Technical Report 52

Solving Games in Parallel with Linear-Time Perfect Hash Functions

Stefan Edelkamp
Hartmut Messerschmidt
Damian Sulewski
TZI, Universität Bremen

Cengizhan Yücel
Technische Universität Dortmund

TZI-Bericht Nr. 52
2009

TZI-Berichte

Herausgeber:
Technologie-Zentrum Informatik
Universität Bremen
Am Fallturm 1
28359 Bremen
Telefon: +49-421-218-7272
Fax: +49-421-218-7820
E-Mail: info@tzi.de
http://www.tzi.de

ISSN 1613-3773

Solving Games in Parallel with Linear-Time Perfect Hash Functions

Stefan Edelkamp, Hartmut Messerschmidt, Damian Sulewski
TZI, Universität Bremen, Germany

Cengizhan Yücel
Technische Universität Dortmund, Germany

July 24, 2009

Abstract

In this paper, we propose an efficient method of solving one- and two-player combinatorial games by mapping each state to a unique bit in memory.

In order to avoid collisions, a concise portfolio of perfect hash functions is provided. Such perfect hash functions then address tables that serve as a compressed representation of the search space and support the execution of exhaustive search algorithms like breadth-first search and retrograde analysis.

Perfect hashing computes the *rank* of a state, while the inverse operation *unrank* reconstructs the state given its rank. Efficient algorithms are derived, studied in detail and generalized to a larger variety of games. We study rank and unrank functions for permutation games with distinguishable pieces, for selection games with indistinguishable pieces, and for general reachability sets. The running time for ranking and unranking in all three cases is linear in the size of the state vector.

To overcome space and time limitations in solving previously unsolved games like *Frogs-and-Toads* and *Fox-and-Geese*, we utilize parallel computing power in form of multiple cores as available on modern central processing units (CPUs) and graphics processing units (GPUs). We obtain an almost linear speedup with the number of CPU cores. Due to its much larger number of cores, even better absolute speed-up are achieved on the GPU.

Contents

1 Introduction **5**

2 Preliminaries **6**

3 Bitvector State Space Search **8**
3.1 Two-Bit Breadth-First Search 8
3.2 One-Bit Reachability 9
3.3 One-Bit Breadth-First Search 10
3.4 Two-Bit Retrograde Analysis 10

4 Hashing Permutation Games **14**
4.1 Efficient Ranking and Unranking 14
4.2 Sliding-Tile Puzzle 16
4.3 Top-Spin Puzzle 18
4.4 Pancake Problem 19

5 Hashing Selection Games **20**
5.1 Hashing with Binomial Coefficients 22
5.2 Hashing with Multinomial Coefficients 23

6 Parallelization **24**
6.1 Multi-Core Computation 25
6.2 GPU Computation 26

7 Experiments **27**
7.1 Permutation Games 28
7.2 Selection Games 29

8 Discussion **32**
8.1 Symmetries 35
8.2 Frontier Search 35
8.3 Pattern Databases 36
8.4 Other Games 37
8.4.1 Permutation Games 37
8.4.2 Selection Games 37
8.5 General Games 38

9 Conclusion **39**

1 Introduction

Strong computer players for combinatorial games like *Chess* [6] 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* [26] have been decided, in the sense that the solvability status of the initial state has been computed. (The game is a draw, assuming optimal play of both players.)

In this paper we consider *solving* a game in the sense of creating an optimal player for *every* possible initial state in the input. 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. In single-agent games the value of a game simply is its goal distance, while for two-player games the value is the best possible reward assuming that both players play optimally.

Our approach is based on perfect hashing, where a perfect hash function is a one-to-one mapping from a set of states to some set $\{0, \ldots, m-1\}$ for a sufficiently small number m. *Ranking* maps a state to a number, while *unranking* reconstructs a state given its rank. One application of ranking and unranking functions is to compress and decompress a state.

Provided the state space on disk, minimum perfect hash functions with a few bits per state can be constructed I/O efficiently. Botelho et al. [4] devise minimal practical hash functions for general state spaces, once the set of reachable states is known. The approach requires some small constant number c of bits per state (typically, $c \approx 4$). Of course, perfect hash functions do not have to be minimal to be space-efficient. Non-minimal hash functions can outperform minimal ones, since the gain in the constant number c of bits per state for the hash function can become smaller than the loss in coverage.

We will see that for many search problems space-efficient perfect hash functions can be constructed prior to executing the search. Sometimes it is even possible to devise a family of perfect hash functions, one for each (forward or backward) search layer. We propose linear time algorithms for invertible perfect hashing for a wide selection of AI search problems, including

- *permutation games*, i.e., games with distinguishable objects. In this class we find *Sliding-Tile* puzzles with numbered tiles, as well as *Top-Spin* and *Pancake* problems. The *parity* of a permutation will prove to be an important concept as it often allows to restrict the range of the hash function to half.

- *selection games*, i.e., games with indistinguishable objects. In this class we find tile games like *Frogs-and-Trouts*, as well as strategic games like *Peg-Solitaire* and *Fox-and-Geese*.

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-efficiencies of our algorithms. 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 cores on the central processing unit (located on the motherboard) and on the graphics processing unit (located on the graphics card).

This paper extends observations made in [14], where only permutation games have been considered, to a wider selection of state space problems. To the best of our knowledge, the only

attempts to use state space search on GPUs was by the authors of this paper in the context of model checking [13, 3]. In [13] they improved large-scale disk-based model checking by shifting complex numerical operations to the graphic card. As delayed elimination of duplicates is the performance bottleneck, the authors performed parallel processing on the GPU to improve the sorting speed significantly. Since existing GPU sorting solutions like Bitonic Sort and Quicksort do not obey any speed-up on state vectors, they propose a refined GPU-based Bucket-Sort algorithm. In [3] algorithms for parallel probabilistic model checking on GPUs were proposed. For this purpose the authors exploit the fact that some of the basic algorithms for probabilistic model checking rely on matrix vector multiplication. Since this kind of linear algebraic operations are implemented very efficiently on GPUs, the new parallel algorithms achieve considerable runtime improvements compared to their counterparts on standard architectures.

The paper is structured as follows. First, we provide preliminaries on perfect, invertible, orthogonal hash functions, on permutation parity and move alternation properties. Then we study constant-bit breadth-first search and constant-bit retrograde analysis. We discuss time-space trade-offs using only one bit per state. Next, we address the design of efficient rank and unrank functions for games with distinguishable and indistinguishable pieces, and adapt the algorithms to a series of state space problems. In order to improve the running time for solving the problems, we parallelize the classification algorithms. We show how to generate successors and how to rank and unrank states on multiple CPU and GPU cores. We provide experimental data for solving the games, discuss the results, and give some final concluding remarks.

2 Preliminaries

In the following, we formalize different characteristics of hash functions.

Definition 1 (Hash Function) *A* hash function h *is a mapping of a universe* U *to an index set* $\{0, \ldots, m-1\}$.

The set of reachable states S of a search problem is a subset of U, i.e., $S \subseteq U$. We are interested in hash functions that are injective[1].

Definition 2 (Perfect Hash Function) *A hash function* $h : S \to \{0, \ldots, m-1\}$ *is* perfect*, if for all* $s \in S$ *with* $h(s) = h(s')$ *we have* $s = s'$.

Definition 3 (Space Efficiency) *The space efficiency of a hash function* $h : S \to \{0, \ldots, m-1\}$ *is the proportion* $m/|S|$ *of available hash values to states.*

Given that every state can be viewed as a bitvector and interpreted as a number, one inefficient design of a perfect hash function is immediate. The space requirements of the corresponding hash table are usually too large. An space-optimal perfect hash function is bijective.

[1] A mapping is injective, if for all $f(x) = f(y)$ we have $x = y$.

Definition 4 (Minimal Perfect Hash Function) *A perfect hash function is* minimal *if its space efficiency is equal to* 1*, i.e., if* $m = |S|$.

Efficient and minimal perfect hash functions allow direct-addressing a bit-state hash table instead of mapping states to an open-addressed or chained hash table. The computed index of the direct access table uniquely identifies the state.

Whenever the averaged number of required bits per state for a perfect hash function is smaller than the number of bits in the state encoding, an implicit representation of the search space is fortunate, assuming that no other tricks like orthogonal hashing are applied.

Definition 5 (Orthogonal Hash Functions) *Two hash functions* h_1 *and* h_2 *are* orthogonal*, if for all states* s, s' *with* $h_1(s) = h_1(s')$ *and* $h_2(s) = h_2(s')$ *we have* $s = s'$.

In case of orthogonal hash functions h_1 and h_2, the value of h_1 can, e.g., be encoded in the file name, leading to a partitioned layout of the search space, and a smaller hash value h_2 to be stored explicitly.

Theorem 1 (Orthogonal Hashing imply Perfect One) *If the two hash functions* $h_1 : S \to \{0, \ldots, m_1 - 1\}$ *and* $h_2 : S \to \{0, \ldots, m_2 - 1\}$ *are orthogonal, their concatenation* (h_1, h_2) *is perfect.*

Proof. We start with two hash functions h_1 and h_2. Let s be any state in S. Given $(h_1(s), h_2(s)) = (h'_1(s), h'_2(s))$ we have $h_1(s) = h_1(s')$ and $h_2(s) = h_2(s')$. Since h_1 and h_2 are orthogonal, this implies $s_1 = s_2$. □

The other important property of a perfect hash function for a state space search is that the state vector can be reconstructed given the hash value.

Definition 6 (Invertible Hash Function) *A perfect hash function* h *is* invertible*, if given* $h(s)$*,* $s \in S$ *can be reconstructed. The inverse* h^{-1} *of* h *is a mapping from* $\{0, \ldots, m - 1\}$ *to* S*. Computing the hash value is denoted as* ranking*, while reconstructing a state given its rank is denoted as* unranking.

For the exploration of the search space, in which array indices serve as state descriptors, invertible hash functions are required.

For the design of minimal perfect hash functions in permutation games, parity will be a helpful concept.

Definition 7 (Inversion) *An* inversion *in a permutation* $\pi = (\pi_1, \ldots, \pi_n)$ *is a pair* (i, j) *with* $1 \leq i < j \leq n$ *and* $\pi_i > \pi_j$.

Definition 8 (Parity) *The* parity *of the permutation* π *is defined as the parity (*mod 2 *value) of the number of inversions in* π*,*

Definition 9 (Parity Preservation) *A permutation game is* parity-preserving*, if all moves preserve the parity of the permutation.*

Parity-preservation allows to separate solvable from insolvable states in several permutation games. If the parity is preserved, the state space can be compressed. For this case we have $|S| = n!/2$.

Definition 10 (Move Alternation Property) *A property $p : S \to \mathbb{N}$ is* move-alternating*, if the parity of p toggles for all actions, i.e., for all s and $s' \in \text{succs}(s)$ we have*

$$p(s') \bmod 2 = (p(s) + 1) \bmod 2.$$

As a result, $p(s)$ is the same for all states s in one BFS layer. In a mixed representation of two subsequent layers, states s' in the next BFS layer can be separated by knowing $p(s') \neq p(s)$.

One example for a move-alternation property is the position of the blank in the Sliding-Tile puzzle. Moreover, many pattern database heuristics [9] have the property either to increase or to decrease by 1 with each applied action.

3 Bitvector State Space Search

As indicated above, 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. Cooperman and Finkelstein [8] 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.

3.1 Two-Bit Breadth-First Search

Two-bit breadth-first has first been used to enumerate so-called *Cayley Graphs* [8]. As a subsequent result the authors proved an upper bound to solve every possible configuration of *Rubik's Cube* [23]. By performing a breadth-first search over subsets of configurations in 63 hours together with the help of 128 processor cores and 7 Tera bytes of disk space it was shown that 26 moves always suffice to rescramble it. Korf [20] has applied the two-bit breadth-first search algorithm to generate the state spaces for hard instances of the *Pancake* problem I/O-efficiently.

In the two-bit breadth-first search algorithm (shown in Algorithm 1) every state is expanded at most once. 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 allows to distinguish generated and visited states from ones expanded in the current breadth-first layer.

The running time is determined by the size of the search space times the maximum breadth-first search layer (times the efforts to generate the children).

Algorithm 1 Two-Bit-Breadth-First-Search(m, *init*)

```
1: for all i := 0, ..., m − 1 do
2:    BFS-Layer[i] := 3
3: BFS-Layer[rank(init)] := level := 0
4: while BFS-Layer has changed do
5:    level := level + 1
6:    for all i := 0, ..., m − 1 do
7:       if BFS-Layer[i] = (level − 1) mod 3 then
8:          succs := expand(unrank(i))
9:          for all s ∈ succs do
10:            if BFS-Layer[rank(s)] = 3 then
11:               BFS-Layer[rank(s)] := level mod 3
```

Algorithm 2 One-Bit-Reachability (m, *init*)

```
1: for all i := 0, ..., m − 1 do
2:    Open[i] := false
3: Open[rank(init)] = true
4: while Open has changed do
5:    i := 0, ..., m − 1
6:    if Open[i] = true then
7:       succs := expand(unrank(i))
8:       for all s ∈ succs do
9:          Open[rank(i)] := true
```

3.2 One-Bit Reachability

Are two bits the best possible compaction for computing the set of all reachable states? Yes and no. The procedure shown in Algorithm 2 illustrates that it is possible to generate the entire state space using one bit per state. However, as it does not distinguish between states to be expanded next (open states) and states already expanded (closed states), the algorithm may expand a state multiple times. Nonetheless, the algorithm is able to determine reachable states. Additional information extracted from a state can improve the running time by decreasing the number of states to be reopened.

If the successor's rank is smaller than the rank of the actual one, it will be expanded in the next scan, otherwise in the same. This observation leads to the following result.

Theorem 2 (Number of Scans in One-Bit Reachability) *The number of scans in the algorithm* One-Bit-Reachability *is bounded by the maximum number of BFS layers.*

Proof. Let *Layer*(i) be the BFS-layer of a state with rank i and *Scan*(i) be the layer in the algorithm *One-Bit-Reachability*. Evidently, *Scan*(*rank*(*init*)) = *Layer*(*rank*(*init*)) = 0. For any path $(s_0, \ldots, s_d)$ generated by BFS, we have *Scan*(*rank*(s_{d-1})) $\leq$ *Layer*(*rank*(s_{d-1})) by induction hypothesis. All successors of s_{d-1} are generated in the same iteration (if their index

value is larger than i) or in the next iteration (if their index value is smaller than i) such that $\mathit{Scan}(\mathit{rank}(s_d)) \leq \mathit{Layer}(\mathit{rank}(s_d))$. □

3.3 One-Bit Breadth-First Search

For some domains, one bit per state suffices for performing breadth-first search [14]. In *Peg-Solitaire*, the number of remaining pegs uniquely determine the breadth-first search layer, so that one bit per state suffices to distinguish newly generated states from expanded one. This saves space compared to the more general two-bit breadth-first search routine.

In the event of a move-alternation property *alternation*, we, therefore, can perform breadth-first search using only one bit per state.

Algorithm 3 One-Bit-Breadth-First-Search (m, *init*)

```
 1: for i = 0, ..., m − 1 do
 2:    Open[i] := false
 3: Open[rank(init)] := true
 4: level := 0
 5: while Open has changed do
 6:    for all i with Open[i] = true do
 7:       s := unrank(i)
 8:       if alternation(s) = level mod 2 then
 9:          succs := expand(unrank(i))
10:          for all s' ∈ succs do
11:             Open[rank(s')] := true
12:    level = level + 1
```

One important observation is that not all visited states that appear in previous BFS layers are removed from the current search layer. So there are states that are reopened, in the worst case once for each BFS layer. Even though some states may be expanded several times, the following result is immediate.

Theorem 3 (Population Count One-Bit-BFS) *Let the population count pc_l be the number of bits set after the l-th scan in Algorithm* One-Bit-BFS. *Then the number of states in BFS-level l is* $|\mathit{Layer}_l| = pc_l - pc_{l-1}$.

If we were able to store the set of reached states on disk, we could subtract the set of reached states. This, however, would imply that the algorithm no longer consumes one bit per state.

3.4 Two-Bit Retrograde Analysis

Retrograde analysis classifies the entire set of positions in backward direction, starting from won and lost terminal ones. Moreover, partially completed retrograde analyses have been used in conjunction with forward-chaining game playing programs to serve as endgame databases.

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 with this property. In detail: 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 in between a layer or from a higher layer to a lower one.

Additional state information indicating the player to move, 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.

Bit-state retrograde analysis applies backward BFS starting from the states that are already decided. Algorithm 4 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 this 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 file.

In the second part a position is marked as 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 as 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 it is asserted. A winning strategy means that one player can win from a given position no matter how the other player moves.

Theorem 4 *A state is marked as won if and only if there exists a winning strategy for this state.*

A state is marked as lost if and only if it is either a winning situation for the opponent, or all successors are marked as won for the opponent.

Proof. The proof is done with 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

Algorithm 4 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(rank(i)) then
5:       Solved[i] := 1
6:    if lost(rank(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[rank(i)] := 2
14:      else
15:         for all s ∈ succs-smaller do
16:            if Solved[rank-smaller(s)] = 1 then
17:               Solved[rank(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[rank(i)] := 2
27:                allone := allone & (Solved[rank(s)] = 1)
28:             if allone then
29:                Solved[rank(i)] := 1
30:          else
31:             alltwo := true
32:             for all s ∈ succs-equal do
33:                if Solved[rank(s)] = 1 then
34:                   Solved[rank(i)] := 1
35:                alltwo := alltwo & (Solved[rank(s)] = 2)
36:             if alltwo then
37:                Solved[rank(i)] := 2
```

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.

Without loss of generality player one is the player to move, the cases for player two are done accordingly.

So 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 with the induction hypothesis 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. □

The algorithm and the theorem can be extended to games with draws by only slightly modifying them, as mentioned above the value 3 can be used to indicate a draw. The problem with a draw is, that it depends on the game how and when it is a draw. Also note that this theorem does not show, that the algorithm always marks all states, because in certain games it is possible to have infinite sequences of moves, and different games have different conditions for this infiniteness; Some leading to draws or preventing cycles by demanding that no game position may occur twice. For these special situations the so called history problem needs to be solved for each game individually.

4 Hashing Permutation Games

In the sequel of this paper, we study efficient perfect hash functions for fast ranking and unranking. We will also look at hash functions that are adapted to the BFS layer.

4.1 Efficient Ranking and Unranking

For ranking and unranking permutations, time- and space-efficient algorithms have already been designed [2].

Definition 11 (Natural or Lexicographic Rank) *The natural or lexicographic rank of a permutation is the position in the lexicographic order of its state vector representation. In the lexicographic ordering of a permutation* $\pi = (\pi_0, \ldots, \pi_{n-1})$ *of* $\{0, \ldots, n-1\}$ *we first have* $(n!-1)$ *permutations that begin with* 0*, followed by* $(n!-1)$ *permutations that begin with* 1*, etc. Threfore, we have*

$$\pi_0 \cdot (n-1)! \leq \text{lex-rank}(\pi, n) \leq \pi_0 \cdot (n-1)!.$$

This leads to the following recursive formula: $\text{lex-rank}((0), 1) = 0$ *and*

$$\text{lex-rank}(\pi, n) \leq \pi_0 \cdot (n-1)! + \text{lex-rank}(\pi', n-1),$$

where $\pi'_i = \pi_{i+1}$ *if* $\pi'_i > \pi_0$ *and* $\pi'_i = \pi_i$ *if* $\pi'_i < \pi_0$.

The following result is widely known [2].

Theorem 5 (Inverted Index, Factorial Base) *The lexicographic rank of permutation π (of size n) is determined as lex-rank$(\pi, n) = \sum_{i=0}^{N-1} d_i \cdot (N-1-i)!$ where the vector d of coefficients d_i is called the* inverted index *or* factorial base*. The coefficients d_i are uniquely determined. The parity of a permutation is known to match $(\sum_{i=0}^{N-1} d_i) \bmod 2$.*

In the recursive definition of *lex-rank* the derivation of π' from π creates a burden that makes an according ranking algorithm non-linear. There have been many attempts, e.g. by Trotter and Johnson's minmal-exchange approach, which still have a non-linear time complexity in the worst-case [22].

Korf and Schultze [21] use two lookup tables with a space requirement of $O(2^n \log n)$ bits to compute lexicographic ranks in linear time. More crucially, given that larger tables do not fit into SRAM, the algorithms does not work well on the GPU. Bonet [2] discusses time-space trade-offs and provides a uniform algorithm that takes $O(n \log n)$ time and $O(n)$ space. Algorithms that are linear in time and space for both operations are not known.

Given that existing ranking and unranking algorithms wrt. the lexicographic ordering are rather slow in particular if executed on the graphics card, next we have a detailed look at the more efficient ordering of Myrvold and Ruskey [24]. They devise another ordering based on the observation that every permutation can be generated uniformly by swapping an element at position i with a randomly selected element $j > i$, while i continously increases. The sequence of j's can be seen as the equivalent to the factorial base for the lexicographic rank.

We show that the parity of a permuation can be derived on-the-fly in the unranking algorithm proposed by Myrvold and Ruskey[2]. For fast execution on the graphics card, we additionally avoid recursion.

The ranking algorithm is shown in Algorithm 5. The input is the number of elements N to permute, the permutation π, and its inverse permutation π^{-1}. The output is the rank of π. As a side effect, we have that both π and π^{-1} are modified. The unranking algorithm is shown in Alg. 6.

Algorithm 5 *rank*(n, π, π^{-1})

1: **for all** i **in** $\{1, \ldots, n-1\}$ **do**
2: $\quad l \leftarrow \pi_{n-i}$
3: $\quad$ *swap*$(\pi_{n-i}, \pi_{\pi^{-1}_{n-i}})$
4: $\quad$ *swap*$(\pi^{-1}_{l}, \pi^{-1}_{n-i})$
5: $\quad$ *rank*$_i \leftarrow l$
6: **return** $\prod_{i=1}^{n-1}$(*rank*$_{n-i+1} + i$)

Theorem 6 (Parity in Myrvold & Ruskey's Unrank Function) *The parity of a permutation for a rank r in Myrvold & Ruskey's permutation ordering can be computed on-the fly with the unrank function shown in Algorithm 6.*

[2]In all our results, we refer to Myrvold and Ruskey's rank1 and unrank1 functions.

Algorithm 6 *unrank*(r)

```
1: π := id
2: parity := false
3: while n > 0 do
4:    i := n − 1
5:    j := r mod n
6:    if i ≠ j then
7:       parity := ¬parity
8:       swap(π_i, π_j)
9:       r := r div n
10:   n := n − 1
11: return (parity, π)
```

Proof. In the *unrank* function swapping two elements u and v at position i and j, resp., with $i \neq j$ we count $2(j-i-1)+1$ transpositions (u and v are the elements to be swapped, x is a wildcard for any intermediate element): $uxx\ldots xxv \rightarrow xux\ldots xxv \rightarrow \ldots \rightarrow xx\ldots xxuv \rightarrow xx\ldots xxvu \rightarrow \ldots \rightarrow vxx\ldots xxu$. As $2(j-i-1)+1 \ \mathit{mod}\ 2 = 1$, each transposition either increases or decreases the parity of the number of inversions, so that the parity toggles for each iteration. The only exception is if $i = j$, where no change occurs. Hence, the parity of the permutation can be determined on-the-fly in our algorithm.

□

Theorem 7 (Folding Permutation Table in Myrvold & Ruskey's Approach) *Let $\pi(r)$ denote the permutation returned by Myrvold & Ruskey's* unrank *function given index r. Then $\pi(r)$ matches $\pi(r+n!/2)$ except for swapping π_0 and π_1.*

Proof. The last call to $\mathit{swap}(\pi_{n-1}, \pi_{r \ \mathit{mod}\ n})$ in Myrvold and Ruskey's *unrank* function is $\mathit{swap}(\pi_0, \pi_{r \ \mathit{mod}\ 1})$, which resolves to either $\mathit{swap}(\pi_1, \pi_1)$ or $\mathit{swap}(\pi_1, \pi_0)$. Only the latter one induces a change.

If $r_1, \ldots, r_{n-1}$ denote the indices of $r \ \mathit{mod}\ n$ in the iterations $1, \ldots, N-1$ of Myrvold and Ruskey's *unrank* function, then $r_{N-1} = \lfloor \ldots \lfloor r/(n-1) \rfloor \ldots /2 \rfloor$, which resolves to 1 for $r \geq n!/2$ and 0 for $r < n!/2$. □

4.2 Sliding-Tile Puzzle

Next, we consider permutation games, especially the ones shown in Fig. 1.

The $(n \times m)$ sliding-tile puzzle [17] consists of $(nm-1)$ numbered tiles and one empty position, called the blank. In many cases, the tiles are squarely arranged, such that $m = n$.

The task is to re-arrange the tiles such that a certain goal arrangement is reached. Swapping two tiles toggles the permutation parity and, in turn, the solvability status of the game. Thus, only half the $nm!$ states are reachable.

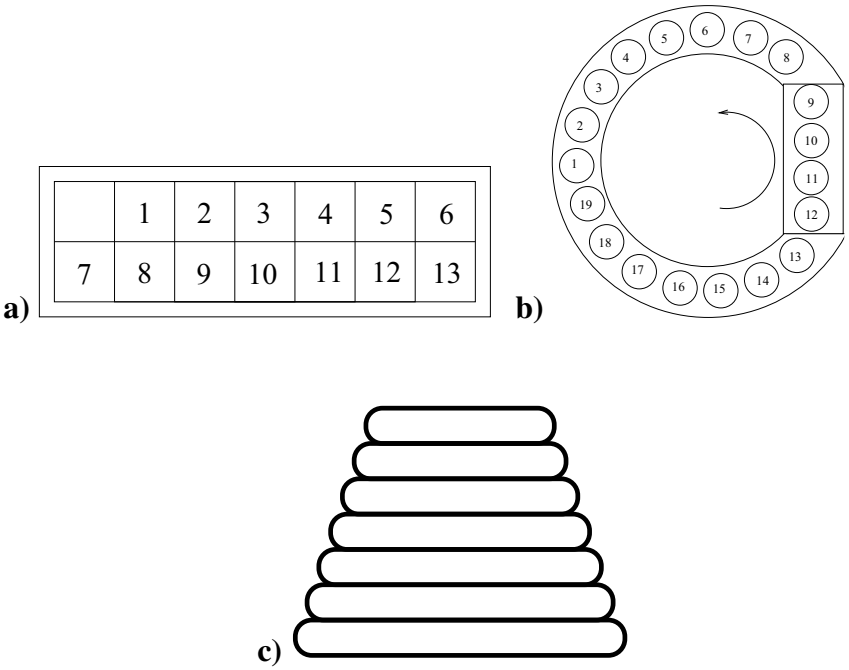

Figure 1: Permutation Games: a) Sliding-Tile Puzzle, b) Top-Spin Puzzle, c) Pancake Problem.

For the Sliding-Tile puzzle, we observe that in a lexicographic ordering every two adjacent permutations with lexicographic rank $2i$ and $2i + 1$ have a different solvability status. In order to hash a sliding-tile puzzle state to $\{0, \ldots, (nm)!/2 - 1\}$, we can, therefore, compute the lexicographic rank and divide it by 2. Unranking is slightly more complex, as it has to determine, which of the two permutations π_{2i} and π_{2i+1} of the puzzle with index i is actually reachable.

There is one subtle problem with the blank. Simply taking the parity of the entire board does not suffice to compute a minimum perfect hash value in $\{0, \ldots, nm!/2\}$, as swapping a tile with the blank is a move, which does not change the parity.

A solution to this problem (shown in Algorithm 7) is to partition the state space wrt. the position of the blank, since for exploring the $(n \times m)$ puzzle it is equivalent to enumerate all $(nm - 1)!/2$ orderings together with the nm positions of the blank. If $S_0, \ldots, S_{nm-1}$ denote the set of "blank-projected" partitions, then each set S_j, $j \in \{0, \ldots, nm - 1\}$ contains $(nm - 1)!/2$ states. Given the index i as the permutation rank and j it is simple to reconstruct the puzzle's state.

As a side effect of this partitioning, horizontal moves of the blank do not change the state vector, thus the rank remains the same. Tiles remain in the same order, preserving the rank.

Since the parity does not change in this puzzle we need another move alternating property, and find it in the position of the blank. The partition into buckets $S_0, \ldots, S_{nm-1}$ has the additional advantage that we can determine, whether the state belongs to an odd or even layer and which bucket a successor belongs to [27]. We observe that in puzzles with an odd number of columns at an even breadth-first level the blank position is even and at an odd breadth-first level the blank

Algorithm 7 One-Bit-Breath-First-Search-Sliding-Tile (*init*)

```
 1: for blank = 0, ..., nm − 1 do
 2:   for i = 0, ..., (nm − 1)!/2 − 1 do
 3:     Open[blank][i] := false
 4: Open[blank(init)][rank(init) mod (nm − 1)!/2] := true
 5: level := 0
 6: while Open has changed do
 7:   blank := level mod 2
 8:   while blank ≤ nm do
 9:     for all d ∈ {R, L, D, U} do
10:       dst := newblank(blank, d)
11:       if d ∈ {L, R} then
12:         Open[dst] := Open[dst] ∨ Open[blank]
13:       else
14:         for all i with Open[blank][i] = true do
15:           (valid, π) := unrank(i)
16:           if ¬valid then
17:             swap(π_0, π_1)
18:           succ := expand(π, d)
19:           r := rank(succ) mod (N − 1)!/2
20:           Open[dst][r] := true
21:     blank = blank + 2
22:   level = level + 1
```

position is odd.

For such a factored representation of the sliding-tile puzzles, a refined exploration in Algorithm 3 retains the breadth-first order, by means that a bit for a node is set for the first time in its BFS layer. The bitvector *Open* is partitioned into nm parts, which are expanded depending on the breadth-first *level* (line 7).

As mentioned above, the rank of a permutation does not change by a horizontal move of the blank. This is exploited in line 11 writing the ranks directly to the destination bucket using a bitwise-or on the bitvector from layer *level* $-$ 2 and *level*. The vertical moves are unranked, moved and ranked from line 13 onwards. When a bucket is done, the next one is skipped and the next but one is expanded. The algorithm terminates when no new successor is generated.

4.3 Top-Spin Puzzle

The next example is the (n, k)-Top-Spin Puzzle [7], which has n tokens in a ring. In one twist action k consecutive tokens are reversed and in one slide action pieces are shifted around. There are $n!$ different possible ways to permute the tokens into the locations. However, since the puzzle is cyclic only the order of the different tokens matters and thus there are only $(n-1)!$ different states in practice. After each of the n possible actions, we thus normalize the permutation by

cyclically shifting the array until token 1 occupies the first position in the array.

Theorem 8 (Parity in Top-Spin Puzzle) *For an even value of k (the default) and odd value of $n > k + 1$, the (normalized) (n, k) Top-Spin Puzzle has $(n-1)!/2$ reachable states.*

Proof. We first observe that due to the normalization for an even value of k, only a twist at the start/end of the normalized array can change the parity. Otherwise, the twist involves reversing k adjacent numbers, an operation with even parity.

Let $n = 2m + 1$ and $(x_0, x_1, \ldots, x_{2m})$ be the normalized state vector. Thus, due to normalization, $x_0 = 0$.

First of all, we observe that the modification of 0 is not counted as a transposition in the normalized representation, so only $k - 1$ elements actually change their relative position and lead to an odd number of transpositions.

Without loss of generality, we look at $k = 4$, which simplifies notation. Larger values of k only increase the number of cases, but lead to no further insight. Assuming $k = 4$, three elements change their relative position and lead to three transpositions.

We now look at the effect of normalization. For $(0, x_1, x_2, x_3, \ldots, x_{2m})$ we have four critical successors:

- $(x_3, x_2, x_1, 0, x_4 .... x_{2m})$,
- $(x_2, x_1, 0, x_{2m}, x_3, ..., x_{2m-1})$,
- $(x_1, 0, x_{2m-1}, x_{2m}, x_2, ..., x_{2m-2})$, and
- $(0, x_{2m-2}, x_{2m-1}, x_{2m}, x_1, ..., x_{2m-3})$.

In all cases, normalization has to move 3 elements either the ones with low index to the end of the array to postprocess the twist, or the ones with large indices to the start of the array to preprocess the operation. The number of transpositions for one such move is $2m - 1$. In total we have $3(2m - 1) + 3$ transpositions. As each transposition changes the parity and the total of $6m$ transpositions is even, all critical cases have even priority. □

As the parity is even for a move in the (normalized) (n, k) Top-Spin Puzzle for an odd value of $n > k + 1$, we obtain the entire set of $(n - 1)!$ reachable states.

4.4 Pancake Problem

The n-Pancake Problem [10] is to determine the number of flips of the first k pancakes (with varying $k \in \{1, \ldots, n\}$) necessary to put them into ascending order. The problem has been analyzed e.g. by [16]. It is known that $(5n + 5)/3$ flips always suffice, and that $15n/14$ flips are necessary.

In the *n-Burned-Pancake* variant, the pancakes are burned on one side and the additional requirement is to bring all burned sides down. For this version it is known that $2n - 2$ flips always suffice and that $3n/2$ flips are necessary. Both problems have n possible operators. The

pancake problem has $n!$ reachable states, the burned one has $n!2^n$ reachable states. For an even value of $\lceil (k-1)/2 \rceil$, $k > 1$ the parity changes, while for an odd one, the parity remains the same.

5 Hashing Selection Games

Figure 2: Initial States of the Two-Player Turn-Taking Game *Fox-and-Geese*.

Fox-and-Geese is a two-player zero-sum game. The lone fox attempts to capture the geese, while the geese try to hem the Fox, so that he can't move. It is played upon a cross-shaped 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. 2. Pieces can move to any empty intersection around them (also diagonally). The fox can additionally jump over a goose to capture it. Geese cannot jump. The geese win if they surround the fox so that it cannot move. The fox wins if it captures enough geese that the remaining geese cannot surround him.

Fox-and-Geese belongs to the set of asymmetric strategy games played on a cross shaped board. 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[3]. To the authors' knowledge, *Fox-and-Geese* has not been solved yet. 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. 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 goose (unprotected ones, empty vertices, which are central to the area around) and moves actively towards them.

[3] see *The Online Guide to Traditional Games*

Potential decoys, which try to lure the fox out of his 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.

Figure 3: Initial State of the Single-Player Game *Peg-Solitaire*.

Fox-and-Geese extends *Peg-Solitaire* (see Fig. 3), a single-agent problem invented in the 17th century. The game asks for the minimum number of pegs that is reachable from a given initial state. The set of pegs is iteratively reduced by jumps. Solutions for the initial state (shown in Fig. 3) with one peg remaining in the middle of the board are widely known [1]. An optimal player for all possible states has been generated by [12].

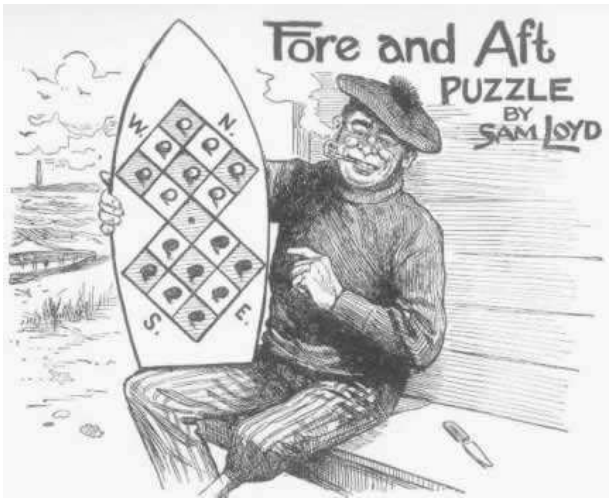

Figure 4: Initial State of the Single-Player Game *Fore and Aft*.

The *Fore and Aft* puzzle (see Fig. 4) has been made popular by the American puzzle creator Sam Loyd. It is played on a part of the 5×5 board consisting of two 3×3 subarrays at diagonally opposite corners. They overlap in the central square. One square has 8 black pieces and the other has 8 white pieces, with the centre left vacant. The objective is to reverse the positions of pieces in the lowest number of moves. Pieces can slide or jump over another pieces of any colour. This game was originally an English invention, having been designed by an English sailor in the 18th century. Henry Ernest Dudeney discovered a quickest solution of just 46 moves. *Frogs-and-Toads* generalizes *Fore and Aft* and larger versions are yet unsolved.

As the number of pegs shows the progress in playing the game *Peg-Solitaire*, we may aim at representing all boards with k of the $n-1$ possible pegs, where n is the number of holes. In fact, the breadth-first level k contains at most $\binom{n}{k}$ states. In contrast to permutation games, pegs are indistinguishable, and call for a different design of a hash function and its inverse.

Such an invertible perfect hash function of all states that have $k = 1, \ldots, n$ pegs remaining on the board reduces the RAM requirements for analyzing the game. As successor generation is fast, we will need an efficient hash function (rank) that maps bitvectors $(s_0, \ldots, s_{n-1}) \in \{0,1\}^n$ with k ones to $\{0, \ldots, \binom{n}{k} - 1\}$ and back (unrank). There is a trivial ranking algorithm that uses a counter to determine the number of bitvectors passed in their lexicographic ordering that have k ones. It uses linear space, but the time complexity by traversing the entire set of bitvectors is exponential. The unranking algorithm works similarly with matching exponential time performance.

The design of a linear time ranking and unranking algorithm is not obvious. The pieces on the board are not labeled, their relative ordering does not matter.

5.1 Hashing with Binomial Coefficients

An efficient solution for perfect and invertible hashing of all bitvectors with k ones to $\{0, \ldots, \binom{n}{k} - 1\}$ is shown in Algorithms 8 and 9. The algorithms utilize binomial coefficients that can either be precomputed or determined on-the-fly. The algorithms rely on the observation that once a bit at position i in a bitvector with n bits and with j zeros is processed, the binomial coefficient $\binom{i}{j-1}$ can be added to the rank. The notation $\max\left\{0, \binom{i}{\mathit{zeros}-1}\right\}$ is shorthand notation to say, if $\mathit{zeros} < 1$ take 0, otherwise take $\binom{i}{\mathit{zeros}-1}$.

The time complexities of both algorithms are $O(n)$. In case the number of zeros exceeds the number of ones, the rank and unrank algorithms can be extended to the inverted bitvector representation of a state.

The correctness argument is based on representing of the binomial coefficients in a grid graph of nodes $B_{i,j}$ with i denoting the position in the bit-vector and j denoting the number of zeros already seen. Let $B_{i,j}$ be connected via a directed edge to $B_{i-1,j}$ and $B_{i-1,j-1}$ corresponding to a zero and an one processed in the bit-vector. Starting at $B_{i,j}$ there are $\binom{i}{j}$ possible non-overlapping paths that reach $B_{0,z}$. These pathcount-values can be used to determine the index of a given bitvector in the set of all possible ones. At the current node (i,j) in the grid graph in case of the state at position i containing a

- 1: all path-counts at $B_{i-1,j-1}$ are added.

Algorithm 8 Ranking-Binomial-Coefficients(s, ones)

```
1: r := 0; i := n; zeros := n − ones
2: while (i > 0) do
3:    i := i − 1
4:    if (s[i] = 0) then
5:       zeros := zeros − 1
6:    else
7:       value_0 := max{0, C(i, zeros − 1)}
8:       r := r + value_0
9: return r
```

Algorithm 9 Unranking-Binomial-Coefficients(r, n, ones)

```
1: i := n; zeros := n − ones
2: while (i > 0) do
3:    i := i − 1
4:    value_0 := max{0, C(i, zeros − 1)}
5:    if (r < value_0) and (zeros > 0) then
6:       zeros := zeros − 1
7:       record s[i] := 0
8:    else
9:       record s[i] := 1
10:      r := r − value_0
11: return s
```

- 0: nothing is added.

5.2 Hashing with Multinomial Coefficients

The perfect hash functions derived for games like *Peg-Solitaire* are often insufficient in games with pieces of different color like *Tic-Tac-Toe* and *Nine-Men-Morris*. For this case, we have to devise a hash function that operates on state vectors of size n that contain zeros (location not occupied), ones (location occupied by pieces of the first player) and twos (location occupied by pieces of the second player). We will determine the value of a position by hashing all state with a fix number of z zeros, and o ones and $t = n - z - o$ twos to a value in $\{0, \ldots, \binom{n}{z,o,t} - 1\}$, where the multinomial coefficient $\binom{n}{z,o,t}$ is defined as

$$\binom{n}{z, o, t} = \frac{n!}{z! \cdot o! \cdot t!}.$$

The implementations of the rank and unrank functions are shown in Algorithms 10 and 11. They naturally extend the code derived for binomial coefficients.

Algorithm 10 Ranking-Multinomial-Coefficients$(s, n, \mathit{ones}, \mathit{twos})$

1: $r := 0$; $\mathit{zeros} := n - \mathit{ones} - \mathit{twos}$; $i := n$
2: **while** $(i > 0)$ **do**
3: $\quad i := i - 1$
4: $\quad$ **if** $(s[i] = 0)$ **then**
5: $\quad\quad \mathit{zeros} := \mathit{zeros} - 1$
6: $\quad$ **else**
7: $\quad\quad$ **if** $s[i] = 1$ **then**
8: $\quad\quad\quad \mathit{value}_0 := \max\left\{0, \binom{i}{\mathit{zeros}-1,\mathit{ones},i-\mathit{zeros}-\mathit{ones}-1}\right\}$
9: $\quad\quad\quad r := r + \mathit{value}_0$
10: $\quad\quad\quad \mathit{ones} := \mathit{ones} - 1$
11: $\quad\quad$ **else**
12: $\quad\quad\quad \mathit{value}_0 := \max\left\{0, \binom{i}{\mathit{zeros}-1,\mathit{ones},i-\mathit{zeros}-\mathit{ones}-1}\right\}$
13: $\quad\quad\quad \mathit{value}_1 := \max\left\{0, \binom{i}{\mathit{zeros},\mathit{ones}-1,i-\mathit{zeros}-\mathit{ones}-1}\right\}$
14: $\quad\quad\quad r := r + \mathit{value}_0 + \mathit{value}_1$
15: **return** r

The correctness argument relies on representing the multinomial coefficients in a 3D grid graph of nodes $B_{i,j,l}$ with i denoting the index position in the vector and j denoting the number of zeros j, and l denoting the number of ones already seen. The number of twos is then immediate. Let $B_{i,j,l}$ be connected via a directed edge to $B_{i-1,j,l}$, $B_{i-1,j,l-1}$ and $B_{i-1,j-1,l}$ corresponding to a value 2, 1 or 0 processed in the bit-vector, respectively. There are $\binom{i}{j,l,n-j-l}$ possible non-overlapping paths starting from each node $B_{i,j,l}$ that reach $B_{0,z,o}$. These pathcount-values can be used to determine the index of a given bitvector in the set of all possible ones. At the current node (i, j, l) in the grid graph in case of the node at position i containing a

- 1: all path-counts values at $B_{i-1,j-1,l}$ are added.
- 2: all path-counts values at $B_{i-1,j,l-1}$ are added.
- 0: nothing is added.

6 Parallelization

Parallel processing is the future of computing. On current personal computer systems with multiple cores on the CPU and (graphics) processing units on the graphics card, parallelism is available "for the masses". For our case of solving games, we aim at fast successor computation. Moreover, ranking and unranking take substantial running time are executed in parallel.

To improve the I/O behavior the partitioned state space was distributed over multiple hard disks. This increased the reading and writing bandwidth and to enable each thread to use its own hard disk. In larger instances that exceed RAM capacities we additionally maintain write buffers

Algorithm 11 Unranking-Multinomial-Coefficients$(r, n, \mathit{ones}, \mathit{twos})$

1: $i := n$; $\mathit{zeros} := n - \mathit{ones} - \mathit{twos}$
2: **while** $i > 0$ **do**
3: $\quad i := i - 1$
4: $\quad \mathit{value}_0 := \max\left\{0, \binom{i}{\mathit{zeros}-1,\mathit{ones},i-\mathit{zero}-\mathit{ones}-1}\right\}$
5: $\quad \mathit{value}_1 := \max\left\{0, \binom{i}{\mathit{zeros},\mathit{ones}-1,i-\mathit{zero}-\mathit{ones}-1}\right\}$
6: $\quad$ **if** $(r < \mathit{value}_0)$ and $(\mathit{zeros} > 0)$ **then**
7: $\quad\quad \mathit{zeros} := \mathit{zeros} - 1$
8: $\quad\quad$ record $s[i] := 0$
9: $\quad$ **else**
10: $\quad\quad$ **if** $(r < \mathit{value}_0 + \mathit{value}_1)$ and $(\mathit{ones} > 0)$ **then**
11: $\quad\quad\quad \mathit{ones} := \mathit{ones} - 1$
12: $\quad\quad\quad$ record $s[i] := 1$
13: $\quad\quad\quad r := r - \mathit{value}_0$
14: $\quad\quad$ **else**
15: $\quad\quad\quad$ record $s[i] := 2$
16: $\quad\quad\quad \mathit{twos} := \mathit{twos} - 1$
17: $\quad\quad\quad r := r - \mathit{value}_0 - \mathit{value}_1$
18: **return** s

to avoid random access on disk. Once the buffer is full, it is flushed to disk. In one streamed access, all corresponding bits are set.

6.1 Multi-Core Computation

Nowadays computers have multiple cores, which reduce the run-time of an algorithm via distribution the workload to concurrently running threads.

We use *pthreads* as additional multi-threading support.

Let S_p be the set of all possible positions in *Fox-and-Geese* (*Frogs-and-Toads*) with p pieces, which together with the fox (blank) position and the player's turn uniquely address states in the game. During play, the number of pieces decreases (or stays) such that we partition backward (forward) BFS layers into disjoint sets $S_p = S_{p,0} \cup \ldots \cup S_{p,n-1}$. As $|S_{p,i}| \leq \binom{n-1}{p}$ is constant for all $i \in \{0, \ldots, n-1\}$, a possible upper bound on the number of reachable states with p pieces is $n \cdot \binom{n-1}{p}$. These states will be classified by our algorithm.

In two-bit retrograde (bfs) analysis all layers $\mathit{Layer}_0, \mathit{Layer}_1, \ldots$ are processed in partition form. The fixpoint iteration to determine the solvability status in one backward (forward) BFS level $\mathit{Layer}_p = S_{p,0} \cup \ldots \cup S_{p,n-1}$ is the most time consuming part. Here, we can apply a multi-core parallelization using pthreads. In total, n threads are forked and joined after completion. They share the same hash function, and communicate for termination.

For improving space consumption we urge the exploration to flush the sets $S_{p,i}$ whenever

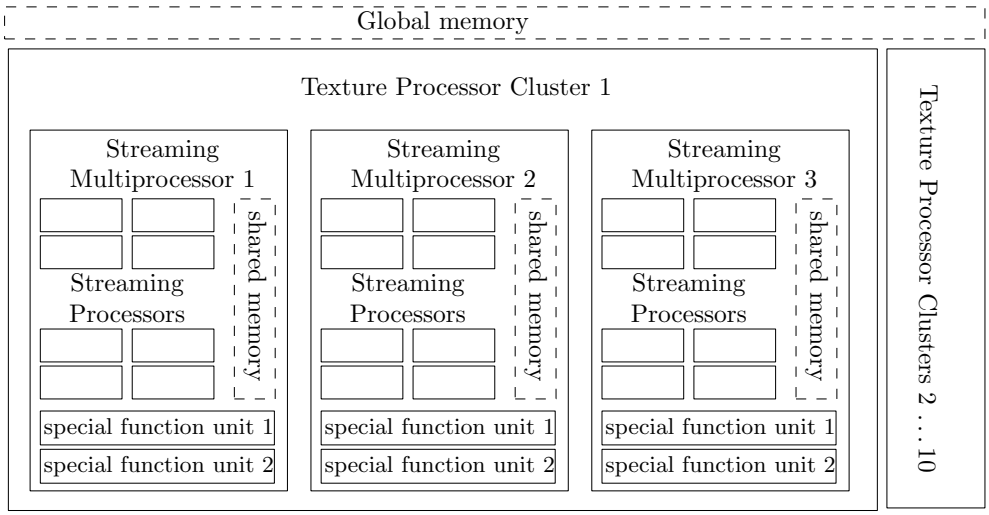

Figure 5: Sample GPU Architecture (G200 Chipset).

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 fixpoint iteration, we release the memory needed to store the previous layer. As a result, the maximum number of bits needed is $\max\{|S_p|, |S_p|/n + |S_{p-1}|)$.

6.2 GPU Computation

In the last few years there has been a remarkable increase in the performance and capabilities of the graphics processing unit. Modern GPUs are not only powerful, but also parallel programmable processors featuring high arithmetic capabilities and memory bandwidths. Deployed on current graphic cards, GPUs have outpaced CPUs in many numerical algorithms. The GPU's rapid increase in both programmability and capability has inspired researchers to map computationally demanding, complex problems to the GPU.

GPUs have multiple cores, but the programming and computational model are different from the ones on the CPU. Programming a GPU requires a special compiler, which translates the code to native GPU instructions. The GPU architecture mimics a single instruction multiply data (SIMD) computer with the same instructions running on all processors. It supports different layers for memory access, forbids simultaneous writes but allows concurrent reads to one memory cell.

If we consider the G200 chipset, as found in state-of-the-art NVIDIA GPUs and illustrated in Figure 5, a core is a streaming processor (SP) with 1 floating point and 2 arithmetic logic units. 8 SPs are grouped together with a cache structure and two special function units (performing e.g. double precision arithmetics) to one streaming multiprocessor (SM), and used like ordinary SIMD processors. Each of the 10 texture processor clusters (TSCs) combines 3 SMs, yielding

240 cores in one chip.

Memory, visualized shaded in the figure, is structured hierarchically, starting with the GPU's global memory (video RAM, or VRAM). Access to this memory is slow, but can be accelerated through *coalescing*, where adjacent accesses with less than 64 bits are combined to one 64-bit access. Each SM includes 16 KB of memory (SRAM), which is shared between all SPs and can be accessed at the same speed as registers. Additional registers are also located in each SM but not shared between SPs. Data has to be copied from the systems main memory to the VRAM to be accessible by the threads.

The GPU programming language links to ordinary C-sources. The function executed in parallel on the GPU is called *kernel*. The kernel is driven by threads, grouped together in *blocks*. The TSC distributes the blocks to its SMs in a way that none of them runs more than $1,024$ threads and a block is not distributed among different SMs. This way, taking into account that the maximal *blockSize* of 512, at most 2 blocks can be executed by one SM on its 8 SPs. Each SM schedules 8 threads (one for each SP) to be executed in parallel, providing the code to the SPs. Since all the SPs get the same chunk of code, SPs in an else-branch wait for the SPs in the if-branch, being idle. After the 8 threads have completed a chunk the next one is executed. Note that threads waiting for data can be parked by the SM, while the SPs work on threads, which have already received the data.

To profit from coalescing, threads should access adjacent memory contemporary. Additionally, the SIMD like architecture forces to avoid if-branches and to design a kernel which will be executed unchanged for all threads. This facts lead to the implementation of keeping the entire or partitioned state space bitvector in RAM and copying an array of indices (ranks) to the GPU. This approach benefits from the SIMD technology but imposes additional work on the CPU. One additional scan through the bitvector is needed to convert its bits into integer ranks, but on the GPU the work to unrank, generate the successors and rank them is identical for all threads. To avoid unnecessary memory access, the rank given to expand should be overwritten with the rank of the first child. As the number of successors is known beforehand, with each rank we reserve space for its successors. For smaller BFS layers this means that a smaller amount of states is expanded.

For solving games on the graphics card [14], storing the bitvector on the GPU yields bad exploration results. Hence, we forward the bitvector indices from the CPU's host RAM to the GPU's VRAM, where they were uploaded to the SRAM, unranked and expanded, while the successors were ranked. At the end of one iteration, all successors are moved back to CPU's host RAM, where they are perfectly hashed and marked if new.

7 Experiments

We divide the presentation of the experiments in permutation games (mostly showing the effect of multi-core GPU computation) and selection games (also showing the effect of multi-core CPU computation).

7.1 Permutation Games

We conducted the permutation game experiments on an AMD Athlon 64 X2 Dual Core Processor 3800+ system with 2 GB RAM. The GPU we used was an NVIDIA graphics card with G-200 chipset and 1 GB VRAM. In all cases we perform forward breadth-first search to generate the entire state space.

For measuring the speed-up on a matching implementation we compare the GPU performance with a CPU emulation on a single core. This way, the same code and work was executed on the CPU and the GPU. For a fair comparison, the emulation was run with GPU code adjusted to one thread. This minimizes the work for thread communication on the CPU. Moreover, we profiled that the emulation consumed most CPU time for state expansion and ranking.

Sliding-Tile Puzzle The results of the first set of experiments shown in Table 1 illustrate the effect of bitvector state space compression with breadth-first search in rectangular *Sliding-Tile* problems of different sizes.

We run both the one- and two-bit breadth-first search algorithms on the CPU and GPU. The 3×3 version was simply too small to show significant advances, while even in partitioned form a complete exploration on a bit vector representation of the 15-Puzzle requires more RAM than available.

We first validated that all states were generated and equally distributed among the possible blank positions. Moreover, as expected, the numbers of BFS layers for symmetric puzzle instances match (53 for 3×4 and 4×3 as well as 63 for 2×6 and 6×2).

For the 2-Bit BFS implementation, we observe a moderate speed-up by a factor between 2 and 3, which is due to the fact that the BFS-layers of the instances that could be solved in RAM are too small. For such small BFS layers, further data processing issues like copying the indices to the VRAM is rather expensive compared to the gain achieved by parallel computation on the GPU. Unfortunately, the next larger instance (7×2) was too large for the amount of RAM available in the machine (it needs $3 \times 750 = 2,250$ MB for *Open* and 2 GB for reading and writing indices to the VRAM).

In the 1-Bit BFS implementation the speed-up increases to a factor between 7 and 10 in the small instances. Many states are re-expanded in this approach, inducing more work for the GPU and exploiting its potential for parallel computation. Partitions being too large for the VRAM are split and processed in chunks of about 250 millions indices (for the 7×2 instance). A quick calculation shows that the savings of GPU computation are large. We noticed that the GPU has the capability to generate 83 million states per second (including unranking, generating the successors and computing their rank) compared to about 5 million states per second of the CPU. As a result, for the CPU experiment that ran out of time (o.o.t), which we stopped after one day of execution, we predict a speed-up factor of at least 16, and a running time of over 60 hours.

Top-Spin Problems The results for the (n, k)-Top-Spin problems for a fixed value of $k = 4$ are shown in Table 7.1 (o.o.m denotes out of memory, while o.o.t denotes out of time). We see that the experiments validate the theoretical statement of Theorem 1 that the state spaces are of

Problem	2-Bit Time GPU	CPU	1-Bit Time GPU	CPU
(2×6)	1m10s	2m56s	2m43s	15m17s
(3×4)	55s	2m22s	1m38s	13m53s
(4×3)	1m4s	2m22s	1m44s	12m53s
(6×2)	1m26s	2m40s	1m29s	18m30s
(7×2)	o.o.m.	o.o.m.	226m30s	o.o.t.

Table 1: Comparing GPU with CPU Performances in 1-Bit and 2-Bit BFS in the Sliding-Tile Puzzle Domain.

n	States	GPU Time	CPU Time
6	120	0s	0s
7	360	0s	0s
8	5,040	0s	0s
9	20,160	0s	0s
10	362,880	0s	6s
11	1,814,400	1s	35s
12	39,916,800	27s	15m20s

Table 2: Comparing GPU with CPU Performances for Two-Bit-BFS in the Top-Spin Domain.

size $(n-1)!/2$ for n being odd[4] and $(n-1)!$ for n even. For large values of n, we obtain a significant speed-up of more than factor 30.

Pancake Problems The GPU and CPU running time results for the n-Pancake problems are shown in Table 7.1. Similar to the Top-Spin puzzle for a large value of n, we obtain a speed-up factor of more than 30 wrt. running the same algorithm on the CPU.

7.2 Selection Games

Experiments are drawn on a Linux-PC with an Intel i7 processor having 8 cores running at 2.66 Ghz. The computer is equipped with 12 GB RAM. It has a NVIDIA graphics card with G-200 Chipset and 1GB VRAM.

Peg-Solitaire The first set of results, shown in Table 4, considers *Peg-Solitaire*. For each BFS-layer, the state space is small enough to fit in RAM. The exploration result show that there are 5 positions with one peg remaining (of course there is none with zero pegs), one of which has the peg in the goal position.

[4]At least the Top-Spin implementation of Rob Holte and likely the one of Ariel Felner/Uzi Zahavi do not consider parity compressed state spaces.

n	States	GPU Time	CPU Time
9	362,880	0s	4s
10	3,628,800	2s	48s
11	39,916,800	21s	10m41s
12	479,001,600	6m50s	153m7s

Table 3: Comparing GPU with CPU Performances in Two-Bit-BFS in Pancake Problems.

In *Peg-Solitaire* we find symmetry, which applies to the entire state space. If we invert the board (exchanging pegs with holes or swapping the colors), the goal and the initial state are the same. Moreover, the entire forward and backward graph structures match.

Hence, a call of backward breadth-first search to determine the number of states with a fixed goal distance is not needed. The number of states with a certain goal distances matches the number of states with a the same distance to the initial state. The total number of reachable states is 187,636,298.

We parallelized the game expanding and ranking states on the GPU. The total time for a BFS we measured was about 12m on the CPU and 1m8s on the GPU.

As the puzzle is moderately small, we consider the GPU speed-up factor of about 6 wrt. CPU computation as being significant.

For validity of the results, we compared the exploration results match with the ones obtained in [12]. For this case we had to alter the reward structure to the one that is imposed by the general game description language that was used there. We found that the number of expanded states matches, but – as expected – the total time to classify the states using the specialized player on the GPU is much smaller than in the general player of [12] running on one core of the CPU.

Frogs-and-Toads Similar to *Peg-Solitaire* if we invert the board (swapping the colors of the pieces), the goal and the initial state are the same, so that forward breadth-first search sufficies to solve the game.

In a BFS of about 0.19 seconds we validated the result of Dudeney for the *Fore and Aft* problem that reversing black and white takes 46 moves. There are two patterns which require 47 moves, namely, after reversing black and white, put one of the far corner pieces in the center. Table 5 also shows that there are 218,790 possible patterns of the pieces.

As *Frogs-and-Toads* generalizes *Fore and Aft*, we next considered the variant with 15 black and 15 white pieces on a board with 31 squares. The BFS outcome computed in 148m is shown in Table 6. We monitored that reversing black and white pieces takes 115 steps (in a shortest solution) and see that the worst-case input is slightly harder and takes 117 steps. A GPU parallelization leading to the same exploration results required about half an hour run-time.

Fox-and-Geese The next set of results shown in Table 7 considers the *Fox-and-Geese* game, where we applied retrograde analysis. For a fixed fox position the remaining geese can be binomially hashed. Moves stay in the same partition.

Holes	Bits	Space	Expanded
0	1	1 B	–
1	33	5 B	1
2	528	66 B	4
3	5,456	682 B	12
4	40,920	4,99 KB	60
5	237,336	28,97 KB	296
6	1,107,568	135 KB	1,338
7	4,272,048	521 KB	5,648
8	13,884,156	1.65 MB	21,842
9	38,567,100	4.59 MB	77,559
10	92,561,040	11.03 MB	249,690
11	193,536,720	23.07 MB	717,788
12	354,817,320	42.29 MB	1,834,379
13	573,166,440	68.32 MB	4,138,302
14	818,809,200	97.60 MB	8,171,208
15	1,037,158,320	123 MB	14,020,166
16	1,166,803,110	139 MB	20,773236
17	1,166,803,110	139 MB	26,482,824
18	1,037,158,320	123 MB	28,994,876
19	818,809,200	97.60 MB	27,286,330
20	573,166,440	68.32 MB	22,106,348
21	354,817,320	42.29 MB	15,425,572
22	193,536,720	23.07 MB	9,274,496
23	92,561,040	11.03 MB	4,792,664
24	38,567,100	4.59 MB	2,120,101
25	13,884,156	1.65 MB	800,152
26	4,272,048	521 KB	255,544
27	1,107,568	135 KB	68,236
28	237,336	28.97 KB	14,727
29	40,920	4.99 KB	2529
30	5,456	682 B	334
31	528	66 B	33
32	33	5 B	5
33	1	1 B	-

Table 4: One-Bit-BFS Results for *Peg-Solitaire*.

Depth	Expanded	Depth	Expanded	Depth	Expanded	Depth	Expanded
1	1	13	1,700	25	15,433	37	1,990
2	8	14	2,386	26	14,981	38	1,401
3	13	15	3,223	27	14,015	39	914
4	14	16	4,242	28	12,848	40	557
5	32	17	5,677	29	11,666	41	348
6	58	18	7,330	30	10,439	42	202
7	121	19	8,722	31	9,334	43	137
8	178	20	10,084	32	7,858	44	66
9	284	21	11,501	33	6,075	45	32
10	494	22	12,879	34	4,651	46	4
11	794	23	13,997	35	3,459	47	11
12	1,143	24	14,804	36	2,682	48	2

Table 5: BFS Results for *Fore and Aft*.

In spite of the work for classification being considerable, it was feasible to complete the analysis with 12 GB RAM. In fact, we observed that the largest problem with 16 geese required resistant space in main memory of 9.2 GB RAM.

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 boarder 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 potentially drawn positions that couldn't been proven won or lost by the geese, are devised to be a win for the fox. The critical point, where the fox looses more than 50% of the game seems to be reached at currently explored level 16. This matches the observation in practical play, that the 13 geese are too less to show an edge for the geese.

The total run-time of about 730h (about a month) for the experiment is considerable. Without multi-core parallelization, 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 of space needed for operating an optimal player is about 34 GB, so that in case geese are captured we would have to reload data from disk. This strategy yields a maximal space requirement of 4.61 GB RAM, which might further be reduced by reloading data in case of a fox moves.

8 Discussion

In this section we discuss further applications of the above approach.

The games have different applications in moving target search. For example, *Fox-and-Geese* is prototypical for chasing an attacker, with applications to computer security, where an intruder

Depth	Expanded	Depth	Expanded	Depth	Expanded	Depth	Expanded
1	1	31	4,199,886	61	171,101,874	91	4,109,157
2	8	32	5,447,660	62	170,182,837	92	3,156,288
3	17	33	6,975,087	63	168,060,816	93	2,387,873
4	26	34	8,865,648	64	164,733,845	94	1,780,521
5	46	35	11,138,986	65	160,093,746	95	1,307,312
6	78	36	13,881,449	66	154,297,247	96	948,300
7	169	37	17,060,948	67	147,342,825	97	680,299
8	318	38	20,800,347	68	139,568,855	98	484,207
9	552	39	25,048,652	69	131,146,077	99	340,311
10	974	40	29,915,082	70	122,370,443	100	235,996
11	1,720	41	35,382,942	71	113,415,294	101	160,153
12	2,905	42	41,507,233	72	104,380,748	102	107,024
13	4,826	43	48,277,767	73	95,379,850	103	69,216
14	7,878	44	55,681,853	74	86,375,535	104	44,547
15	12,647	45	63,649,969	75	77,534,248	105	27,873
16	19,980	46	72,098,327	76	68,891,439	106	17,394
17	31,511	47	80,937,547	77	60,672,897	107	10,256
18	49,242	48	89,999,613	78	52,953,463	108	6,219
19	74,760	49	99,231,456	79	45,889,798	109	3,524
20	112,218	50	108,495,904	80	39,482,737	110	2,033
21	166,651	51	117,679,229	81	33,751,896	111	1,040
22	241,157	52	126,722,190	82	28,607,395	112	532
23	348,886	53	135,363,894	83	24,035,844	113	251
24	497,698	54	143,534,546	84	19,957,392	114	154
25	700,060	55	150,897,878	85	16,394,453	115	42
26	974,219	56	157,334,088	86	13,306,659	116	19
27	1,337,480	57	162,600,933	87	10,695,284	117	10
28	1,812,712	58	166,634,148	88	8,521,304	118	2
29	2,426,769	59	169,360,939	89	6,738,557		
30	3,214,074	60	170,829,205	90	5,286,222		

Table 6: BFS Results for *Frogs-and-Touds*.

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		to be re-run	388m	
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

Table 7: Retrograde Analysis Results for *Fox-and-Geese*.

has to be catched. In a more general setting, the board games are played with tokens on a graph $G = (V, E)$. For example, one move corresponds to pass a token along an edge $(i, j) \in E$. The space complexities of the bit-state analysis now depends on the number of tokens played and the number of nodes. For particular types of Petri nets like safe nets this might yield an appropriate compression for their exploration.

8.1 Symmetries

Symmetries are helpful to reduce the time and space consumption of a classification algorithm.

In many board games we find reflection along the main axes or along the diagonals. If we look at the four possible rotations on the board for *Peg-Solitaire* and *Fox-and-Geese* plus reflection, we count 8 symmetries in total.

The exploitation of state symmetries are of various kinds. For *Fox-and-Geese* we can classify all states that share a symmetrical fox position by simply copying the result obtained for the existing one. Besides the savings of time for not expanding states, this can also save the number of positions that have to be kept in RAM during fixpoint computation.

If the forward and backward search graphs match (as in *Peg Solitaire* and *Frogs-and-Toads*) we may also truncate the breadth-first search proceedure to the half of the search depth. In two-bit BFS, we simply have to look at the rank of the inverted unranked state. Moreover, with the forward BFS layers we also have the minimal distances of each state to the goal state, and, hence, the classification result.

8.2 Frontier Search

Frontier search is motivated by the attempt of omitting the *Closed* list of states already expanded. It mainly applies to problem graphs that are directed or acyclic but has been extended to more general graph classes. It is especially effective if the ratio of *Closed* to *Open* list sizes is large.

Frontier search requires the *locality* of the search space [28] being bounded, where the locality (for breadth-first search) is defined as $\max\{\mathit{layer}(s) - \mathit{layer}(s') + 1 \mid s, s' \in S; s' \in \mathit{succs}(s)\}$, where $\mathit{layer}(s)$ denotes the depth d of s in the breadth-first search tree.

For frontier search, the *space efficiency* of the hash function $h : S \to \{0, \ldots, m-1\}$ boils down to $m / \left(\max_d |\mathit{Layer}_d| + \ldots + |\mathit{Layer}_{d+l}|\right)$, where Layer_d is set of nodes in depth d of the breadth-first search tree and l is the locality of the breadth-first search tree as defined above.

For the example of the Fifteen puzzle, i.e., the 4×4 version of *Sliding-Tile*, the predicted amount of 1.2 TB hard disk space for 1-bit breadth-first search is only slightly smaller than the 1.4 TB of frontier breadth-first search reported by [21].

As frontier search does not shrink the set of states reachable, one may conclude, that frontier search hardly cooperates well with a bitvector representation of the entire state space. However, if layers are hashed individually, as done in all selection games we have considered, a combination of bit-state and frontier search is possible.

8.3 Pattern Databases

The breadth-first traversal in a bitvector representation of the search space is also essential for the construction of compressed pattern databases [5]. The number of bits per state can be reduced to $\log 3 \approx 1.6$. For this case, 5 values $\{0, 1, 2\}$ are packed into a byte, given that $3^5 = 243 < 255$. The observation that $\log 3$ are sufficient to represent all mod-3 values possible and the byte-wise packing was already made by [8].

The idea of pattern database compression is to store the mod-3 value (of the backward BFS depth) from abstract space, so that its absolute value can be computed incrementally in constant time. For the initial state, an incremental computation for its heuristic evaluation is not available, so that a backward construction of its generating path can be used. As illustrated in [5], for an undirected graph a shortest path predecessor with mod-3 of BFS depth k appears in level $k - 1$ *mod* 3.

As the abstract space is generated anyway for generating the database, one could alternatively invoke a shortest path search from the initial state, without exceeding the time complexity of database construction.

By having computed the heuristic value for the projected initial state as the goal distance in the inverted abstract state space graph, as shown in [5] all other pattern database lookup values can then be determined incrementally in constant time, i.e., $h(v) = h(u) + \Delta(v)$, with $v \in \mathit{succs}(u)$ and $\Delta(v)$ found using the mod-3 value of v. Given that the considered search spaces in [5] are undirected, the information to evaluate the successors with $\Delta(v) \in \{-1, 0, 1\}$ is possible.

For directed (and unweighted) search spaces more bits are needed to allow incremental heuristic computation in constant time. It is not difficult to see that the locality in the inverted abstract state space determines the maximum difference in h-values $h(v) - h(u)$, $v \in \mathit{succs}(u)$ in original space.

Theorem 9 (Locality determines Number of Bits for Pattern Database Compression) *In a directed (but unweighted) search space, the (dual) logarithm of the (breadth-first) locality of the inverse of the abstract state space graph plus 1 is an upper bound on the number of bits needed for incremental heuristic computation of bit-vector compressed pattern databases, i.e., for locality* $l_A^{-1} = \max\{\mathit{layer}^{-1}(u) - \mathit{layer}^{-1}(v) + 1 \mid u, v \in A; v \in \mathit{succs}^{-1}(u)\}$ *in abstract state space graph* A *of* S *we require at most* $\log\lceil l_A^{-1}\rceil + 1$ *bits to reconstruct the value* $h(v)$ *of a successor* $v \in S$ *of any chosen* $u \in S$ *given* $h(u)$.

Proof. First we observe that the goal distances in abstract space A determine the h-value in original state space, so that the locality $\max\{\mathit{layer}^{-1}(u) - \mathit{layer}^{-1}(v) + 1 \mid u, v \in A; v \in \mathit{succs}^{-1}(u)\}$ is bounded by $h(u) - h(v) + 1$ for all u, v in original space with $u \in \mathit{succs}(v)$, which is equal to the maximum of $h(v) - h(u) + 1$ for $u, v \in S$ with $v \in \mathit{succs}(u)$. Therefore, the number of bits needed for incremental heuristic computation equals $\lceil \max\{h(v) - h(u) \mid u, v \in A; v \in \mathit{succs}^{-1}(u)\}\rceil + 2$ as all values in the interval $[h(u) - 1, \ldots, h(v)]$ have to be accommodated for. Thus for the incremental value $\Delta(v)$ added to $h(u)$ we have $\Delta(v) \in \{-1, \ldots, h(v) - h(u)\}$,

so that $\lceil \log(\max\{h(v) - h(u) + 2 \mid u, v \in S; v \in \textit{succs}(u)\}) \rceil = \log\lceil l_A^{-1} \rceil + 1$ bits suffice to reconstruct the value $h(v)$ of a successor $v \in S$ for every $u \in S$ given $h(u)$. □

For undirected search spaces we have $\log l_A^{-1} = \log 2 = 1$, so that $1 + 1 = 2$ bits suffice to be stored for each abstract pattern state according to the theorem. Using the tighter packing of the $2 + 1 = 3$ values into bytes provided above, $8/5 = 1.6$ bits are sufficient.

If not all states in the search space that has been encoded in the perfect hash function are reachable, reducing the constant-bit compression to a lesser number of bits might not always be available, as unreached states cannot easily be removed. For this case, the numerical value remaining to be set for an unreachable states in the inverse of abstract state space will stand for h-value infinity, at which the search in the original search space can stop.

For problems with discretized costs, more general notions of locality based on cost-based backward construction have been developed [18]. More formally, the best-first locality has been defined as $\max\{\textit{cost-layer}(s) - \textit{cost-layer}(s') + \textit{cost}(s, s') \mid s, s' \in S; s' \in \textit{succs}(s)\}$, where *cost-layer*$(s)$ denotes the smallest accumulated *cost*-value from the initial state to s. The theoretical considerations on the number of bits needed to perform incremental heuristic evaluation extend to this setting.

8.4 Other Games

We distinguish between permutation games and selection games, and add remarks on general games for which a functional representation of the state space exists.

8.4.1 Permutation Games

Rubik's Cube, invented in the late 1970s by Erno Rubik, is a known challenge for single-agent search [19]. Each face can be rotated by 90, 180, or 270 degrees and the goal is to rearrange a scrambled cube such that all faces are uniformly colored.

Solvability invariants for the set of all dissembled cubes are:

- a single corner cube must not be twisted
- a single edge cube must not be twisted and
- no two cube must be exchanged

For the last issue the parity of the permutation is crucial and leads to $8! \cdot 3^7 \cdot 12! \cdot 2^{11}/2 \approx 4.3 \cdot 10^{19}$ solvable states. Assuming one bit per state, an impractical amount of $4.68 \cdot 10^{18}$ bytes for performing full reachability is needed. For generating upper bounds, however, bitvector representations of subspaces have been shown to be efficient [23].

8.4.2 Selection Games

The binomial and multinomial hashing approach is applicable to many other pen-and-paper and board games.

- In *Awari* [25] the two player redistribute seeds among 12 holes according to the rules of the game, with an initial state having uniformly four seeds in each of the holes. When all seeds are available, all possible layouts can be generated in an urn experiments with 59 balls, where 48 balls represent filling the current hole with a seed and 11 balls indicate changing from the current to the next hole. Thus the binomial hash function applies.

- In *Dots and Boxes* players take turns joining two horizontally or vertically adjacent dots by a line. A player that completes the fourth side of a square (a box) colors that box and must play again. When all boxes have been colored, the game ends and the player who has colored more boxes wins. Here, the binomial hash suffices. For each edge we denote whether or not it is marked. Together with the marking, we denote the number of boxes of at least one player. In difference to other games, all successors are in the next layer, so that one scan suffices to solve the current one.

- *Nine-Men's-Morris* is one of the oldest games still played today. Boards have been found on many historic buildings throughout the world. One of the oldest dates back to about 1400 BC [15]. The game naturally divides in three stages. Each player has 9 pieces, called men, that are first placed alternately on a board with 24 locations. In the second stage, the men move to form mills (a row of three pieces along one of the board's lines), in which case one man of the opponent (except the ones that form a mill) is removed from the board. In one common variation of the third stage, once a player is reduced to three men, his pieces may "fly" to any empty location. If a move has just closed a mill, but all the opponent's men are also in mills, the player may declare any stone to be removed. The game ends if a player has less than three men (the player loses), if a player cannot make a legal move (the player loses), if a midgame or endgame position is repeated (the game is a draw).

  Besides the usual symmetries along the axes, there is one in swapping the inner with the outer circle. Gassner has solved the game by computing large endgame databases for the last two phases together with alpha-beta search for the first phase [15]. His results showed that, assuming optimal play of both players, the game ends in a draw. For this game the multinomial hash is applicable.

8.5 General Games

It is not difficult to extend the above functions to more than two different sets of pieces on the board. For *Chinese Checkers*, for example, three and more colors are needed. In this case a larger multinomial coefficient has to be built, but the construction remains similar to the one above.

We now look at general games with state spaces provided in functional representation. This setting complements the explicit-state setting of Botelho et al.. The state space is present in so-called *functional representation*. It has been constructed in symbolic forward search and a bijection of all states S reached to $\{0, \ldots, |S| - 1\}$ is computed together with its inverse. This approach will have potential applications in action planning, general game playing, and model checking.

Algorithm 12 Rank-BDDs(s, v)

```
1: if v is 0-sink then
2:    return 0
3: if v is 1-sink then
4:    return 1
5: if v is node labeled x_i with 0-succ. u and 1-succ. w then
6:    if s[i] = 1 then
7:       return sat-count(v) + rank(s, w)
8:    if s[i] = 0 then
9:       return rank(s, u)
```

Algorithm 13 Unranking-BDDs(r)

```
1: i := 1
2: start at root
3: while i ≤ n do
4:    at node v for x_i with 0-succ. u and 1-succ. w
5:    if r ≥ sat-count(u) then
6:       r := r − sat-count(u)
7:       follow 1-edge to w, record s[i] := 1
8:    else
9:       follow 0-edge to u, record s[i] := 0
10:   i := i + 1
```

The above algorithms are special cases of according ranking and unranking functions developed for BDDs [11]. For the sake of completeness, the according rank and unrank algorithms are shown in Algorithm 12 and Algorithm 13. The BDD for representing the $\binom{n}{k}$ structure is of polynomial size. Secondly, up to the links to the zero sink that do not contribute to counting the number of satisfying paths, the BDD is quasi-reduced by means that all variables appear on every path.

For simple reachability analysis this does not provide any surplus, but in case of more complex algorithms, like the classification of two-player games, perfect hash function based on BDDs show computational advantages in form of (internal or external) memory gains.

9 Conclusion

In this work we presented and analyzed linear time ranking and unranking functions for games in order to execute breadth-first search and 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. As an interesting time-space trade-off we studied one-bit

reachability and one-bit breadth-first search.The latter imposes the presence of a move alternation property. Some previously unresolved games were solved, mostly in RAM, but sometimes using I/O-efficient strategies.

The approach featured parallel explicit-state traversal of two challenging games on limited space. We studied the application of multiple-core CPU and GPU computation and accelerated the analysis. The speed-ups compare well with alternative results combining external-memory and parallel search on multiple cores [21, 29].

In our experiments, the CPU speed-up is almost linear in the number of cores. For this we exploited independence in the problem, using an appropriate projection function. The GPU speed-up often exceeds the number of CPU cores considerably.

To compute invertible minimal perfect hash functions for permutation games, we extended the already efficient method by Myrvold and Ruskey. For selection games, with binomial and multinomial hashing we proposed an approach that has been inspired by counting the number of paths (lexicographic smaller to the given assignment) in a BDD. Due to the small amount of available shared RAM of 16 KB on the GPU, we prefer the space requirements for the ranking and unranking functions to be small.

References

[1] E. R. Berlekamp, J. H. Conway, and R. K. Guy. *Winning Ways*. Academic Press, 1982.

[2] B. Bonet. Efficient algorithms to rank and unrank permutations in lexicographic order. In *AAAI-Workshop on Search in AI and Robotics*, 2008.

[3] D. Bosnacki, S. Edelkamp, and D. Sulewski. Efficient probabilistic model checking on general purpose graphics processors. In *Model Checking Software (SPIN)*, 2009.

[4] F. C. Botelho and N. Ziviani. External perfect hashing for very large key sets. In *ACM Conference on Information and Knowledge Management (CIKM)*, pages 653–662, 2007.

[5] T. Breyer and R. Korf. 1.6-bit pattern databases. In *Symposium on Combinatorial Search (SOCS)*, 2009.

[6] M. Campbell, J. A. J. Hoane, and F. Hsu. Deep blue. *Artificial Intelligence*, 134(1-2):57–83, 2002.

[7] T. Chen and S. Skiena. Sorting with fixed-length reversals. *Discrete Applied Mathematics*, 71(1–3):269–295, 1996.

[8] G. Cooperman and L. Finkelstein. New methods for using Cayley graphs in interconnection networks. *Discrete Applied Mathematics*, 37/38:95–118, 1992.

[9] J. C. Culberson and J. Schaeffer. Pattern databases. *Computational Intelligence*, 14(4):318–334, 1998.

[10] M. Dweighter. Problem e2569. *American Mathematical Monthly*, (82):1010, 1975.

[11] S. Edelkamp and M. Diezfelbinger. Perfect hashing for state spaces in BDD representation. In *German Conference on Artificial Intelligence (KI)*, 2009.

[12] S. Edelkamp and P. Kissmann. Symbolic classification of general two-player games. In *KI*, pages 185–192, 2008.

[13] S. Edelkamp and D. Sulewski. Model checking via delayed duplicate detection on the GPU. Technical Report 821, TU Dortmund, 2008.

[14] S. Edelkamp and D. Sulewski. State space search on the GPU. In *Symposium on Combinatorial Search (SOCS)*, 2009.

[15] R. Gassner. Solving Nine-Men-Morris. *Computational Intelligence*, (12):24–41, 1996.

[16] W. H. Gates and C. H. Papadimitriou. Bounds for sorting by prefix reversal. *Discrete Mathematics*, 27:47–57, 1979.

[17] E. Hordern. *Sliding Piece Puzzles*. Oxford University Press, 1986.

[18] S. Jabbar. *External Memory Algorithms for State Space Exploration in Model Checking and Planning*. PhD thesis, University of Dortmund, 2008.

[19] R. E. Korf. Finding optimal solutions to Rubik's Cube using pattern databases. In *National Conference on Artificial Intelligence (AAAI)*, pages 700–705, 1997.

[20] R. E. Korf. Minimizing disk I/O in two-bit-breath-first search. In *National Conference on Artificial Intelligence (AAAI)*, pages 317–324, 2008.

[21] R. E. Korf and T. Schultze. Large-scale parallel breadth-first search. In *National Conference on Artificial Intelligence (AAAI)*, pages 1380–1385, 2005.

[22] D. L. Kreher and D. R. Stinson. *Combinatorial Algorithms*. Discrete Mathematics and Its Applications, 1984.

[23] D. Kunkle and G. Cooperman. Twenty-six moves suffice for Rubik's cube. In *International Symposium on Symbolic and Algebraic Computation (ISSAC)*, pages 235 – 242, 2007.

[24] W. Myrvold and F. Ruskey. Ranking and unranking permutations in linear time. *Information Processing Letters*, 79(6):281–284, 2001.

[25] J. W. Romein and H. E. Bal. Awari is solved. *Journal of the ICGA*, 25:162–165, 2002.

[26] J. Schaeffer, Y. Björnsson, N. Burch, A. Kishimoto, and M. Müller. Solving checkers. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 292–297, 2005.

[27] R. Zhou and E. A. Hansen. Structured duplicate detection in external-memory graph search. In *National Conference on Artificial Intelligence (AAAI)*, pages 683–689, 2004.

[28] R. Zhou and E. A. Hansen. Breadth-first heuristic search. *Artificial Intelligence*, 170(4-5):385–408, 2006.

[29] R. Zhou and E. A. Hansen. Parallel structured duplicate detection. In *National Conference on Artificial Intelligence (AAAI)*, pages 1217–1222, 2007.