CPSC 320 2018W1: Assignment 2

September 29, 2018

Please submit this assignment via GradeScope at `https://gradescope.com`. Be sure to identify everyone in your group if you're making a group submission. (Reminder: groups can include a maximum of three students; we strongly encourage groups of two.)

Submit by the deadline **Saturday October 6, 2018 at 10PM**. For credit, your group must make a **single** submission via one group member's account, marking all other group members in that submission **using GradeScope's interface**. Your group's submission **must**:

- Be on time.
- Consist of a single, clearly legible file uploadable to GradeScope with clearly indicated solutions to the problems. (PDFs produced via LaTeX, Word, Google Docs, or other editing software work well. Scanned documents will likely work well. **High-quality** photographs are OK if we agree they're legible.)
- Include prominent numbering that corresponds to the numbering used in this assignment handout (not the individual quizzes). Put these **in order** starting each problem on a new page, ideally. If not, **very clearly** and prominently indicate which problem is answered where! When uploading assignments to gradescope, marks will be docked if pages are not properly assigned to each question.
- Include at the start of the document the **ugrad.cs.ubc.ca e-mail addresses** of each member of your team. (Please do **NOT** include your name on the assignment, however.[1])
- Include at the start of the document the statement: "All group members have read and followed the guidelines for academic conduct in CPSC 320. As part of those rules, when collaborating with anyone outside my group, (1) I and my collaborators took no record but names (and GradeScope information) away, and (2) after a suitable break, my group created the assignment I am submitting without help from anyone other than the course staff." (Go read those guidelines!)
- Include at the start of the document your outside-group collaborators' ugrad.cs.ubc.ca IDs, but **not** their names. (Be sure to get those IDs when you collaborate!)

Before we begin, a few notes on pseudocode throughout CPSC 320: Your pseudocode need not compile in any language, but it must communicate your algorithm clearly, concisely, correctly, and without irrelevant detail. Reasonable use of plain English is fine in such pseudocode. You should envision your audience as a capable CPSC 320 student unfamiliar with the problem you are solving. If you choose to use actual code, note that you may **neither** include what we consider to be irrelevant detail **nor** assume that we understand the particular language you chose. (So, for example, do not write `#include <iostream>` at the start of your pseudocode, and avoid idiosyncratic features of your language like Java's ternary (question-mark-colon) operator.)

[1] If you don't mind private information being stored outside Canada and want an extra double-check on your identity, include your student number rather than your name.

1 Feed Forward Loops

Gene regulatory networks in a cell involve many genes, with some genes (or rather the proteins encoded by such genes) affecting the level of expression of other genes. The expression level of a gene (which can vary over time) is the amount of the protein encoded by the gene that is present in the cell. Such networks can be represented by directed graphs: Nodes in the network represent genes, and there is a directed edge from gene g to gene g' (where g' may equal g) if the protein encoded by g affects the level of expression of the protein encoded by g'.

A **feed-forward loop** (FFL) in a gene regulatory graph has three distinct nodes, say a, b, and c, and has directed edges $(a, b), (b, c)$, and (a, c), but not the directed edges (b, a), (c, b) or (c, a), and no self-loops (a, a), (b, b) or (c, c). The actual feed-forward loops in the real network often ensure rapid, yet robust responses to changes in the cellular environment, and are abundant in regulatory networks of all kinds of organisms. Biologists are are interested in identifying FFLs (and other "motifs") in gene regulatory networks, and figuring out what is their function.

We'll define the *FFL detection problem* as follows: an instance of the problem is a directed graph $G = (V, E)$. The problem is to find all feed-forward loops in G.

1. Indicate which of the following (ignoring constants, i.e., using Θ notation) is the largest possible number of FFLs that a network with n nodes might have, as a function of n.

   If you are using LaTeX (please do!) just change `\fillinMCmath` to `\fillinMCmathsoln` for your choice of answer. This will make it much simpler for us to grade this question.

   ○ n ○ n^2 ○ n^3 ○ n^4 ○ 2^n ○ 3^n

2. Using pseudocode, describe a brute force algorithm that solves the FFL detection problem. Your pseudocode should be at a level of detail that ignores implementation decisions such as how the graph is represented.

3. Suppose that the input graph is represented by an adjacency matrix. Using Θ notation, give the running time of an efficient implementation of your algorithm in the worst case.

   ○ n ○ n^2 ○ n^3 ○ n^4 ○ 2^n ○ 3^n

4. [Challenge, not graded]: Suppose that the input graph is represented by adjacency lists. Using big-O notation, give the running time of your algorithm in the worst case.

   ○ n ○ n^2 ○ n^3 ○ n^4 ○ 2^n ○ 3^n

2 Kidney Exchange

Our description of this problem here is identical to that in Quiz 2. We recommend looking at the quiz problems (and, once you've tried them yourself, also look at the solutions) before working on this problem.

Kidney transplant is a life-saving procedure for people with kidney disease. Ideally, a kidney transplant patient identifies a living donor who can donate a kidney that is highly compatible with the patient, e.g., the patient and donor have the same blood type. The living donor will function just fine with their one remaining kidney, and hopefully the patient will be healthy with the donated kidney.

Often however, when a patient p finds a willing donor d, d may not be highly compatible with p. Given that there may be many such (donor, patient) pairs seeking transplants, it's natural to ask if the donors could be reallocated to the patients so that all patients are better off, i.e., are matched with more compatible donors. The goal of kidney exchange is find an alternative matching that is better. Here we'll formulate the kidney exchange problem in a simple way.

We'll define an instance of the KEP, the Kidney Exchange Problem, as follows. There is a set M of n initial (donor, patient) matches (d, p) (that is, M is a perfect matching of donors and patients). Let D be the set of n donors and P the set of n patients that are matched in M. Each patient p has a complete ranking $R[p]$ of donors; the donor d initially matched with p may not be at the top of p's list.

A problem instance is a tuple (n,M,R), where n is the number of patients (and also the number of donors), M is the initial perfect matching of patients and donors, and R is the collection of rankings R[p] for each of the n patients p.

For an instance I of KEP:

- The **best-preference** graph G_I is as follows: the nodes are pairs (d, p). There is a directed edge from (d, p) to (d', p') if d' is the most highly ranked donor on p's preference list.
- A **valid solution** S for I is any perfect matching of donors and recipients.
- An **instability** is a subset B of pairs of S such that a reallocation of donors of recipients in B would result in a solution S' in which all recipients are strictly better off than they are in S, that is, all are matched with a donor higher on their preference list.

Here is a small instance (3,M,R), where:

- M is the perfect matching d1:p1, d2:p2, d3:p3
- R is given by:

R[p1]:	r2,r3,r1
R[p2]:	r1,r2,r3
R[p3]:	r1,r3,r2

1. Show a valid solution with an instability for this instance, and write down what is the instability.

2. Trace through an execution of the following algorithm for KEP on the above instance, showing what M and S are just before the first iteration of the While loop, and also after each iteration of the `While` loop.

```
Algorithm Exchange ($I = (n$,M,R))
    Initialize $S$ to be the empty set
    While $n > 0$
        Create the best-preference graph $G_I$ for instance $I$
        Find a directed cycle in $G_I$, say $v_1, v_2, ..., v_k = v_1$
        Let $v_i$ be the pair $(d_i, p_i)$, for $1 \leq i < k$
        Update instance $I$ as follows:
            Remove the pairs $(d_i, p_i)$ from $M$, $1 \leq i < k$
            Remove $d_i$ from all patient rankings in R, $1 \leq i < k$
            Set $n$ to $n - k + 1$
        Add the pairs $(d_{i+1}, p_i)$ to $S$ for $1 \leq i < k - 1$, and also add the pair $(d_1, p_{k-1})$ to $S$
    Endwhile
    Return $S$
```

3. Explain why algorithm *Exchange* always produces a solution with no instability. Hint: Let $I = (n,\text{M},\text{R})$ be any instance and S the solution produced. Let B be any subset of S, i.e., a potential instability. Let P_B be the patients that are matched in B. Let V_i be the nodes in the directed cycle of iteration i. Let l be the first iteration in which $V_l \cap B \neq \emptyset$, with recipient $p \in P_B$ being allocated a donor in the iteration l. Explain why no reallocation among the donors in B can make p strictly better off.

3 Oh Graphs (again)

1. As seen in one of the Quiz 2 instances, the worst-case number of topological orderings of a connected, directed acyclic graph (DAG) G with n nodes is $(n-1)!$. Show how to construct DAGs that have this number of topological orderings.

2. [Challenge, not graded]: prove that $(n-1)!$ is the worst-case number of topological orderings of a DAG with n nodes.

3. Describe an $O(n)$ time algorithm to determine if a given graph $G = (V, E)$ is a star. A star has one central node that is adjacent (i.e., has an edge to) to all other nodes, and no other pair of nodes are adjacent. Assume that G is stored using an adjacency list data structure, and that the only way to determine the number of neighbours of a node N is to traverse its list of neighbours.

4. In the quiz, we assumed that each graph G was stored using an adjacency list data structure, and that the only way to determine the number of neighbours of a node N was to traverse its list of neighbours. For this question, suppose now that the list of neighbours for each node is stored in something like Java's `ArrayList`, whose length is available in $\Theta(1)$ time.

   What is the worst-case running time of the most efficient algorithm to solve each of the following problems for a given undirected graph $G = (V, E)$ with n nodes and m edges, under this new assumption?

   If you are using LaTeX (please do, **especially** for this question!a) just change `\fillinMCmath` to `\fillinMCmathsoln` for your choice of answer. This will make it *much *simpler for us to grade the question.

   - Given two nodes i and j of G, determine if they are adjacent.

     ○ n ○ nm
     ○ $n \log n$ ○ n^2m
     ○ n^2 ○ nm^2
     ○ $n + m$ ○ None of these

   - Determine if G is an n-clique (in an n-clique, all pairs of nodes are adjacent).

     ○ n ○ nm
     ○ $n \log n$ ○ n^2m
     ○ n^2 ○ nm^2
     ○ $n + m$ ○ None of these

   - Determine if G is a star.

     ○ n ○ nm
     ○ $n \log n$ ○ n^2m
     ○ n^2 ○ nm^2
     ○ $n + m$ ○ None of these

   - Find a node of G that is as close as possible to every other node of G. That is, you want to find a node N that minimizes $d(N) = \max\{y \in V, dist(N, y)\}$, where $dist(N, y)$ is the length of the shortest path (path with fewest edges) connecting N to y.

     ○ n ○ nm
     ○ $n \log n$ ○ n^2m
     ○ n^2 ○ nm^2
     ○ $n + m$ ○ None of these

5. Several fundamental algorithms for traversing or determining properties of directed and undirected graphs and trees are presented in Chapter 2 of the text, including depth first search (DFS), breadth first search (BFS), computing a topological ordering of nodes in a directed graph, testing if an undirected graph is connected, testing if a directed graph is strongly connected, finding connected components or strongly connected components, testing bipartiteness. Describe which of these algorithms would be most appropriate to solve the following problem and what its running time would be as a function of an appropriately chosen instance size.

   Related Sites: We would like to determine the n web sites that one is most likely to get to, starting from a given starting site.

   - Graph model of problem instance:
     - Nodes:
     - Edges:
   - Measure of instance size:
   - Good solution specification:
   - Well-known algorithm that can find a good solution:
   - Running time: