Problem 16

Víctor Alcázar Kosmas Palios Albert Ribes

March 16, 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. We call the nodes of the first (second) type white (black) nodes. 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 at most $2(2c+1)^{n^2}$ nodes. Also, there are various edges, that take us back to nodes of higher levels, 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.

For convience we will colour the nodes according to whose turn it is: black nodes for Black's turn to play, and white nodes for White's turn to play. Note that the nodes where White has won are black nodes without out-edges.

Tree traversal

Now we have this graph. 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 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.

In our algorithm, we define the $win()$ function. This takes as input a state (remember that a state is a board configuration and a boolean telling us whose turn it is) and returns true if starting from this state, player WHITE has a possibility of winning regardless of the moves of player black. It can take as input either a state in which white plays, or a state in which black plays.

Depending on the value of $white\_plays$ our function's behaviour changes:

- if state.white_plays is true
  - has White lost? Then the function return false.
  - Otherwise, the function checks if *one* of the child-states of this state (in all of which it is black's turn to play) can lead to white's victory, regardless of the black's moves. Equivalently, if $\exists child\_state : win(child\_state) = true$.
- if state.white_plays is false:
  - Has Black lost? Then the function returns true.

- Otherwise the function checks if *all* of the child-states of this state (in all of which it is white's turn to play) can lead to white's victory. Equivalently, if $\forall child\_state : win(child\_state) = true$

It is pretty straightforward to prove that if the win() function returns true, that means that there is a winning strategy for White.

Now, for the algorithm:

1. Beginning with the graph described above, we check if there are any black nodes with out-degree zero. We know the value of win() for these nodes, as the are positions of White's victory. We label them true.

2. Then using acquired information, we check the nodes that are neighbours of the newly labeled nodes. All of them are white nodes, because only white moves can end the game. So, we label all of them as winning, as they all have at least one out-edge that goes to an endgame.

3. Now the nodes labeled "winning" are more, can we determine if one of their neighbours is a winning node? This time, our neighbours are all black, so the condition is that *all* of their out-edges point to a winning node. Suppose there is one that fulfils this condition. We label it winning, and we continue, keeping the rest of the neighbours in a list of "yet unspecified neighbours".

4. We continue like this. With each addition to the "winning" set, we check the neighbours of the newly labeled nodes, as well as the list of "yet unspecified neighbours".

5. We stop either when we reach to a point we cannot designate any more nodes as winning, or when we have labeled the initial node as winning.

6. In the first case, we can safely say that White has no winning strategy.

7. In the second case we can safely say that she has a winning strategy.

Why are the last two prepositions true? There second one is immediately derived from the definition of the win() property.

If the algorithm stops before reaching initial state there is no winning strategy

Consider the point where the algorithm stops. Then, we can be sure that that the set of already labeled nodes do not have in-edges from white unlabeled nodes, because otherwise they would be eligible for labelling by our algorithm. Then, we can say that two cases exist:

1. Neither black nodes exist that have connection with our true-subgraph: In this case, we can deduce that the graph has no directed path from the unlabeled nodes subgraph to the labeled ones. That means that there are no sequence of moves that can lead White to victory whatsoever (remember, all the winning nodes are in the labeled territory).

2. There exist black nodes that have out-edges to labeled nodes, but every one of them has an out-edge pointing towards another unlabeled node. In this case we realize that every sequence of moves leading to White's victory must pass by on of these black nodes. But every time the game is in one of these nodes, Black can send it to a node that does neighbour with the nodes in the labeled "territory", thus creating a loop, possibly infinite. This way, Black can stall the game forever, making White's victory impossible.

In both cases, there doesn't exist a fail-proof winning strategy for White.

Exercise 16b

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

The solution

- We will visit the possible moves graph using DFS algorithm. Note that this graph is not the same an the one above, because it includes states which are equal in different memory space. That way we handle the loops. As it is DFS, we don't have to keep the whole graph in memory, only a branch, which they assure is only polynomially large.
- We do an exhaustive search using only polynomial space.