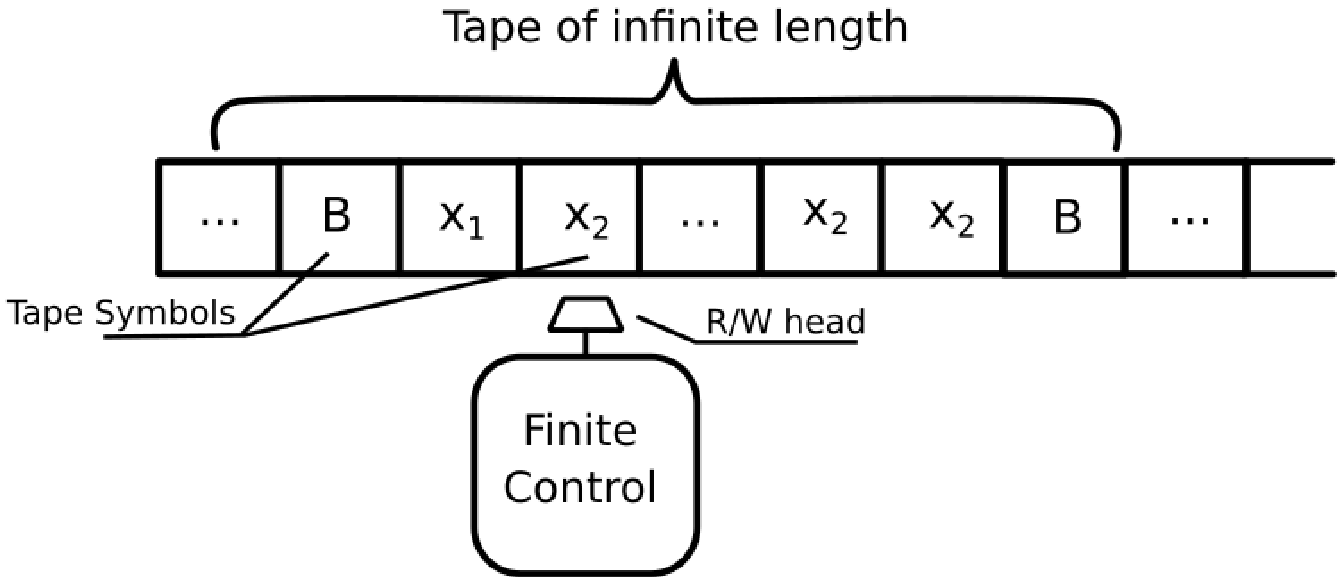
Quantum Algorithms
Lecture 16
Classical computation – more examples

Zhejiang University

Turing machines

TM definition reminder

Turing machine has tape, where input is given and machine can write on it. Machine has finite control that can be described by finite set of states and transition function that describes the behavior of the machine.



Computational complexity

We say that a Turing machine works in time $T(n)$ if it performs at most $T(n)$ steps for any input of size n. Analogously, a Turing machine M works in space $s(n)$ if it visits at most $s(n)$ cells for any computation on inputs of size n.

Time – number of computational steps.

Space – number of cells visited by a TM.

Halting problem

Halting problem is defined for as follows:

$Halting(M, x)$ for TM description M and x (we can consider M as a TM program or as a TM itself.

$Halting(M, x) = 1$ if M stops on input x.

$Halting(M, x) = 0$ if M does not stop on input x.

In a book there is a task with a proof that Halting problem is undecidable. It is possible that M did not stop, but we cannot be sure whether it will stop eventually.

Another idea of a proof

Christopher Strachey outlined a proof by contradiction that the halting problem is not solvable. The proof proceeds as follows: Suppose that there exists a total computable function $halts(f)$ that returns true if the subroutine f halts (when run with no inputs) and returns false otherwise.

Another idea of a proof

Now consider the following subroutine:

```
def g():
    if halts(g):
        loop_forever()
```

halts(g) must either return true or false, because halts was assumed to be total. If halts(g) returns true, then g will call loop_forever and never halt, which is a contradiction. If halts(g) returns false, then g will halt, because it will not call loop_forever; this is also a contradiction. Overall, g does the opposite of what halts says g should do, so halts(g) can not return a truth value that is consistent with whether g halts. Therefore, the initial assumption that halts is a total computable function must be false.

Example problem - One

$One(M,x) = 1$ if M stops on input x and $M(x) = 1$, i.e., gives output 1.

$One(M,x) = 0$ otherwise (either if M does not stop or if stops and gives output 0.

Is this problem decidable – solvable by a TM?

Example problem - One

$One(M, x)$.

If we would be able to solve this problem, then it also would be possible to solve Halting problem.

It is known that Halting problem is not solvable.

We have to conclude, that problem One is also not solvable.

Example problem - One

$One(M, x)$.

We can also consider reductions. If we are able to reduce the given problem to Halting, then the problem is not solvable by a TM.

In our case we can reduce $One(M, x)$ to $Halting(M', x')$.

M' is the same machine M, except that it will always output 1 when it halts. $x' = x$.

As a result, we get an instance of One that solves Halting, i.e., output will be the same.

Example problem – 17 steps

$17\mathrm{steps}(M, x) = 1$ if M stops on input x within 17 computational steps. This means that after no more than 17 computational steps machine M stops.

$17\mathrm{steps}(M, x) = 0$ if M does not stop after 17 computational steps on input x.

Example problem – 17 steps

$17\text{steps}(M, x)$.

This problem is solvable by a TM. The reason is because we can run a TM M for 17 steps and check whether it stopped. If M stopped, then output 1. If M did not stop, output 0.

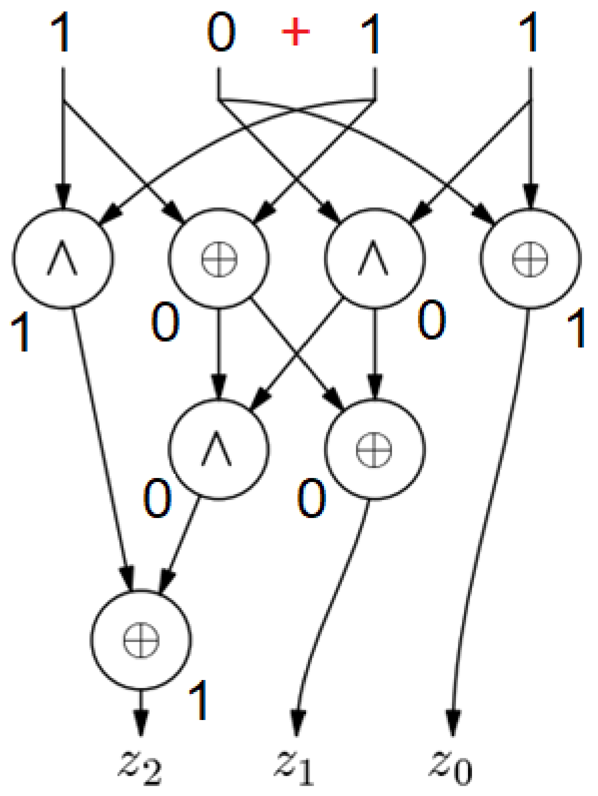
Boolean circuits

Boolean circuit example

$u_1 = 1 \wedge 1 = 1$
$u_2 = 1 \oplus 1 = 0$
$u_3 = 0 \wedge 1 = 0$
$z_0 = 0 \oplus 1 = 1$

$u_4 = 0 \wedge 0 = 0$
$z_1 = 0 \oplus 0 = 0$

$z_2 = 1 \oplus 0 = 1$



Problem 2.1

Construct an algorithm that determines whether a given set of Boolean functions A constitutes a complete basis. (Functions are represented by tables.)

Problem 2.1

Let us find all functions in two variables that can be expressed by formulas in the basis A. We begin with two projections, namely, the functions $p1(x, y) = x$ and $p2(x, y) = y$.

These projections will be allowed in our circuit as from any two variables we can always continue with one of them. Now set F consists of A, p1, p2.

x	y	p1
0	0	0
0	1	0
1	0	1
1	1	1

x	y	p2
0	0	0
0	1	1
1	0	0
1	1	1

Problem 2.1

Then the following procedure is applied to the set F of already constructed functions. We add to the set F all functions of the form

$$f(g_1(x_1, x_2), g_2(x_3, x_4), \ldots, g_k(x_{2k-1}, x_{2k}))$$

, where $x_j \in \{x, y\}, g_j \in F, f \in F$. If the set F increases, we repeat the procedure.

Basically, we extend our set with compositions of functions.

Problem 2.1

Otherwise there are two possibilities: either we have obtained all functions in two variables (then the basis is complete), or not (then the basis is not complete).

Problem 2.1

We estimate the working time of this algorithm. Only 16 Boolean functions in two variables exist; therefore the set F can be enlarged at most 14 times. At each step we must check at most $16^m \cdot |A|$ possibilities, where m is the maximum number of arguments of a basis function.

Problem 2.1

Indeed, for each basis function f each of (at most) m positions can be occupied by any function from F, and $|F| \leq 16$. The length of the input (encoded basis) is at least 2^m (because the table for a function in m Boolean variables has 2^m entries). So, the working time of the algorithm is polynomially bounded in the length of the input.

Problem 2.4

Let C be an $O(logn)$ -depth circuit which computes some function $f\colon B^n \to B^m$. Prove that after eliminating all extra variables and gates in C (those which are not connected to the output), we get a circuit of size $poly(n + m)$.

Problem 2.4

Each output depends on $O(1)$ wires; each of them depends on $O(1)$ other wires, etc. Therefore, the total number of used wires and gates is $O(1)^{depth} = 2^{O(\log(m+n))} = poly(m+n)$.

The class NP: Reducibility and completeness

Nondeterministic Turing machine

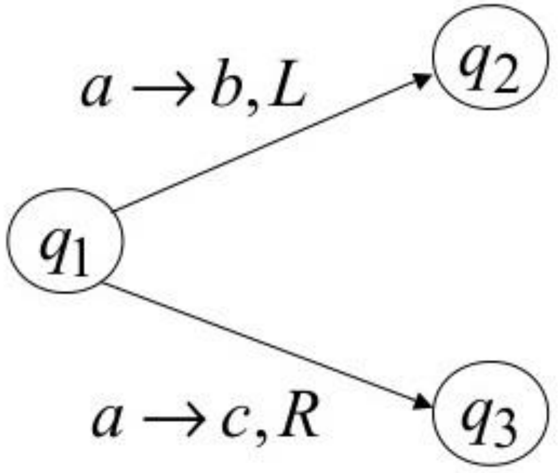
$\delta(q, s_p) = (q', s', \Delta p)$

For each pair (q, s_p) can exist more than one transition, as if machine would be able to do this transitions in parallel.

In this example:

$\delta(q1, a) = (q2, b, \mathrm{L})$

$\delta(q1, a) = (q3, c, \mathrm{R})$



NP - alternative definition

A predicate L belongs to the class NP if it can be represented as

$$L(x) = \exists y((|y| < q(|x|)) \land R(x, y)),$$

where q is a polynomial (with integer coefficients), and R is a predicate of two variables decidable in polynomial time.

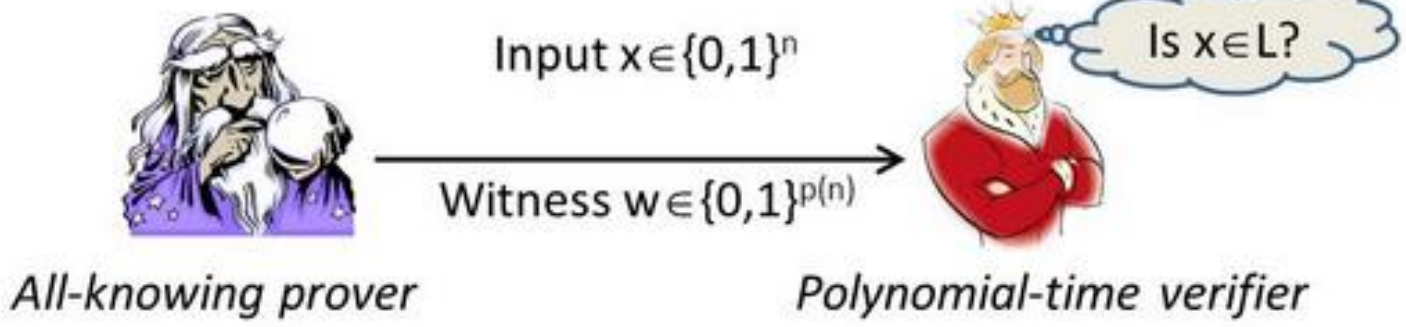
Problems, whose solutions can be checked efficiently, belong to class NP. Given possible solution can be verified in polynomial time.

$R(x, y)$ – here x is given input, and y is provided solution.

Another description of NP

- $L(x) = 1$ => M can convince A that $L(x)$ is true by presenting some proof y such that $R(x, y)$;
- $L(x) = 0$ => whatever M says, A is not convinced: $R(x, y)$ is false for any y.

Moreover, the proof y should have polynomial (in $|x|$) length, otherwise A cannot check $R(x, y)$ in polynomial (in $|x|$) time.



Problem 3.3

Suppose we have an NP-oracle — a magic device that can immediately solve any instance of the SAT problem for us. In other words, for any propositional formula the oracle tells whether it is satisfiable or not. Prove that there is a polynomial-time algorithm that finds a satisfying assignment to a given formula by making a polynomial number of queries to the oracle.

Problem 3.3

Let $F(x1, \dots, xn)$ be a propositional formula. How do we find a satisfying assignment?

First, we ask the oracle whether such an assignment exists.

If the answer is “yes”, we next learn whether there is a satisfying assignment with $x1 = 0$. To this end, we submit the query $F' = F \land \neg x1$.
If the answer is “no” (but F is satisfiable), then there is a satisfying assignment with $x1 = 1$.

Then we try to fix the value of $x2$ and so forth.

We need n many such iterations to find a solution.

Problem 3.8

Prove that the predicate “x is the binary representation of a composite integer” belongs to NP.

Problem 3.8

Merlin tells Arthur the prime factors of x (each factor may repeat several times). Since the multiplication of integers can be performed in polynomial time, Arthur can check whether or not the factorization provided by Merlin is correct.

Problem 3.9

Prove that the predicate “x is the binary representation of a prime integer” belongs to NP.

Problem 3.9

We prove that Primality $\in$ NP by showing how Merlin constructs a polynomial size “certificate” of primality that Arthur can verify in polynomial time.

Problem 3.9

Let $p > 2$ be a prime number. It is enough to convince Arthur that (Z/pZ)$*$ is a cyclic group of order $p - 1$. Merlin can show a generator g of this group, and Arthur can verify whether $g^{p-1} \equiv 1\,(mod\,p)$ (this requires $O(\log p)$ multiplications; see Section 4.2.2).

Theorem A.10. If p is a prime number, then (Z/pZ)$*$ is a cyclic group of order $p - 1$.

Problem 3.9

This is still not enough since the order of g may be a nontrivial divisor of $p-1$. If for some reason Arthur knows the factorization of $p-1$ (which includes prime factors q_1, q_2, ...), he can check whether $g^{(p-1)/q_j} \equiv 1\,(mod\,p)$. But factoring is a difficult problem, so Arthur cannot compute the numbers q_j himself.

Problem 3.9

However, Merlin may communicate the factorization to Arthur. The only problem is that Merlin has to convince Arthur that the factors are indeed prime. Therefore Merlin should recursively provide certificates of primality for all factors.

Problem 3.9

Thus the complete certificate of primality is a tree. Each node of this tree is meant to certify that some number q is prime (the root corresponding to $q = p$). All leaves are labeled by $q = 2$. Each of the remaining nodes is labeled by a prime number $q > 2$ and also carries a generator h of the group (Z/qZ)$*$; the children of this node are labeled by prime factors of $q - 1$.

Problem 3.9

Let us estimate the total size of the certificate. The tree we have just described has at most $n = \lceil \log_2 p \rceil$ leaves (since the product of all factors of $q - 1$ is less than q). The total number of nodes is at most twice the number of leaves (this is true for any tree). Each node carries a pair of n-bit numbers (q and h). Therefore the total number of bits in the certificate is $O(n^2)$.

Problem 3.9

Now we estimate the certificate verification complexity. For each of $O(n)$ nodes Arthur checks whether $h^{p-1} \equiv 1 \ (mod\ q)$. This requires $O(logq) = O(n)$ multiplications, which is done by a circuit of size $O(n^3)$. Similar checks are performed for each parent-child pair, but the number of such pairs is also $O(n)$. Therefore the certificate check is done by a circuit of size $O(n^4)$, which can be constructed and simulated on a TM in time $poly(n)$.

Probabilistic algorithms and the class BPP

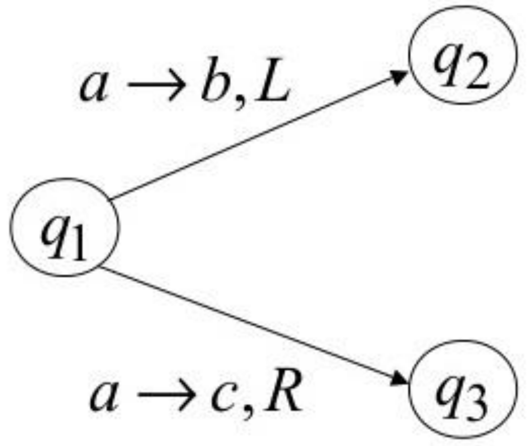
PTM

A probabilistic Turing machine (PTM) is somewhat similar to a nondeterministic one; the difference is that choice is produced by coin tossing, not by guessing.

For example:

$\delta(q1, a) = (q2, b, \mathrm{L})$ with probability 1/2

$\delta(q1, a) = (q3, c, \mathrm{R})$ with probability 1/2



BPP

Let ε be a constant such that $0 < \varepsilon < 1/2$. A predicate L belongs to the class BPP if there exist a PTM M and a polynomial $p(n)$ such that the machine M running on input string x always terminates after at most $p(|x|)$ steps, and

- $L(x) = 1 \Rightarrow M$ gives the answer “yes” with probability $\geq 1 - \varepsilon$;
- $L(x) = 0 \Rightarrow M$ gives the answer “yes” with probability $\leq \varepsilon$.

BPP - equivalent definition

A predicate L belongs to BPP if there exist a polynomial p and a predicate R, decidable in polynomial time, such that

- $L(x) = 1 \Rightarrow$ the fraction of strings r of length $p(|x|)$ satisfying $R(x, r)$ is greater than $1 - \varepsilon$;
- $L(x) = 0 \Rightarrow$ the fraction of strings r of length $p(|x|)$ satisfying $R(x, r)$ is less than ε.

Here strings r denote strings of random bits – e.g., probabilistic choices of PTM. This reminds us proof/solution like in NP definition.

Probabilistic algorithm example

The approximate counting algorithm allows the counting of a large number of events using a small amount of memory. The algorithm is considered one of the precursors of streaming algorithms, and the more general problem of determining the frequency moments of a data stream has been central to the field.

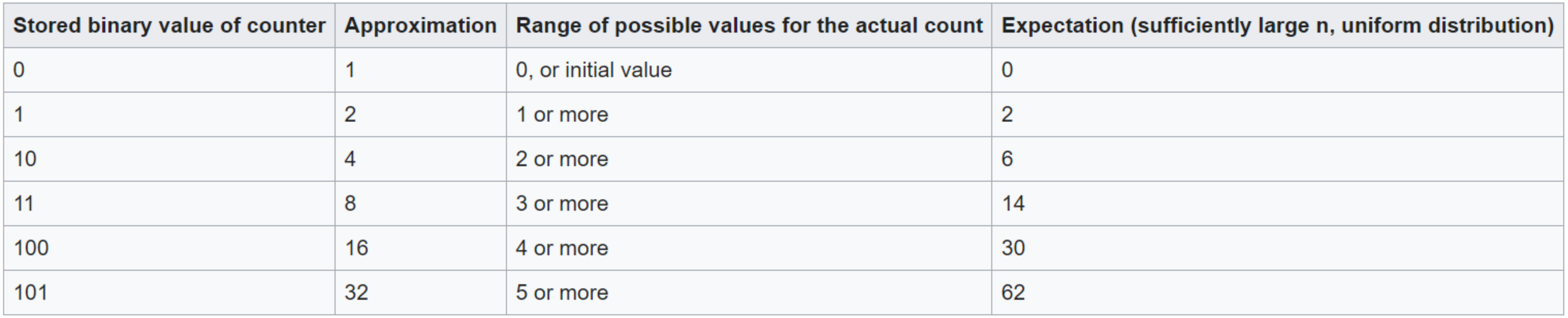
Approximate counting

Using Morris' algorithm, the counter represents an "order of magnitude estimate" of the actual count. The approximation is mathematically unbiased.

To increment the counter, a pseudo-random event is used, such that the incrementing is a probabilistic event. To save space, only the exponent is kept. For example, in base 2, the counter can estimate the count to be 1, 2, 4, 8, 16, 32, and all of the powers of two. The memory requirement is simply to hold the exponent.

Approximate counting

As an example, to increment from 4 to 8, a pseudo-random number would be generated such that a probability of .25 generates a positive change in the counter. Otherwise, the counter remains at 4.



Stored binary value of counter	Approximation	Range of possible values for the actual count	Expectation (sufficiently large n, uniform distribution)
0	1	0, or initial value	0
1	2	1 or more	2
10	4	2 or more	6
11	8	3 or more	14
100	16	4 or more	30
101	32	5 or more	62

Approximate counting

If the counter holds the value of 101, which equates to an exponent of 5 (the decimal equivalent of 101), then the estimated count is 2^5, or 32. There is a fairly low probability that the actual count of increment events was 5:

$$1 * \frac{1}{2} * \frac{1}{4} * \frac{1}{8} * \frac{1}{16} = \frac{1}{1024}$$

The actual count of increment events is likely to be "around 32", but it could be arbitrarily high (with decreasing probabilities for actual counts above 39).

Approximate counting

When incrementing the counter, "flip a coin" the number of times of the counter's current value. If it comes up "Heads" each time, then increment the counter. Otherwise, do not increment it.

Example:

Step	Coin toss	Counter value
start		0
1	Tails	0
2	Heads	1
3	Heads, Tails	1
4	Tails, Heads	1
5	Heads, Heads	2
6	T,H,H	2

Approximate counting

In our previous example we finished after 6 steps. The value of the counter is 2. Then we conclude that counter stored magnitude 2, which means number $2^2 = 4$. Expected value was 6.

This algorithm gives estimate of counted value, so it is approximation. So real value can differ much from counter value.

Advantage – it uses exponentially less space. To count up to number n, binary number stored uses $O(loglogn)$ memory.

Approximate counting

The algorithm is useful in examining large data streams for patterns. This is particularly useful in applications of data compression, sight and sound recognition, and other artificial intelligence applications.

The hierarchy of complexity classes

Two-player game

Consider a game with two players called White (W) and Black (B). A string x is shown to both players. After that, players alternately choose binary strings: W starts with some string $w1$, B replies with $b1$, then W says $w2$, etc. Each string has length polynomial in $|x|$. Each player is allowed to see the strings already chosen by his opponent.

Description of the game

There is a predicate $W(x, w1, b1, w2, b2, ...)$ that is true when W is the winner, and we assume that this predicate belongs to P. If this predicate is false, B is the winner (there are no ties). This predicate (together with polynomial bounds for the length of strings and the number of steps) determines the game.

The game is completed after some prescribed number of steps.

The referee, who knows x and all the strings and who acts according to a polynomial-time algorithm, declares the winner.

Defining classes

Since this game is finite and has no ties, for each x either B or W has a winning strategy.

Therefore, any game determines two complementary sets,

$Lw = \{x\text{: } W\ has\ a\ winning\ strategy\}$,

$Lb = \{x\text{: } B\ has\ a\ winning\ strategy\}$.

Many complexity classes can be defined as classes formed by the sets Lw (or Lb) for some classes of games.

PSPACE – alternative definition

L ∈ PSPACE if and only if there exists a polynomial game such that
$L = \{x\colon W\ has\ a\ winning\ strategy\ for\ input\ x\}$.

By a polynomial game we mean a game where the number of moves is bounded by a polynomial (in the length of the input), players' moves are strings of polynomial length, and the referee's algorithm runs in polynomial time.

Problem 5.1

Prove that any predicate $L(x)$ that is recognized by a nondeterministic machine in space $s = poly(|x|)$ belongs to PSPACE. (A predicate L is recognized by an NTM M in space $s(|x|)$ if for any $x \in L$ there exists a computational path of M that gives the answer “yes” using at most $s(|x|)$ cells and, for each $x \notin L$, no computational path of M ends with “yes”.)

Problem 5.1

If there is an accepting (i.e., ending with “yes”) computational path, then there is such a path of length $\exp(O(s))$. Indeed, we may think of machine configurations as points, and possible transitions as edges. Thus an NTM with space s is described by a directed graph with $N = \exp(O(s))$ vertices, and then an accepting path is just a path between two given vertices. If there is such a path, we can eliminate loops from it and get a path of length $\leq N$.

Problem 5.1

Therefore, the proof for the following statement is still valid for nondeterministic Turing machines:
L ∈ PSPACE if and only if there exists a polynomial game such that
$L = \{x\text{: } W\ has\ a\ winning\ strategy\ for\ input\ x\}$.

Thus we reduce the NTM to a polynomial game; then we simulate this game on a deterministic machine.

Thank you for your attention!