ECE 3790 Lab 1 Report – Winter 2019

Daniel Lovegrove (7763168)

INTRODUCTION

The point of Lab 1 is to create a naive algorithm to split a graph into two equal parts, minimizing the connection between the two groups of nodes. This approach will be built upon in the coming months to make a more sophisticated algorithm.

To minimize the number of connections between two groups of nodes in a graph, we will be using the method of swapping a node with one from the other group and testing the cost of it. If the new solution has a lower cost than the current best, it is kept as the new current best. The cost depends on the number of connections between the groups. This approach will be continuously applied until the solutions do not appreciably get better.

The language I chose to write this lab in is Python 3. There are two main portions of code, one is a tool used to generate graphs and read/write graphs from files, and the other portion is the algorithm itself and all code needed to set up and support it. All of the code I used is in Appendix B, preceded by how to use it in Appendix A.

LAB QUESTIONS

Q1: Algorithm Summary

At its most basic, the algorithm randomly swaps items between the two groups of graph nodes, and if the new configuration has a lower cost than the best one so far, it is considered the best going forward. If worse configurations than the best are found N*N times in a row (where N is the number of nodes in the graph), the algorithm halts. Rather than set a static number of max worse cases to try, I made it so that it depends on the size of the graph.

The pseudo-code for the algorithm is as follows:

```
split graph into two equal groups
while max number of consecutive worse cases has not been reached:
      if not first iteration:
            randomly swap a single element between each group
      calculate cost of the current solution
      if current cost less than best/lowest cost solution:
            assign best solution to current solution
            reset consecutive number of worse cases
      else:
            increment consecutive number of worse cases than best
```

Q2: Minimum Score for Test Cases

The following are the minimum scores for some of the text files given in UMLearn, and for a few randomly generated graphs.

Graph Tested	**Initial # of Connections**	**Final # of Connections**	**Percent Improvement**
AdjMatSym2019.txt	790	689	12.8%
AdjMatCCSparce.txt	0	0	0%
AdjMatRand.txt	734	675	8.0%
Random 100 node graph, 95% sparse	126	95	24.6%
Random 100 node graph, 80% sparse	510	435	14.7%
Random 100 node graph, 70% sparse	747	671	10.2%

(Table 1: Test Scores – Daniel Lovegrove)

Q3: Running Time

Using Python's `time` library I measured the time it took to run the algorithm, ignoring the time to load the graph. I first measured 3 trials for 50 nodes and took the average, then did the same for 100 nodes.

- Average time for 50 nodes: 1.064 seconds
- Average time for 100 nodes: 17.744 seconds

The time increased by 16.7 times. Since $2^4 = 16 \sim= 16.7$, I will say that the time scaled by a power of 4.

Q4: Complexity

Since the problem increased by a power of 4 for a doubling of the problem size in Q3, I will estimate the complexity as $O(n^4)$. Not very good.

Q5: Improvements

There are several possible ways to improve this basic algorithm. One way may be to decide whether or two nodes are worth switching by testing if each node is connected to many nodes in the group they're currently in. If they are, move on and try swapping different nodes (similar to what's said in the lab document). It may also be worth trying a few completely random configurations rather than swapping a single node.

Q6: To Think About

We were trying to minimize the number of connections between two groups of nodes in a graph. There are *n choose n/2* possible solutions. In other words, from a group of *n* nodes, you choose half to go in one group, the rest automatically go in the other group.

Assuming there are *a* optimal solution in a graph of 100 nodes, the chance of selecting an optimal solution on your first try at random is *very* slim (even if *a* is relatively large):

$$p = \frac{a}{(100\ Choose\ 50)} = \frac{a}{1.0089134 * 10^{29}} \simeq a * 10^{-29}$$

APPENDIX A: CODE DOCUMENTATION

To run the code for this lab, you will require Python 3. On my Ubuntu computer, Python 3 can be run from the command line using the `python3` command. On some computers, `python` is the command to run Python 3. Check with `python --version` or `python3 -version`.

If you do not have the Python library `plac` installed on your computer, you will need to install it with `pip3 install plac`. `Plac` is used to parse command line arguments.

The two main files to run the code for the naive algorithm are `lab1_random_graph.py` and `lab1_graph_from_file.py`. These files are self explanatory in what they do, the former runs the algorithm on a randomly generated graph, and the latter runs the algorithm on a graph generated from a text file.

The code itself is in Appendix B.

lab1_random_graph.py

Running `python3 lab_1_random_graph.py --help` from the command line gives information about how to call the script:

```
usage: lab1_random_graph.py [-h] size sparseness

positional arguments:
  size        Size of graph
  sparseness  Percentage of graph that is sparse

optional arguments:
  -h, --help  show this help message and exit
```

The `size` argument is the number of nodes in the graph, but the `sparseness` argument may not be as obvious. Sparseness indicates the percent probability that any node has no connection to another node. If you specify 50, each node has a 50% chance of having no connection to any other node. If you specify 100, each node has a 100% chance of having no connection to any other node, meaning the whole graph will be unconnected.

The following is a sample invocation of `lab1_random_graph.py` on the command line for a graph with 100 nodes and 95% sparseness:

```
> python3 lab1_random_graph.py 100 95
```

lab1_graph_from_file.py

Running `python3 lab1_graph_from_file.py --help` gives information about how to call the script:

```
usage: lab1_graph_from_file.py [-h] filename
```

```
positional arguments:
  filename    Path to file containing adjacency matrix

optional arguments:
  -h, --help  show this help message and exit
```

There is only one argument for this script: `filename`. This is the relative/absolute path to a text file that represents an adjacency matrix.

The following is a sample invocation of `lab1_graph_from_file.py`, giving it the file `AdjMatCCSparce.txt` as input (this file is in the same folder as the script):

```
> python3 lab1_graph_from_file.py ./AdjMatCCSparce.txt
```

APPENDIX B: CODE

FILE: graphtools.py

Responsible for creating adjacency matrices representing graphs created randomly or from file.

```
import random
import numpy

def create_symmetric_graph(_size, sparseness):
    """
    Creates and returns a symmetric 2D numpy array of N x N size.

    _size: Dimension N
    sparseness: Value from 0 to 100 determining the percent chance that a node has no connection
        to another node. 100 means that a node will have a 100% chance of having no connection to
        another node (no nodes will be connected). 0 means that a node will have a 0% chance of
        not being connected to another node (the graph will be complete).
    """
    if _size < 2:
        raise ValueError('_size must not be less than two')

    if sparseness < 0 or sparseness > 100:
        raise ValueError('Sparseness must be between 0 and 100 (inclusive)')

    # Create a matrix of random integers from 1 to 9
    graph = numpy.random.random_integers(1, 9, size=(_size, _size))

    # Apply sparseness
    chance_of_sparseness = numpy.ones(100)
    chance_of_sparseness[0: sparseness] = 0

    for i in range(0, _size):
        for j in range(0, _size):
            graph[i][j] = graph[i][j] * random.choice(chance_of_sparseness)

    # Get the lower triangle of the random matrix (also ensures diagonal is all zeros)
    symmetric_graph = numpy.tril(graph, k=-1)

    # Symmetrize the matrix
    for i in range(0, _size):
        for j in range(0, _size):
            symmetric_graph[i][j] = symmetric_graph[j][i]

    return symmetric_graph


def create_symmetric_half_connected_graph(_size, sparseness):
    """
```

```
    Creates and returns a symmetric 2D numpy array of N x N size, where the nodes are split into
    two disjoint groups.

    _size: Dimension N
    sparseness: Value from 0 to 100 determining the percent chance that a node has no connection
        to another node. 100 means that a node will have a 100% chance of having no connection to
        another node (no nodes will be connected). 0 means that a node will have a 0% chance of
        not being connected to another node (the graph will be complete).
    """
    # Get a symmetric graph
    half_connected_graph = create_symmetric_graph(_size, sparseness)

    # Create two disjoint sets of nodes
    num_nodes_group_1 = int(_size / 2)
    for i in range(0, num_nodes_group_1):
        for j in range(num_nodes_group_1, _size):
            half_connected_graph[i][j] = 0
            half_connected_graph[j][i] = 0

    return half_connected_graph


def convert_text_matrix_to_graph(filepath):
    """
    Converts and returns a 2D numpy array from a text representation of an adjacency matrix.

    filepath: Path to text file
    """
    filelines = []
    with open(filepath, 'r') as f:
        filelines = f.readlines()

    rows_of_numbers = []
    for line in filelines:
        formatted_line = line.replace(' ', '').replace('\n', '')
        numbers = []

        for number in formatted_line:
            numbers.append(int(number))

        rows_of_numbers.append(numbers)

    numpy_array = numpy.array(rows_of_numbers)

    return numpy_array
```

FILE: lab1_random_graph.py

Creates a graph using `graphtools` and starts the algorithm. Note that `plac` is needed to parse command line arguments and may need to be installed with `pip` if it is not available on a host.

```
import graphtools
import grouping_algorithm
import plac

def str_to_int(string):
   try:
       return int(string)
   except ValueError:
       raise ValueError('Value "{}" could not be converted to an integer'.format(string))

def main(size: "Size of graph", sparseness: "Percentage of graph that is sparse"):
   graph = graphtools.create_symmetric_graph(str_to_int(size), str_to_int(sparseness))
   grouping_algorithm.split_into_groups(graph)

if __name__ == '__main__':
   try:
       plac.call(main)
   except KeyboardInterrupt:
       print('Exiting.')
```

FILE: lab1_graph_from_file.py

Loads a graph from file using `graphtools` and starts the algorithm.

```
import graphtools
import grouping_algorithm
import plac
from pathlib import Path

def main(filename: "Path to file containing adjacency matrix"):
   filepath = Path(filename)

   if filepath.is_file():
       graph = graphtools.convert_text_matrix_to_graph(filepath)
       grouping_algorithm.split_into_groups(graph)
   else:
       print('"{}" does not exist.'.format(filename))

if __name__ == '__main__':
   try:
       plac.call(main)
   except KeyboardInterrupt:
       print('Exiting.')
```

FILE: grouping_algorithm.py

The algorithm itself. Uses a naive technique to try and split a graph into two minimally connected groups of components/nodes.

```
import random
import copy
import time

UNEVEN_PENALTY = 5

class Cost:
    """Defines the cost of a solution"""
    def __init__(self, num_connections, uneven_cost):
        self.num_connections = num_connections
        self.uneven_cost = uneven_cost

class AlgorithmSolution:
    """Contains all of the data associated with a solution to the problem"""
    def __init__(self, group_a, group_b, cost: Cost):
        self.group_a = group_a
        self.group_b = group_b
        self.cost = cost

def split_into_groups(adjacency_matrix):
    """
    Splits a matrix into two groups with minimal connection between groups.

    adjacency_matrix: A numpy N x N array representing a symmetric adjacency matrix
    """
    print('Using naive algorithm to split matrix into two groups with minimal connections.')
    print('Press CTRL-C at any point to stop.\n')

    graph_size = len(adjacency_matrix)
    # Number of tries to find a better solution without finding a better one is proportional to
graph size
    consecutive_worse_cost_max = graph_size * graph_size

    component_group_A = []
    component_group_B = []

    # Initially, separate graph into two non-optimized groups
    for i in range(0, graph_size):
        if len(component_group_A) >= (int(graph_size / 2)):
            component_group_B.append(i)
        else:
            component_group_A.append(i)

    # Make the best solution really bad at first (very high cost)
    best_solution = AlgorithmSolution([], [], Cost(graph_size * graph_size, graph_size *
graph_size))
```

```
    iteration = 1
    consecutive_worse_cost = 0
    better_solution_found = True
    start_time = time.time()

    while consecutive_worse_cost < consecutive_worse_cost_max:
        # Swap two nodes between group A and B (except on first iteration)
        if iteration != 1:
            indexA = random.choice(range(0, len(component_group_A)))
            indexB = random.choice(range(0, len(component_group_B)))
            temp = component_group_A[indexA]
            component_group_A[indexA] = component_group_B[indexB]
            component_group_B[indexB] = temp

        # Get the cost for this iteration
        current_cost = calculate_cost(adjacency_matrix, component_group_A, component_group_B)

        # If the new solution is better than the best one so far, make it the new best
        if current_cost.num_connections < best_solution.cost.num_connections:
            better_solution_found = True
            consecutive_worse_cost = 0
            best_solution = AlgorithmSolution(copy.deepcopy(component_group_A),
copy.deepcopy(component_group_B), current_cost)

        # Else the new solution is no better than the current one
        else:
            better_solution_found = False
            consecutive_worse_cost += 1

        # Only print cost of new solution if we found a new best
        if better_solution_found:
            print("Iteration {}".format(iteration))
            print("Cost due to unevenness: {}\nNumber of connections:
{}\n".format(current_cost.uneven_cost, current_cost.num_connections))

        # Don't keep running if cost is zero
        if best_solution.cost.num_connections == 0:
            break

        iteration += 1

    print("> Finished running after {} iterations.".format(iteration))
    print("> Finished in {:.5f} seconds\n".format(time.time() - start_time))
    print_final_solution(best_solution)

def calculate_cost(adjacency_matrix, node_group_A, node_group_B):
    # Get the number of connections between groups
    num_connections = 0
    for i in node_group_A:
        for j in node_group_B:
            if adjacency_matrix[i][j] != 0:
                num_connections += 1
```

```
    # Get uneven penalty
    uneven_cost = UNEVEN_PENALTY * abs(len(node_group_A) - len(node_group_B))

    return Cost(num_connections, uneven_cost)

def print_final_solution(solution: AlgorithmSolution):
    user_input = input("Do you want to print the final groups of components? (Y/N):
").strip().lower()
    if user_input == "y":
        solution.group_a.sort()
        solution.group_b.sort()
        group_a = list(map(lambda x: x + 1, solution.group_a))
        group_b = list(map(lambda x: x + 1, solution.group_b))
        print("Group A: {}".format(group_a))
        print("Group B: {}\n".format(group_b))
```