CS 170 Efficient Algorithms and Intractable Problems
Spring 2016 Alessandro Chiesa and Umesh Vazirani

MT 1 Solution

Name: Joe

SID: 1234

GSI and section time:

Write down the names of the students on your left and right as they appear on their SID.

Name of student on your left:

Name of student on your right:

Name of student behind you:

Name of student in front of you:

Note: For each question part, 20% of the points will be given to any blank answer, or to any *clearly* crossed-out answer.
Note: If you finish in the last 15 minutes, please remain seated and not leave early, to not distract your fellow classmates.

Instructions: You may consult one handwritten, single-sided sheet of notes. You may not consult other notes, textbooks, etc. Cell phones and other electronic devices are not permitted.

There are 8 questions. The last page is page 12.

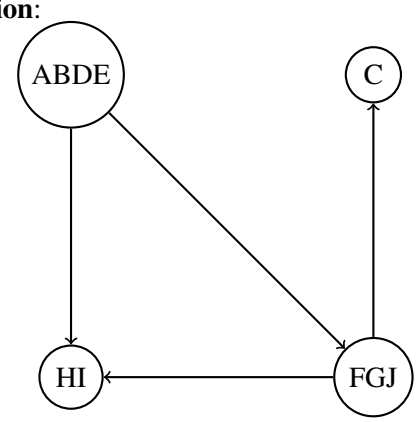
Answer all questions. On questions asking for an algorithm, make sure to respond in the format we request. Be precise and concise. Write in the space provided. Good luck!

Do not turn this page until an instructor tells you to do so.

1. (10 pts.) Strongly Connected Components

(a) **(6 pts.)** For the directed graph below, find the strongly connected components and draw the DAG of the strongly connected components. (Graph in problem left out of solutions for space reasons).

Solution:



(b) **(2 pts.)** List a set of edges which, if added to the graph, would make the graph have exactly one sink SCC and one source SCC. Use the fewest edges possible; if the graph already has one source SCC and one sink SCC, write "None."

Solution:

There are many possible answers, all involving a single edge from C, H, or I to any vertex not in the same connected component.

For example: **(C,F)**

(c) **(2 pts.)** List a set of edges which, if added to the graph, would make the whole graph strongly connected. Use the fewest edges possible; if the graph is already strongly connected, write "None."

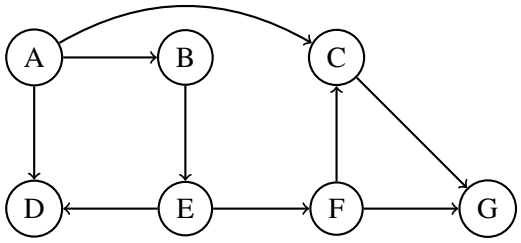
Solution:

The solution must have one edge from C, H, or I to one of A, B, D, E. The other edge must be from H or I (if the first was from C) or from C (otherwise) and go to any other connected component.

For example, {**(C,B), (H,A)**}

2. (15 pts.) Topological Sort

(a) **(5 pts.)** For the directed graph below, give a topological ordering of the vertices.



Solution: There are 4 possible orderings. One such ordering is (A, B, E, D, F, C, G).
For the other orderings, we have the constraints of the partial ordering (A, B, E, F, C, G), and we have that D has to go after E.

(b) **(5 pts.)** Now consider *all possible* topological sorts of this graph. For each following pair of vertices X and Y,

write $<$ if X must be before Y in any topological sort; $>$ if X must be after Y; and $\lesseqgtr$ if X can be either before or after Y.

$C > E$

$B < D$

$D \lesseqgtr G$

(c) **(5 pts.)** State a concise sufficient and necessary condition for when two vertices in a DAG can be in either order in a topological sort. (In other words, either one of them can be first in a valid topological sort of that DAG).

Solution:

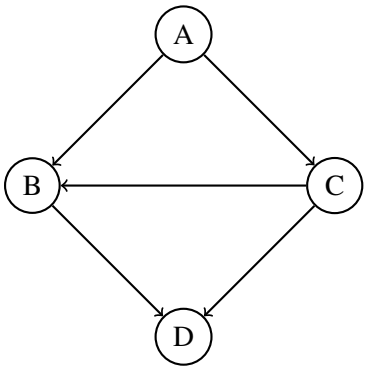
There is no path from either of the vertices to the other.

Equivalently,

Neither of the vertices is an ancestor of the other.

Common mistakes:

- "The two vertices have disjoint pre-post intervals" – in the graph below, DFS traversal could go through the vertices in the order A, B, D, C. The pre-post intervals for C and D are disjoint, but C must come before D in any topological sort.
- "The lengths of the shortest paths from the vertices to the root/parent are the same" – in the graph below, B and C have the same root/parent-to-node shortest path lengths, but C must come before B in any topological sort.



3. (20 pts.) True/False

For each part, determine whether the statement is true or false. If true, prove the statement is true. If false, provide a simple counterexample demonstrating that it is false.

(a) **(5 pts.)** The vertex with the smallest post-number in the DFS on a DAG is necessarily a sink vertex.

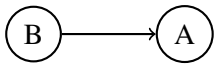
Mark one: **TRUE** or FALSE

Solution: When a vertex has its post-number assigned, it can only have edges pointing to vertices that have been fully explored (which contradicts the claim that it has the *smallest* post-number), or pointing to vertices that are currently in the DFS stack (which can't happen in a DAG). Thus, in a DAG, the vertex with the smallest post-number must be a sink.

(b) **(5 pts.)** Arranging the vertices of a DAG according to increasing pre-number results in a topological sort.

Mark one: TRUE or **FALSE**

Solution: Take the following graph:



If we explore A before B, then A will have a smaller pre-value of 1 (compared to B's pre-value of 3). However the only linearization is (B,A).

(c) **(5 pts.)** Dijkstra's algorithm finds the shortest path even if some edge weights are 0 (and none are negative).

Mark one: **TRUE** or FALSE

Solution: One way to see this is to merge all vertices separated by 0 weight edges together. Dijstra's on the original graph will compute the same paths as Dijkstra's on the original graph, and indeed all path lengths are unchanged by merging those vertices together.

Alternatively, unlike negative edges, 0-weight edges don't alter the invariant that when Dijkstra's pops a vertex off the priority queue and explores it, its distance is correct. Even if there's some length-0 path we haven't yet explored, it can't possibly improve the dist estimate versus what Dijkstra's had when it popped the vertex.

(d) **(5 pts.)** Recall the unique-shortest-paths question on Homework 4, where you are given an undirected graph $G = (V, E)$ with edge lengths $l_e > 0$, starting vertex $s \in V$ and asked to tell, for every vertex u, if there is an unique shortest path from s to u. This can be done by modifying Dijkstra's algorithm (lines 8-10 are added):

```
1:  Initialize usp[·] to true
2:  while H is not empty do
3:      u = DELETEMIN(H)
4:      for all (u, v) ∈ E do
5:          if dist(v) > dist(u) + l(u, v) then
6:              dist(v) = dist(u) + l(u, v)
7:              DECREASEKEY(H, v)
8:              usp[v] = usp[u]
9:          else if dist(v) = dist(u) + l(u, v) then
10:             usp[v] = false
```

Does this algorithm still compute `usp` correctly if some edge weights are zero (and none are negative)?

Mark one: TRUE or **FALSE**

Solution: Consider the graph with four vertices s, t, u, and w, with edges $s-t$, $t-u$, $t-w$, and $s-u$ of length 1, 0, 0, 1, respectively. Starting from s, if t is visited first before u, then `usp`$[w]$ will be set to `true` and will not change afterwards, but there are two shortest paths from s to w.

Common Mistakes:

(a) Using a directed graph as a counterexample

(b) Showing the graph (or something similar) $s \overset{1}{-} t \overset{0}{-} u$ and claiming that `usp`$[t] =$ `false` whereas there is a unique shortest path from s to t, which is not what the algorithm does, and also there are multiple shortest paths from s to t (passing through u).

4. (10 pts.) FFT Fundamentals

(a) **(5 pts.)** What is the FFT of the vector [1, 1, 1]?

Solution: We pad the vector with a 0 so that the degree is a power of 2, and then multiply the vector to $M_4(\omega)$:

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 3 \\ i \\ 1 \\ -i \end{bmatrix}$$

Common Mistakes: Not including the 4th component, $-i$, in your final answer.

Comments: Another accepted solution was using the third roots of unity. The correct answer, observing that the third roots cancel out when summed together, was $\begin{bmatrix} 3 \\ 0 \\ 0 \end{bmatrix}$.

(b) **(5 pts.)** Determine the polynomial of degree at most 3 that is represented by the following points: $f(\omega^0) = 0$, $f(\omega^1) = 1$, $f(\omega^2) = 0$, $f(\omega^3) = 1$. (Here ω denotes the first of the 4th roots of unity.)

Solution: We run the inverse FFT algorithm, multiplying $M_4(\omega^{-1})$ with the points vector $[0,1,0,1]^T$:

$$\frac{1}{4}\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 2/4 \\ 0 \\ -2/4 \\ 0 \end{bmatrix}$$

Thus, the polynomial is $1/2 - 1/2x^2$.

Common Mistakes: Interpreting the final vector in reverse order and getting $1/2x^3 - 1/2x$.

Comments: It would also have worked to use other ways of interpolating polynomials, like solving a system of linear equations or use Lagrange interpolation.

5. (15 pts.) Retroactive Oil Trading

The price of oil has been especially volatile over the last month. You want to figure out the largest profit you could have made in a single trade over that period, by buying oil at a certain time and selling at a later time. Specifically, given the chart of the oil price $P[i]$ for $i = 1$ to n, we're looking for the maximum value of $(P[k] - P[j])/P[j]$ for some indices j, k, with $1 \leq j < k \leq n$.

(a) **(7 pts.)** Fill in the blanks below to produce an algorithm that correctly solves this problem. You do not need to prove that your algorithm is correct.

OILMAX($P[1..n]$):

1. If $n \leq 1$, return $-\infty$.
2. Set $max_L :=$ OILMAX($P[1..\lfloor \frac{n}{2} \rfloor]$).
3. Set $max_R :=$ OILMAX($P[\lfloor \frac{n}{2} + 1 \rfloor..n]$).
4. Set $x := \min(P[1..\lfloor \frac{n}{2} \rfloor])$.
5. Set $y := \max(P[\lfloor \frac{n}{2} + 1 \rfloor..n])$.
6. Return $\max(max_L, max_R, \frac{y-x}{x})$.

(b) **(3 pts.)** What is the runtime of the above algorithm? Give a short justification of your answer.

Solution: There are two recursive calls on a subarray of half the size, plus a linear ($\Theta(n)$) scan to find the smallest/largest of either half. Thus, the recurrence relation is $T(n) = 2T(n/2) + \Theta(n)$, which solves to $\Theta(n \log n)$.

(c) **(5 pts.)** Now, modify this algorithm to make it asymptotically faster. Clearly describe the changes you would make to the algorithm in (a) to speed it up, and state the new runtime. You do not need to justify the runtime or prove your new algorithm correct.

Solution: We can modify the algorithm to also return the smallest and largest elements of the array (i.e. return the triple (best price, min of array, max of array)). This allows the algorithm to avoid the computation of finding the smallest/largest elements of each half of the array, as the recursive calls have already calculated them. (As an additional detail, the base case also needed a minor tweak.)

Our runtime would thus be $T(n) = 2T(n/2) + \Theta(1)$, which solves to $\Theta(n)$.

You did not need to show pseudocode, but here is pseudocode implementing the above modifications:

OILMAX($P[1..n]$):
1. If $n \leq 0$, return $(-\infty, \infty, -\infty)$.
2. If $n = 1$, return $(-\infty, P[1], P[1])$.
3. Set $(OM_L, min_L, max_L) :=$ OILMAX($P[1..\lfloor \frac{n}{2} \rfloor]$).
4. Set $(OM_R, min_R, max_R) :=$ OILMAX($P[\lfloor \frac{n}{2} + 1 \rfloor..n]$).
5. Return $(\max(OM_L, OM_R, \frac{max_R - min_L}{min_L}), \min(min_L, min_R), \max(max_L, max_R))$.

Iterative Solution: There is also an iterative solution that was given credit. There are numerous ways to implement it. The solution sketch is to keep track of the minimum value seen so far as you iterate through the array and seeing the best possible trade. You can also do it the other way. We saw many different implementations of this algorithm, and gave all reasonable implementations credit.

Common Mistakes:

- Greedy algorithms like looking at the global max or min and assuming that one of them will always be chosen will not work. As a counterexample, look at the array $[7, 6, 3, 4, 5, 2, 1]$: the best trade is buying at price 3 and selling at price 5.

6. (10 pts.) Arithmetic

Given three subsets A, B, C of the set of integers $\{1, \ldots, n\}$, determine which elements in C can be written as the sum of a pair of numbers from A and B (one each).

For example, if $n = 6$, $A = \{1,3,4\}$, $B = \{3,6\}$, and $C = \{2,4,5,6\}$, then the result would be $\{4,6\}$.

There is a straightforward n^2 solution; find a faster one.

Please answer in the following format: clearly describe your algorithm (formal pseudocode is not necessary), justify why it is correct, and give and briefly justify the runtime.

Solution: (As an aside, we notice the similarity between this problem and Triple Sum on the homework.)
Main Idea: We first encode A as a polynomial $f_A(x) = \sum_{a \in A} x^a$, and we encode B as a polynomial $f_B(x)$ in a similar manner.
We multiply the two polynomials together using FFT, and call the result $f_{AB}(x)$.
We then iterate through C: for each element c in C, if the coefficient of the term x_c is nonzero, then we include it as a value to return.
Proof: We need to argue that any element in C that can be written as the sum of a pair of numbers from A and B will be returned by our algorithm, and that any number our algorithm returns can be written as such a sum (thus, our algorithm returns all and only the desired numbers).
For the first argument, we note that if there are numbers $a \in A, b \in B, c \in C$ such that $a + b = c$, then we would have a x^a term in $f_A(x)$ and a x^b term in $f_B(x)$; when we multiply using FFT, the product of these terms would be x^{a+b}. This term won't "cancel out" with any negative term, since we are multiplying two polynomials with only nonnegative coefficients; thus, there will be a nonzero coefficient for the x^{a+b} term in $f_{AB}x$, so our algorithm will note that $c \in C$ should be returned.
For the second argument, if our algorithm returns c, then it must be that $c \in C$ and that there is a nonzero coefficient for x^c in $f_{AB}(x)$ so that there exist a, b such that $a + b = c$ and there are nonzero coefficients for x^a, x^b in $f_A(x), f_B(x)$, respectively. Thus, by construction, we must have $a \in A$ and $b \in B$, so that c can be written as the sum of a pair of numbers in A and B.
Thus, we conclude that our algorithm will return only the elements in C that can be written as the sum of a pair of numbers in A and B.
Runtime: $\Theta(n \log n)$. This algorithm has three steps: encoding A and B into polynomials takes linear time, multiplying the polynomials f_A and f_B together using FFT takes $\Theta(n \log n)$ time (as they have degree at most n), and checking which elements in C have a corresponding term in $f_{AB}(x)$ takes linear time. Out of these three operations, the polynomial multiplication dominates.

Common Mistakes:

- It was important to describe *how* the sets were encoded as polynomials, and not just that they should be encoded. It would also have been preferable to not also call the polynomials A and B.
- Not a mistake per se, but some students described (to varying levels of detail) the process of FFT and how it is used for polynomial multiplication. This was not needed: you can always black box any algorithm that we've taught. In particular, it was not necessary to specify that f_A and f_B were polynomials of degree that was a power of 2; FFT will pad the polynomials as necessary.
- It was insufficient to simplify explain that the runtime was that of FFT. There were other steps executed in the algorithm, before and after the polynomial multiplication, whose runtime should be discussed, *before* saying that FFT dominates.
- "This problem is like the one in the homework" is not a proof of correctness.

7. (15 pts.) Weighty Vertices

Consider an *undirected* graph $G(V,E)$, with nonnegative weights $d(e)$ and $w(v)$ for both edges and vertices. You have access to a Dijkstra's algorithm solver, which (in a standard undirected graph, with edge weights but no vertex weights) gives you the shortest path from a start vertex s to all other vertices.

Construct a graph which you can feed into the Dijkstra's solver to find the shortest path from the start vertex $s \in V$ to all other vertices.

Alternatively, for two-thirds credit, solve this problem assuming the edges are directed *instead. If you choose this option, you have access to a Dijkstra's algorithm solver which works on directed graphs.*

Please answer in the following format: clearly describe your graph, and prove that the path Dijkstra's returns on it will be the shortest path in the original graph.

Undirected Graph – Solution 1: Create a new graph G' with the same vertices and edges as before. For each edge $e(u,v)$, set edge weights in the new graph $d'(e) := d(e) + w(u)/2 + w(v)/2$. (This could also be done by replacing each edge with three "segments," with weights equal to $d(e)$, $w(u)/2$, and $w(v)/2$.)

Undirected Graph – Solution 2: Create a new graph G' as follows: for each vertex u in the graph (say it has degree d, so that it is an endpoint of edges $(u,v_1)..(u,v_d)$). Then replace u with d vertices $u_1..u_d$. Then replace all the (u,v_i) edges with (u_i,v_i) edges with the same weight, and also add all edges of the form (u_i,u_j) with weight $w(u)$.

Undirected Graph – Solution 3*: Convert the undirected graph to a directed one by adding directed edges (u,v) and (v,u) with the same weight as before. Then do Directed – Solution 1. This was eligible for 13/15 points.

Directed Graph – Solution 1: Update the weight of each graph edge $e(u,v)$ to $d'(e) = d(e) + w(v)$. Equivalently, set $d'(e) = d(e) + w(u)$ instead.

Directed Graph – Solution 2: Replace each vertex v with v_{in} and v_{out}. Add edge (v_{in}, v_{out}) with weight $w(v)$. Then replace (v,u) edges with (v_{out},u) edges, and (t,v) edges with (t,v_{in}) edges, all with the same weight as before.

Proof: If you got the construction completely correct, we were generally pretty generous with the proof if it was justified in a reasonable manner. IF the construction had significant errors, we may have given some points on the proof but never full points on the proof.

The easiest way to rigorously prove Dijkstra's runs correctly on the new graph was to argue that any path in the new graph corresponds to a path with the same weight in the original graph. We know that Dijsktra's will find shortest paths in the new graph, so if there is a direct correspondence between a path in the new graph and in the old, the same path in the old graph must have been shortest. That proof looks different depending on your solution, here's an example:

Undirected Graph – Solution 1: Any path in the new graph has weight $w(s)/2 + \sum_i (w_i) + w(t)/2 + \sum_e d(e)$, where i iterates through the internal vertices visited and e iterates through the edges used. This is exactly the same weight as the path in the original graph, except for $w(s)$ and $w(t)$, which are incorrectly weighted. But we know any $s-t$ path must go through s and t exactly once, so that this difference is constant for any possible path. Thus whichever path was shortest in the original path will directly correspond to a shortest path in the new graph.

Common Mistakes:

- Splitting the weight of the vertex evenly among all vertices, or not splitting it at all.
- Using a BFS or DFS preprocessing step to figure out the "direction" we'll go across each undirected edge in the Dijkstra's run, and adding the incoming or outgoing vertex weight to that edge accordingly.

This is circular in the sense that before we fix the graph and run Dijkstra's, we can't know in which direction we'll explore edges (eg for edge (u, v), we can't tell whether u is closer than v or vice versa).

- Some solutions failed whenever any vertex had a degree of more than 2.

8. (20 pts.) Super Mario Path

You wish to use your knowledge of CS 170 to get a leg up on playing Super Mario Kart. In Level 17, you must drive through Bowser's Castle, which you can model as a directed graph. Traveling through each passage in the castle (directed edge) adds a positive or negative number of points to your score, depending upon what type of monster/object/power-up is on it. (Call $w(e)$ to find the number of points (which can be negative) each edge adds.) To successfully traverse the level, you must find a path, *perhaps with repeated edges*, from the castle entrance s to the exit t, such that your score never goes negative *at any point in time*. Assume that every vertex is reachable from s and every vertex is able to reach t. Use your knowledge of CS 170 to design an efficient algorithm to find such a path if it exists, or to determine that there is no such path.

Please answer in the following format: describe the main idea of your algorithm, write the pseudocode, justify why it is correct, and give and briefly justify the runtime.

Solution

Main ideas:

1. Use a variant of Bellman-Ford;
2. At each update, verify that the value of the path never becomes negative;
3. If a reachable (from s, via a path that is never negative) positive cycle is found, we are guaranteed to have some legal path from s to t.

Pseudocode:

procedure MARIO($G = (V,E)$; $w : E \to \mathbb{R}$; $s,t \in V$) ▷ Initialize:
  for $v \in V$ **do**
    $Value[v] \leftarrow -\infty$
  $Value[s] \leftarrow 0$
  ▷ Main Loop:
  for $i = 1 \ldots |V| - 1$ **do**
    for $(u \to v) \in E$ **do**
      if $Value[v] < Value[u] + w(e)$ AND $Value[u] \geq 0$ **then** ▷ Idea # 2
        $Value[v] \leftarrow Value[u] + w(e)$
  if $Value[t] \geq 0$ **then return** True
  ▷ Find positive cycles
  for $(u \to v) \in E$ **do**
    if $Value[v] < Value[u] + w(e)$ AND $Value[u] \geq 0$ **then return** True
  return False

Proof:

Claims 1 and 5 below guarantee that if there is an always-nonnegative path from s to t, the algorithm returns true. Claims 2-4 guarantee that if the algorithm returns true, then there is a path.

Claim 1: If there is an always-nonnegative path from s to t of length at most $|V| - 1$, the algorithm discovers it in the Main Loop.

Proof of Claim 1: By induction on path length.

Induction hypothesis: If there is an always-nonnegative path from s to v (for any $v \in V$) of length ℓ and total weight $\bar{w}$, we will have $Value[v] \geq \bar{w}$ by the ℓ-th iteration of the Main Loop.

Base case: $\ell = 0$, $v = s$.

Induction step: Let u be the last vertex before v in the path. By the induction hypothesis, after the $(\ell-1)$-th, $Value[u] \geq \bar{w} - w(u \to v)$, and since the path is always-nonnegative, $Value[u] \geq 0$. Therefore after we tried to perform the ℓ-th update on edge $(u \to v)$, we have that $Value[v] \geq \bar{w}$.

Claim 2: If $Value[t] \geq 0$, then there is an always-nonnegative path from s to t.

Proof of Claim 2: By induction on the Main Loop's updates.

Induction hypothesis: At any point during the run of the algorithm and for every $v \in V$, if $Value[v] \geq 0$, then there is an always-nonnegative path from s to v of total weight $Value[v]$.

Base case: Before the loop begins, only $Value[s] = 0$.

Induction step: Suppose the induction hypothesis is true before the inner loop considers edge $(u \to v)$. Only $Value[v]$ may change. If $Value[v]$ changes to $Value[u] + w(u \to v)$, then by the induction hypothesis, there exists an always-nonnegative path from s to u of total weight at least $Value[u]$. Furthermore, since $Value[u] \geq 0$, we can continue the same path from u to v and it remains always-nonnegative.

Claim 3: If the shortest always-nonnegative path from s to t has length at least $|V|$, then the graph has a positive cycle that is reachable from s via an always-nonnegative path.

Proof of Claim 3: If the path has length $\geq |V|$ it must contain a cycle, if the cycle is not positive, we can remove it and get a shorter always-nonnegative path from s to t.

Claim 4: If the graph has a positive cycle that is reachable from s via an always-nonnegative path, then there is an always-nonnegative path from s to t.

Proof of Claim 4: By the premise, there is an always-nonnegative path from s to the cycle; traverse the cycle an arbitrary number of times to reach an arbitrarily high positive value; then traverse any simple path from the cycle to t.

Claim 5: If the graph has a positive cycle that is reachable from s, the algorithm detects it.

Proof of Claim 5: The path to the cycle and the first time we go through the cycle until we close a loop are an always-nonnegative path of length at most $|V|$ from s to some vertex v. Therefore, by the time we exit the Main Loop, for every vertex v in the cycle, we have $Value[v] \geq 0$. Therefore one of the values on the cycle must be updated in the $|V|$-th iteration.

Running time: Like Bellman-Ford, $O(|V| \cdot |E|)$.