Development of an Android Application for the Game of Quantum Tic-Tac-Toe

Konstantinos D. Smanis
Dept. of Electrical & Computer Engineering
University of Patras
Rion, Patras, Greece
konstantinos.smanis@gmail.com

***Abstract*—This diploma dissertation presents the research, design and development of an Android application for the game of quantum tic-tac-toe. Initially we examined the rules that govern the game of quantum tic-tac-toe, the relevant algorithms for its solution and the capabilities provided by the Android development platform. Subsequently we implemented a Java library for solving the game, which was later used, with minor modifications, for the development of the Android application. The said application allows the user to play quantum tic-tac-toe games against other users or against the computer, in varying levels of difficulty. In addition, the game's capabilities include finding the optimal move in a matter of seconds in every case. This feature comes as a result of extensive optimizations in the game-solving algorithm, such as alpha-beta pruning and move ordering.**

Keywords—quantum tic-tac-toe; android application; minimax algorithm; alpha-beta pruning; move ordering

I. Introduction

Quantum tic-tac-toe is a two-player, zero-sum game of perfect information that was developed by Allan Goff [1] as a teaching metaphor for fundamental concepts of quantum mechanics. It is a quantum generalization of the venerable game "tic-tac-toe" [2], which adds only one rule: superposition.

According to Ishizeki and Matsuura [3], the game of quantum tic-tac-toe can be solved and it is shown that the first player ("X") has an advantage over the second player ("O"). Given perfect play, the first player always wins with a "narrow win". Narrow win is a special case in the game of quantum tic-tac-toe, in which both players achieve three marks in a row, but only the first player to do so wins.

The objective of this thesis is the development of an Android application ("app") for the game of quantum tic-tac-toe. The user should be able to play in real-time against the "computer" or another user, through a modern, interactive interface. Key characteristics of the app should be simplicity, responsiveness and ease of use.

Quantum tic-tac-toe can be played on paper or on a white board, but the use of a digital platform allows for increased usability. The app disallows invalid moves and provides quick navigation to a game's move history, visual feedback for game events, etc. Moreover, it offers an AI ("computer" player) with varying levels of difficulty. At the time of writing of this thesis (January 2016), no other app or publically available software is known to offer similar capabilities.

Quantum tic-tac-toe has relatively simple rules, which can be easily modelled and debugged using the minimax algorithm. However, the search space of the game is exponentially bigger than in classical tic-tac-toe. Given that the app is meant to be used in portable devices with limited hardware, it is obvious that the minimax algorithm is in no way enough for our purposes. Therefore, the game-solving algorithm should be heavily optimized.

The rest of this thesis is structured as follows. In Section II, the theoretical background of the problem at hand is given. In Section III, the implementation of the app is discussed. In Section IV, the operation of the app is explained, followed by improvement suggestions in Section V. Finally, concluding remarks are given in Section VI.

II. Theoretical Background

A. Game Rules

1) Superposition

As in classical tic-tac-toe, two players ("X" and "O") take turns placing their marks on the game board. However, in quantum tic-tac-toe, the players must place *a pair* of marks per turn, in contrast to classical tic-tac-toe, where the players place a single mark per turn. These marks are denoted by the player's name ("X" or "O"), subscripted with the number of the move during which they were played and must be placed in different cells, if possible. Consequently, "X" places the marks X_1, X_3, X_5, X_7, X_9, while "O" places the marks O_2, O_4, O_6, and O_8 (see Fig. 1).

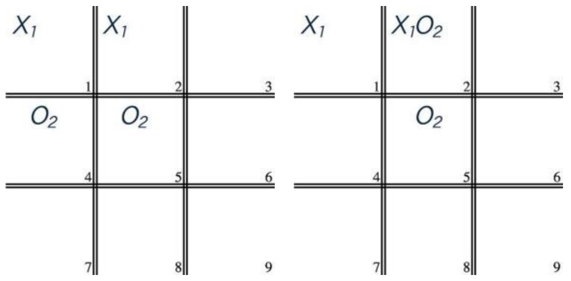


Fig. 1. Examples of (incomplete) quantum tic-tac-toe games. (Source: [1])

These marks are derived from the notion of superposition in quantum mechanics and are, therefore, called “quantum” or “spooky” marks. Quantum marks are not fixed, i.e. they exist in two cells simultaneously, until a special condition is met, “cyclic entanglement”, which will be analyzed in the following subsections. It follows that each cell may contain multiple quantum marks, otherwise by the fifth move, “X” would not be able to place both of his marks.

The superposition rule seems to imply that in reality we are playing multiple games of classical tic-tac-toe, *simultaneously* (see Fig. 2).

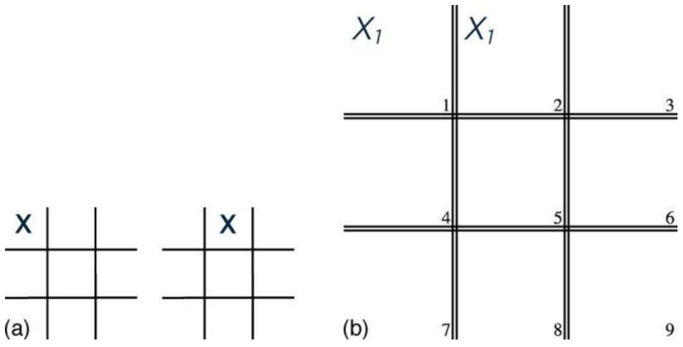


Fig. 2. The game state in quantum tic-tac-toe is equivalent to multiple games of classical tic-tac-toe, which are called the “classical ensemble”. These games are in simultaneous play, but they are not independent. (Source: [1])

2) *Entanglement*

As the game progresses, the players’ moves are inevitably intertwined every time a number of them share the same cell. In this state, they no longer are independent and, instead, influence one another. The game’s term for this state is “entanglement”.

In Fig. 1, we saw two distinct cases of game development. In the first game, the moves are independent, i.e. the states are separable. However, in the second game, the moves are entangled, which leads to contradictory classical games (see Fig. 3). Contradictory games are eliminated (pruned) from the classical ensemble.

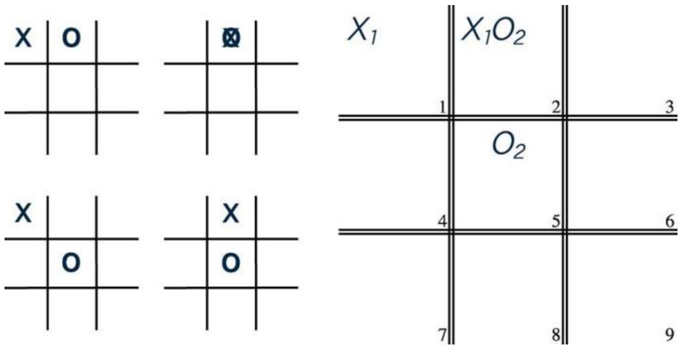
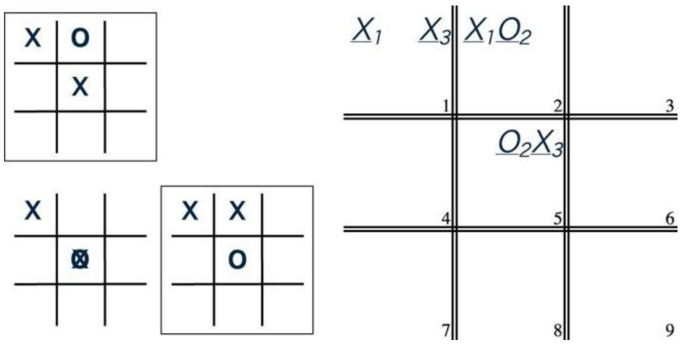


Fig. 3. In the scenario above, if X_1 ends up in cell 2, then O_2 will end up in cell 5. If, however, O_2 ends up in cell 2, then X_1 will end up in cell 1. (Source: [1])

Cyclic entanglement is a special case of entanglement, in which the participating quantum marks form a circular logic. For example, in Fig. 4, if X_1 ends up in cell 1, then X_3 will end up in cell 5, O_2 will end up in cell 2 and X_1 will end up in cell 1. Due to the circular nature of the entanglement, we are lead back to our initial assumption.



$\underline{X_1}$
$\underline{X_3}$
$\underline{X_1}\underline{O_2}$
$\underline{O_2}\underline{X_3}$

Fig. 4. Example of cyclic entanglement in a game of quantum tic-tac-toe. Quantum marks that are part of the cyclic entanglement appear underlined. All, but two, of the classical games are contradictory and are, therefore, eliminated. The only possible outcomes of the game appear in a square outline. (Source: [1])

3) *Collapse*

In the event of a cyclic entanglement, a “measurement” is performed. The game state is “observed”, i.e. the quantum state of the game “collapses” into a single reality, where the cyclically entangled quantum marks are fixed to cells. No matter how complex a cyclic entanglement is, there will always be exactly two realities (possible outcomes) to choose from; all the other outcomes will be contradictory. In order to balance the game strategically, this choice is made by the player who *did not* create the cyclic entanglement. Otherwise, as shown by Ishizeki and Matsuura [3], the first player (“X”) holds an unfair advantage, because, given perfect play, he always wins.

Whenever a pair of quantum marks is observed, one of them collapses into a “real” or “classical” mark, while the other one vanishes. In order to win, a player must have three real marks in a row. Cells that contain a real mark prohibit any further actions; the mark is fixed in the cell until the end of the game.

Much like a normal move, choosing a reality to collapse into is a strategic decision. However, it does not count as a normal move, i.e. the player still has to perform a normal move after selecting an outcome. Consequently, after a cyclic entanglement occurs, the player has two consecutive decisions to make.

4) *Special Cases*

A noteworthy aspect of the measurement process in quantum tic-tac-toe is that quantum marks collapse in groups, thus allowing for simultaneous sets of three marks in a row (see Fig. 5).

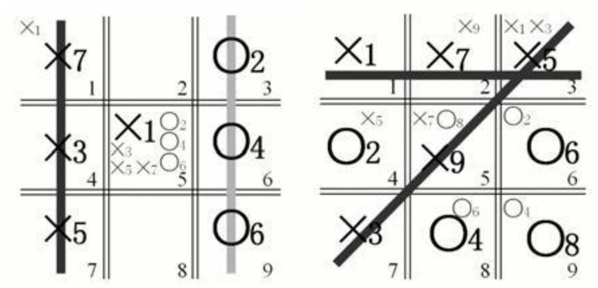


Fig. 5. Multiple sets of three marks in a row. In the first case, both players achieve three marks in a row. In the second case, the first player achieves a double win! (Source: [3])

The most common case is when both players achieve three marks in a row. In this scenario, the winner is the player whose largest subscript of the three marks is smaller. In other words, if the classical marks were added sequentially, as in classical tic-tac-toe, the losing player would have never completed his set of three marks, since the game would have already ended by then. This outcome of the game is called a narrow win.

A somewhat rare scenario is when the first player achieves two sets of three marks in a row simultaneously; a double win! It is worth noting that the second player is not capable of a double win, because it requires five different marks. This is yet another advantage of the first player, although it is not one of great significance, since it requires some very bad moves on behalf of the second player in order to happen.

Last but not least, it is possible that by the ninth move only one cell remains open, i.e. the rest of the cells contain real marks. In this scenario, the first player places both of his quantum marks (X_9) in the same cell and they instantly collapse into a real mark.

B. *Artificial Intelligence & Game Theory*

1) Games

In the field of Artificial Intelligence (AI), adversarial search problems are often known as games [4]. These problems usually come up in multiagent, competitive environments, in which the agents' goals are in conflict. In these environments, the actions of one agent influence the problem-solving process of other agents, by introducing contingencies. Game theory, on the other hand, views any multiagent environment as a game, regardless of whether the agents cooperate or compete, provided they influence one another significantly.

One of the most common cases that AI studies, is deterministic, two-player, turn-taking, zero-sum games of perfect information. In AI terminology, this means that the two agents play alternately in a deterministic, fully observable environment. In addition, the gain of one agent is equal to the loss of the other agent, i.e. the utility values at the end of the game are always equal and opposite. A prime example of this game category is (quantum) tic-tac-toe.

2) Optimal Decisions

At any point of a game, the optimal decision for a player is the action that will lead to the best outcome for him, taking into consideration every possible response from the opponent(s) and assuming perfect play. An optimal strategy is a sequence of optimal decisions, thus defining the player's moves at any point of a game.

The advantage of an optimal strategy is that it maximizes the player's gains against an infallible opponent. In addition, if the opponent is not perfect, an optimal strategy will lead to even better results. Other strategies may do better against suboptimal opponents, but these strategies will necessarily do worse against optimal opponents.

3) Minimax Decision

Given a game tree, we can assign numeric values to its nodes that help us determine an optimal strategy (see Fig. 6). These values are called minimax values. The minimax value of a node is interpreted as the utility of being in the corresponding state for the player MAX, assuming perfect play from both players till the end of the game.

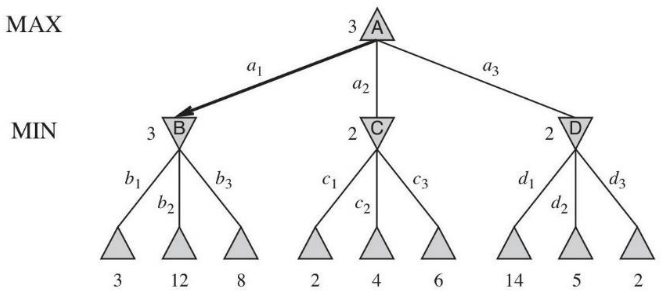


Fig. 6. A game tree with minimax values assigned to nodes. In Δ nodes, it is MAX's turn to move, whilst in ∇ nodes, it is MIN's turn. MAX prefers moves with maximum value, whereas MIN prefers moves with minimum value (hence the players' names). (Source: [4])

According to these, we have the following:

$$\text{Minimax}(n)=\begin{cases}\text{Utility}(n), & \text{if } n \text{ is a terminal node}\\ \max_{s\in \text{Successors}(n)}\text{Minimax}(s), & \text{if } n \text{ is a MAX node}\\ \min_{s\in \text{Successors}(n)}\text{Minimax}(s), & \text{if } n \text{ is a MIN node}\end{cases}$$

The utility function returns a numeric value for a terminal state according to the game rules. The successor function returns the set of states that result from applying every legal move in a state.

The minimax decision is defined as the action that leads to the node with the highest minimax value for player MAX, while for player MIN it is defined as the action that leads to the node with lowest minimax value.

4) Minimax Algorithm

The minimax algorithm (see Fig. 7) computes the minimax decision from the current game state by performing a complete depth-first exploration of the game tree. The algorithm recurses down to the leaves of the tree and then as it unfolds, it copies the calculated minimax values to the parent nodes.

The time complexity of the algorithm is $O(b^m)$ and the space complexity is O(bm), where "b" is the average branching factor and "m" is the maximum depth tree. The time complexity renders the algorithm impractical for the vast majority of games, yet it is often used for the theoretical analysis of games and serves as the basis for more practical algorithms.

```
function MINIMAX-DECISION($state$) returns $an\ action$
  return $\arg\max_{a\ \in\ \text{ACTIONS}(s)}$ MIN-VALUE(RESULT($state$, $a$))

function MAX-VALUE($state$) returns $a\ utility\ value$
  if TERMINAL-TEST($state$) then return UTILITY($state$)
  $v \leftarrow -\infty$
  for each $a$ in ACTIONS($state$) do
    $v \leftarrow$ MAX($v$, MIN-VALUE(RESULT($s$, $a$)))
  return $v$

function MIN-VALUE($state$) returns $a\ utility\ value$
  if TERMINAL-TEST($state$) then return UTILITY($state$)
  $v \leftarrow \infty$
  for each $a$ in ACTIONS($state$) do
    $v \leftarrow$ MIN($v$, MAX-VALUE(RESULT($s$, $a$)))
  return $v$
```

Fig. 7. The minimax algorithm calculates the minimax decision in the current game state for the player that pursues maximisation of utility, assuming the opponent aims at minimisation of utility. The algorithm performs a recursive, depth-first traversal of the game tree, copying the minimax values as the recursion unwinds. (Source: [4])

5) *Alpha-Beta Pruning*

The astute reader may have noticed that it is possible to compute the correct minimax decision without traversing every single node in the game tree. By applying pruning techniques in the minimax algorithm it is possible to reduce the time complexity exponent almost by half, i.e. it is possible to solve a game tree roughly twice as deep as minimax in the same amount of time. The pruning method examined in this subsection is called alpha-beta pruning.

The general principle of alpha-beta pruning is relatively simple. Consider a node “n” somewhere in the game tree (see Fig. 8), such that Player has a choice of getting to it. If at any decision point further up Player has a better choice (e.g. node “m”), then “n” will never be reached in play. Therefore, the subtree with the root node “n” may safely be pruned, as soon as enough of its descendants have been examined.

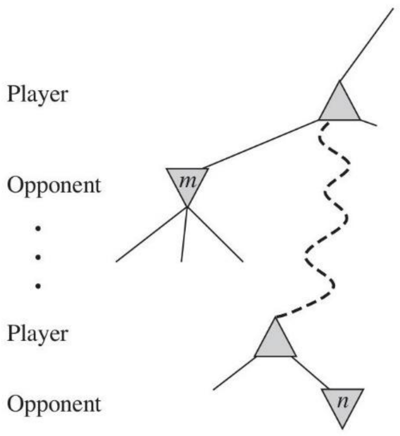


Fig. 8. The general case for alpha-beta pruning. An optimal player would never reach node “n” in actual play, if node “m” is better. (Source: [4])

The alpha-beta algorithm (see Fig. 9) is a variation of the minimax algorithm that produces the same result, while pruning away the game tree branches that cannot possibly influence the final decision. In this manner, large parts of the game tree are completely ignored, even entire subtrees of considerable size. Although the alpha-beta algorithm is surprisingly similar to the minimax algorithm, it is way more efficient.

```
function ALPHA-BETA-SEARCH($state$) returns an action
  $v \leftarrow$ MAX-VALUE($state, -\infty, +\infty$)
  return the $action$ in ACTIONS($state$) with value $v$

function MAX-VALUE($state, \alpha, \beta$) returns $a\ utility\ value$
  if TERMINAL-TEST($state$) then return UTILITY($state$)
  $v \leftarrow -\infty$
  for each $a$ in ACTIONS($state$) do
    $v \leftarrow$ MAX($v$, MIN-VALUE(RESULT($s,a$), $\alpha, \beta$))
    if $v \geq \beta$ then return $v$
    $\alpha \leftarrow$ MAX($\alpha$, $v$)
  return $v$

function MIN-VALUE($state, \alpha, \beta$) returns $a\ utility\ value$
  if TERMINAL-TEST($state$) then return UTILITY($state$)
  $v \leftarrow +\infty$
  for each $a$ in ACTIONS($state$) do
    $v \leftarrow$ MIN($v$, MAX-VALUE(RESULT($s,a$) , $\alpha, \beta$))
    if $v \leq \alpha$ then return $v$
    $\beta \leftarrow$ MIN($\beta$, $v$)
  return $v$
```

Fig. 9. The alpha-beta algorithm is a variation of the minimax algorithm that makes use of alpha-beta pruning. (Source: [4])

III. IMPLEMENTATION

The development process was carried out in two separate stages: a backend and a frontend.

The backend comprises a set of algorithms and structures that are necessary for the implementation of a quantum tic-tac-toe game. The frontend, on the other hand, is a visual wrapper around the backend that allows interacting with it through a pleasant and user-friendly GUI (Graphical User Interface).

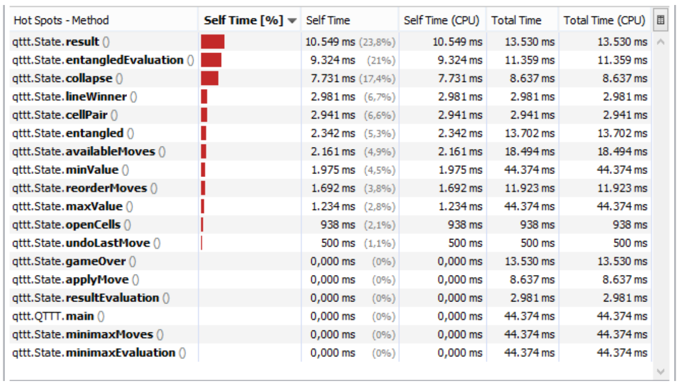
The backend was implemented in pure Java [5] and was later incorporated, with minor modifications, into the Android application, since the Android development platform is mostly compatible with the Java SE API (Application Programming Interface) [6]. The frontend, a native Android app, was built using the Android SDK (Software Development Kit) libraries.

A. *Backend*

The backend is, by far, the most challenging and intriguing part of this thesis. More specifically, it is the game-solving AI that is of interest, because it required numerous optimizations in order to accommodate it in a mobile environment.

A naive minimax implementation for quantum tic-tac-toe immediately reveals that the game is much more complex than classical tic-tac-toe, due to its exponentially larger search space. As a matter of fact, a simple upper bound for the game tree size of classical tic-tac-toe is 9!=362880, while the corresponding limit for quantum tic-tac-toe is $9!^2$ or approximately 131 billion! In such an implementation, traversing the first root subtree (out of 36) takes more than three hours on a modern desktop-class computer system. Given that the main objective of this thesis is real-time gameplay on mobile devices with a responsive interface, it is obvious that the game's algorithm has to be heavily optimized.

In order to analyze the performance of the core algorithms and determine any critical spots, the VisualVM profiling tool was put to use (see Fig. 10). VisualVM allows monitoring the runtime behavior of Java applications and is included in the official JDK (Java Development Kit).



Hot Spots - Method	Self Time [%]	Self Time	Self Time (CPU)	Total Time	Total Time (CPU)
qttt.State.**result** ()		10.549 ms (23,8%)	10.549 ms	13.530 ms	13.530 ms
qttt.State.**entangledEvaluation** ()		9.324 ms (21%)	9.324 ms	11.359 ms	11.359 ms
qttt.State.**collapse** ()		7.731 ms (17,4%)	7.731 ms	8.637 ms	8.637 ms
qttt.State.**lineWinner** ()		2.981 ms (6,7%)	2.981 ms	2.981 ms	2.981 ms
qttt.State.**cellPair** ()		2.941 ms (6,6%)	2.941 ms	2.941 ms	2.941 ms
qttt.State.**entangled** ()		2.342 ms (5,3%)	2.342 ms	13.702 ms	13.702 ms
qttt.State.**availableMoves** ()		2.161 ms (4,9%)	2.161 ms	18.494 ms	18.494 ms
qttt.State.**minValue** ()		1.975 ms (4,5%)	1.975 ms	44.374 ms	44.374 ms
qttt.State.**reorderMoves** ()		1.692 ms (3,8%)	1.692 ms	11.923 ms	11.923 ms
qttt.State.**maxValue** ()		1.234 ms (2,8%)	1.234 ms	44.374 ms	44.374 ms
qttt.State.**openCells** ()		938 ms (2,1%)	938 ms	938 ms	938 ms
qttt.State.**undoLastMove** ()		500 ms (1,1%)	500 ms	500 ms	500 ms
qttt.State.**gameOver** ()		0,000 ms (0%)	0,000 ms	13.530 ms	13.530 ms
qttt.State.**applyMove** ()		0,000 ms (0%)	0,000 ms	8.637 ms	8.637 ms
qttt.State.**resultEvaluation** ()		0,000 ms (0%)	0,000 ms	2.981 ms	2.981 ms
qttt.QTTT.**main** ()		0,000 ms (0%)	0,000 ms	44.374 ms	44.374 ms
qttt.State.**minimaxMoves** ()		0,000 ms (0%)	0,000 ms	44.374 ms	44.374 ms
qttt.State.**minimaxEvaluation** ()		0,000 ms (0%)	0,000 ms	44.374 ms	44.374 ms

Fig. 10. Screenshot from the VisualVM profiling tool. An analysis of the algorithm's execution on a modern desktop computer is shown, broken down in methods. The first three methods consume more than 15% of the total execution time each (which is indicated by the qttt.QTTT.main() method) and would be prime candidates for optimization.

1) *Alpha-Beta Pruning*

Alpha-beta pruning was the first, most simple and yet most important optimization that was applied. After incorporating it into the code base, the algorithm was able to traverse the entire game tree in approximately 15-20 minutes on a desktop computer. The performance is not ideal of course, although the performance gains (compared to a "bare" minimax implementation) are immense.

This implementation constitutes the basis on which all the other optimizations were applied. Additionally, it is noteworthy how such a small optimization (about 10-20 lines of code) led to such a disproportionate increase in performance.

2) *Move Ordering*

A naive (and trivial) implementation of the successor function in quantum tic-tac-toe would generate the successor states in alphabetical order. However, it turns out that changing the order in which the game tree nodes are generated, greatly affects the performance of the game-solving algorithm. This technique is called move ordering and is widely used in conjunction with alpha-beta pruning in order to accelerate the game tree traversal.

The principle behind move ordering is that examining the best successor state first leads to extensive pruning of the game tree by the alpha-beta algorithm. Of course it is not possible to choose the best successor state every time, otherwise the move ordering function could be used to play perfectly. Instead, the move ordering function attempts to reorder the moves according to certain criteria that are more likely to prioritize the best moves first.

Ishizeki and Matsuura [3] suggest that the following criteria be used:

- Moves that cause cyclic entanglement
- Moves that occupy the center cell of the board
- Moves that occupy a corner of the board

In addition, the following criterion was checked, although without successful results, as expected:

- Moves that occupy a side cell of the board (i.e. no corner)

The reasoning behind the criteria above is that cyclic entanglement causes at least two pairs of quantum marks to collapse, thus prohibiting further play in the corresponding cells, which, in turn, results in an exponential reduction of the search space. Moreover, the center cell and the corners of the board are of strategic importance, as in classical tic-tac-toe. On the other hand, the side cells of the board are insignificant and, in fact, have a negative impact on the algorithm's performance.

After thorough testing of all the possible permutations of the reordering criteria, they were made use of as listed above. In other words, moves that cause cyclic entanglement are examined first, followed by moves that occupy the center cell of the board and, finally, by moves that occupy a corner of the board.

The observed performance gains of this optimization were truly remarkable. Move ordering managed to reduce the total execution time of the algorithm from 987 seconds down to 40 seconds, a 96% reduction of execution time!

3) *Search vs. Lookup*

One last optimization was necessary in order to reduce the execution time from 40 seconds on a desktop computer down to 5-10 seconds on a mobile device. This performance jump was achieved by making use of *move lookup* instead of *move search*. Put differently, it is preferable to look up the optimal move in a precomputed database instead of searching for it in the game tree. For instance, there is no need to waste 40 seconds searching for the optimal move at the beginning of every game, since it is already well established by Ishizeki and Matsuura [3] that the first player must place his marks in opposite corners (i.e. 1-9 or 3-7) in order to secure a narrow win against an infallible opponent.

In an attempt to offer the best possible performance in the smallest possible size, the application contains a database of precomputed moves for the first five rounds of the game. In other words, during the first half of the game, no search is performed and, as a result, the optimal moves are retrieved almost instantaneously! After the fifth round, the algorithm reverts to searching the game tree, but at this point the game has progressed deep enough into the game tree so that the search cost is negligible. Even in the worst case scenario, where no marks have collapsed by the sixth round, the algorithm's execution time is usually less than a couple seconds, which guarantees that even the less capable mobile devices perform adequately.

It is worth mentioning that the relevant precomputed files consume approximately 90 Megabytes of storage space, but in the final installation file they are highly compressed, leading to a total package size of only 11 Megabytes. The format of these files is very simple so that it is human-readable, but it is also quite inefficient with regard to file size, lookup performance, etc. A self-explanatory example is shown below.

```
(0,1)(0,1)(1)(7,8):0,13
4,7
4,8
6,7
6,8

(0,1)(0,2)(0,1):-2,13
0
1
```

B. Frontend

Assuming a fully operating backend, the implementation of a frontend is, for the most part, trivial. As a matter of fact, both a textual and a graphical interface were implemented, although only the latter is accessible to the user in the app.

The textual interface (see Fig. 11) was created mainly for debugging purposes, but as a proof of concept, too. It is primitive in a number of ways, yet it allows the user to play either against the AI or another user (on the same device).

```
X1 O2 X3*|         |      X3
         |         |   X5*
         |X7       |
-----------------------------
         |   O2*   |
      O6 |O4       |O4*
X7       |         |
-----------------------------
         |         |X1*
      O6 |         |   X5
         |         |
```

Fig. 11. A textual representation of the game board. Quantum marks with an asterisk have collapsed in the corresponding cells.

The graphical interface, on the other hand, is much more complex (see Fig. 12). It is touch-enabled and uses various visual hints (such as colors and effects) in order to present the game state in a comprehensible and intuitive manner. This user interface will be detailed in the following section.

It is worth mentioning that the frontend allows multiple levels of difficulty for the AI player. This functionality is implemented within the frontend and is not available through the backend, which only supports perfect play. In order to achieve the "illusion" of varying levels of difficulty from a perfect opponent, a certain amount of randomness had to be introduced.

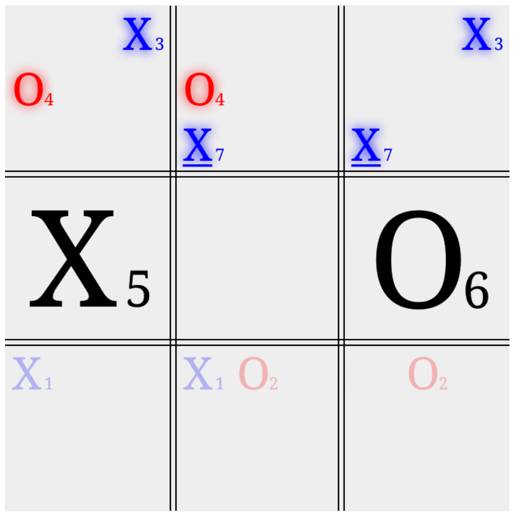


Fig. 12. A graphical representation of the board during the event of cyclic entanglement. The quantum marks X_3, O_4 and X_7 take part in the cyclic entanglement and are, therefore, "glowing". The quantum marks X_1 and O_2 do not participate in the cyclic entanglement and are "faded-out" instead. The quantum marks X_5 and O_6 have already collapsed and consequently occupy the whole cell as a visual aid. The user is hinted to choose between the pair of underlined quantum marks (X_7).

More specifically:

- In the "Random Moves" difficulty level, the AI always chooses a random move.
- In the "Easy" difficulty level, the AI chooses a random with a 50% probability.
- In the "Medium" difficulty level, the AI chooses a random move with a 25% probability.
- In the "Hard" difficulty level, the AI chooses a random move with a 10% probability. In addition, it always chooses the optimal move in the event of a cyclic entanglement.
- In the "Optimal" difficulty level, the AI always chooses an optimal move.

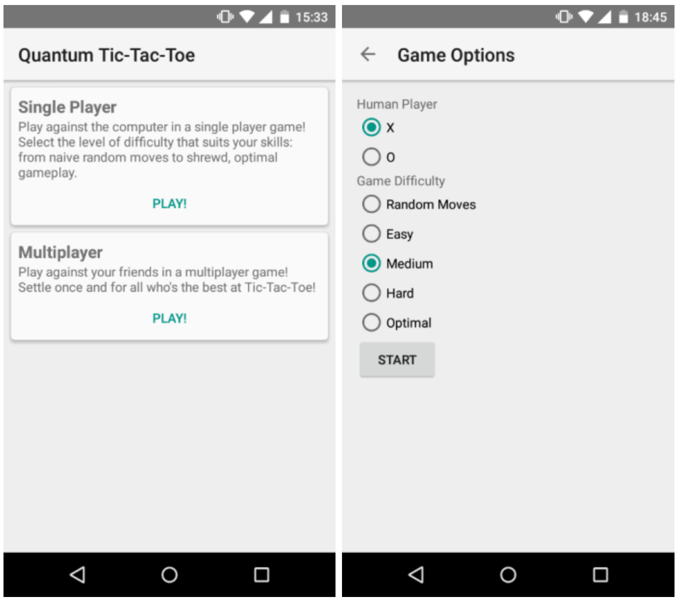
It goes without saying that the AI always chooses a valid move, which may turn out to be optimal, even when selected randomly. The following table sums up the above.

TABLE I. GAME DIFFICULTY LEVELS

Level	Random Move Probability	Optimal Move Probability
Random Moves	100%	>0%
Easy	50%	>50%
Medium	25%	>75%
Hard	10%	>90%
Optimal	0%	100%

IV. OPERATION

The operation of the app is quite straightforward. Initially, the user is greeted with a welcome screen that allows selecting the game mode: either single-player or multiplayer. The single-player game mode allows setting some options before the game begins, that is the user-controlled marks and the difficulty level of the AI (see Fig. 13). The multiplayer game mode requires no settings, therefore it begins immediately.


15:33

Quantum Tic-Tac-Toe

Single Player

Play against the computer in a single player game! Select the level of difficulty that suits your skills: from naive random moves to shrewd, optimal gameplay.

PLAY!

Multiplayer

Play against your friends in a multiplayer game! Settle once and for all who's the best at Tic-Tac-Toe!

PLAY!

18:45

Game Options

Human Player

[✓ X]

O

Game Difficulty

Random Moves

Easy

[✓ Medium]

Hard

Optimal

START

Fig. 13. The welcome screen of the app (left) and the game settings for the single-player mode (right).

Depending on the game mode and settings, the user is usually greeted with an empty game board (unless the user selected the “O” marks in a single-player game). By tapping on the cells, the user is able to interact with the game board (see Fig. 14). Extra effort was put in so that the interactions are as simple and intuitive as possible.

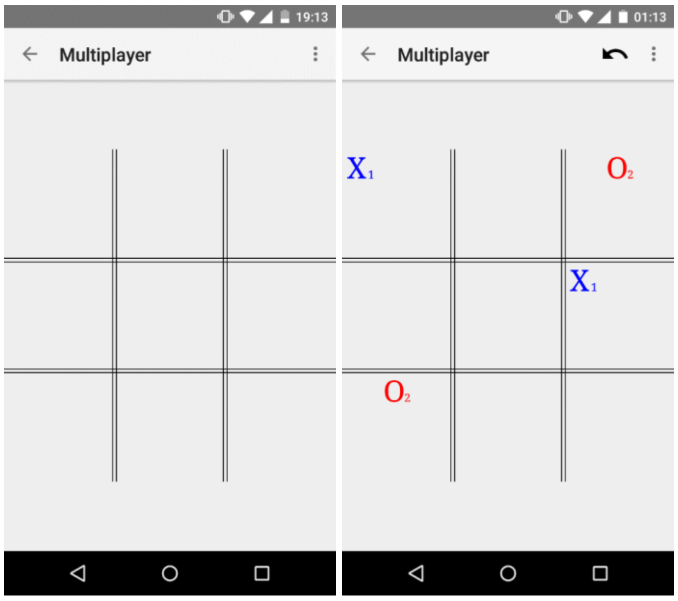


Fig. 14. Empty game board (left) and the third round of a multiplayer game (right).

While using the app, the user may, at any time, navigate to a previous screen by making use of the “Up” navigation arrow on the left of the toolbar. In addition, during a game the user may undo his (own) last move or reset the game board to the initial state (see Fig. 15, 16 and 17).

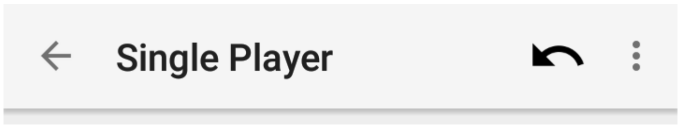


Fig. 15. The application’s toolbar during a single-player game. The “Up” navigation arrow can be found on the left, while the “Undo Move” action is located on the right. The three vertical dots is the overflow menu which contains hidden actions.

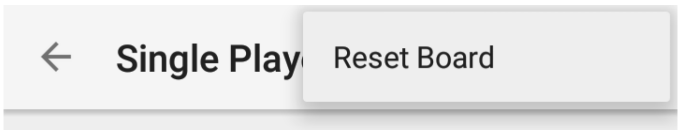


Fig. 16. The application’s overflow menu, when activated, presents a list of hidden actions (“Reset Board” in this case).

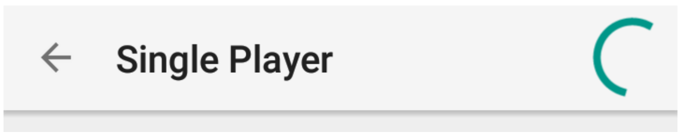


Fig. 17. Toolbar actions are not accessible while the AI is calculating the next move. In this case, the actions are temporarily replaced by a spinning circle progress bar.

The game board makes extensive use of various visual aids in order to make the game as user-friendly as possible. First of all, quantum marks are laid out on an imaginary grid and appear in a small font with varying color: the quantum marks of “X” are blue, while the quantum marks of “O” are red. Collapsed marks appear in black color and expand in order to “fill” the entire cell. However, tapping on any of them shrinks them back to normal size, in order to reveal the full history of moves (see Fig. 18).

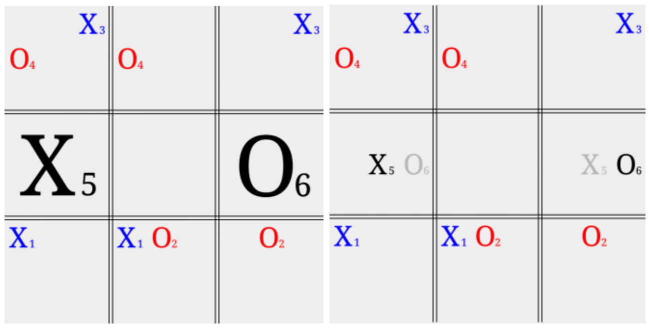
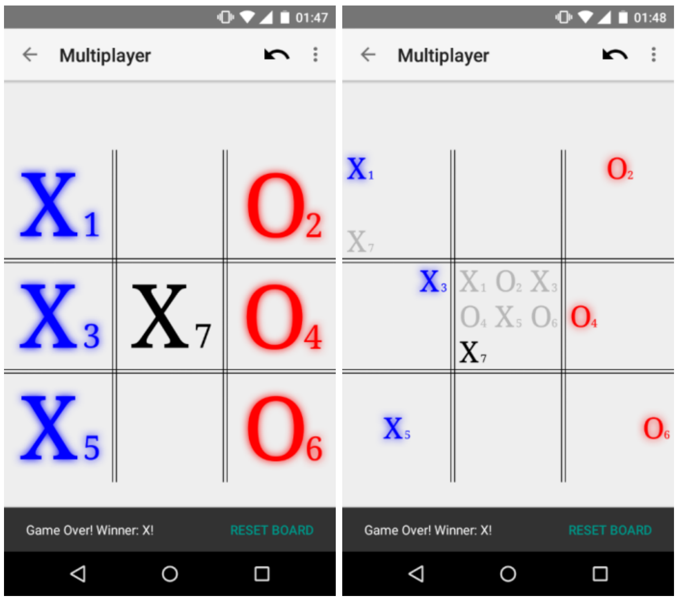


Fig. 18. Tapping on any of the collapsed marks toggles the “history” view of the game, i.e. all marks are shown, even those that have disappeared.

In the event of a cyclic entanglement, the quantum marks that participate start “glowing”, while the rest appear as “ghost” images, using a fade-out effect (see Fig. 12). This behavior allows the user to quickly determine which marks are going to collapse, while ignoring any symbols that are of no immediate interest.

Finally, when the game ends, a notification pops up on the bottom of the screen, informing the user about the game result and providing quick access to the “Reset Board” action. Every set of three marks in a row is highlighted with the corresponding player's color (see Fig. 19).


Game Over! Winner: X!

Game Over! Winner: X!

Fig. 19. The game has ended. Tapping on any of the collapsed marks toggles the "history" view, as usual.

V. Suggestions

Although the application met the requirements that were set forth by this thesis, there is always room for improvement. What follows is a list of steps that could be taken in the future in order to enhance the app.

A. *Move Suggestions*

An idea that would be trivial to implement is move suggestions, i.e. some kind of helper function accessible to the user that suggests moves, evaluates the current state of the game, etc. Given that all the relevant functionality is already available in the backend, the only aspect worth contemplating would be the game mechanics of such a tool, that is how it is presented to the user, which uses should be allowed, what consequences (if any) it would incur, etc.

B. *Internet Multiplayer*

An interesting addition to the app would be multiplayer matchmaking through the Internet. Implementation-wise this suggestion would not require a lot of modifications on the side of the client (the app), but it involves coding a server-side component, as well as buying, setting up and maintaining dedicated servers. Alternatively, in a decentralized, peer-to-peer architecture, the app could be able to connect directly to an IP (Internet Protocol) address, or scan for nearby players on the same (local) network.

C. *Multithreading*

A great way to make the game-solving algorithm more efficient would be to utilize the multiple processing cores that are available in modern mobile devices. After all, it is not uncommon for modern smartphones and tablets to have 4, 8 or even more CPU (Central Processing Unit) cores. Moreover, the minimax algorithm can be easily parallelized, since the traversal of every subtree can be performed independently on a separate thread. Given that the app's responsiveness was found to be up to par, this suggestion was not implemented, although it would, undeniably, lead to major performance gains.

D. *Move Database*

Finally, the current format for the precomputed "opening" files is suboptimal in regard to both the file size and lookup performance. Although the observed performance is adequate, it would be highly advisable that a more efficient implementation were used. To this end, an indexing scheme for opening moves could be applied, in order to speed up the lookup procedure, along with a more compact file format, in order to save up on storage space. Alternatively, an SQL database system could be used, given that the Android development platform includes excellent support for SQLite databases by default.

VI. Conclusions

In the previous sections, a theoretical, technical and operational analysis of the application was laid out. Taking everything into consideration, the end result is considered satisfactory, having fulfilled this thesis' primary objective: a simple, user-friendly Android application for the game of quantum tic-tac-toe. The application's interface is deliberately minimalistic, yet informative and responsive. Regarding functionality, the most common use case scenarios were implemented: a single player and a multiplayer mode, along with an extra feature that allows setting the difficulty level of the AI.

References

[1] A. Goff, "Quantum tic-tac-toe: A teaching metaphor for superposition in quantum mechanics," American Journal of Physics, vol. 74, no. 11, pp. 962–973, 2006.

[2] "Quantum tic-tac-toe," Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Quantum_tic-tac-toe. [Accessed: Jan-2016].

[3] T. Ishizeki and A. Matsuura, "Solving Quantum Tic-Tac-Toe," in Proceedings of the International Conference on Advanced Computing & Communication Technologies, 2011, pp. 330–334.

[4] S. J. Russell and P. Norvig, Artificial Intelligence: A Modern Approach, 3rd ed. Upper Saddle River, NJ: Prentice Hall, 2009.

[5] "Java (programming language)," Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Java_(programming_language). [Accessed: Jan-2016].

[6] "Android (operating system)," Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/Android_(operating_system). [Accessed: Jan-2016].