Minimum-Domination-Set

Methods

Graph Representation

A Graph is represented by:

- vertices: Set of Vertices
- edges: Dictionary of a Vertice as Key and a Set of Vertices as Value

The **vertices** are represented by a Set of Vertices (`Set[Vertice]`), this is optimal because we can not have the same vertice twice inside this Set and also provides us with $O(1)$ search complexity.

The **edges** are represented by a Dictionary containing a Vertice as a key and a Set of Vertices as the value (`Dict[Vertice, Set[Vertice]]`). This structure also provides $O(1)$ search complexity for both the values and the keys.

Generating the graphs

We need to test the algorithms proposed by this project, to fulfill that, graphs need to be generated in order to check results.
To generate graphs we first need to initialize a Graph object

```
graph = Graph()
```

Then we can use the `generate()` method which generates the graph based on some user input such as:

Argument	Type	Function
number_of_vertices	Integer	Number of vertices inside the graph
medium_number_of_edges	Integer	Medium number of edges connecting each vertice

Argument	Type	Function
width	Integer	Width of the canvas where the graph is represented
height	Integer	Height of the canvas where the graph is represented
minimum_distance_between_vertices	Integer	Mininum distance between each vertice
connect_with_closest	Boolean	Connect each vertice with the *n* closest vertices
verbose	Boolean	Prints information about the operation

The graph is constructed in **two main steps** by the `generate()` method:

1. First we need to generate the vertices
2. Then we can generate the edges

Vertices Generation

The vertice generation is of complexity $O(n)$ because iterates though the number of vertices to generate

```
for i in range(number_of_vertices):
    new_vertice = Vertice(name = str(i))
    vertices.add(new_vertice)
```

There is small differentes in this function based on if it is needed to generate also the `width`, `height` and also based on `minimum_distance_between_vertices`. However these are negligable, for example, when both the `width` and `height` are different than `None`, then

```
new_vertice = Vertice(
    name = str(len(vertices)),
    x = random.randint(0, width),
    y = random.randint(0, height)
)
```

Edges Generation

The edges are more complex to generate than the vertices based on the argument `connect_with_closest` because if `True`, it means we need to look to the closest vertices from all

the others vertices to discover which are closer. This is very helpful when debuging the complex algorithms because even for graphs with just 100 vertices it is almost impossible to figure out what vertices the edges are connecting. The implementation of the generation is done by the following pseudo-code

```
for vertice in vertices:
    if connect_with_closest:
        closest_vertices = get_closest_vertices(vertice = vertice, n = remaining_number_
        for vertice_to_connect in closest_vertices:
            add_edge(vertice, vertice_to_connect)
    else:
        vertices_to_connect = random.sample(self.vertices, remaining_number_of_edges)
        for vertice_to_connect in vertices_to_connect:
            add_edge(vertice, vertice_to_connect)
```

The `get_closest_vertices()` method is implemented with a *priority queue* in order to achieve aproximatly $O(log(n))$ complexity instead of the python `sorted()` which runs in $O(nlog(n))$. When generating large graphs, this difference is very noticiable because this function runs for every vertice.

Auxiliary Features

Some features were implemented in order to improve graph visualization (`visualizer.py`) and results (`results.py`).

Visualization

This class is used to provide a visualization method named `show()` which takes in a Graph or a list of graphs and plots them in a new window. The vertices are represented by a white circle with the name of the vertice but for easier distinction, the dominating set vertices are represented at a different color.

Results Extraction

In order to improve results extraction and to facilitate the comparison between multiple algorithms, 2 defined functions compare both the Custom and the Networkx implementation of the mininum dominating set:

1. `show_both_algorithms()` : given the inputs (*number of vertices*, *medium number of edges* and *seed*), returns the plots of both the implementations.
2. `compare_both_algorithms_with_table()` : given a list of inputs (list of the input of the previous function), returns a table with statistics.

Results and discussion

During the whole formal and empirical analysis the **basic operation** considered in these analysis will be the **retrieval of a vertice**.

Formal Analysis

Exhaustive Search

In order to compute this problem in an exhaustive way, we need to run all possible combinations, check if that specific combination is a dominating set and then check if the new dominating set is smaller than the last one.
The pseudo-code would be

```
for current_combination in all_combinations:
    if is_domination_set(graph, current_combination) and len(current_combination) < len(
        min_domination_set = current_combination
```

Since we need to check every combination, the complexity of this algorithm will be

$$O(2^n)$$

However, if we count the `is_domination_set()` method, because it is of complexity $O(n * medium\_number\_of\_edges)$, we end up with a larger complexity $O(2^n \times n \times medium\_number\_of\_edges)$ but the smaller order terms are not taken into account, for that reason, from now on these will not appear again.

Greedy Heuristics

The greedy approache consists in 3 main steps:

1. starting the min dominating set with the vertices that contain 0 edges and if only contains 1 edge, then add the vertice which is connected
2. sort vertices from the number of edges it contains and start adding them to the dominating set until finds a dominating set
3. remove excess vertices from the dominating set by sorting them in decreasing order of number of edges and trying to check if the dominating set remains a dominating set without that vertice

The preprocessing complexity is $O(n)$ and can be seen as

```
for vertice in graph.vertices:
    vertice_list = graph.edges.get(vertice, list())
    if len(vertice_list) == 0:
        domination_set.add(vertice)
    elif len(vertice_list) == 1:
        domination_set.add(vertice_list[0])
```

We can find the dominating set by the following block of pseudo-code with $O(n)$ complexity

```
covered = set()
for vertice in cardinality_invert_sorted_vertices:
    if vertice in covered: continue

    covered.update(graph.edges[vertice] + vertice)

    domination_set.add(vertice)

    if is_domination_set(graph, domination_set):
        break
```

And finally remove redundant nodes with also $O(n)$ complexity

```
for vertice in cardinality_sorted_domination_vertices:
    domination_set.remove(vertice)
    if not is_domination_set(graph, domination_set):
        domination_set.add(vertice)
```

Therefore, the final complexity of this algorithm is $O(3n)$ which simplifies to

$$O(n)$$

Experimental Results

All results are calculated with `seed = 100` .

Exhaustive Search

The exhaustive algorithm does not allow us to test the algorithm much further than the number of vertices being more than 22. This is explained by the formal analysis of the exhaustive search. The table with the inputs and outputs can be seen bellow

Vertices	Edges	Custom Exhaustive	Networkx
2	1	2 vertices	2 vertices
		0.02 ms	0.03 ms
4 + 100.0%	5 + 400.0%	2 vertices	3 vertices
		0.05 ms + 149.5%	0.01 ms - 46.4%
8 + 100.0%	12 + 140.0%	4 vertices	5 vertices
		0.71 ms + 1301.2%	0.02 ms + 54.8%
16 + 100.0%	22 + 83.3%	9 vertices	9 vertices
		226.8 ms + 32004.3%	0.03 ms + 34.1%
20 + 25.0%	34 + 54.5%	9 vertices	12 vertices
		5024.77 ms + 2115.5%	0.03 ms + 15.1%
22 + 10.0%	39 + 14.7%	11 vertices	14 vertices
		21665.12 ms + 331.2%	0.04 ms + 7.4%

We can see as expected that the number of vertices returned by the exhaustive search is better than the Networkx version, however the time needed to calculate each graph is much larger for each iteraction.

Greedy Heuristics

(adicionar a contagem com os vertices q foram lidos)

The greedy algorithm has a much better ratio between being computationally fast and having a good solution, remaining always better than the Networkx however, being slower as seen bellow

Vertices	Edges	Custom Greedy	Networkx
125	584	38 vertices	49 vertices
		1.3 ms	0.11 ms
250 + 100.0%	1247 + 113.5%	74 vertices	95 vertices
		5.08 ms + 292.0%	0.17 ms + 58.5%

Vertices	Edges	Custom Greedy	Networkx
500 + 100.0%	2551 + 104.6%	139 vertices	188 vertices
		21.48 ms + 322.6%	0.35 ms + 98.3%
1000 + 100.0%	5126 + 100.9%	290 vertices	353 vertices
		95.76 ms + 345.8%	0.64 ms + 84.1%
2000 + 100.0%	10129 + 97.6%	570 vertices	744 vertices
		356.95 ms + 272.8%	1.32 ms + 108.1%
4000 + 100.0%	20045 + 97.9%	1147 vertices	1482 vertices
		1420.4 ms + 297.9%	2.88 ms + 117.3%
8000 + 100.0%	40127 + 100.2%	2319 vertices	2958 vertices
		6107.14 ms + 330.0%	5.84 ms + 102.8%

When not applying the pós-processing we get better results, however, the ratio between being good and having better results is worse.

Vertices	Edges	Custom	Networkx
125	584	45 vertices	49 vertices
		0.54 ms	0.11 ms
250 + 100.0%	1247 + 113.5%	84 vertices	93 vertices
		2.34 ms + 329.4%	0.17 ms + 52.9%
500 + 100.0%	2551 + 104.6%	167 vertices	187 vertices
		7.63 ms + 226.2%	0.35 ms + 106.9%
1000 + 100.0%	5126 + 100.9%	328 vertices	353 vertices
		33.0 ms + 332.5%	0.64 ms + 81.8%
2000 + 100.0%	10129 + 97.6%	647 vertices	739 vertices
		129.05 ms + 291.1%	1.33 ms + 108.3%
4000 + 100.0%	20045 + 97.9%	1303 vertices	1483 vertices

Vertices	Edges	Custom	Networkx
		515.49 ms + 299.5%	2.72 ms + 104.2%
8000 + 100.0%	40127 + 100.2%	2671 vertices	2949 vertices
		2213.42 ms + 329.4%	5.69 ms + 109.4%

Conclusion

In conclusion