NEC - Exercise 4

Optimization with Genetic Algorithms

Carlos García

February 2017

Contents

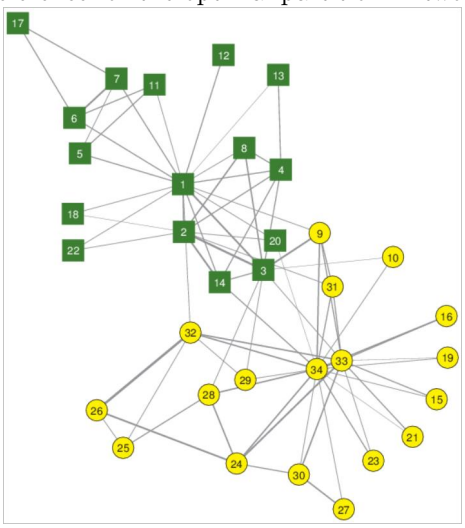
Exercise 4: Optimization with Genetic Algorithms

Objective

Implementation of a genetic algorithm (GA) for the clustering of the nodes of a graph, by means of the optimization of modularity.

- Modularity
- Genetic algorithm
- Fitness

Example This is the "real" partition of the Zachary Karate Club network, as a reference for the optimal partition in two communities.



1 Main concepts

1.1 Modularity (networks)

Modularity is one measure of the structure of networks or graphs. It was designed to measure the strength of division of a network into modules (also called groups, clusters or communities). Networks with high modularity have dense connections between the nodes within modules but sparse connections between nodes in different modules. Modularity is often used in optimization methods for detecting community structure in networks. However, it has been shown that modularity suffers a resolution limit and, therefore, it is unable to detect small communities. Biological networks, including animal brains, exhibit a high degree of modularity [1].

1.2 Genetic algorithm

In computer science and operations research, a genetic algorithm (GA) is a metaheuristic inspired by the process of natural selection that belongs to the larger class of evolutionary algorithms (EA). Genetic algorithms are commonly used to generate high-quality solutions to optimization and search problems by relying on bio-inspired operators such as mutation, crossover and selection [2].

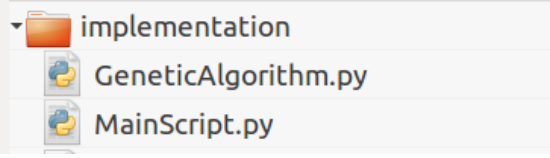
1.3 Fitness (biology)

Fitness is the quantitative representation of natural and sexual selection within evolutionary biology. It can be defined either with respect to a genotype or to a phenotype in a given environment. In either case, it describes individual reproductive success and is equal to the average contribution to the gene pool of the next generation that is made by individuals of the specified genotype or phenotype. The fitness of a genotype is manifested through its phenotype, which is also affected by the developmental environment. The fitness of a given phenotype can also be different in different selective environments [3].

2 Description of the implementation approach

As for prior exercises, I chose working with Python language because it is easy to write, read, and it has lots of documentation on the web. Besides because I'm currently trying to learn it and use it a bit more.

First I split the code in two main parts, one script with all the tools or functions related to the genetic algorithm, it is called "GeneticAlgorithm.py" and another script which calls and uses all functions into latest script, it is called "MainScript.py".



2.1 GeneticAlgorithm.py

Now we can check the "GeneticAlgorithm.py", this is the most important script because it holds all the important logic and procedures for every step in the survival process, in the following image we have an overview with the declaration of all this methods:

```
 9  # ------------------------------------------------------------------------------
10  def createAdjacenceMatrix(fileName):
38  # ------------------------------------------------------------------------------
39  def createInitialPopulation(numIndividuals, codeSize):
45  # ------------------------------------------------------------------------------
46  def computeModularity(matrix, pop, L, Ki, Kj, codeSize):
62  # ------------------------------------------------------------------------------
63  def computeFitnessOld(modularities, popSize):
86  # ------------------------------------------------------------------------------
87  def computeFitness(modularities, popSize):
93  # ------------------------------------------------------------------------------
94  def createNextGeneration(population, fitness, popSize, codeSize, mutationProb,
    elitismSize):
130 # ------------------------------------------------------------------------------
131 def getParentAccordingToFitness(population, fitness, popSize, codeSize):
136 # ------------------------------------------------------------------------------
137 def printCluFile(individual, codeSize, fileName, popSize, elitismSize,
    numGenerations, mutationProb, bestModularity):
```

Figure 1: GeneticAlgorithm.py overview.

Now let us make a small review from every method:

2.1.1 createAdjacenceMatrix

This is a method just to read the file "*.net" and use it to create the adjacency matrix:

```
10  def createAdjacenceMatrix(fileName):
11      isEdges = False
12      lineCounter = 0
13      edges = []
14      with open(fileName) as infile:
15          for line in infile:
16              if isEdges:
17                  # Remove blank spaces
18                  line = " ".join(line.split())
19                  edge = line.split(' ')
20  ›               if len(edge) > 2:⋯
24              else:
25                  lineCounter += 1
26              #
27              if not isEdges and line.startswith("*Edges"):
28                  isEdges = True
29                  matrixSize = lineCounter - 2
30      # ------------------------------------------------------
31      adjMat = np.zeros((matrixSize,matrixSize))
32      for edge in edges:
33  ›           # Replicate the connection in both ways⋯
35          adjMat[edge[0] - 1,edge[1] - 1] = 1
36          adjMat[edge[1] - 1,edge[0] - 1] = 1
37      return adjMat, matrixSize
```

2.1.2 createInitialPopulation

This method uses random numbers to create the first generation according to the desired population size and the size of the chromosomes code:

```
"
def createInitialPopulation(numIndividuals, codeSize):
    population =  np.random.randint(2, size=(numIndividuals, codeSize))
    # Make a threshold over 0.5
    population[population > 0.5] = 1
    population[population < 0.5] = 0
    return np.matrix(population)
"
```

2.1.3 computeModularity

This method computes the modularity according to the instructions detailed in the definition of the problem:

```
46  def computeModularity(matrix, pop, L, Ki, Kj, codeSize):
47      modularities = np.zeros(len(pop))
48      # iterate individuals and then positions of adjacency matrix
49      for ind,individual in enumerate(pop):
50          individual = individual.getA1() # Return self as a flattened ndarray.
51          modularity = 0
52          for j in range(codeSize):⋯
59          modularity = modularity * (1/L) # We don't need the number 2 because we
            are using a symmetric matrix
60          modularities[ind] = modularity
61      return modularities
```

2.1.4 computeFitness

This method adapt the modularity output so it can be used as a fitness in the evolution process, for this it make use of a ranking procedure and then assign a value to every item according to its position in the rank:

```
87  def computeFitness(modularities, popSize):
88      # Get index order, if all values are the same it returns [1,2,3,4...]
89      fitness = np.argsort(np.argsort(modularities))
90      # Force values to sum 1 (this will be used in numpy.random.choice function)
91      fitness = fitness/float(np.sum(fitness))
92      return fitness
```

2.1.5 createNextGeneration and getParentAccordingToFitness

With those two methods we use all the information gathered until this point and use it to create a better new generation of descendant individuals. To select parents of new generation we use a process where we give higher priority to those parents with higher modularity using the fitness function we saw in the step before:

```
94  def createNextGeneration(population, fitness, popSize, codeSize, mutationProb, elitismSize):
95      children = []
96      # Include elite individuals
97      if (elitismSize > 0):
98          elitismIndices = fitness.argsort()[-1*elitismSize:]
99          for i in range(elitismSize):
100             children.append(population[elitismIndices[i]].getA1())
101     # Breed to complete the population
102     while len(children) < popSize:
103         fatherInd = getParentAccordingToFitness(population, fitness, popSize, codeSize)
104         motherInd = getParentAccordingToFitness(population, fitness, popSize, codeSize)
105         if fatherInd != motherInd:
106             father = population[fatherInd].getA1()
107             mother = population[motherInd].getA1()
108             child = [0] * codeSize
109             if True:   # Uniform crossover
110                 # Mix chromosomes using uniform crossover
111                 crossoverSelector = np.random.randint(2, size=codeSize)
112                 for i in range(codeSize):
113                     child[i] = father[i] if (crossoverSelector[i] == 1) else mother[i]
114             if False:    # One point crossover
115                 fatherCodeSize = codeSize / 2
116                 motherCodeSize = codeSize / 2
117                 # Add 1 chromosome to father if odd number
118                 if (motherCodeSize + fatherCodeSize) != codeSize:
119                     fatherCodeSize += 1
120                 child = np.hstack((father[:fatherCodeSize],mother[codeSize-motherCodeSize:]))
121             # ------------------------
122             # Add mutation, Select random position in code and change it
123             mutate = np.random.rand(1)
124             if mutate < mutationProb:
125                 randPos = np.random.randint(codeSize)
126                 child[randPos] = 1 if child[randPos] == 0 else 0
127             # ------------------------
128             children.append(child)
129     return np.matrix(children)
```

```
130 # ------------------------------------------------------------------------------
131 def getParentAccordingToFitness(population, fitness, popSize, codeSize):
132     # print "getParentAccordingToFitness"
133     choice = np.random.choice(popSize, 1, p=fitness)
134     # print choice
135     return choice
```

2.1.6 printCluFile

This method is only to save the new generated "*.clu" file using the information from the best individual obtained after the training:

```
137 def printCluFile(individual, codeSize, fileName, popSize, elitismSize,
    numGenerations, mutationProb, bestModularity):
138     outName = fileName.split('/')[-1]
139     outName = outName.split('.')[0]
140     outName = '../outputs/' + outName + '_out'
141     outName += '_[popS=%d, elitS=%d, numGen=%d, mutProb=%.2f, bestMod=%.3f]' %
        (popSize, elitismSize, numGenerations, mutationProb, bestModularity)
142     outName += '.clu'
143     f = open(outName, 'w')
144     f.write('*Vertices ' + str(codeSize) + '\n')
145 ›   for i in range(codeSize):⋯
147     f.close()
```

2.2 MainScript.py

In this script we simply declare the settings we want to use during the training:

```
27 popSize = 100 #100 or 50
28 elitismSize = int(popSize * 0.2)
29 numGenerations = 200
30 mutationProb = 0.25
31 modularityHistory = []
32 verbose = 1
33 restingTime = 0.25
```

Then we read the input file, create the adjacency matrix, and create the initial random population:

```
38 # Create adjacency matrix from input file
39 adjMatrix, codeSize = createAdjacenceMatrix(inputFile)
40 # Compute L, Ki and Kj
41 L = np.sum(adjMatrix)
42 Ki = np.sum(adjMatrix, axis=0)
43 Kj = np.sum(adjMatrix, axis=1)
44 if verbose > 1: print "Adjacency matrix"; print adjMatrix
45 # ---------------
46 # Create initial population using random numbers
47 population = createInitialPopulation(numIndividuals=popSize, codeSize=codeSize)
48 if verbose > 1: print "Population"; print population
```

After this we compute the modularity and fitness for the first generation:

```
50 # Compute modularity
51 modularities = computeModularity(matrix=adjMatrix, pop=population, L=L, Ki=Ki,
   Kj=Kj, codeSize=codeSize)
52 if verbose > 0: print "Modularities"; print modularities
53 modularityHistory.append(modularities)
54 # ---------------
55 # Get fitness from modularity
56 fitness = computeFitness(modularities=modularities, popSize=popSize)
57 if verbose > 0: print "Fitness"; print fitness
```

Then we loop over the same process for every generation we want to compute:

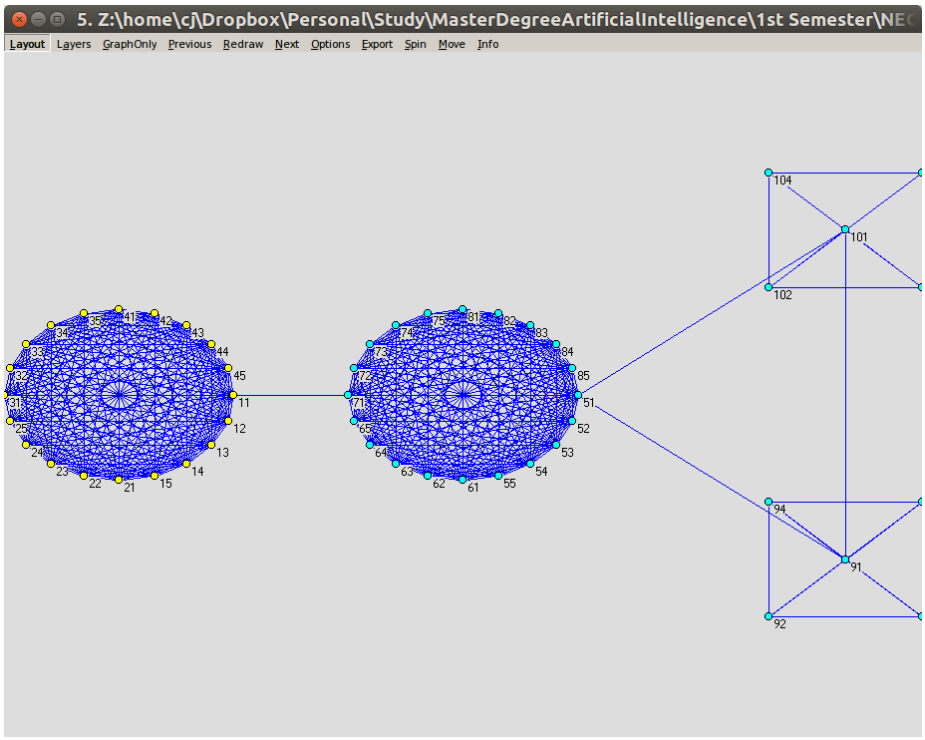
```
59  # Iterate for future generations
60  bestModularity = np.max(modularities)
61  lastLoopBest = bestModularity
62  bestIndividual = population[np.argmax(modularities)]
63  for i in range(numGenerations):
64      if i%10 == 0 or i == (numGenerations-1):
65          print "Generation= %d" % i
66          print '[popS=%d, elitS=%d, numGen=%d, mutProb=%f, bestMod=%.3f]' %
 •          (popSize, elitismSize, numGenerations, mutationProb, lastLoopBest)
67      # Get next generation
68      population = createNextGeneration(population=population, fitness=fitness,
 •      popSize=popSize, codeSize=codeSize, mutationProb=mutationProb,
 •      elitismSize=elitismSize)
69      if verbose > 0: print "newGeneration"; print population
70      # ---------------
71      # Compute modularity
72      modularities = computeModularity(matrix=adjMatrix, pop=population, L=L,
 •      Ki=Ki, Kj=Kj, codeSize=codeSize)
73      if verbose > 0: print "Modularities"; print modularities
74      modularityHistory.append(modularities)
75      # ---------------
76      # Get fitness from modularity
77      fitness = computeFitness(modularities=modularities, popSize=popSize)
78      if verbose > 0: print "Fitness"; print fitness
79      # Save best individual
80      lastLoopBest = np.max(modularities)
81      if lastLoopBest > bestModularity:
82          bestModularity = lastLoopBest
83          bestIndividual = population[np.argmax(modularities)]
```

3 Results (partitions) and evaluation of the partitions

In this section we want to show the results obtained after every simulation using Pajek and our genetic algorithm implementation.

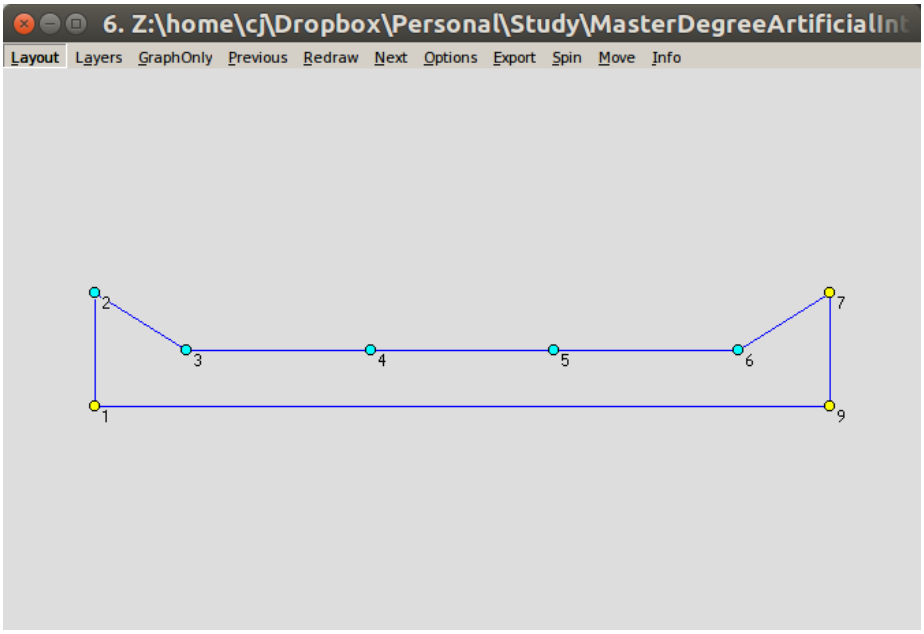
3.1 20x2+5x2.net

Summary of the test: 20x2+5x2_out_[popS=100, elitS=50, numGen=100, mutProb=0.25, bestMod=0.496].clu



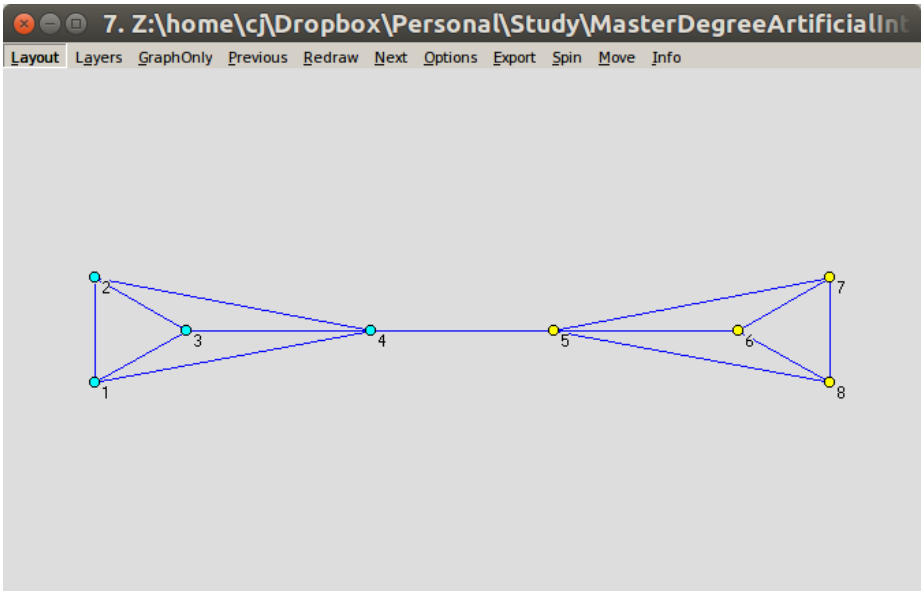
3.2 circle9.net

Summary of the test: circle9_out_[popS=100, elitS=50, numGen=100, mutProb=0.25, bestMod=0.272].clu



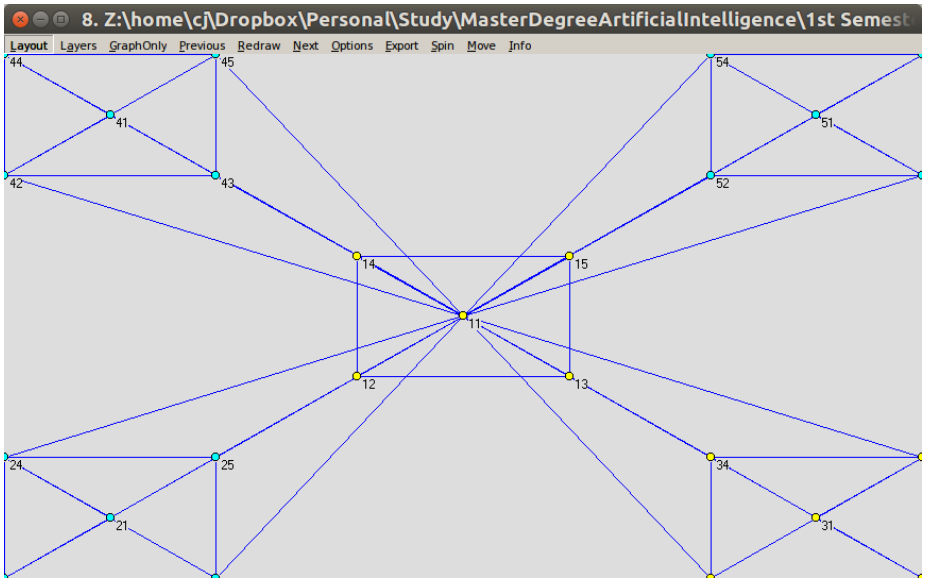
3.3 graph3+2+3.net

Summary of the test: graph3+2+3_out_[popS=100, elitS=50, numGen=100, mutProb=0.25, bestMod=0.423].clu



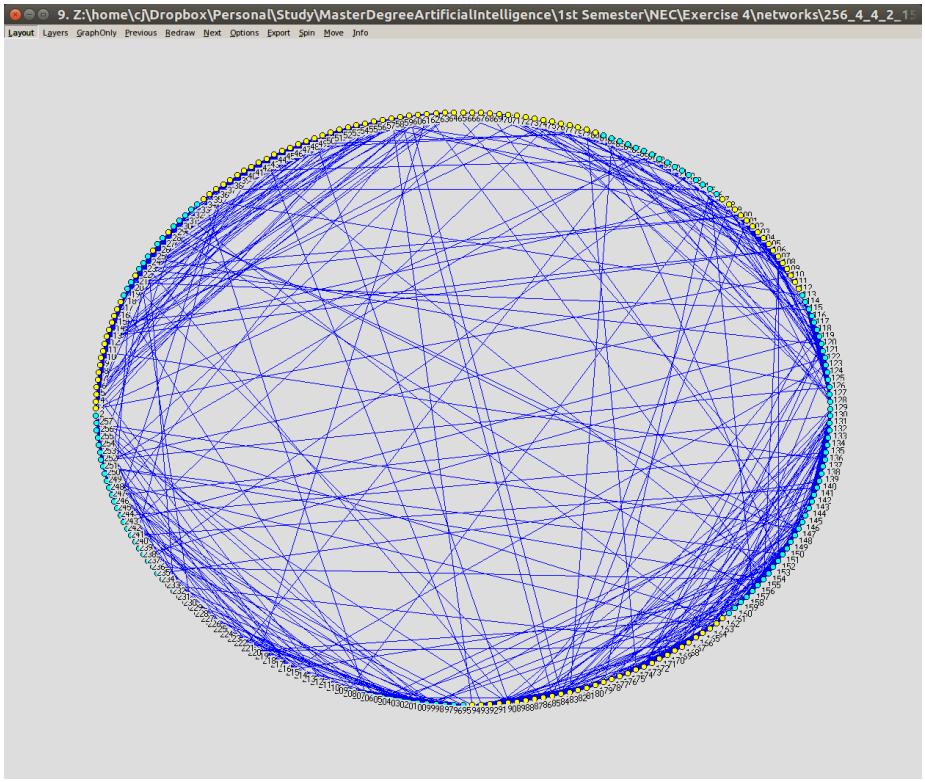
3.4 rb25.net

Summary of the test: rb25_out_[popS=100, elitS=50, numGen=100, mutProb=0.25, bestMod=0.314].clu



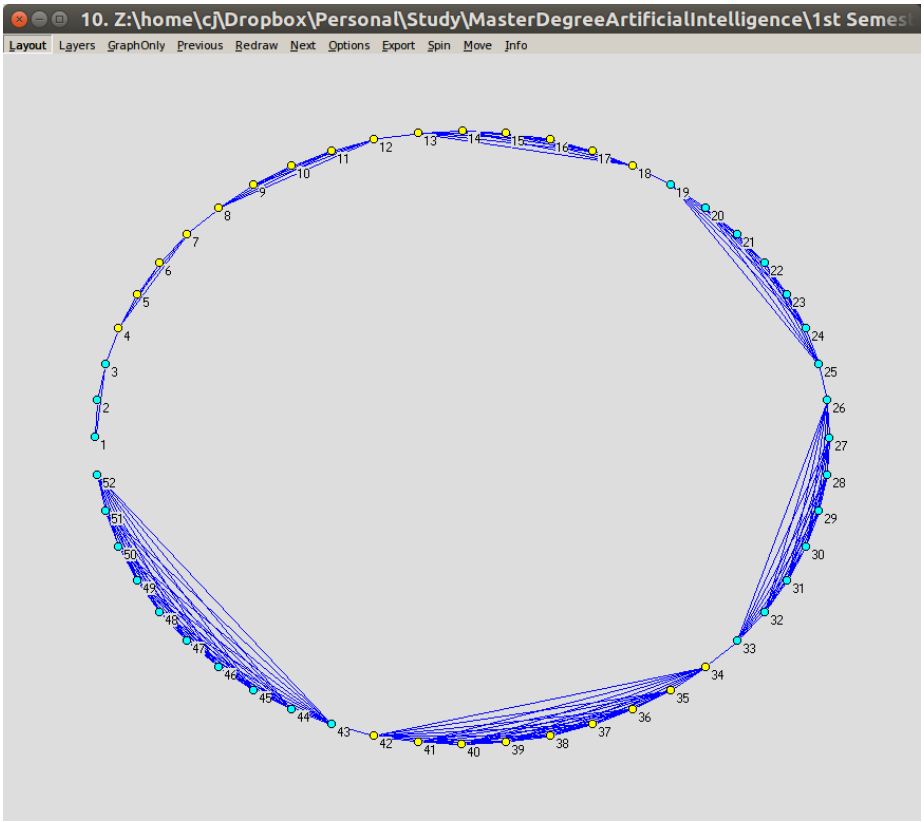
3.5 256_4_4_2_15_18_p.net

Summary of the test: 256_4_4_2_15_18_p_out_[popS=100, elitS=50, numGen=100, mutProb=0.25, bestMod=0.410].clu



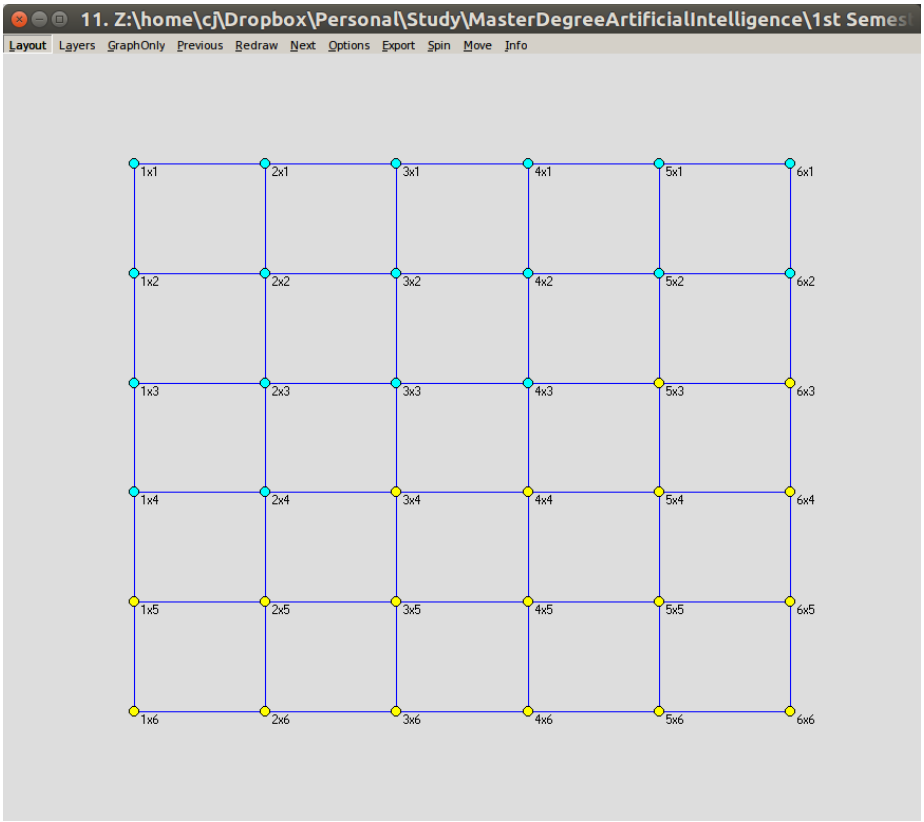
3.6 cliques_line.net

Summary of the test: cliques_line_out_[popS=100, elitS=50, numGen=100, mutProb=0.25, bestMod=0.462].clu



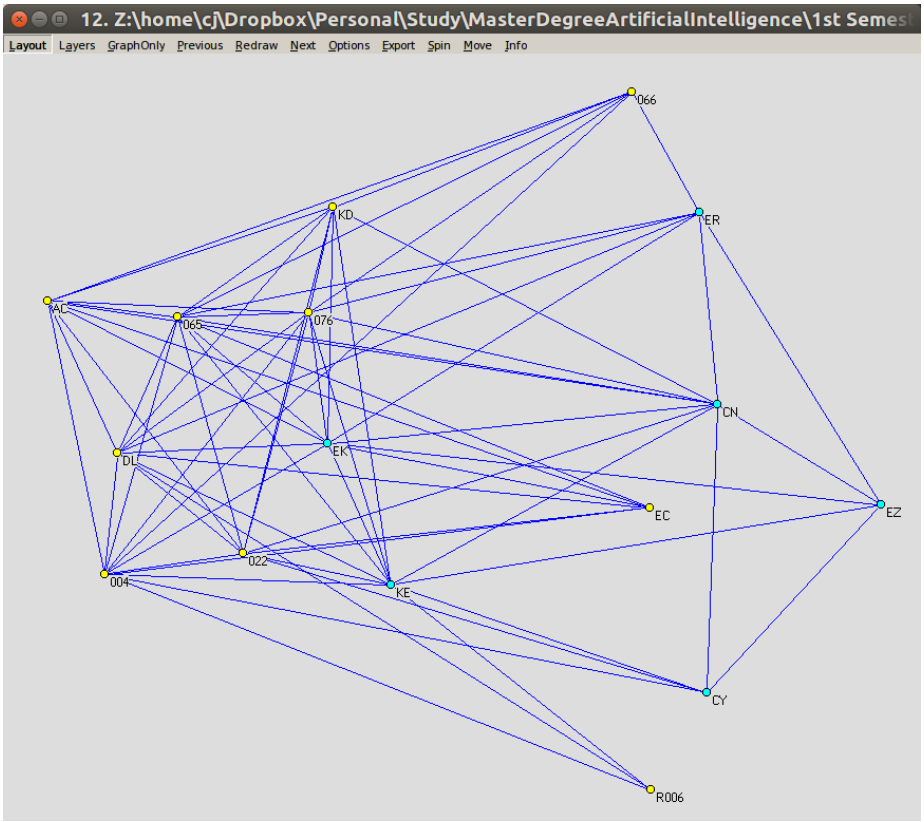
3.7 grid-6x6.net

Summary of the test: grid-6x6_out_[popS=100, elitS=50, numGen=100, mutProb=0.25, bestMod=0.367].clu



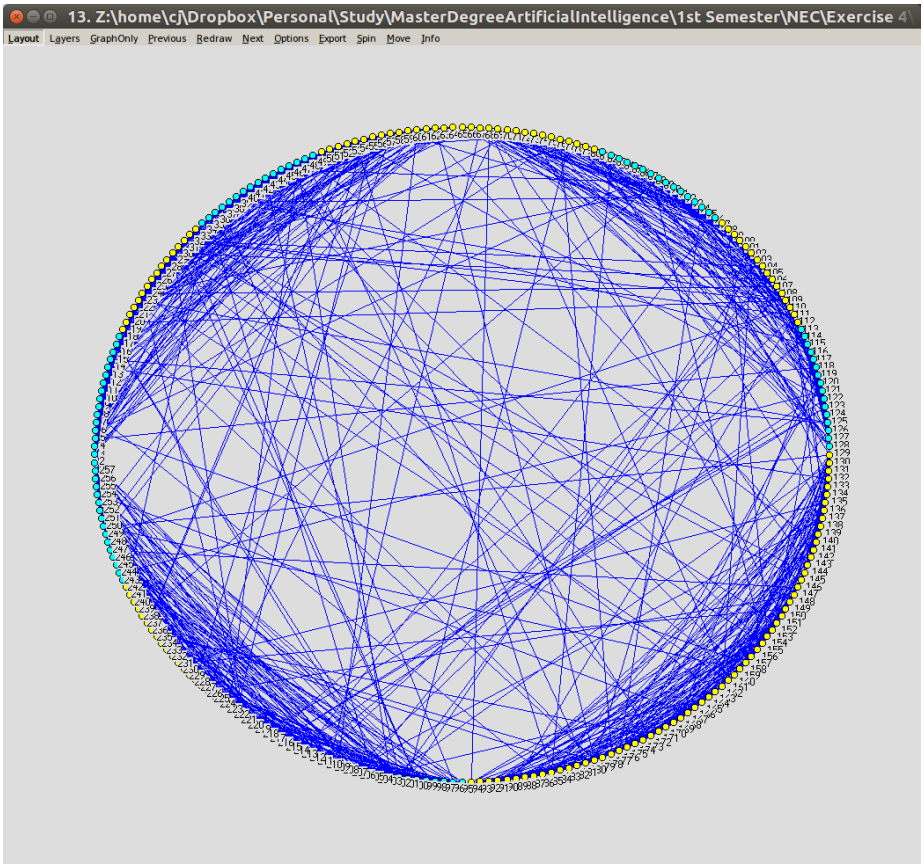
3.8 rhesus_simetrica.net

Summary of the test: rhesus_simetrica_out_[popS=100, elitS=20, numGen=200, mutProb=0.25, bestMod=0.096].clu



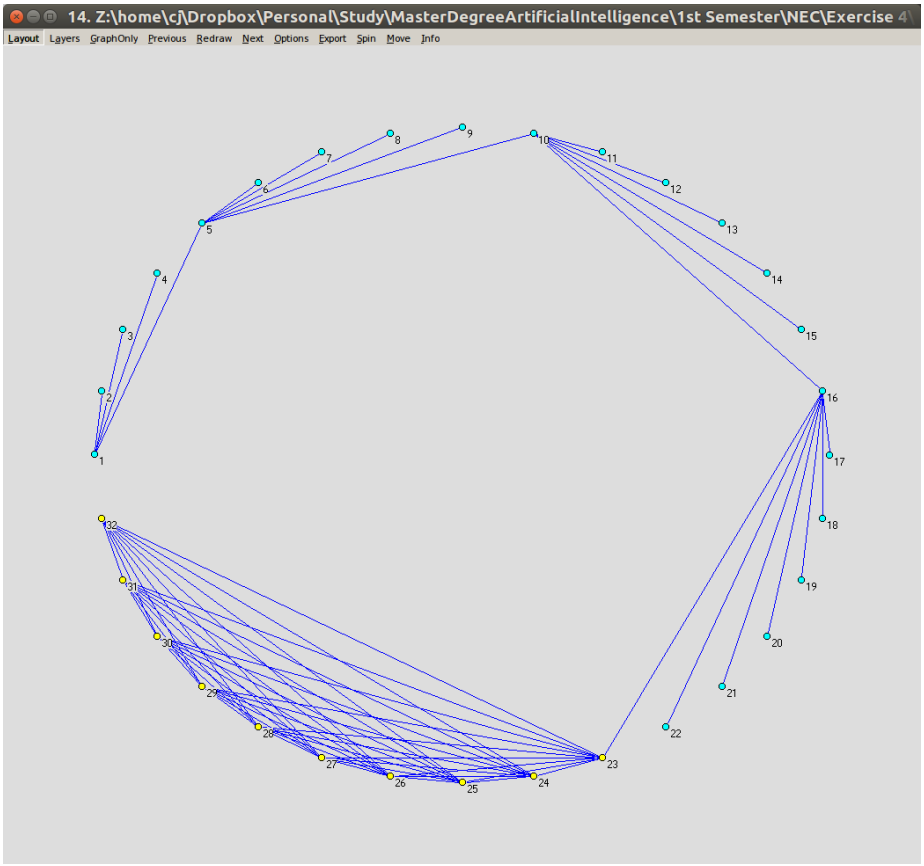
3.9 256_4_4_4_13_18_p.net

Summary of the test: 256_4_4_4_13_18_p_out_[popS=100, elitS=20, numGen=200, mutProb=0.25, bestMod=0.364].clu



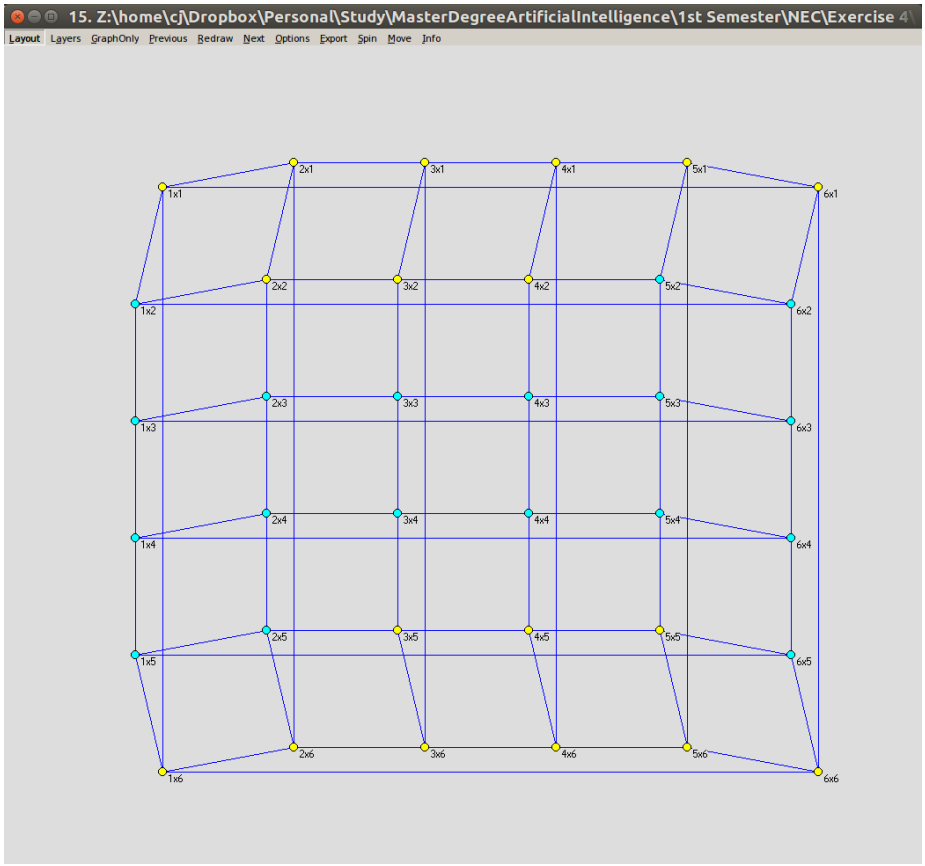
3.10 clique_stars.net

Summary of the test: clique_stars_out_[popS=100, elitS=20, numGen=200, mutProb=0.25, bestMod=0.421].clu



3.11 grid-p-6x6.net

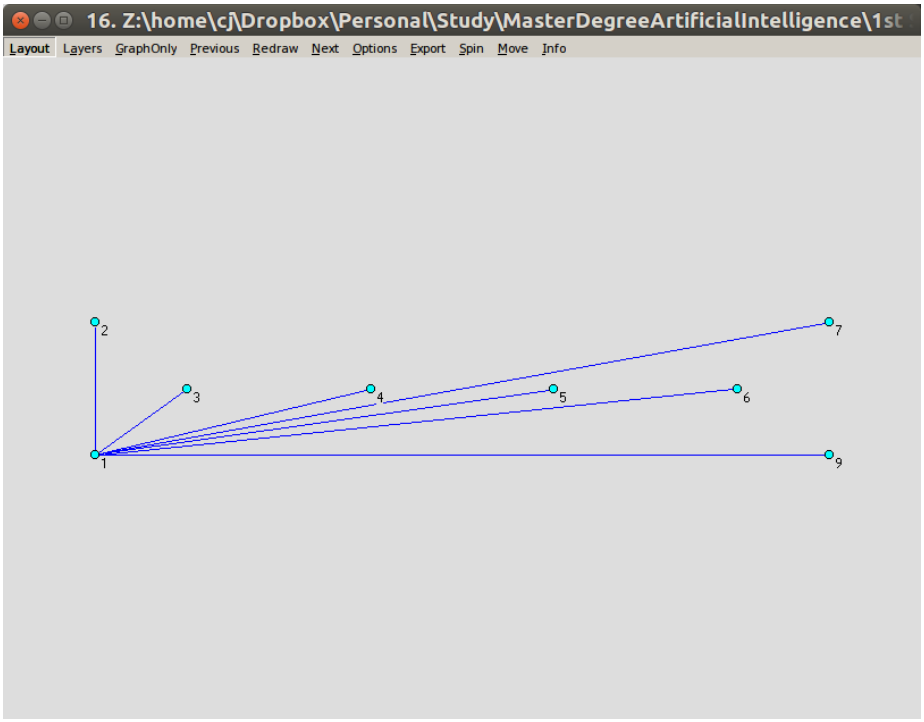
Summary of the test: grid-p-6x6_out_[popS=100, elitS=20, numGen=200, mutProb=0.25, bestMod=0.278].clu



3.12 star.net

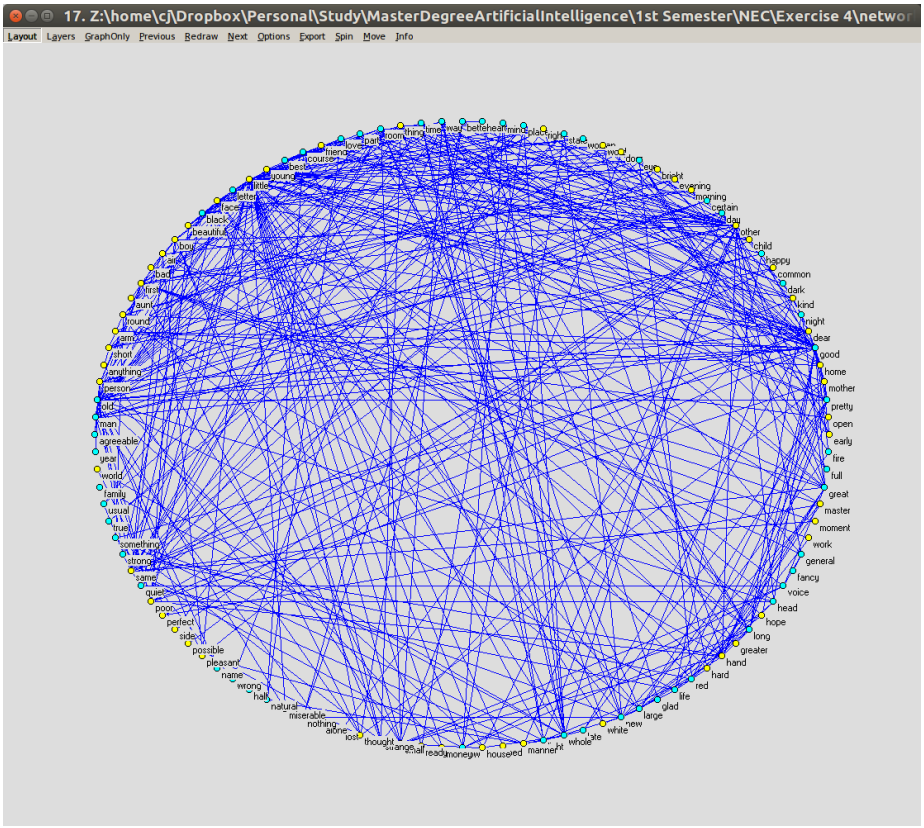
Summary of the test: star_out_[popS=100, elitS=20, numGen=200, mutProb=0.25, bestMod=0.000].clu

Note: This sample is interesting because this network disposition results being impossible to be split in two groups, that's why the biggest modularity comes when all nodes belong to same cluster and modularity equals zero.



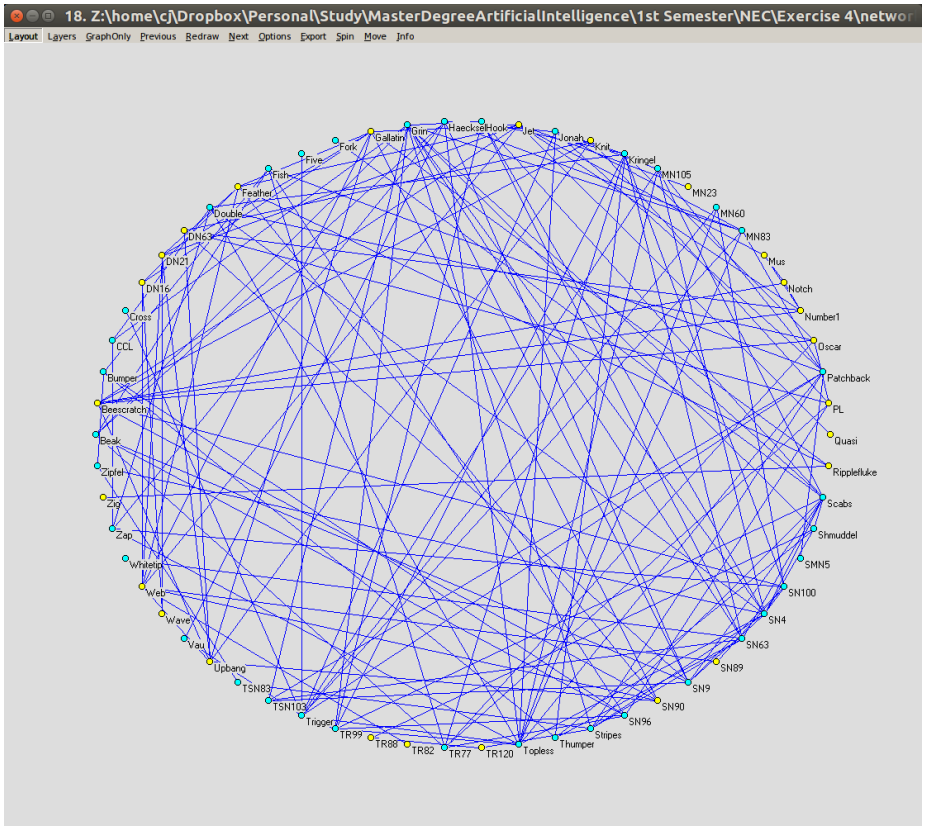
3.13 adjnoun.net

Summary of the test: adjnoun_out_[popS=100, elitS=20, numGen=200, mutProb=0.25, bestMod=0.178].clu



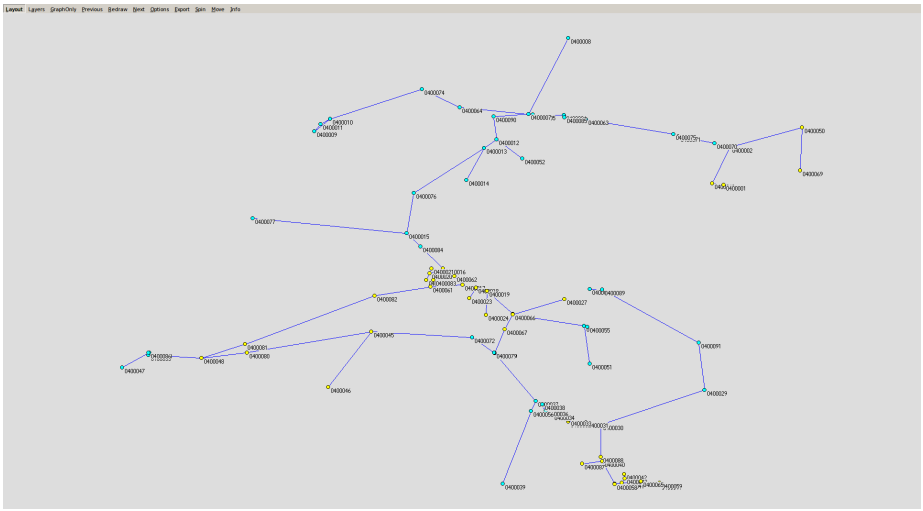
3.14 dolphins.net

Summary of the test: dolphins_out_[popS=100, elitS=20, numGen=200, mutProb=0.25, bestMod=0.386].clu



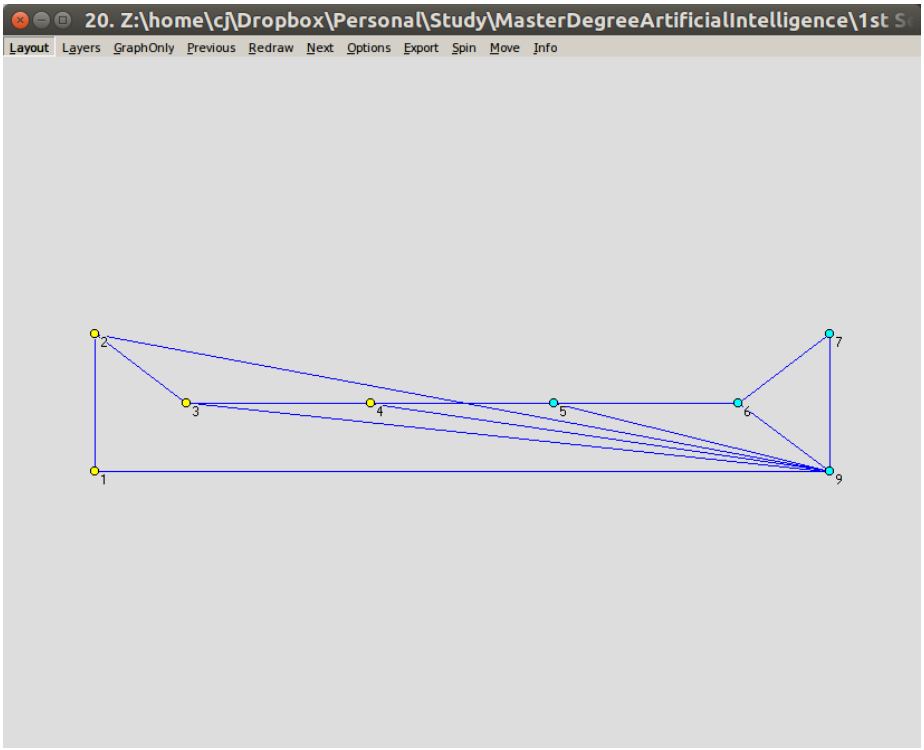
3.15 qns04_d.net

Summary of the test: qns04_d_out_[popS=100, elitS=20, numGen=200, mutProb=0.25, bestMod=0.410].clu



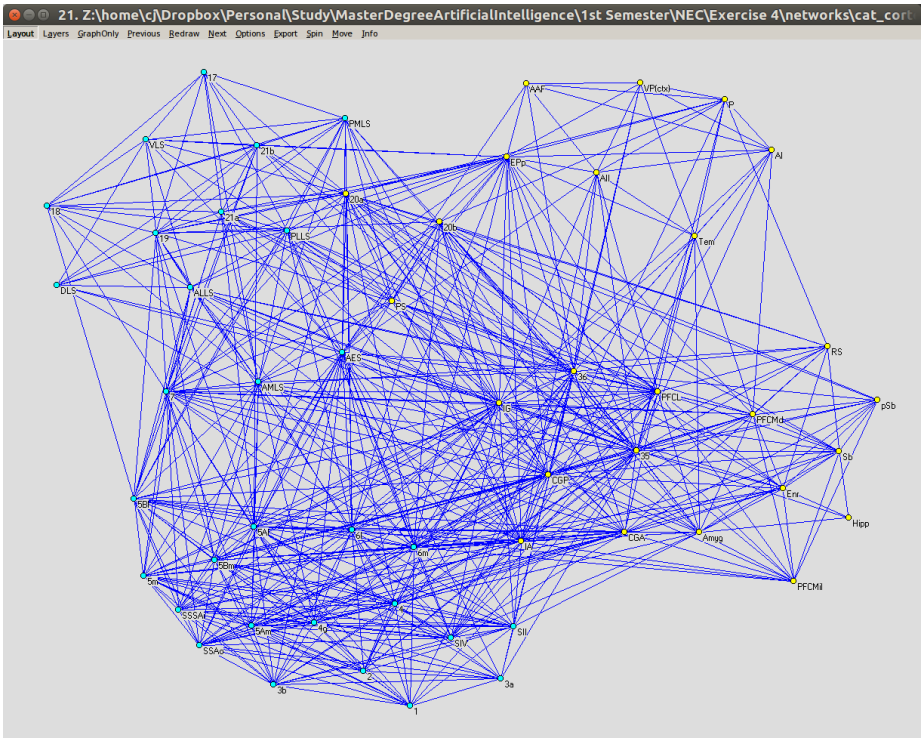
3.16 wheel.net

Summary of the test: wheel_out_[popS=100, elitS=20, numGen=200, mutProb=0.25, bestMod=0.094].clu



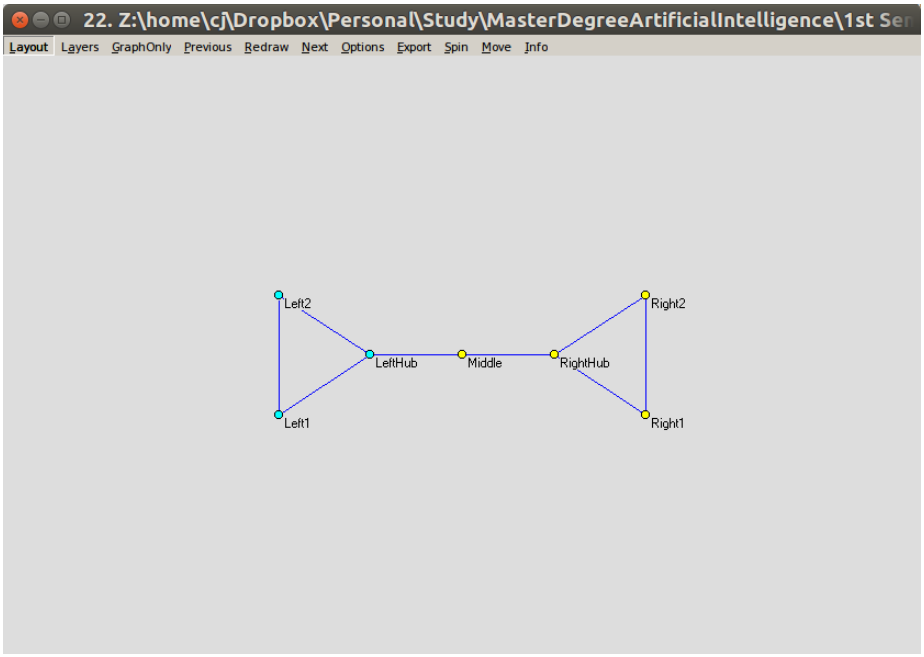
3.17 cat_cortex_sim.net

Summary of the test: cat_cortex_sim_out_[popS=100, elitS=20, numGen=200, mutProb=0.25, bestMod=0.201].clu



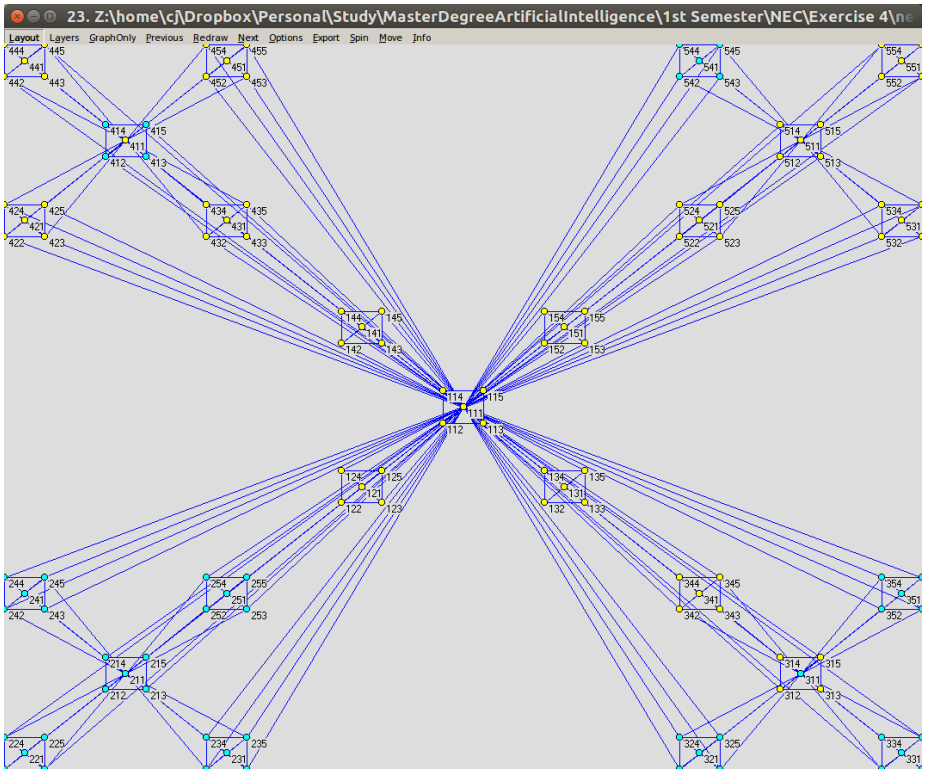
3.18 graph3+1+3.net

Summary of the test: graph3+1+3_out_[popS=100, elitS=50, numGen=100, mutProb=0.25, bestMod=0.367].clu



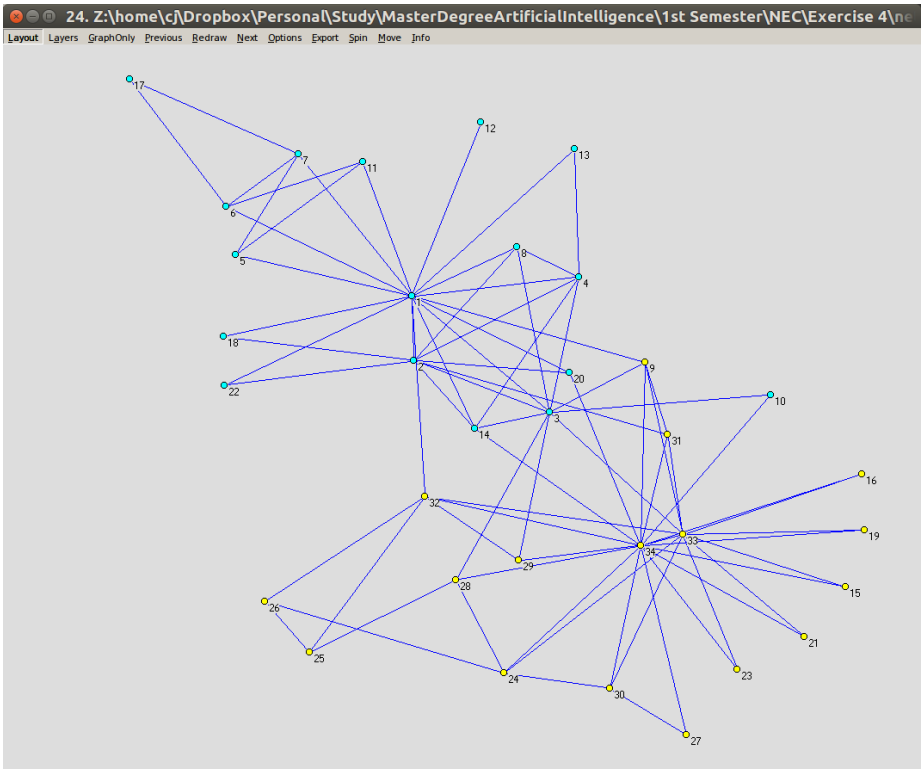
3.19 rb125.net

Summary of the test: rb125_out_[popS=100, elitS=20, numGen=200, mutProb=0.25, bestMod=0.330].clu



3.20 zachary_unwh.net

Summary of the test: zachary_unwh_out_[popS=1000, elitS=500, numGen=100, mutProb=0.50, bestMod=0.372].clu



References

[1] Wikipedia. Modularity (networks) — wikipedia, the free encyclopedia, 2016. [Online; accessed 9-February-2017].

[2] Wikipedia. Genetic algorithm — wikipedia, the free encyclopedia, 2017. [Online; accessed 9-February-2017].

[3] Wikipedia. Fitness (biology) — wikipedia, the free encyclopedia, 2017. [Online; accessed 9-February-2017].