Homework #1: Hill Climbing for Partitioning and Exhaustive Search for Cycle Counting in Directed Graphs

Instructions:
The assignment consists of a written part (designated by W), and a programming assignment (designated by P), as well as exercises (designated by E). There is no need to turn in the exercise. Turn in the programming assignments (.cpp and .h files – make sure they compile). Turn in ONE .doc or .pdf or .txt for your written assignments. Make sure that we can use your code to reproduce any of your results in your written assignments as appropriate. Do not email assignments, unless instructed to do so.

All test cases are on the class web: www.cs.tufts.edu/comp/150GA/homeworks/hw1
To submit your code and your write-up, enter at the command prompt, “provide comp150ga  hw1 file1 file2 file3.. ”

Start early. None of these problems are ones that you can solve in one sitting. Also, some of the algorithm might take more than a few minutes to run. Plan ahead. Ask for help/clarification from the TA or the instructor if needed. For Problem #1, you may use C/C++ or Matlab. For problem #2, plan on using C/C++.

The problems presented here are real research problems that I am currently working on! So the discovery generated here is actually of great value!

Problem #1: Hill Climbing for Partitioning

A biochemical model for a fat cell is described by \( n \) independent reactions. These reactions are to be partitioned into two separate groups based on a property that measures their cyclical interactions. In short, this property is stored in a matrix B where if \( B_{ij} \) is positive, then reactions i and j should be placed in the same group and if \( B_{ij} \) is negative, the reactions should be placed in different groups. As such, the modularity metric \( Q \) is given by:

\[
Q = \sum_{i=1}^{n} \sum_{j=1}^{n} B_{ij} \cdot s_i \cdot s_j
\]

The idea is that \( s_i \) is an entry in vector \( S \) that takes values of -1 or 1, each representing the left or right grouping in the partition. If \( B_{ij} \) is positive, then we want either \( s_i \) and \( s_j \) to both take on a value of 1 or -1. This is because we want to reward putting reactions i and j in the same grouping by increasing our Q score because the product \( s_i \cdot s_j \) will be positive. By contrast, if \( B_{ij} \) is negative, then we want to reward this case by putting reactions i and j in different groups, which is done by assigning \( s_i \) and \( s_j \) opposite signs. However, it is obviously not possible to always increase Q because sometimes two reactions will have to be misplaced in order to better increase Q for another reaction pair. As such, the objective is to find a vector \( S \) that maximizes Q.

You will explore in this homework using hill climbing to partition the reactions into two sets. The optimal partition maximizes an objective function Q.

You will be given a set of examples to partition. Your job is to report the partitions and the maximum value of Q that can be attained for that partition.

Here is a concrete example:
For the vector $S = [-1, -1, -1, 1, 1, 1, -1, -1]$, the value for $Q = 43.2$

Feel free to consult the literature on portioning algorithms, and make sure that you cite your references. There are SO many of them!

**W1.** How many possible 2-way non-zero partitions exist? Your answer should be recorded in problem1.txt (or .pdf or .doc), with the W1 label.

**P2.** Implement a purely random algorithm to partition the reactions. Run your algorithm 100,000 different times. Report the maximum 100 values obtained over the 100,000 runs. Your code should have a function `generateRandomPartition` (int k, vector S), where k is the number of iterations that random will run for, and a function `evaluatePartition` (B, S). The code must be turned in files random.cpp and random.h. Your 100 top values should be recorded in problem1.txt (or .pdf or .doc), with the P2 label. Don’t worry about repeat random values. They are unlikely to occur.

**P3.** Implement another algorithm using any of the techniques we discussed in class, or any combination thereof. Feel free to use any method to improve your algorithm. Your code should re-use the function `evaluatePartition` from P1 and should have a function `improvePartition` (vector S). Your code should be turned in files partition.cpp and partition.h. Your algorithm probably is iterative in some way and calls the function “evaluatePartition” repeatedly. Write code to record the Q value after 100, 200, 300, … 1,000 calls to `evaluatePartition`, and provide the data so you can generate a histogram of that data. If it turns out that you benefit greatly after 1000 evaluations, please feel free to extend your histogram accordingly.

**W4.** Run your code on the test cases given in the homework directory. Generate histograms for each case.

**W5.** Short answers: (some of these answers are easy...)
   a. What is your objective function for P3
   b. What representation did you use for P3. Why?
   c. What was your evaluation function for P3?
   d. Which algorithm did you choose, and WHY?
   e. Explain how you fixed any of the parameters of the problem, if any.
   f. If you tried ANY other techniques, please state why it did or did not work. Feel free to try multiple possible algorithms.

The higher the Q – the better!
Problem #2: Exhaustive Search for Cycle Counting in Directed Graphs

Exhaustive search is, to say the least, exhausting!

There is nothing better than trying to reduce the amount of work to generate an exhaustive solution.

You are faced with the problem of enumerating cycles in a directed graph. Obviously there may be an exponential number of cycles. Examine the following graph courtesy of Wikipedia. Imagine that it is directed. There are lots of cycles. You really don’t want to count them by hand.

In this problem, you will learn the fastest current algorithm to count cycles!!

Also, since it takes so long to run it, we will attempt to use “randomness” to allow us to come up with a possible approximation. You will be given code to look at and examine to understand the algorithm, but all questions require a written answer.

E1. Let’s work with a simpler example:
The graph has two cycles: BCDE, BFDE.

Imagine using a depth first (DFS) traversal to discover the cycles.
A->B->C->D->E. we find a back edge. We declare a cycle. Recall that DFS marks nodes as “done/black” once you explore all their outgoing edges. After backtracking to B, nodes E, D, and C are already explored.

From B, DFS traverses to F. It then does not explore D because it is already colored black. The second cycle is never found!

Ahm.. DFS is \(O(V+E)\) – so clearly it would not explore all CYCLES. Clearly depth-first traversal will not yield the desired cycles either. It just takes so much more computation power to do so – a price must be paid somehow to discover each cycle.

Nothing is for free!

One way of dealing with this situation is to remove the “DONE” designation once a cycle gets discovered.

However, for a very large graph, it gets pretty complicated.

A clever man by the name of Donald Johnson proposed in 1975 a systematic way of counting cycles. His ideas are elegant. Here are the key things that his algorithm accomplishes:
1) Focus on counting all cycle originating from each node in the graph. That is, each node in the graph gets a turn being the starting node $s$.

2) Combining aspects of breadth/depth traversal. When the algorithm is at any one particular node $v$, and if ANY of $v$’s children result in a cycle back to $s$, then $v$ gets unmarked. Why? Because $v$ was a fruitful node that resulted in a cycle. If the traversal ever gets to $v$ again through another path, $v$ better be “available” for traversal to re-discover the path back to $s$.

3) To minimize all useless traversals, we keep record of all fruitless traversals. So if a node $v$ does not lead to the discovery of a cycle back to $s$, then we keep track of all the possible nodes that led us to $v$ by creating for each node a list of fruitless incoming edges. I emphasize here that if $v$ leads to the discovery of a cycle not including $s$, then this is considered an unfruitful traversal. The list of fruitless searches originating from a node $v$ only gets cleared when either starting a search from a new $s$, or when a cycle is discovered that involves $v$.

Here is an example that illustrates all these points.

From node 0, only the cycle 0→1→0 is discovered.

Before the search begins start from node 1, here is the fruitless list:
- Fruitless[0] = {}
- Fruitless[1] = {}
- Fruitless[2] = {5 6}
- Fruitless[3] = {2}
- Fruitless[4] = {3}
- Fruitless[5] = {7 4}
- Fruitless[6] = {4}
- Fruitless[7] = {6}

From node 1, only the cycle 1→0→1 is discovered.

…

Let’s consider the cycle counting starting from node 6.

6 gets blocked. Traversal begins and visits 2, 3, 4, 5, blocking each node as it traverses it. When 2 is reached again, it is already blocked. So Fruitless[2] = 5 indicates that node 2 was reached fruitlessly from node 5. The search backtracks to node 4, where the other children of 4 are explored. We reach node 6. We have found a cycle that involves 6 (6→2→3→4→6). At this point, every one of these nodes gets unblocked. Moreover, because Fruitless[2] = 5, then node 5 also gets unblocked.

When the traversal from 6 continues through edge 7, the cycle 6→7→5→2→6 is discovered.

If we did not unblock 5 based on 2’s fruitless list, then we would never discovered the cycle through 6 utilizing the edge 6→7!

What about the runtime? If there are “c” cycles in the graph, then the runtime will be $O((V+E)\cdot c)$. That’s more like it! We need to do work to count each cycle.
W1. Examine the code carefully (johnson.cpp in the homework directory on the web). Write pseudo-code summary of the algorithm. Feel free to look up the write up for the original algorithm (also in the hw directory). The pseudo code should be formatted similarly to the code in the algorithm textbook.

W2. You are given the network below, with $k=1$. Describe the state of the fruitless list at the termination of the traversal from node 5.

W3. For the network below, how many times will the cycle $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 1$ be found?

W4. Assume you have a magic aura that assigns each edge a weight that corresponds to the number of cycles in which each edge participates. Russell tells you that he can use this data to reduce the amount of traversals by Johnson’s algorithm needed to find all the cycles. Here is what he suggests:

Sort all the edges in the graph from smallest to largest participation number
Hide all edges in the graph.
For each edge (from smallest to largest):

Unhide the edge
Run DFS starting with the edge. If no back edges are found, continue.
Otherwise, run Johnson’s on the unhidden edges recording all found cycles.

a) Does this method find all cycles in the graph? Does it ever fail in discovering a cycle? Explain your answer or provide a counter example.

b) Is this algorithm runtime faster than Johnson’s? What is the runtime for this algorithm and how does it compare with Johnson’s run time?

c) Let’s say you are willing to tolerate a bit of “inaccuracy” here, but would like to finish your search before the next century. Would you recommend using this method over Johnson’s algorithm to find the number of cycles? Explain your answer.

![Network Diagram](image.png)

**Fig. 1. A worst-case example for Tarjan’s algorithm**