1º Assignment ---- Algorithm Development Strategies

Authors: Isaac dos Anjos

***Resumo* – Neste trabalho trata se em desenvolver uma solução que encontra o maior subgrafo completo de um grafo G. A solução foi optomizado e comparado com a versão original utilizar métodos estatísticas.**

***Abstract* – In this report, the goal is to develop a algorithm which extracts the a clique from a given graph G. The solution was then optimized and compared with the original using statistical methods.**

I. Introduction

Within the Advanced Algorithms Curricular Unit, a proposal was made to students to choose an algorithm for them to analyze, implement, perform a set of tests on it and briefly summarize them.

The problem chosen for this assignment was to determine the largest clique of a given graph using exhaustive search.

......

II. Exhaustive search

Exhaustive search [1], or bruteforce search is a problem-solving technique that consists of systematically enumerating all possible solutions which satisfies the problem’s statement.

III. Clique

A Clique [2], mathematical term, is a subset of vertices of an undirected graph such that every vertices is an adjacent of all vertices except it self, in other words, a complete subgraph.

Fig. 1 - Unidirectional graph, green area indicates a ‘clique’ k=3

In Figure 1 an example of an unidirectional graph, 4 vertices and 4 edges. The goal here is to identify the maximum clique within the domain, in this case, its visual possible to identify that there is a clique of k=3. Shown in green.

The maximum clique of a graph is denoted as $\omega(G)$[2], where the number of vertices corresponds to maximum number of vertices of the maximum clique in G.

IV. Clique Calculation using Exhaustive Search

The entire process of calculating cliques are done through the following algorithm:

Algorithm 1 Find the largest clique in G with brute force

```
Require: G = (V, E) an undirected graph
N <- |V|
while N >= 1 do
        solutions <- { }
        for c ϵ Combinations(V, N) do
                if isClique(c) then
                        solutions <- solutions + c
                end if
        end for
        N <- N – 1
        If solutions.length() > 0 then
                return solutions
        end if
end while
```

Fig. 2 - Pseudo code to calculate the maximum cliques of G

Algorithm 4 Check if it's a clique

Require:

```
G = {V, E}  // Graph
Seq = {W}, {W} ϵ {V}
for i ϵ 0,...,|V| do
        if i ϵ/ Seq then
                continue
        end if
        for j ϵ 0,...,|V| do
                if j ϵ/ Seq then
                        continue
                end if
                if E(Vi, Vj) ϵ/ G then
                        return False
                end if
        end for
end for
return True
```

Fig. 3 - Pseudo code to verify if a given combination is a clique of G

Consider a graph G with V vertices and E edges, the algorithm begins by considering all nodes of G. If G is a complete graph k=N, then it itself is the maximum clique.

After initializing the set of solutions, the algorithm begins to generate combinations of the N nodes. For each combination, is followed by checking if it's a clique, in Figure 3.

If no solution was found within the array of combinations, the algorithm reduces N by 1, and retart the process all over again.

V. Code Complexity

The algorithm in Figure 2 generates hand load of combinations, followed by a large amount of comparisons in the algorithm in Figure 3.

The code complexity generally uses the big-O notation, exposing the worse case or the worse set of iterations the program will preform.

In this project the complexity of this program is $O(2^{n+3} - 1)$

VI. Libraries Used

A. random

the random library [3] implements pseudo-random number generators for various distributions. Most module functions rely on the basic function *random()*, which returns a random float in a semi-open range [0.0,1.0).

This module was used to generate graphs for testing.

B. argparse

The argparse library [4] brings a user-friendly handler for command-line interfaces. The application defines arguments which it requires and argparse will parse those out of sys.argv. The argparse module also generates help and usage messages and throws when given invalid arguments.

This module was used to dynamically interact with the script.

C. matplotlib

The matplotlib library [5] is a python 2d plotting which generates high quality figures in various formats and interactive environments across platforms.

This module was used for to properly compare statistic data.

D. os

The os library [6] is a portable way in using the operating system dependent functionality.

This module was used to run other scripts.

E. itertools

The itertools library [7] implements various interator building blocks, a set of fast, memory efficient tools.

This module was used to create combinations of nodes.

F. time

The time library [8] provides time-related functions.

This module was used to extract execute time and limit cpu processing.

VII. Important Functions

. Graph Generator

```python
def generateGraph(N,p=0.5):
    output = [["" for i in range(N)] for j in range(N)]
    for i in range(0,N):
        for j in range(0,N):
            if i != j:
                if output[j][i] == "":
                    output[i][j]=1 if random.random() > p else 0
                else:
                    output[i][j] = output[j][i]
    return output
```

Fig. 4 - Graph generator function

The function generateGraph creates a graph of a given size N of nodes. Edges are later generate with a probability of p, where 1 returns a complete graph and 0 return N isolated nodes.

```
['', 1, 0, 1, 0]
[1, '', 0, 1, 0]
[0, 0, '', 1, 0]
[1, 1, 1, '', 1]
[0, 0, 0, 1, '']
```

Fig. 7 - Graph in a matrix

VIII. Analysis Results

First Version

In this chapter, shows a presentation of the results obtained, smalls changes made to the algorithms and side by side comparisons of the changes on all significant levels.

The results were obtained by running the program 10 times and calculating the average to avoid bias results. The results include, the behaviour within the probability domain, executions time of each N nodes and the number of operations made by the algorithm.

In Figure 5, with a probability of 0, shows the total time average of each N ϵ {1 … 20}. Concluding that time increases exponentially along the N axis. The reason behind using a probability 0 is due to being the worse case for the algorithm. since it’s goal is to find the largest set of nodes which forms a clique, beginning from the largest set is recommended.

In Figure 6, proves that the code complexity stated above is $O(2^{n+3} - 1)$.   .

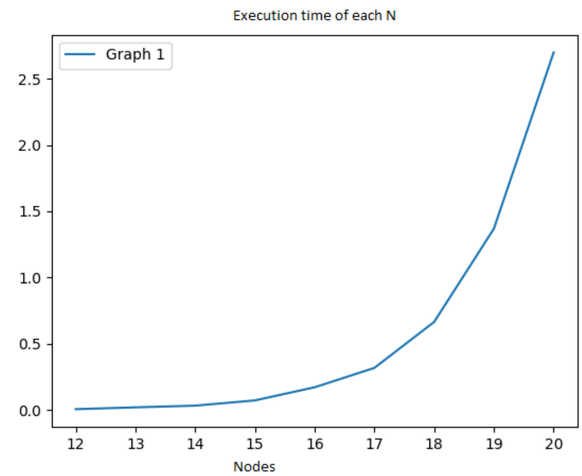

Fig. 5 - Execution time average of each number of nodes

Edge probability: 0%

N	Solution Length	Num. Operations	Time AVG (s)
12	12	40540	0.0063
13	13	81433	0.0158
14	14	163281	0.0306
15	15	327044	0.0667
16	16	654642	0.1385
17	17	1309915	0.2693
18	18	2620543	0.5274
19	19	5241886	1.1224
20	20	10484664	2.1787

Fig. 6 - Execution time average and number of operations of each number of nodes

Second Version

After reviewing the algorithm, there were some unnecessary iterations being made when solving the randomly generated graph.

In this project, graphs were digitally represented by a matrix, where each cell represents an edge of two nodes. This meant that both the superior and inferior triangle of the matrix were the same.

Fig. 8 - Indicating both triangles of the matrix

To properly interate the matrix, the first index should iterate all nodes except the last one, i ϵ {0 … |V|-1}. The second index should only iterate the nodes after i, j ϵ {i+1 … |V|}.

Algorithm 4 Check if it's a clique

Require:

G = {V, E} // Graph

Seq = {W}, {W} ϵ {V}

for i ϵ 0,...,|V|-1 **do**

  if i ϵ/ Seq **then**

    continue

  end if

  for j ϵ i+1,...,|V| **do**

    if j ϵ/ Seq **then**

      continue

    end if

    if E(Vi, Vj) ϵ/ G **then**

      return False

Fig. 9 - Algorithm Clique iterating the superior triangle of the matrix

The matrix in theory should like this:

```
['', 0, 0, 0, 1]
['', '', 0, 0, 1]
['', '', '', 1, 1]
['', '', '', '', 1]
['', '', '', '', '']
```

Fig. 10 - Superior triangle of matrix

Edge probability: 0%

N	Solution Length	Num. Operations	Time AVG (s)
12	12	20310	0.0063
13	13	40763	0.0158
14	14	81694	0.0306
15	15	163583	0.0667
16	16	327390	0.1385
17	17	655035	0.2693
18	18	1310358	0.5274
19	19	2621039	1.1224
20	20	5242438	2.1787

Fig. 11 - Execution time average and number of operations of each number of nodes

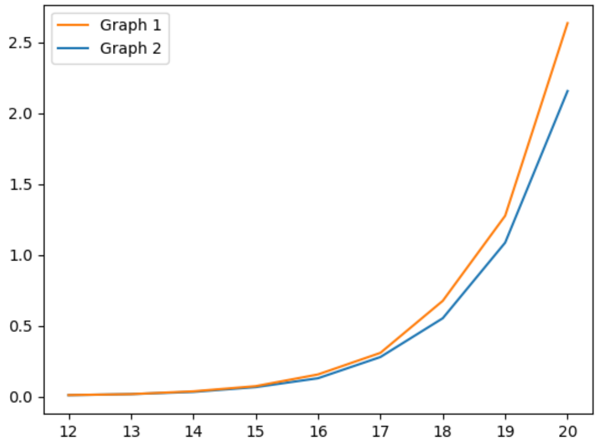

Fig. 12 - Execution time average of each number of nodes

The overall performance of the new version increased by roughly 20%, and the number of iterations per N was cut down in half.

Third Version

After analysing carefully both versions of the algorithm in the probability domain, an idea appeared. The algorithm was designed to solve randomly generated graphs, where the probability of 0 edges is considered extremely rare.

The maximum number of edges a graph can persue is (|N|*(|N|-1)) / 2, and with a probability of connection is 0.5 so on average there should be (|N|*(|N|-1)) / 4 edges.

This being, lets assume that the node with the most edges is likely to appear in the solution. Instead of iterating all of the combinations, only iterate combinations where the with the most connections/edges appears.

```
def getBestNode(self):
    bestNode = []
    best = 0
    temp = 0
    for line in range(0,len(self.matrix)):
        temp = 0
        if best > (len(self.matrix)-(line+1)):
            break
        for cell in self.matrix[line]:
            if cell == 1:
                temp += 1
        if temp > best:
            best = temp
            bestNode = [line]
        elif temp == best:
            bestNode += [line]
    #print("Best nodes")
    #print(bestNode)
    return bestNode
```

Fig. 13 – Algorithm to find the best nodes

In the algorithm above, returns set of nodes with most the connections(since there might be nodes with same number of connections).

```python
def getCombinations(self,n):
    if n>= self.n:
        return self.combinations
    temp = list(itertools.combinations(range(0,self.n),n))
    #print(output)
    output = []
    for tup in temp:
        for node in self.bestNodes:
            if node in tup:
                output += [tup]
                break
    return output
```

Fig. 14 – Filtering non favourable combinations with the best nodes

The best nodes are then used to filter out combinations that do not have these nodes.
The only problem here is when the node with the most connections doesn't belong to the maximum clique and other solutions might not be viewed after filtering.

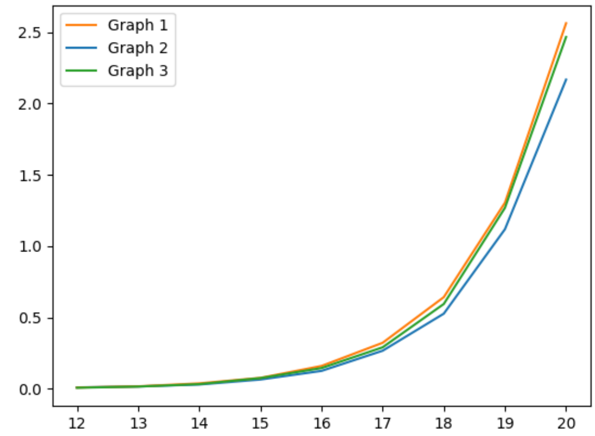

Fig. 15 – All three versions in their worse case scenario

After testing all three versions in the worse case scenario, its obvious how adding extra loops extends the execution time of the third version.

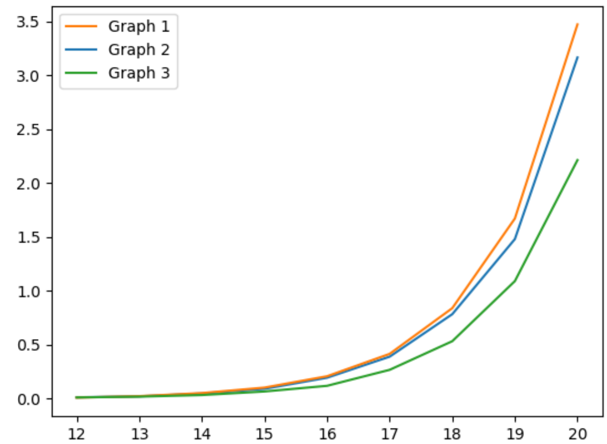

Fig. 16 – Results with a probability of 50%

When testing them with a probability > 0, the results of the third solution shows how its possible to rapidly find the solution while sacrificing the 100% certainty.

Edge probability 50%

Graph 1

N	Solution Length	Num. Operations	Time AVG (s)
12	1	31298	0.009854316711425781
13	1	33212	0.020059823989868164
14	1	114598	0.03973197937011719
15	1	232695	0.06994485855102539
16	1	510332	0.19034218788146973
17	1	1210543	0.36466145515441895
18	1	2213682	1.5115206241607666
19	1	2915577	2.0401113033294678
20	1	6712771	2.8351569175720215

Graph 2

N	Solution Length	Num. Operations	Time AVG (s)
12	1	50950	0.010106801986694336
13	1	59974	0.010382890701293945
14	1	180522	0.0302581787109375
15	1	407295	0.08000683784484863
16	1	823499	0.22005271911621094
17	1	2028216	0.5349476337432861
18	1	3818898	1.8199570178985596
19	1	5527526	2.4898452758789062
20	1	12296939	3.125030755996704

Graph 3

N	Solution Length	Num. Operations	Time AVG (s)
12	1	25378	0.0
13	1	18865	0.009614944458007812
14	1	89560	0.029818058013916016
15	1	116016	0.059607267379760074
16	1	416062	0.14971303939819336
17	1	955055	0.6502954959869385
18	1	1141227	1.2902064323425293
19	1	1225131	0.7698450088500977
20	1	3201463	1.5699751377105713

Fig. 17 - Execution time average and number of operations of each number of nodes with a probability of 50%

IX. Conclusion

Exhaustive search, in my opinion, is such a great starting point to solve these types of problem, but not as an ending point. Optimizations and elaborated strategies tend to perform much better.

References

Use the Style "referencia" to the references. Example:

[1] https://en.wikipedia.org/wiki/Brute-force_search

[2] https://en.wikipedia.org/wiki/Clique_(graph_theory)

[3] https://docs.python.org/3/library/random.html

[4] https://docs.python.org/3/library/argparse.html

[5] https://matplotlib.org/

[6] https://docs.python.org/3/library/os.html

[7] https://docs.python.org/2/library/itertools.html

[8] https://docs.python.org/3/library/time.html