CS 517
Homework 5

Ameyassh Nagarajan

May 2024

1 Problem 1

Proof

To show that $\mathrm{NP}^{\mathrm{NP}\cap\mathrm{coNP}} = \mathrm{NP}$, we need to prove two inclusions: $\mathrm{NP}^{\mathrm{NP}\cap\mathrm{coNP}} \subseteq \mathrm{NP}$ and $\mathrm{NP} \subseteq \mathrm{NP}^{\mathrm{NP}\cap\mathrm{coNP}}$.

Proof

To show that $\mathrm{NP}^{\mathrm{NP}\cap\mathrm{coNP}} = \mathrm{NP}$, we need to prove two inclusions: $\mathrm{NP}^{\mathrm{NP}\cap\mathrm{coNP}} \subseteq \mathrm{NP}$ and $\mathrm{NP} \subseteq \mathrm{NP}^{\mathrm{NP}\cap\mathrm{coNP}}$.

Proof

$\mathbf{NP}^{\mathbf{NP}\cap\mathbf{coNP}} \subseteq \mathbf{NP}$ $(\rightarrow)$

We need to show that any language that can be decided by a nondeterministic polynomial-time Turing machine with an oracle for $\mathrm{NP} \cap \mathrm{coNP}$ can also be decided by a nondeterministic polynomial-time Turing machine without such an oracle.

Suppose $L \in \mathrm{NP}^{\mathrm{NP}\cap\mathrm{coNP}}$. By definition, there exists a nondeterministic polynomial-time Turing machine M that decides L with access to an oracle for $\mathrm{NP} \cap \mathrm{coNP}$.

Since $\mathrm{NP} \cap \mathrm{coNP} \subseteq \mathrm{NP}$, any oracle query that M makes to $\mathrm{NP} \cap \mathrm{coNP}$ can be simulated by an NP oracle. Therefore, M can be simulated by a nondeterministic polynomial-time Turing machine M' with access to an NP oracle.

We know that $\mathrm{NP}^{\mathrm{NP}} = \mathrm{NP}$. Thus, a nondeterministic polynomial-time Turing machine with access to an NP oracle is still within the complexity class NP. Hence, $\mathrm{NP}^{\mathrm{NP}\cap\mathrm{coNP}} \subseteq \mathrm{NP}$.

For each oracle call y, the NP machine can simulate this call by making its own nondeterministic guesses. The nondeterministic machine can guess a certificate for y and verify it in polynomial time using its own verification process.

The overall process remains polynomial because each step (including the simulated oracle calls) is polynomial in time and the number of oracle calls is polynomially bounded. This means that the overall running time of the NP machine with an NP oracle remains polynomial.

Since the NP machine can simulate the NP oracle using its own capabilities, any problem solvable by $\mathrm{NP}^{\mathrm{NP}}$ can also be solved by an NP machine without the oracle. Therefore, $\mathrm{NP}^{\mathrm{NP}}$ does not have more computational power than NP alone.

We have shown that a nondeterministic polynomial-time Turing machine with access to an NP oracle can be simulated by a standard nondeterministic polynomial-time Turing machine. Hence, $\mathrm{NP}^{\mathrm{NP}} = \mathrm{NP}$.
Therefore, $\mathrm{NP}^{\mathrm{NP} \cap \mathrm{coNP}} \subseteq \mathrm{NP}$.

$\mathbf{NP} \subseteq \mathbf{NP}^{\mathbf{NP} \cap \mathbf{coNP}}$ $(\leftarrow)$

We need to show that any language in NP can also be decided by a nondeterministic polynomial-time Turing machine with access to an oracle for $\mathrm{NP} \cap \mathrm{coNP}$.

Suppose $L \in \mathrm{NP}$. By definition, there exists a nondeterministic polynomial-time Turing machine M that decides L. We need to show that L can also be decided by a nondeterministic polynomial-time Turing machine N with access to an oracle for $\mathrm{NP} \cap \mathrm{coNP}$.

Any language in NP can be decided by a nondeterministic polynomial-time Turing machine with access to an oracle for $\mathrm{NP} \cap \mathrm{coNP}$ because it does not need to use the oracle. Therefore, N can simulate M without using its oracle. Hence, $\mathrm{NP} \subseteq \mathrm{NP}^{\mathrm{NP} \cap \mathrm{coNP}}$.

Conclusion

Since we have shown both $\mathrm{NP}^{\mathrm{NP} \cap \mathrm{coNP}} \subseteq \mathrm{NP}$ and $\mathrm{NP} \subseteq \mathrm{NP}^{\mathrm{NP} \cap \mathrm{coNP}}$, it follows that

$$\mathrm{NP}^{\mathrm{NP} \cap \mathrm{coNP}} = \mathrm{NP}.$$

2 Problem 2

Let M be an oracle Turing machine that is shrinking, meaning M queries its oracle only on strings strictly shorter than its input. Suppose a language A can be written as

$$A = \{x \mid M^A(x) = 1\},$$

where M is a polynomial-time shrinking oracle Turing machine. We aim to prove that such a language A is in PSPACE.

Proof

To prove that $A \in$ PSPACE, we need to show that there exists a polynomial-space algorithm that decides membership in A.

The shrinking property of M ensures that on input x, any oracle query made by M is to a string y such that $|y| < |x|$. This recursive querying structure inherently limits the depth of queries.

Simulation in Polynomial Space

We will simulate M using polynomial space without directly using the oracle for A. Since M is a polynomial-time Turing machine, it uses at most $p(|x|)$ time steps for some polynomial p.

Recursive Simulation

For input x, we simulate M step-by-step. Whenever M makes an oracle query to a string y (with $|y| < |x|$), we pause the current simulation and recursively simulate M on y.

Stack Management

We use a stack to keep track of the simulation context at each level of recursion. Each stack frame stores the state of the Turing machine at the point of making an oracle query, which includes:

- The tape content up to that point.
- The head position.
- The current state of the finite control.
- The position in the input string x.

The amount of information stored in each stack frame is $O(p(|x|))$.

Depth of Recursion

The depth of recursion is bounded by the length of the input string $|x|$ because each oracle query is to a strictly shorter string. The maximum depth of the recursion is therefore $|x|$.

Overall Space Usage

At each level of recursion, we use $O(p(|x|))$ space. The total space used by the stack is $O(|x| \times p(|x|))$, which is polynomial in $|x|$.

Polynomial-Space Algorithm

We can construct a polynomial-space algorithm that simulates M on input x as follows:

1. **Base Case**: If x is sufficiently short (e.g., length 0), directly simulate M without any oracle queries.
2. **Recursive Case**: For input x, simulate M and handle each oracle query y by recursively simulating M on y.
3. Use a stack to keep track of the recursive calls, ensuring that at each level, the space used is polynomial in the input size.

The recursion stack depth is at most $|x|$, and each level uses polynomial space.

Conclusion

By constructing a polynomial-space algorithm that simulates the shrinking oracle Turing machine M, we have shown that the language A is in PSPACE. Thus, if A can be decided by a polynomial-time shrinking oracle Turing machine, then A is in PSPACE.

3 Problem 3

State Representation

A state is defined by:

- The position of the player (x, y).
- The positions of all k movable blocks.

Each state can be represented using $O((n \times m) + k \log(n \times m))$ bits, which is polynomial in n and m.

State Space Calculation

- The player can be in any of the $n \times m$ cells.
- Each of the k movable blocks can be in any of the $n \times m$ cells, with no two blocks occupying the same cell.

The total number of states S is:

$$S = (n \times m) \times \binom{n \times m}{k}$$

where

$$\binom{n \times m}{k} = \frac{(n \times m)!}{k! \times (n \times m - k)!}$$

Why the algorithm uses PSPACE

While the total number of states is factorial, we can solve the problem using polynomial space by simulating state transitions without storing all states simultaneously.

State Representation in Polynomial Space

Each state requires $p((n \times m)$ bits, which is polynomial.
A nondeterministic algorithm can guess the next move (player movement or block push) at each step, using polynomial space to keep track of the current state and sequence of moves. This approach simulates exploring the state space without explicitly storing the set of visited states.
Since we have shown that **NPSPACE** = **PSPACE**. We can conclude that by using a nondeterministic approach to explore the state space and using the fact that **NPSPACE** = **PSPACE**, we can show that the problem is solvable using polynomial space. Thus, the problem of determining if the player can reach the target cell is in **PSPACE**, as it can be solved using a polynomial amount of space relative to the size of the input.

Depth-First Search (DFS) Algorithm

- Start from the initial state.
- For the current state, generate all possible next states by making legal moves.
- Recursively explore each of the next states.
- If a state leads to the target, the problem is solved.
- If no more moves are possible, backtrack to the previous state and try the next possible move.

Handling Exponentially Many States

If there are exponentially many states after the current state, DFS remains feasible in terms of space complexity but faces challenges in terms of time complexity.

Space Complexity

- **State Representation**: Each state is represented by the configuration of the board, which requires $O(n \times m)$ space.
- **Maximum Depth**: The maximum depth of the search tree corresponds to the maximum number of moves needed to reach the target, which is $O(n \times m)$ in the worst case.
- **Space Used**: The space required for the stack to store the current path is $O(d \times s)$, where d is the depth of the tree and s is the space required to store a state. This results in $O((n \times m)^2)$ space, which is polynomial in the size of the board.

Conclusion

Therefore, we have proved that the given problem is in **PSPACE**