10 $\frac{38}{40}$ +2 = $\frac{40}{40}$ Sandipan Dey CMSC 651 HW-7

1. To prove: $A \le_L B \wedge B \le_L C \Rightarrow A \le_L C$.

Proof: $A \le_L B \Rightarrow \exists$ logspace computable $f : \Sigma^* \to \Sigma^* \mid \forall w,\ w \in A \Leftrightarrow f(w) \in B$

$B \le_L C \Rightarrow$ " " " $g : \Sigma^* \to \Sigma^* \mid \forall w,\ w \in B \Leftrightarrow f(w) \in C$

(By definition)

Let

M_f: logspace transducer computing f

M_g: logspace transducer computing g.

Then $g \circ f : \Sigma^* \to \Sigma^* \mid \forall w,\ w \in A \Leftrightarrow (g \circ f) w \in C$ with $(g \circ f)(w) = g(f(w))$

Wrong proof:

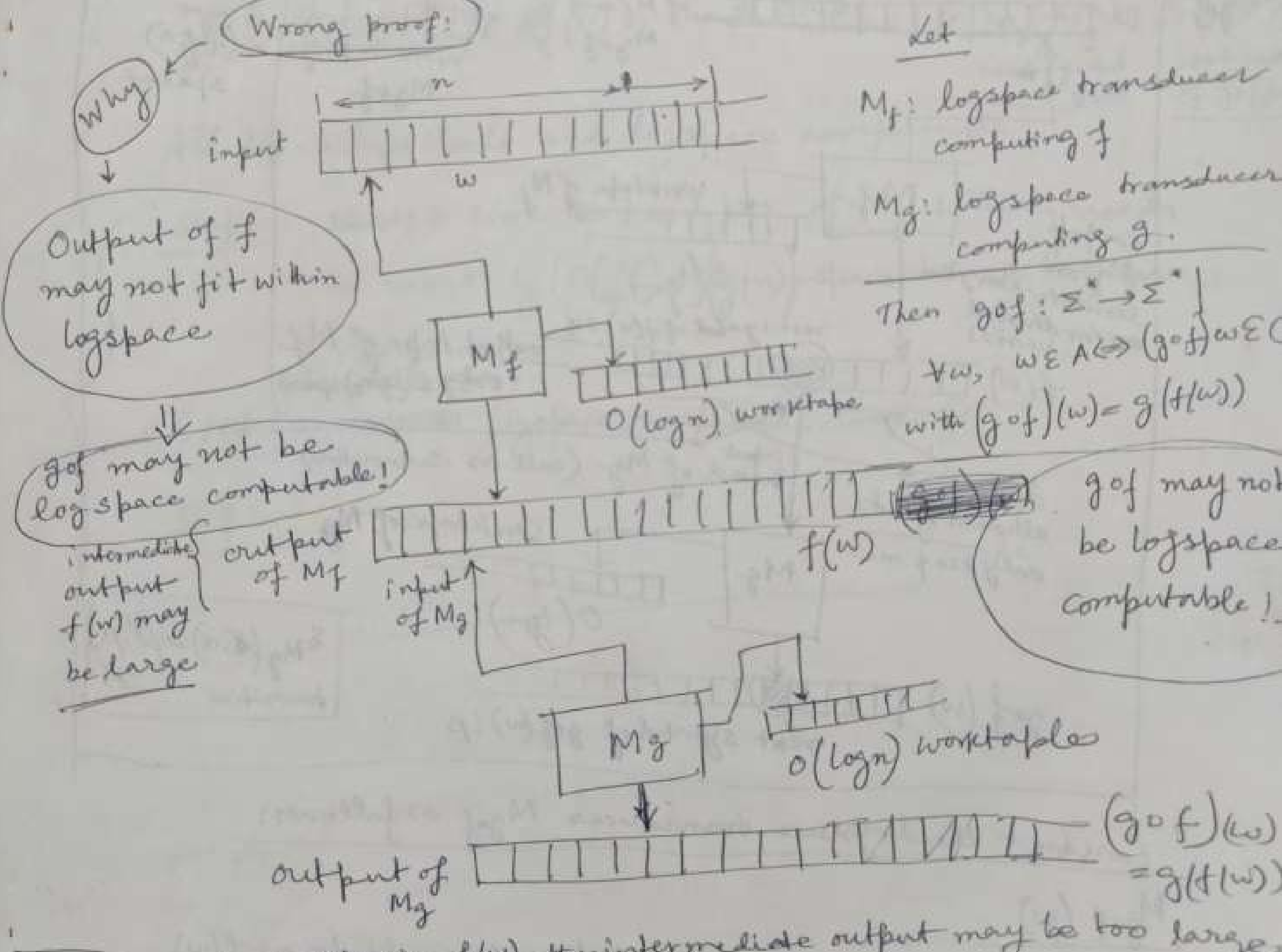
Why

Output of f may not fit within logspace

⇓

$g \circ f$ may not be logspace computable!

intermediate output $f(w)$ may be large



$g \circ f$ may not be logspace computable!

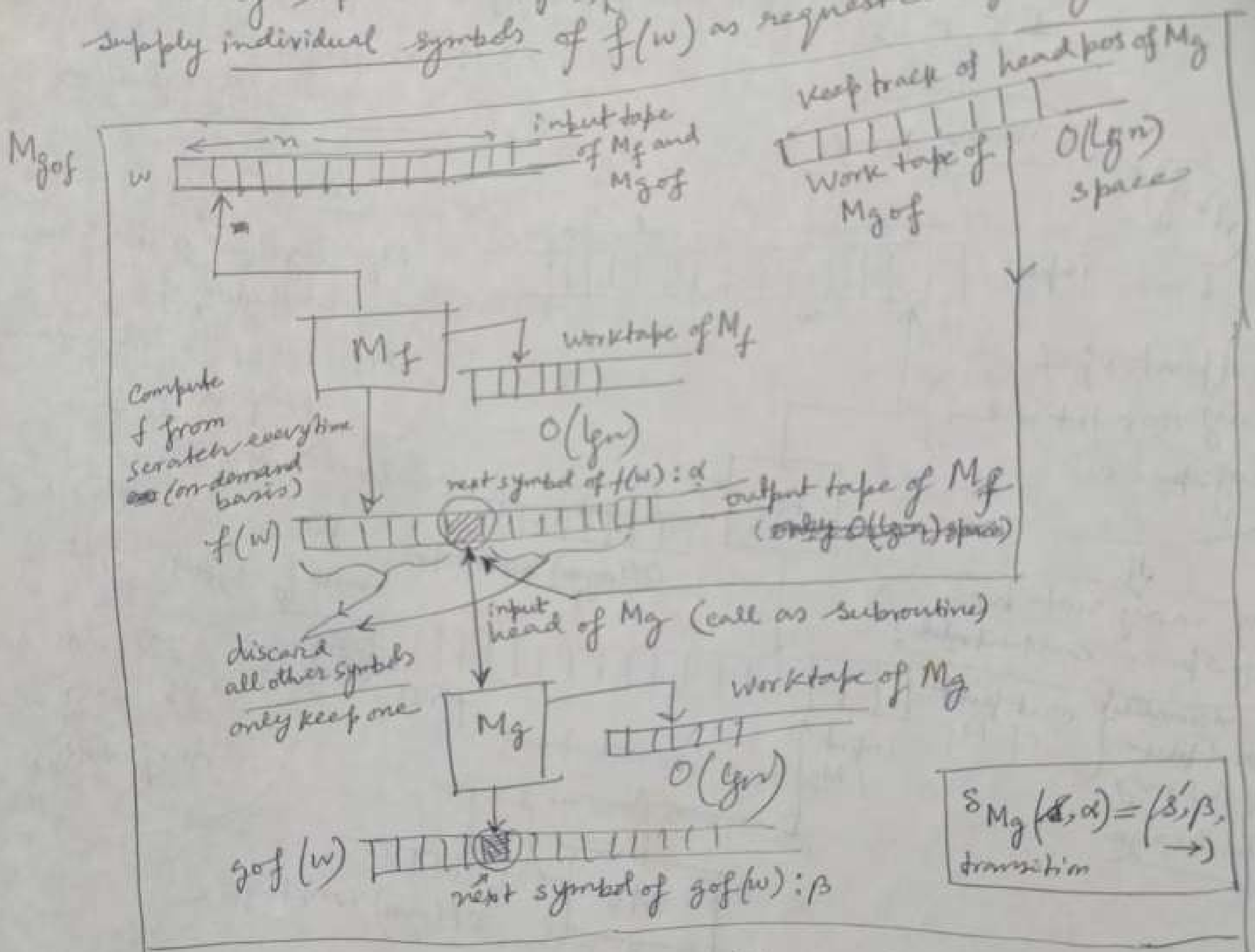
The problem is: $f(w)$, the intermediate output may be too large to fit with logspace bound. Since M_f runs in $O(\lg n)$ space on input w of length n, number of configurations of M_f of n ~~an~~ is $O(n\,2^{O(\lg n)} \ldots$ $= O(n^2_{\lg n})$ ~~and~~ which is an upper bound on the running time of M_f w and also on the length of $f(w)$, which is not in logspace. Hen

~~f(w)~~ $g \circ f$ is no longer logspace Computable!

[illegible]

Correct Proof:

Instead of storing the entire $f(w)$ (which may need much larger space than $\log n$), M_f recomputes $f(w)$ everytime to supply individual symbols of $f(w)$ as requested by M_g.



Construct the logspace transducer $M_{g \circ f}$ as follows:

$M_{g \circ f}(w)$

1. Keep track of where M_g's input head would be on $f(w)$ /* e.g., initialize with the start pos. of $f(w)$ */ If the input head is on the right of $f(w)$, exit.
2. (Re-) compute $f(w)$ by running M_f on w from the begining and ignore all the output except the desired location (where M_g's input head is). /*

need to store all of $f(w)$, only a single symbol is needed everytime the head of M_g is moved */

3. Run M_g, only ~~[illegible]~~ by a single step /* e.g. step execution mode, a single transition each time */ on the input symbol $f(w)$ obtained from M_f and compute the next symbol for $g(f(w))$. Go to Step 1. /* $\delta_{M_g}(s, \underset{\substack{\uparrow \\ \text{next symbol} \\ \text{of } f(w)}}{\alpha}) = (s', \underset{\substack{\uparrow \\ \text{next symbol} \\ \text{of } g \circ f(w)}}{\beta}, R)$ */

All the 3 steps halt and logspace computable.

Step 1: Needs to store the head pos. of M_g, which needs at most $\lg(O(\frac{n^2}{4})) = O(\lg(n))$ space hence logspace computable.

Step 2: ~~Needs~~ logspace computable, since f is.

Step 3: " " , " g " ✓

2. Let's assume that the input adjacency list is represented as ~~our~~ d

10

pairs of 2-tuples, i.e., if the list looks like $(u_1, v_1), (u_2, v_2), (u_3, v_3)$
then we have $(u_1, v_1) < (u_2, v_2) < (u_3, v_3) < \cdots$ where the relation $<$
defined as: $(u_1, v_1) < (u_2, v_2) \iff (u_1 < u_2) \vee (u_1 = u_2 \wedge v_1 < v_2)$
(standard representation of adjacency matrix). According to the assumption $(1, 2), (2, 1256798), (2, 3)$ is not a valid represe
instead it should be $(1, 2), (2, 3), (2, 1256798)$. Similarly $(1, 2), ($
is a valid

should be able to fix the adjacency list if it doesn't have this format

~~The input~~ Also let's assume the output adjacency matrix ~~list~~ format i
going to be like the following: if the graph has m nodes it will
be $\underbrace{\delta_{11}\delta_{12}\cdots\delta_{1m}\delta_{21}\delta_{22}\cdots\delta_{2m}\cdots\delta_{m1}\delta_{m2}\cdots\delta_{mm}}_{\text{length } m^2 \text{ binary string}}$ with $\delta_{ij} = \begin{cases} 1, & \text{iff } (i,j) \in \text{ad} \\ 0, & \text{ow} \end{cases}$

There exists no isolated vertices (given).

Obviously, we need to compute the following things in logarithmic s

1. Number of vertices of the graph, m

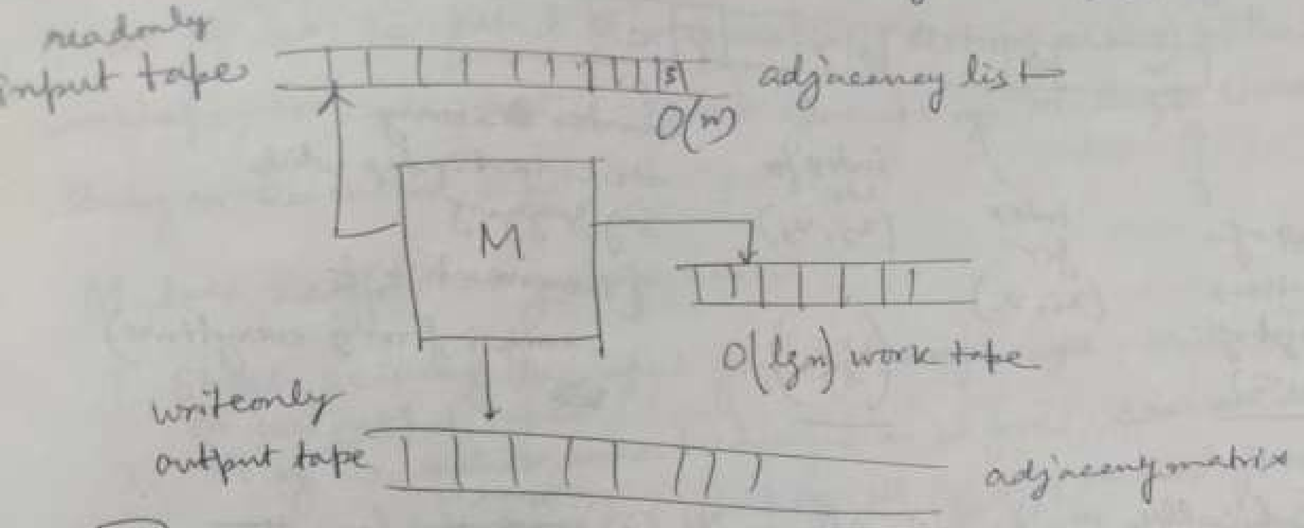
~~2: $\delta_{ij}, \forall i, j$~~

~~2. If the graph is sparse~~ change the vertex numbers accordingly

3. Compute $\delta_{ij}, \forall (i, j) = 1, \ldots, m$

For ~~the~~ sparse matrix we note that output will not have any redundant rows, i.e. $(\forall i)(\exists j)(\delta_{ij} \neq 0)$. Also since we are concerne about space here we shall not ~~to~~ be efficient w.r.t. time.

With all that in mind let's describe the algorithm for the logspace transducer that is going to convert the adjacency list representation to an adjacency matrix representation st let's concentrate on computing the count of vertices (m).

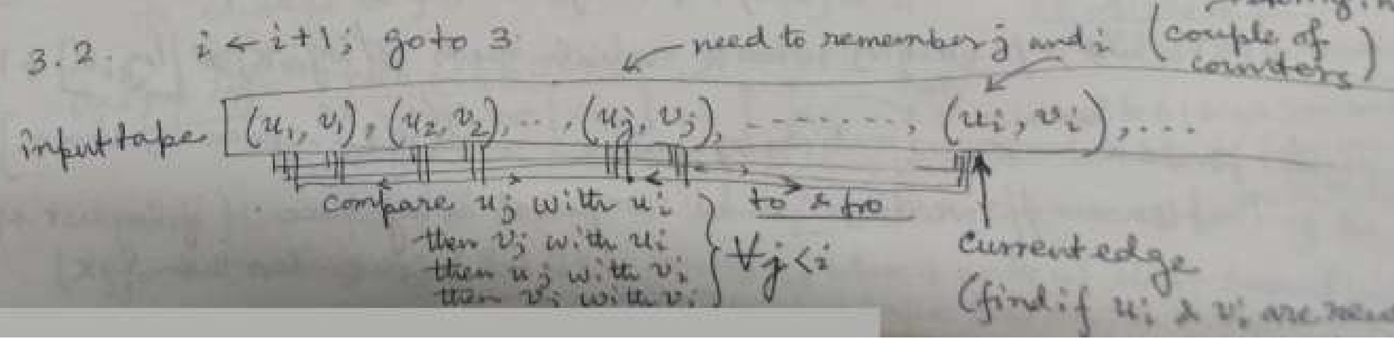


st $\boxed{M}$ does ~~does~~ the following:

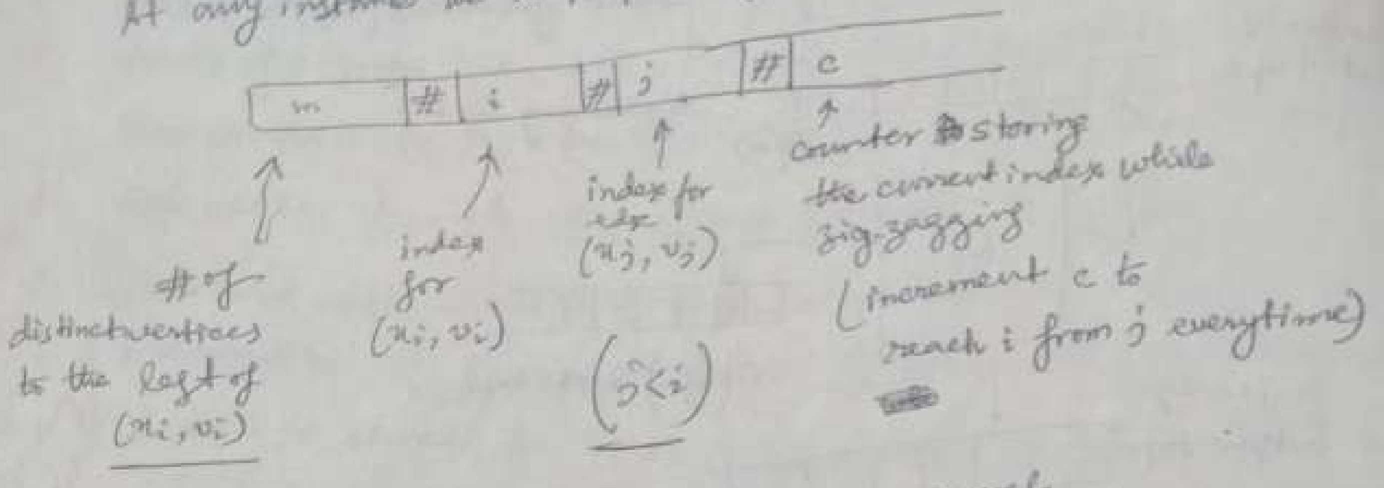
1. Initialize m with 0 and store it on the worktape. /* $m \leftarrow 0$ */
2. ~~Scan the input~~ Store the ~~current~~ next/ index (i) of the edge to be scanned ~~next~~ on the worktape as well, initialize it with 1. ~~e.g.~~ /* e.g., if $(u_1, v_1), (u_2, v_2), (u_3, v_3)$ ... in the adjacency list (u_1, v_1) and (u_2, v_2) are already scanned, $i \leftarrow 3$. Before start of scanning the input, $i \leftarrow 1$, whenever '),' is encountered, increment i */
3. Repeatedly do the following until the end of input (e.g. '\$') is reached:
    3.1. ~~Initialize a counter~~ (Re)start scanning the input from the begining. ~~c on the worktape by 1. While ($c < i$) { move head to right and if '),' is encountered increment c by 1}. Now extract the next edge (u_i, v_i) from the input~~
    $(\forall j < i)$ compare u_j and v_j with u_i in a zigzag manner, if all are different from u_i (which means a new vertex),
    $m \leftarrow m+1$;
    Do the similar comparison to check if v_i is a new symbol.

✓ Good.

    3.2. $i \leftarrow i+1$; goto 3



At any instance the worktape may look like:



Typically m, i, j, c all will be ~~[illegible]~~ much less than $O(\log n)$, since i, j, c stores the index of the edge only.

(e.g. for if $(1,2), (2, 123576798), (33,2)$ be the adj-list.

$$\underline{(1,2), (\overset{\uparrow}{\underset{j}{2}}, 123576798)}, (\overset{\uparrow}{\underset{i}{33}},2)$$

i is going to store 3 (since 3rd edge)

j " " " " 2 (" 2nd ")

m " " " 4 (" 4 different vertices)]

~~also in the worst case if we have adj-list as (1,2), (2,3), (3,4), ..., (k-1,k)~~

~~input len n = ⌊log₁₀ 1⌋+1 + 2⌊log₁₀ 2⌋+1 + 2 log 3 + ... + 2 log₁₀(k-1) + log₁₀ k + 3k + k-1~~

~~Now the next step~~

[Also, in the worst case if we have adj-list as $(1,2),(2,3),(3,4),\ldots,(k-1,k)$,

input length $n = \lfloor \log_{10} 1 \rfloor + 1 + 2(\lfloor \log_{10} 2 \rfloor + 1) + 2(\lfloor \log_{10} 3 \rfloor + 1) + \cdots + 2(\lfloor \log_{10} k-1 \rfloor + 1)$

$$+ \lfloor \log_{10} k \rfloor + 1 + 3.\underset{(,)}{k} + \underset{,}{k-1}$$

$$\simeq 2.\log_{10} 2.3 \ldots (k-1) + 4k + \log_{10} k$$

$$= 2 \log_{10} (k-1)! + 4k + \log_{10} k$$

$$\geq 2.\log_{10} 10^{k-1} + 4k + \log_{10} k, \quad \forall k \geq 11$$

$$= 2(k-1) + 4k + \log_{10} k = O(k).$$

While, # vertices $m = k$, which needs ~~$O(\log_{10} k)$~~ $\lfloor \log_{10} k \rfloor + 1$

for same reason i, j needs also log space $= O(\lg k)$ space to store]

Hence # vertices can be computed in logspace. (if the list sparse it will be even less than $\lg k$)

Now the next step:

Once M has computed m (number of distinct vertices) on the worktape, it knows that it needs to generate an m^2-length binary string on the output tape.

M does the following steps next.

1. Start scanning the input from left ~~away~~ to right. Write m^2 zeros on the output tape.
2. Using the ~~entire~~ same technique as before, scan an edge (u_i, v_i). Again $\forall j<i$ compare all (u_j, v_j)s on the left with current edge to find the ~~task~~ new vertex number (in case of sparse matrix it will be different) of u_i and v_i. ~~Write a~~ (let's say u_i' and v_i') The renaming of vertices is explained in the example below. ~~3. Start from the beginning of the output tape and see move the head towards right giving identity transition~~ ~~(a, a)~~ ~~(blank, blank) etc~~)

gSpace

In between two consecutive (u_i, v_i) and (u_i, w_i) if v_i' and w_i' are not consecutive integers (obviously $v_i' < w_i'$), write the intermediate bits as 0s on the tape. But write 1 at the position (then write 0's) and write 1 at "
$(m-1)*u_i + v_i$
$(m-1)*u_i + w_i$ on the output tape

Not sure about the details of your renaming scheme, but something like this will work

3. Go to step 2 and repeat until $i > m$.

Vertex renaming

Let's see an example.

Count # vertices ~~with~~ having less number than the current one = 4 hence rename to 5

logSpace counting

input tape (1,2) (2,3) (3, 10100) (5,1) (5, 12345678910)

rename to 5 — rename to 4 — rename to 6

M finds m=6

M extracts (1,2) first and finds that ~~both are~~ it does not need to change vertex numbers. So it ~~can~~ writes 0's until it reaches s_{12}, where it writes a 1.

output tape 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 ~~1~~ 0 0 0 0 1

s_{11} s_{12} … s_{16} s_{21} s_{23} s_{31} s_{35} s_{41} s_{46}

12345678910 renamed

5 renamed to 4 (since 3 vertices <)

10100 renamed to 5 (since 4 vertices with less …)

Then M scans (2,3) and sees that $u_2 = 2$, hence it goes on writing 0's for the remaining s_{1j}'s and till it reaches s_{23} where it writes a 1. Next scan (3,10100) and scanning the input tape find there are 4 vertices with less numbers. Having …

3. To prove: Immerman-Szelepcsenyi Theorem extends to

10 space bounds, i.e., $\forall \underbrace{S(n)}_{\text{not space constructible}} \geq \lg n$, $NSPACE(S(n)) = co\text{-}NSPACE(S(n))$

Proof ~~[illegible]~~ M which is $S(n)$-space bounded $\Rightarrow \overline{L(M)} \in co\text{-}NSPACE(S(n))$

1. M be a NTM which is $S(n)$-space bounded. We have to construct another NTM N accepting $\overline{L(M)}$. If M has finite alphabet set ~~[illegible]~~ Δ, with $|\Delta| = d$, every configuration can be represented as a string in $\Delta^{S(n)}$, where n is length of input.

2. Assume M has unique accepting configuration w.l.o.g., accept $\in \Delta^{S(n)}$ on inputs of length n. Let start $\in \Delta^{S(n)}$ represent start configuration on input x, $|x| = n$.

Define $\underset{\text{set of configuration}}{A_m} = \{\alpha \in \Delta^{S(n)} \mid \text{start} \xrightarrow{\leq m} \alpha\}$ configuration reachable within at most m steps from start

if x is accepted by M, the length of accepting config can be at most $d^{S(n)}$

Obviously $A_0 = \{\text{start}\}$, $|A_0| = 1$, Since $|A_m| \leq d^{S(n)}$, $\forall m$ to compute any $|A_m|$ only $O(S(n))$ space is needed

$x \in L(M) \Leftrightarrow \text{accept} \in A_{d^{S(n)}}$

3. Now, since $S(n)$ is NOT space constructible, we can't start by marking of $S(n)$ space (note: this much space is required in computing any $|A_m|$ and N uses the concept of census function to nondeterministically test the nonmembership of accept in $A_{d^{S(n)}}$) on N's worktape.

4. Instead ~~[illegible]~~ do the following steps (5-6) for successive values of $S = 1, 2, 3, \ldots$ approximating the true space bound $S(n)$ (in order to get rid of space constructibility). For each S, if we ever encounter a configuration reachable from the start configuration that wants to use more than S space, set $S \leftarrow S+1$ and restart. We shall eventually hit $S(n)$, at which point no reachable configuration will try to use more space.

5. Lay off S space ~~[illegible]~~ on N's worktape. ~~[illegible]~~ $|A_0| = 1$, Compute $|A_{m+1}|$ from $|A_m|$: successively write down $\beta \in \Delta^{S}$ in lexicographic order and for each one determine if $\beta \in A_{m+1}$, if so, increment a counter by one.

To test whether $\beta \in A_{m+1}$, nondeterministically guess $|A_m|$ elements of A_m in lexicographic order, verify that each $\alpha \in A_m$ by guessing the computation path $start \xrightarrow{\leq m} \alpha$ and for each α check whether $\alpha \xrightarrow{\leq 1} \beta$. If true then $\beta \in A_{m+1}$ ow $\beta \notin A_{m+1}$ (using census function concept $\overset{N}{}$ nondeterministically decided the non-reachability problem)

6. After $|A_{d^s}|$ has been computed test accept $\notin A_{d^s}$ non-membership again in the similar manner by guessing $|A_{d^s}|$ elements α of A_{d^s} and verifying $\alpha \in A_{d^s}$ by guessing computation path $start \xrightarrow{\leq d^s} \alpha$, also verifying each α is different from accept.

/* Repeat steps 5, 6 for s=1, 2, 3, .. since the function is not space constructible, can't mark S(n) space initially on N's tape */

N runs in $S(n)$ space and decides $L(\overline{M}) \in$ co-NSPACE$(S(n))$ (Proved) ✓

$\Rightarrow$ NSPACE$(S(n))$ = co-NSPACE$(S(n))$

Extra-credit

4. To prove: UCYCLE $\in$ L

8 +2 =10

Proof We have to find whether the graph G is a forest or not. We assume that the vertices of G are arbitrarily labeled 1 through n. First let's define a cyclic ordering of the edges incident on a vertex u in the following manner: if $(u, v_1), (u, v_2), \ldots, (u, v_k) \in E[G]$ with $d(u) = k$, the cyclic ordering is defined by $(u, v_{i_1}) < (u, v_{i_2}) < \cdots < (u, v_{i_k}) < (u, v_{i_1})$

Where $i_1, i_2, \ldots, i_k$ is a permutation of $1, 2, \ldots, k$ and $i_1 < i_2 < \ldots < i_k$.

If $d(u) = 0$, no ordering. $d(u) = 1$, $(u, v) \in E$: the ordering is $(u, v) < (u, v)$

Next, let's define a traversal algorithm starting from any vertex u. The algorithm starts with a vertex edge pair (u, e), where e is incident on u. At each step, the algorithm enters a vertex on an edge e_1 and leaves the vertex on the edge e_2 that comes after e_1 in the circular order.

Claim 1: The algorithm starting from u eventually returns to u.

Proof: Since the traversal algorithm starting from u can only reach the nodes v that are reachable (by a path) from u and reachibility is symmetric (u will also be reachable from v) and since the circular ~~ordering~~ ordering guarantees that all the ~~paths~~ edges ~~from~~ incident on every node must be traversed eventually, it must reach u. [If it reaches to a vertex $v \neq u$ from u via path P_1 it will return back to u via the path P_1 ~~or a different path P_2~~ if $\exists$ only one path P_1, $u \overset{P_1}{\rightsquigarrow} v$, otherwise it will return via a different path P_2, if $d(v) \geq 2$ and $\exists$ a different path $u \overset{P_2}{\rightsquigarrow} v$]

← C_1 is a component of G

Claim 2: If a vertex $u \in C_1$ is part of a cycle iff $\exists$ k such a traversal starting from u and traversing ~~all the~~ k reachable vertices from u, ~~must~~ coming back to u via a different edge. $k = 1, 2 \ldots, |C_1|$

part of a cycle $u \in C_1$, C_1 a component in G

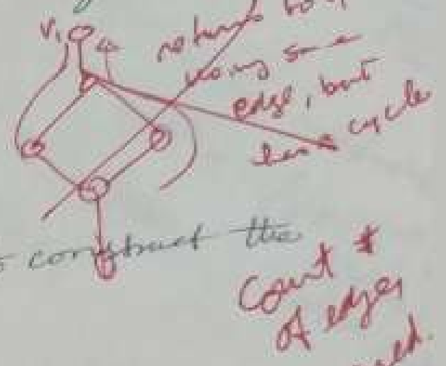
u not part of a cycle

Here $1 \leq k \leq |C_1|$, where C_1 is the component where u belongs to

Proof: It's easy to see that if ~~there is~~ u is not part of a cycle, $\forall v \in V(G)$ that are reachable from u, $\exists$ exactly one path from u to v. Hence ~~the~~ any traversal starting from u and exiting u with edge e_1 must return to u via edge e_1 only.

If u is part of a cycle, $\exists$ at least one more vertex $v \neq u$, on the same cycle, hence $\exists$ at least 2 paths from u to v and ~~since~~ by circular ordering, $\exists$ at least one traversal startin from u will come back ~~to~~ u in a different edge (and path).

Claim 3: G is a forest (does not contain a cycle) iff all possible such traversals starting from any vertex of G returns to the same vertex via the same edge through which it left that vertex.



Proof: Obvious from Claim 1 & 2

With all the above claims, we are now ready to construct the logspace transducer M that decides UCYCLE:

Count # of edges traversed.

```
        input length n
M(G(V, E))          /* Decides whether G contains CYCLE */
  1. $\forall v \in V(G)$  /* read the next vertex from input tape */
     do the following:
     1.1. Store the start vertex v, the previous vertex p and current vertex c only
          on the worktape. Initialize $c \leftarrow v$, $p \leftarrow v$   /* only O(log(n)) space */
     1.2. if (c == v) /* first vertex */ choose the first edge e by
          in the cyclic ordering ~~from V by choosing the~~
          finding the corresponding vertex $v_1 \in adj(c)$ /* read the
          input tape to find such $v_1$'s from $E[G]$ */
          Also store this edg $(v, v_1) = e$ (start edge)
          else choose the next edge $e_2$ incident on ~~adjacent to~~ c ~~in~~ such
               that it's next to $e_1 = (p, c)$ in the cyclic ordering
               /* scan each $e = (c, v) \in E[G]$ from input tape */
               /* $e_2 = (c, v_1)$ is such edge next to $e_1$, $e_1 < e_2$ */
          ~~$p \leftarrow c$, $c \leftarrow v_1$ goto 1.1~~
     1.3. if ($v_1 == v$) /* returned to start */
              if $(c, v_1) != e$ /* Not same return edge */ → CYCLE
                  output 'yes'; else goto 1;
          else goto 1.1; /* if $v_1 != v$ */
```

only uses logspace

Always terminates

Forest: → output 'No'.