# Introduction to Complexity Theory*
# Lecture 5: Non-Deterministic Space

Notes taken by Yoad Lustig and Tal Hassner

January 14, 1999

**Summary:** We recall two basic facts about deterministic space complexity, and then define non-deterministic space complexity. Three alternative models for measuring non-deterministic space complexity are introduced : the standard non-deterministic model, the online model and the offline model. The equivalence between the non-deterministic and online models and their exponential relation to the offline model are proved. After the relationships between the non-deterministic models are presented we turn to investigate the relation between the non-deterministic and deterministic space complexity. Savitch's theorem is presented and we conclude with the translation lemma.

## 1 Preliminaries

During the last lectures we have introduced the notion of space complexity, and in order to be able to measure sub-linear space complexity, a variant model of a Turing machine was introduced. In this model in addition to the work tape(s) and the finite state control, the machine contains two special tapes : an input tape and an output tape. These dedicated tapes are restricted each in it's own way. The input tape is read only and the output tape is write only and unidirectional (i.e. the head can only move in one direction).
In order to deal with non-deterministic space complexity we will have to change the model again, but before embarking on that task, two basic facts regarding the relations between time and space complexity classes should be reminded.

To simplify the description of asymptotic behaviour of functions we define :

**Definition 1** *Given two functions* $f : \mathsf{N} \rightarrow \mathsf{N}$ *and* $g : \mathsf{N} \rightarrow \mathsf{R}$
$f$ is at least $g$ *if there exists an* $n_0 \in \mathsf{N}$ *s.t. for all* $n \geq n_0$ $f(n) \geq \lceil g(n) \rceil$.
$f$ is at least linear *if there exists a linear function* $g$ *s.t.* $f$ *is at least* $g$ *(there exists a constant* $c > 0$ *s.t.* $f$ *is at least* $cn$*).*

**Fact 1.1** *For every function* $S(\cdot)$ *which is at least* $log(\cdot)$ $DSPACE(S) \subseteq DTIME(2^{O(S)})$.

**Proof:** Given a Turing machine $M$, a complete description of it's computational state on a fixed input at time $t$ can be given by specifying :

- The contents of the work tape(s).

* Lecture Notes for a course given by Oded Goldreich at the Weizmann Institute of Science, Israel.

- The location of the head(s) on work tape(s).
- The location of the head on the input tape.
- The state of the machine.

Denote such a description a *configuration of* $M$. (Such a configuration may be encoded in many ways, however in the rest of the discussion we will assume a standard encoding was fixed, and would not differentiate between a configuration and it's encoding. For example we might refer to the space needed to hold such a configuration. This is of course the space needed to hold the representation of the configuration and therefore this is a property of the encoding method, however from an asymptotic point of view the minor differences between reasonable encoding methods make little difference). A complete description of an entire computation can be made simply by specifying the configuration at every time $t$ of the computation.

If during a computation at time $t$, machine $M$ reached a configuration in which it has already been in at time $t_1 < t$, (i.e. the configurations of $M$ at times $t_1$ and $t$ are identical), then there is a cycle in which the machine moves from one configuration to the next ultimately returning to the original configuration after $t - t_1$ steps. Since $M$ is deterministic such a cycle cannot be broken and therefore $M$'s computation will never end.

The last observation shows that during a computation in which $M$ stops, there are no such cycles and therefore no configuration is ever reached twice. It follows that the running time of such a machine is bounded by the number of possible configurations, so in order to bound the time it is enough to bound the number of possible configurations.

If a machine $M$ never uses more than $s$ cells, then on a given input $x$, the number of configurations is bounded by the number of possible contents of $s$ cells (i.e. $|\Gamma_M|^s$, where $\Gamma_M$ is the tape alphabet of machine $M$),times the number of possible locations of the work head (i.e. $s$), times the number of possible locations of the input head (i.e. $|x|$), times the number the possible states (i.e. $|S_M|$). If the number of cells used by a machine is a function of the input's length the same analysis holds and gives us a bound on the number of configurations as a function of the input's length.

For a given machine $M$ and input $x$, denote by $\#conf(M, x)$ the number of possible configurations of machine $M$ on input $x$. We have seen that for a machine $M$ that works in space $S(\cdot)$ on input $x$, $\#conf(M, x) = |\Gamma_M|^{S(|x|)} \cdot S(|x|) \cdot |x| \cdot |S_M| = 2^{O(S(|x|))} \cdot |x|$

Therefore in the context of the theorem (i.e. $S(|x|) = \Omega(log(|x|))$) we get that on input $x$ the time of $M$'s computation is bounded by : $\#conf(M, x) = 2^{O(S(|x|))}$ ■

**Fact 1.2** *For every function* $T(\cdot)$ $DTIME(T) \subseteq DSPACE(T)$.

**Proof:** Clearly no more then $T(|x|)$ cells can be reached by the machine's head in $T(|x|)$ steps. ■

Note : In the (far) future we will show a better bound (i.e. $DTIME(T) \subseteq DSPACE(\frac{T}{log(T)})$) which is non-trivial.

# 2 Non-Deterministic space complexity

## 2.1 Definition of models

During our discussion on $\mathcal{NP}$ we noticed that the idea of a non-deterministic Turing machine can be formalized in two approaches, the first approach is that the transition function of the machine is non-deterministic (i.e. the transition function is a multi-valued function), in the second approach the transition function is deterministic but in addition to the input the machine gets an extra string (viewed as a guess): the machine is said to accept input $x$ iff there exists a guess $y$ s.t. the machine's computation on (x,y) ends in an accepting state. (In such a case $y$ is called a witness for $x$).

In this section we shall try to generalize these approaches and construct a model suitable for measuring non-deterministic space complexity. The first approach can be applied to our standard turing machine model.
Put formally, the definition of a non-deterministic Turing machine under the first approach is as follows :

**Definition 2** (non-deterministic Turing machine): *A* non deterministic Turing machine *is a Turing machine with a non-deterministic transition function, having a work tape, a read-only input tape, and a unidirectional write-only output tape. The machine is said to accept input $x$ if there exists a computation ending in an accepting state.*

Trying to apply the second approach in the context of space complexity a natural question arises : should the memory used to hold the guess be metered?
It seems reasonable not to meter that memory as the machine does not "really" use it for computation. (Just as the machine does not "really" use the memory that holds the input). Therefore a special kind of memory (another tape) must be dedicated to the guess and that memory would not be metered. However if we do not meter the machine for the guess memory, we must restrict the access to the guess tape, just as we did in the case of the input tape. (surely if we allow the machine to write on the guess tape without being metered and that way get "free" auxiliary memory that would be cheating).
It is clear that the access to the guess tape should be read only.

**Definition 3** (offline non-deterministic Turing machine): *An* offline non-deterministic Turing machine *is a Turing machine with a work tape, a read-only input tape, a two-way read-only guess tape, and a unidirectional write-only output tape, where the contents of the guess tape is selected non-deterministically. The machine is said to accept input $x$ if there exists contents to the guess tape (a guess string $y$) s.t. when the machine starts working with $x$ in the input tape and $y$ in the guess tape it eventually enters an accepting state.*

As was made explicit in the definition, there is another natural way in which access to the guess tape can be farther limited: the tape can be made unidirectional (i.e. allow the head to move only in one direction).

**Definition 4** (online non-deterministic Turing machine): *An* online non-deterministic Turing machine *is a Turing machine with a work tape,a read-only input tape, a unidirectional read-only guess tape (whos contents are selected non-deterministicly), and a unidirectional write-only output tape. Again, the machine is said to accept $x$ if there exists a guess $y$ s.t. the machine working on (x,y) will eventually enter an accepting state.*

An approach that limits the guess tape to be unidirectional seems to correspond to an online guessing process – a non-deterministic machine works and whenever there are two (or more) possible ways to continue the machine guesses (online) which way to choose. If such a machine "wants" to "know" which way it guessed in the past, it must record it's guesses (use memory). On the other hand, the approach that allows the guess tape to be two-way corresponds to an offline guessing process i.e. all the guesses are given before hand (as a string) and whenever the machine wants to check what was guessed at any stage of the computation, it can look at the guesses list.

It turns out that the first non-deterministic model and the online model are equivalent. (Although the next claim is phrased for language decision problems, it holds with the same proof for other kinds of problems).

**Claim 2.1** *For every language $L$ there exists a non-deterministic Turing machine $M_N$ that identifies $L$ in time $O(T)$ and space $O(S)$ iff there exists an online Turing machine $M_{on}$ that identifies $L$ in time $O(T)$ and space $O(S)$.*

**Proof:** Given $M_N$ it can be easily transformed to an online machine $M_{on}$ in the following way: $M_{on}$ simulates $M_N$ and whenever $M_N$ has several options for a next move (it must choose non-deterministicaly which option to take), $M_{on}$ decides which option to take according to the content of the cell scanned at the guess tape, then move the guess tape head one cell to the right.

In some cases we may want to restrict the alphabet of the guess (for example to $\{0, 1\}$). In those cases there is a minor flaw in the above construction as the number of options for $M_N$'s next move may be bigger than the guess alphabet thus the decision which option to take cannot be made according to the content of a single guess tape cell. This is only an apparent flaw since we can assume with out loss of generality that $M_N$ has at most two options to choose from. Such an assumption can be made since a choice from any number of options can be transformed to a sequence of choices from two options at a time by building a binary tree with the original options as leaves. This kind of transformation can be easily implemented on $M_N$ by adding states that correspond to the inner nodes of the tree. The time of the transformed machine has increased at most by a factor of the height of the tree which is constant in the input size.

The transformation from an online machine $M_{on}$ to a non-deterministic machine is equally easy: If we would have demanded that the guess head of $M_{on}$ must advance every time, the construction would have been trivial i.e. at every time $M_{on}$ moves according to it's state and the contents of the cells scanned by the input-tape, work-tape and guess-tape heads, if the contents of the guess cell scanned are not known there may be several moves possible (one for each possible guess symbol), $M_N$ could have simply choose non-deterministically between those. However as we defined it, the guess tape head may stay in place, in such a case the non-deterministic moves of the machine are dependendent (are fixed by the same symbol) untill the guess head moves again. This is not a real problem, all we have to do is remember the current guess symbol, i.e. $M_N$ states would be $S_{M_{on}} \times \Sigma$ where $S_{M_{on}}$ is $M'_{on}s$ states and $\Sigma$ is the guess alphabet, ($M_N$ being in state $(s, a)$ corresponds to $M_{on}$ being in state $s$ while it's guess head scans $a$). The transition function of $M_N$ is defined in the natural way. Suppose $M_N$ is in state $(s, a)$ and scans symbols $b$ and $c$ in it's work and input tapes, this correspond to $M_{on}$ being in state $s$ while scanning $a$,$b$ and $c$. In this case $M_{on}$ transition function is well defined, (denote the new state by $s'$), $M_N$ will move the work and input heads as $M_{on}$ moves it's heads, if the guess head of $M_{on}$ stays fixed then the new state of $M_N$ is $(s', a)$, otherwise $M_{on}$ reads a new guess symbol, so $M_N$ chooses non-deterministically a new state of the form $(s', a')$ (i.e. guesses what is read from the new guess tape cell). ■

These models define complexity classes in a natural way. In the following definitions $M(x,y)$ should be read as "the machine $M$ with input $x$ and guess $y$".

**Definition 5** ($NSPACE_{on}$): *For any function $T : \mathsf{N} \to \mathsf{N}$*

$$NSPACE_{on}(T) \stackrel{\text{def}}{=} \left\{ L \subseteq \Sigma^* \;\middle|\; \begin{array}{l} \textit{There exists an online Turing machine } M_{on} \textit{ s.t. for any input } x \in \Sigma^* \\ \textit{there exists a witness } y \in \Sigma^* \textit{ for which } M_{on}(x,y) \textit{ accepts iff } x \in L, \\ \textit{and that for any } y \in \Sigma^* \; M_{on} \textit{ uses at most } T(|x|) \textit{ space.} \end{array} \right\}$$

**Definition 6** ($NSPACE_{off}$): *For any function $T : \mathsf{N} \to \mathsf{N}$*

$$NSPACE_{off}(T) \stackrel{\text{def}}{=} \left\{ L \subseteq \Sigma^* \;\middle|\; \begin{array}{l} \textit{There exists an offline Turing machine } M_{off} \textit{ s.t. for any input } x \in \Sigma^* \\ \textit{there exists a witness } y \in \Sigma^* \textit{ for which } M_{off}(x,y) \textit{ accepts iff } x \in L, \\ \textit{and that for any } y \in \Sigma^* \; M_{off} \textit{ uses at most } T(|x|) \textit{ space.} \end{array} \right\}$$

## 2.2 Relations between $NSPACE_{on}$ and $NSPACE_{off}$

In this section the exponential relation between $NSPACE_{on}$ and $NSPACE_{off}$ will be established.

**Theorem 7** *For any function $S : \mathsf{N} \to \mathsf{N}$ so that $S$ is at least logarithmic and $\log S$ is space constructible,*
*$NSPACE_{on}(S) \subseteq NSPACE_{off}(log(S))$.*

Given an online machine $M_{on}$ that works in space bounded by $S$ we shall construct an offline machine $M_{off}$ which recognizes the same language as $M_{on}$ and works in space bounded by $O(log(S))$. We will see later (Theroem 8) the opposite relation i.e. given an offline machine $M_{off}$ that works in space $S$, one can construct an online machine $M_{on}$ that recognizes the same language and works in space $2^{O(S)}$.

The general idea of the proof is that if we had a full description of the computation of $M_{on}$ on input $x$, we can just look at the end of the computation and copy the result (many of us are familiar with the general framework from our school days). The problem is that $M_{off}$ does not have a computation of $M_{on}$ however it can use the power of non-determinism to guess it. This is not the same as having a computation, since $M_{off}$ cannot be sure that what was guessed is really a computation of $M_{on}$ on $x$. This has to be checked before copying the result. (The absence of the last stage caused many of us great troubles in our school days).

To prove the theorem all we have to show is that checking that a guess is indeed a computation of a space $S(\cdot)$-online machine can be done in $log(S(|x|))$ space. To do that we will first need a technical result concerning the length of computations of such a machine $M_{on}$, this result is obtained using a similar argument to the one used in the proof of Fact 1.1 ($DSPACE(S) \subseteq DTIME(2^{O(S)})$).

**Proof:** *(Theorem 7: $NSPACE_{on}(S) \subseteq NSPACE_{off}(log(S))$):*
Given an online machine $M_{on}$ that works in space bounded by $S$ we shall construct an offline machine $M_{off}$ which recognize the same language as $M_{on}$ and works in space bounded by $O(log(S))$. Using claim 2.1, there exists a non-deterministic machine $M_N$ equivalent to $M_{on}$, so it is enough to construct $M_{off}$ to be equivalent to $M_N$.

As in the proof of Fact 1.1 ($DSPACE(S) \subseteq DTIME(2^{O(S)})$) we would like to describe the state of the computation by a configuration. (As $M_N$ uses a different model of computation we must redefine configuration to capture the full description of the computation at a given moment, however after re-examination we discover that the state of the computation in the non-deterministic model is fully captured by the same components i.e. the contents of the work tape, the location of the work and input tape heads and the state of the machine, so the definition of a configuration can remain the same).

**Claim 2.2** *If there exists an accepting computation of $M_N$ on input $x$ then there exists such a computation in which no configuration appears more than once.*

**Proof:** Suppose that $c_0, c_1, \ldots, c_n$ is a description of an accepting computation as a sequence configurations in which some configuration appear more than once. We can assume, without loss of generality that both $c_0$ and $c_n$ appear only once. Assume for $0 < k < l < n$, $c_k = c_l$. We claim that $c_0, \ldots, c_k, c_{l+1}, \ldots, c_n$ is also a description of an accepting computation. To prove that, one has to understand when is a sequence of configurations a description of an accepting computation, This is the case if the following hold :

1. The first configuration (i.e. $c_0$) describes a situation in which $M_N$ starts a computation with input $x$ (initial state, the work tape empty).
2. Every configuration $c_j$ is followed by a configuration (i.e. $c_{j+1}$) that is possible in the sense that, $M_N$ may move in one step from $c_j$ to $c_{j+1}$.
3. The last configuration (i.e. $c_n$) describes a situation in which the $M_N$ accepts.

When $c_{k+1}, \ldots, c_l$ (the cycle) is removed properties 1 and 3 do not change as $c_0$ and $c_n$ remain the same. Property 2 still holds since $c_{l+1}$ is possible after $c_l$ and therefore after $c_k$.
$c_0, \ldots, c_k, c_{l+1}, \ldots, c_n$ is a computation with a smaller number of identical configurations and clearly one can iterate the process to get a sequence with no identical configurations at all. ■

Remark : The proof of the last claim follows a very similar reasoning to the proof of Fact 1.1 ($DSPACE(S) \subseteq DTIME(2^{O(S)})$), but with an important difference. In the context of non-determinism it is possible that a computation of a given machine is arbitrarily long (the machine can enter a loop and leave it non-deterministicaly). The best that can be done is to prove that short computations exist.

We saw that also arbitrarily long computations may happen, these computations do not add power to the model since the same languages can be recognized if we forbid long computations. A similar question may rise regarding infinite computations. A machine may reject either by halting in a rejecting (non-accepting) state, or by entering an infinite computation, it is known that by demanding that all rejecting computations of a turing machine will halt, one reduces the power of the model (the class R as opposed to RE), the question is is the same true for space bounded machines ? It turns out that this is not the case (i.e. we may demand with out loss of generality that every computation of a space bounded machine halts). By Claim 2.2 machine that works in space $S$ works in time $2^{O(S)}$, we can transform such a machine to a machine that always halts by adding a time counter that counts untill the time limit has passed and then halts in a rejecting state (time out). Such a counter would only cost $log(2^{O(S)}) = O(S)$ so adding it does not change the space bound significantly.

*Now we have all we need to present the idea of the proof.*

Given input $x$ machine $M_{off}$ will guess a sequence of at most $\#conf(M, x)$ of configurations of $M_N$, and then check that it is indeed an accepting computation by verifying properties 1–3 (in the proof of Claim 2.2). If the guess turns out to be an accepting computation, $M_{off}$ will accept otherwise reject.

*How much space does $M_{off}$ need to do the task?*

The key point is that in order to verify these properties $M_{off}$ need only look at 2 consecutive configurations at a time and even those are already on the guess tape, so the work tape only keeps a fixed number of counters (pointing to the interesting cell numbers on the guess and input tapes).

$M_{off}$ treats it's guess as if it is composed of blocks, each contains a configuration of $M_{on}$.
To verify property 1, all $M_{off}$ has to do is check that the first block (configuration) describes an initial computational state i.e. check that $M_N$ is in the initial state and that the work tape is empty. That can be done using $O(1)$ memory.
To verify property 2 for a specific couple of consecutive configurations $M_{off}$ has to check that the contents of the work tape in those configurations is the same except perhaps the cell on which $M_N$'s work head was, that the content of the cell the head was on, the state of the machine and the new location of the work head are the result of a possible move of $M_N$. To do that $M_{off}$ checks that these properties hold for every two consecutive blocks on the guess tape. This can be done using a fixed number of counters (each capable of holding integers upto the length of a single block) + $O(1)$ memory.
To verify property 3 all $M$ has to do is to verify the last block (configuration) describes an accepting configuration. That can be done using $O(1)$ memory.
All that is left is to calculate the space needed to hold a counter. This is the maximum between log the size of a configuration and $log(|x|)$. A configuration is composed of the following parts :

- The contents of the work-tape – $O(S(|x|))$ cells
- The location of the work head – $log(O(S(|x|)))$ cells
- The state of the machine $M_N$ – $O(1)$ cells
- The location of the input head – $O(log(|x|))$ cells

Since $S$ is at least logaithmic, the length of a configuration is $O(S(|x|))$, and the size of a counter which points to location in a configuration is $O(1) + \log(S(|x|)))$.

*Comment*: Two details which were omitted are (1) the low-level implementation of the verification of property 2, and (2) dealing with the case that the guess is not of the right form (i.e., does not consists of a sequence of configurations of $M_{on}$). ■

**Theorem 8** *For any space constractable function $S : \mathsf{N} \rightarrow \mathsf{N}$ which is at least logarithmic.* $NSPACE_{off}(S) \subseteq NSPACE_{on}(2^{O(S)})$.

As in the last theorem, given a machine of one model we would like to find a machine of the other model accepting the same language. This time an offline machine $M_{off}$ is given and we would like to construct an online machine $M_{on}$.

In such a case the naive approach is simulation, i.e. trying to build a machine $M_{on}$ that simulates $M_{off}$. This approach would not give us the space bound we are looking for, however, trying to follow that approach will be instructive, so that is what we will do.

The basic approach is to try and simulate $M_{off}$ by an online machine $M_{on}$ (in the previous theorem we did even better than that by guessing the computation and only verifying it's correctness (that way the memory used to hold the computation was free). This kind of trick will not help us here because the process of verification involves comparing two configurations and in an online machine that would force us to copy a configuration to the work tape. Since holding a configuration on the work tape costs $O(S(|x|))$ space we might as well try to simulate $M_{off}$ in a normal way).

Since we only have an online machine which cannot go back and forth on the guess tape, the straightforward approach would seem to be : guess the content of a guess tape for $M_{off}$ then copy it to the work tape of the online machine $M_{on}$. That gives $M_{on}$ two way access to the guess and now $M_{on}$ can simulate $M_{off}$ in a straight forward way. The only question remains how much space would be needed ? (clearly at least as long as the guess)

The length of the guess can be bounded using a similar analysis to the one we saw at Fact 1.1 ($DSPACE(S) \subseteq DTIME(2^{O(S)})$), only this time things are a bit more complicated.

If we look on $M_{off}$'s guess head during a computation it moves back and forth thus it's movement forms a "snake like path" over the guess tape.

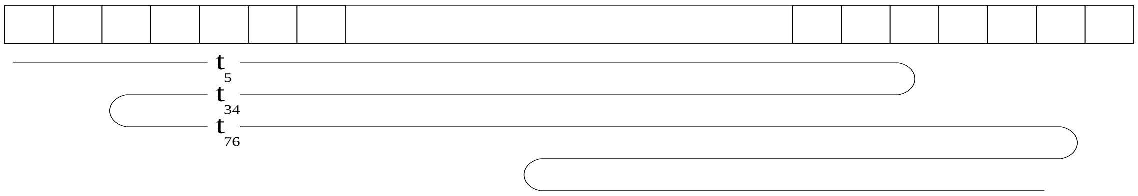


Figure 1: The guess head movement

The guess head can visit a cell on the guess tape many times, but we claim the number of times a cell is visited by the head can be bounded. The idea is, as in Fact 1.1, that a machine cannot be in the exact same situation twice without entering an infinite loop.

To formalize the last intuition we would need a notion of configuration (a machine's exact situation) this time for an offline machine. To describe in full the computational state of an offline machine one would have to describe all we described in a deterministic model (contents of work tape, location of work and input head and the machine state) and in addition the contents of the guess tape and the location of the guess head. However we intend to use the configuration notion for a very specific purpose, in our case we are dealing with a specific cell on the guess tape while the guess is fixed. Therefore denote by *CWG* (configuration without guess) of $M_{off}$ its configuration without the the guess tape contents and the guess head location. (exactly the same components as in the non-deterministic configuration). Once again the combinatorial analysis shows us that the number of possible CWGs is $|\Gamma|^{S(|x|)}S(|x|)|S_M|log(|x|)$ which is equal to $\#conf(M,x)$.

**Claim 2.3** *The number of times during an accepting computation of $M_{off}$ in which the guess tape head visits a specified cell is lesser or equal to $\#conf(M,x)_M = 2^{O(S)}$.*

**Proof:** If $M_{off}$ visits a single cell twice while all the parameters in the CWG (contents of work tape, location of work and input head and state of the machine) are the same then the entire computation state is the same, because the contents of the guess tape and the input remains fixed throughout the computation. Since $M_{off}$'s transition function is deterministic this means that $M_{off}$ is in an infinite loop and the computation never stops.

Since $M_{off}$ uses only $S(|x|)$ space there are only $\#conf(M,x)$ possible CWGs and therefore $\#conf(M,x)$ bounds the number of times the guess head may return to a specified cell. ■

Now we can (almost) bound the size of the guess.

**Claim 2.4** *If for input $x$ there exists a guess $y$ s.t. the machine $M_{off}$ stops on $x$ with guess $y$, then there exists such a guess $y$ satisfying $|y| < |\Gamma| \cdot \#conf(M,x)^{\#conf(M,x)} = 2^{2^{O(S(|x|))}}$.*

**Proof:** Denote the guess tape cells $c_0 c_1 \ldots c_{|y|}$ and their content $y = g_o \ldots g_{|y|}$. Given a computation of $M_{off}$ and a specified cell $c_i$ the guess head may have visited $c_i$ several times during the computation, each time $M_{off}$ was in another CWG. We can associate with every cell $c_i$ the sequence of CWGs $M_{off}$ was in when it visited $c_i$, denote such a sequence by *visiting sequence of* $c_i$. (Thus the first CWG in the visiting sequence of $c_i$ is the CWG $M_{off}$ was in the first time the guess head visited $c_i$, the second CWG in the visiting sequence is the CWG $M_{off}$ was in the second time the guess head visited $c_i$ and so on). By the last claim we get that the length of a visiting sequence is between 0 and $\#conf(M,x)$.

Suppose that for $k < l$, $c_k$ and $c_l$ both have the same visiting sequence and the same content i.e. $g_k = g_l$. Then the guess $g_0, \ldots, g_k, g_{l+1}, \ldots, g_{|y|}$ is also a guess that will cause $M_{off}$ to accept input $x$. The idea is the same as we saw in the proof of Claim 2.2, i.e. if there are two points in the computation in which the machine is in the exact same situation, then the part of the computation between these two points can be cut off and the result would still be a computation of the machine. To see that this is the case here, we need just follow the computation, when the machine first tries to move from cell $c_k$ to cell $c_{k+1}$ (denote this time $t_1^k$) it's CWG is the same CWG that describes the machine's state when first moving from cell $c_l$ to $c_{l+1}$ (denote this time $t_1^l$) therefore we can "skip" the part of the computation between $t_1^k$ and $t_1^l$ and just put the guess head on $c_{l+1}$ and still have a "computation" (the reason for the quatation marks is that normal computations do not have guess head teleportations). By similar reasoning whenever the machine tries to move from $c_{l+1}$ to $c_l$ (or from $c_k$ to $c_{k+1}$) we can just put the guess head on $c_k$ (respectively $c_{l+1}$) and "cut off" the part of the computation between the time it moved from $c_{l+1}$ to the correponding time it arrived at $c_k$ (respectively $c_k$ and $c_{l+1}$). If we would have done exactly that i.e. always "teleporting" the head and cutting the middle part of the computation, we would get a "computation" in which the guess head never entered the part of the guess tape between $c_k$ and $c_{l+1}$ so actually we would have a real computation (this time with out the quotation marks) on the guess $g_0 g_1 \ldots g_k g_{l+1} g_{l+2} \cdots g_{|y|}$.

Since we can iterate cut and paste process until we get a guess with no two cells with identical visiting sequences and content, we can assume the guess contains no two such cells.

There are $\#conf(M,x)$ possible CWGs therefore $\#conf(M,x)^n$ sequences of $n$ CWGs. Each visiting sequence is a sequence of CWGs of length at most $\#conf(M,x)$ so over all there are $\sum_{i=1}^{\#conf(M,x)} \#conf(M,x)^i \leq \#conf(M,x) \cdot \#conf(M,x)^{\#conf(M,x)} = \#conf(M,x)^{\#conf(M,x)+1} = 2^{2^{O(S(|x|))}}$ possibilities for a visiting sequence. Multiplied by the $|\Gamma|$ possibilities for the guess itself at each guess tape cell, this bounds the length of our short guess. ■

We have succeeded in bounding the length of the guess and therefore the space needed to simulate $M_{off}$ in an online machine using a straightforward approach. Unfortunately the bound is a double exponential bound and we want better. The good news is that during the analysis of the naive approach to the problem we have seen almost all that is necessary to prove Theorem 8.

**Proof:** *(Theorem 8: $NSPACE_{off}(S) \subseteq NSPACE_{on}(2^{O(S)})$.):*
Given an offline machine $M_{off}$ we shall construct an online machine $M_{on}$ that accepts the same language.

In the proof of the last claim (bounding the length of the guess) we saw another way to describe the computation. If we knew the guess, instead of a configuration sequence (with time as an index), one can look at a sequence of visiting sequences (with the guess tape cells as index). Therfore if we add the contents of the guess cell to each visiting sequence, the sequence of the augumented visiting sequences would describe the computation.

Our online machine $M_{on}$ will guess an $M_{off}$ computation described in the visiting sequences form and check whether indeed the guess is an accepting computation of $M_{off}$ (accept if so, reject otherwise). The strategy is very similar to what was done in the proof of Theorem 7 (where an offline machine guessed a computation of an online machine and verified it).

To follow this strategy we need to slightly augment the definition of a visiting sequence. Given a computation of $M_{off}$ and a guess tape cell $c_i$ denote by *directed visiting sequence (DVS) of* $c_i$ :

- The content of the guess cell $c_i$
- The visiting sequence of $c_i$
- For every CWG in the visiting sequence, the direction from which the guess head arrived to the cell (either R, L or S standing for Right, Left or Stay)

We shall now try to characterize when a string of symbols represents an accepting computation in this representation.

A DVS has the *reasonable returning direction property* if : whenever according to a CWG and cell content the guess head should move right, then the direction associated with the next CWG (returning direction) is left. (respectively the returning direction from a left head movement is right, and from staying is stay).

An ordered pair of DVSs is called *locally consistent* if they appear as if they may be consecutive in a computation i.e. whenever according to the CWG and the guess symbol in one of the DVSs the guess head should move to the cell that the other DVS represents then the CWG in the other DVS that corresponds to the consecutive move of $M_{off}$ is indeed the CWG $M_{off}$ would be in according to the transition function. (The corresponding CWG is well defined because we can count how many times did the head leave the cell of the first DVS in the direction of the cell of other DVS and the corresponding CWG can be found by counting how many time sthe head arrived from that direction). In addition to that, both DVSs must be first entered from the left, and both must have the reasonable returning property.

What must be checked in order to verify a candidate string is indeed an encoded computation of $M_{off}$ on input $x$ ?

1. The CWG in the first DVS is describing an initial configuration of $M_{off}$.
2. Every two consecutive DVSs are locally consistent.
3. In some DVS the last CWG is describing an accepting configuration.
4. In the last (most right) DVS, there is no CWG that according to it and the symbol on the guess tape the guess head should move to the right.

$M_{on}$ guesses a sequence of DVSs and checks the properties 1–4. To do that, $M_{on}$ never has to hold more then two consecutive DVSs + $O(1)$ memory. Since by Claim 2.4 the space needed for a DVS is $log(2^{2^{O(S(|x|))}}) = 2^{O(S(|x|))}$, $M_{on}$ works in space $2^{O(S(|x|))}$. ■

The online model is considered more natural for measuring space complexity (and is equivalent to the first formulation of a non-deterministic Turing machine), therefore it is considered the standard model. In the future we will use the following convention :
For any function $T : \mathsf{N} \to \mathsf{N}$, $NSPACE(T) \stackrel{\text{def}}{=} NSPACE_{on}(T)$.

## 3 Relations between Deterministic and Non-Deterministic space

### 3.1 Savitch's theorem

In this section we present the basic result regarding the relations between deterministic and non deterministic space complexity classes. It is easy to see that for any function $S : \mathsf{N} \rightarrow \mathsf{N}$, $DSPACE(S) \subseteq NSPACE(S)$ as deterministic machines are in particular degenerated non-deterministic machines. The question is how much can be "gained" by allowing non-determinism.

**Theorem 9** (Savitch): *For every space constractable function $S(\cdot)$ which is at least logarithmic $NSPACE(S) \subseteq DSPACE(S^2)$.*

For any non-deterministic machine $M_N$ that accepts $L$ in space $S$, we will show a deterministic machine $M$ that accepts $L$ in space $S^2$.

**Definition 10** ($M$'s configuration graph over $x$) : *Given a machine $M$ which works in space $S$ and an input string $x$, $M$'s configuration graph over $x$, $G_M^x$, is the directed graph in which the set of vertices is all the possible configurations of $M$ (with input $x$) and there exists a directed edge from $s_1$ to $s_2$ iff it is possible for $M$, being in configuration $s_1$, to change to configuration $s_2$.*

Using this terminology, $M$ is deterministic iff the out degree of all the vertices in $G_M^x$ is one.

Since we can assume without loss of generality that $M$ accepts only in one specific configuration (assume $M$ clears the work tape and move the head to the initial position before accepting), denote that configuration by $accept_M$ and the initial configuration by $start_M$. The question whether there exists a computation of $M$ that accepts $x$ can now be phrased in the graph terminology as "is there a directed path from $start_M$ to $accept_M$ in $G_M^x$".

Another use of this terminology may be in formulating the argument we have repeatedly used during the previous discussions : if there exists a computation that accept $x$ then there exists such a computation in which no configuration appears more than once. Phrased in the configuration graph terminology this reduces to the obvious statement that if there exists a path between two nodes in a graph then there exists a simple path between them. If $M$ works in space $S(|x|)$ then the number of nodes in $G_M^x$ is $|V_M^x| = \#conf(M, x)$ therefore if there exists a path from $start_M$ to $accept_M$ then there is one of length at most $|V_M^x|$.

We reduced the problem of whether $M$ accepts $x$ to a graph problem of the sort "is there a directed path in G from $s$ to $t$ which is at most $l$ long ?". This kind of problem can be solved in $O(log(|l|) \cdot log(|G|))$ space. (The latter is true assuming that the graph is given in a way that enables the machine to find the vertices and the vertices neighbors in a space efficient way, this is the case in $G_M^x$).

**Claim 3.1** *Given a graph $G = (V, E)$, two vertices $s, t \in V$ and a number $l$, in a way that solving the question of whether there exists an edge between two vertices can be done in $O(S)$ space, the question "is there a path of length at most $l$ from $s$ to $t$" can be answered in space $O(S \cdot log(l))$.*

**Proof:** If there is a path from $s$ to $t$ of length at most $l$ either there is an edge from $s$ to $t$ or there is a vertex $u$ s.t. there is a path from $s$ to $u$ of length at most $\lceil l/2 \rceil$ and a path from $u$ to $t$ of length at most $\lfloor l/2 \rfloor$. It is easy to implement a recursive procedure PATH$(a, b, l)$ to answer the question.

```
boolean PATH(a, b, l)
  if there is an edge from a to b then return TRUE
  (otherwise continue as follows :)
    for every vertex v
      if PATH(a, v, ⌈l/2⌉) and PATH(v, b, ⌊l/2⌋)
        then return TRUE
    otherwise return FALSE
```

*How much space does PATH($a, b, l$) use?*

When we call PATH with parameter $l$ it uses $O(S)$ space to store $a$, $b$ and $l$, check whether there is an edge from $s$ to $t$, and handle the for-loop control variable (i.e. $v$). In addition it invokes PATH twice with parameter $l/2$, but the key point is that both invocations use the same space (or in other words, the second invocations re-uses the space used by the first). Letting $W(l)$ denote the space used in invoking PATH with parameter $l$, we get the recursion $W(l) = O(S) + W(l/2)$, with end-condition $W(1) = O(S)$. The solution of this relation is $W(l) = O(S \cdot log(l))$.

(The solution is obvious because we add $O(S)$, $log(l)$ times (halving $l$ at every iteration, it will take $log(l)$ iterations to get to 1). The solution is also easily verified by induction, denote by $c_1$ the constant from the $O(S)$ and $c_2 = 2c_1$, the induction step : $W(l) \leq c_1 S + c_2 S \cdot log(l/2) = c_1 S + c_2 S log(l) - c_2 S = c_2 S log(l) + (c_1 - c_2)S$ and for $c_2 > c_1$ we get $W(L) \leq c_2 S log(l/2)$). ■

Now the proof of Savitch's theorem is trivial.

**Proof:** *(Theorem 9 (Savitch's theorem): $NSPACE(S) \subseteq DSPACE(S^2)$) :*
The idea is to apply Claim 3.1 by asking "is there a path from $start_M$ to $accept_M$ in $G_M^x$ ?" (we saw that this is equivalent to "does $M$ accept $x$"). It may seem that we cannot apply Claim 3.1 in this case since $G_M^x$ is not given explicitly as an input, however since the deterministic machine $M$ get $x$ as the input, it can build $G_M^x$ so $G_M^x$ is given implicitly. Our troubles are not over since storing all $G_M^x$ is too space consuming, but there is no need for that, our deterministic machine can build $G_M^x$ on the fly i.e. build and keep in memory only the parts it needs for the operation it performs now then reuse the space to hold other parts of the graph that may be needed for the next operations. This can be done since the vertices of $G_M^x$ are configurations of $M_N$ and there is an edge from $v$ to $u$ iff it is possible for $M_N$ being in configuration $v$ to change for configuration $u$, and that can easily be checked by looking at the transition function of $M_N$. Therefore If $M$ works in $O(S)$ space then in $G_M^x$ we need $O(S)$ space to store a vertex (i.e. a configuration), and $log(O(S))$ space to check if there is an edge between two stored vertices, all that is left is to apply the Claim 3.1. ■

## 3.2 Translation lemma

**Definition 11** (NL) *The complexity class* Non-Deterministic logarithmic space*, denoted $\mathcal{NL}$, is defined as $NSPACE(O(log(n)))$.*

Sometimes Savitch's theorem can be found phrased as:

$$\mathcal{NL} \subseteq DSPACE(log(n)^2).$$

This looks like a special case of the theorem as we phrased it, but is actually equivalent to it. What we miss in order to see the full equivalence is a proof that containment of complexity classes "translates upwards".

**Lemma 3.2** (Translation lemma): *Given $S_1, S_2, f$ space constractable functions s.t. $S_2(f)$ is also space constractible and $S_2(n) \geq log(n), f(n) \geq n$ then if $NSPACE(S_1(n)) \subseteq DSPACE(S_2(n))$ then $NSPACE(S_1(f(n))) \subseteq DSPACE(S_2(f(n)))$.*

Using the translation lemma, it is easy to derive the general Savitch's therorem from the restricted case of NL: Given that $\mathcal{NL} \subseteq DSPACE(log(n)^2)$, given a function $S(\cdot)$ choose $S_1(\cdot) = log(\cdot), S_2(\cdot) = log(\cdot)^2$ and $f(\cdot) = 2^{S(\cdot)}$ ($f$ would be constractible if $S$ was) now, applying the translation lemma, we get that $NSPACE(log(2^S)) \subseteq DSPACE(log(2^S)^2)$ which is equivalent to $NSPACE(S) \subseteq DSPACE(S^2)$.

**Proof:** Given $L \in NSPACE(S_1(f(n)))$ we must prove the existence of a machine $M$ that works in space $S_2(f(n))$ and accepts $L$.

The idea is simple, transform our language $L$ of non-deterministic space complexity $S_1(f)$ to a language $L^{pad}$ of non-deterministic space complexity $S_1$ by enlarging the input, this can be done by padding. Now we know that $L^{pad}$ is also of deterministic space complexity $S_2$. Since the words of $L^{pad}$ are only the words of $L$ padded, we can think of a machine that given an input pads it and then checks if it is in $L^{pad}$. The rest of the proof is just carrying out this program carefully while checking that we do not step out of the space bounds for any input.

There exists $M_1$ which works in space $S_1(f(n))$ and accepts $L$. Denote by $L^{pad}$ the language $L^{pad} \stackrel{\text{def}}{=} \{x\$^i | x \in L$ and $M_1$ accpets $x$ in $S_1(|x| + i)$ space.$\}$ where \$ is a new symbol.

We claim now that $L^{pad}$ is of non-deterministic space complexity $S_1$. To check whether a candidate string $s$ is in $L^{pad}$ we have to check that it is of form $x\$^j$ for some $j$ (that can be done using $O(1)$ space). If so (i.e. $s = x\$^j$), we have to check that $M_1$ accepts $x$ in $S_1(f(|x|+j))$ space and do that without stepping out of the $S_1$ space bound on the original input (i.e. $S_1(|s|) = S_1(|x|+j)$). This can be done easily by simulating $M_1$ on $x$ while checking that $M_1$ does not step over the space bound (the space bound $S_1(|x|+j)$ can be calculated since $S_1$ is space constructable). (The resulting machine is referred to as $M_2$.)

Since $L^{pad}$ is in $NSPACE(S_1)$ it is also in $DSPACE(S_2)$; i.e., there exists a deterministic machine $M_3$ that recognizes $L^{pad}$ in $S_2$ space.

Given the deterministic machine $M_3$ we will construct a deterministic machine $M_4$ that accepts the original $L$ in space $S_2(f)$ in the following way:

On input $x$, we simulate $M_3$ on $x\$^j$ for $j = 1, 2, \ldots$ as long as our space permits (i.e., using space at most $S_2(f(|x|))$, including all our overheads). This can be done as follows: If the head of $M_3$ is within $x$, $M_4$'s input head will be on the corresponding point in the input tape, whenever the head of $M_3$ leaves the $x$ part of the input, $M_4$ keeps a counter of $M_3$'s input head position (and supplies the simualted $M_3$ with either \$ or black as appropriate). Recall that we also keep track that $M_3$ does not use more that $S_2(f(|x|))$ (for that reason we need $S_2(f)$ to be constractible), and if $M_3$ tries to step out of this bound we will treat it as if $M_3$ rejected. If during our simulations $M_3$ accept so does $M_4$ otherwise $M_4$ rejects.

Basicly $M_4$ is trying to find a right $j$ that will cause $M_3$ to accept, if $x$ is not in $L$ then neither is $x\$^j$ in $L^{pad}$ (for any $j$) and therefore $M_3$ will not accept any such string untill $M_4$ will eventually reject $x$ (which will happen when $j$ is sufficiently large so that $\log j$ superseeds $S_2(f(|x|))$ which is our own space bound). If on the other hand $x$ is in $L$ than $M_1$ accepts it in $S_1(f(|x|))$ space therefore $M_3$ accepts $x\$^j$ for some $j \leq f(|x|) - |x|$ (since to hold $f(|x|) - |x|$ one needs only a counter of size $log(f(|x|)$ and $S_2$ is bigger then $log$ this counter can be kept within the space bound of $S_2(f(|x|))$ and $M_4$ will get to try the right $x\$^i$ and will eventually accept $x$). ■

**Remark:** In the last proof thre was no essential use of the model deterministic or non-deterministic, so by similar argument we can prove analogous results (for example, $DSPACE(S_1) \subseteq DSPACE(S_2)$ implies $DSPACE(S_1(f)) \subseteq DSPACE(S_2(f))$ ).

By a similar argument we may also prove analogous results regarding time complexity classes. In this case we cannot use our method of searching for the correct padding since this method (while being space efficient) is time consuming. On the other hand, under suitable hypothesis, we can can compute $f$ directly and so do not need to search for the righ padding. We define $L_2^{pad} = \{x\$^{f(|x|)-|x|} : x \in L\}$ and now $M_4$ can compute $f(|x|)$ and run $M_3$ on $x\$^{f(|x|)-|x|}$ in one try. There are two minor modifications that have to be done. Firstly, we assume all the functions involved $S_1, S_2, f \geq n$ (this is a reasonable assumption when dealing with time-complexity classes). Secondly, $M_2$ has to check whether the input $x\$^j$ is indeed $x\$^{f(|x|)-|x|}$; this is easy if it can compute $f(|x|)$ within it's time bounds (i.e., $S_1(|x\$^j|)$), but may not be the case if the input $x\$^j$ is much shorter than $f(|x|)$. To solve that, $M_2$ only has to time itself while computing $f(|x|)$ and if it fails to compute $f(|x|)$ within the time bound it rejects.