Strongly Solving Fox-and-Geese on Multi-core CPU

Stefan Edelkamp and Hartmut Messerschmidt

TZI, Universität Bremen, Germany

Abstract. In this paper, we apply an efficient method of solving two-player combinatorial games by mapping each state to a unique bit in memory. In order to avoid collisions, such perfect hash functions serve as a compressed representation of the search space and support the execution of an exhaustive retrograde analysis on limited space. To overcome time limitations in solving the previously unsolved game *Fox-and-Geese*, we additionally utilize parallel computing power and obtain a linear speed-up in the number of CPU cores.

1 Introduction

Strong computer players for combinatorial games like *Chess* [2] have shown the impact of advanced AI search engines. For many games they play on expert and world championship level, sometimes even better. Some games like *Checkers* [7] have been decided, in the sense that the solvability status of the initial state has been computed.

In this paper we strongly solve *Fox-and-Geese* (*Fuchs-und-Gänse*[1]), a challenging two-player zero-sum game. To the authors knowledge, *Fox-and-Geese* has not been solved yet.

Fox-and-Geese belongs to the set of asymmetric strategy games played on a cross shaped board. The lone fox attempts to capture the geese, while the geese try to block the fox, so that it cannot move.

The first probable reference to an ancestor of the game is that of *Hala-Tafl*, which is mentioned in an Icelandic saga and which is believed to have been written in the 14th century. According to various Internet sources, the chances for 13 geese are assumed to be an advantage for the fox, while for 17 geese the chances are assumed to be roughly equal.

The game requires a strategic plan and tactical skills in certain battle situations[2]. The portions of tactic and strategy are not equal for both players, such that a novice often plays better with the fox than with the geese. A good fox detects weaknesses in the set of geese (unprotected ones, empty vertices, which are central to the area around) and moves actively towards them. Potential decoys, which try to lure the fox out of its burrow have to be captured early enough. The geese have to work together in form of a swarm and find a compromise between risk and safety. In the beginning it is recommended to choose safe moves, while to the end of the game it is recommended to challenge the fox to move out in order to fill blocked vertices.

[1] http://en.wikipedia.org/w/index.php?title=Fox_games&oldid=366500050

[2] An online game can be played at http://www.osv.org/kids_zone/foxgeese/index.html

R. Dillmann et al. (Eds.): KI 2010, LNAI 6359, pp. 291–298, 2010.

2 Preliminaries

The game Fox-and-Geese is played on a cross-shaped (peg solitaire) board consisting of a 3×3 square of intersections in the middle with four 2×3 areas adjacent to each face of the central square. One board with the initial layout is shown in Fig. 1. Pieces can move to any empty intersection around them (also diagonally). The fox can additionally jump over a goose to devour it. Geese cannot jump. The geese win if they surround the fox so that it can neither jump nor move. The fox wins if it captures enough geese that the remaining geese cannot block him.

Fig. 1. Initial State of the Two-Player Turn-Taking Game *Fox-and-Geese*

We consider *strongly solving* a game in the sense of creating an optimal player for every possible initial state. This is achieved by computing the game-theoretical value of each state, so that the best possible action can be selected by looking at all possible successor states. For two-player games the value is the best possible reward assuming that both players play optimally.

For analyzing the state space, we utilize a bitvector that covers the solvability information of all possible states. Moreover, we apply symmetries to reduce the time- and space-efficiency of our algorithm. Besides the design of efficient perfect hash functions that apply to a wide selection of games, one important contribution of the paper is to compute successors states on multiple CPU cores.

3 Bitvector State Space Search

Our approach is based on minimal perfect hashing [8]. Perfect hash functions are injective mappings of the set of reachable states to a set of available indices. They are invertible, if the state can be reconstructed given the index. A perfect hash function is minimal if it is bijective.

Ranking maps a state to a unique number, while *unranking* reconstructs a state given its rank. One application of the ranking and unranking functions is to compress and decompress a state.

3.1 Two-Bit Breadth-First Search

Cooperman and Finkelstein [3] showed that, given a perfect and invertible hash function, two bits per state are sufficient to perform a complete breadth-first exploration of the search space. Two-bit breadth-first search has first been used to enumerate so-called *Cayley Graphs* [3]. As a subsequent result the authors proved an upper bound to solve every possible configuration of *Rubik's Cube* [6]. By performing a breadth-first search over subsets of configurations in 63 hours together with the help of 128 processor cores and 7 TBs of disk space it was shown that 26 moves always suffice to unscramble it. More recently, Korf [4] has applied two-bit breadth-first search to generate the state spaces for hard instances of the *Pancake* problem I/O-efficiently, and, together with Breyer, he has extended the idea to construct compressed pattern databases having 1.6 bits [1].

In the two-bit breadth-first search algorithm every state is looked at once per layer. The two bits encode values in $\{0, \ldots, 3\}$ with value 3 representing an unvisited state, and values 0, 1, or 2 denoting the current search depth *mod* 3. This suffices to distinguish generated and visited states from ones expanded in the current breadth-first search layer.

3.2 Two-Bit Retrograde Analysis

Retrograde analysis classifies the entire set of positions in backward direction, starting from won and lost terminal ones. Partially completed retrograde analyses have been used in conjunction with forward-chaining game playing programs as endgame databases. Moreover, large endgame databases are usually constructed on disk for an increasing number of pieces. Since captures are non-invertible moves, a state to be classified refers only to successors that have the same number of pieces (and thus are in the same layer), and to ones that have a smaller number of pieces (often only one less).

The retrograde analysis algorithm works for all games, where the game positions can be divided into different layers, and the layers are ordered in such a way that movements are only possible within the same layer or from a higher layer to a lower one. Here each layer is defined by the number of geese on the game board. The rank and unrank functions are different for each layer. In the algorithm rank and unrank stands for the rank, unrank function for the given layer, while the suffix -smaller indicates, that the rank and unrank function belongs to a lower layer. Accordingly the expand-smaller function returns the successors in a lower layer, while expand-equal returns the successors in the same layer.

Retrograde analysis for zero-sum games requires 2 bits per state for executing the analysis on a bitvector representation of the search space: denoting if a state is unsolved, if it is a draw, if it is won for the first, or if it is won for the second player. Additional state information determines the player to move next.

Bit-state retrograde analysis applies backward BFS starting from the states that are already decided. Algorithm 1 shows an implementation of the retrograde analysis in pseudo code. For the sake of simplicity, in the implementation we look at two-player zero-sum games that have no draw. (For including draws, we would have to use the unused value 3, which shows, that two bits per state are still sufficient.) Based on the players' turn, the state space is in fact twice as large as the mere number of possible

game positions. The bits for the first player and the second player to move are interleaved, so that it can be distinguished by looking at the *mod 2* value of a state's rank.

Under these conditions it is sufficient to do the lookup in the lower layers only once during the computation of each layer. Thus the algorithm is divided into three parts. First an initialization of the layer (lines 4 to 8), here all positions that are won for one of the players are marked, a 1 stands for a victory of player one and a 2 for one of player two. Second a lookup of the successors in the lower layer (lines 9 - 18) is done, and at last an iteration over the remaining unclassified positions is done in lines 19 - 34. In the third part it is sufficient to consider only successors in the same layer.

In the second part, a position is marked won, if it has a successor that is won for the player to move, here (line 10) *even*(i) checks who is the active player. If there is no winning successor, the position remains unsolved. Even if all successors in the lower layer are lost, the position remains unsolved. A position is marked lost only in the third part of the algorithm, because not until then it is known how all successors are marked. If there are no successors in the third part, then the position is also marked as lost, because it has either only loosing successors in the lower layer, or no successor at all.

In the following it is shown that the algorithm indeed behaves as expected. The runtime is determined by the number of iterations that dictate work for the passes over the bitvector including ranking, unranking and lookups. A winning strategy means that one player can win from a given position no matter how the other player moves.

Theorem 1. *(Correctness) In Algorithm 1 a state is marked won, if and only if there exists a winning strategy for this state. A state is marked lost, if and only if it is either a winning situation for the opponent, or all successors are marked won for the opponent.*

Proof. The proof is done by induction over the length of the longest possible path, that is the maximal number of moves to a winning situation. As only two-player zero-sum games are considered, a game is lost for one player if it is won for the opponent, and, as the turns of both players alternate, the two statements must be shown together.

The algorithm marks a state with 1 if it assumes it is won for player one and with 2 if it assumes it is won for player two. Initially all positions with a winning situation are marked accordingly, therefore for all paths of length 0 it follows that a position is marked with 1, 2, if and only if it is won for player one, two, respectively. Thus for both directions of the proof the base of the induction holds.

The induction hypothesis for the first direction is as follows: for all non-final states x with a maximal path length of $n-1$ it follows that:

1. If x is marked as 1 and player one is the player to move, then there exists a winning strategy for player one.
2. If x is marked as 2 and player one is the player to move, then all successors of x are won for player two.
3. If x is marked as 2 and player two is the player to move, then there exists a winning strategy for player two.
4. If x is marked as 1 and player two is the player to move, then all successors of x are won for player one.

W.l.o.g. player one is the player to move; the cases for player two are analogously.

Algorithm 1. Two-Bit-Retrograde(m, *lost*, *won*)

```
 1: for all i := 0,...,m-1 do
 2:     Solved[i] := 0
 3: for all i := 0,...,m-1 do
 4:     if won(unrank(i)) then
 5:         Solved[i] := 1
 6:     if lost(unrank(i)) then
 7:         Solved[i] := 2
 8:     if Solved[i] = 0 then
 9:         succs-smaller := expand-smaller(unrank(i))
10:         if even(i) then
11:             for all s ∈ succs-smaller do
12:                 if Solved[rank-smaller(s)] = 2 then
13:                     Solved[i] := 2
14:         else
15:             for all s ∈ succs-smaller do
16:                 if Solved[rank-smaller(s)] = 1 then
17:                     Solved[i] := 1
18: while (Solved has changed) do
19:     for all i := 0,...,m-1 do
20:         if Solved[i] = 0 then
21:             succs-equal := expand-equal(unrank(i))
22:             if even(i) then
23:                 allone := true
24:                 for all s ∈ succs-equal do
25:                     if Solved[rank(s)] = 2 then
26:                         Solved[i] := 2
27:                     allone := allone & (Solved[rank(s)] = 1)
28:                 if allone then
29:                     Solved[i] := 1
30:             else
31:                 alltwo := true
32:                 for all s ∈ succs-equal do
33:                     if Solved[rank(s)] = 1 then
34:                         Solved[i] := 1
35:                     alltwo := alltwo & (Solved[rank(s)] = 2)
36:                 if alltwo then
37:                     Solved[i] := 2
```

Assume that x is marked as 1 and the maximal number of moves from position x are n. Then there exists a successor of x, say x', that is also marked as 1. There are two cases how a state can be marked as 1, x' is in a lower layer (lines 17,18) or in the same layer (lines 32,33). In both cases the maximal number of moves from x' is less than n, therefore with the induction hypothesis it follows that all successors of x' are won for player one, therefore there is a winning strategy for player one starting from state x.

Otherwise, if a state x is marked as 2 and the maximal number of moves from position x are n, then there is only one possible way how x was marked by the algorithm

(line 34), and it follows that all successors of x are marked with 2, too. Again it follows that there exists a winning strategy for all successors of x, and therefore they are won for player two. Together the assumption follows. The other direction is done quite similar, here from a winning strategy it follows almost immediately that a state is marked. For all paths of length less than n from a state x it follows that:

1. If there exists a winning strategy for player one and player one is the player to move, then x is marked as 1.
2. If all successors of x are won for player two and player one is the player to move, then x is marked as 2.
3. If there exists a winning strategy for player two and player two is the player to move, then x is marked as 2.
4. If all successors of x are won for player one and player two is the player to move, then x is marked as 1.

Assume that the maximal path length from a state x is n. If there exists a winning strategy for player one from x, then this strategy states a successor x' of x such that all successors of x' are won for player one, or x' is a winning situation for player one. In both cases it follows that x' is marked with 1 and therefore x is marked with 1 as well (line 17 or 34).

On the other hand, if all successors of x are won for player two. The successors in the lower layer do not effect the value of x because only winning successors change it (lines 16, 17). In line 31 alltwo is set to true and as long as there are only loosing successors which are marked with 2 by induction hypothesis, it stays true, and therefore x is marked with 2 in line 37, too.

4 Multi-core Parallelization

Modern computers have several cores, and efficient parallelizations reduce the run-time of algorithms by the distribution of the workload to concurrently running threads.

Let S_p be the set of all possible positions in *Fox-and-Geese* with p pieces on a board of size n. The arrangement of geese and the position of the fox together with the player's turn uniquely address a state in the game.

We can partition the state space S_p into disjoint sets $S_{p,0} \cup \ldots \cup S_{p,n-1}$, where the second index is the position of the fox. As $|S_{p,i}| \leq \binom{n-1}{p}$ for all $i \in \{0, \ldots, n-1\}$, an upper bound on the number of reachable states with p pieces is $n \cdot \binom{n-1}{p}$. These states are to be classified in parallel.

In parallel two-bit retrograde (BFS) analysis layers are processed in partition form with increasing number of geese. The fix point iteration to determine the solvability status in one BFS level is the most time consuming computation. Here, we apply a multi-core CPU parallelization. In each layer, n threads (one for each fox location) are created and joined after completion. They share the same hash function.

For improving the space consumption we urge the exploration to flush the sets $S_{p,i}$ whenever possible and to load only the ones needed for the current computation. In the retrograde analysis of *Fox-and-Geese* the access to positions with a smaller number of pieces S_{p-1} is only needed during the initialization phase. As such initialization is a

simple scan through a level we only need one set $S_{p,i}$ at a time. To save space for the fix point iteration, we release the memory needed to store the previous layer. As a result, the maximum number of bits needed is $\max_p\{|S_p|, |S_p|/n + |S_{p-1}|)$.

5 Experiments

Table 1 shows the results in strongly solving the game, where we applied parallel retrograde analysis for a increasing number of geese. By using this partitioning the entire analysis stayed in RAM. In fact, we observed that the exploration in the largest problem with 16 geese consumed main memory in the order of about 9.2 GB. The column Time real stands for the time and Time User is the sum of the CPU times.

Table 1. Retrograde Analysis Results for *Fox-and-Geese*

Geese	States	Space	Iterations	Won	Time Real	Time User
1	2,112	264 B	1	0	0.05s	0.08s
2	32,736	3.99 KB	6	0	0.55s	1.16s
3	327,360	39 KB	8	0	0.75s	2.99s
4	2,373,360	289 KB	11	40	6.73s	40.40s
5	13,290,816	1.58 MB	15	1,280	52.20s	6m24s
6	59,808,675	7.12 MB	17	21,380	4m37s	34m40s
7	222,146,996	26 MB	31	918,195	27m43s	208m19s
8	694,207,800	82 MB	32	6,381,436	99m45s	757m0s
9	1,851,200,800	220 MB	31	32,298,253	273m56s	2,083m20s
10	4,257,807,840	507 MB	46	130,237,402	1,006m52s	7,766m19s
11	8,515,615,680	1015 MB	137	633,387,266	5,933m13s	46,759m33s
12	14,902,327,440	1.73 GB	102	6,828,165,879	4,996m36s	36,375m09s
13	22,926,657,600	2.66 GB	89	10,069,015,679	5,400m13s	41,803m44s
14	31,114,749,600	3.62 GB	78	14,843,934,148	5,899m14s	45,426m42s
15	37,337,699,520	4.24 GB	73	18,301,131,418	5,749m6s	44,038m48s
16	39,671,305,740	4.61 GB	64	20,022,660,514	4,903m31s	37,394m1s
17	37,337,699,520	4.24 GB	57	19,475,378,171	3,833m26s	29,101m2s
18	31,114,749,600	3.62 GB	50	16,808,655,989	2,661m51s	20,098m3s
19	22,926,657,600	2.66 GB	45	12,885,372,114	1,621m41s	12,134m4s
20	14,902,327,440	1.73 GB	41	8,693,422,489	858m28s	6,342m50s
21	8,515,615,680	1015 MB	36	5,169,727,685	395m30s	2,889m45s
22	4,257,807,840	507 MB	31	2,695,418,693	158m41s	1,140m33s
23	1,851,200,800	220 MB	26	1,222,085,051	54m57	385m32s
24	694,207,800	82 MB	23	477,731,423	16m29s	112m.35s
25	222,146,996	26 MB	20	159,025,879	4m18s	28m42s
26	59,808,675	7.12 MB	17	44,865,396	55s	5m49s
27	13,290,816	1.58 MB	15	10,426,148	9.81s	56.15s
28	2,373,360	289 KB	12	1,948,134	1.59s	6.98s
29	327,360	39 KB	9	281,800	0.30s	0.55s
30	32,736	3.99 KB	6	28,347	0.02s	0.08s
31	2,112	264 B	5	2001	0.00s	0.06s

The first three levels do not contain any state won for the geese, which matches the fact that four geese are necessary to block the fox (at the middle border cell in each arm of the cross). We observe that after a while, the number of iterations shrinks for a raising number of geese. This matches the experience that with more geese it is easier to block the fox. Recall that all potential draws (that could not been proven won or lost by the geese) are devised to be a win for the fox. The transition, where the fox looses more than 50% of the game is reached at about 15–16 geese. This confirms practical observations that 13 geese are insufficient to win.

The total run-time of about a month for the experiment is considerable. Without multi-core parallelization, we estimated that more than 7 month would have been needed to complete the experiments. Even though we parallelized only the iteration stage of the algorithm, the speed-up on the 8-core machine is larger than 7, showing an almost linear speed-up.

The total space needed for operating our optimal player is about 34 GB, so that in case a goose is captured data is reloaded from disk.

6 Conclusion

In this work we applied linear-time ranking and unranking functions for games in order to execute retrograde analysis on sparse memory. We reflected that such constant-bit state space traversal to solve games is applicable, only if invertible and perfect hash functions are available. The approach features parallel explicit-state traversal of one challenging game on limited space. We studied the application of multiple-core computation and accelerated the analysis. In our experiments, the CPU speed-up is linear in the number of cores. For this we exploited independence in the problem, using an appropriate projection function. The speed-ups compare well with alternative results on parallel search on multiple cores, e.g. [5,9].

References

1. Breyer, T., Korf, R.E.: 1.6-bit pattern databases. In: AAAI (to appear 2010)
2. Campbell, M., Hoane Jr., A.J., Hsu, F.: Deep blue. Artificial Intelligence 134(1-2), 57–83 (2002)
3. Cooperman, G., Finkelstein, L.: New methods for using Cayley graphs in interconnection networks. Discrete Applied Mathematics 37/38, 95–118 (1992)
4. Korf, R.E.: Minimizing disk I/O in two-bit-breath-first search. In: National Conference on Artificial Intelligence (AAAI), pp. 317–324 (2008)
5. Korf, R.E., Schultze, T.: Large-scale parallel breadth-first search. In: National Conference on Artificial Intelligence (AAAI), pp. 1380–1385 (2005)
6. Kunkle, D., Cooperman, G.: Twenty-six moves suffice for Rubik's cube. In: International Symposium on Symbolic and Algebraic Computation (ISSAC), pp. 235–242 (2007)
7. Schaeffer, J., Björnsson, Y., Burch, N., Kishimoto, A., Müller, M.: Solving checkers. In: International Joint Conference on Artificial Intelligence (IJCAI), pp. 292–297 (2005)
8. Sulewski, D., Edelkamp, S., Yücel, C.: Perfect hashing for state space search on the GPU. In: International Conference on Automated Planning and Scheduling, ICAPS (2010)
9. Zhou, R., Hansen, E.A.: Parallel structured duplicate detection. In: National Conference on Artificial Intelligence (AAAI), pp. 1217–1222 (2007)