Approximating Optimal Strategy in Match-Three Puzzle Games

Gregory McAdams
Computer Science and Electrical Engineering
University of Maryland Baltimore County
Baltimore, USA
mcadams1@umbc.edu

***Abstract*—Match-three puzzles are a class of games that are interesting not only due to their inherently random nature, but also because they are computationally intractable, even in the simplest case. Previous attempts have been made to create a solver that provides a best-guess to these games, but most are either too simplistic or provide no empirical evidence showing any success. A tree-based algorithm is presented, along with several heuristic functions, in order to provide a strategy that capitalizes on all of the limited state information given to the player. Furthermore, a lookahead can be configured in order to weight an immediate value against a possible greater potential value in a future state. An implementation of this algorithm was created, and it has shown remarkable superiority over simply random guessing. Specifically, a two-tailed t-test was used to show that the new algorithm achieves a higher average score with 99.9% confidence when varying only game state size. A comparison was also done between two implemented heuristic functions used with the new algorithm; the first heuristic is simplistic and only considers score, whereas the second heuristic considers not only score, but also the game state itself. Two major trends were observed among all trials. First, as the game state size increased, the more of a performance gap there was between the new algorithm and random guessing, and between the first and second heuristic. Second, as the maximum depth limit was increased when building the tree, the less of a performance gap there was the first and second heuristic. In other words, after a certain depth is reached, there is no benefit in doing further lookahead; this limit was found to be 3. Lastly, the new algorithm was modified to break ties between moves based on the average row the move affected. It is shown that moves further down tend to make the resulting game state more volatile, and thus result in slightly higher average scores over time.**

I. Introduction

Match-three games are a class of puzzle games that have existed since the 1980s[1], but have exploded in popularity recently through games such as Bejeweled and Candy Crush. In fact, Candy Crush became the most popular game on Facebook in 2013[2], and Bejeweled has been downloaded around 150 million units since it was released in 2001[5]. A major reason for success of such games can be attributed to their casual, mobile, and free-to-play nature. A more interesting reason is that these games are actually quite challenging for the player.

Even in the simplest form of gameplay, where the player is only concerned with matching three, these games have proven to be NP-hard[3][4]. What makes many match-three games so challenging is that much of the state of the game at each move is generated at random, and the player must create patterns in this sea of randomness in order to score. There are many variations of match-three games with varying rules and gameplay mechanics, but the focus in this paper is primarily the variation represented by Candy Crush.

This paper assumes the reader knows the basic rules, mechanics, and functionality of games such as Candy Crush and Bejeweled. These games are constantly being updated, with new levels and new mechanics of increasing complexity added quite frequently. However, this paper looks at the basic three, four, and five matching mechanics of the game, where the goal is to maximize or meet a score threshold after a number of valid moves.

Although there are hint features within these games, they are very basic and nearly always give an obviously sub-optimal move. Because of this, and due to the popularity of the game and its challenging nature, there have been several attempts at creating a program to find the most optimal move. These programs – deemed “solvers” – currently reside in the app store for both Android and iOS devices. Unfortunately, many of these programs only look in the current state of the game to determine the “best” move. Furthermore, many do not attempt to look ahead beyond a single move to estimate how the chosen move might later affect the game. In other words, many of these programs may be only marginally better than the in-game hints, and at worst a complete waste of money.

An author of one of these apps confided his strategy, and he revealed that a more complex algorithm was indeed used. Coincidentally, his strategy is very similar to the strategy formulated in this paper, but with a slightly more simplistic heuristic function. None of these solvers have any empirical data illustrating that they are actually worth the money they charge. A potential customer is limited to blind faith that the solver will actually give them a higher score than if they had a monkey play. Thus, a major goal of this paper is to show that, on average, such a solver is measurably better than a player who chooses moves randomly.

In this research study, three main objectives were met. First, a text-based match-three puzzle game, similar to a basic form of Candy Crush, was created in Python in order to automate a large number of games being run simultaneously.

This game was designed to be highly configurable such that any size game board can be used along with any number of moves, etc. to study the effects that these variables have among different strategies. Second, a new algorithm is proposed that attempts to play each game in the most optimal way possible. Third, this paper empirically proves that this new algorithm is indeed better than a player that selects every move purely at random.

II. Match-three Puzzle Simulator

Most existing solvers use image processing to determine the current game state, as there is no API provided by Candy Crush or other such games. Because of this, and other factors, playing just a single game takes quite some time. Furthermore, every game is completely random from the player's perspective, as there is no control over the pseudo-random number generator, and each game level has its own, unchangeable, parameters. To overcome these limitations and to obtain meaningful results in a reasonable amount of time, a simulator program was created.

A. *Implemented Features*

The simulator looks for a match of three, four, or five to determine a valid move, much like Candy Crush/Bejeweled. Upon detecting a match, the pieces located in those positions are deleted, and every other piece is moved down accordingly. The simulator checks for a valid move starting from the top-left corner, going top to bottom, left to right. This contrasts with Candy Crush, as in that game this process occurs simultaneously. Upon matching four, a single special piece replaces the left-most piece. This special piece mimics the mechanics of the “striped candy” in Candy Crush, with its direction also determined in the same way. Upon matching five, a piece that mimics the mechanics of the “color bomb” is generated in the same way. Note that the “wrapped candy” special piece is absent for the time being. As in Candy Crush, these special pieces can be combined with one another for unique mechanics. The following combinations of special candy are implemented, and mimic the mechanics of Candy Crush: “striped candy” and “striped candy”, “striped candy” and “color bomb”, and “color bomb” and “color bomb.”

Like in Candy Crush/Bejeweled, any move can trigger a chain of moves, creation of special pieces, or any combination of those until there are no more three or more matching pieces present. Also, if no moves are currently available, all pieces in the game will be randomly shuffled until there is at least one valid move. A script was also written to drive and run the simulator program as many times as needed, for any player type desired. Furthermore, this script keeps track of the following statistics for all combined games: average score, median score, standard deviation of score, average moves, and average run-time until completion. It also provides a running average of scores, so one does not have to wait until all games are played to determine the real-time average.

Scoring is quite a bit different in the simulator than in Candy Crush, mainly for simplicity. There is no official documentation on how exactly these games score certain moves or combination of moves in every situation. To simplify scoring, the simulator simply gives 1 point for each piece that is matched or otherwise removed. This does not take the desirability away from special pieces, but they do give relatively less score themselves when removed. There are no multipliers or any such mechanic implemented when a long chain of pieces are removed originating from a single move.

B. *Configurable Parameters of Simulator*

- $M(rows) \times N(columns)$ size of the game board
- Number/value of “colors”
- Max number of moves allowed
- Goal score to be achieved
- Pseudo-random number generator seed
- Player “type” and heuristic function (if applicable)

III. A New Algorithm

A new algorithm was developed where, for each move, a new tree is constructed in a depth-first manner, with the current game state as the root. A child node is generated at each level for each possible legal move in the current game state. For each child, we copy the parent's game state and make the next move. Note that once a move is made in a child game state, it may trigger any number of cascading moves such that the child's game state afterwards may be very different from its parent. The number of nodes at level i, on average, is the average number of moves raised to i.

$$Avg.\#\,Nodes, level\ i\ =\ (Avg.\#\,Moves)^{i} \qquad (1)$$

A. *Reducing Time Complexity*

For a game state with many possible moves, it becomes unfeasible to build the full tree up to an arbitrary level, as the number of nodes at subsequent levels grows exponentially. Due to this, a maximum depth limit is set beforehand. Additionally, a variable beam width was added to the algorithm, with a decay rate equal to the floor of the square root function at each level. As with maximum depth, this parameter is set beforehand. For example, if the beam width at the first level is 9, at the second level it is 3.

$$Beam\ Width, level\ i = \sqrt{Beam\ Width, level\ i-1} \qquad (2)$$

In other words, the 9 best children are kept at the first level and the best 3 children at kept at the second level. The beam width at the first level depends on the maximum depth, and is equal to the maximum depth squared.

$$Beam\ Width, level\ 1 = (Max\ Depth)^{2} \qquad (3)$$

The quantitative measurement for the quality of each child is given by a heuristic function, discussed in the next section. If there are a multitude of equally high quality children to choose from, the tie-breaker is whichever move was computed last, which is the move furthest down since moves are done top-down. This optimization was suggested by a number of avid Candy Crush players, and results have shown this to be a promising optimization. Furthermore, when each level is generated, the nodes are first sorted by average depth, and then sorted by heuristic value. Average depth is the average row between all pieces to be removed. Note that if the move is horizontal, then all rows are equal.

B. *Optimizations*

Since a new tree is generated after each move, there is a possibility of recomputing the children for a particular game state. Thus, an optimization was made such that all game states, along with their children, are saved in a Python dictionary data structure. When a game state's hash value matches a key in the dictionary, all of its children are reused for that node and added as its own children. The hash value for a game state is simply a string representing the game's current state in a compressed form, meaning that all whitespace is removed. Here is an example of a game state:

R	*G*	*B*	*P*	*O*
B	*O*	*G*	*P*	*B*
O	*R*	*G*	*B*	*P*
O	*P*	*B*	*G*	*R*
R	*G*	*B*	*O*	*O*

Fig. 3.1. A Game State

The hash value would then be *RGBPOBO...* in Fig. 3.1. Of course, as the number of possible states increases, this optimization becomes less useful, and quickly to the point where it does not help at all.

The tree is not built beyond the number of moves allowed. If any level of the tree determines that the current move is equal to the max moves allowed, it does not build beyond that level. This is important not only from an efficiency standpoint, but it dictates that the choice for the last several moves is based on actual score earning potential. For the last move, only the score for the immediate move matters, and so the last move is always greedy regardless of depth limit.

A performance optimization was made based on the determination of an adjacent piece. Only the pieces to the immediate right and below any given piece need to be considered. Thus, checking pieces to the left and above are redundant in this case. This is because adjacent pieces are commutative with one another. Note that if traversing the game state bottom to top, right to left, the adjacent pieces would instead be to the left and above a given piece. The same optimization was made to the heuristic function H2 introduced in the next section.

C. *Pseudocode*

```
Algorithm 1 A New Algorithm
procedure NEXT MOVE($CurrentState, MaxMoves, MaxDepth$)
    $root \leftarrow$ BUILD TREE(NODE($CurrentState$), $MaxMoves, MaxDepth$)
    $NextMove \leftarrow root.children[0]$
    for all $Nodes$ in $root.children$ do                          ▷ Find best next move
        if $Node.value >= NextMove.value$ then
            $NextMove \leftarrow Node$
    return $NextMove.obj.move$                                     ▷ Return best next move

procedure BUILD TREE($Node, MaxMoves, MaxDepth$)
    $children \leftarrow Empty\ List$
    if $Node.obj.movecounter = MaxMoves$ then                      ▷ Reached Max Moves
        return
    for all $valid\ moves$ in $Node.obj$ do
        $NewChild \leftarrow$ COPY($Node$)                          ▷ Clone Node
        do $valid\ move$ in $NewChild.obj$                          ▷ Perform next move
        $NewChild.value \leftarrow$ HEURISTIC($NewChild.obj$)
        $NewChild.parent \leftarrow Node$
        $children \leftarrow$ APPEND($NewChild$)
        if $MaxDepth > 1$ then                                     ▷ Keep building tree depth-first
            return BUILD TREE($NewChild, MaxMoves, MaxDepth - 1$)
    if LENGTH($children$) $= 0$ then                               ▷ If no valid moves
        return
    sort $children$ by $Average\ Depth$ of $child.obj.move$        ▷ Optimization
    sort $children$ by $child.value$                               ▷ Sort children by score
    $BeamWidth \leftarrow MaxDepth^2$                              ▷ Keep only the top children
    remove $children[BeamWidth]$ to $children[$LENGTH$(children) - 1]$
    $Node.children \leftarrow children$
    if $Node.parent = NULL$ then                                   ▷ If root node
        return $Node$
    $Node.value \leftarrow$ FIND AVERAGE VALUE($node$)              ▷ Node's final value
    return

procedure FIND AVERAGE VALUE($Node$)
    if LENGTH($children$) $= 0$ then                               ▷ No children
        return $Node.value$                                        ▷ Node's value is only its own
    $ChildValues \leftarrow 0$
    for all $children$ in $Node.children$ do
        $ChildValues \overset{+}{=} (child.value)/$LENGTH$(Node.children)$
        return $(node.value + ChildValues)/2$                      ▷ Avg. of Node + children
```

Fig. 3.2. Deciding the Next Move

IV. HEURISTIC FUNCTIONS

Two heuristics, used as a way to rank different game states, were employed by the new algorithm.

A. *Heuristic H1*

The first heuristic is a simple heuristic that only takes into account the current score of the game state and the number of moves taken up to that point. This heuristic serves as the baseline to compare the more advanced heuristic, H2, described hereafter. The heuristic value given by H1 is simply the current score divided by the total number of moves.

$$H1 = \frac{Score}{Moves} \tag{4}$$

B. *Heuristic H2*

The second heuristic is a more complex heuristic that takes into account the current score as well as the current configuration of the game state. Essentially, for each piece in the game state, the heuristic assigns points to it depending on its scoring potential. The following in the game state are used in the configuration of the heuristic value:

- The number of pieces that are adjacent to another piece of the same color
- All "striped candy" pieces
- All "color bomb" pieces

Each piece that is not a special piece is assigned a value of 1 only if there is an adjacent piece with an identical color. Note that this can be either a "striped candy" or another non-special piece. If it is not adjacent to a piece of an identical color, it has no value assigned to it. The reasoning behind this is that if there are already two pieces of a particular color adjacent to one another, there is a chance of them contributing to the score in the next several moves since only one other color is needed to match three. However, if alone, there is a much smaller chance, and in the heuristic this is determined to be negligible.

For each "striped candy" piece, if it is adjacent to a non-special piece with the same color, its value is $\frac{N}{3}$ if its direction is horizontal and $\frac{M}{3}$ if its direction is vertical. Like with normal pieces, this value is assigned because there is a chance that this will contribute to the score in the next several moves. The reasoning behind the given value is that the piece is assigned one point for every possible 3 pieces in either a single row or column. Likewise, if a striped candy is not adjacent to any pieces with the same color, it is not given a value at all. If the "striped candy" piece is adjacent to another "striped candy" piece, these pieces can of course be combined, and a guaranteed minimum immediate value is added to the heuristic value equal to $M + N$. If a "striped candy" piece is adjacent to a "color bomb" piece, the value is approximated by assigning a random direction to each piece with the same color since they will all be transformed into "striped candy." Each of these new "striped candy" pieces are assigned a value of either N or M depending on their assigned direction, as they are guaranteed to all be immediately activated by the combo.

For each "color bomb," every adjacent piece is considered, and the value of the color bomb is simply the maximum of all possibilities. For each adjacent normal piece, the number of those colored pieces on the board is counted; each normal piece is given a value of 1, and each striped piece is given a value of either M or N depending on its direction. If there is an adjacent "striped candy" piece, the method just described in the previous paragraph for this combination applies identically. If another color bomb piece is adjacent, then the value is $M \times N$, since the entire board will be cleared from this combination.

The final heuristic value given is the sum of the values just described. This value is stored at the node itself when child nodes are created. The intention of this was to weight the value of each node based on the level at which the node resides. However, different weighting was experimented with in this regard, but just taking the average value of all nodes in a particular path from root to leaf came to have the best results. The average heuristic values determined for each path is done in a bottom-up manner, starting with the leaf nodes and percolating up. After building the tree, the first level of child nodes all have the average score for their respective paths, and the child with the highest average score is chosen as the next move.

More complex aspects were experimented with in this heuristic, but the relatively more simple method just described gave the best results. For example, the number of pieces that differed between the child state and parent state were counted, as a way to compare the percentage chance that one state could become another after a single move. This value would be subtracted from the difference in score from the move itself, which is a guaranteed change of state not subject to random chance after the move. We then use this value as the power raised to the number of unique colors, and divide 1 by it to get the final probability in decimal form. Finally, the heuristic value would be multiplied by this probability to weight the heuristic value based on how likely the state would come to be. Again, doing this process actually resulted in worse results, and so was removed.

Algorithm 2 Heuristic H2

```
procedure HEURISTIC H2($CurrentState, NewState$)
    $Value \leftarrow NewState.score - CurrentState.score$          ▷ Add score diff.
    $ComparedList \leftarrow An\ Empty\ List$          ▷ List of prev. compared pieces
    for each $piece\ p_1$ in $NewState$ do          ▷ Examine each piece in state
        for each $adjacent\ piece\ p_2$ do          ▷ Either to the right or below
            if $(p_1, p_2)$ or $(p_2, p_1)$ in $ComparedList$ then
                continue          ▷ Avoid double counting
            append $(p_1, p_2)$ to $ComparedList$
            if $p_1$ is a $Color\ Bomb$ then
                $Value \mathrel{+}=$ COLOR BOMB VALUE($NewState, p_2$)
            else if $p_1$ is a $Striped\ Candy$ then
                $Value \mathrel{+}=$ STRIPED VALUE($NewState, p_1, p_2$)
            else if $p_1.color = p_2.color$ then          ▷ Normal + Normal
                $Value \mathrel{+}= 1$
    return $Value$          ▷ Return final value

procedure COLOR BOMB VALUE($NewState, c_2$)
    if $c_2$ is a $Color\ Bomb$ then          ▷ Color Bomb + Color Bomb combo
        return $NewState.M \times NewState.N$          ▷ Rows × columns
    if $c_2$ is a $Striped\ Candy$ then          ▷ Color Bomb + Striped combo
        $Value \leftarrow 0$
        for each $piece\ p_1$ in $NewState, p_1 \neq c_2$ do
            if $p_1.color = c_2.color$ then          ▷ If colors match
                $Randomly\ Pick\ 'H'$ or $'V'$          ▷ Pick a direction
                if $'V'$ then
                    $Value \mathrel{+}= NewState.M$          ▷ Add the # of rows
                else
                    $Value \mathrel{+}= NewState.N$          ▷ Add the # of columns
        return $Value$
    $Value \leftarrow 0$          ▷ Color Bomb + Normal
    for each $piece\ p_1$ in $NewState, p_1 \neq c_2$ do
        if $p_1.color = c_2.color$ then
            $Value \mathrel{+}= 1$
    return $Value$

procedure STRIPED VALUE($NewState, c_1, c_2$)
    if $c_2$ is a $Color\ Bomb$ then          ▷ Striped + Color Bomb combo
        return COLOR BOMB VALUE($NewState, c_1$)
    if $c_2$ is a $Striped\ Candy$ then          ▷ Striped + Striped combo
        return $NewState.M + NewState.N$          ▷ Rows + columns
    if $c_1.color = c_2.color$ then          ▷ Striped + Normal
        if $c_1.dir = 'V'$ then          ▷ If $c_1$ has a vertical direction
            return $M/3$
        else
            return $N/3$
    return 0          ▷ Colors do not match
```

Fig. 4.1. Pseudocode for Heuristic H2

V. Experiments

There are many parameters that can be changed and/or varied to obtain meaningful results. In all trials, the focus is on keeping the following parameters static: 6 unique colors, 20 valid moves, no score ceiling. In other words, the goal is to achieve the maximum possible score in 20 moves with the standard 6 different colors. The following parameters are independent: board size, player type, and heuristic function for the new algorithm. There is neither a time limit set for each move nor for an entire game. The seed value used in each game is equal to the game's number in the series. For example, for the first game in the series, a seed value of 1 is used. The experiments were run on a PC with an Intel Core i5-3570K 3.6-GHz processor and 8 GB RAM, equipped with Linux Fedora 20.

Each experiment was run on 3 different players. The first is the "random" player, which chooses a valid move completely at random. The second and third players are "smart" players that implement the new algorithm. The second player uses the simple heuristic function – H1 – and the third player uses the more complex heuristic – H2. The simplest smart player is the one that uses the simple heuristic, and only builds its tree to a single level (i.e. greedy). If we can conclude that the simplest smart player achieves an empirically higher score than the random player, then every other instance of the smart player will be at least as good. Experiments were also produced for a comparison between the two heuristic functions employed by the smart player.

We compare two players at a time, and determine which player in each pair of trials achieves a higher average score using a two-tailed test. We use a two-tailed test because we do not know a priori which player will obtain a higher average score, so we need to test in both directions. A t-test is a valid statistical significance test here since each trial consists of independent and identically distributed (i.i.d.) random numbers generated from the Python built-in library random[1] such that, by the central limit theorem, all averages are normally distributed. Since we are comparing two sample means (x_1 and x_2) and their standard deviations (s_1 and s_2), we find the t-value according to the following equation:

$$t = \frac{x_1 - x_2}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}} \tag{5}$$

Games are played between the random and smart player until there is ~1000 degrees of freedom, given by:

$$d.o.f. = \frac{\left(\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}\right)^2}{\frac{1}{n_1 - 1}\left(\frac{s_1^2}{n_1}\right)^2 + \frac{1}{n_2 - 1}\left(\frac{s_2^2}{n_2}\right)^2} \tag{6}$$

The number of trials is rounded to 500 for both n_1 and n_2 across all testing. The null hypothesis we are trying to refute is H_0: $P_1 = P_2$, and our alternate hypothesis then is H_1: $P_1 != P_2$ for the average score of competing players P_1 and P_2.

[1] This module computes a uniform random floating-point number between [0.0, 1.0) using the Mersenne Twister PRNG.

In the first graph of each section that follows, the x-axis is labeled according to:

$$Player.Heuristic.Max\ Depth \tag{7}$$

For example, for the smart player using heuristic H1 with a max depth of 2, the label is *S.H1.2*. The random player's label is simply *R*.

A. *Game Size of 5x5*

The average number of moves for an arbitrary game state with this configuration was found to be 3.28. Therefore, we should expect that having a beam width greater than 4 at the first level will result in a statistically insignificant difference in the average score. However, a trial was still run with a max depth of 3 to confirm our claim, but in future sections these will be omitted. With a t-value of 14.59 between the random player and the simplest smart player, we are well over 99.9% confident that the latter results in a higher average score.

Between the two heuristics used for the smart player, we are less than 90% confident that heuristic H2 is better for the greedy, max depth of 1 case. For the max depth of 2 case, we are less than 90% confident that heuristic H1 is better. For the max depth of 3 case, we are over 98% confident that heuristic H1 is better.

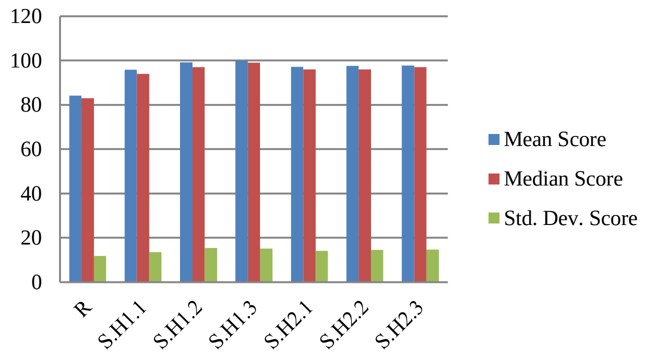

Fig. 5.1. Statistics for 5x5

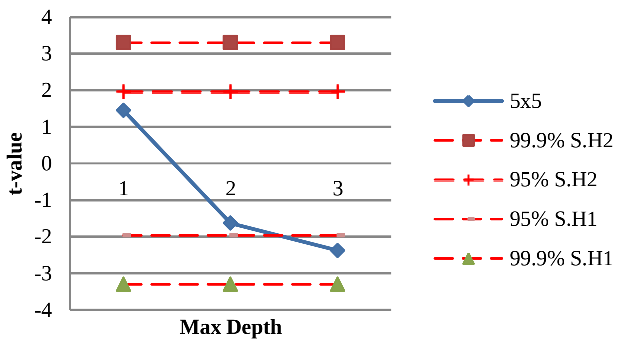

Fig. 5.2. S.H1 vs. S.H2 for 5x5

B. *6x6*

The average number of moves for an arbitrary game state with this configuration was found to be 4.99, and thus there are no trials included of max depth greater than 3. With a t-value of 20.623 between the random player and the simplest smart player, we are well over 99.9% confident that the latter results in a higher average score.

Between the two heuristics used for the smart player, we are over 99.9% confident that heuristic H2 is better in the case when the max depth is 1. For the max depth of 2 case, we are over 98% confident that heuristic H1 is better. For the max depth of 3 case, we are less than 50% confident of either heuristic resulting in a higher score, and thus the null hypothesis holds.

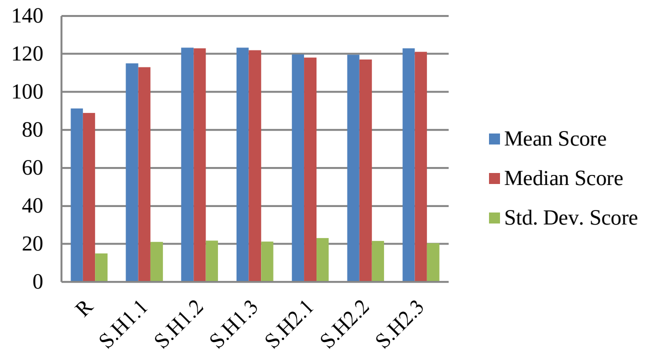

Fig. 5.3. Statistics for 6x6

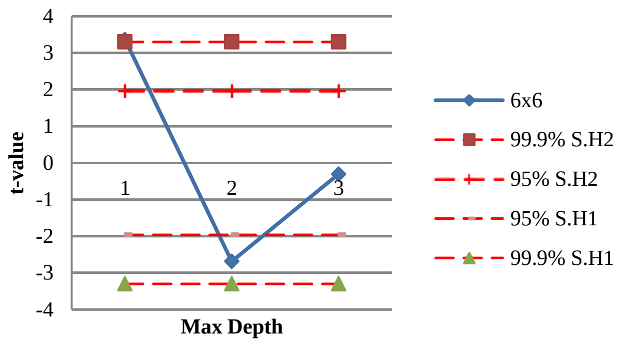

Fig. 5.4. S.H1 vs. S.H2 for 6x6

C. 7x7

The average number of moves for an arbitrary game state with this configuration was found to be 7.72, and thus there are no trials included of max depth greater than 3. With a t-value of 26.131 between the random player and the simplest smart player, we are well over 99.9% confident that the latter results in a higher average score.

Between the two heuristics used for the smart player, we are over 99.9% confident that heuristic H2 is better in the case when the max depth is 1. For the max depth of 2 case, we are less than 50% confident of either heuristic resulting in a higher score, and thus the null hypothesis holds. This is also the result for the max depth of 3 case.

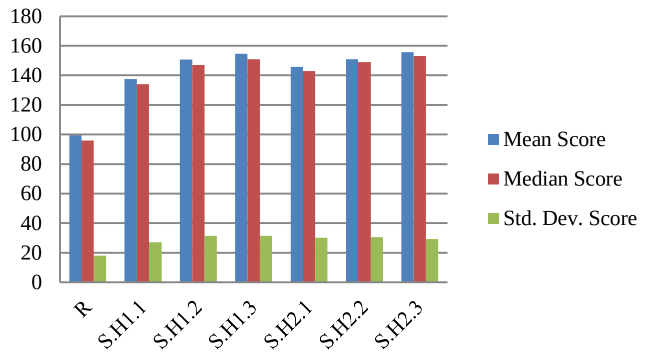

Fig. 5.5. Statistics for 7x7

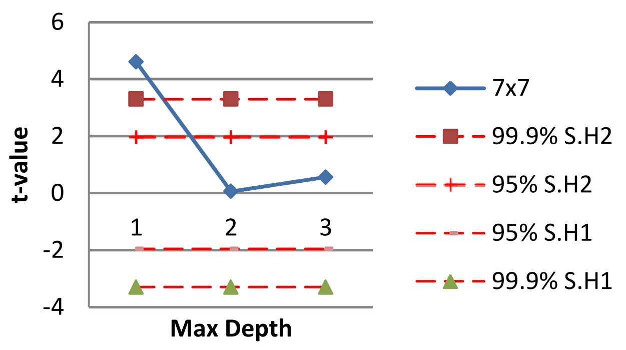

Fig. 5.6. S.H1 vs. S.H2 for 7x7

D. 8x8

The average number of moves for an arbitrary game state with this configuration was found to be 10.59. With a t-value of 28.634 between the random player and the simplest smart player, we are well over 99.9% confident that the latter results in a higher average score.

Between the two heuristics used for the smart player, we are well over 99.9% confident that heuristic H2 is better in the case when the max depth is 1. For the max depth of 2 case, we are less than 50% confident of either heuristic resulting in a higher score, and thus the null hypothesis holds. This is also the result for the max depth of 3 case. For the max depth of 4 case, we are less than 80% confident that heuristic H2 is better.

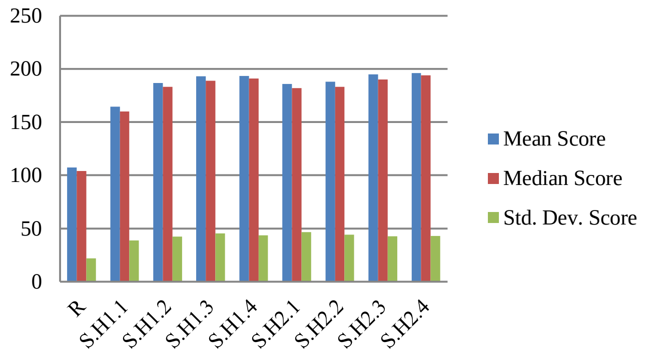

Fig. 5.7. Statistics for 8x8

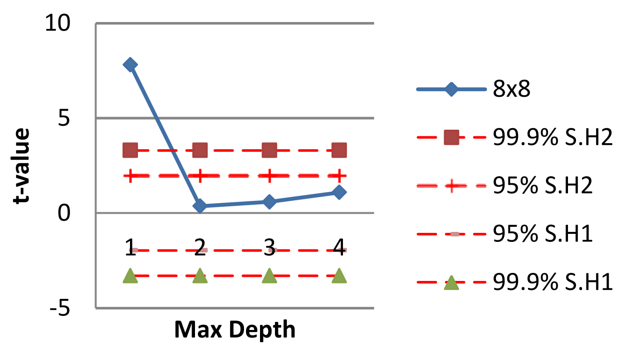

Fig. 5.8. S.H1 vs. S.H2 for 8x8

E. *9x9*

The average number of moves for an arbitrary game state with this configuration was found to be 14.31. With a t-value of 32.540 between the random player and the simplest smart player, we are well over 99.9% confident that the latter results in a higher average score.

Between the two heuristics used for the smart player, we are well over 99.9% confident that heuristic H2 is better in the case when the max depth is 1. For the max depth of 2 case, we are also well over 98% confident that heuristic H2 is better. For the max depth of 3 case, we are less than 95% confident that heuristic H2 is better. For the max depth of 4 case, we are over 99.9% confident that heuristic H2 is better.

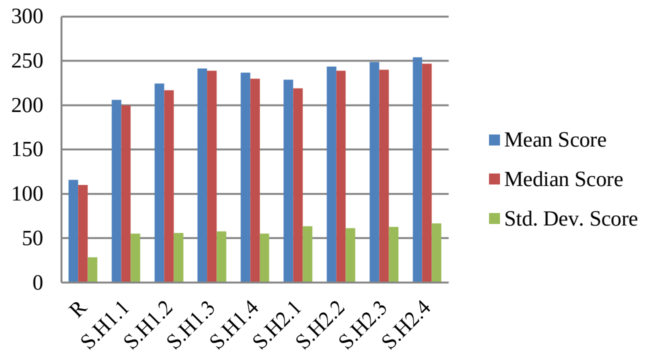

Fig. 5.9. Statistics for 9x9

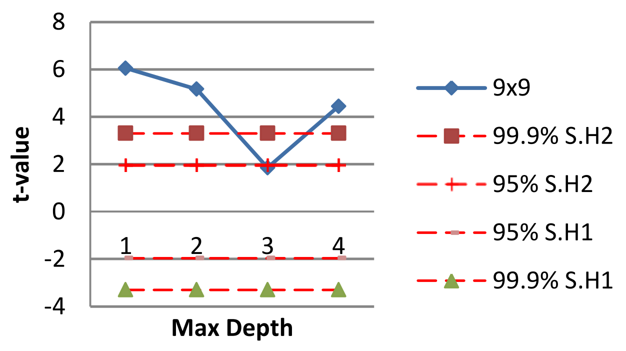

Fig. 5.10. S.H1 vs. S.H2 for 9x9

F. *10x10*

The average number of moves for an arbitrary game state with this configuration was found to be 18.96. With a t-value of 36.756 between the random player and the simplest smart player, we are well over 99.9% confident that the latter results in a higher average score.

Between the two heuristics used for the smart player, we are well over 99.9% confident that heuristic H2 is better in the case when the max depth is 1. For the max depth of 2, 3, and 4 cases, we are over 99.9% confident that heuristic H2 is better.

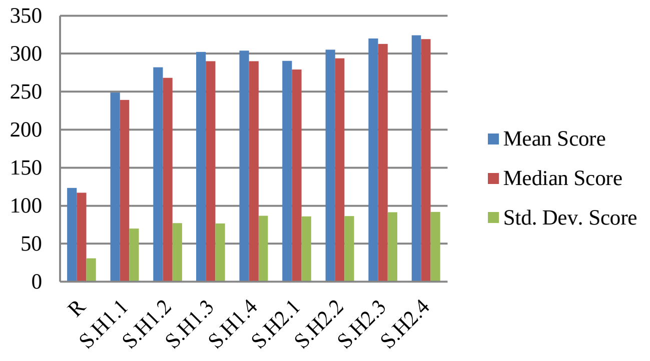

Fig. 5.11. Statistics for 10x10

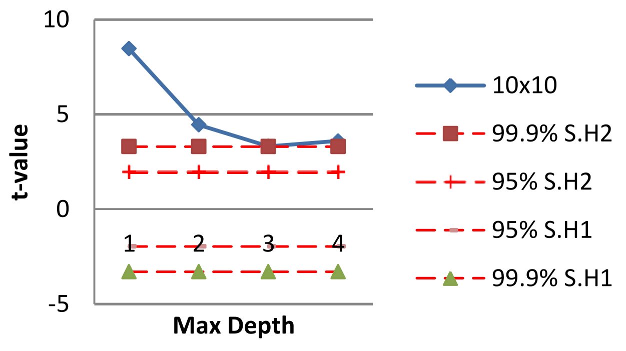

Fig. 5.12. S.H1 vs. S.H2 for 10x10

G. *Optimization – Sorting by Average Depth*

In this special case experiment, we show that an optimization that was added to the algorithm provides a statistically significant benefit. This optimization involves first sorting possible moves by the average row in which the move would have an effect (i.e. average depth). To demonstrate this is a valid optimization, an experiment was created with a game of size $10x5$. We compare the smart player, which first sorts its possible moves based on average depth (S), to the smart player which does not (NS). Additionally, the player which first sorts by average depth always chooses the bottom-most move when there is a tie, where the other player always chooses the top-most move. Both smart players use heuristic H1 in this experiment, and both also sort all moves at each level based on their heuristic values as normal.

The average number of moves for an arbitrary game state with this configuration was found to be 7.284, and thus there are no trials included of max depth greater than 3. With a t-value of 3.439 between the depth-sorted player and the non-depth-sorted player, we are over 99.9% confident that the former results in a higher average score when the max depth is 1. For the max depth of 2 case, we are less than 98% confident that the depth-sorted player results a higher average score. For the max depth of 3 case, we are less than 50% confident of either player showing a higher score, and thus the null hypothesis holds.

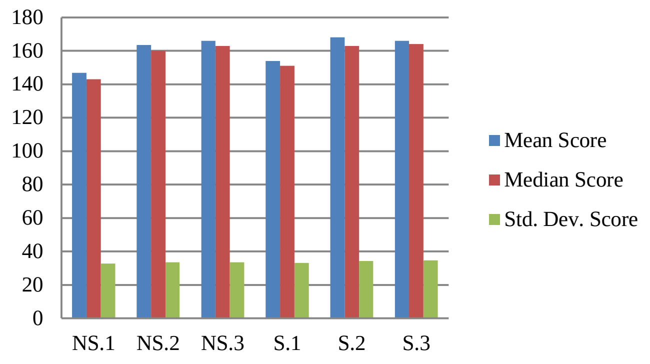

Fig. 5.13. Statistics for 10x5

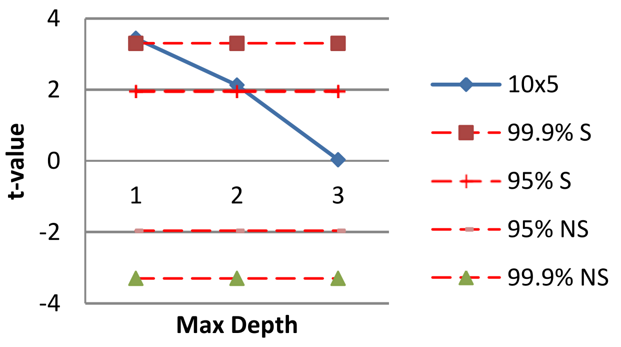

Fig. 5.14. S vs. NS for 10x5

VI. Discussion

Generally, as the size of the game board increases, so does the confidence level that the average score for the smart player is greater than the average score for the random player. Even at the smallest game board size of 5x5, there is incredibly strong evidence that the simplest instance of the smart player has a higher average score than the random player. In all the testing that was done, in no instance are we less than 99.9% confident that the new algorithm achieves a higher score than the random player. Thus, the more interesting results are between the two competing heuristics – H1 and H2.

Generally, as the size of the game board increases, so does the confidence that H2 gives a higher average score than H1 at all maximum depth levels. However, it could not be shown with 95% or greater confidence that H2 is better than H1 at any max depth greater than 1 until the board size reached at least 9x9. This trend is expected: the smaller the game state is, the more a single move will affect its state. Since each move affects its state more, the less important the current state of the board is for future states. The opposite is the case for a larger game state. This makes sense because as the size of the game state increases, the minimum valid moves remain static: match-three, match-four, or match-five.

Randomness is a significant factor in match-three games. Even if a player takes the most optimal move given the current game state and potential future states, it may not even be remotely close to the actual optimal move. This is because the game is not fully deterministic from the player's point of view, and thus the player has no idea what new pieces will drop down after a move. Because of this, the value added by lookahead exponentially diminishes with the distance of the lookahead. In all the testing that was done, in no instance was there a confidence of 95% or greater that a max depth of 4 resulted in a higher average score than a max depth of 3. Thus, having a lookahead of greater than 3 added no significant value to the new algorithm operating in this search space. Generally, as the maximum search depth level increases, the less of a statistical significance there is between the average scores of the competing heuristics.

An interesting result was that H1 had a higher average score than H2 with greater than 95% confidence for a board size of 5x5, max depth 3 and for a board size of 6x6, max depth 2. Even more interesting about the 6x6 test is that at 1 level of lookahead, H2 has over 99.9% confidence, but at level 2 H1 has over 95% confidence. It seems that there may be a slight trend that in smaller game states, choosing the next move based purely on average score might be the better strategy. Since average score is such a dominant factor in smaller states, factoring in anything else may actually lead to a more suboptimal result.

Another interesting result is the fact that standard deviation increases between the random player and the simplest smart player in each experiment. This may be happening due to the fact that when the game board size increases, the more a single move can affect the state in general. This affecting more of the state may translate into many chains of matching pieces after a single move. Additionally, two moves may have a significant difference in how each affects the game state. For example, one move may produce no chaining while the other produces dozens. To see this is the case, consider a game board of size 10x10. For one move, we match-three across the very top row, 3 new pieces drop down, and only 3 squares are affected in the game state. In the other move, we match-three across the very bottom row, 3 new pieces drop down, but now 30 squares are affected since the pieces above must now drop down one row. Thus, this variation among different moves translates into more variation in scores among different games.

For the optimization special case experiment, heuristic H1 was chosen as the heuristic for both smart players because there is less variation in its heuristic values, resulting in more moves that are tied. Thus, we expect there to be a more significant variation in the average score over 500 games. The results of this experiment generally show that this is a valid optimization, with a 99.9% confidence that the greedy player using this optimization achieves a higher score than the greedy player which does not. As we expect, the optimization makes less importance as the level of lookahead increases. This coincides with the general trend of decreasing difference between strategies as the level of lookahead increases.

REFERENCES

[1] J. Juul (2007). "Swap adjacent gems to make sets of three: A history of matching tile games." *Artifact* 1.4:205-216.

[2] “Application Analytics for Facebook, iOS and Android”. Internet: http://www.AppData.com, 2013-04-27.

[3] T. Walsh (2014). “Candy Crush is NP-hard.” arXiv preprint arXiv:1403.1911.

[4] L. Gualà, S. Leucci, and E. Natale (2014). "Bejeweled, Candy Crush and other Match-Three Games are (NP-) Hard." *arXiv preprint arXiv:1403.5830*.

[5] M. Ward (2008). "Casual games make a serious impact." *BBC News*: 135-136.