Ioana-Sabina STOIAN
sabina.stoian@gmail.com

Lab session # 1
ALTEGRAD 2019

10/13/19

1 Question 1

The density of the graph increases as the window size increases. This can be explained by the fact that the connections within the graph are created based on the co-occurences of words in the same window, thus more connections will appear in the graph when a larger window size is used. As there will be more connections, the overall number of edges will increase. By inspecting the formula of how the density of a graph is computed, the density is directly-proportional with the number of edges, thus the density of the graph increases. In Fig. 1, we show how the density of the graph increases as the size of the window increases from using a window of size 2 up to a size equal to the number of terms which resulted from the text after the preprocessing step, where the maximum density is reached.

We may observe that for small window sizes, the differences between densities are significant (e.g. for a window of size 2, the density is approximately equal to 0.106, and for a window of size 3, the density is 0.21). For greater window sizes, the density will increase slowly, up to not at all, from one value to another (e.g. for a window of size 16, the density is 0.7954, for a window of size 17, the density is equal to 0.803 and for a window of size 21, the density is equal to 0.803). So as the window size approaches the number of the words resulting from the preprocessing step, the density tends to reach the maximal value that can be reached for that specific graph.

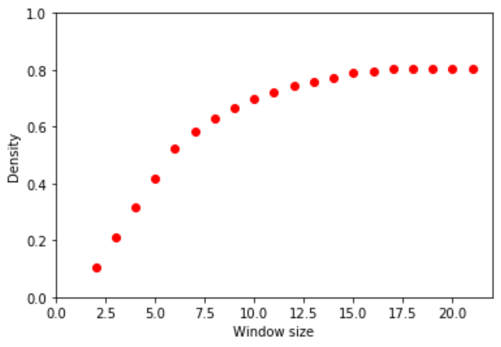

Figure 1: Variation of the density of the graph with the window size.

2 Question 2

For the graph $G(V, E)$, we denote the number of edges m ($m = |E|$), the number of vertices $n = |V|$ and the average number of neighbours of a node a. In the following, we will describe the time complexity of each line of the algorithm and, after that, we will evaluate the overall complexity of the algorithm. We suppose that both the list of vertices and the adjacency lists are linked lists and they are not indexed, so we need to go through the whole list in order to find a specific node. The adjacency list for a specific node will contain only the neighbours with whom that specific node is connected through outer edges (edges for which that specific node is the source and the neighbour is the target).

- Line 1: $p \leftarrow \{v : degree(v)\}, \forall v \in V$.

  In order to create the dictionary p that will contain every vertex with its associated degree, we need to go through the whole list of vertices (n steps) and for every vertex, we need to go through its adjacency list. In this way, we will go through all the edges of the graph and we can compute the degree of every vertex. In order to compute the degree of a specific vertex, it is not sufficient to count the number of neighbours in its adjacency list, because we will only compute the out degree of the node (we will count the edges for which that specific node is the source and the neighbour is the target). We also need to scan the entire list of lists in order to compute the degree of a single node and it will also be sufficient for computing the degree of all nodes at the same passing through the list of lists (by using the strategy that when we find a node in an adjacency list, we will increment the value from the dictionary for the key represented by the vertex). Thus we will execute $n \cdot a$ steps $\implies O(n \cdot a)$.

- Line 2: **while** $|V| < 0$ **do**:

  By looking at Line 6, we can see that every line of code within this while loop will be executed n times. The comparison step ($|V| < 0$) takes constant time, so $O(1)$.

- Line 3: $v \leftarrow$ element of p with lowest value.

  Considering the fact that the dictionary p is not sorted, in order to find the minimum degree vertex, we have to scan through the whole dictionary of vertices. For every vertex, we need to extract the value of its degree, which can be done in constant time, thus $O(1)$. We need to do this step for every vertex in the dictionary, so we do this step n times $\implies O(n)$.

- Line 4: $c[v] \leftarrow p[v]$

  This line is a simple assignment step, which can be split into smaller operations. The extraction of the value from a specific key from a dictionary ($p[v]$) can be done in constant time, so $O(1)$. Writing a new value at a specific key in a dictionary can also be done in constant time, so $O(1)$. Thus, this line of code requires constant time $\implies O(1)$.

- Line 5: $neighbors \leftarrow N(v; V)$

  In this line of code, we extract the neighbours of a vertex and we copy them to a new list. We need to go through the whole list of vertices and their adjacency lists. We need to find all the connections of the node v, not only the neighbours from its adjacency list, because we have a directed graph and the adjacency list contains only the neighbours incident to the outer edges (so we need to find all the neighbours of the node v incident to the inner edges). We thus need to go trough all the nodes and look at all their neighbours. Considering the average number of neighbours for a node, this operation can be done in $O(a)$ steps per node, resulting in an overall complexity of $O(n \cdot a)$ for this line.

- Line 6: $V \leftarrow V \setminus \{v\}$

  By executing this line of code, we delete the vertex node v (which we selected in the Line 3 as the vertex with the lowest degree) from the vertex list. Finding the node in the linked list costs $O(n)$ time complexity and deleting the list of its neighbours takes $O(a)$ (considering the fact that we need to free the allocated memory). The deletion of v takes $O(1)$, because the deletion of an element after finding it requires $O(1)$, so the overall computational complexity of the line is $O(n + a)$.

- Line 7: $E \leftarrow E \setminus \{(u; v) | u \in V\}$

  We need to go through every vertex of the list and its adjacency list of neighbours. On average, it will require $n \cdot a$ steps $\implies O(n \cdot a)$.

- Line 8: **for** $u \in neighbors$ **do**

  This line represents a simple passing through all the elements from a list, so this means that every instruction within the for loop will be executed a times.

- Line 9: $p[u] \leftarrow max(c[v]; degree(u))$

  This line of code is a simple maximum function between two numbers, which takes constant time. Both terms are obtained by extracting the value from a specific key in dictionaries ($c[v]$ and $degree(u)$, which is actually $p[u]$) $\implies O(1)$.

  Line 8 + Line 9 $\implies O(a)$ (the for loop requires $O(a)$ time complexity).

  All the complexity terms from Line 2 to Line 9 need to be multiplied by n because of the while loop at Line 2. The overall complexity of the algorithm will be:

  $$O(n \cdot a) + O(n \cdot (1 + n + 1 + (n \cdot a) + (n + a) + (n \cdot a) + a)) = O(n^2 \cdot a) \tag{1}$$

  a is the average number of neighbours for a node, so if we multiply it with the number of nodes, we will obtain the number of edges, so the overall complexity of the algorithm can also be written: $O(n \cdot m)$.

3 Question 3

In Table 1, we present the results achieved using 4 different algorithms on the same dataset: unweighted k-core decomposition, weighted k-core decomposition, PageRank and TF-IDF. We can observe that the maximum precision is obtained when using the weighted k-core decomposition algorithm and the maximum recall is achieved by the unweighted k-core decomposition algorithm, surpassing the other methods considerably. The maximum F-1 score is achieved by the unweighted k-core decomposition, which means that it gives the best result on the chosen dataset. This method has a very low precision (the lowest precision of all the used methods), but a very good recall.

The TF-IDF method is surpassed in performance (F-1 score) by all the other graph-based methods (weighted / unweighted k-core, PageRank) and this can be explained by the fact that it doesn't capture the dependencies and the order between the words (in TF-IDF, every word is considered independent from all the other words).

The PageRank algorithm is outclassed by both k-core decomposition methods (weighted and unweighted), because they also take into consideration the cohesiveness of the keywords in the graph-of-words, not only their centrality.

Method	Precision (%)	Recall (%)	F-1 score (%)
k-core (unweighted)	51.86	**62.56**	**51.55**
k-core (weighted)	**63.86**	48.64	46.52
PageRank	60.18	38.3	44.96
TF-IDF	59.21	38.5	44.85

Table 1: Comparison between results achieved with the the k-core algorithm (unweighted/weighted), Page Rank and TF-IDF.

4 Question 4

Advantages of the k-core approach:

- Unlike TF-IDF, it uses Graph-of-Words representations, so it retains information about term order (in a directed graph) and term dependency (the weighted k-core decomposition uses the weights of the graph as indicators of the strength of the dependency between two words).
- It also captures the cohesiveness of a keyword and not only its centrality (unlike in PageRank).
- The number of keywords does not need to be chosen ahead of time, because in this algorithm, the number of keywords is selected automatically based on the size of the graph.

Disadvantages of the k-core approach:

- The naive version of the k-core algorithm is computationally expensive.
- In the case of realising a computationally effective version of the method, the implementation is not easy, because it requires the use of more complex data structures.

5 Question 5

The k-core decomposition can be realised on parallel devices by using specific techniques and algorithms, as described in [1]. This can improve the overall time required by the algorithm because the computation can be distributed to multiple parallel nodes, thus the result will be computed faster.

Another possible improvement can be the use of an ensemble method that can combine all the results from the four different approaches (weighted k-core, unweighted k-core, PageRank and TF-IDF) and could use a majority vote to pick a fixed number of keywords. As an intuition for this, we may think that each algorithm can capture different types of information (e. g. k-core decomposition captures the cohesion of the keywords and TF-IDF captures the term frequency in multiple documents). Furthermore, based on the results depicted in Table 1, we may see that the unweighted k-core algorithm has a high recall but a low precision, while the weighted version of the algorithm has a high precision and a low recall. By combining these two methods, that have complementary advantages and disadvantages, we may obtain a more powerful and precise algorithm.

References

[1] Hossein Esfandiari, Silvio Lattanzi, and Vahab Mirrokni. Parallel and streaming algorithms for k-core decomposition. In *ICML'18*, 2018.