The Art of Playing Games

Marko Cvijović[1], Mislav Unger[2], Jan Corazza[3]
[1]Gymnasium Užice, Užice, Serbia
[2]Gymnasium Bjelovar, Bjelovar, Croatia
[3]V. gymnasium, Zagreb, Croatia
aginor96@gmail.com
mislavunger@gmail.com
yannbane@yannbane.com

Abstract **- In our project, we applied artificial intelligence algorithms to games, in order to see how computers can compete against themselves and humans. The programming language we utilized for our main application was Java. We also used various open source software, such as the OpenCV computer vision library and the RXTX [6] serial communication library in order to connect our Java game-playing application to a robotic arm so it could see the game state and play against humans in the real world. Google Code and the NetBeans IDE were used as collaboration and software development tools, respectively.**

Keywords **- Artificial intelligence; Algorithms; Strategy; Robotics; Board games; Educational robotics; Computer vision**

I. INTRODUCTION

Artificial intelligence (abbreviated as AI) is a branch of computer science that deals with development and application of intelligent behavior in machines and software. AI is further divided into subsectors that explore concepts such as computer vision, planning, learning, communication, prediction, spatial orientation, representation and reasoning [7].

Artificial intelligence algorithms are finding their way into various levels of society with an astonishing pace: Google's self-driving cars, industrial processes such as aircraft design, complex data analysis for the stock market, multidisciplinary science (especially biology and astronomy), etc. The reason for their huge success is the fact that they allow us to use the advantages of our own brains' ways of analyzing data, learning, and memorizing, and combine them with the vast computing power and other technological conveniences of our machines.

A concrete example of such a system would be Google self-driving cars[8], which use these algorithms and many other intriguing technologies in order to solve real-world problems and make the lives of humans easier. The project was originally led by Sebastian Thrun and Chris Urmson in secret, but Google is currently performing tests in public.

The games we have analyzed were one-player or two-player games that could easily be expressed with simple logical conditions. We restricted our scope to games where the game playing progress can be partitioned to a sequence of elementary pick and place moves with optional modification of states of game elements. This included board games like mill and reversi, but excluded real-time video games. Even very complex board games such as chess could be described in our system, but we focused on those games, where general algorithms could be applied without having to feed the system with substantial expert knowledge and game-specific information.

A similar, but quite more advanced application, is Watson, an IBM machine that won the Jeopardy game in a TV show [9].

II. MATERIALS AND METHODS

As the crux of our project were games, we also played them a lot, and manually discovered strategies for them and did some further mathematical analysis. At some point, these games needed to be transferred to the computer, so we needed a common language for describing them and how they progressed, and we had to find a method of representing them in the computer.

The common language, with which we could define the games, was a custom expression language made by our project leaders. Using it, we managed to develop a game called Connect Four. The goal of the game is to align four tokens in a line, by dropping them into a grid. The game is played by two players, each of which use differently-colored tokens.

Figure 1. Traditional Connect Four game.

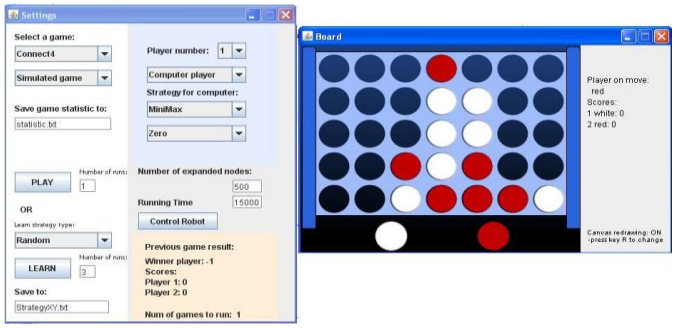

Figure 2. The Java game interface.

```
[Expressions]

SameElementType
  ELTYPE(CONTENT("b($X,$Y)")) ==
$TP
  $CNT = ($CNT + 1)
  $X = ($X + $DX)
```

```
Count($DX,$DY)
  $X = ($L + $DX)
  $Y = ($M + $DY)
  $CNT = 0
  FORALL($A,1,3,SameElementType) OR
true
  $CNT
END
```

Code 2

The example shows a function that verifies whether the player has placed a 4th winning token in direction dx, dy. It uses variables, assignments, built-in operators ==, OR, +, and functions ELTYPE, CONTENT, FORALL. Full specification of the language is in the project wiki [1], and full rules of Connect4 can be found in games/ folder in the open-source project repository.

A. Internal representation

The board games we studied, had a finite number of game states, which described the game configuration in a particular moment in time. Any game state could be expanded to obtain the set of game states reachable by performing one move.

One of the problems that had to be overcome was the fact that game states could roll back into a loop – for example, if moving a figure A to a certain position, and then moving it back again, was a legal sequence of moves, then there are infinite moves in the game – which could be quite problematic to our algorithms, so we had to remember which states we had visited before. Thus the resulting state space formed a rooted tree – known from the graph theory as a connected graph without loops with states represented by nodes and moves represented by edges.

B. S3Games

Our main tool for this project was a Java framework called S3Games. It was prepared by our project leaders, and its goal was to simplify the definition of new games and the creation of new algorithms. It includes a GUI for playing and monitoring games, as well as a custom language for rules.

The project is maintained at [1], and is about 8k lines of code long.

C. Algorithms

We used algorithms that were compatible with graphs, that is, they could be applied to some form of graph traversal and analysis.

Our algorithms had two main characteristics: the most important one being whether they were deterministic or stochastic. Deterministic algorithms would give the same output each time they were applied to the same input (some initial game state). This practically meant that the algorithms would try to explore all of the possibilities, one by one (exactly how it did that depended on the concrete implementation). This might seem as a desirable feature, but in reality it imposes serious limitations on the whole process. We learned how to mitigate these shortcomings by introducing stochastic elements.

The stochastic algorithms rely on statistics, and traverse only a subset of the game state neighborhood. Again, the details depend on the implementation, but we usually only chose a percentage of the state neighborhood and ignored the rest. This allowed the program to simulate the game much further into the future given the same amount of time, and still be able to get satisfying results in terms of the moves it chooses to make.

1. Deterministic

BFS and DFS

The Breadth First Search algorithm is one of the two simplest graph traversal algorithms which we learned about. It tries to expand all of the nodes, and progresses level through level. If, for example, state A led to states B, C, D, and E, the algorithm would process them in that order – ignoring the children of the nodes in the second row.

This algorithm wasn't efficient since the number of nodes it needed to process increased exponentially, and it took a very long time to reach the end. Statistically, it is not needed to process all of the states to get good results.

Depth First Search is just as primitive (for this type of task)

as BFS. It functions on a similar yet opposite principle: it tries to get to the leaf (ending) nodes first, and only then returns from there to process others.

E.g., if there was a graph with a root node A, two children D and E, and if D had yet another two children, B and C, the algorithm would process the nodes as follows: A, D, B, C, E. It expanded the D node and processed its children before processing E, even if they were on the same depth.

Just as BFS was too broad to return useful results in this particular case, DFS was too specific. It would *definitely* reach ending states (leafs), but not nearly enough of them to actually make it statistically relevant.

AStar search

The nodes are stored in a priority queue, i.e. when retrieving the next node, we always take the node with the highest priority. The priority depends on the function (lower is better):

$$f(state) = h(state) + g(state)$$

Equation1:

where $h(state)$ is a heuristic evaluation of the state of that node, i.e. the expected distance to the goal state, however it must comply with the restriction $h(state) <= d$, where d is the real distance to the goal state, and $h(state) > 0$. And $g(state)$ is the distance from the root state, i.e. the number of moves we would have to make from the state where we started searching.

MiniMax

All of the aforementioned algorithms suffer yet another fatal flaw: they never account for the opponent. BFS and other algorithms' results are of little to no value in two player games, where it is not enough to simply iterate over all possible states, since in two player games the decision making process alternates between and depends on two parties.

Thus, an algorithm which considered the opposing player was needed to be developed. The name MiniMax comes from the fact that the algorithm tries to predict the moves of the opponent based on minimizing its success, and tries to make moves based on maximizing it.

2. Stochastic

Monte Carlo method

The Monte Carlo method is the general method of constructing stochastic algorithms based on statistics already described above. Specifically, in our case, it works by evaluating only a subset of all of the possible paths a game could take after a move is taken, thus

MiniMax Stochastic

With this algorithm we combined the predictive power of the original MiniMax algorithm and the efficiency of the Monte Carlo method. We supposed that by doing this, we would develop a strategy better than both of them. This modification was a relatively simple one, as we only needed to restrict the possible pool of next moves from a node to random subset, as described above.

Improvements over the original Monte Carlo method

We noticed how the original Monte Carlo algorithm had a serious issue, which was the fact that it didn't consider how many moves it takes to reach some final state, and how big was the chance of an opponent making a move to a particular node. That is, a victory in a hundred moves would be viewed the same as a victory in the next two moves – which is obviously a suboptimal consideration of the current state. We fixed this by making the deeper nodes less relevant. At first, this was done by dividing the value of a node by the breadth of branching in all nodes all the way from the root node to the considered winning state.

We also implemented another, less strict method of trivialization – multiplication by a constant factor. The factor we used was 0.9, but it was only an estimate, not an empirically satisfying value (to achieve that, we could, for example, use evolutionary programming).

D. Robotics

Since most humans are familiar with these games in their physical, as opposed to virtual, form, it was only logical to add the ability to play against the computer in this much more natural way of interaction.

For this purpose, a Lynxmotion robotic arm [4] which was assembled by our mentors was used. The arm consisted of six servo motors, controlled by an on-board AVR microcontroller. The microcontroller received commands from a PC over a serial port [5]. The RXTX library [6] was used in order to facilitate this connection and make our application capable of controlling the robot.

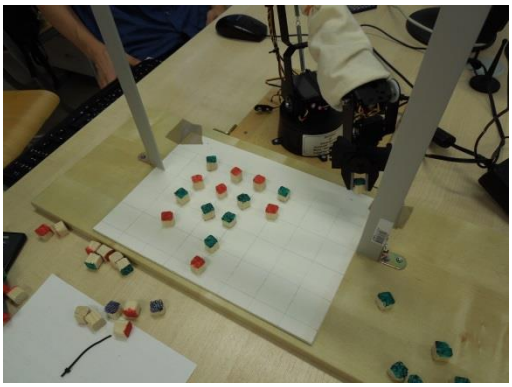

Figure 3. The robotic arm playing a game.

An additional component of the whole setup was a web camera, which allowed the computer to "see" the current game state, and interpret it in its internal data structures. We used the

OpenCV computer vision library [3] to detect the variously colored pads on the game board, and determine their positions.

E. Overnight simulation marathon

Our most important experiment was our game simulation marathon. We used twelve computers in order to compare various combinations of our algorithms playing against each other overnight.

III. Results

We measured our algorithms according to the Connect Four game we had implemented ourselves.

Table I shows the results we got from the simulations.

These data suggest that the classic Monte Carlo method is the best algorithm we implemented, since the simpler algorithms such as BFS and DFS could not really compare.

It is evident that the Monte Carlo method (labeled MC in Table I), is the best performing, since it had a win percentage of over 50 against every other algorithm.

We also had humans play against the machines – both simulated and real board. They mostly won against the random control, and lost against more advanced algorithms, however, it would be statistically incorrect to draw any conclusions, since the number of games was too low.

Table I: Percentage of the games won in all different algorithm pairs. Pairs played 52-100 games, changing the starting player after half of them.

	MM	**MMS**	**MC**	**MCR2**
MM	x	82,93	41,30	48,08
MMS	8,54	x	11,67	3,19
MC	54,35	88,33	x	51,72
MCR2	44,23	96,81	46,55	x
	MCR	**MCR2**		
MCR	x	31,93		
MCR2	68,07	x		

Legend: MCR2 – Monte Carlo with a fixed point factor, MC – original Monte Carlo, MMS – stochastic MiniMax, MM – MiniMax, MCR – Monte Carlo with breadth-dependent division.

IV. Discussion

The most surprising result we've had was the poor performance of the stochastic MiniMax. We had hoped that it would outperform both Monte Carlo family and the original MiniMax, but it managed to do neither. Our explanation is that the program doesn't manage to expand enough nodes and do the required computations on them for the resulting decision to be statistically accurate. It is also too likely to ignore an important move that is close to the root node, thus giving the opponent a huge advantage. We hypothesized that the performance could be improved by providing more running time, and skipping only own moves, but not the moves of the opponent.

A lot more useful data could be gathered by more testing with varying parameters – how much time each algorithm is given, how many nodes it is allowed to expand, etc.

Another expansion of this study would also be implementing more different games.

Acknowledgment

Pavel Petrovič and Zuzana Koyšová were our project leaders and we're all very grateful to them for this interesting and informative project.

References

[1] Project open-source repository and wiki at Google Code. Available online: http://code.google.com/p/s3games/

[2] JAVATM Platform, Standard Edition 7 API Specification, Oracle. Available online: http://docs.oracle.com/javase/7/docs/api/

[3] OpenCV online documentation, Available online: http://docs.opencv.org/

[4] Lynxmotion Robot Arm AL5D. Available online: http://www.lynxmotion.com/c-130-al5d.aspx

[5] Jim Frye: SSC-32 Manual, Lynxmotion, 2010. Available online: http://www.lynxmotion.com/images/html/build136.htm

[6] Keane Jarvi: RXTX Library. Available online: http://rxtx.qbang.org

[7] Stuart Russell, Peter Norvig: Artificial Intelligence: A Modern Approach (3rd Edition), Prentice Hall, 2009.

[8] Erico Guizzo: How Google's Self-Driving Car Works, IEEE Spectrum, 2011. Available online: http://spectrum.ieee.org/automaton/robotics/artificial-intelligence/how-google-self-driving-car-works

[9] Eric Brown, Eddie Epstein, J William Murdock, Tong-Haing Fin: Tools and Methods for Building Watson, IBM Research Report RC25356, 2013.