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-

All people are with CRIL-CNRS UMR 8188, Université d'Artois, F-62307 Lens, France (phone: +33-3-21-79-17-90; fax: +33-3-21-79-17-70; email: {chetcuti, delorme, lagrue, stackowiak}@cril.univ-artois.fr.

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;

Fig. 1. Some tiles

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

Fig. 2. Dice

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.

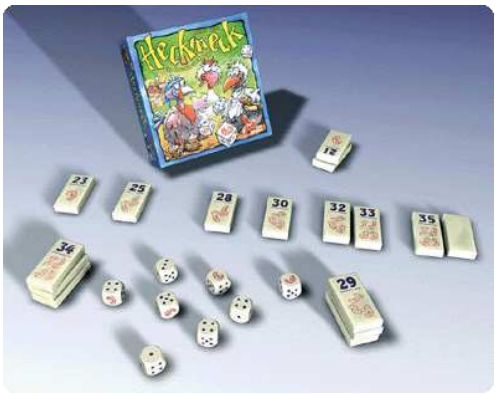


Fig. 3. The game in progress with 3 players

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:

She chooses the two 4-valued dice. Her score is $2 \times 4 = 8$*. Now she throws the 6 remaining dice and gets:*

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:

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.

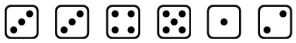
2) Stop or go: When the player was able to choose some dice, he must then decide either to stop and pick some tile if possible or to roll again in the hope of being able to pick a *greater* tile but risking reaching a dead end.

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 × 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 × 5 + 5 = 15. Bob throws the 5 remaining dice and gets:

He takes the 3 3-valued dice. Now his score is: 2 × 5 + 5 + 3× 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:

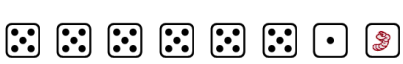
Intuitively, in order to make the greatest result as possible, it is not a good idea to take the 1's.

TABLE I
SIMPLE3AI IN ACTION

choice	gain	average of remaining values	expected return
1	$3 = 3 \times 1$	4	$23 = 3 + 4 \times 5$
4	$12 = 3 \times 4$	3.4	$29 = 12 + 3.4 \times 5$
5	5	3.2	$27.4 = 5 + 3.2 \times 7$
6	6	3	$27 = 6 + 3 \times 7$

For each choice, the program 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:



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 determinate 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 II
MATCHES BETWEEN SIMPLE AIS

vs.	S1	S2	S3
S1	-	4094	3347
S2	15906	-	9133
S3	16653	10867	-

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, is it 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 ⚃ ⚃ ⚃ ⚃, then ⚂ ⚂ is different from obtaining first ⚂ ⚂, then ⚃ ⚃ ⚃ ⚃. 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: ⚃ ⚃ ⚃ ⚃ then ⚁ and last ⚀. 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.

Algorithm 1: Initializing a table of risk

$Risk$ and $Fail$ are two global tables which contain the final results
procedure *Init()*
begin
  for $i \leftarrow 1$ *to* 6 **do**
    for $j \leftarrow 21$ *to* 37 **do**
      $Risk[i][j] \leftarrow 0$
    $Fail[i] \leftarrow 0$
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 ⚅ $\in D_K$ **then**
    if $Sum(D_K) \in \{21, 22, ..., 36\}$ **then**
      $Risk[n][Sum(D_K)] \leftarrow Risk[Sum(D_K)] + 1$
    else if $Sum(D_K) > 36$ **then**
      $Risk[36] \leftarrow Risk[37] + 1$
  $D \leftarrow random(8 - NbDice(D_K)$
  $D' \leftarrow \{d : d \in D$ and $d \notin D_K\}$
  if $D' = \emptyset$ **then**
    $Fail[nbTry] \leftarrow Fail[n] + 1$
  else
    foreach $d \in D'$ **do**
      $simul(D_K \cup \{d\}, A, n + 1)$
end

Example 5: The current player took previously the following dice: ⚂ ⚂ ⚀. Figure 4 sums up a simulation from this situation. The initial score is 7, the algorithm simulates a dice roll, which is ⚃ ⚅ ⚃ ⚂ ⚁. Only two different dice can

be chosen: 3 or 1. The algorithm explores both branches by simulating dice rolls.

Underlined scores in the tree represents 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.*

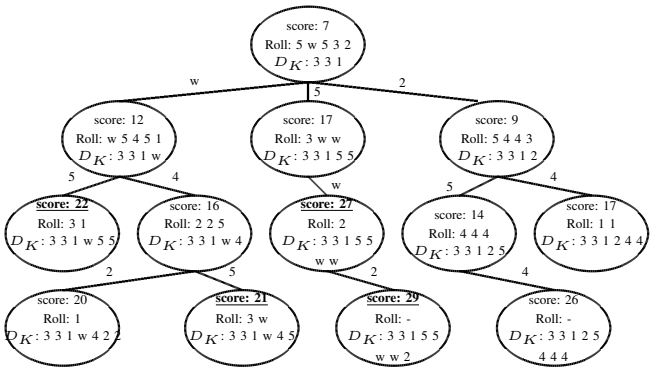


Fig. 4. Simulation of risk

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): 3 3 5 3 5 1 5 5.

TABLE III
3 RISK TABLES

Dice	Roll #	21	22	23	24	25	26	27	28	29	30	...
1	1	5	0	0	0	0	3	0	0	0	0	...
	2	89	12	10	10	6	28	2	3	1	0	...
	3	123	142	166	162	137	25	33	37	42	23	...
	4	613	458	384	167	89	241	139	114	29	12	...
	5	0	141	159	148	314	35	72	63	37	51	...
	6	0	0	0	0	0	0	0	0	0	0	...
5	1	0	0	0	0	193	0	0	0	0	59	...
	2	0	0	0	0	0	146	158	135	161	0	...
	3	0	0	0	0	0	0	0	123	122	253	...
	4	0	0	0	0	0	0	0	0	0	0	...
	5	0	0	0	0	0	0	0	0	0	0	...
	6	0	0	0	0	0	0	0	0	0	0	...
3	1	92	0	75	16	97	0	16	0	0	15	...
	2	370	462	268	274	134	175	93	131	92	9	...
	3	176	178	379	460	406	329	326	192	97	213	...
	4	0	0	0	0	175	209	203	248	287	0	...
	5	0	0	0	0	0	0	0	0	0	150	...
	6	0	0	0	0	0	0	0	0	0	0	...

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 $evalRisk$. 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), $evalRisk$ computes the best sum of columns:

$$\max_{val \in A} \sum_{i=0}^{6} Risk[i][val] \quad (1)$$

where A denotes the set of accessible values of pickominos.

Algorithm 3: Making decision

function 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
  $r_{max} \leftarrow +\infty$
  $val \leftarrow 0$
  foreach $(d, p) \in D_R$ **do**
    $init()$
    for $i \leftarrow 1$ *to* n **do**
      $simul(D_K \cup \{(d, n)\}, 1)$
      $r \leftarrow evalRisk(Risk, A)$
      if $r > r_m in$ **then**
        $r_{max} \leftarrow r$
        $val \leftarrow d$;
  return *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 20′000 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

TABLE IV
MC VS SIMPLE'S

vs.	S1	S2	S3	MC
S1	-	4094	3347	2949
S2	15906	-	9133	8178
S3	16653	10867	-	9064
MC	17051	11822	10936	-

is over 2%, but the gain between 500 and 1000 iterations is less than 0.4%.

TABLE V
ITERATIONS

vs.	MC	MC100It	MC1000It
MC	-	10345	9925
MC100It	9655	-	9626
MC1000It	10075	10374	-

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:

$$\sum_{val \in A} \sum_{i=0}^{6} Risk[i][val] \tag{2}$$

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:

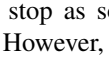
- the choice of pickomino,
- taking more risk in dice rolling,
- a better management of the end of the game.

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 player while decreasing 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  after the first roll, the only danger with the next roll is to obtain 3 worms. The probability of such an event is only $1/6^3$, i.e. 0.4%. The probability to improve the score is 99.6%. It is 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:

$$\frac{(|distinct(D_K)|)^{8-|D_K|}}{6^{8-|D_k|}} \tag{3}$$

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

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

vs.	MC	MCPr5	MCPr10	MCPr25	MCPr50
MC	-	15174	9667	10868	15233
MCPr5	4826	-	4496	4746	9990
MCPr10	10333	15504	-	11188	15440
MCPr25	9132	15254	8812	-	15118
MCPr50	4767	10010	4560	4882	-

Algorithm MC is beaten only by MCPr10 (MC with a threshold of 10%) with a 48,335% of lost games. Algorithms MCPr5 and MCPr50 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.

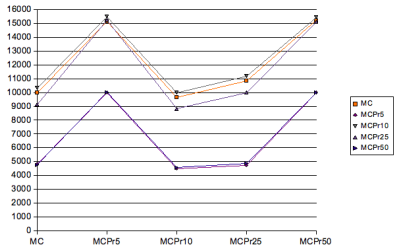


Fig. 5. Taking more risk

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.

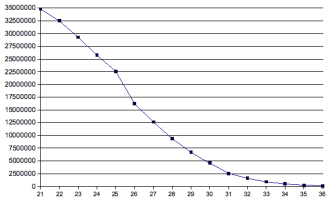


Fig. 6. What is a little pickomino?

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

TABLE VII
TAKING RISK

vs.	MC	MC4	MCC	MC4C
MC	-	9562	9903	9487
MC4	10438	-	10376	9907
MCC	10097	9624	-	9508
MC4C	10513	10093	10492	-

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.

VI. CONCLUSION AND PERSPECTIVES

This article focuses on an original dice game, Pickomino. We investigate several ways to make an efficient program

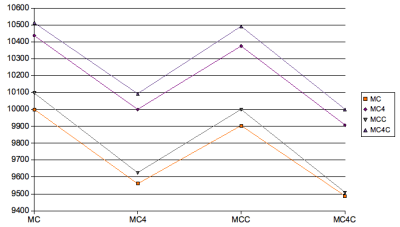


Fig. 7. Best programs

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

[1] M. Campbell, A. J. Hoane Jr., and F.-h. Hsu, "Deep blue," *Artificial Intelligence*, vol. 134, no. 1-2, pp. 57–83, 2002.
[2] J. Schaeffer, Y. Björnsson, N. Burch, A. Kishimoto, M. Müller, R. Lake, P. Lu, and S. Sutphen, "Solving checkers," in *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI'05)*, Edinburgh, 2005, pp. 292–297.
[3] R. Coulom, "Efficient selectivity and backup operators in monte-carlo tree search," in *Proceedings of the fifth International Conference on Computers and Games (CG'06)*, ser. Lecture Notes in Computer Science, vol. 4630/2007. Springer, 2006.
[4] H. J. Berliner, "Backgammon computer program beats world champion," *Artificial Intelligence*, vol. 14, no. 2, 1980.
[5] G. Tesauro, "Programming backgammon using self-teaching neural nets," *Artificial Intelligence*, vol. 134, no. 1-2, 2002.
[6] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 2nd ed. Prentice Hall, 2002.
[7] M. L. Ginsberg, "Gib: Steps toward an expert-level bridge-playing program," in *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI'99)*, 1999.
[8] ——, "Gib: Imperfect information in a computationally challenging game," *Journal of Artificial Intelligence Research (JAIR)*, vol. 14, pp. 303–358, 2001.
[9] http://www.zoch-verlag.com/fileadmin/user_upload/Spielregeln/Huehnerregeln/Heckmeck/SR-Heckmeck-en.pdf.
[10] http://www.brettspielwelt.de/Hilfe/Anleitungen/Heckmeck/.
[11] http://www.heckmeck-wm.de.

APPENDIX

TABLE VIII
ALL THE 7'600'000 MATCHES

vs.	S1	S2	S3	MC	MC2	MC3	MC4	MCPr5	MCPr10	MCPr25	MCPr-50	MC100It	MC-1000It	MCSF	MC4Pr10	MC4-Pr100	MCR2	MCR4	MCC	MC4C
S1	-	4094	3347	2949	3302	4152	2777	10138	2845	4119	10267	3114	2903	17605	2713	3495	12182	2980	2910	2844
S2	15906	-	9133	8178	9148	10306	7978	14699	8013	9392	14608	8506	8287	19485	7993	8560	15625	8187	8292	7899
S3	16653	10867	-	9064	9833	11372	8827	15337	8883	10301	15405	9381	9099	19558	8888	9480	15980	9075	9062	8961
MC	17051	11822	10936	-	10866	12104	9562	15174	9667	10868	15233	10345	9925	19572	9690	10020	15477	9843	9903	9487
MC2	16698	10852	10167	9134	-	11380	8827	14660	9049	10089	14479	9509	9040	19445	8829	9399	14964	9214	9167	8853
MC3	15848	9694	8628	7896	8620	-	7549	14200	7577	8972	14325	8035	7785	19429	7607	8342	15052	7693	7833	7455
MC4	17223	12022	11173	10438	11173	12451	-	15820	10087	11220	15744	10675	10370	19648	10247	10682	16490	10238	10376	9907
MCPr5	9862	5301	4663	4826	5340	5800	4180	-	4496	4746	9990	4960	4729	16859	4479	5001	12760	4831	4594	4118
MCPr10	17155	11987	11117	10333	10951	12423	9913	15504	-	11188	15440	10435	10099	19616	9999	10244	15802	10022	10228	9734
MCPr25	15881	10608	9699	9132	9911	11028	8780	15254	8812	-	15118	9242	9088	19503	8809	9334	16297	9028	8956	8620
MCPr50	9733	5392	4595	4767	5521	5675	4256	10010	4560	4882	-	4907	4764	16887	4491	4969	12627	4826	4875	4241
MC100It	16886	11494	10619	9655	10491	11965	9325	15040	9565	10758	15093	-	9626	19484	9426	9822	15474	9659	9740	9276
MC1000It	17097	11713	10901	10075	10960	12215	9630	15271	9901	10912	15236	10374	-	19536	9670	10183	15504	9912	10052	9565
MCSF	2395	515	442	428	555	571	352	3141	384	497	3113	516	464	-	399	587	5559	446	438	293
MC4Pr10	17287	12007	11112	10310	11171	12393	9753	15521	10001	11191	15509	10574	10330	19601	-	10197	15862	10240	10220	9705
MC4Pr100	16505	11440	10520	9980	10601	11658	9318	14999	9756	10666	15031	10178	9817	19413	9803	-	15912	9780	9778	9326
MCR2	7818	4375	4020	4523	5036	4948	3510	7240	4198	3703	7373	4526	4496	14441	4138	4088	-	4366	4404	3435
MCR4	17020	11813	10925	10157	10786	12307	9762	15169	9978	10972	15174	10341	10088	19554	9760	10220	15634	-	9920	9583
MCC	17090	11708	10938	10097	10833	12167	9624	15406	9772	11044	15125	10260	9948	19562	9780	10222	15596	10080	-	9508
MC4C	**17156**	**12101**	**11039**	**10513**	**11147**	**12545**	**10093**	**15882**	**10266**	**11380**	**15759**	**10724**	**10435**	**19707**	**10295**	**10674**	**16565**	**10417**	**10492**	-

TABLE IX
SUMMARY OF ALL PROGRAMS

Algo.	Hard rules	Simulations	Cumul	Risk ?	Threshold	Break Cycles	Misc.
S1	Y	N	N	N	-	N	-
S2	Y	N	N	N	-	N	-
S3	Y	N	N	N	-	N	-
MC	N	Y	N	N	-	N	-
MC2	N	Y	N	N	-	N	takes first pickominos on table
MC3	N	Y	N	N	-	N	try to exploit Failure Table
MC4	N	Y	Y	Y	10%	Y	-
MCPr5	N	Y	N	Y	5%	N	-
MCPr10	N	Y	N	Y	10%	N	-
MCPr25	N	Y	N	Y	25%	N	-
MCPr50	N	Y	N	Y	50%	N	-
MC100It	N	Y	N	N	-	N	idem than MC with 100 iterations
MC1000It	N	Y	N	N	-	N	idem than MC with 1000 iterations
MCSF	N	Y	N	N	-	N	makes the sum of failures
MC4Pr10	N	Y	N	Y	10%	N	
MC4Pr100	N	Y	N	Y	100%	N	
MCR2	N	Y	N	N	-	N	rolls again if 4 dice left
MCR4	N	Y	N	N	-	N	rolls again if 2 dice left
MCC	N	Y	Y	N	N	N	-
MC4C	N	Y	Y	Y	10%	Y	-

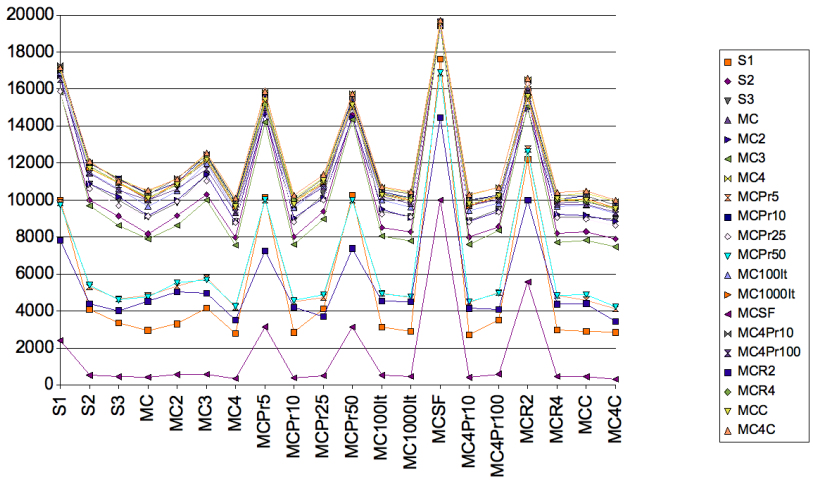


Fig. 8. All Results in one graph