Space Complexity

1 Introduction

So far, we have used time as a resource that has to be optimized during computation. This week, we shall study also time as a resource and the implications of the same. In our exploration, we still use the Turing machine as our computational model. Our interest is to be able to define appropriate complexity classes based on the space used. We shall also study how these complexity classes relate to complexity classes defined with respect to time. We will study how the deterministic and the nondeterministic machines may differ in their space complexity. Further, we can also see if there is a hierarchy of complexity classes with respect to space.

1.1 Space

We start with the following definition. We consider TMs with three tapes, an input tape that is read-only, a work-tape that is both read-write, and an output tape that is writable. The TM cannot write in the input tape but can read any cell of this tape. The number of cells used by a TM is the number of cells written on the work tape.

Definition 1.1 (DSpace) *Let $s : \mathbb{N} \rightarrow \mathbb{N}$ and let $L \subseteq zo$. Let M be a deterministic Turing machine that decides L with the property that for any $w \in \{0,1\}^*$, M executing on input w uses at most $c \cdot s(|w|)$ tape cells, for a constant c before deciding whether $w \in L$. Then, we say that $L \in$ DSpace$(s(n))$.*

Notice the similarity to that of deterministic time complexity that one can define. However, there is a fundamental difference between the resources space and time. Unlike time, space can be reused and this property has far reaching implications. Consider for example a TM that implements a counter for bit strings of length n. While it has run for 2^n time steps, it can run in a space of n. We first show the following example and then generalize the same.

Example. 1.2 *Recall the problem CLIQUE defined as:*

$$\textit{CLIQUE} = \{\langle G, k\rangle \mid G \textit{ has a clique of size } k\}.$$

It is known that CLIQUE $\in$ NP and also that CLIQUE $\in$ NP-complete. Therefore, it is unlikely that there will be a polynomial time algorithm for CLIQUE. However, a deterministic TM can be designed to solve the CLIQUE problem using space in $O(n)$ as follows.

Recall that each candidate solution to the CLIQUE problem can be represented as a subset of k vertices out of the n vertices. Such a subset can be in turn represented as a bit string of length n. The TM M does this by writing bit strings corresponding to k subsets of n in a lexicigraphic order. To verify if a particular subset is a solution, the TM can verify in polynomial time if the vertices belonging to the subset are mutual neighbors. In this case, the machine can accept the input. If all subsets fail to be valid solutions, the machine rejects the input. The spae used by M on its work tape is in $O(n)$.

Notice also how separating the input tape and the work tape is convenient. In the above example, it is still true that the input can also be represented in polynomial space. As we define compelxity classes with respect to space that are sub-linear, this difference becomes more important.

The above example is not an exception with respect to problems in NP. As the following theorem shows, it appears that space is more powerful than time. Let PSPACE $= \cup_{k \in \mathbb{N}} \text{DSpace}(n^k)$.

Theorem 1.3 *$P \subseteq NP \subseteq PSPACE$.*

The proof of the above theorem is left as an assignment. The above proof can be shown by using the following claim on the following notion of *configuration graphs* of TMs.

Let M be a TM that is either deterministic of nondeterministic. A *configuration* of M is a description of the contents of all the non-blank cells of the work tape of M, along with the state of M and the position of its head. Given a TM M and an input w to M, the configuration graph G_{Mw}, a directed graph, can be defined as follows. The nodes in G_{Mw} are the configurations that M can reach from the starting configuration with w on the input tape and the state being the start state, and the head at the first symbol in w. An edge between configurations C_l and C_2 exists in gmw iff from C_1 the machine can reach C_2 in one step. If M is deterministic, then the outdegree of G_{Mw} is 1 and if M is nondeterministic, then without loss of generality we can take that G_{Mw} has outdegree 2. Further, we can assume that M has only one accepting state, and also one accepting configuration, $\mathcal{C}_{\text{accept}}$. So, M accepts w iff there is a directed path from $\mathcal{C}_{\text{start}}$ to $\mathcal{C}_{\text{accept}}$ in G_{Mw}. The following claim is easy to show.

Claim 1.4 *Let M be a machine that decides some language $L \in \{0,1\}^*$ using space $s(n)$. Let G_{Mw} be the configuration graph of M on input w. Then,*

1. *G_{Mw} has $2^{O(s(n))}$ nodes.*
2. *Two configurations C_1 and C_2 are neighbors in G_{Mw} if and only if there exists a Boolean formula of size $s(n)$ in CNF that is satisfiable.*

Proof. A basic counting argument establishes item (i) as follows. A configuration is completely determined by the symbols on the tape, the position of the head, and the state of the machine. If the TM uses at most $s(n)$ tape cells, then the number of different ways the symbols can be arranged on the tape is $2^{s(n)}$, assuming that the alphabet is $\{0,1\}$. The number of states $|Q|$ can be taken to be a constant, and the position of the head has at most $s(n)$ choices. Put together, the total number of nodes in the graph is $O(2^{s(n)})$.

For item (ii), this observation is the essence of the NP-completeness of SAT as shown by Cook in his famous theorem. Essentially, we can use a sequence of Boolean And operators to check if two configurations are neighbors in the above graph. Each such Boolean formula has a constant size, constant based on the number of states and the size of the alphabet. □

To continue further, let us see how DTMs and NTMs can differ in their space usage. In this direction, we first extend Definition as follows.

Definition 1.5 (NSpace) *Let $s : \mathbb{N} \to \mathbb{N}$ and let $L \subseteq \{0,1\}^*$. Let M be a nondeterministic Turing machine that decides L with the property that for any $w \in \{0,1\}^*$, M executing on input w uses at most $c \cdot s(|w|)$ tape cells, for a constant c before deciding whether $w \in L$. Then, we say that $L \in NSpace(s(n))$.*

Recall that simulating a computation of an NTM on a DTM seems to require an exponential increase in time. For instance, an NTM running in $t(n)$ time can be simultaed by a DTM in time $O(2^{t(n)})$. In a remarkable result, Savitch showed that however deterministic TMs can simulate nondeterministic TMs with a very small space overhead.

Theorem 1.6 (Savitch) *Let $s : \mathbb{N} \rightarrow \mathbb{N}$ be such that $s(n) \geq n$. Then,*

$$\mathit{NSpace}(s(n)) \subseteq \mathit{DSpace}(s(n)^2)$$

Proof. To think about the proof, here is an idea. We need to show that a deterministic TM can simulate the actions of a non-deterministic TM that uses a space of $s(n)$. Using Claim **??** is a possibility. We need to check in space $s(n)^2$ if the starting configuration leads to the accepting configuration. The number of nodes in the graph is indeed $2^{O(s(n))}$. But given that the outdegree is 2, the number of paths in the graph is in $O(2^{2^{O(s(n))}})$. If we have to systematically explore all these paths, to represent a path by its number, we need a space of $O(2^{s(n)})$, which is much more than $O(s(n)^2)$ as promised by the theorem.

So we need a new trick here. Indeed, counting all possible paths is rather naive. There are several simple graphs with polynomial number of nodes but exponential number of paths. ⟦***Create one such graph to convince yourself.***⟧ The trick is to think of the graph reachability problem which need not be solved in such a brute-force manner.

The trick is to therefore mimic the graph reachability solution by using as little space as possible. The details follow. Let C_1 and C_2 be two configurations and let the predicate $\mathrm{Reach}(C_1, C_2, t)$ return Yes or No depending on whether from configuration C_1 there is path to configuration C_2 in at most t steps. The predicate Reach can be used recursivley starting from $\mathrm{Reach}(C_1, C_{\text{accept}}, t)$, where $t = 2^{cs(n)})$ for a suitable constant c. ⟦***Why should t be so high?***⟧. The predicate can be answered recursively as follows. To decide $\mathrm{Reach}(C_1, C_2, t)$ one can recursively check if there exists a configuration C_{mid} such that both $\mathrm{Reach}(C_1, C_{\text{mid}}, t/2)$ and $\mathrm{Reach}(C_{\text{mid}}, C_2, t/2)$ are both true. (In the above, we can take that t is an exact power of 2, or consider ceil$(t/2)$.) The recursive program can be written as shown below:

Algorithm Reach(C_1, C_2, t)
begin
  If $t = 1$ then
    Check if either $C_1 = C_2$ or C_1 and C_2 are neighboring configurations.
    If so, return True, else return False;
  end-if
  else
    For each configuration C_{mid} do
      result1 = Reach$(C_1, C_{\text{mid}}, t/2)$;
      result2 = Reach$(C_{\text{mid}}, C_2, t/2)$;
      return result1 and result2;
    end-for
  End-if
End-Algorithm

The determinstic machine M that simulates the $s(n)$-space nondeterministic machine N simply implements the above program. Let us now see how much space M needs for this simulation. For each recursive call, M needs to store three variables namely, C_1, C_2, and t. Since each is in $2^{O(s(n))}$, each of them can be stored by using $O(s(n))$ bits. There are also $\log 2^{O(s(n))}$ stages in the recursion. So, the overall space used by M is in $O(s(n)^2)$. □

An immediate implication of Svaitch;s theorem is the following corollary.

Corollary 1.7

$$P \subseteq NP \subseteq PSPACE = NSpace \subseteq EXPTIME.$$

Figure 1: Containments among complexity classes, as is widely beleived.

In the above list of containments, we do not know if any of the containments are proper. It is possible to show that P $\neq$ EXPTIME. Therefore, in the list of containments, one more of them has to be proper. A picture of the above containments, assuming that all are proper except the one indicated to be equal, is shown in Figure 1. It is widely beleived that the situation is as shown in Figure 1.

2 PSPACE and PSPACE-Completeness

A natural question to ask when we define a new complexity class is to seek problems that are complete with respect to that class. Recall the class PSPACE of problems. We now seek the notion of PSPACE-completeness for the class PSPACE.

Definition 2.1 *A language L is said to be PSPACE-complete if:*

- *L is in PSPACE, and*
- *Every language L' in PSPACE is polynomial time reducible to L.*

Notice the limitation on the reduction in the above definition. We are following the general rule that the comptuational class the reduction function belongs to must be weaker than the class for which we want to establish reduction. Analogous to the satisfiability problem for NP-completeness, we have a quantied formual satisfiability that can be shown to be PSPACE-complete.

Quantified Boolean formulae can be defined as follows. Recall the universal quantifiers $(\forall)$ and the existential quantifier $(\exists)$. Certain mathematical statements require the use of quantifiers to indicate the scope of applicability of the statement. For instance, when one considers natural numbers, the statement $x^2 + y^2 = z^2$ is true only some triples x, y, and z. So, one can write the above as $\exists x \exists y \exists z\ x^2 + y^2 = z^2$. Such a statement is called a quanitified formula. Another example is the following: $\forall x \langle \exists y \exists d\ x = y \cdot d \rangle \wedge \langle \exists z \exists e\ x = z \cdot e$. What does the above statement capture? Is it true? Assume that x, y, and z are positive integers.

The difference between the above two examples is that in the former all quantifiers appear at the beginning of the statement. Such a statement is called to be *prenex normal form*. When we consider that the variables are from $\{0, 1\}$ then quantified statements in prenex normal form are called quantified Boolean statement. Further, when all variables have some quantifier associated with the varaible, then the statement is called as a *fully qualified*. We define the language $TQBF$ as:

$$TQBF = \{\Phi | \Phi \text{ is a true fully quantifiable Boolean formula}\}.$$

We show now that $TQBF$ is PSPACE-complete.

Theorem 2.2 *TQBF is PSPACE-complete.*

Proof. As usual there are two items to show. One is to show that TQBF is in PSPACE and the second is to show that every other language in PSPACE is reducible to TQBF.

For the first item, consider a TQBF formula Φ. A deterministic TM for checking whether $\Phi \in TQBF$ can be designed as follows. There are at most a polynomial number of variables in Φ. For each variable, there are only two possible values: 0 and 1. The TM can systematically explore all possible values for all variables. For each such assignment, the TM checks if Φ is true. The TM accepts if there is an assignment under which Φ is true. Otherwise the TM rejects. Since the space used for writing the current assignment can be reused, the total space used by the TM is polynomial in the length of the input. Hence, $TQBF \in$ PSPACE.

The second item is more involved. Let L be a PSPACE language and let M be a TM that recognizes L. We wish to show that the action of M on an input w can be coded into a fully quantified Boolean formula Φ so that Φ is true iff $w \in L$.

Notice the similarity between the present problem and that of reducing any language L in NP-complete to SAT. Briefly, the proof of the latter constructs a Boolean formula that is satisfiable iff the machine for L accepts an input w.

One can check that the above approach fails for the PSPACE machine M for the following reason. M, being bound by polynomial space can run for an exponential amount of steps. Therefore, the Boolean formula that represents the actions of M on input w can be exponentially long. The reduction however is limited to be polynomial in time. So, such a reduction would not work.

Let us see if the configuration graph we defined for M on input w, $G_{M,w}$ helps. One can use the proof technique of Savitch's theorem to say that M accepts w iff there exists a configuration C_{mid} so that the formula corresponding to [$\mathcal{C}_{\text{start}}$ can reach $\mathcal{C}_{\text{mid}}$ in $t/2$ steps] and the formula corresponding to [$\mathcal{C}_{\text{mid}}$ can reach $\mathcal{C}_{\text{accept}}$ in $t/2$ steps] are both true. Let us represent the above by saying that $\Phi_{c_1,c_2,t}$ corresponds to the Boolean formula with c_1 as the starting configuration and c_2 as the ending configuration and t is the time allowed for reaching from c_1 to c_2. The formula $\Phi_{c_1,c_2,t}$ can be constructed recursively as:

$$\Phi_{c_1,c_2,t} = \exists c' \Phi_{c_1,c',t/2} \wedge \Phi_{c',c_2,t/2}$$

For $t = 1$, we just have to check if the configurations in Φ are neighboring configurations in $G_{M,w}$. While the above is technically correct, there is one problem.

The length of the expression $\Phi_{\mathcal{C}_{\text{start}},\mathcal{C}_{\text{accept}},t}$ for $t \in 2^{O(n^k)}$ can be super-polynomial in length. The recursion is controlled by the parameter t and every recursive call reduces t to $t/2$, but also doubles the size of the formula. So, the formula ends up having an exponential length as t can be exponential. The reduction cannot write such a long formula.

We therefore need another method to encode the actions of M into a polynomial sized fully quantified Boolean formula. This can be done as follows. The trick is to not double the length of the formula every recursive step. This is done by redefining $\Phi_{c_1,c_2,t}$ as:

$$\Phi_{c_1,c_2,t} := \exists \mathcal{C}_{\text{mid}} \forall (c,d) \in \{(\mathcal{C}_{\text{start}}, \mathcal{C}_{\text{mid}}), (\mathcal{C}_{\text{mid}}, \mathcal{C}_{\text{accept}})\} \ \langle \Phi_{c,d,t/2} \rangle.$$

While this does not increase the size of the Boolean formula every recursive step, the scope of the variables c and d is no longer Boolean. However, that can be easily fixed by writing the subexpression $\forall (c,d) \in \{(\mathcal{C}_{\text{start}}, \mathcal{C}_{\text{mid}}) \ \langle \Phi_{c,d,t/2} \rangle$ as $(c,d) = (\mathcal{C}_{\text{start}}, \mathcal{C}_{\text{mid}}) \cap (c,d) = (\mathcal{C}_{\text{mid}}, \mathcal{C}_{\text{start}}) \rightarrow \ \langle \Phi_{c,d,t/2} \rangle$. This way, the length of the formula $\Phi_{\mathcal{C}_{\text{start}},\mathcal{C}_{\text{accept}},t}$ is polynomial in $|w|$. □

An important observation regarding languages in PSPACE is that winning strategies for most games can be captured using quantified Boolean formulae. Most board games can thus be shown to be PSPACE-hard.

3 Sub-linear Space

In this section, we explore problems that can be solved by using space is that less than the space consumed by the input itself. Hence, if n denotes the size of the input, we are interested in problems that can be solved in sublinear space. Notice that the machine does have enough time to actually read the entire input while still working in sublinear space. So this notion is not ill-defined. Interestingly however, there is an equivalent notion with respect to time called sub-linear algorithms which were studied recently in the works of Indyk. While we may not have time to study some of the sublinear time works, plesae see [] for more details.

Given that we are consdiering sublinear space, what are the right sublinear space functions that are interesting? Should we seek problems that can be solved in $O(\sqrt{n})$-space?, $O(n^\epsilon)$-space for some constant $\epsilon < 1$?, or should we investigate $O(\log n)$-space?, or further small space such as $O(\log \log n)$-space? It turns that logarithmic space is a good candidate to study as it provides some natural intuitions to think about. In logarithmic space, one can essentially have space just enough to store a constant number of pointers into the input! Still it turns out that there is a class of interesting problems that can be solved in logarithmic space. Further, this class of functions is invariant to simple changes to the Turing machine model such as addingg more tapes, or increasing the size of the alphabet, etc. Therefore, studying logarithmic space is interesting.

Definition 3.1 *We define LogSpace to be the class of languages that can be decided by a deterministic Turing machine M using at most $O(\log n)$ cells of the tape where n is the size of the input. In other words,*

$$\textit{LogSpace} = \textit{DSpace}(\log n)$$

The following example shows a simple language that can be decided in logarithmic space.

Example. 3.2 *Let L be the language $\{\langle G\rangle | G$ has no triangle$\}$. Then L can be recognized in LogSpace as follows. On the work tape, the machine M can write all 3 tuples of the vertices of G. For each tuple written, the machine checks if there is a trinalge passing through those vertices. If all tests fail, then M accepts, otherwise M rejects. The space used by M is space enoug to store three vertex identifiers, therefore L is in LogSpace.*

Analogous to the definition of NSpace, also LogSpace has a nondeterministic equivalent.

Definition 3.3 *We define NLogSpace to be the class of languages that can be decided by a nondeterministic Turing machine M using at most $O(\log n)$ cells of the tape where n is the size of the input. In other words,*

$$\textit{NLogSpace} = \textit{NSpace}(\log n)$$

The following example illustrates a problem that can be solved nondeterministically using logarithmic space.

Example. 3.4 *Extend the above example to define the language L as the language $\{\langle G\rangle | G$ is bipartite$\}$. It can be shown that L is in NLogSpace as follows. The nondeterminitic machine guesses an odd integer k such that $1 \leq k \leq n$ where G has n vertices. For this value of k it starts by guessing k vertices that form a cycle in G. Notice that the k vertices need not be stored all at the same time. Only neighbouring vertices on the cycle need be stored on the tape. Therefore $L \in$ NLogSpace.*

Example. 3.5 *Let PATH be defined as the language $\{\langle G, u, v\rangle \mid G$ is a directed graph and $u \rightsquigarrow v\}$. PATH essentially tries to see if the given directed graph G has a path from a vertex u to vertex v in the graph G. Recall that there are standard algorithms that run in polynomial time and space that can decide the PATH problem. The challenge however is to show that indeed a small space suffices.*

A nondeterministic machine M that decides PATH can be designed as follows. On its work tape, M guesses the next node on a path from u to v. M starts by guessing one of the out-neighbors of u, say u_1. M writes u_1 on its work tape. It then guesses an outneighbor of u_1 and writes the id of this node on the work tape in the same space u_1 is written. If M is able to reach v via these guesses, then M accepts. If M makes more than $n = |V(G)|$ gusses, then M rejects the input.

Notice that M does not keep the history of its guesses; M just has to decide to accept or reject and need not show a path in case of acceptance.

The above example is rather remarkable. It turns out that if G is an undirected graph, then nondeterminism is not required. The undirected version of the above problem can be solved using logarithmic space by a deterministic machine.

3.1 Savitch's Theorem for LogSpace and NLogSpace

Recall from Savitch's theorem that for space beyond linear, it holds that NSpace$(s(n)) =$ DSpace$(s(n)^2)$. In this section, we argue that a similar result holds also when $s(n) \geq \log n$. One of the important pieces in the proof of Savitch's theorem is the relation between the space used by a TM and the time the TM takes to decide on any input. Specifically, we showed that an $s(n)$-space bounded TM runs in time $O(2^{s(n)})$. Is this relation still true for logarithmic space TMs? Not so at a first glance. A TM running in $O(1)$ space can just read the entire input, and do nothing with it thereby consuming $O(n)$ time. We therefore introduce the following definition that relates the space and time for every function $s(n)$.

Definition 3.6 *Consider a multitape TM M with a separate read-only input tape. Let w be an input to M. A configuration of M on w consists of the state of M, the contents on its work tape, and the positions of its heads, including the head of the read-only tape.*

The essence of this definition is to separate the input from the configuration of a TM. This is not a problem as the input is read-only, and the position of the head is included in the configuration.

Using the above definition, it now holds that if a TM runs in space $s(n)$, then on an input w of length $|w|$, the number of configurations is in $O(n \cdot 2^{s(n)})$. How?

Finally, Savitch's theorem can be proved using the above number of configurations. The configuration graph again has $n2^{O(s(n))}$ nodes and we have solve a reachability problem. Storing the parameters for every recursive call now requires a space of $\log(n2^{O(s(n))}) = \log n + O(s(n))$. Therefore, so long as $s(n) \geq \log n$, Savitch's theorem holds.

3.2 NLogSpace-Completeness

Analogous to the definition of time based complexity classes P and NP, we have now defined logarithmic space based complexity classes LogSpace and NLogSpace. So, it is natural to ask the relation between the classes LogSpace and NLogSpace. Clearly, LogSpace $\subseteq$ NLogSpace, and the question therefore is whether LogSpace $=$ NLogSpace or not. This sounds very similar to the P versus NP question. To create evidence that P is not equal to NP, there is the notion of NP-complete that shows that certain problems in NP are not likely to be in P. In the same flavour, one can build evidence to show that LogSpace $\neq$ NLogSpace by searching for complete problems with respect to NLogSpace.

As with the general definition of a problem complete for a class, an NLogSpace-complete problem represents a problem that is most difficult in the class NLogSpace. Therefore, a problem A is NLogSpace complete if it is in NLogSpace and every other problem in NLogSpace reduces to A. What has to be specified in the above notion is the time allowed for the reduction function. If we use polynomial reducibility, then there is a small inconsistency. All problems in NLogSpace are solvable in polynomial time. So, every

pair of problems in NLogSpace, except the pair Φ and Σ^* are reducible to each other. This suggests that polynomial reducibility is too powerful a notion in the case of NLogSpace completeness. (In general, when defining complete problems with respect to a class, the power allowed to the reduction must be smaller than the resources sufficient to decide the class itself). We therefore introduce the following notion of log space reducibility.

Definition 3.7 *A language L_1 is log space reducible to a language B, written $A \leq_L B$ if positive instances of A can be converted to positive instances of B using a function $f : \Sigma^* \to \Sigma^*$ where f is a log space computable function. A log space computable function is a function such that there is a Turing machine M with a read-only input tape, a $O(\log n)$ long work tape, and a write-only output tape, which on input $w \in \Sigma^*$ on the input tape computes $f(w)$ and writes $f(w)$ on the output tape.*

Using the above definition, a problem L_1 is NLogSpace complete if $L_1 \in$ NLogSpace and every problem in NLogSpace is log space reducible to L_1. Standard results of the above definition that follow are given below.

Theorem 3.8 *The following are both true.*

- *If $A \leq_L B$ and $B \in$ LogSpace then also $A \in$ LogSpace.*
- *If any NLogSpace-complete language is in LogSpace then LogSpace $=$ NLogSpace.*

To end this section, we finally show that the PATH problem defined earlier is NLogSpace-complete.

Theorem 3.9 *PATH is NLogSpace-complete.*

Proof. There are two items to show. Firstly, we need to show that PATH $\in$ NLogSpace. But that is already done in the example. Secondly, we need to show that every other problem in NLogSpace is log space reducible to PATH. For this, we need to exhibit a log space reduction that converts positive instances of any problem in NLogSpace to positive instances of PATH. The reduction works as follows.

Imagine for a moment that there is no restriction on the space used for the reduction. Let us understand how to reduce any problem in NLogSpace to the PATH problem. Let $B \in$ NLogSpace and let M be a nondeterministic TM that deciedes B. One can think of constructing a graph $G_{M,w}$ for input w such that nodes in $G_{M,w}$ correspond to the configurations of the machine M on w. Now, if there is a path in $G_{M,w}$ from the start configuration to the accepting configuration, as an instance of PATH, then w is accepted by M.

Let us now see how this graph can be constructed using only logarithmic space. The nodes of $G_{M,w}$ are the configurations of M on input w. There are $O(n^k)$ nodes in this graph as there are only $n2^{O(\log n)}$ configurations. An edge exists in $G_{M,w}$ from configuration C_1 to C_2 iff either $C_1 = C_2$ or C_2 is reached in one step of the machine from C_1. This can be checked given the definition of M.

To compute the graph in logarithmic space, one can proceed as follows. Notice that only the work tape is bounded by logarithmic space, but not the output tape. To describe a graph, one needs to simply describe its nodes and edges. The nodes can be listed in a lexicogrpahic order as follows. Each node in $G_{M,w}$ can be represented in $O(\log n)$ bits. So, start by listing all bit strings of length $O(\log n)$ on the work tape. For each bit string b, check if b is a valid configuration of M on input w. If so, write b on the output tape. Write the string next to b in the lexicographic order on the work tape, and repeat until all bit strings of length $O(\log n)$ are checked. To list the edges, a similar technique can be used. Now, we list pairs of bit strings of length $O(\log n)$ in lexicographic order on the work tape. We check if the current pair is a valid edge, and if so, write the edge on to the output tape. This check can be done by using the transition function of M. □

An immediate corollary of the above theorem is the following.

Corollary 3.10 *NLogSpace $\subseteq$ P.*

3.3 The Class coNLogSpace

We now turn our attention to another space based complexity class along with a surprising result. The class coNLogSpace is the set of languages whose complement can be decided by a nondeterministic Turing machine using logarithmic space. The class coNLogSpace can be seen as analogous to the class coNP. It is generally beleived that NP $\neq$ coNP. However, as the following theorem shows, NLogSpace $=$ coNLogSpace.

Theorem 3.11 *NLogSpace= coNLogSpace*

Proof. We have to show that every problem in coNLogSpace is also in NLogSpace. Here is where complete problems come to our rescue. Recall that PATH is NLogSpace-complete. So, if $\overline{\text{PATH}}$ were shown to be in NLogSpace, then also every problem in coNLogSpace would be in NLogSpace. The language $\overline{\text{PATH}}$ is the set of directed graphs with two vertices u and v such that there is *no* path from u to v. We now show a nondeterministic loagrithmic space machine for $\overline{\text{PATH}}$.

The nondeterministic machine should somehow conclude that no path exists between two vertices u and v in a given directed graph. Let us ignore the space aspect for a moment. Then, one way of doing this is as follows. The nondeterministic machine tries to partition the vertex set of the given graph into two parts V_R and V_{NR}. V_R contains the set of vertices that are reachable from s and V_{NR} contains the set of vertices that are not reachable from s. If $v \in V_{NR}$ then M accepts and if $v \in N_R$ then M rejects. To see whether a vertex x is to be placed in V_R, M can check using nondeterminism if there is a path from u to x. This can be done also in logarithmic space. If such a verification fails, then it means that the vertex x should be in V_{NR}. Once vertex t is characterized, the machine can decide accordingly.

In the above, we are storing the n bits to indicate whether a vertex is in V_R or its complement. This is space more than what we can afford to use. To reduce space, we not the following observations. Firstly, we need not know for every vertex x if x is in V_R or not. It suffices if we can match up the number of vertices in V_R. How do we know $|V_R|$? Assume so for a moment that we know $|V_R|$. Then, convince yourself that the machine M can run in logarithmic space using nondeterminism.

To find $|V_R|$ we proceed as follows. V_R can be written as the union of sets V_R^i where V_R^i is the set of vertices that are at a shortest distance of i from u. Note that $V_R^0 = \{u\}$. And, V_R^{i+1} can be computed from V_R^i as follows. For each vertex x, it is verified if the x is in V_R^i. (Note: We cannot store this information). If so, then it checks if for every vertex y, (x, y) is an edge in G. If so, then y is in V_R^{i+1}. This is similar to a space reduced version of BFS. □