Martello - AI Summer Project - Daniel Graf

Supervised by Prof. Luke Zettlemoyer, University of Washington, June - August 2013

1. Introduction

After taking the class 'Introduction to Artificial Intelligence' during my half-year exchange in spring quarter 2013 at UW CSE, I wanted to apply the learned techniques in a small summer project. So I recalled the board game *Martello* that I received as a gift a few years ago. *Martello* is a combinatorial one-person board game. It was invented by the Swiss mathematician Peter Hammer in the 1980s. One can roughly think of it as a combination of peg solitaire and Uno, making it an intricate puzzle where the player has to find the right sequence of moves in order to eliminate all but one stone.

First, I will present the rules of the game and then present the analysis I made. The exponentially large state space makes it also an interesting problem for finding algorithmic strategies. I implemented a DFS-solver and then custom tailored it to the problem at hand using backtracking techniques and randomization. Finally, I embedded this solver in an iOS application in order to visualize the found solutions and in order to play the game while looking at hints from the solver. I end with a short overview over the code base.

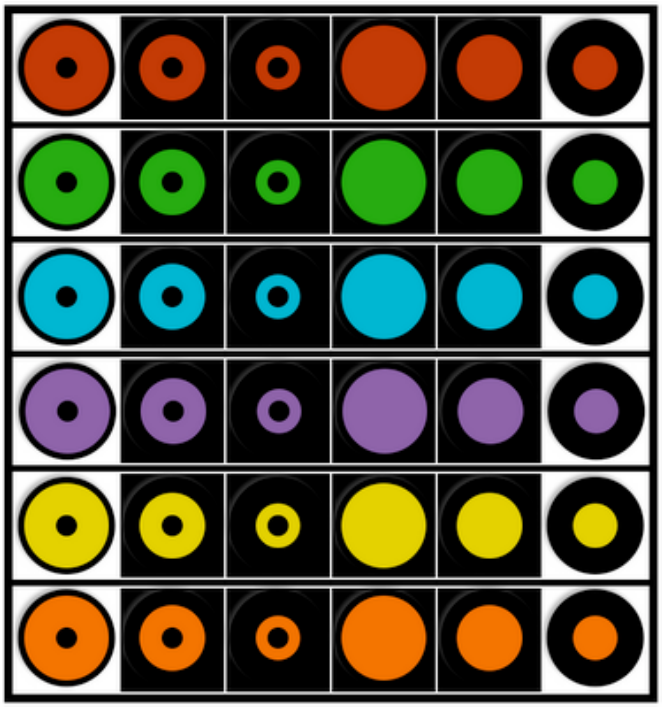


Figure 1: The 6-by-6 Martello board with all 36 stones.

2. Game

Idea

On a square board of 36 stones, one tries to eliminate one stone after the other.

There are six different colors and six different symbols (dots and rings in three sizes each) and there is one stone for each combination of color and symbol.

A stone can "eat" another stone if they show the same color or the same symbol.

Allowed Moves

At the start all 36 stones are shuffled and randomly placed on the 36 spots on the board. The board has 6 rows and 6 columns, with the first and last column having a special property and therefore being marked in white. The goal of the game is to remove as many stones as possible from the board. A stone can be removed by placing a stone on top of one of its neighboring stones. The subjacent stone gets removed.

Removal of Stones

Neighboring stones can "eat" each other only if the stones either share the same symbol or the same color. Next, we have to specify the neighboring relation precisely.

Horizontal Neighbors: Two stones are horizontal neighbors if they are in the same row of the board and no other stone is placed between them.

Vertical Neighbors: Two stones are vertical neighbors if they both lie in the white zone (first or last column) and the two cells are direct neighbors.

Horizontal Movement

The stones can be moved horizontally without eating other stones at any point in the game as long as the following conditions are met: Stones can only move over free spots into neighboring fields on the left or on the right. It is not allowed to jump over other stones when moving horizontally. The outermost stones of each row have to stay in the white zone at all times. So if a spot in the white zone opens up, the next stone in the same row has to be moved there. Only if there are less than two stones left in a row, it is ok for a spot in the white zone not to be occupied. If there is only one stone left it can be moved to the left or right border column at will.

Again, leaping over other stones or moving stones vertically without removing stones is not allowed and you have to make sure the white zone is always filled as far as possible.

Double Jump

As soon as only one stone in a row is left, this remaining stone can only cover one of the two spots in the white zone and a gap opens up on the other side. In a 'double jump' two stones are also considered vertical neighbors if there is exactly one empty white field in between. Note that you can move a single stone in a row horizontally to decide on which side this gap should be. This vertical jump is limited to leaping over one empty spot. Jumping any further is not allowed.

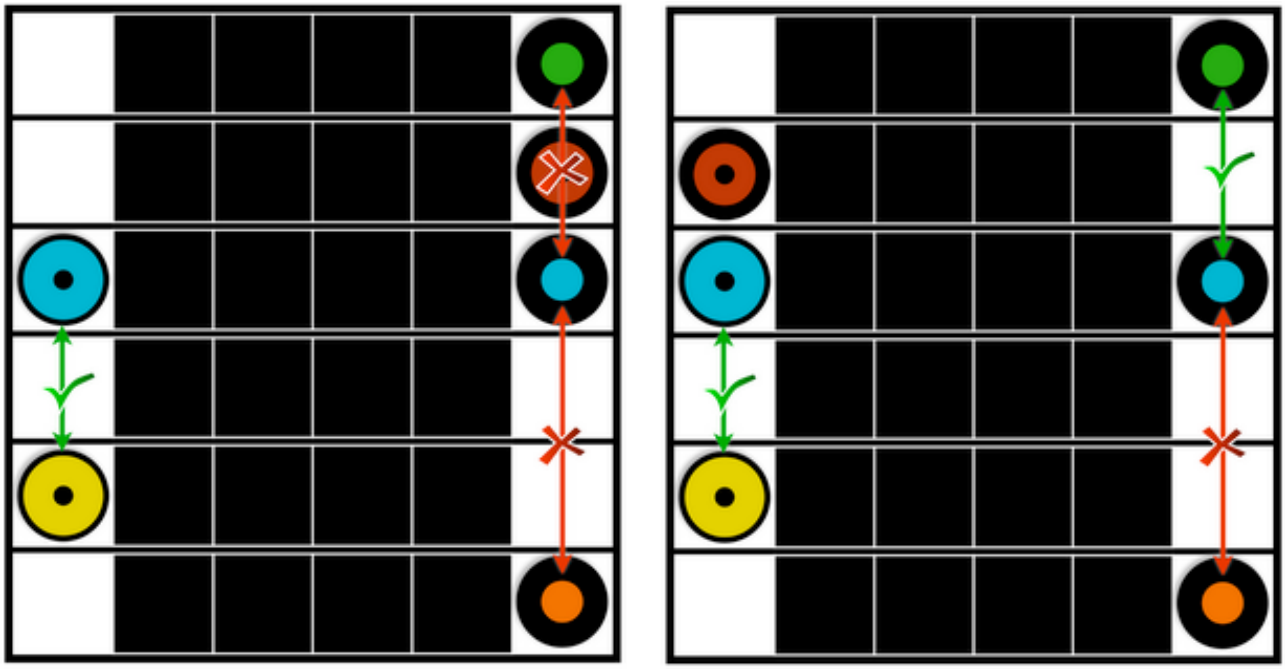


Figure 2: Admissible and inadmissible double jumps. Note that after moving the red stone to the right, another double jump in the upper right corner is valid.

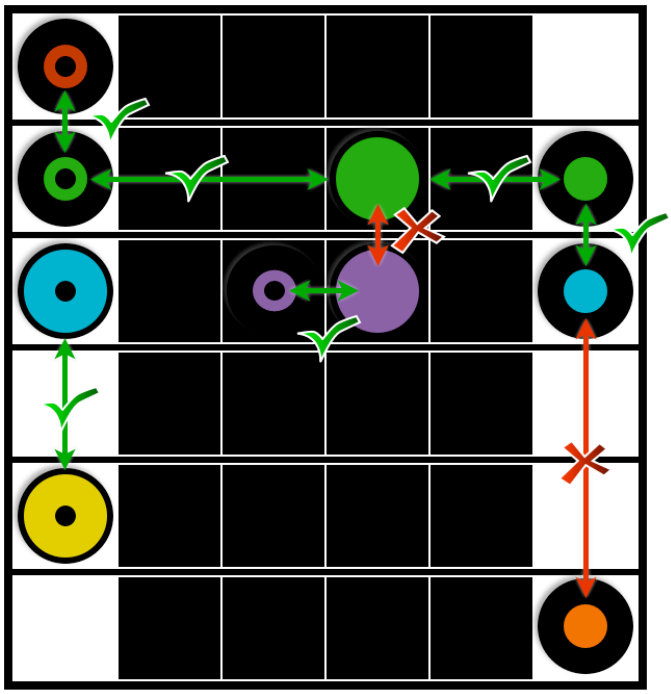


Figure 3: The image below illustrates admissible and inadmissible moves. Note how the double jump over one empty white field is ok, but not over two.

This is the complete set of allowed moves in the single player variant of Martello. There are also two-player variants of the game called 'martello duo', 'martello colore' and 'martello rondo'. In this project I only considered ‘martello solo’.

Terminology

We say the game is *won* if the player manages to remove all but one stone from the board. The word *state* denotes any configuration of stones on the board, also after some *moves* have already been performed. A *start permutation* (or short a *game*) is *solvable*, if there exists a sequence of moves that can be used to win the game. If there exists no such sequence, we call the game *unsolvable*.

In any case, we denote with the *minimum stone count*, the smallest number of stones that are left after any sequence of valid moves. So for solvable games, the minimum stone count is 1 and unsolvable games it is somewhere in [2, 36].

Generalization to n-by-n board

This set of rules easily transfers to an n-by-n board with n^2 colors and symbols. For any $n \geq 2$, the first and last column are marked as the special white column. A game on an 4-by-4 board could therefore look as follows:

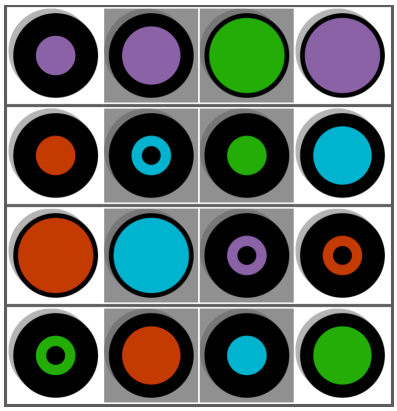


Figure 4: A 4-by-4 board with a random start configuration.

3. Analysis

In this section I will present the results I got from analyzing various properties of *Martello*. All of them were achieved using the solver, whose construction is presented in Section 4. For now you might just think of it as a black box that takes the state of a game and returns its solvability, the minimum stone count and a sequence of moves achieving that.

Solvability

Up front by just looking at the rules, it is not clear at all, why it should even be possible to solve any game. In fact, by playing the game myself I hardly ever managed to solve it and usually got stuck with a few stones left. However, sometimes I managed to eliminate all but one stone, so it can not be impossible either. By generating about 3 million random permutations and by solving each of these start configurations individually, I got a more precise idea of the solvability of *Martello*.

In *98.35%* of all random start configurations the game is in fact solvable, so it is possible to remove 35 of 36 stones. Also very interesting is the distribution of the minimum stone count within the remaining *1.65%* of unsolvable games. Given, that the game is unsolvable it is most likely that only at most 6 to 14 stones can be removed before the player gets stuck. The distribution has another peak at a minimum stone count of 2, which means to get stuck right before the end.

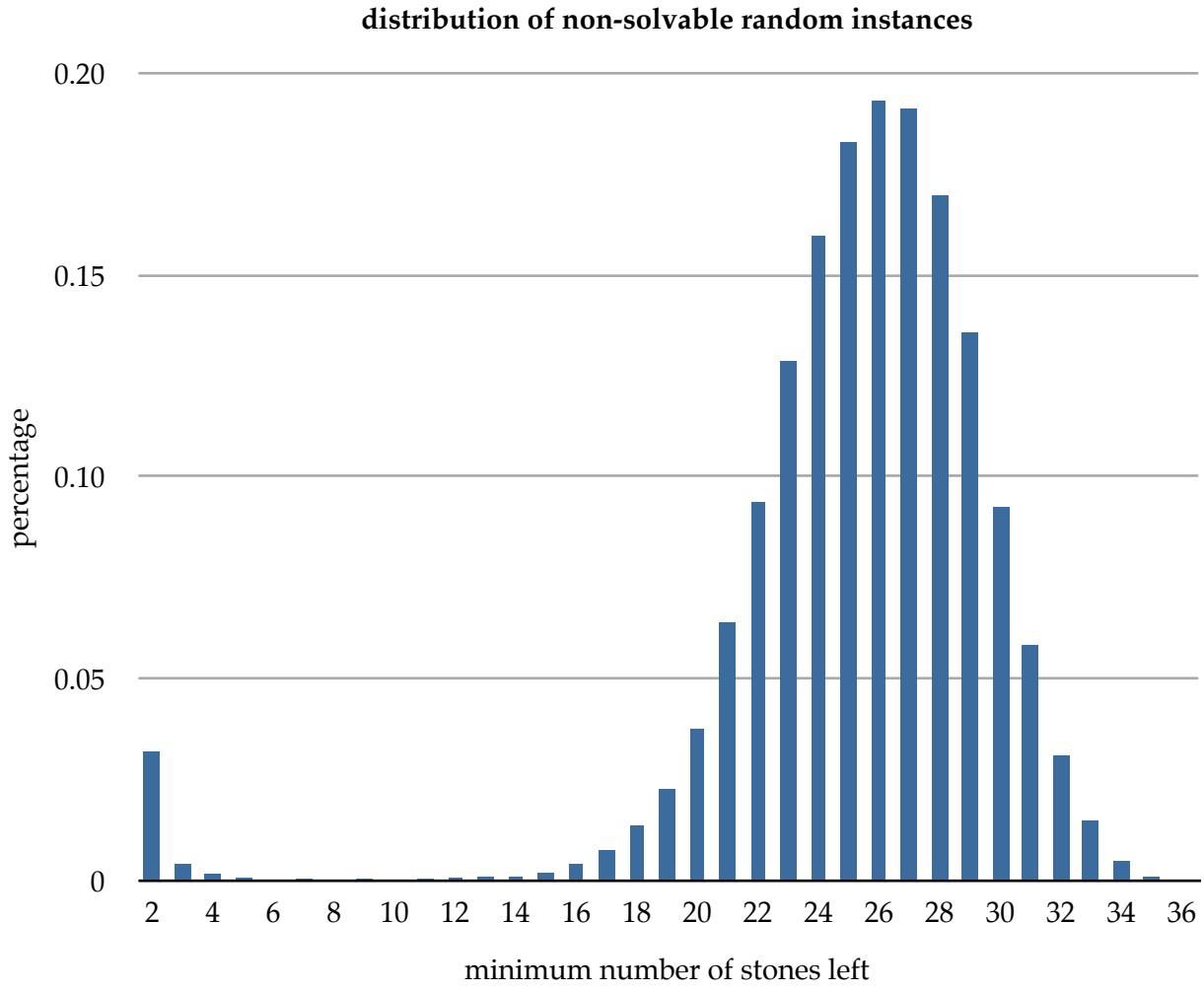


Figure 5: Minimum stone count distribution within the 1.65% of unsolvable games

Every single number between 1 and 36 has non-zero probability of being the min stone count of a random game. For instance in only 2 of the 2918608 analyzed games the minimum stone count was 36, so no move was possible from the very beginning. Figure 6 shows one such start configuration.

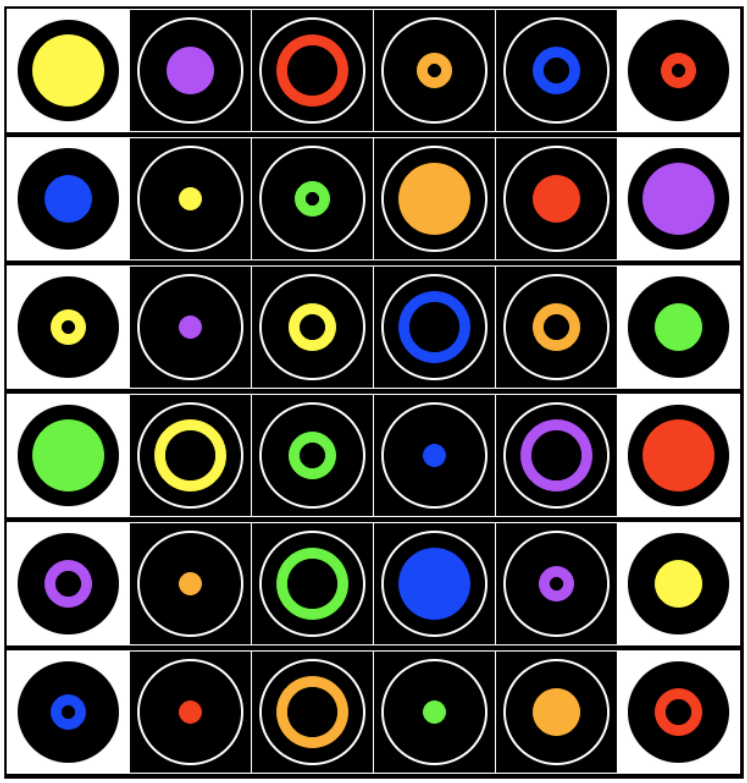
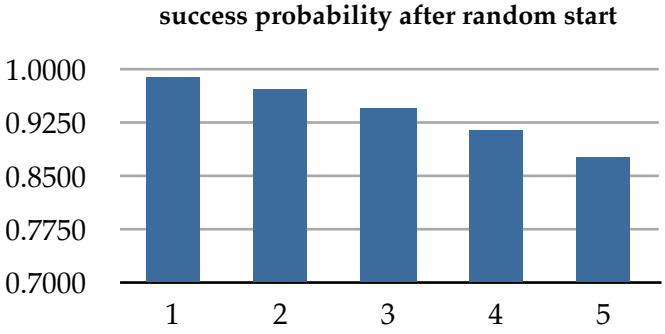


Figure 6: Start configuration with minimum stone count of 36.

Randomized Starts

Given, that a human player can hardly think 35 steps ahead, the first few moves of each game are most likely to be played according to some heuristic or just randomly. This poses the question whether the first few moves matter at all or if any solvable game is likely to be still solvable after a few random moves. I generated several thousand solvable games and then for each of them applied a sequence of l moves, picked uniformly at random out of all allowed moves, and then applied the solver to the resulting state. Figure 7 shows the percentages of a random starting configuration still being solvable after applying the sequence of random moves.



l	probability overall	for the additional l-th move
1	0.9887	0.9887
2	0.9715	0.9826
3	0.9447	0.9724
4	0.9141	0.9676
5	0.8753	0.9576

Figure 7: Percentage of solvable games after l random moves at the beginning

The percentage of good choices is surprisingly high: 98.8% of all options in the first move are good. Even after a start with five random moves the game is still solvable with a probability of 87.5%.

Board sizes

By generalizing the original game to an n-by-n board, I was hoping to get more insights into the structure and the solvability of *Martello*. As my solver takes several seconds to handle some of the harder 6-by-6 instances, I only looked at boards of at most that size. Obviously the only 1-by-1 and all 2-by-2 games are solvable. More interestingly, for all board sizes from 3-by-3 to 6-by-6 there are a variety of possible minimum stone counts including 1 and n^2. Figure 8 lists the empirically found solvability probabilities for random n-by-n games. Generally the bigger n, the more likely a random game is unsolvable. But still, for any $n>2$ the random sample contained games, that were stuck from the beginning, so where the minimum stone count was n^2.

n	1	2	3	4	5	6
ratio	1.00000	1.00000	0.99596	0.99767	0.99393	0.98349

Figure 8: solvability ratio for random games of various size.

4. Solution Strategies

In this Section I describe how I constructed and extended a DFS-based solver for *Martello*. First we discuss some essential observations about the problem that were needed to make the solver feasible. After looking at runtime results and the analysis from the previous Section, I describe a randomized algorithm that allowed me to reduce the average runtime by 29.5%.

DFS Preparations

I did not use the set of board configurations as the set of states for this search problem but an equivalence class on them. Each equivalence class is represented by a so called *normalized* state. A state is normalized whenever all stones in it are moved as far to the left as possible, so whenever no more horizontal moves to the left without eating other stones can be made. For any state its unique normalized state can be found by just repeatedly performing any non-eating horizontal move to the left. Obviously, normalization does not change the solvability of a game, but can help us immensely to reduce the number of states that need to be considered. Figure 9 gives an example for such a normalization:

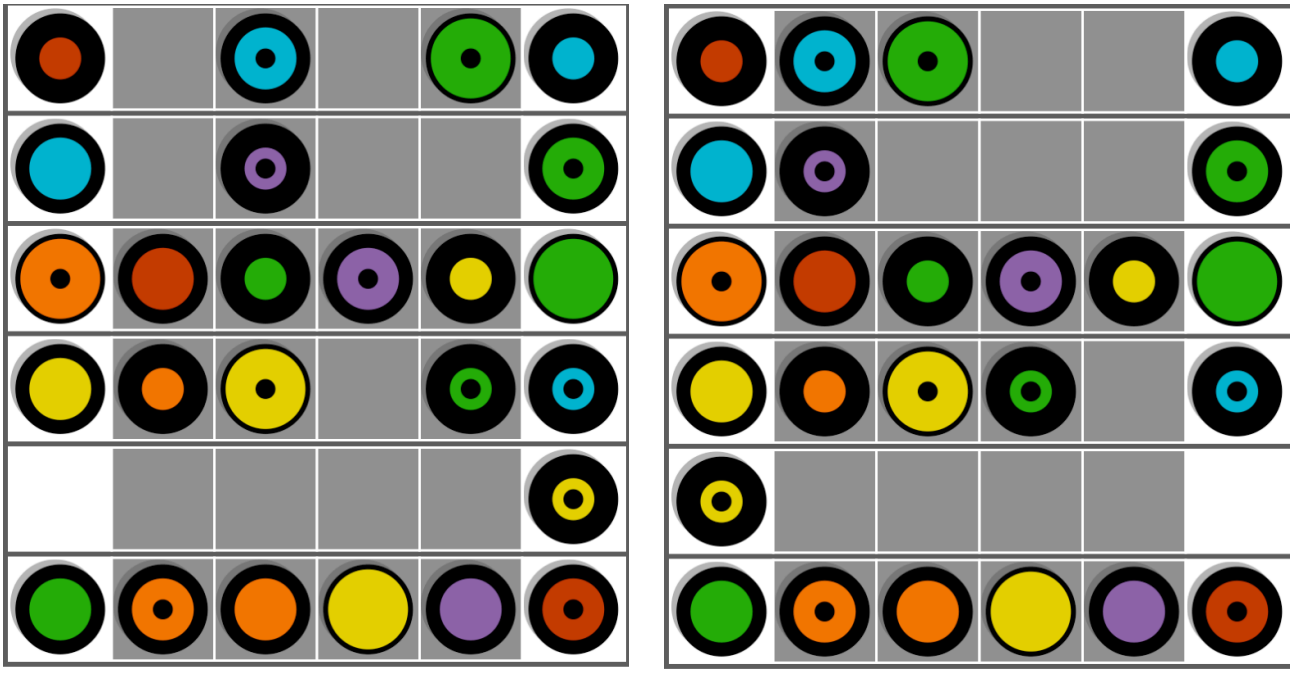


Figure 9: A state within a game, before and after normalization

On this space of normalized states we now define *operations* as a transition from one normalized state to another by eliminating exactly one stone. Such an operation must correspond to a sequence of regular *Martello*-moves: first none or some non-eating horizontal moves, than exactly one eating move and finally again none or some non-eating horizontal moves for the *renormalization*. Figure 10 illustrates these three phases.

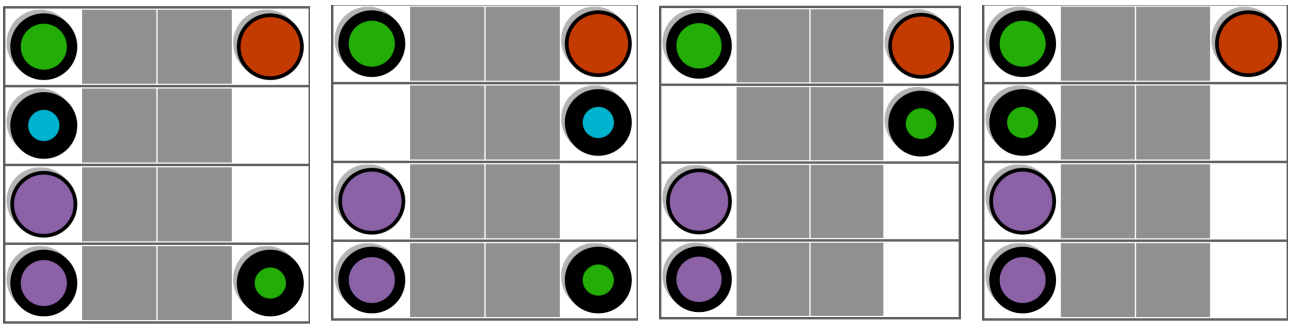


Figure 10: Starting with a normalized state, we move a stone horizontally before performing an eating move. Afterwards we move a stone horizontally again in order to end up with a normalized state.

Note that when just considering any valid sequence of moves, this sequence might be of infinite length, as it is allowed to move a stone horizontally back and forth without ever eating another stone. But now we can use the fact that each operation eliminates one stone to bound the depth of the search graph to exactly 35 operations.

The number of possible sequences of operations grows exponentially fast with n, because as long as the game is not stuck, there are always at least two valid moves due to the symmetry of 'eating' and 'getting eaten'. So distinguishing between games with a minimum stone count of one or two by simply enumerating all valid sequences of operations is not feasible. But we can make use of the fact that the game is *memoryless*, meaning that only the current state of the board determines all possible moves and it does not matter which moves lead to this state. Similar to the *Markov property* of *stochastic processes*. Therefore we should extend the depth first search with *memoization*. We maintain a set of all explored states to make sure we follow each bad path only once.

Representing and finding (partial) solutions

While we can just use a C-array-representation of the board as the data type for representing the states in this memoization set, we also want some handier representation of states for storying them in a database, copying them into a visualization etc. For this I use a simple string of 36 characters where I use the 26 lower case letters and the first 10 upper case letters to represent the different stones. Sequences of states that represent a solution path are encoded the same way, separated with an 'X' between each state.

As soon as the depth first search finds a sequence to solve the game, the search stops and reports the current states on the search stack as a valid solution. If the game is unsolvable the DFS has to explore all reachable states and then return any list of operations that achieves the minimum stone count. One way to implement this would be to continuously maintain a list of the currently explored path (in parallel to the call stack of a recursive DFS implementation) and then create a copy of that list whenever the currently best known minimum stone count decreases. I opted for a simpler implementation that only keeps track of the minimum stone count. Once all states are explored and it is clear that no solution exists, I run the depth first search again and stop as soon as a path for that minimum stone count is found. This keeps the DFS-implementation simple and fast, but might lead to every state being explored twice. But since there are often many ways to reach the minimum stone count, the second DFS run is likely to be very short. Also keep in mind that this only matters for the *1.65%* of unsolvable games.

DFS work distribution results

This well-defined depth first search algorithm then allowed me to solve any *Martello* instance. However, it became clear quickly, that some states are much harder to analyze than others. My implementation could explore about 170’000 states per second on a single thread on a 2.0 GHz Intel Core Duo CPU. For many games the solution was found almost instantly, but some games took several seconds to solve. Figure 11 shows the average number of state explorations needed for different minimum stone counts.

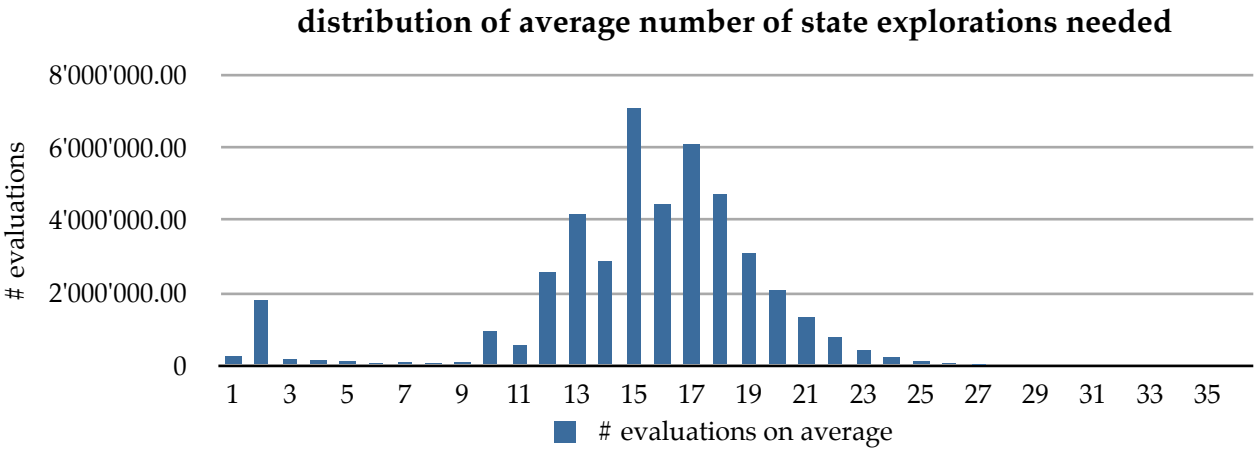


Figure 11: The distribution of necessary evaluations is clearly dominated by the non-solvable games. This because it is necessary to evaluate every reachable state of unsolvable games. For solvable games the search can stop as soon as a solution is found and so after an average of about 230’000 state explorations the search can stop. It is interesting, that games with minimum stone count of 2 are significantly harder to analyze than such with a minimum stone count of 3 or 4. Also shows that games with a minimum stone count of 12 to 20 allow the most amount of different states to be reached.

As all the unsolvable instances are very rare, it makes sense to also look at the distribution of work within the class of solvable games, as shown in Figure 12. Note that most of the games could be solved within 100’000 explorations and almost all games fall below the average of 230’000 explorations. This can be explained by the significant amount of games which need more than a million explorations.

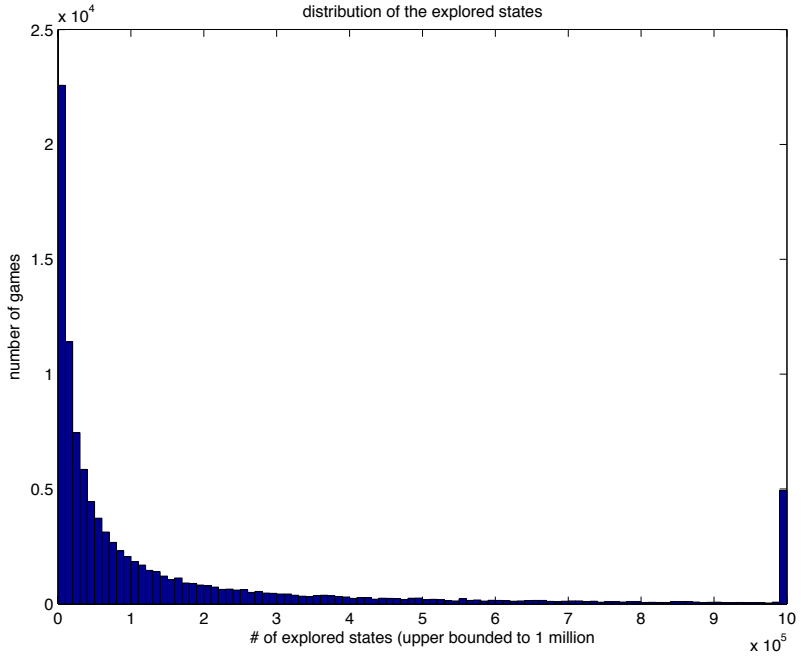


Figure 12: Histogram of the number of state explorations needed for solvable games only.

Exploration Order Heuristics

Figure 12 suggests that it might be worthwhile to try different exploration orders, so apply some heuristics that tell us which of the possible moves should be tried first. So far, the order that the possible moves get tried out in, is the one from the definition of the rules in Section 2, so first all horizontal moves, then all vertical moves and finally all double jumps. In an attempt to improve on this I came up with and implemented the following heuristics:

Most-options-first Heuristic

In order not to get stuck too soon, it is reasonable to try to keep as many options available as possible. Therefore we explore that move first whose resulting state has itself the most moves available, meaning that we order the available paths in the search tree according to the branching factor of the neighboring nodes. This is similar to the *least constraining value* technique that we saw for *constraint satisfaction problems.*

Monte Carlo Heuristic

The branching factor might be a very bad indicator of whether we are on the right path towards the solution. Looking for a fast way to approximate the minimum stone count of a state, I implemented the following heuristic: Given a state to evaluate, we perform a random sequence of moves until we get stuck, which gives us an upper bound to the minimum stone count.

If we are lucky we might accidentally solve the game. And if not we might at least get closer when starting in a solvable state then when starting in an unsolvable state. A single random run does not tell us much, so we repeat this 100 times and take the minimum of all runs to get a tighter approximation of the minimum stone count. The nodes get expanded in the order of this estimate. The ones who are more promising to solve randomly get explored first.

Since evaluating this heuristic is considerably more expensive than just checking the branching factor, I only activated it for the first three levels of the search tree.

Random Heuristic

As a comparison, I also added a heuristic that simply explores the possible moves in a random order. As the resulting depth first search is no longer deterministic, I run this search 10 times on each game and then took the average as well as the minimum number of state explorations. Surprisingly the minimum was about 19.7 times smaller than the average, which gave me the idea for the technique explained in the next subsection.

Results

Unfortunately none of the heuristics improved the search time. The most-options-first heuristic even performed significantly worse than using the standard expansion order. The fact that the average for the standard exploration order with about 219’000 is lower than the average of 230’000 in Figure 11 is puzzling me, as it was the same setup. Maybe the sample size of 5200 games for Figure 13 was too small compared to the 2’800’000 samples in Figure 11.

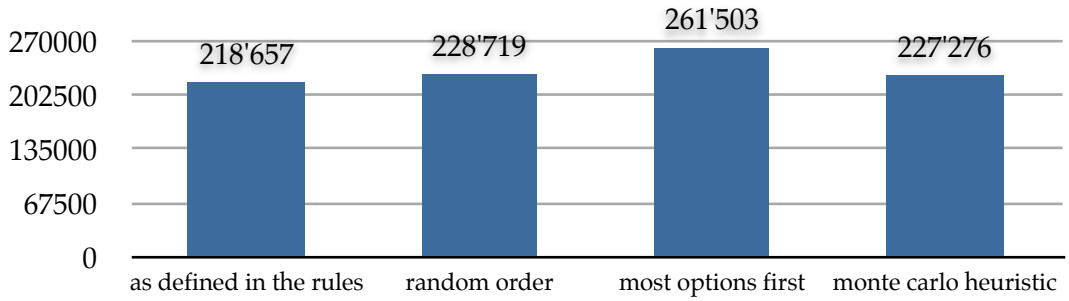


Figure 13: Comparison of the average number of expansions needed for finding a solution of a solvable game.

Randomized Restart Solver

It became clearer why the heuristics did not work once I analyzed the properties of the randomized starts, as described in the Section 3. The first few moves simply do not matter much. I also saw that the number of expansions needed after such a random start varied heavily. A lucky start might make the difference between a few hundred or several thousand expansions, even for the same game. To investigate that I ran the solver with a limited budget of state expansions and counted how often a solution was found, depending on the size of this limit and the number of random moves in the beginning. The results of this experiment can be seen in Figure 14.

depth \ expansions	100	250	500	1000	2500	5000	10000
1	0.60087	0.88466	0.95365	0.9795	0.99281	0.99647	0.99811
2	0.60995	0.88554	0.95516	0.98095	0.99313	0.99653	0.99798
3	0.61186	0.88767	0.95251	0.97893	0.99231	0.99596	0.99773
4	0.61352	0.88588	0.95342	0.97904	0.99217	0.99596	0.99779
5	0.62547	0.8882	0.9532	0.97838	0.99182	0.99537	0.99753

Figure 14: Success rate of finishing a solvable game in at least 1 of 1000 tries with limited expansions. So in roughly 40% of all solvable games I have to assume that the probability of finding a solution within 100 expansions is negligible.

All of these experiments were made using the random heuristic. We see that the depth does not matter. This makes sense, as using the random heuristic from the root is equivalent to using it after 3 random moves as long as only one branch on the third level can be explored within the budget. The increase from 0.60087 to 0.62547 in the first column can be explained with the fact that the depth gives a few additional state expansions, so it increases from 101 to 105 expansions with bigger depths. With 1000 attempts using 1000 expansions each, *98%* of all solvable games can be solved.

This success probability of each combination of depth and budget allowed me to use the expectation of a geometric distribution in order to estimate the expected number of expansions until the first success, as in Figure 15.

depth \ expansions	100	250	500	1000	2500	5000	10000
1	56252	64246	67118	69726	77072	85349	95257
2	56194	63127	68187	71627	78053	86860	98014
3	55555	63689	67529	71430	78632	89604	1.01E+05
4	55118	64006	68239	72097	82064	94503	1.10E+05
5	55040	63804	68767	73507	84556	98278	1.18E+05

Figure 15: Expected number of expansions until a successful run under the condition that at least 1 of the 1000 attempts were successful (see Figure 14). So while the entries in the first column with roughly 55'000 are very low, they are also the most unlikely to occur with only 61% probability.

Finally we are interested in the overall work needed, so including the amount of expansions needed for solving it precisely whenever the budgeted, randomized attempts do not work. This additional effort is non-trivial to bound. We can not simply take the overall average, as this conditional expectation is only over some of the hardest games. I used 4 million as an estimate as this was the average of the 2% hardest games in my database of analyzed games. As figure 16 shows, I could hope for roughly 150'000 expansions on average using such a limit-and-repeat solver with a budget of 1'000 expansions in each run.

depth \ expansions	100	250	500	1000	2500	5000	10000
1	1.65E+06	5.26E+05	2.53E+05	1.52E+05	1.06E+05	99475	1.03E+05
2	1.62E+06	5.21E+05	2.48E+05	1.48E+05	1.06E+05	1.01E+05	1.06E+05
3	1.61E+06	5.13E+05	2.58E+05	1.56E+05	1.09E+05	1.06E+05	1.11E+05
4	1.60E+06	5.20E+05	2.55E+05	1.56E+05	1.13E+05	1.11E+05	1.19E+05
5	1.55E+06	5.11E+05	2.56E+05	1.60E+05	1.17E+05	1.17E+05	1.28E+05

Figure 16: Expected overall averages with a bound of 4'000'000 explorations for games where the random technique is not successful.

I adopted the DFS-solver to make use of this series of runs. I make sure that only fully explored states remain in the memoization set. So whenever the search is aborted, I remove the states that are currently on the stack from the memoization set, as these were in the middle of exploration and might need to be looked at again in order to find an optimal solution. With this, it is not necessary to restart the solver each time from scratch, so that we can reuse the gained knowledge of previous DFS-runs and make sure that every state gets explored only once.

After that, I implemented a combination of such solvers, which I called *meta-solver*. I ended up using a variety of expansion budgets in this order:

- once using 10'000 expansions
- 40 times using 250 expansions
- 60 times using 1'000 expansions
- 4 times 2'5000 expansion
- once using 10'000 expansions

If none of these 100'000 explorations leads to a solution, then the regular limitless solver kicks in. The idea is to find the best balance between restarting often and making sure that a considerable amount of options is looked at before each restart. The numbers I picked above are somewhat arbitrary and are based mainly on a few examples that I tried out by hand. After running it on a big amount of random games I was happy to see that the average amount of work over random games (including unsolvable ones) significantly decreased by *29%* down to only *144'000* expansions per game.

5. Interactive Visualization

Python Visualization Script

As with any board game, debugging its states and actions only using textual output can be very tedious. Therefore I wrote a simple python script that takes a string representation of a sequence of states and presents a simple UI to click through them back and forth. That allowed me to follow the solutions that the solver found and to make sure that all the rules were met.

iOS Application

In order to play the game on various board sizes myself and to interactively show the results of the solver, I wrote an iOS application. The user can move the stones around using a simple tap and drag gesture.

The solver runs on a separate thread in the background and continuously analyses the current state. The app can display the minimum stone count as well as arrows for each of the allowed moves. Figure 17 shows this user interface.

The color of the arrows encodes how good the corresponding move is. Green arrows mean that they are on a path towards solving the game and red arrows symbolize worse options. A yellow arrow stands for the best option once the game is unsolvable and a blue arrow is used whenever the resulting state could not be completely analyzed within a limited amount of time.

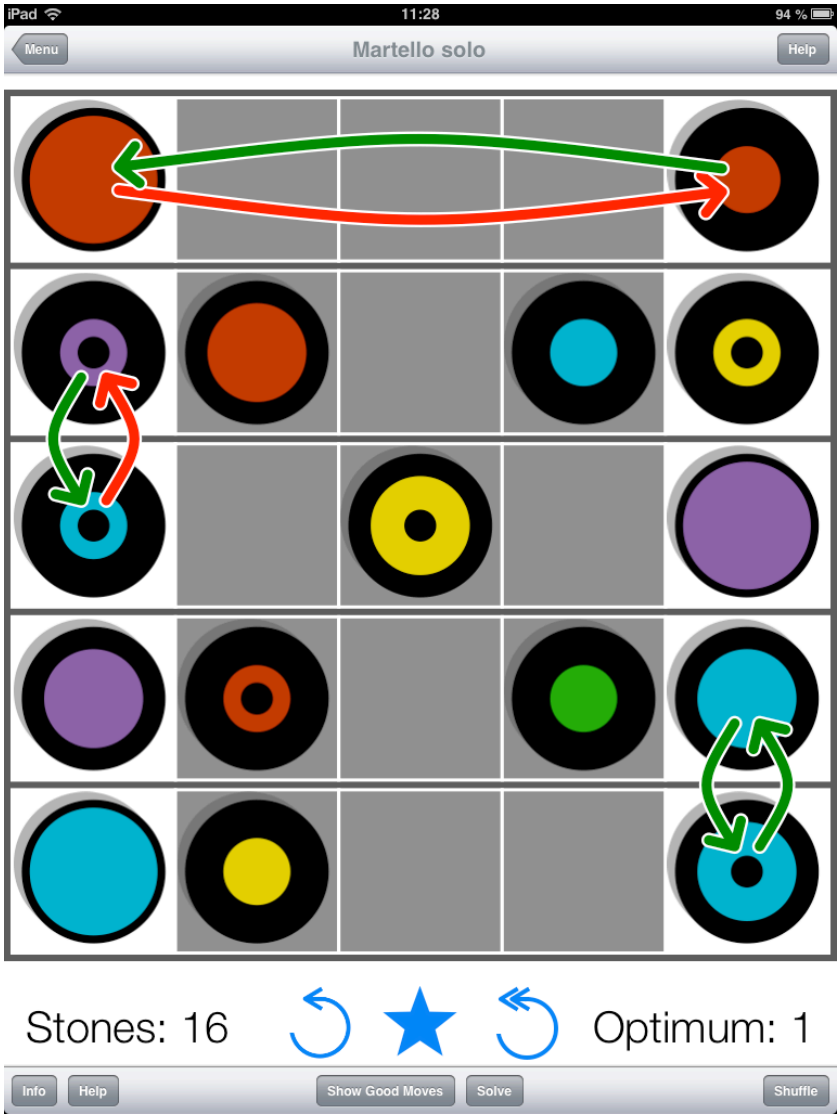


Figure 17: Screenshot of the iOS implementation on an iPad

One thing I also had to add was a translation between moves on the normalized state used for the solver and the potentially multiple moves on the unnormalized state shown on the screen. As each stone is unique, this can be achieved by just cross-referencing all the positions of the stones in both states.

The three buttons below the board allow the user to undo the last move, show and hide the arrows and to restart the game from the beginning.

6. Code Overview

This section provides a brief overview of my code base.

Martello Solver

The solver is written in C++ and comes with a makefile as well as an Xcode-project. In *main.cpp* all the high level functions for running the different experiments can be found. The short shell script *multi_run_script.sh* can be used to repeatedly start the solver and feed its output into a *mysql*-database. I used a separate database table for each experiment and then either summarized the data directly using SQL-queries or by exporting the relevant columns and processing them in MATLAB. Here are two typical queries:

To compare the average amount of state explorations of the meta solver with the regular solver:
SELECT SUM(metaStatesExplored)/count(*), SUM(statesExplored)/count(*) FROM metaSolver;

To list the number of boards for each combination of board size and minimum stone count:
SELECT N, minStoneCount, count(*), task FROM variableSize GROUP BY N, minStoneCount ORDER BY N, minStoneCount ASC;

The C++ code contains useful data structure in *State.cpp* and *Solution.cpp* that represent stones, moves and states. For instance, the method *allAllowedMoves()* can be called on every state and returns a list of all allowed moves according to the rules of the game. *Solver.cpp* implements the depth first search and *SolverAnalyzer.cpp* does implement the experiments I did with the solver. Finally, *MetaSolver.cpp* contains the implementation of the randomized restart technique.

Python Visualization

The script *displaymartello.py* takes any string representing a sequence of states and paints a board view using TkInter. Two buttons allow to go forward and backward in the sequence of states.

iOS Application

The app launches with a *UINavigationController* and a screen with a button to start the game. After that the *MartelloGameViewController* takes over and manages the *GameBoardView* and all the labels and buttons. It also takes care of starting the solver whenever a move is made and to synchronize its results with the presentation in the user interface. The *GameBoardView* paints the grid of the board using the dimensions calculated in a *GameBoardLayout* instance. Each stone is painted by a *StoneView* instance. The *MoveOverlayView* is responsible for painting the arrows. I use the *CoreAnimation* framework for animating moves and for fading out eaten stones.

7. Conclusion

I am very happy with the outcome of this project. The solver works accurately and reasonably fast and its visualization in the app is engaging and fun to play with. For instance it is now possible for me to play the game and see at what point I got off the track to solving the game and restart from there. Before publicly releasing the app some more polishing of the user interface would be needed. For instance a tutorial should be provided for the user to learn the rules of the game.

The analysis of the game gave new insights as well. We saw that in almost all random start states the game is solvable and that starting with a few random moves usually does not hurt the solvability. Furthermore, we found that also smaller board sizes work well and that they keep similar properties.

In future work one might try out other solver variants and search for the best budget sizes of the random restart solver. It would also be worthwhile to investigate the two-player variants of the game.