

# **Computer Games Workshop at ECAI 2012**

Chairs : Tristan Cazenave, Jean Méhat, Mark Winands

Monday 27 August 2012

# Beam Nested Rollout Policy Adaptation

Tristan Cazenave<sup>1</sup> and Fabien Teytaud<sup>1,2</sup>

<sup>1</sup> LAMSADE, Université Paris Dauphine, France,

<sup>2</sup> HEC Paris, CNRS, 1 rue de la Libération 78351 Jouy-en-Josas, France

**Abstract.** The Nested Rollout Policy Adaptation algorithm is a tree search algorithm known to be efficient on combinatorial problems. However, one problem of this algorithm is that it can converge to a local optimum and get stuck in it. We propose a modification which limits this behavior and we experiment it on two combinatorial problems for which the Nested Rollout Policy Adaptation is known to be good at.

## 1 Introduction

Recently, the Nested Monte-Carlo Search (NMC) has been proposed for solving combinatorial problems [2, 13]. Based on this algorithm, a new algorithm has been successfully introduced, the Nested Rollout Policy Adaptation algorithm [14]. This algorithm is efficient for numerous combinatorial problems, and in particular, the Traveling Salesman Problem with time Windows [3] and the Morpion Solitaire puzzle [14].

The idea behind the NMC algorithm can be seen as a Meta Monte-Carlo algorithm. This is a recursive algorithm. The first level of the search consists in simply performing a Monte-Carlo simulation, i.e. each decision is chosen randomly until no more possible decision are available. At the end, the score of the position that has been reached is sent back. This first level is called the level 0. For each other level  $lvl > 0$ , the search consists in launching a NMC algorithm with a level  $lvl - 1$  for each possible decision. Such as for the level 0, the score for each reached position is sent back, and the decision with the best score is chosen. This algorithm is presented in Section 3. The NRPA algorithm is based on this idea, except that a level 0 policy is learned by gradient ascent and is used instead of the Monte-Carlo policy. This algorithm is presented in Section 4. One problem of this algorithm is that it can converge to local optima due to its simple learning. In this work, we propose a modification in order to improve the behavior of the NRPA algorithm in front of local optima. The principle of the modification consists in keeping a beam of different sequences (with their corresponding policies).

The paper is organized as follows. The next section (Section 2) presents the two problems studied in this work, the Traveling Salesman Problem with Time Windows in Section 2.1 and the Morpion-Solitaire puzzle in Section 2.2. In Section 3, the Nested Monte-Carlo Search is presented, in Section 4 we present the Nested Rollout Policy Adaptation algorithm, and in Section 5 the improvement done on the NRPA algorithm. Finally, in Section 6 we present comparisons between the NRPA algorithm and the algorithm designed in this work.

## 2 Problems

In this Section we present two well-known combinatorial problems. The first one, presented in Section 2.1 is the Traveling Salesman Problem with Time Windows. The second problem is the puzzle called Morpion-Solitaire and is presented in Section 2.2. The NRPA algorithm has been already used for solving these two problems [14, 3].

### 2.1 The Traveling Salesman Problem with Time Windows

The Traveling Salesman Problem (TSP) is a well-known logistic problem. Given a list of cities and their pairwise distances, the goal is to find the shortest possible path that visits each city only once. The path has to start and finish at a given depot. The TSP problem is NP-hard [9]. The Traveling Salesman Problem with Time Windows (TSPTW) is a problem based on the TSP. Inputs are the same, but a difficulty is added. In this version, a time interval is defined for each city, and each city has to be visited within its corresponding period of time.

Formally, the TSPTW can be defined as follows. Let  $G$  be an undirected complete graph.  $G = (N, A)$ , where  $N = 0, 1, \dots, n$  corresponds to a set of nodes and  $A = N \times N$  corresponds to the set of edges between the nodes. The node 0 corresponds to the depot. Each city is represented by the  $n$  other nodes. A cost function  $c : A \rightarrow \mathbb{R}$  is given and represents the distance between two cities. A solution to this problem is a sequence of nodes  $P = (p_0, p_1, \dots, p_n)$  where  $p_0 = 0$  and  $(p_1, \dots, p_n)$  is a permutation of  $[1, N]$ . Set  $p_{n+1} = 0$  (the path must finish at the depot), then the goal is to minimize the function defined in Equation 1.

$$cost(P) = \sum_{k=0}^n c(a(p_k, p_{k+1})) \quad (1)$$

As said previously, the TSPTW version is more difficult because each city  $i$  has to be visited in a time interval  $[e_i, l_i]$ . This means that a city  $i$  has to be visited before  $l_i$ . It is possible to visit a city before  $e_i$ , but in that case, the new departure time becomes  $e_i$ . Consequently, this case may be dangerous as it generates a penalty. Formally, if  $r_{p_k}$  is the real arrival time at node  $p_k$ , then the departure time  $d_{p_k}$  from this node is  $d_{p_k} = \max(r_{p_k}, e_{p_k})$ .

In the TSPTW, the function to minimize is the same as for the TSP (Equation 1), but a set of constraints is added and must be satisfied. Let us define  $\Omega(P)$  as the number of violated windows constraints by tour (P).

Two constraints are defined. The first constraint is to check that the arrival time is lower than the fixed time. Formally,

$$\forall p_k, r_{p_k} < l_{p_k}.$$

The second constraint is the minimization of the time lost by waiting at a city. Formally,

$$r_{p_{k+1}} = \max(r_{p_k}, e_{p_k}) + c(a_{p_k, p_{k+1}}).$$

With algorithms used in this work, paths with violated constraints can be generated. As presented in [13], a new score  $Tcost(p)$  of a path  $p$  can be defined as follows:

$$Tcost(p) = cost(p) + 10^6 * \Omega(p),$$

with, as defined previously,  $cost(p)$  the cost of the path  $p$  and  $\Omega(p)$  the number of violated constraints.  $10^6$  is a constant chosen high enough so that the algorithm first optimizes the constraints.

A survey of efficient methods for solving the TSPTW can be found in [10]. Existing methods for solving the TSPTW are numerous. First, branch and bound methods were used [1, 4]. Later, dynamic programming based methods [6], heuristics based algorithms [15, 8] and methods based on constraint programming [7, 11] have been published. More recently, ant colony optimization algorithms have been used [10] and have established new state of the art scores. Works based on the NMC have been proposed in [13] and on the NRPA in [3].

## 2.2 Morpion-Solitaire

Morpion-Solitaire is an NP-hard pencil-and-paper puzzle played on a square grid. A move consists in adding a circle (on one possible intersection on the grid) such that a line containing five circles can be drawn. The new line is then added to the grid. Lines can be horizontal, vertical or diagonal. The initial grid contains some starting circles, as shown in Figure 1. Two versions of this puzzle exist, the touching version and the disjoint version. In this paper, we are interested in the first one, the disjoint version, for which a circle can not belong to two lines that have the same direction. The best human score for this version of the puzzle is 68 moves [5]. The Nested Monte-Carlo search found a score of 80 moves [2], and [14] found a new record with 82 moves.

## 3 Nested Monte-Carlo Search

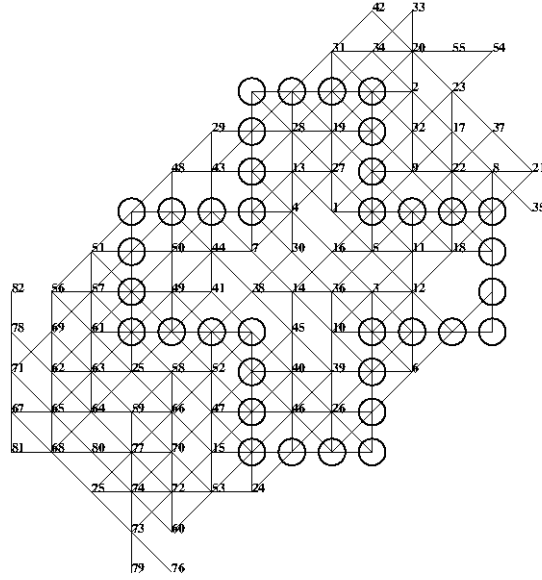
The basic idea of Nested Monte-Carlo Search is to perform a principal playout with a bias on the selection of each decision based on the results of a Monte-Carlo tree search [2].

The base level of the search build random solutions (i.e. playouts), random decision are chosen until the end at this level. When a solution is completely built, the score of the position that has been reached is sent back.

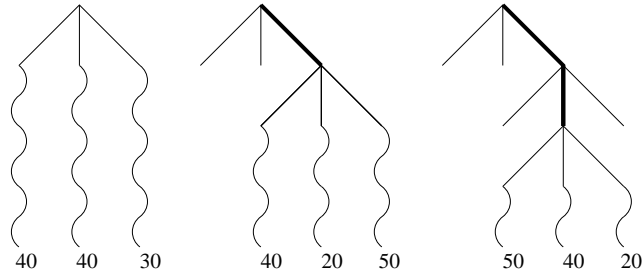
At each decision of a playout of level 1 it chooses the decision that gives the best score when followed by a random playout. Similarly for a playout of level  $n$  it chooses the decision that gives the best score when followed by a playout of level  $n - 1$ .

When a search at the highest level is finished and there is time left, another search is performed at the highest level, and so on until the thinking time is elapsed.

Nested Monte-Carlo search has been successful in establishing world records in single player games such as Morpion Solitaire or SameGame [2]. It provides a good balance between exploration and exploitation and it automatically adapts its search behavior to the problem at hand without parameters tuning.



**Fig. 1.** Example of a puzzle. Circles represent initial points and numbers represent the moves. This 82 moves grid found by our algorithm equalizes the world record established by Rosin [14] through a different solution.



**Fig. 2.** At each step of the principal playout shown here with a bold line, an NMC of level  $n$  performs a NMC of level  $n - 1$  (shown with wavy lines) for each available decision and selects the best one. At level 0, a simple pseudo-random playout is used.

Figure 2 illustrates a level 1 Nested Monte-Carlo search. Three selections of cities at level 1 are shown. The leftmost tree shows that, at the root, all possible cities are tried and that for each possible decision a playout follows it. Among the three possible cities at the root, the rightmost city has the best result of 30, therefore this is the first decision played at level 1. This brings us to the middle tree. After this first city choice, playouts are performed again for each possible city following the first choice. One of the cities has result 20 which is the best playout result among his siblings. So the algorithm continues with this decision as shown in the rightmost tree. This algorithm is presented in Algorithm 1.

---

**Algorithm 1** Nested Monte-Carlo search

---

```

nested(level,node)
  if level==0 then
    ply  $\leftarrow$  0
    seq  $\leftarrow$  {}
    while num_children(node) > 0 do
      CHOOSE seq[ply]  $\leftarrow$  child i with probability 1/num_children(node)
      node  $\leftarrow$  child(node,seq[ply])
      ply  $\leftarrow$  ply+1
    end while
    RETURN (score(node),seq)
  else
    ply  $\leftarrow$  0
    seq  $\leftarrow$  {}
    best_score  $\leftarrow$   $\infty$ 
    while num_children(node) > 0 do
      for children i of node do
        temp  $\leftarrow$  child(node,i)
        (results,new)  $\leftarrow$  nested(level-1,temp)
        if results < best_score then
          best_score  $\leftarrow$  results
          seq[ply]=i
          seq[ply+1..]=new
        end if
      end for
      node=child(node,seq[ply])
      ply  $\leftarrow$  ply+1
    end while
    RETURN (best_score,seq)
  end if

```

---

At each choice of a playout of level 1 it chooses the city that gives the best score when followed by a single random playout. Similarly for a playout of level  $n$  it chooses the city that gives the best score when followed by a playout of level  $n - 1$ .

## 4 The Nested Rollout Policy Adaptation algorithm

The Nested Rollout Policy Adaptation algorithm (NRPA) is an algorithm that learns a playout policy. There are different levels in the algorithm. Each level is associated to the best sequence found at that level. The playout policy is a vector of weights that are used to calculate the probability of choosing a city. A city is chosen proportionally to the exponential of its associated weight. Learning the playout policy consists in increasing the weights associated to the best cities and decreasing the weights associated to the other cities. The algorithm is given in Algorithm 2.

---

### Algorithm 2 Nested Rollout Policy Adaptation

---

```

NRPA (level, pol)
if level = 0 then
    node  $\leftarrow$  root
    ply  $\leftarrow$  0
    seq  $\leftarrow$  {}
    while there are possible decisions do
        CHOOSE seq[ply]  $\leftarrow$  child i the with probability proportional to  $\exp(\text{pol}[\text{code}(\text{node}, i)])$ 
        node  $\leftarrow$  child(node, seq [ply])
        ply  $\leftarrow$  ply + 1
    end while
    return (score (node), seq)
else
    bestScore  $\leftarrow$   $\infty$ 
    for N iterations do
        (result, new)  $\leftarrow$  NRPA (level - 1, pol)
        if result  $\leq$  bestScore then
            bestScore  $\leftarrow$  result
            seq  $\leftarrow$  new
        end if
        pol  $\leftarrow$  Adapt(pol, seq)
    end for
end if
    return (bestScore, seq)

Adapt (pol, seq)
node  $\leftarrow$  root
pol'  $\leftarrow$  pol
for ply  $\leftarrow$  0 to length(seq) - 1 do
    pol'[code(node, seq[ply])] += Alpha
    z  $\leftarrow$  SUM  $\exp(\text{pol}[\text{code}(\text{node}, i)])$  over node's children i
    for children i of node do
        pol'[code(node, i)] -= Alpha  $\times \exp(\text{pol}[\text{code}(\text{node}, i)]) / z$ 
    end for
    node  $\leftarrow$  child(node, seq [ply])
end for
    return pol'

```

---

## 5 The Beam Nested Rollout Policy Adaptation algorithm

The idea of Beam Nested Rollout Policy Adaptation is to combine a beam search with the Nested Rollout Policy Adaptation algorithm. Instead of memorizing one sequence at each level of the algorithm, a set of the best sequences is memorized at each level. The size of the beam for a given level is the number of sequences in the set of this level. Note that the sequences are not memorized alone. Each memorized sequence is associated to a score and a policy. The algorithm is given in Algorithm 3. In the algorithm  $r$  is a score,  $s$  is a sequence and  $p$  is a policy.

As can be seen in the algorithm, a recursive call is performed for each sequence in the set of best sequences for each level. At the end of the algorithm a set of the best sequences and the associated policies and scores is returned. This set is used to adapt the policies at the upper level and these adapted policies are inserted in the set of best sequences at the upper level. When all the sequences coming from the calls at the lower level have been inserted, only the  $B$  best ones are kept ( $B$  being the size of the beam at that level).

The Adapt function that learns the policy is the same as in the original NRPA algorithm.

---

### Algorithm 3 Beam Nested Rollout Policy Adaptation

---

```

beamNRPA (level, pol)
if level = 0 then
    node  $\leftarrow$  root
    ply  $\leftarrow$  0
    seq  $\leftarrow$  {}
    while there are possible decisions do
        CHOOSE seq[ply]  $\leftarrow$  child i the with probability proportional to  $\exp(\text{pol}[\text{code}(\text{node}, i)])$ 
        node  $\leftarrow$  child(node, seq [ply])
        ply  $\leftarrow$  ply + 1
    end while
    return (score (node), seq, pol)
else
    beam  $\leftarrow$  {( $\infty$ , {}, pol)}
    for N iterations do
        newBeam  $\leftarrow$  {}
        for (r, s, p) in beam do
            insert (r, s, p) in newBeam
            beam1  $\leftarrow$  beamNRPA (level - 1, p)
            for (r1, s1, p1) in beam1 do
                p1  $\leftarrow$  Adapt(p, s1)
                insert (r1, s1, p1) in newBeam
            end for
        end for
        beam  $\leftarrow$  B best scores of newBeam
    end for
    return beam
end if

```

---

## 6 Experimental Results

We apply the beam NRPA algorithm to two applications, the TSPTW, presented in Section 2.1 and the Morpion-Solitaire puzzle, presented in Section 2.2. Results are presented respectively in Section 6.1 and in Section 6.2. We define the complexity of the algorithm as the total number of evaluations (rollout) done by the algorithm. Formally, for the beam NRPA algorithm, the complexity is

$$C = (N * B)^{lvl}$$

with  $B$  the size of the beam,  $lvl$  the level of the algorithm and  $N$  the number of iterations done for the learning. Experimentally, we have found that having a beam size  $B > 1$  only for the level 1 was the best choice in terms of complexity. The complexity becomes then

$$C = N^{lvl} * B$$

. For all our experiments, the size of the beam is fixed to 1 for all levels above 1 and is changed at level 1. Consequently, increasing the complexity comes to increase the size of the beam at level 1. In order to have comparable complexities for both beam NRPA and NRPA algorithms, we repeat the NRPA algorithm  $B$  times, and we take the best value found during the  $B$  runs as the return value of the algorithm.

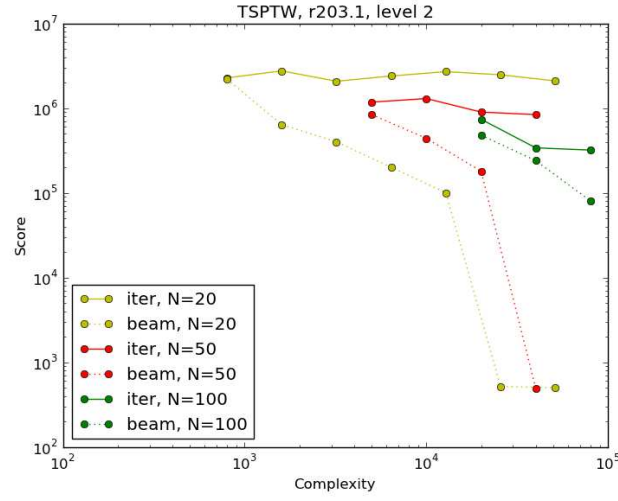
### 6.1 Traveling Salesman Problem with Time Windows

In a first experiment on the TSPTW, we compare the best score found by the two algorithms on two fixed problems from the set of problems from [12]. The two problems are the problem rc203.1, which is a simple one, with 19 cities, and the rc202.3, which has 29 cities and is then harder. We measure the average traveling score as a function of  $C$ . We experiment  $N = \{20, 50, 100\}$  and  $B = \{2, 4, 8, 16, 32, 64\}$  for  $N = 20$ ,  $B = \{2, 4, 8, 16\}$  for  $N = 50$  and  $B = \{2, 4, 8\}$  for  $N = 100$ . Results for the problem rc203.1 are presented in Figure 3. We experiment three different values of  $N$  in level 2. The beam NRPA is always better than the classic algorithm for all complexities (i.e., for all different sizes of beam).  $N = 20$  and  $N = 50$  for the beam algorithm are the only versions that are able to find valid paths (i.e. without violated constraints).

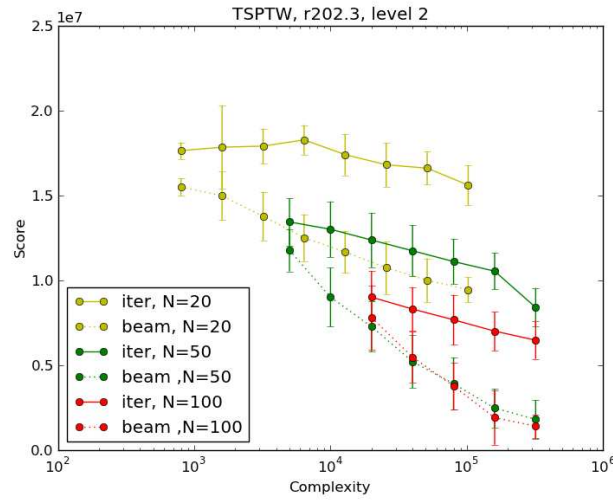
For the second problem (rc202.3), results are presented in Figure 4. Here again, it is always better to use the beam NRPA algorithm. We can note that, because this problem is harder, a larger value of  $N$  is needed, meaning that more time need to be spent during the learning phase. Best results are found with the beam NRPA algorithm with  $N = 50$  and  $N = 100$ .

The last experiment on the TSPTW, is to run the beam NRPA algorithm on all problems from the set of problems from [12], and to compare our results with the results found by the NRPA algorithm from [3]. Results are presented in Table 1.

As expected, we can see that the beam NRPA algorithm is always able to find better scores than the NRPA algorithm. The beam NRPA is able to find 63% of state of the art scores, and this without any expert knowledge. Expert knowledge can be added to the beam NRPA algorithm, in the same way as in the NRPA version from [3].



**Fig. 3.** Experience on the problem rc203.1 with level 2. The lower the better. Average on 30 runs. Best results are found by the beam NRPA algorithm with  $N = 20$ . In this experiment, only the beam NRPA algorithm is able to find a valid path, without violated constraints. The best known score for this problem is 453.48. This score is reached for the Beam NRPA with  $N = 20$ .



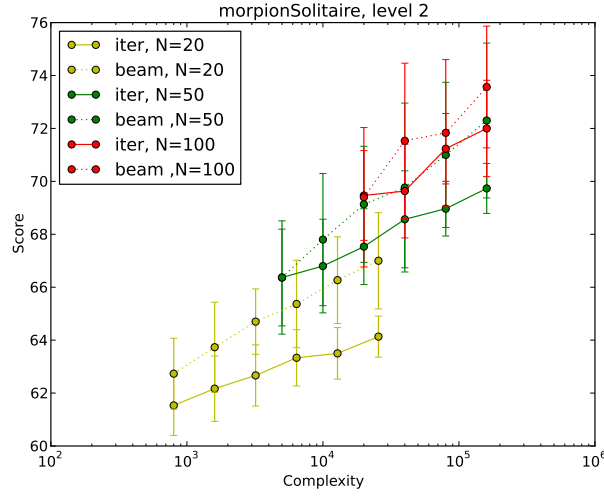
**Fig. 4.** Experience on the problem rc202.3 with level 2. The lower the better. Average on 30 runs. Best results are found by the beam NRPA algorithm with  $N = 50$  and  $N = 100$ . The best known score for this problem is 837.72

Problem	City	State of the art	NRPA	beam NRPA
rc206.1	4	117.85	<b>117.85</b>	<b>117.85</b>
rc207.4	6	119.64	<b>119.64</b>	<b>119.64</b>
rc202.2	14	304.14	<b>304.14</b>	<b>304.14</b>
rc205.1	14	343.21	<b>343.21</b>	<b>343.21</b>
rc203.4	15	314.29	<b>314.29</b>	<b>314.29</b>
rc203.1	19	453.48	<b>453.48</b>	<b>453.48</b>
rc201.1	20	444.54	<b>444.54</b>	<b>444.54</b>
rc204.3	24	455.03	<b>455.03</b>	<b>455.03</b>
rc206.3	25	574.42	<b>574.42</b>	<b>574.42</b>
rc201.2	26	711.54	<b>711.54</b>	<b>711.54</b>
rc201.4	26	793.64	<b>793.64</b>	<b>793.64</b>
rc205.2	27	755.93	<b>755.93</b>	<b>755.93</b>
rc202.4	28	793.03	800.18	<b>793.03</b>
rc205.4	28	760.47	765.38	765.38
rc202.3	29	837.72	839.58	839.58
rc208.2	29	533.78	537.74	<b>533.78</b>
rc207.2	31	701.25	702.17	702.17
rc201.3	32	790.61	796.98	795.43
rc204.2	33	662.16	673.89	663.19
rc202.1	33	771.78	775.59	772.17
rc203.2	33	784.16	<b>784.16</b>	798.73
rc207.3	33	682.40	688.50	<b>682.40</b>
rc207.1	34	732.68	743.72	<b>732.68</b>
rc205.3	35	825.06	828.36	<b>825.06</b>
rc208.3	36	634.44	656.40	649.93
rc203.3	37	817.53	820.93	<b>817.53</b>
rc206.2	37	828.06	829.07	842.17
rc206.4	38	831.67	831.72	<b>831.67</b>
rc208.1	38	789.25	799.24	795.57
rc204.1	46	868.76	883.85	878.76

**Table 1.** Results on all problems from the set from Potvin and Bengio [12]. First Column corresponds to the problem, second column is the number of cities, third column is the state of the art score, found in [10]. Fourth column is the best score found by the NRPA algorithm in [3] and fifth column is the best score found by the beam NRPA algorithm. The problems for which we find the state of the art solutions are in bold. With the beam NRPA 63% of state of the art scores are found, where as with the classic NRPA algorithm only 43% state of the art scores are found.

## 6.2 Morpion-Solitaire

The second experimented application is the Morpion-Solitaire puzzle. As for the two first experiments on the TSPTW, we measure the best score found by the beam NRPA and the NRPA algorithms as a function of the complexity in level 2. For this application, the higher scores the better. Results are presented in Figure 5. For beam sizes larger than 2, results are always better for the beam NRPA algorithm. For  $B = 2$ , results are equivalent.



**Fig. 5.** Experience on the Morpion-Solitaire puzzle with level 2. The higher the better. Best results are found by the beam NRPA algorithm with  $N = 100$ . Each point is an average of 30 runs.

## 7 Conclusion

In this work we show how to improve the Nested Rollout Policy Adaptation algorithm. For both applications, results are good for a beam size of 4. When the size of the beam increases, results are even better. On the first experimented application, the traveling salesman problem with time windows, we do not use any expert knowledge. Our goal was then, not to find new records, but to show the efficiency of having numerous learned policies. The classic NRPA algorithm find 43% of state of the art records, whereas the beam NRPA algorithm is able to find 63% of records. Only for 2 problems we are not able to find equal or better scores than the NRPA algorithms. For all other problems, scores are equal or better for the beam NRPA algorithm. On the Morpion Solitaire puzzle, we reach the current record (82 moves), but we are not able to beat it. However, as shown in Figure 5 best scores are found faster with the beam algorithm than with

the classic NRPA algorithm. This behaviour has been also observed for the traveling salesman problem with time windows (Figures 4 and 3).

As pointed out in the future works of the NRPA algorithm’s author in [14], realizing a parallel version of the NRPA algorithm is a challenging work. The beam NRPA algorithm has the advantage to be easily parallelizable, because, all policies from the beam can be evaluated in parallel.

An interesting future work is to keep distances between all the sequences from the beam. Having such a modification should be much more robust in front of local optima.

## References

1. E.K. Baker, ‘An exact algorithm for the time-constrained traveling salesman problem’, *Operations Research*, **31**(5), 938–945, (1983).
2. T. Cazenave, ‘Nested Monte-Carlo search’, in *IJCAI*, pp. 456–461, (2009).
3. T. Cazenave and F. Teytaud, ‘Application of the nested rollout policy adaptation algorithm to the traveling salesman problem with time windows’, in *LION 6*. Springer, (2012).
4. N. Christofides, A. Mingozzi, and P. Toth, ‘State-space relaxation procedures for the computation of bounds to routing problems’, *Networks*, **11**(2), 145–164, (1981).
5. E.D. Demaine, M.L. Demaine, A. Langerman, and S. Langerman, ‘Morpion solitaire’, *Theory of Computing Systems*, **39**(3), 439–453, (2006).
6. Y. Dumas, J. Desrosiers, E. Gelinass, and M.M. Solomon, ‘An optimal algorithm for the traveling salesman problem with time windows’, *Operations Research*, **43**(2), 367–371, (1995).
7. F. Focacci, A. Lodi, and M. Milano, ‘A hybrid exact algorithm for the tsptw’, *INFORMS Journal on Computing*, **14**(4), 403–417, (2002).
8. M. Gendreau, A. Hertz, G. Laporte, and M. Stan, ‘A generalized insertion heuristic for the traveling salesman problem with time windows’, *Operations Research*, **46**(3), 330–335, (1998).
9. D.S. Johnson and C.H. Papadimitriou, *Computational complexity and the traveling salesman problem*, Mass. Inst. of Technology, Laboratory for Computer Science, 1981.
10. Manuel López-Ibáñez and Christian Blum, ‘Beam-ACO for the travelling salesman problem with time windows’, *Computers & OR*, **37**(9), 1570–1583, (2010).
11. G. Pesant, M. Gendreau, J.Y. Potvin, and J.M. Rousseau, ‘An exact constraint logic programming algorithm for the traveling salesman problem with time windows’, *Transportation Science*, **32**(1), 12–29, (1998).
12. J.Y. Potvin and S. Bengio, ‘The vehicle routing problem with time windows part II: genetic search’, *INFORMS journal on Computing*, **8**(2), 165, (1996).
13. A. Rimmel, F. Teytaud, and T. Cazenave, ‘Optimization of the nested monte-carlo algorithm on the traveling salesman problem with time windows’, *Applications of Evolutionary Computation*, 501–510, (2011).
14. Christopher D. Rosin, ‘Nested rollout policy adaptation for monte carlo tree search’, in *IJCAI*, pp. 649–654, (2011).
15. M.M. Solomon, ‘Algorithms for the vehicle routing and scheduling problems with time window constraints’, *Operations Research*, **35**(2), 254–265, (1987).

# A new self-acquired knowledge process for Monte Carlo Tree Search

André Fabbri, Frédéric Armetta, Éric Duchêne and Salima Hassas

Laboratoire GAMA

**Abstract.** Computer Go is one of the most challenging field in Artificial Intelligence Game. In this area the use of Monte Carlo Tree Search has emerged as a very attractive research direction, where recent advances have been achieved and permitted to significantly increase programs efficiency.

These enhancements result from combining tree search used to identify best next moves, and a Monte Carlo process to estimate and gradually refine a position accuracy (estimation function). The more the estimation process is accurate, the better the Go program performs.

In this paper, we propose a new approach to extract knowledge from the Go tree search which allows to increase the evaluation function accuracy (BHRF: Background History Reply Forest). The experiments results provided by this new approach are very promising.

## 1 Introduction

The game of Go is a deterministic fully observable two player game. Despite its simple rules, it stands for one of the great challenge in the field of AI Game.

Go tactical and strategic decisions are extremely difficult to tackle, since each move could have a high impact a long time after having been played. A single stone move could completely change the board configuration and therefore, there is no static evaluation function available during the game [22]. Professional human players succeed where programs fail, thanks to their expertise and knowledge acquired from their long experience.

Recent developments in Monte Carlo Tree Search allowed to considerably increase program's efficiency. These enhancements result from combining Tree Search used to identify best next moves to a Monte Carlo process based on random playing attempts to estimate and gradually refine a position accuracy (estimation function).

Monte Carlo Tree Search programs are able to reach up a professional level by simulating thousands of pseudo random games [16]. However these program's performances does not scale with computational power increasing [2].

A promising way to increase program's efficiency arises from learning on-line local knowledge to enhance the relevance of random simulations used to evaluate positions [1,17,18].

In this context, one can note that the issue is not only focused on the computing power available, but in our ability to manage additional more sophisticated

self-acquired knowledge. Then, in the rest of the paper, we will focus on this ability. Further work will propose additional enhancements to optimize the process associated to knowledge management in order to decrease the computing-resources consumption.

In this paper we propose a new approach to collect the knowledge acquired through the tree search of estimated positions. Our proposal appears as a complementary data structure. As described in section 3, a tree search is used to define what are the best next moves to play from a completely described current position. The structure we propose to build addresses a more abstract question: if a set of relatives moves are played, what to play next ? We show with our approach presented in section 4 that a natural manner to tackle this question is to constitute a Background History Reply Forest (BHRF). We then describe how to build and maintain this generic forest and how to exploit it. These two points raise new questions, but we show that a first straightforward setup produces good results. Some promising experimental results are presented in 5 and show that using BHRF allows to play better.

The next section presents the general structure of knowledge involved in a learning process. The section 3 focus on the kind of knowledge used in the existing Go programs. Section 4 introduces a Background History Reply Forest (BHRF) for the Monte Carlo Tree Search. The last section sums up experiments carried on the model. A conclusion is presented in 6.

## 2 Knowledge in computer Go

While a game is running, for each step, a Go player aims at selecting the best move according to the game overall state. The overall game state involves any information related to the current game such as the board setup, the history of played moves, local fights and also the opponent's level. During the decision process, players need to select the relevant parts of the game to focus on. Human players accumulate knowledge by studying standard techniques, watching professional games or interacting with other players. Computer programs encode directly Go expert knowledge (apriori knowledge) or create their own knowledge using machine learning technic. This information is evaluated and stored in a data structure (knowledge acquisition or *learning*) for a further exploitation (knowledge exploitation or *planning*). Reinforcement learning methods iteratively apply these two steps to progressively build an effective policy [21,19].

### 2.1 Go knowledge acquiring

Knowledge is useful to evaluate a position (current state or targeted state resulting from a set of planned moves). Either generic (fitting many cases) or specific knowledge could be used to do this evaluation.

For instance, understanding the shapes formed by the stones on a Goban is a relevant way to solve local fights. In this case, local patterns are a powerful way to encode Go expert knowledge [19,12].

An other way is to consider a knowledge based on high-level characteristics of the game: groups, groups stability, influences between groups, etc [11]. The representation can then focus on the intersection around the last move played to find local replies [22].

The value of this knowledge generally represents the probability to win if we reach this game state representation. The quality of the estimator determines the accuracy of the considered knowledge. The value given by poor estimators are not reliable and have to be considered with caution. Several estimators for the same knowledge can be combined to produce a more effective estimation. The knowledge values are computed based on professional game records [4,20], professional players comments [13] or self-playing [19]. These knowledge values will be exploited to choose an action according to the current game state.

## 2.2 Go knowledge exploitation

The policy maps an action to each game state. The action is selected based on the knowledge matching that game state. A high knowledge value means a good move but the knowledge nature and its estimator might be misleading. Therefore uncertainty is generally inserted in the policy to mitigate the influence of wrong estimations. The  *$\epsilon$ -greedy* policy selects with a probability  $\epsilon$  the action associated to the highest knowledge value otherwise it generally plays a random move. The *softmax* policy selects each action with a probability depending on the associated knowledge value [21].

A policy would be selected according to its context of application. During a tutorial process, a policy might tolerate exploratory moves but during a challenge the policy has to select the best possible action.

## 2.3 Reinforcement learning

Reinforcement learning is a way to manage learning in an autocatalytic way. The program is not taught what to do and learns therefore by its own experience [21]. *Temporal difference learning* is one of the most applied method for such problems and has been successfully applied to computer games. This method dynamically bootstraps its knowledge from simulated games.

The policy evolves as the simulations runs and will influence the incoming simulations. Since the knowledge is built on-line, the policy has to deal with exploiting the accumulated knowledge (*exploitation*) or gathering more knowledge (*exploration*). A good policy should produce accurate simulations to enable the learning process. A too strong policy can actually lead to a weaker program [21].

$TD(\lambda)$  methods reinforce their knowledge values according to the estimation difference of the game state between two time-step. A Monte Carlo method is a *temporal difference* method which reinforcement depends on the final game state of the simulation [19].

### 3 Monte Carlo Go programs

The main idea of Monte Carlo Tree Search is to build a tree of possible sequence of actions. The tree root corresponds to the current board situation and each child node is a possible future game state. The tree will be progressively expanded towards the most promising situation by repeating the 4 phases: *descent*, *roll-out*, *update* and *growth* (Fig.1).

During the *descent* phase, the simulation starts from the root node and progresses through the tree until it reaches a game state outside of the tree.

For each node the program will iteratively select the best action with respect to the *descent* policy. Once it leaves the Monte Carlo Tree, the *roll-out* phase generates the remaining moves according to the *roll-out* policy until the game reaches a final state. The *update* phase propagates the final game results in the node reached during the *descent* and the *growth* phase appends the first game situation outside of the tree to the overall structure [5].

Monte Carlo Go programs involve two kinds of knowledge to guide the simulations from the current game state. The Monte Carlo Tree stores knowledge on-line. This knowledge is exploited during the *descent* phase. The current best programs exploit Go domain specific knowledge in the *roll-out* phase because the tree knowledge is no more available. In the last section we will present methods that attempt to dynamically build up knowledge for the *roll-out* phase.

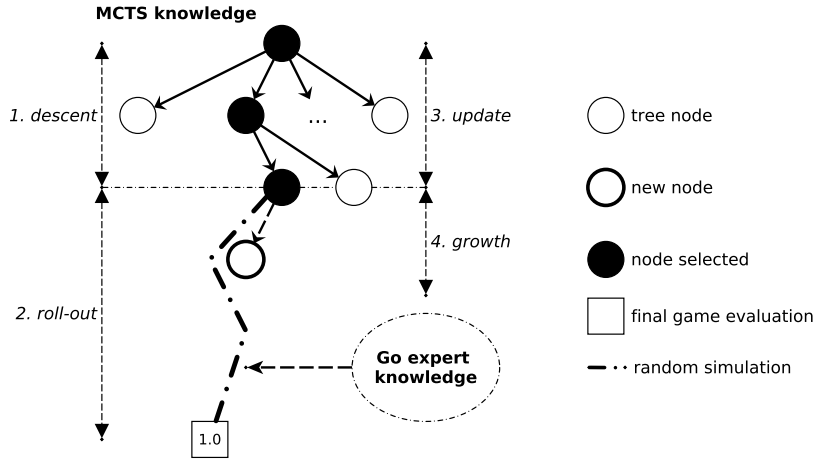


Fig. 1. General MCTS process

### 3.1 Monte Carlo tree

In the Monte Carlo tree, each node is a knowledge piece that associates a possible future board setups with an accuracy value estimated from previous Monte Carlo simulations. Board setups are very precise game state representations which will be progressively selected as the simulations run. However the built tree depends on the current board situation. Local structures are not shared between branches leading to a new kind of horizon effect [6] and at each new turn only a subtree is kept. All previously acquired knowledge in other branches is forgotten.

At each step of the *descent* phase, the policy selects in a greedy way the best action based on the child's node value or a default value for the resulting states outside of the tree [22]. The node's estimator will have a huge influence on the policy's behaviour. Since the policy is not perfect and the estimation sample is too small, the mean of Monte Carlo rewards (MCTS value) is not a reliable indicator on the long-term run [8]. The Upper Confidence bound applied to Tree estimator (UCT value) reveals to be a good trade-off between exploration and exploitation [15]. Recent results show that, in practice, a combination between the MCTS value and a biased mean estimator called RAVE ensures a good exploratory behaviour [10,8] and also minimizes the knowledge lost at each new turn, but without control.

### 3.2 Go a priori knowledge

Contrary to the Monte Carlo tree, the Go expert knowledge is independent from the overall stones disposition and therefore can be exploited for a broader set of a board setup. The purpose of this knowledge is to guide the simulation after having left the tree structure. However the *roll-out* policy should not be too strong to do not disturb the learning process [9]. The knowledge encoded corresponds generally to static Go tactical rules based on the current board configuration. Sequence-like policies brought a substantial improvement by searching around the last opponent move [22]. If no rules fit the current situation, the policy plays randomly. Other *roll-out* policies use a linear combination of local pattern. The weights are generally computed with off-line intensive machine learning [9] but recent works obtained promising results by tuning them on-line [14,19].

### 3.3 Knowledge collected from Game

Several works aim at dynamically learn knowledge relative to the current game in order to exploit it during the *roll-out* phase (see Fig.2). The game chosen state representations are generally small move sequences or patterns. As for the Monte Carlo tree, this knowledge will be built in an autocatalytic manner but this knowledge will persist over the turns.

Pool RAVE [18] exploits the RAVE values in the Monte-Carlo tree to influence the *roll-out*. The best RAVE values from the covered nodes (*descent* phase) are selected to influence the next *roll-out*. The *roll-out* policy proposed for the game of Go uses first a "fill-board heuristic" [3] to ensure a good exploratory

behaviour and then applies the moves associated with one of the pooled RAVE values. Hence the selected RAVE value contributes directly to its own reinforcement.

Contextual Monte Carlo [17] stores tiles of two moves played by the same player. Each tiles has Monte Carlo mean value updated when the two moves appears in the same simulation. The main idea is to link the expected success with a couple of moves played by the same player. During the *roll-out* phase a  $\epsilon$ -greedy policy will select the best move according to the last player's move.

Last Good Reply Forgetting heuristic [1] associates a reply to each sequence of one or two consecutive moves. For each encountered sequence the program stores a single reply. In the *update* phase all the reply sequences are updated or forgotten according to the simulation result. Over the simulations the replies are frequently updated but the most persisting moves are spatially local replies [6,1]. In the *roll-out* phase the policy successively tries to apply the reply to the two previous ones, the reply to the previous move or a move generated from a Go expert knowledge policy if no appropriate reply is stored.

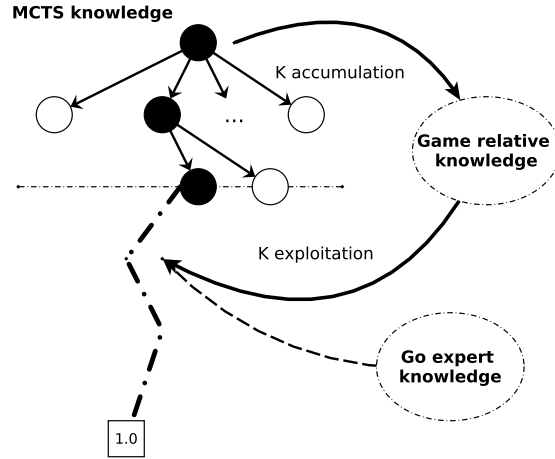


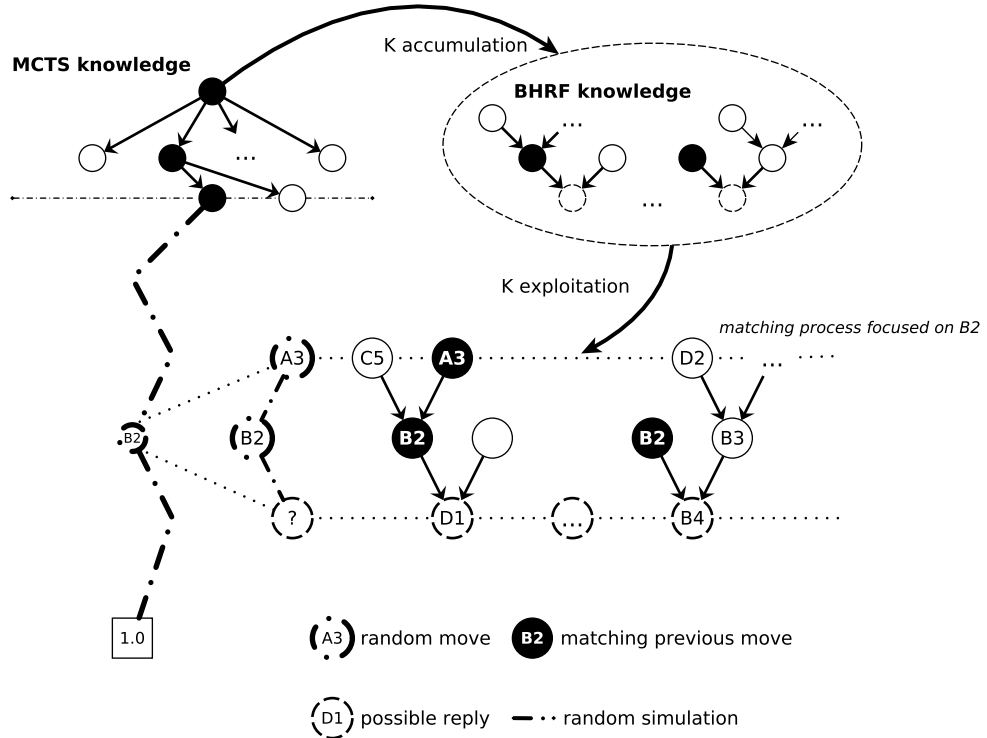
Fig. 2. Game relative knowledge for MCTS process

## 4 Proposal

In this paper, we propose a new method to learn knowledge collected from game (game-based knowledge) persistent over the turns. We actually assume that we can extract knowledge from the Monte Carlo tree and reuse it during the *roll-out* phase. Therefore we build an independent data structure similar to the Monte Carlo tree to store game based knowledge. The *roll-out* policy will be modified to consider this new form of knowledge.

#### 4.1 Background History Reply Forest

A Monte Carlo node represents a future possible board setup. If we consider the background history of the board rather than the position of the stones, a board setup is also the sequence of moves played since the beginning of the match. Hence each child node of the tree is a possible reply to that sequence. Our idea is to extend the LGRF heuristic to catch the knowledge stored in the tree over the simulations. The size of the previous sequence will be lengthened and the reply estimation will be based on the tree values. This knowledge will be more biased than the one provided by Monte Carlo nodes but it will be exploited in the *roll-out* phase and will persist over the turns.



**Fig. 3.** BHRF knowledge exploitation

Hence we build a forest of search trees such that each tree root is a potential reply and each child node a previous sequence to this reply (Fig.3). The value associated to each node comes from the Monte Carlo tree. In the *update* phase all nodes corresponding to sequences chosen in the *descent* phase are reinforced. The search trees are progressively expanded in the *growth* phase as done in

the Monte Carlo tree [5]. This knowledge is then exploited during the *roll-out* phase, according to the previous moves played. The selected nodes with the longest corresponding sequences will be the most dependent on the game state.

## 4.2 BHRF exploitaiton

For each *roll-out* game state, we choose a reply among all legal positions on the board. The program progresses through the associated tree search to select the nodes with the longest previous sequences (up to a maximum size parameter called *depth*). The policy computes on-line the UCT value for the nodes with the same previous sequence according to their last Monte Carlo rewards.

Due to the biased nature of this knowledge and in order not to restrain the simulation diversity, we implemented a non determinist policy to generate the moves at each *roll-out* step. The policy plays the root reply associated with a selected node among those with the longest matching sequences. If no reply move was played by the policy, a default *roll-out* policy is applied.

The *softmax* policy selects a node according to a *softmax* distribution based on their UCT value and plays the associated move with a probability depending on its UCT value multiplied by a parameter  $\alpha$ .

---

**Algorithm 1** *softmax* policy for BHRF knowledge

---

```

 $curLength = 0$ 
for  $m$  in  $legalMoves$  do
     $i = replyTree[m].selectNode(depth, lastmoves)$ 
     $nodeSelection.add(i)$ 
    if  $n.length > curLength$  then
         $curLength = n.length$ 
    end if
end for
 $n = nodeSelection.softMaxUct(curLength)$ 
if  $randomValue() < \alpha \times n.uctValue$  then
    return  $n.rootReply$ 
else
    return  $defaultRandomMove()$ 
end if

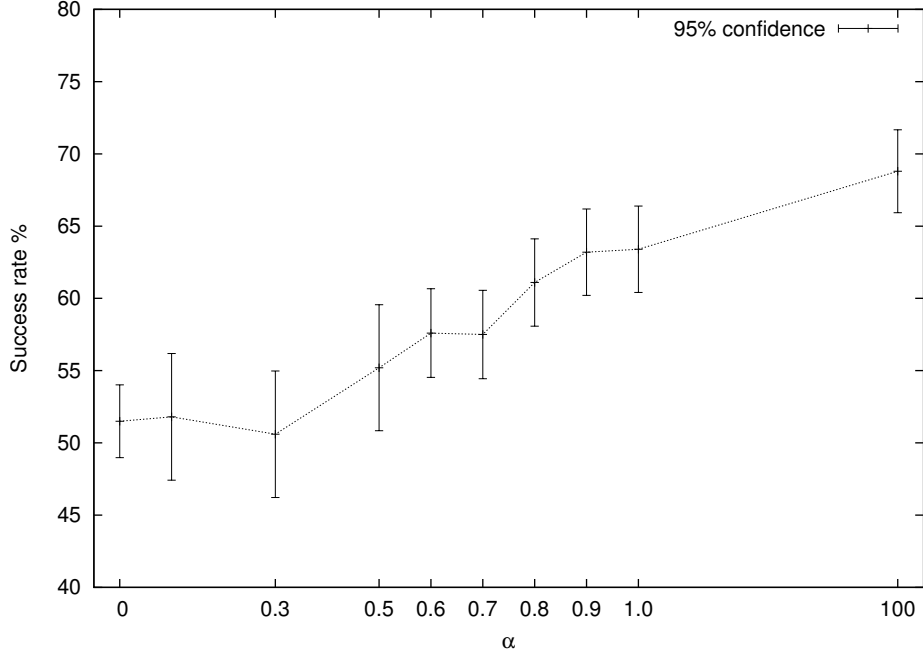
```

---

## 5 Experimental results

We implemented this proposed knowledge-based model on top of the state of art program Fuego [7]. To prove the effectiveness of our approach we tested the BHRF player against a Monte-Carlo player using a random *roll-out* policy. We disabled the expert heuristics involved in the Fuego *roll-out* policy except for

the random move generation<sup>1</sup>. The experimental results were carried out on a 9x9 goban against a baseline program without the BHRF heuristic. The current implementation is not thread-safe and is time-consuming. Therefore to provide a fair comparison we removed the clock limitation and each turn both programs run 10000 Monte-Carlo games on a single thread.

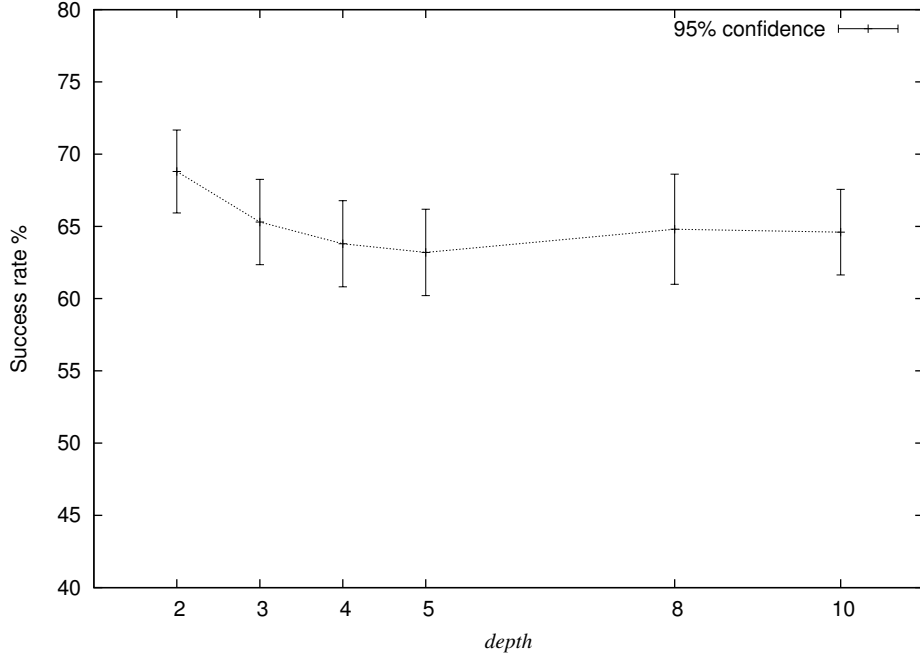


**Fig. 4.** BHRF success rate over  $\alpha$  against the baseline program

We first study the influence of the parameter  $\alpha$  for the *softmax* policy. This parameter tunes the involvement of the BHRF knowledge during the *roll-out* phase (see Algo. 1). As we see in Fig. 4, the performance increases with  $\alpha$ . The program actually performs better for  $\alpha = 100$  where the BHRF replies are applied quite systematically if a BHRF node matches (only 2% doesn't). These results show that the *softmax* policy provides a good simulation diversity by itself.

Hence we next set  $\alpha$  at 100 and focus on the *depth* parameter. The *depth* parameter determines the “history relevance” of the selected knowledge. As the maximum *depth* increases, the suggested moves should be more accurate. The experiments were carried out up to a *depth* of 10 because of time consuming constraints. Further experiments will require an optimised BHRF implementation.

<sup>1</sup> Fuego default random move generation use heuristics to avoid “suicidal” moves



**Fig. 5.** BHRF success rate over *depth* against the baseline program

The figure 5 reveals that the performance of our algorithm slightly decreases as the *depth* parameter grows.

To validate our approach, we compare the performance of the BHRF heuristic according to the number of Monte-Carlo games simulated at each turn. Table 1 shows that BHRF performs better for 10000 simulations search<sup>2</sup>. The resulting Background History Reply Forest for deep Monte Carlo search produces better estimators and actually matches to a wider set of game states. About 90% of the *roll-out* game states benefit from a BHRF reply when the MCTS algorithm runs 10000 games against 80% for 1000 games simulated.

**Table 1.** Comparative between 1000 and 10000 games simulated for the *softmax* policy

$\alpha$	1		100	
games simulated	1000	10000	1000	10000
success rate %	$58.7 \pm 2.49$	$63.4 \pm 2.99$	$58.5 \pm 2.49$	$68.8 \pm 2.87$

These results show an overall improvement when reusing game-based knowledge, that is extracted from the Monte Carlo tree. As we mentioned before, our main concern was to build self-acquired knowledge to improve Monte Carlo

<sup>2</sup> The results are given with a 95 % confidence interval

simulations. The computing time of this algorithm is still higher than the baseline algorithm (up to 10 times) but improvements could be made. Indeed the heuristic searches at each *roll-out* step among all the potential moves through the whole board contrary to the Go expert policy which one focuses around the last played move [22,7]. Furthermore we would like to point out that our main concern was to show the good impact of Game-based knowledge model. An optimised version will requires a deep modification of the Fuego libraries.

## 6 Conclusion

In this paper we present a new approach to complement the Monte Carlo Tree Search programs. The proposed framework is inspired from the reinforcement learning paradigm, and aims at distinguishing incorporated knowledge from its exploitation.

The proposed model extracts game-based knowledge from the Go Tree Search to introduce more accurate moves through simulations. The incorporated knowledge is based on history data and tends to find a trade-off between the specific knowledge stored in the tree and more biased game-relative heuristics.

We show with our approach that a natural manner to complement Go tree search knowledge is to constitute a Background History Reply Forest (BHRF). We then describe how to build and maintain this generic forest and how to exploit it. These two points raise new questions, but we show that a first straightforward setup provides good results.

The first proposed results underline the efficiency of our model, and allows to outperform the baseline MCTS algorithm with similar setting for up to 2/3 of the considered games. This encourages us to carry on the experiments on a larger board or with other settings (longer sequences, use of BHRF to compute initial values of leaf for the Go tree search, etc).

## References

1. Baier, H., Drake, P.: The Power of Forgetting: Improving the Last-Good-Reply Policy in Monte-Carlo go. *IEEE Transactions on Computational Intelligence and AI in Games* 2(4), 303–309 (2010)
2. Bourki, A., Chaslot, G., Coulm, M., Danjean, V., Doghmen, H., Hooek, J., H  rault, T., Rimmel, A., Teytaud, F., Teytaud, O., et al.: Scalability and Parallelization of Monte-Carlo Tree Search. In: *Proceedings of Advance in Computer Games* 13 (2010)
3. Chaslot, G., Fiter, C., Hooek, J., Rimmel, A., Teytaud, O.: Adding expert knowledge and exploration in Monte-Carlo Tree Search. *Advances in Computer Games* pp. 1–13 (2010)
4. Coulom, R.: Computing Elo Ratings of Move Patterns in the Game of Go. In: *Computer Games Workshop, Amsterdam, The Netherlands* (2007)
5. Coulom, R.: Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. *Computers and Games* pp. 72–83 (2007)

6. Drake, P.: The Last-Good-Reply Policy for Monte-Carlo Go. *International Computer Games Association Journal* 32(4), 221–227 (2009)
7. Enzenberger, M., Muller, M., Arneson, B., Segal, R.: FUEGO - an Open-Source Framework for Board Games and Go Engine Based on Monte-Carlo Tree Search. *IEEE Transactions on Computational Intelligence and AI in Games* 2(4), 259–270 (2009)
8. Gelly, S., Kocsis, L., Schoenauer, M., Sebag, M., Silver, D., Szepesvári, C., Teytaud, O.: The Grand Challenge of Computer Go: Monte Carlo Tree Search and Extensions. *Communications of the ACM* 55(3), 106–113 (2012)
9. Gelly, S., Silver, D.: Combining Online and Offline Knowledge in UCT. In: *Proceedings of the 24th international conference on Machine learning*. pp. 273–280. ACM (2007)
10. Gelly, S., Silver, D.: Monte-Carlo Tree Search and Rapid Action Value Estimation in Computer Go. *Artificial Intelligence* (2011)
11. Graepel, T., Goutrie, M., Krüger, M., Herbrich, R.: Learning on Graphs in the Game of Go. *Artificial Neural Networks—ICANN 2001* pp. 347–352 (2001)
12. Helmut, A.M.: Board Representations for Neural Go Players Learning by Temporal Difference. In: *Computational Intelligence and Games, 2007. CIG 2007. IEEE Symposium on*. pp. 183–188. IEEE (2007)
13. Hoock, J., Lee, C., Rimmel, A., Teytaud, F., Wang, M., Teytaud, O.: Intelligent Agents for the Game of Go. *Computational Intelligence Magazine, IEEE* 5(4), 28–42 (2010)
14. Huang, S., Coulom, R., Lin, S.: Monte-Carlo Simulation Balancing in Practice. *Computers and Games* pp. 81–92 (2011)
15. Kocsis, L., Szepesvári, C.: Bandit based Monte-Carlo Planning. *Machine Learning: ECML 2006* pp. 282–293 (2006)
16. Lee, C., Wang, M., Chaslot, G., Hoock, J., Rimmel, A., Teytaud, O., Tsai, S., Hsu, S., Hong, T.: The Computational Intelligence of MoGo Revealed in Taiwan’s Computer Go Tournaments. *Computational Intelligence and AI in Games, IEEE Transactions on* pp. 73–89 (2009)
17. Rimmel, A., Teytaud, F.: Multiple Overlapping Tiles for Contextual Monte Carlo Tree Search. *Applications of Evolutionary Computation* pp. 201–210 (2010)
18. Rimmel, A., Teytaud, F., Teytaud, O.: Biasing Monte-Carlo Simulations through RAVE Values. *Computers and Games* pp. 59–68 (2011)
19. Silver, D., Sutton, R., Müller, M.: Temporal-difference search in computer Go. *Machine Learning* pp. 1–37 (2012)
20. Stern, D., Herbrich, R., Graepel, T.: Bayesian Pattern Ranking for Move Prediction in the Game of Go. In: *Proceedings of the 23rd international conference on Machine learning*. pp. 873–880. ACM (2006)
21. Sutton, R., Barto, A.: *Reinforcement Learning: An Introduction*. Cambridge Univ Press (1998), [online] available at <http://webdocs.cs.ualberta.ca/~sutton/book/ebook/the-book.html>
22. Wang, Y., Gelly, S.: Modifications of UCT and sequence-like simulations for Monte-Carlo Go. In: *IEEE Symposium on Computational Intelligence and Games, Honolulu, Hawaii*. pp. 175–182 (2007)

# Detecting cheating activities in online duplicate Bridge game tournaments: first results

Sylvain Lagrue and Karim Tabia

CRIL UMR CNRS 8188 - Artois University - France

{lagrue, tabia}@cril.univ-artois.fr

<http://www.cril.fr/~lagrue>

<http://www.cril.fr/~tabia>

**Abstract.** Online games have become very popular in recent years and the number of players is steadily increasing. The number of game web sites, tournaments and the underlying economic issues are now considerable. Unlike face-to-face games, online ones offer several cheating methods such as exchanging information through incontrollable channels such as phone, instant messaging or any other communication means. This paper focuses on online duplicate Bridge tournaments where the same deals are played by several players. We argue that anomaly-based approaches, which are widely used in several application domains such as computer security, are very appropriate for detecting potential cheating activities for a number of reasons. We finally provide preliminary experimental evaluations showing the effectiveness of the proposed approaches.

**Keywords:** Online games, duplicate Bridge tournaments, anomaly detection

## 1 Introduction

Online games are very attractive and increasing numbers of players spend lots of time and money in playing. Some online games are played in tournaments like card games. It is very important to mention that most of these games are free or require paid subscriptions, consequently their business model heavily lies on the confidence and the reputation of that online game. If a doubt bears on a given online game that some players can cheat, this will result in catastrophic disinterest and turn away of "honest" players. Moreover, some online tournaments are officially recognized and taken into account for establishing player rankings hence the need to deploy automated means to detect and prevent any cheating attempt. Unfortunately, this problem has not received much interest and only some preliminary works [14] address it.

Unlike face-to-face card games where the communication of information is really restricted and controlled, in online ones, a player may use incontrollable channels to obtain useful information. For example, in an online Bridge tournament where the same deal is played in duplicate on multiple boards, one can play with two distinct accounts (using different logins) and obtain relevant information from

one account to use it in the second account. Checking and detecting cheating attempts in face-to-face games is traditionally done by humans (game opponents and organizers) but due to the games' complexity and the number of players to analyze and control in each tournament, this task is time consuming and complex. We propose in this paper approaches from the AI field and inspired from the anomaly detection problem to detect cheating activities in online duplicate Bridge tournaments.

The starting point of this work is the recommendations of the prestigious AAAI conference challenge on detecting collusive activities in online Bridge [14]. In this paper, we point out several technical issues related to cheating scenarios that can be performed in online Bridge tournaments where several players, on different boards, play the same deal (card distribution) and overall scores are computed to rank the players. Cheating can occur during the bidding phase or the card play one. Our goal is to detect suspicious activities. We propose approaches for detecting cheating activities inspired by those used in intrusion detection (in computer security) to detect attacks [2]. Namely, our objective is to design a tool capable of analyzing in almost real-time the data of on-going tournaments and flag the suspicious actions and players as cheating or normal. This problem is closely related to detecting computer attacks where an intrusion detection system IDS [4][2] analyzes network traffic (and potentially other information such as application log files, etc.) and flag as anomalous every suspicious activity with respect to authorized activities. More precisely, we propose anomaly-based approaches where significantly deviating activities from a normal activities reference profile are detected as potential attacks. This approach is very relevant since we can build normal profiles using the plays of the remaining players, historical data of the same players, robot players, etc. Moreover, in addition to the game data, we can use network-oriented data to help detecting cheating activities. For instance, IP addresses of the players, the duration of each step played by a player, the sequence of play, etc. are relevant features to consider for detecting cheating activities in duplicate Bridge tournaments.

The contribution of this paper is an anomaly detection-based approach for detecting cheating activities in online duplicate Bridge tournaments. Moreover, we provide preliminary results showing that our approach can be efficiently implemented and deployed in real online duplicate Bridge tournaments.

The rest of this paper is organized as follows: Section 2 provides basic background on online games and duplicate Bridge tournaments. Section 3 provides insights into anomaly detection approaches in computer security. In Section 4, we present our anomaly-based approach for detecting cheating activities. Finally, Section 5 provides details of our experimental studies.

## 2 Online games and cheating activities

### 2.1 Online duplicate Bridge tournaments

Online games are those played by players through Internet. In some games, there are tournaments that are organized and results of the participants are available.

In many player games, robots often play against human players. Bridge is a 52 card game with several specificities. It is played by four players, each two compose a team (North-South team and East-West one). After the cards are distributed (each player receives a hand of 13 cards), the game starts with a bidding session aiming to set the contract denoting the objective to attain. The first player to bid is called the declarer while the player having set the contract is called the dealer and his partner is the dummy. Once the contract set, the card game starts where the declarer's team tries to reach the contract while the opponent one tries to prevent them.

Bridge programs and robots have reached an acceptable level in the late 1990s, when the author in [8] proposed a Monte Carlo method combining a solver of open deals with a sampling method for deals generated randomly and compatible with the knowledge of the player. One of their weaknesses in the game with a dummy is their inability to deal with opponents playing in a non-optimal way when they are bluffing or simply because of their weakness. Another weakness is to play the card that has the best chance, thinking that the opponents will play perfectly, while an expert player may play a card that increases the likelihood that the opponents make a mistake. It may be noted that significant improvements have recently been made using learning methods [1][6].

Winning a Bridge game depends on the expertise of the player, his opponents but also the distribution of cards. To better score a player, he is compared to other players playing the same deal. This is the aim and principle of duplicate Bridge. A player is better than another one if he gains more points than the other having the same hand and playing the same position (South, North, ...). Hence, duplicate Bridge tournaments are organized where several players play a set of deals and their results over each deal are taken into account in order to compute scores and rankings. Such tournaments are organized by Bridge clubs, federations, etc. Online duplicate Bridge tournaments are similar to face-to-face ones except that the players are either human players or Bridge robots. The advantage of Bridge robots is that they can be very strong, and can play as long as one wants and at any time. Several efficient online Bridge sites exist such as Jack Bridge<sup>1</sup>, Wbridge5<sup>2</sup>, Bridge Baron<sup>3</sup> and GIB<sup>4</sup>, BBO<sup>5</sup>, etc. Some of these sites propose duplicate Bridge tournaments.

## 2.2 Cheating in online duplicate Bridge tournaments

As we pointed out in the introduction, a cheater can use uncontrollable channels to cheat in online duplicate Bridge tournaments. For instance, he can play as two different players using two different logins and access relevant information. Another uncontrollable channel is the one of communicating game information

<sup>1</sup> <http://www.jackbridge.com/>

<sup>2</sup> <http://www.wbridge5.com/>

<sup>3</sup> <http://www.bridgebaron.com/>

<sup>4</sup> <http://www.gibware.com/>

<sup>5</sup> <http://online.bridgebase.com/>

to another human player through the phone, instant messaging, etc. A player may also make use of specific software and robots to help him to find the optimal decisions. There are several online testimonies with former cheaters showing that cheating does exist. The authors in [14] provide some cheating scenarios. Our objective in this paper is to focus not only on some known cheating scenarios but on all potentially abnormal activities which may be cheating ones. This approach allows on the one hand to detect new cheating scenarios and on the other hand to not encode and search for many known cheating scenarios in huge amounts of data making this task complex and tedious.

In the following, we argue that anomaly-based approaches are relevant for detecting cheating activities without any prior knowledge on cheating.

### 3 Anomaly detection: Approaches and challenges

To some extent, the problem of detecting cheating activities in online duplicate Bridge tournaments is similar to the one of detecting intrusions in computer networks. There are two main approaches for intrusion detection:

1. **Misuse-based approaches:** These approaches detect only known attacks by identifying their "signatures" in the analyzed data. Signature-based intrusion detection systems (like the well-known Snort IDS<sup>6</sup>) rely on a signature database and some pattern matching techniques to detect attacks. The major drawback of these approaches is that they cannot detect new attacks, namely those for which there is no corresponding signature in the database.
2. **Anomaly-based approaches:** Anomaly-based approaches adopt a different and complementary strategy where the underlying assumption is that attacks and malicious activities have a behaviour which is different from the one of normal and legitimate ones. Hence, in order to detect abnormal activities, one has only to build a profile or a specification of what is normal. Such a profile can for instance be learnt after observing the normal activities or from a security policy or protocol specification, etc. During the detection phase, every significant deviation from the normal profile is considered as an abnormal event and can potentially be an attack. The main advantage of anomaly-based approaches is their ability to detect any attack (be it new or known). In practice, anomaly-based approaches suffer from a high false alarm rate because it is very difficult to obtain acceptable detection and false alarm rate tradeoffs.

In anomaly-based approaches, an anomaly score relative to a given event often depends on several local deviations measuring how much anomalous the analyzed event is with respect to the different normal profiles. Critical issues in anomaly detection are normal profile definition and anomaly scoring and thresholding. The first issue is concerned with extracting and selecting the best features to analyze in order to effectively detect anomalies. The second issue is also critical as it provides the anomaly scores determining

---

<sup>6</sup> [www.snort.org](http://www.snort.org)

whether an event should be flagged normal or anomalous. The problem of bad tradeoffs between detection rates and the underlying false alarm ones characterizing most anomaly-based approaches are in part due to problems in anomaly measuring, aggregating and thresholding methods [3].

### 3.1 Why anomaly-based approaches are more relevant

In [14], the author pointed out only few cheating scenarios. While it is very easy to implement a signature-based approach for detecting known cheating scenarios, this method firstly needs to list and encode the cheating scenarios and it cannot detect any novel cheating scenario that is not already present in the cheating scenarios database. In order to overcome these limitations, an anomaly-based approach may be more relevant since the normal profile can easily be obtained from the past tournaments or from the actions of the remaining players in duplicate Bridge tournaments. Note that a signature-based approach can be combined with an anomaly-based one in order to exploit their mutual complementarities.

### 3.2 Model definition and validation

During the model construction phase, one has to define the relevant variables for each local anomaly model. Such an effort needs a knowledge on the duplicate Bridge tournaments. It also needs analyzing both normal data and some cheating scenarios. Once the variable definition has been conducted, one has to choose which model to build and then collect and preprocess historical data in order to build the model. Model validation should consider normal plays that should be detected by the model as normal and cheating scenarios that should be detected as cheating actions. The model effectiveness can be measured through an ROC curve (Receiving Operating Curve) highlighting the detection rate with respect to the corresponding false alert one.

## 4 An anomaly-based approach for detecting cheating activities in duplicate online Bridge tournaments

### 4.1 Online Bridge data to analyze

Due to the different nature of data (sequential, relational, etc.) which may be relevant to analyze, we propose a multi-model approach where each normal behaviour in a given data type is represented by its specific profile. Hence, the anomaly score will be computed using local anomaly measures associated with each local model. An aggregating function will then be used to derive a global anomaly score and a thresholding schema is needed to decide whether the analyzed event is normal or anomalous. Let us give the different data types to analyze and an idea on the corresponding profile to capture the features of that behaviour.

1. *Relational data (attribute-values data)*: Some attribute-value data is relevant to detect some anomalies regarding the whole game of a player or during each action during the play. For example, the player's *rank*, the *contract*, the *lead* and the *result* of the game. It is possible to collect historical data and build a model of normal plays. Now, at the end of each play, we use an anomaly score to estimate the normality of the play. An example of anomaly scoring function that can be used here is a probabilistic one where anomaly scores are inversely proportional to the probability of the analyzed event. Moreover, such an anomaly score can take into account the dependences and correlations among the attributes. These correlations can be for instance provided by an expert or automatically learnt from the historical data.
2. *Sequential data*: Sequential data can be collected in Bridge games during the bidding sequence, the card game sequence or the actions of the different players in a duplicate tournament. For instance, the behaviour of a player who knows only his cards and the one who knows in addition the next actions of the remaining ones if he bids in a given way, can adapt his bidding more advantageously. The same scenario can happen during the card play. In addition to the sequence of actions during the bidding and card play phases, another source of relevant information is relative to the sequence of actions of the different plays, the durations between each two consecutive actions, etc. Such behaviours can be captured by sequence anomaly detection models such as hidden Markov models HMM [13], etc. Here again, a probability measure can be used to associate an anomaly score for each action sequence. It is clear that a good estimation of a global anomaly score should take into account the local scores. Clearly, using probabilistic-based models and probabilistic-based anomaly scoring functions has the advantage of preventing incommensurability problems and can be fused in many ways (by summation, averaging, maximum, etc.).

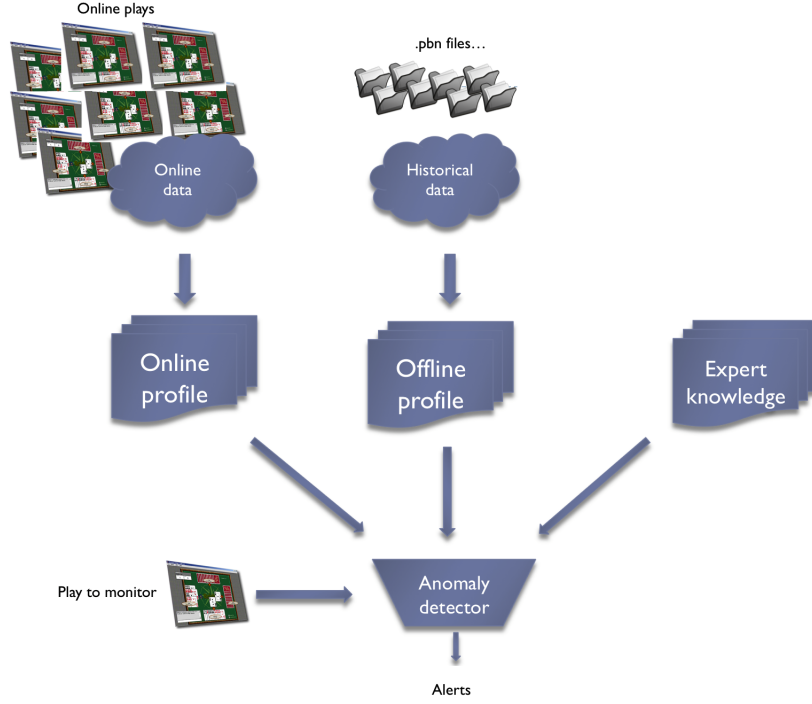
#### 4.2 Anomaly detection-based model architecture

Figure 1 depicts an anomaly-based model where profiles of normal activities are built using (i) online data consisting in the plays of the players of the same tournament on the same deal, (ii) offline data built on the past tournament on the same deals and potentially (iii) expert knowledge formalizing for example known cheating activities, game rules, etc.

In our experimentations, our anomaly-based approaches use an outlier detection technique and a probabilistic classifier where the training data is labeled as *normal* or *abnormal*.

### 5 Experimental studies

In this section, we provide preliminary experimental studies showing that anomaly-based approaches can efficiently detect cheating activities in online duplicate Bridge tournaments.



**Fig. 1.** Building blocks of an anomaly-based approach for detecting cheating activities in duplicate online Bridge tournaments.

### 5.1 Experimentation setup

In this experimentation, we are concerned with detecting anomalies in asynchronous duplicate Bridge tournaments<sup>7</sup>. Our experimental evaluation is carried out on **real data collected from a real online duplicate Bridge tournament** where many players play the same deals. Note that this tournament is asynchronous and each deal is played a given day but the participating players do not necessarily play at the same time. On each virtual board, the player plays with a Bridge robot against two Bridge robots.

We propose in this paper two experimentations using anomaly-based approaches for detecting cheating and abnormal activities in the analyzed data. They are implemented using classification<sup>8</sup> techniques allowing the prediction of cheating attempts. More precisely, in order to detect cheating attempts, we propose

<sup>7</sup> A short description of duplicate Bridge tournaments can be found here [http://en.wikipedia.org/wiki/Duplicate\\_bridge](http://en.wikipedia.org/wiki/Duplicate_bridge)

<sup>8</sup> Formally, classification consists in predicting the value of a non observable variable (here the nature of a play which can be normal or cheating) given the values of observed variables. Namely, given observed variables  $A_1, \dots, A_n$  describing the objects to classify, it is required to predict the *right* value of the class variable  $C$  among a predefined set of class instances. In order to build a classifier, one can either

classification-based approaches as final objective is checking whether there is a cheating attempt or not. Namely, we want to know for each play whether it is *normal* or *cheating*. Hence, the most straightforward way to adopt is a classification-based approach where the classes to predict are *Normal* (denoting the absence of cheating) and *Abnormal* (revealing an abnormal, potentially cheating attempt). In our problem, the variables are defined as follows:

1. **Predictors (attribute variables):** The set of attributes (observed variables) consists of the relevant features for detecting cheating attempts. For example, attributes describing the player, the game itself and the game's result will be needed for detecting some cheating attempts.
2. **Class variable:** The class variable  $C$  represents the game category variable whose domain involves the following values *Normal* and *Abnormal*.

**Naive Bayes classifier** In Experimentation 1, we use a naive Bayesian classifier [7] which is the simplest form of Bayesian network classifiers (which are Bayesian networks used for classification purposes). Naive Bayes classifiers are particularly simple to build while they allow very interesting classification performances [10]. Moreover, this classifier is efficiently used in many works for detecting intrusions and anomalies [3]. In addition to the naive classifier, we conducted other experimentations using other classifiers like C4.5 decision [12] and TAN Bayesian classifiers [10] but the results are very similar to those of naive Bayes. That is why the results of the other classifiers are not provided in this paper.

**Input data preparation: preprocessing and labelling** The data used in our experimentations is extracted from audit Bridge files (having the *.bpn* extension). This data is extracted and preprocessed in order to format it and extract only the relevant information. The play of each human player is described by the following features:

In our case, each play is described by the features of Table 1 and labeled either as *Normal* or *Abnormal* by an expert. Each play data is real (collected from real tournaments) and it is analyzed and then flagged manually. Because of this manual analysis, the experimentations are carried out on small datasets. The following is a snapshot of our dataset in the CSV format:

```
-----
Duration,Serie,Rank,Contract,Declarer,Lead,Vulnerable,Result,Score,Class
141,A,85,4HX,N,DA,All,-4,-1100, Normal
172,A,85,4HX,N,DA,All,-4,-1100, Normal
593,A,1,2SX,N,DK,All,=,670, Abnormal
```

---

rely on expert knowledge (for example, provide classification rules) or simply use machine learning techniques. In the latter case, there is need to build a training and labeled data set in order to train the classifier. Examples of well-known classifiers are Bayesian network classifiers [10], decision trees [12], etc.

Feature	Description	Type
Duration	Duration of the play in seconds	Numeric
Serie	Serie of the player	Symbolic
Rank	Rank of the player	Numeric
Contract	The deal set by the players after the bidding phase	Symbolic
Declarer	The declarer in the current play	Symbolic
Lead	The first card played by the dealer	Symbolic
Vulnerable	The vulnerable team	Symbolic
Result	Result of the play (it depends on the player's score and the ones of the other players)	Numeric
Score	Player's score after the current play	Numeric

**Table 1.** Feature set

207,C,2,3NT,N,SQ,A11,-3,300, Abnormal  
 91,B,3,5HX,N,SA,A11,-1,200, Abnormal  
 85,C,3,5H,S,D8,None,-2,200, Normal  
 345,A,3,3NT,N,SQ,A11,-2,200, Abnormal  
 307,D,3,4HX,N,D8,A11,-1,200, Abnormal  
 210,A,8,5H,N,SA,NS,-1,100, Normal  
 146,A,8,4H,N,D8,A11,-1,100, Normal

-----

## 5.2 Experimentation 1: A classification-based approach for detecting cheating activities

Table 2 shows the distribution of the dataset used in Experimentation 1. In this dataset, each deal is played by nearly a hundred of different human players (in each play, the human player plays with a robot against two robots). After manual analysis, we found 19 cheating/abnormal plays while the remaining ones seem normal.

Number of <i>nomal</i> plays	1020
Number of <i>abnomal</i> plays	19
Number of deals	10

**Table 2.** Dataset statistics used in Experimentation 1

In Table 3, we provide the results of evaluating a naive Bayes classifier on the dataset of Table 2. The evaluation is performed using 10-fold cross-validation<sup>9</sup>.

<sup>9</sup> Cross-validation is a technique for assessing the performances of a predictive model. In an N-fold cross-validation, the training data is divided into N subsets. The model is trained on N-1 subsets and then evaluated on the remaining subset. This operation is repeated changing the testing subset in each iteration. The results are then aggregated.

	Number	Rate
<b>Detected cheating/abnormal events</b>	15	78.94%
<b>Detected normal events</b>	1004	98.43%
<b>False alarms</b>	16	51.61%
<b>False negatives</b>	4	0.39%

**Table 3.** Naive Bayes results

Table 3 shows that most of the abnormal plays (15 among 19) are detected correctly and the rate of false alarms is acceptable given that in the analyzed tournament, the number of plays is around one hundred per deal and a human expert can analyze the raised alerts. In Experimentation 2, we detect abnormal plays by comparing each play with a set of plays on the same deal. Other evaluation methods like splitting the dataset into two separate sets, one for training and the other for testing, give nearly the same results as the ones of Table 3.

### 5.3 Experimentation 2: An outlier detection approach for detecting cheating activities

In this experimentation, we propose an anomaly-based method for detecting cheating activities using an outlier detection approach. Namely, detecting cheating activities is done by detecting outliers corresponding to significantly atypical items. Outlier and novelty detection is generally performed by computing a similarity score of an item from a reference dataset or a reference model representing a concept. In Experimentation 2, we use a one class-classifier as an implementation of an outlier detection method [9]. Namely, the technique learns only one target class and rejects any data item whose similarity with the learnt target class is less than a predefined threshold. Table 4 provide details on the used dataset and the obtained results. Note that in this experimentation, we ran the technique on each deal which is played a given number of times (see *#ofplays* column and between parantheses the number of abnormal/cheating plays). Each deal contains only few abnormal plays. We evaluated the model using a 10-fold cross validation as in Experimentation 1. The true negative rates<sup>10</sup> show that most of normal plays are recognized as normal. Note also that the model triggered only few alerts most of which are false ones but most of the abnormal ones are detected (see the number of existing abnormal plays in column *#ofplays* and the number of detected abnormal plays in column *#ofalerts* where the number of true positives (denoting true alarms) is between parantheses). Given that the number of raised alerts by day is acceptable then this method can be considered efficient since most of the cheating plays are detected. The evaluation using two separate sets (different training and testing sets) gives similar results.

<sup>10</sup> A true negative is a normal activity that is detected as a normal one by a predictive model.

Deal number	# of plays (Abnormal)	True negative rate	# of alerts (True posi- tives)	False positive rate
1	79 (1)	92,41%	8 (1)	87,50%
2	90 (2)	94,32%	6 (1)	83,33%
3	128 (1)	95,28%	7 (0)	100,00%
4	125 (3)	95,90%	8 (3)	62,50%
5	113 (1)	96,43%	5 (1)	80,00%
6	114 (1)	96,46%	5 (1)	80,00%
7	195 (3)	97,92%	4 (0)	100,00%
8	120 (5)	97,39%	6 (3)	50,00%
9	108(2)	97,17%	4 (1)	75,00%
10	95 (1)	96,81%	4 (1)	75,00%

Table 4. Outlier detetction approach results

## 6 Summary and conclusions

The paper dealt with the problem of detecting cheating activities in online games. More precisely, it focused on detecting cheating activities in duplicate online Bridge tournaments. We argued that anomaly-based approaches which are widely used in some domains like intrusion detection are particularly suitable for our problem. The results presented in this paper on real data and using anomaly-based approaches show the effectiveness of such approaches for detecting cheating activities in duplicate Bridge tournaments. Moreover, these results can be improved in several directions:

- **Using more relevant datasets / models:** Possible improvements can be obtained by defining more relevant variables and more representative datasets. For instance, the number of training instances can be increased and data distribution can be improved to decrease the negative impact of the class imbalance problem (in our experimentations, the proportion of abnormal plays is very low in comparison with normal ones). As for the used models, one can obtain some improvements by choosing more efficient classifiers (SVM [5], decision trees [12], etc.), combining several models and mixing them with models analyzing sequential data (like HMMs [13]).
- **Combining anomaly-based approaches with signature-based ones:** Given that some cheating scenarios are known, then one can combine an anomaly-based approach (in serial or in parallel) along with a signature-based one. This latter will efficiently detect known cheating scenarios while the anomaly-based one will detect unknown ones.

A potentially interesting technique for collecting cheating attempts data is deploying honeypots [11] that can lure cheaters. Honeypots are widely used in intrusion detection for collecting intruder attempts. The key issue here is how to build a duplicate Bridge tournament platform allowing cheating and how to automatically generate deals that can lure cheaters or generate deals where

only cheaters can win while non cheating players cannot. Such data will help developing efficient cheating detection systems.

## References

1. Asaf Amit and Shaul Markovitch. Learning to bid in bridge. *Machine Learning*, 63(3):287–327, 2006.
2. Stefan Axelsson. Intrusion Detection Systems: A Survey and Taxonomy. Technical Report 99-15, Chalmers Univ., March 2000.
3. Salem Benferhat and Karim Tabia. New schemes for anomaly score aggregation and thresholding. In *Proceedings of the International Conference on Security and Cryptography, Porto, Portugal, July 26-29*, pages 21–28. INSTICC Press, 2008.
4. Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM Comput. Surv.*, 41:15:1–15:58, July 2009.
5. Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine Learning*, 20(3):273–297, 1995.
6. Downey J. DeLooze L. Bridge bidding with imperfect information. In *Computational Intelligence and Games, CIG 2007*, pages 368–373. Honolulu: IEEE Press, 2007.
7. Nir Friedman, Dan Geiger, and Moises Goldszmidt. Bayesian network classifiers. *Mach. Learn.*, 29:131–163, November 1997.
8. Matthew L. Ginsberg. Gib: Steps toward an expert-level bridge-playing program. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence, IJCAI '99*, pages 584–593, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.
9. Kathryn Hempstalk, Eibe Frank, and Ian H. Witten. One-class classification by combining density and class probability estimation. In *Proceedings of the 2008 European Conference on Machine Learning and Knowledge Discovery in Databases - Part I, ECML PKDD '08*, pages 505–519, Berlin, Heidelberg, 2008. Springer-Verlag.
10. Pat Langley, Wayne Iba, and Kevin Thompson. An analysis of bayesian classifiers. In *In Proceedings of the 10th national conference on artificial intelligence*, pages 223–228. MIT Press, 1992.
11. Iyatiti Mokube and Michele Adams. Honeypots: concepts, approaches, and challenges. In *Proceedings of the 45th annual southeast regional conference, ACM-SE 45*, pages 321–326, New York, NY, USA, 2007. ACM.
12. J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.
13. Lawrence R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. In *Proceedings of the IEEE*, pages 257–286, 1989.
14. Jeff Yan. Collusion detection in online bridge. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*. AAAI Press, 2010.

# Sparse Sampling for Adversarial Games

Marc Lanctot<sup>1</sup>, Abdallah Saffidine<sup>2</sup>, Joel Veness<sup>1</sup>, Chris Archibald<sup>1</sup>

<sup>1</sup>University of Alberta, Edmonton, Alberta, Canada

<sup>2</sup>LAMSADE, Université Paris-Dauphine, Paris, France

**Abstract.** This paper introduces Monte Carlo \*-Minimax Search (MCMS), a Monte-Carlo search algorithm for finite, turned based, stochastic, two-player, zero-sum games of perfect information. Through a combination of sparse sampling and classical pruning techniques, MCMS allows deep plans to be constructed. Unlike other popular tree search techniques, MCMS is suitable for densely stochastic games, i.e., games where one would never expect to sample the same state twice. We give a basis for the theoretical properties of the algorithm and evaluate its performance in three games: Pig (Pig Out), EinStein Würfelt Nicht!, and Can't Stop.

## 1 Introduction

Monte-Carlo Tree Search (MCTS) has recently become one of the dominant paradigms for online planning in large sequential games. Since its initial application to Computer Go [Gelly et al., 2006, Coulom, 2007a], numerous extensions have been proposed, allowing this general approach [Chaslot et al., 2008a] to be successfully adapted to a variety of challenging problem settings, including real-time strategy games, imperfect information games and General Game Playing [Finnsson and Björnsson, 2008, Szita et al., 2010, Winands et al., 2010, Auger, 2011, Ciancarini and Favini, 2010]. At first, the method was applied to games lacking strong Minimax players, but recently has been shown to compete against strong Minimax players in such games [Ramanujan and Selman, 2011, Winands and Björnsson, 2010].

One class of games that has proven more resistant is stochastic games. Unlike classical games such as Chess and Go, stochastic game trees include chance nodes in addition to decision nodes. How MCTS should account for this added uncertainty remains unclear. The classical algorithms for stochastic games, Expectimax (exp) and \*-Minimax (Star1 and Star2), perform look-ahead searches to a limited depth. However, both algorithms scale exponentially in the branching factor at chance nodes as the search horizon is increased. Hence, their performance in large games often depends heavily on the quality of the heuristic evaluation function, as only shallow searches are possible.

One way to handle the uncertainty at chance nodes is to simply sample a single outcome when encountering a chance node. This is common practice in MCTS when applied to stochastic games; however, the general performance of this method is questionable. Large stochastic domains still pose a significant challenge. For example, MCTS is outperformed by \*-Minimax in the game of Carcassonne [Heyden, 2009]. Unfortunately, the literature on the application of Monte Carlo search methods to stochastic

games is relatively small, possibly due to the lack of a principled and practical algorithm for these domains.

In this paper, we introduce a new algorithm, called Monte-Carlo Minimax Search (MCMS), which can increase the performance of search algorithms in stochastic games by sampling a subset of chance event outcomes. We describe a sampling technique for chance nodes based on sparse sampling [Kearns et al., 1999]. We present a theorem that shows that MCMS approaches the optimal decision as the number of samples grows. We evaluate the practical performance of MCMS in three domains: Pig (Pig Out), EinStein Würfelt Nicht! (EWN), and Can't Stop. Finally, we show in Pig (Pig Out) that the estimates returned by MCMS have lower mean-squared error and lower regret than the estimates returned by MCTS.

## 2 Background

A finite, two-player zero-sum game of perfect information can be described as a tuple  $(\mathcal{S}, \mathcal{T}, \mathcal{A}, \mathcal{P}, u_1, s_1)$ , which we now define. The state space  $\mathcal{S}$  is a finite, non-empty set of states, with  $\mathcal{T} \subseteq \mathcal{S}$  denoting the finite, non-empty set of terminal states. The action space  $\mathcal{A}$  is a finite, non-empty set of actions. The transition probability function  $\mathcal{P}$  assigns to each state-action pair  $(s, a) \in \mathcal{S} \times \mathcal{A}$  a probability measure over  $\mathcal{S}$  that we denote by  $\mathcal{P}(\cdot | s, a)$ . The utility function  $u_1 : \mathcal{T} \mapsto [v_{\min}, v_{\max}] \subset \mathbb{R}$  gives the utility of player 1, with  $v_{\min}$  and  $v_{\max}$  denoting the minimum and maximum possible utility respectively. Since the game is zero-sum, the utility of player 2 in any state  $s \in \mathcal{T}$  is given by  $u_2(s) := -u_1(s)$ . The player index function  $\tau : \mathcal{S} \rightarrow \{1, 2\}$  returns the player to act in given state  $s$  if  $s \in \mathcal{S} \setminus \mathcal{T}$ , otherwise it returns 1 in the case where  $s \in \mathcal{T}$ .

Each game starts in the initial state  $s_1$  with  $\tau(s_1) := 1$ . It proceeds as follows, for each time step  $t \in \mathbb{N}$ : first, player  $\tau(s_t)$  selects an action  $a_t \in \mathcal{A}$  in state  $s_t$ , with the next state  $s_{t+1}$  generated according to  $\mathcal{P}(\cdot | s_t, a_t)$ . Player  $\tau(s_{t+1})$  then chooses a next action and the cycle continues until some terminal state  $s_T \in \mathcal{T}$  is reached. At this point player 1 and player 2 receive a utility of  $u_1(s_T)$  and  $u_2(s_T)$  respectively.

### 2.1 Classical Game Tree Search

We now describe the two main search paradigms for adversarial stochastic game tree search. We begin first by describing the classical techniques, that differ from modern approaches in that they do not use Monte-Carlo sampling. The minimax value of a state  $s \in \mathcal{S}$  is defined by

$$V(s) := \begin{cases} \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} \mathcal{P}(s' | s, a) V(s') & \text{if } s \notin \mathcal{T} \text{ and } \tau(s) = 1 \\ \min_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} \mathcal{P}(s' | s, a) V(s') & \text{if } s \notin \mathcal{T} \text{ and } \tau(s) = 2 \\ u_1(s) & \text{otherwise,} \end{cases}$$

Note that we always treat player 1 as the player maximizing  $u_1(s)$  (*Max*), and player 2 as the player minimizing  $u_1(s)$  (*Min*). In most large games, computing the minimax value for a given game state is intractable. Because of this, an often used approximation is to instead compute the *finite horizon minimax value*. This requires limiting

the recursion to some fixed depth  $d \in \mathbb{N}$  and applying a heuristic evaluation function  $h : \mathcal{S} \mapsto [v_{\min}, v_{\max}]$  when this depth limit is reached. Thus given a heuristic evaluation function  $h_1(s) : \mathcal{S} \rightarrow \mathbb{R}$  defined with respect to player 1 that satisfies the requirement  $h_1(s) = u_1(s)$  when  $s \in \mathcal{T}$ , the finite horizon minimax value is defined recursively by

$$V_d(s) := \begin{cases} \max_{a \in \mathcal{A}} V_d(s, a) & \text{if } d > 0, s \notin \mathcal{T}, \text{ and } \tau(s) = 1 \\ \min_{a \in \mathcal{A}} V_d(s, a) & \text{if } d > 0, s \notin \mathcal{T}, \text{ and } \tau(s) = 2 \\ h_1(s) & \text{otherwise,} \end{cases}$$

where

$$V_d(s, a) := \sum_{s' \in \mathcal{S}} \mathcal{P}(s' | s, a) V_{d-1}(s'), \quad (1)$$

For sufficiently large  $d$ ,  $V_d(s)$  coincides with  $V(s)$ . The quality of the approximation depends on both the heuristic evaluation function and the search depth parameter  $d$ .

A direct computation of  $\arg \max_{a \in \mathcal{A}(s)} V_d(s, a)$  or  $\arg \min_{a \in \mathcal{A}(s)} V_d(s, a)$  is equivalent to running the well known EXPECTIMINIMAX algorithm [Russell and Norvig, 2010]. The base EXPECTIMINIMAX algorithm can be enhanced by a technique similar to alpha-beta pruning for deterministic game tree search. This involves correctly propagating the  $[\alpha, \beta]$  bounds and performing an additional pruning step at each chance node. This pruning step is based on the simple observation that if the minimax value has already been computed for a subset of successors  $\tilde{\mathcal{S}} \subset \mathcal{S}$ , the negamax value of state-action pair  $(s, a)$  must lie within

$$L_d(s, a) \leq V_d(s, a) \leq U_d(s, a),$$

where

$$\begin{aligned} L_d(s, a) &:= \sum_{s' \in \tilde{\mathcal{S}}} \mathcal{P}(s' | s, a) V_{d-1}(s') + \sum_{s' \in \mathcal{S} \setminus \tilde{\mathcal{S}}} \mathcal{P}(s' | s, a) v_{\min} \\ U_d(s, a) &:= \sum_{s' \in \tilde{\mathcal{S}}} \mathcal{P}(s' | s, a) V_{d-1}(s') + \sum_{s' \in \mathcal{S} \setminus \tilde{\mathcal{S}}} \mathcal{P}(s' | s, a) v_{\max}. \end{aligned}$$

These bounds form the basis of the pruning mechanisms in the \*-Minimax [Ballard, 1983] family of algorithms. In the Star1 algorithm, each  $s'$  from the equations above represents the state reached after a particular outcome is applied at a chance node following  $(s, a)$ . In practice, Star1 maintains lower and upper bounds on  $V_{d-1}(s')$  for each child  $s'$  at chance nodes, using this information to stop the search when it finds a proof that any future search is pointless.

To better understand when cutoffs occur in \*-Minimax, we now present an example adapted from Ballard's original paper. Consider Figure 1. The algorithm recurses down from state  $s$  with a window of  $[\alpha, \beta] = [4, 5]$  and encounters a chance node. Without having searched any of the children the bounds for the values returned are  $(v_{\min}, v_{\max}) = (-10, +10)$ . The subtree of a child, say  $s'$ , is searched and returns  $V_{d-1}(s') = 2$ . Since this is now known, the upper and lower bounds for that outcome become 2. The lower bound on the minimax value of the chance node becomes  $(2 - 10 - 10)/3$  and the upper bound becomes  $(2 + 10 + 10)/3$ , assuming a uniform distribution over chance events. If ever the lower bound on the value of the chance

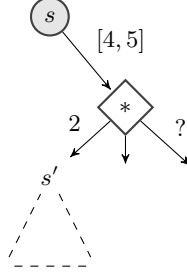


Fig. 1: An example of the STAR1 algorithm.

**Algorithm 1** Star1

---

```

1: Star1( $s, a, d, \alpha, \beta, c$ )
2:   if  $d = 0$  or  $s \in \mathcal{T}$  then return  $(h_1(s), \text{null})$ 
3:   else if  $\neg c$  then return  $\text{alphabeta1}(s, d, \alpha, \beta)$ 
4:   else
5:      $o \leftarrow \text{genOutcomeSet}(s, a, v_{\min}, v_{\max})$ 
6:      $N \leftarrow |o|$ 
7:     for  $i \in \{0, \dots, N - 1\}$ 
8:        $\alpha' \leftarrow \text{computeChildAlpha}(o, \alpha, i)$ ;  $\beta' \leftarrow \text{computeChildBeta}(o, \beta, i)$ 
9:        $s' \leftarrow \text{applyActionAndChanceOutcome}(s, a, i)$ 
10:       $(v, a') \leftarrow \text{Star1}(s', \text{null}, d - 1, \max(v_{\min}, \alpha'), \min(v_{\max}, \beta'), \text{false})$ 
11:       $o_{il} \leftarrow v$ ;  $o_{iu} \leftarrow v$ 
12:      if  $v \geq \beta'$  then return  $(\text{lowerBound}(o), \text{null})$ 
13:      if  $v \leq \alpha'$  then return  $(\text{upperBound}(o), \text{null})$ 
14:   return  $\text{exactValue}(o)$ 

```

---

node exceeds  $\beta$ , or if the upper bound for the chance node is less than  $\alpha$ , the subtree is pruned. In addition, this bound information is used to compute new bounds to send to the other child nodes.

The algorithm is summarized in Algorithm 1. The `alphabeta1` procedure recursively calls `Star1`. The parameter  $c$  is a boolean representing whether or not a chance node is the next node in the tree. The outcome set  $o$  is an array of tuples, one per outcome. The  $i$ th tuple has three attributes: a lower bound  $o_{il}$  initialized to  $v_{\min}$ , an upper bound  $o_{iu}$  initialized to  $v_{\max}$ , and the outcome's probability  $o_{ip}$ . The `lowerBound` function returns the current lower bound on the chance node  $\sum_{i \in \{0, \dots, N-1\}} o_{ip} o_{il}$ . Similarly, `upperBound` returns the current upper bound on the chance node using  $o_{iu}$  in place of  $o_{il}$ . Finally, the functions `computeChildAlpha` and `computeChildBeta` return the new bounds on the value of the respective child below. Continuing the example above, suppose the algorithm is ready to descend down the middle outcome. The lower bound for the child is derived from the equation  $(2 + o_{1p}\alpha' + 10)/3 = \alpha$ . Solving for  $\alpha'$  here gives  $\alpha' = (3\alpha - 12)/o_{1p}$ . In general:

$$\alpha' = \frac{\alpha - \text{upperBound}(o) + o_{ip}o_{iu}}{o_{ip}}, \quad \beta' = \frac{\beta - \text{lowerBound}(o) + o_{ip}o_{il}}{o_{ip}}.$$

**Algorithm 2** Star2

---

```

1: Star2( $s, a, d, \alpha, \beta, c, p$ )
2:   if  $d = 0$  or  $s \in \mathcal{T}$  then return ( $h_1(s)$ , null)
3:   else if  $\neg c$  then return alphabeta2( $s, d, \alpha, \beta, p$ )
4:   else
5:      $o \leftarrow \text{genOutcomeSet}(s, a, v_{\min}, v_{\max})$ 
6:      $N \leftarrow |o|$ 
7:     for  $i \in \{0, \dots, N - 1\}$ 
8:        $\alpha' \leftarrow \text{computeChildAlpha}(o, \alpha, i)$ ;  $\beta' \leftarrow \text{computeChildBeta}(o, \beta, i)$ 
9:        $s' \leftarrow \text{applyActionAndChanceOutcome}(s, a, i)$ 
10:      ( $v, a'$ )  $\leftarrow \text{Star2}(s', \text{null}, d - 1, \max(v_{\min}, \alpha'), \min(v_{\max}, \beta'), \text{false}, \text{true})$ 
11:      if  $\tau(s') = 1$  then
12:         $o_{il} \leftarrow v$ 
13:        if  $\text{lowerBound}(o) \geq \beta$  then return ( $\text{lowerBound}(o)$ , null)
14:        else if  $\tau(s') = 2$  then
15:           $o_{iu} \leftarrow v$ 
16:          if  $\text{upperBound}(o) \leq \alpha$  then return ( $\text{upperBound}(o)$ , null)
17:      for  $i \in \{0, \dots, N - 1\}$ 
18:         $\alpha' \leftarrow \text{computeChildAlpha}(o, \alpha, i)$ ;  $\beta' \leftarrow \text{computeChildBeta}(o, \beta, i)$ 
19:         $s' \leftarrow \text{applyActionAndChanceOutcome}(s, a, i)$ 
20:        ( $v, a'$ )  $\leftarrow \text{Star2}(s', \text{null}, d - 1, \max(v_{\min}, \alpha'), \min(v_{\max}, \beta'), \text{false}, \text{false})$ 
21:         $o_{il} \leftarrow v$ ;  $o_{iu} \leftarrow v$ 
22:        if  $v \geq \beta'$  then return ( $\text{lowerBound}(o)$ , null)
23:        if  $v \leq \alpha'$  then return ( $\text{upperBound}(o)$ , null)
24:      return  $\text{exactValue}(o)$ 

```

---

The performance of the algorithm can be improved significantly by applying a simple look-ahead heuristic. Suppose the algorithm encounters a chance node. When searching the children of each outcome, one can temporarily restrict the legal actions at a successor (decision) node. If only a single action is searched at the successor, then the value returned will be a bound on  $V_{d-1}(s')$ . If the successor is a Max node, then the true value can only be larger, and hence the value returned is a lower bound. Similarly, if it was a Min node, the value returned is a lower bound. The Star2 algorithm applies this idea via a preliminary *probing phase* at chance nodes in hopes of pruning without requiring full search of the children. If probing does not lead to a cutoff, then the children are fully searched, but bound information collected in the probing phase can be re-used. When moves are appropriately ordered, the algorithm can often choose the best single move and effectively cause a cut-off with exponentially less search effort. Since this is applied recursively, the benefit compounds as the depth increases. The algorithm is summarized in Algorithm 2. The alphabeta2 procedure is analogous to alphabeta1 except when  $p$  is true, a subset (of size one) of the actions are considered at the next decision node. The recursive calls to Star2 within alphabeta2 have  $p$  set to false and  $a$  set to the chosen action.

Note that Star1 and Star2 are typically presented using the negamax formulation. In fact, Ballard originally restricted his discussion to regular \*-minimax trees, ones that strictly alternate Max, Chance, Min, Chance. We intentionally present the more

general  $\alpha - \beta$  formulation here because it handles a specific case encountered by two of our three test domains. In games where the outcome of a chance node determines the next player to play, the cut criteria during the STAR2 probing phase depends on the child node. The bound established by the STAR2 probing phase will either be a lower bound or an upper bound, depending on the child's type. This distinction is made in lines 11 to 16. Also note: when implementing the algorithm, for better performance it is advisable to incrementally compute the bound information [Hauk et al., 2006].

## 2.2 Monte Carlo Tree Search

In recent years Monte Carlo methods have seen a surge of popularity in tree search methods for games. The main idea is to iteratively run simulations from the game's current position to a leaf, and incrementally grow a model of the game tree rooted at the current position. The tree starts empty and each simulation adds (expands) the tree by adding a single node (a leaf) to the tree. The node added to the tree may not be a terminal state, so a rollout policy takes over and chooses actions until a terminal state is reached. This idea of using random rollouts to estimate the value of individual positions has proved very successful in Go and many other domains [Coulom, 2007a, Browne et al., 2012]. When a simulation encounters a terminal state, it returns (back-propagates) the utility of the state up to all the nodes reached during the simulation, updating reward estimates for actions maintained at the nodes in the tree.

During a simulation through the tree, actions must be chosen. A popular way to select actions is to do so in a way that balances exploration and exploitation as in the well-known multi-armed bandit scenario [Auer et al., 2002]. UCT is an algorithm that recursively applies this selection mechanism to trees [Kocsis and Szepesvári, 2006]. An improvement of practical importance has been established called Progressive Unpruning / Widening [Coulom, 2007b, Chaslot et al., 2008b]. The main idea here is to purposely restrict the number of actions; this width is gradually increased so that the tree grows deeper at first and then slowly wider over time.

The progressive widening idea is extended to include chance nodes in the double progressive widening algorithm (DPW) [Couetoux et al., 2011]. When DPW encounters a chance or decision node, it computes a maximum number of actions or outcomes to consider  $k = \lceil Cv^\alpha \rceil$ , where  $C$  and  $\alpha$  are parameter constants and  $v$  represents a number of visits to the node. At a decision node, then only the first  $k$  actions from the action set are available. At a chance node, a set of outcomes is stored and incrementally grown. An outcome is sampled; if  $k$  is larger than the size of the current set of outcomes and the newly sampled outcome is not in the set, it is added to the set. When the branching factor at chance nodes is extremely high, double progressive widening prevents MCTS from degrading into 1-ply rollout planning. We will use MCTS enhanced with double progressive widening as one of the baseline algorithms for our experimental comparison.

## 2.3 Sampling Methods for Markov Decision Processes

Computing optimal policies in large Markov Decision Processes (MDPs) is a significant challenge. Since the size of the state space is often exponential in the properties

describing each state, much work has focused on finding efficient methods to compute approximately optimal solutions. One way to do this, given only a generative model of the domain, is to employ *sparse sampling* [Kearns et al., 1999]. When faced with a decision to make from a particular state, a local sub-MDP is built using finite horizon look-ahead search. When transitioning to successor states, a fixed number  $c \in \mathbb{N}$  of successor states are sampled for each action. Kearns et al. showed that for an appropriate choice of  $c$ , this procedure produces value estimates that are accurate (with high probability). Importantly,  $c$  was shown to have no dependence on the number of states  $|\mathcal{S}|$ , effectively breaking the curse of dimensionality.

This method of sparse sampling was improved by using adaptive decision rules based on the multi-armed bandit literature to give the AMS algorithm [Chang et al., 2005]. Also, the Forward Search Sparse Sampling (FSSS) [Walsh et al., 2010] algorithm was recently introduced, which exploits bound information to add a form of sound pruning to sparse sampling. The pruning mechanism used by FSSS is analogous to what Star1 performs in adversarial domains.

### 3 Sparse Sampling in Adversarial Games

The performance of classical game tree search suffers from a dependence on  $|\mathcal{S}|$ . Like Sparse Sampling for MDPs [Kearns et al., 1999], we remove this dependence using Monte-Carlo sampling. We now define the *estimated finite horizon minimax value* as

$$\hat{V}_d(s) := \begin{cases} \max_{a \in \mathcal{A}} \hat{V}_d(s, a) & \text{if } d > 0, s \notin \mathcal{T}, \text{ and } \tau(s) = 1 \\ \min_{a \in \mathcal{A}} \hat{V}_d(s, a) & \text{if } d > 0, s \notin \mathcal{T}, \text{ and } \tau(s) = 2 \\ h(s) & \text{otherwise.} \end{cases}$$

where

$$\hat{V}_d(s, a) := \frac{1}{c} \sum_{i=1}^c \hat{V}_{d-1}(s_i),$$

for all  $s \in \mathcal{S}$  and  $a \in \mathcal{A}$ , with each successor state  $s_i \sim \mathcal{P}(\cdot | s, a)$  for  $1 \leq i \leq c$ .

We now state a result which shows that the previously defined value estimates are accurate (with high probability), provided  $c$  is sufficiently large.

**Theorem 1** *Given  $c \in \mathbb{N}$ , for any state  $s \in \mathcal{S}$ , for all  $\lambda \in (0, 2v_{\max}] \subset \mathbb{R}$ , for any depth  $d \in \mathbb{Z}_+$ ,*

$$\mathbb{P} \left( \left| \hat{V}_d(s) - V_d(s) \right| \leq \lambda d \right) \geq 1 - (2c|\mathcal{A}|)^d \exp \left\{ -\lambda^2 c / 2v_{\max}^2 \right\}.$$

The proof is a straightforward generalisation of the result of Kearns et al. [1999] for finite horizon, adversarial games. As it is quite long, we defer its presentation to a forthcoming technical report.

The MCMS variants can be easily described in terms of the descriptions of Star1 and Star2. To enable sampling, one need only change the implementation of `getOutcomeSet` on line 5 of Algorithm 1 and line 5 of Algorithm 2. Instead of generating the full list of





moves, the new function samples  $c$  outcomes *with replacement* and assigns a uniform distribution over the new outcome set of size  $c$ . We call these new variants s1SS and s2SS. If all pruning is disabled, we obtain EXPECTIMINIMAX with sparse sampling (EXPSS), which computes  $\hat{V}_d(s)$  directly from definition. The s1SS method computes exactly the same value as EXPSS, but can avoid useless work. The same can be said for s2SS, provided exactly the same set of chance events is used whenever a state-action pair is visited; this additional restriction is needed due to the extra probing phase in Star2. Note that while sampling without replacement may work better in practice, Theorem 1 only holds in the case of sampling with replacement. We aim to extend our analysis to cover the without replacement case in future work.




## 4 Experiments

In this section, we describe our empirical evaluation of MCMS. The algorithm abbreviations are expanded and summarized in Table 1. Before describing our experiments, we describe our domains.

Table 1: Algorithms summary and abbreviations.

Abbreviation	Algorithm
exp	Plain expectimax search
Star1	Star1 search, as described in Algorithm 1
Star2	Star2 search, as described in Algorithm 2
MCTS	MCTS, as described in Section 2.2
DPW	MCTS with double progressive widening, as described in Section 2.2
expSS	Expectimax search with sparse sampling
s1SS	Star1 search with sparse sampling
s2SS	Star2 search with sparse sampling

**Pig (Pig Out)** is a two-player dice game [Scarne, 1945]. Players each start with 0 points; the goal is to be the first player to achieve 100 or more points. Each turn, players roll two dice and then, if there are no  showing, add the sum to their turn total. At each decision point, a player may continue to roll or stop. If they decide to stop, they add their turn total to their total score and then it becomes the opponent's turn. Otherwise, they roll dice again for a chance to continue adding to their turn total. If a single  is rolled the turn total will be reset and the turn ended (no points gained); if a   is rolled then the players turn will end along with their *total score* being reset to 0.

**EinStein Würfelt Nicht! (EWN)** is a tactical dice game played on a 5 by 5 grid. Each player, red or blue, starts with their 6 dice from (, , ..., ) in the top left corner squares or bottom-right corner squares. The goal is to reach the opponent's corner

square with a single die or capture every opponent piece. Each turn starts with the player rolling a white die off the board which indicates which of their dice they can move this turn. Pieces can only move toward the opponent’s corner square or off the board; if they move a die over a square containing another die (belonging to them or the opponent), it is captured. EWN is a popular game played by humans and computer opponents on the Little Golem online board game site<sup>1</sup>; at least one MCTS player has been developed to play it [Lorentz, 2011].

**Can’t Stop** is a dice game that is currently popular on online gaming sites [Sackson, 1980]. The goal is to obtain three complete columns by reaching the highest level in each of the 2-12 columns. This done by repeatedly rolling 4 dice and playing zero or more pairing combinations. Once a pairing combination is played, a marker is placed on the associated column and moved upwards. Only three distinct columns can be used during any given turn. If dice are rolled and no legal pairing combination can be made, the player loses all of the progress made towards completing columns on this turn. After rolling and making a legal pairing, a player can chose to lock in their progress by ending their turn. A key component of the game involves correctly assessing the risk associated with not being able to make a legal dice pairing given the current board configuration.

To evaluate our algorithm, we performed two separate experiments. Our first experiment compares statistical properties of the estimates returned and actions recommended by MCMS and MCTS. At a decision point, each algorithm returns a recommended move  $a \in \mathcal{A}$  and acts as an estimator of its minimax value  $\hat{V}(s)$ . Since Pig (Pig Out) has fewer than one million states, we solve it using the technique of value iteration which has been applied to previous smaller games of Pig [Neller and Pressor, 2004], obtaining the true value of each state  $V(s)$ . From this, we estimate the *mean squared error*, *variance*, and *bias* of each algorithm:  $\text{MSE}[\hat{V}(s)] = \mathbb{E}[(\hat{V}(s) - V(s))^2] = \text{Var}[\hat{V}(s)] + \text{Bias}(V(s), \hat{V}(s))^2$  by taking 30 samples of each algorithm at each decision point. Define the regret of taking action  $a$  at state  $s$  to be  $\text{Regret}(s, a) = V(s) - V(s, a)$ , where  $a$  is the action chosen by the algorithm from state  $s$ . We measure the average value of  $\text{Regret}(s, a)$  over the 30 samples at each decision point for each algorithm. The results of this experiment is shown in Figure 2.

In our second experiment, we computed the performance of each algorithm by playing 500 test matches for each paired set of players. Each match consists of two games where players swap seats and a single random seed is generated and used for both games in the match. The performance of each pairing of players is shown in Table 2.

In all of our experiments, a time limit of 0.1 seconds of search is used. MCTS uses utilities in  $[-100, 100]$  and a tuned exploration constant value of 50. For a more fair comparison, MCTS returns the value of the heuristic evaluation function at the leaves rather than using a rollout policy. MCTS with double-progressive widening (DPW) uses parameters  $C$  and  $\alpha$  described in Section 2.2. All experiments were single-threaded and run on the same hardware (equipped with Intel Core i7 3.4Ghz processors). The best sample widths for expSS, s1SS, s2SS, and  $(C, \alpha)$  for DPW for Pig (Pig Out) were (20, 2, 8, (5, 0.4)). For EWN and Can’t Stop these parameters were set to

<sup>1</sup> <http://www.littlegolem.net>

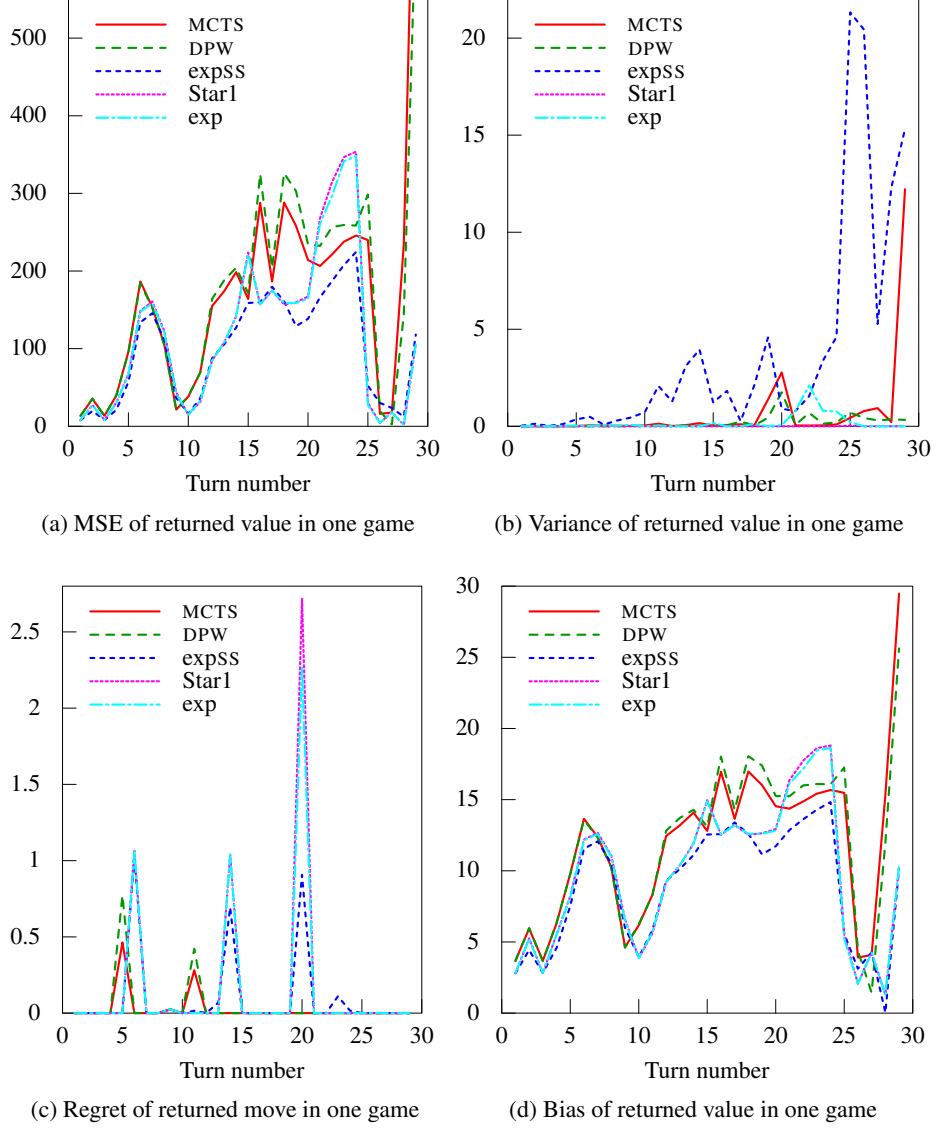


Fig. 2: Properties of MCMS on Pig (Pig Out). exp and expSS represent EXPECTIMAX without and with sparse sampling, respectively. DPW represents MCTS using double progressive widening.

Table 2: Number of wins for  $p_1$  in a  $p_1$ - $p_2$  match of 1000 games in Pig (Pig Out), EWN, and Can’t Stop.

	expSS-exp	s1SS-Star1	s2SS-Star2	expSS-DPW	s1SS-DPW	s2SS-DPW	MCTS-DPW
Pig (Pig Out)	555±15.7	514±15.8	485±15.8	499±15.8	405±15.5	461±15.8	483±15.8
EWN	519±15.8	507±15.8	501±15.8	497±15.8	452±15.7	501±15.8	513±15.8
Can’t Stop	811±12.4	844±11.5	846±11.4	804±12.6	801±12.6	779±13.1	189±12.4

(5, 2, 4, (3, 0.05)) and (5, 50, 18, (10, 0.2)) respectively. These values were determined by running a number of round-robin tournaments between players of the same type.

We see from Figure 2 that the estimated mean-squared error and bias of the values returned by expSS are lower than both MCTS and the non-sampling algorithms. However, this comes at cost of higher variance value estimates, especially toward the end of the game. This makes sense in Pig (Pig Out) since the chance outcomes affect the game more heavily near the end. For future investigation, it might make sense to consider using one or more variance reduction techniques [Veness et al., 2011] to improve the performance of MCMS. expSS also exhibits lower regret than MCTS but not as low as MCTS with double-progressive widening. While these graphs only represent a single game, the results on other games looked similar.

The results from Table 2 show that the MCMS variants outperform their equivalent non-sampling counterparts in all but one instance; this might be explained by the fact that since there are only 2 actions to choose from in Pig (Pig Out), it is easy to determine a move ordering so that the Star2 probing phase works well enough without the need for sampling. DPW outperforms MCMS on Pig (Pig Out); this was somewhat expected since it exhibited lower regret from the first experiment. In EWN, MCMS performs evenly with DPW. Of the MCMS algorithms, s2SS does best in EWN, likely due to the strictly alternating roles which give higher chances for cutoffs to occur during the Star2 probing phase. Finally, we see that MCMS wins by large margins in Can’t Stop, the domain with the largest branching factor at chance nodes. This seems to imply that MCMS is well suited for densely stochastic games.

## 5 Conclusion and Future Work

This paper has introduced MCMS, a sparse sampling algorithm for stochastic, adversarial games. The algorithm is a natural generalization of Ballard’s ideas to a Monte-Carlo setting. We show that in one game that the mean squared error and bias are lower than MCTS. Finally, we show performance results showing MCMS outperforming classic \*-Minimax in eight of nine instances and MCTS in a fairly complex domain.

For future work, we aim to extend our investigation to the case where sampling without replacement is used. Finally, we plan to apply the algorithm in larger domains such as Backgammon, Carcassonne, Dominion, or Ra.

## Bibliography

- Peter Auer, Nicol Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2):235–256, 2002.
- David Auger. Multiple tree for partially observable monte-carlo tree search. In *Proceedings of the 2011 International Conference on Applications of Evolutionary Computation*, 2011.
- Bruce W. Ballard. The \*-minimax search procedure for trees containing chance nodes. *Artif. Intell.*, 21(3):327–350, 1983.
- C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of monte carlo tree search methods. *Computational Intelligence and AI in Games, IEEE Transactions on*, 4(1):1–43, march 2012. ISSN 1943-068X. doi: 10.1109/TCIAIG.2012.2186810.
- Hyeon S. Chang, Michael C. Fu, Jiaqiao Hu, and Steven I. Marcus. An Adaptive Sampling Algorithm for Solving Markov Decision Processes. *Operations Research*, 53(1):126–139, January 2005.
- Guillaume Chaslot, Sander Bakkes, Istvan Szita, and Pieter Spronck. Monte-Carlo Tree Search: A New Framework for Game AI. In *Fourth Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE 2008)*, 2008a.
- Guillaume Chaslot, Mark Winands, H. Jaap van den Herik, Jos Uiterwijk, and Bruno Bouzy. Progressive strategies for monte-carlo tree search. *New Mathematics and Natural Computation*, 4(3):343–357, 2008b.
- Paolo Ciancarini and Gian Piero Favini. Monte carlo tree search in kriegspiel. *Artificial Intelligence*, 174(11):670–684, 2010.
- Adrien Couetoux, Jean-Baptiste Hoock, Nataliya Sokolovska, Olivier Teytaud, and Nicolas Bonnard. Continuous Upper Confidence Trees. In *LION’11: Proceedings of the 5th International Conference on Learning and Intelligent Optimization*, Italy, January 2011. URL <http://hal.archives-ouvertes.fr/hal-00542673>.
- Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *Proceedings of the 5th international conference on Computers and games*, CG’06, pages 72–83, Berlin, Heidelberg, 2007a. Springer-Verlag. ISBN 3-540-75537-3, 978-3-540-75537-1. URL <http://dl.acm.org/citation.cfm?id=1777826.1777833>.
- Rémi Coulom. Computing ELO ratings of move patterns in the game of go. *International Computer Games Association*, 30(4):198–208, 2007b.
- Hilmar Finnsson and Yngvi Björnsson. Simulation-based approach to general game playing. In *The Twenty-Third AAAI Conference on Artificial Intelligence*, pages 259–264. AAAI Press, 2008.
- Sylvain Gelly, Yizao Wang, Rémi Munos, and Olivier Teytaud. Modification of uct with patterns in monte-carlo go. Technical Report RR-6062, Institut National de Recherche en Informatique et en Automatique (INRIA), November 2006.
- Thomas Hauk, Michael Buro, and Jonathan Schaeffer. Rediscovering \*-minimax search. In *Proceedings of the 4th international conference on Computers and Games*, CG’04, pages 35–50, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-32488-7, 978-3-540-32488-1. doi: 10.1007/11674399\_3. URL [http://dx.doi.org/10.1007/11674399\\_3](http://dx.doi.org/10.1007/11674399_3).
- Cathleen Heyden. Implementing a Computer Player for Carcassonne. Master’s thesis, Department of Knowledge Engineering, Maastricht University, 2009.
- Michael J. Kearns, Yishay Mansour, and Andrew Y. Ng. A sparse sampling algorithm for near-optimal planning in large Markov Decision Processes. In *IJCAI*, pages 1324–1331, 1999.

- Levente Kocsis and Csaba Szepesvári. Bandit based Monte-Carlo planning. In *ECML*, pages 282–293, 2006.
- Richard J. Lorentz. An MCTS program to play einstein würfelt nicht! In *Proceedings of the 12th International Conference on Advances in Computer Games*, 2011.
- Todd W. Neller and Clifton G.M. Pressor. Optimal play of the dice game pig. *Undergraduate Mathematics and Its Applications*, 25(1):25–47, 2004.
- Raghuram Ramanujan and Bart Selman. Trade-offs in sampling-based adversarial planning. In *ICAPS*, 2011.
- Stuart J. Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach* (3. internat. ed.). Pearson Education, 2010. ISBN 978-0-13-207148-2.
- Sid Sackson. Can’t Stop. *Ravensburger*, 1980.
- John Scarne. Scarne on dice. *Harrisburg, PA: Military Service Publishing Co*, 1945.
- István Szita, Guillaume Chaslot, and Pieter Spronck. Monte-carlo tree search in settlers of catan. In *Proceedings of Advances in Computer Games (ACG)*, pages 21–34, 2010.
- Joel Veness, Marc Lanctot, and Michael Bowling. Variance reduction in monte-carlo tree search. In J. Shawe-Taylor, R.S. Zemel, P. Bartlett, F.C.N. Pereira, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems 24*, pages 1836–1844. 2011.
- Thomas J. Walsh, Sergiu Goschin, and Michael L. Littman. Integrating sample-based planning and model-based reinforcement learning. In *AAAI*, 2010.
- Mark H. M. Winands and Yngvi Björnsson. Evaluation function cased monte-carlo lines of action. In *Proceedings of the 12th International Conference on Advances in Computer Games, ACG’09*, pages 33–44, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-12992-7, 978-3-642-12992-6. doi: 10.1007/978-3-642-12993-3\_4. URL [http://dx.doi.org/10.1007/978-3-642-12993-3\\_4](http://dx.doi.org/10.1007/978-3-642-12993-3_4).
- Mark H.M. Winands, Yngvi Björnsson, and Jahn-Takeshi Saito. Monte carlo tree search in lines of action. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):239–250, 2010.

# An Overview of Search Techniques in Multi-Player Games

J. (Pim) A. M. Nijssen and Mark H. M. Winands

Department of Knowledge Engineering, Faculty of Humanities and Sciences,  
Maastricht University, Maastricht, The Netherlands  
{pim.nijssen,m.winands}@maastrichtuniversity.nl

**Abstract.** In this paper we compare several search techniques for multi-player games. We test the performance of the minimax-based search techniques  $\max^n$ , paranoid search and Best-Reply Search. Furthermore, we investigate how the tree structure of each of the minimax-based techniques can be applied in MCTS. The test domain consists of four different multi-player games: Chinese Checkers, Focus, Rolit and Blokus. Based on the experimental results, we may conclude that Best-Reply Search is generally the best minimax-based search technique. Monte-Carlo Tree Search performs best with the  $\max^n$  tree structure.

## 1 Introduction

Multi-player games are games that can be played by more than 2 players. They have several properties that makes them an interesting challenge for computers. First, contrary to 2-player games, pruning in search trees is considerably more difficult. Second, the opponents' moves are more unpredictable, as coalitions may occur.

Over the past years, several tree search techniques have been developed for playing multi-player games. In 1986, Luckhardt and Irani proposed a modification of the minimax-search technique to play multi-player games, called  $\max^n$  [10]. In 2000, Sturtevant and Korf proposed the paranoid search algorithm [17]. With this technique they showed, in the trick-based card game Sergeant Major, that much more pruning is possible than in  $\max^n$ . However, due to the, often incorrect, assumption that all opponents cooperate against the root player, a paranoid player often plays too defensively. Trying to overcome this shortcoming of the paranoid algorithm, several techniques have been developed to make the algorithm less paranoid. In 2005, Lorenz and Tscheuschner proposed the coalition-mixer algorithm for 4-player chess [9] and in 2009, Zuckerman *et al.* proposed the MP-Mix algorithm [19]. This algorithm uses an evaluation function to determine which search technique should be used. Another algorithm was proposed by Schadd and Winands in 2011, namely Best-Reply Search (BRS) [14]. This algorithm performs significantly better than paranoid search in various multi-player games.

Over the past years, Monte-Carlo Tree Search (MCTS) [6, 8] has become a popular technique for playing multi-player games as well. MCTS is a best-first

search technique that, instead of an evaluation function, uses simulations to guide the search. This algorithm is able to compute mixed equilibria in multi-player games [16], contrary to  $\max^n$ , paranoid and BRS. MCTS is used in a variety of multi-player games, such as Focus [11, 12], Chinese Checkers [11, 12, 16], Hearts [16], Spades [16], and multi-player Go [4].

In this paper we compare several search techniques that have been developed over the years. We test the performance of  $\max^n$ , paranoid and BRS in four different multi-player games. Furthermore, we investigate how the tree structure of these techniques can be applied in the MCTS framework.

The paper is structured as follows. In Section 2 we give an overview of the search techniques used in this paper. Next, in Section 3 we explain the rules and applied domain knowledge of the four games. In Section 4 we describe the experiments and the results. Finally, in Section 5 we provide the conclusions and an outline of future research.

## 2 Search Techniques

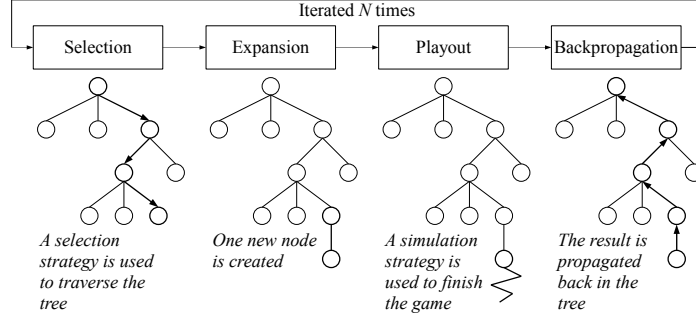
In this section we discuss the search techniques investigated in this paper. In Subsection 2.1 we explain the three minimax-based search techniques:  $\max^n$ , paranoid and BRS. In Subsection 2.2 we briefly discuss MCTS and explain how the tree structure of each of the minimax-based techniques can be applied in MCTS.

### 2.1 Minimax-based search techniques

The traditional algorithm for playing multi-player games is  $\max^n$  [10]. This technique is an extension of minimax search to multi-player games. In the leaf nodes of the search tree, each player is awarded a payoff. Each player chooses the child with the highest payoff. A disadvantage of  $\max^n$  is that only a limited amount of pruning is possible. Shallow pruning [17] is the easiest and safest way to achieve some cut-offs.

*Paranoid* search [17] assumes that all opponents have formed a coalition against the root player. Using this assumption, the game can be reduced to a 2-player game where the root player is represented in the tree by MAX nodes and the opponents by MIN nodes. The advantage of this assumption is that  $\alpha\beta$ -like pruning [7] is possible in the search tree, allowing deeper searches in the same amount of time. The disadvantage is that, because of the often incorrect paranoid assumption, the player may become too defensive.

In 2011, Schadd and Winands proposed a new algorithm for playing multi-player games, namely *Best Reply Search* (BRS) [14]. This technique is similar to paranoid search, but instead of allowing all opponents to make a move, only one opponent is allowed to do so. The advantage of this technique is that more MAX nodes are investigated. The disadvantage is that, if passing is not allowed, illegal positions or positions that are unreachable in the actual game are taken into account.



**Fig. 1.** Monte-Carlo Tree Search scheme (Slightly adapted from [5]).

## 2.2 Monte-Carlo Tree Search

Monte-Carlo Tree Search (MCTS) [6, 8] is a search technique that gradually builds up a search tree, guided by Monte-Carlo simulations. In contrast to classic search techniques such as  $\alpha\beta$ -search, it does not require a heuristic evaluation function.

The MCTS algorithm consists of four phases [5]: selection, expansion, payout and backpropagation (see Fig. 1). By repeating these four phases iteratively, the search tree is constructed gradually. The tree is traversed using the Upper Confidence bounds applied to Trees (UCT) [8] selection strategy. In our program, UCT has been enhanced with Progressive History [11]. The child  $i$  with the highest score  $v_i$  in Formula 1 is selected.

$$v_i = \bar{s}_i + C \sqrt{\frac{\ln(n_p)}{n_i}} + W \frac{\bar{s}_a}{n_i(1 - \bar{s}_i) + 1} \quad (1)$$

In this formula,  $\bar{s}_i$  denotes the win rate of child  $i$ , where  $\bar{s}_i \in [0, 1]$ . The variables  $n_i$  and  $n_p$  denote the total number of times that child  $i$  and parent  $p$  have been visited, respectively.  $C$  is a constant that determines the exploration factor of UCT. In the Progressive History part,  $\bar{s}_a$  represents the win rate of move  $a$ .  $W$  is a constant that determines the influence of Progressive History.

The basic MCTS algorithm uses a tree structure which is analogous to the  $\max^n$  search tree. It is possible to apply the paranoid and BRS tree structures to MCTS as well. The idea of using a paranoid tree structure in MCTS was presented by Cazenave [4], however he did not implement or test it. When using paranoid search or BRS in MCTS, the opponents use a different UCT formula. Instead of considering their own win rate, they try to minimize the win rate of the root player. In the MIN nodes of the tree, the following modified version of Formula 1 is used.

$$v_i = (1 - \bar{s}_i) + C \sqrt{\frac{\ln(n_p)}{n_i}} + W \frac{(1 - \bar{s}_a)}{n_i \bar{s}_i + 1} \quad (2)$$

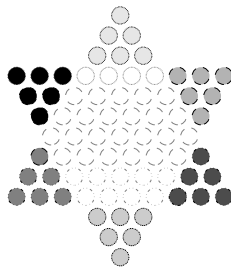


Fig. 2. A Chinese Checkers board.

### 3 Test Domains

The performance of the search techniques is tested in four different games: Chinese Checkers, Focus, Rolit and Blokus. In this section we briefly discuss the rules and the properties of these games in Subsections 3.2 – 3.4. In Subsection 3.5 we explain the move and board evaluators for the games.

#### 3.1 Chinese Checkers

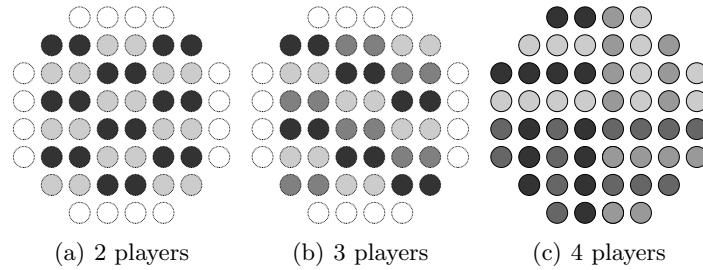
Chinese Checkers is a board game that can be played by 2 to 6 players. This game was invented in 1893 and has since then been released by various publishers under different names. Chinese Checkers is played on a star-shaped board. The most commonly used board contains 121 fields, where each player starts with 10 checkers. We decided to play on a slightly smaller board [16] (see Fig. 2). In this version, each player plays with 6 checkers. The advantage of a smaller board is that games take a shorter amount of time to complete, which means that more Monte-Carlo simulations can be performed and more experiments can be run. Also, it allows the use of a stronger evaluation function.

The goal of each player is to move all his pieces to his home base at the other side of the board. Pieces may move to one of the adjacent fields or they may jump over another piece to an empty field. It is also allowed to make multiple jumps with one piece in one turn, making it possible to create a setup that allows pieces to jump over a large distance. The first player who manages to fill his home base wins the game.

#### 3.2 Focus

Focus is an abstract multi-player strategy board game, which was invented in 1963 by Sid Sackson [13]. This game has also been released under the name *Domination*. Focus is played on an  $8 \times 8$  board where in each corner three fields are removed. It can be played by 2, 3 or 4 players. Each player starts with a number of pieces on the board. In Fig. 3, the initial board positions for the 2-, 3- and 4-player variants are given.

In Focus, pieces can be stacked on top of each other. A stack may contain up to 5 pieces. Each turn a player may move a stack orthogonally as many fields as

**Fig. 3.** Set-ups for Focus.

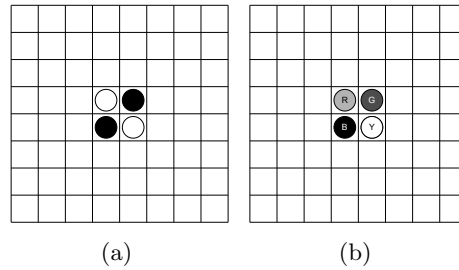
the stack is tall. A player may only move a stack of pieces if a piece of his color is on top of the stack. It is also allowed to split stacks in two smaller stacks. If a player decides to do so, then he only moves the upper stack as many fields as the number of pieces that are being moved.

If a stack lands on top of another stack, the stacks are merged. If the merged stack has a size of  $n > 5$ , then the bottom  $n - 5$  pieces are captured by the player, such that there are 5 pieces left. If a player captures one of his own pieces, he may later place one piece back on the board, instead of moving a stack. This piece may be placed either on an empty field or on top of an existing stack.

There exist two variations of the game, each with a different winning condition. In the standard version of the game, a player has won if all other players cannot make a legal move. However, such games can take a long time to finish. Therefore, we chose to use the shortened version of the game. In this version, a player has won if he has either captured certain number of pieces in total, or a number of pieces from each player. In the 2-player variant, a player wins if he has captured at least 6 pieces from the opponent. In the 3-player variant, a player has won if he has captured at least 3 pieces from both opponents or at least 10 pieces in total. In the 4-player variant, the goal is to capture at least 2 pieces from each opponent or capture at least 10 pieces in total.

### 3.3 Rolit

Rolit is a multi-player variant of the 2-player game Othello. This game was introduced in 1975. It is similar to a game invented around 1880, called Reversi. This game was invented by either Lewis Waterman or John W. Mollett. At the end of the 19th century it gained much popularity in England and in 1898, games publisher Ravensburger started producing the game as one of its first titles. Othello is played by 2 players, Black and White, on an  $8 \times 8$  board. On this board, so-called discs are placed. Discs have two different sides: a black one and a white one. If a disc on the board has its black side faced up, it is owned by player Black and if it has its white side up, it belongs to player White. The game starts with four discs on the board, as shown in Fig. 4(a). Black always starts the game, and the players take turns alternately. When it is a player's turn he has to place a disc on the board in such a way that he captures at least one of the opponents discs. A disc is captured when it lies on a straight line between



**Fig. 4.** Set-ups for Othello (a) and Rolit (b).

the placed disc and another disc of the player making the move. Such a straight line may not be interrupted by an empty square or a disc of the player making the move. All captured discs are flipped and the turn goes to the other player. If a player cannot make a legal move, he has to pass. If both players have to pass, the game is over. The player who owns the most discs, wins the game.

For Rolit, the rules are slightly different. Rolit can be played by up to 4 players, called Red, Yellow, Green and Blue. The initial board position is shown in Fig. 4(b). The largest difference is that if a player cannot capture any pieces, which will occur during the first few rounds of a 4-player game, he may put a piece orthogonally or diagonally adjacent to any of the pieces already on the board. Using this rule, passing does not occur and the game is finished when the entire board is filled. The scoring is similar to Othello; the player owning the most pieces wins. We remark that, contrary to Focus and Chinese Checkers, Rolit can end in a draw between several players.

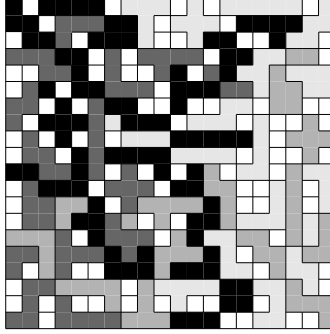
### 3.4 Blokus

Blokus is a 4-player tile placement game developed by Bernard Tavitian in 2000. The board consists of  $20 \times 20$  squares. Each player receives 21 pieces varying in size from one to five squares in all possible shapes. Alternately, the players place one of their pieces on the board. The pieces may be rotated in any way. The difficulty in this game is that any square may only be occupied by one piece and two pieces of the same player may not be orthogonally adjacent. However, they have to be adjacent diagonally to any of the player's pieces already on the board. The first pieces of the players should all be placed in one of the corners.

The game finishes when none of the players can place a piece on the board anymore. The player who has the largest number of squares on the board occupied is the winner. Note that, similar to Rolit, draws can occur. However, there is one tie breaker in this game. If more than one player manages to place all pieces on the board, the winner is the player who placed the piece of size 1 on the board during the last round.

### 3.5 Domain Knowledge

For the minimax-based techniques, a board evaluator is necessary to evaluate the leaf nodes of the search tree. This evaluator computes a heuristic value for each



**Fig. 5.** A finished game of Blokus.

player, based on the current board position. A move evaluator is used for static move ordering. The evaluator assigns a value to a move, without considering the board position. The move evaluator is also used in the MCTS-framework, for determining the moves in the  $\epsilon$ -greedy playouts [18].

For Chinese Checkers, the board evaluator uses a lookup table [16]. This table stores, for each possible configuration of pieces, the minimum number of moves a player should perform to get all pieces in the home base, assuming that there are no opponents' pieces on the board. For any player, the value of the board equals  $28 - m$ , where  $m$  is the value stored in the table which corresponds to the piece configuration of the player. We remark that 28 is the highest value stored in the table. The move evaluator of Chinese Checkers uses the function  $d_s - d_t$ , where  $d_s$  is the distance of the source location of the piece that is moved to the home base, and  $d_t$  the distance of the target location to the home base.

For Focus, the board evaluator is based on the minimum number of pieces each player needs to capture to win the game,  $r$ , and the number of stacks each player controls,  $c$ . For each player, the score is calculated using the formula  $600 - 100r + c$ . The move evaluator applies the function  $10(n + t) + s$ , where  $n$  is the number of pieces moved,  $t$  is the number of pieces on the target location, and  $s$  is the number of stacks the player gained. The value of  $s$  can be 1, 0, or -1.

For Rolit, the board evaluator is similar to the pattern based evaluation function used by Buro in his Othello program LOGISTELLO [3]. Over 90,000 games from the WTHOR database<sup>1</sup> were analyzed on 12 different patterns for 15 stages (4 moves per stage) in the game. For each pattern, the average score at the end of the game is stored. To use this pattern database in Rolit, we use the assumption that all of the opponent's pieces have the same color [14]. This reduces the accuracy, but it is unfeasible to create a pattern database for four colors. The move evaluator depends on the location of the square where the piece is placed. The values of the squares are displayed in Fig. 6.

<sup>1</sup> <http://www.ffothello.org/info/base.php>

5	2	4	3	3	4	2	5
2	1	3	3	3	3	1	2
4	3	4	4	4	4	3	4
3	3	4	4	4	4	3	3
3	3	4	4	4	4	3	3
4	3	4	4	4	4	3	4
2	1	3	3	3	3	1	2
5	2	4	3	3	4	2	5

**Fig. 6.** The values of the squares in Rolit.

For Blokus, the board evaluator counts the number of squares that each player has occupied. The move evaluator depends on the size of the piece that is played on the board. Large pieces are preferred over small ones.

For all board and move evaluators, a small random factor is added to the score. This factor is only to differentiate between board positions or moves that have the same value. This random factor is added to prevent the players from being deterministic.

## 4 Experiments

In this section, we describe the experiments performed. The program is written in Java [11]. For Formula 1, the constant  $C$  is set to 0.2 and  $W$  is set to 5. All MCTS-based players use  $\epsilon$ -greedy playouts with  $\epsilon = 0.05$ . These values were achieved by systematic testing. All minimax-based players use a DEEP transposition table [2] and static move ordering. Furthermore, the paranoid and BRS players use killer moves [1] and the history heuristic [15]. Finally, for the  $\max^n$  player, shallow pruning is applied [17], while the paranoid and BRS players use  $\alpha\beta$  pruning [7].

The experiments were run on a cluster consisting of AMD64 Opteron 2.4 GHz processors. For the games, there may be an advantage regarding the order of play and the number of different players. Games where not all player types are playing are not interesting, so these are not considered. Table 1 shows in how many ways the player types can be assigned. Each assignment is played multiple times until at least 1,000 games are played and each assignment was played equally often.

**Table 1.** The number of ways 2 or 3 different player types can be assigned. Between brackets is the number of games that are played per match.

Number of players	2 player types	3 player types
3	6 (1050)	6 (1050)
4	14 (1050)	36 (1044)
6	62 (1054)	540 (1080)

**Table 2.** Results of max<sup>n</sup> vs. paranoid vs. BRS

Game	Players	Time (ms)	Max <sup>n</sup>		Paranoid		BRS	
			Win rate (%)	Depth (ply)	Win rate (%)	Depth (ply)	Win rate (%)	Depth (ply)
Chinese Checkers	3	250	0.6±0.5	3.22	32.7±2.8	4.38	66.8±2.8	4.87
Chinese Checkers	3	1000	0.1±0.2	3.45	24.9±2.6	5.11	75.0±2.6	5.40
Chinese Checkers	3	5000	0.0±0.0	4.23	25.9±2.7	5.64	74.1±2.7	6.61
Chinese Checkers	4	250	4.5±1.3	3.09	7.2±1.6	3.64	88.3±1.9	4.39
Chinese Checkers	4	1000	3.5±1.1	3.66	21.2±2.5	4.83	75.3±2.6	5.01
Chinese Checkers	4	5000	3.9±1.1	4.23	19.3±2.4	5.38	76.8±2.6	5.73
Chinese Checkers	6	250	16.4±2.2	3.07	16.4±2.2	3.61	67.2±2.8	3.77
Chinese Checkers	6	1000	14.3±2.1	3.84	10.4±1.8	4.07	75.2±2.6	4.70
Chinese Checkers	6	5000	24.2±2.6	4.15	11.2±1.9	4.59	64.7±2.9	5.10
Focus	3	250	4.8±1.3	3.43	41.8±3.0	4.13	53.4±3.0	4.16
Focus	3	1000	4.0±1.2	3.87	34.0±2.9	4.84	62.0±2.9	4.96
Focus	3	5000	3.6±1.1	4.60	31.0±2.8	5.18	65.4±2.9	5.75
Focus	4	250	8.6±1.7	3.29	17.7±2.3	3.37	73.7±2.7	3.99
Focus	4	1000	6.0±1.4	3.69	22.7±2.5	4.48	71.3±2.7	4.82
Focus	4	5000	7.3±1.6	4.32	28.9±2.7	5.15	63.8±2.9	5.20
Rolit	3	250	0.8±0.5	4.48	41.4±3.0	6.31	57.9±3.0	6.15
Rolit	3	1000	11.0±1.9	5.26	28.3±2.7	7.74	60.7±3.0	7.39
Rolit	3	5000	0.3±0.3	6.01	66.5±2.9	8.97	33.2±2.8	8.61
Rolit	4	250	16.8±2.3	4.52	46.6±3.0	5.88	36.7±2.9	5.45
Rolit	4	1000	13.4±2.1	5.27	40.4±3.0	6.84	46.2±3.0	6.60
Rolit	4	5000	11.1±1.9	6.01	40.7±3.0	7.87	48.2±3.0	7.68
Blokus	4	250	22.2±2.5	1.89	29.0±2.8	2.37	48.8±3.0	2.46
Blokus	4	1000	19.6±2.4	2.27	25.8±2.7	2.97	54.7±3.0	3.36
Blokus	4	5000	11.8±2.0	2.76	20.8±2.5	3.38	67.3±2.9	4.03

#### 4.1 Minimax-based techniques

In the first set of experiments we match the three basic minimax-based players against each other: max<sup>n</sup>, paranoid and BRS. The win rates and the average search depths of the players in the different games are displayed in Table 2.

The results show that max<sup>n</sup> is by far the weakest algorithm. In every game with any number of players and time setting, max<sup>n</sup> has a significantly lower win rate than both paranoid and BRS. The exception is 6-player Chinese Checkers. Because paranoid also has barely any pruning, max<sup>n</sup> plays at least as strong as paranoid. Max<sup>n</sup> also plays relatively well in Blokus, where all players have difficulty reaching a decent search depth. Only the BRS player can reach a second level of MAX nodes. In most games, BRS is the best search technique. Overall, the BRS players can search slightly deeper than the paranoid players. The most notable exception is Rolit. In this game, the paranoid players can generally search slightly deeper. Also, with some settings paranoid outperforms BRS. This is comparable to the results achieved by Schadd and Winands [14].

#### 4.2 MCTS variants

In the second set of experiments, we test the performance of three different MCTS players. Each player uses a different tree structure: max<sup>n</sup> (MCTS-max<sup>n</sup>), paranoid (MCTS-paranoid) or BRS (MCTS-BRS). The win rates and the median number of payouts per move are summarized in Table 3.

The results reveal that MCTS clearly performs best using the standard max<sup>n</sup> tree structure. Only in Blokus, MCTS-max<sup>n</sup> is not significantly stronger than MCTS-paranoid. Paranoid and BRS perform well in the minimax framework

**Table 3.** Results of MCTS-max<sup>n</sup> vs. MCTS-paranoid vs. MCTS-BRS

Game	Players	Time (ms)	MCTS-max <sup>n</sup>		MCTS-paranoid		MCTS-BRS	
			Win rate (%)	Playouts (median)	Win rate (%)	Playouts (median)	Win rate (%)	Playouts (median)
Chinese Checkers	3	250	40.5±3.0	1,166	35.5±2.9	1,168	24.0±2.6	1,163
Chinese Checkers	3	1000	43.7±3.0	5,004	27.5±2.7	5,008	28.8±2.7	4,968
Chinese Checkers	3	5000	54.2±3.0	26,058	19.0±2.4	25,951	26.8±2.7	25,786
Chinese Checkers	4	250	42.5±3.0	898	33.0±2.9	900	24.4±2.6	891
Chinese Checkers	4	1000	49.1±3.0	4,042	29.8±2.8	4,022	21.1±2.5	3,962
Chinese Checkers	4	5000	62.1±2.9	21,680	17.9±2.3	21,531	20.1±2.4	20,893
Chinese Checkers	6	250	45.0±3.0	711	30.2±2.7	723	24.8±2.6	713
Chinese Checkers	6	1000	51.4±3.0	3,140	25.8±2.6	3,212	22.8±2.5	3,167
Chinese Checkers	6	5000	63.6±2.9	17,155	18.9±2.3	18,113	17.5±2.3	16,780
Focus	3	250	38.1±2.9	3,370	33.0±2.8	3,389	29.9±2.8	3,473
Focus	3	1000	37.4±2.9	12,745	29.0±2.7	13,058	33.5±2.9	12,837
Focus	3	5000	38.4±2.9	57,450	27.9±2.7	60,144	33.7±2.9	57,459
Focus	4	250	40.1±3.0	2,212	37.2±2.9	2,182	22.7±2.5	2,144
Focus	4	1000	39.1±3.0	9,189	33.0±2.9	9,220	27.9±2.7	8,881
Focus	4	5000	41.6±3.0	50,260	28.3±2.7	51,442	30.1±2.8	48,206
Rolit	3	250	44.3±3.0	2,011	31.6±2.8	2,024	24.1±2.6	2,017
Rolit	3	1000	55.6±3.0	8,333	26.0±2.7	8,356	18.4±2.3	8,320
Rolit	3	5000	67.3±2.8	41,553	16.6±2.3	41,459	16.1±2.2	42,049
Rolit	4	250	41.6±3.0	2,111	33.8±2.9	2,109	24.6±2.6	2,078
Rolit	4	1000	44.9±3.0	8,510	30.2±2.8	8,483	24.9±2.6	8,331
Rolit	4	5000	56.6±3.0	42,999	23.1±2.6	42,489	20.3±2.4	41,095
Blokus	4	250	34.9±2.9	184	36.2±2.9	185	28.9±2.7	177
Blokus	4	1000	33.3±2.9	776	34.7±2.9	780	32.1±2.8	749
Blokus	4	5000	33.0±2.9	4,170	34.5±2.9	4,212	32.6±2.8	4,061

because they increase the amount of pruning. Because  $\alpha\beta$ -pruning does not occur in MCTS, this advantage is nonexistent in the MCTS framework. It also becomes clear that MCTS-paranoid significantly outperforms MCTS-BRS. A possible explanation for this result is that illegal positions are reached in the tree. Performing a playout from these illegal or unreachable positions apparently leads to unreliable results.

#### 4.3 BRS versus MCTS-max<sup>n</sup>

Based on the previous sets of experiments, we can conclude that BRS is the strongest minimax-based technique and that MCTS-max<sup>n</sup> is the strongest MCTS technique. To determine which technique performs best in multi-player games, we let these two players play against each other in the final set of experiments. The results are displayed in Table 4.

From the results we can conclude that there is no clear winner. BRS significantly outperforms MCTS-max<sup>n</sup> in Focus, while MCTS-max<sup>n</sup> is stronger in Blokus and Rolit. In Chinese Checkers, the winner depends on the time settings. With a higher time setting, MCTS-max<sup>n</sup> becomes stronger than BRS. In all games, MCTS-max<sup>n</sup> performs relatively better with higher time settings.

## 5 Conclusions

Among the three minimax-based search techniques we tested, BRS turns out to be the strongest one. Overall, it reaches the highest search depth, and because of its tree structure more MAX nodes are investigated than in paranoid and max<sup>n</sup>. BRS significantly outperforms max<sup>n</sup> and paranoid in Chinese Checkers, Focus and Blokus. Only in Rolit, paranoid outperforms BRS with some settings.

**Table 4.** Results of MCTS-max<sup>n</sup> against BRS

Game	Players	Time (ms)	MCTS-max <sup>n</sup>		BRS	
			Win rate (%)	Playouts (median)	Win rate (%)	Depth (ply)
Chinese Checkers	3	250	21.7±2.5	1,145	78.3±2.5	4.75
Chinese Checkers	3	1000	42.1±3.0	5,206	57.9±3.0	5.34
Chinese Checkers	3	5000	60.4±3.0	27,639	39.6±3.0	6.34
Chinese Checkers	4	250	26.1±2.7	874	73.9±2.7	4.22
Chinese Checkers	4	1000	55.2±3.0	4,007	44.8±3.0	4.93
Chinese Checkers	4	5000	71.5±2.7	21,603	28.5±2.7	5.52
Chinese Checkers	6	250	35.4±2.9	703	64.6±2.9	3.53
Chinese Checkers	6	1000	66.9±2.8	3,227	33.1±2.8	4.40
Chinese Checkers	6	5000	90.1±1.8	16,603	9.9±1.8	4.71
Focus	3	250	9.9±1.8	1,481	90.1±1.8	4.19
Focus	3	1000	19.4±2.4	7,449	80.6±2.4	4.88
Focus	3	5000	28.3±2.7	42,326	71.7±2.7	5.59
Focus	4	250	23.2±2.6	1,546	76.8±2.6	3.88
Focus	4	1000	29.1±2.8	6,883	70.9±2.8	4.69
Focus	4	5000	41.5±3.0	39,786	58.5±3.0	5.06
Rolit	3	250	83.6±2.2	1,954	16.4±2.2	6.19
Rolit	3	1000	93.4±1.5	8,103	6.6±1.5	7.44
Rolit	3	5000	93.6±1.5	41,397	6.4±1.5	8.60
Rolit	4	250	78.7±2.5	1,946	21.3±2.5	5.54
Rolit	4	1000	85.8±2.1	8,162	14.2±2.1	6.67
Rolit	4	5000	88.6±1.9	42,860	11.4±1.9	7.60
Blokus	4	250	47.8±3.0	165	52.2±3.0	2.46
Blokus	4	1000	68.8±2.8	818	31.2±2.8	3.25
Blokus	4	5000	83.7±2.2	4,412	16.3±2.2	3.83

In the MCTS framework, the max<sup>n</sup> tree structure appears to perform best. The advantages of paranoid and BRS in the minimax framework do not apply in MCTS, because  $\alpha\beta$ -pruning is not applicable in MCTS. MCTS-paranoid outperforms MCTS-BRS and a possible reason for this is that MCTS-BRS performs playouts starting from an illegal or unreachable board position. This may lead to inaccurate results.

Finally, in a comparison between MCTS-max<sup>n</sup> and BRS, it turns out that there is no clear winner. In Focus, BRS is considerably stronger, while in Rolit and Blokus MCTS-max<sup>n</sup> significantly outperforms BRS. In Chinese Checkers, the winner depends on the thinking time. Overall, with higher time settings, the MCTS-based player performs relatively better.

In this research we investigated three basic tree search algorithms, i.e. max<sup>n</sup>, paranoid and BRS. We did not consider algorithms derived from these techniques, such as the Coalition-Mixer [9] or MP-Mix [19]. They use a combination of max<sup>n</sup> and (variations of) paranoid search. They also have numerous parameters that need to be tuned. Tuning and testing such algorithms in multi-player games is a direction of future research. Another possible future research direction is the application of paranoid search and BRS in the playout phase of MCTS. Cazenave [4] used paranoid playouts in multi-player Go, improving the performance of an MCTS player significantly.

## References

1. S.G. Akl and M.M. Newborn. The Principal Continuation and the Killer Heuristic. In *Proceedings of the ACM Annual Conference*, pages 466–473, New York, NY, USA, 1977. ACM.

2. D.M. Breuker, Uiterwijk J.W.H., H., and H.J. van den Herik. Replacement Schemes and Two-Level Tables. *ICCA Journal*, 19(3):175–180, 1996.
3. M. Buro. Experiments with Multi-ProbCut and a new high-quality evaluation function for Othello. *Games in AI Research*, pages 77–96, 1997.
4. T. Cazenave. Multi-player Go. In H.J. van den Herik, X. Xu, Z. Ma, and M.H.M. Winands, editors, *Computers and Games (CG 2008)*, volume 5131 of *LNCS*, pages 50–59, Berlin, Germany, 2008. Springer.
5. G.M.J-B. Chaslot, M.H.M. Winands, J.W.H.M. Uiterwijk, H.J. van den Herik, and B. Bouzy. Progressive strategies for Monte-Carlo Tree Search. *New Mathematics and Natural Computation*, 4(3):343–357, 2008.
6. R. Coulom. Efficient selectivity and backup operators in Monte-Carlo Tree Search. In H.J. van den Herik, P. Ciancarini, and H.H.L.M. Donkers, editors, *Computers and Games (CG 2006)*, volume 4630 of *LNCS*, pages 72–83, Berlin, Germany, 2007. Springer.
7. D.E. Knuth and R.W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975.
8. L. Kocsis and C. Szepesvári. Bandit based Monte-Carlo planning. In J. Fürnkranz, T. Scheffer, and M. Spiliopoulou, editors, *Machine Learning: ECML 2006*, volume 4212 of *LNAI*, pages 282–293, Berlin, Germany, 2006. Springer.
9. U. Lorenz and T. Tscheuschner. Player Modeling, Search Algorithms and Strategies in Multi-player Games. In H.J. van den Herik, S.-C. Hsu, T.-S. Hsu, and H.H.L.M. Donkers, editors, *Advances in Computer Games (ACG11)*, volume 4250 of *LNCS*, pages 210–224, Berlin, Germany, 2006. Springer.
10. C. Luckhardt and K.B. Irani. An algorithmic solution of n-person games. In *Proceedings of the 5th National Conference on Artificial Intelligence (AAAI)*, volume 1, pages 158–162, 1986.
11. J.A.M. Nijssen and M.H.M. Winands. Enhancements for Multi-Player Monte-Carlo Tree Search. In H.J. van den Herik, H. Iida, and A. Plaat, editors, *Computers and Games (CG 2010)*, volume 6515 of *LNCS*, pages 238–249, Berlin, Germany, 2011. Springer.
12. J.A.M. Nijssen and M.H.M. Winands. Payout Search for Monte-Carlo Tree Search in Multi-Player Games. In *Advances in Computer Games (ACG13)*, volume 7168 of *LNCS*, pages 72–83, Berlin, Germany, 2012.
13. S. Sackson. *A Gamut of Games*. Random House, New York, NY, USA, 1969.
14. M.P.D. Schadd and M.H.M. Winands. Best Reply Search for Multiplayer Games. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(1):57–66, 2011.
15. J. Schaeffer. The history heuristic. *ICCA Journal*, 6(3):16–19, 1983.
16. N.R. Sturtevant. An analysis of UCT in multi-player games. In H.J. van den Herik, X. Xu, Z. Ma, and M.H.M. Winands, editors, *Computers and Games (CG 2008)*, volume 5131 of *LNCS*, pages 37–49, Berlin, Germany, 2008. Springer.
17. N.R. Sturtevant and R.E. Korf. On pruning techniques for multi-player games. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, pages 201–207. AAAI Press / The MIT Press, 2000.
18. R.S. Sutton and A.G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, USA, 1998.
19. I. Zuckerman, A. Felner, and S. Kraus. Mixing Search Strategies for Multi-Player Games. In C. Boutilier, editor, *Proceedings of the Twenty-first International Joint Conferences on Artificial Intelligence (IJCAI-09)*, pages 646–651, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

# Heuristics and Fishing in Scrabble

Alejandro González Romero<sup>1</sup>, René Alquézar<sup>2</sup>, Arturo Ramírez Flores<sup>3</sup> and  
Francisco González Acuña<sup>4</sup>

<sup>1,2</sup>Dept. LSI. Universitat Politècnica de Catalunya (UPC)

C/ Jordi Girona 1-3, Edifici Omega, 08034 Barcelona, Spain

<sup>3,4</sup>Centro de Investigación en Matemáticas A.C. (CIMAT)

Callejón Jalisco S/N. Col. Valenciana, 36240 Guanajuato, Gto. México

<sup>4</sup>Instituto de Matemáticas, Universidad Autónoma de México (UNAM)

Circuito Exterior Ciudad Universitaria. Coyoacán 04510 México, D.F., México

[yarnalito@gmail.com](mailto:yarnalito@gmail.com), [alquezar@lsi.upc.edu](mailto:alquezar@lsi.upc.edu), [ramirez@cimat.mx](mailto:ramirez@cimat.mx), [ficomx@yahoo.com.mx](mailto:ficomx@yahoo.com.mx)

**Abstract.** Computer Scrabble has been studied mainly by using simulation [Sheppard, 2002b; Katz-Brown *et al.*, 2006]. In [Ramírez *et al.*, 2009] an alternative method which uses a heuristic function that involves probability calculations to evaluate moves is presented. This paper presents improvements of this heuristic function, tackles the problem of finding the best move for every initial 7-letter rack and reports results and future work suggested by this study.

**Keywords:** Computer Scrabble, Probabilistic Heuristic, Leave, Fishing Moves

## 1 Introduction

Leading Computer Scrabble until now rely mainly on simulation to achieve quality play [Sheppard, 2002b; Katz-Brown *et al.*, 2006]. The program Maven [Sheppard, 2002a] developed by Brian Sheppard is one of the references for this paradigm and its excellent results against top-level players are the best demonstration of the appropriateness of this approach. Jason Katz-Brown and John O’Laughlin [Katz-Brown *et al.*, 2006] implemented Quackle, a program distributed under the GNU license, which also exploits the simulation technique. Recently Opponent modeling was implemented using Bayes’ theorem to infer the opponent’s tiles based on their last move [Richards and Amir, 2007].

A difference between Heuri, a Scrabble engine presented in [Ramírez *et al.*, 2009] and other programs is that the evaluation of a *leave* (rack residue) is calculated as a sum of products of probabilities times expected rewards (see 2.2) rather than calculating values of individual tiles, summing them, and finally adjusting for synergy and other factors ([Sheppard, 2002b], Chapter 5).

Heuri uses probability techniques to select the “best” move, which is frequently a fish. A fish is a move which seeks a bingo ( in which all 7 tiles of the rack are played thereby obtaining a 50-point bonus ).

This paper gives a more ambitious evaluation function which measures board

openness and considers more 8-letter bingos. These two factors significantly improve the evaluation function in [Ramírez *et al.*, 2009].

In section 2 the evaluation functions Heuri\_2 and Heuri\_4 are given. In section 3 a formal definition of fish is proposed and also the problem of finding a presumably optimal initial move is considered. Section 4 gives the results of a 1000 game match between Heuri\_4 and Heuri\_2. Finally section 5 contains conclusions and suggestions for future work.

## 2 Probabilistic Heuristic Functions

### 2.1 Previous Heuristic

An important part of a program that plays Scrabble is the decision of what to leave on the rack. In a move  $t$  tiles are played or exchanged and  $n-t$  tiles make up the leave  $r$  (excluding the endgame,  $n = 7$ ).

In [Ramírez *et al.*, 2009] it was proposed to give a numerical evaluation of all potential moves as follows:

The following heuristic function is used to evaluate all potential moves:

$$v = j + e - d \quad (1)$$

where  $j$  is the number of points made by the move, in which  $t$  tiles are played ( it is assumed here that the bag has at least  $t$  tiles;  $j=0$  if  $t$  tiles are changed rather than played on the board );  $e$  is the expected value of a bingo, given a leave  $r$ , if  $t$  tiles are drawn randomly from the *augmented bag*, that is, the union of the bag and the opponent's rack;  $d$  is a nonnegative number which is zero if the move is not weak from a defensive point of view. From all potential moves one with maximal  $v$  is chosen.

To explain our estimate of  $e$  define a *septet* to be a lexicographically ordered string of seven characters of  $\{A,B,\dots,Z,\#\}$  (where  $\#$  is a blank) from which a 7-letter word can be constructed; for example  $\{AAAA\tilde{N}RR\}$  (yielding ARAÑARA) and  $\{ACEINR\# \}$  (yielding RECIBAN) are septets but  $\{AEEQRY\# \}$  and  $\{ADEILOS\}$  are not.

There are 130065 septets. For the calculation of  $e$  the following formula was used

$$e = \sum_{i=1}^{130065} p_i (50 + k\sigma_i) . \quad (2)$$

Let us call this heuristic **Heuri\_2**

Heuri\_2 is the heuristic used in the program Heuri presented in [Ramírez *et al.*, 2009]

Here  $p_i$  is the probability (which might be zero) of obtaining the  $i$ -th septet, given a leave  $r$  consisting of  $7 - t$  tiles, if  $t$  tiles are drawn randomly from the augmented bag;  $\sigma_i$  is the sum of the values of the characters of the  $i$ -th septet. The existence of premium squares, hook words, bingos of length greater than 7 and experimentation have led us to take presently 2.5 as the value of the constant  $k$ .

It is better to explain the calculation of  $p_i$  using an example:

Suppose that, in the beginning of the game, the first player has the rack {AAAN̄HQP} and puts aside {HQP} to exchange them, keeping the leave {AAAN̄}.

What is the probability  $p_i$  of obtaining the  $i$ -th septet {AAAĀNRR} (from which one can form the 7-letter bingo ARAN̄ARA) if one has the leave {AAAN̄}, the augmented bag is "total bag - {AAAN̄HQP}" and one draws 3 tiles from it?

Answer: If {AAAN̄} were not contained in {AAAĀNRR}  $p_i$  would be 0. However {AAAN̄} is contained in {AAAĀNRR} so we consider the difference set {AAAĀNRR} - {AAAN̄} = {ARR} = {AR<sup>2</sup>} and the augmented bag:

$$\{AAAAAAAAABBCCCC...RRRRR...XYZ##\} = \{A^9 B^2 C^4 ... R^5 ... XYZ\#^2\}$$

and one then computes  $p_i = C(9,1) * C(5,2) / C(93,3)$  (93 is the number of tiles of the augmented bag and the 3 in  $C(93,3)$  is the number of tiles that are taken from the bag) where  $C(m, n)$  is the binomial coefficient:

$$C(m,n) = m(m-1)(m-2)...(m-n+1)/n!$$

The numerator has  $n$  factors and  $C(m,n)$  is the number of  $n$ -element subsets of a set consisting of  $m$  elements. Notice that the denominator  $C(93,3)$  does not depend on the septet.

## 2.2 Improved Heuristics

Although the formula (2) is a good estimate of  $e$  when the board is open, it ignores the fact that one may have a septet in the rack which cannot be placed as a bingo on the board. This often happens when the board is closed. To account for this, one can write:

$$e = \sum_{i=1}^{130065} \delta_i p_i (50 + k \sigma_i) . \quad (3)$$

where  $\delta_i$ , which depends on the board, is 1 if the  $i$ -th septet can be placed as a bingo on the board and 0 otherwise. The calculation of  $\delta_i$ , for all  $i$ , which is independent of the rack, is a task that is feasible because Heuri has a fast move generator based on anagrams.

A collection of 1000 Scrabble games was gathered with the purpose of classifying the bingos made in those games into 3 categories according to its length : 1) seven-letter bingos, 2) eight-letter bingos and 3) X-letter bingos,  $X > 8$ . The results gave

6073 bingos divided as follows: 2003 seven-letter bingos, 3972 eight-letter bingos and 98 bingos with more than 8 letters. This reinforced a previous thought: the heuristic function should take into account the probability of making 8-letter words besides the 7-letter words.

Let us define an *octet* as a lexicographically ordered string of eight characters of  $\{A,B,\dots,Z,\#\}$  (where  $\#$  is a blank) from which an 8-letter word can be constructed.

A *reduced octet* is a lexicographically ordered string of seven characters of  $\{A,B,\dots,Z,\#\}$  which forms an octet when adding a certain new character.

Let a *pure septet* be a septet which is not a reduced octet, in other words when added any character to the pure septet it does not become an octet (there is no valid 8-letter word).

There are 130065 septets, 5712 pure septets and 284805 reduced octets. Notice that the set of septets contain all pure septets and 124353 reduced octets.

One desires to give an estimate of the probability of obtaining a 7-letter or 8-letter bingo given a leave and a board. The pure septets and the reduced octets give a total of 290517.

Then the calculation of  $e$  is given by the formula

$$e = \sum_{i=1}^{290517} \delta_i p_i (50 + k \sigma_i) . \quad (4)$$

Let us call this heuristic **Heuri\_4**

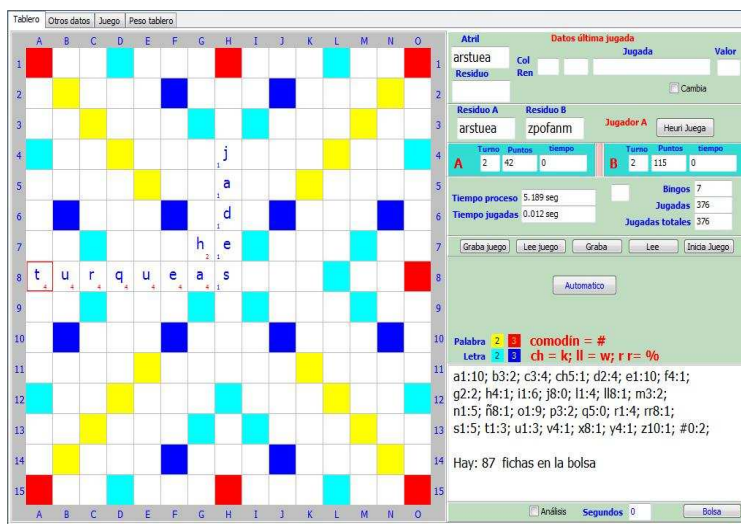
where  $p_i$ ,  $\sigma_i$ ,  $k$  and  $\delta_i$  were explained in (2) and (3).

### 3 Fishing

#### 3.1 Fishing Moves

Intuitively a *fish* is a move which seeks a bingo in the next turn. More precisely, using Heuri's probabilistic heuristic,  $v = j + e - d$ , let us define a *fish* as a move where  $e > j$  (its expected value of a bingo is greater than the points made by the move). The most common example of a fish is an exchange with  $e > 0$ . Almost all exchanges are fishing moves; however in the pre-endgame (when there are few tiles in the bag) we might change a Q tile to avoid getting stuck with it at the end; it is likely that  $e = 0$  since no more bingos can be placed on the board, or it is impossible to construct a bingo with your rack leave and the tiles left inside the bag.

The following is an example of a possible line of play using Heuri that shows and involves fishing moves:



**Figure 1.** Example of fishing moves using Heuri’s engine

The first player has played ( H4 ) jades 42pts. Then it is Heuri’s turn. Heuri has the following rack: [ r h t u q e ] Using (1)  $v = j + e - d$  with  $d = 0$ , and using (4) to calculate  $e$  Heuri’s engine gives us the following fishing moves:

**Table 1.** Examples of fishing moves for the rack: [ r h t u q e ]. See Figure 1 above

Coordinate and Exchange and leave	word or	$v$	$j$	$e$	$d$
7G	he	43.39	9	34.39	0
7H	eh	43.39	9	34.39	0
14	uh	39.85	19	20.85	0
5G	ah	39.39	5	34.39	0
5H	ah	39.39	5	34.39	0
Exch. h, Leave: ( e q r t u )		34.39	0	34.39	0
5E	huta	33.21	14	19.21	0

Therefore Heuri plays 7G he for 9 points and takes an “a” out of the bag. The first player exchanged letters and Heuri with the rack [ r a t u q e ] plays 8A turqueas 106 points ! See Figure 1 above.

Fishing plays are important because they seek high scoring moves in next turns. Brian Sheppard the developer of Maven writes the following in [Sheppard,2002b].

“Maven’s lack of a move generator for fishing may be its biggest weakness. All of the moves that Maven overlooked in its match against Adam Logan were fish.”

Brian Sheppard [2002b] also mentions that there are less than 4 million distinct racks of 7 or fewer tiles, and proposes to learn a value for every single one. Quackle

follows this advice using simulations to learn these values for different languages and lexicons. They are known as the *superleaves* and they are precalculated before the engine plays. Quackle can play without them but the quality of play drops considerably.

### 3.2 The best Initial Move

As in Chess it might be possible to build openings for Scrabble. They could consist of the first couple of moves. For the moment let us study only the very first move. In 2002 Steve Gordon proposed to build a database of all possible opening racks together with their recommended moves under simulation. There are 3,199,724 seven-letter racks in the English version using the full bag [59].

As mentioned in [Sheppard,2002b] the proposal may be impractical since it would take too much CPU time to complete the simulations of the 3,199,724 racks. Instead he suggests that a simple evaluator should be able to improve upon the evaluation of opening turns.

To start working on answering the question: given an initial 7-letter rack, what is the best move?, it is convenient to follow two different approaches simultaneously. The probabilistic heuristic shown in (4) is used as well as the simulation approach with the aid of the Quackle simulator tool.

The two Scrabble engines Heuri and Quackle examined some random initial 7-letter racks and proposed a move. Then using the Quackle simulator tool a move, which we called “best”, was obtained; comparisons and deductions through observations were then made.

### 3.3 Results, observations and deductions

Table 2 shows some Spanish initial rack examples. Let us analyze them next.

Examples 1,2 and 3 show how good a fisher Heuri is, especially when exchanging tiles. This is due to Heuri’s probabilistic heuristic function. Quackle can not find the best move because the authors did not precalculate the superleaves for the Spanish lexicon. Quackle is using the English superleaves to play in Spanish. Sometimes this error can be fixed by the simulation employed by Quackle, but the candidate solution must be among the first 21 candidates since, because of time constraints of the game, only 21 moves are simulated.

Example 4 shows the need for a method to break ties. A possible way to solve this problem is to build a defense module that penalizes moves for opening bingo lines and hot spots.

Finally in example 5 where Heuri and Quackle played 8H DETALL for 42 points, the best play is an exchange because 8H DETALL opens with an A a possible four timer bingo and it keeps a very bad leave [HQLL].

**Table 2.** Examples of initial moves in spanish

<b>Rack</b>	<b>Heuri</b>	<b>Quackle</b>	<b>Best</b>
<b>1. [NAMXQGO]</b>	Exch. 4 , leave [MANO]	8D MONGA	Exch. 4 , leave [MANO]
<b>2. [SIABRHD]</b>	Exch. 2 , leave [RIBAS]	8D HIDRA	Exch. 2 , leave [RIBAS]
<b>3. [DLAIOLS]</b>	Exch. 1 , leave [LAIDOS]	8D DOLIAS	Exch. 1 , leave [LAIDOS]
<b>4. [IVDGORY]</b>	8F, 8G, 8H VOY	8G VOY	8G VOY
<b>5. [LL HDTAEQ]</b>	8H DETALL	8H DETALL	Exch. 3 , leave [ DATE]

Heuri recently started playing in English; it is now possible to present the following English initial rack examples:

**Table 3.** Examples of initial moves in english

<b>Rack</b>	<b>Heuri</b>	<b>Quackle</b>	<b>Best</b>
<b>1. [ERTTVVW]</b>	Exch. 4, leave [RET]	Exch. 4, leave [RET]	Exch. 4, leave [RET]
<b>2. [DDGSTVY]</b>	Exch. 5, leave [ST]	Exch. 5, leave [ST]	Exch. 5, leave [ST]
<b>3. [IRLTAOE]</b>	Exch. 1, leave [LATIRE]	Exch. 1, leave [LATIRE]	Exch. 1, leave [LATIRE]
<b>4. [HNOQSTW]</b>	8H SOWTH	Exch. 3, leave [NOST]	Exch. 3, leave [NOST]
<b>5. [ADEHORT]</b>	(8G, 8H OH)=(8G, 8H HO)	8D HORDE	8G HO, leave [RATED]

Quackle is now using superleaves for the corresponding English lexicon. This can be observed in examples 1-4 where Quackle gets the best move every time. Heuri demonstrates good fishing qualities in English too. In Example 5 it almost gets the best move due to good calculation of the bingo probability of the leave [RATED], but fails to discover it due to the lack of a defensive adjustment which could break ties properly. In example 4 Heuri opens two lucrative bingo lines for the opponent (a four timer with a T and a triple timer with an S); besides, it is unlikely Heuri could use one of these bingo lines in the next turn because of the bad rack leave [QN].

## 4 Experimental Results

Table 4 shows the results from 1000 games in which Heuri\_2 heuristic competed against Heuri\_4 heuristic. Heuri\_4 scored, on average, 31 points per game more than Heuri\_2 and won 82 more games. The difference is statistically significant with  $p < 0.01$  ( a 99% confidence interval ).

**Table 4.** Results for 1000 games between Heuri\_2 and Heuri\_4

	<b>Heuri_2</b>	<b>Heuri_4</b>
Wins	459	541
Mean Score	479	510
Biggest Win	278	311

The significant game winning percentage shown by Heuri\_4 strategy and its 31 points average spread against Heuri\_2 indicate a substantial improvement between the new Heuri\_4 heuristic and the old Heuri\_2 heuristic. Another factor, not shown in the table, that indicates the superiority of Heuri\_4 vs. Heuri\_2 is that Heuri\_4 averaged 42.5 points per move while Heuri\_2 averaged 39.9 pts. This 2.6-point spread is due to a better ability of Heuri\_4 to find playable bingos.

## 5 Conclusions and Future work

Heuri\_4, unlike Heuri\_2, pays attention to the fact that one may have on the rack a bingo which is not playable on the board. Also much more importance is given to 8-letter bingos. These two characteristics are very important since it is not desirable to have a bingo that is not playable; this happens more frequently in Heuri\_2 than in Heuri\_4 due to the  $\delta_i$  factor which measures board openness. As mentioned in section 2.2, 8-letter bingos are more likely to occur than 7-letter bingos. This also helps Heuri\_4 perform better than Heuri\_2.

Besides gathering several statistics from game collections to discover useful Scrabble domain knowledge, we believe that a combination of simulation, probability methods and opponent modelling might improve even more Computer Scrabble.

One question that comes to mind is: Is it possible to find the best move for any given initial rack without using simulation in every single rack? Indeed it seems possible to achieve this task, or at least a quite good approximation. A possibility is to precalculate dangers of any given board such as bingo lines and hot spots. Then it would be possible to calculate a measure of defense for a given move. This could be done using special census for many board situations, or by using precalculated simulations in a similar fashion as the calculus of the superleaves.

More difficult is to try to find the best move in any given situation without using simulation when actually playing. A similar approach could work, but it would need many more board situations which would act as training examples.

While analyzing games it is possible to seek common situations and try to build a defense for them. An important feature one can contemplate is to take into account the opponents' last moves to try to deduce information about the opponents' rack. In other words, it is possible to include opponent modeling in the precalculations of the defense and to capture key defensive strategies while analyzing games.

Finally the probabilistic heuristic (4) can be improved by adjusting the constant value  $k$  ( currently  $k=2.5$  ). In the opening move a few experiments indicated that  $k=2.7$ , but still more experiments are needed. Besides it is likely that  $k$  is not a constant value since it probably changes throughout the game.

## References

- [Katz-Brown *et al.*, 2006] Jason Katz-Brown, John O'Laughlin, John Fultz and Matt Liberty. Quackle is an open source crossword game program released in March 2006.
- [Ramírez *et al.*, 2009] Arturo Ramírez, Francisco González Acuña, Alejandro González Romero, René Alquézar, Enric Hernández, Amador Roldán Aguilar and Ian García Olmedo. A Scrabble Heuristic Based on Probability That Performs at Championship Level. In *Proceedings of the 8<sup>th</sup> Mexican International Conference on Artificial Intelligence MICAI*, 2009.
- [Richards and Amir, 2007] Richards, M., Amir, E., Opponent modeling in Scrabble, *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence*, p.1482-1487, Hyderabad, India, January 2007.
- [Shapiro, 1979] Shapiro, S.C. A scrabble crossword game playing program. *Proceedings of the Sixth IJCAI*, p. 797-799, 1979.
- [Sheppard, 2002a] Brian Sheppard, World-championship-caliber Scrabble, *Artificial Intelligence*, v.134 n.1-2, p.241-275, January 2002.
- [Sheppard, 2002b] Brian Sheppard, *Towards Perfect Play of Scrabble*. PhD thesis, IKAT/Computer Science Department, Universiteit Maastricht, July 2002.
- [Steven, 1994] Steven A. Gordon. A Faster Scrabble Move Generation Algorithm. *Software—Practice and Experience*, v. 24(2), p.219–232, February 1994.

# Model Checking Games in GDL-II

Ji Ruan and Michael Thielscher

The University of New South Wales, Australia  
Email: {jiruan, mit}@cse.unsw.edu.au

**Abstract.** The game description language GDL has been developed as a logic-based formalism for representing the rules of arbitrary games in general game playing. A recent language extension called GDL-II allows the description of nondeterministic games with any number of players who may have incomplete, asymmetric information. In this paper, we apply *model checking* to address the problem of verifying that games specified in GDL-II satisfy appropriate temporal and knowledge conditions. We present a systematic translation of a GDL-II description to a model checking tool, and show the feasibility by two case studies.

## 1 Introduction

The general game description language GDL, which has been established as input language for general game-playing systems [7, 10], has recently been extended to GDL-II to incorporate games with nondeterministic actions and where players have incomplete/imperfect information [20]. However, not all GDL-II descriptions correspond to games, let alone meaningful, non-trivial games. [7, 10] list a few properties that are necessary for well-formed GDL games, e.g., it terminates after finite steps and all players have at least one legal move in non-terminal states. The introduction of incomplete information to GDL-II also raises new questions, e.g., can players *always know* their legal moves in non-terminal states or *know* their goal values in terminal states?

Temporal Logics have been applied to the verification of computer programs, or more broadly computer systems, initially by A. Pnueli and Z. Manna et al. [14, 11], and by E. Clarke and E. A. Emerson et al. [4]. The programs are in certain states at each time instance, and the correctness of the programs can be expressed as temporal specifications, such as “*AG¬deadlock*” meaning *the program can never enter a deadlock state*. Epistemic logics, on the other hand, are the formalisms of knowledge and beliefs. Its application in verification was originally motivated by the need to reason about communication protocols. One is typically interested in what knowledge different parties to a protocol have before, during and after a run (an execution sequence) of the protocol. [5] gives a comprehensive study on epistemic logic for multiple interacting agents.

We have previously analysed the epistemic logic behind GDL-II and in particular shown that the situation at any stage of a game can be characterised by a multi-agent epistemic (i.e., S5-) model [16]. Yet, this result only provides a static characterisation of what players know (and don’t know) at a certain stage. This paper extends such analysis with a temporal dimension, and also provides a practical method for verifying temporal and epistemic properties using a model checker named MCK [6]. The main idea is to translate a GDL-II description into the model specification language of MCK in a

systematic and equivalent way. Checking whether a property  $\varphi$  holds for description  $G$  is then equivalent to checking whether  $\varphi$  holds for the translation  $\pi(G)$ . The latter can be automatically checked in MCK.

The paper is organised as follows. Section 2 introduces GDL-II and MCK. Section 3 gives the main translation and some optimisations that can be applied to the translation. Experimental results are given for two cases in Section 4. The paper concludes with a discussion of related work and directions for further research.

## 2 GDL-II and MCK

*GDL-II* A complete game description consists of the names of (one or more) players, a specification of the initial position, the legal moves and how they affect the position, and the terminating and winning criteria. The emphasis of game description languages is on high-level, declarative game rules that are easy to understand and maintain. At the same time, GDL and its successor GDL-II have a precise semantics and are fully machine-processable. Moreover, background knowledge is not required—a set of rules is all a player needs to know to be able to play a hitherto unknown game. The description language GDL-II uses these *keywords*:

<code>role(?r)</code>	<code>?r</code> is a player
<code>init(?f)</code>	<code>?f</code> holds in the initial position
<code>true(?f)</code>	<code>?f</code> holds in the current position
<code>legal(?r, ?m)</code>	<code>?r</code> can do move <code>?m</code>
<code>does(?r, ?m)</code>	player <code>?r</code> does move <code>?m</code>
<code>next(?f)</code>	<code>?f</code> holds in the next position
<code>terminal</code>	the current position is terminal
<code>goal(?r, ?v)</code>	goal value for role <code>?r</code> is <code>?v</code>
<code>sees(?r, ?p)</code>	<code>?r</code> perceives <code>?p</code> in the next position
<code>random</code>	the random player

GDL (without `sees` and `random`) is suitable for describing finite, synchronous, and deterministic  $n$ -player games with complete information about the game state [10]. The extended game description language GDL-II allows the specification of games with randomness and imperfect/incomplete information [20]. Valid game descriptions must satisfy certain syntactic restrictions; for details we have to refer to [10] for space reasons.

The GDL-II rules in Fig. 1 formalise a simple but famous game called *Monty Hall* where a car prize is hidden behind one of three doors and where a candidate is given two chances to pick a door. The intuition behind the rules is as follows. Line 1 introduces the players' names (the game host is modelled by `random`). Lines 3–4 define the four features that comprise the initial game state. The possible moves are specified by the rules for `legal`: in step 1, the `random` player must decide where to place the car (line 6) and, simultaneously, the candidate chooses a door (line 10); in step 2, `random` opens a door that is not the one that holds the car nor the chosen one (lines 7–8); finally, the candidate can either stick to their earlier choice (noop) or switch to the other, yet unopened door (line 12 and 13, respectively). The candidate's only percept throughout the game is to see the door opened by the host (line 15) and where the car is after step 3

```

1 role(candidate). role(random).
2
3 init(closed(1)). init(closed(2)). init(closed(3)).
4 init(step(1)).
5
6 legal(random,hide_car(?d)) <= true(step(1)), true(closed(?d)).
7 legal(random,open_door(?d)) <= true(step(2)), true(closed(?d)),
8                               not true(car(?d)), not true(chosen(?d)).
9 legal(random,noop) <= true(step(3)).
10 legal(candidate,choose(?d)) <= true(step(1)), true(closed(?d)).
11 legal(candidate,noop) <= true(step(2)).
12 legal(candidate,noop) <= true(step(3)).
13 legal(candidate,switch) <= true(step(3)).
14
15 sees(candidate,?d) <= does(random,open_door(?d)).
16 sees(candidate,?d) <= true(step(3)), true(car(?d)).
17
18 next(car(?d)) <= does(random,hide_car(?d)).
19 next(car(?d)) <= true(car(?d)).
20 next(closed(?d)) <= true(closed(?d)), not does(random,open_door(?d)).
21 next(chosen(?d)) <= does(candidate,choose(?d)).
22 next(chosen(?d)) <= true(chosen(?d)), not does(candidate,switch).
23 next(chosen(?d)) <= does(candidate,switch),
24                               true(closed(?d)), not true(chosen(?d)).
25 next(step(2)) <= true(step(1)).
26 next(step(3)) <= true(step(2)).
27 next(step(4)) <= true(step(3)).
28
29 terminal <= true(step(4)).
30
31 goal(candidate,100) <= true(chosen(?d)), true(car(?d)).
32 goal(candidate, 0) <= true(chosen(?d)), not true(car(?d)).
33 goal(random,0).

```

**Fig. 1.** A GDL-II description of the Monty Hall game adapted from [21].

(line 16). The remaining rules specify the state update (rules for next), the conditions for the game to end (rule for terminal), and the payoff for the player depending on whether they got the door right in the end (rules for goal).

We refer the formal semantics of GDL-II to [16] due to limited spaces. The semantics enables us to derive a game model from a given game description.

*MCK* In this paper, we will use MCK, for ‘Model Checking Knowledge’, which is a model checker for temporal and knowledge specifications [6, 12]. The overall setup of MCK supposes a number of agents acting in an environment. This is modelled by an interpreted system where agents perform actions according to protocols. Actions and the environment may be only partially observable at each instant in time. In MCK, different approaches to the temporal and epistemic interaction and development are implemented. Knowledge may be based on current observations only, on current observations and clock value, or on the history of all observations and clock value. The last corresponds to *synchronous perfect recall*. In the temporal dimension, the specification formulas may describe the evolution of the system along a single computation, i.e., using linear time temporal logic, or they may describe the branching structure of all possible computations, i.e., using branching time or computation tree logic. We give the basic syntax of Computation Tree Logic of Knowledge (CTLK).

**Definition 1.** *The language of CTLK (with respect to a set of atomic propositions  $\Phi$ ), is given by the following grammar:*

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \psi \mid AX\varphi \mid AF\varphi \mid AG\varphi \mid A\varphi\mathcal{U}\psi \mid E\varphi\mathcal{U}\psi \mid K_i\varphi$$

where  $p \in \Phi$ . Other logic constants and connectives  $\top, \perp, \vee, \rightarrow$  are defined as usual.

We only explain the semantics informally here (cf. [17] for more details). The formulas of CTLK can be interpreted on states of game models. A game model consists of a set of agents, a set of possible states (esp., one initial state and a subset of terminal states), and a transition function for states. Intuitively  $AX\varphi$  means that for all the next states  $\varphi$  must hold;  $AF\varphi$  means that for all the paths of the game  $\varphi$  will eventually hold in the future;  $AG\varphi$  means that for all the paths of the game  $\varphi$  always hold in the future;  $A\varphi\mathcal{U}\psi$  means that for all the paths of the game,  $\varphi$  holds until  $\psi$  holds;  $E\varphi\mathcal{U}\psi$  means that there exists a path of the game,  $\varphi$  holds until  $\psi$  holds; and  $K_i\varphi$  means that  $\varphi$  holds in all the states that agent  $i$  can not distinguish from. An agent with synchronous perfect recall, can not distinguish two states if it made the same moves and had the same perceptions along two histories from the initial state.

### 3 Translation from GDL-II to MCK

Given a GDL-II description  $G$ , our program generates a translation  $\pi(G)$  as the input for MCK. The result of the translation,  $\pi(G)$ , is equivalent to  $G$  in the sense that, the game model derived from  $G$  using GDL-II semantics satisfies same formulas as the model that is derived from  $\pi(G)$  using MCK operational semantics.

**Proposition 1.** *Given a GDL-II description  $G$ , let  $\pi(G)$  be the translation from GDL-II to MCK and  $\varphi$  a temporal epistemic property, then:*

$$G \models_{GDL} \varphi \text{ iff } \pi(G) \models_{MCK} \varphi$$

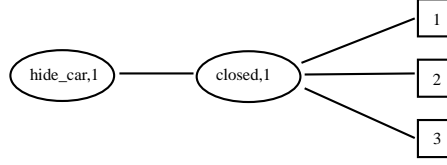
This enables us to check temporal epistemic properties against  $G$  by checking them against  $\pi(G)$ , which can be done by MCK automatically. For detailed proof, see [17].

We use the GDL-II description of Monty Hall game in Fig. 1, denoted as  $G_{MH}$ , to illustrate the whole process. The translation  $\pi$  can be divided into the following steps.

#### Computing Domains

The first step is to compute the domains, or rather supersets of the domains, of all predicates and functions of the game description. This is done by generating a dependency graph from the rules of the game description, following [19]. The nodes of the graph are the arguments of functions and predicates in game description, and there is an edge between two nodes whenever there is a variable in a rule of the game description that occurs in both arguments. Connected components in the graph share a (super-)domain.

Take, for example, the Monty Hall rules, line 3 and 6 give us the domain graph in Fig. 2, from which it can be seen that the argument of both *closed* and *hide\_car* ranges over the domain  $\{1, 2, 3\}$ .



**Fig. 2.** A domain graph for calculating domains of functions and predicates.

Once we have computed the domains, we instantiate all the variables in  $G$ . E.g., the rule in line 6-7 in the above example are turned into three rules

```

legal(random,hide_car(1))  <= true(step(1)), true(closed(1)).
legal(random,hide_car(2))  <= true(step(1)), true(closed(2)).
legal(random,hide_car(3))  <= true(step(1)), true(closed(3)).

```

### Deriving MCK Variables

The second step is to derive all the variables for  $\pi(G)$ . For this, we distinguish predicates that occur as arguments of `init` or `true`, and those that do not. The former are translated to boolean variables. For example, `step(1)` and `closed(1)` appear in  $\pi(G_{MH})$  as

```

step_1: Bool
closed_1: Bool

```

Predicates of the second type typically depend on the first type of expressions. E.g., the following rule shows that the *legal* predicate depends on two *true* predicates:

```

legal(random,hide_car(1))  <= true(step(1)), true(closed(1)).

```

There are two ways to deal with these cases: (1) translate them into booleans like above and then use valuation statements or (2) directly *define* them in terms of the booleans translated from the first type of predicates. For example, we can translate `legal(random,hide_car(1))` to a boolean (and assign a proper value later):

```

legal_random_hide_car_1: Bool
legal_random_hide_car_1:= step_1 /\ closed_1

```

or as a definition,

```

define legal_random_hide_car_1 = step_1 /\ closed_1

```

where `step_1` and `closed_1` are both booleans and `/\` is the symbol for conjunction.

The advantage of using definitions is that no new variables are introduced to the state representation. This reduces the number of variables in the overall translation and therefore potentially saves model checking time. The predicates `terminal` and `goal` can also be treated this way.

In GDL-II, *sees* predicates specify the perceptions of agents. Such predicates depend on the first type of predicates as well but they cannot be given as definitions in the translation because they have to be observable for the relevant agents in agent protocols (given below). Therefore we translate such predicates into separate boolean variables.

Since agents can recall their past moves, we make moves as part of the history, along with the perceptions of agents. While MCK's algorithms for CTLK with perfect

recall semantics do not include moves as part of the history, we need to embed such information as part of a state. Therefore we introduce an extra boolean for each `legal` instance, and replace `legal` with `did`. E.g., for `legal(random, hide_car(1))`, we add

```
did_random_hide_car_1: Bool.
```

The above procedure can already generate all the variables needed. We can do further optimisation on two kinds of predicates: these appearing in the rules with empty bodies and those never appearing in the head of rules. Under GDL-II semantics, the first kind is always true and the second kind is always false. Therefore we can replace them universally with their corresponding truth values. E.g., consider the following program:

```
1 succ(1,2)
2 succ(2,3)
3 next(step(?y)) <= true (step(?x)), succ(?x, ?y).
```

We can first translate the program to the following by using the dependency graph:

```
4 succ(1,2)
5 succ(2,3)
6 next(step(2)) <= true(step(1)), succ(1,2).
7 next(step(2)) <= true(step(2)), succ(2,2).
8 next(step(3)) <= true(step(2)), succ(2,3).
9 next(step(3)) <= true(step(3)), succ(3,3).
```

Because both `succ(2,2)`, `succ(3,3)` are always false, and `succ(1,2)`, `succ(2,3)` are always true, we replace them using their truth values. Then we can further simplify this program by removing the rules with a “False” conjunct, and by removing the “True” conjuncts universally:

```
10 next(step(2)) <= true(step(1)).
11 next(step(3)) <= true(step(2)).
```

It is easy to check that lines 10–11 are equivalent to lines 1–3 (and also lines 4–9) in terms of changes over `step` predicates. This will effectively reduce the number of variables in the translation.

### Initial Conditions

This step specifies the initial condition of  $\pi(G)$ . All the booleans translated from the predicates included within `init` are made true and all other predicates are made false. Taking  $G_{MH}$  (lines 3-4) for example, we have

```
init_cond =
closed_1 == True /\ closed_2 == True /\ closed_3 == True /\
step_1 == True /\ step_2 == False /\ step_3 == False /\ step_4 == False /\
car_1 == False /\ car_2 == False /\ car_3 == False /\ ...
```

### Agent Protocols

This step specifies the agents and their protocols during the game play. The agent binding operation binds distinct agent names to the protocols they run, and instantiate each protocol’s parameters. In GDL-II, the names of the agents are read off from the rules for the `role` predicate, and moves are read off from the `legal` predicates. Each agent has its own protocol. In MCK, protocol parameters are typed and some have *observable* before the type to indicate that agents can *see* these parameters, which are then used for agents’ accessibility relations. Taking the role candidate in  $G_{MH}$  for example, the following protocol is constructed in  $\pi(G_{MH})$ ,

```

protocol "candidate" (
  step_1: Bool, step_2: Bool, step_3: Bool, ... ,
  sees_Candidate_1: observable Bool, sees_Candidate_2: observable Bool, ...,
  did_Candidate_Choose_1: observable Bool, ... )
begin do
  legal_Candidate_Choose_1 -> <<Choose_1>>
  [] legal_Candidate_Choose_2 -> <<Choose_2>>
  [] legal_Candidate_Choose_3 -> <<Choose_3>>
  [] legal_Candidate_Noop -> <<Noop>>
  [] legal_Candidate_Switch -> <<Switch>>
od
end

```

Here the parameters prefixed with `sees_` or `did_` are observable to the candidate. The variables prefixed with `legal_` are booleans or definitions (explained above) and they represent the preconditions of moves, e.g., `legal_Candidate_Choose_1` is the precondition for agent to chose move `<< Choose_1 >>`. “[]” means non-deterministic choice, so in each step, one of these statements within `do...od` will be non-deterministically executed whenever their guards are true.

We bind agent `Candidate` to the above protocol `candidate` as this:

```

agent Candidate "candidate" ( parameter variables )

```

A protocol can be bound to multiple names, so this gives potential to code reuse when several agents share the same protocol.

### State Transition

This step specifies the statements that update the variables after agents have decided which moves to make. The first part is update the variables prefixed with `did_`. E.g., the following lines in  $\pi(G_{MH})$  indicates that agent `Candidate` made move `Choose_1` on the previous state.

```

if Candidate.Choose_1 -> did_Candidate_Choose_1 := True
[] otherwise -> did_Candidate_Choose_1 := False
fi;

```

Essentially, the effects of the agents’ joint actions will be computed so this section is connected to the `does` and `next` predicates in the description  $G$ . Here we use Clark Completion to update these variables. This can be illustrated by the variable `chosen_1` from our running example. First take all the rules that have `chosen(1)` in the head from the original description  $G_{MH}$  and instantiate ?d with 1:

```

1 next(chosen(1)) <= does(candidate,choose(1)).
2 next(chosen(1)) <= true(chosen(1)),
3   not does(candidate,switch).
4 next(chosen(1)) <= does(candidate,switch),
5   true(closed(1)),
6   not true(chosen(1)).

```

Then translate the bodies of these three rules and take their disjunction to be the guard of the resulting ‘if’ statement as follows: (note that `chosen(1)` is translated to `chosen_1` and so are the other ground atoms)

```

if (did_Candidate_Choose_1 /\ (chosen_1 /\ neg did_Candidate_Switch) /\
(did_Candidate_Switch /\ closed_1 /\ neg chosen_1) -> chosen_1 := True
[] otherwise -> chosen_1 := False
fi;

```

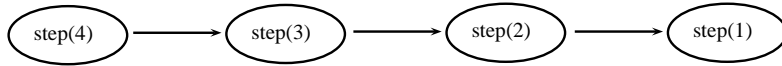
In addition, we may need to arrange the order of MCK statements carefully, because MCK's input language is imperative, which means that the statements are executed in a given order. In contrast, GDL-II is a declarative language and the order of the rules does not change the meaning (or semantics) of the whole description. Take the following example:

```
1 next(step(2)) <= true(step(1)).
2 next(step(3)) <= true(step(2)).
3 next(step(4)) <= true(step(3)).
```

The first rule means that if `step(1)` is true in the current state, then `step(2)` will be true in the next state. Note that this rule is the only rule with head `next(step(2))`, so we can apply the Clark Completion and get a statement: `step_2 := step_1`. This is fine by itself, but we have a problem when the three rules are translated together in the original order:

```
step_2 := step_1;
step_3 := step_2;
step_4 := step_3;
```

The problem is this: if `step_1` is true originally, then after executing this three statements we have all the variables to be true, whereas in GDL-II `step(3)` and `step(4)` would still be false in the next state. In fact, when we update the value of `step_3` in MCK, we need to make sure to use the guard value `step_2` from the previous state. When we follow the exact order of above, then `step_2` is updated before `step_3` is getting updated. One solution to this problem is to use a dependency graph:



This graph indicates the order in which the variables need to be updated: `step_4, ..., step_1`. Thus the correct program in MCK is as follows:

```
step_4 := step_3;
step_3 := step_2;
step_2 := step_1;
```

But what if the dependency graph has loops? Consider this example:

```
1 next(holds(x,a)) <= true(holds(x,a)).
2 next(control(x)) <= true(control(o)).
3 next(control(o)) <= true(control(x)).
```

The first loop is a self loop and does not create any problem, but the second one does pose a problem. The following program does not capture the meaning of lines 2–3 in the above description:

```
control_x := control_o;
control_o := control_x;
```

where the second statement uses a boolean updated by the first.

Our solution is to break the loop by cutting one dependency and then creating a new variable to record the last variable in the new dependency graph. Back to the above example, we can cut the dependency from `control(o)` to `control(x)`, and then use a new variable to give the following correct translation:

```

control_x_old := control_x;
control_x := control_o;
control_o := control_x_old;

```

Put in words, `control_x_old` is used to remember the old value of `control_x`.

There is another more general way to solve this problem: give all ground atoms a new variable (appended with `old`) and use them to record the values of their corresponding part in the beginning of the translation. But this can be computationally expensive for MCK in practice, because the number of variables will be doubled. It will not increase the number of reachable states of the game, but the total state space of the MCK program will be increased exponentially (i.e.,  $2^n$  for  $n$  new variables). So our above solution by using dependency graphs will be much more efficient in practice.

### Specifications

The last step is to encode the temporal and epistemic properties to be verified, using:

```

<specification type> = ... temporal and epistemic formula ...

```

The specification types we will use are “`spec_spr_nested`”, “`spec_obs_ctl`” and “`spec_spr_bmc n`”, where *spr* indicates that the model checking algorithm will use synchronous perfect recall semantics, and *obs* indicates observational semantics, *bmc* means bounded model checking. The first two algorithm use Ordered Binary Decision Diagram (OBDD) encoding and the second uses SAT encoding. We will explain the difference when we present the experimental results in the next session. Our temporal and epistemic formulas are given in CTLK syntax.

MCK checks a property  $\varphi$  on the initial state of the translated game  $\pi(G)$  with specification type  $x$ , and then when the computation is done, it returns either ‘holds’ (i.e.,  $\pi(G) \models_x \varphi$ ) or ‘fails’ (otherwise).

## 4 Experimental Results

We present some experimental results on two incomplete information games: Monty Hall and Krieg-Tictactoe. The machines have Intel Core i5-2500 Quad CPU 3.3 GHz and 8GB Ram running under GNU Linux OS 2.6.32. The MCK version is 1.0.0.

### Monty Hall

Following the method presented in the previous section, we compare the efficiency of two different translations of the Monty Hall game from Fig. 1. The first translation  $\pi_1(G_{MT})$  is the straightforward one without optimisation. It contains 43 boolean variables. The second translation  $\pi_2(G_{MT})$  is the result of applying the various optimisations given above, resulting in only 28 boolean variables.

The following properties have been checked using MCK on these two translations:

$$- \varphi_1 = (\bigwedge_m \text{legal}(\text{Candidate}, m) \rightarrow K_{\text{Candidate}} \text{legal}(\text{Candidate}, m))$$

This property intuitively means that ‘*Candidate*’ knows his legal moves at the current state. Furthermore, we define  $\varphi_2 = AX\varphi_1$  which intuitively means that

- ‘Candidate’ knows his legal moves at all next states;  $\varphi_3 = AXAX\varphi_1$ ; and  $\varphi_4 = AXAXAX\varphi_1$ .
- $\varphi_5 = AXAXAX(\text{terminal} \wedge K_{\text{Candidate}}\text{terminal})$ . This property intuitively means that for all states after three steps, the game is terminal and the candidate knows this.
- $\varphi_6 = \neg AXAXAX(\text{goal}(\text{Candidate}, 100)) \wedge \neg AXAXAX(\text{goal}(\text{Candidate}, 0))$ . This property intuitively means that it is not always the case that *Candidate* will win (i.e.,  $\text{goal}(\text{Candidate}, 100)$  is true) after three steps, nor that he will always lose.
- $\varphi_7 = AF\text{terminal}$ . This property means the game will eventually reach a terminal state.

We check  $\varphi_1$  to  $\varphi_5$  using the *spr\_nested* algorithm associated with *synchronous perfect recall* (spr) semantics because they all involve knowledge, and then check  $\varphi_6$  and  $\varphi_7$  using the *obs\_ctl* algorithm associated with the *observational* (obs) semantics because these formulas do not involve knowledge (which reduces the model checking time). For comparison, we also check  $\varphi_6$  under spr semantics. The following table shows the model checking time (measured in seconds) for these seven formulas. These formulas all hold in the initial state of  $G_{MT}$  and MCK returns corrected results.

Translation	$\varphi_1$	$\varphi_2$	$\varphi_3$	$\varphi_4$	$\varphi_5$	$\varphi_6(\text{spr})$	$\varphi_6(\text{obs})$	$\varphi_7$
$\pi_1(G_{MT})$	6.70	20.63	49.37	129.66	222.06	561.24	3.41	6.47
$\pi_2(G_{MT})$	0.53	3.10	9.01	19.59	17.25	39.32	0.50	0.49

We can see that our second translation needs notably less time than the first translation under both semantics. Also when a formula contains more temporal depth, it tends to need more time. The result on  $\varphi_6$  shows that the model checking under obs-semantics may need much less time than that of spr-semantics. For  $\varphi_7$ , it cannot be checked under spr-semantics because operator AF is not supported.

### Krieg-TicTacToe

We also studied a more complex game called Krieg-TicTacToe, an incomplete information version of TicTacToe. In this game, two players cannot see their opponent’s markings, and if one player tries to mark a position that has been occupied by the opponent, then the game master will tell the player that the move is not valid and ask it to try again. The turn-taking and winning conditions remain the same. The GDL-II description of this game (call it  $G_{KT}$ ) can be found on [ggpservers.general-game-playing.de](http://ggpservers.general-game-playing.de).

Our first translation  $\pi_1(G_{KT})$  has 111 boolean variables, and the optimised translation  $\pi_2(G_{KT})$  has 70 boolean variables. Both are around six times larger than the translations of the Monty Hall game.

We select a few representative properties:

- $\psi_1 = (\bigwedge_m \text{legal}(xplayer, m) \rightarrow K_{xplayer}\text{legal}(xplayer, m))$ . This property intuitively means that ‘xplayer’ knows his legal moves at the current state. Similarly we define  $\psi_2 = AX\psi_1$ ,  $\psi_3 = AXAX\psi_1$  and  $\psi_4 = AXAXAX\psi_1$ .

- $\psi_5 = AXAXcontrol(xplayer)$ . This property says that *after two steps xplayer is in control in all the resulting states*. This property would be true for the original TicTacToe due to the turn taking under complete information. But under incomplete information, this is not true anymore. It is because after one step, *oplayer* has control and she might try an invalid move, in that case, she will be given another chance to select a move for the next step.
- $\psi_6 = AG(tried(xplayer, 1, 1) \rightarrow AXtried(xplayer, 1, 1))$ . This property says that it is always the case that if xplayer already tried to mark the position (1,1), then in all the next states, this is still true.

We first check  $\psi_1$  to  $\psi_4$  using the *spr\_nested* algorithm. The following table only shows the model checking time (in seconds) for the second translation  $\pi_2(G_{KT})$ :

Translation	$\psi_1$	$\psi_2$	$\psi_3$	$\psi_4$
$\pi_2(G_{KT})$	86.24	1539.24	26782.95	NA

It indicates that the time complexity increases quickly with the depth of the formula. In the case of  $\varphi_4$  we could not obtain a result within 24 hours. This led us to bounded model checking (BMC) in which the specification is required to be a formula in the so-called universal fragment of a logic. The universal fragment of a logic requires that the negation operator may apply only to atomic propositions, and the modal operators can only be  $AX$ ,  $AF$ ,  $AG$ ,  $AU$  and  $K_i$ . Each specification will also be given a bound number  $n$  to indicate the depth of the game tree to be checked by MCK. The following table shows the model checking time (in seconds) for both translations:

Translation	$\psi_1(b\ 1)$	$\psi_2(b\ 2)$	$\psi_3(b\ 3)$	$\psi_4(b\ 4)$	$\psi_5(b\ 3)$	$\psi_6(b\ 5)$	$\psi_6(b\ 4)$
$\pi_1(G_{KT})$	0.27	1.70	4.40	10.34	3.69	11.87	6.87
$\pi_2(G_{KT})$	4.63	8.69	32.00	588.38	24.33	113.62	48.81

Note that each formula is given a bound when being fed to MCK; the bound  $n$  is indicated as  $(b\ n)$ . It is interesting to see that under BMC, the first translation has a better efficiency now. The main disadvantage of BMC is that it only check the model up to bound  $n$ . So if there is no counter example found under bound  $n$ , it usually does not mean that no counter example can be found at bound  $n + 1$ . E.g., formula  $\varphi_6$  has no counter example under bound 4, but it has a counter example under bound 5.

We can partially answer why a seemingly more optimised translation yields a worse result in BMC. Unlike the OBDDs, SAT algorithms are more sensitive to the complexity of boolean statements which can express complicated relations between booleans, rather than to the number of booleans. In the optimised version  $\pi_2(G_{KT})$ , we use “*definitions*” to reduce the number of variables but that, on the other hand, increases the complexity of boolean statements.

## 5 Related Work and Further Research

There are a few papers on reasoning about games in GDL and its extension GDL-II. [8] uses Answer Set Programming for verifying finitely-bounded temporal invariance properties against a given game description by structural induction. [9] extends [8] to deal

with epistemic properties for GDL-II. That formalism restricts on positive-knowledge formulas while the approach in this paper does not have such restriction and can handle more expressive epistemic and temporal formulas. [15] provides a reasoning mechanism for strategic and temporal properties but it is restricted on the original GDL for complete information games. [16] exams the epistemic logic behind GDL-II and in particular shows that the situation at any stage of a game can be characterised by a multi-agent epistemic (i.e., S5-) model. [18], an extension to [15, 16], provides both semantic and syntactic characterisations of GDL-II descriptions in terms of a strategic and epistemic logic, and shows the equivalence of these two characterisations. The current paper does not handle strategies but is more applied than [18] as we can directly using a model checker.

Some other work are related to this paper more generally in terms of planning and model checking. [1] applies symbolic planning to solve parity games equivalent to  $\mu$ -calculus model checking problems. [2] solves planning problems based on a high-level action language and model checking; and [3] gives automatic plan generation for non-deterministic domains using OBDD (which is also used by MCK). [13] introduces an approach to conformant planning (where the initial situation is not fully known and actions may have non-deterministic effects) by converting such problems into classical planning problems. It is similar to our approach in spirit but the actual formalisms are rather different.

We conclude by pointing out some directions for further research. Our case study on Krieg-TicTacToe suggests that the optimisation we have applied allows us to verify some formulas in a reasonable amount of time but is not yet fully functional for more complex formulas. However a hand-made version of Krieg-TicTacToe (with more abstraction) in MCK does suggest that MCK has no problem to cope with the amount of reachable states of Krieg-TicTacToe. So the question is, what other optimisation techniques can we find for the translation? On the other hand, we would like to investigate how to make MCK language more expressive by allowing n-ary predicates, fixpoints, loops in transition relations. This may result in a more direct translation.

Also there are logics that deal with strategic and epistemic reasoning, so we are interested in generalising this model checking approach to such logics (see [18] for a first theoretical result). Similar to [1–3, 13], we would like to also explore how plans (or strategies) can be generated via model checking for general game playing.

*Acknowledgements* We thank Xiaowei Huang, Ron van der Meyden and two anonymous reviewers for their helpful comments. This research was supported under Australian Research Council’s (ARC) *Discovery Projects* funding scheme (DP 120102023). The second author is the recipient of an Australian Research Council Future Fellowship (FT 0991348) and is also affiliated with the University of Western Sydney.

## References

1. Bakera, M., Edelkamp, S., Kissmann, P., Renner, C.D.: Solving  $\mu$ -calculus parity games by symbolic planning. In: Peled, D., Wooldridge, M. (eds.) *MoChArt. Lecture Notes in Computer Science*, vol. 5348, pp. 15–33. Springer (2008)

2. Cimatti, A., Giunchiglia, E., Giunchiglia, F., Traverso, P.: Planning via model checking: A decision procedure for ar. In: Proceedings of 4th European Conference on Planning ECP'97. pp. 130–142. Springer-Verlag (1997)
3. Cimatti, A., Roveri, M., Traverso, P.: Automatic obdd-based generation of universal plans in non-deterministic domains. In: Mostow, J., Rich, C. (eds.) AAAI/IAAI. pp. 875–881. AAAI Press / The MIT Press (1998)
4. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Kozen, D. (ed.) Logics of Programs — Proceedings 1981 (LNCS Volume 131). pp. 52–71. Springer-Verlag: Berlin, Germany (1981)
5. Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.Y.: Reasoning About Knowledge. The MIT Press: Cambridge, MA (1995)
6. Gammie, P., van der Meyden, R.: MCK: Model checking the logic of knowledge. In: Alur, R., Peled, D. (eds.) Proceedings of the 16th International Conference on Computer Aided Verification (CAV 2004). pp. 479–483. Springer (2004)
7. Genesereth, M., Love, N., Pell, B.: General game playing: Overview of the AAAI competition. AI Magazine 26(2), 62–72 (2005)
8. Haufe, S., Schiffel, S., Thielscher, M.: Automated verification of state sequence invariants in general game playing. Artificial Intelligence Journal 187–188, 1–30 (2012)
9. Haufe, S., Thielscher, M.: Automated verification of epistemic properties for general game playing. In: Proceedings of the 13th International Conference on Principles of Knowledge Representation and Reasoning (KR 2012) (2012)
10. Love, N., Hinrichs, T., Haley, D., Schkufza, E., Genesereth, M.: General Game Playing: Game Description Language Specification. Tech. Rep. LG-2006-01, Stanford Logic Group, Computer Science Department, Stanford University (2006)
11. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems. Springer-Verlag: Berlin, Germany (1992)
12. van der Meyden, R., Gammie, P., Baukus, K., Lee, J., Luo, C., Huang, X.: User manual for mck 0.5.0. Tech. rep., University of New South Wales (2010)
13. Palacios, H., Geffner, H.: Compiling uncertainty away in conformant planning problems with bounded width. J. Artif. Intell. Res. (JAIR) 35, 623–675 (2009)
14. Pnueli, A.: The temporal logic of programs. In: Proceedings of the Eighteenth IEEE Symposium on the Foundations of Computer Science. pp. 46–57 (1977)
15. Ruan, J., van der Hoek, W., Wooldridge, M.: Verification of games in the game description language. Journal Logic and Computation 19(6), 1127–1156 (2009)
16. Ruan, J., Thielscher, M.: The epistemic logic behind the game description language. In: Proceedings of the Conference on the Advancement of Artificial Intelligence (AAAI). pp. 840–845. San Francisco (2011)
17. Ruan, J., Thielscher, M.: Model checking games in GDL-II: the technical report. Tech. Rep. CSE-TR-201219, University of New South Wales (2012)
18. Ruan, J., Thielscher, M.: Strategic and epistemic reasoning for the game description language GDL-II. In: Proceedings of the European Conference on Artificial Intelligence (ECAI 2012) (2012)
19. Schiffel, S., Thielscher, M.: Fluxplayer: A successful general game player. In: Proceedings of the 22nd AAAI Conference on Artificial Intelligence (AAAI-07). pp. 1191–1196. AAAI Press (2007)
20. Thielscher, M.: A general game description language for incomplete information games. In: Proceedings of AAAI. pp. 994–999 (2010)
21. Thielscher, M.: The general game playing description language is universal. In: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI). pp. 1107–1112. Barcelona (2011)

# A General Multi-Agent Modal Logic K Framework for Game Tree Search

Abdallah Saffidine\* and Tristan Cazenave\*\*

LAMSADE, Université Paris-Dauphine, 75775 Paris Cedex 16, France

**Abstract.** We present an application of Multi-Agent Modal Logic K (MMLK) to model dynamic strategy game properties. We also provide several search algorithms to decide the model checking problem in MMLK. In this framework, we distinguish between the solution concept of interest which is represented by a class of formulas in MMLK and the search algorithm proper. The solution concept defines the shape of the game tree to be explored and the search algorithm determines how the game tree is explored. As a result, several formulas class and several of search algorithms can represent more than a dozen classical game tree search algorithms for single agent search, two-player games, and multi-player games. Among others, we can express the following algorithms in this work: depth-first search, Minimax, Monte Carlo Tree Search, Proof Number Search, Lambda Search, Paranoid Search, Best Reply Search.

## 1 Introduction

### 1.1 Motivation

Deterministic perfect information strategy games constitute a broad class of games ranging from western classic CHESS and eastern GO to modern abstract games such as HEX or multiplayer CHINESE CHECKERS [22]. Single-agent search problems and perfect information planning problems can also naturally be seen as one-player strategy games. A question in this setting is whether some agent, can achieve a specified goal from a given position. The other agents can either be assumed to be cooperative, or adversarial.

For example, an instance of such a question in CHESS is: “Can *White* force a capture of the Black Queen in exactly 5 moves?” In CHINESE CHECKERS, we could ask whether one player can force a win within ten moves. Ladder detection in GO and *helpmate* solving in CHESS also belong to this framework. The latter is an example of a cooperative two player game.

### 1.2 Intuition

The main idea of this article is that we should see the structure of a game and the behaviour of the players as two distinct parts of a game problem.

Thus, a game problem can be seen as the combination of a Game Automaton (the structure of the game) and a solution concept represented by a modal logic formula (the behaviour of the players).

---

\* abdallah.saffidine@dauphine.fr

\*\* cazenave@lamsade.dauphine.fr

### 1.3 Contributions and Outline

Our contributions in this work are:

- We establish a relation between strategy games and the Multi-Agent Modal Logic K (MMLK). Then, we show that many abstract properties of games such as those mentioned in the introduction can be formally expressed as model checking problems in MMLK with an appropriate formula (Section 2).
- We describe three possible algorithms to solve the model checking problem in MMLK. These algorithms are inspired by depth-first search, effort numbers, and Monte Carlo playouts (Section 3).
- We show that numerous previous game tree search algorithms can be directly expressed as combinations of model checking problems and model checking algorithms (Section 4).
- We demonstrate that the MMLK allows new solution concepts to be rigorously defined and conveniently expressed. Moreover, many new algorithms can be derived through new combinations of the proposed search algorithms and existing or new solution concepts (formulas). Finally, it is a convenient formal model to prove some kind of properties about game algorithms (Section 5).

We believe that these contributions can be of interest to a broad class of researchers. Indeed, the games that fall under our formalism constitute a significant fragment of the games encountered in General Game Playing [11]. We also express a generalization of the Monte Carlo Tree Search algorithm [10] that can be used even when not looking for a winning strategy. Finally, the unifying framework we provide makes understanding a wide class of game tree search algorithms relatively easy, and the implementation is straightforward.

## 2 Strategy Games and Modal Logic K

### 2.1 Game model

We now define the model we use to represent games, namely the Game Automaton (GA). We focus on a subset of the strategy games that are studied in Game Theory. The games we are interested in are turn-based games with perfect and complete information. Despite these restrictions, the class of games considered is quite large, including classics such as CHESS and GO, but also multiplayer games such as CHINESE CHECKERS, or single player games such as SOKOBAN.

Informally, the states of the game automaton correspond to possible positions over the board, and a transition from a state to another state naturally refers to a move from a position to the next.

Although the game is turn-based, we do not assume that positions are tied to a player on turn. This is natural for some games such as GO or HEX. If the turn player is tightly linked to the position, we can simply consider that the other players have no legal moves, or we can add a *pass* move for the other players that will not change the position.

We do not mark final states explicitly, neither do we embed the concept of game outcome and reward explicitly in the following definition. We rather rely on a labelling of

the states through atomic propositions. It is then possible to generate an atomic proposition for each possible game outcome and label each final state with exactly one such proposition.

**Definition 1.** A Game Automaton is a 5-tuple  $G = (\Pi, \Sigma, Q, \pi, \delta)$  with the following components :

- $\Sigma$  is a non-empty finite set of agents (or players)
- $\Pi$  is a non-empty set of atomic propositions
- $Q$  is a set of game states
- $\pi : Q \rightarrow 2^\Pi$  maps each state  $q$  to its labels, the set of atomic propositions that are true in  $q$
- $\delta : Q \times \Sigma \rightarrow 2^Q$  is a transition function that maps a state and an agent to a set of next states.

We write  $q \xrightarrow{a} q'$  when  $q' \in \delta(q, a)$ . We understand  $\delta$  as: in a state  $q$ , agent  $a$  is free to choose as the next state any  $q'$  such that  $q \xrightarrow{a} q'$ .

Note that we lift the restriction that the turn order is fixed and that in a given position, only one player can move. That is, we assume that any player can move from a given position if asked to. This generalisation is straightforward for many games. For the other games where moves for non-turn players cannot be conceived easily, we either add a single `pass` move or simply accept that there are no legal moves for non-turn players.

We will assume for the remainder of the paper that one distinguished player is denoted by  $A$  and the other players (if any) are denoted by  $B$  (or  $B^1, \dots, B^k$ ). Assume two distinct atomic propositions  $w$  and  $l$ , such that  $w$  is understood as a label of terminal positions won by  $A$ , while  $l$  is understood as a label of terminal positions not won by  $A$ .<sup>1</sup>

## 2.2 Multi-Agent Modal Logic K

Modal logic [5] is often used to reason about the knowledge of agents in a multi-agent environment. In such environments, the states in the GA are interpreted as possible worlds and additional constraints are put on the transition relation which is interpreted through the concepts of knowledge or belief. In this work, though, the transition relation is interpreted as a *legal move* function, and we do not need to put additional constraints on it. Since we do not want to reason about the epistemic capacities of our players, we use the simplest fragment of Multi-Agent Modal Logic K (MMLK) [5].

**Syntax** Let  $\Pi$  be a finite set of state labels and  $\Sigma$  be finite set of agents. We define the *Multi-Agent Modal Logic K (MMLK)* over  $\Pi$  and  $\Sigma$ , noted  $T$ , as follows:

**Definition 2.** The MMLK  $T$  is defined inductively.

$$\begin{aligned} & \forall p \in \Pi, p \in T \\ & \forall \phi_1, \phi_2 \in T, \neg\phi_1 \in T, (\phi_1 \wedge \phi_2) \in T \\ & \forall a \in \Sigma, \forall \phi \in T, \Box_a \phi \in T \end{aligned}$$

---

<sup>1</sup> Note that the atom  $l$  is not formally needed, as it can be defined using  $w$  and  $\delta$ .

That is a formula (or *threat*) is either an atomic proposition, the negation of a formula, the conjunction of two formulas, or the modal operator  $\Box_a$  for a player  $a$  applied to a formula. We read  $\Box_a \phi$  as *all moves for agent  $a$  lead to states where  $\phi$  holds*.

We define the following syntactic shortcuts.

- $\phi_1 \vee \phi_2 \equiv \neg(\neg\phi_1 \wedge \neg\phi_2)$
- $\Diamond_a \phi \equiv \neg \Box_a \neg\phi$

We read  $\Diamond_a \phi$  as *there exists a move for agent  $a$  leading to a state where  $\phi$  holds*. The precedence of  $\Diamond_a$  and  $\Box_a$ , for any agent  $a$ , is higher than  $\vee$  and  $\wedge$ , that is,  $\Diamond_a \phi_1 \vee \phi_2 = (\Diamond_a \phi_1) \vee \phi_2$ .

**Semantics** For a GA  $G = (\Pi, \Sigma, Q, \pi, \delta)$ , a state  $q$  in  $Q$ , and a formula  $\phi$ , we write  $G, q \models \phi$  when state  $q$  satisfies  $\phi$  in game  $G$ . We omit the game  $G$  when obvious from context. The formal definition of satisfaction is as follows.

- $q \models p$  with  $p \in \Pi$  if  $p$  is a label of  $q$ :  $p \in \pi(q)$
- $q \models \neg\phi$  if  $q \not\models \phi$
- $q \models \phi_1 \wedge \phi_2$  if  $q \models \phi_1$  and  $q \models \phi_2$
- $q \models \Box_a \phi$  if for all  $q'$  such that  $q \xrightarrow{a} q'$ , we have  $q' \models \phi$ .

It can be shown that the semantics for the syntactic shortcuts defined previously behave as expected.

**Proposition 1.**

- $q \models \phi_1 \vee \phi_2$  if and only if  $q \models \phi_1$  or  $q \models \phi_2$
- $q \models \Diamond_a \phi$  if there exists an actions of agent  $a$  in  $q$ , such that the next state satisfies  $\phi$ :  $\exists q' \xrightarrow{a} q', q' \models \phi$ .

### 2.3 Formalization of some game concepts

We now proceed to define several classes of formulas to express interesting properties about games.

**Reachability** A natural question that arises in one-player games is *reachability*. In this setting, we are not interested in reaching a specific state, but rather in reaching any state satisfying a given property.

**Definition 3.** We say that a player  $A$  can reach a state satisfying  $\phi$  from a state  $q$  in exactly  $n$  steps if  $q \models \underbrace{\Diamond_A \dots \Diamond_A}_{n \text{ times}} \phi$ .

**Winning strategy** We now proceed to express the concept of having a winning strategy in a finite number of moves in an alternating two-player game.

**Definition 4.** Player  $A$  has a winning strategy of depth less or equal to  $n$  in state  $q$  if  $q \models \text{WS}_{\alpha_n}$ , where  $\text{WS}_{\alpha_n}$  is defined as

- $\text{WS}_{\alpha_0} = \text{WS}_{\beta_0} = w$
- $\text{WS}_{\alpha_n} = w \vee (\neg l \wedge \Diamond_A \text{WS}_{\beta_{n-1}})$
- $\text{WS}_{\beta_n} = w \vee (\neg l \wedge \Box_B \text{WS}_{\alpha_{n-1}})$

*Ladders* The concept of *ladder* occurs in several games, particularly GO [16] and HEX. A threatening move for player  $A$  is a move such that, if it was possible for  $A$  to play a second time in a row, then  $A$  could win. A ladder is a sequence of threatening moves by  $A$  followed by defending moves by  $B$ , ending with  $A$  fulfilling their objective.

**Definition 5.** Player  $A$  has a ladder of depth less or equal to  $n$  in state  $s$  if  $q \models L_{\alpha_n}$ , where  $L_{\alpha_n}$  is defined as

- $L_{\alpha_0} = L_{\beta_0} = w$
- $L_{\alpha_n} = w \vee (\neg l \wedge \Diamond_A (w \vee (\Diamond_A w \wedge L_{\beta_{n-1}})))$
- $L_{\beta_n} = w \vee (\neg l \wedge \Box_B L_{\alpha_{n-1}})$

For instance, Figure 1a presents a position of the game HEX where the goal for each player is to connect their border by putting stones of their color. In this position, *Black* can play a successful ladder thereby connecting the left group to the bottom right border.

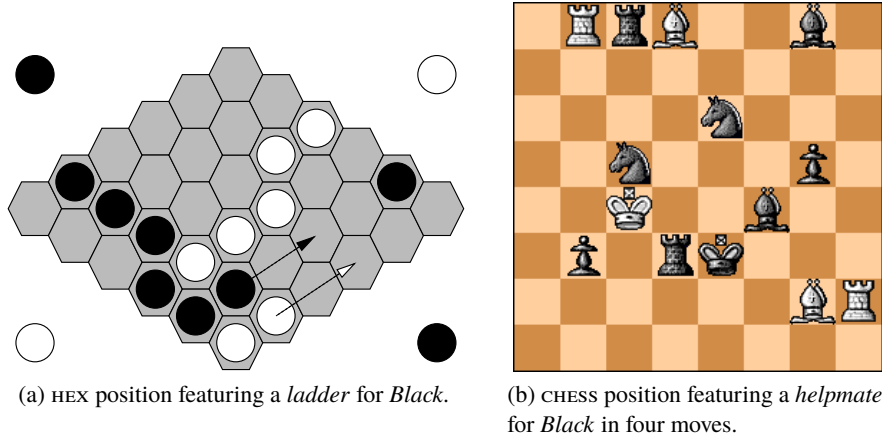


Fig. 1: Game positions illustrating the concepts of *ladder* and *helpmate*.

*Helpmates* In a CHESS *helpmate*, the situation seems vastly favourable to player *Black*, but the problemist must find a way to have the Black king checkmated. Both players move towards this end, so it can be seen as a cooperative game. *Black* usually starts in helpmate studies. See Figure 1b for an example. A helpmate in at most  $2n$  plies can be represented through the formula  $H_n$  where  $H_0 = w$  and  $H_n = w \vee \Diamond_B \Diamond_A H_{n-1}$ .

*Selfmates* A *selfmate*, on the other hand, is a situation where *Black* forces *White* to checkmate the Black King, while *White* must do their best to avoid this. *Black* starts moving in a selfmate and a position with a selfmate satisfies  $S_n$  for some  $n$ , where  $S_0 = w$  and  $S_n = w \vee \Diamond_B \Box_A S_{n-1}$ .

### 3 Search paradigms

We now define several model checking algorithms. That is, we present algorithms that allow to decide whether a state  $q$  satisfies a formula  $\phi$  ( $q \models \phi$ ).

#### 3.1 Depth First Threat Search

Checking whether a formula is satisfied on a state can be decided by a depth-first search on the game tree as dictated by the semantics given in Section 2.2. Pseudo-code for the resulting algorithm, called Depth First Threat Search (DFTS) is presented in Algorithm 1.

```

dfts (state  $q$ , formula  $\phi$ )
  switch on the shape of  $\phi$  do
    case  $p \in \Pi$ 
      return  $p \in \pi(q)$ 
    case  $\phi_1 \wedge \phi_2$ 
      return  $\text{dfts}(q, \phi_1) \wedge \text{dfts}(q, \phi_2)$ 
    case  $\neg\phi_1$ 
      return  $\neg \text{dfts}(q, \phi_1)$ 
    case  $\Box_a \phi_1$ 
      let  $l = \{q', q \xrightarrow{a} q'\}$ ;
      foreach  $q'$  in  $l$  do
        if not  $\text{dfts}(q', \phi_1)$  then
          return false
      return true

```

**Algorithm 1:** Pseudo-code for the DFTS algorithm.

#### 3.2 Best-first Search Algorithms

We can propose several alternatives to the DFTS algorithm to check a given formula in a given state. We present a generic framework to express best first search model checking algorithms. Best-first search algorithms must maintain a partial tree in memory, the shape of which is determined by the formula to be checked.

Nodes are mapped to a (state  $q$ , formula  $\phi$ ) label. A leaf is terminal if its label is an atomic proposition  $p \in \Pi$  otherwise it is non-terminal. Each node is associated to a unique position, but a position may be associated to multiple nodes.<sup>2</sup>

The following static observations can be made about partial trees:

- an internal node labelled  $(q, \neg\phi)$  has exactly one child and it is labelled  $(q, \phi)$ ;
- an internal node labelled  $(q, \phi_1 \wedge \phi_2)$  has exactly two children which are labelled  $(q, \phi_1)$  and  $(q, \phi_2)$ ;
- an internal node labelled  $(q, \Box_a \phi)$  has as many children as there are legal transition for  $a$  in  $q$ . Each child is labelled  $(q', \phi)$  where  $q'$  is the corresponding state.

```

bfs (state  $q$ , formula  $\phi$ )
let  $r$  = new node with label  $(q, \phi)$ ;
 $r$ .info  $\leftarrow$  init-leaf ( $r$ );
let  $n = r$ ;
while  $r$  is not solved do
    while  $n$  is not a leaf do
         $n \leftarrow$  select-child ( $n$ );
    extend ( $n$ );
     $n \leftarrow$  backpropagate ( $n$ );
return  $r$ 

extend (node  $n$ )
switch on the label of  $n$  do
    case  $(q, p)$ 
         $n$ .info  $\leftarrow$  info-term ( $n$ );
    case  $(q, \phi_1 \wedge \phi_2)$ 
        let  $n_1$  = new node with label  $(q, \phi_1)$ ;
        let  $n_2$  = new node with label  $(q, \phi_2)$ ;
         $n_1$ .info  $\leftarrow$  init-leaf ( $n_1$ );
         $n_2$ .info  $\leftarrow$  init-leaf ( $n_2$ );
        Add  $n_1$  and  $n_2$  as children of  $n$ ;
    case  $(q, \neg \phi_1)$ 
        let  $n'$  = new node with label  $(q, \phi_1)$ ;
         $n'$ .info  $\leftarrow$  init-leaf ( $n'$ );
        Add  $n'$  as a child of  $n$ ;
    case  $(q, \Box_a \phi_1)$ 
        let  $l = \{q', q \xrightarrow{a} q'\}$ ;
        foreach  $q'$  in  $l$  do
            let  $n'$  = new node with label  $(q', \phi_1)$ ;
             $n'$ .info  $\leftarrow$  init-leaf ( $n'$ );
            Add  $n'$  as child of  $n$ ;

backpropagate (node  $n$ )
let new_info = update ( $n$ );
if new_info =  $n$ .info  $\vee n = r$  then
    return  $n$ 
else
     $n$ .info  $\leftarrow$  new_info;
    return backpropagate ( $n$ .parent)

```

**Algorithm 2:** Pseudo-code for a best-first search algorithm.

The generic framework is described in Algorithm 2. An instance must provide a data type for node specific information which we call *node value* and the following procedures. The `info-term` defines the value of terminal leaves. The `init-leaf` procedure is called when initialising a new leaf. The `update` procedure determines how the value of an internal node evolves as a function of its label and the value of the children. The `select` procedure decides which child is best to be explored next depending on the node's value and label and the value of each child. We present possible instances in Sections 3.3 and 3.4.

### 3.3 Proof Number Threat Search (PNTS)

We present a first instance of the generic best-first search algorithm described in Section 3.2 under the name PNTS. This algorithm uses the concept of *effort numbers* and is inspired from Proof Number Search (PNS) [2, 28].

The node specific information needed for PNTS is a pair of numbers which can be positive, equal to zero, or infinite. We call them *proof number* (PN) and *disproof number* (DN). Basically, if a subformula  $\phi$  is to be proved in a state  $s$  and  $n$  is the corresponding node in the constructed partial tree, then the PN (resp. DN) in a node  $n$  is a lower bound on the number of nodes to be added to the tree to be able to exhibit a proof that  $s \models \phi$  (resp.  $s \not\models \phi$ ). When the PN reached 0 (and the DN reaches  $\infty$ ), the fact has been proved and when the PN reached  $\infty$  (and the DN reaches 0) the fact has been disproved.

The `info-term` and `init-leaf` procedures are described in Table 1, while Table 2 and 3 describe the `update` and `select-child` procedures, respectively.

Table 1: Initial values for leaf nodes in PNTS.

	Node label	PN	DN
info-term	$(q, p)$ when $p \in \pi(q)$	0	$\infty$
	$(q, p)$ when $p \notin \pi(q)$	$\infty$	0
init-leaf	$(q, \phi)$	1	1

Table 2: Determination of values for internal nodes in PNTS.

Node label	Children	PN	DN
$(q, \neg\phi)$	$\{c\}$	DN( $c$ )	PN( $c$ )
$(q, \phi_1 \wedge \phi_2)$	$C$	$\sum_C$ PN	$\min_C$ DN
$(q, \Box_a \phi)$	$C$	$\sum_C$ PN	$\min_C$ DN

<sup>2</sup> While it is possible to store the state  $q$  associated to a node  $n$  in memory, it usually is more efficient to store move information on edges and reconstruct  $q$  from the root position and the path to  $n$ .

Table 3: Selection policy for PNTS.

Node label	Children	Chosen child
$(q, \neg\phi)$	$\{c\}$	$c$
$(q, \phi_1 \wedge \phi_2)$	$C$	$\arg \min_C \text{DN}$
$(q, \Box_a \phi)$	$C$	$\arg \min_C \text{DN}$

### 3.4 Monte Carlo Proof Search (MCPS)

Monte Carlo Tree Search (MCTS) [9, 8] is a recent game tree search technique based on multi-armed bandit problems [4]. MCTS has enabled a huge leap forward in the playing level of artificial Go players. MCTS has been extended to prove wins and losses under the name MCTS Solver [31] and it can be seen as the origin of the algorithm presented in this section which we call MCPS.

The basic idea in MCPS is to evaluate whether a state  $s$  satisfies a formula via probes in the tree below  $s$ . A probe, or Monte Carlo playout, is a random subtree of the tree below  $s$  whose structure is given by the formula to be checked in  $s$ . In the original MCTS algorithm, the structure of playouts is always a path. We lift this constraint here as we want to model check elaborate formulas about states. A probe is said to be *successful* if the formulas at the leaves are satisfied in the corresponding states. Determining whether a new probe generated on the fly is successful can be done as demonstrated in Algorithm 3.

```

probe (state  $q$ , formula  $\phi$ )
  switch on the shape of  $\phi$  do
    case  $p \in \Pi$ 
      return  $p \in \pi(q)$ 
    case  $\phi_1 \wedge \phi_2$ 
      return probe ( $q, \phi_1$ )  $\wedge$  probe ( $q, \phi_2$ )
    case  $\neg\phi_1$ 
      return  $\neg$  probe ( $q, \phi_1$ )
    case  $\Box_a \phi_1$ 
      let  $q'$  be a random state such that  $q \xrightarrow{a} q'$ ;
      return probe ( $q', \phi_1$ )

```

**Algorithm 3:** Pseudo-code for a Monte-Carlo Probe.

Like MCTS, MCPS explores the GA in a best first way by using aggregates of information given by the playouts. For each node  $n$ , we need to know the total number of probes rooted below  $n$  (denoted by  $N$ ) and the number of successful probes among them (denoted by  $R$ ). We are then faced with an exploration-exploitation dilemma between running probes in nodes which have not been explored much ( $N$  is small) and running probes in nodes which seem successful (high  $\frac{R}{N}$  ratio). This concern is addressed using the UCB formula [4].

Similarly to MCTS Solver, we will add another label to the value of nodes called P. P represents the proof status and allows to avoid solved subtrees. P can take three values:  $\top$ ,  $\perp$ , or  $?$ . These values respectively mean that the corresponding subformula was proved, disproved, or neither proved nor disproved for this node.

We describe the `info-term`, `init-leaf`, `update`, and `select-child` procedures in Table 4, Table 5, and Table 6.

Table 4: Initialisation for leaf values in MCPS for a node  $n$ .

	Node label	P	R	N
info-term	$(q, p)$ where $p \in \pi(q)$	$\top$	1	1
	$(q, p)$ where $p \notin \pi(n)$	$\perp$	0	1
init-leaf	$(q, \phi)$	$?$	probe $(q, \phi)$	1

Table 5: Determination of values for internal nodes in MCPS.

Node label	Children	P	R	N
$(q, \neg\phi)$	$\{c\}$	$\neg P(c)$	$N(c) - R(c)$	$N(c)$
$(q, \phi_1 \wedge \phi_2)$	$C$	$\bigwedge_C P$	$\sum_C R$	$\sum_C N$
$(q, \Box_a \phi)$	$C$	$\bigwedge_C P$	$\sum_C R$	$\sum_C N$

Table 6: Selection policy for MCPS in a node  $n$ .

Node label	Children	Chosen child
$(q, \neg\phi)$	$\{c\}$	$c$
$(q, \phi_1 \wedge \phi_2)$	$C$	$\arg \max_{C, P(c)=?} \frac{N-R}{N} + \sqrt{\frac{2 \log N(n)}{N}}$
$(q, \Box_a \phi)$	$C$	$\arg \max_{C, P(c)=?} \frac{N-R}{N} + \sqrt{\frac{2 \log N(n)}{N}}$

## 4 Simulation of existing game tree algorithms

By defining appropriate formulas classes, we can simulate many existing algorithms by solving model checking problems in MMLK with specific search algorithms.

**Definition 6.** Let  $\phi$  be a formula,  $S$  be a model checking algorithm and  $A$  be a specific game algorithm. We say that  $(\phi, S)$  simulates  $A$  if for every game, for every state  $q$  where

*A can be applied, we have the following: solving  $q \models \phi$  with  $S$  will explore exactly the same states in the same order and return the same result as algorithm  $A$  applied to initial state  $q$ .*

Table 7 presents how combining the formulas defined later in this section with the model checking algorithms defined in Section 3 allows to simulate many important algorithms. For instance, using the DFTS algorithm to model-check an  $\text{APS}_n$  formula on a HEX position represented as a state of a GA is exactly the same as running the Abstract Proof Search algorithm on that position.

Table 7: Different algorithms expressed as a combination of a formula class and a search paradigm.

Formula	Search Paradigm		
	DFTS	PNTS	MCPS
$\pi_n$	Depth-first search		Single-player MCTS [21]
$\text{WS}_{\alpha_n}$	$\alpha\beta$ [14]	PNS [2]	MCTS Solver [31]
$\text{PA}_n$	Paranoid [26]	Paranoid PNS [19]	Multi-player MCTS [17]
$\text{LS}_{d,n}$	Lambda-search [27]	Lambda-PNS [33] <sup>1</sup>	
$\text{BRS}_n$	Best Reply Search [20]		
$\text{APS}_n$	Abstract proof search [6]		

<sup>1</sup> We actually need to change the update rule for the PN in internal  $\phi_1 \wedge \phi_2$  nodes in PNTS from  $\sum_C \text{PN}$  to  $\max_C \text{PN}$ .

#### 4.1 One-player games

Many one-player games, the so-called puzzles, involve finding a path to a terminal state. Ideally this path should be the shortest possible. Examples of such puzzles include the 15-PUZZLE and RUBIK’S CUBE.

Recall that we defined a class of formulas for reachability in exactly  $n$  steps in Definition 3. Similarly we define now a class of formulas representing the existence of a path to a winning terminal state within  $n$  moves.

**Definition 7.** *We say that agent  $A$  has a winning path from a state  $q$  if  $q$  satisfies  $\pi_n$  where  $\pi_n$  is defined as  $\pi_0 = w$  and  $\pi_n = w \vee \Diamond_A \pi_{n-1}$  if  $n > 0$ .*

#### 4.2 Two-player games

We already defined the *winning strategy* formulas  $\text{WS}_{\alpha_n}$  and  $\text{WS}_{\beta_n}$  in Definition 4. We will now express a few other interesting formulas that can be satisfied in game states in two player games.

*$\lambda$ -Trees*  $\lambda$ -trees have been introduced [27] as a generalisation of ladders as seen in Section 2.3. We will refrain from describing the intuition behind  $\lambda$ -trees here and will be satisfied with giving the formal corresponding property as they only constitute an example of the applicability of our framework.

**Definition 8.** A state  $q$  has an  $\lambda$ -tree of order  $d$  and maximal depth  $n$  for player  $A$  if  $q \models \text{LS}_{\alpha_d, n}$ , where  $\text{LS}_{\alpha_d, n}$  is defined as follows.

- $\text{LS}_{\alpha_0, n} = \text{LS}_{\alpha_d, 0} = \text{LS}_{\beta_0, n} = \text{LS}_{\beta_d, 0} = w$
- $\text{LS}_{\alpha_d, n} = w \vee \Diamond_A (\neg l \wedge \text{LS}_{\alpha_{d-1}, n-1} \wedge \text{LS}_{\beta_d, n-1})$
- $\text{LS}_{\beta_d, n} = w \vee \Box_B (\neg l \wedge \text{LS}_{\alpha_d, n-1})$

$\lambda$ -trees are a generalisation of ladders as defined in Definition 5 since a ladder is a  $\lambda$ -tree of order  $d = 1$ .

*Abstract proof trees* Abstract proof trees were introduced to address some perceived practical limitations of  $\alpha - \beta$  when facing a huge number of moves. They have been used to solve games such as PHUTBALL or ATARI-GO. We limit ourselves here to describing how we can specify in MMLK that a state is root to an abstract proof tree. Again, we refer the reader to the literature for the intuition about abstract proof trees and their original definition [6].

**Definition 9.** A state  $q$  has an abstract proof tree of order  $n$  for player  $A$  if  $q \models \text{APS}_{\alpha_n}$ , where  $\text{APS}_{\alpha_n}$  is defined as follows.

- $\text{APS}_{\alpha_0} = \text{APS}_{\beta_0} = w$
- $\text{APS}_{\alpha_n} = w \vee \Diamond_A (\neg l \wedge \text{APS}_{\alpha_{n-1}} \wedge \text{APS}_{\beta_{n-1}})$
- $\text{APS}_{\beta_n} = w \vee \Box_B (\neg l \wedge \text{APS}_{\alpha_{n-1}})$

*Other concepts* Many other interesting concepts can be similarly implemented via a class of appropriate formulas. Notably minimax search with iterative deepening, the Null-move assumption, and Dual Lambda-search [25] can be related to model checking some MMLK formulas with DFTS.

### 4.3 Multiplayer games

*Paranoid Algorithm* The Paranoid Hypothesis was developed to allow for more  $\alpha - \beta$  style safe pruning in multi-player games [26]. It transforms the original  $k + 1$ -player into a two-player game  $A$  versus  $B$ . In the new game, the player  $B$  takes the place of  $B^1, \dots, B^k$  and  $B$  is trying to prevent player  $A$  from reaching a won position. Assuming the original turn order is fixed and is  $A, B^1, \dots, B^k, A, B^1, \dots$ , we can reproduce a similar idea in MMLK.

**Definition 10.** Player  $A$  has a paranoid win of depth  $n$  in a state  $q$  if  $q \models \text{PA}_{\alpha_n}$ , where  $\text{PA}_{\alpha_n}$  is defined as follows.

- $\text{PA}_{\alpha_0} = \text{PA}_{\beta_0^i} = w$
- $\text{PA}_{\alpha_n} = w \vee \Diamond_A (\neg l \wedge \text{PA}_{\beta_{n-1}^1})$
- $\text{PA}_{\beta_n^k} = w \vee \Box_{B^k} (\neg l \wedge \text{PA}_{\alpha_{n-1}})$
- $\text{PA}_{\beta_n^i} = w \vee \Box_{B^i} (\neg l \wedge \text{PA}_{\beta_{n-1}^{i+1}})$  for  $1 \leq i < k$

*Best Reply Search* Best Reply Search (BRS) [20] is a new search algorithm for multi-player games. It consists of performing a minimax search where only one opponent is allowed to play after  $A$ . For instance a principal variation in a BRS search with  $k = 3$  opponents could involve the following turn order  $A, B_2, A, B_1, A, B_1, A, B_3, A, \dots$  instead of the regular  $A, B_1, B_2, B_3, A, B_1, B_2, B_3, \dots$ .

The rationale behind BRS is that the number of moves studied for the player in turn in any variation should only depend on the depth of the search and not on the number of opponents. This leads to an artificial player selecting moves exhibiting longer term planning. This performs well in games where skipping a move does not influence the global position too much, such as CHINESE CHECKERS.

**Definition 11.** *Player  $A$  has a best-reply search win of depth  $n$  in a state  $q$  if  $q \models \text{BRS}_{\alpha_n}$ , where  $\text{BRS}_{\alpha_n}$  is defined as follows.*

- $\text{BRS}_{\alpha_0} = \text{BRS}_{\beta_0} = w$
- $\text{BRS}_{\alpha_n} = w \vee \Diamond_A (\neg l \wedge \text{BRS}_{\beta_{n-1}})$
- $\text{BRS}_{\beta_n} = w \vee \bigwedge_{i=1}^k \Box_{B^i} (\neg l \wedge \text{BRS}_{\alpha_{n-1}})$

## 5 Creation of new game tree algorithms

We now turn to show how MMLK Model Checking framework can be used to develop new research in game tree search. As such, the goal of this section is not to put forward a single well performing algorithm, nor to prove strong theorems with elaborate proofs, but rather to demonstrate that the MMLK Model Checking is an appropriate tool for designing and reasoning about new game tree search algorithms.

*Progress Tree Search* It occurs in many two-player games that at some point near the end of the game, one player has a winning sequence of  $n$  moves that is relatively independent of the opponent's moves. For instance Figure 2 presents a HEX position won for *Black* and a CHESS position won for *White*. In both cases, the opponent's moves cannot even delay the end of the game.

To capture this intuition, we define a solution concept we name *progress tree*. The idea giving its name to the concept of progress trees is that we want the player to focus on those moves that brings them closer to a winning state, and discard the moves that are out of the winning path.

**Definition 12.** *Player  $A$  has a progress tree of depth  $2n + 1$  in a state  $q$  if  $q \models \text{PT}_{\alpha_{2n+1}}$ , where  $\text{PT}_{\alpha_{2n+1}}$  is defined as follows.*

- $\text{PT}_{\beta_0} = w$
- $\text{PT}_{\alpha_{2n+1}} = w \vee \Diamond_A (\neg l \wedge \pi_n \wedge \text{PT}_{\beta_{2n}})$
- $\text{PT}_{\beta_{2n}} = w \vee (\neg l \wedge \Box_B \text{PT}_{\alpha_{2n-1}})$

We can check states for progress trees using any of the model checking algorithms presented in Section 3, effectively giving rise to three new specialised algorithms. Note that if a player has a progress tree of depth  $2n + 1$  in some state, then they also have a winning strategy of depth  $2n + 1$  from that state (see Proposition 2). Therefore, if we

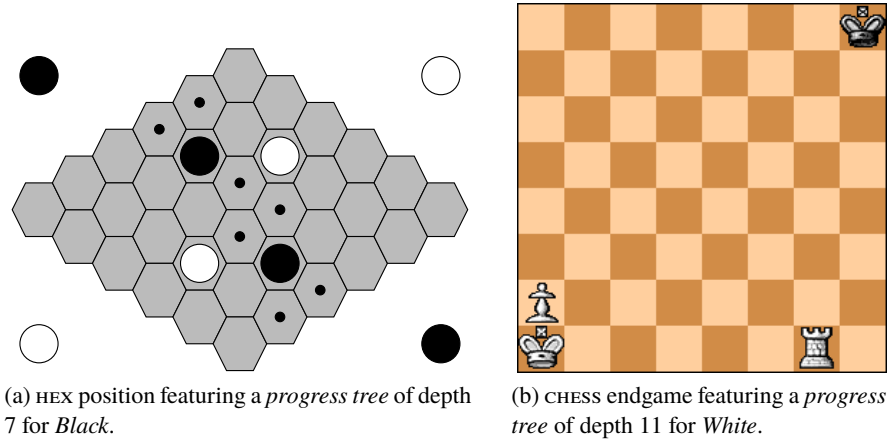


Fig. 2: Positions illustrating the concepts of *progress tree*.

prove that a player has a progress tree in some position, then we can deduce that have a winning strategy.

We tested a naive implementation of the DFTS model checking algorithms on the position in Figure 2 to check for progress trees and winning strategies. The principal variations consists for *White* in moving the pawn up to the last row and move the resulting queen to the bottom-right hand corner to deliver checkmate. To study how the solving difficulty increases with respect to the size of the formula to be checked, we model checked every position on a principal variation and present the results in Table 8.

We can see that proving that a progress tree exists becomes significantly faster than proving an arbitrary winning strategy as the size of the problem increases. We can also notice that the overhead of checking for a path at each  $\alpha$  node of the search is more than compensated by the early pruning of moves not contributing to the winning strategy.

*Examining new combinations* We have seen in Section 3 that we could obtain previously known algorithms by combining model checking algorithms with solution concepts. On the one hand, some solution concepts such a winning strategy and paranoid win, were combined with the three possible search paradigms in previous work. On the other hand, other solution concepts such as best-reply search win were only investigated within the depth-first paradigm.

It is perfectly possible to model check a best-reply search win using the MCPS algorithm, for instance, leading to a new Monte Carlo Best Reply Search algorithm. Similarly model checking abstract proof trees with PNTS would lead to a new Proof Number based Abstract Proof Search (PNAPS) algorithm. Preliminary experiments in HEX without any specific domain knowledge added seem to indicate that PNAPS does not seem to perform as well as Abstract Proof Search, though.

Finally, most of the empty cells in Table 7 can be considered as new algorithms waiting for an optimised implementation and a careful evaluation.

Table 8: Search statistics for a DFTS on positions along a principal variation of the CHESS problem in Figure 2b.

MC problem	Time (s)	Number of queries		
		atomic	listmoves	play
PT <sub>α<sub>5</sub></sub>	0.1	6040	328	5897
WS <sub>α<sub>5</sub></sub>	0.2	11172	624	5587
PT <sub>α<sub>7</sub></sub>	1.4	99269	5312	98696
WS <sub>α<sub>7</sub></sub>	3.5	194429	10621	97217
PT <sub>α<sub>9</sub></sub>	23.6	1674454	88047	1668752
WS <sub>α<sub>9</sub></sub>	63.8	3382102	181442	1691055
PT <sub>α<sub>11</sub></sub>	260.4	25183612	1297975	25106324
WS <sub>α<sub>11</sub></sub>	953.6	52209939	2759895	26104986

*Expressing properties of the algorithms* We now demonstrate that using the MMLK model checking framework for game tree search makes some formal reasoning straightforward. Again, the goal of this section is not to demonstrate strong theorems with elaborate proofs but rather show that the framework is convenient for expressing certain properties and helps reasoning on them.

It is easy to prove by induction on the depth that lambda trees, abstract proof trees, and progress trees are all refinements of winning strategies.

**Proposition 2.** *For all order  $d$  and depth  $n$ , we have  $LS_{\alpha_{d,n}} \Rightarrow WS_{\alpha_n}$ ,  $APS_{\alpha_n} \Rightarrow WS_{\alpha_n}$ , and  $PT_{\alpha_n} \Rightarrow WS_{\alpha_n}$ .*

Therefore, whenever we succeed in proving that a position features, say, a lambda tree, then we know it also has a winning strategy for the same player:  $\forall q, q \models LS_{\alpha_{d,n}} \rightarrow q \models WS_{\alpha_n}$ .

On the other hand, in many games, it is possible to have a position featuring a winning strategy but no lambda tree, abstract proof tree, or even progress tree. Before studying the other direction further, we need to rule out games featuring *zugzwangs*, that is, positions in which a player would rather pass and let an opponent make the next move.

**Definition 13.** *A  $\phi$ -zugzwang for player  $A$  against players  $B^1, \dots, B^k$  is a state  $q$  such that  $q \models \neg\phi \wedge (\Box_{B^1} \phi \vee \dots \vee \Box_{B^k} \phi)$ . A game is zugzwang-free for a set of formulas  $\Phi$  and player  $A$  against players  $B^1, \dots, B^k$  if for every state  $q$ , and every formula  $\phi \in \Phi$ ,  $q$  is not a  $\phi$ -zugzwang for  $A$  against  $B^1, \dots, B^k$ .*

The usual understanding of zugzwang is in two player games with  $\phi$  a winning strategy formula or a formula representing forcing some material gain in CHESS.

We can now use this definition to show that in games zugzwang-free for winning strategies, such as HEX or CONNECT-6, an abstract proof tree and a progress tree are equivalent to a winning strategy of the same depth.

**Proposition 3.** *Consider a two-player game zugzwang-free for winning strategies. For any depth  $n$  and any state  $q$ ,  $q \models APS_{\alpha_n} \leftrightarrow q \models PT_{\alpha_n} \leftrightarrow q \models WS_{\alpha_n}$ .*

## 6 Conclusion and discussion

We have defined a general way to express the shape of a search tree using MMLK. We have shown it is possible to use different search strategies to search the tree shape. This combination of a tree shape and of a search strategy yields a variety of search algorithms that can be modelled in the same framework. This makes it easy to combine strategies and shapes to test known algorithms as well to define new ones.

We have shown that the Multi-Agent Modal Logic K was a convenient tool to express various kind of threats in a game independant way. Victor Allis provided one of the earliest study of the concept of threats in his *Threat space search* algorithm used to solve GOMOKU [1].

Previous work by Schaeffer et al. was also concerned with providing a unifying view of heuristic search and the optimizations tricks that appeared in both single-agent search and two-player game search [23].

Another trend of related previous work is connecting modal logic and game theory [29, 32, 15]. In this area, the focus is on the concept of Nash equilibria, extensive form games, and coalition formation. As a result, more powerful logic than the restricted MMLK are used [3, 30, 12]. Studying how the model checking algorithms presented in this article can be extended for these settings is an interesting path for future work.

The model used in this article differs from the one used in General Game Playing (GGP) called Multi-Agent Environment (MAE) [24]. In an MAE, a transition correspond to a joint-action. That is, each player decide a move simultaneously and the combinaison of these moves determines the next state. In a GA, as used in this article, the moves are always sequential. It is possible to simulate sequential moves in an MAE by using *pass* moves for the non acting agents, however this ties the turn player into the game representation. As a result, testing for solution concepts where the player to move in a given position is variable is not possible with an MAE. For instance, it is not possible to formally test for the existence of a ladder in a GGP representation of the game of go because we need to compute the successors of a given position after a white move and alternatively after a black move.

Effective handling of transpositions is another interesting topic for future work. It is already nontrivial in PNS [13] and MCTS [18], but it is an even richer subject in this model checking setting as we might want to prove different facts about a given position in the same search.

Table 7 reveals many interesting previously untested possible combinations of formula classes and search algorithms. Implementing and optimising one specific new combination for a particular game could lead to insightful practical results. For instance, it is quite possible that a Monte Carlo version of Best Reply Search would be successful in MULTIPLAYER GO [7].

## References

- [1] Louis Victor Allis, H. Jaap van den Herik, and M. P. H. Huntjens. Go-Moku solved by new search techniques. *Computational Intelligence*, 12:7–23, 1996.
- [2] Louis Victor Allis, M. van der Meulen, and H. Jaap van den Herik. Proof-Number Search. *Artificial Intelligence*, 66(1):91–124, 1994.
- [3] Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman. Alternating-time temporal logic. *Journal of the ACM*, 49(5):672–713, 2002.
- [4] Peter Auer, Nicolás Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2):235–256, 2002.
- [5] Patrick Blackburn, Maarten De Rijke, and Yde Venema. *Modal Logic*, volume 53. Cambridge University Press, 2001.
- [6] Tristan Cazenave. Abstract Proof Search. In T. Anthony Marsland and Ian Frank, editors, *Computers and Games 2000*, volume 2063 of *Lecture Notes in Computer Science*, pages 39–54. Springer, 2002.
- [7] Tristan Cazenave. Multi-player Go. In H. Jaap van den Herik, Xinhe Xu, Zongmin Ma, and Mark H.M. Winands, editors, *Computers and Games*, volume 5131 of *Lecture Notes in Computer Science*, pages 50–59. Springer Berlin / Heidelberg, 2008.
- [8] Guillaume Chaslot, Sander Bakkes, Istvan Szita, and Pieter Spronck. Monte-carlo tree search: A new framework for game AI. In *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference*, pages 216–217. AAAI Press, 2008.
- [9] Rémi Coulom. Efficient selectivity and back-up operators in monte-carlo tree search. In *Proceedings of the 5th Conference on Computers and Games (CG'2006)*, volume 4630 of *LNCS*, pages 72–83, Torino, Italy, 2006. Springer.
- [10] Sylvain Gelly and David Silver. Achieving master level play in  $9 \times 9$  computer go. In *Proceedings of the 23rd national conference on Artificial Intelligence (AAAI'08)*, pages 1537–1540, 2008.
- [11] Michael Genesereth and Nathaniel Love. General game playing: Overview of the AAAI competition. *AI Magazine*, 26:62–72, 2005.
- [12] Valentin Goranko and Govert van Drimmelen. Complete axiomatization and decidability of alternating-time temporal logic. *Theoretical Computer Science*, 353(1):93–117, 2006.
- [13] Akihiro Kishimoto and Martin Müller. A solution to the GHI problem for depth-first proof-number search. *Information Sciences*, 175(4):296–314, 2005.
- [14] Donald E. Knuth and Ronald W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975.
- [15] Lena Kurzen. *Complexity in Interaction*. PhD thesis, Universiteit van Amsterdam, 2011.
- [16] Martin Müller. Computer go. *Artificial Intelligence*, 134(1-2):145–179, 2002.
- [17] J.A.M. Nijssen and Mark H.M. Winands. Enhancements for multi-player Monte-Carlo Tree Search. In *Proceedings of the 7th international conference on Computers and games, CG'10*, pages 238–249, Berlin / Heidelberg, 2011. Springer-Verlag.
- [18] Abdallah Saffidine, Tristan Cazenave, and Jean Méhat. UCD: Upper Confidence bound for rooted Directed acyclic graphs. In *International Conference on Technologies and Applications of Artificial Intelligence*, pages 467–473, Hsinchu, Taiwan, November 2010. IEEE Computer Society.
- [19] Jahn-Takeshi Saito and Mark H.M. Winands. Paranoid Proof-Number Search. In Georgios N. Yannakakis and Julian Togelius, editors, *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games (CIG-2010)*, pages 203–210, 2010.

- [20] Maarten P.D. Schadd and Mark H.M. Winands. Best reply search for multiplayer games. *IEEE Transactions Computational Intelligence and AI in Games*, 3(1):57–66, 2011.
- [21] Maarten P.D. Schadd, Mark H.M. Winands, H. Jaap van den Herik, Guillaume Chaslot, and Jos W.H.M. Uiterwijk. Single-player monte-carlo tree search. In *Computers and Games*, pages 1–12, 2008.
- [22] Jonathan Schaeffer. A gamut of games. *AI Magazine*, 22(3):29–46, 2001.
- [23] Jonathan Schaeffer, Aske Plaat, and Andreas Junghanns. Unifying single-agent and two-player search. *Information Sciences*, 135(3-4):151–175, July 2001.
- [24] Stephan Schiffel and Michael Thielscher. A multiagent semantics for the game description language. In Joaquim Filipe, Ana Fred, and Bernadette Sharp, editors, *Agents and Artificial Intelligence*, volume 67 of *Communications in Computer and Information Science*, pages 44–55. Springer Berlin Heidelberg, 2010.
- [25] Shunsuke Soeda, Tomoyuki Kaneko, and Tetsuro Tanaka. Dual lambda search and shogi endgames. In H. van den Herik, Shun-Chin Hsu, Tsan sheng Hsu, and H. Donkers, editors, *Advances in Computer Games*, volume 4250 of *Lecture Notes in Computer Science*, pages 126–139. Springer Berlin / Heidelberg, 2006.
- [26] Nathan R. Sturtevant and Richard E. Korf. On pruning techniques for multi-player games. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, AAAI/IAAI 2000, pages 201–207, 2000.
- [27] Thomas Thomsen. Lambda-search in game trees - with application to Go. *ICGA Journal*, 23(4):203–217, 2000.
- [28] H. Jaap van den Herik and Mark Winands. Proof-Number Search and its variants. *Oppositional Concepts in Computational Intelligence*, pages 91–118, 2008.
- [29] Wiebe van der Hoek and Marc Pauly. Modal logic for games and information. *Handbook of modal logic*, 3:1077–1148, 2006.
- [30] Wiebe van der Hoek and Michael Wooldridge. Cooperation, knowledge, and time: Alternating-time temporal epistemic logic and its applications. *Studia Logica*, 75(1):125–157, 2003.
- [31] Mark H. Winands, Yngvi Björnsson, and Jahn-Takeshi Saito. Monte-Carlo tree search solver. In *Proceedings of the 6th international conference on Computers and Games*, CG '08, pages 25–36, Berlin, Heidelberg, 2008. Springer-Verlag.
- [32] Michael Wooldridge, Thomas Agotnes, Paul E. Dunne, and Wiebe van der Hoek. Logic for automated mechanism design — a progress report. In *Proceedings of the National Conference on Artificial Intelligence (AAAI-07)*, volume 22, page 9. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2007.
- [33] Kazuki Yoshizoe, Akihiro Kishimoto, and Martin Müller. Lambda Depth-First Proof Number Search and its application to Go. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI-07)*, pages 2404–2409, 2007.

# An Analysis of Voting Algorithm in Games

Yuichiro Sato, Alessandro Cincotti, and Hiroyuki Iida

Japan Advanced Institute of Science and Technology,  
1-1 Asahidai, Nomi, Ishikawa, Japan  
{sato.yuichiro,cincotti,iida}@jaist.ac.jp

**Abstract.** Multiple choice systems in the domain of games, for example 3-Hirn and consultation algorithm, have been researched and their advantages have been reported. However, little is known about the reason why these systems work well. In this paper, we introduce a mathematical representation of multiple choice systems to determine the necessary and sufficient condition of successful decision making on voting algorithm. Then, we derive a reasonable explanations for 3-Hirn and consultation algorithm in this context.

**Keywords:** 3-Hirn, consultation algorithm, multiple choice system, voting algorithm, decision making

## 1 Introduction

In game engine research, improving performance by multiple choice systems has been researched. In 1985 Althöfer started 3-Hirn with a seminal experiment in the game of chess [1]. 3-Hirn is a system such that “one or more programs compute a clear handful of candidate solutions and a human chooses amongst these candidates” [2]. In chess, the 3-Hirn consists of two different strong chess engines and one human weak chess player. When the human plays a game by choosing a move from moves which the engines suggest as candidates, his/her performance is improved and overcome the each individual engine. This result is surprising because if the weakest player chooses a move from moves which are suggested by the other stronger human players, the outcome is expected to be the opposite.

After Althöfer’s delightful success, Obata et al. reported consultation algorithm where many game engines choose one move by simple majority rule, which improves the performance on Shogi game. This is “a method where a machine chooses a move automatically without human intervention” [3]. They also reported optimistic consultation algorithm [4]. The consultation algorithm adopts consultation between many individual engines. To make many engines, they apply noises on an evaluate function which BONANZA, a strong engine has. Also, they reported consultation of three strong Shogi programs: YSS, GPS, and BONANZA plays better games than any of the three individual programs.

This algorithm works well in other games like chess and Go [5, 6]. In 2010, AKARA, which is a game player uses consultation algorithm defeated a top player in the Ladies Professional Players Group [7].

Even though the advantage of multiple choice systems is clear in practice, the reason why these systems work well is not clear. In 3-Hirn, both engines are stronger than the human who decides which candidate to play. Also, in some consultation algorithm, weak engines which are made from a strong original engine with random numbers suggest candidate moves. In other words, contribution of weak players improves total output. This is a paradox. In this paper, we report a reasonable explanation of this paradox.

In section 2, we introduce a mathematical representation of these systems, especially for consultation with random numbers. In section 3, we introduce an analysis of these systems. This is an extension of the discussion in [3] and a reasonable explanation of the paradox. In section 4, we apply our representation to 3-Hirn and give a reasonable explanation for it. In section 5, we discuss our result. In section 6, we conclude the paper.

## 2 Mathematical representation of general consultation algorithm with majority rule

In this section we introduce a mathematical representation of consultation algorithm. Then, in the next section, we apply this representation to analyze experiments of consultation which are reported in [3]. To analyze consultation, the most important point of view is that an engine is a program which chooses a move in a position in a deterministic way, except if a random algorithm is adopted. Therefore, we can represent an engine as a mapping from positions to moves. To do that, an order for positions and moves is needed to be introduced. This order indexes positions and moves. We do not need to specify this order. The only restriction is that this order needs to be total, *i.e.*, all the positions and moves are needed to be indexed in this order. After indexing, we can represent an engine as a function from indexes of positions to indexes of moves. In this way, we can treat an engine as a function from natural numbers to natural numbers.

Let us denote the set of all positions of a game as  $P$  and the set of all moves as  $M$ . These sets are indexed by natural numbers. Engine decides a move in each position, and this choice is expressed as deciding an index of move for each index of position. This representation of an engine makes our discussion clear and nothing important is missed for analysis. We represent an engine as a function from a set of natural numbers of 1 to  $|P|$  to a set of natural numbers of 1 to  $|M|$  in this paper.

Also, an engine must have an evaluation function  $f$ . An evaluation function is a function which evaluates the advantage of a move in a position. If an engine does not have an evaluation function, it is impossible to decide which move is better. As a result, it has no choice and must return a random move. Random players are not suitable for our purpose. Therefore, we omit systems which do not have an evaluation function.

An evaluation function is a mapping from Cartesian products of positions and moves to real numbers. This mapping also becomes a function if positions

and moves are indexed. Therefore, an evaluation function is a function which is defined on 2-dimensional lattice points. To evaluate each move which is chosen by engine, let us suppose that there exists a perfect player, and denote its evaluation function as  $f^*$ . When  $p$  is a position,  $m$  is a move and  $x$  is a real number, formally

$$f^*(p, m) = x, \quad (1)$$

This  $f^*$  is used to calculate the exact advantage of consultation.  $M$  is the all moves which the game has, therefore  $m$  could be an illegal move in  $p$ . If it is, define the evaluation value as the minimum.

Now, we can start making the mathematical representation of the consultation algorithm. Let us denote  $p \in P = \{1, 2, \dots, |P|\}$  as the position from which the engine needs to play and  $m \in M = \{1, 2, \dots, |M|\}$  as the move which the engine chooses in that position, then the engine is defined as follows.

$$AI(p) = m, \quad (2)$$

If you want an analytical function, use a polynomial function as like  $AI(p) = \sum_{i=1}^{|P|} \lambda_i p^i = m$ . It is possible to choose  $\lambda_i$  to mimic the target engine's decision, because this engine is deterministic. The important point is that this mathematical function returns completely the same move as a real engine which is written as a program. If there are  $n$  engines, let us denote them as  $AI_1, AI_2, \dots, AI_n$ .

To analyze the consultation algorithm, we need to make a matrix as

$$M_{ij}(p) = AI_i(p) - AI_j(p) \quad (3)$$

$M_{ij}(p) = 0$  if and only if  $AI_i(p) - AI_j(p) = 0$ , *i.e.*, the matrix element is 0 if and only if corresponding engines choose the same move. To convert this matrix to an easy to use one, let us use the function  $\delta(x)$  which returns 1 if  $x = 0$  and 0 if  $x \neq 0$ . Then,

$$\begin{aligned} V_{ij}(p) &= \delta(M_{ij}(p)) \\ &= \begin{cases} 1 & (AI_i(p) = AI_j(p)) \\ 0 & (AI_i(p) \neq AI_j(p)) \end{cases} \end{aligned} \quad (4)$$

hence if  $AI_i(p) = AI_j(p)$  then  $M_{ij}(p) = 0$  and if  $AI_i(p) \neq AI_j(p)$  then  $M_{ij}(p) \neq 0$ . Let us call this the voting matrix. Then,  $\sum_{j=1}^n V_{ij}(p)$  is the number of engines who agree with  $AI_i$ . This is greater than or equals to 1, because  $AI_i$  always chooses the same candidate as  $AI_i$ .

In the consultation algorithm, a weight vector  $\mathbf{w}$  is used. This vector represents a priority of each engine. Heavily weighted engines have more priority than lightly weighted ones. For example, in simple majority consultation algorithm, all the elements of weight vector are 1. In consultation algorithm with a leader, the leader is weighted as 1.5 and the others are 1. In this way, voting vector  $\mathbf{v}$  is calculated as follows.

$$\begin{pmatrix} V_{11}(p) & V_{12}(p) & \dots & V_{1n}(p) \\ V_{21}(p) & V_{22}(p) & \dots & V_{2n}(p) \\ \vdots & \vdots & \ddots & \vdots \\ V_{n1}(p) & V_{n2}(p) & \dots & V_{nn}(p) \end{pmatrix} \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{pmatrix} = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix} \quad (5)$$

The index of the max coordinate in  $\mathbf{v}$  represents the accepted engines in the consultation algorithm.

Of course, there is the case where two or more different candidates get the same number of votes. For example, in a 5 engines consultation, the leader is alone, 2 engines choose some candidate and the other 2 engines choose another candidate. In this case, we need to decide which candidate to play. Therefore, a conflict resolution is needed. To do so, resolution function  $r$  is used. Then, a consultation algorithm with majority rule is represented as follows

$$C(p) = AI_{r(\mathbf{v})}(p) \quad (6)$$

where

$$r(\mathbf{v}) = r \left( \begin{pmatrix} \sum_{j=1}^n w_1 \delta(AI_1(p) - AI_j(p)) \\ \sum_{j=1}^n w_2 \delta(AI_2(p) - AI_j(p)) \\ \vdots \\ \sum_{j=1}^n w_n \delta(AI_n(p) - AI_j(p)) \end{pmatrix} \right) \quad (7)$$

One example of  $r$  is random choose from candidates which have a conflict. This  $r$  is used in [3]. Another is to use the evaluation function  $f^s$  of the strongest engine. In this case, the index of engine which is used in consultation algorithm is written as follows

$$Max[f^s(p, AI_i(p))] \quad (8)$$

where  $i \in \{v_i | \forall j. v_j \leq v_i\}$  and  $v_i = \sum_{j=0}^n w_j V_{ij}(p)$ .

Now, we finished the mathematical representation of consultation algorithm with majority rule. We want to analyze the conditions in which the consultation algorithm works well. Therefore, we need a reasonable definition of working well. The following definition is suitable for our purpose.

**Definition 1** *The consultation algorithm absolutely works well if and only if*

$$\forall i. \forall p. f_{AI_i}^*(p) \leq f_C^*(p)$$

where  $f_{AI_i}^*(p) = f^*(p, AI_i(p))$  and  $f_C^*(p) = f^*(p, C(p))$  and  $f^*$  is the evaluation function of the perfect player.

This definition is too strict in practice. What we expect is an average improvement. In other words, what we expect from the consultation algorithm is choosing a better move for most of the positions, but not necessarily for all of the positions. Then, the practical definition becomes.

**Definition 2** *The consultation algorithm works well under a distribution of positions  $D$  if and only if*

$$\forall i. \sum_{p=1}^{|P|} Pr(p) f_{AI_i}^*(p) < \sum_{p=1}^{|P|} Pr(p) f_C^*(p) \Leftrightarrow \forall i. Ave_D f_{AI_i}^*(p) < Ave_D f_C^*(p)$$

where  $Pr(p)$  is the probability of occurrence of position  $p$  under distribution  $D$  and  $Ave_D$  is an average.

If the engine is stochastic, we need an extended definition as follows.

**Definition 3** *The consultation algorithm is expected to work well under distribution of positions  $D$  and distribution of moves  $D'$  if and only if*

$$\begin{aligned} \forall i. \sum_{p=1}^{|P|} Pr(p) \sum_{m=1}^{|M|} \pi(p, m) f_{AI_i}^*(p) &< \sum_{p=1}^{|P|} Pr(p) \sum_{m=1}^{|M|} \Pi(p, m) f_C^*(p) \\ \Leftrightarrow \forall i. Ave_D Ex[f_{AI_i}^*(p)] &< Ave_D Ex[f_C^*(p)] \end{aligned}$$

where  $\pi(p, m)$  and  $\Pi(p, m)$  are the probabilities that  $AI_i$  and  $C$  choose move  $m$  under distribution  $D'$ , and  $Ex[\cdot]$  is an expectation value. If a move  $m$  is illegal in  $p$ ,  $\pi(p, m)$  and  $\Pi(p, m)$  are 0.

Under these definitions, we continue to analyze the consultation algorithm in this paper.

### 3 Mathematical analysis of consultation algorithm with random numbers

In the previous section we formed a mathematical representation of the consultation algorithm and defined when the consultation algorithm works well. Now, it is time to analyze reported experimental results and give a reasonable explanation of the consultation algorithm. We analyze consultation with random numbers as reported in [3].

Obata et al. reported experiments of consultation algorithm with random numbers. In these experiments, they prepared many engines that contribute to consultation which are generated with random numbers in the evaluation function of the original strong Shogi engine, BONANZA. Let us denote the original engine as  $AI_O$  and derived noisy engine as  $AI'$ . Then, the difference of these engines can be represented by noise function  $N_O$ . In other word, a prepared engine is represented as follows

$$AI'(p) = AI_O(p) + N_O(p), \quad (9)$$

$N_O(p)$  is a function which is generated by a fixed list of random numbers, once  $N_O(p)$  is generated, it is fixed. Therefore,  $AI'$  is still deterministic.  $N_O(p)$  depends on  $p$  and the original engine, because some positions or engines could be sensitive for noises, some could be not.

If one makes  $n$  of engines which join consultation, let us denote them as  $AI_i(p) = AI_O(p) + N_O^i(p)$ . Then, a voting matrix becomes as follows

$$V = \delta \left( \begin{pmatrix} N_O^1(p) - N_O^1(p) & N_O^1(p) - N_O^2(p) & \dots & N_O^1(p) - N_O^n(p) \\ N_O^2(p) - N_O^1(p) & N_O^2(p) - N_O^2(p) & \dots & N_O^2(p) - N_O^n(p) \\ \vdots & \vdots & \ddots & \vdots \\ N_O^n(p) - N_O^1(p) & N_O^n(p) - N_O^2(p) & \dots & N_O^n(p) - N_O^n(p) \end{pmatrix} \right) \quad (10)$$

because for each  $AI_i(p)$ ,  $AI_O(p)$  is common. As you see, if the voting matrix does not depend on positions nor types of the original engine, it only depends on random numbers. Therefore, consultation views the engine as playing randomly. However  $N_O$  depends on positions and the original engine. This dependency is important for an improvement of performance.

In the experiments reported in [3], all engines which join in the consultation are weaker than the original engine. In other words,

$$\forall i. Ave_D f_{AI_i}^*(p) < Ave_D f_{AI_O}^*(p) \quad (11)$$

This equation means random noise makes the decision worse. This is reasonable, because random choice is not expected to be better than careful choice. Even though no engine is stronger than the original engine, the majority of them choose better candidates compared to the original one. This seems a paradox. However, there exists a reasonable explanation.

As definition, if consultation algorithm works well, the average of the evaluation values by the perfect player becomes better. In these experiments, consultation worked well. Therefore, the following condition must be satisfied.

$$\forall i. Ave_D f_{AI_i}^*(p) < Ave_D f_C^*(p) \quad (12)$$

Additionally, they reported that the consultation algorithm is stronger than the original engine. Therefore,

$$\forall i. Ave_D f_{AI_i}^*(p) < Ave_D f_{AI_O}^*(p) < Ave_D f_C^*(p) \quad (13)$$

must be satisfied. This equation could be satisfied for some noise function  $N_O$ . However,  $N_O$  are generated randomly. Therefore, whether consultation algorithm works well or not is stochastic in this case. Therefore, we need to treat probability explicitly.

Before discuss about it, we need a partition of  $M$  which is made according to a classification as follows.

$$b(p) = \{m | f^*(p, m) > f_{AI_O}^*(p)\} \quad (14)$$

$$e(p) = \{m | f^*(p, m) = f_{AI_O}^*(p)\} \quad (15)$$

$$w(p) = \{m | f^*(p, m) < f_{AI_O}^*(p)\} \quad (16)$$

In  $b(p)$ , the moves have a better evaluation value, in  $e(p)$  and  $w(p)$ , they do not. This partition depends on  $p$  because better or worse is only relative property. Illegal moves have the minimum as its evaluate value, therefore they are classified in  $w(p)$ . What we expect for the consultation algorithm is that the majority of noisy engines choose a candidate from  $b(p)$ .

At position  $p$ , engines which join in the consultation have three behaviors. One is to choose an equivalent candidate of the original engine. The other is to choose a better or worse candidate compared to the original one. Therefore, any stochastic changes on  $AI_i$  by noise are classified as three types, *i.e.*, going on  $b(p)$  or  $w(p)$ , and staying on  $e(p)$ .

Suppose there exists exact probabilities of  $AI_O$  changing its move from  $AI_O(p)$  to  $m$ . Let us denote this probability as  $\pi(p, m)$ . We assume this probability is common for all  $AI_i$ . Then, from experiment which is reported in [3],

$$Ave_D Ex[f_{AI_i}^*(p)] < Ave_D Ex[f_{AI_O}^*(p)] \quad (17)$$

$$\Leftrightarrow Ave_D \sum_{m=1}^{|M|} \pi(p, m) f^*(p, m) < Ave_D f_{AI_O}^*(p) \quad (18)$$

$M$  is divided into a partition by evaluation as  $b(p)$ ,  $e(p)$  and  $w(p)$ . Therefore, a summation on  $M$  is divided into a summation on  $b(p)$ ,  $e(p)$  and  $w(p)$ . Let us denote such a summation as  $\sum^{b(p)}$ ,  $\sum^{e(p)}$  and  $\sum^{w(p)}$ . Then,

$$\begin{aligned} \sum_{m=1}^{|M|} \pi(p, m) f^*(p, m) &= \sum^{b(p)} \pi(p, m) f^*(p, m) \\ &\quad + \sum^{e(p)} \pi(p, m) f^*(p, m) + \sum^{w(p)} \pi(p, m) f^*(p, m) \\ &= \sum^{b(p)} \pi(p, m) f^*(p, m) + \sum^{e(p)} \pi(p, m) f_{AI_O}^*(p) + \sum^{w(p)} \pi(p, m) f^*(p, m) \end{aligned} \quad (19)$$

because in  $e(p)$ ,  $f^*(p, m) = f_{AI_O}^*(p)$ .

Therefore, equation (18) is fixed as follows.

$$\begin{aligned} Ave_D \sum_{m=1}^{|M|} \pi(p, m) f^*(p, m) &< Ave_D f_{AI_O}^*(p) \\ \Leftrightarrow Ave_D \sum_{m=1}^{|M|} \pi(p, m) f^*(p, m) &< Ave_D \left( \sum^{b(p)} \pi(p, m) f_{AI_O}^*(p) \right. \\ &\quad \left. + \sum^{e(p)} \pi(p, m) f_{AI_O}^*(p) + \sum^{w(p)} \pi(p, m) f_{AI_O}^*(p) \right) \quad (20) \\ \Leftrightarrow Ave_D \sum_{m=1}^{b(p)} \pi(p, m) \{f^*(p, m) - f_{AI_O}^*(p)\} \\ &< Ave_D \sum_{m=1}^{w(p)} \pi(p, m) \{f_{AI_O}^*(p) - f^*(p, m)\} \end{aligned} \quad (21)$$

because  $\sum^{b(p)} \pi(p, m) + \sum^{e(p)} \pi(p, m) + \sum^{w(p)} \pi(p, m) = 1$  and  $Ave_D$  is linear. This equation means an average of an expected improvement on  $b(p)$  is less than an average of an expected reduction in quality and is a reasonable condition of

this experiment. Let us denote the expected improvement of  $AI'$ , i.e., expected improvement by changing  $AI_O$  to  $AI'$  as  $Ex_i^{AI'}(p)$ . Then,

$$Ex_i^{AI'}(p) = \sum^{b(p)} \pi(p, m) \{f^*(p, m) - f_{AI_O}^*(p)\} \quad (22)$$

Also, let us denote the expected reduction of  $AI'$ , i.e., expected reduction by changing  $AI_O$  to  $AI'$  as  $Ex_r^{AI'}(p)$ . Then,

$$Ex_r^{AI'}(p) = \sum^{w(p)} \pi(p, m) \{f_{AI_O}^*(p, m) - f^*(p, m)\} \quad (23)$$

The experimental condition becomes as follows

$$\forall i. Ave_D Ex_i^{AI'}(p) < Ave_D Ex_r^{AI'}(p) \quad (24)$$

**Theorem 1**

$$\forall i. Ave_D Ex[f_{AI_i}^*(p)] < Ave_D Ex[f_{AI_O}^*(p)] \Leftrightarrow \forall i. Ave_D Ex_i^{AI_i}(p) < Ave_D Ex_r^{AI_i}(p)$$

*Proof.* As above.  $\square$

Let denote us the consultation algorithm with random numbers works well if and only if  $\forall i. Ave_D Ex[f_{AI_i}^*(p)] < Ave_D Ex[f_{AI_O}^*(p)] < Ave_D Ex[f_C^*(p)]$ .

**Theorem 2** *The consultation algorithm with random numbers works well if and only if*

$$\begin{cases} \forall i. Ave_D Ex_i^{AI_i}(p) < Ave_D Ex_r^{AI_i}(p) \\ Ave_D Ex_r^C(p) < Ave_D Ex_i^C(p) \end{cases}$$

*Proof.*

$$Ave_D Ex[f_{AI_O}^*(p)] < Ave_D Ex[f_C^*(p)] \quad (25)$$

$$\Leftrightarrow Ave_D f_{AI_O}^*(p) < Ave_D \sum_{m=1}^{|M|} \Pi(p, m) f_C^*(p) \quad (26)$$

$$\begin{aligned} \Leftrightarrow Ave_D f_{AI_O}^*(p) < Ave_D \left( \sum^{b(p)} \Pi(p, m) f^*(p, m) \right. \\ \left. + \sum^{e(p)} \Pi(p, m) f^*(p, m) + \sum^{w(p)} \Pi(p, m) f^*(p, m) \right) \end{aligned} \quad (27)$$

$$\begin{aligned} \Leftrightarrow Ave_D \sum^{w(p)} \Pi(p, m) \{f_{AI_O}^*(p) - f^*(p, m)\} \\ < Ave_D \sum^{b(p)} \Pi(p, m) \{f^*(p, m) - f_{AI_O}^*(p)\} \end{aligned} \quad (28)$$

$$\Leftrightarrow Ave_D Ex_r^C(p) < Ave_D Ex_i^C(p) \quad (29)$$

because a summation on  $M$  is divided into  $\sum^{b(p)}$ ,  $\sum^{e(p)}$  and  $\sum^{w(p)}$  and  $f_C^*(p) = f_{AI_O}^*(p)$  in  $e(p)$ .  $\square$

**Theorem 3** *The lower bound of the consultation algorithm chooses a better move,  $\Pi_b(p, m)$  and the upper bound of the consultation algorithm chooses a worse move,  $\Pi_w(p, m)$  is as follows*

$$\begin{aligned}\Pi_b(p, m) &= \sum_{i=2}^n \binom{n}{i} \pi(p, m)^i - \sum_{i=2}^{\lfloor n/2 \rfloor} \binom{n}{i} \pi(p, m)^i \sum_{j=i}^n \sum^{e(p), w(p)} \binom{n}{j} \pi(p, m')^j \\ \Pi_w(p, m) &= \sum_{i=1}^n \binom{n}{i} \pi(p, m)^i - \sum_{i=1}^{\lfloor n/2 \rfloor - 1} \binom{n}{i} \pi(p, m)^i \sum_{j=i}^n \sum^{e(p), b(p)} \binom{n}{j} \pi(p, m')^j\end{aligned}$$

*Proof.* If consultation algorithm chooses a move, the move needs to be the majority of the candidates. If more than half of engines choose the same candidate, the candidate is chosen as a move in any case. Moreover, if some move is chosen by relatively many engines it is majority. Therefore, the sum of these two is the probability of a move being chosen by consultation.

The probability of a better move  $m \in b(p)$  being chosen by more than half of the engines is

$$\sum_{i=\lfloor n/2 \rfloor + 1}^n \binom{n}{i} \pi(p, m)^i \quad (30)$$

and the probability of relatively many engines choosing the better move is

$$\sum_{i=2}^{\lfloor n/2 \rfloor} \binom{n}{i} \pi(p, m)^i \left( 1 - \sum_{j=i}^n \sum^{e(p), w(p)} \binom{n}{j} \pi(p, m')^j \right) \quad (31)$$

where  $\sum^{e(p), w(p)}$  is a summation on  $e(p)$  and  $w(p)$ . We eliminated the case such that the same number of engines choose a better move and another worse move. In this case, conflict resolution needed. For example, in the experiments in [3], this is done by randomly choosing a move from the candidates. Therefore, in such a case, the better move is not necessarily chosen. This is the reason why we omit this case. This case is counted in the probability that the consultation chooses a worse move.

The summation of these two

$$\begin{aligned}\Pi_b(p, m) &= \sum_{i=\lfloor n/2 \rfloor + 1}^n \binom{n}{i} \pi(p, m)^i \\ &\quad + \sum_{i=2}^{\lfloor n/2 \rfloor} \binom{n}{i} \pi(p, m)^i \left( 1 - \sum_{j=i}^n \sum^{e(p), w(p)} \binom{n}{j} \pi(p, m')^j \right) \\ &= \sum_{i=2}^n \binom{n}{i} \pi(p, m)^i - \sum_{i=2}^{\lfloor n/2 \rfloor} \binom{n}{i} \pi(p, m)^i \sum_{j=i}^n \sum^{e(p), w(p)} \binom{n}{j} \pi(p, m')^j\end{aligned} \quad (32)$$

is the lower bound of  $m \in b(p)$  is chosen by consultation at  $p$ .

In the same context, the upper bound of  $m \in w(p)$  is chosen by consultation at  $p$  is

$$\Pi_w(p, m) = \sum_{i=1}^n \binom{n}{i} \pi(p, m)^i - \sum_{i=1}^{\lfloor n/2 \rfloor - 1} \binom{n}{i} \pi(p, m)^i \sum_{j=i}^n \sum^{e(p), b(p)} \binom{n}{j} \pi(p, m')^j \quad (33)$$

□

These probabilities are useful to calculate the lower bound of expected improvement and the upper bound of expected reduction.  $Ave_D Ex_r^C(p) < Ave_D Ex_i^C(p)$  is satisfiable under  $\forall i. Ave_D Ex_i^{AI_i}(p) < Ave_D Ex_r^{AI_i}(p)$ . There exists such situations and it is easy to find them by numerical trial and error.

## 4 Reasonable explanation for 3-Hirn

3-Hirn is a system which is made of two engines and a human [1, 2]. In this system, the human chooses a better candidate from candidates which are suggested from engines. The human is weaker than each engine in the game, but this system is stronger than each engine. The weak player's contribution improves the final outcome. It looks unreasonable that this system becomes stronger than any individual engines. However, there is a reasonable explanation of this puzzle in almost the same context as consultation algorithm, as follows.

Let us denote two engines in this system as  $AI_1$  and  $AI_2$  and the evaluation function of the human as  $f^h$ . Then the 3-Hirn system is written as follows

$$H_3(p) = \begin{cases} AI_1(p) & (f_{AI_1}^h(p) - f_{AI_2}^h(p) > 0) \\ AI_2(p) & (f_{AI_1}^h(p) - f_{AI_2}^h(p) \leq 0) \end{cases} \quad (34)$$

If 3-Hirn works well, the following condition is satisfied

$$\forall i. Ave_D f_{AI_i}^*(p) < Ave_D f_{H_3}^*(p) \quad (35)$$

where  $i$  is 1 or 2.

This condition is satisfiable. An engine has an evaluation function to improve its search speed. This function tends to be rough compared to the one a human has. Even though the engine has a rough evaluation function, the computer is really powerful, so it is able to cover the weakness by making a vast search tree. For human, the situation is opposite. A strong human player has a really good evaluation function and cuts the search tree efficiently.

Mathematically, this situation is represented as follows

$$Ave_D |f^h(p, m) - f^*(p, m)| < Ave_D |f^{AI_i}(p, m) - f^*(p, m)| \quad (36)$$

for almost all of  $m \in M$ .

This situation is satisfiable. An evaluation function is a function from Cartesian product of positions and moves to real numbers. This means it only depends

on the current position, not on search depth. However the strength of a player depends on an evaluation function and search depth. Therefore, if the human is good at evaluation but not for search, and engines are oppositely good at search but not for evaluation, the situation could arise.

If the above equations are satisfied, the human can choose the best candidate at a glance from candidates which engines found by wide and deep search effort. In other word, this system works well “By combining the gifts and strengths of humans and machines in appropriate ways” as Althöfer mentioned in [1]. This is the reason why 3-Hirn works well.

## 5 Discussion

Now, we have an analysis and reasonable explanation for consultation algorithm with random numbers and 3-Hirn. There exists many situations in which consultation works well. However, it is difficult to derive a property which is common for all cases. Everything depends on the game and the engine. Therefore, consultation algorithm with random numbers is not a general solution. The domain which is available to consultation is limited. A clear explanation of 3-Hirn is obtained in the same context.

The consultation algorithm with random numbers is formally explained by our approach, but we have another type of consultation, *i.e.*, optimistic consultation. Unfortunately, it is impossible to find the origin of an advantage of optimistic consultation in this approach. To analyze optimistic consultation algorithm, we need to analyze detailed structures of an evaluation function. This is difficult.

Also, consultation of different types of engines are reported [3]. This is a slightly different situation from our analyses. It is possible to apply our analyses for this case. This consultation makes a large improvement on the original engine. No longer using a noisy weak engine but different type of engine. Therefore, it could be possible to approximate this consultation using our analyses.

We conclude that the effectiveness of the consultation algorithm depends on the game. In our representation, a game has positions and moves, *i.e.*, current states and decisions to make. Therefore, some human activity is included in our analyses. For example, the case in which one human considers his idea from many points of view has an analogy with our analyses. To get different point of view, one needs to come up with options which is not the first choice. This means generating worse options under his evaluation. If his evaluation is close enough to perfect, this is the same as adding random noise to his thinking. Consultation with many humans could also be approximated by our approach. Solomon describes a result of social epistemology as “If group deliberation does take place, outcomes are better when members of the group are strangers, rather than colleagues or friends.” [8]. By taking a group of friends, and adding randomness to the decision process, a group of strangers is formed. This is a case of human activity to which our approach is applicable.

## 6 Conclusions

In this paper, we clearly explained the origin of an advantage of consultation algorithm with random numbers, and 3-Hirn. The consultation algorithm with random numbers works well if and only if the expected improvement of consultation is greater than the expected reduction of consultation and the expected improvement of each engine is less than the expected reduction of each engine. In this paper a new definition is derived of the necessary and sufficient condition for consultation algorithm working well. This new definition is an improvement of the existing one because it considers all choices. A explanation of 3-Hirn is given, elaborating on the existing one. Formulas for bounds of probability of consultation algorithm improving/reducing are given. Our analysis has a possibility to be applicable to human activities not only engine.

## References

1. Althöfer I., and Snatzke, G.R.: Playing Games with Multiple Choice Systems. Lecture Notes in Computer Science 2883, pp. 142-153 (2003)
2. Althöfer I.: Improved game play by multiple computer hints. Theoretical Computer Science 313, 315-324 (2004)
3. Obata, T., Sugiyama, T., Hoki, K., and Ito, T.: Consultation Algorithm for Computer Shogi Move Decisions by Majority. Lecture Notes in Computer Science 6515, pp. 156-165 (2011)
4. Sugiyama, T., Obata, T., Hoki, K., and Ito, T.: Optimistic Selection Rule Better Than Majority Voting System. Lecture Notes in Computer Science 6515, pp. 166-175 (2011)
5. Omori, S., Hoki, K., and Ito, T.: Consultation Algorithm for Computer Chess. SIG Technical Reports 2011-GI-26(5), 1-7 (2011)
6. Manabe, K., and Muramatsu, M.: Boosting Approach for Consultaton by Weighted Majority Vote in Computer Go. IPSJ Symposium Series 2011 6, 128-134 (2011)
7. 清水市代女流王将 vs. あから 2010 速報,  
<http://www.ipsj.or.jp/50anv/shogi/20101012.html> (News:a top player in the Ladies Professional Players Group vs. AKARA 2010)
8. Solomon, M.: Groupthink versus The Wisdom of Crowds: The Social Epistemology of Deliberation and Dissent. The Southern Journal of Philosophy 44, 28-42 (2006)

# A Bayesian Tactician

Gabriel Synnaeve (gabriel.synnaeve@gmail.com) and Pierre Bessière (pierre.bessiere@imag.fr)

Université de Grenoble (LIG), INRIA, CNRS, Collège de France (LPPA)

**Abstract.** We describe a generative Bayesian model of tactical attacks in strategy games, which can be used both to predict attacks and to take tactical decisions. This model is designed to easily integrate and merge information from other (probabilistic) estimations and heuristics. In particular, it handles uncertainty in enemy units’ positions as well as their probable tech tree. We claim that learning, being it supervised or through reinforcement, adapts to skewed data sources. We evaluated our approach on StarCraft<sup>1</sup>: the parameters are learned on a new (freely available) dataset of game states, deterministically re-created from replays, and the whole model is evaluated for prediction in realistic conditions. It is also the tactical decision-making component of a competitive StarCraft AI.

## 1 Introduction

### 1.1 Game AI

We believe video game AI is central to new, fun, re-playable gameplays, being them multi-player or not. Cooperative (player versus AI) games are enjoying a new boom, recent RTS games delegate more micro-management to the AI as ever, and ever more realistic first-person shooters (FPS) immersion hardly cope with scripted (unsurprising) non-playing characters (NPC). In their study on human like characteristics in RTS games, Hagelbäck and Johansson [1] found out that “tactics was one of the most successful indicators of whether the player was human or not”. No current non-cheating AI consistently beats good human players in RTS (aim cheating is harder to define for FPS games), nor are fun to play many games against. Finally, as the world is simulated but the players are not, multi-player game AI research is in between real-world robotics and more theoretical AI, and so can benefit both fields.

### 1.2 RTS Gameplay

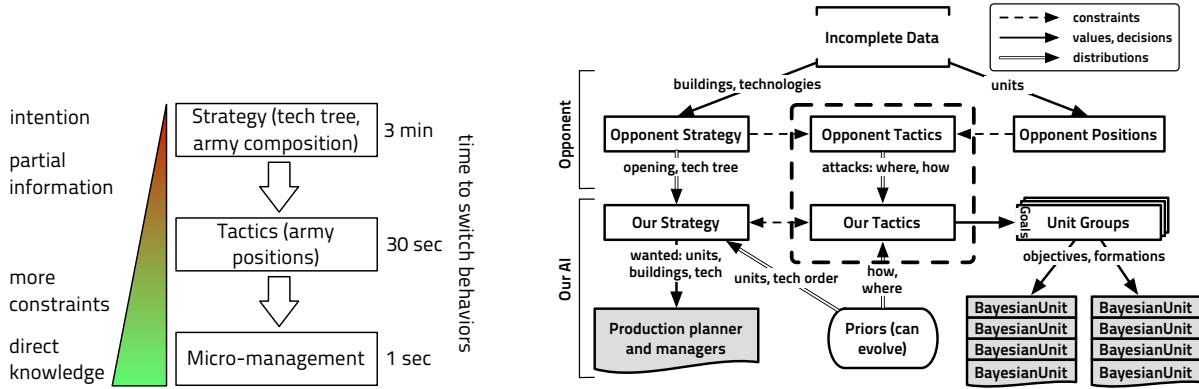
Real-time strategy (RTS) gameplay consist in producing and managing group of units with attacks and movements specificities in order to defeat an enemy. Most often, it is required to gather resources and build up an economic and military power while expanding a technology tree. Parts of the map not in the sight range of the player’s units are under *fog of war*, so the player only has partial information about the enemy buildings and army. The way by which we expand the tech tree, the specific units composing the army, and the general stance (aggressive or defensive) form what we call *strategy*. At the lower level, the actions performed by the player (human or not) to optimize the effectiveness of its units is called *micro-management*. In between lies *tactics*: where to attack, and how. A good human player takes much data in consideration when choosing: are there flaws in the defense? Which spot is more worthy to attack? How much am I vulnerable for attacking here? Is the terrain (height, chokes) to my advantage? etc.

In this paper, we focus on tactics, in between strategy (high-level) and micro-management (lower-level), as seen in Fig. 1. We propose a model which can either predict enemy attacks or give us a distribution on where and how to attack the opponent. Information from the higher-level strategy constrains what types of attacks are possible. As shown in Fig. 1, information from units’ positions (or possibly an enemy units particle filter as in [2]) constrains where the armies can possibly be in the future. In the context of our StarCraft AI (“bot”), once we have a decision: we generate a goal (attack order) passed to units groups (see Fig.1). A Bayesian model for micro-management [3], in which units are attracted or repulsed by dynamic (goal, units, damages) and static (terrain) influence maps, actually moves the units in StarCraft. Other previous works on strategy prediction [4, 5] allows us to infer the enemy tech tree and strategies from incomplete information (due to the fog of war).

### 1.3 StarCraft Tactics

We worked on StarCraft: Brood War, which is a canonical RTS game. It had been around since 1998, sold 9.5 million licenses and reigned on competitive RTS for more than a decade. StarCraft (like most RTS) has a mechanism, *replays*, to record every

<sup>1</sup> StarCraft and its expansion StarCraft: Brood War are trademarks of Blizzard Entertainment<sup>TM</sup>



**Fig. 1.** Left: Gameplay levels of abstraction for RTS games, compared with their level of direct (and complete) information and orders of magnitudes of time to change their policies. Right: Information centric view of the StarCraft bot player, the part presented in this paper is inside dotted lines (tactics). Dotted arrows represent constraints on what is possible, plain simple arrows represent simple (real) values, either from data or decisions, and double arrows represent probability distributions on possible values. The grayed surfaces are the components actuators (passing orders to the game).

player's actions such that the state of the game can be deterministically re-simulated. Numerous international competitions and professional gaming (mainly in South Korea) produced a massive amount of data of highly skilled human players, performing about 300 actions per minute while following and adapting their strategies. In StarCraft, there are two types of resources, often located close together, minerals (at the base of everything) and gas (at the base of advanced units and technologies). There are 3 factions (Protoss, Terran and Zerg) which have workers to gather resources, and all other characteristics are different: from military units to "tech trees", gameplay styles.

Units have different abilities, which leads to different possible tactics. Each faction has invisible (temporarily or permanently) units, flying transport units, flying attack units and ground units. Some units can only attack ground or air units, some others have splash damage attacks, immobilizing or illusion abilities. Fast and mobile units are not cost-effective in head-to-head fights against slower bulky units. We used the gamers' vocabulary to qualify different types of tactics:

- *ground* attacks (raids or pushes) are the most normal kind of attacks, carried by basic units which cannot fly,
- *air* attacks (air raids), which use flying units' mobility to quickly deal damage to undefended spots.
- *invisible* attacks exploit the weaknesses (being them positional or technological) in detectors of the enemy to deal damage without retaliation,
- *drops* are attacks using ground units transported by air, combining flying units' mobility with cost-effectiveness of ground units, at the expense of vulnerability during transit.

This will be the only four types of tactics that we will use in this paper: *how* did the player attack or defend?

RTS games maps, StarCraft included, consist in a closed arena in which units can evolve. It is filled with terrain features like uncrossable terrain for ground units (water, space), cliffs, ramps, walls. Particularly, each RTS game which allows production also give some economical (gathering) mechanism and so there are some resources scattered on the map, where players need to go collect. It is way more efficient to build expansion (auxiliary bases) to collect resources directly on site. So when a player decides to attack, she has to decide *where* to attack, and this decision takes into account *how* it can attack different places, due to their geographical remoteness, topological access possibilities and defense strength. Choosing *where* to attack is a complex decision to make: of course it is always wanted to attack poorly defended economic expansions of the opponent, but the player has to consider if it places its own bases in jeopardy, or if it may trap her own army. With a perfect estimator of battles outcomes (which is a hard problem due to terrain, army composition combinatorics and units control complexity), and perfect information, this would result in a game tree problem which could be solved by  $\alpha - \beta$ . Unfortunately, StarCraft is a partial observation game with complex terrain and fight mechanics.

## 2 Background

### 2.1 Related Work

Aha et al. [6] used case-based reasoning (CBR) to perform dynamic tactical plan retrieval (matching) extracted from domain knowledge in Wargus. Ontaño et al. [7] based their real-time case-based planning (CBP) system on a plan dependency graph

which is learned from human demonstration in Wargus. A case based behavior generator spawn missing goals which are missing from the current state and plan according to the recognized state. In [8, 9], they used a knowledge-based approach to perform situation assessment to use the right plan, performing runtime adaptation by monitoring its performance. Sharma et al. [10] used richly parametrized CBR for strategic and tactical AI in Spring (Total Annihilation open source clone). [11] combined CBR and reinforcement learning to enable reuse of tactical plan components. Cadena and Garrido [12] used fuzzy CBR (fuzzy case matching) for strategic and tactical planning. Chung et al. [13] applied Monte-Carlo planning to a capture-the-flag mod of Open RTS. Inspired by successes of MCTS with upper confidence bounds on trees (UCT) policies [14] on the game of Go, Balla and Fern [15] applied UCT to tactical assault planning in Wargus.

In Starcraft, Weber et al. [16, 17] produced tactical goals through reactive planning and goal-driven autonomy, finding the more relevant goal(s) to follow in unforeseen situations. Kabanja et al. [18] performed plan and intent recognition to find tactical opportunities. On spatial and temporal reasoning, Forbus et al. [19] presented a tactical qualitative description of terrain for wargames through geometric and pathfinding analysis. Perkins [20] automatically extracted choke points and regions of StarCraft maps from a pruned Voronoi diagram, which we used for our regions representations. Wintermute et al. [21] used a cognitive approach mimicking human attention for tactics and units control. Ponsen et al. [22] developed an evolutionary state-based tactics generator for Wargus. Finally, Avery et al. [23] and Smith et al. [24] co-evolved influence map trees for spatial (tactical) reasoning in RTS games.

Our approach (and bot architecture, depicted in Fig. 1) can be seen as goal-driven autonomy [16] dealing with multi-level reasoning by passing distributions (without any assumption about how they were obtained) on the module input. Using distributions as messages between specialized modules makes dealing with uncertainty first class, this way a given model do not care if the uncertainty comes from incompleteness in the data, a complex and biased heuristic, or another probabilistic model. We then take a decision by sampling or taking the most probable value in the output distribution. Another particularity of our model is that it allows for prediction of the enemy tactics using the same model with different inputs. Finally, our approach is not exclusive to most of the techniques presented above, and it could be interesting to combine it with UCT [15] and more complex/precise tactics generated through planning.

## 2.2 Bayesian Programming

Probability is used as an alternative to classical logic and we transform incompleteness (in the experiences, observations or the model) into uncertainty [25]. We introduce Bayesian programs (BP), a formalism that can be used to describe entirely any kind of Bayesian model, subsuming Bayesian networks and Bayesian maps, equivalent to probabilistic factor graphs [26]. There are mainly two parts in a BP, the **description** of how to compute the joint distribution, and the **question(s)** that it will be asked.

The description consists in explaining the relevant *variables*  $\{X^1, \dots, X^n\}$  and explain their dependencies by *decomposing* the joint distribution  $P(X^1 \dots X^n | \delta, \pi)$  with existing preliminary knowledge  $\pi$  and data  $\delta$ . The *forms* of each term of the product specify how to compute their distributions: either parametric forms (laws or probability tables, with free parameters that can be learned from data  $\delta$ ) or recursive questions to other Bayesian programs.

Answering a question is computing the distribution  $P(\text{Searched} | \text{Known})$ , with *Searched* and *Known* two disjoint subsets of the variables.

$$P(\text{Searched} | \text{Known}) = \frac{\sum_{\text{Free}} P(\text{Searched}, \text{Free}, \text{Known})}{P(\text{Known})}$$

$$= \frac{1}{Z} \times \sum_{\text{Free}} P(\text{Searched}, \text{Free}, \text{Known})$$

$$BP \left\{ \begin{array}{l} \text{Desc.} \left\{ \begin{array}{l} \text{Spec.}(\pi) \left\{ \begin{array}{l} \text{Variables} \\ \text{Decomposition} \\ \text{Forms (Parametric or Program)} \end{array} \right. \\ \text{Identification (based on } \delta) \end{array} \right. \\ \text{Question} \end{array} \right.$$

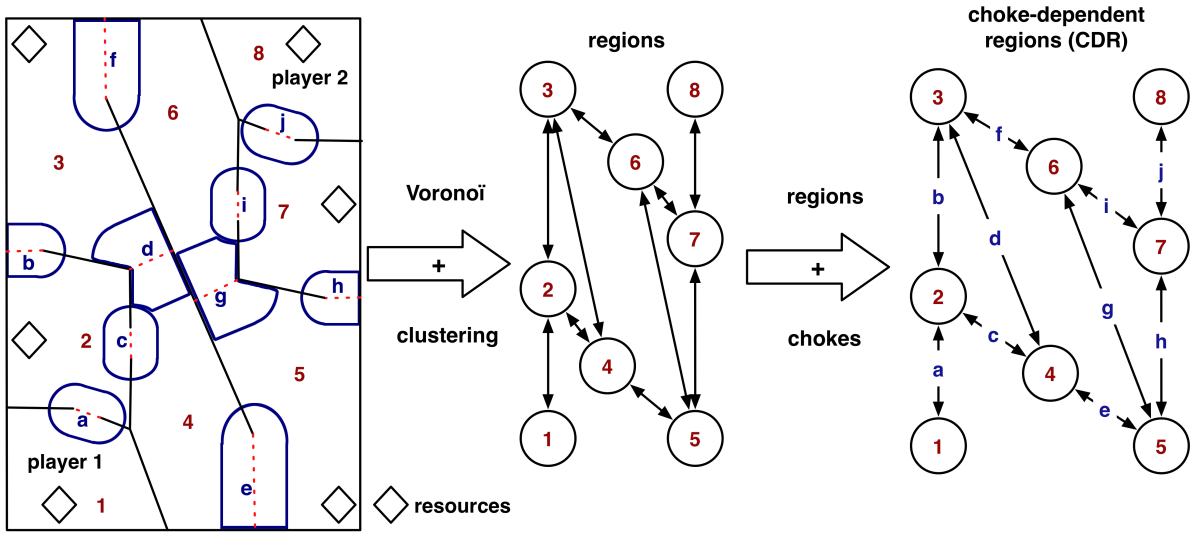
Bayesian programming originated in robotics [27] and evolved to all sensory-motor systems [28]. For its use in cognitive modeling, see [29] and for its first use in video games (FPS, Unreal Tournament), see [30]; for MMORPG, see [31].

### 3 Methodology

#### 3.1 Dataset

We downloaded more than 8000 replays to keep 7649 uncorrupted, 1v1 replays of very high level StarCraft games (progamers leagues and international tournaments) from specialized websites<sup>2,3,4</sup>, we then ran them using BWAPI<sup>5</sup> and dumped units positions, pathfinding and regions, resources, orders, vision events, for attacks (we trigger an attack tracking heuristic when one unit dies and there are at least two military units around): types, positions, outcomes. Basically, every BWAPI event was recorded, the dataset and its source code are freely available<sup>6</sup>.

#### 3.2 Spatial Reasoning



**Fig. 2.** A very simple map on the left, which is transformed into regions (between chokes in dotted red lines) by Voronoi tessellation and clustering. These plain regions (numbers in red) are then augmented with choke-dependent regions (letters in blue)

We used two kinds of regions: BroodWar Terrain Analyser (BWTa) regions and choke-dependent (choke-centered) regions. BWTa regions are obtained from a pruned Voronoi diagram on walkable terrain [20] and give regions for which chokes are the boundaries. As battles often happens at chokes, choke-dependent regions are created by doing an additional (distance limited) Voronoi tessellation spawned at chokes, its regions set is  $(regions \setminus chokes) \cup chokes$ . Figure 2 illustrate regions and choke-dependent regions (CDR). Results for choke-dependent regions are not fully detailed.

#### 3.3 Value Heuristics

The idea is to have (most probably biased) lower-level heuristics from units observations which produce information exploitable at the tactical level, and take some advantage of strategic inference too. The advantages are that 1) learning will de-skew the model output from biased heuristic inputs 2) the model is agnostic to where input variables' values come from 3) the updating process is the same for supervised learning and for reinforcement learning.

We note  $s_{unit\ type}^{a|d}(r)$  for the balanced score of units from attacker or defender ( $a|d$ ) of a given type in region  $r$ . The balanced score of units is just the sum of units multiplied by each unit score ( $= minerals\_value + \frac{4}{3}gas\_value + 50supply\_value$ ).

<sup>2</sup> <http://www.teamliquid.net>

<sup>3</sup> <http://www.gosugamers.net>

<sup>4</sup> <http://www.iccup.com>

<sup>5</sup> <http://code.google.com/p/bwapi/>

<sup>6</sup> <http://snippyhollow.github.com/bwreplaydump/>

The heuristics we used in our benchmarks (which we could change) are:

$$economical\_score^d(r) = \frac{s_{workers}^d(r)}{\sum_{i \in regions} s_{workers}^d(i)}$$

$$tactical\_score^d(r) = \sum_{i \in regions} s_{army}^d(i) \times dist(i, r)^{-1.5}$$

We used “ $-1.5$ ” such that the tactical value of a region in between two halves of an army, each at distance 2, would be higher than the tactical value of a region at distance 4 of the full (same) army. For flying units,  $dist$  is the Euclidean distance, while for ground units it takes pathfinding into account.

$$ground\_defense^d(r) = \frac{s_{can\_attack\_ground}^d(r)}{s_{ground\_units}^a(r)}$$

$$air\_defense^d(r) = \frac{s_{can\_attack\_air}^d(r)}{s_{air\_units}^a(r)}$$

$$invis\_defense^d(r) = number_{detectors}^d$$

### 3.4 Tactical Model

We preferred to discretize continuous values to enable quick complete computations. Another strategy would keep more values and use Monte Carlo sampling for computation. We think that discretization is not a concern because 1) heuristics are simple and biased already 2) we often reason about imperfect information and this uncertainty tops discretization fittings.

**Variables** With  $n$  regions, we have:

- $A_{1:n} \in \{true, false\}$ ,  $A_i$ : attack in region  $i$  or not?
- $E_{1:n} \in \{no, low, high\}$ ,  $E_i$  is the discretized economical value of the region  $i$  for the defender. We choose 3 values: *no* workers in the regions, *low*: a small amount of workers (less than half the total) and *high*: more than half the total of workers in this region  $i$ .
- $T_{1:n} \in discrete\ levels$ ,  $T_i$  is the tactical value of the region  $i$  for the defender, see above for an explanation of the heuristic. Basically,  $T$  is proportional to the proximity to the defender’s army. In benchmarks, discretization steps are 0, 0.05, 0.1, 0.2, 0.4, 0.8 ( $\log_2$  scale).
- $TA_{1:n} \in discrete\ levels$ ,  $TA_i$  is the tactical value of the region  $i$  for the attacker (see above).
- $B_{1:n} \in \{true, false\}$ ,  $B_i$  tells if the region belongs (or not) to the defender.  $P(B_i = true) = 1$  if the defender has a base in region  $i$  and  $P(B_i = false) = 1$  if the attacker has one. Influence zones of the defender can be measured (with uncertainty) by  $P(B_i = true) \geq 0.5$  and vice versa.
- $H_{1:n} \in \{ground, air, invisible, drop\}$ ,  $H_i$ : in predictive mode: how we will be attacked, in decision-making: how to attack, in region  $i$ .
- $GD_{1:n} \in \{no, low, med, high\}$ : ground defense (relative to the attacker power) in region  $i$ , result from a heuristic. *no* defense if the defender’s army is  $\geq 1/10th$  of the attacker’s, *low* defense above that and under half the attacker’s army, *medium* defense above that and under comparable sizes, *high* if the defender’s army is bigger than the attacker.
- $AD_{1:n} \in \{no, low, med, high\}$ : same for air defense.
- $ID_{1:n} \in \{no\ detector, one\ detector, several\}$ : invisible defense, equating to numbers of detectors.
- $TT \in [\emptyset, building_1, building_2, building_1 \wedge building_2, techtrees, \dots]$ : all the possible technological trees for the given race. For instance  $\{pylon, gate\}$  and  $\{pylon, gate, core\}$  are two different Tech Trees.
- $HP \in \{ground, ground \wedge air, ground \wedge invis, ground \wedge air \wedge invis, ground \wedge drop, ground \wedge air \wedge drop, ground \wedge invis \wedge drop, ground \wedge air \wedge invis \wedge drop\}$ : **how possible** types of attacks, directly mapped from  $TT$  information. In prediction, with this variable, we make use of what we can infer on the opponent’s strategy [5, 4], in decision-making, we know our own possibilities (we know our tech tree as well as the units we own).

Finally, for some variables, we take uncertainty into account with “soft evidences”: for instance for a region in which no player has a base, we have a soft evidence that it belongs more probably to the player established closer. In this case, for a given region, we introduce the soft evidence variable(s)  $B'$  and the coherence variable  $\lambda_B$  and impose  $P(\lambda_B = 1|B, B') \text{ iff } B = B'$ , while  $P(\lambda_B|B, B')P(B')$  is a new factor in the joint distribution. This allows to sum over  $P(B')$  distribution (soft evidence).

**Decomposition** The joint distribution of our model contains soft evidence variables for all input family variables ( $E, T, TA, B, GD, AD, ID, HP$ ) to be as general as possible, *i.e.* to be able to cope with all possible uncertainty (from incomplete information) that may come up in a game. To avoid being too verbose, we explain the decomposition only with the soft evidence for the family of variables  $B$ , the principle holds for all other soft evidences. For the  $n$  considered regions, we have:

$$\begin{aligned}
& P(A_{1:n}, E_{1:n}, T_{1:n}, TA_{1:n}, B_{1:n}, B'_{1:n}, \lambda_{B,1:n}, \\
& H_{1:n}, GD_{1:n}, AD_{1:n}, ID_{1:n}, HP, TT) \\
&= \prod_{i=1}^n [P(A_i)P(E_i, T_i, TA_i, B_i|A_i) \cdot \quad (1) \\
& P(\lambda_{B,i}|B_{1:n}, B'_{1:n})P(B'_{1:n}) \\
& P(AD_i, GD_i, ID_i|H_i)P(H_i|HP)]P(HP|TT)P(TT)
\end{aligned}$$

**Forms and Learning** We will explain the forms for a given/fixed  $i$  region number:

- $P(A)$  is the prior on the fact that the player attacks in this region, in our evaluation we set it to  $n_{battles}/(n_{battles} + n_{not\ battles})$ . In a given match it should be initialized to uniform and progressively learn the preferred attack regions of the opponent for predictions, learn the regions in which our attacks fail or succeed for decision-making.
- $P(E, T, TA, B|A)$  is a covariance table of the economical, tactical (both for the defender and the attacker), belonging scores where an attacks happen. We just use Laplace succession law (“add one” smoothing) [25] and count the co-occurrences, thus almost performing maximum likelihood learning of the table.
- $P(\lambda_B|B, B') = 1.0$  *iff*  $B = B'$  is just a coherence constraint.
- $P(AD, GD, ID|H)$  is a covariance table of the air, ground, invisible defense values depending on how the attack happens. As for  $P(E, T, TA, B|A)$ , we use a Laplace’s law of succession to learn it.
- $P(H|HP)$  is the distribution on how the attack happens depending on what is possible. Trivially  $P(H = ground|HP = ground) = 1.0$ , for more complex possibilities we have different maximum likelihood multinomial distributions on  $H$  values depending on  $HP$ .
- $P(HP|TT)$  is the direct mapping of what the tech tree allows as possible attack types:  $P(HP = hp|TT) = 1$  is a function of  $TT$  (all  $P(HP \neq hp|TT) = 0$ ).
- $P(TT)$ : if we are sure of the tech tree (prediction without fog of war, or in decision-making mode),  $P(TT = k) = 1$  and  $P(TT \neq k) = 0$ ; otherwise, it allows us to take uncertainty about the opponent’s tech tree and balance  $P(HP|TT)$ . We obtain a distribution on what is possible ( $P(HP)$ ) for the opponent’s attack types.

There are two approaches to fill up these probability tables, either by observing games (supervised learning), as we did in the evaluation section, or by acting (reinforcement learning). In match situation against a given opponent, for inputs that we can unequivocally attribute to their intention (style and general strategy), we also refine these probability tables (with Laplace’s rule of succession). To keep things simple, we just refine  $\sum_{E,T,TA} P(E, T, TA, B|A)$  corresponding to their aggressiveness (aggro) or our successes and failures, and equivalently for  $P(H|HP)$ . Indeed, if we sum over  $E, T$  and  $TA$ , we consider the inclination of our opponent to venture into enemy territory or the interest that we have to do so by counting our successes with aggressive or defensive parameters. In  $P(H|HP)$ , we are learning the opponent’s inclination for particular types of tactics according to what is available to their, or for us the effectiveness of our attack types choices.

The model is highly modular, and some parts are more important than others. We can separate three main parts:  $P(E, T, TA, B|A)$ ,  $P(AD, GD, ID|H)$  and  $P(H|HP)$ . In prediction,  $P(E, T, TA, B|A)$  uses the inferred (uncertain) economic ( $E$ ), tactical ( $T$ ) and belonging ( $B$ ) scores of the opponent while knowing our own tactical position fully ( $TA$ ). In decision-making, we know  $E, T, B$  (for us) and estimate  $TA$ . In our prediction benchmarks,  $P(AD, GD, ID|H)$  has the lesser impact on the results of the three main parts, either because the uncertainty from the attacker on  $AD, GD, ID$  is too high or because our heuristics are too simple, though it still contributes positively to the score. In decision-making, it allows for reinforcement learning to have pivoting tuple values for  $AD, GD, ID$  at which to switch attack types. In prediction,  $P(H|HP)$  is used to take  $P(TT)$  (coming from strategy prediction [4]) into account and constraints  $H$  to what is possible. For the use of  $P(H|HP)P(HP|TT)P(TT)$  in decision-making, see the Results sections.

**Questions** For a given region  $i$ , we can ask the probability to attack here,

$$P(A_i = a_i|e_i, t_i, ta_i, \lambda_{B,i} = 1)$$

$$\begin{aligned}
&= \frac{\sum_{B_i, B'_i} P(e_i, t_i, ta_i, B_i | a_i) P(a_i) P(B'_i) \cdot P(\lambda_{B,i} | B_i, B'_i)}{\sum_{A_i, B_i, B'_i} P(e_i, t_i, ta_i, B_i | A_i) P(A_i) P(B'_i) P(\lambda_{B,i} | B_i, B'_i)} \\
&\propto \sum_{B_i, B'_i} P(e_i, t_i, ta_i, B_i | a_i) P(a_i) P(B'_i) P(\lambda_{B,i} | B_i, B'_i)
\end{aligned}$$

and the mean by which we should attack,

$$\begin{aligned}
&P(H_i = h_i | ad_i, gd_i, id_i) \\
&\propto \sum_{TT, P} [P(ad_i, gd_i, id_i | h_i) P(h_i | HP) P(HP | TT) P(TT)]
\end{aligned}$$

For clarity, we omitted some variables couples on which we have to sum (to take uncertainty into account) as for  $B$  (and  $B'$ ) above. We always sum over estimated, inferred variables, while we know the one we observe fully. In prediction mode, we sum over  $TA, B, TT, P$ ; in decision-making, we sum over  $E, T, B, AD, GD, ID$ . The complete question that we ask our model is  $P(A, H | FullyObserved)$ . The maximum of  $P(A, H)$  may not be the same as the maximum of  $P(A)$  or  $P(H)$ , for instance think of a very important economic zone that is very well defended, it may be the maximum of  $P(A)$ , but not once we take  $P(H)$  into account. Inversely, some regions are not defended against anything at all but present little or no interest. Our joint distribution (1) can be rewritten:  $P(Searched, FullyObserved, Estimated)$ , so we ask:

$$\begin{aligned}
&P(A_{1:n}, H_{1:n} | FullyObserved) \quad (2) \\
&\propto \sum_{Estimated} P(A_{1:n}, H_{1:n}, Estimated, FullyObserved)
\end{aligned}$$

## 4 Results

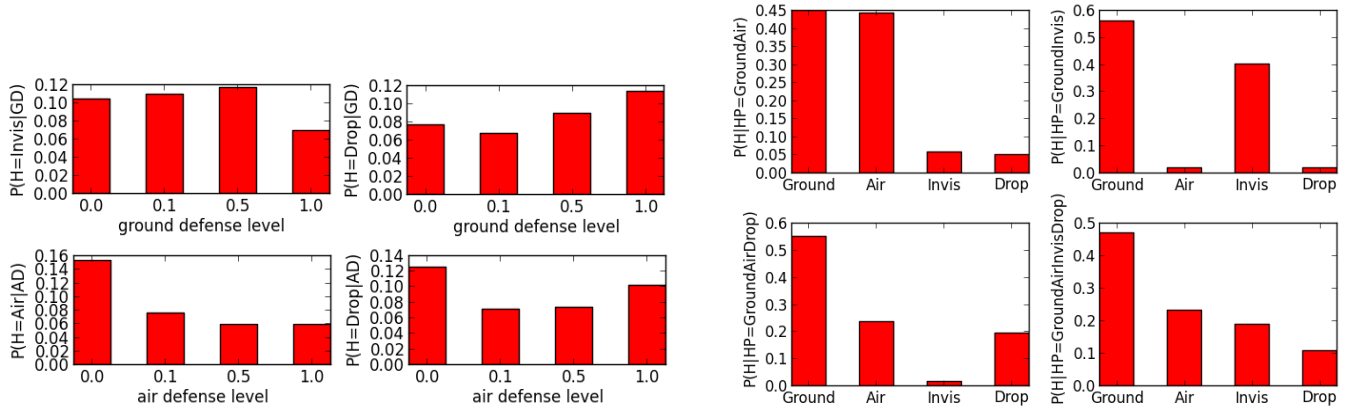
### 4.1 Learning

To measure fairly the prediction performance of such a model, we applied “leave-100-out” cross-validation from our dataset: as we had many games (see Table. 1), we set aside 100 games of each match-up for testing (with more than 1 battle per match: rather  $\approx \llbracket 11 \dots 35 \rrbracket$  battles/match) and train our model on the rest. We write match-ups  $XvY$  with  $X$  and  $Y$  the first letters of the factions involved (Protoss, Terran, Zerg). Note that mirror match-ups (PvP, TvT, ZvZ) have fewer games but twice as many attacks from a given faction. Learning was performed as explained in III.B.3: for each battle in  $r$  we had one observation for:  $P(e_r, t_r, ta_r, b_r | A = true)$ , and  $\#regions - 1$  observations for the  $i$  regions which were not attacked:  $P(e_{i \neq r}, t_{i \neq r}, ta_{i \neq r}, b_{i \neq r} | A = false)$ . For each battle of type  $t$  we had one observation for  $P(ad, gd, id | H = t)$  and  $P(H = t | p)$ . By learning with a Laplace’s law of succession [25], we allow for unseen event to have a non-zero probability.

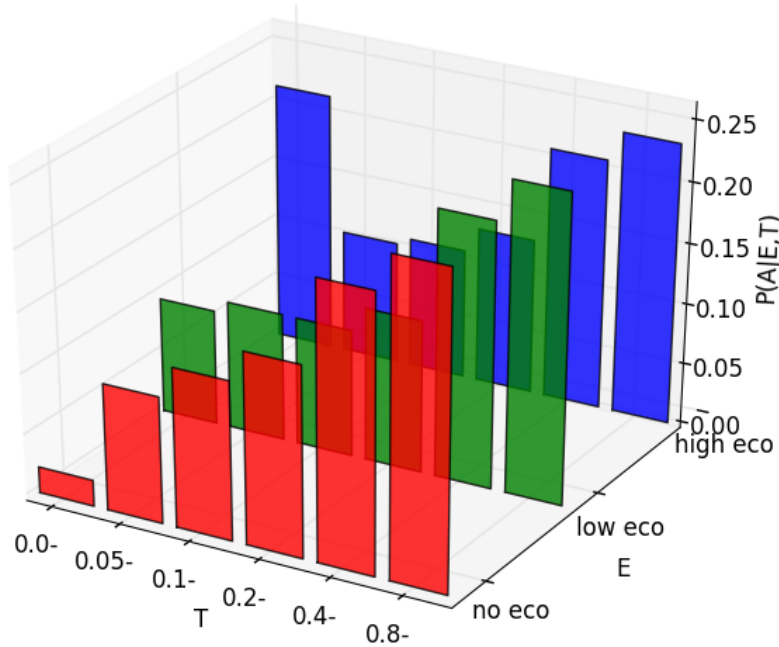
An exhaustive presentation of the learned tables is out of the scope of this paper, but we displayed interesting cases in which the posteriors of the learned model concur with human expertise in Figures 3 and 4. In Fig. 3, we see that air raids/attacks are quite risk averse and it is two times more likely to attack a region with less than 1/10th of the flying force in anti-aircraft warfare than to attack a region with up to one half of our force. We can also notice that drops are to be preferred either when it is safe to land (no anti-aircraft defense) or when there is a large defense (harassment tactics). In Fig. 3 we can see that, in general, there are as many ground attacks at the sum of other types. The two top graphs show cases in which the tech of the attacker was very specialized, and, in such cases, the specificity seems to be used. In particular, the top right graphic may be corresponding to a “fast Dark Templars rush”. Finally, Fig. 4 shows the transition between two types of encounters: tactics aimed at engaging the enemy army (a higher  $T$  value entails a higher  $P(A)$ ) and tactics aimed at damaging the enemy economy (at high  $E$ , we look for opportunities to attack with a small army where  $T$  is lower).

### 4.2 Prediction Performance

We learned and tested one model for each race and each match-up. As we want to predict *where* ( $P(A_{1:n})$ ) and *how* ( $P(H_{battle})$ ) the next attack will happen to us, we used inferred enemy  $TT$  (to produce  $P$ ) and  $TA$ , our scores being fully known:  $E, T, B, ID$ . We consider  $GD, AD$  to be fully known even though they depend on the attacker force, we should have some uncertainty on them, but we tested that they accounted (being known instead of fully unknown) for 1 to 2% of  $P(H)$  accuracy (in prediction) once  $P$  was known. We should point that pro-gamers scout very well and so it allows for a highly accurate  $TT$  estimation with [4]. Training requires to recreate battle states (all units’ positions) and count parameters for 5000 to 30000 battles (depending on the match-up). Once that is done, inference is very quick: a look-up in a probability table for known values and  $\#F$  look-ups for free variables  $F$  on which we sum. We chose to try and predict the next battle 30 seconds before it happens, 30 seconds



**Fig. 3.** Left: (top)  $P(H = invis)$  and  $P(H = drop)$  for varying values of  $GD$  (summed on other variables); (bottom)  $P(H = air)$  and  $P(H = drop)$  for varying values of  $AD$  (summed on other variables), for Terran in TvP. We can see that it is far more likely that invisible (“sneaky”) attacks happen where there is low ground presence (top left plot). For drops, we understand that the high value for  $P(H = drop|GD = 1.0)$  is caused by the fact that drop armies are small and this value corresponds to drops which are being expected by the defender. Drops at lower values of  $GD$  correspond to unexpected (surprise) drops. As ground units are more cost efficient than flying units in a static battle, we see that both  $P(H = invis|AD = 0.0)$  and  $P(H = drop|AD = 0.0)$  are much more probable than situations with air defenses. Right:  $P(H|HP)$  for varying values of  $H$  and for different values of  $P$  (derived from inferred  $TT$ ), for Protoss in PvT. Conditioning on what is possible given the *tech tree* gives a lot of information about what attack types are possible or not. More interestingly, it clusters the game phases in different tech levels and allows for learning the relative distributions of attack types with regard to each phase. For instance, the last (bottom right) plot shows the distribution on attack types at the end of a technologically complete game.



**Fig. 4.**  $P(A)$  for varying values of  $E$  and  $T$ , summed on the other variables, for Terran in TvT. Higher economical values is strongly correlated with surprise attacks with small tactical squads and no defenses, which almost never happens in open fields (“no eco”) as this would lead to very unbalanced battles (in terms of army sizes): it would not benefit the smaller party, which can flee and avoid confrontation, as opposed to when defending their base.

being an approximation of the time needed to go from the middle of a map (where the entropy on “next battle position” is maximum) to any region by ground, so that the prediction is useful for the defender (they can position their army).

The model code<sup>7</sup> (for learning and testing) as well as the datasets (see above) are freely available. Raw results of predictions of positions and types of attacks 30 seconds before they happen are presented in Table. 1: for instance the bold number (38.0) corresponds to the percentage of good positions (regions) predictions (30 sec before event) which were ranked 1st in the probabilities on  $A_{1:n}$  for Protoss attacks against Terran (PvT). The measures on *where* corresponds to the percentage of good prediction and the mean probability for given ranks in  $P(A_{1:n})$  (to give a sense of the shape of the distribution). As the most probable The measures on *how* corresponds to the percentage of good predictions for the most probable  $P(H_{battle})$  and the number of such battles seen in the test set for given attack types. We particularly predict well ground attacks (trivial in the early game, less in the end game) and, interestingly, Terran and Zerg drop attacks. The *where & how* row corresponds to the percentage of good predictions for the maximal probability in the joint  $P(A_{1:n}, H_{1:n})$ : considering only the most probable attack (more information is in the rest of the distribution, as shown for *where*!) according to our model, we can predict *where and how* an attack will occur in the next 30 seconds  $\approx 1/4$ th of the time. Finally, note that scores are also useful 60 seconds before the attack (obviously, *TT*, and thus *P*, are not so different, nor are *B* and *E*): PvT *where* top 4 ranks are 35.6, 8.5, 7.7, 7.0% good versus 38.0, 16.3, 8.9, 6.7% 30 seconds before; *how* total precision 60 seconds before is 70.0% vs. 72.4%, *where & how* maximum probability precision is 19.9% vs. 23%. This gives even more time for the player to adapt their tactics.

**Table 1.** Results summary for multiple metrics at 30 seconds before attack. The number in bold (38.0) is read as “38% of the time, the region  $i$  with probability of rank 1 in  $P(A_i)$  is the one in which the attack happened 30 seconds later”.

%: good predictions Pr: mean probability total # games		Protoss						Terran						Zerg					
		P		T		Z		P		T		Z		P		T		Z	
		445		2408		2027		2408		461		2107		2027		2107		199	
measure	rank	%	Pr	%	Pr	%	Pr	%	Pr	%	Pr	%	Pr	%	Pr	%	Pr	%	Pr
<i>where</i>	1	40.9	.334	<b>38.0</b>	.329	34.5	.304	35.3	.299	34.4	.295	39.0	0.358	32.8	.31	39.8	.331	37.2	.324
	2	14.6	.157	16.3	.149	13.0	.152	14.3	.148	14.7	.147	17.8	.174	15.4	.166	16.6	.148	16.9	.157
	3	7.8	.089	8.9	.085	6.9	.092	9.8	.09	8.4	.087	10.0	.096	11.3	.099	7.6	.084	10.7	.100
	4	7.6	.062	6.7	.059	7.9	.064	8.6	.071	6.9	.063	7.0	.062	8.9	.07	7.7	.064	8.6	.07
measure	type	%	N	%	N	%	N	%	N	%	N	%	N	%	N	%	N	%	N
<i>how</i>	G	97.5	1016	98.1	1458	98.4	568	100	691	99.9	3218	76.7	695	86.6	612	99.8	567	67.2	607
	A	44.4	81	34.5	415	46.8	190	40	5	13.3	444	47.1	402	14.2	155	15.8	19	74.2	586
	I	22.7	225	49.6	337	12.9	132	NA	NA	NA	NA	36.8	326	32.6	227	NA	NA	NA	NA
	D	55.9	340	42.2	464	45.2	93	93.5	107	86	1183	62.8	739	67.7	535	81.4	86	63.6	588
total		76.3	1662	72.4	2674	71.9	983	98.4	806	88.5	4850	60.4	2162	64.6	1529	94.7	674	67.6	1802
where & how (%)		32.8		23		23.8		27.1		23.6		30.2		23.3		30.9		26.4	

When we are mistaken, the mean ground distance (pathfinding wise) of the most probable predicted region to the good one (where the attack happens) is 1223 pixels (38 build tiles, or 2 screens in StarCraft’s resolution), while the mean max distance on the map is 5506 (172 build tiles). Also, the mean number of regions by map is 19, so a random *where* (attack destination) picking policy would have a correctness of 1/19 (5.23%). For choke-centered regions, the numbers of good *where* predictions are lower (between 24% and 32% correct for the most probable) but the mean number of regions by map is 42. For *where & how*, a random policy would have a precision of  $1/(19*4)$ , and even a random policy taking the high frequency of ground attacks into account would at most be  $\approx 1/(19*2)$  correct.

For the location only (*where* question), we also counted the mean number of different regions which were attacked in a given game (between 3.97 and 4.86 for regions, depending on the match-up, and between 5.13 and 6.23 for choke-dependent regions). The ratio over these means would give the best prediction rate we could expect from a *baseline heuristic* based solely on the location data. These are attacks that actually happened, so the number of regions a player have to be worried about is at least this one (or more, for regions which were not attacked during a game but were potential targets). This *baseline heuristic* would yield (depending on the match-up) prediction rates between 20.5 and 25.2% for regions, versus our 32.8 to 40.9%, and between 16.1% and 19.5% for choke-dependent regions, versus our 24% to 32%.

Note that our current model considers a uniform prior on regions (no bias towards past battlefields) and that we do not incorporate any derivative of the armies’ movements. There is no player modeling at all: learning and fitting the mean player’s tactics is not optimal, so we should specialize the probability tables for each player. Also, we use all types of battles in our training and testing. Short experiments showed that if we used only attacks on bases, the probability of good *where* predictions for the maximum of  $P(A_{1:n})$  goes above 50% (which is not a surprise, there are far less bases than regions in which attacks happen). To conclude on tactics positions prediction: if we sum the 2 most probable regions for the attack, we are right at least half the time; if we sum the 4 most probable (for our robotic player, it means it prepares against attacks in 4 regions as opposed to 19), we are right  $\approx 70\%$  of the time.

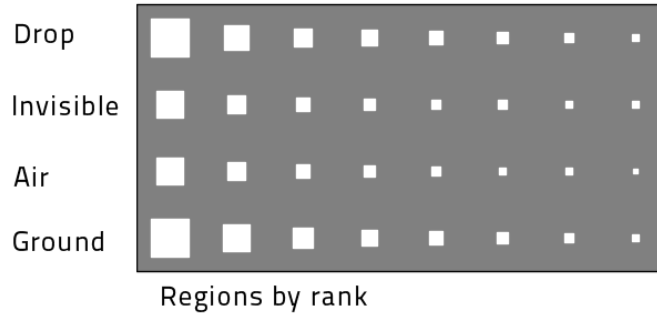
<sup>7</sup> <https://github.com/SnippyHollow/AnalyzeBWData> 422

Mistakes on the type of the attack are high for invisible attacks: while these tactics can definitely win a game, the counter is strategic (it is to have detectors technology deployed) more than positional. Also, if the maximum of  $P(H_{battle})$  is wrong, it doesn't mean that  $P(H_{battle} = good) = 0.0$  at all! The result needing improvements the most is for air tactics, because countering them really is positional, see our discussion in the conclusion.

### 4.3 In Game Decision-Making

In a StarCraft game, our bot has to make decisions about where and how to attack or defend, it does so by reasoning about opponent's tactics, bases, its priors, and under strategic constraints (Fig. 1). Once a decision is taken, the output of the tactical model is an offensive or defensive goal. There are different military goal types (base defense, ground attacks, air attacks, drops...), and each type of goal has pre-requisites (for instance: a drop goal needs to have the control of a dropship and military units to become active). The spawned goal then autonomously sets objectives for Bayesian units [3], sometimes procedurally creating intermediate objectives or canceling itself in the worst cases.

The destinations of goals are from  $P(A)$ , while the type of the goal comes from  $P(H)$ . In input, we fully know tactical scores of the regions according to our military units placement  $TA$  (we are the attacker), what is possible for us to do  $P$  (according to units available) and we estimate  $E, T, B, ID, GD, AD$  from past (partial) observations. Estimating  $T$  is the most tricky of all because it may be changing fast, for that we use a units filter which just decays probability mass of seen units. An improvement would be to use a particle filter [2], with a learned motion model. From the joint (2)  $P(A_{1:n}, H_{1:n}|ta, p, tt)$  may arise a couple  $i, H_i$  more probable than the most probables  $P(A_i)$  and  $P(H_j)$  taken separately (the case of an heavily defended main base and a small unprotected expand for instance). Fig. 5 displays the mean  $P(A, H)$  for Terran (in TvZ) attacks decision-making for the most 32 probable type/region tactical couples. It is in this kind of landscape (though more steep because Fig. 5 is a mean) that we sample (or pick the most probable couple) to take a decision. Also, we may spawn defensive goals countering the attacks that we predict from the opponent.



**Fig. 5.** Mean  $P(A, H)$  for all  $H$  values and the top 8  $P(A_i, H_i)$  values, for Terran in TvZ. The larger the white square area, the higher  $P(A_i, H_i)$ . A simple way of taking a tactical decision according to this model, and the learned parameters, is by sampling in this distribution.

Finally, we can steer our technological growth towards the opponent's weaknesses. A question that we can ask our model (at time  $t$ ) is  $P(TT)$ , or, in two parts: we first find  $i, h_i$  which maximize  $P(A, H)$  at time  $t + 1$ , and then ask a more directive:

$$P(TT|h_i) \propto \sum_P P(h_i|HP)P(P|TT)P(TT)$$

so that it gives us a distribution on the tech trees ( $TT$ ) needed to be able to perform the wanted attack type. To take a decision on our technology direction, we can consider the distances between our current  $tt^t$  and all the probable values of  $TT^{t+1}$ .

## 5 Conclusions

### 5.1 Possible Improvements

There are three main research directions for possible improvements: improving the underlying heuristics, improving the dynamic of the model and improving the model itself. The heuristics presented here are quite simple but they may be changed, and even removed or added, for another RTS or FPS, or for more performance. In particular, our “defense against invisible” heuristic

could take detector positioning/coverage into account. Our heuristic on tactical values can also be reworked to take terrain tactical values into account (chokes and elevation in StarCraft). For the estimated position of enemy units, we could use a particle filter [2] with a motion model (at least one for ground units and one for flying units). There is room to improve the dynamics of the model: considering the prior probabilities to attack in regions given past attacks and/or considering evolutions of the  $T, TA, B, E$  values (derivatives) in time. The discretizations that we used may show their limits, though if we want to use continuous values, we need to setup a more complicated learning and inference process (MCMC sampling). Finally, one of the strongest assumptions (which is a drawback particularly for prediction) of our model is that the attacking player is always considered to attack in this most probable regions. While this would be true if the model was complete (with finer army positions inputs and a model of what the player thinks), we believe such an assumption of completeness is far fetched. Instead we should express that incompleteness in the model itself and have a “player decision” variable  $D \sim Multinomial(P(A_{1:n}, H_{1:n}), player)$ .

## 5.2 Final Words

We have presented a Bayesian tactical model for RTS AI, which allows both for opposing tactics prediction and autonomous tactical decision-making. Being a probabilistic model, it deals with uncertainty easily, and its design allows easy integration into multi-granularity (multi-scale) AI systems as needed in RTS AI. Without any temporal dynamics, its exact prediction rate of the joint position and tactical type is in [23-32.8]% (depending on the match-up), and considering the 4 most probable regions it goes up to  $\approx 70\%$ . More importantly, it allows for tactical decision-making under (technological) constraints and (state) uncertainty. It can be used in production thanks to its low CPU and memory footprint. The dataset, its documentation<sup>8</sup>, as well as our model implementation<sup>9</sup> (and other data-exploration tools) are free software and can be found online. We plan to use this model in our StarCraft AI competition entry bot as it gives our bot tactical autonomy and a way to adapt to our opponent.

## References

1. Hagelbäck, J., Johansson, S.J.: A Study on Human like Characteristics in Real Time Strategy Games. In: Proceedings of CIG, IEEE (2010)
2. Weber, B.G., Mateas, M., Jhala, A.: A Particle Model for State Estimation in Real-Time Strategy Games. In: Proceedings of AIIDE, Stanford, Palo Alto, California, AAAI Press (2011) 103–108
3. Synnaeve, G., Bessière, P.: A Bayesian Model for RTS Units Control applied to StarCraft. In: Proceedings of CIG, Seoul, South Korea, IEEE (September 2011)
4. Synnaeve, G., Bessière, P.: A Bayesian Model for Plan Recognition in RTS Games applied to StarCraft. In: Proceedings of AIIDE, Palo Alto, CA, USA, AAAI Press (October 2011) 79–84
5. Synnaeve, G., Bessière, P.: A Bayesian Model for Opening Prediction in RTS Games with Application to StarCraft. In: Proceedings of CIG, Seoul, South Korea, IEEE (September 2011)
6. Aha, D.W., Molineaux, M., Ponsen, M.J.V.: Learning to Win: Case-Based Plan Selection in a Real-Time Strategy Game. In: ICCBR. (2005) 5–20
7. Ontañón, S., Mishra, K., Sugandh, N., Ram, A.: Case-based planning and execution for real-time strategy games. In: Proceedings of the 7th international conference on Case-Based Reasoning: Case-Based Reasoning Research and Development. International Joint Conference on Neural Networks (ICCBR-07), Springer-Verlag (2007) 164–178
8. Mishra, K., Ontañón, S., Ram, A.: Situation Assessment for Plan Retrieval in Real-Time Strategy Games. In: ECCBR. (2008) 355–369
9. Meta, M., Ontañón, S., Ram, A.: Meta-Level Behavior Adaptation in Real-Time Strategy Games. In: ICCBR-10 Workshop on Case-Based Reasoning for Computer Games, Alessandria, Italy. (2010)
10. Bakkes, S.C.J., Spronck, P.H.M., Jaap van den Herik, H.: Opponent modelling for case-based adaptive game AI. Entertainment Computing 1(1) (January 2009) 27–37
11. Sharma, M., Holmes, M., Santamaria, J., Irani, A., Isbell, C.L., Ram, A.: Transfer Learning in Real-Time Strategy Games Using Hybrid CBR/RL. In: International Joint Conference of Artificial Intelligence, IJCAI. (2007)
12. Cadena, P., Garrido, L.: Fuzzy Case-Based Reasoning for Managing Strategic and Tactical Reasoning in StarCraft. In: Proceedings of MICA (1), Springer (2011) 113–124
13. Chung, M., Buro, M., Schaeffer, J.: Monte Carlo Planning in RTS Games. In: Proceedings of CIG, IEEE (2005)
14. Gelly, S., Wang, Y.: Exploration exploitation in Go: UCT for Monte-Carlo Go. In: NIPS: Neural Information Processing Systems Conference On-line trading of Exploration and Exploitation Workshop, Canada (December 2006)
15. Balla, R.K., Fern, A.: UCT for Tactical Assault Planning in Real-Time Strategy Games. In: IJCAI. (2009)
16. Weber, B.G., Mateas, M., Jhala, A.: Applying Goal-Driven Autonomy to StarCraft. In: Artificial Intelligence and Interactive Digital Entertainment (AIIDE). (2010)
17. Weber, B.G., Mawhorter, P., Mateas, M., Jhala, A.: Reactive Planning Idioms for Multi-Scale Game AI. In: Proceedings of CIG, IEEE (2010)

<sup>8</sup> <http://snippyhollow.github.com/bwrepdump/>

<sup>9</sup> <https://github.com/SnippyHollow/AnalyzeBWDat>

18. Kabanza, F., Bellefeuille, P., Bisson, F., Benaskeur, A.R., Irandoust, H.: Opponent Behaviour Recognition for Real-Time Strategy Games. In: AAAI Workshops. (2010)
19. Forbus, K.D., Mahoney, J.V., Dill, K.: How qualitative spatial reasoning can improve strategy game ais. *IEEE Intelligent Systems* **17** (July 2002) 25–30
20. Perkins, L.: Terrain Analysis in Real-Time Strategy Games: An Integrated Approach to Choke Point Detection and Region Decomposition. In: Proceedings of AIIDE, AAAI Press (2010)
21. Wintermute, S., Joseph Xu, J.Z., Laird, J.E.: SORTS: A Human-Level Approach to Real-Time Strategy AI. In: Proceedings of AIIDE, AAAI Press (2007) 55–60
22. Ponsen, M.J.V., Muñoz-Avila, H., Spronck, P., Aha, D.W.: Automatically Generating Game Tactics through Evolutionary Learning. *AI Magazine* **27**(3) (2006) 75–84
23. Avery, P., Louis, S., Avery, B.: Evolving Coordinated Spatial Tactics for Autonomous Entities using Influence Maps. In: Proceedings of the 5th international conference on Computational Intelligence and Games. Proceedings of CIG, Piscataway, NJ, USA, IEEE (2009) 341–348
24. Smith, G., Avery, P., Houmanfar, R., Louis, S.: Using Co-evolved RTS Opponents to Teach Spatial Tactics. In: Proceedings of CIG, IEEE (2010)
25. Jaynes, E.T.: Probability Theory: The Logic of Science. Cambridge University Press (June 2003)
26. Diard, J., Bessière, P., Mazer, E.: A Survey of Probabilistic Models Using the Bayesian Programming Methodology as a Unifying Framework. In: Conference on Computational Intelligence, Robotics and Autonomous Systems, CIRAS. (2003)
27. Lebeltel, O., Bessière, P., Diard, J., Mazer, E.: Bayesian Robot Programming. *Autonomous Robots* **16**(1) (2004) 49–79
28. Bessière, P., Laugier, C., Siegwart, R.: Probabilistic Reasoning and Decision Making in Sensory-Motor Systems. 1st edn. Springer Publishing Company, Incorporated (2008)
29. Colas, F., Diard, J., Bessière, P.: Common Bayesian Models for Common Cognitive Issues. *Acta Biotheoretica* **58** (2010) 191–216
30. Le Hy, R., Arrigoni, A., Bessière, P., Lebeltel, O.: Teaching Bayesian behaviours to video game characters. *Robotics and Autonomous Systems* **47** (2004) 177–185
31. Synnaeve, G., Bessière, P.: Bayesian Modeling of a Human MMORPG Player. In: 30th international workshop on Bayesian Inference and Maximum Entropy, Chamonix, France (July 2010)

# Monte-Carlo Tree Search for the Simultaneous Move Game Tron

Niek G.P. Den Teuling and Mark H.M. Winands

Games and AI Group, Department of Knowledge Engineering,  
Faculty of Humanities and Sciences,  
Maastricht University, Maastricht, The Netherlands  
{n.denteuling@student., m.winands@}maastrichtuniversity.nl

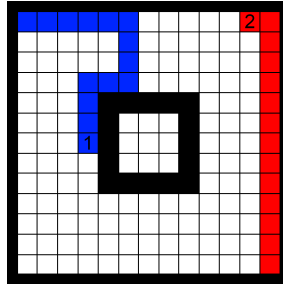
**Abstract.** Monte-Carlo Tree Search (MCTS) has been successfully applied to many games, particularly in Go. In this paper, we investigate the performance of MCTS in Tron, which is a two-player simultaneous move game. We try to increase the playing strength of an MCTS program for the game of Tron by applying several enhancements to the selection, expansion and play-out phase of MCTS.

Based on the experiments, we may conclude that Progressive Bias, altered expansion phase and play-out cut-off all increase the overall playing strength, but the results differ per board. MCTS-Solver appears to be a reliable replacement for MCTS in the game of Tron, and is preferred over MCTS due to its ability to search the state space for a proven win. The MCTS program is still outperformed by the best  $\alpha\beta$  program A1K0N, which uses a sophisticated evaluation function, indicating that there is quite some room for improvement.

## 1 Introduction

The classic way of exploring the game tree is using  $\alpha\beta$ -search [9] with a domain-specific evaluation function. However, for games that require a complex positional evaluation function, this approach might not be the best way. An alternative approach is Monte-Carlo Tree Search (MCTS) [6, 10]. In contrast to  $\alpha\beta$ -search, MCTS does not require a positional evaluation function as it relies on stochastic simulations. MCTS has proven itself to be a viable alternative in, for instance, the board games Go [6], Hex [1], Amazons [11] and Lines of Action [19].

A challenging new game is Tron. It is a two-player game that bears resemblance to Snake, except that in Tron, players leave a wall behind at each move. An interesting aspect of Tron is that it is a simultaneous move game, rather than the usual turn-taking game. In 2010, the University of Waterloo Computer Science Club organized an AI tournament for the game of Tron [18]. Overall, the MCTS programs were outperformed by  $\alpha\beta$  programs. In this paper, the performance of MCTS in Tron is investigated, continuing the pioneering work performed by Samothrakis *et al.* [12]. We examine approaches to improve the program's playing strength, by trying out different evaluation functions and



**Fig. 1.** A Tron game on a board with obstacles after 13 moves. The blue player (1) has cut off the upper part of the board, restricting the space the red player (2) can fill.

MCTS enhancements. Our MCTS program equipped with the enhancements is subsequently matched against the top  $\alpha\beta$  program A1K0N.

This paper is organized as follows. Section 2 gives a brief description of the game of Tron and the difficulties that a program has to be able to handle, to play at a decent skill level. Section 3 explains the MCTS and MCTS-Solver method applied to Tron. The enhancements applied to MCTS regarding the selection strategy are described in Section 4, followed by an enhanced expansion strategy in Section 5. Play-out strategies are described in Section 6. Experiments and results are given in Section 7. Finally, in Section 8, conclusions from the results are drawn and future research is suggested.

## 2 The Game of Tron

The game of Tron originates from the movie *Tron*, released by Walt Disney Studios in 1982. The movie is about a virtual world where motorcycles drive at a constant speed and can only make  $90^\circ$  angles, leaving a wall behind them as they go. The game of Tron investigated in this paper is a board version of the game played in the movie.

Tron is a two-player board game played on an  $m \times n$  grid. It is similar to Snake: each player leaves a wall behind them as they move. In Snake, the player's wall is of a limited length, but Tron does not have such a restriction. At each turn, the red and blue player can only move one square straight ahead, or to the left or right. Both players perform their moves at the same time; they have no knowledge of the other player's move until the next turn. Players cannot move to a square that already contains a wall. If both players move to the same square, it is considered a draw. If a player moves into a wall, he loses and the other player wins. Usually the boards are symmetric, such that none of the players has an advantage over the other player. A typical board size is  $13 \times 13$ .

The game is won by outlasting your opponent such that the opponent has no moves left other than moving into a wall. At the early stage of the game, it is difficult to find good moves as the number of possible move sequences is quite large and it is difficult to predict what the opponent will do. Boards can contain

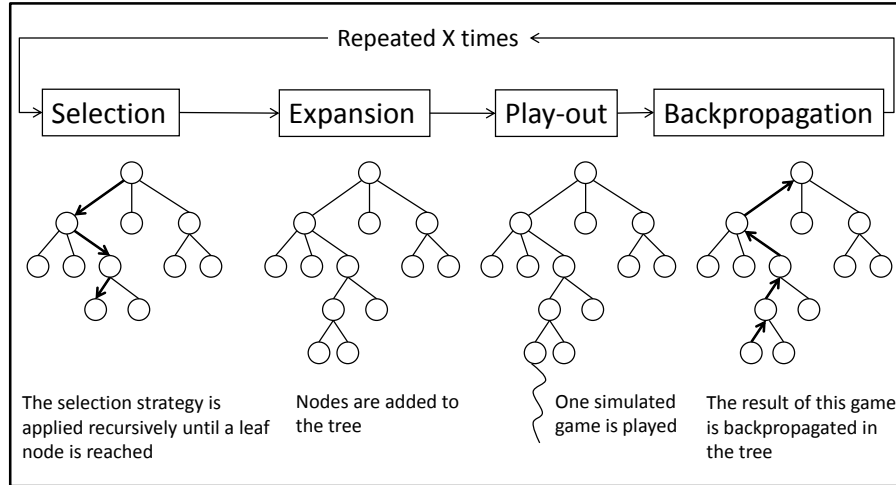
obstacles (see Figure 1), further increasing the difficulty of the game because filling the available space becomes a more difficult task. Obstacles can provide opportunities to cut off an opponent, reducing the opponent's free space while maximizing your own.

### 3 Monte-Carlo Tree Search

Monte-Carlo Tree Search (MCTS) is a best-first search method that constructs a search tree using many simulations (called play-outs) [6, 10]. Play-outs are used to evaluate a certain position. Positions with a high winning percentage are preferred over those with a lower winning percentage. MCTS constructs a search tree consisting of nodes, where each node represents a position of the game. Each node  $i$  has a value  $v_i$  and a visit count  $n_i$ . The search starts from the root node, which represents the current position. The tree is explored at random, but as the number of simulated nodes increases, it gains a better evaluation of the nodes and can focus on the most promising nodes.

Although Tron is a simultaneous move game, it is possible to represent it as a turn-taking game. The player under consideration is always first to move inside the tree, followed by the other player. An issue arises when the players can run into each other. A solution is given in Subsection 3.1.

MCTS is divided into four phases [5]. These phases are executed until time is up. The phases are illustrated in Figure 2. We explain the phases in detail below.



**Fig. 2.** Outline of the Monte-Carlo Tree Search [5].

**Selection.** In the selection phase, a child node of a given node is selected according to some strategy until a leaf node is reached. The selection task is an important one, as the goal is to find the best move. Because moves are evaluated by simulation, promising nodes should be played (exploited) more often than unpromising nodes. However, to find these nodes, unvisited nodes have to be tried out as well (exploration). Considering that in Tron, a player has at most 3 different moves at any turn (except for the first turn, where there could be 4 moves), this is not a problem. Because the number of simulations that can be performed is limited, a good balance has to be found between exploring and exploiting nodes. The simplest selection strategy is selecting a child node at random. A selection strategy that provides a balance between exploration and exploitation, is UCT (Upper Confidence Bound applied to Trees) [10]. It is based on the UCB1 algorithm (Upper Confidence Bound) [2]. UCT selects a child node  $k$  from node  $p$  as follows:

$$k \in \operatorname{argmax}_{i \in I} \left( v_i + C \times \sqrt{\frac{\ln n_p}{n_i}} \right) \quad (1)$$

$C$  is a constant, which has to be tuned experimentally. Generally, UCT is applied after the node has first been visited a certain number of times  $T$ , to ensure all nodes have been sufficiently explored to apply UCT. If a node has a visit count less than  $T$ , the random-selection strategy is applied [6].

**Expansion.** In the expansion phase, the selected leaf node  $p$  is expanded. Since the number of child nodes is at most 3 in Tron, all nodes are added. The selection strategy is then applied to node  $p$ , returning the node from which the play-out starts.

**Play-out.** In this phase, the game is simulated in self-play, starting from the position of the selected node. Moves are performed until the game ends, or when the outcome can be estimated reliably. In contrast to the selection phase, both players move simultaneously in the play-out phase. The strategy used for selecting the moves to play can either be performing random moves, or using domain-specific knowledge that increases the quality of the simulation. The play-out phase returns a value of 1, 0 or -1 for the play-out node  $p$ , depending on whether the simulated game resulted in a win, draw, or loss, respectively. The same values are awarded to terminal nodes in the search tree. If the play-out node belongs to the player under consideration, the other player will first perform a move, such that both players have performed the same number of moves.

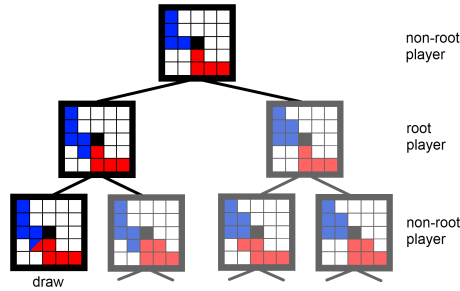
**Backpropagation.** The result of the simulation is backpropagated from the leaf node all the way back to the root node of the search tree. The values of the nodes are updated to match the new average of the play-outs.

After the simulations, the final move to be played by the MCTS program has to be chosen. The move is selected by taking the most ‘secure’ child of the root node [5]. The secureness of a node  $i$  is defined as:  $v_i + \frac{A}{\sqrt{n_i}}$ , where  $A$  is a constant. In the experiments, based on trial-and-error for the MCTS program,  $A = 1$  is used.

### 3.1 Handling Simultaneous Moves

Treating Tron as a turn-taking game inside the tree works out quite well in almost every position of the game. However, if a position arises where the MCTS program has the advantage, but the players are at risk of crashing into each other, the program might play the move that leads to a draw. This happens because inside the search tree, the root player is always the first to move (as done in [12]). Because the root player already moved to the square that was accessible to both, the non-root player can no longer move to this square.

This problem is solved by adding an enhancement to the expansion strategy: if a node  $n$  belongs to the root player, and the non-root player could have moved to the square the root player is currently at, a terminal node is added to  $n$  with the value of a draw (i.e., 0). An example is shown in Figure 3.



**Fig. 3.** A game tree of Tron. In the left-most branch, both players moved to the same square, resulting in a terminal node that ends in a draw.

### 3.2 Monte-Carlo Tree Search Solver

Monte-Carlo Tree Search Solver (MCTS-Solver) [19] is an enhancement for MCTS that is able to prove the game-theoretic value of a position. MCTS combined with UCT may require significantly more time to find the best move, because it requires a large number of play-outs to converge to the game-theoretic value. Running MCTS-Solver requires a negligible amount of additional computation time on top of MCTS. Since proven positions do not have to be evaluated again, time can be spent on other positions that have not been proven yet. The original MCTS-Solver only considered win and loss outcomes, because in the test domain Lines of Action draws are exceptional [19]. Since draws occur often in Tron, an enhanced MCTS-Solver is used that does handle draws.

The *Score-Bounded MCTS-Solver* [4] extends MCTS-Solver to games that have more than two game-theoretic outcomes. It attaches an interval to each node, as done in the B\* algorithm [3]. The interval is described by a pessimistic and optimistic bound. The pessimistic score represents the lowest achievable outcome for the root player, and the optimistic score represents the best achievable

outcome for the root player. Given sufficient time, the pessimistic and optimistic score of a node  $n$  will converge to its true value. An advantage of Score-Bounded MCTS is that the bounds enable pruning as in  $\alpha\beta$  search, skipping unpromising branches. The initial bound of a node is set to  $[-1.0, 1.0]$ . Score-Bounded MCTS has been shown to solve positions considerably faster than MCTS-Solver [4].

### 3.3 Progressive Bias

Although UCT gives good results compared to other selection strategies that do not use knowledge of the game, there is room for improvement. The random strategy applied when the visit count of the node is small, can be replaced by the more promising play-out strategy. Since the accuracy of the UCT selection strategy increases as the number of visits increases, it is desirable to introduce a so-called ‘progressive strategy’. The progressive strategy provides a soft transition between the two strategies [5]. A popular choice is *progressive bias* (PB) that combines heuristic knowledge with UCT to select the best node [5]. By using Tron knowledge, node selection can be guided in a more promising direction, one that might not have been found by using play-outs only. This domain knowledge can be computationally expensive. A trade-off has to be made between simulating more games and spending more time for computing the domain knowledge.

When few games have been played, the heuristic knowledge has a major influence on the decision. The influence gradually decreases when the node is visited more often. The PB formula is as follows:  $\frac{W \times P_{mc}}{l_i + 1}$  [19].  $W$  is a constant (set to 10 in the experiments).  $P_{mc}$  is the transitional probability of a move category  $mc$  [17].  $l_i$  denotes the number of losses in node  $i$ , this way, nodes that do not turn out well are not biased for too long. The formula of UCT and PB combined is:

$$k \in \operatorname{argmax}_{i \in I} \left( v_i + C \times \sqrt{\frac{\ln n_p}{n_i}} + \frac{W \times P_{mc}}{l_i + 1} \right) \quad (2)$$

The transitional probability of each move category is acquired from games played by expert players. Since no such games are available, the probabilities are obtained from self-play experiments of the MCTS program. The transitional probability  $P_{mc}$  of a move belonging to the move category  $mc$  is given by:

$$P_{mc} = \frac{n_{played(mc)}}{n_{available(mc)}} \quad (3)$$

$n_{played(mc)}$  denotes the number of positions in which a move belonging to move category  $mc$  was played.  $n_{available(mc)}$  is the number of positions where a move belonging to move category  $mc$  could have been played.

We distinguish six move features in Tron:

- Passive: The player follows the wall that it is currently adjacent to.
- Offensive: The player moves towards the other player, when close to each other.

- Defensive: The player moves away from the other player, when close to each other.
- Territorial: The player attempts to close off a space by moving across open space towards a wall.
- Reckless: The player moves towards a square where the other player could have moved to, risking a draw.
- Obstructive: The player moves to a square that contains paths to multiple subspaces, closing off these spaces (at least locally).

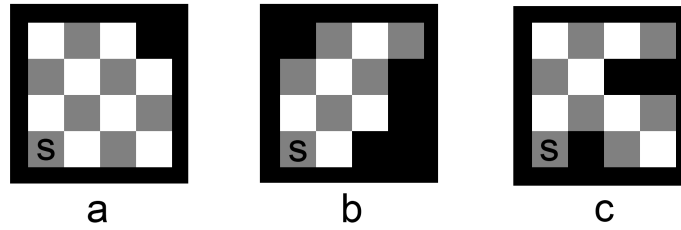
## 4 Heuristic Knowledge in Tron

Estimating the remaining available space of a player is a useful heuristic in Tron because the game is won by filling a larger space than the opponent. *Space estimation* is valuable when, for instance, the program is at a square where it has to choose between two spaces. Biasing the selection towards moving to larger spaces saves time on simulating less-promising nodes that lead towards smaller spaces. We only focus on estimating the number of moves a player can make in a space that is not reachable by the other player.

Counting the number of empty squares does not always give an accurate estimation of the number of moves a program requires to fill the available space. Spaces can contain squares that can be reached, but offer no path back to fill the rest of the space.

One way to get an estimation of the available space is by filling up the space in a single-player simulation, and counting the number of moves [13]. The simulation uses a greedy *wall-following heuristic*. This heuristic works as follows. A move is selected such that the player moves to a square that lies adjacent to one or more walls (excluding the wall at its current square). If any of the moves cuts the player off from the other possible moves, the available space of each move is estimated and the move leading having the largest available space is selected. If there are multiple possible moves of equal score, a move is selected at random. This method does not always give the correct number of moves, but it gives a good lower bound on the amount of available space.

Instead of counting the number of empty squares, a tighter upper bound can be obtained by treating the board as a checker board. The difference in the number of grey and white squares gives an indication of the number of moves that can be performed [7]. The estimated number of moves  $M$  is computed by:  $M = Z - |c_g - c_w|$ .  $Z$  is the total available space,  $c_g$  is the number of grey tiles (including the one the player is currently standing on), and  $c_w$  is the number of white tiles. The estimated number of moves can be substantially off if the space contains large subspaces that offer no way back to other subspaces. Three example spaces are shown in Figure 4. The estimated number of moves for boards  $a$ ,  $b$  and  $c$  are 13, 9 and 11, respectively. The true number of moves for the boards are 13, 9 and 10. Note that the estimation of board  $c$  is off because of the two separated subspaces. Had  $Z$  been solely used as the move estimation, the estimated number of moves would have been 14, 10 and 12.



**Fig. 4.** Three example boards where the player is isolated from the other player. The player starts at square  $S$ .

## 5 Predictive Expansion Strategy

In Tron, players will often get separated from each other (if not, the game ends in a draw). If a game is in such a position, the outcome of a game can be predicted in reliable way. The *predictive expansion* strategy uses space estimation to predict the outcome of a position. It has to be noted that only nodes of the non-root player are evaluated, so both players have performed their moves when the position is evaluated.

If the outcome of a node can be predicted with certainty, the node does not have to be simulated, and is treated as a terminal node. This works as follows. The node candidate for expansion is evaluated using the space estimation heuristic. If there is a way for the players to reach each other, the node is expanded in the default way. If the players are isolated from each other and the outcome can be predicted, the node is not expanded and it is treated as a terminal node. The result of the prediction is backpropagated.

This strategy has two advantages over the old expansion strategy. First, applying space estimation is faster than performing a play-out. If a sufficient number of games are cut off, the time spent on space estimation is regained, and more time can be spent on searching through other parts of the tree. Second, the outcome prediction is more reliable than performing multitudes of play-outs. This prevents the program from underestimating positions where one or more players are closed off in a large space.

Once the game reaches the endgame phase, the enhanced expansion strategy is no longer applied since the MCTS program is has shown to be capable of efficiently filling up the remaining space.

## 6 Play-out Strategies

The simplest play-out strategy is the *random-move* strategy. During the play-out, players perform a random move. Moves that result in an immediate defeat are excluded from the selection.

The authors observed that play-outs performed using a random-move strategy give surprisingly good results in Tron, considering that the randomly moving

Tron programs frequently trap themselves. The reliability of the play-outs can be further increased by using a more advanced play-out strategy [8].

We propose six play-out strategies. The resulting playing strength of each of these strategies is determined in the experiments.

**Wall-following strategy.** This strategy is inspired by the wall-following heuristic described in Section 4. The strategy selects the move leading to the square with the most number of walls (but smaller than 3). If multiple of moves lead to squares of the same number of walls, one of the moves is randomly selected. A problem with the wall-following strategy is that it does not leave much room for a rich variety of simulations. During each play-out, the moves performed will roughly be the same. It means that running more simulations does not necessarily increase the accuracy of the move value.

**Offensive strategy.** The offensive strategy selects moves that bring the player closer to the opponent player. If more than one move brings the player closer, one of the moves is selected at random. If there is no move that brings it closer to the opponent, a random move is performed.

**Defensive strategy.** This play-out strategy selects the move that increases the distance to the opponent player. If there is no such move, a random move is performed. If more than one move increases the distance from the opponent player, one of the moves is played at random.

**Mixed strategy.** The mixed strategy is a combination of the random play-out strategy and the previously mentioned strategies. At each move, a strategy is randomly selected according to a certain probability. The reasoning behind this strategy is that none of the strategies are particularly strong, and combining them may give better results.

The wall-following strategy has a 50% probability of being played, whereas the random-move, defensive and offensive are played 20%, 25% and 5% of the time, respectively.

**Move-category strategy.** This strategy uses the move category statistics used by the Progressive Bias enhancement to select a move. Moves are selected by roulette-wheel selection.

**$\epsilon$ -greedy strategy.** This strategy has a probability of  $1 - \epsilon$  (i.e. 90%) of playing the wall-following strategy, and a probability of  $\epsilon$  (i.e. 10%) of playing a random other play-out strategy [14, 15].

## 6.1 Endgame Strategies

The game of Tron can be split into two phases: the phase where players try to maximize their own available free space, and the phase where the players are isolated and attempt to outlast the other player by filling the remaining space as efficiently as possible, referred to as the endgame phase.

During the endgame phase, the same play-out strategies as mentioned above can be used, with the exception of the offensive and defensive strategy since there is no point in biasing the move on the position of the other player.

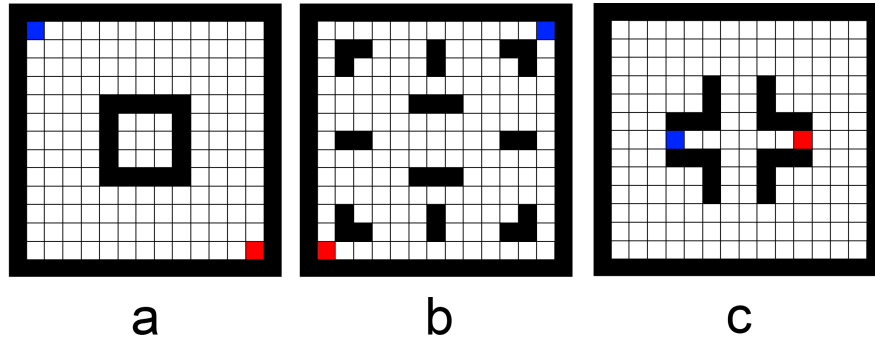


Fig. 5. The three boards used in the experiments.

## 6.2 Play-out Cut-Off

Although a Tron game is guaranteed to terminate, as each move brings the game moves closer to a terminal position, the number of moves performed during the play-out phase can be reduced. This saves time, leaving room for more simulations. Using heuristic knowledge, the result of a game can be predicted without the need to completely simulate it. A major problem with applying heuristic knowledge is that it costs much computation time compared to playing moves. Therefore, the positions are only evaluated once every 5 moves. An additional advantage of predicting the play-out outcome is that the accuracy of the play-outs is increased, because the player with the largest space can still lose a portion of the simulated games due to the weak play of the play-out strategies. The heuristic to predict the outcome of a game is the same as used in Subsection 4.

## 7 Experiments and Results

In this section, the proposed enhancements of Section 4, 5 and 6 are tested in our MCTS program [16]. The experiments are conducted on a 2.4 GHz AMD Opteron CPU with 8 GB of RAM. Experiments are conducted for three different boards, each providing different difficulties for MCTS. The boards are shown in Figure 5. Although all boards are symmetric, experiments are run for both colours to eliminate the possibility that the playing strength of the program is affected by its colour.

The following settings are used for the experiments, unless mentioned otherwise. In each experiment, 100 games are played on each board for both setups. In total 600 games are played. The time players can spend on computing the next move is 1 second. The programs have no knowledge about the move the other program will be performing.

Progressive Bias is tested in Subsection 7.1. Next, Subsection 7.2 describes the results of the various play-out strategies. Subsequently, Subsection 7.3 evaluates the play-out cut-off enhancement. The Predictive Expansion strategy is reported in Subsection 7.4. The playing strength of MCTS-Solver is tested in

Subsection 7.5. Finally, MCTS programs combined with the enhancements are tested against the winning program of the Tron Google AI Challenge in Subsection 7.6.

### 7.1 Progressive Bias

In the first series of experiment the Progressive Bias (PB) strategy is tested for different values of  $W$ . The MCTS-PB program is tested against the MCTS-UCT program [16]. Table 1 shows the transitional probabilities, obtained from 600 self-play games by the MCTS-UCT program at the three boards (200 games per board).

Table 2 shows that Progressive Bias does not improve the playing strength at boards  $a$  and  $b$  for any of the tested values of  $W$ , but it noticeably increases the playing strength at board  $c$ . Furthermore, the exact value of  $W$  does not seem to matter that much.

**Table 1.** Move categories and their respective transitional probabilities. **Table 2.** Win rates of MCTS-PB vs. MCTS-UCT.

Move category	$P_{mc}$
Defensive	28.9%
Defensive/Territorial	36.2%
Offensive	29.6%
Offensive/Reckless	0.0%
Offensive/Territorial	21.6%
Passive/Defensive	77.1%
Passive/Defensive/Obstructive	92.8%
Passive/Offensive	78.9%
Passive/Offensive/Obstructive	86.5%
Passive/Offensive/Reckless	6.6%
Passive/Off./Obs./Reck.	15.9%

W	Board $a$	Board $b$	Board $c$	Total
0.5	45%	48%	65%	$53 \pm 4 \%$
1	49%	45%	67%	$54 \pm 4 \%$
5	55%	45%	63%	$54 \pm 4 \%$
10	48%	49%	63%	$53 \pm 4 \%$
20	36%	52%	66%	$51 \pm 4 \%$

### 7.2 Play-out Strategy Experiments

In the next of series of experiments the various play-out strategies proposed are tested in the MCTS program. They are matched against each other in a round-robin tournament. The tournament is run at all three boards. Table 3 gives the results for each board. Table 4 shows the averaged results over all three boards.

The results show that the boards have a large influence on the effectiveness of the strategies. As such, it is difficult to select the best strategies based on these results. Overall, the random-move, defensive and wall-following strategies seem to be the best strategies. The random-move strategy performs well for all boards, whereas the defensive strategy stands out for board  $c$ . The wall-following strategy works well against the random-move strategy on board  $b$ . The random-move strategy is used in the experiments of the next subsections.

**Table 3.** Play-out strategy results for board *a*, *b* and *c*.

Board <i>a</i>	Rand.	Wall	Off.	Def.	Mixed	Cat.	$\epsilon$ -g.
Random		58%	90%	83%	71%	67%	52%
Wall	42%		90%	52%	21%	40%	26%
Offensive	10%	10%		31%	16%	13%	12%
Defensive	17%	48%	69%		55%	35%	32%
Mixed	29%	79%	84%	45%		49%	28%
Category	33%	60%	87%	65%	51%		38%
$\epsilon$ -greedy	48%	74%	88%	68%	72%	62%	

Board <i>b</i>	Rand.	Wall	Off.	Def.	Mixed	Cat.	$\epsilon$ -g.
Random		15%	100%	70%	56%	53%	58%
Wall	85%		99%	50%	78%	86%	88%
Offensive	0%	1%		0%	0%	0%	0%
Defensive	30%	50%	100%		77%	96%	51%
Mixed	44%	22%	100%	23%		37%	45%
Category	47%	14%	100%	4%	63%		36%
$\epsilon$ -greedy	42%	12%	100%	49%	55%	64%	

Board <i>c</i>	Random	Wall	Off.	Def.	Mixed	Cat.	$\epsilon$ -g.
Random		91%	98%	18%	43%	50%	49%
Wall	9%		81%	15%	7%	44%	9%
Offensive	2%	19%		9%	20%	12%	15%
Defensive	82%	85%	91%		76%	80%	90%
Mixed	57%	93%	80%	24%		40%	50%
Category	50%	56%	88%	20%	60%		57%
$\epsilon$ -greedy	51%	91%	85%	10%	50%	43%	

### 7.3 Play-out Cut-off

The play-out cut-off (PC) enhancement is tested by matching the MCTS-PC program against the MCTS-UCT program. MCTS-PC runs considerably fewer play-outs (25,000 per second on average) due to the computation time required by the play-out cut-off heuristic.

Table 5 shows the win rate for MCTS-PC against MCTS-UCT. At boards *a* and *b*, 800 games were run to ensure that the observed win rate was not influenced too much by statistical noise. 400 games were run for board *c*. The bad performance at board *c* might have to do with the difficulty for a player to isolate itself on this board.

### 7.4 Expansion Strategy Experiments

In the subsequent series of experiments are conducted for the predictive expansion (PDE) strategy. The MCTS-PDE program is tested against the MCTS-UCT program. The MCTS-PDE program ran 60,000 play-outs per second on average. The results are shown in Table 6. For each board, 600 games were run.

Similar to the results of the play-out cut-off experiment, the MCTS-PDE program appears to be slightly better than the MCTS-UCT program at board *a*

**Table 4.** Averaged play-out strategy results for all boards.

	Random	Wall	Off.	Def.	Mixed	Cat.	$\epsilon$ -g.
Random		55%	96%	57%	56%	57%	53%
Wall	45%		90%	39%	35%	57%	41%
Offensive	4%	10%		13%	12%	8%	9%
Defensive	43%	61%	87%		69%	71%	58%
Mixed	44%	65%	88%	31%		42%	41%
Category	43%	43%	92%	29%	58%		44%
$\epsilon$ -greedy	47%	59%	91%	42%	59%	56%	

**Table 5.** Win rates of MCTS-PC vs. MCTS-UCT.

	Board <i>a</i>	Board <i>b</i>	Board <i>c</i>	Total
Win	54%	56%	33%	$48 \pm 2\%$

and *b*. The poor win rate at board *c* is likely caused by the behaviour of MCTS for this board. The programs spiral around the centre, leaving the outer edges of the board open. Because the space estimation heuristic used by the predictive expansion strategy is only applicable when the players are isolated from each other, the MCTS-PDE program is squandering computation time on a mostly useless heuristic. Since the MCTS-UCT program spends all of its time on the play-outs, it can look further ahead and therefore has an advantage over the MCTS-PDE program.

### 7.5 MCTS-Solver Experiments

In this series of experiments the Score-Bounded MCTS-Solver is tested. The MCTS-Solver program ran at the same speed as the MCTS-UCT program. In the experiments of MCTS-Solver, MCTS-Solver-PDE and MCTS-Solver-PDE-PC, 400 games were run on each board.

As shown in Table 7, the MCTS-Solver program shows a slight improvement over MCTS-UCT. MCTS-Solver in combination with PDE or PDE-PC performs poorly at board *a* because it tends to cut off one side of the board, and due to lower number of simulations, cannot look ahead far enough to see the resulting outcome (i.e. loss). PDE and PDE-PC perform well at boards *b* and *c*, probably due to the obstacles and mistakes made by MCTS-UCT.

In terms of computation time and playing style, the MCTS-Solver program is the preferred choice since it requires no additional computations once a move has been proven to lead to a guaranteed win (or draw, when this is the best achievable outcome). Furthermore, the program can look up and play the shortest move sequence leading to a win by searching for the shortest winning path in the tree.

### 7.6 Playing against an $\alpha\beta$ program

In the final series of experiment, the MCTS program is tested against the winning program of the Tron Google AI Challenge, A1K0N [13]. The A1K0N program uses

**Table 6.** Win rates of MCTS-PDE vs. MCTS-UCT.

	Board <i>a</i>	Board <i>b</i>	Board <i>c</i>	Total
Win	53%	58%	48%	53 ± 2 %

**Table 7.** Win rates of MCTS-Solver variants against MCTS-UCT.

Win	Board <i>a</i>	Board <i>b</i>	Board <i>c</i>	Total
Solver	50%	52%	57%	53 ± 3 %
Solver-PDE	32%	74%	53%	53 ± 3 %
Solver-PDE-PC	30%	82%	70%	61 ± 3 %

$\alpha\beta$ -search [9] together with a good evaluation function that is primarily based on the tree of chambers heuristic.

**Table 8.** Win rates of various MCTS players against A1K0N.

	Board <i>a</i>	Board <i>b</i>	Board <i>c</i>	Total
MCTS-UCT	40%	0%	0%	14 ± 3 %
MCTS-PDE	44%	0%	0%	15 ± 3 %
Solver-PDE	13%	12%	0%	8 ± 3 %
Solver-PDE-PC	28%	10%	16%	18 ± 3 %
Solver-PDE-PC-PB	12%	18%	0%	10 ± 3 %

As can be seen in Table 8, A1K0N is the stronger player by far, achieving a win rate of 82% against MCTS-Solver-PDE-PC and a win rate higher than 85% against the other MCTS programs. Applying PB to MCTS-Solver does not seem to give an improvement in overall playing strength. Although the MCTS-Solver reaches a decent level of play, it still makes mistakes, mainly due to the fact that the reliability of the play-outs rapidly drops as the players get more distant from each other. By the time MCTS-Solver sees that it is in a bad position, it is already too late to correct.

## 8 Conclusion and Future Research

In this paper we developed an MCTS program for the game of Tron. Several enhancements were made to the selection, expansion and play-out phase. All of the enhancements were tested against an MCTS-UCT program.

The enhancement made to the selection phase, the progressive bias strategy, showed no improvement over UCT at two out of three boards. At board *c* the enhancement scored a consistent win rate of over 63%.

The experiments of the play-out strategies have shown that the board configuration has a large influence on the game and the effectiveness and accuracy of

the play-out strategies. The random-move strategy appeared to be the most robust choice, doing reasonably well on all three boards. The wall-following strategy outperformed the other strategies only at board *b*, whereas the defensive strategy outperformed the other strategies at board *c*.

Applying play-out cut-off showed an increase in playing strength at boards *a* and *b* (54% and 56%, respectively), but significantly decreased the playing strength at board *c*. The bad performance at board *c* may have to do with the difficulty for a player to isolate itself on this board.

Similar to the play-out cut-off enhancement, the predictive expansion strategy showed a slight increase in playing strength at board *a* (53%) and *b* (58%), but not at board *c*. The poor win rate at board *c* is likely caused by the behaviour of MCTS for this board. The MCTS programs keep a large space open behind them, up until late in the game. In such positions, the space estimation heuristic used is not helpful.

The Score-Bounded MCTS-Solver was tested against MCTS, and turned out to be an improvement, although MCTS-Solver with PDE performs poorly at board *a*, in comparison to MCTS-PDE. MCTS-Solver is preferred for its ability to look up the shortest path leading to a win (or draw, when a draw is the best achievable outcome) once one or more moves have been proven. In contrast, MCTS-UCT usually postpones the victory.

Using PDE on MCTS-Solver enables the program to prove a position more quickly, however, the extra time spent on computing the heuristic did not work out well for all boards. PDE and PC in combination with MCTS-Solver further increased the overall playing strength of the program. MCTS-Solver-PDE-PC achieves a surprisingly high win rate on board *c* (70%), where MCTS-PC only scored (33%).

The experiment involving the  $\alpha\beta$  program showed that the MCTS programs struggle at evaluating positions where the players are distant from one another (further than 10 steps away). Overall, the Solver-PDE-PC program is the best performing program, winning approximately 1 out of 5 games on average against the  $\alpha\beta$  program, and achieving a win rate of 61% against MCTS-UCT.

The experiments show that the board configuration has a large influence on the playing strength of the enhancements tested. Since our goal was to create a Tron program capable of playing on any  $13 \times 13$  map, the experiments should be conducted for many more boards.

As future research, applying a more sophisticated play-out strategy may increase the playing strength of MCTS in Tron. It would be interesting to see whether the play-out strategy and selection strategy can be improved such that MCTS can correctly look far ahead. The play-out phase might even have to be replaced completely by a sophisticated evaluation function (e.g. tree of chambers), as used by the  $\alpha\beta$  program A1K0N.

## References

1. B. Arneson, R.B. Hayward, and P. Henderson. Monte Carlo Tree Search in Hex. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):251–257,

- 2010.
2. P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time Analysis of the Multiarmed Bandit Problem. *Machine learning*, 47(2):235–256, 2002.
3. H.J. Berliner. The B\* Tree Search Algorithm: A Best-first Proof Procedure. *Artificial Intelligence*, 12(1):23–40, 1979.
4. T. Cazenave and A. Saffidine. Score Bounded Monte-Carlo Tree Search. In *Computers and Games (CG 2010)*, volume 6515 of *Lecture Notes in Computer Science (LNCS)*, pages 93–104, Berlin Heidelberg, Germany, 2011. Springer.
5. G.M.J-B. Chaslot, M.H.M. Winands, H.J. van den Herik, J.W.H.M. Uiterwijk, and B. Bouzy. Progressive Strategies for Monte Carlo Tree Search. *New Mathematics and Natural Computation*, 4(3):343–357, 2008.
6. R. Coulom. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In *Computers and Games (CG 2006)*, volume 4630 of *LNCS*, pages 72–83. Springer-Verlag, 2007.
7. Dmj. Survival Mode. <http://www.ai-contest.com/forums/viewtopic.php?p=1568>, 2010.
8. S. Gelly, Y. Wang, R. Munos, and O. Teytaud. Modification of UCT with Patterns in Monte-Carlo Go. Technical Report 6062, INRIA, 2006.
9. D.E. Knuth and R.W. Moore. An Analysis of Alpha-beta Pruning. *Artificial intelligence*, 6(4):293–326, 1975.
10. L. Kocsis and C. Szepesvári. Bandit Based Monte-Carlo Planning. In *Machine Learning: ECML 2006*, volume 4212 of *LNCS*, pages 282–293. Springer, 2006.
11. R.J. Lorentz. Amazons Discover Monte-Carlo. In *Computers and Games (CG 2008)*, volume 5131 of *LNCS*, pages 13–24. Springer, 2008.
12. S. Samothrakis, D. Robles, and S.M. Lucas. A UCT Agent for Tron: Initial Investigations. In *2010 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 365–371. IEEE, 2010.
13. Sloane, A. Google AI Challenge Post-mortem. <http://a1k0n.net/blah/archives/2010/03/>, Mar 2010.
14. N.R. Sturtevant. An analysis of uct in multi-player games. In *Computers and Games (CG 2008)*, volume 5131 of *LNCS*, pages 37–49. Springer, 2008.
15. R.S. Sutton and A.G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, USA, 1998.
16. N.G.P. Den Teuling. Monte-Carlo Tree Search for the Simultaneous Move Game Tron, 2011. B.Sc. thesis, Maastricht University.
17. Y. Tsuruoka, D. Yokoyama, and T. Chikayama. Game-tree search algorithm based on realization probability. *ICGA Journal*, 25(3):132–144, 2002.
18. University of Waterloo Computer Science Club. Google AI Challenge. <http://csclub.uwaterloo.ca/contest/>, Feb 2010.
19. M.H.M. Winands, Y. Björnsson, and J.-T. Saito. Monte Carlo Tree Search in Lines of Action. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):239–250, 2010.