MITRO209 Project

Pedro Orlando

January 2022

1 Introduction

The main objective of this project is to implement Charikar's greedy 2-approximation algorithm for the densest subgraph problem as seen in course, which was implemented in Python3 and tested on some of the graphs available from SNAP's Large Network Dataset Collection of various shapes and sizes.

In this project, the following datasets were used:

- `twitch` [1]: Graph representing connections between different English-speaking streamers on Twitch
- `facebook` [2]: Friends lists on Facebook
- `wiki` [3]: Paths in the online game Wikispeedia
- `deezer` [4]: Friendship networks of users from 3 European countries on Deezer
- `git` [5]: A large social network of GitHub developers which was collected from the public API in June 2019
- `fb-artist` [6]: Data from Facebook pages from November 2017, "Artist" category
- `dblp` [7]: DBLP collaboration network
- `twitter` [8]: Social circles on Twitter
- `google` [9]: Graph representing websites and hyperlinks between them, released in 2002 by Google
- `twitch-gamers` [10]: A social network of Twitch users which was collected from the public API in Spring 2018
- `california` [11]: Road network of California
- `internet` [12]: Internet topology graph from traceroutes ran in 2005

2 Data Structure

The code uses one main class called `Graph` to initialise the data and compute the algorithm, containing the following attributes:

- `edges` (dict of sets): contains every node and its edges as an adjacency list – except instead of lists it uses sets;
- `degrees` (dict of sets): for every degree d in the graph, contains a set of every node with degree d;
- `nodes` (dict of ints): for every node, saves their degree, essentially a sort of dual of `degrees`;
- `n_edges` (int): total number of edges in the graph;
- `n_nodes` (int): total number of nodes in the graph;
- `density` (float): average degree density of the graph, i.e. $\rho(G) = \frac{|E|}{|V|}$;
- `min_deg` (int): keeps track of the current minimum degree, avoiding re-calculating this at every iteration.

As one can tell, the main data structures used in this class are the `dict` and the `set` classes in Python, which are very similar in implementation: both being hash maps. The difference with sets is that the "key" part of the dictionary is an internal hash key produced by Python and is not shown when manipulating the set. This way we have uniqueness with the values in the set and the keys in the dicts, and constant time for lookups, inserts (via `set.add()`) and deletes (via `set.remove()`).

In the following example we'll denote dictionaries as tuples of `{key: value, ...}` while we'll denote sets by `{value, ...}`.

Let's take the following simple graph as an example:

$$\begin{tikzcd} \bullet_A \arrow[r, dash] & \bullet_B \arrow[r, dash] & \bullet_C \end{tikzcd}$$

In this case, the Graph class would have the following attributes:

```
edges   = {A: {B}, B: {A,C}, C:{B}}
degrees = {1: {A,C}, 2: {B}}
nodes   = {A: 1, B: 2, C: 1}
n_edges = 2
n_nodes = 3
density = 2/3
min_deg = 1
```

3 Implemented code

The code works in three parts: Graph building in section 3.1, the algorithm for the 2-approximation to a densest subgraph in section 3.2, and the reconstruction of the densest subgraph found by the algorithm in section 3.3. All of the implementations can be found in Appendix A.

3.1 Graph building

In order to build the class as defined above, a file is expected in the form of pairs of nodes (u, v) representing an edge with a separator between them. In the previous example, this would correspond to the following, with a comma as a separator:

```
A,B
B,C
```

The `edges`, `degrees` and `nodes` dictionaries are initialised using Python's `defaultdict` to simplify the process of adding an edge to a new node that wasn't yet added.
The `n_edges`, `n_nodes` and `density` attributes are initialised as 0, while `min_deg` is initialised as $+\infty$.

We then run the following, over the file as specified previously:

```
1: for line in file do
2:     Split the line on the separator, save the nodes $n$ and $t$
3:     if $n \neq t$ then
4:         Save $t$ on $n$'s edges
5:         Save $n$ on $t$'s edges
```

Notice how, since `edges[n]` and `edges[t]` are sets, repeated edges will be ignored, because each element in a set appears only once.
With this, the resulting graph will be a simple (undirected) graph.

Afterwards, we define the rest of the attributes of the graph with a loop over the now-defined nodes:

```
1: for $v \in V_G$ do
2:     $d \leftarrow |E_v|$
3:     Add $v$ to degrees[d]
4:     Save $d$ as $v$'s degree on nodes
5:     n_nodes ← n_nodes + 1
6:     n_edges ← n_edges + $d$
7:     If $d <$ min_deg: min_deg ← $d$
8: n_edges ← n_edges / 2          ▷ Divide n_edges by 2 since we're counting each edge twice
9: density ← n_edges/n_nodes
```

3.2 Algorithm

The code implemented in this project is a variation on the original densest subgraph algorithm shown in class where instead of saving the result directly to another graph H, we keep track of which nodes need to be removed in order to rebuild the final densest subgraph. This is done, particularly, to avoid having to copy the entirety of G every time we compute a denser subgraph.

Algorithm 1 Densest Subgraph Algorithm Variation

```
1: $max_G \leftarrow \rho(G)$
2: to_remove : empty list
3: densest_to_remove : empty list
4: while G has nodes do
5:     let $v$ be the node with minimum degree $\delta_G(v)$ in $G$
6:     remove $v$ and all its edges from $G$
7:     save $v$ in to_remove
8:     if $\rho(G) > max_G$ then
9:         $max_G \leftarrow \rho(G)$
10:        update densest_to_remove with the values in to_remove
11:        empty to_remove
12: return densest_to_remove
```

This function is called `densest_subgraph()` and it's a method of the Graph class. It contains three major operations:

1. Getting the node v with minimum degree, called `__update_minimum_degree()`;
2. Removing the node v and its edges from the graph, called `remove_node(v)`;
3. Managing the list of nodes to remove, which is done inside of `densest_subgraph()`.

3.2.1 Getting a node with minimum degree

For this, we mostly use the `min_deg` attribute of the class, updating it during the removal of a node. However, there is a case that must be taken care of: when we no longer have nodes with the previous `min_deg`, we're forced to search for the next available minimum degree.

Because of this case, the method `__update_minimum_degree()` is called, which checks if there are still nodes with the current minimum degree, only calling Python's `min()` when we've exhausted them.
This, in practice, runs in $\mathcal{O}(1)$ since the `min` function is only called when there are no longer nodes with `min_deg`, which is not enough to be noticeable because even in such cases it runs in $\mathcal{O}(\log n)$ in Python.
Further evidence for this will be given in section 4.1.

Once d_t has been well-defined, the code runs a `set.pop()` on `degrees[` d_t `]` to get some node with minimum degree, this operation being of constant time complexity.

3.2.2 Removing a node

For removing a node, the `remove_node(v)` method is called, which consists of a loop over all nodes t connected to v, i.e. a loop over $E_v = \{t \in V_G \mid (v, t) \in E_G\}$, followed by the removal of v itself from the Graph object.

In this method, we do the following operations:

```
1: for $t \in E_v$ do
2:     Remove $v$ from the set of edges of $t$, i.e. edges[t]
3:     Update the position of $t$ in the degrees dictionary:
4:     Remove $v$ from degrees[$d_t$], where $d_t$ is the degree of $t$
5:     $d_t \leftarrow d_t - 1$                                    ▷ Update $t$'s degree
6:     if $d_t > 0$ then
7:         Add $t$ to degrees[ $d_t$ ]
8:         If $d_t <$ min_deg : min_deg $\leftarrow d_t$
9:     else
10:        Remove $t$ from the graph
11: Remove $v$ from the graph
```

This step of the code has complexity $\mathcal{O}(1 + d_v)$, where d_v is the degree of the node v, most of the time the algorithm takes to run comes from this function, due to the amount of lookups, inserts and deletes that are called in it.

3.2.3 List of nodes to remove

In order to keep track of what's the densest subgraph found by the algorithm, we need a way to record it at each iteration. For this, we use two lists: `to_remove` and `densest_to_remove`.

The latter being a list of all nodes that need to be removed in order to achieve the final densest subgraph produced by the algorithm, and the former being kept as a temporary list of nodes to remove, updating `densest_to_remove` every time we find a denser subgraph than the one we already have, and being reset when this happens.

By the end of the algorithm, the resulting list `densest_to_remove` will contain all of the nodes that need to be removed from the original graph G in order to reconstruct its densest subgraph.

All operations run in $\mathcal{O}(1)$ thanks to the `.append()` and `.extend()` methods for lists.

3.3 Graph rebuilding

After the result from the `densest_subgraph()` method is acquired, the `densest_to_remove` list returned is converted into a set so that we may reconstruct the densest subgraph found by the algorithm.

This is done in a very similar fashion described in section 3.1 except we check if either of the nodes `n` or `t` at each line are in `densest_to_remove` (which is done in $\mathcal{O}(1)$ because it's now a set): if not, then both are in the densest subgraph and we add them.

This step has the same complexity as the graph building step at worst-case (when the original graph is already the densest). In general, however, it's much faster, as it runs $|E| - |$`densest_to_remove`$|$ accesses to memory and inserts.

4 Results

4.1 Complexity analysis

The building complexity, as mentioned in section 3.1, is linear on the size of the dataset. More specifically, the initial loop runs $|E|$ times, with two constant time operations (adding the two nodes to their respective `edges` dictionary, i.e. a lookup on a dictionary and an insert on a set).
The loop to set the rest of the attributes is over all nodes and thus runs $|V|$ times, and only containing therein constant time operations.
Therefore the building phase has overall complexity $\mathcal{O}(|V| + |E|)$.

As for the `densest_subgraph()` function, it has a major loop over all of G's nodes, in it calling its three sub-procedures: getting a node with minimum degree (3.2.1), removing said node and its edges (3.2.2) and managing the `densest_to_remove` list (3.2.3).
The complexity of the third part being rather straightforward as $\mathcal{O}(1)$, we're left with the other two:

The removal of the node v, as discussed previously, is of $\mathcal{O}(1 + d_v)$. Due to the fact that this function is called as a part of a loop over all nodes, its complexity is: $\mathcal{O}\left(\sum_{v \in V_G}(1 + d_v)\right) = \mathcal{O}(|V| + |E|)$.
And, finally, to argue about the complexity of getting a node with minimum degree, take the worst-case example of k isolated cliques of various sizes $(d_1, ..., d_k)$ – the `min()` function is called when we run out of nodes with the current minimum degree, which would happen every time the algorithm removes completely a clique. Python's built-in `min()` being $\mathcal{O}(\log n)$ on a set of integers means we have final complexity $\mathcal{O}(k \log |V|)$ for this part when ran in the `densest_subgraph()` method.

But for large values of k, we have: $k \log |V| \ll k|V|$, since $|V| \propto c \cdot k$, where $c > 1$ is some value depending on the degrees of the cliques.

As such, since $|V|$ grows at a faster rate than k, for large datasets, the calls to `min()` are overshadowed by the linear complexity of the node removal part of the algorithm, and for small datasets it does not make a significant impact on the performance. This is seen on Figure 3.

Thus the complexity of this implementation of the densest subgraph algorithm is:

$$\mathcal{O}(1 + |V| + |E| + k \log |V|) \rightarrow \mathcal{O}(|V| + |E|)$$

4.2 Plotting the run time

Let's analyse the run time of the code for each of the datasets to argue that the implementation is linear on the graph size. All time data is measured in seconds.

In Table 1, we can see the data for the graph building phase, which is over the original size of the graph, plotted in Figure 1:

Source	Original #V	Original #E	Total Size (V+E)	Graph Building
twitch	7,126	35,324	42,450	0.0397
facebook	4,039	88,234	92,273	0.0687
wikispeedia	4,592	119,882	124,474	0.1341
git	37,700	289,003	326,703	0.3042
deezer	54,573	498,202	552,775	0.5580
fb-artist	50,515	819,306	869,821	0.8885
dblp	317,080	1,049,866	1,366,946	1.7029
twitter	81,306	2,420,766	2,502,072	2.6844
google	875,713	5,105,039	5,980,752	8.0068
twitch-gamers	168,114	6,797,557	6,965,671	8.0899
california	1,965,206	5,533,215	7,498,421	9.2416
internet	1,696,415	11,095,298	12,791,713	19.165

Table 1: Original graph size and graph building time

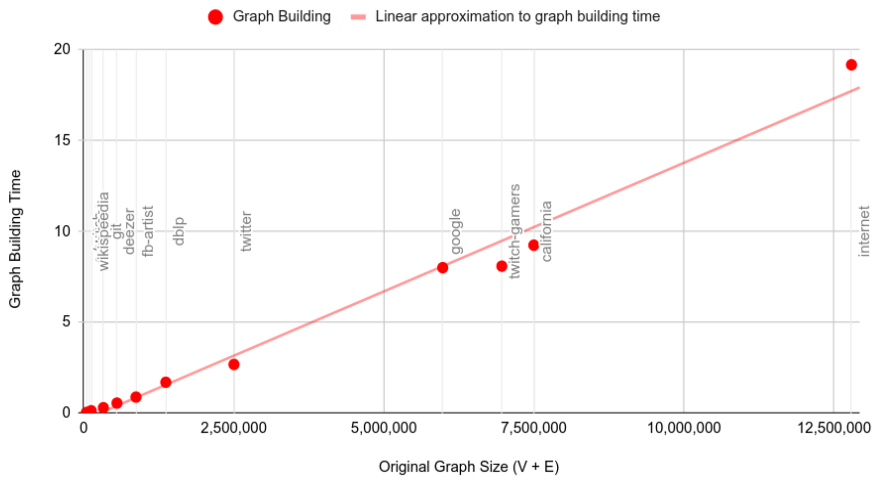


Figure 1: Graph build time plotted against original graph size

The time elapsed for the graph building phase seems to follow a linear relationship with the total size of the dataset.

In Table 2, we can see the processed graph size (after removing self-loops and repeated edges) and algorithm run time, followed by its plot on Figure 2:

Source	Processed #V	Processed #E	Processed V+E	Algorithm Time
twitch	7,126	35,324	42,450	0.0633
facebook	4,039	88,234	92,273	0.1118
wikispeedia	4,592	106,537	111,129	0.1639
git	37,700	289,003	326,703	0.5596
deezer	54,573	498,202	552,775	1.1205
fb-artist	50,515	819,090	869,605	1.6467
dblp	317,080	1,049,866	1,366,946	2.9276
twitter	81,306	1,342,296	1,423,602	2.9572
california	1,965,206	2,766,607	4,731,813	8.5874
google	875,713	4,322,051	5,197,764	9.9867
twitch-gamers	168,114	6,797,557	6,965,671	16.905
internet	1,696,415	11,095,298	12,791,713	32.693

Table 2: Processed graph size and algorithm running time

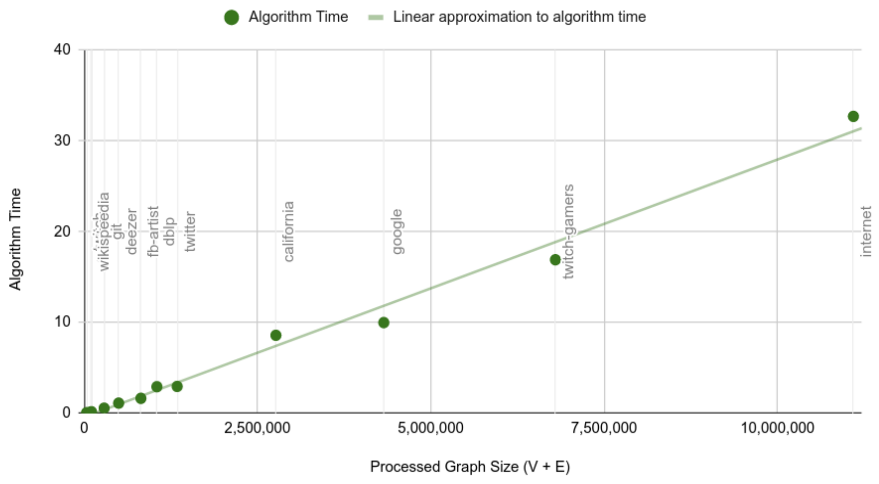


Figure 2: Algorithm time plotted against processed graph size

Thus, the algorithm run time seems to follow a linear relationship with the size of the processed graph, even for datasets with total size over $10,000,000$.

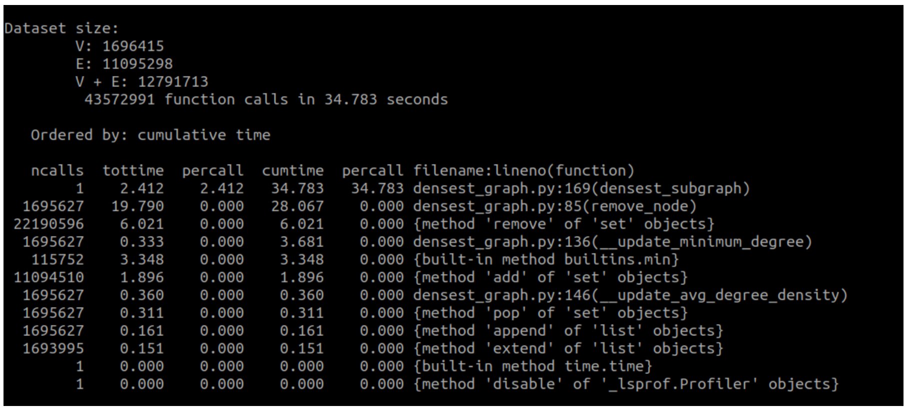
Finally, the resulting number of nodes, edges and density for the densest subgraphs given by the algorithm along with their rebuild time:

Source	# V in Solution	# E in Solution	Max Density	Rebuild time
twitch	459	5,475	11.9281	0.0135
facebook	205	15,624	76.2146	0.0313
wikispeedia	436	17,739	40.6858	0.0692
git	713	21,565	30.2454	0.1173
deezer	6,209	101,646	16.3708	0.2710
fb-artist	1,726	100,219	58.0643	0.3426
dblp	114	6,441	56.5	0.4783
twitter	734	50,216	68.4142	1.1192
google	240	6,523	27.1792	2.0457
twitch-gamers	4,372	600,965	137.458	3.5455
california	17,006	28,984	1.7043	2.6281
internet	431	38,437	89.181	4.3268

Table 3: Size of the densest subgraph produced by the algorithm and time elapsed to rebuild it

4.3 Code Profiling

Using the cProfile library in Python, we're able to analyse in-depth how many times each function is called and how long each call takes. Image 3 is an excerpt of this analysis for the `internet` dataset. For a brief explanation of the columns available: `ncalls` is the number of calls made to the function, `tottime` is the total time spent in a function (not counting subfunctions), and `cumtime` is the cumulative time spent in a function (counting subfunctions). Each `percall` column is just the previous column divided by `ncalls`.



```
Dataset size:
        V: 1696415
        E: 11095298
        V + E: 12791713
         43572991 function calls in 34.783 seconds

   Ordered by: cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    2.412    2.412   34.783   34.783 densest_graph.py:169(densest_subgraph)
  1695627   19.790    0.000   28.067    0.000 densest_graph.py:85(remove_node)
 22190596    6.021    0.000    6.021    0.000 {method 'remove' of 'set' objects}
  1695627    0.333    0.000    3.681    0.000 densest_graph.py:136(__update_minimum_degree)
   115752    3.348    0.000    3.348    0.000 {built-in method builtins.min}
 11094510    1.896    0.000    1.896    0.000 {method 'add' of 'set' objects}
  1695627    0.360    0.000    0.360    0.000 densest_graph.py:146(__update_avg_degree_density)
  1695627    0.311    0.000    0.311    0.000 {method 'pop' of 'set' objects}
  1695627    0.161    0.000    0.161    0.000 {method 'append' of 'list' objects}
  1693995    0.151    0.000    0.151    0.000 {method 'extend' of 'list' objects}
        1    0.000    0.000    0.000    0.000 {built-in method time.time}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

Figure 3: Code profiling of the densest_subgraph() function

Here we can see that most of the complexity of the `densest_subgraph()` function comes from the `remove_node()` method which, by itself, has a noticeable amount of lookups, inserts and deletes. Despite all of these operations being $\mathcal{O}(1)$ with our chosen data structure (they're all run on dictionaries and sets), they certainly add up as the number of edges in the graph goes up.
Notice, particularly, how the cumulative time of Python's built-in `min()` function is only 3s – a comparatively small fraction when compared to the total run-time of the function (34 seconds).

Another interesting part is the number of calls made to each function, with `remove_node()` being called $|V|$ times, in which we call `set.remove()` $2|E|$ times and `set.add()` $|E|$ times, while Python's `min()` is only called one tenth of the time.

The difference in `tottime` and `cumtime` in the `densest_subgraph()` and `remove_node()` functions comes from all the lookups being performed in each one – rather noticeable in `remove_node()` as there are 6 lookups on the huge dictionaries `edges`, `degrees` and `nodes`.

The biggest drawback to this implementation is the space complexity – with data structured around the worst-case for a hash map (that is, data that may have a lot of clashes in the hash function, or data that may have very large names for the nodes and thus large space complexity for storing the hashes), the graph construction phase can take a surprisingly long time, or even run out of memory.
That being said, after the construction, the `densest_subgraph()` algorithm still runs in linear time.

5 References

[1] *musae-twitch*. URL: http://snap.stanford.edu/data/twitch-social-networks.html.

[2] *ego-Facebook*. URL: http://snap.stanford.edu/data/ego-Facebook.html.

[3] *wikispeedia*. URL: http://snap.stanford.edu/data/wikispeedia.html.

[4] *gemsec-Deezer*. URL: http://snap.stanford.edu/data/gemsec-Deezer.html.

[5] *musae-github*. URL: http://snap.stanford.edu/data/github-social.html.

[6] *gemsec-Facebook*. URL: http://snap.stanford.edu/data/gemsec-Facebook.html.

[7] *com-DBLP*. URL: http://snap.stanford.edu/data/com-DBLP.html.

[8] *ego-Twitter*. URL: http://snap.stanford.edu/data/ego-Twitter.html.

[9] *web-Google*. URL: http://snap.stanford.edu/data/web-Google.html.

[10] *twitch-gamers*. URL: http://snap.stanford.edu/data/twitch-gamers.html.

[11] *roadNet-CA*. URL: http://snap.stanford.edu/data/roadNet-CA.html.

[12] *as-Skitter*. URL: http://snap.stanford.edu/data/as-Skitter.html.

Appendix A

Code Implementation in Python

The following are all the implementations mentioned in this project, so as to facilitate reading. Note that the following functions are all methods in the Graph class.

A.1 Graph initialisation

```
def __init__(self, filepath = '', sep = '', nodes_to_remove = set()):
    self.edges = defaultdict(set)
    self.degrees = defaultdict(set)
    self.nodes = defaultdict(int)

    self.n_edges = 0
    self.n_nodes = 0
    self.density = float(0)
    self.min_deg = float('inf')

    # We're transforming the graph into a simple graph
    with open(filepath) as f:
        csvfile = csv.reader(f, delimiter=sep)
        for n, t in csvfile:
            if n != t: # Ignore self-loops
                # To simplify the "rebuilding the graph" part of the code,
                # it's O(1) because it's a set
                if n not in nodes_to_remove and t not in nodes_to_remove:
                    # Add each edge once but make it bi-directional
                    self.edges[n].add(t)        # O(1) because it's a set
                    self.edges[t].add(n)        # O(1) because it's a set

    # Defining the other attributes by looping over all nodes
    for node, edges in self.edges.items():
        d = len(edges)  # degree of this particular node

        self.degrees[d].add(node)   # adding this node to the `degrees` dict
        self.nodes[node] = d    # adding its degree to the `nodes` dict

        self.n_nodes += 1   # +1 node to total
        self.n_edges += d   # +d edges to total

        # Computing min_deg while we build the initial Graph
        if d < self.min_deg:
            self.min_deg = d

    # NOTE: since we're building an undirected graph and duplicating every edge,
    #       we must divide n_edges by 2 to take that into account
    self.n_edges = self.n_edges//2

    # And, lastly, computing the initial density
    self.__update_avg_degree_density()
```

Listing 1: Graph initialisation

Where the `__update_avg_degree_density()` method simply returns $\frac{|E|}{|V|}$ as follows:

```
def __update_avg_degree_density(self):
    self.density = self.n_edges / self.n_nodes if self.n_nodes else 0 # sanity check for the
    case n_nodes = 0, which happens in the final iteration
```

A.2 Densest subgraph function

The code for the main body of the algorithm, `densest_subgraph()`:

```
def densest_subgraph(self):
    max_den = self.density
    to_remove = list()
    densest_to_remove = list()

    # repeat while G isn't empty
    while self.edges:
        # find v in G with minimum degree
        self.__update_minimum_degree()
        min_nodes = self.degrees[ self.min_deg ]
        v = min_nodes.pop()

        if not self.degrees[self.min_deg]:
            del self.degrees[ self.min_deg ]

        self.remove_node(v) # remove v and its edges from G
        to_remove.append(v)

        if self.density > max_den:
            max_den = self.density
            densest_to_remove.extend(to_remove)
            to_remove = list()

    return densest_to_remove
```

Listing 2: densest_subgraph method

A.2.1 Update minimum degree function

The method to update the minimum degree if needed, `__update_minimum_degree()`:

```
def __update_minimum_degree(self):
    # If there are no longer nodes with the current min_deg, look for the new min_deg in the
    graph
    if self.min_deg not in self.degrees:
        self.min_deg = min(self.degrees)
```

Listing 3: Updating minimum degree

A.2.2 Remove node function

The `remove_node()` method is as follows:

```python
def remove_node(self, v):
    for t in self.edges[v]:
        # For each target node t connected to v, we remove v from their edges
        self.edges[t].remove( v )                      # O(1) with a set

        # Updating their position on the degrees dict
        degree_t = self.nodes[t]
        self.degrees[ degree_t ].remove( t )        # O(1) with a set

        # Updating total number of edges and t's degree
        self.n_edges  -= 1
        self.nodes[t] -= 1

        # If t's new degree is non-zero, place it accordingly in the 'degrees' dict
        if degree_t - 1 > 0:
            self.degrees[ degree_t - 1 ].add( t )      # O(1) with a set

            # And update min_deg in case t now has a smaller degree than the previous min_deg
            if degree_t - 1 < self.min_deg:
                self.min_deg = degree_t - 1

        # Removing t from the graph in case it has no more edges,
        # and subtracting 1 to total number of nodes
        else:
            del self.edges[ t ]              # O(1) with a dict
            del self.nodes[ t ]              # O(1) with a dict
            self.n_nodes -= 1


        # Removing the entry for t's old  degree in case it's now empty
        if not self.degrees[ degree_t ]:
            del self.degrees[ degree_t ]             # O(1) with a dict

    # Removing the node v from the graph
    self.n_nodes -= 1
    del self.edges[v]                                # O(1) with a dict
    del self.nodes[v]                                # O(1) with a dict

    # And, finally, saving the new density
    self.__update_avg_degree_density()
```

Listing 4: Remove node v from graph

This is where the `densest_subgraph()` algorithm takes the longest time, due to the amount of lookups and calls to `add()` and `remove()`.