Lab #1: Greedy Algorithms (Naïve anyway)

Student ID \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Section # \_\_\_\_\_\_\_\_

Marks \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Introduction

**Objective:** In this Lab you will become very familiar with implementing and experimenting with a greedy or naïve algorithm. From the course lectures and previous experiences you should be fairly/somewhat familiar with optimization. In this lab you will become familiar with some techniques that you can use to solve a combinatorial optimization problem. Combinatorial mean that there are many combination associated with an answer. The greedy process is not difficult and should also be somewhat familiar. People are inherently greedy so you basically already understand the decision process. The gist is, you make a change and see if it improves the solution if it does, great, keep the change, if it is not an improvement, reject the change. There are more sophisticated greedy strategies but this will do for our purposes.

In this lab a problem suitable for solution by a greedy algorithm is investigated. The problem is denoted the Component Allocation Problem. This is an allocation problem where the object is to allocate an equal numbers (or as close as equal as possible) of components into two racks of equipment minimizing the degree of interconnect between the two racks. Review the course notes for a detailed example albeit very small example. In this lab make sure you use a considerably larger problem. Be able to solve problems at least of 100 components. This problem is similar/identical to the problem for Lab 3 and 4 although the algorithms are different. This allows some degree of reuse as well as allowing for docking or comparison (verification). In addition, this problem is of historical significance as it was one of the first problems addressed by a non-deterministic optimization algorithm at IBM in the early 80s. I will include an adjacency matrix in umLearn that must be used as a benchmark or test case. This also facilitates comparison with other peoples’ solutions. Complete problem specification can be found in the course notes.

Problem Ref: Kirkpatrick, Scott, D. Gelatt Jr., and Mario P. Vecchi. "Optimization by simulated annealing." science 220, no. 4598 (1983): 671-680. Cited: 42000 times (That is a lot). In future labs you will solve the same problem using a Genetic Algorithm as well as by Simulated Annealing

**Preparation:** Overview the section in the notes covering the genetic algorithm section as it describes the problem in detail. Discussed below as well.

**Part 1 Objective:** The objective of part 1 is to write a utility function that you can use for the labs 1-3. The specification is as follows: Provide a utility that generates an adjacency matrix (text if you like) of integers which specify the connectivity between nodes on a graph.

Parameterize the matrix in terms of sparseness and size. Generate the edge weights as integers from 1-9 if an edge is present. The matrix should be symmetric.

Also generate a matrix that represents the case where 1/2 of the elements are connected to each other and the other 1/2 connected to each other. This will serve as a benchmark to see how well your algorithms performs for subsequent labs.

Generate a utility that allows you to read in an adjacency matrix. The format that I will provide for you will be text with spaces separating weights. So you may need to do some data formatting.

**Part 2 Objective:** The objective of part two is to implement a greedy algorithm to solve the component allocation problem. Use your generator as input of various size problems to estimate the run time versus problem size. Keep the matrix sparse as that is pretty reasonable assumption for many interconnection networks. For a sparse matrix the number of edges per node is (O(1)), that is, a constant.

For the component allocation problem the input is a list of components denoted Ci with connectivity represented by an adjacency matrix. The adjacency matric represents the degree of connectivity or edge cost associated with connecting component i to j. The links are bi-directional. The objective function is to minimize the cost of the interconnect between two groups of components while trying to maintain an equal number of components in each set. An additional cost that may be associated with an unequal number of components, could be x times the difference in the cardinality of the respective groups, if your algorithm considers roughly equal set partitions. That is, if x=5, if one group has 11 and the other 9, the difference is 2 and the increase in cost is 10. Adjacency matrices will also be provided in umLearn for testing your algorithm and seeing how well it does. Also generate and use a circular connection of nodes or a linear connection of nodes as a test cases (corner). Here is a trick of the trade for evaluating test cases. Initialize the problem with the known solution, then mix it up randomly, then run the algorithm and see how close you get to the optimal solution. This trick is also how you select from a set at random without replacement.

What would Bob do: (WWBD) I would configure a solution.

Half the nodes in one compartment, half in the other.

Calculate the cost of the solution.

Swap a pair of nodes.

Calculate the cost of the solution.

If improved, keep the new solution.

Repeat until tired (diminishing return).

**Note**: Your algorithm may not exactly be a greedy algorithm if it basically selects candidate component to swap, and swaps them if they reduce the crossover cost. Something closer and more famous (min-cut) employs a heuristic that would select candidate nodes more intelligently. For example, if nodes are selected that have more connections to nodes in the other compartment than their own, this would quite likely be better in terms of performance. This illustrates another general observation or trade-off: The simpler the algorithm, the easier its implementation the less optimal the outcome.

**Questions:**

1) Summarize your algorithm. Provide pseudo-code for your algorithm/s? Did it run as expected? What was the stopping criteria?

2) What was your minimum score or solution for the test cases? What was the percent improvement from and original or initial “solution”?

3) How long did the algorithm take to run? If you doubled the size of your problem did the running time scale linearly, quadratically or by some other means?

4) Vary the component population size. How did this affect the running time? Can you estimate a Big-O complexity?

5) How might your basic algorithm be improved?

6) To think about: What were you trying to minimize? What is the possible number of solutions? If the number of components were 100, what is the chance you could select the optimal solution at random?

This is sort of Bonus: Utility for “Visualization”

Write a utility that takes as input solution(s) and displays the result visually.

**Report**

Report Submission Instructions The lab report is due at the start of the next lab, and must be submitted to umLearn.

• Each student is required to write and hand in a separate well organized report.

• In the body of the lab report include answer each of the questions posed in the lab.

• Document all Java/Python Programs with at least comments in the program code.

• Lab write ups will only be accepted in PDF format. • Programs should be in plain text.

• As Appendices: • Code you have written, and documentation.

Zip your lab up if you like.