EC 704 - VLSI Design Automation Assignment 5

-Shruti Masand, 181EC245

Problem Statement:

Using the netlist, generate the adjacency matrix and do the partitioning using Simulated Annealing.

Solution:

Approach:

STEP 1 : Firstly, we need to convert the given netlist into a graph. To do that, I have tried to read the Netlist File line by line (using the readline() function) and tried to interpret the meaning of each line. As we have to read the gates, we take all the equations that start with "G".
Then, we can simply read LHS and RHS separately. The LHS part of that particular line will tell us about the output and the RHS part will tell us about the inputs. Hence, we create 2 lists by the name of LHS and RHS that will be used to store the values of the outputs and the inputs respectively.

These (input, output) pairs will then be appended to a list that is given as input to the Networkx library functions.The commands written using this library called Networkx, very conveniently plot these functions into graphs with gates as nodes and directed connections shown as Edges.
For an even enriching visual aid, we can change the size and color of these edges and vertices as well. The root code for the given library ws thoroughly understood and is also given as one of the links in the references.

STEP 2 : Next step is to use that graph to form the adjacency list/matrix. The adjacency Matrix gives us the information about the nodes that are in the neighbourhood of a particular node. This can be accomplished by the use of the networkx and the numpy library. This is accomplished using a single command that is shown in the code.

STEP 3 : Now we need to apply the Simulated Annealing Algorithm for performing efficient partitioning. This can be done by coding in accordance with the algorithms discussed in the class.

Code, implemented in Python:

```
import networkx as nx
import matplotlib.pyplot as plt
from math import *
import numpy as np
from numpy import asarray
from numpy import exp
from numpy.random import randn
from numpy.random import rand
from numpy.random import seed

def read_gates(l,index):
        t = ''
        t = t + l[index]
        j = index + 1
        while ((l[j] != ' ') and (l[j] != ',') and (l[j] != ')')):
        t = t + l[j]
        j = j +1
        return(t)

# objective function
def objective(x):
        return x[0]**2.0

# simulated annealing algorithm
def simulated_annealing(objective, bounds, n_iterations, step_size, temp):

        best = bounds[:, 0] + rand(len(bounds)) * (bounds[:, 1] - bounds[:, 0])

        best_eval = objective(best)
        # current working solution
        curr, curr_eval = best, best_eval
        # run the algorithm
        for i in range(n_iterations):
        # take a step
        candidate = curr + randn(len(bounds)) * step_size
        # evaluate candidate point
        candidate_eval = objective(candidate)
        # check for new best solution
        if candidate_eval < best_eval:
        # store new best point
        best, best_eval = candidate, candidate_eval
        # report progress
```

```
    print('>%d f(%s) = %.5f' % (i, best, best_eval))
    # difference between candidate and current point evaluation
    diff = candidate_eval - curr_eval
    # calculate temperature for current epoch
    t = temp / float(i + 1)
    # calculate metropolis acceptance criterion
    metropolis = exp(-diff / t)
    # check if we should keep the new point
    if diff < 0 or rand() < metropolis:
    # store the new current point
    curr, curr_eval = candidate, candidate_eval
    return [best, best_eval]



def load_data(filename):

    lhs = []
    rhs = []
    exray = []
    vega = []

    abc = 0

    dict = {}

    file1 = open(filename, 'r')
    Lines = file1.readlines()

    for line in Lines:
    if(len(line)>0):
    if(line[0] == 'G'):
        gate_lhs = read_gates(line, 0)
        if(dict.get(gate_lhs)):
        gate_lhs_index = dict[gate_lhs]

        else:
        gate_lhs_index = abc
        dict[gate_lhs] = abc
        abc = abc+1


        for i in range(1, len(line)):
        if (line[i] == "G"):
        gate_rhs = read_gates(line,i)
        if(dict.get(gate_rhs)):
            gate_rhs_index = dict[gate_rhs]

        else:
            gate_rhs_index = abc
            dict[gate_rhs] = abc
            abc = abc+1

        lhs.append((gate_rhs_index,gate_lhs_index))

    #To get a list of all Vertices:
        exray.append(gate_rhs)
        exray.append(gate_lhs)

    lhs.sort()

    G = nx.DiGraph()

    G.add_edges_from( lhs )

    pos = nx.spring_layout(G)

    nx.draw_networkx_nodes(G, pos, node_size = 500)
    nx.draw_networkx_edges(G, pos, edgelist = G.edges(), edge_color = 'black')
    nx.draw_networkx_labels(G, pos)
    plt.show()

    return lhs


def main():

    seed(1)
    our_data = asarray(load_data('s27bench'))

    n_iterations = 1000

    step_size = 0.1

    temp = 10

    best, score = simulated_annealing(objective, our_data, n_iterations, step_size, temp)
    print('Simulated Annealing Completed')
    print('f(%s) = %f score' % (best, score))


if __name__ == "__main__":
    main()
```

Screenshots of the results obtained:

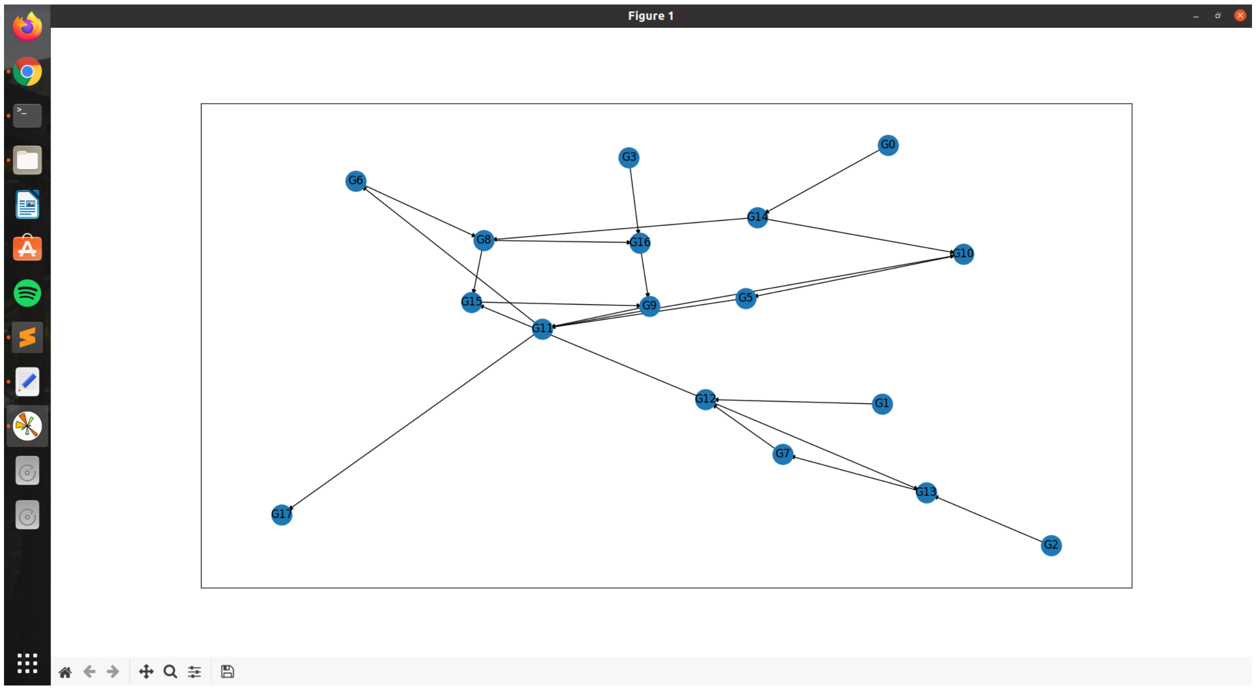
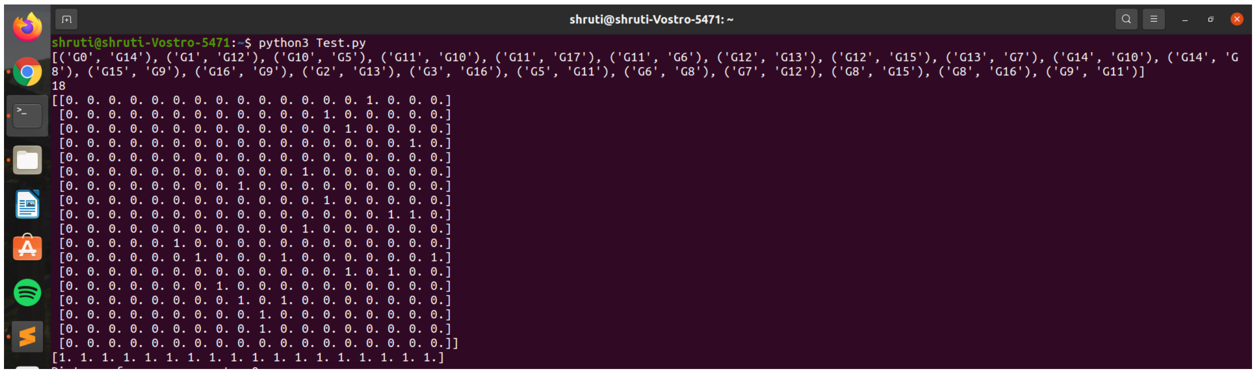


Fig1. Displays the graph obtained using the netlist file : s27.bench



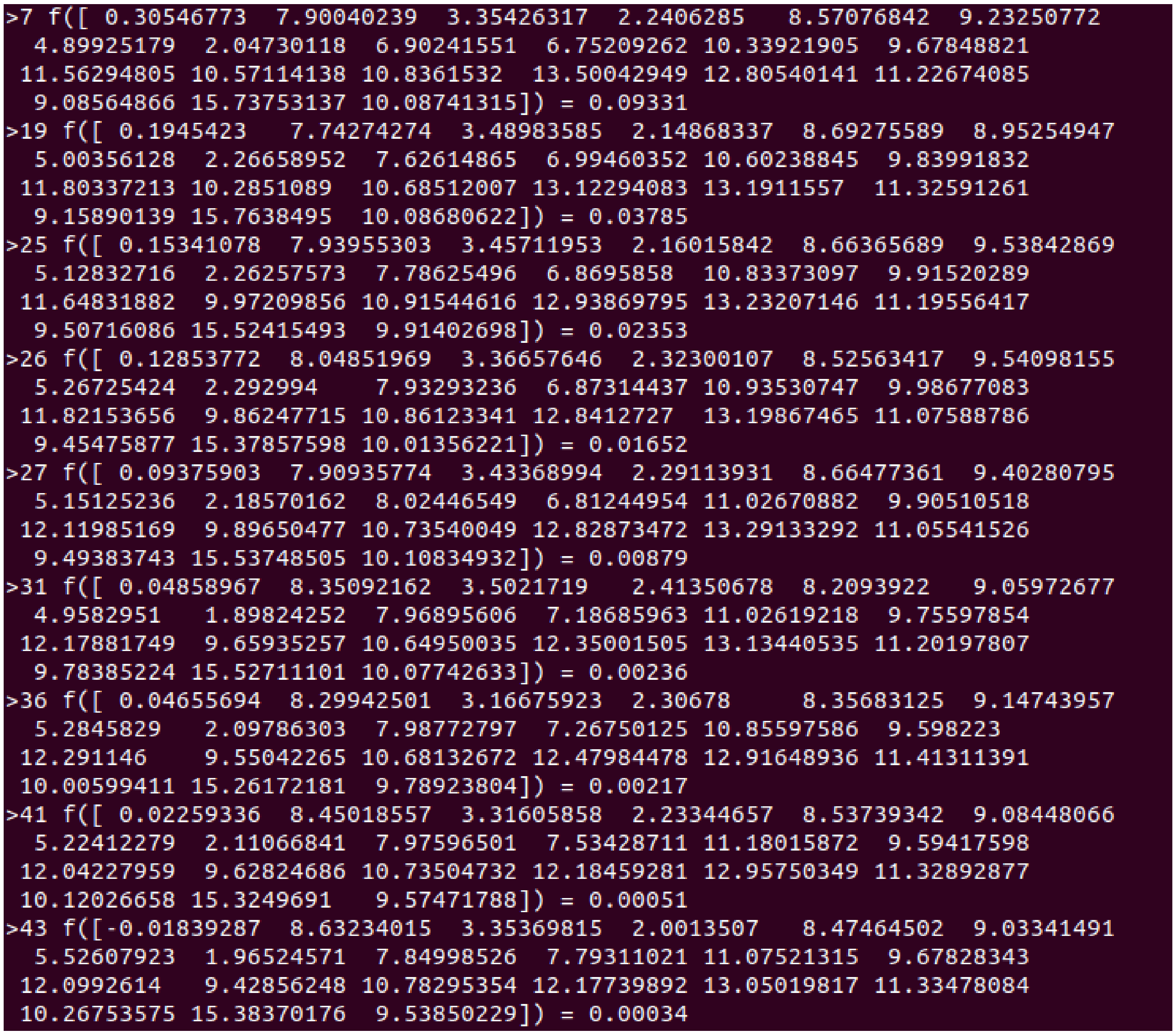
```
shruti@shruti-Vostro-5471:~$ python3 Test.py
[('G0', 'G14'), ('G1', 'G12'), ('G10', 'G5'), ('G11', 'G10'), ('G11', 'G17'), ('G11', 'G6'), ('G12', 'G13'), ('G12', 'G15'), ('G13', 'G7'), ('G14', 'G10'), ('G14', 'G
8'), ('G15', 'G9'), ('G16', 'G9'), ('G2', 'G13'), ('G3', 'G16'), ('G5', 'G11'), ('G6', 'G8'), ('G7', 'G12'), ('G8', 'G15'), ('G8', 'G16'), ('G9', 'G11')]
18
[[0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 1.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
```

Fig2. The following results display for s27.bench file:

a) A list of all the edge connections with vertices in the order (input, output)

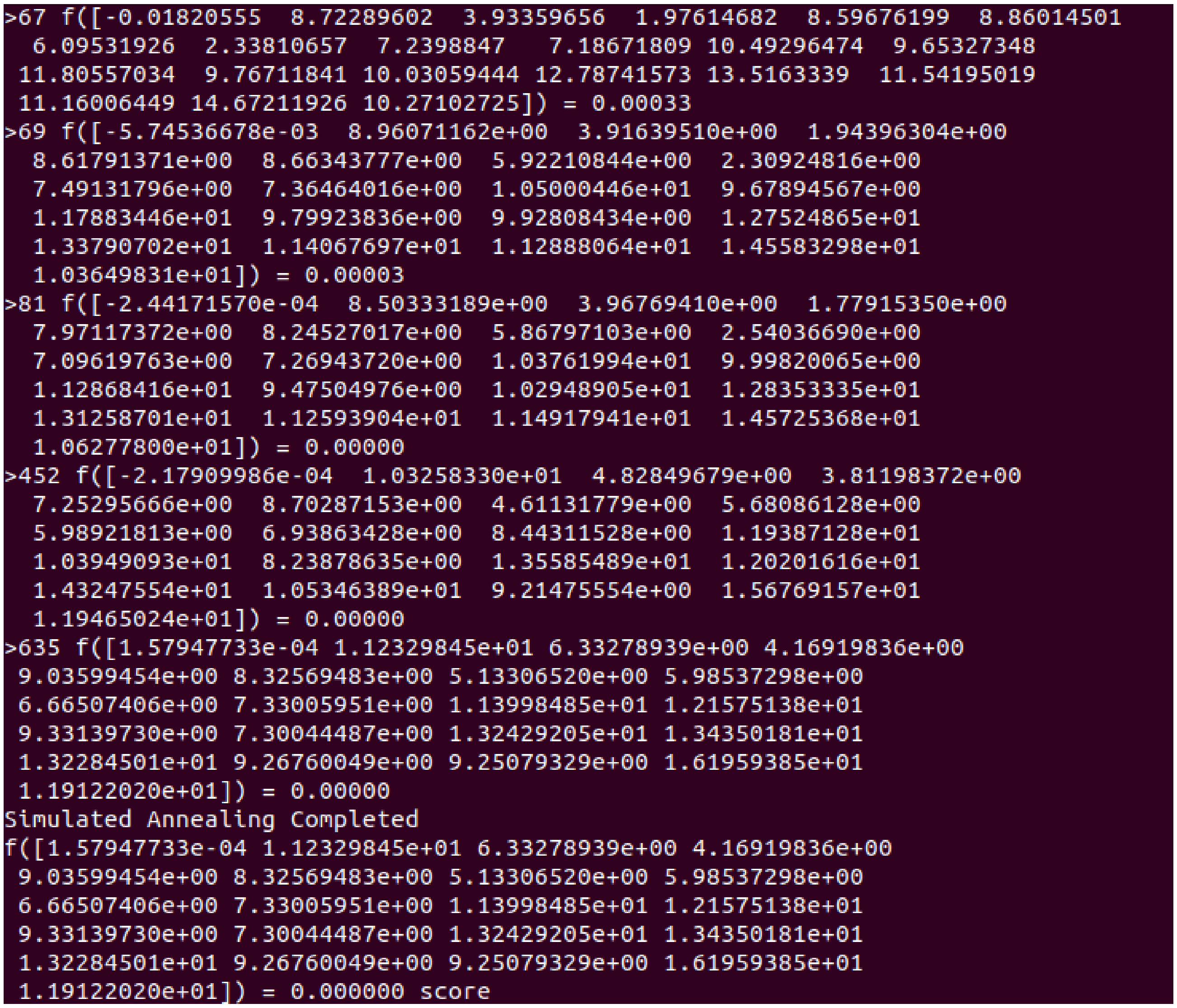
b) Adjacency Matrix for the above shown Directed Graph

c) List of weights(here all have weight = 1)



```
>7 f([ 0.30546773  7.90040239  3.35426317  2.2406285   8.57076842  9.23250772
  4.89925179  2.04730118  6.90241551  6.75209262 10.33921905  9.67848821
 11.56294805 10.57114138 10.8361532  13.50042949 12.80540141 11.22674085
  9.08564866 15.73753137 10.08741315]) = 0.09331
>19 f([ 0.1945423   7.74274274  3.48983585  2.14868337  8.69275589  8.95254947
  5.00356128  2.26658952  7.62614865  6.99460352 10.60238845  9.83991832
 11.80337213 10.2851089  10.68512007 13.12294083 13.1911557  11.32591261
  9.15890139 15.7638495  10.08680622]) = 0.03785
>25 f([ 0.15341078  7.93955303  3.45711953  2.16015842  8.66365689  9.53842869
  5.12832716  2.26257573  7.78625496  6.8695858  10.83373097  9.91520289
 11.64831882  9.97209856 10.91544616 12.93869795 13.23207146 11.19556417
  9.50716086 15.52415493  9.91402698]) = 0.02353
>26 f([ 0.12853772  8.04851969  3.36657646  2.32300107  8.52563417  9.54098155
  5.26725424  2.292994    7.93293236  6.87314437 10.93530747  9.98677083
 11.82153656  9.86247715 10.86123341 12.8412727  13.19867465 11.07588786
  9.45475877 15.37857598 10.01356221]) = 0.01652
>27 f([ 0.09375903  7.90935774  3.43368994  2.29113931  8.66477361  9.40280795
  5.15125236  2.18570162  8.02446549  6.81244954 11.02670882  9.90510518
 12.11985169  9.89650477 10.73540049 12.82873472 13.29133292 11.05541526
  9.49383743 15.53748505 10.10834932]) = 0.00879
>31 f([ 0.04858967  8.35092162  3.5021719   2.41350678  8.2093922   9.05972677
  4.9582951   1.89824252  7.96895606  7.18685963 11.02619218  9.75597854
 12.17881749  9.65935257 10.64950035 12.35001505 13.13440535 11.20197807
  9.78385224 15.52711101 10.07742633]) = 0.00236
>36 f([ 0.04655694  8.29942501  3.16675923  2.30678     8.35683125  9.14743957
  5.2845829   2.09786303  7.98772797  7.26750125 10.85597586  9.598223
 12.291146    9.55042265 10.68132672 12.47984478 12.91648936 11.41311391
 10.00599411 15.26172181  9.78923804]) = 0.00217
>41 f([ 0.02259336  8.45018557  3.31605858  2.23344657  8.53739342  9.08448066
  5.22412279  2.11066841  7.97596501  7.53428711 11.18015872  9.59417598
 12.04227959  9.62824686 10.73504732 12.18459281 12.95750349 11.32892877
 10.12026658 15.3249691   9.57471788]) = 0.00051
>43 f([-0.01839287  8.63234015  3.35369815  2.0013507   8.47464502  9.03341491
  5.52607923  1.96524571  7.84998526  7.79311021 11.07521315  9.67828343
 12.0992614   9.42856248 10.78295354 12.17739892 13.05019817 11.33478084
 10.26753575 15.38370176  9.53850229]) = 0.00034
```

Fig3. In this first half of the output, we can see 9 improvements at iterations 7,19,25,26,27,31,36,41and 43 reducing the solution from 0.09331 to 0.00034



```
>67 f([-0.01820555  8.72289602  3.93359656  1.97614682  8.59676199  8.86014501
  6.09531926  2.33810657  7.2398847   7.18671809 10.49296474  9.65327348
 11.80557034  9.76711841 10.03059444 12.78741573 13.5163339  11.54195019
 11.16006449 14.67211926 10.27102725]) = 0.00033
>69 f([-5.74536678e-03  8.96071162e+00  3.91639510e+00  1.94396304e+00
  8.61791371e+00  8.66343777e+00  5.92210844e+00  2.30924816e+00
  7.49131796e+00  7.36464016e+00  1.05000446e+01  9.67894567e+00
  1.17883446e+01  9.79923836e+00  9.92808434e+00  1.27524865e+01
  1.33790702e+01  1.14067697e+01  1.12888064e+01  1.45583298e+01
  1.03649831e+01]) = 0.00003
>81 f([-2.44171570e-04  8.50333189e+00  3.96769410e+00  1.77915350e+00
  7.97117372e+00  8.24527017e+00  5.86797103e+00  2.54036690e+00
  7.09619763e+00  7.26943720e+00  1.03761994e+01  9.99820065e+00
  1.12868416e+01  9.47504976e+00  1.02948905e+01  1.28353335e+01
  1.31258701e+01  1.12593904e+01  1.14917941e+01  1.45725368e+01
  1.06277800e+01]) = 0.00000
>452 f([-2.17909986e-04  1.03258330e+01  4.82849679e+00  3.81198372e+00
  7.25295666e+00  8.70287153e+00  4.61131779e+00  5.68086128e+00
  5.98921813e+00  6.93863428e+00  8.44311528e+00  1.19387128e+01
  1.03949093e+01  8.23878635e+00  1.35585489e+01  1.20201616e+01
  1.43247554e+01  1.05346389e+01  9.21475554e+00  1.56769157e+01
  1.19465024e+01]) = 0.00000
>635 f([1.57947733e-04 1.12329845e+01 6.33278939e+00 4.16919836e+00
 9.03599454e+00 8.32569483e+00 5.13306520e+00 5.98537298e+00
 6.66507406e+00 7.33005951e+00 1.13998485e+01 1.21575138e+01
 9.33139730e+00 7.30044487e+00 1.32429205e+01 1.34350181e+01
 1.32284501e+01 9.26760049e+00 9.25079329e+00 1.61959385e+01
 1.19122020e+01]) = 0.00000
Simulated Annealing Completed
f([1.57947733e-04 1.12329845e+01 6.33278939e+00 4.16919836e+00
 9.03599454e+00 8.32569483e+00 5.13306520e+00 5.98537298e+00
 6.66507406e+00 7.33005951e+00 1.13998485e+01 1.21575138e+01
 9.33139730e+00 7.30044487e+00 1.32429205e+01 1.34350181e+01
 1.32284501e+01 9.26760049e+00 9.25079329e+00 1.61959385e+01
 1.19122020e+01]) = 0.000000 score
```

Fig4. In the second half of the output, we can see improvement at iterations 67,69,81,452 and 635, improving the result until we get 0 as a solution.

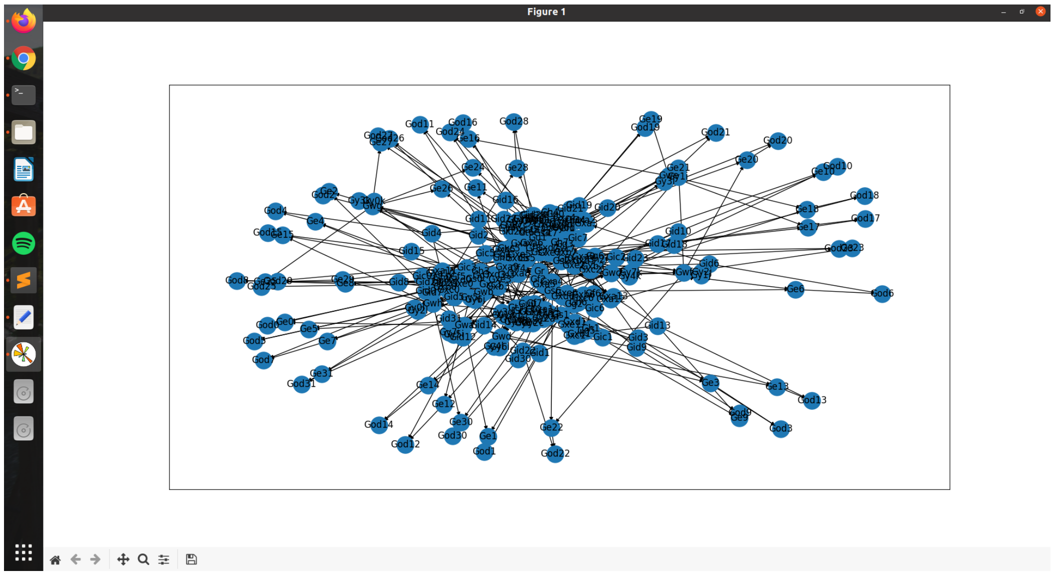


Fig5. Similarly, the graph can also be obtained for something as complex as the c499.bench file, as given in the figure