Solving Mahjong Solitaire Positions

T. Stam

December 4, 2007

Abstract

Mahjong Solitaire is a tile-removal game with imperfect information. This paper describes a research on solving Mahjong positions and offers a comparison of four strategies and two heuristics aimed at solving Mahjong game positions. Up to 62% of the positions can be solved using a strategy that picks stones based on a simple heuristic, thereby outperforming more complex search strategies.

1 Introduction

Mahjong[1] is the name of two different tile-based games that share nothing but the tileset and the name with each other. One of the games is a two-to-four-player game, which according to one myth was developed by Confucius in 500 BC [2]. The other, the subject of this paper, is a single-player game created by Brodie Lockard from 1981 onwards. In 1986, the game gained popularity when it was released by Activision under the trademarked name Shanghai, also programmed by Lockard. From then on, literally hundreds of versions of the game were written, under names such as Mahjong, Mahjong Solitaire, or "the Turtle Game" [5].

1.1 Research Question

In the subdomain of Artificial Intelligence which is concerned with computers playing board games, results vary widely. In English checkers the computer plays perfectly since 2007 [10], and will never lose. In chess, computers have been able to beat the human world champion since a decade ago. In go the strength of the computer is depending on the board size. go-5×5 is solved. In go-9×9, a computer can play at the same level as champions. For go-19× 19, computer players can be beaten even by mediocre players.

The question we will answer in the course of this study is: *How well do various solving strategies handle the solving of Mahjong Solitaire positions?*

[1]There are many spelling variations on Mahjong, such as Mah yongg, Mahjing, or Mah-jong. In this paper, "Mahjong" is used. See http://www.sloperama.com/mjfaq/mjfaq06.htm for more information on the spelling and history of the name.

1.2 Goal

The goal of the research in this paper is to describe and compare four strategies that solve positions of the imperfect-information variant of Mahjong.

1.3 Outline

Section 2 describes the game of Mahjong Solitaire. It starts with an overview of available research. Then the rules of the game are described. The section is concluded with a discussion on the game-tree complexity.

Section 3 describes the methods to solve a Mahjong position. It starts with a definition list. Next is a discusion on ways to set up a Mahjong game position, followed by a description of the heuristics. The section is concluded with the description of the four solving methods researched.

Section 4 starts with a description of how we set up our experiments. It then gives the results of these experiments and concludes with a discussion of the results.

Section 5 gives the conclusion to our research, and some suggestions for further research.

2 The Game

This section starts with an overview of the available research on Mahjong Solitaire, followed by the rules of the Mahjong game. It is concluded by a description of the game-tree complexity for Mahjong.

2.1 Prior Work

There is not much prior work about Mahjong Solitaire.

Christopher James Emmett wrote a report [4] that focused more on creating a working computer game. One of his aims was to generate methods that could solve imperfect-information game positions. However, due to time constraints, he only managed to create a solver for the perfect-information game and a *brute-force* solver.

Pedro Gemino Fortea described a method [6] to solve the perfect-information variant of the game. This method will solve any position if a solution exists. However, his method takes a few days to solve a board in the worst case.

2.2 Game Rules

A Mahjong position initially consists of a pile of 144 stones. The goal of the game is to remove all stones. In the context of this paper, a *solved* position is one where all stones are removed.

Each of the 144 stones in the game has a *face*. There are 36 distinct faces. Per face, there are four stones with that face.[2] Stones have to be removed in matching pairs.

A Mahjong position is a pile of stones, all face up. For some stones, the face is obscured by other stones above. A stone can be removed when it is possible to "slide it" to the left (west) or right (east), with no stone (partially) covering it. A stone can not be removed when it is touched by other stones on the left *and* on the right, even when it is not touched in the north or south. A stone can also not be removed when there is a stone exactly or partially on top of it.

A stone that can be removed is called a *free* stone.

In Figure 1, stone 2 is blocked by stones 1 and 3 because it can not move to the left or right, and it is not allowed to lift the stone vertically or to slide the stone towards the north or south. The stones 4, 5, 6, and 7 are blocked because stone 8 lies partially on top of them. Stone 10 is blocked on the left by stone 11, but free nonetheless as it can move freely to the right.

There are many ways to create a positions of 144 stones. The one studied in this paper will be the so-called "turtle" layout, because it is the layout used in Lockards's original 1981 Mahjong game [9], and it is a layout available as default option in various Mahjong Solitaire games. See Figure 2 for an example turtle layout, with only some faces filled in for clarity.

While this game can also be played with real Mahjong stones, the set-up process for a position reveals some of the stones that are later obscured, thereby giving the player information he should not have and spoiling some of the "fun".[3] In this paper, the solving strategies have no information on the faces of stones fully covered by other stones above, but full information on stones that are not or only partially covered. Also, it is not possible to backtrack, as otherwise the strategy could "cheat" and gather information on invisible stones by taking almost all stones, then backtracking up to the original pile, then use a perfect-information solver.

[2]In most implementations, as in the original Chinese Mahjong game, there are 34 quadruplets of faces, and 8 unique faces in 2 groups of four stones called *seasons*. The four stones of one season group can be picked as if they have the same face. In this paper, we regard the four stones in each of the two season groups to have the same face.

[3]Some computer implementations of Mahjong, such as the game in the Neopets.com game world called Koujong, briefly show each stone as a position gets built up, obscuring them later by stones above. Quickly recognizing and memorizing these stones might help in solving the position.

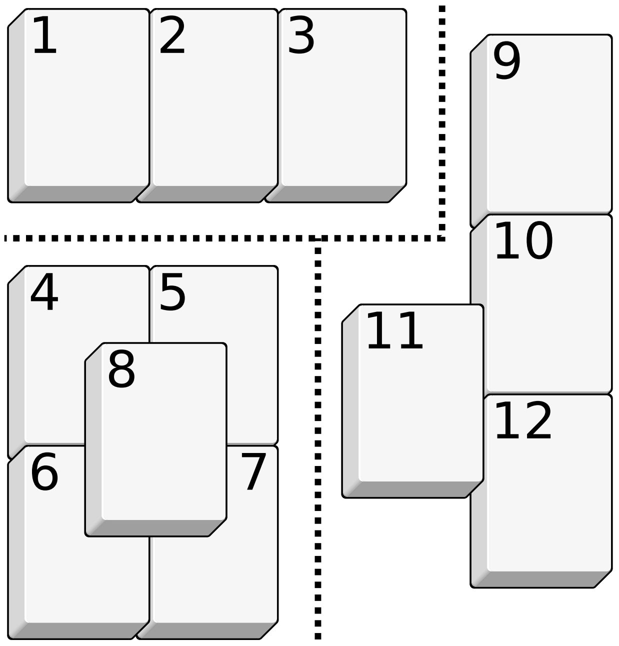

Figure 1: Stone 2 is blocked by stones 1 and 3. Stones 4-7 are blocked by stone 8. The other stones are free to be removed.

2.3 Complexity

The game-tree complexity is determined by the number of iterations or *plies* it takes to solve a position of the game, and the number of search options there are at each ply.

There are 144 stones in Mahjong, but for the first three algorithms described below, only the free stones matter. At the start of a position, 35 stones are free and thus the branching factor is 35. Testing over 1000 positions using the Random strategy and Random heuristic (both will be explained later) shows the average number of free stones is 21 for positions that are eventually solved and 18 for positions that are not. Using the MaxBlock heuristic with the Random strategy, the numbers are 24 for solved positions and 22 for unsolved positions.

For the Random and Greedy strategies, there are exactly 72 plies in a solved position. For the MultipleFirst strategy there might be less than 72 plies.

Setting the branching factor at 24, the game-tree complexity is $24^{72} \approx 7.5 \cdot 10^{116}$. This is comparable to the size of the game tree for Chess, which was estimated by Shannon to be about 10^{120} [11] and by Allis to be 10^{123} [1], but far greater than the game tree for the perfect-information variant of Mahjong, which has $3^{72} \approx 2.3 \cdot 10^{34}$ nodes, using Gemino Fortea's method [6].

For the Obstruction-Tree strategy, the case is different. Multiple stones might be picked in one ply, but there are also plies without solution. On average, there are 78 plies for solved positions. Per ply, there are up to 3 matches for the root pick, then up to 11 blockers per stone, each having up to 3 matches, and so on. The average maximum search level is 4. This means that an upper bound on the branching factor per ply is $(3 \cdot 11)^4 = 14,641$. The plies and branching factor combined give $((3\cdot 11)^4)^{78} \approx 8.2\cdot 10^{324}$ as an upper bound on the game-tree size using the Obstruction-Tree strategy. However, a lot of this game tree can be pruned, thereby decreasing the actual game-tree size considerably.

3 Methods Used

This section begins with some notes on the creation on the pile of stones. Then the heuristics used are described. This section is concluded with the four solving strategies that are the subject of our research.

3.1 Definitions

blocked A stone is blocked when there are stones touching it to the left and to the right, or when there are stones (partially) covering it from above.

free A stone is free when it is not blocked.

pair A pair is two stones with the same face.

pick To pick a stone is to remove the stone from the pile. A picked stone plays no further role in the game position.

position A position is one instance of the Mahjong game with a particular permutation of the faces over the available stones.

obstructor An obstructor is a stone blocking another stone either directly, or indirectly by blocking other blocking stones. A stone that has obstructors can still be free, as long as it *only* has obstructors either to the left or to the right, and none above.

solved A solved game position (or simply position) is a position where all stones are picked.

3.2 Creating a Position

The position can be set up in several ways. One way is to randomly distribute the 144 faces over the available stone positions. Care must be taken to use a "good" random generator. Using a linear congruential generator using period 144 is not a "good" random generator, as it can be shown that the results of this generator lie on at most 12 hyperplanes [7]. This could become visible in that certain stones could have a fixed "face distance", a fact that could be used by solving strategies to get some information they should not have on stones obscured by others. The Mersenne Twister random generator is a better alternative, as the serial correlation is very low [8].

Randomly assigning all the 4×36 faces over the 144 available stones, it is possible to generate a position that is not solvable. A trivial example is when all four stones of the same face are in one vertical column: only the top stone is able to become free, therefore it is impossible to generate a free pair of this face.

Another way to set up a pile is to generate a "solvable" position. Solvable in this context means there is at least one way to solve the position. However, it can be shown that, even in a solvable position, one can reach a situation where there is no deterministic way to solve the position [3]. To generate a solvable position, a position is built where all 144 stones have the same face. Then a random solver picks pairs until the board is empty. Then, 72 faces (2 of each) are randomly applied to the 72 removed pairs of the original empty board. An exception is when not all stones can be picked and only 1 free stones (and an odd number of non-free stones) remain in the game: no more pairs can be removed. In this case, the algorithm is restarted.

```
set up a pile of 144 stones with all the same face;
while *not solved* and $|free\ stones| > 1$ do
    randomly select $S_1, S_2$ from freeStones;
    pick $S_1$, $S_2$;
    add $S_1$, $S_2$ to pickOrder;
end
if *game not solved* then restart algorithm;
foreach $0 \le i < 72$ do faces[$i$] $\leftarrow i$;
foreach $0 \le i < 72$ do    /* permute stones */
    $n \leftarrow Random \cdot 72$;
    swap faces[$i$], faces[$n$];
end
foreach $0 \le i < 72$ do
    pickOrder[$2i$] $\leftarrow$ faces[$i$];
    pickOrder[$2i+1$] $\leftarrow$ faces[$i$];
end
rebuild stonepile with stones in pickOrder;
```

Algorithm 1: Generating a solvable position.

In the research described in this paper, we use a position generator that produces solvable positions, as described in Algorithm 1.

3.3 Selection Heuristics

The solving strategies described below all have certain points in their algorithms where a choice can be made between two or more options to proceed with. An option to deal with this choice is to randomly pick one stone. However, there might be an advantage to take one of the options over the others. In the imperfect-information game we are studying, it is usually not possible to oversee the consequences of the choice until after it is made.

An answer to this dilemma might be to use a heuristic which outcome might provide better results. One simple heuristic is to use the *blocking value* of each stone. Informally, the blocking value can be explained as how many other stones a stone is blocking. More formally, it is made up of by a horizontal and vertical component.

The horizontal component *hor* for stone S is defined as

$$hor(S) = \lfloor abs(|obstructorsLeft(S)| - |obstructorsRight(S)|)/2\rfloor \tag{1}$$

where $obstructorsLeft(S)$ and $obstructorsRight(S)$ are the stones touching stone S on the left or right and the stones touching those stones to their side, and so on. For the stone with ID 79 in Figure 2, there are 4 stones obstructing it to the left and 7 to the right; per formula 1 its horizontal blocking value is 1.

The vertical component *ver* is defined as

$$ver(S) = \begin{cases} \{\sum (hor(b_i) + 1) : b \in B\} & : Z > 0 \\ 0 & : Z = 0 \end{cases} \tag{2}$$

where B is the set of stones touching S from below and Z is the vertical level of stone S, where stones with $Z = 0$ are the bottom layer.

The blocking value is simply the sum of the horizontal and vertical component:

$$blockingvalue(S) = hor(S) + ver(S) \tag{3}$$

This heuristic, called "MaxBlock" will return the stone or pair with the highest (combined) blocking value when given a set of stones or pairs.

In our research, we studied the effect of the heuristic described versus a random choice. Therefore, the random choice was implemented as a "fake heuristic". When the "Random" heuristic is asked for the best stone or pair in a set, it will return a random one.

3.4 Solving Methods

This section describes the four strategies that were implemented to solve Mahjong game positions. In the pseudocode algorithms below, some small details might be omitted for clarity. These will be described in the accompanying text.

In the strategies and algorithms described below, there are some calls to a general heuristic. This can both be the MaxBlock heuristic or the Random heuristic.

Random strategy

The Random strategy searches for pairs in free stones, then picks a pair according to the heuristic. This process is repeated until the board is cleared or there are no more free pairs.

In finding pairs, stones with the same face are permutationally paired. When there are 2 free stones of the same face, this results in 1 pair. With 3 free stones, there are 3 pairs and 4 free stones result in 6 pairs.

```
initialize;
while $|Free\ pairs| > 0$ do
    Get freeStones;
    find pairs;
    pick best pair according to heuristic;
end
```

Algorithm 2: Random strategy.

MultipleFirst strategy

The MultipleFirst strategy tries to remove the set of free quadruples first, then the set of free triplets, then the set of free pairs. For the triplets, obviously only two out of three stones can be removed, as only one pair, accompanied by a "loose" stone, can be formed at the same time.

This strategy removes all stones in a set before reiterating. Note that when more than one quadruplet is found, all quadruplet sets will be picked before researching the free stones.

```
initialize;
while $|Free\ pairs| > 0$ do
    Get freeStones;
    foreach face i in freeStones do
    $n_i \leftarrow |face[i]|$;
    if $\max(n) = 4$ then
        foreach $n$ where $n_i = 4$ do
            pick first pair;
            pick remaining pair;
        end
    else if $\max(n) = 3$ then
        foreach $n$ where $n_i = 3$ do heuristically
        pick best pair;
    else if $\max(n) = 2$ then
        foreach $n$ where $n_i = 2$ do pick pair;
    end
end
```

Algorithm 3: MultipleFirst strategy.

Greedy strategy

For every stone that gets removed, zero or more stones become *free*. The Greedy strategy picks pairs in such a way that the number of free stones is maximally increased. When there is a tie, the best pair according to a heuristic is picked. After every pick, the available pairs are re-evaluated. The strategy does, by default, not pick quadruplets in advance.

```
initialize;
while |$Free\ pairs$| > 0 do
    Get freeStones;
    foreach $Stone\ S_i$ in $freeStones$ do
        $S_i \leftarrow$ |stones freed by removal of i|;
    end
    Pick pair $S_i, S_j$ such that $S_i + S_j$ is
    maximalized;
end
```

Algorithm 4: Greedy strategy.

Obstruction-Tree strategy

The remainder of this section lists the general workings of the Obstruction-Tree strategy. It is concluded with an example of the application of the Obstruction-Tree strategy.

The Obstruction-Tree strategy is a two-step search-tree strategy.

This algorithm will try to find a match for a "pick candidate", and will then try to find all stones obstructing it. It does this by alternating between finding matches for a particular stone, then finding the *obstructors* for that stone, then finding matches for these obstructors, and so on, and so on.

The algorithm creates a search tree where the pick candidate is root. Its children are zero to three matches. Each match has three sets as node, the stones obstructing it from the left, from above and from the right. Each obstructor has another zero to three matches, and so on.

Each node, both matchFinding and obstructorFinding, will return a pickPath. The pickPath holds a set of stones to be picked and a status, as follows:

- A negative status means no possible path can be found in this node or in its subtree.
- A zero status means there is no solvable pickPath nor a fatal obstruction has been found yet in the subtree of the current node, and a conclusive answer can be found by searching deeper in the tree in a later iteration.
- A positive status means there is at least one possible path found in this node or its subtree. Along with a positive status, this pickPath is returned.

The resulting tree is traversed by an iterative-deepening search. The tree is in fact traversed depth-first, but not deeper than a preset search depth. As long as no nonzero status is returned by the root node, the search depth will be increased by one, and the tree is researched. This traversal method is similar to a breadth-first search. The difference is that with this algorithm each level is completely finished, whereas using breadth-first the search is stopped as soon as a match is found.

Once a pickPath is found at the root node, it is picked depth-first. The pick candidate and its match, who form the subject of the root node, is always picked last, after all stones blocking it have been cleared

When the path is picked, the findMatches subalgorithm ends. The main algorithm will invoke it again until the whole stone pile is picked.

This algorithm will, at very shallow levels, create a tree with more nodes than there are stones in the game. Therefore, there are a few pruning methods:

- Whenever a node returns a positive or negative status, this value is stored. On later iterations, it is returned immediately without further deepening the tree. A negative effect of this is that a possible "better" pickPath deeper in the tree will not be found.
- To prevent cyclic searching, each node is given a list of stones higher up in the search tree. If a match or obstructor is found that already was used higher in the tree, the current node will return a negative status.
- In the search for obstructions, not all obstructors need to be researched. For each of the directions left, above and right, just one node with a negative status means it is not possible to unblock the subject stone in that direction, i.e., is not solvable. Whenever one of the obstructors from above or both on the left and the right are not solvable, the node as a whole is not solvable and will return a negative status. On the converse, when all of the obstructors from above have a positive status and either all the obstructors to the left or all the obstructors to the right (or both) have a positive status for all obstructors in those directions, the positive status will be returned, even though the left or right obstructors' status might be negative or undecided.
- In the main algorithm, the rootPick is chosen according to a heuristic. When no pickPath is found, the first (best) stone returned by the heuristic is skipped and the next best stone is tried as a rootPick, and so on, until either a pickPath is found and the next iteration will use the first stone given by the heuristic again, or all free stones are skipped in which case the algorithm ends.

The different paths in the node do not communicate with each other, so a situation can arise were a certain stone is used in two or more parallel paths in the search tree. A stone can not be picked more than once during a position. When a certain stone appears two or more times in a pickPath, it might happen that all stones directly or indirecly blocked by the stone that appears twice or its match can not be picked during this iteration. This is not a problem, as usually in later iterations the root or parts of this path will be picked.

In the implementation, the algorithm has two main supporting classes, TwoStageNode and MatchObstructorSet. TwoStageNode holds the matches and the

MatchObstructorSets for a certain stone. A MatchObstructorSet holds all the stones obstructing a *match*, or actually the TwoStageNodes that hold those matches. The main algorithm alternates the methods findMatches and expandObstructors alternatively with an increasing depth level. They build up the obstruction search tree, until either a pickPath is found, or a negative status, meaning no pickPath can be found, is reported back.

Below are the algorihtms that make op the Obstruction-Tree strategy. Algorithm 5 lists the main algorithm for the Obstruction-Tree strategy. Algorithms 6 and 7 lists the code for findMatches in the TwoStageNode and MatchObstructorSet classes, respectively. Algorithms 8 and 9 do the same for expandObstructors.

```
initialize;
endcondition ← false ;
skipFirst ← 0;
while !endcondition do
    rootPick ← heuristic.getBest(freeStones, skipFirst);
    // 0 is depth in tree
    rootTSN ← new TwoStageNode(rootPick, 0);
    level ← 0;
    while rootTSN.status = 0 and rootTSN.expandable = true do
        pickPath ← findMatches(rootTSN, level);
        expandObstructors(level);
        status ← pickPath.getStatus();
        level++;
    end
    if status = 1 then /* pickPath exists */
        pick all stones in pickPath;
        skipFirst ← 0;
    else /* no pickPath */
        skipFirst++;
    end
    if |freeStones| = 0 then
        endcondition ← true;
    end
    if |freeStones| = skipFirst then
        endcondition ← true;
    end
end
```

Algorithm 5: Obstruction-Tree strategy main algorithm.

```
if depth = level then
    matches ← visible matches;
    if |matches| = 0 then
        return status -1;
    end
    if |matches where status = free| > 0 then
        return first pickpath with free status;
    else
        return status 0;
    end
end
else if depth ≠ level then
    for all MatchObstructorSets get pickPath;
    return pickPath with highest status;
end
```

Algorithm 6: findMatches subalgorithm within TwoStageNode.

```
// LAR = Left, Above, Right
// each <variable>LAR should be read as three variables, one for each direction
// obstructorsLAR holds the TwoStageNodes
foreach stone S in blockersLAR do
    if level = depth+1 then
        obstructorsLAR.add new TwoStageNode(S, depth+1);
    end
    statusLAR ← min(obstructorsLAR.status);
    if statusAbove < 0 or (statusLeft < 0 and statusRight < 0) then There is no solvable pickPath
        return status -1;
    end
    if statusAbove and (statusLeft or statusRight) ≥ 1 then There is a pickPath
        return new pickPath(combined pickPaths above and left or right, where status is higher);
    elsethere is no decision yet, iterate deeper later
        return status 0;
    end
end
```

Algorithm 7: findMatches subalgorithm within MatchObstructorSet.

```
initialize;
if $depth = level$ then
    // per match, create new
       MatchObstructorSet
    foreach stone $S$ in matches do
        MOSkeeper.add(new
        MatchObstructorSet($S$));
    end
    return true;
end
else if $depth \neq level$ then
    // return true if at least 1
       MatchObstructorSet is true
    foreach MatchObstructorSet in MOSkeeper
    do
        if MatchObstruc-
        torset.expandObstructors(level) = true
        then
            return true;
        end
    end
    return false;
end
```

Algorithm 8: expandObstructors subalgorithm within TwoStageNode.

```
initialize;
// Check whether TSNs are solvable
foreach TwoStageNode in obstructorsLAR do
    solvableLAR ←
    min(TwoStageNode.expandObstructors);
end
if solvableAbove = true and (solvableLeft = true
or solvableRight = true) then
    return true;
else
    return false;
end
```

Algorithm 9: expandObstructors subalgorithm within MatchObstructorSet.

For an example of the workings of Obstrution Search Tree strategy, see Figure 2. The search tree for this example is in Figure 3. Assume the stone with ID 123 is taken as pick candidate. It has two (visible) matches: stones 136 and 142.

As none of these matches is free, it is not possible to pick a combination of stone 123 with one of them. After expanding the obstructors for both matches, it is found that stone 136 has one obstructor to the left, none above and two to the right. Stone 142 has one on top of it and one to the left of it.

After finding the matches for all obstructors, 2 possible pickpaths appear. Stone 143 can be picked together with 44, such that stone 142 is free from above (it already was from the right). Stone 123 can then be picked together with stone 142.

Alternatively, stone 136 can be unblocked from the left by removing the blocker 135 along with its match 92. Stone 136 is then free to be picked with stone 123.

Note that stone 136 is blocked by the right by stones 137 and 138. While 99 is a free match to 137, the match for stone 138, 81, is not free. Therefore, stone 136 can not yet be freed from the right. If the two pickpaths drawn out before would not exist, the blockers for stone 81 could be expanded to find if a path existed that way.

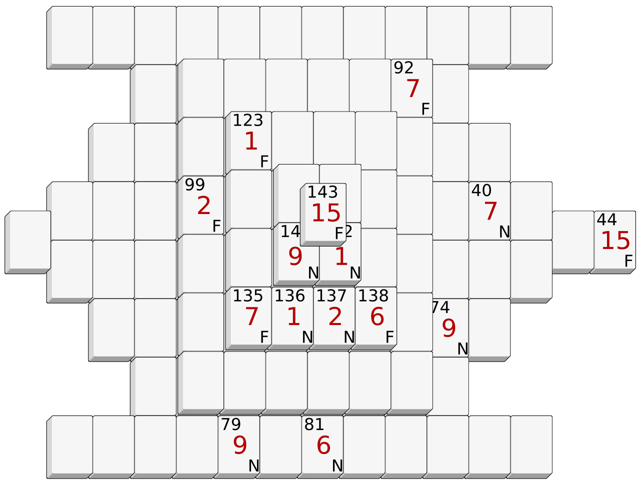

Figure 2: Turtle layout with most stones omitted for clarity. See Figure 3 for a legend. The stones under the top stone have ID 141 (with face 9) and ID 142 (face 1).

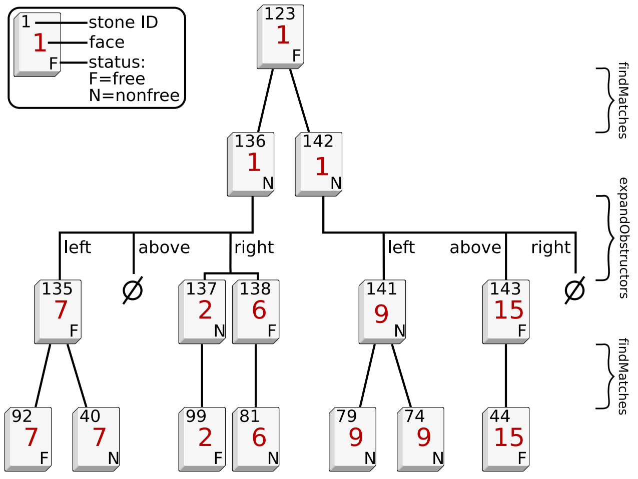

Figure 3: The obstruction search tree for Figure 2.

4 Experiments

This section starts with a description of how we set up our experiments. It continues with the results of these experiments, and concludes with a discussion of the results.

4.1 Experimental Setup

The algorithms and heuristics as described above were incorporated into a computer program using the Java programming language. The combination of 4 strategies and 2 heuristics makes 8 test configurations. A test set with 20,000 positions was created. Each of the solver's 8 configurations was run with this test set.

Data recorded per configuration were the number of solved positions, the number of stones remaining if a position was not fully solved, and the run time. It should be taken into account that there are many factors having an influence on the run time, and the algorithms were designed nor implemented with computation speed in mind. The run time should only be seen as a very general comparator for speed classes and not as a perfomance indicator.

4.2 Results

The following table lists the results for the different strategies and heuristics. Given are the number of solved positions s and percentage $s\%$, the average remaining number of stones in positions not solved r, and time t in seconds the strategy took to run 20,000 times.

Table 1: Results.

	s	$s\%$	r	t
	Random heuristic			
Random	7934	39.67	35.79	360
MultipleFirst	9602	48.01	32.19	277
Greedy	9249	46.25	26.02	387
Obstruction Tree	9484	47.42	28.27	781
	MaxBlock heuristic			
Random	12425	62.13	25.28	692
MultipleFirst	12182	60.91	25.40	313
Greedy	11206	56.03	22.32	514
Obstruction Tree	11928	59.64	22.94	1295

4.3 Discussion

The most striking result is that the Random strategy is the worst performing strategy of all when combined with the Random heuristic, but the best performing when combined with the MaxBlock heuristic. The converse is not true: the MultipleFirst is best of all strategies when combined with the Random strategy, but still performs second-best together with MaxBlock. Nonetheless it is interesting to note that more complex strategies do not benefit from a heuristic as much as a simple one does.

All strategies perform better using the MaxBlock heuristic, and in fact, the MaxBlock heuristic absolutely dominates the Random heuristic for the four selected strategies for the number of solved positions. The increase in solving percentage for MaxBlock relative to the Random heuristic is 1.57 for the Random strategy, 1.27, 1.21, and 1.26 for the MultipleFirst, Greedy and Obstruction-Tree, respectively. Also, the number of remaining stones for unsolved stones is lower for the MaxBlock heuristic than when using the Random heuristic.

When looking at the time needed to run the strategies, the MultipleFirst strategy is the fastest strategy of all, and yields the best solving/time factor. The Obstruction-Tree strategy is a factor 2 to 4 slower than the other strategies while performing worse than the fast MultipleFirst strategy. However, we repeat that none of the algorithms was designed with computing speed in mind and all might be significantly optimized in speed.

A small investigation over 20 positions (positions 247-266) on the website www.mahjongcravings.com revealed that on average 14% of the positions is solved, with 23% as a maximum. It is not possible to conclude from this that all algorithms above perform better than humans, as there is no information on whether any of the positions was abandoned midway. Also there is no information on whether the positions were played multiple times (giving players more insight in the position on successive tries) and whether all positions were generated randomly or solvable. It seems unlikely that a human who is forced to finish a position until either the pile is solved or there are no more free pairs, performs much worse than the Random strategy/Random heuristic pair, indicating that these statistics are not over finished positions. However, when asked, four people in the environment of the author who played the game but not often, estimated they solved between $\frac{1}{8}$ and $\frac{1}{3}$ of the positions they played, suggesting that the Random strategy/MaxBlock heuristic pair has a significant better solving percentage than at least basic players.

5 Conclusions and Further Research

The conclusion of this research is that, to solve positions of the perfect-information game of Mahjong, using a heuristic without using a complex search strategy is a simple yet very effective way to achieve that goal.

Also, there is evidence a computer player can play at a level at least the same or higher than novice human players.

5.1 Suggestions for Further Research

During our research, we encountered many tracks that could have been explored with more attention than they received now. Our suggestions for further research include:

- Researching the effects of various stonepile generation methods, and in particular various random-number generators, on the playability and predictability of a Mahjong game position.
- Researching the effect of using other layouts than the Turtle layout. Some layouts have far fewer free and visible stones and thus a much more reduced information set. Do the algorithms perform as well on other layouts as on the Turtle layout, and equally well relative to each other?
- Researching whether other strategies than search-based strategies, such as neural networks, perform as well as or better than the algorithms described in this research.
- Researching how well human players can play the game of Mahjong, and analizing human strategies to see if these can be translated into winning computer search strategies.

References

[1] Allis, L. Victor (1994). *Searching for Solutions in Games and Artificial Intelligence*. Ponsen & Looijen, Wageningen.

[2] Butler, Jonathan (2001). The tiles of mah jong. `http://www.atdesk.com/jon/mahjong.html`.

[3] Elonen, Jarno (2003 - 2004). There is no deterministic way to solve a mahjongg solitaire game. `http://elonen.iki.fi/code/misc-notes/no-alg-mahj-solit/`.

[4] Emmett, Christopher James (2007). Mahjong solitaire. Technical report, The University of Manchester.

[5] Fregger, Brad (1999). *Lucky That Way*. Sunstar Publishing,U.S.

[6] Gimeno Fortea, Pedro (1998-2005). Mahjongg solitaire solver. `http://www.formauri.es/personal/pgimeno/mj/mjsol.html`.

[7] Marsaglia, George (1968). Random numbers fall mainly in the planes. *Proc Natl Acad Sci U S A.*, Vol. 61(1), pp. 25–28.

[8] Matsumoto, Makoto and Nishimura, Takuji (1998). Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, Vol. 8(1), pp. 3–30.

[9] Petersen, Vegard Krog (2003-2005). Solitaire mahjongg - a guide to the world of the computer tile-matching, solitaire mahjongg games - history. `http://home.halden.net/vkp/vkp/`.

[10] Schaeffer, Jonathan, Burch, Neil, Bjornsson, Yngvi, Kishimoto, Akihiro, Muller, Martin, Lake, Rob, Lu, Paul, and Sutphen, Steve (2007). Checkers is solved. *Science*, Vol. 317.

[11] Shannon, Claude E. (1950). Programming a computer for playing chess. *Philosophical Magazine*, Vol. 41, No. 314.