CPSC 320 2016W2: Assignment #2

January 23, 2017

Please submit this assignment via GradeScope at https://gradescope.com. Detailed instructions about how to do that are pinned to the top of our Piazza board: https://piazza.com/ubc.ca/winterterm22016/cpsc320/. Briefly, your GradeScope account **must** use the "GradeScope Student #" we distributed in your Connect gradebook so that we can link your account with you!

Submit by the deadline **Friday 3 Feb 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 quiz postings). Put these **in order** starting each problem on a new page, ideally. If not, **very clearly** and prominently indicate which problem is answered where!
- Include at the start of the document the **GradeScope Student #s** of each member of your team. (No names are necessary.)
- 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' GradeScope student #s or account names. (Be sure to get those when you collaborate!)

1 Cluedo Something to Me

The new game Clue II (CII, for short) is about a group of n people, each intent on stealing one of n objects. Each person moves about a mansion picking up and putting down objects but never holding the same object twice or holding any one object more than once. When they reach the object they want to steal, they pick it up and immediately flee the mansion with it, picking up no more objects.

Crucially, however, no one discovers anyone else has stolen anything. So, no one ever goes to pick up an object, only to find it missing.

As a player, you receive a list of clues, each of which is in one of two forms:

- "person" clues: person p_i was supposed to hold object o_j before object o_k (denoted (i, j, k) for short), and
- "object" clues: object o_i was supposed to be held by person p_j before person p_k (also denoted (i, j, k) for short)

(The clues describe what happens up to the point where a person steals an item; from that point on, the person picks up no more items, no matter what the clues say.)

So, an instance of the game is a value n, a list of person clues (i, j, k), and a list of object clues (i, j, k). No clue is repeated. No set of clues for a particular person (or a particular object) is inconsistent in the sense that it demands (directly or indirectly via a chain of clues) that person p_i hold o_j both before and after o_k (and similarly for object clues).

Your goal is to propose an item for each person to steal so that the thefts could have happened without anyone discovering that any other item was stolen.

For example, with the object clue "the candle was supposed to be held by Squirrel Girl before Koi Boy" and the person clue "Koi Boy was supposed to hold the candle before the pipe", it could **not** be the case that Squirrel Girl stole the candle and Koi Boy stole the pipe because:

- before Koi Boy can pick up and steal the pipe, Koi Boy must pick up the candle, and
- before Koi Boy can pick up the candle, Squirrel Girl (who held it first) must have already stolen it,
- but then Koi Boy would discover the theft of the candle.

You may use the topological sort algorithm (topo-sort) as needed in this problem. Topological sort takes a directed graph with no cycles (a directed, acyclic graph) and produces a left-to-right ordering of the vertices in the graph such that all edges point to the right. It runs in $O(m + n)$ time (where $m = |E|$ and $n = |V|$).

1.1 Instealbility

Here's a slight variant on the problem, which we'll call C3. Instead of receiving individual clues like "p_1 was supposed to hold o_1 before o_2", you receive a full list of the order that people are supposed to hold items and items are supposed to be held.

For example, with three people/items, you might be told for p_1 that "p_1 was supposed to hold o_1 before o_3 and o_3 before o_2" and for o_2 that "o_2 was supposed to be held by p_3 before p_2 and by p_2 before p_1".

So, a C3 instance is n, a planned ordering of objects for each person, and then a planned ordering of people for each object.

Give a correct reduction from C3 to SMP. Note that this means you have a correct solver for SMP and you use it as part of your reduction to solve C3.

FOR THE ASSIGNMENT ONLY: Sketch the key points in a proof that your reduction is correct. (That is, clearly explain why only an instability in SMP instance's solution could lead to the discovery of a theft in C3 instance/solution.)

1.2 Stealability

Here's a slight variant on the problem, which we'll call C3. Instead of receiving individual clues like "p_1 was supposed to hold o_1 before o_2", you receive a full list of the order that people are supposed to hold items and items are supposed to be held.

For example, with three people/items, you might be told for p_1 that "p_1 was supposed to hold o_1 before o_3 and o_3 before o_2" and for o_2 that "o_2 was supposed to be held by p_3 before p_2 and by p_2 before p_1".

So, a C3 instance is n, a planned ordering of objects for each person, and then a planned ordering of people for each object.

Give a correct reduction from CII to C3. Note that this means you have a correct solver for C3 and you use it as part of your reduction to solve CII.

FOR THE ASSIGNMENT ONLY: Sketch the key points in a proof that your reduction is correct. (That is, if no one discovers a theft in your C3 instance/solution, then no one will discover a theft in the CII instance/solution.)

2 Labour of Love

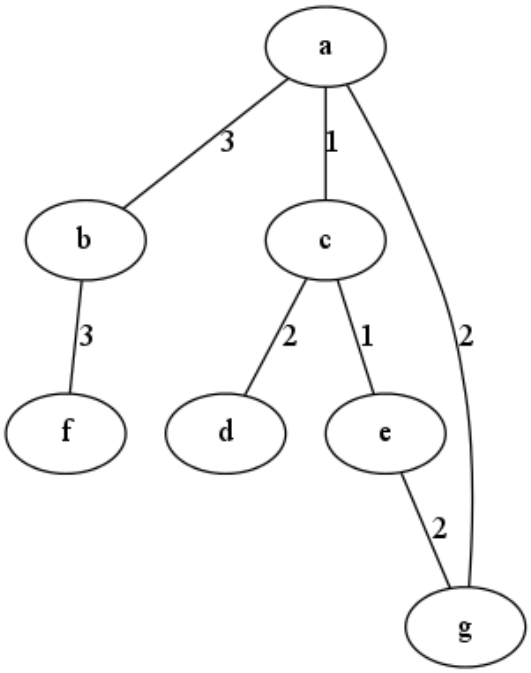
We define an undirected, positive-integer-weighted graph to be $G = (V, E, W)$, where V is a set of vertices, E is a set of edges (u, v) where $u, v \in V$ and we consider the tuples to be unordered, and W is a function that maps edges (u, v) to their positive, integer weights w. (The graph may not have self-loops.)

We define the result of contracting an edge (u, v) in such a graph to be the new graph $G' = (V - \{v\}, E', W')$. E' contains all "uninvolved" edges (s, t), those where neither s nor t is u or v. The weights of these edges under W' are unchanged from their weights under W. Furthermore, E' contains an edge (t, u) (for t not equal to u or v) exactly when E contains an edge (t, u) or (t, v) or both. If E only has (t, u) and not (t, v) or if E only has (t, v) and not (t, u), then the weight of (t, u) under W' is the same as that one corresponding E edge's weight under W. If E contains both (t, u) and (t, v), then the weight of (t, u) under W' is the sum of the weights of (t, u) and (t, v) under W.

In other words, when we contract an edge, we merge the two vertices on either side of the edge into one, remove the edge that connected them, and "sum" pairs of edges that used to lead from one other vertex to each of the now-merged vertices into a single edge with the total weight of the two old edges.

2.1 Treeting a Graph With Kindness

Consider the following graph:



Sketch the result of performing the following two contractions one after another: (a, c) (removing c) and then (a, e) (removing e). The intermediate graph is not required... but **is** critical for partial credit!

2.2 Death of a Thousand Min-Cuts

Consider the following algorithm:

```
FindCut(G):                                  // where G = (V, E, W)
  if |E| = 0:
    return 0
  while |E| > 1:
    pick an arbitrary edge (u, v) in E
    contract (u, v) in G                     // i.e., update G so (u, v) has been contracted

  let (u, v) be the remaining edge in E
  return W(u, v)
```

Answer the following questions in terms of $n = |V|$ and $m = |E|$. (Reminder: a connected graph is one where every vertex is reachable from every other vertex.)

1. What is the **maximum** number of iterations the loop can run **in a connected graph**? Briefly but clearly and completely justify your argument. This should **NOT** be an asymptotic bound.

2. What is the **minimum** number of iterations the loop can run **in a connected graph**? Briefly but clearly and completely justify your argument. This should **NOT** be an asymptotic bound.

3. **FOR THE ASSIGNMENT ONLY:** A cut in a graph is a partitioning of the vertices of the graph into two non-empty subsets (such that no vertex is in both sets and every vertex is in one set). The minimum cut in a graph is the cut that minimizes the total weight of the edges crossing from one set of the cut to the other. How does the result of this function relate to the minimum cut's value? Briefly but clearly and completely justify your answer.

2.3 How Far Apart Are They?

Consider a graph $G = (V, E, W)$ represented as an adjacency list L that also stores edge weights. Specifically, the vertices in V are numbers $\{1, 2, \ldots, |V|\}$ that are used as one-based indexes into the array L, and each node of the doubly-linked lists stored in L contains both the index of the vertex on the other side of the edge and the weight of that edge (and, of course, pointers to the next and previous nodes in the linked list). Each edge (u, v) with weight w is represented twice: once in $L[u]$ (by a node in the list containing v and w) and once in $L[v]$ (by a node in the list containing u and w).

Consider the following algorithm to perform an edge contraction with such a graph representation. Note that unlike in the definition of a contraction above, when this function contracts (u, v), v is **not** removed from V; however, the graph no longer contains any edge to or from v.

```
Contract(G = (V, E, W), (u, v)):
    // Precondition: (u, v) is in E and so u and v are each in V
    //               G is represented as described above as a list L
    For each pair (s, w) of a vertex index and weight in L[v]:
        If s is not equal to u:
            Find a node n in L[u] with vertex index s
            If such a node n exists, add w to the weight of node n
            Otherwise:
                Add to the front of L[u] a new linked list node n' = (s, w)
        Find a node n'' in L[s] with vertex index v (which must exist)
        Delete n'' from L[s]
    Delete all entries from L[v]
```

Give a good worst-case asymptotic bound on the runtime of this algorithm in terms of $n = |V|$ and $m = |E|$. Briefly justify your bound, including annotating **all** lines in the body of the function above to make clear their contribution to the runtime.

3 The Biggest Slice of Π

You have been given an array of daily gains G for a stock over the course of n days. A gain of 1.35 for a day, for example, means that \$1.00 invested in the stock at the start of the day would be worth \$1.35 at the end of the day (and the start of the next day). A "gain" of 0.5 would mean that \$1.00 invested at the start of the day would be worth only \$0.50 at the end of the day. All gains are positive numbers. You are reviewing potential earnings from investments in this stock.

3.1 Smashing the Π

Imagine a scenario in which you invest a sum of money in the stock at the start of one day i and pull all the money back out (sell) at the start of another day j. The overall gain for your money would be $\prod_{k=i}^{j-1} G[k] = G[i] * G[i+1] * \cdots * G[j-2] * G[j-1]$.

Write pseudocode for and **analyse the worst-case asymptotic runtime of** a brute force algorithm to find the highest overall gain for any pair of days i and j, where $1 \leq i \leq n$ and $i \leq j \leq n+1$. (If $i = j$, then you put your money in and take it straight back out, with no change in value. If $j = n+1$, you leave your money in up to and including the last day and pull it out at the end of that day, which is also the start of day $n+1$.)

You should assume that multiplication takes constant time, regardless of how big the result becomes.

4 Access Allowed

Based on accessibility guidelines, an institution has classified all of its campus's pathways connecting points of interest into three groups, those that are: at recommended specifications (R), at minimum specifications (M), and below minimum specifications (X). For each pathway that is rated M or X, you also have a cost to upgrade to the higher specification(s). Occasionally, that cost is indicated at ∞ where it's considered impossible to perform the upgrade.

Note that a "pathway" may take any of various forms (e.g., a flight of steps), but that it will always connect exactly two points of interest. Furthermore, although two points of interest may be physically connected "in the real world" by multiple pathways (e.g., a flight of stairs and an elevator), the "logical" pathway will appear only once in the data rated according to the most accessible of the physical pathways connecting the two points.

4.1 A Little Help from my Friends

Ideally, every point on the campus should be reachable from every other point using only pathways rated at R. However, the institution wants to determine whether all areas are reachable with a "single point of assistance". That is, can every point on campus be reached from every other point using at most one pathway between neighbouring points that is rated below the recommended specifications (i.e., at M or X)?

1. Give a small example of a graph that **does** pass this test even though it has four points of interest with the property that no one of the four points can be reached from any other of the points without using at least one pathway rated M or X. Clearly label each edge with a R, M, or X.

2. Consider the function `DFSFromStackROnly` which implements standard Depth-First Search, except that it begins from a provided stack that may contain many vertices already:

```
DFSFromStackROnly(G, S):
    while S is not empty:
        pop the first vertex v off of S
        if not v.visited:
            set v.visited to true
            // Remember that (v, u) and (u, v) are the same edge
            for each edge (v, u) in G's edges:
                if (v, u) has the label R and u.visited is false:
                    push u onto S
```

   We assume each vertex has a Boolean `visited` field that is initially set to false before your algorithm runs. Manipulate it as you need in your algorithm (below). `DFSFromStackROnly` also uses this field.[1]

   Give clear pseudocode for a correct and efficient—$O(m+n)$—algorithm that takes an accessibility graph and determines whether all vertices in the graph are reachable with a "single point of assistance". Call `DFSFromStackROnly` as many times as you need (but stay within the $O(m+n)$ bound!).

[1] We could use a hash table mapping vertices to Booleans to simulate the `visited` field's presence if it didn't already exist.

4.2 A Bridge Too Far

Ideally, every point on the campus should be reachable from every other point using only pathways rated at R. However, the institution wants to determine whether all areas are reachable with a "single point of assistance". That is, can every point on campus be reached from every other point using at most one pathway between neighbouring points that is rated below the recommended specifications (i.e., at M or X)?

Consider the following algorithm, which assumes each vertex has extra fields `visited` (a Boolean that is initially `false`) and `steps` (a non-negative integer that is initially `0`):

```
BFSWithSteps(G):
    if G has no vertices, return

    let Q be an empty queue (where each entry is a pair of a vertex and an integer)
    let v be an arbitrary vertex in G
    enqueue (v, 0) into Q
    while Q is not empty:
        dequeue the first pair (v, s) off of Q
        if v.visited is false:
            set v.visited to true
            set v.steps to s

            // Remember that (v, u) and (u, v) are the same edge
            for each edge (v, u) in the graph:
                if (v, u) has the label R:
                    enqueue (u, s) into Q
                else:
                    enqueue (u, s+1) into Q
```

This is like BFS except that it keeps track of how many assisted steps it took along the path to a particular point. The intent of the algorithm is that we would run it and report that all areas are reachable with a single point of assistance if every vertex has a `steps` of 0 or 1.

1. Give an example of a graph (with each edge clearly labeled as R, M, or X) for which the algorithm labels every vertex with `steps` of 0 or 1, yet it is **NOT** the case that all areas are reachable using only a single point of assistance. **Label each vertex** with the `steps` value the algorithm assigns to it.

   NOTES: you should choose (and clearly indicate!) which vertex the algorithm enqueues first. Your graph **must** be quite small for credit.

CONTINUE TO THE SECOND PROBLEM ON THE NEXT PAGE.

2. Give an example of a graph (with each edge clearly labeled as R, M, or X) for which the algorithm labels at least one vertex with `steps` of 2 or larger, yet all areas **are** reachable using only a single point of assistance. **Label each vertex** with the `steps` value the algorithm assigns to it.

   NOTES: you should choose (and clearly indicate!) which vertex the algorithm enqueues first. Your graph **must** be quite small for credit.