Symbolic Classification of General Two-Player Games[⋆]

Stefan Edelkamp and Peter Kissmann

Technische Universität Dortmund, Fakultät für Informatik
Otto-Hahn-Str. 14, D-44227 Dortmund, Germany

Abstract. In this paper we present a new symbolic algorithm for the classification, i. e. the calculation of the rewards for both players in case of optimal play, of two-player games with general rewards according to the Game Description Language. We will show that it classifies all states using a linear number of images concerning the depth of the game graph. We also present an extension that uses this algorithm to create symbolic endgame databases and then performs UCT to find an estimate for the classification of the game.

1 Introduction

In General Game Playing (GGP), games are not known beforehand. Thus, for writing an algorithm to play or solve general games no specialized knowledge about them can be used. One language to describe such games is the Game Description Language (GDL) [11]. Since 2005, an annual GGP Competition [9] takes place, which was last won 2007 by Yngvi Björnsson's and Hilmar Finnsson's CADIAPLAYER [6]. GDL is designed for the description of general games of full information satisfying the restrictions to be finite, discrete, and deterministic.

When the games end, all players receive a certain reward. This is an integer value within $\{0, \ldots, 100\}$ with 0 being the worst and 100 the optimal reward. Thus, each player will try to get a reward as high as possible (and maybe at the same time keep the opponent's reward as low as possible). The description is based on the Knowledge Interchange Format (KIF) [8], which is a logic based language. It gives formulas for the initial state (*init*), the goal states (*terminal*), the preconditions for the moves (*legal*), further preconditions to get certain effects (*next*), the rewards in each terminal state for each player (*goal*) and some domain axioms (constant functions).

In this paper, we focus on non-simultaneous two-player games. Opposed to the competition's winners [4,13,6], we are not mainly interested in playing the games, but in classifying (solving) them. That is, we want to get the rewards for both players in case of optimal play (optimal rewards) for each of the states reachable from the initial state. Thus, when the classification is done, we can exploit the information to obtain a perfect player. Our implementation for the classification of these games originally worked with a variation of the Planning Domain Definition Language (PDDL) [12] that we called GDDL [5]. In the meantime, we implemented an instantiator for the KIF files, which results in a format quite close to instantiated PDDL but with *multi-actions* to represent the moves. These multi-actions consist of a global precondition, which is the disjunction

[⋆] Thanks to DFG for the financial support of the authors.

of all corresponding *legal* formulas, and several precondition/effect pairs where the preconditions can be arbitrary formulas while the effects are atoms. These pairs are the result of instantiating the *next* formulas and are interpreted as follows. A multi-action consisting of the global precondition *global* and the precondition/effect pairs pre_1, eff_1, $\ldots$, pre_n, eff_n might also be written as $\mathit{global} \wedge (\mathit{pre}_1 \Leftrightarrow \mathit{eff}_1) \wedge \ldots \wedge (\mathit{pre}_n \Leftrightarrow \mathit{eff}_n)$. If *global* holds in the current state, we can take this action. It will create the current state's successor by applying the precondition/effect pairs. An effect has to hold in the successor iff the corresponding precondition holds in the current state. All variables not in any of the effects are set to *false*.

The algorithm to classify two-player turn-taking games with general rewards proposed in [5] is quite inefficient. In this paper, we present a new one that is much faster. We also show how to adapt it to build endgame databases that can be used for the estimation of the optimal rewards of the reachable states using the UCT algorithm [10].

2 Symbolic Search for General Games

Symbolic search is concerned with checking the satisfiability of formulas. For this purpose, we use Binary Decision Diagrams (BDDs) [3], so that we work with state sets instead of single states. In many cases, this saves a lot of memory. E. g., we are able to calculate the complete set of reachable states for Connect Four. The precise number of states reachable from the initially empty board is $4{,}531{,}985{,}219{,}092$, compared to Allis's estimate of $70{,}728{,}639{,}995{,}483$ in [2]. In case of explicit search, for the same state encoding (two bits for each cell (player 1, player 2, empty) plus one for the current player, resulting in a total of 85 bits per state), nearly 43.8 terabytes would be necessary. When using BDDs, $84{,}088{,}763$ nodes suffice to represent all states.

In order to perform symbolic search, we need BDDs to represent the initial state (*init*), the goal states (*goal*), the transition relation (*trans*) and the conditions for each player p to achieve a specified reward r ($\mathit{reward}\,(p,r)$, $r \in \{0,\ldots,100\}$). We construct one BDD trans_a for each action a resulting in the transition relation $\mathit{trans} = \bigvee_a \mathit{trans}_a$. Furthermore, we need two sets of variables, one, S, for the predecessor states and the other, S', for the successor states.

To calculate the successors of a state set *state*, we build the (weak) *image*, defined as $\mathit{image}\,(\mathit{trans}, \mathit{state}) = \bigvee_a \exists S.\ \mathit{trans}_a\,(S,S') \wedge \mathit{state}\,(S)$. The predecessor calculation, the (weak) *pre-image*, works similar: $\mathit{preImage}\,(\mathit{trans}, \mathit{state}) = \bigvee_a \exists S'.\ \mathit{trans}_a\,(S,S') \wedge \mathit{state}\,(S')$. If we need all the states whose successors hold in *state*, we use the *strong pre-image*, which is defined as $\mathit{strongPreImage}\,(\mathit{trans}, \mathit{state}) = \bigwedge_a \forall S'.\ \mathit{trans}_a\,(S,S') \Rightarrow \mathit{state}\,(S') = \neg \mathit{preImage}\,(\mathit{trans}, \neg \mathit{state})$. Note that the image and pre-image calculations result in states in the other set of variables. Thereto, after each image or pre-image we perform a replacement of the variables to the ones used before.

3 The Algorithm UCT

In order to calculate an estimate for the current state often Monte-Carlo sampling can be applied. It performs a random search from the current state to a goal state and updates the estimated values of intermediate nodes.

The algorithm UCT [10] (*Upper Confidence Bounds applied to Trees*) is an algorithm to work for trees, such as game trees. Starting at the root of the tree, it initially works similar to Monte-Carlo search. If not all actions of the current node were performed at least once, it chooses an unused action randomly. Once this is done for all actions in a node (state s in depth d), it chooses the action that maximizes $Q_t(s, a, d) + c_{N_{s,d}(t),N_{s,a,d}(t)}$, with $Q_t(s, a, d)$ being the estimated value of action a in state s at depth d and time t, $N_{s,d}(t)$ the number of times s was reached up to time t in depth d, $N_{s,a,d}(t)$ the number of times action a was selected when s was visited in depth d up to time t and $c_{n_1,n_2} = 2C_p\sqrt{\ln(n_1)/n_2}$ with appropriate constant C_p. The value c_{n_1,n_2} trades off between exploration and exploitation in that actions with a low estimate will be taken again if the algorithm runs long enough: If an action is chosen, this factor will decrease as the number of times this action was chosen increases; for the actions not chosen, this factor will increase as the number of visits to this state increases.

UCT is successfully used in several game players. In [7], the authors show that in Go UCT outperforms classical α-β-search and give three reasons: UCT can be stopped at any time and still perform well, it is robust by automatically handling uncertainty in a smooth way and the tree grows asymmetrically.

4 Classification of General Two-Player Games

In [5], we proposed a first symbolic algorithm to classify general non-simultaneous two-player games, which has a bad runtime-behavior. In the following, we will present a greatly improved version and a symbolic variant of UCT that uses this algorithm to construct endgame databases.

4.1 Improved Symbolic Classification Algorithm

When classifying a game, we are interested in the optimal rewards of all reachable states. Thereto, we construct a game tree by symbolic breadth-first search (SBFS) with the initial state as its root and each action corresponding to an edge to a succeeding node. As the rewards are only applied in goal states, the internal nodes initially have no known rewards, whereas those of the leaves can be easily determined.

To determine the optimal rewards of each internal node, we propagate these values towards the root until all necessary nodes are classified. As we are concerned with two-player games, this propagation is quite simple. All states whose successors are already classified are given rewards (cf. Fig. 1) according to certain rules. This we repeat iteratively, until finally all states are classified.

Our symbolic algorithm works exactly like that. For it we need a 101×101 matrix (*bucket*) with one BDD for each theoretically possible reward combination. Furthermore, we need a BDD to represent all *classified* states, i. e. states that are in the matrix, a BDD *classifyable* to store those unclassified states whose successors are already classified, and a BDD for the remaining *unclassified* states.

The algorithm's pseudocode is shown in Algorithm 1. First, we determine all reachable states by performing SBFS. Next, we initialize the matrix by setting each possible *bucket* (i, j) to the conjunction of the reachable goal states achieving reward i for player

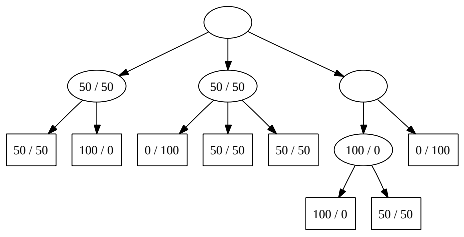

Fig. 1. Example of a game tree. The classification has performed one step.

Algorithm 1: Improved algorithm for general rewards (*classifyGame*).

```
**Input**: $\textit{reward}, \textit{goal}$
$\textit{reach} \leftarrow \textit{reachable}\,()$;
**forall** $i, j \in \{0, \ldots, 100\}$ **do**
  $\textit{bucket}\,(i, j) = \textit{reach} \wedge \textit{goal} \wedge \textit{reward}\,(0, i) \wedge \textit{reward}\,(1, j)$
$\textit{classified} \leftarrow \bigvee_{0 \leq i,j \leq 100} \textit{bucket}\,(i, j)$;
$\textit{unclassified} \leftarrow \textit{reach} \wedge \neg \textit{classified}$;
**while** $\textit{unclassified} \neq \perp$ **do**
  **foreach** $\textit{player} \in \{0, 1\}$ **do**
    $\textit{classifyable} \leftarrow \textit{strongPreImage}\,(\textit{trans}, \textit{classified}) \wedge \textit{unclassified} \wedge \textit{control}\,(\textit{player})$;
    **if** $\textit{classifyable} \neq \perp$ **then**
      $\textit{bucket} \leftarrow \textit{classifyStates}\,(\textit{classifyable}, \textit{player}, \textit{bucket})$;
      $\textit{classified} \leftarrow \textit{classified} \vee \textit{classifyable}$;
      $\textit{unclassified} \leftarrow \textit{unclassified} \wedge \neg \textit{classifyable}$;
```

0 and j for player 1. The *classified* states are the disjunction of all the states in the buckets and the *unclassified* ones the remaining reachable states.

While there are still unclassified states, we continue working on those states, whose successors are all classified. They are found by computing the strong pre-image. Of these states, we only take those where the current player has *control* and classify them. For this, we have two different functions, dependent on the way we wish to compare two states. Either we first maximize the reward for the current player and afterwards minimize that for the opponent, or we maximize the difference between the current player's and its opponent's reward. The classification procedure works as shown in Algorithm 2. When going through the buckets in the specified order, we calculate the states that have to be inserted in each bucket by calculating the corresponding predecessors for the classifyable states using the pre-image function. These states are inserted into this bucket and the classifyable states are updated by deleting the newly classifed ones. After each classification step, the classified states as well as the unclassified ones are updated by adding and deleting the newly classified states, respectively.

To predict the runtime of the algorithm, we count the number of pre-images. As the strong ones can be converted to the weak ones we make no difference between them. Let d be the depth of the inverse game graph, which equals the number of traversals through the main algorithm's loop. During each iteration we calculate two strong pre-images (one for each player). This results in at most $2 \times d$ strong pre-image calculations. In the classification algorithm, we calculate one pre-image for each possible pair of rewards. Let $r = |\{(i, j) \mid (\textit{reward}\,(0, i) \wedge \textit{reward}\,(1, j)) \neq \perp\}|$ be the number

Algorithm 2: Classification of the classifyable states (*classifyStates*).

```
Input: classifyable, player, bucket
if player $= 0$ then
  foreach $i, j \in \{0, \dots, 100\}$ do          // Go through the buckets in specific order.
    if bucket $(i, j) \neq \bot$ then
      newStates $\leftarrow$ preImage (trans, bucket $(i, j)$) $\wedge$ classifyable;
      bucket $(i, j) \leftarrow$ bucket $(i, j) \vee$ newStates;
      classifyable $\leftarrow$ classifyable $\wedge \neg$newStates;
      if classifyable $= \bot$ then break
else // The same for the other player, with swapped bucket indices.
return bucket;
```

of possible reward pairs. So, for classification we calculate r weak pre-images. In each iteration through the main algorithm's loop, we call the classification algorithm at most twice (once for each player). Thus, we get at most $2 \times d \times r$ weak pre-images and $2 \times d \times (1 + r)$ pre-images in total.

4.2 UCT with Symbolic Endgame Databases

The above algorithm is good for classifying all states of a game. But this total classification takes time – the calculation of the set of reachable states as well as the classification itself might take hours or days for complex games. If one is only interested in an estimate of the optimal rewards and has only a limited amount of time, the combination of the classification algorithm and UCT might be interesting. Thereto, we adapted the classification algorithm to create an endgame database and applied UCT to this.

To be able to create the endgame database as quickly as possible, it is best to omit the calculation of reachable states. This way, many states that are not reachable in forward direction will be created and classified, but the forward calculation drops out. In order to get better databases, one should calculate as many backward steps as possible, but if time is short, only few iterations should be performed.

In our UCT algorithm, we encode states as BDDs. Starting at the initial state, for each new node we first have to verify if it is in the endgame database (*classification*; cf. Algorithm 3). If it is, we get the state's value by subtracting the corresponding bucket's second index from the first to get the difference of the rewards. This value we set as the state's estimate, update its number of visits to 1, and return the value.

If the state is not in the database, we compute the applicable actions. Thereto, we build the conjunction of the current state with each action. Each applicable action is stored in a list along with the node. When reaching a node that was visited before, we verify if it has applicable actions. If it has not, it is a leaf node that is stored in the database and we only have to return its value. Otherwise, we further check if all actions have been applied at least once. If not, we randomly choose one of the unapplied actions and create its successor by calculating the image. If all actions have been used at least once, we take the action that maximizes the UCT value. In both cases we increment the number of applications of the action, update the estimate of this node, increment its number of visits, and return the value. For the calculation of the UCT value, we transform the estimated values into the interval $[0, 1]$ by adding 100 and dividing the result by 200, if it is the first player's move, or by -200, if it is the second player's.

Algorithm 3: UCT search function (*search*).

```
Input: state, classification
if $state.visits = 0$ then
  if $\exists x, y.\ (state.bdd \land classification\ (x, y) \neq \bot)$ then
    $state.visits \leftarrow 1$;
    $state.estimate \leftarrow x - y$ ;                    // maximize the difference
    return $state.estimate$;
  foreach $action \in trans$ do
    if $(state.bdd \land action) \neq \bot$ then  $state.actions \leftarrow state.actions \cup action$;
if $state.actions = \emptyset$ then  return $state.estimate$;
if $|\{a \mid a \in state.actions \land a.visits = 0\}| > 0$ then
  $action \leftarrow chooseActionRandomly\ (state.actions)$;
  $successor \leftarrow image\ (action, state.bdd)$;
else
  $action \leftarrow findActionWithBestUCT\ (state.actions)$;
  $successor \leftarrow getSuccessor\ (state, action)$;
$value \leftarrow search\ (classification, successor)$;
$action.visits \leftarrow action.visits + 1$;
$state.estimate \leftarrow (state.estimate \cdot state.visits + value)\ /\ (state.visits + 1)$;
$state.visits \leftarrow state.visits + 1$;
return $value$;
```

5 Experimental Results

We implemented the algorithms in Java using JavaBDD, which provides a native interface to Fabio Somenzi's CUDD package, and performed them on an AMD Opteron with 2.6 GHz and 16 GB RAM.

For *Clobber* [1], we set the reward for a victory depending on the number of moves taken. This way, we performed the classification for 3×4 and 4×5 boards that are initially filled with the pieces of both players alternating. While our older algorithm takes more than an hour for the 3×4 board, our improved version finishes after less than five seconds. The result is that the game is a victory for the second player with two pieces of each player remaining. On the 4×5 board, nearly 26.8 million states are reachable, while less than half a million BDD nodes suffice to represent them. The older algorithm cannot classify the game within 20 days, whereas the improved one is able to classify the reachable states within 2 hours and 15 minutes. This game is won for the first player with three own and two opponent pieces remaining.

With Clobber we also performed UCT with a timeout of one minute. Here, we compare the estimates for the initial state for different endgame database sizes (number of backward steps). Fig. 2 shows the difference of the estimates on the two players' rewards (higher values mean higher rewards for the first player). As can be seen, we get a solution much nearer to the optimal outcome of -40 after a shorter time with the larger databases. In fact, for databases that result from very few backward steps, after one minute the estimate for the initial state is far from the optimal one and thus might be misleading. Also, with the better databases the algorithm can perform more iterations.

The game of *Crisscross* is a two-player adaptation of Chinese Checkers and was one of the games used in the qualification phase of the 2006 competition. Our algorithm can classify it completely in about twelve seconds. Nearly $150{,}000$ states are reachable and

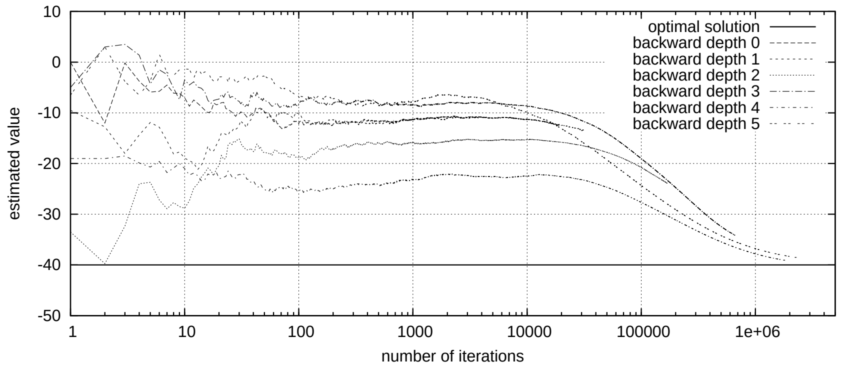

Fig. 2. Estimates for 3×4 Clobber with different endgame databases (averaged over 10 runs).

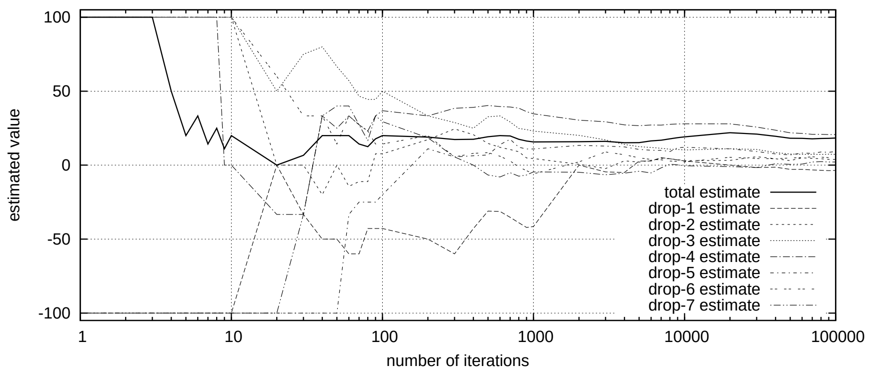

Fig. 3. The results for Connect Four.

the optimal outcome is $25/100$, i. e. both players can reach their goal positions but the second player is the first to reach it.

For *Connect Four* we performed UCT, as it takes too long (several hours) – and also too much RAM (more than 16 GB) – to be classified. Fig. 3 shows the results after about 2.5 hours. Nearly $100{,}000$ iterations were performed with a final estimate of 18.3 at the initial state (again the value gives the difference of the estimate on the two players' rewards, higher being better for the first player). As the game is won for the first player, the correct value is 100. In Fig. 3, we can see that the estimate for the optimal move, dropping the first piece in the middle column (*drop-4*), is the highest (20.6 vs. 9.3 for the second best). Furthermore, the estimates for the moves resulting in a draw (*drop-3* and *drop-5*) are slightly higher than those for the losing moves. After no more than 300 iterations, which are performed in less than ten seconds, this already is the case. We also found out that in the end the optimal move has been analyzed about ten times as often as the other possible moves, thus it has the most reliable estimate.

6 Conclusion

In this paper, we have proposed a new symbolic algorithm for the classification of general non-simultaneous two-player games. We have shown that a linear number (in the depth of the game graph) of pre-images suffices to classify such a game. Furthermore, we have presented a symbolic variant of UCT that uses our algorithm beforehand to construct endgame databases.

We have presented results for several games. For Clobber we performed the classification algorithm as well as the UCT algorithm, while for Crisscross we performed only the classification. In Connect Four, we could not finish the classification and so used the classification calculated so far as an endgame database for the UCT algorithm.

With the classification, we are able to check how well players did in real matches. To do this, we can start with the initial state and perform the actions the players chose during the matches. For the successor state we can find the bucket it is in, which gives the optimal reward achievable from this state. Whenever the bucket is changed, the current player has performed a suboptimal move. This might help analyzing the quality of existing players.

References

1. M. Albert, J. P. Grossman, R. J. Nowakowski, and D. Wolfe. An introduction to clobber. *INTEGERS: The Electronic Journal of Combinatorial Number Theory*, 5(2), 2005.
2. V. Allis. A knowledge-based approach of connect-four. Master's thesis, Vrije Universiteit Amsterdam, October 1988.
3. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.
4. J. Clune. Heuristic evaluation functions for general game playing. In *AAAI*, pages 1134–1139, 2007.
5. S. Edelkamp and P. Kissmann. Symbolic exploration for general game playing in PDDL. In *ICAPS-Workshop on Planning in Games*, 2007.
6. H. Finnsson and Y. Björnsson. Simulation-based approach to general game playing. In *AAAI*, pages 259–264, 2008.
7. S. Gelly and Y. Wang. Exploration exploitation in Go: UCT for Monte-Carlo Go. In *NIPS-Workshop on On-line Trading of Exploration and Exploitation*, 2006.
8. M. R. Genesereth and R. E. Fikes. Knowledge interchange format, version 3.0 reference manual. Technical Report Logic-92-1, Stanford University, 1992.
9. M. R. Genesereth, N. Love, and B. Pell. General game playing: Overview of the AAAI competition. *AI Magazine*, 26(2):62–72, 2005.
10. L. Kocsis and C. Szepesvári. Bandit based Monte-Carlo planning. In *ECML*, volume 4212 of *LNCS*, pages 282–293, 2006.
11. N. C. Love, T. L. Hinrichs, and M. R. Genesereth. General game playing: Game description language specification. Technical Report LG-2006-01, Stanford Logic Group, April 2006.
12. D. V. McDermott. The 1998 AI planning systems competition. *AI Magazine*, 21(2):35–55, 2000.
13. S. Schiffel and M. Thielscher. Fluxplayer: A successful general game player. In *AAAI*, pages 1191–1196, 2007.