212
2	Hub → Auth	0.939	0.273	0.424	0.152	0.152	0.212	0.061	0.212	0.212
2	Hub ← Auth	0.939	0.273	0.424	0.939	0.939	1.212	0.273	1.364	1.364
2	Normalize	0.574	0.167	0.259	0.154	0.154	0.199	0.045	0.224	0.224

2.2. Q4

2.2.1. a)

All the nodes maintain their score even after one step of Basic PageRank update, which means the web pages are forming equilibrium.

2.2.2. b)

This webpage is not in equilibrium state as node A will achieve 1/2 after one step update.

3. PageRank

3.1. Loop: PageRank Algorithm implementation

Pagerank (1997) implementation with loop is quite trivial as it just need to follow the algorithm description from the paper. The implementation with python is described on figure 1.

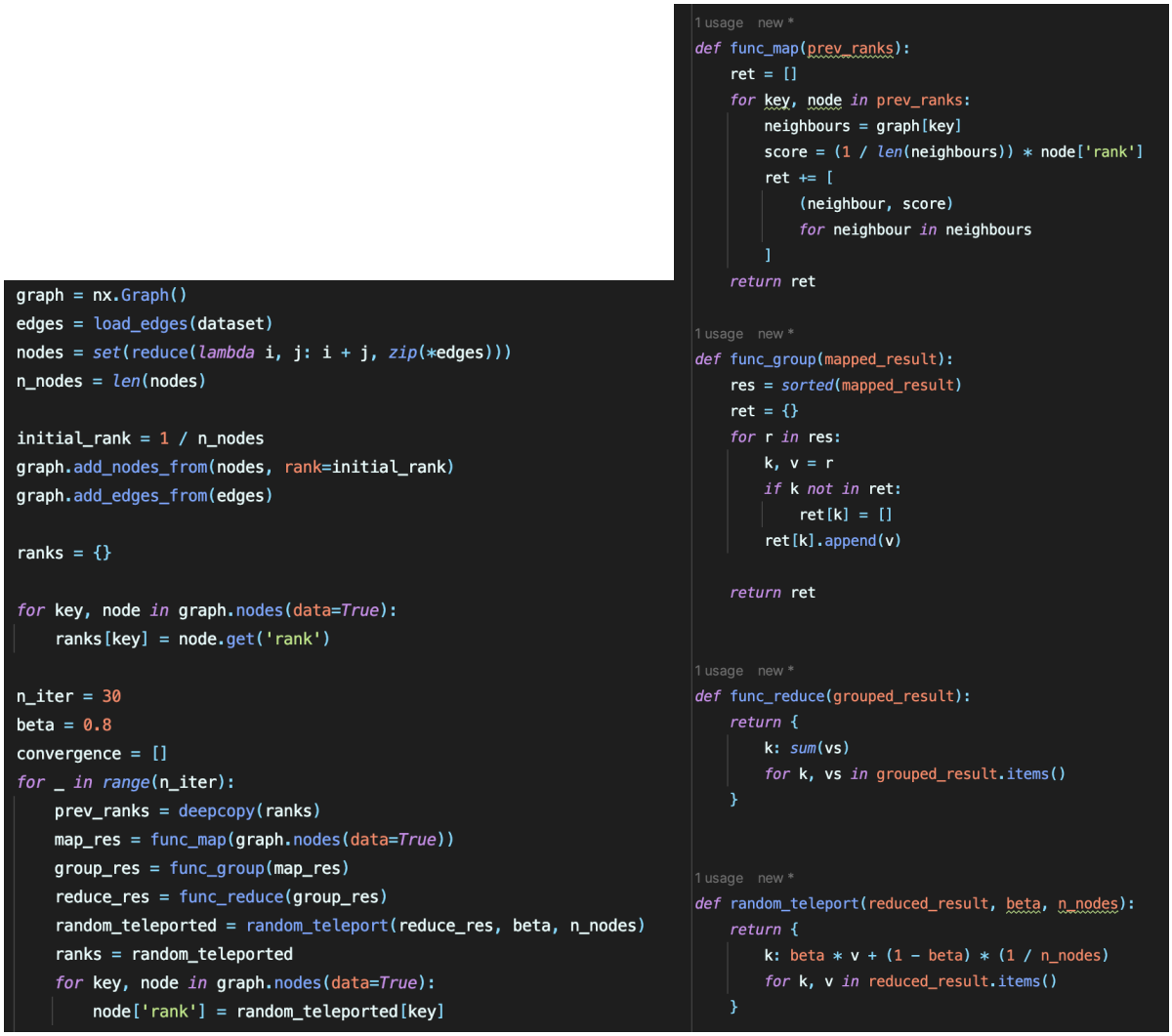
```
graph = nx.Graph()
edges = load_edges(dataset)
nodes = set(reduce(lambda i, j: i + j, zip(*edges)))
n_nodes = len(nodes)

initial_rank = 1 / n_nodes
graph.add_nodes_from(nodes, rank=initial_rank)
graph.add_edges_from(edges)

ranks = {}

for key, node in graph.nodes(data=True):
    ranks[key] = node.get('rank')

n_iter = 30
beta = 0.8
convergence = []
for _ in range(n_iter):
    prev_ranks = deepcopy(ranks)
    for key, node in graph.nodes(data=True):
        rank_sum = 0.0
        neighbours = graph[key]
        for n in neighbours:
            if ranks[n] is not None:
                outlinks = len(list(graph.neighbors(n)))
                rank_sum += (1 / outlinks) * ranks[n]
        ranks[key] = beta * rank_sum + (1 - beta) * (1 / n_nodes)
```

Figure 1: Pagerank algorithm loop way implementation

3.2. Map-Reduce: PageRank Algorithm implementation

The implementation of pagerank with map reduce is slightly different compare to loop way. Map function maps <node, score> to [<neighbour1_of_node, out_score>, <neighbour2_of_node, out_score>, …]. Group function groups the out scores by the key which results <node, [out_score1, out_score2, …]. Finally reduce function sums up the scores on value side, <node, final_updated_score>. However, the pagerank(1997) implementation include random suffer model, therefore, random score should be injected by the proportion by the beta, the hyper-parameter (often set as 0.8 with step size k=5).

Although this implementation followed the instruction from the lecture, the map-reduce is not implemented as it designed. The map-reduce is originally designed to perform on distributed computing circumstance. Therefore, the implementation must see part of the graph, while the given implementation on figure 2 does not. To achieve original implementation of the map-reduce on distributed computing resources, there must be a method `broadcast` which tells all the distributed node to receive global variable. For this case, each node's number of outlinks should be broadcasted as immutable global variable. This methods all well implemented on Spark (either using Scala or Python) which can be used on top of Hadoop ecosystem.



```python
graph = nx.Graph()
edges = load_edges(dataset)
nodes = set(reduce(lambda i, j: i + j, zip(*edges)))
n_nodes = len(nodes)

initial_rank = 1 / n_nodes
graph.add_nodes_from(nodes, rank=initial_rank)
graph.add_edges_from(edges)

ranks = {}

for key, node in graph.nodes(data=True):
    ranks[key] = node.get('rank')

n_iter = 30
beta = 0.8
convergence = []
for _ in range(n_iter):
    prev_ranks = deepcopy(ranks)
    map_res = func_map(graph.nodes(data=True))
    group_res = func_group(map_res)
    reduce_res = func_reduce(group_res)
    random_teleported = random_teleport(reduce_res, beta, n_nodes)
    ranks = random_teleported
    for key, node in graph.nodes(data=True):
        node['rank'] = random_teleported[key]
```

```python
1 usage  new *
def func_map(prev_ranks):
    ret = []
    for key, node in prev_ranks:
        neighbours = graph[key]
        score = (1 / len(neighbours)) * node['rank']
        ret += [
            (neighbour, score)
            for neighbour in neighbours
        ]
    return ret

1 usage  new *
def func_group(mapped_result):
    res = sorted(mapped_result)
    ret = {}
    for r in res:
        k, v = r
        if k not in ret:
            ret[k] = []
        ret[k].append(v)

    return ret


1 usage  new *
def func_reduce(grouped_result):
    return {
        k: sum(vs)
        for k, vs in grouped_result.items()
    }


1 usage  new *
def random_teleport(reduced_result, beta, n_nodes):
    return {
        k: beta * v + (1 - beta) * (1 / n_nodes)
        for k, v in reduced_result.items()
    }
```

Figure 2: Pagerank algorithm implementation with Map-Reduce
(left: overall implementation, right: functions used for map-reduce implementations)

3.3. Pagerank results

The pagerank obtained by both loop approach and map-reduce approach can be found on table1 and 2 for dataset dolphins.csv and lesmis.csv, respectively.

3.4. Analysis between loop method and map-reduce method

Primarily, measuring the time consumption only for pagerank calculation for 100 times results average10.54 ms for loop approach and average 8.25 ms for map-reduce approach. This time measurement means that using map-reduce for small volume of data on single machine will not lead to dramatic performance improvement. With multi-processing that uses multiple cores may achieve performance improvement for map-reduce approach. Moreover, the small time difference may be caused due the implementation of the map-reduce following the lecture note. The implementation does not really splits the input data into size of workers and also multiple processes are not involved to the program, therefore, achieving time efficiency may be a difficult goal for this implementation.

The trend of the convergence of each method can be analyzed. Both loop method and map-reduce method converges their score quite quickly and similarly. The convergence of the score can be found on figure 3 and 4. One difference can be found on two figures is loop method converges to some value dramatically on first 2 stages while map-reduce method converges smoothly through first 5 steps.

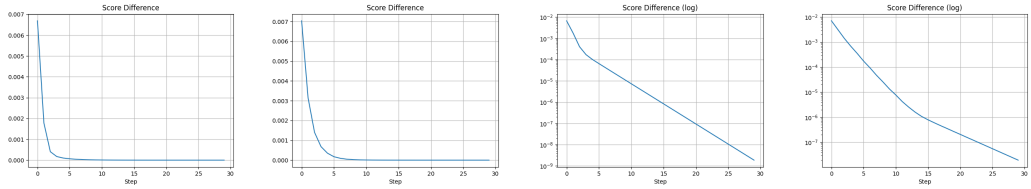


Figure 3: Score convergence in Dolphins dataset
(left to right, 1: loop, 2: map-reduce, 3: loop logscale, 4: map-reduce logscale)

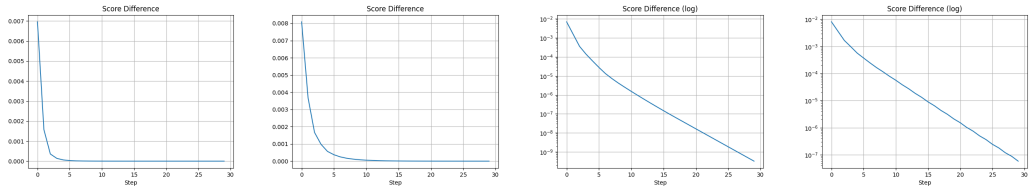


Figure 4: Score convergence in Lesmis dataset
(left to right, 1: loop, 2: map-reduce, 3: loop logscale, 4: map-reduce logscale)

As two methods converges to some value similarly but show slight difference, the final value after 30 steps also showed minor difference. The difference between two method on average over nodes is 6.56e-8 and 6.26e-8 in dolphins and lesmis dataset, respectively. This very minor difference may caused the difference in summation. The loop methods keep sums-up the scores on `rank_sum` variable while map-reduce method gathers out-score on group_by stage and sums up the scores on reduce stage. The machine has limit on expressing the decimal values on its floating point precision which is usually set to IEEE 754 doubles these days. On M2(Apple) machine, python uses 18 decimals to express float value. However, internally python will use IEEE 754 doubles to calculate the value. Therefore, summing the value on explicit variable such as `rank_sum` will truncate (or rounded up) the decimals by 18 after summation while group and reduce steps on map-reduce will use `sum(*)` method, implemented internally in python, calculates without ore less truncation.

Table 1: Dolphins dataset result

Node	Loop	MR
Beak	0.016684055689292900	0.016684002901478900
Beescratch	0.024101833091359300	0.024101942694052300
Bumper	0.013664189448587000	0.013664149096473100
CCL	0.009892276817471500	0.009892247731197200
Cross	0.0057393960346273	0.005739384176673880
DN16	0.01456880048775200	0.014568903898690500
DN21	0.019797062246735700	0.01979720853734170
DN63	0.015558757306695500	0.015558800032157700
Double	0.016864137382726400	0.01686408625714760
Feather	0.023060503866115200	0.02306068184401050
Fish	0.01512041084054970	0.015120364963913400
Five	0.0057393960346273	0.005739384176673880
Fork	0.005450484118870760	0.005450472742299690
Gallatin	0.025551965078548600	0.025552161292726100
Grin	0.03089634054093350	0.03089621180410150
Haecksel	0.01957065018278880	0.01957057890388910
Hook	0.01624387611816290	0.01624381147107800
Jet	0.03169228583097030	0.03169248812183140
Jonah	0.01891859203092530	0.018918513734228600
Knit	0.012981672661326000	0.012981716021888500
Kringel	0.02388816783708610	0.023888090594936400
MN105	0.0166118528677463	0.016611784993195100
MN23	0.006042898525476930	0.006042922153348900
MN60	0.010128966014001500	0.01012893957296570
MN83	0.016590147446676100	0.016590078234278400
Mus	0.011873958716707100	0.011874017881235400
Notch	0.011552714205722700	0.011552767225337500
Number1	0.01718959658657450	0.017189682051893200
Oscar	0.014736402408739200	0.014736393334813300
PL	0.01528194755070020	0.01528195259694020

Node	Loop	MR
Patchback	0.026152241234961600	0.026152138155064300
Quasi	0.006042898525476930	0.006042922153348900
Ripplefluke	0.013979408647106000	0.013979490249194800
SMN5	0.005550450116942830	0.005550438342925270
SN100	0.020317408913147600	0.020317389299005300
SN4	0.028784678384779100	0.028784566783958700
SN63	0.023785392009704200	0.023785304419248500
SN89	0.008188436947458070	0.008188458259819110
SN9	0.021299999807227500	0.021299940964726200
SN90	0.015878482650013800	0.015878593018325100
SN96	0.017469190131924300	0.017469145481591900
Scabs	0.027808470840723300	0.027808359702725600
Shmuddel	0.016054257802227100	0.016054201542644600
Stripes	0.021637274762531400	0.021637193706643000
TR120	0.009543373279640020	0.009543432219601800
TR77	0.017125167460719300	0.017125131634333700
TR82	0.005866447357384060	0.005866471423386310
TR88	0.009611837245947400	0.009611812610148880
TR99	0.018709878256825500	0.018709804857582100
TSN103	0.012118745334698200	0.012118703825253300
TSN83	0.008782677004758350	0.008782653781044440
Thumper	0.013029257968292400	0.013029215604247600
Topless	0.028429218698928300	0.028429101187222600
Trigger	0.031419869787680000	0.031419754122495900
Upbang	0.021097729213035500	0.021097861818901400
Vau	0.007976041995591410	0.007976019438221720
Wave	0.008779174848728060	0.008779227608585440
Web	0.029707210189925500	0.029707416737578600
Whitetip	0.005604345899466650	0.005604334398309760
Zap	0.014738513680080500	0.014738470574803000
Zig	0.006953649081147020	0.006953677393091940
Zipfel	0.011565146818924800	0.011565113642814200

Table 2: Lesmis dataset Pagerank result

Node	Loop	MR
Anzelma	0.006577169159816710	0.006577158212128500
Babet	0.016037837949459000	0.016037809689212000
Bahorel	0.016328914043213200	0.016328771506808800
Bamatabois	0.015289205051171000	0.01528926638457950
BaronessT	0.005579608647987410	0.005579598967391250
Blacheville	0.012507452585847400	0.012507425313416300
Bossuet	0.017965707156695600	0.01796556900709050
Boulatruelle	0.003951940571746880	0.003951937253934920
Brevet	0.012385937019101800	0.012385997230152500
Brujon	0.011585681382518600	0.011585651401665400
Champmathieu	0.012385936910476200	0.012385997230152500
Champtercier	0.006143389145791940	0.006143553285745370
Chenildieu	0.012385936847149000	0.012385997230152500
Child1	0.0064113802202313500	0.006411359124022040
Child2	0.006411380273106980	0.006411359124022040
Claquesous	0.015899263671344400	0.01589923492891650
Cochepaille	0.012385937060319200	0.012385997230152500
Combeferre	0.015126844469940000	0.015126714540707700
Cosette	0.020195692609750400	0.02019567939514130
Count	0.006143389145791940	0.006143553285745370
CountessDeLo	0.0061433891164743	0.006143553285745370
Courfeyrac	0.017580662079418900	0.017580512806208900
Cravatte	0.006143389145791940	0.006143553285745370
Dahlia	0.012507452617496400	0.012507425313416300
Enjolras	0.020620601821761000	0.02062045873470350
Eponine	0.017119304415124400	0.017119246310185100
Fameuil	0.012507452555867800	0.012507425313416300
Fantine	0.02631738095268600	0.026317367282621100
Fauchelevent	0.012377257269396100	0.012377277286339300
Favourite	0.012507452723059500	0.012507425313416300
Feuilly	0.015126844259559100	0.015126714540707700
Gavroche	0.03435920162361400	0.03435900805063650

Node	Loop	MR
Geborand	0.006143389145791940	0.006143553285745370
Gervais	0.004251183279331230	0.0042511895296269600
Gillenormand	0.015189464255420200	0.01518944737760450
Grantaire	0.013823596737536400	0.013823474518906100
Gribier	0.005072854095872640	0.005072860714486200
Gueulemer	0.016037837884146700	0.016037809689212000
Isabeau	0.004251183279331230	0.0042511895296269600
Javert	0.029264155115379500	0.029264152751272100
Joly	0.01632891445229400	0.016328771506808800
Jondrette	0.00608254981372984	0.006082536888220000
Judge	0.012385936980357500	0.012385997230152500
Labarre	0.004251183279331230	0.0042511895296269600
Listolier	0.012507452500258400	0.012507425313416300
LtGillenormand	0.00896197254441402	0.008961959772830460
Mabeuf	0.017074154251326500	0.017074031463534200
Magnon	0.005713529672287880	0.005713526028892160
Marguerite	0.005654777002498390	0.005654782191077980
Marius	0.0295988459251392	0.029598709609097200
MlleBaptistine	0.010632504286897800	0.01063270582217290
MlleGillenormand	0.016743880106647500	0.016743863993281400
MlleVaubois	0.004510988895305170	0.004510987146346140
MmeBurgon	0.008712867961972430	0.008712844455333220
MmeDeR	0.004251183279331230	0.0042511895296269600
MmeHucheloup	0.010417138791333300	0.010417055559051200
MmeMagloire	0.010632504331855900	0.01063270582217290
MmePontmercy	0.006601318257265050	0.006601310903354820
MmeThenardier	0.0189775886708358	0.018977567254706900
Montparnasse	0.014639397174189900	0.014639370457005300
MotherInnocent	0.006726634733210450	0.006726647646710560
MotherPlutarch	0.003839159305967260	0.003839146981142670
Myriel	0.04432483148839630	0.04432489016880100
Napoleon	0.006143389145791940	0.006143553285745370
OldMan	0.006143389145791940	0.006143553285745370

Node	Loop	MR
Perpetue	0.005877585933632130	0.005877587520734070
Pontmercy	0.00783873510734957	0.007838719617258000
Prouvaire	0.012617407140194400	0.012617297860704800
Scaufflaire	0.004251183279331230	0.0042511895296269600
Simplice	0.009382948081367550	0.009382952705673000
Thenardier	0.027090759486885700	0.027090707714164700
Tholomyes	0.01529258019080040	0.01529254390729800
Toussaint	0.0070970976349886900	0.0070971018846427100
Valjean	0.07442013068678860	0.07442044932807940
Woman1	0.005628320065439120	0.00562832562814704
Woman2	0.007097097669438810	0.0070971018846427100
Zephine	0.012507452685841000	0.012507425313416300