Playout Optimization

Lilian Buzer
LIGM
Université Gustave Eiffel, CNRS,
F-77454 Marne-la-Vallée, France
lilian.buzer@esiee.fr

Tristan Cazenave
LAMSADE
Université Paris-Dauphine, PSL, CNRS
Paris, France
Tristan.Cazenave@dauphine.psl.eu

Abstract—To generate playouts, we successively play moves and update the current list of possible moves. This last task can be costly if we analyze a too large area of the gameboard. We introduce the Range Query technique which speeds up playouts’ generation by a factor of about 10. By using a careful numbering of the moves, this technique locates the moves in the list in relation with the played move and updates them by analyzing only relevant squares of the gameboard. This technique uses very little memory and all the game data can remain in the L1 CPU cache which helps to improve performance. We also use SSE2 optimization to improve the performance of list processing. This rather general technique can be reused in any game where the invalidated and the new moves are in the neighborhood of the move that has been played. The performance gain can impact any algorithm based on playouts’ generation. For the experiments, we test our technique with the NMCS and NRPA algorithms on the 5D and 5T variants of the Morpion Solitaire.

Index Terms—Range Query, Monte Carlo Search, Morpion Solitaire

I. INTRODUCTION

Monte Carlo Tree Search (MCTS) has been successfully applied to many games and problems [1]. Nested Monte Carlo Search (NMCS) [2] is an algorithm that works well for puzzles. It biases its playouts using lower level playouts. At level zero NMCS adopts a uniform random playout policy. Online learning of playout strategies combined with NMCS has given good results on optimization problems [3]. Other applications of NMCS include Single Player General Game Playing [4], Coding Theory [5], Cooperative Pathfinding [6], Software testing [7], heuristic Model-Checking [8], the Pancake problem [9], Games [10], Cryptography [11] and the RNA inverse folding problem [12].

Online learning of a playout policy in the context of nested searches has been further developed for puzzles and optimization with Nested Rollout Policy Adaptation (NRPA) [13]. NRPA has found new world records in Morpion Solitaire and crosswords puzzles. Edelkamp, Cazenave and co-workers have applied the NRPA algorithm to multiple problems. They have optimized the NRPA for the Traveling Salesman with Time Windows (TSPTW) problem [14], [15]. Other applications deal with 3D Packing with Object Orientation [16], the physical traveling salesman problem [17], the Multiple Sequence Alignment problem [18], Logistics [19], [20], Graph Coloring [21] and Inverse Folding [22]. The principle of NRPA is to adapt the playout policy so as to learn the best sequence of moves found so far at each level.

The performance of NRPA, NMCS and MCTS is closely related to the speed of the playouts. Scalability studies of MCTS [2], [13], [23] usually show a linear increase of the scores with the logarithm of the number of playouts. It means that decreasing 10 fold the time of a playout gives an increase in score proportional to log(10) for the same search time. For problems such as Morpion Solitaire a search at a higher level usually gives much better result than the search at the lower level. Reducing the playouts time enables to do more search at a high levels thus increasing the probability of breaking records. This is the main reason why we are interested in optimizing playouts time.

The paper is organized as follows. In the next Section, we introduce our new technique named Range Query. We also present the Morpion Solitaire game that will be used to benchmark our technique. In Section 3, we present different implementation strategies, especially the management of the list of possible moves using SSE2 instructions. Finally, in Section 4, we analyze the speed of the NMCS and NRPA algorithms based on massive playouts’ generation. We compare their speed with and without the Range Query technique. Finally, we run a broader series of simulations to obtain a better estimation of the distribution of the scores.

II. UPDATING THE LIST OF POSSIBLE MOVES

A. Philosophy

The performance of NRPA and NMCS algorithms is linked to the time spent on generating playouts. To generate a playout, we play moves until the game is over. An iteration of the playout main loop can be broken down into three steps: select a move among the possible moves, play this move by updating the game data and finally list all possible moves for this new game state. This way, each time a move is played, the entire game data may be analyzed to detect possible moves. This step is costly and it slows down the playouts’ generation.

B. Range Query

In Database, a Range Query is a common operation that retrieves all records where some value is between an upper and a lower boundary. In Computational Geometry, a Rectangular Range Query allows us to search for all the points located in a
rectangular region of the plane. We generalize these problems to playouts’ generation by implementing an oracle function capable of removing all possible moves from the current list that are lying next to the move that has just been played. Thus, we set up a two steps approach. First, we remove some possible moves from the current list using a Range Query on a specific area. Second, we update the game data and look for all the possible moves lying in this area and add them to the current list. After these two steps, the list of possible moves has been correctly updated.

C. Morpion Solitaire

Morpion-Solitaire is a pencil-and-paper single-player game played on a square grid with an initial configuration of 36 dots depicted in Figure 1. At each move, the player puts one dot on an empty grid position and draws a line traversing five consecutive dots including the last drawn dot. The line can be horizontal, vertical or diagonal. It cannot overlap with a specific area. Second, we update the game data and look for all the possible moves lying in this area and add them to the current list. After these two steps, the list of possible moves has been correctly updated.

D. Coordinate Transform

A move can be represented by its position \((x, y)\) and its direction. Each direction is coded as follows: 0: top-left, 1: upwards, 2: top-right and 3: to the right. When we play a move in Morpion Solitaire, only the horizontal / vertical / diagonals squares centered on the new dot have to be considered. In this star-shaped configuration, a Rectangular Range Query would be awkward because the invalidated area would be too large and there would be many more points to process in the end. To perform an efficient Range Query, consecutive lines/moves in the same direction must have an increasing code. For this purpose, we can no more use the Cartesian coordinate system, but we have to use a local coordinate system \((u, v)\) where \(v_d\) identifies a strip of direction \(d\) containing all consecutive lines and \(u_d\) identifies a line in this strip. See Figures 2 and 3 for some examples. To clarify the transformations we use, we present the associated formulas hereafter. For a gameboard of size \(L \times L\), we have:

- \(u_0 = y\) \quad \(v_0 = x + y\)
- \(u_1 = y\) \quad \(v_1 = x\)
- \(u_2 = y\) \quad \(v_2 = x - y + L - 1\)
- \(u_3 = x\) \quad \(v_3 = y\)

E. Coding Moves

For a gameboard of size \(L \times L\), the coordinate \(u_d\) varies from 0 to \(L - 1\) because the \(x\) and \(y\) coordinates vary from 0 to \(L - 1\). According to our formulas, the coordinate \(v_d\) varies from 0 to \(2(L - 1)\) in the worst case. Thus, with \(4 \times L \times 2L = 8L^2\) codes, we can represent all the moves on the gameboard. For a grid of size 64 \(\times 64\), this represents 32 768 different codes. So we can code each move using an unsigned 16 bits integer value (u16). For this, we use: 2 bits to represent its direction \(d\), 7 bits to represent its coordinate \(v_d\) and the last 6 bits to store its coordinate \(u_d\). Doing so, we guarantee that two consecutive moves/lines in the same direction have two consecutive codes.
F. Range Query Process

We present the Range Query technique for the 5D variant of Morpion Solitaire, the 5T version being similar. When a move is played, we decode its direction $d$ and its local coordinates $u_d$ and $v_d$. Indeed, by reversing these coordinates, we can determine the location $(x, y)$ of the move. By analyzing the five consecutive squares, we detect the hole where the new dot lies and thus we know its coordinate $(x_h, y_h)$. Then, we process the four directions independently: when processing direction $d'$, we transform the position of the new dot $(x_h, y_h)$ to obtain its local coordinates denoted by $(u_h, v_h)$. This new dot modifies the possible moves four squares away. The code $idmove$ of the move starting at the position $(u_h, v_h)$ in the direction $d'$ is given by $idmove = code(d', u_h, v_h)$. The code of the move whose line ends at the position $(u_h, v_h)$ is given by $idmove - 4$ because we carefully numbered the consecutive moves by increasing order. The code of all the other moves that have to be removed are in the interval $[idmove - 4, idmove]$, see Figure 4 for an example. Now, we can perform a Range Query to remove all these codes from the list of possible moves.

We now focus on the moves located next to the newly created line. The line forbids any move at a distance of four squares from its both ends. When a move is played, its code $idmove$ is known. The code of the move whose rightmost position corresponds to the left extremity of the line is $idmove - 4$. The code of the move whose leftmost position corresponds to right extremity of line is $idmove + 4$. The codes of all the other moves that have to be removed are in the interval $[idmove - 4, idmove + 4]$. For the same direction, if we consider the range query interval associated with the new dot and the range query interval associated with the new line, the first interval is always included in the second. So, we perform only one Range Query corresponding to the largest interval. All the process is presented in Algorithm 1.

G. Detecting Possible Moves

After removing all possible moves associated with the neighborhood of the line, we analyze each group of 9 squares around the new dot to determine the possible moves. It is useless to analyze the squares where the line lies because no move is possible in this direction. Indeed, there is no room to insert a new line, see Figure 4 for an example.

To describe the state of each square, we use 2 bits: one bit to tell if a line is present and one bit to tell if a dot is present. To process a group of five consecutive squares, we analyze 10 bits. For example the value 0b0101010001 indicates that no line is lying on these five squares, that four dots are present plus one hole. This configuration allows us to play one move at this position. We can use five if statements to detect the five configurations associated with a possible move, but if statements are expensive for a processor and risky. If a CPU mispredicts a condition, this results in a pipeline stall in the execution unit, implying a penalty cost of many CPU clocks. To remove all these if statements, which is always a good idea for CPU as for GPU, we prefer to use a precomputed array of size $2^{10}$. Each entry in this array is equal to 0 and 1, the value 1 indicates that a move can be played at this position. Its size of 1024 bytes is not expensive and may remain L1 cache (CPU as for GPU) allowing an access of 1 clock cycle which is extremely fast.

It remains to determine the code of the corresponding move. When we analyze the group of nine consecutive squares centered on the new dot, we describe their state using 18 bits. The last ten bits, corresponds to the group of five squares where the new dot lies in the leftmost square. The rule of insertion of the Morpion Solitaire are the same regardless of the direction, they can be seen as isotropic. Thus, when we extract of group of squares, we always arrange them horizontally as in Figure 4 regardless of their orientation. So, we can use the terms: leftmost or rightmost even if we are...
processing a vertical group of squares. We compute the code of the move starting at the new dot. Then, all the codes in the group of squares will be \( c - 1, c - 2 \ldots \). Finally, we present Algorithm 2 used to detect the possible moves present in a group of 9 squares:

\textbf{Algorithm 2: Detecting Moves in a group of 9 squares}

\textbf{Input:} \( B18: 18 \) bits describing the state of 9 squares  
\hspace{1cm} ListPossMove: Possible moves array  
\hspace{1cm} nb: size of the array  
\hspace{1cm} x, y: position of the new dot  
\hspace{1cm} d: direction of the 9 squares  

\textbf{Data:} PossMov\( [] \): 1 if you can play, 0 otherwise

\begin{verbatim}
1  // code of the rightmost move
2  idmove = code(d, uh, vh)

3// Processing the 5 possible moves
4  for p = 0 to 4 do
5      // Analyzing the last 5 squares
6      B10 = B18 & 0b1111111111
7      // Bypassing an if statement
8      // Insertion is valid when nb += 1
9      ListPossMoves[nb] = idmove
10     nb += PossMov[B10]
11
12    // Moving to the next group
13    B18 = B18 >> 2
14    idmove = idmove − 1
15  end
\end{verbatim}

H. Another approach

To describe the state of nine consecutive squares, 18 bits are required. Thus, using a large array of \( 2^{18} = 262 \, 144 \) entries, we can easily store the information needed to update the set of possible moves. Thus, we can speed up the Range Query approach by storing for each entry in the list of codes describing the incoming and outgoing moves. For example, let us suppose that we process a group of squares whose smallest code is 0. If the list of possible moves goes from \{5, 7\} to \{7, 8\}, we note that the move coded by 5 has been removed, and that the move coded by 8 has been inserted. So, the incoming and outgoing moves are 5 and 8 giving the list \{5, 8\}. If we choose to work with sorted lists, updating the list of possible moves consists in merging two sorted lists. To do this, we add a slight modification during the merging process: if two values are identical, both have to be removed. Thus, merging the list \{5, 7\} with the list \{5, 8\} produces the following results: \{7, 8\}. Merging two sorted lists can be done efficiently in linear time in the number of values. But, as each entry in the array has at most five elements, the array size is in the order of megabytes exceeding the size of the L1 cache causing a slowdown of memory accesses. During our tests, this approach proved to be as fast as the Range Query technique. If one should maintain a sorted list of possible moves, this approach should be considered.

III. Implementation Strategies

A. Sequence of moves

To facilitate the readability of the program, we create a class to specifically manage the list of played moves. We choose to model this list using a constant size array. This choice is motivated by a search for efficiency. Indeed, we bypass any resize operation. For the Morpion Solitaire, for the 5D variant as well as the 5T variant, a realistic maximum size of about 200 moves can be chosen. The code of the moves are stored using 16 bits integer, so the size of the array is about 400 bytes. This size can be perceived as important and it can be thought that any copy of this list will slow down the processing. Nevertheless, modern processors have a memory bus of 256 bits allowing the reading and the writing of a block of 32 bytes in one clock cycle, see [25] for further explanations. So, with such a bandwidth, copying a sequence of moves takes about 15 clock cycles which is a negligible amount compared to the whole process of creating a playout.

B. Random Number Generator

We use the Intel Fast Random Number Generator based on SSE2 instructions [26]. As SSE2 is available on all machines nowadays, we decided to use this algorithm whose benchmarks have revealed an ability to generate a random number in 10 clock cycles on average.

For seed generation, we use the recent C++ RdRand intrinsic based on an on-chip hardware random number generator using an entropy source like thermal noise. This function remains slow with about 1000 clock cycles per call but as it is only used at program startup, this has no impact on the performance. The seeds we use are generated from run to run to ensure that there is no repeats in our experiments.

C. List of Possible Moves

We create a specific class to manage this list. For the Range Query technique, the central task consists in removing values belonging to an interval. We run performance benchmarks using different implementations and strategies and it appears that SSE2 instructions have made it possible to gain in performance by a factor 2 for playouts’ generation. Indeed, SSE2 allows to load 8 codes at once and checks whether each of these values is belonging to an interval in few operations. The result is a vector of 8 Boolean masks corresponding to the values 0x0000 or 0xFFFF. It remains to perform a logical AND+NOT operation between the input values and these masks, thus each value to be removed is replaced by 0. We present the SSE2 source code using C++ intrinsics in Algorithm 3. When all Range Queries have been performed, the list contains some 0 values that have to be removed: they can be seen as empty cells. So, we finally pack the non-zero values to conclude. This pack operation is a native AVX512 operation not available on all platforms. So we had no choice but to write this last step in standard C.
Algorithm 3: Range Query Remove() using SSE2

**Input:** BaseAdr: array address of the list of possible moves

\( nb \): number of moves in the list\n
\( min, max \): two unsigned 16 bits int used to describe the interval \( I = [min, max] \)

**Result:** All values in the array belonging to the interval \( I \) are replaced by 0

```c
1 // Import C+ SSE intrinsics
2 #include <emmintrin.h>
3 // Fill a 8xu16 register with u16 min
4 __m128i Mins = __mm_set1_epi16(min-1)
5 // Fill a 8xu16 register with u16 max
6 __m128i Maxs = __mm_set1_epi16(max+1)
7 // Process the list per block of 8
8 nb_blocks = 1 + (nb-1) / 8
9 for i = 0 to nb_blocks do
10   // Load 8 values from the array
11   __m128i values = *BaseAdr
12   // Check > min-1, \( \geq \) not available
13   __m128i tl = __mm_cmpgt_epi16(values, Mins)
14   // Check < max+1, \( \leq \) not available
15   __m128i t2 = __mm_cmpgt_epi16(Maxs, values)
16   // Packed logical AND
17   // Eight masks: 0x0000 or 0xFFFF
18   __m128i masks = __mm_and_si128(t1, t2)
19   // Values in the interval \( I \) \( \rightarrow \) 0
20   newvalues = __mm_andnot_si128(masks, values)
21   // Store 8 packed codes in memory
22   *BaseAdr = newvalues
23   // Move to the next block
24   BaseAdr += 1
25 end
```

D. Data Structure for the GameBoard

When we process a move near the border, we have to use some if statements to detect when we are outside the gameboard. This implies an extra cost in the processing and this extra cost seems to be a waste of time because it does not impact the quality of the solution. We prefer to avoid this extra cost by expanding the size of the gameboard. For example, in the Morpion Solitaire, it is sufficient to add 4 squares around the gameboard to guarantee that no moves will cross the border.

For the Range Query technique, we need to quickly extract the information concerning nine consecutive squares. For each square, we use one bit to signify the presence a dot and another bit to tell that a line is present. We represent the game area as a two-dimensional array where each entry represents a square of the game. Each entry indicates with one bit denoted \( D \) if a dot is present and with four bits \( L_0, L_1, L_2 \) and \( L_3 \) if a line is crossing the associated direction. When we process the direction \( d \), we need to determine the value of the bit \( D \) and of the bit \( L_d \). To avoid time consuming bits’ manipulation, we choose to repeat the bit \( D \) four times in order to place this bit near to each bit \( L_0, L_1, L_2 \) and \( L_3 \). Thus, each byte of the array \( \text{Game}[x,y] \) is structured like this:

\[
\text{Game}[x,y] = L_0 \ D \ L_1 \ D \ L_2 \ D \ L_3 \ D
\]

When we want to read the information associated with direction number 1, we only need to perform a bit shift and a mask operation: \( DL_1 = (\text{Game}[x,y] \gg 4) \& 0b11 \). When we put a dot at position \((x,y)\), we only have to perform the following logical OR operation: \( \text{Game}(x,y) = 0b10101011 \).

E. Why optimizing playouts is difficult?

When we optimize the performance of an algorithm, we generally enter different loops and finally find some lines of code that are executed lots of time. We carefully optimize these lines to achieve better performance. When we work with playouts, we use a main loop to play each turn. Inside this loop, we find all the logic of the game. If the logic is not optimized, we may find a double loops to analyze the gameboard. Inside this double loops, we find some tests to detect if a move is possible. In this case, we are in the standard case where we have to optimize some lines of code inside large loops. With the Range Query technique, only one loop exists. This implies that the remaining code inside the last loop has no longer 10 lines of code but rather 200. To optimize performance, each of the 200 lines containing the game logic must be optimized. If only one part remains slow, it will reduce overall performance.

Analyzing performance on modern CPU is a tricky task. We must recall that the measured performances are the result of three things: the code/algorithm, the compiler and the strategies used by your CPU to optimize performance during execution. The compiler makes some optimizations. Small variations in the code style may lead to a different choice with different performances. On their side, CPUs use many tricks to perform operations in advance or in parallel. When a CPU senses an optimization, the execution can speed up by a factor 2.5. When the CPU processes some instructions whose dependency chain is intricate, performance can slow down by half. We can control the code of the program, but the compiler and the CPU act as two black-boxes. To find the suitable version of a function, we need to write many versions of the same function to detect the one which is well understood by the compiler and well optimized by the CPU. Fortunately, once the right version has been found, performance is stable across the different configurations we have tested: AMD Ryzen9, AMD Epyc with compilers: MSVC++ 14 and G++ 9.

IV. Performance Testing

A. Playouts’ Generation Benchmark

The following benchmarks are done on a Ryzen9 3900X processor @4.1GHz. The Range Query technique is efficient because it processes a very small area of the gameboard. To illustrate the gain achieved by narrowing the work area, we
perform some tests by expanding the search area. We present the results for the 5D Morpion Solitaire with a gameboard of $32 \times 32$ in Table I.

To analyze the performance of playout's generation, we create playouts using a uniform random selection of the moves. We count the CPU clock cycles spent and we divide this quantity by the total number of turns in all the playouts. Thus, we obtain an average number of CPU clock cycles used to play one move. This quantity does not represent the time spent for a specific move. Indeed, in the first half of the game, the number of children in the game tree oscillates between 28 and 10. In the second half of the game, this quantity oscillates between 9 and 2. The playouts generated by advanced algorithms like NMCS and NRPA have longer sequences. But, the decreasing of the number of children during the game is similar, so the average number of CPU clock cycles of playouts with shorter sequences also corresponds to the average performance of longer sequences. In Table II we present the average clock cycles required to play a move for the 5D and 5T variants. Next using CPU profiling, we present the time spent for each activity in Table III.

### Table I
#### Average Time to Play One Move Relative to the Size of the Area.

<table>
<thead>
<tr>
<th>Size of the region analyzed</th>
<th>CPU cycles</th>
<th>Speed gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Whole gameboard</td>
<td>$\sim 150,000$</td>
<td>$\times 1$</td>
</tr>
<tr>
<td>All squares in the four directions</td>
<td>$\sim 15,000$</td>
<td>$\times 10$</td>
</tr>
<tr>
<td>The surrounding rectangular area</td>
<td>$\sim 6,000$</td>
<td>$\times 25$</td>
</tr>
<tr>
<td>Range Query technique</td>
<td>$\sim 300$</td>
<td>$\times 500$</td>
</tr>
</tbody>
</table>

### Table II
#### Average Performance for the Range Query Technique.

<table>
<thead>
<tr>
<th>Average performance to play one move</th>
<th>Number of CPU clock cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>5D Variant - Gameboard of $32 \times 32$</td>
<td>288</td>
</tr>
<tr>
<td>5T Variant - Gameboard of $64 \times 64$</td>
<td>298</td>
</tr>
</tbody>
</table>

### Table III
#### Average Time Spent on Each Activity to Play One Move.

<table>
<thead>
<tr>
<th>Different types of activity</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random selection</td>
<td>9%</td>
</tr>
<tr>
<td>Coordinates transform</td>
<td>6%</td>
</tr>
<tr>
<td>Gameboard data structure management</td>
<td>31%</td>
</tr>
<tr>
<td>Phase I of Range Query (removals)</td>
<td>34%</td>
</tr>
<tr>
<td>Phase II of Range Query (detecting moves)</td>
<td>20%</td>
</tr>
</tbody>
</table>

### B. NMCS algorithm

We compare the performance of two different implementations of the NMCS algorithm: the one used in [2] and our NMCS version based on the Range Query technique. Both versions are written in C++ and benchmarks are performed on a Ryzen9 3900X processor @4.1GHz for the 5D variant of the Morpion Solitaire. Results are presented in Table IV.

The speed gain obtained with the Range Query technique is almost a factor of ten. The ratio between the time spent to produce a Level 3 sequence and a Level 2 sequence is close for the two implementations, which proves that both versions are similar from an algorithmic point of view: both versions use the same algorithm and also a two-dimensional array to represent the gameboard. So the speed gain is only attributable to the Range Query technique and to the SSE2 implementation of the list of possible moves.

### Table IV
#### Time Spent to Generate a NMCS Sequence for the 5D Variant.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Level 2</td>
<td>3.2 s</td>
<td>0.33 s</td>
</tr>
<tr>
<td>Level 3</td>
<td>582 s</td>
<td>66.1 s</td>
</tr>
<tr>
<td>Ratio L3/L2</td>
<td>182</td>
<td>200</td>
</tr>
</tbody>
</table>

In [2], the author was able to present the results obtained by the NMCS algorithm up to level 3 for the 5D version of Morpion Solitaire. Using a dual AMD EPYC 7702 64-Core machine, we are now able to present scores' distribution for level 4 of the NMCS algorithm in Figure 5. This computation uses 7 hours on average for 100 simulations running one simulation per core.

### C. NRPA algorithm

The NRPA algorithm [13] can be considered as the state of the art for solving the Morpion Solitaire problem. Indeed, this algorithm holds the double record for the 5D and for the 5T Morpion Solitaire. We set up two different optimizations presented hereafter. During our benchmarks, to generate a level 3 NRPA with the 5D variant, our optimizations saved 36% of computation time compared to a vanilla NRPA version. For the 5T variant, we saved 32% of computation time.
- Avoiding the copy of the policy: in the vanilla version, the current policy is copied and then used as a buffer value to compute the gradient and to update the current policy. We prefer to copy the policy values of the current sequence which limits the amount of data to be copied.
- Limiting the number of times an exponential is calculated: the NRPA algorithm associates a weight to each move. Then, when a move has to be chosen among the list of possible moves, a softmax function is used to compute the probability for each possible move of being selected. Nevertheless, computing this function at each node of the game tree during each playout generation is costly. So, instead of storing the weight of each move, we prefer to store the exponential value of the weights.

To benchmark the NRPA algorithm, we use $\alpha = 1$ and $N = 100$ for the number of iterations. We present the time required to run a simulation using one core per simulation in Table V using a Ryzen9 3900X processor @4.1GHz. For the 5T variant, we find the scaling factor of $\times 100$ between two levels of simulation, because a simulation of level $n$ requires 100 simulations of level $n-1$. In fact, this factor is a little higher because the more the simulation level increases, the longer the sequences are. For example, from level 3 to level 4 of the 5T variant, the average length of the sequences increases from 150 to 160.

<table>
<thead>
<tr>
<th>NRPA</th>
<th>Level 1</th>
<th>Level 2</th>
<th>Level 3</th>
<th>Level 4</th>
<th>Level 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>5D Version</td>
<td>&lt; 1 s</td>
<td>&lt; 1 s</td>
<td>14 s</td>
<td>24 min</td>
<td>41 h</td>
</tr>
<tr>
<td>5T version</td>
<td>&lt; 1 s</td>
<td>&lt; 1 s</td>
<td>17 s</td>
<td>33 min</td>
<td>57 h</td>
</tr>
</tbody>
</table>

Using a dual AMD EPYC 7702 64-Core machine, we were able to run 1000 runs of level 5 in 4 weeks for the 5T variant.

As a rough comparison, the author in [13] set up 28 runs in 28 weeks. The time gain of $\times 250$ is explained by the gain factor of the machine and the gain factor of the Range Query technique. For the 5D variant, the sequences of level 5 reached the actual record of 82 in only 10 iterations. The 90 remaining iterations we used to try to improve these best sequences, but no better solutions were found. For the 5T variant, only 2% of our simulations reach the record of 178. To try to exceed this score, we perform multiple “warm starts” initializing the start policy with an already known best sequence. Nevertheless, all our warm tests were only able to reach the scores of 176, 177 and 178. No simulation was able to exceed the known record.

D. Source code

To facilitate the reuse of the Range Query method, all source code for the 5D and 5T versions can be downloaded at the following address: [https://github.com/liliancode/RangeQuery](https://github.com/liliancode/RangeQuery). We use encapsulation to separate the different elements of the algorithm: the list of possible moves created using SSE2 has been put in a class to be reused as is. Other parts have been encapsulated like the gameboard and the different algorithms.

V. Conclusion

We have presented the Range Query technique as well as other optimizations that speed up Monte Carlo Search by a factor close to 10 for Morpion Solitaire. Using these optimizations we were able to generate 1000 NRPA search at level 5 for the 5T version, reaching many times the current world record. For the 5D version a level 5 search almost always reaches the current world record, indicating that it is extremely difficult or even impossible to break.

The techniques we use are optimized for Morpion Solitaire but the same ideas can be used for other games or problems. As it is important for Monte Carlo Search to generate playouts as fast as possible in most of its applications, we expect our
techniques to improve Monte Carlo Search for other games and problems.

REFERENCES


