Project : Single-Document Keyword Extraction as a Dense Subgraph Discovery Problem

Alexandre Carton & Hugues Derogis

Abstract

In this project, we transform the single-document keyword extraction problem into the research of dense subgraphs, using the graph-of-word representation of documents. After some preprocessing steps, a paper is thus turned into a graph were each node represents a word, and edges the links between them. We then apply different algorithms in order to extract the densest subgraph, and we retain as keywords those in the dense subgraph. Our results are compared to the existing approaches.

1 Introduction

Today, many methods are used to determine the Golden Keywords of different texts. The main methods used often repose on calculating the keywords that appear the most in the texts. Although it is an easy and quite reliable method, in some cases this method is not as good as it should. On the other hand, many methods exist to discover the Densest Subgraph of a graph. The idea is to use different algorithms that discover the densest subgraph of a graph to compute the Golden Keywords of a text.

2 Graph-of-word

The first step to test all the following algorothms is to construct the *graph-of-words*. The text is preprocessed to retain the most relevant information, then it is transformed into a graph $\mathcal{G}$. In this graph, nodes are words and they are connected if a link of proximity between them exists in the paper. The edges are weighted, the weight represents the number of links that occur in the paper.

2.1 Preprocessing

We consider a text that we want to represent in a graph-of-words. Not all words are meaningful, which is why it requires preprocessing. We applied the following steps :

Tokenization Transforms the text in a list of words. We used python's nltk package with the function *nltk.word_tokenize*, which also separates punctuation from the rest of the words.

Part-of-speech tagging This step adds a suffix for each word in the list, corresponding to its grammatical function in the sentence. We used the nltk algorithm, *nltk.pos_tag*. We then keep only the nouns and adjectives, like Rousseau and Vazirgiannis in [1]

Stopwords removal Some common words that are not relevant or meaningful are removed. The list is in the *stopwords.txt* file and comes from [2]. Other elements are also removed, for example those containing numbers from the references that nltk couln't tag.

Stemming The Porter and Stemming [3] algorithm cuts word endings, so that 'document', 'documents', 'documentation' are gathered into a unique node 'document'

2.2 Graph

To construct the graph from the preprocessed list of words, we use a sliding window of size 4 (value used in [1]) : every pair of words both appearing in the window generates an edge between the two corresponding nodes. The edges are weighted so that if this situation happens several times we keep the number of co-occurences in memory. The graph is computed in Python using the NetworkX library.

3 Dense subgraph extraction

Now, we have a graph-of-words built from the words in the text. We now have to analyze the graph-of-words in order to determine the most important keywords of the text. For this purpose, we use algorithms that are used to find the densest subgraph in a graph. Finding the densest subgraph is a tough question and it is mainly used in order to detect communities in large graphs.

However, we will present different algorithms used to find the densest subgraph and compare the results – both the keywords computed by the algorithm and the complexity of the algorithm – of these algorithms. We have chosen different algorithms: the k-core algorithm; another algorithm which uses the average degree of the subgraph as a definition of the density and an algorithm which uses the theory of flows and determines the minimum-cut of the graph to return the densest subgraph. Lastly, we chose an algorithm which finds the maximum clique of the graph.

Then, we compare it with the PageRank algorithm, asking for a number of keywords. The first advantage of using the densest subgraph theory to determine the best keywords of the text is that we do not have to choose the number of

keywords to compute; the algorithm chooses itself the keywords and adapts itself to the text.

Considering a 1000-word text, how many keywords should we get? Three, ten, fifty, one hundred? Actually, it mainly depends on the text, and on its concentration of keywords.

With densest-subgraphs algorithms, we may have 5 keywords for one text and 50 for another text.

3.1 K-cores

3.1.1 Definition

First, we must define what is a K-Core, what is the *Core Number* of a node and how we can determine the densest subgraph using the K-Cores :

- A K-Core is a subgraph of an undirected graph $\mathcal{G}$ in which considering only the nodes and edges of the subgraph, all the nodes of the subgraph have at least K neighbors.
- The *Core Number* of a node is the maximum K for which the node belongs to the K-Core subgraph.
- We define the densest subgraph as the maximum K-Core found.

3.1.2 Algorithm

NetworkX algorithm The library NetworkX has already an algorithm to find the maximum K-Core of the graph :

```
G.remove_edges_from(G.selfloop_edges());
S4 = nx.k_core(G);
```

Our algorithm However, the algorithm from NetworkX has one problem: our graph-of-words has weighted edges whereas the algorithm does not take into account these weights. Thus, we had to write another K-Core algorithm that will take into account these weights. Here is the algorithm :

The algorithm reposes on the computation of the Core Number of every node of the graph. Basically, we have a list with all the nodes and their number of links. We sort this list so that the first element of the list is the node with the lowest number of links computed with the weights. Then, the degree of the first node corresponds to its Core Number. We remove this node from the list and insert it in a "Core Number List", having all the nodes and their Core Number. Parallel to this, we update the 1st list by computing the new number of links of the nodes in the list. Actually, only the neighbors of the node that was removed have to be updated.

Once we have entirely filled the "Core Number List", we just have to compute the highest Core Number and take all the nodes that have this Core Number. We then get the Densest Subgraph considering K-Core.

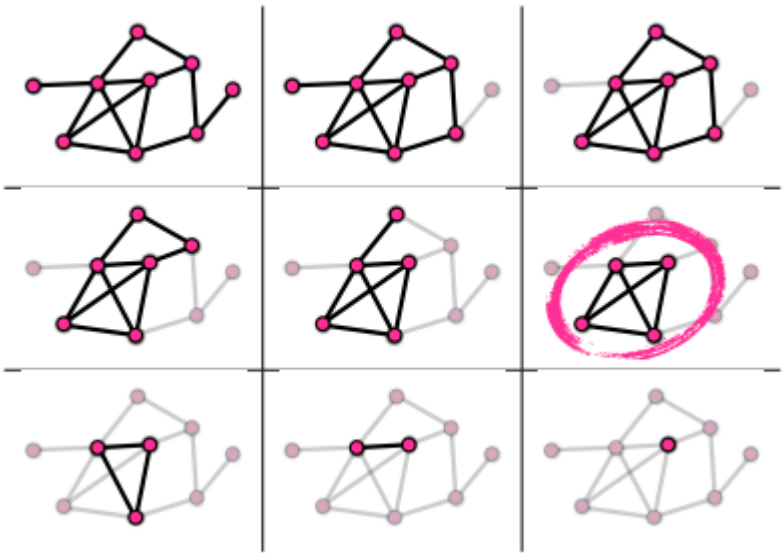
3.2 Best average degree

3.2.1 Definitions

In this algorithm, we use a new definition of the density. Here, the density of the graph $\mathcal{G}$ is the Average Degree, defined as : AvgDeg $= 2\frac{E}{N}$ where E is the number of edges and N the number f edges of $\mathcal{G}$.

3.2.2 Greedy Algorithm

This first algorithm is a greedy algorithm. Here is the algorithm given by :



Basically, we first consider the whole graph. We compute the node with the lowest degree (considering the weight of the edges) of the graph. Then, we remove this node from the graph, we compute the density of the subgraph that we have and we come back to first step. We do this until there is no more node in the graph, and we get a list with all the subgraphs and their density. We just have to choose the subgraph with the highest density to get the densest subgraph.

3.2.3 Mincut Algorithm

The Greedy Algorithm gives good results but it takes a quite a long time. Actually, as said by [4], the research of the densest subgraph can be replaced by the Linear Programming Problem and we can see this problem as a min-cut

problem :

$$
\begin{array}{lll}
(1) & \max \sum_{i,j} x_{ij} & \\
(2) & \forall i,j \in E & x_{ij} \leq y_i \\
(3) & \forall i,j \in E & x_{ij} \leq y_j \\
(4) & \sum_i y_i \leq 1 & \\
(5) & x_{ij}, y_i \geq 0 &
\end{array}
$$

The algorithm we used is the Goldberg's algorithm and there are three steps to create the directed graph from the undirected graph $\mathcal{G}$. Given m the number of nodes of G, a number g and d_i the degree of the node i in the original graph, we :

- Add a source s and a sink t to the undirected graph $\mathcal{G}$
- Replace each undirected edge of the graph by two directed edges with capacity 1
- Add directed edges from s to all the nodes of $\mathcal{G}$ with capacity m AND we add directed edges from all the nodes to t with capacity $E + 2g–d_i$

Then the algorithm is ([4]) :

Algorithm 10 FindDensestSubgraph(G)

$mind \leftarrow 0; maxd \leftarrow m;$
$V_s \leftarrow \emptyset;$
while $maxd - mind \geq \frac{1}{n(n-1)}$ **do**
  $g \leftarrow \frac{maxd+mind}{2}$,
  Construct new network as we have mentioned;
  Generate S and T utilizing max-flow min-cut algorithm;
  if $S = \{s\}$ **then**
    $maxd \leftarrow g$;
  else
    $mind \leftarrow g$;
    $V_s \leftarrow S - \{s\}$;
  end if
end while
return subgraph induced by V_s;

3.3 Maximum clique

3.3.1 Definitions

A k-clique is a graph with k nodes where every pair of nodes is connected, so that the average degree defined above is maximal and equal to $k - 1$. The maximum cliques we are looking for are subgraphs of $\mathcal{G}$ having the maximum

size. There may be several solutions. Finding the maximal cliques in a graph is a **NP**-hard problem, however since the graphs we use are not too big - typically 100-1000 nodes - this approach remains possible.

3.3.2 Implementation

We use NetworkX function which extracts all cliques as below :

```
def find_densest_subgraph(G) :
    Slist=list(nx.find_cliques(G))
    avg_best = max(len(clique) for clique in Slist)
    cliques=[c for c in Slist if len(clique)==avg_best]
    return cliques, avg_best
```

3.4 Pagerank

Contrary to the other methods, this is not a dense subgraph extraction method. We tried this algorithm to compare its results with the other, as it is a well-known way and already explored way to retrieve keywords.

4 Conclusion and results

Therefore, we have many different algorithms which search Densest Subgraphs. Now, we have to compare these algorithms to find which one will be the most suitable to detect Keywords in papers. Comparison reposes on two points :

- Keywords computed by the algorithm. We compare the keywords computed by the algorithms to the keywords "computed" by humans. Then, we want the algorithm which has the highest number of True Positives AND the lowest numbers of False Positives and False Negatives.
- Complexity of the algorithm. Even if an algorithm gives much better keywords, it is not worth using it if the Time Computation is way higher than other algorithms. Thus, we aslo compute the time of each algorithm.

4.1 First test - Paper.txt

First, we tested our algorithms on one paper to have an idea of which one could be the best one.

The file paper.txt has about 4300 words. It is Rousseau and Vazirgiannis' paper [1]on which this work is mainly based, captions, references and the mast paragraph - which gives its own keywords as a final example - excluded. The first step was to compute the graph-of-words from the paper. Using our algorithms, we obtain a graph-of-words with 450 nodes and 3953 edges, contructed in about 2 seconds.

From the graph-of-words, we can now use the different algorithms of Denses Subgraph Discovery to get different Keywords for this paper. We also compute the time taken by the algorithms.

There is not a precise number of keywords for all the papers but the number of keywords depends on the text. Thus, algorithms based on Densest Subgraph Discovery are expected to have better results than PageRank or classic algorithms.

4.1.1 Golden Keywords

The Golden Keywords of the file paper.txt are :

```
golden_keywords = ['keyword', 'degeneracy', 'document',
'single', 'extraction', 'graph', 'representation','text',
'weighted', 'graph-of-words', 'k-core', 'decomposition']
```

4.1.2 Unweighted K-Core : NetworkX Algorithm

From the paper, we get these results :

```
Nodes in densest subgraph using unweighted k-cores :
['origin', 'represent', 'set', 'bigram', 'edg', 'weight',
'golden', 'process', 'text', 'Section', 'direct',
'dataset', 'paper', 'result', 'human', 'cohes', 'co-occur',
'total', 'extract', 'size', 'variabl', 'precis', 'number',
'network', 'author', 'graph', 'top', 'graph-of-word',
'n-gram', 'length', 'textual', 'co-occurr', 'rel',
'import', 'main', 'approach', 'method', 'higher', 'node',
'core', 'curv', 'recal', 'lot', 'subgraph', 'vertex',
'account', 'decomposit', 'k-core', 'increas', 'unigram',
'vertic', 'case', 'term', 'document', 'central', 'keyword',
'algorithm', 'work', 'annot', 'structur', 'degeneraci',
'unweight', 'degre', 'statist', 'neighbor', 'time',
'model', 'order', 'undirect', 'experi']
Number of nodes :  70 / 450
Computation time unweighted k-cores :  0.0149603713253
```

4.1.3 Weighted K-Core : Our Algorithm

From the paper, we get these results :

```
Nodes in densest subgraph using weighted k-cores :
['graph', 'top', 'document', 'number', 'keyword']
Number of nodes :  5 / 450
Avg deg :  52
Computation time weighted k-cores :  0.146862833292
```

4.1.4 Average Degree : Greedy Algorithm

From the paper, we get these results :

```
Nodes in densest subgraph using avg degree :
['represent', 'term', 'main', 'edg', 'keyword', 'core',
'k-core', 'text', 'top', 'work', 'number', 'graph-of-word',
'length', 'set', 'graph', 'approach', 'vertic', 'extract',
'document']
Number of nodes :  19 / 450
Avg deg :  147.368421053
Computation time avg degree :  10.2549911783
```

4.1.5 Average Degree : MinCut Algorithm

From the paper, we get these results :

```
Nodes in densest subgraph using mincut :
['represent', 'term', 'set', 'extract', 'edg', 'keyword',
'core', 'text', 'top', 'work', 'number', 'graph-of-word',
'length', 'graph', 'vertic', 'k-core', 'main', 'approach',
'document']
Number of nodes :  19 / 450
Avg deg :  147.368421053
Computation time mincut :  4.63712610227
```

4.1.6 Maximum Clique

From the paper, we get these results :

```
 Nodes in densest subgraph using cliques :

 printing clique no :  1
['keyword', 'co-occurr', 'term', 'graph', 'text',
'graph-of-word', 'core', 'document', 'number', 'edg',
'k-core']

 printing clique no :  2
['keyword', 'main', 'core', 'graph', 'k-core', 'term',
'document', 'approach', 'graph-of-word', 'text', 'work']

 printing clique no :  3
['keyword', 'extract', 'document', 'work', 'k-core',
'graph', 'term', 'core', 'text', 'approach',
'graph-of-word']

 printing clique no :  4
```

```
['keyword', 'extract', 'document', 'number', 'k-core',
'graph', 'core', 'edg', 'graph-of-word', 'term', 'text']

 printing clique no :  5
['keyword', 'extract', 'document', 'number', 'k-core',
'graph', 'core', 'edg', 'graph-of-word', 'term', 'vertic']

 printing clique no :  6
['keyword', 'extract', 'document', 'number', 'k-core',
'graph', 'core', 'edg', 'graph-of-word', 'set',
'vertic']

Number of nodes :  11 / 450
Avg deg :  123.6875
Computation time cliques :  0.0371106631076
```

4.1.7 PageRank

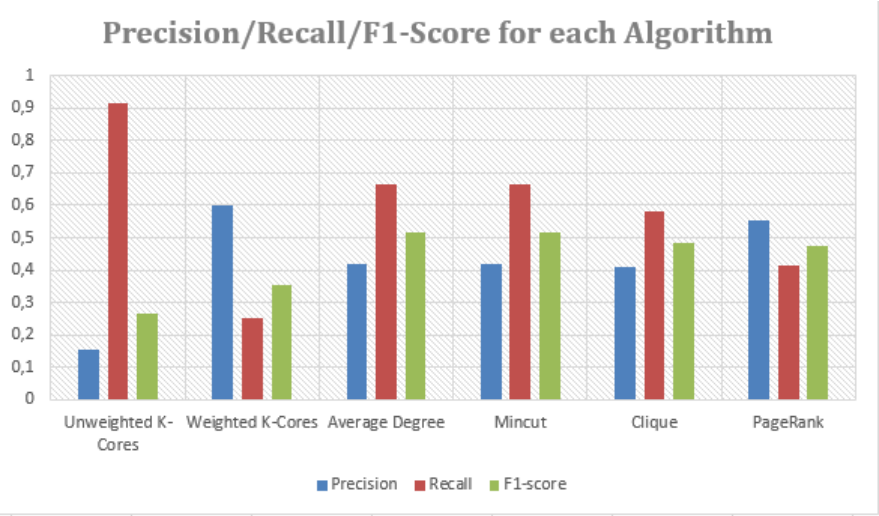
From the paper, we get these results :

```
Number of nodes chosen for pagerank : 10 / 450
Nodes :  [('graph-of-word', 0.0126273413256538), ('vertic',
0.01277112356916248), ('edg', 0.013157160492818436),
('k-core', 0.017021367280321425), ('term',
0.018101530631287683), ('graph', 0.01825211628306198),
('extract', 0.018376073905439164), ('number',
0.021564368478025904), ('document', 0.02339065876223866)]
Computation time pagerank :  0.210977067041
```

4.1.8 Comparison of the Algorithms

To compare the different algorithms we had, we computed the Recall & Precision for each Algorithm.

Algorithm	Number of Nodes	Time (s)	Precision	Recall	F1-score
Unweighted K-Cores	70	0,016	0,157	0,91667	0,26808459
Weighted K-Cores	5	0,136	0,6	0,25	0,35294118
Average Degree	19	12,95	0,421	0,6667	0,51609948
Mincut	19	5,14	0,421	0,6667	0,51609948
Clique	11	0,044	0,4118	0,5833	0,48277146
PageRank	10	0,263	0,5556	0,41667	0,47620898



Thus, we can see that :

- The Unweighted K-Cores gives the best Recall score and the worst Precision score. It is normal as the number of keywords computed by this algorithm is 70/450 !
- The Weighted K-Cores gives the best Precision score but has a low Recall Score.
- Average Degree and Mincut give the same results because it is just two ways to achieve the same "algorithm". We can see that they have both a medium/high Recall & Precision scores. However, the time taken by the algorithm is quite high (around 15 secondes for Average Degree and 5 secondes for Mincut)
- Lastly, the Clique Algorithm has a medium Precision score and a medium Recall score and the time taken taken by the algorithm is quite low : 0.037 secondes.

4.2 Test on the Databases Hulth2003 and Krapivin2009

To do the tests, we used the Databases named Hulth2003 and Krapivin3009 that are divided in two types of files :

1. The excerpt : abstracts from research papers
2. The corresponding keywords computed by humans

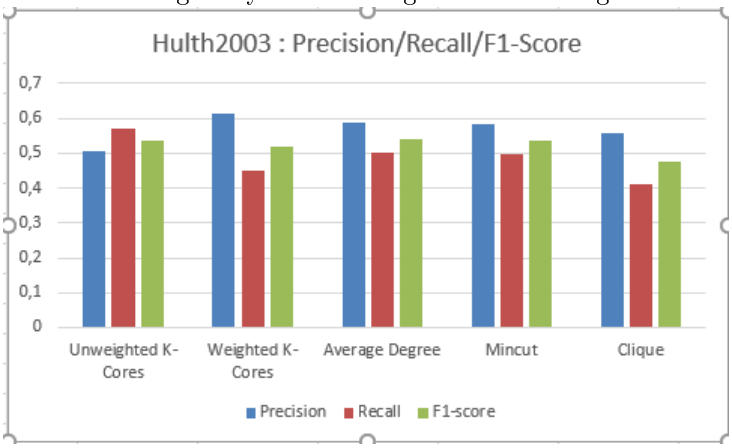
4.2.1 Hulth2003

The Database Hulth2003 is composed of around 500 papers of 30-40 words each. The Database is composed of the *.abstr* files which are the papers and the *.uncontr* files which are the Keywords computed by humans.

Here are our results :

Algorithm	Time (s)	Precision	Recall	F1-score
Unweighted K-Cores	0,5	0,505	0,5721	0,53645994
Weighted K-Cores	1,72	0,6156	0,4506	0,5203327
Average Degree	13	0,5865	0,5032	0,54166615
Mincut	51	0,5847	0,496	0,53670991
Clique	0,55	0,5594	0,4112	0,47398574

Here are the results in a graphic. We can see that the results are not that better with our algorithms compared to the classic NetworkX Algorithm. The explanation is simple : the Database Hutlh2003 is composed of papers of about 30-40 words so the number of keywords computed by the algorithms is very low and does not change very much among the different algorithms.

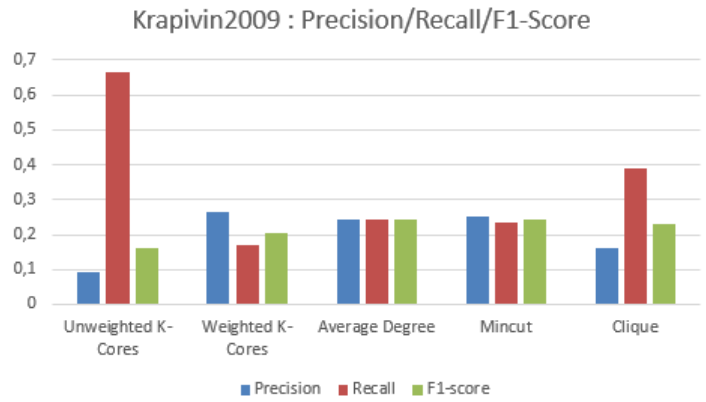


4.2.2 Krapivin2009

The Database is composed of around 2300 papers of 500-700 words each. The Database is composed of *.txt* files which are the papers and the *.key* files which are the Keywords computed by humans.

Here are our results :

Algorithm	Time (s)	Precision	Recall	F1-score
Unweighted K-Cores	0,94	0,091828224	0,66622464	0,16140886
Weighted K-Cores	9,24	0,262656	0,16851456	0,2053079
Average Degree	940	0,24496128	0,24297984	0,24396654
Mincut	299	0,2515968	0,23530291	0,24317722
Clique	4	0,16178688	0,3877632	0,22831404



Here, we can see that the results are very different with the Unweighted K-Cores Algorithm from NetworkX and with our Algorithms. This is because the papers are 500-700 word long. We can see that the Weighted K-Cores algorithm gives rhe best precision score but a low Recall score. On the contrary, the Clique Algorithm gives the best Recall score but a low Precision Score. Lastly, Avg Deg and Mincut Algorithms give quite the same results : medium Precision and Recall scores, and a good F1-Score. However, the time taken by these algorithms is much higher than for the other algorithms.

5 Conclusion

Thus, we can see that using Densest-Subgraph Discovery Methods is very powerful to discover Golden Keywords of some papers. For papers with only few words, using the NetworkX Algorithm K-Core seems to be a good idea to balance time & results. However, when the papers have many words, the Mincut Algorithm is the one which gives the best results but it is also the one which takes the greatest time to compute.

6 Appendices

6.1 K-Core Algorithm

```
def find_densest_subgraph(G):

    S = copy.deepcopy(G)
    E_init = compute_number_of_edges(S)
    N_init = S.number_of_nodes()
    avg_degree_max = (2.0*E_init)/N_init
    S_max = copy.deepcopy(S);

    while S.number_of_nodes() > 0 :
```

```
        E = compute_number_of_edges(S)
        N = S.number_of_nodes()
        avg_degree = (2.0 * E)/N
        if avg_degree_max <= avg_degree :
            avg_degree_max = avg_degree;
            S_max = copy.deepcopy(S);
                nodes = {}
        liste = []
        for node,deg in S.degree_iter() :
            deg1 = S.degree(node, weight='weight')
            liste.append([node, deg1])
        nodes_min = find_lowest_degree(liste);
        S.remove_node(nodes_min[0]);

    return S_max, avg_degree_max;
```

6.2 Average Degree : Greedy Algorithm

```
def compute_number_of_edges (S) :
        sum = 0
        for node in S.degree_iter() :
                deg = S.degree(node, weight='weight');
                sum = deg.values()[0] + sum;
        return sum

def find_densest_subgraph(G):

        S = copy.deepcopy(G)
        E_init = compute_number_of_edges(S)
        N_init = S.number_of_nodes()
        avg_degree_max = (2.0*E_init)/N_init
        S_max = copy.deepcopy(S);

    while S.number_of_nodes() > 0 :
        E = compute_number_of_edges(S)
                N = S.number_of_nodes()
                avg_degree = (2.0 * E)/N
                if avg_degree_max <= avg_degree :
                        avg_degree_max = avg_degree
                        S_max = copy.deepcopy(S)
                        liste = []
        for node,deg in S.degree_iter():
                        deg1 = S.degree(node, weight='weight')
                        liste.append([node, deg1])

        nodes_min = find_lowest_degree(liste);
```

```
                S.remove_node(nodes_min[0]);

    return S_max, avg_degree_max;
```

6.3 Average Degree : Min-Cut

```
def compute_number_of_edges (S) :
        sum = 0
        for node in S.degree_iter() :
                deg = S.degree(node, weight='weight');
                sum = deg.values()[0] + sum;
        return sum

def find_densest_subgraph(G):
        E=compute_number_of_edges(G);
        N=G.number_of_nodes();
        mind = 0;
        maxd = N;
        Vs = [];
        threshold = 1.0/ (N * (N-1));
        cut_value_max=0;

        while (maxd - mind )> threshold :
                g = (maxd + mind)/2.0
                liste_nodes = G.nodes(data = False)
                source = 'source'
                sink = 'sink'
                DG = nx.DiGraph()
                DG.add_node(source)
                DG.add_node(sink)

#We construct the Directed Graph before using min-cut algorithm
                for u,v in G.edges():
                        DG.add_edge(u,v, capacity = G.edge[u][v]['weight']);
                        DG.add_edge(v,u,capacity=G.edge[u][v]['weight']);
                for x in G.nodes():
                        if x != source and x!=sink:
                                DG.add_edge(source,x, capacity=E);
                for x in G.nodes():
                        if x != source and x!= sink :
                                deg = G.degree(x, weight='weight')
                                DG.add_edge(x,sink, capacity=E+2*g-deg);
                                cut_value, partition = nx.minimum_cut(DG,source,

                if (len(partition[0]) == 1 ) and (partition[0].pop()==source ):
                        maxd = g;
```

```
                else :
                        mind = g;
                        Vs = [x for x in partition[0] if x != source ]
                        cut_value_max=cut_value;
        length_Vs = len(Vs);

        V_subgraph = G.subgraph(Vs)
        avg_degree_max = 2.0*compute_number_of_edges(V_subgraph)/V_subgraph.numb

        return V_subgraph, avg_degree_max;
```

6.4 Maximum Clique

```
def find_densest_subgraph(G) :
        Slist=list(nx.find_cliques(G))
        avg_best = max(len(clique)
        for clique in Slist)
        cliques=[clique for clique in Slist if len(clique)==avg_best]
    avg_deg = 0;
        for clique in cliques:
                clique_subgraph = G.subgraph(clique)
                N = clique_subgraph.number_of_nodes()
                E = compute_number_of_edges(clique_subgraph)
        avg_deg = 0.5*(avg_deg+(2.0*E)/N)
    return cliques, avg_deg
```

References

[1] F. Rousseau and M. Vazirgiannis. Main Core Retention on Graph-of-Words for Single-Document Keyword Extraction. In ECIR, 2015.

[2] http://jmlr.org/papers/volume5/lewis04a/a11-smart-stop-list/english.stop

[3] http://www.tartarus.org/~martin/PorterStemmer

[4] A Survey of Algorithms for Dense Subgraph Discovery, Victor E. Lee, Ning Ruan, Ruoming Jin, Charu Aggarwal

[6] Hulth. Improved automatic keyword extraction given more linguistic knowledge. In Proceedings of EMNLP '03 , page 216–223, 2003