2º Assignment ---- Algorithm Development Strategies

Author: Isaac dos Anjos

***Resumo* – Neste trabalho trata se em desenvolver uma solução que encontra o maior subgrafo completo de um grafo G. Uma abordagem probabilística foi implementada 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. A probabilistic approach was implemented and compared with previous versions 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 and briefly summarize their work.

The problem chosen for this assignment was to determine the largest clique of a given graph using probabilistic 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. Monte Carlo Algorithm

Monte Carlo Algorithm [9] is a problem-solving technique that consists of randomly selecting k elements from an array for evaluation. The algorithm stops if it finds a solution or the k elements are all invalid answers.

IV. Clique

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

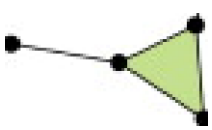

Fig. 1 - Unidirectional graph, green area indicates a 'clique' k=3

In Figure 1, illustrates an example of an unidirectional graph, 4 vertices and 4 edges. The goal here is to identify the maximum clique within the graph, in this case the solution is 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 Monte Carlo Algorithm

The process of calculating cliques is done through the following algorithm:

```
Algorithm 2 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 ∈ RandomCombinations(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 find the largest clique in G

```
Algorithm 5 Get Random Combinations
Require:
G = {V, E}
K = 100 // number of combinations
N = |V| // Number of nodes
output = [] // Output array

while |output| < K do
        if |output| >= 2^N then
                break
        end if
        x <- randomNumber(1,((2^N)-1))
        if x ϵ/ output then
                output <- output + x
        end if
end while
return SortReverse(output)
```

Fig. 4 - Pseudo code to get N random combinations of nodes

```
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 generating K random combinations of G.

After initializing the set of solutions, the algorithm begins to verify each combination.

If no solution was found within the given array of combinations, the algorithm returns 0 solutions.

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 $O(2^{n+3} - 1)$

VII. Important Functions

. *Graph Generator*

Algorithm 6 Graph Generator

Require:

N <- |Nodes|

P <- Probability of Edge

output = [] // Output array

for i ϵ 0,...,N **do**

    for j ϵ 0,...,N **do**

        output[i][j] <- None

    endfor

endfor

for i ϵ 0,...,N **do**

    for j ϵ 0,...,N **do**

        if i != j **and** output[i][j] == "" **then**

            if randomValue() > P **then**

                output[i][j] <- 0

            else

                output[i][j] <- 0

            endif

        endif

    endfor

endfor

return output

Fig. 5 - Pseudo code to generate a graph

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.

VIII. Analysis Results

In the previous report [10], there were 3 versions of the exhaustive search algorithm, solving the largest clique of a subgraph. Adjustments were made, showing the first solution encountered, since the combination array is sorted by size, the program will always return the best solution of the array.

Graph 1 Exhaustive Search V1

N	Best solution	Num. Operations	Time AVG (s)
12	(0, 5, 10, 11)	39812	0.0
13	(3, 6, 9, 10)	92644	0.015619516372680664
14	(0, 1, 4, 5, 6)	233954	0.04686427116394043
15	(0, 3, 5, 9)	350553	0.09372687339782715
16	(0, 1, 4, 7, 9)	910866	0.21866202354431152
17	(1, 8, 9, 13, 16)	1388785	0.34366869926452637
18	(0, 3, 4, 5)	3219696	0.7498579025268555
19	(0, 3, 8, 9, 10)	6723004	1.6402082443237305
20	(3, 8, 11, 12, 17)	13312609	3.3742053508758545

Graph 2 Exhaustive Search V2

N	Best solution	Num. Operations	Time AVG (s)
12	(0, 5, 10, 11)	22574	0.015625953674316406
13	(3, 6, 9, 10)	51907	0.01558542251586914
14	(0, 1, 4, 5, 6)	142546	0.062485456466674805
15	(0, 3, 5, 9)	190497	0.09372663497924805
16	(0, 1, 4, 7, 9)	553256	0.1874547004699707
17	(1, 8, 9, 13, 16)	751056	0.3280484676361084
18	(0, 3, 4, 5)	1779522	0.6873044967651367
19	(0, 3, 8, 9, 10)	3802933	1.4059183597564697
20	(3, 8, 11, 12, 17)	7423680	3.4835216999053955

Graph 3 Using Best Node

N	Best solution	Num. Operations	Time AVG (s)
12	(1, 4, 6, 10)	11916	0.0
13	(0, 2, 8)	37204	0.01565837860107422
14	(0, 1, 4, 5, 6)	97824	0.031243562698364258
15	(0, 3, 5, 9)	127417	0.03124213218688965
16	(0, 1, 4, 7, 9)	340000	0.12500524520874023
17	(4, 7, 9, 11, 16)	353444	0.15621519088745117
18	(0, 3, 4, 5)	725142	0.35929203033447266
19	(0, 3, 8, 9, 10)	1354197	0.734238862991333
20	(3, 8, 11, 12, 17)	5330912	2.4681692123413086

Fig. 6 - Results of previous versions

Graph 4 Monte Carlo Algorithm k=1000

N	Best solution	Num. Operations	Time AVG (s)
12	(4, 6, 9, 10)	5626	0.01558828353881836
13	(2, 3, 9)	6759	0.01558542251586914
14	(1, 4, 5, 6)	11293	0.015619277954101562
15	(7, 9, 11)	6499	0.03124237060546875
16	(3, 6, 9, 15)	9771	0.031209230422973633
17	(4, 12, 14)	6208	0.015619993209838867
18	()	6865	0.015620231628417969
19	()	7592	0.015613794326782227
20	()	7392	0.015650033950805664

Fig. 7 - Monte Carlo results, k=1000

The solutions encountered using the Monte Carlo algorithm, k = 1000, weren't the best though the processing time was a huge upgrade compared to the exhaustive search results.

```
Graph 1 Exhaustive Search V1
N       Best solution           Num. Operations         Time AVG (s)
12      (0, 1, 4, 6)            44122                   0.0
13      (0, 1, 4, 6, 11, 12)    54591                   0.0
14      (1, 2, 3, 6, 13)        188629                  0.031242132186889 65
15      (1, 5, 7, 12, 13)       461274                  0.124965190887451 17
16      (2, 3, 4, 9, 15)        866509                  0.203076601028442 38
17      (1, 4, 6, 11, 13, 15)   1138695                 0.296808242797851 56
18      (2, 9, 13, 16, 17)      3158261                 0.874794721603393 6
19      (0, 1, 2, 7, 18)        8455389                 1.983907461166381 8
20      (3, 6, 10, 13, 17)      13084804                3.124262571334839
Graph 2 Exhaustive Search V2
N       Best solution           Num. Operations         Time AVG (s)
12      (0, 1, 4, 6)            26417                   0.015621423721313477
13      (0, 1, 4, 6, 11, 12)    32064                   0.015620470046997 07
14      (1, 2, 3, 6, 13)        112836                  0.031243562698364258
15      (1, 5, 7, 12, 13)       281547                  0.093729257583618 16
16      (2, 3, 4, 9, 15)        510277                  0.18745613098144 53
17      (1, 4, 6, 11, 13, 15)   602258                  0.281182289123535 16
18      (2, 9, 13, 16, 17)      1749661                 0.687340736389160 2
19      (0, 1, 2, 7, 18)        5129924                 1.858974218368530 3
20      (3, 6, 10, 13, 17)      7261899                 3.046188831329345 7
Graph 3 Using Best Node
N       Best solution           Num. Operations         Time AVG (s)
12      (0, 1, 4, 6)            15592                   0.015620946884155273
13      (0, 1, 4, 6, 11, 12)    23451                   0.015650033950805664
14      (1, 2, 3, 6, 13)        57333                   0.046900749206542 97
15      (0, 1, 8, 13)           186950                  0.078135858985900879
16      (2, 3, 4, 9, 15)        406441                  0.156248331069946 3
17      (1, 4, 6, 11, 13, 15)   286815                  0.140592575073242 2
18      (0, 3, 14, 16)          1212501                 0.562364101409912 1
19      (0, 1, 2, 7, 18)        2975393                 1.468368053436279 3
20      (3, 6, 10, 13, 17)      3589553                 2.046361684799194 3
Graph 4 Monte Carlo Algorithm k=10000
N       Best solution           Num. Operations         Time AVG (s)
12      (0, 1, 4, 6)            26417                   0.015621185302734375
13      (0, 1, 4, 6, 11, 12)    32064                   0.0
14      (1, 2, 3, 6, 13)        69956                   0.156178236007690 43
15      (5, 6, 7, 12, 13)       85113                   0.187421321868896 48
16      (4, 9, 10, 15)          82470                   0.203039646148681 64
17      (4, 6, 11, 13, 15)      52676                   0.234317779541015 62
18      (2, 9, 14, 16, 17)      64217                   0.187455654144287 1
19      (17, 18)                102474                  0.218733787536621 1
20      (1, 3, 8, 17)           70567                   0.2811853885650635
```

Fig. 7 – Results of 4 versions, k=10000

After adjusting k to 10000, the algorithm can properly find solutions for graphs over 18 nodes. The average time also increased proportionally with k.

IX. Conclusion

Probabilistic search, statistically, solves problems much faster than exhaustive search but in contrast, returns low quality or no results. This algorithm is useful in scenarios where there are a lot of factors involved and the best solution is as great as any solution.

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.

REFERENCES

[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
[9] https://elearning.ua.pt/pluginfile.php/913972/mod_resource/content/0/AA_08_Intro_Randomized_Algorithms_III.pdf
[10] ./previousReport.pdf