Problem 16

Víctor Alcázar
Kosmas Palios
Albert Ribes

March 6, 2017

Generalized board games

Consider a board game such as Go or Chess, where the state space of possible positions on an $n \times n$ board is exponentially large and there is no guarantee that the game only lasts for a polynomial number of moves. Assume for the moment that there is no restriction on visiting the same position twice.

Exercise 16i

We have to prove that board games like chess or go in $n \times n$ boards are in EXP.

The solution

We only have to give an exponential algorithm that solves such a game.

Main idea

We construct a tree T of possible states of the game ,produced from the initial state by moving pieces around. A state is combination of a configuration of pieces on the board and a variable telling whose turn it is to play. For every configuration of pieces, there are (at most) two states/nodes in the tree. One for the case it is white's turn to play, and one for the case that it is black's turn. This tree can be used to discover if a certain player has a winning strategy or not.

Details

In this tree every node does not have a constant number of children. Every possible state can have polynomially many neighbouring states. We can define $max\_number\_of\_neighbours = p(n)$ to be the maximum number of neighbours a state can have. The exact value depends on the game rules.

Let us consider that in any game there are c kinds of pieces. For example, in chess we have $c = 6$ (king,queen,rook,bishop,knight,pawn). That means that in every square of the board can have none or one of 2*c pieces (every piece can be either black or white). Therefore, the number of possible configurations on a $n \times n$ board is $(2c+1)^{n^2}$.

From the above observation, we conclude that our directed tree has $2(2c+1)^{n^2}$ nodes. Also, there are back edges, although these will not trouble us a lot, as shown later. While creating this tree, we find some states in which either black or white wins. These are the leaves of the tree.

Tree construction

The creation of this tree takes exponential time. Why? Let us describe the constuction process with in detail.

Each state is in the form $s_i =< board_i, white\_plays_i >$. The first variable gives us the board configuration and the second is a boolean that is true if it is white's turn to play and false otherwise.

We begin with an initial state s_0, given as input to the problem. Then, we start exploring in a BFS manner. Firstly, we create the $p(n)$ states that can be generated by this initial state, where $p(n)$ is a polynomial of size n. We add the newly created states to a queue, and then we repeat the process for each one of the states in the queue, but without creating duplicate states. In this manner, we shall create no more than $2(2c+1)^{n^2}$ nodes, and since for each node we try p(n) moves, the creation of tree is complete after $O((2c+1)^{n^2}p(n))$ steps.

Tree traversal

Now we have this tree. What do we do with it? Consider how the condition "Player I has a winning strategy" is check-able in this tree.

The condition "Player white has a winning strategy from the initial state <b,true>" is equivalent to "$\exists$ a first move for white s.t. $\forall$ second moves of black, $\exists$ a third move for white s.t. $\forall$ fourth moves of black $\ldots \exists$ a k-th move for black that wins the game. "

This can be verified in our tree, given exponential time. We will describe how in the algorithm given below. Note that the back-edges do not actually affect the solution, as they lead us to previous nodes.

Add algorithm here...

This concludes the proof.

Exercise 16b

Given that the game ends after p(n) moves, we must show that the above problem is in PSPACE.

The solution

This solution is based on a crucial observation.

In the previous problem, we did not really have to create the whole graph and then start traversing it. We could have avoided using so much memory, as we can build the tree as we go, and only keep the current branch we are traversing.

How much memory is used in this case? Roughly, it would be

$$memory\_consumption = O(longest\_branch\_length * max\_number\_of\_neighbours)$$

$$= O(longest\_branch\_length * p(n))$$

To prove this, we will describe the improved version of the algorithm.

Space efficient algorithm

Add algorithm here...

In conclusion, in the special case of having polynomially limited duration q(n) of the game, we have polynomial space complexity.