ECE 3790 Lab 2 Report – Winter 2019

Daniel Lovegrove (7763168)

INTRODUCTION

The point of Lab 2 is to create a simulated annealing based algorithm to split a graph into two minimally connected groups of components. This lab builds upon the naive approach from lab 1.

The general idea of simulated annealing for this lab is to get our solution out of a local minimum by somtimes taking solutions that are worse with some probability. The algorithm starts by creating two random sets of nodes. Then, one node is randomly swapped between the groups. If the new solution is better, we take it unconditionally as the new best, but if it is worse we take that solution with a probability depending on the delta cost value and on the current control parameter. The control parameter geometrically decreases throughout the runtime of the algorithm to allow the algorithm to converge to a solution. The lower the control parameter, the less chance that a worse solution will be selected as the new solution.

The language I chose to write this lab in is Python 3. All of the code I used is in Appendix B, preceded by the documentation for it in Appendix A.

LAB QUESTIONS

Q1: Algorithm Summary

The algorithm I wrote for lab 2 performed slightly better than lab 1. The simulated annealing algorithm consistently got a better lowest scores compared to lab 1 in my implementation, but sometimes this cost was barely better. The stopping criteria for my algorithm is when the number of consecutive solutions not taken reaches the size of the graph squared.

The pseudo-code for the algorithm is as follows:

```
set control parameter to average delta over (n^2)/2 iterations times some integer
randomly split graph into two equal groups
get cost for random solution

while (number of consecutive solutions not taken is less than max allowed):
      for (graph size * some integer iterations):
            randomly swap a single element between each group
            calculate cost of the current solution
            calculate delta (best cost – current cost)
            decide whether to take this solution based on delta
            if taking solution:
                  best solution = current solutions
                  set consecutive solutions not taken to zero
            else:
                  increment consecutive solutions not taken
      geometrically decrease control parameter after loop
```

Q2: Minimum Score for Test Cases

The following are the minimum scores for some of the text files given in UMLearn.

Graph Tested	Initial Cost	Final Cost	% Improvement
AdjMatSym2019.txt	3806	3334	12.4%
AdjMatCCSparce.txt	1909	1545	19.1%
AdjMatRand.txt	3733	3282	12.1%
AdjMatCCDense.txt	6193	5102	17.6%

(Table 1: Test Scores – Daniel Lovegrove)

Q3: Cooling Schedule

The following is the cooling schedule I used:

- if n is the size of the graph, iterate n * 2 times at the current kT value, then set kT = 0.92kT and iterate again.

I found that n * 2 iterations at a control parameter value had a good balance between running too long and not running long enough. The same goes for my choice of 0.92 as the geometric decrease factor. The algorithm consistently performed worse at 0.90 since it was getting less time to run, and the algorithm took too long to run when the value was 0.95 or above. As well, when the value was 0.95 or above, the algorithm did not produce consistently better results than a decreasing factor of 0.92.

Q4: How Long Did the Algorithm Take to Run?

Figure 1 below shows how varying the size of the problem affects the running time. I applied an exponential regression and polynomial regressions to the data using Wolfram Alpha and found that it had a very good fit for the exponential regression. Because of this, I believe that the algorithm runs in exponential time, with $O(e^n)$.

Q5: Varying Component Connectivity

Because my algorithm runs for a constant amount of iterations for a certain graph size and calculating the cost does not depend on the connectivity, varying the connectivity did not appreciably change the runtime of my algorithm.

Q6: Algorithm Improvements

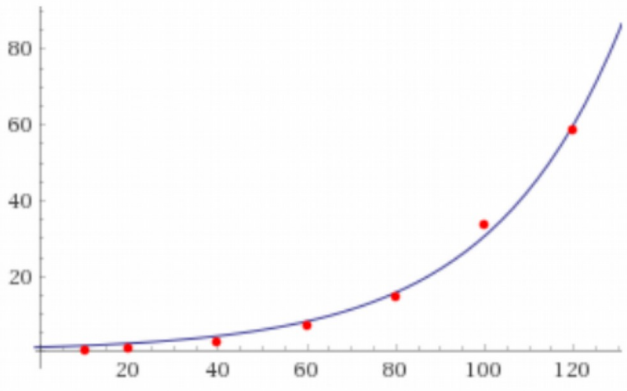
I think that my algorithm could be improved more if I didn’t iterate a constant amount at a specific control parameter, and actually checked if the solution was getting better or not.

TEST DATA	
Problem Size	Time to Complete (Avg. over 3 tries)
10	0.09 s
20	0.57s
40	2.64 s
60	6.91 s
80	14.54 s
100	33.23 s
120	38.42 s

Least-squares best fit:

$$1.02169\, e^{0.0338659\, x}$$

Plot of the least-squares fit:



Fit diagnostics:

AIC	BIC	R^2	adjusted R^2
32.5477	32.3855	0.996387	0.994941

(Figure 1: Exponential Regression – Daniel Lovegrove)

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`.

The libraries `plac` and `matplotlib` are required to be installed to run the code. These can be installed with `pip install plac`, and `pip install matplotlib`.

The two main files to run the code for the naive algorithm are `lab2_random_graph.py` and `lab2_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.

lab2_random_graph.py

Running `python3 lab_2_random_graph.py --help` from the command line gives information about how to call the script:

```
usage: lab2_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 `lab2_random_graph.py` on the command line for a graph with 100 nodes and 95% sparseness:

```
> python3 lab2_random_graph.py 100 95
```

lab2_graph_from_file.py

Running `python3 lab2_graph_from_file.py --help` gives information about how to call the script:

```
usage: lab2_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 `lab2_graph_from_file.py`, giving it the file `AdjMatCCSparce.txt` as input (this file is in the same folder as the script):

```
> python3 lab2_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: lab2_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_anneal
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: lab2_graph_from_file.py

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

```
import graphtools
import grouping_algorithm_anneal
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_anneal.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
import math
import cost_plotter

UNEVEN_PENALTY = 5

# Simulated annealing variables. These can be tweaked to change the performance of the algorithm
CONTROL_PARAM_DECREASE_RATE = 0.92
INIT_CONTROL_PARAM_MULTIPLIER = 2
GRAPH_SIZE_ITERATION_MULTIPLIER = 5

class Cost:
    """Defines the cost of a solution"""
    def __init__(self, connection_cost, uneven_cost):
        self.connection_cost = connection_cost
        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 simulated annealing to split matrix into two groups with minimal connections.')
    print('Press CTRL-C at any point to stop.\n')

    graph_size = len(adjacency_matrix)

    consecutive_solutions_unchanged_max = graph_size * graph_size

    (component_group_A, component_group_B) = get_initial_component_groups(graph_size)

    control_param = get_initial_control_parameter(adjacency_matrix, component_group_A,
component_group_B)
    initial_cost = calculate_cost(adjacency_matrix, component_group_A, component_group_B)

    best_solution = AlgorithmSolution(\
```

```
        copy.deepcopy(component_group_A),\
        copy.deepcopy(component_group_B),\
        initial_cost)

    solution_list = []
    iteration = 1
    consecutive_solutions_unchanged = 0
    new_solution = True
    start_time = time.time()

    while consecutive_solutions_unchanged < consecutive_solutions_unchanged_max:
        # Edge case: stop running when cost is zero
        if best_solution.cost.connection_cost == 0:
            break

        for i in range(0, graph_size * GRAPH_SIZE_ITERATION_MULTIPLIER):
            swap_one_item_between_groups(component_group_A, component_group_B)

            current_cost = calculate_cost(adjacency_matrix, component_group_A, component_group_B)

            # Simulated annealing
            delta = best_solution.cost.connection_cost - current_cost.connection_cost
            try:
                probability_to_take = math.exp(delta / control_param)
            except OverflowError:
                probability_to_take = 1.00
            random_num = random.random()
            take_solution = (random_num < probability_to_take)

            #print("kT: {:.4f}, DELTA: {}, PROB: {:.4f}, RAND: {:.4}, TAKE: {}".format(\
            #    control_param, delta, probability_to_take, random_num, take_solution))

            # Take better solution unconditionally
            # - OR -
            # Take worse solution with some probability
            if take_solution:
                solution_list.append(current_cost.connection_cost)
                new_solution = True
                consecutive_solutions_unchanged = 0
                best_solution = AlgorithmSolution(\
                    copy.deepcopy(component_group_A),\
                    copy.deepcopy(component_group_B),\
                    current_cost)

            else:
                new_solution = False
                consecutive_solutions_unchanged += 1

            if new_solution:
                print("Iteration {}".format(iteration))
                print("Cost due to unevenness: {}\nConnection cost:
{}\n".format(current_cost.uneven_cost, current_cost.connection_cost))
```

```
            iteration += 1
        # end for

        # Decrease control parameter geometrically
        control_param = control_param * CONTROL_PARAM_DECREASE_RATE

    # end while

    print("> Finished running after {} iterations.".format(iteration))
    print("> Finished in {:.5f} seconds".format(time.time() - start_time))
    print("> Initial cost: {}".format(initial_cost.connection_cost))
    print("> Final cost: {}\n".format(best_solution.cost.connection_cost))
    print_final_solution(best_solution)
    show_cost_plot(solution_list)

def get_initial_control_parameter(adjacency_matrix, node_group_A, node_group_B):
    initial_cost = calculate_cost(adjacency_matrix, node_group_A, node_group_B)
    number_iterations = int((len(node_group_A) **2) / 2) # Experimentally derived
    delta_cost_sum = 0

    node_group_A_copy = copy.deepcopy(node_group_A)
    node_group_B_copy = copy.deepcopy(node_group_B)

    for i in range(0, number_iterations):
        swap_one_item_between_groups(node_group_A_copy, node_group_B_copy)
        current_cost = calculate_cost(adjacency_matrix, node_group_A_copy, node_group_B_copy)
        delta_value = abs(initial_cost.connection_cost - current_cost.connection_cost)
        delta_cost_sum += delta_value

    return (INIT_CONTROL_PARAM_MULTIPLIER * (delta_cost_sum / number_iterations))

def swap_one_item_between_groups(node_group_A, node_group_B):
    indexA = random.choice(range(0, len(node_group_A)))
    indexB = random.choice(range(0, len(node_group_B)))
    temp = node_group_A[indexA]
    node_group_A[indexA] =node_group_B[indexB]
    node_group_B[indexB] = temp

def get_initial_component_groups(graph_size):
    # Shuffle component list
    all_components = list(range(0, graph_size))
    random.shuffle(all_components)

    # Put half of the components in one array, the other half in the other
    half_size = int(graph_size / 2)
    component_group_A = all_components[0: half_size]
    component_group_B = all_components[half_size:]

    return (component_group_A, component_group_B)

def calculate_cost(adjacency_matrix, node_group_A, node_group_B):
```

```
    # Get the number of connections between groups
    connection_cost = 0
    for i in node_group_A:
        for j in node_group_B:
            connection_cost += adjacency_matrix[i][j]

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

    return Cost(connection_cost, uneven_cost)

def print_final_solution(solution: AlgorithmSolution):
    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("> Final groups of nodes:")
    print("> Group A: {}".format(group_a))
    print("> Group B: {}\n".format(group_b))

def show_cost_plot(cost_list):
    user_input = input("> Do you want to plot the costs changing over time? (Y/N):
").strip().lower()
    if user_input == "y":
        cost_plotter.plot_costs(cost_list)
```

FILE: cost_plotter.py

Plots the changing solution over time.

```
import matplotlib
matplotlib.use('TKAgg')
import matplotlib.pyplot as plt

def plot_costs(cost_list):
   plt.plot(cost_list)
   plt.title('Change in Cost of Solution Over Time')
   plt.ylabel('Cost of solution')
   plt.xlabel('Solution number')
   plt.show()
```