Determination and Evaluation of Efficient Strategies for a Stop or Roll Dice Game: Heckmeck am Bratwurmeck (Pickomino)

Nathalie Chetcuti-Sperandio, Fabien Delorme, Sylvain Lagrue, Denis Stackowiak

To cite this version:
Nathalie Chetcuti-Sperandio, Fabien Delorme, Sylvain Lagrue, Denis Stackowiak. Determination and Evaluation of Efficient Strategies for a Stop or Roll Dice Game: Heckmeck am Bratwurmeck (Pickomino). IEEE Symposium on Computational Intelligence and Games(CIG’08, 2008, Australia. IEEE Press, pp.175-182, 2008. <hal-00801338>
Determination and Evaluation of Efficient Strategies for a Stop or Roll Dice Game: Heckmeck am Bratwurmeck (Pickomino)

Nathalie Chetcuti-Sperandio  Fabien Delorme  Sylvain Lagrue  Denis Stackowiak

Abstract—This paper deals with a nondeterministic dice-based game: Heckmeck am Bratwurmeck (Pickomino). This game is based on dice rolling and on the stop or roll principle. To decide between going on rolling or stopping a player has to estimate his chances of improving his score and of losing. To do so he takes into account the previous dice rolls and evaluate the risk for the next ones.

Since the standard methods for nondeterministic games cannot be used directly, we conceived original algorithms for Pickomino presented in this paper. The first ones are based on hard rules and not really satisfactory as their playing level proved to be weak. We propose then an algorithm using a Monte-Carlo method to evaluate probabilities of dice rolls and the accessibility of resources. By using this tactical computing in different ways the programs can play according to the stage of the game (beginning or end). Finally, we present experimental results comparing all the proposed algorithms. Over 7'500'000 matches opposed the different AIs and the winner of this contest turns out to be a strong opponent for Human Players.

I. INTRODUCTION

Games represent an exciting challenge for Artificial Intelligence. The ability of computers to confront human beings in a convincing manner, or even to defeat them, fascinate most people. Besides, games are a good framework to test algorithms developed for more general problems. Thus games are a good area to test out AI techniques and develop new approaches.

Deterministic games (ie. games where the result of each action is certain) have been substantially investigated. For instance the best chess programs defeated the world champion [1]. More recently checkers were totally solved [2]. On the contrary, the playing level of the best Go programs remains low, even if great progress has been made [3].

Unlike deterministic ones, nondeterministic games have been less studied. In this kind of games the actions of a player are affected by randomness. Dice games are good examples of nondeterministic games. But dice games are little studied in Artificial Intelligence. Their haphazard side dissuades most people from really studying this kind of games. However by taking randomness into account, some strategies can be drawn to make strong artificial opponents. Moreover despite randomness good human players often win this kind of games against other human players. Backgammon is an exception. It is the only nondeterministic game truly studied, with outstanding results [4]. The best programs for backgammon are based on neural networks [5].

In this paper, we focus on a stop or roll dice game Heckmeck am Bratwurmeck, (Pickomino in England and France). This game was created by Reiner Knizia and is edited by Zoch. "Heckmeck am Bratwurmeck" can be translated by the "skewered roasted worms" but, in the sequel of this paper, we will used the English name. The word "worms" comes from the farm packaging used by the editor for this game and the word "pickomino" is a blend of "to pick up" and "domino" and refers to the equipment of the game. The main principles and mechanisms of this game are very simple (it can be played from 7) and are based on the stop or roll principle, choice of dice and number decomposition.

It is a multi-player game based on dice rolling and exploiting the results at best. More precisely the aim is to pile up the most resources, represented by worms on tiles. Worms represented on pickominos are collected with respect to the score provided by a sequence of rolls. After each roll, under conditions described in section 2, the player can stop and takes worms. The player can also throw the dice again in the hope of taking more worms, but at the risk of being blocked and losing previously collected worms. The choice of dice can change all the initial probabilities consequently the initial decision will be modified too. To be efficient a program for Pickomino should estimate the risk of rolling again and take the appropriate decision. Unfortunately a priori probabilities are extremely hard to be computed, the number of possible states being exponential.

We propose in this paper different algorithms for a two-player Pickomino. The first ones, based on hard rules, are not really satisfactory and the playing level of such methods turns out to be weak. We propose then an algorithm using a Monte-Carlo method to evaluate probabilities of dice rolls, thus the accessibility of resources. Monte-Carlo methods are already used for bridge [7], [8] or Go [3], except that the algorithm we propose deals with simulation-trees. By using this tactical computing in different ways the programs can play according to the stage of the game (beginning or end). Finally, we present experimental results comparing all the proposed algorithms. Over 7'500'000 matches opposed the different AIs and the winner of this contest turns out to be a strong opponent for Human Players.

First, Section 2 presents the detailed rules for Pickomino. Then, Section 3 proposes different naive algorithms, based on hard rules and on expected return. Section 4 introduces a simulation-based algorithm that evaluate the risk in Pick-
omino and several programs based on this algorithm. Finally, Section 5 focuses on some improvements on programs presented in the previous section. Experimental results are provided in each section for all proposed algorithm. The final Appendix sums up all the experimentations.

II. GAME RULES

Pickomino is a game for 2 to 7 players, aged from 8. The goal of the game is to make high scores with dice to pick the most worms. To make high scores one has to take chances as the more one rolls the dice the higher the score is but the more likely it is to lose one’s turn and some worms. Detailed rules can also be found on Zoch’s website [9] or on brettspielwelt [10].

The equipment consists of

• 16 tiles, called pickominos, numbered from 21 to 36 and bearing from 1 to 4 worms, laying face upwards in the center of the table;

• 8 six-sided dice; the sides are numbered from 1 to 5, the sixth side bearing a worm.

Turn after turn each player builds a stack of tiles by trying, when it is his turn, to pick a tile either in the center of the table or on top of the stack of tiles of some other player. To do so the player has to roll the dice several times in order to get a high enough score.

A. A turn in progress

An ongoing turn can be broken down into three steps: first the player rolls the dice, then if he did not reach a dead end he chooses the dice he wants to keep (else he loses his turn), last he decides either to stop or to roll again.

1) Rolling and choosing the dice: After rolling the available dice, the player puts aside all the dice of some value he chooses among the values not chosen previously in the turn. The put aside dice are no longer available for the current turn.

Example 1: It is Alice’s turn, she throws the 8 dice and gets:

![Dice](image1)

She chooses the two 4-valued dice. Her score is $2 \times 4 = 8$.

Now she throws the 6 remaining dice and gets:

![Dice](image2)

She cannot choose the 4-valued die as she already chose this value. She chooses the 5-valued die. Her score is $2 \times 4 + 1 \times 5 = 13$.

If all the current available values have already been chosen the player reached a dead end and loses his turn.

Example 2: [continued] Alice kept 2 4-valued dice and one 5-valued die. She throws the 5 remaining dice and gets:

![Dice](image3)

She chooses the 2 2-valued dice and throws the 3 remaining dice. She gets:

Alice cannot choose any value as she already chose both 2-valued dice and 5-valued dice.

One can notice that going on throwing the dice is increasingly risky as the numbers of available values and of available dice decrease.

2) Stop or go: When the player was able to choose some dice, he must then decide either to stop and pick some tile if possible or to roll again in the hope of being able to pick a greater tile but risking reaching a dead end.

A player can pick a tile and put it on top of his stack if first he kept at least one worm-bearing die and second his
score is either greater or equal to the number of some tile in the center of the table or equal to the number of the top-stack tile of another player. Note that a worm-bearing die is worth 5 points.

Example 3: It is Bob's turn, he throws the 8 dice and gets:

He chooses the 2 worm-bearing dice. His score is: $2 \times 5 = 10$. His score is not high enough so he throws the 6 remaining dice and gets:

He chooses the 5-valued die. His score is: $2 \times 5 + 5 = 15$.

Bob throws the 5 remaining dice and gets:

He takes the 3 3-valued dice. Now his score is: $2 \times 5 + 5 + 3 \times 3 = 24$.

Bob kept at least one worm-bearing die and his current score is 24, so he can either stop (if some 24-or-less-valued tile is available) or throw the dice again.

3) Failing turn: A player loses his turn in three cases: first he could not choose any die after some dice rolling (all the values having been chosen already), second he gathered no worm through dice rolling, third his score is too low to get an available tile.

Then the player has to give back his top-stack tile and the greater available tile in the center of the table is returned face downwards (making it unavailable for the rest of the game) unless the player put back the current greater available tile. The intent in making the greatest tile unavailable when a tile is back in the center of the table is to lessen the risk of looping.

B. End of the game

The game ends when no more tile is available in the center of the table. The winner is the player having picked the most worms. In case of a tie the winner is the player having the highest-numbered tile.

III. Simple AIs

In order to compare experimentally the different algorithms provided in this paper, we present in this section several naive artificial intelligences based on very simple rules. However, these artificial intelligences have a good behaviour at the beginning of the game and they regularly defeat humans and more advanced programs. Moreover, most evolved programs can be compared with these simple programs and should beat them. As previously mentioned in Introduction, we only consider account 2-player Pickomino game.

A. Simple1AI and Simple2AI

Roughly speaking, a program playing Pickomino has to take several decisions:

- What is the best dice-value to keep?
- Should I stop or roll?
- Which pickomino should I take?

The programs presented in this section take these decisions with very simple hard rules. The first one, called Simple1AI (S1), acts like a child when he learns to play the game.

1) Choice of dice: S1 takes worms first, then 5's, then a value at random.
2) Stop or Roll: S1 stops as soon as possible (when a pickomino is accessible)
3) Choice of pickomino: in the stack of another player first, else in the center of the table

The second one, named Simple1AI (S2), refines the choice of dice of Simple1AI. It takes worms first, except if there are more 5's than worms. Last it takes the greatest value.

For instance, let us consider the following roll:

Simple1AI takes worms, while Simple2AI takes 5's. If worms and 5's have already been chosen, Simple1AI takes 1's, whereas Simple2AI takes 4's.

B. Taking probabilities into account

In order to obtain more convincing results, the algorithm Simple3AI computes the expected return for each choice. To favour worms, we raise their value to 6.

Example 4: For instance, consider the following roll:

Intuitively, in order to make the greatest result as possible, it is not a good idea to take the 1's.

The procedure computes the expected return. In this case, it chooses the 4's and it expects to obtain 29 in average.

Nevertheless Simple3AI has some limitations. For instance, it does not check the validity of the sequence of dice. Suppose that, at the beginning of the game, the program obtains the following roll:

\[
\begin{array}{c}
\blacklozenge \\
\blacklozenge \\
\blacklozenge \\
\blacklozenge \\
\blacklozenge \\
\blacklozenge
\end{array}
\]
In this case, taking the 5’s, one has a probability of $1/12$ (8%) only to get a worm afterwards, so to obtain a valid sequence.

C. Experimental results

In order to compare these three first programs, they were confronted in 20’000 matches. One algorithm is the first player for the 10’000 first matches, then the algorithms switch for the last 10’000 matches. The order of the players change to avoid a possible bias. In fact, several matches opposing different algorithm to themselves were played to determine the importance of the order of the players. Experimental results (out of 30’000 matches) show that the first player wins 50.6% of the matches.

Table II sum up the results. For instance the line corresponding to $S_1$ means that Simple1AI won 4094 times (out of 20’000 matches) against Simple2AI and 3347 times (out of 20’000 other matches) against Simple3AI.

<table>
<thead>
<tr>
<th>vs</th>
<th>$S_1$</th>
<th>$S_2$</th>
<th>$S_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_1$</td>
<td>-</td>
<td>4094</td>
<td>3347</td>
</tr>
<tr>
<td>$S_2$</td>
<td>15906</td>
<td>-</td>
<td>9133</td>
</tr>
<tr>
<td>$S_3$</td>
<td>16653</td>
<td>10867</td>
<td>-</td>
</tr>
</tbody>
</table>

As expected, Simple3AI is the best program. It beats the two other programs (83% of victories against Simple1AI and 54% against Simple2AI).

IV. A SIMULATION-BASED ALGORITHM FOR PROBABILITY ESTIMATION

The previous programs are unsatisfactory for different reasons. First, they fail to reach high pickominos: Simple1 and Simple2 choose the greatest dice value even if it means taking only one die, whereas Simple3 is too pessimistic. Moreover, they do not take into account the accessible pickominos.

A good program has to take the best decisions according to dice rolls. Note that the decisions are not independent one from the other so that an efficient program should adapt its strategy for each dice roll, for each situation and for each stage of the game.

A good Pickomino-playing program has to use a simulation-based algorithm in order to evaluate risk. Such a component is needed not only to choose beween stopping and rolling, but also to choose the initial goals, to adjust these goals and to determine the dice to be taken. Previous naive algorithms are not able to adapt their goals. For instance, at the end of the game, it is easier to aim for pickomino 21 on the top of the stack of some opponent (that is, to obtain exactly 21) or to aim for pickominos 31 and 32 on the table? Hence, the objective depends on the first roll but also on the accessible resources (pickominos on the table and pickominos on the top of opponents’ stacks).

The main problem, in this game, is that the a priori probabilities are extremely hard to compute. They depends especially on the decomposition of the values of pickominos, on the number of rolls, on the order of choices. For example the probability to obtain first the sequence $\square \square \square \square$, then $\square \square \square \square$ is different from obtaining first $\square \square \square \square$, then $\square \square \square \square$. As the second roll is concerned, in the first case, one rolls 3 dice, whereas in the second one, one rolls 5 dice. Moreover, the probabilities to obtain different values are entangled. For instance, one can obtain 23 by the following sequence of rolls: $\square \square \square \square \square \square$ and last $\square$. But after the second roll, 22 was reached. Thus events are not independent and are entangled.

A. A Simulation Algorithm

Algorithms based on Monte-Carlo simulations cannot take into account this tangle property. Therefore we propose an algorithm that produces trees of possibilities and not consecutive dice rolls only. This algorithm fills a table that can be used to make decision for dice choice. Algorithm 2 presents the algorithm used for simulation. It considers all the possible dice choices and continue recursively the simulation. The following example illustrates a single-tree simulation.

\begin{verbatim}
Algorithm 1: Initializing a table of risk
Risk and Fail are two global tables which contain the final results

procedure Init()
begin
   for i ← 1 to 6 do
      for j ← 1 to 37 do
         Risk[i][j] ← 0
         Fail[i] ← 0
end

Algorithm 2: Evaluating risk

procedure simul(D_K, n)
Input: D_K, the multiset of already-selected dice
n, the number of rolls

begin
   if $\square \in D_K$ then
      if Sum(D_K) ∈ {21, 22, ..., 36} then
         Risk[36][Sum(D_K)] ← Risk[36][Sum(D_K)] + 1
      else if Sum(D_K) > 36 then
         Risk[37] ← Risk[37] + 1
   D ← random(8 - NbDice(D_K))
   D′ ← {d : d ∈ D and d \∉ D_K}
   if D′ = \emptyset then
      Fail[nbTry] ← Fail[nbTry] + 1
   else
      foreach d ∈ D′ do
         simul(D_K ∪ {d}, A, n + 1)
end

Example 5: The current player took previously the following dice: $\square \square \square$. Figure 4 sums up a simulation from this situation. The initial score is 7, the algorithm simulates a dice roll, which is $\square \square \square \square \square$. Only two different dice can
\end{verbatim}
be chosen: \( \text{\textbullet} \) or \( \text{\textbullet} \). The algorithm explores both branches by simulating dice rolls.

Underlined scores in the tree represent valid scores, i.e. scores that are greater than 20 with at least one worm. Even if it reaches a valid score, the simulation goes on evaluating the risk or the gain for new rolls.

![Simulating risk](image)

When a sufficient number of simulations is realized, the table of risk is filled. Table VII gives the evaluation of the risk for the following initial roll (at the very beginning of the game): \( \text{\textbullet} \text{\textbullet} \text{\textbullet} \text{\textbullet} \text{\textbullet} \text{\textbullet} \text{\textbullet} \text{\textbullet} \).

### Table III 3 Risk Tables

| DK | Roll | 01 | 21 | 31 | 02 | 22 | 32 | 03 | 23 | 33 | 04 | 24 | 34 | 05 | 25 | 35 | 06 | 26 | 36 | 07 | 27 | 37 |
|----|-----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 |
| 1  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
| 1  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
| 1  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
| 1  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
| 1  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
| 1  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |
| 1  | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 |

It is the concatenation of 3 risk tables obtained after 500 simulations (for readability’s sake only values less than 30 were considered). The upper part of the table is the risk part associated with the choice of dice 1, the middle one for dice 5 and the lower one for worms. Dice 5 are clearly not a good choice. On the contrary, the choice between 1 and worms is more debatable and will be solved differently by two algorithms presented further. Moreover, choice depends on the goal: if the objective is to take pickomino 21, the choice of 1 seems to be the best one, on the contrary, if one wants to take a pickomino equal to 26 at least, worms are more adequate. One remarks that risk tables have 6 lines, because, according to the rules, the maximum number of dice rolls is 6: if one rolls again, he cannot obtain a value that was not previously kept.

### B. MC Algorithm

Different algorithms can be developed in order to take decision using a risk table. For instance algorithm 3 is a generic algorithm in which the decisions depend on a function \( \text{evalRisk} \). This function returns a value representing the risk of a choice (the greater is the value, the safer is the choice), according to the set of accessible pickominos (on the table or on the top of a stack of another player). In the first program, called MC (for Monte-Carlo), \( \text{evalRisk} \) computes the best sum of columns:

\[
\max_{\text{val} \in A} \sum_{i=0}^{6} \text{Risk}[i][\text{val}]
\]

where \( A \) denotes the set of accessible values of pickominos.

**Algorithm 3: Making decision**

**function** \( \text{choice}(D_R,D_K,A,p) \)

**Input:** \( D_R \), the multiset of rolled dice

\( D_K \), the multiset of already-selected dice

\( A \), the set of accessible pickominos

\( p \), the number of simulations

**begin**

\[ r_{\max} \leftarrow +\infty \]

\[ \text{val} \leftarrow 0 \]

**foreach** \( (d,p) \in D_R \) **do**

**init()**

**for** \( i \leftarrow 1 \) **to** \( n \) **do**

\[ \text{simul}(D_K \cup \{(d,p)\}, 1) \]

\[ r \leftarrow \text{evalRisk}(\text{Risk}, A) \]

**if** \( r > r_{\text{min}} \) **then**

\[ r_{\max} \leftarrow r \]

\[ \text{val} \leftarrow d; \]

**return** \( \text{val} \)

**end**

Anytime a valid value (i.e. a sequence of rolls with at least one worm kept and a score greater than 20) is reached, the program increments the associated value in a risk table, initialized by Algorithm 1. Several simulations are done and from 100 to 1000 trees are developed so as to estimate the risk for all possible decisions (another table of failures is also filled but it is currently not really useful).

Using Table VII, the program takes die 1, because column 21 has the maximum score (812). This algorithm gets pickominos as soon as possible and if it has to choose between two pickominos, he takes first the one on the top of an opponent’s stack. Experimental results show that this elementary algorithm is better than any simple algorithm presented in the previous section. Table IV sums up these simulations. MC wins most of its 20000 matches versus Simple1 (85%), Simple2 (59%) and Simple3 (53%).

The algorithm MC launches 500 different simulations. Now, the quality of this kind of algorithm depends on the number of simulations. Moreover, the runtime is exponentially affected by this number of simulations. The number of 500 iterations turns out to be an excellent compromise, as it is shown in Table V. The gain from 100 to 500 iterations
is over 2%, but the gain between 500 and 1000 iterations is less than 0.4%.

### TABLE IV
**MC vs Simple’s**

<table>
<thead>
<tr>
<th>vs.</th>
<th>S1</th>
<th>S2</th>
<th>S3</th>
<th>MC</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>4094</td>
<td>3347</td>
<td>2949</td>
<td></td>
</tr>
<tr>
<td>S2</td>
<td>15906</td>
<td>-</td>
<td>9133</td>
<td>8178</td>
</tr>
<tr>
<td>S3</td>
<td>16653</td>
<td>10867</td>
<td>-</td>
<td>9064</td>
</tr>
<tr>
<td>MC</td>
<td>17051</td>
<td>11822</td>
<td>10936</td>
<td>-</td>
</tr>
</tbody>
</table>

### TABLE V
**Iterations**

<table>
<thead>
<tr>
<th>vs.</th>
<th>MC</th>
<th>MC100It</th>
<th>MC1000It</th>
</tr>
</thead>
<tbody>
<tr>
<td>MC</td>
<td>9655</td>
<td>-</td>
<td>9925</td>
</tr>
<tr>
<td>MC100It</td>
<td>10075</td>
<td>10374</td>
<td>-</td>
</tr>
</tbody>
</table>

C. Adding up Chances

The main problem of the algorithm MC is that it does not add up chances: it only focuses on one possibility (the best one for each table). Formula (1) can be modified to add up chances, by changing the max operator into the sum operator:

\[
\sum_{\text{val} \in A} \sum_{i=0}^{6} \text{Risk}[i][\text{val}]
\]

where \(A\) denotes the set of accessible values of pickominos.

Cumulating Algorithm (MCC) is based on this formula. In that case, if one considers again Table VII at the very beginning of the game, contrary to MC, that algorithm does not choose die 1, but worms. Both algorithms were tested in 20'000 matches and MCC won only 10'097 times (50.5%).

### V. Improving Algorithms

This section studies different methods to improve the algorithms proposed in the previous section. More particularly, this section focuses on:

- the choice of pickomino,
- taking more risk in dice rolling,
- a better management of the end of the game.

These methods were all experimentally evaluated. And the program MC4C, that includes all these improvements, turns out to be the best program and a strong opponent against human players.

A. Which Pickomino to Take?

One crucial moment in Pickomino is the choice of one pickomino. Indeed, this choice can strongly change the course of the game. There are at most two available pickominos: the one on top of the opponent’s stack or a lesser valued pickomino on the table. In the previous section, all the algorithms take first the pickomino on the top of the opponent’s stack. This strategy has two advantages: increasing the number of worms of the opponent. In a duel between MC and MC2 (a variant of MC that takes a pickomino on the table first) MC wins 10’866 times on 20’000 matches (54%).

### B. Taking More Risk

Should programs take more risk? This essential point needs also to be evaluated. Actually, MC and MCC algorithms stop as soon as possible, when a pickomino can be taken. However, if one gets a roll with all the dice values already taken, is it reasonable to try another roll.

More formally, the simple probability to fail, i.e. to have a roll with all the dice values already taken, is:

\[
\frac{|\text{distinct}(D_K)|^6 - |D_K|}{6^8 - |D_k|}
\]

where \(D_K\) denotes the multiset of already kept dice, \(|D_K|\) the cardinality of \(D_K\), \(\text{distinct}(D_K)\) the set of distinct elements in \(D_K\) and \(|\text{distinct}(D_K)|\) the cardinality of \(\text{distinct}(D_K)\).

Variations on MC were tested. These versions roll again if the risk estimated by formula (3) is less than some limit. Four thresholds were tried out: 5%, 10%, 25% and 50%. Experimental results, described in Table VI and by Figure 5, are quite surprising: taking too little or too big risk is not efficient. A good compromise should be used.

### TABLE VI
**Taking risk**

<table>
<thead>
<tr>
<th>vs.</th>
<th>MC</th>
<th>MCP5</th>
<th>MCP10</th>
<th>MCP25</th>
<th>MCP50</th>
</tr>
</thead>
<tbody>
<tr>
<td>MC</td>
<td>-</td>
<td>15174</td>
<td>9667</td>
<td>10868</td>
<td>15233</td>
</tr>
<tr>
<td>MCP5</td>
<td>4826</td>
<td>-</td>
<td>4496</td>
<td>4746</td>
<td>9990</td>
</tr>
<tr>
<td>MCP10</td>
<td>10333</td>
<td>15504</td>
<td>-</td>
<td>11188</td>
<td>15440</td>
</tr>
<tr>
<td>MCP25</td>
<td>9132</td>
<td>15254</td>
<td>8812</td>
<td>-</td>
<td>15118</td>
</tr>
<tr>
<td>MCP50</td>
<td>4767</td>
<td>10010</td>
<td>4560</td>
<td>4882</td>
<td>-</td>
</tr>
</tbody>
</table>

Algorithm MC is beaten only by MCP10 (MC with a threshold of 10%) with a 48,335% of lost games. Algorithms MCP5 and MCP50 are very close and inefficient. The best compromise on these tests is 10%, but more precise evaluation of the threshold should be made in the future.

![Fig. 5. Taking more risk](image)
C. End of the Game and Vicious Circles

Last, we concentrate on a particular stage of the game (studied in all games): the end. The algorithm used in the middle of a game is often ineffective at the end of the game. Pickomino is not an exception.

1) What is a little pickomino: In Pickomino, we consider that the end of the game begins when only 3 "little" pickominos remain on the table. More than 50’000 simulations were launched in order to determine what a little pickomino is. Previously, dice were rolled once before starting the simulations, here the simulations start from scratch. Figure 6 sums up these simulations. Values 21 to 25 gather 80.45% of the valid rolls. In the sequel of the paper, we will consider that little pickominos go from 21 to 26 and hard pickominos go from 27 to 36.

Fig. 6. What is a little pickomino?

2) Breaking Vicious Circles: Vicious circles can appear at the end of the game. Suppose that pickomino 21 is the top-stack tile of current player Bob (the next one being pickomino 22) and that pickomino 32 is the top-stack tile of his opponent Alice. Three pickominos remain on the table: 33, 34, 35. In this case, Bob will most probably fail. Then he gives back pickomino 21, which is much easier to take than 33, 34 or 35. In this case, if Alice obtains 22, she should take 21 on the table and not 22 on Bob’s stack, else Bob will most probably take 21 and Alice will enter a vicious circle.

Two programs have been developed in order to take into account vicious circles. These programs take more risk at the end of the game when the number of the remaining little pickominos is even. The first one, MC4, is based on MCPr10. The second one, MC4C, which is based on MCC, takes risk with a threshold of 10%. Table VII and Figure 7 sums up all the matches. Algorithms MC and MCC are added to have a broader comparison. Algorithm MC4C proved to be the best algorithm presented in this paper.

### Table VII

<table>
<thead>
<tr>
<th>Taking Risk</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>vs.</strong></td>
</tr>
<tr>
<td><strong>MC</strong></td>
</tr>
<tr>
<td><strong>MC4</strong></td>
</tr>
<tr>
<td><strong>MCC</strong></td>
</tr>
<tr>
<td><strong>MC4C</strong></td>
</tr>
</tbody>
</table>

VI. Conclusion and Perspectives

This article focuses on an original dice game, Pickomino. We investigate several ways to make an efficient program for this game. Some of them prove to be dead ends. On the contrary the combination of complementary algorithms (Monte-Carlo techniques, parsimonious risk taking, vicious circles breaking) leads to a strong program: MC4C. It beats all the other algorithms and it is the best winner (having the most victories) against any algorithm, except for S1 and MC2, where it is nearly the best (see table VIII). All the algorithms were confronted in 20’000 matches for each duel (see table VIII).

Some matches were organized against human players and MC4C won most of them. We plan to develop the evaluation against human players, for instance by the participation to a league (e.g. on the brettspielwelt website [10]) or with a match against the best European players (a first European cup was organized by Zoch [11]).

Moreover some ways need to be explored. For instance, the threshold of risk leading to the best result (10%) is somewhat arbitrary and further simulations should help estimating the finest threshold. Nondeterministic game trees [6] could be used to improve the end of the game. Besides, the proposed algorithms need to be generalized for more than 2 players, the difficult point being the management of the end of the game. Finally, it would be interesting to adapt the proposed algorithms to other advanced dice games, for instance Yahtzee/Yams.

**References**

### Table VIII

#### Summary of all Programs

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>-</td>
<td>N</td>
<td>-</td>
</tr>
<tr>
<td>S2</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>-</td>
<td>N</td>
<td>-</td>
</tr>
<tr>
<td>S3</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>-</td>
<td>N</td>
<td>-</td>
</tr>
<tr>
<td>MC</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>-</td>
<td>N</td>
<td>-</td>
</tr>
<tr>
<td>MC2</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>-</td>
<td>N</td>
<td>-</td>
</tr>
<tr>
<td>MC3</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>-</td>
<td>N</td>
<td>-</td>
</tr>
<tr>
<td>MC4</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>10%</td>
<td>Y</td>
<td>-</td>
</tr>
<tr>
<td>MC4P5</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>5%</td>
<td>N</td>
<td>-</td>
</tr>
<tr>
<td>MC4P10</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>10%</td>
<td>N</td>
<td>-</td>
</tr>
<tr>
<td>MC5</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>-</td>
<td>N</td>
<td>-</td>
</tr>
<tr>
<td>MC5P10</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>10%</td>
<td>N</td>
<td>-</td>
</tr>
<tr>
<td>MC5P25</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>25%</td>
<td>N</td>
<td>-</td>
</tr>
<tr>
<td>MC5P50</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>50%</td>
<td>N</td>
<td>-</td>
</tr>
<tr>
<td>MC1000H</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>-</td>
<td>N</td>
<td>-</td>
</tr>
<tr>
<td>MC1000R</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>-</td>
<td>N</td>
<td>-</td>
</tr>
<tr>
<td>MC4P10</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>10%</td>
<td>N</td>
<td>-</td>
</tr>
<tr>
<td>MC4P100</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>100%</td>
<td>N</td>
<td>-</td>
</tr>
<tr>
<td>MC4C</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>-</td>
<td>N</td>
<td>-</td>
</tr>
<tr>
<td>MCC</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>-</td>
<td>N</td>
<td>-</td>
</tr>
<tr>
<td>MCCC</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>10%</td>
<td>Y</td>
<td>-</td>
</tr>
</tbody>
</table>

### Fig. 8

All Results in one graph