Backward algorithm and abstract graphs in zero-sum games

by

Zangir Iklassov

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Arts

in

Economics

at

NAZARBAYEV UNIVERSITY

SCHOOL OF HUMANITIES AND SOCIAL SCIENCE

2020

1 Abstract

With the beginning of the computer age, solving many problems in game theory has become easier. However, there is a whole class of well-known problems such as chess, checkers, go and so on, the methods of solving which use brute force technique to find solutions of the game. This technique requires analysis of all states of the game and it has the exponential complexity of running the algorithm due to which many games cannot be solved on modern computers. Considered class of games include zero-sum games with perfect information described in discrete space. For such problems, there is no smooth solution that would allow solving problems without going through all the states of the game. This work proposes a new algorithm for finding solutions to such problems. The algorithm uses a new data structure, called abstract graphs and backwards analysis to find solutions. The algorithm still has the exponential complexity of the analysis, however, finding a solution does not require going through all the possible states of the game, which reduces the complexity of analysis. For a clear example, the algorithm was used on a tic-tac-toe game, for which brute force technique requires analysis of around 15k states, while the Backwards algorithm analyzed just 5 states to find all existing solutions. In the future, this study can be continued for a deeper study of the mathematical properties of the algorithm and to use it on games such as chess and go.

2 Introduction

Game Theory concepts and algorithms can be useful in solving real world strategic situations. They are used in policy estimation, utility maximization, auctions, advertising campaign, etc. However, for certain types of the games, modern algorithms use brute force

technique, which is not optimal, while the set of actions and states cannot fit computer memory.

New concepts are proposed: Abstract Graphs and Backward Algorithm. The first one is a data structure that helps to describe game in compact format. This format stores any set of states and actions using small amount of memory. The second one is algorithm that helps to work with this data structure and find solution. The difference from standard Backward Induction used in Game Theory is that Backward Algorithm does not consider each concrete state of the game in the tree, but analyzes abstract states which will make algorithm less complex. The standard tree used in Backward Induction consists of leaves that denote all states of the game, while Backward algorithm uses tree with leaves that denote abstract states. The result of algorithm would be the set of rules of how to play the game that allow to never lose the game regardless of opponents' the actions.

The algorithm can potentially be useful for many zero-sum games. An additional requirement for the usefulness of this algorithm will be that the game must be in discrete space, like tic tac toe, chess and go games. Their actions and payoffs are described in discrete space and there is no smooth function of the dependence of their payoffs on the actions of the player; otherwise, these games could be solved using simple function optimization like Stackelberg competition.

This work introduces concepts of Abstract Graph and Backward Induction. These concepts used to solve tic-tac-toe game, since it has full set of solutions and it is easy to compare results of algorithm with existing solutions.

3 Literature review

According to Shannon (1950) there are 10^{40} positions that are possible in chess game. Number of possible games exceeds 10^{120}. These are extremely huge numbers to solve on the modern computers using simple brute force method, which checks every possible position and all its continuations in the game. The modern chess programs use brute force limiting it to certain depth. They also have preloaded game debuts that are well developed through chess history; and they estimate each position based on some human created utility function. This combination of characteristics allows us to create program that can win any player in the world. However, such programs do not really solve the game. They are restricted by their depth of analysis. They cannot analyze game from the beginning till the end.

According to Bratko (2018) such a situation had interesting change with creation of so-called AlphaZero algorithm. It is an artificial neural network that behaves more like a human. It does not estimate each possible future position but given state of the game it discards wittingly 'bad' moves and analyzes only moves that are potentially 'good'. This allows program to analyze to a much greater depth. However; at the same time, it has certain drawbacks. Firstly, as any neural network algorithm this is the black box program, meaning we can give it input and get certain output but we are unable to understand why it took certain decision, why one move is better than the other. Secondly, it is not appropriate for our case, since it does not solve the game. This algorithm still incapable to analyze such a great number of positions in chess.

This work will introduce the algorithm that is both interpretable and solves the game, meaning it can find complete set of rules to never lose the game. The algorithm is useful for zero-sum games described in discrete space, such as chess, go and tic-tac-toe. Payoffs

of these games are not described using smooth functions in continuous spaces, therefore all algorithms that aim to find a solution for the these games use brute force technique of analyzing the game. It requires exhaustive search of all game states to find optimal strategy. Thus, the effectiveness of any algorithm will directly depend on the number of possible states in the considered game. Therefore, the fewer states an algorithm will have to consider, the more efficient it will be. This is the purpose of the study to take a different look at set of states of zero-sum games, making it more compact and thereby less difficult to analyze.

4 Methodology

4.1 Game

This study will only consider zero-sum games, one type of non-cooperative games, where the final payoffs of the players are opposite to each other, that is, the sum of all final payoffs is equal to zero. For zero-sum games there does not exist a strategy that is profitable for all players. Hereinafter, simple tic-tac-toe game will be considered. In tic-tac-toe there is 3x3 board with nine empty cells at the beginning of the game. Each player chooses a sign 'x' or 'o'. Players take turns placing their signs in the empty cells. The player who first puts his three signs in one line, horizontal, vertical or diagonal, will win the game. If no player has reached this goal, but the board is no longer empty, the game ends in a draw. Hereinafter the term line will refer to a straight line on the board of tic-tac-toe, either some abstract line on the board or some specific, term sign will refer to 'x', 'o' or both of them. For use of Backward Induction algorithm this game can be described as follows,

G=(P;A;S), where

$$P = \{p_1, p_2\} \ (1)$$

is a set of two players, p_1 plays 'x', p_2 plays 'o'.

$$A = \{a_1, a_2, a_3, a_4, a_5, a_6, a_7, a_8, a_9, \} \ (2)$$

is a set of actions. In tic-tac-toe there are nine cells, and each ith action means putting sign in i^{th} cell.

s_0 is an initial state of the game, where the board is empty and no sign was put on the board yet.

$$s = \{s_0, s_1, s_2, ...s_{19682}]\} \ (3)$$

is a set of all states of the game, including initial state. There are nine cells, each one can have 'x', 'o' or be empty. Therefore total length of the set of states will be $3^9 = 19683$.

This is the usual way to describe a set of actions and states in a game. However, in section 3.4 the concept of an abstract graph will be explained. This data structure will allow to change the action and state sets to make them smaller, which will make their analysis less complex (See section 3.8).

4.2 Rule

Rule is a data structure that describes a specific move at a certain position. The rule is described using three structures: input state, action and output state. The input state is the set of states of the game, where the rule is used. Action is the object from the set of actions that must be used according to rule, when the player is in the input state. The

output state is the result of an action made in the input state. If a player plays according to the rule, then, when he finds himself in the input state during the game, he takes corresponding action and goes into the output state. In this study, finding the solution to the game is to find such set of rules using which the players will never lose the game.

The purpose of the Backward Algorithm, described in this work is to create set of Rules

$$R = \{r_1, r_2, ..., r_i\} \ (4)$$

using which any player can play the game without losses.

When playing a game, the player finds himself in a certain state. He has a set of rules, where each rule's input is some abstract state. Player compares his current state in the game with each rule's input starting from the first rule. If player finds that input is equal to his current state, then he makes action of the corresponding rule and goes to corresponding output. If player does not find same state in the rule set, then he makes any random action. If player finds same input in the rule set, but he does not make corresponding action, then opponent will have chance to use previous rule, and attack with sure win.

4.3 Extensive form game

In this section standard method of the game analysis such as extensive form game is described. The extensive form of the game is a representation of a certain game in the form of a tree where collected all the players, all possible actions and all positions available in the game. This is the standard tree used in Game Tree analysis. A tree is a collection of states and edges that is directed from top to bottom, where the upper state is the starting position of the game and all the bottom states are the final positions of the game.

The state of the game in the tree is displayed using the so-called leaf. The leaves are connected to each other using the edges, that is, each edge displays certain action and connects two different states of the game. The tree begins to be built from the initial position of the game, then its edges are represented by all possible actions available from this initial position; the leaves of the next level of the tree are represented by all positions available to be reached in one action from the initial position. Thus, we get the next level of leaves in our tree. From each leaf of a new level of the tree we build new edges that are represented by all possible actions available from this leaf. From here we get new leaves of the tree for the next level. We build a tree until all the final leaves end in the states that are final for this game. That is, all the last leaves of the tree should end with draw or the victory of one of the players in considered zero-sum games. If there will be a subtree from current position of the game, where no matter what our opponent does there is a strategy by which we win for sure for the remaining part of the game, we will find such subtree using the method of analyzing the tree of all the states of a particular game. Similarly, we can avoid such sure win from the opponent, by finding corresponding subtree. This will allow us to get necessary set of strategies for solving the game from any position of the game. However, in this algorithm it is necessary to consider all the leaves of the current subtree. Such an analysis requires exponential time for running, where this time is proportional to the average number of leaves at one level of the tree to the power of depth (number of levels) of the tree. Such a solution is not optimal and efficient, since it is impossible to solve a plenty of games with a large number of possible positions. For example, if we consider the game of tic-tac-toe, the starting position of the game, that is, the first leaf of the tree is an empty board, then the first player has 9 options of possible action. Thus, the second level of tree has 9 possible positions (leaves). Then, the second

player makes her move, he has 8 options for possible action. Thus, at the second level, the tree has 72 possible positions. At each subsequent level, the number of possible positions is multiplied by the number reduced by one. Thus, by the end we have 9 factorial of possible positions. However, some of the leaves do not continue until the last 9th move. The game ends with the victory of one of the players earlier. In total, it is estimated that the number of possible leaves in the tree is equal to 15,000. Aycock (2002).

Using modern computational resources, it impossible to draw such a tree for a large number of games. It would be more effective to consider the tree, where the certain node includes several game positions simultaneously, since at the moment only specific type of move is important. There arises a question: if we estimate actions and states combining them into certain abstract node, is it still possible to find the perfect set of rules? That is can we find all rules necessary to be an ideal player by analyzing several actions and states simultaneously during one iteration of the algorithm, not analyzing each action and state separately; to answer this question, new data structure to combine several states and actions together is proposed.

4.4 Abstract Graph

In computer science and graph theory there is a concept of graph, which is the set of objects (nodes) and connections (edges) between those objects, where the connections represent some relation between the objects. Abstract graph uses abstract nodes rather than some concrete objects. For example, in tic-tac-toe abstract graph can be represented using the following nodes: an abstract line node, an empty abstract object denoting one of the nine cells on the board and an abstract sign node (Figure 1).

There are eights lines on the TTT (tic-tac-toe) board. Line 1 is an abstract object

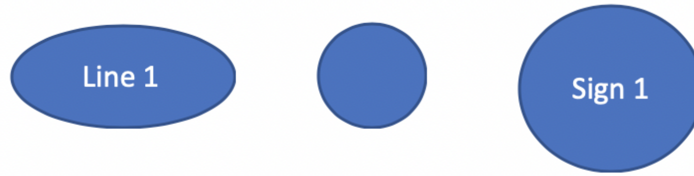

Figure 1: Abstract nodes in TTT game

that denotes one of these lines. If we would use object line 2, it will stand for one of the seven lines left, thus it will be any other line rather than line 1. The right object in the figure above with label 'Sign 1' denotes one of the two signs 'x' or 'o' possible to use in the game. The edge connecting the line and the cell indicates that this cell belongs to this line (Figure 2). Thus, according to the rules of the game each line has three connected cells, the edges between them cannot change during the game since they are constant. There can be a maximum of eight lines and nine cells.

Figure 2: Line connection with cell

The edge between the sign object and the cell object indicates that this sign was put in this cell (Figure 3).

Figure 3: Sign connection with cell

Such edges can change during the game: they cannot be removed, but can be placed, since the sign player cannot remove the sign, but can put it in one of the free cells.

The Figure 4 represents initial position of the game. Eight lines each one connected with three cells from the total set of nine cells and two signs not connected yet to any of

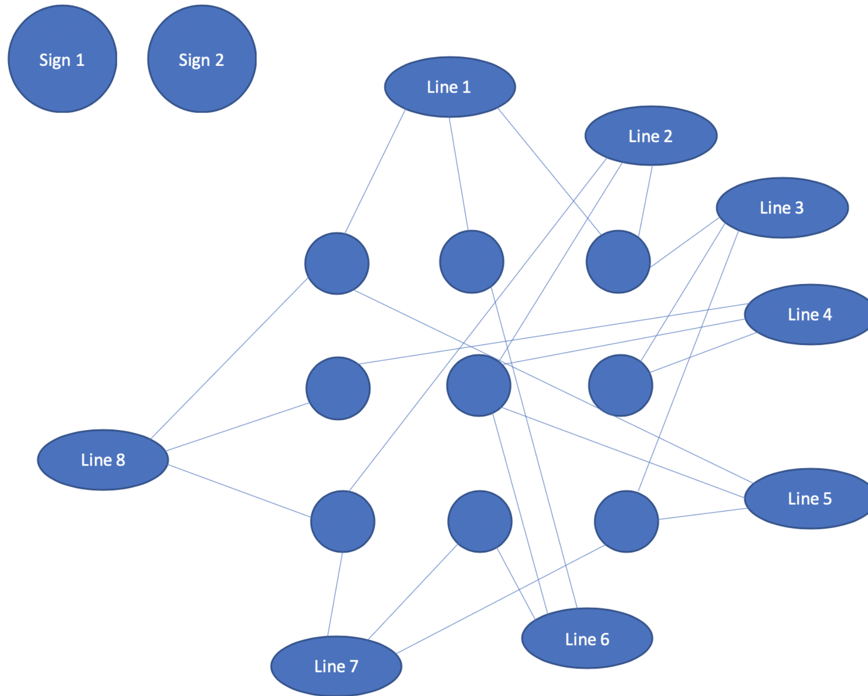

Figure 4: Initial board as a graph

the cells. As players will make actions, they will add connections between their signs and empty cells. Each cell can have at most one connection with one of the signs.

As the algorithm will work with graph data structure, there is need to describe main operations on the graphs such as Sum and Difference.

4.5 Sum of Graphs

To use the algorithm, the function of summing two graphs will be needed. This is a procedure in which two graphs are combined based on their common objects. Suppose we have two graphs shown in the Figure 5.

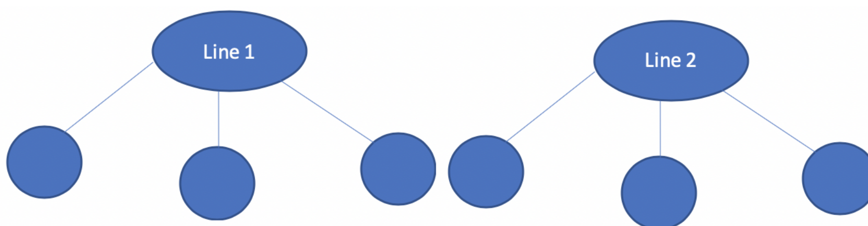

Figure 5: 2 graphs

These are two graphs each of which shows one of the lines on the board connected with three empty cells. The figure shows line one and line two, these are two different lines. In order to produce the sum of two graphs, it is necessary to identify which of the objects of the first graph can also be the objects of the second graph. Line one and line two are two different objects, but each of them is associated with three abstract cells. As far as we know, two arbitrary lines on the tic-tac-toe board can have one common cell. Thus, one of the three cells of the first line can also be one of the three cells of the second line that is, these two lines intersect in one of the abstract cells. This cell is the common object for two graphs. We can combine the two graphs as shown in the Figure 6.

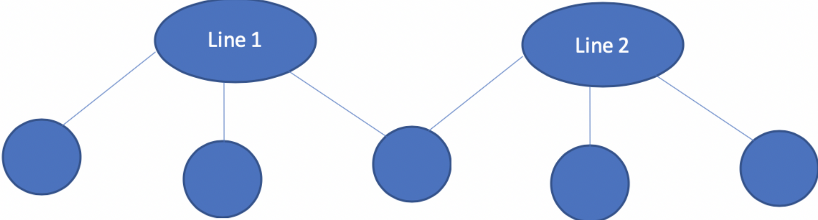

Figure 6: Graph Sum with one shared cell

If, according to the game, two lines could have two common cells, then these graphs could be combined as shown in the Figure 7.

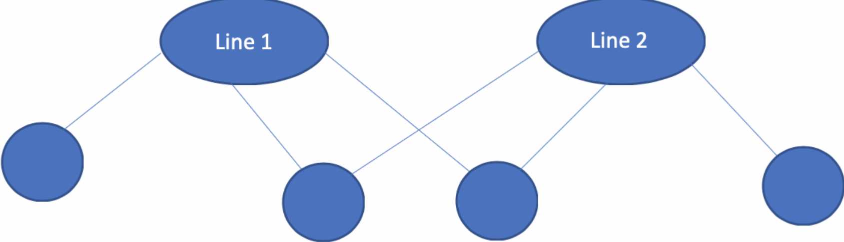

Figure 7: Graph Sum with two shared cells

Mathematically, sum of two graphs can be shown as

$$GraphSum(G_1, G_2) = G_1 \cup G_2 - G_1 \cap G_2$$

where G_1 and G_2 are two graphs, their union is combination of all of their objects and edges. Intersection is the set that includes all similar objects between two graphs. If graphs have some similar objects, these objects repeated twice, we need to subtract one intersection of G_1 and G_2 to get their sum.

4.6 Difference of the Graphs

Let's suppose, there are two graphs G_1 and G_2. G_1 has the set of its edges E_1 and G_2 has respective set E_2. G_1 is shown in the Figure 8, and G_2 is shown in the Figure 9.

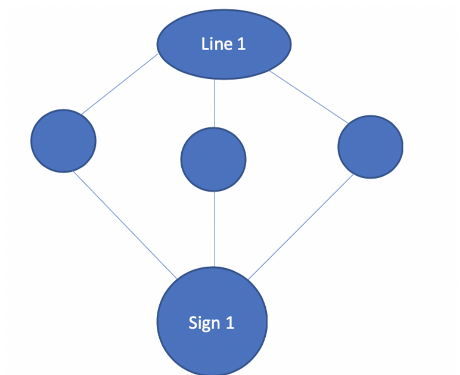

Figure 8: First term of difference

Figure 9: Second term of graph difference

E_1 has 6 edges and E_2 has one edge. If we take difference between G_1 and G_2, we delete all edges that are in E_2 from E_1. Thus, from G_1 we should delete one edge that connects 'Sign' object and 'Cell' object and we will get graph shown in the Figure 10.

It is important to note that G_1 has three edges that connect 'Sign' object and 'Cell' object, but we need to delete only one, since E_1 consists of one such an edge. There is no difference which edge out of these three to delete from G_1, the resulting difference

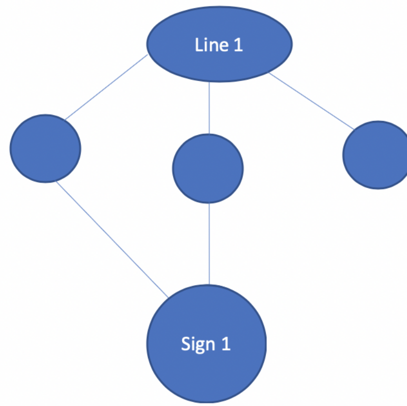

Figure 10: Graph difference

will denote the same abstract graph. All signs of the sum and differences shown in the equations below will refer exclusively to the sum and difference of the graphs.

4.7 Action as the difference of the Graphs

Action in our algorithm is the difference between two graphs,

$$a = G_1 - G_2 \ (5)$$

that indicates the transition from one graph to another. This difference can be shown using any basic operation on the graphs, including adding and removing edges and nodes. In such a way, we can show the action on the tic-tac-toe board as the difference between two graphs, where in the first graph there is a sign and a cell that are not connected to each other, and on the second graph they become connected, since one of the players has put his sign in this cell (Figure 11).

This is the only possible abstract action in the game - to put sign in a free cell, there are no more abstract actions.

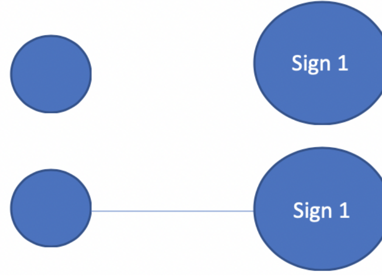

Figure 11: Action Graph in TTT game

4.8 Abstract description of the game

Using abstract data structure, tic-tac-toe game can be described in another way rather than in section 3.1. This will make it possible to use the Backward Algorithm with this game. The set of players, set of actions and final win state should be described.

$$P = \{p_1, p_2\} \quad (6)$$

Again, there is a set of players that consists of two players; however, at the moment the players are abstract, that is, each of them is not assigned to a specific sign.

$$A = \{a\} \quad (7)$$

There is a set of actions, that consists of single action. 'a' describes abstract action to put any sign in an empty cell. This is an abstract action that is not assigned to concrete cells, but to any empty cell as shown on the Figure 11 above.

$$s_{f_w} = \{s_f\} \quad (8)$$

This is a set of final win states. A state in which one player wins another. The state 's_f' describes situation when one of the players has put his sign in one line on the board.

The sets of actions and states described in this way are sufficient for use in the Backward Algorithm

4.9 Backward Algorithm

To use backward algorithm, there must be the following property of the considered game. For each action out of the set of actions of the game, there should exist such another action that may prevent implementation of the first action. Let's call these actions as preventive. For the use of algorithm, it is necessary to compose a set of the preventive actions.

$$I = \{i\} \ (9)$$

This is such a set in which there are actions that interfere with the execution of actions from the usual set of actions (7).

Set of preventive actions in TTT consists of only one action. According to the the game, preventing another player from placing his sign in a free cell is possible only by putting sign in this same cell before him. Two graphs of this preventive action are shown in Figure 12.

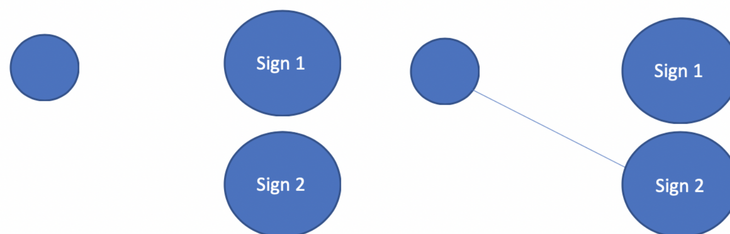

Figure 12: Prevention graph in TTT game

Player who plays with Sign 2 prevents another player from putting Sign 1 in the considered cell.

The Backward algorithm is an iterative algorithm, that will create the set of rules.

Each rule has input part, action and output parts. The algorithm will operate in reverse direction, describing initially the output part of each rule, then describing the action and finally getting the input by taking difference between output and action.

$$out_{r_1} = s_f$$

This is output of the first rule. It is equal to win of one of the players.

$$a_{r_k} = i, \text{where } i \in I$$

$$or$$

$$a_{r_k} = a, \text{where } a \in A$$

This is action of kth rule. It is equal to preventive action from I set or action from A set. When action in the previous rule equals to some a, $a_{r_{k-1}} = a$, action in current rule must be equal to such i that is preventive for this a, $a_{r_k} = i$. When action in the previous rule equals to i, $a_{r_{k-1}} = i$, action in current rule must be equal to any $a \in A$, that can be done in this state of the game, $a_{r_k} = a$.

$$in_{r_k} = out_{r_k} - a_{r_k}$$

This is input of kth rule. It is equal to difference between output and action of corresponding rule.

$$out_{r_{k+1}} = GraphSum(in_{r_k}, i) \ (10)$$

$$or$$

$$out_{r_{k+1}} = GraphSum(in_{r_k}, in_{r_j}) \text{ where } j \leq k \ (11)$$

Output of any subsequent rule is equal to GraphSum of input of previous rule and preventive action to this rule (10). This will create new rule which will make previous rule impossible to use. This will be useful when player needs to defend from attack of another player. Also, output can be equal to GraphSum of input of previous rule and input of any existing rule (11). Suppose, $player_1$ is at the $(k+1)^{th}$ step of creating rules, he already has k rules with their outputs, actions and inputs, he gets sum of the input of kth rule and input of any rule he already has. This will create a situation when an opponent is attacked in two different directions, so opponent needs to defend in both of them, but in games with one move at a time it is not possible. This is frequently called 'fork' situation. An example for such rule will be given in the following sections.

For tic-tac-toe game there will be five rules.

4.10 First Rule

The output signal of the first rule is equal to the final state of the game in which one of the players wins.

$$out_{r_1} = s_f$$

Without loss of generality, suppose this is the first player and he plays using the first abstract sign. It can be either 'x' or 'o'. The condition of his victory is that three of his signs are standing on the same line. (Figure 13)

Action of the first rule is the only abstract action from the action set.

$$a_{r_1} = a, \text{where } a \in A$$

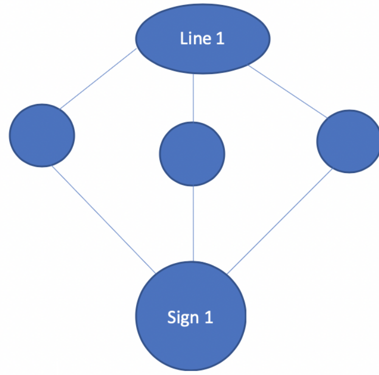

Figure 13: First rule output; Pl1 victory

It indicates action, when he puts third of his sign on the line. Then we subtract second term of the action graph from the output graph and get the input of the first rule (Figure 14).

$$in_{r_1} = s_f - a$$

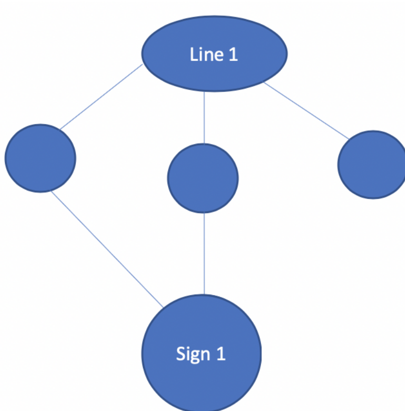

Figure 14: First rule input

There shown one possible example of real game position, where 'x' player can use first rule to put third 'x' in the cell with thick (Figure 15).

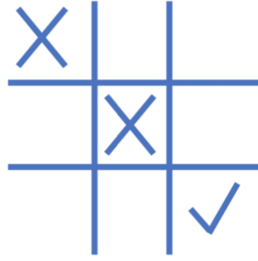

Figure 15: Example of first input. Win of 'x' player.

4.11 Second Rule

The output of the second rule is equal to the input of the first rule plus second term of a single abstract preventive action (Figure 16).

$$out_{r_2} = GraphSum(s_f - a, i)$$

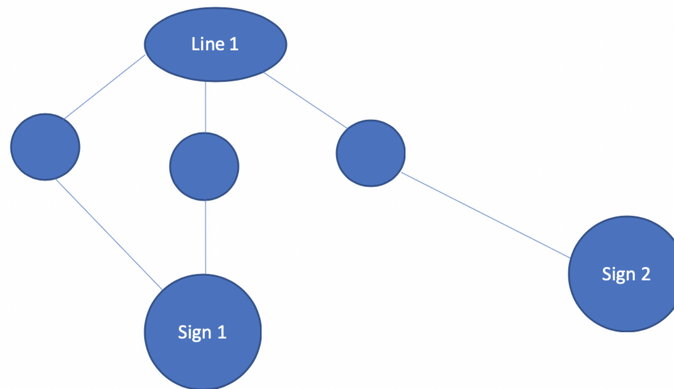

Figure 16: Second rule output

The second player will try to prevent the first player from putting his sign in an empty cell on the line, and thereby try to stop him from winning the game, having at the input two occupied cells with his opponent's sign (Figure 17).

$$a_{r_2} = i, \text{where } i \in I$$

$$in_{r_2} = s_f - a$$

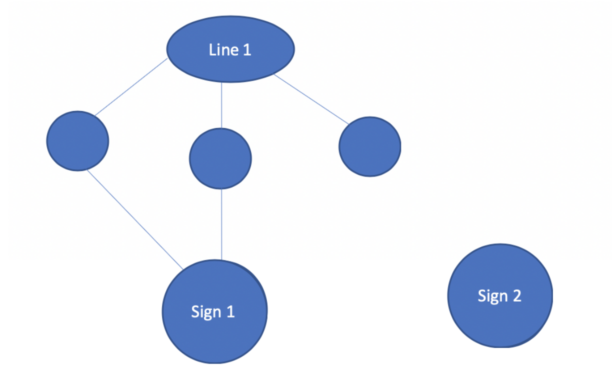

Figure 17: Second rule input

4.12 Third Rule

Creating an output of the third rule the first player will try to force second player to put the sign in another place, rather than the position described in the second rule. For this, he must know how he can force second player to perform some other action. At the moment, this is only possible using existing rules. The first player will try to create an input of two rules simultaneously.

$$out_{r_3} = GraphSum(s_f - a, in_{r_2})$$

The first player will have two connected lines that will be almost completely occupied by his sign except for one free cell in each line (Figure 18).

In TTT such a situation is called a 'fork' position. Similar to the procedure of the first rule, we subtract the only abstract action from this graph and get the input graph of the third rule (Figure 19).

$$a_{r_3} = a$$

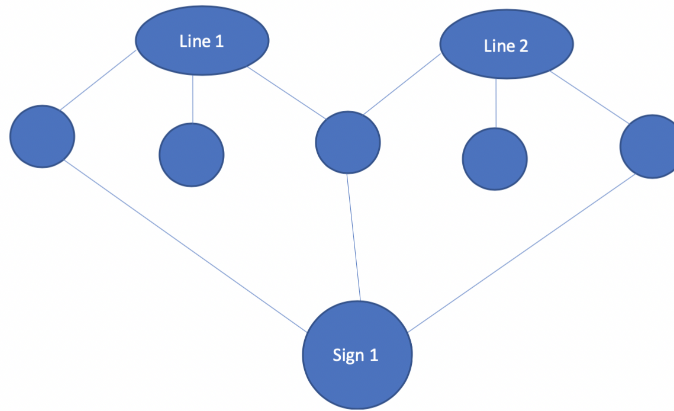

Figure 18: Third rule output, Pl1 fork

$$in_{r_3} = GraphSum(s_f - a, in_{r_2}) - a,$$

such that $in_{r_3} \neq in_{r_i}$, where i $<$ 3

Here is important to choose such an action from the action set, so that input of this rule will not be equal to input of any of previous rules, otherwise, the player will have to use the previous rule action.

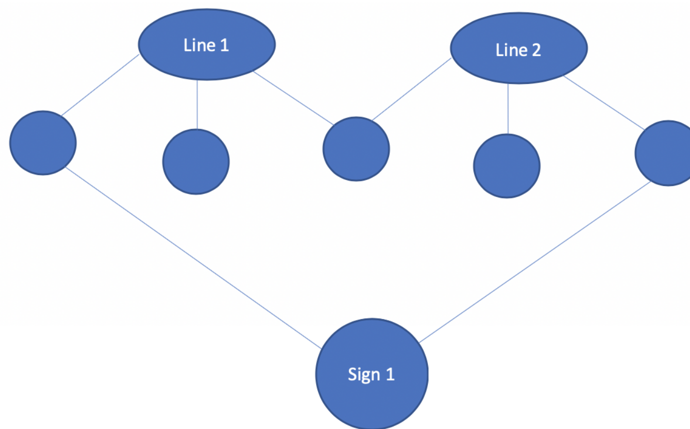

Figure 19: Third rule input

Figure 20 shows how player 'x' can put his sign in the cell with thick to create 'fork' situation.

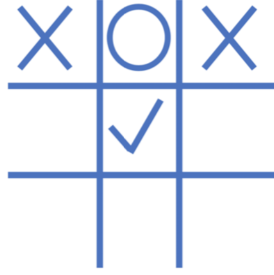

Figure 20: Example of third input. Fork of 'x' player

4.13 Fourth Rule

The second player again knows about the action of the first player and he adds the graph of preventive action to the input of the graph of the previous rule.

$$out_{r_4} = GraphSum(in_{r_3}, i)$$

If the second player sees that his opponent can almost occupy two lines in one move, then he will try to prevent him from using third rule by putting his sign at the intersection of these two lines and the first player will not be able to use the previous rule (Figures 21 and 22).

$$a_{r_4} = i$$

$$in_{r_4} = in_{r_3}$$

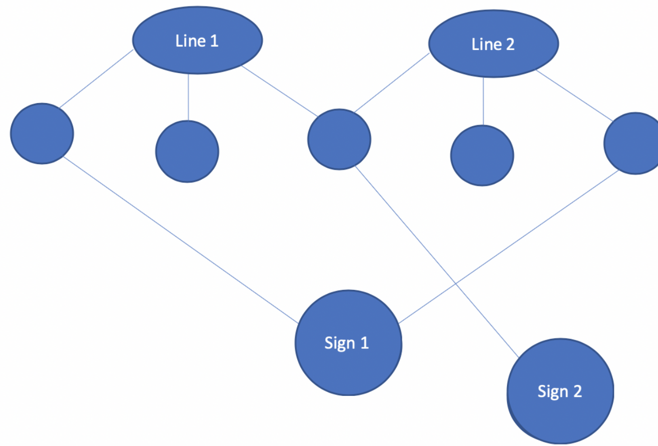

Figure 21: Fourth rule output

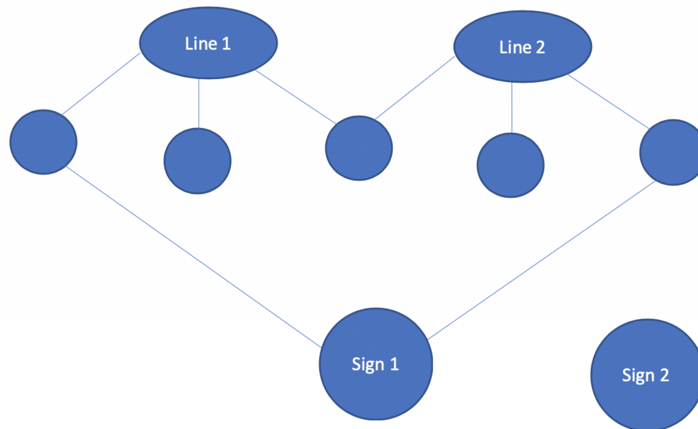

Figure 22: Fourth rule input

4.14 Fifth Rule

The first player, trying to interfere with the previous rule, will add the input of one of the previous rules again as in the third rule, in order to force the opponent to make another move (Figure 23).

$$out_{r_5} = GraphSum(in_{r_4}, in_{r_2})$$

Thus, in tic-tac-toe, when three lines are free from the sign of the second player and the intersection of two of the lines is occupied by the sign of the first player, the first player can put his sign at the intersection of other two lines, and thereby continue to use

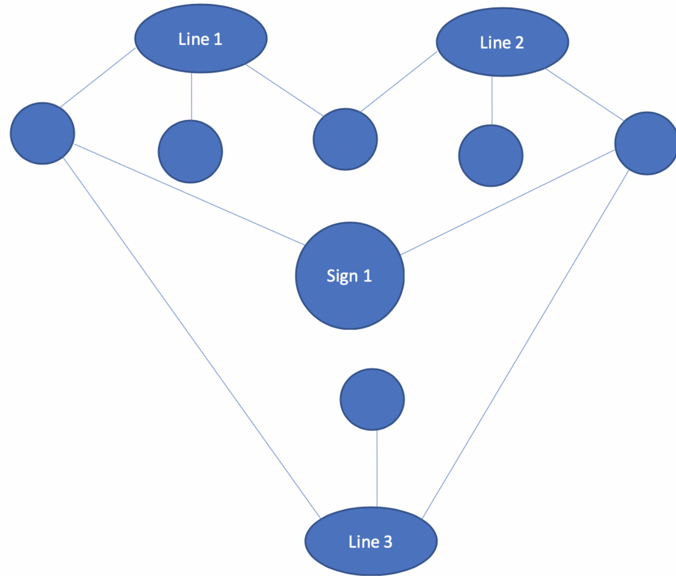

Figure 23: Fifth rule output

the odd attack rules.

$$a_{r_5} = a$$

$$in_{r_5} = out_{r_5} - a$$

This will allow him to win the game with certainty, if he finds himself in the situation of the input of this rule (Figure 24).

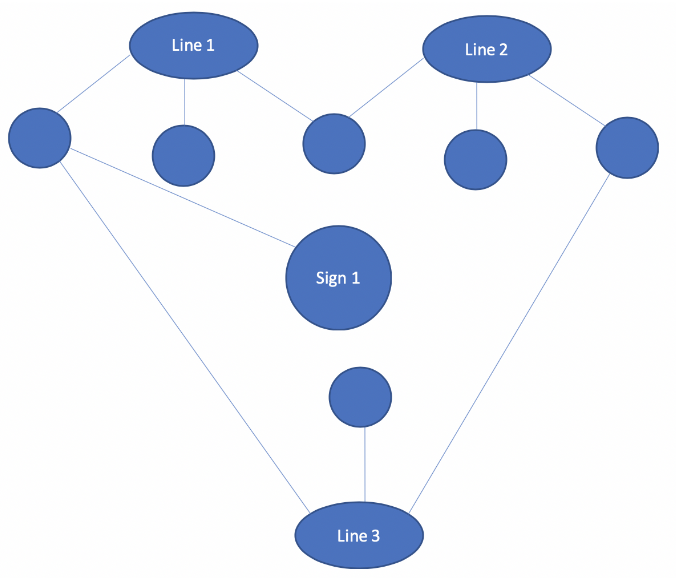

Figure 24: Fifth rule input

Thus we get a three structures of fifth rule. The example of real situation in the game is shown on Figure 25, where 'x' player can put his sign instead of one of the thick marks, which will allow him to create 'fork' situation further.

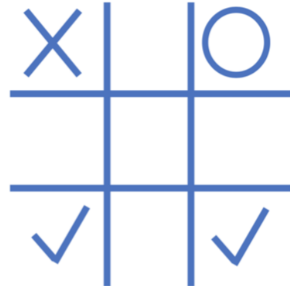

Figure 25: Example of fifth input. Example of 'x' player

The sixth rule here is impossible, since its input will be equal to the input of the third rule. But even using these five rules from the beginning of the game any player can guarantee that he will never lose the game.

5 Results of tic-tac-toe solution

The algorithm was tested on the game of tic-tac-toe, since it can be verified easily whether the rules are optimal in this case and graph is easy to create. If we would solve the game using tree, we need to consider nine possible actions on the first move, eight on the second and so on, in total we have 15120 possible positions. Using Backward algorithm there were five iterations of the program, from which we found five rules. Derived rules were compared to Aycock (2002) strategies, and appeared to be the same, together they create the set of optimal rules that allows to never lose the game.

The example of tic-tac-toe showed following logic, if we have state of two same signs in a line and third cell in a line is empty and it is our turn now, we choose action of putting third sign in this empty cell. Then we go one step backwards and estimate opponent's

rule in this situation. he needs to prevent first rule and go to abstract state, where it is impossible to put third sign for first player, so her optimal rule is to put her sign in this line. Remember, we do not consider other cells now, we only assume that the game was not ended yet and there is certain position on considered abstract line. Next rule was to create 'fork' situation on the board, when there are two possibilities to win on the second move. Then, this rule was defeated by 'safe' play of not allowing to create fork situation. This analysis provides us with the set of rules that allow to never lose the game, and this was reached in just five algorithm iterations.

In general, output of the first rule is always equal to final win state of the game. The input is equal to output minus abstract action from the action set. The odd rules in the rule set are attacking rules, player tries to prevent previous rule by either adding prevention action or by adding input of any of previous rules. By this, he guarantees that he can prevent the defense of the opponent. Each even rule is defense rule and it is created by adding preventive action to the input of previous attacking rule. If one of the players is in a situation which is the input state for an even rule, but at the same time, he does not use the action of this rule, that is, he deviates from the rule, then this allows the second player to use previous odd rule. If one of the players is in a state that is an input state for the attacking odd rule, this allows him to make impossible to conduct the previous protective even rule, which in turn will allow him to reach the very first rule and win the game.

It is important to notice that rules are applicable for abstract player and it does not matter what sign he plays. In the set of the rules, the left rules are more important because, they are closer to win situation. During the usual game each player should consider each possible action at the current state, and for each action he should compare the result

state of this action with the input of each rule starting from the very first rule. They should do it twice assuming one of the players as attacking and another as defending and vice versa. If one of the rule inputs is presented on the current board they should do the corresponding action of this rule. This will lead to playing such games without any losses.

6 Conclusion

To conclude, simple version of algorithm for solving finite zero-sum games with perfect information was developed. It showed sufficient results on the tic-tac-toe game, which was solved fully in only 5 iterations. This is much more effective than solving game using extensive form analysis, where we need to consider each of 15k possible positions. This algorithm as well as the Backwards Induction finds solution using game states' analysis. However, unlike the standard method of Game Theory, this analysis considers abstract states. One abstract state can describe several concrete states of the game at the same time, so the number of iterations of the algorithm can be reduced. Thus, this algorithm should be more effective in terms of the time spent searching for a solution. The main essence of the algorithm is that we describe the game using abstract graphs, and begin their analysis with the final state of the game; victory of one of the players. The first rule is created, where the player is one step from the victory, then new rules are added, where each rule prevents the execution of the previous rule. This process continues until there is no opportunity to add new rules that are different from previously created rules. As a result, a set of rules is created and it can be used for playing the game later by comparing current state's action results with the input state of each of the rules in the rules' set. To use this algorithm on tic-tac-toe, it is necessary to describe the actions of this game and the final state in the form of abstract graphs. For this game, this is a simple task,

since it requires only one graph for set of actions and one graph to describe the final win state. The number of such graphs may vary from game to game; perhaps chess will require a larger number of graphs due to the greater complexity of the game. Further research requires a more detailed mathematical study of the properties of this algorithm. The algorithm can also be tested on games such as chess and go, on which similar algorithms are often tested. A possible difficulty can be that creating an abstract graph that may require additional tricks.

References

[1] Aycock, Ryan. How to Win at Tic-Tac-Toe. 2002, http://www.cs.jhu.edu/ jorgev/cs106/ttt.pdf

[2] Bratko, Ivan. AlphaZero–What's Missing? In *Informatica* 42(1), 2018.

[3] Chess Arbiters' Association. FIDE Laws of Chess. 2018.

[4] Shannon, Claude. Programming a computer for playing chess. In *Philosophical Magazine* 41 (314), 1950.

[5] Ullmann, Julian R. An algorithm for subgraph isomorphism. Journal of the ACM, 23 (1), 1976.