First Midterm Solutions

Name:

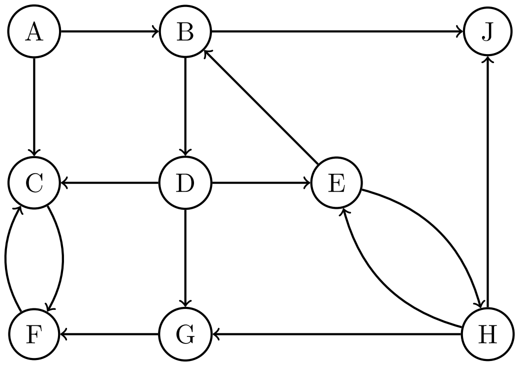
SID:

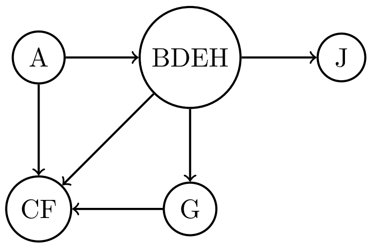
GSI and section time:

Answer all questions. Read them carefully first. Be precise and concise. The number of points indicate the amount of time (in minutes) each problem is worth spending. Not all parts of a problem are weighted equally. Write in the space provided, and use the back of the page for scratch. Box numerical final answers. Good luck!

When an explanation is required, answer with one or two short sentences. *(20 points)*

1. For the directed graph below, find the strongly connected components and draw the DAG of strongly connected components.



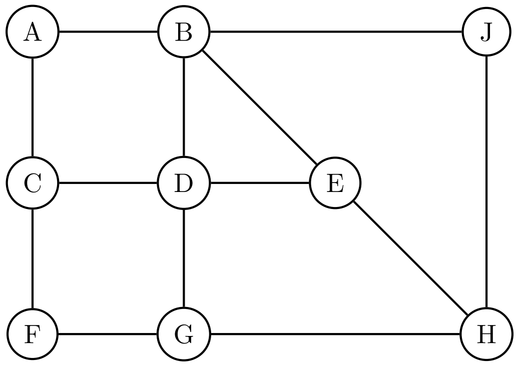


The strongly connected components are $\{A\}, \{B, D, E, H\}, \{C, F\}, \{G\}, \{J\}$, and the DAG is in the above right graph.

Common Mistakes:

- Many students forgot the edge between $\{B, D, E, H\}$ and $\{C, F\}$.
- While nothing is wrong with drawing the graph linearized, this was not necessary, and made it harder to read and grade :(
- The DAG of the SCCs is not a multi-edge graph. For example, there should not be two edges between $\{B, D, E, H\}$ and $\{G\}$.
- A strongly connected component may be a single vertex, yes.

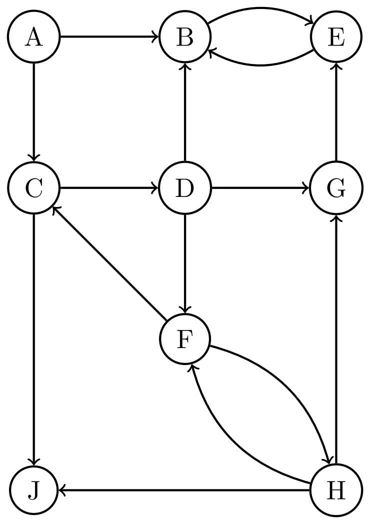
2. Execute DFS on the following undirected graph starting at node D breaking ties alphabetically. Mark the pre and post values of the nodes.

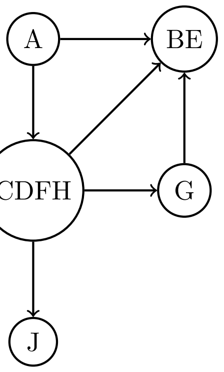


Node	pre	post
A	3	16
B	2	17
C	4	15
D	1	18
E	8	9
F	5	14
G	6	13
H	7	12
J	10	11

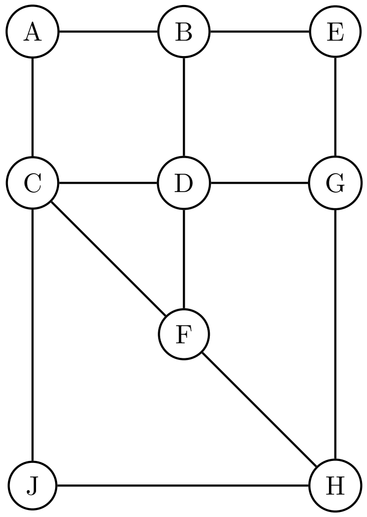
When an explanation is required, answer with one or two short sentences. *(20 points)*

1. For the directed graph below, find the strongly connected components and draw the DAG of strongly connected components.



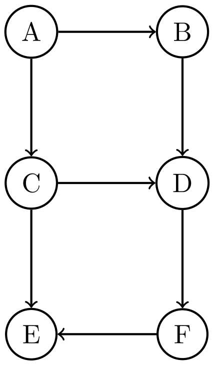


2. Execute DFS on the following undirected graph starting at node D breaking ties alphabetically. Mark the pre and post values of the nodes.



Node	pre	post
A	3	16
B	2	17
C	4	15
D	1	18
E	8	9
F	5	14
G	7	10
H	6	13
J	11	12

3. In an implementation of Bellman-Ford, starting with the initialization $dist(A) = 0, dist(B) = dist(C) = dist(D) = dist(E) = dist(F) = \infty$, the following sequence of updates are applied on the graph shown below.



1) $update(A \to C)$
2) $update(B \to D)$
3) $update(A \to B)$
4) $update(C \to D)$
5) $update(F \to E)$
6) $update(C \to E)$
7) $update(D \to F)$
8) $update(B \to D)$
9) $update(D \to F)$
10) $update(F \to E)$
11) $update(C \to D)$
12) $update(A \to C)$
13) $update(A \to B)$
14) $update(C \to E)$
15) $update(D \to F)$
16) $update(B \to D)$
17) $update(A \to B)$
18) $update(C \to D)$
19) $update(F \to E)$
20) $update(C \to E)$

What is the earliest step at which the distance to F guaranteed to be correct, for all possible weights on the edges? Justify your answer.

Solution: $ACDF$ and $ABDF$ are the only two paths from A to F, and both of them appear as subsequences of the updates by step 9. Hence, the distance to F is guaranteed to be correct by step 9.

More specifically, the distance to C is guaranteed by step 1, the distance to B is guaranteed by step 3, the distance to D is guaranteed by step $\max(4, 8) = 8$, and the distance to F is guaranteed by step 9.

4. Describe the naive algorithm for Fourier transform. What is its running time?
(Briefly and precisely describe the algorithm, no need to prove the correctness)

`Input:` $a_0, \ldots, a_{n-1} \in \mathbb{R}$
`Output: the Fourier transform` $b_0, \ldots, b_{n-1}$ `using` ω `the` n^{th} `root of unity`

Fourier transform involves evaluating the polynomial $p(x) = a_0 + a_1 \cdot x + a_2 \cdot x^2 + \cdots + a_{n-1} \cdot x^{n-1}$ at the n-th roots of unity, $1, \omega, \omega^2, \ldots, \omega^{n-1}$, and giving as output, $b_0 = p(1), b_1 = p(\omega), b_2 = p(\omega^2), \ldots, b_{n-1} = p(\omega^{n-1})$.
In this class we devised the fast Fourier transform algorithm in order to do these computations efficiently, **but** the question was asking for the naive way of doing these computations. The naive Fourier transform naively evaluates $p(x)$ at the n-th roots of unity. Based on how naively we calculate $\omega^{i \cdot j}$ for $i, j \in \{0, 1, 2, \ldots, n-1\}$, this computation can take from $O(n^2)$ up to $O(n^4)$ time. All correctly explained answers in this range were given full credit.
Note that we can give a compact representation of the evaluation of the polynomial at the n-th roots of unity, using linear algebra notation, i.e.

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ \vdots \\ b_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \ldots & 1 \\ 1 & \omega & \omega^2 & \ldots & \omega^{n-1} \\ 1 & \omega^2 & (\omega^2)^2 & \ldots & (\omega^2)^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{n-1} & (\omega^{n-1})^2 & \ldots & (\omega^{n-1})^{n-1} \end{bmatrix} \cdot \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

Using this representation the naive Fourier tranform involves constructing the $n \times n$ matrix which can take from $O(n^2)$ up to $O(n^4)$ time (based on how naive the construction is), and then multiplying the matrix with the vector which takes $O(n^2)$ time. Correct solutions interpreting the matrix-vector multiplication as a matrix-matrix multiplication and so deriving that $O(n^3)$ time is needed for this operation was given full credit as well.

Find the bug (10 points)

5. Are these algorithms and/or their proofs correct? Justify your answers (If the algorithm is correct, justify why and if the algorithm is incorrect, either give a counterexample or justify why).

   (a) **Divide and Conquer Algorithm for MST** `MST(`G`: graph on `n` vertices)`

   ```
   • $T_1 \leftarrow$ MST($G_1$: subgraph of $G$ induced on vertices $\{1,\dots,n/2\}$)
   • $T_2 \leftarrow$ MST($G_2$: subgraph of $G$ induced on vertices $\{n/2+1,\dots,n\}$)
   • $e \leftarrow$ cheapest edge across the cut $\{1,\dots,\frac{n}{2}\}$ and $\{\frac{n}{2}+1,\dots,n\}$.
   • return $T_1 \cup T_2 \cup \{e\}$.
   ```

   Proof of correctness *By the cut property, the cheapest edge e across the cut* $\{1,\dots,\frac{n}{2}\}$ *and* $\{\frac{n}{2}+1,\dots,n\}$ *belongs to an MST* T*. On removing the edge e from* T*, the resulting subtrees must be the minimum spanning trees connecting* $\{1,\dots,\frac{n}{2}\}$ *and* $\{\frac{n}{2}+1,\dots,n\}$.

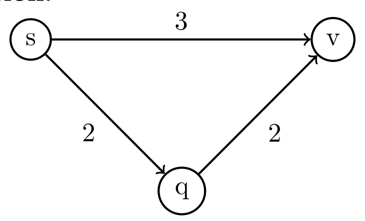
   Solution:
   This algorithm does not work; multiple edges of the MST could cross this particular cut. Another way to see this is that the MSTs of the subgraph needn't also be part of the MST of the whole graph.
   As a concrete counterexample, consider a wide rectangle and the horizontal cut between the top two vertices and the bottom two. Both edges on this cut should be in the MST.

   (b) **Greedy DFS for shortest paths**

   ```
   Input: Graph $G = (V,E)$,a starting vertex $s$ and non-negative lengths $\ell_e$ for each
   edge $e \in E$.
   Goal: Compute shortest path to a vertex $v$

   Run DFS, but at each node explore the shortest outgoing edge first until $v$ is reached.
   Return the $s$ to $v$ path in the DFS tree.
   ```

   Solution:



   As a counterexample, consider the above graph. The greedy DFS will output the path $s-q-v$ rather than the shortest path, $s-v$. This illustrates the intuition for why this algorithm fails: the shortest path between two vertices need not make use of the shortest individual edges.

True or false? Circle the right answer. No explanation needed (15 points)

1) **T** **F** The solution of the recurrence $T(n) = 3T(n/3) + O(n^3)$ is $T(n) = O(n^3)$.

Solution: T; this is a straightforward application of the master theorem.

2) **T** **F** By starting with number 3 and repeatedly squaring it 1000 times, we can compute $3^{2^{1000}}$ within a day on a laptop.

Solution: F; the final number will be around 2^{1000} digits long, and thus dwarf the world's available storage space.

3) **T** **F** Given a polynomial of degree $2^n - 1$, the FFT works by recursively computing 2^n points of two polynomials of degree $2^{n-1} - 1$ and then combining the results.

Solution: F; the power of the FFT lies in that it only needs 2^{n-1} points of each of the lower-degree polynomials to compute 2^n points of the bigger polynomial.

4) **T** **F** In a graph, if one raises the lengths of all edges to the power 3, the minimum spanning tree will stay the same.

Solution: T; the MST algorithms care about *relative* edge lengths, and raising all edge lengths the 3rd power preserves this relationship.

5) **T** **F** $FFT(1, 2, 3, 4) + FFT(-1, -2, -3, -4) = [0, 0, 0, 0]$

Solution: T; FFT is linear.

6) **T** **F** If $a^{n-1} \equiv 1 \pmod{n}$ for some positive integers $a < n$, then n is a prime.

Solution: F; $a = 1$ is the obvious counter-example.

7) **T** **F** The randomized algorithm to find the median is always faster than running mergesort to find the median.

Solution: F; the median-finding algorithm has worst-case running time $O(n^2)$.

8) **T** **F** The first edge added by Kruskal's algorithm can be the last edge added by Prim's algorithm.

Solution: T; the graph $d(A, B) = 1$, $d(B, C) = 2$, and we start running Prim's at C.

9) **T** **F** $\log^*(2^n) = 2\log^* n$

Solution: F; By definition, $\log^*(2^n) = 1 + \log^*(\log 2^n) = 1 + \log^* n$.

10) **T** **F** The family consisting of all possible functions $f : \{1, \ldots, 2^{16}\} \to \{1, \ldots, 256\}$ is a universal hash family.

Solution: T; Consider distinct x, y in the domain, and let $f(x) = q$. Exactly 1/256 of all possible such functions f will satisfy $f(y) = q$, because for any mapping of the other $2^{16} - 2$ elements in the domain, exactly one of the 256 functions will map y to q.

11) **T** **F** If $\mathcal{H} : \mathbb{Z} \to \{1, \ldots, 170\}$ is a universal hash family of functions, then for every pair of distinct keys x, y there is some function $f \in \mathcal{H}$ such that $f(x) \neq f(y)$.

Solution: T; we have $P(f(x) = f(y)) = \frac{1}{170}$, so there must be such an f.

12) **T** **F** If all edge weights in a graph are either 1 or 2, then the shortest path can be computed in $O(|V| + |E|)$ time.

Solution: T; split edges of length 2 in two by adding a dummy vertex, and run BFS.

13) **T** **F** The heaviest edge in a graph cannot belong to the minimum spanning tree.

Solution: F; this edge may be connecting two otherwise-disconnected subgraphs.

14) **T** **F** The maximum spanning tree (spanning tree of maximum cost) can be computed by negating the cost of all the edges in the graph and then computing minimum spanning tree.

Solution: T; this works, and none of the proofs of our MST algorithms depended on edge weights being nonnegative.

15) **T** **F** The longest path in a graph can be computed by negating the cost of all the edges in the graph and then running Bellman-Ford.

Solution: F; this may introduce negative cycles.

Semi-Connected Graphs (15 points)

6. A directed graph $G = (V, E)$ is semi-connected if for every pair of vertices u, v either there is a path from u to v or there is a path from v to u or both.

   (a) Give an example of a DAG that is not semi-connected.

   Solution: A disconnected DAG:

   (b) State a necessary and sufficient condition for a DAG to be semi-connected.

   Solution: There is a single path that goes through all vertices (i.e. there is a Hamiltonian path); this is equivalent to saying that when linearized, there is an edge between every consecutive pair of vertices. A third equivalent statement is that there is exactly one linearization of the DAG; however, this formulation is harder to base a correct algorithm for in part (c) with.

   We now prove that the first formulation is a necessary and sufficient condition for semi-connectedness. (The corresponding proofs for the other formulations are similar, and will be left as an exercise to the reader.)
   When there are k vertices, let the linearized order be $v_1, v_2, \ldots, v_k$.
   We first show that this is a necessary condition. For any i: there cannot be a path from v_{i+1} to v_i, and the only possible path from v_i to v_{i+1} is a direct edge between then, so this edge must be present in order for the graph to be semi-connected. Since there are edges between all consecutive pairs of vertices in the linearized order, there is a path through all vertices. Thus, this is a necessary condition.
   We now show that this is a sufficient condition. If there is a path that goes through all vertices, there is a path between any pair of vertices v_i, v_j by simply following the relevant part of this global path.

   Common Mistakes:

   - Some students interpreted the problem incorrectly, and confused "path" with "edge".
   - Some students may have confused "semi-connected" and "weakly-connected" graphs.
   - Many students said that the condition was that the DAG had exactly one single source or exactly one sink. While this is a necessary condition, it is not a sufficient one: imagine a graph with the edges $A \to B$, $A \to C$, $B \to D$, $C \to D$. There is neither a path from B to C nor from C to B.
   - Some students, either in this part or in part (c), suggested that each explore call was equivalent to finding a single weakly connected component. This is not necessarily the case, as can be seen in the following graph: $A \to B$, where we start DFS from B first, and would then have to make a separate explore call from A.

(c) Given a DAG, exhibit an algorithm to check if it is semi-connected. (Formal pseudocode is unnecessary, briefly but precisely describe the algorithm and argue its correctness)

Solution 1:
As per part (b), we linearize the graph and check if there is a pair of vertices between every pair of consecutive vertices. The proof of why this works is already explained in part (b).
Solution 2:
There is a path that goes through all vertices in a DAG iff the longest path in the DAG is of length $|V| - 1$. This condition could be checked via linearization and then iterating through the vertices in order, to find the longest path. The proof of correctness of this algorithm is also in part (b).
Solution 3:
An alternate solution is to check if there is exactly one source in the graph; if there is not, then the graph is not semi-connected, and if there is, then the algorithm deletes the vertex and recurses on the remaining vertices.
If a graph has more than one source, then as neither source can have a path to each other, the graph cannot be semi-connected. If a DAG has exactly one source, that source must be an ancestor of and thus be able to reach every other vertex. In this latter case, we thus only need to check if the subgraph excluding the source vertex is semi-connected.

Common Mistakes:

- First, the proof of correctness sought required relating the algorithm, and likely the condition proposed in part (b), to the definition of semi-connectedness. Relating the algorithm to just the condition in part (b) was insufficient (unless the correctness of that condition was proven.)
- In addition, for correctness, it was important to argue both that your algorithm rejects non-semi-connected graphs *as well as* finds the semi-connected graphs, i.e. proving that the algorithm is correct for both types of graphs. Some students only proved one (i.e. algorithm returns true implies graph is semi-connected, or graph is semi-connected implies algorithm returns true, but not both).
- Some students tried to find all possible linearizations of the graph, or count the number of such linearizations. We have not discussed any algorithm to do so, and this is potentially a very inefficient process: the DAG with n vertices and 0 edges has $n!$ linearizations. Similarly, multiple calls to DFS may find the same linearization, even if started on a different vertex, so using DFS is not a reliable way of determining if there is only one linearization.
- Some students checked that there was exactly one source and one sink, and then ran DFS from the source to see if every vertex was reached on the way to the sink, in an attempt to follow solution 2. However, DFS does not guarantee that we explore the vertices in linearized order: imagine the graph with edges $A \to B$, $B \to C$, $A \to C$; in this graph, if we take the path from A to C first, we did not explore the other vertices. Rather, this is why the idea of linearization is so useful for DAGs, instead of using DFS.
- Some students wrote proofs that did not match their algorithm (e.g. abstracted a level away and proved the *intent* of their algorithm). These did not prove the algorithm, regardless of whether or not the algorithm was a successful implementation of the main idea.
- Naive algorithms received, as usual, no credit. In this case, naive would be $O(n^2)$, doing something like run DFS from every vertex.
- Finally, a reminder that in a DAG, every vertex is its own strongly connected component. Generally, the notion of a SCC is unlikely to be relevant for DAGs, even if mentioning SCCs wasn't wrong, per se.

Shortest Path with Time-Dependent Edges (20 points)

7. Mr. Albert is on a vacation in Switzerland. There are n cities in Switzerland and m trains $T_1, \ldots, T_m$ between cities. Each train T_i departs city *origin*$[i]$ at time *dep*$[i]$ and arrives in city *destination*$[i]$ at time *arr*$[i]$. Different trains between the same pair of cities could have different journey times.

   Albert can switch trains at a station instantaneously, i.e., Albert can arrive at time t, switch trains and depart on a train leaving at time t.

   Albert starts his journey at time 0. The arrival and departure times *arr*[] & *dep*[] are specified in the units – *"hours from the time Albert started his journey"*.

   (a) Modify Djikstra's algorithm to compute the duration of the quickest route to city B starting in city A at time 0. (proof of correctness and running time bound not required)

```
for all cities $v$, $dist[v] = \infty$
$dist[A] = 0$
$H \leftarrow makeQueue()$; // priority queue containing cities with dist values
while $H$ is nonempty
    $v \leftarrow deleteMin(H)$
    for every train $T_i$ from city $v$ do
        (write your pseudocode here)

    Solution:
        if $dist[v] \leq dep[i]$ :
            if $arr[i] < dist[destination[i]]$ :
                $dist[destination[i]] \leftarrow arr[i]$
                $H.decreaseKey(destination[i], arr[i])$
return $dist[B]$
```

Explanation:
The values in *dist* will tell us the earliest time at which it is possible for Albert to reach each city. As such, the first if statement ensures that we only consider trains which it is possible for Albert to catch. Each train acts as an edge in a graph where cities are vertices. Rather than explicitly computing edge weights and factoring in both travel and weight time, we update *dist* values with the arrival times of trains. Since we are running Dijkstra's algorithm, we must update the priority queue so that we consider cities ordered by the earliest time that they could be reached.

(b) Albert does not mind the train journeys, but really hates waiting at the stations. Albert would like to find an itinerary that reaches B within 3 days, but minimizes the total wait time at the train stations. Design an algorithm to find such a path.

(Hint: Model the problem using a different graph whose nodes specify more than just the city where Albert is. Running Djikstra's algorithm in this graph would give the desired path.)

(Briefly but precisely describe the graph, and argue the correctness of the algorithm.)

Solution

Very few responses received any credit. To show how to approach the problem, we develop several, increasingly refined solutions.

Naive Layering

First, remove all trains whose arrival times are later than 3 days from Albert's start time. This settles the issue of paths that take too long overall.

We now apply the strategy of creating many copies or "layers" of the original input.

i. Create the ordered set

$$S = \left(\bigcup_{i=1}^{m} dep[i] \right) \cup \left(\bigcup_{i=1}^{m} arr[i] \right)$$

of times at which any train departs or arrives anywhere.

ii. For each time s_k in S, create a copy V_k of the original set of cities V.

iii. For each train T_i, add an edge from city *origin*$[i]$ in layer *dep*$[i]$ to city *destination*$[i]$ in layer *arr*$[i]$. Give these edges zero weight, to signify that Albert doesn't mind time spent traveling.

iv. For each city v_k in layer V_k, add an edge to its copy v_{k+1} in the layer V_{k+1} immediately "above" it; that is, the chronologically next time in S. Give this edge weight $s_{k+1} - s_k$. This penalizes waiting at city v_k from time s_k to s_{k+1}.

v. Run Dijkstra's algorithm on the constructed graph and recover the shortest A-B path.

This is a correct algorithm because the total length of the recovered path will be the total time spent waiting in cities. The zero-length train edges ensure that the cities in this graph are connected the same way as in the original input, and that time spent traveling is cost-free. Also, we eliminated all trains that would enable longer-than-3-day travel, without destroying any other paths.

Since there are up to $2m$ unique times in S, we create $O(mn)$ nodes. We also create $O(mn)$ "waiting" edges and m train edges, for $O(mn)$ edges overall. Thus Dijkstra's algorithm costs $O(mn \log(mn))$ time.

Optimized Layering

Observe that the naive layering solution created numerous useless city nodes. Simply eliminating those city-times v_k that do not interact with any trains at time S_k brings the number of nodes down to $O(m + n)$ (do you see why?). The number of edges is $O(m)$. Thus Dijkstra will run in time $O((m + n) \log(m + n))$.

A Direct Approach

When we notice that the (zero-cost) train edges can be turned into nodes that connect directly via waiting edges, we arrive at the following solution (which can also be reached independently):

i. For each train T_i, create a node (which for simplicity we also call T_i.)

ii. For each city v, create edges (T_i, T_j) between each inbound train T_i and each outbound train T_j *whose departure time from v is later than the arrival time of T_i at v.* Set the weight of each such edge to the time Albert would wait between taking those trains: $dep[j] - arr[i]$.

iii. Create nodes for cities A and B, zero-weight edges (A, T_i) for all trains T_i that depart from A, and zero-weight edges (T_j, B) for all trains T_j that arrive at B.

iv. Run Dijkstra on this graph from A.

The shortest A-B path returned is clearly the sequence of train rides that minimizes total wait time (ignoring nodes A and B in the path). Unfortunately, it is possible to devise an example input for which this constructed graph would have $\Omega(mn)$ edges, making the running time not much better than the naive layering approach. However, this method is also very natural and easy to visualize.