Mahjong Solitaire
Computing the number of unique and solvable arrangements

J.N. van Rijn
Leiden Institute of Advanced Computer Science
Universiteit Leiden
jvrijn@liacs.nl

March 25, 2011

Abstract

In this paper we will provide definitions on various aspects of the game Mahjong Solitaire. We will show in how many ways n tiles can be arranged and provide a recurrence relation on this sequence. We will also provide a recurrence relation on the number of ways we can divide these tiles over a various number of sets, and even a closed formula when we apply the constraint that each set may only contain two tiles. We will show that the product of these two sequences is not equal to the number of arrangements that can be created. We will provide algorithms for generating all arrangements and calculating the number of arrangements that can be won.

1 Introduction

Mahjong Solitaire is a one player puzzle game mainly played on the computer in which the player a randomly arranged stack of tiles is presented. An example is shown in Figure 1. The goal is to remove all tiles in matching pairs of two, restricted by the rules that will be described in Section 2. The game was originally created by Brodie Lockard in 1981 for the PLATO computer

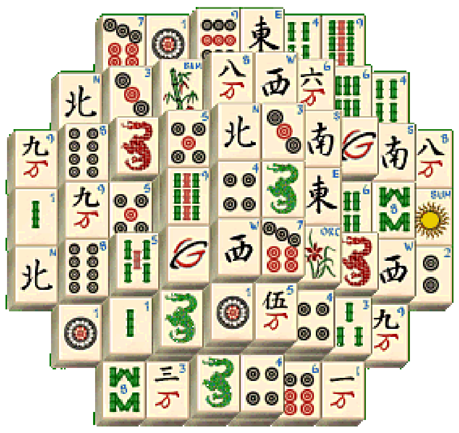


Figure 1: An example of the game Mahjong Solitaire, taken from [14].

system, but it gained a great popularity in the late 1980s when it was ported to other platforms [10].

Despite the great popularity which is still evident nowadays, not much research has been done on this game [2]. Most noticeable contributions are [1] and [11], which both examined the complexity of a different variant of the game. In [13], an algorithm is presented which can determine whether a game of Mahjong can be won or not, and if so, what strategy should be used, provided that we have perfect information. In [12] it is proven that if we don't have perfect information, no such infallible algorithm exists.

The goal of our research is to gain a greater knowledge about the game Mahjong, in particular about what the probability is that such a game can be won. In Section 2 we will provide basic definitions about the variants we will consider. In Section 3 other basic theory from various websites is presented, along with one of our contributions, a definition which can be used to find patterns of an unwinnable game. Section 4 decomposes the game into its core aspects, and provides some sequences that can be used to predict how many different arrangements can be created, using a certain number of tiles. Section 5 provides some brute force algorithms which are able to determine which and how many instances can be won. In Section 6 these algorithms are used in order to gain information about the smaller instances of Mahjong. In Section 7 a conclusion is drawn, and some suggestions for future research are mentioned. Finally, in de Appendices pseudo code is provided for the algorithms that are used in this research.

2 Mahjong solitaire

For describing the game Mahjong Solitaire, we allow ourselves to use a slightly more general definition derived from the excellent version provided by [1].

The game uses Mahjong tiles (comparable with cards), that are divided into m sets $\mathcal{T}_p$ of $|\mathcal{T}_p| = s_p$ matching tiles, where s_p is an even number ($p = 1, 2, \ldots, m$). The collection of all these sets is called S. Formally $S = \{\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_m\}$. We generalize the standard game simply by assuming that there is an arbitrarily large, finite number of tiles. Initially, the tiles are organized in a preset arrangement of rows, and the rows may be stacked on top of each other. As a result, some tiles are hidden under other tiles; it is assumed that each possible arrangement is equally likely. A tile that is not hidden and is at the end of a row is said to be *available*. In a legal move, any pair of available matching tiles may be removed, resulting in a new configuration of the tiles in which up to four previously hidden tiles are uncovered. The player wins the game if all tiles are removed by a sequence of legal moves.

To define the game more formally, we define the set of Mahjong tiles to be $\mathcal{T} = \bigcup_p \mathcal{T}_p$ where $\mathcal{T}_1, \ldots, \mathcal{T}_m$ are disjoint sets of tiles. The size of set $\mathcal{T}_p$ is denoted by s_p, where s_p is even. We say tiles a and b *match*, if and only if for some p it holds that $a, b \in \mathcal{T}_p$. A *configuration* C is a set of positions (i, j, k), where each of i, j, k is a non-negative integer, satisfying the following constraints:

1. If $(i, j, k) \in C$ and $(i, j', k) \in C$ where $j < j'$, then for every j'' in the range $[j, j']$, $(i, j'', k) \in C$.
2. If $(i, j, k) \in C$ where $k > 0$ then $(i, j, k - 1) \in C$.

Intuitively, this captures the fact that tiles are arranged in three dimensions. Tiles can be stacked on top of each other; all tiles with common k are at the same height. Tiles at the same height, with common i index, form a row. The first condition ensures that there cannot be *gaps* in a row; the second, that a tile at height $k > 0$ must have a tile underneath it (in fact, at position $(i, j, k - 1)$).

With respect to a given configuration, a position (i, j, k) is *hidden* if in the configuration also a position $(i, j, k + 1)$ exists; the other positions are called *visible*. An *arrangement* consists of a set of tiles $\mathcal{T}$, a configuration C

of size $|\mathcal{T}|$, and a 1-1 function f from the positions of C to all tiles in $\mathcal{T}$. Intuitively, this means that $f(i, j, k)$ is the tile at position (i, j, k). If this function maps position (i, j, k) to tile t we say t is in position (i, j, k). The elements of $\mathcal{T}$ will be mapped to the elements of C in such a way, that every possible combination is equally likely. With respect to a given arrangement, we say a position (i, j, k) is *available* if it is not hidden, and either position $(i, j-1, k) \notin C$ or position $(i, j+1, k) \notin C$ or both. An arrangement is called empty if $\mathcal{T}$ is empty.

Let $X = (\mathcal{T}, C, f)$ be an arrangement. Each pair $\{(i_1, j_1, k_1), (i_2, j_2, k_2)\}$ of available positions at which there are matching tiles $\{f(i_1, j_1, k_1), f(i_2, j_2, k_2)\} \subseteq \mathcal{T}_p$ for some p defines a *match* of X. We say arrangement X' is obtainable from X via match $\{(i_1, j_1, k_1), (i_2, j_2, k_2)\}$ if $X' = (\mathcal{T}', C', f')$, where $\mathcal{T}' = \mathcal{T} - \{f(i_1, j_1, k_1), f(i_2, j_2, k_2)\}$, $C' = C - \{(i_1, j_1, k_1), (i_2, j_2, k_2)\}$ and finally, f' is almost the same as f, with the only difference that it is not defined on the positions $\{(i_1, j_1, k_1), (i_2, j_2, k_2)\}$, since they are no longer in C. A sequence of moves from arrangement X leads to a win if it results in the empty arrangement.

2.1 Game variations

In the definition of the game given in the section above, there is a strict separation between the tiles that are hidden, and the tiles that are visible. We want to emphasize the fact that when a tile is visible, this doesn't necessarily have to be available. I.e., all tiles on the top row are visible, but only those on the sides (where $(i, j-1, k)$ or $(i, j+1, k)$ is not in the configuration) are also available and can be played, if a match is found.

For computational and optimization issues, it can in some cases be interesting to have the possibility to know what the values of the hidden tiles are. A situation were we allow the player to peak into the hidden tiles, is called an *open instance* of Mahjong. Whenever this is not allowed, this is just a regular instance of Mahjong. We also refer to this as a *closed instance.*

2.2 Definitions

Here we will provide an overview of definitions we use in this report. Some of these definitions might have already been mentioned, but will be further explored here.

- A *position* (typically denoted as (i, j, k)) is an element of the configuration C.
- A *configuration* C is a collection of positions, respecting the conditions given in Section 2.
- A *row* $(R \subseteq C)$ (for given i, k) consists of all positions $(i, j, k) \in C$, that share the same value on i, k.
- A *cross section* $(CS \subseteq C)$ (for given i) consists of all positions $(i, j, k) \in C$, that share the same value on i.
- Every *tile* $t \in \mathcal{T}$ has a certain value, which has no numerical or hierarchical meaning. This value is just to determine which tiles belong to the same group of *matching tiles*, that are denoted by $\mathcal{T}_p$.
- A *candidate pair* is a set of two tiles $a, b \in \mathcal{T}_p$ for some p. The number of candidate pairs of a certain $\mathcal{T}_p$ is $\binom{|\mathcal{T}_p|}{2}$.
- A *move sequence* of length n is a chronological series of n moves performed over an arrangement X such that $X^{(n)}$ is obtained. Here, $X = X^{(0)} \to X^{(1)} \to X^{(2)} \to \ldots \to X^{(n)}$, where each arrow denotes a move. This move sequence is called a *winning move sequence*, if as a result of these moves $X^{(n)}$ is empty. If there exists at least one winning move sequence, we call the arrangement *solvable*. Otherwise we call the original arrangement *blocked*. A move sequence is called an *optimal move sequence* if the number of tiles in $X^{(n)}$ is minimized, or equivalently: if n is maximal. Note that a winning move sequence is always an optimal move sequence. When a sequence is blocked, a move sequence that results in the least remaining tiles is called the optimal move sequence.

3 Theory

This section will provide basic theory, gathered from various sources.

3.1 Complexity

In [1] it is proven that in the closed version it is PSPACE-hard to approximate the maximum probability of removing all tiles within a certain factor,

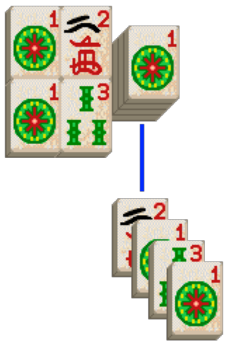


Figure 2: An instance of Mahjong, with one winning move sequence. Image is taken from [12].

assuming that there are arbitrarily many quadruples of matching tiles and that the hidden tiles are uniformly distributed. In [11] it is proven that in the perfect-information version of this puzzle, in which all tile positions are known, it is NP-complete to decide whether all tiles can be removed.

3.2 Closed instance

In [12] it is mentioned, that it is impossible to come up with an algorithm that could solve any closed instance of Mahjong, even when one or more winning move sequences actually exist. Without further analysis, it is immediately clear that the only way to win the game from Figure 2 is to start by taking out the 1-tiles on top of the stack and one of the two on the left. There is, however, no way to know which of the left-hand side 1-tiles to pick first without peeking into the tall stack. Therefore, the statement made in [12] is correct.

3.3 Game blockings

Intuitively, it is not very hard to come up with an example that there will not always exist a winning move sequence. The most trivial case, for example, is when all tiles are stacked on positions on top of one another. In this situation all tiles in $\mathcal{T}$ share the same i and j coordinate, but have a different value for k. Since there is only one tile that is available, no pairs can be made, hence there is no winning sequence for this arrangement, so this game blocked.

We have analyzed all possible ways in which an arrangement with size smaller than 12 can be blocked, and all of them can be categorized by means of the same definition, after a certain number of moves is done.

Let $B \subseteq C$ be a non-empty set of positions, where all corresponding tiles are different. I.e., if $(i,j,k),(i',j',k') \in B$ with $(i,j,k) \neq (i',j',k')$ then $f(i,j,k)$ and $f(i',j',k')$ are not in the same $\mathcal{T}_p$ (for some $p \in \{1,\dots,n\}$). If we can find a set of positions such that the following so-called *blocking property* holds, the game is blocked. Hence there is no winning move sequence. In this case, B is called the *blocking set*. The property is the following: If $r = f(i,j,k)$ with $(i,j,k) \in B$ and for some $(i',j',k') \neq (i,j,k)$ in C we also have $f(i',j',k') = r$, then

- $\exists (i',j',k'') \in B$ with $k'' > k'$, or
- $\exists (i'',j',k'') \in B$ and $\exists (i''',j',k''') \in B$ with $i'' < i' < i'''$ and $k'', k''' \geq k'$

Note that the first one is a special case of the second one, if we allow $i'' \leq i' \leq i'''$. Informally, we intend to select a set of positions (which we previously defined as B), from which we know that we can never play one of the tiles at those positions, because all the tiles that have the same value as the tiles on one of these positions, are already blocked by one or more of the other tiles, with a position in B. If such a set exists, the game can never be won. The most trivial case, obviously, is when tiles with the same value are stacked on top of each other (as shown in Figure 3(a)), but also more complex blocking situations can emerge from this. In Figure 3(b), an example is shown where a blocking set consisting of two tiles a and b is found. Note that this arrangement is blocked because all tiles that are element of the same $\mathcal{T}_p$ as a or b, are in the area that is colored gray.

In order to know whether a game is blocked, in a worse case situation one has to test for a number of at most

$$\sum_{A \subseteq \{1,\dots,m\}} \left(\prod_{a \in A} s_a \right)$$

different subsets of C whether the blocking property holds. If so, this subset is a valid blocking set, hence there is no winning move sequence. If this is not the case, we don't know whether the arrangement is blocked, unless the condition $|\mathcal{T}_p| = 2$ for each p holds true.

The reason for this is as follows: The blocking definition looks for a pattern in the arrangement, where a group of tiles from different sets are positioned

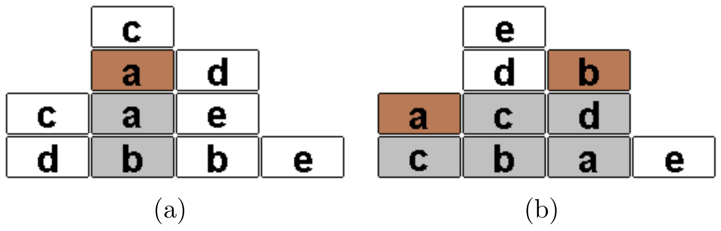


Figure 3: In both (a) and (b) an arrangement is shown which is blocked. Here, $\mathcal{T}_1$ consists of two a-tiles, $\mathcal{T}_2$ consists of two b-tiles and so on. The blocking set is colored brown, the tiles that are impossible to play by means of the blocking set are colored gray.

in such a way, that all other tiles from these sets are blocked by this group of tiles. Having more than two tiles in a certain set, can disturb these patterns. We can think of a situation where an arrangement is not covered by the definition, but after any legal move sequence resulting in an arrangement where for all $p : |\mathcal{T}_p| = 2$ any of these arrangements will be covered by this definition. In Figure 4 such an example is shown. Here an arrangement is displayed which is clearly blocked, however it is not covered by the blocking definition because two of the three free a-tiles disturb the pattern. Doing an arbitrary move will result in having an arrangement consisting of sets with size two, all covered by the blocking definition. Since this is the case we can conclude that the initial arrangement was blocked, despite it was not covered by the blocking definition. From this we can conclude that when we have a blocking set where the condition $|\mathcal{T}_p| = 2$ for all p does not hold true and where no blocking set exists, we do not know whether this arrangement is solvable. In order to find out whether this arrangement is really solvable, we will have to look further in the game tree. In the game tree every edge represents a legal move, every vertex represents an arrangement and a vertex is considered a leaf whenever a blocking set is found, or the arrangement obeys the condition that for all p it holds that $|\mathcal{T}_p| = 2$. If there exists at least one leaf where there does not exist a blocking set in the arrangements, we can consider the initial arrangement solvable. Otherwise the initial arrangement is blocked.

The blocking definition as it is now seems pretty useless because it seems only feasible to use it on arrangements where for all p it holds that $|\mathcal{T}_p| = 2$. If one were to expand it in such a way that it would work on any arrangement without looking into the game tree, it would be a great result. In Section 6.2

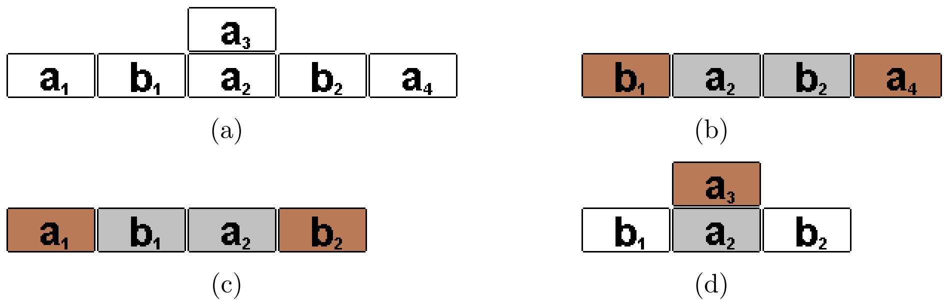


Figure 4: In (a) an arrangement consisting of four a-tiles and two b-tiles is shown. Initially, there are three moves possible. Playing a_1 with a_3 will result in the arrangement shown in (b), playing a_3 with a_4 will result in the arrangement shown in (c) and finally playing a_1 with a_4 will result in the arrangement shown in (d). The blocking set, if it exists, is colored brown, the tiles that are impossible to play by means of the blocking set are colored gray.

we will report on the percentage of blocked arrangements that can be classified by means of the current blocking definition.

4 Data representation

In this section we will give definitions, propose conventions and present algorithms on various aspects of Mahjong.

4.1 Configuration

Section 2 already provided a solid definition for configurations; in this section we will elaborate on how to generate all configurations of a certain size. We will only consider configurations consisting of one cross section. One important property of the algorithm we are going to present is that it filters out all duplicates by reflection. The pseudo code for the presented algorithm is available in Appendix A. We propose to represent a configuration consisting of one cross section as an array of integers C, where an element of C (with index j) denotes the number of tiles that are stacked op top of each other at column j. We want to emphasize once more that by means of the definition provided in Section 2, a configuration is only valid when none of the cross sections contain any gaps.

We are using a recursive solution, as we are first going to generate the small configurations (starting with size 0) and from these we are generating all bigger configurations. The first thing we need to define is which size the configurations are that we are interested in. After this a call will be made to the recursive function, *allConfigurations*. The parameters this function accepts are the array *currentConfig*, which is the array representing the current configuration we are trying to expand, the integer *last*, which contains the last element of *currentConfig*, and finally the boolean *desc*, which keeps track of the question whether we already had a column with a strictly smaller value than the column before. This is for preventing that any gaps will occur in the configuration.

The first thing the algorithm checks, is whether *currentConfig* already reached the required size. If not, the algorithm will try to expand a copy of *currentConfig* in the next loop, otherwise we will check whether *currentConfig* is lexicographically smaller than or equal to its reversed version, so we can add it to our found solutions. Whether this is the case or not, we will return from the function. The last part of the algorithm is where we expand all solutions that were smaller than the requested size. We will only do this in such a way that gaps will be prevented. Some sort of pyramid form will arise: every column will be larger than or equal to its previous column, until a certain column t. Note that t can be every arbitrary column, also the first or the last column. From t on every column should be smaller than or equal to its previous column. We'll keep track on whether we already have made our first descending move by the boolean *desc*. When *desc* is *true*, no columns greater than its previous column will be added to the configuration, otherwise a gap would arise and the configuration would not be valid.

The number of valid configurations with size n is presented in the next table. Here, C_{tot} is the number of configurations the algorithm generated, C_{refl} is the number of configurations, when we don't mind having configurations double because of reflections, and C_{sym} is the number of generated configurations that are symmetric.

n	C_{tot}	C_{refl}	C_{sym}
0	1	1	1
1	1	1	1
2	2	2	2
3	3	4	2
4	6	8	4
5	9	15	3
6	17	27	7
7	26	47	5
8	45	79	11
9	69	130	8
10	113	209	17
20	5673	11240	106
30	135938	271404	472
40	2104752	4207764	1740
50	24323418	48641220	5616
100	317880216	635111413	649019

There is no sequence reported in [3] that corresponds with the C_{tot} sequence. However, in [4, 8] a sequence is reported that corresponds with our C_{refl} sequence, called "the number of stacks, or planar partitions of n". In [5, 9] also a sequence is reported that corresponds with our C_{sym} sequence, "the number of palindromic and unimodal compositions of n". From these two, we can derive the number of valid configurations. Note that in C_{refl}, all configurations are counted twice, except those that are symmetric. It is not hard to see that we can compute the C_{tot} sequence by

$$C_{tot} = \frac{C_{refl} - C_{sym}}{2} + C_{sym}$$

which is equal to

$$C_{tot} = \frac{C_{refl} + C_{sym}}{2}$$

Another nice thing to notice, is that every element with an odd index in the C_{sym} sequence, is smaller than or equal to the previous element in the sequence. We don't have a formal proof that this will always hold, but intuitively speaking it is correct. When having a configuration of size n with n being an odd number, we can only make symmetric configurations with an odd number of columns. Suppose the number of columns of a configuration is c. When for a configuration with an odd size c is even, in order for the configuration to be symmetric, the columns with indexes $1, \ldots, c/2$ should be the same as the columns with indexes $c/2 + 1, \ldots, c$ reversed. This can only be the case when both of these subsets have the same size, which can never be the case when the total size is an odd number. This constraint does not apply when having a configuration of size n with n being an even number, since now we can make symmetric configurations with both an even number of columns and an odd number of columns.

4.2 Deck configuration

A *deck configuration* describes for each value of p, the size of $\mathcal{T}_p$, respecting the rule that $|\mathcal{T}_p| > 0$ and $|\mathcal{T}_p|$ is even. We represent a deck configuration as a set of integers, where the value of element p represents the size of $\mathcal{T}_p$. In the following table, we will show the number of deck configurations for every $n < 12$. Here n is the total number of tiles, DC is the total number of deck configurations with n tiles, and in the column Deck Configurations we show some of them.

n	DC	Deck Configurations
2	1	$\{2\}$
4	2	$\{4\}, \{2, 2\}$
6	3	$\{6\}, \{4, 2\}, \{2, 2, 2\}$
8	5	$\{8\}, \{6, 2\}, \{4, 4\}, \{4, 2, 2\}, \{2, 2, 2, 2\}$
10	7	$\{10\}, \{8, 2\}, \{6, 4\}, \{6, 2, 2\}, \{4, 4, 2\}, \{4, 2, 2, 2\}, \{2, 2, 2, 2, 2\}$
12	11	$\{12\}, \{10, 2\}, \{8, 4\}, \ldots, \{2, 2, 2, 2, 2, 2\}$

In [6], this sequence is recognized as the *partition numbers*. This makes sense since this is actually the same as what we are doing: dividing the n stones into all possible partitions. The formula is as follows: A sequence of positive integers $q = q_1 \ldots q_k$ is a descending partition of the positive integer n if $q_1 + \ldots + q_k = n$ and $q_1 \geq \ldots \geq q_k$. If formally needed, $q_j = 0$ is appended to q for $j > k$. Let Q_n denote the set of these partitions for $n \geq 1$. Then

$$DC(n) = 1 + \sum_{q \in Q_n} \lfloor (q_1 - 1)/(q_2 + 1) \rfloor$$

4.3 Tile layout

A *tile layout* describes the way how the positions in C are mapped to the various sets $\mathcal{T}_p$. Here we distinguish two cases. In the first case we apply the constraint that for each p, it holds that $|\mathcal{T}_p| = 2$. This simplifies our problem, since we can evaluate each arrangement now in quadratic time, as will be shown in Section 5.1. We call this the *easy case*. In the other case we drop this constraint, which results in the fact that the number of arrangements grows, and computational time for most arrangements grows as $|\mathcal{T}_p|$ grows for some p. We call this the *hard case*.

We represent a tile layout as a set of subsets, where each subset contains all positions that are occupied by all tiles from a certain $\mathcal{T}_p$. Because every single tile layout should be compatible with every existing configuration, in this context we prefer to represent each position not as a set of coordinates, but as the index of the position in C_o, which is an ordered set with all elements of C sorted respectively on their i, j and k value. Informally, in an arrangement, we assign the tile with index 1 always to the lowest position in the leftmost column, the tile with index 2 will be assigned to the position on top of the previous tile, if that position exists according to the configuration. Otherwise it will be assigned at the bottom of the next column. This process is repeated, until at last tile with index n will be the assigned to the top-most position in the rightmost column. In Figure 5(a) and 5(d) examples are given, showing how these numbers are assigned.

An important thing to notice is that two tile layouts are considered the same, if we can translate the elements of one of these tile layouts into the other. For example, Figure 5(b) shows an arrangement where $\mathcal{T}_1$ consists of the four a-tiles (at position 0, 2, 5 and 7), $\mathcal{T}_2$ consists of the four b-tiles (at position 1, 3, 9 and 9) and finally $\mathcal{T}_3$ consists of the two c-tiles (at position 4 and 8). Figure 5(c) shows a similar arrangement where $\mathcal{T}_1$ consists of two a-tiles (at position 4 and 8), $\mathcal{T}_2$ consists of four b-tiles (at position 0, 2, 5 and 7) and $\mathcal{T}_3$ consists of four c-tiles (at position 1, 3, 6 and 9). If we were to translate all a-tiles in 5(c) to b-tiles, all b-tiles to c-tiles and all c-tiles to a-tiles, the same arrangement as in 5(b) would arise. Hence we say that these tile layouts are simular.

Just like the configurations we want to know, given a number of tiles, how many tile layouts we can generate. In Appendix B, pseudo code of a recursive algorithm is provided, which works as follows.

First we need to determine the number of tiles we want to consider.

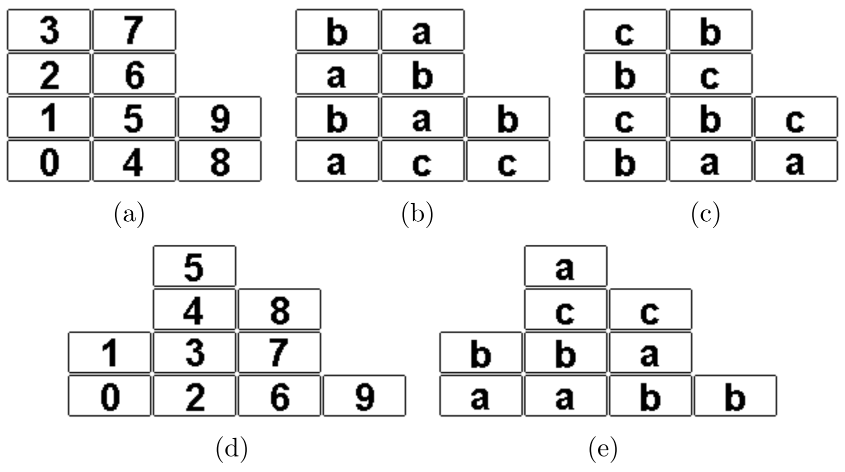


Figure 5: Both (a) and (d) show for every position in a certain configuration, what number will be assigned to the tile at that position in the tile layout. In (b), (c) and (e) it is shown how the tile layout $\{\{0,2,5,7\},\{1,3,6,9\},\{4,8\}\}$ can be implemented in various arrangements.

The recursive function *recCreateAll* needs two parameters, *current_layout*, which is a two dimensional array, representing the tile layout we are creating at this moment, and *tiles_left_over*, which is an array consisting of all tile numbers that are not used yet. We call the recursive function with an empty element provided for *current_layout*, and the array $[1,\ldots,n]$ for *tiles_left_over*. As we go deeper, the array that we provide for *current_layout* will grow, and the one for *tiles_left_over* will shrink. The first thing we are going to check in *recCreateAll*, is whether we already used all tiles. If the size of *tiles_left_over* $= 0$, this is the case. The algorithm adds *current_layout* to the collection of results, and returns from the function. If this is not the case, the algorithm iterates over all possible subsets of *tiles_left_over*, which obey the following rules: i) it includes the first element and ii) it consists of an even number of elements. (The first condition is important, because otherwise we would get the exact same tile layout more than once. The second condition is important, because these subsets are going to represent a $\mathcal{T}_p$. These should always have an even number of elements.)

For every one of these subsets, we make a call to the function *recCreateAll*, but now with the following parameters. For *current_layout*, we will use the same value as the one we already had, but we will add the current subset as an element to this array. For *tiles_left_over*, we also use the same value as we already had, but now without the elements that are in the current subset *comb*.

The number of stone layouts with size n are presented in the next table. Here, n is the number of tiles used and TL_{easy} and TL_{hard} are the number of tile layouts generated respectively in the easy case and the hard case. Note that only the number of tile layouts with an even number of tiles are counted, otherwise the arrangement can never be solvable at all.

n	TL_{easy}	TL_{hard}
0	1	1
2	1	1
4	3	4
6	15	31
8	105	379
10	945	6556
12	10395	150349
14	135135	4373461
16	2027025	156297964
18	34459425	6698486371
20	654729075	337789490599
22	13749310575	19738202807236
24	316234143225	1319703681935929
26	7905853580625	99896787342523081
28	213458046676875	8484301665702298804
30	6190283353629375	802221679220975886631

One thing that draws the attention is that both sequences follow a clear pattern. The sequence emerging from the easy case can be recognized as

$$(n-1)!!$$

Note that this is a double factorial formula, which means that we only multiply all the odd numbers. The proof that this formula is correct, relies on the fact that we will not count tile layouts that can be translated to another tile layout double. We can also represent the tile layout as a string, where each set of tiles is represented by a certain character, and where the positions of that character indicate at what positions the tiles from that set occur in the arrangement. For example the tile layout $\{\{0,5\},\{1,2\},\{3,4\}\}$ can also be

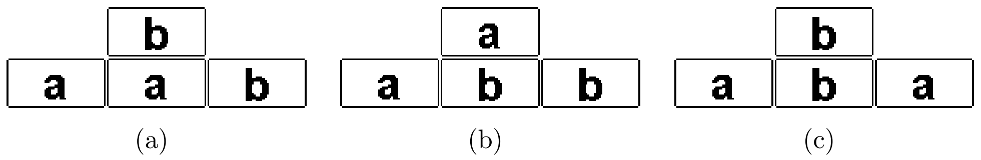


Figure 6: The three arrangements that can be created from configuration $[1, 2, 1]$.

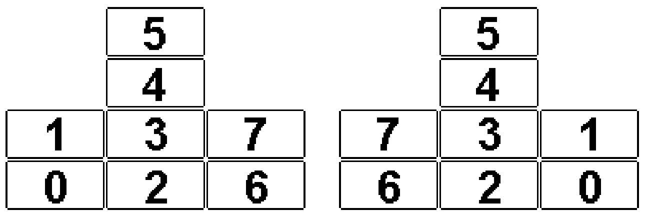
represented as **abbcca**. We can assume that there will always be an a-tile before the first b-tile and the first c-tile, and there will always be a b-tile before the first c-tile, otherwise we can translate the sets in such a way that this will be the case. The first a-tile can only be placed at one position, in fact position 0. For the second a-tile there are $(n-1)$ possible positions. The first b-tile can also be positioned at only one position, the first position that is not occupied yet by the a-tiles. For the second b-tile there are $(n-3)$ possible positions, and so on. This way there are $(n-1) * (n-3) * \ldots * (n-(n-1))$ possible tile layouts, hence $(n-1)!!$.

The hard case can be recognized as the number of partitions of an n-set into even blocks, this is also shown in [7].

4.4 Arrangement

Having both a configuration and a tile layout, an arrangement can be created. An interesting question is, given all possible configurations consisting of n positions and all tile layouts consisting of n tiles, how many unique arrangements can be created. Given the fact that all configurations and deck tile layouts are unique, a tempting assumption would be that the number of unique arrangements is the multiple of the number of configurations and the number of tile layouts. This however is not the case. Consider all arrangements consisting of 4 stones, with the following configuration: $[1, 2, 1]$. The tile layouts would be $\{\{0, 1\}, \{2, 3\}\}$,$\{\{0, 2\}, \{1, 3\}\}$ and $\{\{0, 3\}, \{1, 2\}\}$. These are shown in Figure 6. By mirroring the arrangement in Figure 6(a), the same arrangement as in Figure 6(b) would arise. Therefore, the assumption that the number of unique arrangements is equal to the multiple of the number of configurations and the number of tile layouts is incorrect.

In order to know how many unique arrangements there exist, given a specific number of positions and tiles, one should iterate over all arrangements. If a certain arrangement has a configuration that is not symmetric, there is



(a) (b)

Figure 7: Figure (a) shows the pattern in which a number is assigned to a tile in the tile layout, for configuration $[2, 4, 2]$. In (b) is shown what number is assigned to each tile in the complement tile layout.

no possibility that under the current constraints that are applied to the tile layouts this arrangement can be a duplicate. However, when the configuration is symmetric, we need to determine whether the *complement tile layout* also exists in the collection of all possible tile layouts. If so, this arrangement would be the same as another arrangement, hence we should count it only once. We will now describe how to compute the complement tile layout.

In Section 4.3 it is described in which position a tile would get in a certain arrangement. The pattern is that we start by assigning tiles to the lowest position in the leftmost column and put the next tile on top op the previous tile, until none of the positions in that column are vacant. Then we shift one column to the right and repeat this process. Here both the tiles with index 1 and 2 are element of $\mathcal{T}_p$ for some p, and both tiles 3 and 4 are element of $\mathcal{T}_p$ for some different value of p. The complement tile layout is what you get when having a configuration and a normal tile layout, and translate every tile number in the tile layout to the number that it would have got when the assigning pattern would be from right to left. In Figure 7 it is shown how the numbers from the tile layout can be translated to the complement tile layout. In this example we have configuration $[2, 4, 2]$ and tile layout $\{\{0, 5\}, \{1, 7\}, \{2, 6\}, \{3, 4\}\}$, here the complement tile layout would be $\{\{0, 2\}, \{1, 7\}, \{3, 4\}, \{5, 6\}\}$. Since the tile layout and the complement tile layout are both different from each other and both legal, this arrangement will be counted double, if we were to pair all configurations with all tile layouts. We need to filter these out. The number of arrangements that remain after this filter operation, is shown in the following tables. Here we distinguish once again the easy case from the hard case. In these tables, n is the number of tiles considered, C_{tot} is the number of configurations that can be created with this number of tiles and TL is the number of tile layouts that can be created from this number of tiles. The product of C_{tot} and TL is represented by A_{exp}, which stands for expected arrangements. The actual number of arrangements is represented by A, and finally the number of arrangements that are counted double in A_{exp}, is represented by A_{double}. Note that $A + A_{double} = A_{exp}$.

Easy case

n	C_{tot}	TL	A_{exp}	A	A_{double}
4	6	3	18	17	1
6	17	15	255	225	30
8	45	105	4725	4286	439
10	113	945	106785	99675	7110
12	269	10395	2796255	2669392	126863

Hard case

n	C_{tot}	TL	A_{exp}	A	A_{double}
4	6	4	24	23	1
6	17	31	527	463	64
8	45	379	17055	15431	1624
10	113	6556	740828	690998	49830

We were not yet able to find a pattern or well-known sequence for the numbers of A and A_{double}.

5 Brute force strategies

An interesting question that would arise when having an open instance of Mahjong is whether a winning move sequence exists or not. We will report on two ways to check whether this winning move sequence exists or not.

5.1 Tile pairing

An interesting observation reported by [13] is that for an open instance it is not important in what order the tiles are removed, but rather the choice of *pairings*. (Pairing is when we have a group of tiles, $\mathcal{T}_p$, and we make a move by removing two of these tiles; these tiles are said to be paired. We can also decide to pair two tiles, before we make the actual move.)

The proof of this relies on the obvious fact that removing a pair of tiles does never cause any previously free tile to become blocked. When we have a group of tiles, and we have already decided which tiles will be paired with each other, it does not matter in which order we remove the pairs. For example, if we can choose between removing pair i and pair j, removing pair i will never result in the fact that pair j can not be removed any more and vice versa.

In [13] it is also reported that if we have an arrangement consisting of n tiles, and for all p it holds that $|\mathcal{T}_p| = 2$, it is possible to determine in quadratic time whether a winning move sequence exists or not. An algorithm which can do this is shown in Appendix C.

Based on this information, the author of [13] constructed an algorithm that could answer the boolean question whether a winning move sequence exists for any arbitrary arrangement. This is done by generating all possible tile pairings in advance, and checking whether there exists at least one way to pair these tiles in such a way that the algorithm of Appendix C results in the empty arrangement. In that situation, a winning move sequence exists. There are

$$\prod_{i=1}^{m} \binom{|\mathcal{T}_i|}{2}$$

possible ways to pair all tiles. The pseudo code for this algorithm is shown in Appendix D.

5.2 Dynamic programming

Another way to check whether an instance of Mahjong is solvable or not, is to use a *dynamic programming* algorithm. The idea behind this approach is as follows. In case we want know whether an arrangement of size n is solvable, we check all possible moves that can be performed on this arrangement, and for each of these moves, we play it from the start position. If we have m moves, we now would get m arrangements of size $n - 2$. For all these arrangements, we will check whether this one is solvable in the same way.

With a little optimization, this approach could be a very strong tool for generating all possible arrangements, that can be solved. If we start with all arrangements of size 2, and we store the ones that are solvable (which is in fact just one) than for every arrangement of size 4 we are able to check in quadratic time whether this arrangement is solvable or not. After we have stored all the solvable arrangements of size 4, we could do the same for every solution of size 6, and so on.

The pseudo code for this algorithm is provided in Appendix E. It starts by initiating an associative array in which all arrangements that are solvable can be stored. After that the only arrangement of size 2 that is solvable is stored in this array and the variable *n_done*, which keeps track of the size of the arrangements we've already computed, is initiated and set to 2. We have computed manually all arrangements of size 2, so this is true. After we have computed all arrangements of size 4, this value will be set to 4, and so on. From here on, we can make calls to the function *calculate*.

The first thing this function does, is checking whether arrangements of this size are already calculated. This may seem very trivial, but otherwise the function would not be able to handle a call with $n = 2$. Another reason for this check is when we were to implement this function in such a way that this function could be called multiple times, this small line of code could save us lots of computational power, when we already calculated up to a certain size, and the requested n would be lower than this.

When this is not the case, we start a loop which will first compute all possible arrangements of size $n\_done + 2$, then increments *n_done* with two, and repeats this process as long as *n_done* is smaller than the requested size. Note that we will always increment with two, all arrangements have an even size. Because we will always remove two tiles, this will always be the case.

In this loop, we would like to evaluate all possible arrangements, which are all possible combinations of configurations and tile layouts. As stated in Section 4.4, some duplicates should be left out. In order to keep the pseudo code as clean and clear as possible, this process is not described in detail. For each arrangement a we will check which are the legal moves. Then, for each of these moves we will perform it on a copy of a, and check whether the result was already classified as solvable during the previous iteration. Note that due to the fact that we leave out many arrangements because they are the same by means of reflection, we will also have to check on this. If the arrangement or the reflected arrangement is found, this means we can also solve this arrangement, hence we add it to *solutions*. We do not have to check any further on this arrangement, so we can also break the most inner loop. When the function is done running, all solvable arrangements of size n can be obtained from *solutions*.

6 Experiments

In this section we show the results of our experiments.

6.1 Computing all arrangements

An interesting question is, given all possible arrangements of size n consisting of one cross section, how many of these contain a winning move sequence? We will provide the results of experiments covering arrangements with a small number of tiles. In order to get these results we used the dynamic programming algorithm presented in Section 5.2, and afterwards the tile pairing algorithm reported on in Section 5.1 for validation purposes (up to $n = 10$).

We distinguish the hard case from the easy case. In the results we show the following data. First the number n of tiles we used. In C_{tot} we show the number of configurations consisting of one cross section that exists. In the column DC we show the number of deck configurations consisting of n tiles that exists. Note that in the easy case, this value is by means of the additional constraint always 1. In the column TL we show the number of tile layouts that consist of n tiles.

The column A shows the total number of possible arrangements that can be generated. In the column A_s, we show the number of arrangements for which there is at least one winning move sequence, the column A_b shows the number of arrangements for which no such sequence exists. Note that the sum of the value in solvable and the value in blocked is always the same as the number of arrangements.

Easy case

n	C_{tot}	TL	A	A_s	A_b
2	2	1	2	1	1
4	6	3	17	4	13
6	17	15	225	26	199
8	45	105	4286	224	4062
10	113	945	99675	2335	97340
12	269	10395	2669392	27510	2641882

Hard case

n	C_{tot}	DC	TL	A	A_s	A_b
2	2	1	1	2	1	1
4	6	2	4	23	8	15
6	17	4	31	463	101	362
8	45	8	379	15431	2174	13257
10	113	16	6556	690998	64771	626227

6.2 Classified arrangements

In Section 3.3, a definition was proposed to classify whether an arrangement would be blocked or solvable. As was stated there, this definition is only solid when handling an arrangement where for all p it holds that $|\mathcal{T}_p| = 2$, in all other cases we would have to go deeper into the game tree before we would be able to determine whether an arrangement is blocked or not. As this could be very time consuming, an interesting question could be in what percentage of the cases that are blocked is covered by this definition without going deeper in the game tree.

We obtained these results by checking from all blocked instances whether they also adhere to the blocking definition. The results are presented in the next table. Here, n is the number of tiles in the arrangements, A is the number of arrangements that can be created using n tiles, A_b is the number of blocked arrangements, and A_c is the number of blocked arrangements that are also classified by means of the definition.

n	A	A_b	A_c
4	23	15	15
6	463	362	330
8	15431	13257	11753

The results show clearly that the more tiles are used, the higher the percentage of arrangements that are not correctly classified becomes.

7 Conclusion and future work

The main purpose of this research was to gain information about what percentage of arrangements are solvable. We have provided definitions of several aspects of the game Mahjong Solitaire, and most of them along with an already known number sequence such that it is easy to calculate for a certain number of tiles, how many different arrangements there actually exist.

Getting information about what percentage of these arrangements are solvable, is a bit harder. We provided a dynamic programming algorithm that can do so relatively fast, at the cost of a high number of space resources. We also have reported on the algorithm that was proposed by [13], which can be used for this purpose by evaluating every arrangement individually.

The results that we obtained regarding the percentage of solvable arrangements are at first sight not very significant. The dynamic programming algorithm however, is perfectly usable for arrangements consisting of an arbitrary number of tiles and cross sections, provided that enough computation power is available.

As a recommendation for future work, one issue that seems very promising is the blocking definition. At the moment we can only be sure of its correct working if we are dealing with the case that for all p it holds that $|\mathcal{T}_p| = 2$, otherwise we will have to go deeper into the game tree as described in Section 3.3. If the blocking definition would be extended in such a way that more (or eventually all) patterns of blocked arrangements could be detected, without the mentioned restriction, it would give rise to various possibilities for answering the solvable arrangements question.

If someone were to find a recurrence relation for the number of arrangements given n tiles, or even the number of solvable arrangements, this would be a mayor achievement.

Acknowledgements

I would like to thank W.A. Kosters, J. de Ruiter, M.A. Swenne and L. van Rijn-Parlevliet for their useful comments, the interesting discussions and other comments they provided.

A Generating configurations

This appendix provides the pseudo code for implementing an algorithm that can generate all configurations until a certain number.

```
// maximum size of configurations we want to compute.
```

```
int n = 14;
// array containing all configurations as a array<int>
array<array<int>> allConfigs;
// initial call to function
allConfigurations( new array<int>, 0, false );

void allConfigurations(array<int> currentConfig, int last, bool
    desc){
  if(sum(currentConfig)==n){
    // ignore potential reflections:
    array<int> reversed = reverseArray(currentConfig);
    if(currentConfig<=reversed){
      allConfigs.add(currentConfig);}

    // exit loop when target size is reached
    return;
  }

  // generate versions with an extra column
  for(int i=1;sum(currentConfig)+i<=n;i++){
    if(desc&&i>last)break;
    array<int> copyOfConfig = config;
    copyOfConfig.add( i );
    bool direction=desc;
    if(i<last) direction=true;
    allConfigurations(copyOfConfig,i,direction);
  }
}
```

B Generating tile layouts

This appendix provides the pseudo code for implementing an algorithm that can generate all tile layouts until a certain number.

```
int n = 12;
array results = new array();
array tiles = { 1,..., n };
recCreateAll( {∅} , tiles );

void recCreateAll( int[][] currentLayout, int[] tiles_left_over
    ){
  if(|tiles_left_over| > 0){
    foreach ( comb ⊆ tiles_left_over ){
      if( |comb|%2 == 0 && tiles_left_over[0] ∈ comb ){
        currentLayout[] = comb;
        recCreateAll( currentLayout, leftOverCopy-comb );
      }
    }
  } else {
    results[] = current;
  }
}
```

C Easy case solve algorithm

This appendix provides the pseudo code for implementing an algorithm that solves any instance of Mahjong where holds that for all p it holds that $|\mathcal{T}_p| = 2$.

```
function solvableEasy( array<int> configuration, array<array<int
    >> tile_layout ) {
  array<Tile> tiles;
  array<array<Tile>> tilesPerType;
  int done = 0;

  // initialize the tiles
  for( i = 1 until i = configuration.length ){
    for( j = 1 until j = configuration.get(i) ){
      type = determine type of tile #done;
      Tile current = new Tile( i, j, type,
        (configuration.get(i)-1==j),
        (i==1||configuration.get(i-1)<=j),
        (i==configuration.size() ||
          configuration.get(i+1)<=j)
        );
      tiles.add( current );
      tilesPerType.get(type).add( current );
      done++;
    }
  }

  int tilesLeft = tiles.size();
  while( removeTiles() ){ }
  return (tilesLeft == 0) ? true : false;
}

bool removeTiles(){
  bool removedTiles = false;
  foreach( tilePair in tilesPerType ){
    if( tilePair[0].playable == true &&
        tilePair[1].playable == true ){
      foreach( currentTile in tilePair ){
        if( ∃tile in tiles with x = currentTile.x and y =
            currentTile.y−1 )tile.topFree = true;
        if( ∃tile in tiles with x = currentTile.x+1 and y =
            currentTile.y )tile.leftFree = true;
        if( ∃tile in tiles with x = currentTile.x−1 and y =
            currentTile.y )tile.rightFree = true;
        tilesLeft−−;
      }
      removedTiles = true;
    }
  }
  return removedTiles;
}

class Tile {
  int x,y,type;
  bool leftFree,rightFree,topFree;
  constr Tile(x,y,type,topFree,leftFree,rightFree){
    set global.x = x, global.y = y;
    set global.topFree = topFree;
    set global.leftFree = leftFree;
    set global.rightFree = rightFree;
  }
  bool playable(){
    return (topFree&&(leftFree||rightFree));
  }
}
```

D Tile pairing algorithm

This appendix provides the pseudo code for implementing an algorithm that solves an open instance of Mahjong using the tile pairing algorithm, provided by [13]. This algorithm relies on the function described in Appendix C.

```
array<array<array<int>>> paired_tile_layouts;

function solvable( array<int> configuration, array<array<int>>
    tile_layout ) {
  generate( tile_layout, 0 );

  foreach( paired_tile_layout in paired_tile_layouts ){
    if( solvableEasy( configuration, paired_tile_layout ) ==
        true ){
      return true;
    }
  }
  return false;
}

private void make( array<array<int>> tileLayoutRest, array<array
    <int>> created ){
  if( tileLayoutRest.size() == 0 ){
    if( paired_tile_layouts.contains( created ) == false )
      paired_tile_layouts.add( created );
  } else {
    for( int i = 0; i < tileLayoutRest.get(0).size()-1; i++ ){
      for( int j = i+1; j < tileLayoutRest.get(0).size(); j++ ){
        created.add( tileLayoutRest.get(0).get(i) );
        created.add( tileLayoutRest.get(0).get(j) );

        tileLayoutRest.get(0).remove(j);
        tileLayoutRest.get(0).remove(i);

        make( tileLayoutRest, created );
      }
    }
  }
}
```

E Dynamic programming algorithm

This appendix provides the pseudo code for implementing an algorithm that generates all arrangements of size n that are solvable, using dynamic programming. The class `Arrangement` is initiated using a constructor consisting of a configuration as first parameter and a tile layout as second parameter.

```
// create storage for all solvable solutions, any size
hashmap<integer, array<Arrangement>> solutions;
// add all possible solutions for n = 2
solutions.get(2).add( new Arrangement( [1,1], {{0,1}} ) );
// keep track on which n we solved last
int n_done = 2;

```

```
calculate( 20 );

function calculate( int n ):void{
  if( n >= n_done ) return;
  for( ; n_done < n; n_done+=2){
    array<array<int>> configurations = allConfigurations( n_done
        +2, 0, false );
    array<array<array<int>>> tile_layouts = recCreateAll( { 1,...,
         n_done+2 }, ∅ );
    array<Arrangement> smaller_solutions = solutions.get( n_done
         );
    iterate over all unique combinations of( configurations,
        tile_layouts ){
      array legalMoves = all legal moves on this arrangement
      foreach( legalMove as move ){
        Arrangement current = new arrangement( configuraion,
            tile_layout );
        current.doMove( move );
        if( solution.get( n_done ).contains( current ) ){
          solutions.get( n_done+2 ).add( new arrangement(
              configuraion, tile_layout ) );
          break;
        }
      }
    }
  }
}
```

References

[1] A. Condon, J. Feigenbaum, C. Lund and P. Shor,
Random debaters and the hardness of approximating stochastic functions,
SIAM Journal on Computing, Vol. 26, No. 2, 1997, pages 369–400

[2] G. Kendall, A. Parkes and K. Spoerer,
A Survey of NP-complete Puzzles,
International Computer Games Association Journal (ICGA), 2008, 31(1), 13–34

[3] The OEIS Foundation,
The On-Line Encyclopedia of Integer Sequences

http://oeis.org/
[retrieved March 4, 2011]

[4] The OEIS Foundation,
Number of stacks, or planar partitions of n
http://oeis.org/A001523
[retrieved March 4, 2011]

[5] The OEIS Foundation,
Number of palindromic and unimodal compositions of n
http://oeis.org/A096441
[retrieved March 4, 2011]

[6] The OEIS Foundation,
The Partition Numbers
http://oeis.org/A000041
[retrieved March 4, 2011]

[7] The OEIS Foundation,
Number of partitions of an n-set into even blocks
http://oeis.org/A005046
[retrieved March 4, 2011]

[8] F. C. Auluck,
On some new types of partitions associated with generalized Ferrers graphs,
Proc. Cambridge Philos. Soc. 47, (1951), 679–686

[9] K. Baur and N. Wallach,
Nice parabolic subalgebras of reductive Lie algebras,
Represent. Theory 9 (2005), 1–29.

[10] The Wikimedia Foundation,
Mahjong solitaire,
http://en.wikipedia.org/wiki/Mahjong_solitaire
[retrieved February 28, 2011]

[11] D. Eppstein,
Computational Complexity of Games and Puzzles,
http://www.ics.uci.edu/~eppstein/cgt/hard.html#shang
[retrieved February 28, 2011]

[12] J. Elonen,
There is no deterministic way to solve a mahjongg solitaire game,
`http://elonen.iki.fi/code/misc-notes/no-alg-mahj-solit/`
[retrieved February 28, 2011]

[13] P. Gimeno,
Mahjongg Solitaire Solver,
`http://www.formauri.es/personal/pgimeno/mj/mjsol.html`
[retrieved February 28, 2011]

[14] Rubl.com Games,
Mahjong Solitaire online,
`http://www.rubl.com/games/mahjong/`
[retrieved February 28, 2011]