MTAT.07.004 — Complexity Theory September 29th, 2011

Lecture 5

Lecturer: Peeter Laud *Scribe(s): Alisa Pankova*

1 Introduction

In the previous lectures, we have looked through the time resource and its complexity classes. This lecture is about memory space resource. It introduces definitions of the main space complexity classes and the correlations between time and space.

2 Space Complexity

First of all, we will formally define space complexity classes. They are very similar to the definitions of time complexity classes.

A language $L \subseteq \{0,1\}^*$ belongs to the class DSPACE(f), if there exists a DTM M that accepts L, and a constant c, such that $M(x)$ writes to at most $c \cdot f(|x|)$ cells on its work tapes. The class NSPACE(f) is defined similarly for nondeterministic Turing machines.

Some standard classes:

- PSPACE= $\bigcup_{c\in\mathbb{N}}$DSPACE($\lambda n.n^c$)
- NPSPACE= $\bigcup_{c\in\mathbb{N}}$NSPACE($\lambda n.n^c$)
- L=DSPACE($\lambda n.\log n$)
- NL=NSPACE($\lambda n.\log n$)

Note that we did not have any logarithmic classes in case of time complexity. There was no sense to define them since even copying the input onto the output tape would already be linear. In the case of space complexity, we will make some exceptions in the definition in order to avoid some unreasonable excess in complexty estimation.

When we are talking about Turing machines, let us assume that we take into account only the working tapes. Otherwise, we would get that even a constant function has at least linear comlpexity. The input tape is always linear according to the input size since we write the whole input onto it.

Examples:

- Show that SAT has linear space (SAT $\in$ DSPACE($\lambda n.n$)).

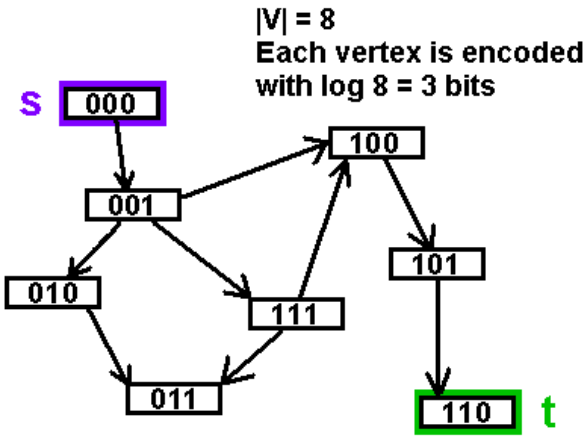
  Suppose that we are given a boolean formula with free variables $x_1, \cdots, x_n$. We need to look through all the possible evaluations of the variables. Each evaluation must be stored on a tape, and the length of evaluation is linear to the number of variables. We will also need to store the intermediate values of subformulas, but we need even less space for them, and the upper bound of memory space does rise above linear.

We do not need to store all the 2^n evaluations at once. After we compute the formula with one evaluation, we may erase the contents of the work tapes and try the next evaluation.

- Let PATH= $\{\langle G, s, t\rangle \mid G$ is directed graph with path from s to $t\} \in$NL.

  The Turing machine that decides PATH receives a directed graph $G = (V, E)$ and two vertices s and t as the input. It returns true iff there exists a path from s to t. We will show that PATH $\in$ NL.

  The space complexity directly depends on the number of vertices, since each vertex needs to be somehow encoded. We take $n = |V|$ as the complexity measurement. If we use binary encoding, we need $\lceil \log n \rceil$ memory cells to store one vertex.



  The graph itself can be encoded as a bitstring by its adjacency matrix representation. Although the contents of the input tape are squared according to the number of vertices, we decided not to consider the input tape in our space complexity definition.

  Now we try to describe an algorithm that holds in memory a constant number of vertices. Since each vertex requires $\lceil \log n \rceil$ cells, our memory will be still $O(\log n)$. Let $G(v)$ denote all the neighbours of a vertex $v \in G$. The algoritm will be the following:

  for i = n-1 **to** 0 **step** -1:
  if $s == t$ **return** true
  if $i == 0$ **return** false
  choose $v \in G(s)$
  $s := v$

  The most important thing in this algorithm is the "choose" operation. It is a non-deterministic operation, and we can use it since we are proving that PATH $\in$ NL. We may assume that each neighbour of the current vertex is being computed in a separate computational path.

  On each step, we need to store the values of i, s, and t. All of them are bounded by n, and therefore the length of each variable is bounded by $\lceil \log n \rceil$. We need $3\lceil \log n \rceil$ space, and it is in $O(\log n)$.

3 Computing a Function

Before we have considered the Turing machines that decide a language. In the end, they write either 0 or 1 on the output tape, and therefore do not add additional space complexity. Now suppose that we compute a function $f : \{0,1\}^* \to \{0,1\}^*$. We want to define that it is computable in space g. If we take into account all the work tapes, we get that the identity function $f(x) = x$ is linear because the contents of the output tape are linear. That is not a good way of evaluating functions since the identity function does not do anything! We need to somehow discount the ouptut tape. There are two ways of doing this:

1. Let us define a special ouptut tape. As the input tape has been read-only, the output tape will be write-only, and we forbid any rewriting of the output (the head is permitted to move only to the right). All the intermediate actions and comparisons will be done only on the work tapes.

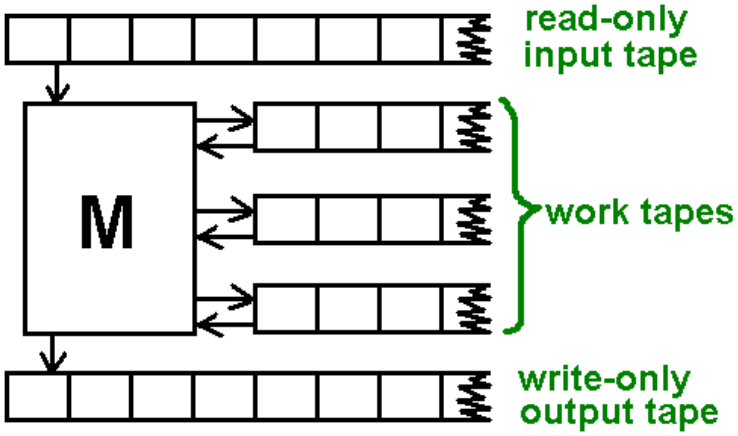
2. We may decide not to write the result anywhere at all and try to extract the information by bits. We define the following languages:

   - $L_i = \{x \,|\, |f(x)| \geq i\}$
   - $L'_i = \{x \,|\, x \in L_i \wedge i\text{-th bit of } f(x) \text{ is } 1\}$

   How we can get the value of $f(x)$? We start computing L'_1, L'_2, $\cdots$. If we get *acc*, it means that the i^{th} bit of $f(x)$ is 1. Otherwise, it is either 0 or the bit string has ended. We may call $L_i(x)$ in order to check if it has ended.

   If all these languages L_i and L'_i are in DSPACE(g), we get the whole value of $f(x)$ in DSPACE(g).

If we have $g \in \Omega(\lambda n.\, \log n)$, these two variants are actually the same. Suppose that we have a TM that has a separate input and output tapes, and all the work tapes compose another Turing machine M. We show that M can decide L'_i (and therefore L_i since L'_i uses it). This machine would take two arguments: x and i, but we will look only how it depends on $|x|$.



Suppose that we want to compute the i^{th} bit of $f(x)$. We cannot read it directly from the output tape since it is write-only. But we may start writing the bits onto a working tape using a counter. Initially, the counter equals to i. On each iteration, we decrease the

counter, erase the previous bit from the tape, and write a new bit instead. If $i == 0$, then M stops, and the bit on the work tape is the one that we are looking for. If $f(x)$ ends before the counter exceeds, we get that $x \notin L_i$.

We get that, although the output tape is write-only, we can still read its contents by deciding L'_i and L_i.

Why it would not work if $g \notin \Omega(\lambda n. \log n)$? The length of the input encoded as a bit string is $\log n$. If we want to decide L'_i for the last bit, we need to set the counter $i := \log n$. The TM cannot therefore use less cells than $\log n$.

4 Relations Between Time and Space Complexity Classes

In this section, we will cover relationships not only of different space classes, but also how time and space depend on each other.

Theorem 1 $DTIME(f) \subseteq DSPACE(f) \subseteq NSPACE(f) \subseteq \bigcup_{c \in \mathbb{N}} DTIME(\lambda n.c^{f(n)})$

Proof: Let us prove these relations one by one.

- $\mathbf{DTIME}(f) \subseteq \mathbf{DSPACE}(f)$. Suppose by contrary that there exists a language L such that $L \in \text{DTIME}(f)$, but $L \notin \text{DSPACE}(f)$. It means that there exists a TM M that decides L in time $O(f)$, but requires $\Omega(f)$ memory space. We need to do at least one step to write one bit into memory. If the number of bits is $\Omega(f)$, it means that the time that we spend to write this value is also $\Omega(f)$. We get a contradiction!

- $\mathbf{DSPACE}(f) \subseteq \mathbf{NSPACE}(f)$. It is directly implied by the definition of DSPACE and NSPACE. We use the fact that each DTM is a NTM.

- $\mathbf{NSPACE}(f) \subseteq \bigcup_{c \in \mathbb{N}} \mathbf{DTIME}(\lambda n.c^{f(n)})$. This relation requires a little bit more complex proof.

  Recall that each Turing machine can be represented as a graph, where vertices are all the possible configurations, and there exists an arc from C to C' iff the TM can go directly from C to C' by transition (C and C' are configurations). The adjacency matrix of this graph represents the transition relation of TM.

  Suppose that we are given a TM M that decides a language $L \in$NSPACE(f). Given an input x, we can check if it belongs to L just by examining the graph. We check whether it is possible to reach any accepting state from the initial configuration.

  What do we know about the size of the graph? How many possible configurations do we have for M? A configuration contains information about the state, the contents of the tapes, and the head positions. The number of different states is constant. The amount of different tape contents is upper-bounded by f since the language $L \in$ NSPACE. We get that the number of different configurations is $O(\lambda n.c^{f(n)})$ for some constant c, which depends only on M.

  We see that our graph search time is bounded by $O(\lambda n.c^{f(n)})$. We can perform this search deterministically, and even in the worst case it is still $O(\lambda n.c^{f(n)})$. Since we used graph search to decide L, and it holds for any $L \in$ NSPACE, we get that NSPACE(f) $\subseteq \bigcup_{c \in \mathbb{N}}$ DTIME($\lambda n.c^{f(n)}$). □

Theorem 2 *If $f \in o(g)$ then DSPACE(f) $\subsetneq$DSPACE(g).*

There has been a similar theorem for time hierarchy, where we proved that if $f \log f = o(g)$, then DTIME(f) $\subsetneq$DTIME(g). The proof would be analogical, and it has not been shown on the lecture.

A similar theorem exists about NSPACE(f) $\subsetneq$NSPACE(g). It could be simpler to prove it for time, but the proof similar to NTIME would also be sufficient.

Corollary 3 *$L \subseteq NL \subseteq P \subseteq NP \subseteq PSPACE$*

Now we are considering certain complexity classes. We will prove these relations one by one.

- **L $\subseteq$ NL**. This is directly imlplied by definition of L and NL: every DTM is a NTM.
- **NL $\subseteq$ P**. We will again use the graph representation here. In Theorem 1, we have shown that the number of possible configurations is bounded by $\lambda n.c^{f(n)}$, where f defines the space complexity. If we have a language $L \in$ NL, it means that the use of memory is bounded by $\lambda n.c' \log n$ for some constant c'. We want to show that $c^{f(n)} = c^{c' \log n}$ is a polynomial. We will do some mathematical reductions.

  $$c^{c' \log n} = c^{\log n^{c'}} = (2^{\log c})^{\log n^{c'}} = 2^{\log c \log n^{c'}} = 2^{\log n^{c \cdot log c'}} = n^{c' \cdot \log c}.$$

  As the result, we get a polynomial. We have shown that any $L \in$ NL can be decided in polynomial time.
- **P $\subseteq$ NP**. This has already been shown on the previous lectures.
- **NP $\subseteq$ PSPACE**. In Theorem 1, we have shown that DTIME(f) $\subseteq$ DSPACE(f). Take $f = \lambda n.p(n)$ (for a polynomial p). We get that P $\subseteq$ PSPACE. Why it holds also for NP?

  We will prove further a very important theorem that implies SAT $\in$ PSPACE. At the same time, in the previous lectures we have proved that SAT is NP-complete. If we take any language $L \in$ NP, we can reduce it polynomially to SAT. Since SAT $\in$ PSPACE, we get that $L \in$ PSPACE, because PSPACE is closed under polynomial reduction. This holds for any $L \in$ NP. □

In the previous lectures, we have shown that any TM that works in time T can be simulated in time $T \log T$, by means of the Universal Turing Machine. What about the space complexity of simulation?

- **Input/output tapes:** are not involved into space complexity.
- **Description tape:** its contents can be very long, but it depends only on the TM, not on the input.
- **State tape:** does not depend on the input.
- **Work tape:** here we do not need to write more information that we did in the initial Turing machine.

In total, we get that we do not have any increase in space complexity as we had in the case of time complexity of DTM.

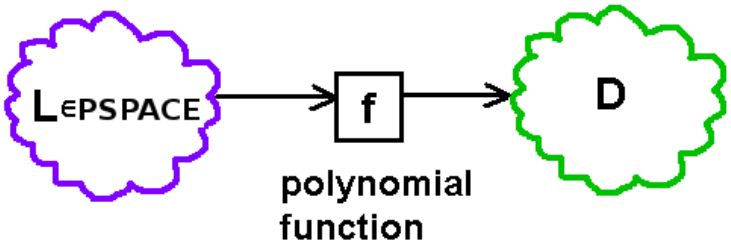
5 PSPACE-completeness

The definition is similar to NP-completeness that we have defined for time complexity classes. A language L is PSPACE-hard if for any $L' \in$ PSPACE we have $L' \leq_m^p L$. Language L is PSPACE-complete if it is PSPACE-hard and belongs to PSPACE.

Theorem 4 *The language* $D = \{\langle M, x, 1^n\rangle \mid$ *DTM* M *accepts* x *in space* $n\}$ *is PSPACE-complete.*

Proof: We will prove it in the similar way as we proved the existence of an NP-complete language. In order to show that D is PSPACE-complete, we need to show that $D \in PSPACE$, and that L is PSPACE-hard.

- D **is in PSPACE:** By definition of D, we check if M accepts x in space n. If the space grows larger, then M rejects x. The space is limited from the outside, and it does not depend on the input x.
- D **is PSPACE-hard:** let $L \in PSPACE$, and let M be the DTM that accepts L in space $O(p(|x|))$. We want to reduce L to D.



Take $f(x) = \langle M, x, 1^{p(|x|)}\rangle$. If M finishes its work on input x, accepts x, and the memory space does not exceed the limit $p(|x|)$, then the function returns true (accepted). On the other hand, if M rejects x or requires more than $p(|x|)$ space units, the function returns false (rejected). In this way we may reduce any language $L \in$ PSPACE to the language D. □

6 Quantified Boolean Formulas

We will give formal recursive defiition of a quantified boolean formula. It can be any boolean formula, and it may in addition use quantifiers. Here x is a variable, *op* is any binary boolean operation $(\vee, \wedge, \ldots)$, and Q is either $\forall$ or $\exists$.

- $\varphi = x$
- $\varphi = \neg\varphi'$
- $\varphi = \varphi_1 \text{ op } \varphi_2$
- $\varphi = Qx\,\varphi'$

When we are using quantifiers, we would like to distinguish free and bounded variables, since we may evaluate only free variables from outside. A variable is free if it does not belong to any quantifier. Here is a formal definition of a function FV that maps a boolean formula to the set of its free variables. It is also defined recursively.

- if $\varphi = x$, then $FV(\varphi) = \{x\}$
- if $\varphi = \neg\varphi'$, then $FV(\varphi) = FV(\varphi')$
- if $\varphi = \varphi_1 \operatorname{op} \varphi_2$, then $FV(\varphi) = FV(\varphi_1) \cup FV(\varphi_2)$
- if $\varphi = Qx\,\varphi'$, then $FV(\varphi) = FV(\varphi')\backslash\{x\}$

We just throw away all the variables that have been under a quantifier.

Now we would like to define evaluation of a boolean function. Let each boolean formula φ define a high-order function $[\![\varphi]\!]$. This function takes an evaluation function f as an argument and applies it to φ. The function f in order takes φ as an argument, finds all the free variables $x_1, \ldots, x_n$, and computes the value of φ by evaluating $x_1 \leftarrow b_1, \ldots, x_n \leftarrow b_n$, where $b_1, \ldots, b_n$ are boolean values defined inside f. The $[\![\varphi]\!]$ just returns the value that f has returned. As the result, the function $[\![\varphi]\!]$ applied to an evaluation returns the value of φ that has been computed by applying that evaluation to its free variables.

We can write it down more formally. Let x be a free variable, and let V be an evaluation function. Each boolean formula φ defines a Boolean function $[\![\varphi]\!]$ from $FV(\varphi) \to \mathbb{B}$ to $\mathbb{B}$, which is computed as follows:

- $[\![x]\!](V) = V(x)$
- $[\![\neg\varphi]\!](V) = \neg([\![\varphi]\!](V))$
- $[\![\varphi_1 \operatorname{op} \varphi_2]\!](V) = [\![\varphi_1]\!](V)\ [\![\mathrm{op}]\!]\ [\![\varphi_2]\!](V)$
- $[\![\forall x\,\varphi]\!](V) = [\![\varphi]\!](V[x \mapsto true]) \wedge [\![\varphi]\!](V[x \mapsto false])$. If the formula is true for any $x \in \{true, false\}$, we check if it is true for the evaluation V where x is set to true, and if it is true for the same evaluation V where x is set to false.
- $[\![\exists x\,\varphi]\!](V) = [\![\varphi]\!](V[x \mapsto true]) \vee [\![\varphi]\!](V[x \mapsto false])$. If there must exist an $x \in \{true, false\}$ such that the fomula is true, we check if it is true for the evaluation V where x is set either to true or false.

Now let us define a totally quantifiable boolean formula. It is a formula that does not have any free variables, but is nevertheless true without any evaluation. We may define a language TQBF that consists of all the totally quantifiable boolean formulas:

$$\text{TQBF}=\{x \text{ is QBF } \mid \text{FV}(x) = \emptyset, [\![x]\!]() = true\}.$$

All quantifiers can be moved to the front of the formula without increasing the length.

In order to check if a boolean formula is totally quantifiable, we need to look through all the values of bounded variables in the formula. In this way, SAT is an instance of TQBF. If we get a formula $\varphi(x_1, x_2, \ldots, x_n)$ and want to check if it is saisfiable, we just bind the free

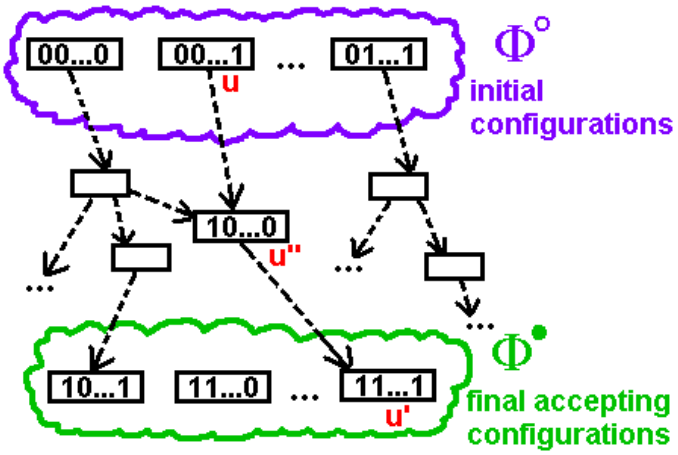
variables to existence quantifiers and check if $\exists x_1, \exists x_2, \ldots, \exists x_n \varphi(x_1, x_2, \ldots, x_n)$ is totally quantifiable.

In the case of time complexity, we have proved that SAT is NP-complete. In the same way, we are going to show that TQBF is PSPACE-complete.

Theorem 5 *TQBF is PSPACE-complete.*

According to the definition of PSPACE-completeness, we need to prove that TQBF $\in$ PSPACE, and that TQBF is PSPACE-hard.

- **TQBF $\in$ PSPACE**. We need to implement the function $[\![\cdot]\!]$. According to its definition, it works in polynomial space. Each value needs only one bit, and evem if the formula is long, we do not need to store all the intermediate values in memory.
- **TQBF is PSPACE-hard**. We need to show that all the languages in PSPACE are polynomially reducible to TQBF. We already know that the language $D = \{\langle M, x, 1^n \rangle \mid \text{DTM } M \text{ accepts } x \text{ in space } n\}$ is PSPACE-complete. We only need to show that we can reduce D to TQBF. This can be done by means of succinct representation of the computational graph.



First, note that each configuration is encoded as bits. Although we have $c^{p(|x|)}$ different configurations for the machine that decides TQBF, we need only $p(|x|)$ bits to encode a single configuration. We define the following:

- $S(u_1, \ldots, u_n)$: is the bit string $u_1 u_2 \ldots u_n$ a configuration or it does not make any sense?
- $R(u_1, \ldots, u_n, u'_1, \ldots, u'_n)$: can we go directly from configuration $C = u_1 \ldots u_n$ to the configuration $C' = u'_1 \ldots u'_n$ by just one transition?
- Φ° is the set of all initial configurations.
- $\Phi^\bullet$ is the set of all final configurations that accept the input.

The sizes of Φ° and $\Phi^\bullet$ depend on the input not more than polynomially. The sizes of S and R are bounded by number of possible configurations (the length of a single configuration). Everything is inside polynomial space.

In order to check if $x \in$ TQBF, we need to check if any accepting final state is reachable from the given initial state (which is determined by the input). We want to write the formula $R_i(u_1, \ldots, u_m, u'_1, \ldots, u'_m)$ expressing $C \xrightarrow{*} C'$ (C' is reachable from C indirectly) in at most 2^i steps.

We know that $2^i = 2^{i-1} \cdot 2^{i-1}$. Therefore, we may define R_i recursively:

- $R_0(u_1, \ldots, u_m, u'_1, \ldots, u'_m) =$
  $R(u_1, \ldots, u_m, u'_1, \ldots, u'_m) \vee (u_1 = u'_1 \wedge \cdots \wedge u_m = u'_m)$
  Meaning: we are already in this state, or it is reachable in 1 step.
- $R_i(u_1, \ldots, u_m, u'_1, \ldots, u') = \exists u''_1, \ldots, u''_m : S(u''_1, \ldots, u''_m)$
  $\wedge R_{i-1}(u_1, \ldots, u_m, u''_1, \ldots, u''_m) \wedge R_{i-1}(u''_1, \ldots, u''_m, u'_1, \ldots, u'_m)$
  Meaning: we check if there exists an intermediate state $u''_1, \ldots, u''_n$ that which is reachable from the source state $u_1, \ldots, u_n$, and if we can reach the target state $u'_1, \ldots, u'_n$ from there. Additionally, we have to check that it is an actual configuration, not a random bit string.

If we want to compute R_i, we need to write out all the previous calls of R_j (for $j < i$) and store them in memory. The space that we need to store R_i is $|R_i| = O(2^i) \cdot |R|$, and it is too much. We need to rewrite the R_i. If it incudes R_{i-1} only once, there will be no space problems, since it will be linear. We rewrite R_i as follows:
$R_i(u_1, \ldots, u_m, u'_1, \ldots, u'_m) =$
$\exists u''_1, \ldots, u''_m : S(u''_1, \ldots, u''_m) \wedge \forall v_1, \ldots, v_m, v'_1, \ldots, v'_m :$
$\Big(\Big(\bigwedge_{k=1}^m v_k = u_k \wedge \bigwedge_{k=1}^m v'_k = u''_k\Big) \vee \Big(\bigwedge_{k=1}^m v_k = u''_k \wedge \bigwedge_{k=1}^m v'_k = u'_k\Big)\Big)$
$\Rightarrow R_{i-1}(v_1, \ldots, v_m, v'_1, \ldots, v'_m)$

As the result, we check the both cases by encoding them into conjunctions. Before that we have "and" ($\wedge$) and therefore had to check if both calls of R_{i-1} can be true at once. Now we have "or" ($\vee$), and therefore may check them separately. We get space $O(i) \cdot |R|$.

We are ready to decide D in polynomial space! Suppose that we are given M, x, and 1^n. We construct the formulas S, R, Φ°, and $\Phi^\bullet$. Our goal is to decide if x accepts x in space n. Take n' such that M has $\leq 2^{n'}$ configurations of size n. We get that $n' = p(n)$ for some polynomial p since the total number of configurations is $2^{p(n)}$. In order to decide x, we compute the following:
$\exists u_1, \ldots, u_m, u'_1, \ldots, u'_m : \Phi^\circ(u_1, \ldots, u_m) \wedge \Phi^\bullet(u'_1, \ldots, u'_m) \wedge R_{n'}(u_1, \ldots, u_m, u'_1, \ldots, u'_m)$
In other words, we ask: is there a path that starts from the Φ°, ends in $\Phi^\bullet$, and whose length is bounded by $2^{n'}$ steps. We know that n' is actually a polynomial. Let's count the space that we need to compute this query:

- $\Phi^\circ(u_1, \ldots, u_m) \to$ polynomial
- $\Phi^\bullet(u_1, \ldots, u_m) \to$ polynomial
- $R_{n'}(u_1, \ldots, u_m, u'_1, \ldots, u'_m) \to O(i) \cdot |R| =$ linear $\cdot$ polynomial

In total we get: polynomial $\cdot$ polynomial $\cdot$ linear $\cdot$ polynomial = polynomial. We have polynomially reduced D to TQBF. We may now claim that TQBF is PSPACE-complete. $\square$

7 Relations Between PSPACE and NPSPACE

Theorem 6 *TQBF is NSPACE-complete.*

Since TQBF$\in PSPACE$, we also have that TQBF$\in NPSPACE$. We only need to show that all the languages in NSPACE can be polynomially reduced to TQBF. The proof of this theorem is analogical to the previous one, but instead of a DTM we would have a NTM. There is no detailed proof of this theorem in this lecture. Actually, the previous proof nowhere used the fact that the Turing machine it encoded was deterministic. The determinism was present only in the formula R and such formula can be written also if the machine is non-deterministic (see the proof that $SAT \in NP$). □

We have reached an interesting result. We know that TQBF$\in PSPACE$. On the other hand, it is NPSPACE-complete. It means that we can solve all the problems in $NSPACE$ by solving the corresponding problem in TQBF, that belongs to $PSPACE$. It means that PSPACE=NPSPACE.

Theorem 7 *(Savitch's theorem) PATH$\in$ DSPACE* $(\lambda n.\log^2 n)$.

Let the number of vertices in the input graph be n. Each vertex can be encoded by $\lceil \log n \rceil$ bits. Take $i = \lceil \log n \rceil$. We get an initial state s, a final state t, and check if we can reach the final state in space $2^{\lceil \log n \rceil}$. We define the function $\mathsf{REACH}(u, v, i)$ that returns true iff there is a path from u to v of length at most 2^i.
$\mathsf{REACH}(u, v, i) = \exists w : \mathsf{REACH}(u, w, i-1) \wedge \mathsf{REACH}(w, v, i-1)$.

The stack frames that will be collected by invocation of REACH:

$\mathbf{REACH}(s_0, t_0, \lceil \log n \rceil)$
$\mathbf{REACH}(s_1, t_1, \lceil \log n \rceil - 1)$
...
$\mathbf{REACH}(s_{\lceil \log n \rceil}, v_{\lceil \log n \rceil}, 0)$

In total, we have $\log n$ stack frames. Each frame keeps three values. Among them, s and t need $\lceil \log n \rceil$ bits of memory. The intermediate vertex w also needs $\lceil \log n \rceil$ bits. Additionally, we store the value i, and since it may have only $\lceil \log n \rceil$ different values, we need only $\lceil \log(\log n) \rceil$ bits to store it. In total, we need $(3 \log n + \log(\log n)) \log n \in O(\log^2 n)$ bits. □

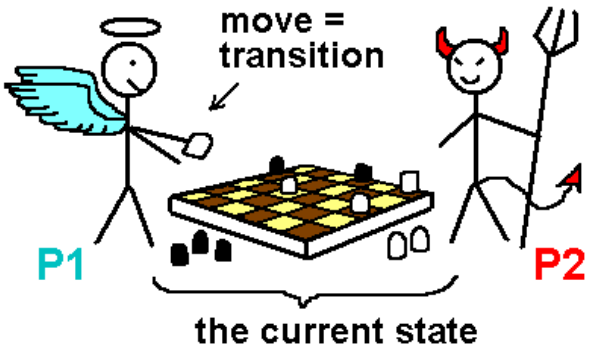
Corollary 8 *NSPACE*$(f) \subseteq$ *DSPACE*$(\lambda n. f(n)^2)$ *for any space-constructible function* $f \in \Omega(\lambda n. \log n)$.

If we consider this graph as the transition relation of a Turing machine that computes a function f, then its number of configurations is bounded by $c^f(n)$ for some constant c. We get that $O(\log^2 |V|) = O(\log^2(c^f(n))) = O(f(n)^2)$. In general, for any $f \in \Omega(\log n)$, NSPACE$(f) \subseteq$DSPACE(f^2).

8 PSPACE and Game-Playing

When we were talking about NP, it was about search problem. In a similar way, PSPACE can be associated with game-playing. We can imagine a game with two players with perfect

information, where each player knows everything about the current state of the game (for example, sees the entire game board). There are starting states, transitions (legal moves), and the possible final states that indicate who has won and who has lost.



There is one question that we may want to ask: does there exist a winning strategy for the player P1? A winning strategy means that whatever moves his opponent P2 performs, it is still possible for P1 to win the game.

$$\exists \text{ my move } \forall \text{ opponent's move } \exists \text{ my move } \ldots \text{ I win!}$$

This formula states that P1 will definitely win if he knows how to play. P2 cannot do anything to prevent it, and the only way for P1 to lose is to make wrong moves. Nothing depends on P2.