Université Libre de Bruxelles
INFO-F203
Coordinator: Jean CARDINAL

Report for the project concerning degeneracy, coloration and cores of a graph.

Author:

Enrollment number: 000509461 Name: Harsh DUA

Partner:

Enrollment number: 000478023 Name: Omar Guizani

April 29, 2022

Abstract

The report details algorithms and its features that compute :-

- The degeneracy of a given graph
- The coloring number of a given graph
- The core number of a given vertex

0.1 Implementation of a graph

The implementations described in this report assumes a graph implemented using an adjacency list, with the methods.

An abstract data type for the vertices was not implemented. I chose to represent them just as integers within the adjacency list, because it seemed redundant to implement a whole class that basically just acts like a wrapper for integers, with no additional functionality.

The adjacency list was implemented by a hash-map of integers to a set of integers. This way, retrieving the neighbors of a vertex, finding the degree of a vertex and the amount of vertices is done in O(1) time, The code is also more readable when looking through the neighbors.

1 Degeneracy

1.1 Algorithm to find the degeneracy

The degeneracy can be found by finding the maximum k value such that a k^{th}-core of a graph exists. The k-core of a graph is defined as a sub-graph H of G, where the minimum degree in H is greater or equal to the smallest k value. The algorithm to get the the k-core sub-graph is as follows:

```
kCore (graph, k)
    // $\delta(graph) \leq k$
    while $(\exists v : v \in graph.VertexSet \wedge deg(v) \leq k)$ do
        $graph.remove(v)$
    end
```

[1]

The intuition for the algorithm above comes from programming the procedure to turn a sketch of a graph into it's k-core graph by hand. To solve the problem manually, the decision making that one does is described as simply crossing out vertices that have a degree less than k, starting from the smallest one, while keeping in mind that removing a vertex will cause its neighbors to decrease their degrees by one, until the minimum degree of the graph is equal to k.

As defined in the project statement, the degeneracy of a graph is the largest k value for which the k-core graph exists. Thus, the algorithm to find the degeneracy can be described as follows :

```
findDegeneracy (graph )
    $k \leftarrow 0$;
    do
        $k \leftarrow k + 1$;
        $kCore(graph, k)$;
    while (! $graph.vertexSet.isEmpty()$);
    $k \leftarrow k - 1$;
    return $k$;
```

1.2 Complexity of the algorithm and comparison to the lower bound

As stated earlier, finding the kCore of a graph is solved manually by erasing every vertex that has a degree less than k, and updating the degrees of each neighbor, until the minimum degree on the graph is equal to k. Since finding the degeneracy can be defined as the maximum value of k such that a k-core exists, in other words, one less than the first k value that produces a null graph; trivially, there are |V| degrees that need to be removed, and each time a vertex has been removed, their corresponding neighbors must have their degrees updated. In essence, eventually all the edges will be accessed, from both sides, and thus it is dependant on |V| and 2|E|. Therefore, the lower bound of the complexity for any degeneracy algorithm is Ω(|V|+|E|). This is how the algorithm described above was programmed, so the algorithm runs at O(|V|+|E|), and is therefore efficient, but not necessarily optimized.

1.3 Implementation

Since the removal of vertices would evidently change the graph itself, any further computation on a graph would require reloading the graph. An approach to remedy this inspired by the procedure to obtain the degeneracy described by Matula and Beck, that uses a smallest-last ordering bucket queue and maintains an array of removed vertices[2],is described below

Two maps that act like a bidirectional map, vdMap and dvMap, were used. vdMap would map vertex V to an integer representing its degree, initially set to the degree(v); and dvMap would map a degree D to the set of vertices of degree D.

The condition that is stated in the while loop in the kCore procedure is calculated by finding the minimum degree mapped in dvMap. If the minimum degree is less than k, then the loop continues. Otherwise, it breaks. Finding the minimum key value in a hashmap can take O(n) time in the worst case. For this reason, a TreeSet dvKeys was implemented that simply held a sorted copy of dvMap.keySet(). This way, the minimum would be found in constant time. Setting dvMap as a treemap would not be efficient since many retrievals, insertions and deletions will be done while accessing the neighbors, which will run at O(log n) instead of O(1).

Each time a vertex needs to be removed, instead of making the graph delete the vertex, it would be added to a LinkedHashSet removedVertices and be considered removed. Then for each of the removed vertex's neighbor that is not in removedVertices, it's degree is reduced by 1 in dvMap and the vdMap is also correspondingly updated. A LinkedHashSet was used instead of an array so that the contains method can be used in O(1) time, instead of O(n), which is used to check whether a neighbor has been removed, and it can maintain the order of removal, which is used to color a graph (see section 2.2).

The algorithm establishes the maps in O(V) since there will be V entries, for both maps. Once established,finding the set of vertices, or the degree of a vertex, will be done in O(1).

The existence of vdMap might seem redundant, but finding the corresponding degree of each neighbor, would run in linear time without it, since I would have to look through all the keys in dvMap and see if it contains the neighbor within the value of the set of vertices. With vdMap, it will take constant time. All these calculations of runtimes is excluding the insanely unlikely event that a hash collision is encountered.

2 Vertex Coloring

2.1 Greedy Algorithm to find a proper coloring

A greedy coloring algorithm can be described as follows

```
greedyColoring (graph, vertexOrdering)
    for $v \in vertexOrdering$ do
        $assign\ v\ the\ smallest\ color\ that\ is\ not\ used\ by\ any\ N(v)$
    end
```

[3]

2.2 Proof of lemma that the chromatic number is less than or equal to k+1

A degenerate ordering of the vertices $v_1, v_2, ..., v_n$ is such that for any vertex v_i, has at most k-neighbors preceding it, where k is the degeneracy of the graph.

Let the ordered set D, $(v_1, v_2, ..., v_n)$, be the degenerate ordering of the vertices of a k-degenerate graph G.

Consider using the greedy coloring algorithm with the ordering D. When any v_i will be colored, since it has at most k neighbors that have been colored, it will have to choose a different color, which can at most be k+1. Thus, the degenerate ordering results in a proper vertex coloring of a graph that uses a maximum of k+1 colors.Therefore, the chromatic number is bound to be a maximum of k+1. [3]

2.3 Description of a coloring algorithm that uses at most k+1 colors

If the findDegeneracy function, saves the order of removal, from last to first, then it saves the degenerate ordering. Then, as explained in the previous subsection, a call to the greedyColoring routine with the degenerate ordering as input, results in a proper coloring of the graph using at most k+1 colors. The order of removal is last to first to minimize the need to use a new color, by dealing with the vertices from the highest core to the lowest core.

2.4 Complexity of the algorithm and comparison to the lower bound

The algorithm to obtain a proper coloring mapping, would obviously have to look at each neighbor that has already been colored, to make sure the vertex being mapped does not get a color of it's neighbors. This would mean the lower bound to color a graph is (|V|+2|E|), since all the vertices will be iterated to be colored, and all the edges will be iterated over, from both ends, to make sure the color of the vertex is not shared by any of the neighbors. The greedyColoring algorithm runs at this time complexity since it performs the computation that was just described.

The algorithm to find a proper coloring of the graph with k+1 colors, needs to run a modified version of findDegeneracy, where the removed vertices are saved in a list, descending in order of removal. Maintaining a list will not affect the time complexity, so it still runs at O(|V|+|E|). Then, the algorithm needs to run greedyColoring with the ordering produced by findDegeneracy, which also runs at O(|V|+|E|). Therefore, the algorithm has a time complexity is O(|V|+|E|), and is efficient.

2.5 Implementation in Java

A new function, findDegenOrdering, was implemented. The function is exactly the same as findDegeneracy described in section 1.3, but returns removedVertices instead of k. The k value is modified within the function instead. An atomic integer was passed as a parameter to achieve this.

After which, the vertices would be popped from the ordering, one by one, until all the vertices will be colored. Each vertex would be assigned the smallest color that is not shared by their neighbors.

3 Vertex Core Number

3.1 Algorithm to find a core number of a vertex

The findDegeneracy routine essentially calls kCore, core by core, and kCore removes all vertices in the cores preceding the k^{th} core. Or rather, it finds the kCore of the last vertex to be removed, ie: the vertex within the maximum non void kCore. This can be modified to stop when the target vertex instead, rather than the very last vertex. Then simply, the core number of the vertex will be the value of the core of the previous iteration. The algorithm would be as follows:

```
findCoreNumber (graph, targetVertex)
    $k \leftarrow 0$
    while $(!graph.VertexSet.isEmpty())$ do
        $k+=1$
        while $(\exists v : v \in graph.VertexSet \wedge deg(v) \leq k)$ do
            if $v = targetV$ then
                $return\ k-1$
            end
            $graph.remove(v)$
        end
    end
```

3.2 Lower bound on finding the core number of a vertex

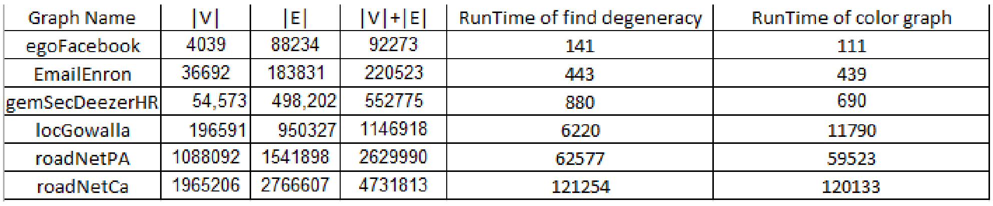
Consider the vertex v that is last removed when findDegeneracy is called. This vertex, has a core number, equal to the degeneracy. If findCoreNumber(v) was called, the function becomes identical to findDegeneracy. Therefore, the lower bound is exactly the same as finding the degeneracy, $\Omega(|V| + |E|)$. Since findCoreNumber becomes identical in the worst case to findDegeneracy, their complexities are also identical. Thus, the algorithm runs at $O(|V| + |E|)$ for the same reasons mentioned in section 1.2, and is efficient.

3.3 Implementation

The kCore function was modified to include a default parameter targetVertex, which was achieved by overloading the function. If targetVertex is not provided, the function works normally. Else, it will return true when targetVertex has been removed. Then for k in 1,2,.. infinity, kCore(k) is called, and if it returns true, then the targetVertex belongs in the core k-1.

4 Experiments on real data

The graphs following graphs from SNAP were tested and their results were documented below:

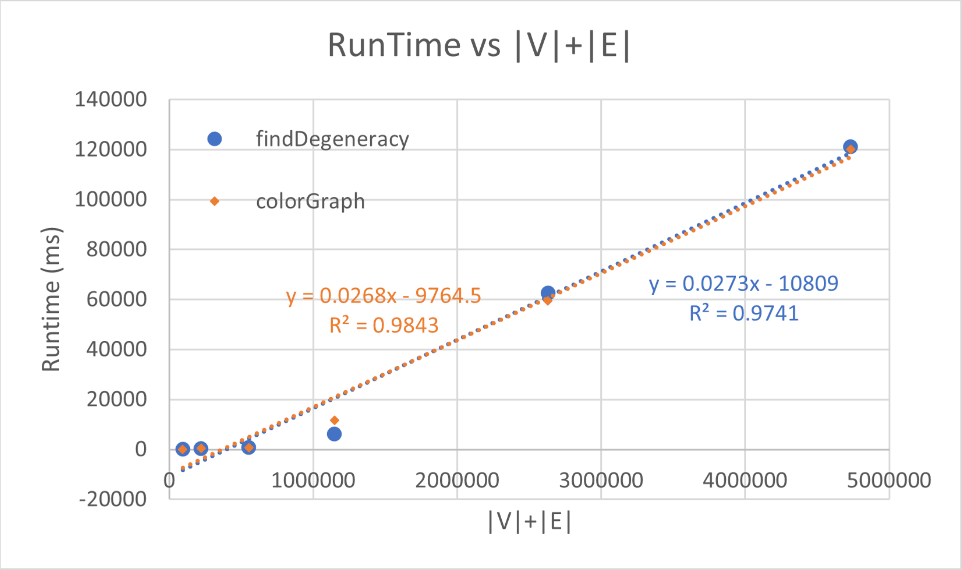


Graph Name	|V|	|E|	|V|+|E|	RunTime of find degeneracy	RunTime of color graph
egoFacebook	4039	88234	92273	141	111
EmailEnron	36692	183831	220523	443	439
gemSecDeezerHR	54,573	498,202	552775	880	690
locGowalla	196591	950327	1146918	6220	11790
roadNetPA	1088092	1541898	2629990	62577	59523
roadNetCa	1965206	2766607	4731813	121254	120133

I chose these particular graphs, because each graph has around double the value of |V|+|E| as the previous one, since all the algorithms' run-time is O(|V|+|E|). This allowed me to test a wide range of graphs that vary a lot in terms of their nodes and edges. Undirected graphs on the SNAP website that had more edges than roadNetCA were not able to be created as they are way larger, to large to even parse the file to the graph.

In general, we see a linear increase in time, this is confirmed by plotting the runtimes against the |V|+|E| and seeing the coefficient of determination of a linear relationship being close to 1.00, as shown in the graph on the next page.

I would say my algorithms are as optimized as I can make it to be, within the timeframe I was able to implement everything. In reality it does not seem to work well on big graphs, such as roadNetCA and roadNetPA. I have done all I can to make sure that any processing of data is done in constant time by using data structures such as sets and hashmaps, to have constant time lookup, retrieval, insertion and deletion, and to minimize computation where possible by for example, deleting empty entries in the dvmap as the vertices would get deleted, to minimize the time it takes to lookup for the minimum value of keys. Unfortunately, I am blind to how I can further make my algorithm more efficient.I was told by my peers that it should take seconds to find the



degeneracy on the roadNet graphs, yet it takes me minutes.

It is odd that it takes slightly less time to color the small graphs because it uses the same function as findDegeneracy, but does way more computation after, since it has to assign colors to the graph. I have no explanation as to why this is the case. Finding the core number of a particular vertex would depend at which core it was located, but obviously, in the worst case, it took the same amount of time as finding the degeneracy, so it was not included in the data since it would be redundant. However, this was tested and it was the case.

All the methods were tested by creating external tests, for example, the proper coloring of a graph was verified by making a method that checks that no vertex has the same color as any of its neighbors. The degeneracy was simply verified by first doing tests on small graphs, and double checking that the order of removal, and the degeneracy achieved was the same as what I would have done manually. For larger graphs, an answer that was the same as other students was a sign of approval. The cores were simply verified by choosing the last removed vertex while finding the degeneracy as the target vertex, and seeing if the result was equal to the degeneracy. I also chose random vertices and asked other students what they got. I realize however, that it is not scientific to rely on the results of others since it is possible that everyone just did not solve it correctly.

References

[1] Allan Bickle. *The K-cores of a graph*. Dec. 2010.

[2] David W. Matula and Leland L. Beck. “Smallest-last ordering and clustering and graph coloring algorithms”. In: *Journal of the ACM* 30.3 (1983), pp. 417–427. DOI: 10.1145/2402.322385.

[3] Adrian Kosowski and Krzysztof Manuszewski. *Classical coloring of graphs*. 2004.