Charles University in Prague

Faculty of Mathematics and Physics

BACHELOR THESIS

Pavel Dvořák

Advanced methods of searching the game tree of 3-dimensional Tic-Tac-Toe

Department of Applied Mathematics

Supervisor of the bachelor thesis:	RNDr. Tomáš Valla, Ph.D.
Study programme:	Computer Science
Specialization:	General Computer Science

Prague 2013

I would like to thank my supervisor Tomáš Valla for introducing me to combinatorial game theory (together with Robert Šámal) and for organizing our mini game seminary, where we discussed our game-theoretical thesis with Jitka Novotná and Pavel Veselý. I also would like to thank Michaela Žaloudková for the correction of my English.

Název práce: Pokročilé metody prohledávání herního stromu trojrozměrných piškvorek

Autor: Pavel Dvořák

Katedra: Katedra aplikované matematiky

Vedoucí bakalářské práce: RNDr. Tomáš Valla, PhD., Informatický ústav Univerzity Karlovy

Abstrakt: V této práci zkoumáme poziční hry, zejména pak vícerozměrné piškvorky. Porovnáváme existující pokročilé algoritmy (Pn-search, Db-Search, λ-search) pro řešení pozic v pozičních hrách. Algoritmy nasazujeme na domény her 4^3 a 5^3, což jsou první netriviální případy trojrozměrných piškvorek. Paralelizujeme Pn-search pro případ, kdy existuje více počátečních pozic. Pn-search aplikujeme jako jednovláknovou úlohu a řešíme, jak sdílet transpoziční tabulku s vyřešenými pozicemi. Hlavním a čistě teoretickým výsledkem je charakterizace grupy automorfismů kombinatorické krychle n^d se stejnou množinou linií jako vícerozměrné piškvorky. Toto je zobecnění Silvera [The American Mathematical Monthly, Vol. 74, No. 3, 1967], který popsal automorfismy hry 4^3.

Klíčová slova: vícerozměrné piškvory, paralelizace, automorfismy

Title: Advanced methods of searching the game tree of 3-dimensional Tic-Tac-Toe

Author: Pavel Dvořák

Department: Department of Applied Mathematics

Supervisor: RNDr. Tomáš Valla, PhD., Computer Science Institute of Charles University

Abstract: In this thesis we study positional games, especially multidimensional tic-tac-toe. We compare present advanced algorithms (Pn-search, Db-search and λ-search) for position solving in positional games. We apply the algorithms on the domain of 4^3 and 5^3 games, which are the first nontrivial cases of 3-dimensional tic-tac-toe. We parallelize Pn-search for cases when there are more starting positions. We apply Pn-search as a single-thread task and we solve how to share the transposition table with solved positions. Our main and clearly theoretical result is the characterization of the group of all automorphisms of combinatorial cube n^d with the same set of lines as multidimensional tic-tac-toe has. This is a generalization of Silver [The American Mathematical Monthly, Vol. 74, No. 3, 1967], who characterized the automorphisms of the game 4^3.

Keywords: multidimensional tic-tac-toe, parallelization, automorphisms

Contents

Introduction

Playing games has always been a pastime of the human race. Along with every game comes a question: How to play the game? There are two approaches how to answer this question. The first one is by experience. Usually when we play a game repeatedly, we learn with some time how to play this game good. The second approach is more formal. We can use mathematics to describe the rules and try to find the properties of the game. Mathematicians usually study games, where no randomness occurs, because the state of the game is determined only by players' moves. Classical game theory studies games with imperfect information. However in this thesis we are interested in tic-tac-toe, which is a game for 2 players with perfect information. These games are studied by combinatorial game theory.

For some simple games with small state space only paper and pen suffice to find out if the first player can force the win. However, when the games are more complex the paper and pen do not suffice and we need computers. Computers are mainly used for 2 problems according to games: finding the winning strategy or creating the artificial intelligence for the game. Actually these problems are very similar, both are solved by searching algorithms. However, a lot of games have huge state space and searching the whole state space for finding the winning strategy would last very long. So AI searches only part of the state space and selects the best ply with respect to the searched part of the state space.

Tic-tac-toe is one of the most famous games for two players. Even little children play it, because it has very simple rules. The size of a basic board of tic-tac-toe is 3×3 and it is easy to find by case analysis that the game ends with a draw, if both players play optimally. However when the size and dimension of the game board are bigger, the manual case analysis is impossible.

We are interested if the first player can force the win in 3-dimensional tic-tac-toe. In tic-tac-toe with the board $2 \times 2 \times 2$ and $3 \times 3 \times 3$ the first player can force the win and it can be shown by simple case analysis. Hence the first non-trivial size of the board is 4 and tic-tac-toe with board $4 \times 4 \times 4$ is called Qubic. Upon our knowledge, Qubic was solved twice in history by Oren Patashnik [5] in 1980 and by Louis V. Allis [1] in 1994. However, we are the first ones, as far as we know, who used the parallel algorithm. We apply Pn-search as a single-thread task and each starting position is solved in one thread. We found that parallel algorithm is easy to implement and it gives quite good results.

Usually people are not interested how many winning strategies exist for the first or second player, but only if a winning strategy exists. Therefore automorphisms are used very often, when a game is analyzed. All automorphisms for Qubic were characterized by Rolland Silver [7] in 1967. When we analyzed the $5 \times 5 \times 5$, we found that the group of automorphism of $5 \times 5 \times 5$ is isomorphic to the group of Qubic automorphisms. Hence we started to study automorphisms a little bit more. Some people ask about the automorphisms of general tic-tac-toe. We answer these questions and we characterized all automorphisms for tic-tac-toe played in general cube with arbitrary length of edge and dimension.

This thesis was motivated by solving the game $5 \times 5 \times 5$, which is an open problem. Unfortunately we did not manage to solve this game. The game $5 \times 5 \times 5$ has a much bigger state space than Qubic and also the algorithms which we used are not as efficient as for Qubic. Nonetheless, we still have two results:

- We propose an easy-implemented parallelization of Pn-search, when there are more starting positions.
- We characterize all automorphisms of the combinatorial cube n^d with the multidimensional tic-tac-toe set of lines.

In the first chapter, we present positional games and tic-tac-toe, how we divide the games and we also formalize the rules. Chapter 2 is about algorithms, which we use for Qubic. In Chapter 3 we present our results of Qubic solving and how we use parallelization. In the last chapter we characterize the automorphism for general tic-tac-toe n^d.

1. Positional games

Games can be divided by many parameters like:

- Number of players
- If the game contains any random element
- Perfect/imperfect information, i.e. if players know the whole state of the game or not

Positional games are games for two players with perfect information and no random element. J. Beck [2] started to study positional games and he did a lot of work in this theory. He defines the positional game as follows:

Definition 1.1. Let H be a finite hypergraph, then the game with following rules is a *strong positional game*:

1. Two players alternately colour colourless vertices of the hypergraph, each player by his own colour.
2. Each player colours exactly one vertex per ply.
3. If a player makes a monochromatic edge of the hypergraph, he wins and the game ends.
4. If all vertices are coloured and no player has monochromatic edge, the game ends with a draw.

To solve the strong positional game means to find who win the game—if the first player can force the win or if the second player can force the draw.

Theorem 1.1 (Strategy stealing). *The second player in strong positional game cannot force the win.*

Idea of Proof. For contradiction let us suppose that the second player can force the win and he writes his strategy on some paper. Now suppose that the first player steals this paper. If the first player makes his first ply randomly and then he plays by the stolen winning strategy, he cannot lose, which is a contradiction.

The positional games are connected to Ramsey theory. Let $H = (V, E)$ be the hypergraph, such that for every vertex colouring with two colours, there exists a monochromatic edge $e \in E$. We know that such hypergraphs exist by Ramsey theory. It implies that a draw does not exist in strong positional games played on H. By strategy stealing we know that the second player cannot force the win. Therefore the first player can force the win in the strong positional games played on such hypergraphs H. There is other type of positional games motivated by strategy stealing:

Definition 1.2. Let H be a finite hypergraph, then the game with the following rules is a *weak positional game*:

1. Two players alternately colour colourless vertices of the hypergraph, each player by his own colour.

2. Each player colours exactly one vertex per ply.

3. If the first player makes a monochromatic edge of the hypergraph, he wins and the game ends.

4. If all vertices are coloured and the first player has not monochromatic edge, the second player wins.

The first player in weak positional games is usually called the maker and the second player the breaker.

Definition 1.3. Let $H = (E, V)$ be a finite hypergraph and $e = \{v_1, \dots, v_n\}$ be the arbitrary edge in E. Then *threat* of the player p in strong positional game played on H is state, when the player p colours vertices $\{v_1, \dots v_{n-1}\}$ and v_n is uncoloured.

When a player makes a threat, the other player must colour the last vertex of the edge, otherwise he loses immediately. In this thesis, the attacker is player, who creates the threat, and the defender is the other player, who tries to block it. Positional games are good subjects for testing new searching algorithms, because they usually have simple rules and a quite large state space. Weak positional games are easier to solve, because the second player can not make threats. There are a lot of known strategies for the weak games. However, for a lot of strong games we have only strategy stealing. Therefore we need a brutal force to solve these games. Moreover, the existence of the first player winning strategy in the strong game played on hypergraph $H(V, E)$ does not imply the existence of the first player winning strategy in the strong game played on $H(V, E \cup e)$, where e is the new edge, $e \notin E$. An example of this extra edge paradox in strong games is in Figure 1.1.

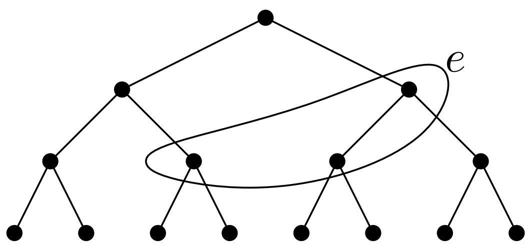


Figure 1.1: If we consider the strong game on tree $T(V, E)$, where E contains all branches of T (each has size 4) and $e \notin E$. The first player clearly has the winning strategy. He colours the root of T and then he colours the roots of the subtrees with increasing depth, such that the second player did not colour any vertex in the subtree. However if we add the edge e to E, the second player can make a threat, break the first player strategy and force the draw.

1.1 Tic-tac-toe

This thesis is about the strong version of tic-tac-toe, which is quite an old game. There is evidence, that people in ancient Rome and Egypt played some variations of this game. It is played on a 2-dimensional grid and each player puts his tokens (usually crosses for first player, and rings for the second one) into the points on the grid. A player wins, if he makes a line with his token vertically, horizontally or diagonally (with same length as grid size). There are winning lines as hypergraph edges and tokens as colours. There are many variations of this game, which are more complicated than the basic tic-tac-toe 3×3. The most famous variation is Go-moku (also known as Five in a Row). The grid has size 19×19 (the same as game Go) and a player wins, when he makes lines of length 5. Tic-tac-toe can be generalized into more dimensions. Chapter 4 is about multidimensional tic-tac-toe. This thesis is mainly about the game Qubic, which is 3-dimensional tic-tac-toe played in a combinatorial cube with the edge length 4, winning lines have also length 4.

1.1.1 Qubic solving

For the first time Qubic was solved by Oren Patashnik in 1980 [5]. He used DFS with threat sequence searching for both players. Some plies of the first player were not made by a computer but by Patashnik himself. He called them strategic moves and there were 2929 of them. He supposed that the first player can force the win Qubic. He also wanted a strategy for the first player (if it exists), not only the verification if his assumption was true. It took about 1500 hours to solve it, but about half of the time was wasted because of a hardware (memory) problem and bad selections of strategic moves. His result, that the first player can win the Qubic, was verified by Ken Thompson. His program did not create the strategy and used the same 2929 strategic moves and it took 50 hours. Qubic was solved a second time by Louis V. Allis in 1994 [1]. He introduced two new algorithms for game solving Proof-number search (Pn-search) and Dependency-based search (Db-search). He combined it for solving. The main algorithm was Pn-search, it is a type of best-first search. Db-search was used for threat sequence searching for both players. His program took 15 hours to solve Qubic. We used Allis algorithms (Pn-search and Db-search) and parallelized the Pn-search. Recent computers can solve the Qubic under 1 minute.

2. Game tree searching algorithms

2.1 Proof-Number search

Pn-search was invented by Louis V. Allis and it was introduced in his PhD thesis [1]. It is the best-first search algorithm. It creates a search tree and in every step it expands "the best" node, which should be the node on the shortest path to the goal. Pn-search is very popular and there are many improvements of the basic algorithm.

2.1.1 AND/OR Tree

Pn-search builds AND/OR tree and tries to prove or disprove the root of the tree.

Definition 2.1. *AND/OR tree* is game a tree (nodes represent the states of game, edges represent plies), where first player nodes are OR nodes, and second player nodes are AND nodes.

Each node N of the tree has the following properties:

- Type: AND/OR—denoted as $type(N)$
- Value: denoted as $val(N)$
  - True: first player can force the win from the represented state of the game
  - False: second player can force the win or draw
  - Unknown: value cannot be determined in the actual state of the algorithm
- Proof and Disproof number: denoted as $p(N)$ and $d(N)$

Definition 2.2. To *prove the node* N means to determine, that $val(N)$ is True. To *disprove the node* N means to determine, that $val(N)$ is False.

Let S_N be the set of sons of node N. Leaves of the AND/OR tree are nodes representing terminal states of the game (first or second player wins or it is a draw). If the first player wins, the leaf has value true, if the second player wins or it is a draw, the leaf has value false. The values of the internal node N are counted from values of their sons:

- If $type(N) = \text{OR}$:
  1. If there exists $S_1 \in S_N$, such that $val(S_1) = \text{True}$, then $val(N) = \text{True}$.
  2. If there exists $S_2 \in S_N$, such that $val(S_2) = \text{Unknown}$, then $val(N) = \text{Unknown}$.
  3. If for all $S_3 \in S_N$ $val(S_3) = \text{False}$, then $val(N) = \text{False}$.

- If $type(N)$ = AND:

    1. If there exists $S_1 \in S_N$, such that $val(S_1)$ = False, then $val(N)$ = False.
    2. If there exists $S_2 \in S_N$, such that $val(S_2)$ = Unknown, then $val(N)$ = Unknown.
    3. If for all $S_3 \in S_N$ $val(S_3)$ = True, then $val(N)$ = True.

That is only the formal description of: “the player can win if he can make a such ply, that his opponent can make only plies leading to lose”. The calculation of values and types of nodes resembles the properties of logic operations conjunction and disjunction. To prove OR node, it is needed to prove at least one of its sons. To disprove OR node, it is needed to disprove all of its sons.

Definition 2.3. The *proof number* of the node N is a minimal count of nodes, which have to be proved to prove the node N, the set of these nodes is *proof set* of node N. The *disproof number* of the node N is a minimal count of nodes, which have to be disproved to disproved the node N, the set of these nodes is *disproof set* of node N.

If the $val(N)$ = True, then $p(N) = 0$ (no other node has to be proved to prove the node N) and $d(N) = \infty$ (there is no such set of nodes, where disproving all its nodes leads to disproving the node N). And vice versa if $val(N)$ = False, $p(N) = \infty$ and $d(N) = 0$. If $val(N)$ = Unknown, the proof/disproof numbers are counted from the numbers of sons of N (the calculation of the numbers resembles the calculation of the values and properties of conjunction and disjunction):

- If $type(N)$ =OR:
    - $p(N) = min\{p(S)|S \in S_N\}$
    - $d(N) = \sum\limits_{S \in S_N} d(S)$
- If $type(N)$ = AND:
    - $p(N) = \sum\limits_{S \in S_N} p(S)$
    - $d(N) = min\{d(S)|S \in S_N\}$

If a node representing non-terminal state is created, it gets the value Unknown. It gets an initial proof and disproof number (it cannot be counted from its sons, because the new node has not any sons). The basic initial proof and disproof number is 1 (at least one node has to be prove/disprove to prove/disprove the recently created node). However, usually heuristic initial numbers are used.

2.1.2 Algorithm description

Pn-search creates the root of the AND/OR tree. The root represents some initial state of the game, for which Pn-search counts if the first player can force the win or the second player can force the draw (or win). The initial state can be a blank game (without any plies) or a game where some plies were made as well. And then it repeats the following 3 steps until the root is proved or disproved:

- Select most proving node (MPN) in the tree
- Develop MPN
- Update MPN ancestors

When the algorithm stops and the root is proved, then the first player can force the win from the initial state of the game. If the root is disproved, the second player can force the win or draw. Pseudocode of the algorithm entry point is in Algorithm 1.

```
Procedure PnSearch(Game G)
{
    Data: G—representation of the current game state
    Result: True—if the first player can force the win
      False—if the second player can force the draw
    initialization of the root R;
    while (R is not proved nor disproved)
    {
        Node N = SelectMpn(R);
        DevelopNode(N);
        update proof and disproof numbers of all ancestors of N;
    }
    return val(R);
}
```

Algorithm 1: Entry point of Pn-search

Selecting the most proving node

The question in the best-first algorithm is what is the best node and how to find it. The best node in Pn-search is the most proving node.

Definition 2.4. The *most proving node* in AND/OR tree with root R is the leaf L, which by proving L reduces the $p(R)$ by 1 and by disproving L reduces the $d(R)$ by 1.

Surprisingly, in every AND/OR tree with root R, such that $val(R)$ = Unknown, the most proving node exists. Allis proved a stronger claim in his work [1]:

Theorem 2.1 (Allis). *Let R be the root of AND/OR tree and $val(R)$ = Unknown. Then each pair of smallest proof set of R and disproof set of R has non-empty intersection.*

This theorem proves that the most proving node exists in every AND/OR tree, where root has unknown value. Corollary is that the same effort is needed to prove or to disprove the root. Pseudocode for selecting the most proving node is in Algorithm 2.

Node developing

During developing node N all its possible sons are created and inserted into the tree. The sons of the node N represent the state of the game after one ply after

```
Procedure SelectMpn(Root R)
{
    Data: $R$—root of the AND/OR tree
    Result: Most proving node
    Node $N = R$;
    while ($N$ is not leaf)
    {
        if ($type(N) = OR$)
        {
            $N = S \in S_N$, such that $p(S) = p(N)$
        }
        else
        {
            $N = S \in S_N$, such that $d(S) = d(N)$
        }
    }
    return $N$;
}
```
Algorithm 2: Selecting the most proving node

the state, which is represented by node N. The sons are evaluated after creating. The simple evaluation method is by terminal game state. Let S be the newly created node, then:

1. If the state of S is victory for the first player, then $val(S)$ = True.
2. If the state is victory for the second player or the state is a draw, then $val(S)$ = False.
3. Otherwise $val(S)$ = Unknown.

However, smarter algorithm can be used as the evaluation method. We use Db-search and λ-search. After evaluation the node gets a proof and disproof number by its value.

Update ancestors

After the developing of the node N, its ancestors (nodes on the path from the node N to the root) are updated. Their values and proof and disproof numbers are recalculated by formulas described in Section 2.1.1.

2.1.3 Transposition table

In lot of games the same (or isomorphic) state can be reached by different ways. A transposition table (usually implemented as a hash table) is storage for solved states. Before developing the node, the transposition table is searched. If the solved state is in the table, the node is not developed but immediately evaluated. This reduces the run-time by a large amount, but still some states can be solved more than once in this case:

1. Node J is selected as MPN, it is developed and node $M \in S_J$ represented the state S, which was not solved yet. Therefore the state S is not in the transposition table.

2. Node K is selected as MPN, it is developed and also has son represented the state S. If M was not solved before developing K, state S is not in the transposition table yet. So both nodes J and K have the child representing same state S and it can be solved two times.

To avoid this problem, all states, even unsolved, should be stored in the transposition table. Therefore transposition table is searched during developing node N, before every $M \in S_N$ is created and there are 3 possible cases:

1. The M is not found: node M is a new son of N, and M is stored in the table as unsolved.

2. The M is found as unsolved: add N as a new parent of M (node M has more parents now).

3. The M is found as solved: node M is a new child of N, with known value (True/False).

Now all states are solved once at the most, but the searched graph is not a tree anymore. It is only DAG (directed acyclic graph). However it does not matter very much. The algorithm is still valid, but it can overestimate proof and disproof numbers of some nodes (see Figure 2.1). Pseudocode for node developing is in Algorithm 3.

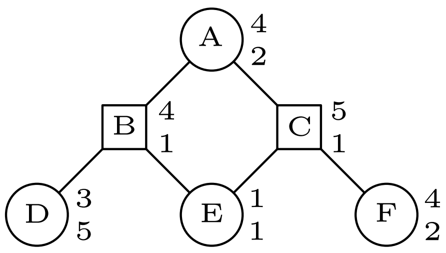


Figure 2.1: Overestimation of proof and disproof numbers in Pn-search on DAG: Circle nodes are OR nodes, square nodes are AND nodes, the top number is the proof number, the bottom one is the disproof number. MPN is clearly node E. If E is disproved, node A is disproved as well, but it has disproved number 2.

```
Procedure DevelopNode(Node $N$)
{
    Data: $N$—node to develop
    GenerataAllSons($N$);
    for (Node $S \in S_N$)
    {
        if (player on turn win)
        {
            if (type($S$) is OR)
            {
                $val(S)$ = True;
            }
            elif (type($S$) is AND)
            {
                $val(S)$ = False;
            }
            save $S$ as solved node in transposition table;
        }
        else
        {
            $val(N)$ = Unknown;
        }
    }
}
Procedure GenerateAllSons(Node $N$)
{
    $G_S$ = states created from $N$ by adding one ply;
    for (Game $G$ in $G_S$)
    {
        search transposition table for $G$;
        if (nothing found in transposition table)
        {
            $C$ = new node represented state $G$;
            add $C$ into $S_N$;
            insert $C$ into transposition table as unsolved node;
        }
        elif (node $C$ is in transposition table as unsolved)
        {
            add $C$ as a new son to $N$;
        }
        elif (node is in transposition table as solved)
        {
            add new solved son to $N$;
        }
    }
}
```

Algorithm 3: Node developing with transposition table

2.1.4 Weak Proof-number search

As is shown in the last section, if every states is solved at most once, the Pn-search graph is DAG and it brings the problem with proof and disproof numbers overestimation. This problem is partly solved by Weak Pn-search, which was published by T. Ueada, T. Hashimoto, J. Hashimoto and H. Iida in 2008 [9]. It is original Pn-search, only proof and disproof numbers calculation is changed. Let k be the count of non-terminal sons of node N.

1. If N is leaf, then it gets the same numbers as in the original Pn-search: Leaves with terminal state of the game get 0 or ∞ by its type. Leaves with non-terminal states get initial values.
2. $\mathit{type}(N) = \text{OR}$:
   - $p(N) = \min\limits_{M \in S_N} \{p(M)\}$
   - $d(N) = \max\limits_{M \in S_N} \{d(M)\} + k - 1$
3. $\mathit{type}(N) = \text{AND}$:
   - $p(N) = \max\limits_{M \in S_N} \{p(M)\} + k - 1$
   - $d(N) = \min\limits_{M \in S_N} \{d(M)\}$

2.2 Dependency-based search

Dependency-based search (Db-search) was published also by L. V. Allis in his PhD thesis [1]. Pn-search tries to find the solution in the quickest way by developing the most proving node. Db-search tries to reduce the size of the searching graph by structured states and by combining the states of the game, which can be played together.

2.2.1 Structure states

Pn-search used atomic states, which means that nodes of the graph represent only single states and nothing else. The structured state also has information on how it was created from the previous state. Therefore we can recognize if two states are independent, i.e. plies lead to that states do not interact. For example, they are on different sides of board and it does not matter, which one is played first. Db-search represents the states of a game as a set of attributes. What is an attribute depends on the game. However in a positional game played on hypergraph $H(V, E)$, an attribute is a pair (v, c), where $v \in V$ and c is a state of the vertex v (empty, first player colour, second player colour). Nodes in Db-search graph represent Db-operators:

Definition 2.5. *Db-operator o* is 3-tuple $(o^{pre}, o^{del}, o^{add})$, where $o^{pre}, o^{del}, o^{add}$ are sets of attributes.

o^{pre} is a precondition, which state S has to comply with to apply Db-Operator o on it (formally: if $o^{pre} \subseteq S$, then o can be applied to S). Applying Db-operator o to state S means to delete attributes from o^{del} and add attributes from o^{add} (formally: $o(S) = (S \backslash o^{del}) \cup o^{add}$). Note that o^{del} is not empty in positional games. For example, if the first player puts a cross on point p, then Db-operator is:

$$\big(\{(p, \text{empty})\}, \{(p, \text{empty})\}, \{(p, \text{cross})\}\big)$$

Point p must be empty, then it is removed from state as an empty point and added as a point with cross.

2.2.2 Paths

Definition 2.6. *Path* $P = (o_1, o_2, \ldots, o_n)$ is a sequence of Db-operators. Path P is *applicable* to state S, if P is empty or o_1 is applicable to S and path $(o_2, \ldots o_n)$ is applicable to $o_1(S)$. *Key operator* of the path is the last one: $\text{key}(P) = o_n$.

Let $\mathbb{P}$ be the set of all applicable paths to initial states.

Definition 2.7. Two paths $P, Q \in \mathbb{P}$ are *equivalent* ($P \equiv Q$) if P is a permutation of Q.

Usually we do not look for every path but only if a path exists. Therefore we do not have to traverse whole state space, but only equivalence classes (elements of $\mathbb{P}/_{\equiv}$). Searching algorithms with transposition table work this way. But Db-search restricts state space even more.

Definition 2.8. Let C be a element of $\mathbb{P}/_{\equiv}$. C is *key class* if: for all $P, Q \in C$: $\text{key}(P) = \text{key}(Q)$.

Definition 2.9. The set of states $\mathbb{S}$ is *singular* if for all $S \in \mathbb{S} : |S| = 1$.

Definition 2.10. The set of path $\mathbb{Q}$ is *monotonous*, if for every path $Q = (o_1, \ldots o_n) \in \mathbb{Q}$ applicable to initial state S holds:

$$(\forall i \neq j : o_i^{add} \cap o_j^{add} = \emptyset) \wedge (\forall i : S \cap o_i^{add} = \emptyset)$$

Definition 2.11. Path $P \in \mathbb{P}$ is a solution, if goal state g exists, such that $g = P(s)$, where s is the initial state.

Allis proves the following theorem in his work [1]:

Theorem 2.2 (Allis). *Let every path $P \in \mathbb{P}$ be monotonous and set of goals be singular. Then set of key classes C is complete, i.e. each solution path $q \in \mathbb{P}$ is element of class in C, and each class in C either consist of only solutions, or no solutions.*

It means that if $\mathbb{P}$ is monotonous and the set of goals is singular then it is sufficient to traverse only key classes. It is easy to change the set of goal states to singular: we add a new state with one new attribute and new operators which change the previous goal states to the new one. But monotonicity of $\mathbb{P}$ depends on the game rule (or another problem which we want to solve). But the set $\mathbb{P}$ for positional games, where coloured vertices are not uncoloured during the game, nor change their colours, is monotonous.

2.2.3 Traversing the key classes

To traverse the key classes Allis defines the meta-operator, which combines and creates new key classes.

Definition 2.12. Let o_1 and o_2 be Db-operators. We define two relations between operators:

1. $o_1 \prec o_2$ (o_1 *supports* o_2) if $o_1^{add} \cap\ o_2^{pre} \neq \emptyset$
2. $o_1 \ll o_2$ (o_1 *precedes* o_2) if $o_1^{pre} \cap\ o_2^{del} \neq \emptyset$

Definition 2.13. Let $P_1, \dots P_n$ be paths applicable to state S. Then the *merge* of the paths $P_1 || \dots || P_n$ is the set of all paths Q applicable to S, such that Q is the permutation of the set of all operators from the merging paths. The merge of equivalence classes is defined as the merge of their representatives: $[P_1]_{\equiv} || \dots || [P_n]_{\equiv} = P_1 || \dots || P_n$.

Definition 2.14. Let $N = \{C_1, \dots C_n\}$ be the set of key classes and all key classes and N are not empty. Let $C = C_1 || \dots || C_n$ and $C \neq \emptyset$. Let o be Db-operator such that:

$$\forall i \in \{1, \dots, n\} : (\text{key}(C_i) \prec o)\ \vee\ (\text{key}(C_i) \ll o)$$

And o is applicable to $P(S)$ ($P \in C, S$ is the initial state). Then o is *valid* in N. *Meta-operator* $M(N, o)$ is applicable if and only if o is valid in N and there is no proper subset $K \subset N$, such as o is valid in K. If $M(N, o)$ is applicable, then $M(N, o) = [P.o]_{\equiv}$, where $P.o$ is path P with operator o added to the end.

Allis proved that meta-operator is sound (it creates only key classes) and complete (every key class can be created by meta-operator). Thus every step of Db-search is a choice of a subset of created key classes, trying to apply meta-operator and adding new key class to the set. However, trying meta-operator to every subset would be very expensive. Fortunately, we will see that it is not necessary to try meta-operator to every subset of created key classes.

2.2.4 Db-search and Qubic

We used Db-search in the same way as Allis. Db-search is started during evaluation node in Pn-search. It searches sequences of threat, which leads to attacker win (threat winning sequence). There are 12 Db-operators for each line and it depends on the player, for which the forced win is searched. Let $\ell = (p_1, p_2, p_3, p_4)$ be a Qubic line, so one of Db-operators o for this line (in case we search the first player forced win) is:

1. $o^{pre} = \{(p_1,\text{cross}), (p_2,\text{cross}), (p_3,\text{empty}), (p_4,\text{empty})\}$
2. $o^{del} = \{(p_3,\text{empty}), (p_4,\text{empty})\}$
3. $o^{add} = \{(p_3,\text{cross}), (p_4,\text{ring})\}$

Other Db-operators for ℓ are the same, only the points are changed. To apply o to line ℓ, the first player must have crosses on points p_1 and p_2 and the other two points must be empty, then he plays one cross on p_3 (he makes a threat) and the second player has only one choice—to play the ring on p_4. Db-operators for the second player's forced win are the same, only crosses and rings are switched.

Qubic meta-operator

It is not necessary to try applying meta-operator to key class sets greater than 2:

1. Db-operator o is valid in key class set N if and only if for every $C \in N$: key(C) $\prec o$. If exists $C \in N$ such as key(C) $\ll o$ and key(C) $\not\prec o$, then o would not be applicable to the merge path, because key(C) delete the empty point (p, empty), which is in o^{pre}.

2. There cannot be 4 different Db-operators o_1, o_2, o_3, o_4 such that $o_1, o_2, o_3 \prec o_4$ that can be applied in one path. Every Db-operator adds one cross (ring) and it needs exactly 2 crosses (rings) to be applied.

To apply meta-operator Db-search repeats two stages: dependency and combination. In the dependency stage the meta-operator tries applying only to sets of size 1. If a meta-operator can be applied to new sets of size 1, it is applied immediately. In the combination stage sets of size 2 are selected for the meta-operator. Key classes created during this stage can not be selected in the same combination stage. Pseudocode for Db-search is in Algorithm 4.

```
Procedure DbSearch(Game G)
{
    Data: G—representation of the game current state
    Result: True—if the attacker can force the win only by threats
      False—otherwise
    initialization of the root R;
    while (new nodes are created)
    {
        apply meta-operator to combination nodes (or root) created in the
        last cycle;
        insert new nodes into tree as dependency nodes;
        if (node, where attacker win, was created)
        {
            return True;
        }
        apply meta-operator to the pairs of dependency node, where at
        least one node was created in the previous step;
        insert new nodes into the tree as combination nodes;
        if (node, where attacker win, was created)
        {
            return True;
        }
    }
    return False;
}
```

Algorithm 4: Db-search algorithm

Failing forced win

There are 3 situations, when Db-search can find invalid forced win:

Definition 2.15. *Defender four* is a state of Qubic, where line ℓ exists, such that every point of ℓ is occupied by defender tokens. *Closed defender three* is a state of Qubic, where line ℓ exists, such that 3 points of ℓ are occupied by defender tokens and the last point of ℓ is occupied by attacker tokens. *Open defender three* is a state of Qubic, where line ℓ exists, such that 3 points of ℓ is occupied by defender tokens and the last point of ℓ is empty.

Each of these situations has to be handled in Db-search:

1. **Defender four**: If the defender creates his line only by forced move, then the attacker forced win fails. There must be a control for defender four in both stages of Db-search.

2. **Closed defender three**: Closed defender three is not a problem after merging. However, a path must exists there, such that when the defender creates open three, the attacker has to close it immediately. This is not a problem in the dependency stage (first must open defend three must be created) but it has to be handled in the combination stage. It is quite expensive to search every path after key classes are merged. So a random path is searched. By this Db-search is not complete: Forced win can exist for some state, but it is not found, because in the random path the attacker does not close the defender open three immediately. However this does not occur very often and Db-search is still very efficient.

3. **Open defender three**: If the defender creates open three, the attacker has to close immediately. This problem is handled during the combination stage. It tries to create the path of Db-operator, which closes every defender open three. If this path does not exist or if the defender creates more than one open three, forced win does not exist.

2.3 λ-search

λ-search was introduced by Thomas Thomsen [8] in 2000. It is described as improvement of alpha-beta pruning algorithm. However, any other game graph searching algorithm can be used. λ-search uses order of threat, denoted as λ. Informally λ is approximately the amount of turns to win. Searching algorithm is executed repeatedly with increasing λ. λ-search is quite popular and there are some improvements of the basic algorithm.

Definition 2.16. *λ^n tree* is a search tree, which contains only λ^n moves. *λ^n_a tree* is λ^n tree, where the attacker moves first. *λ^n move* is a legal move in the game with the following properties:

- If the attacker is to move and the defender passes, then there exists at least one λ^i_a tree with value 1 (attacker win) and $0 \leq i \leq n-1$.

- If the defender is to move, then there does not exist any λ^i_a tree with value 1 and $0 \leq i \leq n-1$.

We used λ-search in the same way as Db-search to evaluate node in Pn-search. It searches if the attacker can win only with λ moves with limited λ and limited depth. Pseudocode for main method of λ-search is in Algorithm 5.

```
Procedure LambdaSearch(Game G)
{
    Data: G—representation of the current game state
    Result: True—if the attacker can force the win with depth limited by
                MAX_DEPTH λ limited by MAX_LAMBDA
      False—otherwise
    initialization of the searching tree;
    for (i = 1 to MAX_LAMBDA)
    {
        if (Lambda(i, MAX_DEPTH))
        {
            return True;
        }
    }
    return False;
}
Procedure Lambda(int Lambda, int Depth)
{
    Data: Lambda—limit for λ in searching
      Depth—limit for searching depth
    Result: True—if the attacker can force the win with input limits
      False—otherwise
    if (Depth ≤ 0)
    {
        return False;
    }
    for (i = 0 to Lambda)
    {
        if (i = 0)
        {
            try to win only with one move;
        }
        else
        {
            if (SearchAlgorithm(i, depth))
            {
                return True;
            }
        }
    }
    return False;
}
```

Algorithm 5: Entry point of λ-search

Any searching algorithm such as *SearchAlgorithm* can be used. Pseudocode for this part is in Algorithm 6.

λ moves are found recursively using the same search algorithm (Algorithm 5).

```
Procedure SearchAlgorithm(int Lambda, int Depth)
{
    Data: Lambda—limit for $\lambda$ in searching
      Depth—limit for searching depth
    Result: True—if the attacker can force the win with input limits
      False—otherwise
    if (Depth $\leq$ 0)
    {
        return False;
    }
    while (LambdaMove(Lambda, Depth) finds $\lambda$ moves M)
    {
        apply M to current state;
        if (attacker can win in SearchAlgorithm(Labmda, Depth - 1)
        {
            return True;
        }
        remove M from current state;
    }
}
```

Algorithm 6: Search algorithm for λ-search

Pseudocode for searching the λ moves is in Algorithm 7.

```
Procedure LambdaMove(int Lambda, int Depth)
{
    Data: Lambda—limit for $\lambda$ in searching
      Depth—limit for searching depth
    Result: $\lambda$ move
    while (exists some valid move $M$)
    {
        apply $M$ to current state;
        if ($M$ is attacker move)
        {
            if (Lambda(Lambda - 1, Depth - 2))
            {
                return $M$;
            }
        }
        else
        {
            if (Lambda(Lambda - 1, Depth - 1))
            {
                return $M$;
            }
        }
        remove $M$ from current state;
    }
}
```

Algorithm 7: Searching for λ moves

3. Solving Qubic

3.1 Properties of Qubic and its game tree

The branching factor at the root of the game tree is 64 and it is decreased by 1 in each level. Maximal depth is also 64. A draw is possible (as you can see in Figure 3.1), so we can not use only strategy stealing for answering the question, if the first player can force the win. The number of all different states of the game is:

$$\sum_{i=0}^{64} \binom{64}{i} \binom{i}{\lceil \frac{i}{2} \rceil} \doteq 410.79 \times 10^{27}$$

Explanation of the formula: First we choose some occupied points: $\binom{64}{i}$. Then we choose, on which points crosses are: $\binom{i}{\lceil \frac{i}{2} \rceil}$ (on the remaining points are rings). It is quite a large number, fortunately we do not have to search all states. Searching algorithm does not create a lot of them, because searching is stopped when one player wins. Qubic has 192 automorphisms, which is quite a lot and it is very useful at the top of the game tree (see Section 3.5.1 and Chapter 4). Threats are very important in Qubic, because it is very easy to make them. A player needs only two tokens in one line and the other two points in the line have to be free and he can make a threat in his turn. Therefore some procedure for searching threats sequence should be in the searching algorithm.

3.2 Choice of algorithm

First we tried to solve Qubic with simple DFS algorithm, but it is a harder problem than we thought. DFS algorithm failed, so we tried better ones. The second algorithm was Pn-search. It failed as well. The first success was Pn-search with Db-search as the evaluation method (Allis used these algorithms in his thesis [1]). The first version of Pn-search uses tree as game graph and it takes 3 $\frac{3}{4}$ minutes to solve. The second version uses DAG (directed acyclic graph). It shortened the computing time by half a minute. However standard Pn-search has a problem with overestimating the proof and disproof number of the DAG nodes. This problem is partly solved by Weak Pn-search. Qubic is solved in 2 $\frac{1}{4}$ minutes by Weak Pn-search on DAG with Db-search as the evaluation method. Then we tried λ-search instead of Db-search and it also failed. λ-search is much more slower and it evaluates less nodes than Db-search. It takes over 10 hours to solve Qubic with λ-search and Db-search together. If only Db-search is used it takes a few minutes. After parallelization it is solved under 1 minute when 8 threads are used.

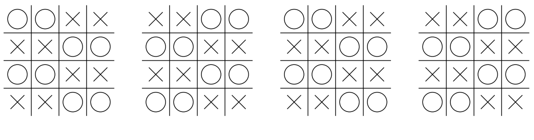


Figure 3.1: Example of draw in Qubic

3.3 Program architecture

The fastest algorithm for Qubic solving, which we implemented, is Pn-search with Db-search as the evaluation method. Pn-search does not start on a blank game. We generated a set of starting games in depth 4 (as Allis [1] did). Pn-search loads one starting game and solves it. Solved and unsolved Pn-search nodes are saved in the transposition table during solving. When the game is solved, the whole Pn-search tree is deleted and unsolved nodes are removed from transposition table. After that Pn-search loads another game. The diagram of architecture is in Figure 3.2. Documentation of the program is in Appendix A.

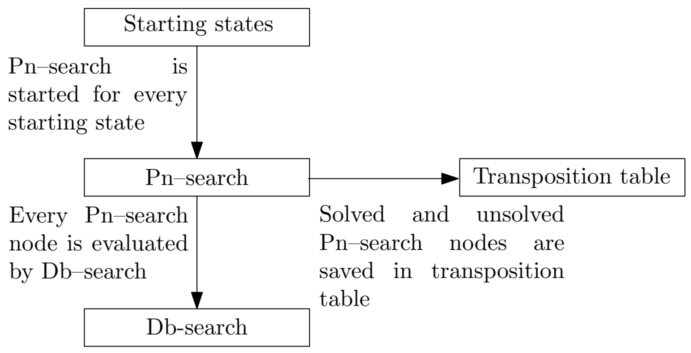

Starting states
Pn–search is started for every starting state
Pn–search
Transposition table
Every Pn–search node is evaluated by Db–search
Solved and unsolved Pn–search nodes are saved in transposition table
Db-search

Figure 3.2: Diagram of single-thread Qubic solver

3.3.1 Parallelization

A lot of work in parallelization of Pn-search exists. As far as we know, A. Kishimoto and Y. Kotani brought the first parallelization of Pn-search [4] in 1999. They tested it on the game Othello and the speed-up rate was 3.6 on 16 distributed processors. Other parallelizations brought better speed-up rate, but usually the procedures of Pn-search are parallelized, like randomized parallel Pn-Search [6] and job-level Pn-search [10] parallelize the nodes expanding and parallel depth first Pn-search parallelizes the selecting of the most proving node. We focus on how to parallelize Pn-search from the "outside". We consider Pn-search as a single-thread task. We solve how it can be executed in more threads and how the threads should be synchronized. Our speed-up rate is 2.7 on quadcore processor with hyperthreading, when we use 8 threads.

If there is more than one starting game, it is not very difficult to parallelize the solving algorithm. Each thread loads one starting game and solves it like in single thread algorithm. When a game is solved, the thread loads another one. The whole solving of one starting game is within one thread, so the most of algorithm is not changed. There are two main changes from single thread to multithread algorithm:

- In loading the starting games
- In transposition table

The starting game loading has to be done with a lock, otherwise one starting game could be solved more than once. The transposition table is divided into two tables (local and global). Every thread has a local table, so it is implemented without locks. It is for unsolved nodes. Local tables are cleared after a thread solved the starting game, because the whole Pn-search tree is deleted. All threads have a common transposition table for solved Pn-search nodes. This table needs a lock, because it is shared with all threads. Items are only inserted into the global table. Therefore the table is locked only for inserting. Reading is done without a lock. A diagram of 2-threads program is in Figure 3.3.

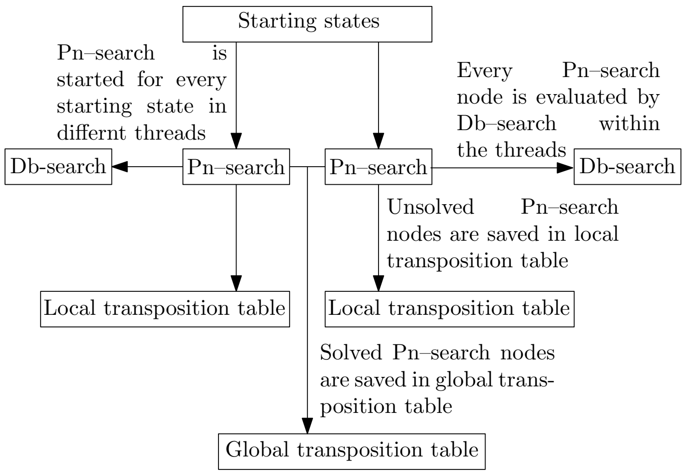

Starting states
Pn–search is started for every starting state in differnt threads
Every Pn–search node is evaluated by Db–search within the threads
Db-search
Pn–search
Pn–search
Db-search
Unsolved Pn–search nodes are saved in local transposition table
Local transposition table
Local transposition table
Solved Pn–search nodes are saved in global transposition table
Global transposition table

Figure 3.3: Diagram of Qubic solver with 2 threads

3.4 Our Result

3.4.1 Starting depth

For Pn-search it is very important to have a not-regular game tree. If the tree is almost regular (nodes have very similar degrees), Pn-search works as standard BFS and it is not very efficient. Hence it is good when the evaluating method evaluates (with the value true or false) some nodes as soon as possible and makes the search tree non-regular. Therefore there is quite a difference (see Table 3.1) when solving starts in depth of 3 (as in Patashnik solution [5]) or 4 (as in Allis solution [1]).

3.4.2 Single-thread algorithms

In Table 3.1 the variants of Pn-search are compared. Nodes counts differed slightly (about hundreds) in every run, because random numbers are used in one part of Db-search. So all nodes counts are rounded to thousands. There were not any significant differences in times (about one second). The program is written in language C and all runs were made on the computer with Intel Core i7-2600 Quadcore 3,4 GHz, 8GB RAM and operation system Ubuntu 10.

3.4.3 Parallel algorithm

Some states can be solved more than once in parallel algorithm by the following process:

		Nodes			
Algorithm	Start depth	Created	Expanded	Evaluated	Time
Pn-search on tree	4	31.002.000	609.000	16.724.000	3:46
Pn-search on DAG	4	21.525.000	419.000	11.298.000	3:10
Weak Pn-search on DAG	4	15.910.000	304.000	7.753.000	2:13
Weak Pn-search on DAG	3	27.862.000	562.000	13.915.000	4:10

Table 3.1: Algorithm comparison: Created/Expanded nodes—nodes created/expanded by Pn-search, Evaluated nodes—nodes evaluated by Db-search (with value true or false) or Pn-search and nodes which get their value from transposition table.

1. Thread T_1 creates node N_1 representing state S and saves it in local table T_{t_1}.
2. Thread T_2 creates node N_2 representing state S and saves it in local table T_{t_2}
3. Thread T_1 solve the node N_1, but N_2 is still in local table T_{t_2}, so it can be solved for the second time.

Solving this problem would be quite complicated. It would mean that the Pn-search tree would have its nodes in different threads. Global table would be more complicated (entries would also have to be deleted not only inserted). Therefore we think that overhead of the treatment this problem would be quite big. Moreover, it does not occur very often and algorithm, where this problem is not handled, gives good result. We counted every event, when solved game is inserted into global table, but the value is already there, to know how many nodes are solved more than once. There are less than a thousand of thees nodes, which is less than per mile of whole count of created Pn-search nodes. In Table 3.2 are the results of parallelized Weak Pn-search on DAG where we changed the number of threads. You can see from the table that it is almost useless to use more than 4 threads for this problem on a quadcore processor. Usage of hyperthreading shortens the computation time only a little. We use the same computer for parallel algorithm as for single-thread algorithm.

Num. of Threads	Hyperthreading	Nodes solved then once	Time
2	yes	578	1:11
4	yes	579	0:50
4	no	578	0:52
8	yes	714	0:48
8	no	709	0:58

Table 3.2: Comparison of parallel algorithm with different numbers of threads

3.5 Discussion

3.5.1 Automorphisms

All created nodes are saved in the transposition table to avoid solving any node more than once. To solve even less nodes, we tried to add automorphisms into searching transposition table. An automorphism image is counted for every node, which is searched in the transposition table. Qubic has 192 automorphism, so counting all images for every created node is quite expensive. We tried to accelerate the image-counting by a pre-generated code. We generated a method for every one automorphism. In the automorphism method, there are only bit operations between Qubic game representation (two 64-bit integer) and integer constants. However, it is still slower than solving without using the automorphisms.

Nonetheless, automorphisms are very important in generating starting games. Our solving algorithm starts with games in depth 4 (with 2 crosses and 2 rings). With assumption that the first player makes his first ply into the points with the highest degree (points on diagonals), there is only one possibility how to start the game. Points in corners and inside the cube are isomorphic. If the second ply of the first player is also to the point with the highest degree, there are only 7 non-isomorphic games with depth 3. If one ring is added to these 7 games, there are only 195 games with depth 4 when the automorphisms are used. If automorphisms are not used in this step there are 422 games created from that 7 3-ply games:

1. For every 3-ply game the second player chooses one of the 61 free points, which give 427 starting games.

2. There are 5 games which were created twice.

And there are 3.812.256 games with depth 4 if automorphisms are not used at all and assumption about first player plies is not made:

1. There are 635.376 (= $\binom{64}{4}$) possibilities how to put 4 tokens into the 64 points.

2. There are 6 possibilities how to put 2 crosses and 2 rings into the 4 given points.

3.5.2 Distributed calculation

The solving algorithm can be easily distributed to more computers. Each computer would get some starting games to solve and after solving it would send results to the server, where the results would be processed. The problem is how to implement the global transposition table. There are two basic ways:

1. Each computer has its own global table and the tables are not synchronized. This is the easier solution for programming and there would not be any synchronization overhead. Nonetheless, maybe many nodes would be solved more than once and this way would be slower than the way with synchronization.

2. The global table is common for all computers. The synchronization of the global table has to be implemented and it would not be very easy.

Verification, which way is faster, would be done by experiments. We did not distribute the calculation, because Qubic is solved under a minute, so distributing the calculation is unnecessary. Other games discussed in next section are quite hard problems and it cannot be solved, even with distributed calculation.

3.5.3 Other games

After success with Qubic we tried the fastest algorithm (Weak Pn-search with Db-search) to solve another two games, which are open problems:

1. **$5 \times 5 \times 5$**: It has the same rules as Qubic, but it is played in a cube with edge length 5 (the length of a winning line is also 5).
2. **3D connect four**: It is played in the same cube as Qubic, but players only choose a column (two coordinates) and the third one is computed as lowest point in the column. In reality it is played with 16 strings (or sticks) in a square and tokens are put on them (a maximum of 4 tokens can be on each string). The winning lines are the same as in Qubic.

Unfortunately, our program failed at both games. Db-search is not as efficient as in Qubic, because creating the threats is more difficult in these games. In $5 \times 5 \times 5$ the player needs three of his tokens in one line to create threads. In 3D connect four the player needs only two of his tokens in one line, but the other two points in that line must be the lowest free points in their columns. In Qubic, Db-search succeeds at almost half of nodes, which are evaluated by Db-search. But in $5 \times 5 \times 5$ and 3D connect four it is only one tenth. It can not solve some deep $5 \times 5 \times 5$ nodes with 25 crosses and 25 rings under 1 hour. Hence solving whole $5 \times 5 \times 5$ with this algorithm and present computers is impossible. We think that 3D connect four can be solved, because it has a smaller state space than $5 \times 5 \times 5$ (even then Qubic). However, it needs more edits in algorithm, so that Db-search would be more efficient.

4. Automorphisms of the game n^d

4.1 Introduction

Automorphisms are often used in game solving, because it is not necessary to solve every position in the game, but only the non-automorphic. It is quite useful to know how many automorphisms the game has and how to generate them. In this chapter we find all automorphisms of multidimensional tic-tac-toe. The d-dimensional tic-tac-toe is played in a d-dimensional combinatorial cube with edge of length n and it is often called game n^d. This chapter is motivated by the game tic-tac-toe, but the automorphisms can be used for any other problem with a combinatorial cube with the same set of lines. This result generalized Silver [7], where all automorphisms were characterized for the game 4^3.

4.2 More formally

Definition 4.1. *Combinatorial cube* n^d is a set of points $[n]^d$, where $[n]$ is the set $\{0, \dots, n-1\}$.

Informally, the combinatorial cube contains points of d-dimensional hypercube, which have integer coordinates. In this chapter we often omit the word combinatorial and call it only cube n^d.

Definition 4.2. Let s be the the sequence $(k_1, \dots, k_n)$. The *type* of sequence s, *type*(s) is:

- $+$ if the sequence s is strictly increasing
- $-$ if the sequence s is strictly decreasing
- c if $k_i = c$ for every $1 \leq i \leq n$
- ? otherwise

Definition 4.3. *Line* ℓ of combinatorial cube n^d is every set of points $\{p^1, p^2, \dots, p^n\}$, such that it can be ordered into the sequence $(q^1, q^2, \dots, q^n)$, such that for each $0 \leq j \leq d$, the sequence $s_j = (q^1_j, q^2_j, \dots q^n_j)$ has *type*$(s) \neq$?. *Type* of line ℓ, *type*(ℓ) is $(\mathit{type}(s_1), \dots, \mathit{type}(s_d))$ and $(\mathit{type}(\ell))_i$ is the i-th element of *type*(ℓ), i.e. *type*(s_i). We denote the set of lines of combinatorial cube n^d by $\mathbb{L}(n^d)$.

Note that for every line $\ell \in \mathbb{L}(n^d)$ there exists at least one $1 \leq j \leq d$, such that *type*$(\ell)_j$ is $+$ or $-$. If every coordinate sequences of line would be constant, the line would be only one point. However, line $\ell \in \mathbb{L}(n^d)$ is defined as a set with n elements. For example points sequences of the game 4^3:

1. $\{[0,0,3], [0,1,2], [0,2,1], [0,3,0]\} \in \mathbb{L}(4^3)$.
2. $\{[0,0,2], [0,1,2], [0,2,2], [0,3,1]\} \notin \mathbb{L}(4^3)$, because the set can not be ordered, such that the sequence of the third coordinates would be increasing, decreasing or constant.

Note that each line $\ell \in \mathbb{L}(n^d)$ has actually 2 types. For example: $type(\ell_1) = (+,-,0,\dots,0)$ and $type(\ell_2) = (-,+,0,\dots,0)$ are types of the same line $\ell = \big\{[i, n-i-1, 0, \dots, 0] | i \in [n]\big\}$.

Definition 4.4. *Dimension of the line* $\ell \in \mathbb{L}(n^d)$ is

$$\dim(\ell) = \big|\{i \in \{1,\dots,d\} | type(\ell)_i \in \{+,-\}\}\big|$$

Degree of the point $p \in n^d$ is

$$\deg(p) = \big|\{\ell \in \mathbb{L}(n^d) | p \in \ell\}\big|$$

Definition 4.5. Two points p_1, p_2 are *collinear*, if there exists line $\ell \in \mathbb{L}(n^d)$, such that $p_1 \in \ell$ and $p_2 \in \ell$.

Definition 4.6. Point $p \in n^d$ is called *corner*, if p has coordinates only 0 and $n-1$. Line $\ell \in \mathbb{L}(n^d)$ is *edge* with $\dim(\ell) = 1$ connecting two corners. 2 corners are *neighbours* if they are connected by arbitrary edge. Line $\ell \in \mathbb{L}(n^d)$ with $\dim(\ell) = n$ is called *main diagonal.* We denote the set of all main diagonals by $\mathbb{L}_m(n^d)$.

Definition 4.7. Point $p = [p_1, \dots, p_d] \in n^d$ is called *outer point* if there exists at least on $i \in \{1,\dots,d\}$, such that $p_i \in \{0, n-1\}$. If point $p \in n^d$ is not outer point, then p is *inner point.*

Definition 4.8. Permutation $P\colon n^d \to n^d$ is *automorphism* if and only if:
$\{v_1, \dots, v_n\} \in \mathbb{L}(n^d) \Rightarrow \{P(v_1), \dots, P(v_n)\} \in \mathbb{L}(n^d)$.

Informally, the automorphism of the game n^d is a permutation of the cube points, which preserves the lines.

Definition 4.9. The point $p \in n^d$ is *fixed* by automorphism a, if $a(p) = p$. The set of points $\{p_1, \dots, p_k\}$ is fixed, if $\{p_1, \dots, p_k\} = \{a(p_1), \dots, a(p_k)\}$.

Note that if some set S is fixed, it does not mean every point of S is fixed.

We denote the set of all automorphism by T_n^d. Note that all automorphism with composition form a group $\mathbb{T}_n^d = (T_n^d, \circ, Id)$. In this chapter we find the generator of the group $\mathbb{T}_n^d$ and then the order of the group. First we count the order of the group $\mathbb{T}_2^d$, whose structure is different from other automorphism groups. Then for better understanding we find the generator for $\mathbb{T}_{2k}^3$ and then the generator and order of the general group $\mathbb{T}_n^d$.

4.2.1 Order of $\mathbb{T}_2^d$

The game 2^d is different from other games, because every two points are collinear. So we have following proposition:

Proposition 4.1. *Order of the group* $\mathbb{T}_2^d$ *is* $(2^d)!$.

Proof. Every permutation on the points of the cube 2^d is an automorphism, because every point is collinear with all others. $\mathbb{T}_2^d$ is subset of $\mathbb{S}_{2^d}$ (symmetry group on 2^d elements), so $|\mathbb{T}_2^d|$ can not be bigger than $(2^d)!$. □

From now we will suppose, that $n > 2$.

4.3 Automorphisms of $(2k)^3$

In this section we find the generator of the group $\mathbb{T}^3_{2k}$. After that we prove it for general cubes. In this section we suppose that $n = 2k$.

4.3.1 Line and point classes

Definition 4.10. We divide lines into 4 sets:

1. *Main diagonals* $\mathbb{L}_m\big((2k)^3\big)$
2. *Face diagonals* $\mathbb{L}_f\big((2k)^3\big) = \{\ell \in \mathbb{L}\big((2k)^3\big) \,|\, \dim(\ell) = 2\}$
3. *Rich lines*
$$\mathbb{L}_r\big((2k)^3\big) = \Big\{\ell \in \mathbb{L}\big((2k)^3\big) \Big| (\dim(\ell) = 1) \wedge (\exists m \in \mathbb{L}_m\big((2k)^3\big) : m \cap \ell \neq \emptyset)\Big\}$$
4. *Poor lines*
$$\mathbb{L}_p\big((2k)^3\big) = \Big\{\ell \in \mathbb{L}\big((2k)^3\big) \Big| (\dim(\ell) = 1) \wedge (\forall m \in \mathbb{L}_m\big((2k)^3\big) : m \cap \ell = \emptyset)\Big\}$$

Observation 4.1. *1. Each point $p \in n^d$ is incident with at least 3 lines $\ell \in \mathbb{L}\big((2k)^3\big)$.*

2. If line $\ell \in \mathbb{L}\big((2k)^3\big)$ with $\dim(\ell) = 1$ intersects with face diagonal $f \in \mathbb{L}_f\big((2k)^3\big)$, then there exists another face diagonal $g \in \mathbb{L}_f\big((2k)^3\big), g \neq f$, such that $\ell \cap g \neq \emptyset$.

3. If line $\ell \in \mathbb{L}\big((2k)^3\big)$ with $\dim(\ell) = 1$ intersects with main diagonal $m \in \mathbb{L}_m\big((2k)^3\big)$, then there exists another main diagonal $k \in \mathbb{L}_m\big((2k)^3\big), k \neq m$, such that $\ell \cap k \neq \emptyset$.

Definition 4.11. We define 3 sets of points:

1. Point $p \in (2k)^3$ is called *rich point*, if $\deg(p) = 7$.
2. Point $p \in (2k)^3$ is called *common point*, if $\deg(p) = 4$.
3. Point $p \in (2k)^3$ is called *poor point*, if $\deg(p) = 3$.

Lemma 4.1. *1. Main diagonal contains n rich points.*

2. Face diagonal contains 2 rich points and $n - 2$ common points.

3. Rich line contains 2 rich points and $n - 2$ common points.

4. Poor line contains 4 common points and $n - 4$ poor points.

Proof. 1. Without loss of generality type of main diagonal $m \in \mathbb{L}_m\big((2k)^3\big)$ is *type*$(m) = (+, +, -)$. Each point $p_i = [i, i, n - i - 1] \in m$ is contained:

- In 3 rich lines $r^i_1, r^i_2, r^i_3 \in \mathbb{L}_r\big((2k)^3\big)$:
  - *type*$(r^i_1) = (+, i, n - i - 1)$

  - $type(r_2^i) = (i, +, n-i-1)$
  - $type(r_3^i) = (i, i, -)$
- In 3 face diagonal $f_1^i, f_2^i, f_3^i \in \mathbb{L}_f\big((2k)^3\big)$:
  - $type(f_1^i) = (+, +, n-i-1)$
  - $type(f_2^i) = (i, +, -)$
  - $type(f_3^i) = (+, i, -)$
- In the main diagonal m itself.

2. Without loss of generality the type of face diagonal $f \in \mathbb{L}_f\big((2k)^3\big)$ is $type(f) = (+, -, c)$, where $c \in [n]$. f intersects two main diagonals $m_1, m_2 \in \mathbb{L}_m\big((2k)^3\big)$:

   - $type(m_1) = (+, -, +)$ and $q_1 = m_1 \cap f = \big\{[c, n-c-1, c]\big\}$
   - $type(m_2) = (+, -, -)$ and $q_2 = m_2 \cap f = \big\{[n-c-1, c, c]\big\}$

   Hence f contains two rich points q_1, q_2. Every point $p_i = [i, n-i-1, c] \in f$ is contained in 3 rich lines $r_1^i, r_2^i, r_3^i \in \mathbb{L}_r\big((2k)^3\big)$:

   - $type(r_1^i) = (+, n-i-1, c)$
   - $type(r_2^i) = (i, +, c)$
   - $type(r_3^i) = (i, n-i-1, +)$

   Therefore the remaining points $p_i \in f$, such that $p_i \neq q_1, q_2$ are common points (they are on the face diagonal f and 3 rich lines r_1^i, r_2^i, r_3^i).

3. Without loss of generality the type of rich line $r \in \mathbb{L}_r\big((2k)^3\big)$ is $type(r) = (+, 0, 0)$. r intersects 2 main diagonals $m_1, m_2 \in \mathbb{L}_m\big((2k)^3\big)$:

   - $type(m_1) = (+, +, +)$ and $q_1 = m_1 \cap r = \big\{[0, 0, 0]\big\}$
   - $type(m_2) = (-, +, +)$ and $q_2 = m_2 \cap r = \big\{[n-1, 0, 0]\big\}$

   Hence r contains 2 rich points q_1 and q_2. Rich line r intersects face diagonal $f_i \in \mathbb{L}_f\big((2k)^3\big)$ in every point $p_i = [i, 0, 0] \in r$: $type(f_i) = (i, +, +)$. Therefore the remaining points $p_i \in r$, such that $p_i \neq q_1, q_2$, are common points.

4. Without loss of generality the type of poor line $\ell \in \mathbb{L}_p(n^d)$ is $type(\ell) = (+, 1, 2)$. Line ℓ intersects 4 face diagonal $f_1, f_2, f_3, f_4 \in \mathbb{L}_f((2k^3))$:

   - $type(f_1) = (+, +, 2)$ and $q_1 = f_1 \cap \ell = \big\{[1, 1, 2]\big\}$
   - $type(f_2) = (+, -, 2)$ and $q_2 = f_2 \cap \ell = \big\{[n-2, 1, 2]\big\}$
   - $type(f_3) = (+, 1, +)$ and $q_3 = f_3 \cap \ell = \big\{[2, 1, 2]\big\}$
   - $type(f_4) = (+, 1, -)$ and $q_4 = f_4 \cap \ell = \big\{[n-3, 1, 2]\big\}$

   Points q_1, q_2, q_3 and q_4 are common points. Remaining points $p \in \ell, p \notin \{q_1, q_2, q_3, q_4\}$ are poor points and there are incident with only 3 poor lines. □

	Main diagonals	Face diagonals	Rich lines	Poor lines
Rich point	1	3	3	0
Common point	0	1	3	0
Poor point	0	0	0	3

Table 4.1: Count of lines, which are incident with points

	Rich points	Common points	Poor point
Main diagonal	n	0	0
Face diagonal	2	$n-2$	0
Rich line	2	$n-2$	0
Poor line	0	4	$n-4$

Table 4.2: Count of points, which are incident with lines

Counts of incidences of points and lines of the cube $(2k)^3$ are in Table 4.1 and Table 4.2.

Definition 4.12. *Star* ℓ^* of the line $\ell \in \mathbb{L}(n^d)$ is set:

$$\ell^* = \{p \in n^d | p \notin \ell \wedge \exists q \in \ell : p \text{ and } q \text{ are collinear}\}$$

Lemma 4.2. *Let* $f \in \mathbb{L}_f\big((2k)^3\big)$ *and* $r \in \mathbb{L}_r\big((2k)^3\big)$. *Then:*

- $|f^*| = 2n(n+1) - 6$,
- $|r^*| = 3n(n-1)$.

Proof. Without loss of generality, *type*$(f) = (+,+,0)$. Star f^* for the cube 4^3 is depicted in Figure 4.1. Points in f^* are from:

- Face $F = \big\{[x,y,0] | x,y \in [n]\big\}$, which contributes $n(n-1)$ points.
- Rich lines $R = \big\{r \in \mathbb{L}_r\big((2k)^3\big) | r \cap f \neq \emptyset, r \not\subset F\big\}$. Rich lines from R are orthogonal to f. $|R| = n$. Hence lines from R add $n(n-1)$ points to the star f^*.
- Face diagonals $D = \big\{d \in \mathbb{L}_f\big((2k)^3\big) | d \cap f \neq \emptyset\big\}$. There are 4 face diagonals in D:
    - *type*$(f_1) = (+,0,+), f_1 \cap f = \big\{[0,0,0]\big\}$
    - *type*$(f_2) = (n-1,+,-), f_2 \cap f = \big\{[n-1,n-1,0]\big\}$
    - *type*$(f_3) = (0,+,+), f_3 \cap f = \big\{[0,0,0]\big\}$
    - *type*$(f_4) = (+,n-1,-), f_4 \cap f = \big\{[n-1,n-1,0]\big\}$

  $f_1 \cap f_2 = \big\{[n-1,0,n-1]\big\}$ and $f_3 \cap f_4 = \big\{[0,n-1,n-1]\big\}$, so D adds $4(n-1)-2$ to the star f^*.

Note that points from main diagonals $m_1, m_2 \in \mathbb{L}_m\big((2k)^3\big)$, which intersect d, are in R. When we add the numbers of points from sets F, R, D we get the size of the star $|f^*| = n(n-1) + n(n-1) + 4(n-1) - 2 = 2n^2 + 2n - 6$.

Without loss of generality, type of r is *type*$(r) = (+,0,0)$. Star r^* the cube 4^3 is depicted in Figure 4.2. Points in r^* are from:

F	F	F	f
F	F	f	F
F	f	F	F
f	F	F	F

		f_4	R
		R	f_2
f_3	R		
R	f_1		

	f_4		R
f_3		R	
	R		f_2
R		f_1	

$f_3 f_4$			R
		R	
	R		
R			$f_1 f_2$

Figure 4.1: Star f^* of the face diagonal $f \in \mathbb{L}_f(4^3)$

- Faces $F_1 = \{[x, y, 0] | x, y \in [n]\}$ and $F_2 = \{[x, 0, z] | x, z \in [n]\}$. Faces F_1 and F_2 add $2n(n-1)$ points to the star r^*.
- Face diagonals $D = \{d \in \mathbb{L}_f((2k)^3) | d \cap r \neq \emptyset, d \not\subset F_1, F_2\}$. $|F| = n$, so D adds $n(n-1)$ points to the star r^*.

Note that points from main diagonals $m_1, m_2 \in \mathbb{L}_m((2k)^3)$, which intersect r, are in D. After adding the numbers we get $|r^*| = 3n(n-1)$.

F_1	F_1	F_1	F_1
F_1	F_1	F_1	F_1
F_1	F_1	F_1	F_1
r	r	r	r

D	D	D	D
F_2	F_2	F_2	F_2

D	D	D	D
F_2	F_2	F_2	F_2

D	D	D	D
F_2	F_2	F_2	F_2

Figure 4.2: Star r^* of rich line $r \in \mathbb{L}_r(4^3)$

$\square$

Note that $|f^*| = |r^*|$ if and only if $n = 2$ or $n = 3$.

Definition 4.13. Set of lines S in *invariant* if for every automorphism $a \in \mathbb{T}_n^d$: $\ell \in S \Rightarrow a(\ell) \in S$.

Theorem 4.1. *Sets* $\mathbb{L}_m((2k)^3), \mathbb{L}_f((2k)^3), \mathbb{L}_r((2k)^3), \mathbb{L}_p((2k)^3)$ *are invariant.*

Proof. Every automorphism has to preserve a degree of points, therefore the main diagonals have to be mapped onto the main diagonals, the poor lines have to be mapped onto the poor lines (by Lemma 4.1). Every automorphism has to preserve the size of the star for each line, therefore the face diagonals have to be mapped onto the face diagonals and the rich lines have to be mapped onto the rich lines (by Lemma 4.2). $\square$

4.3.2 Generator of $\mathbb{T}_{2k}^3$

In this section we find the generator of the group $\mathbb{T}_{2k}^3$. We use 2 types of automorphism:

1. Group of rotations $\mathbb{R}$ of 3-dimensional cube. The generator of $\mathbb{R}$ are rotations:
   - $R_x([x, y, z]) = [x, n-z-1, y]$
   - $R_y([x, y, z]) = [n-z-1, y, x]$
   - $R_z([x, y, z]) = [n-y-1, x, z]$

2. Group of permutation automorphisms $\mathbb{F}_n$ (called flexible group in Silver's proof [7]). This group contains the mappings:

$$F_\pi\big([c_1, \dots, c_d]\big) = [\pi(c_1), \dots, \pi(c_d)]$$

where $\pi \in \mathbb{S}_n$, such that it has *symmetry* property: if $\pi(p) = q$ then $\pi(n-p-1) = n-q-1$.

In proofs we use some easy observations:

Observation 4.2. *If automorphism $a \in \mathbb{T}_n^d$ fixes two collinear points $p, q \in n^d$, it also fixes the line $\ell \in \mathbb{L}(n^d)$, such that $p, q \in \ell$.*

Observation 4.3. *If two lines $\ell_1, \ell_2 \in \mathbb{L}(n^d)$ are fixed by $a \in \mathbb{T}_n^d$, their intersection, point $p = \ell_1 \cap \ell_2$, is fixed.*

Definition 4.14. The group of automorphism $\mathbb{A}_{2k}^3$ is generated by elements from $\mathbb{R} \cup \mathbb{F}_{2k}$.

We prove that $\mathbb{A}_{2k}^3 = \mathbb{T}_{2k}^3$. The idea of the proof, that resembles similar proof of Silver [7], is composed of 2 steps:

1. For any automorphism $t \in \mathbb{T}_{2k}^3$ we find automorphism $a \in \mathbb{A}_{2k}^3$, such that $t \circ a$ fixes every point from certain set S.

2. If an automorphism $a' \in \mathbb{T}_{2k}^3$ fixes every point from S, then a' is identity.

Hence for every $t \in \mathbb{T}_{2k}^3$ we find an inverse element t', such that t' is composed only by elements from $\mathbb{R} \cup \mathbb{F}_{2k}$, therefore $t \in \mathbb{A}_{2k}^3$. Before the main proof, we prove a technical lemma, which we formulate and prove for arbitrary size and dimension of the cube.

Lemma 4.3. *Let F be the front 2-dimensional face of combinatorial cube n^d, i.e. $F = \big\{[x, y, 0, \dots, 0] | x, y \in [n]\big\}$ and automorphism $\alpha \in \mathbb{T}_n^d$ fix all 4 corners of F, i.e. points $[0, \dots, 0], [n-1, 0, \dots, 0], [0, n-1, 0, \dots, 0], [n-1, n-1, 0, \dots, 0]$. Then if α fixes point $[i, 0, \dots, 0], i \in [n]$ it also fixes point $[n-i-1, 0, \dots, 0]$.*

Proof. α fixes all 4 corners, therefore it fixes both diagonals $d_1, d_2 \subset F$ (by Observation 4.2). Type of d_1 and d_2 are $type(d_1) = (+, +, 0, \dots, 0)$ and $type(d_2) = (-, +, 0, \dots, 0)$.

Suppose that α fixes point $p = [i, 0, \dots, 0]$, where $i \in \{1, \dots, n-2\}$ (corners are already fixed). We show that the point $p_5 = [n-i-1, 0, \dots, 0]$ is fixed in 3 steps (note that if $i = n-i-1$ the case is trivial). Fixed points in face 7×7 are depicted in Figure 4.3.

1. We show that $p_1 = [i, i, 0, \dots, 0]$ is fixed by α. $\alpha(p_1)$ must be on d_1 and it must be collinear with p. There are 2 collinear points with p on d_1: $[i, i, 0, \dots 0]$ and $[0, \dots, 0]$, but the second one is already fixed as a corner, so p_1 is fixed. The point $p_2 = [i, n-i-1, 0, \dots, 0] \in d_2$ is fixed by a similar argument.

2. We show that α fixes point $p_3 = [n-i-1, i, 0, \dots, 0]$. $\alpha(p_3)$ must be on d_2 and it must be collinear with p_1. There are two points on d_2 collinear with p_1: p_2 and p_3, but p_2 is fixed from step 1. The point $p_4 = [n-i-1, n-i-1, 0, \dots, 0]$ is fixed by a similar argument.

3. Now line $\ell_1 = \{[n-i-1, j, 0, \dots, 0] | j \in [n]\}$ is fixed (by Observation 4.2). Line $\ell_2 = \{[j, 0, \dots, 0] | j \in [n]\}$ is also fixed, therefore point p_5 is fixed (by Observation 4.3).

□

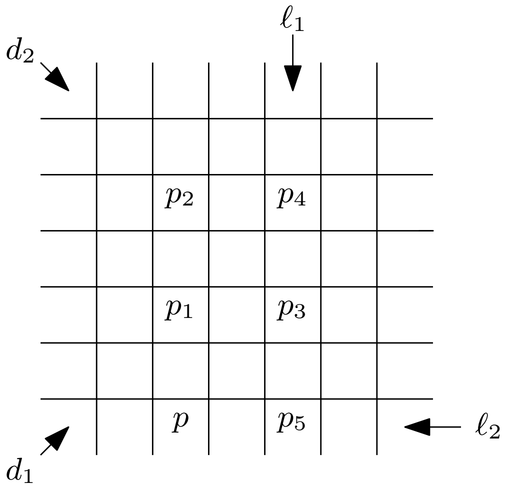


Figure 4.3: How to fix points by diagonals in front 2-dimensional face

Theorem 4.2. *For all $t \in \mathbb{T}^3_{2k}$ exists $a \in \mathbb{A}^3_{2k}$, such that $t \circ a$ fixes all corners and every points of line $\ell = \{[i, 0, 0] | i \in [n]\}$.*

Proof. First we find automorphism $a' \in \mathbb{A}^3_{2k}$, such that a' fixes all corners.

1. We start with point $p_0 = [0,0,0]$. $t(p_0)$ has to be on main diagonal (by Theorem 4.1). Without loss of generality $t(p_0) = [i, i, n-i-1]$, where $i \in [n]$. So we take $f_\pi \in \mathbb{F}_n$, such that:

   - $\pi(i) = 0, \pi(0) = i$
   - $\pi(n-i-1) = n-1, \pi(n-1) = n-i-1$
   - $\pi(k) = k$ otherwise

   Therefore $t \circ f(p_0)$ is corner. Then we take $r_1 \in \mathbb{R}$, such that automorphism $a_1 = t \circ f \circ r_1$ fixes p_0.

2. Line $a_1(\ell)$ must be mapped onto rich line $r \in \mathbb{L}_r\big((2k)^3\big)$, such that $p_0 \in r$. If corner $p_1 = [n-1, 0, 0]$ is fixed by a_1 we take $a_2 = a_1$. Otherwise it can be mapped onto $[0, n-1, 0]$ (or $[0, 0, n-1]$). We take rotation $r_2\big([x,y,z]\big) = [y,x,z]$ (or $[z,y,x]$). So $a_2 = a_1 \circ r_2$ fixes corners p_1 and p_0, note that $r_2\big([0,0,0]\big) = [0,0,0]$.

3. If corner $p_2 = [0, n-1, 0]$ is fixed by a_2 we take $a_3 = a_2$. Otherwise it can be mapped only onto $[0, 0, n-1]$. We take rotation $r_3\big([x,y,z]\big) = [n-x-1, n-z-1, n-y-1]$ and permutation automorphism f_σ, where $\sigma(i) = n-i-1$. Hence $a_3 = a_2 \circ r_3 \circ f_\sigma$ fixes points:

   - $p_0 : a_2 \circ r_3 \circ f_\sigma\big([0,0,0]\big) = r_3 \circ f_\sigma\big([0,0,0]\big) = f_\sigma\big([n-1, n-1, n-1]\big) = [0,0,0]$

- $p_1 : a_2 \circ r_3 \circ f_\sigma\big([n-1,0,0]\big) = r_3 \circ f_\sigma\big([n-1,0,0]\big) = f_\sigma\big([0,n-1,n-1]\big) = [n-1,0,0]$
- $p_2 : a_2 \circ r_3 \circ f_\sigma\big([0,n-1,0]\big) = r_3 \circ f_\sigma\big([0,0,n-1]\big) = f_\sigma\big([n-1,0,n-1]\big) = [0,n-1,0]$

4. Corner $p_3 = [0,0,n-1]$ is fixed automatically, because it is connected by rich line to p_0 and other corners connected to p_0 by rich line (p_1 and p_2) are already fixed.

5. Now we show that corner $p_4 = [n-1,n-1,n-1]$ is fixed by a_3. There are 3 face diagonals incident with p_1:

   (a) $d_1 = \big\{[n-i-1,i,0] | i \in [n]\big\}$. d_1 is fixed, because $p_1, p_2 \in d_1$.

   (b) $d_2 = \big\{[n-i-1,0,i] | i \in [n]\big\}$. d_2 is fixed, because $p_1, p_3 \in d_2$.

   (c) $d_3 = \big\{[n-1,i,i] | i \in [n]\big\}$. d_3 is fixed because it cannot be mapped onto any other face diagonal incident with p_1.

   Therefore $p_4 \in d_3$ is fixed by a_3.

6. Corners $[n-1,n-1,0], [n-1,0,n-1]$ and $[0,n-1,n-1]$ are fixed by a similar argument as in previous step. Note that each corner is incident with 3 edges. So edges can be used as face diagonals were used in previous step.

Automorphism $a' = a_3$ fixes all corners of the cube $(2k)^3$. Now we find the automorphism a, which fixes all corners and the points on line ℓ. Line ℓ is fixed, because points p_0 and p_1 are fixed by a' (by Observation 4.2). Let $q_i = [i,0,0]$ and $q'_i = [n-i-1,0,0]$. We construct the automorphism a by induction by $i \in \{0,\dots,k\}$:

1. $i = 0$. Let automorphism $y_0 = a'$. It fixes all corners (points q_i and q'_i are also corners).

2. $i > 0$. We have automorphism y_{i-1} by induction hypothesis, which fixes all corners and every point of set $Q_{i-1} = \{q_j, q'_j | j \in \{0,\dots,i-1\}\}$. Note that Q_0 contains only corners $[0,0,0]$ and $[n-1,0,0]$. If $y_{i-1}(q_i) = q_i$, then $y_i = y_{i-1}$. Otherwise $y_{i-1}(q_i) = q_j$, where $i < j < n-i-1$, because points from Q_{i-1} are already fixed. Let us consider $f^i_\pi \in \mathbb{F}_{2k}$, where:

   - $\pi(j) = i, \pi(i) = j$
   - $\pi(n-j-1) = n-i-1, \pi(n-i-1) = n-j-1$
   - $\pi(k) = k$, where $k \notin \{j, n-j-1, i, n-i-1\}$

   Automorphisms $y_i = y_{i-1} \circ f^i_\pi$ fixes:

   (a) All corners, because automorphism y_{i-1} fixes all corners by induction hypothesis and $\pi(0) = 0$ and $\pi(n-1) = n-1$.

   (b) Set Q_{i-1}, because automorphism y_{i-1} fixes the set Q_{i-1} by induction hypothesis and for all $k < i$ and $k > n-i-1 : \pi(k) = k$.

   (c) Point q_i: $y_{i-1} \circ f^i_\pi\big([i,0,0]\big) = f^i_\pi\big([j,0,0]\big) = [i,0,0]$.

(d) Point q'_i by Lemma 4.3.

So automorphism $a = y_k$ fixes all points of line ℓ and all corners of the cube. $\square$

Theorem 4.3. *If automorphism $\alpha \in \mathbb{T}^3_{2k}$ fixes all corners of the cube and line $\ell = \{[i,0,0] | i \in [n]\}$, then α is identity.*

Proof. Face diagonals d_1 and d_2 of front face $F = \{[x,y,0] | x,y \in [n]\}$ are fixed, because all corners are fixed (by Observation 4.2). Let $p \in d_1 \cup d_2$, such that p is not corner. Then p is collinear with the only one point $q \in \ell$, such that q is not corner. Therefore p is fixed. Every line of front face F has two fixed points, because every line of F crosses d_1 and d_2. Therefore every line of front face F is fixed (by Observation 4.2). Every point of front face F is fixed, because every point is the intersection of at least two fixed lines (by Observation 4.3). Points of other faces can be fixed by similar arguments. Every line $m \in \mathbb{L}((2k)^3)$ is fixed, because every outer point is fixed, and every line has at least two outer points. Every inner point is fixed, because it is intersection of at least 3 lines. $\square$

4.4 Automorphisms of n^d

4.4.1 Corners, main diagonals and edges

In this section we show lemmas how every automorphism $a \in \mathbb{T}^d_n$ maps main diagonals, edges and corners.

Lemma 4.4. *Every automorphism $a \in \mathbb{T}^d_n$ maps main diagonal $m \in \mathbb{L}_m(n^d)$ onto main diagonal $m' \in \mathbb{L}_m(n^d)$.*

Proof. For n even, the proof is trivial. Every point $p \in m$ has a maximal degree, which is $2^n - 1$. Every automorphism $a \in \mathbb{T}^d_n$ has to preserve the point degree. So the point p has to be mapped onto the point $p' \in m'$.

Now we prove the proposition for n odd. Let $\alpha = \frac{n-1}{2}$. The center of the cube $c = [\alpha, \alpha, \dots, \alpha]$ is always mapped onto c (it is the only point with degree $\frac{3^d-1}{2}$). Therefore the main diagonal $m \in \mathbb{L}_m(n^d)$ has to be mapped onto line $\ell \in \mathbb{L}(n^d)$, such that $c \in \ell$. Without loss of generality $type(\ell) = (+, \dots, +, -, \dots, -, \alpha, \dots, \alpha)$. Let $p \in \ell$, such that $p \neq c, p = [i, \dots, i, n-i-1, \dots, n-i-1, \alpha, \dots, \alpha] = [p_1, p_2, \dots, p_d]$, where $i \in [n], i \neq \alpha$. Let us split the coordinates of p into 2 blocks: i-block (coordinates i and $n-i-1$) and α-block. Let k be the size of the i-block. Now we count the degree of p. There are 2 types of the lines incident with p:

1. The lines, which have non-constant coordinate sequences in i-block. Let I be the coordinate indices of i-block, $I = \{1, \dots, k\}$. Every $J \subseteq I, J \neq \emptyset$ defines line m_J, such that $p \in m_J$ and $type(m_J)_j, j \in \{1, \dots, d\}$ is:
   - $type(m_J)_j = p_j$, if $j \notin J$
   - $type(m_J)_j = +$, if $j \in J \wedge p_j = i$
   - $type(m_J)_j = -$, if $j \in J \wedge p_j = n-i-1$

For example, $J = \{a, b\}$

$$type(m_J) = (i, \dots, \underset{a}{+}, \dots, i, n-i-1, \dots, \underset{b}{-}, \dots, n-i-1, \alpha, \dots, \alpha)$$

Therefore the number of these lines is equal to the number of non-empty subset of I, which is $2^k - 1$.

2. The line, which has non-constant coordinate sequences in α-block. Let I be the set of coordinate indices of α-block, $I = \{k+1, \dots, d\}$. Every $J \subseteq I, J \neq \emptyset$ defines line m_J, such that $p \in m_J$ and $type(m_J)_j, j \in \{1, \dots, d\}$ is:

   - $type(m_J)_j = p_j$, if $j \notin J$
   - $type(m_J)_j = +$, if $j \in J$

   However, every $K \subseteq J$ defines line m'_{JK}, such that $p \in m'_{JK}$ and $type(m'_{JK})_j, j \in \{1, \dots, d\}$ is:

   - $type(m'_{JK})_j = type(m_J)_j$, if $j \notin K$
   - $type(m'_{JK})_j = -$, if $j \in K$

   For example:

$$type(m'_{JK}) = (i, \dots, i, n-i-1, \dots, n-i-1, \alpha, \dots, \alpha, \overbrace{+, \dots, +, \underbrace{-, \dots, -}_{K}}^{J})$$

   We have $\sum\limits_{i=1}^{d-k} \binom{d-k}{i}$ choices for set J. For every J of size i, we have $\sum\limits_{j=0}^{i} \binom{i}{j}$ choices for $K \subseteq J$. Every line is created twice by this way, therefore the number of these lines is: $\frac{1}{2} \sum\limits_{i=1}^{d-k} \binom{d-k}{i} \sum\limits_{j=0}^{i} \binom{i}{j} = \frac{3^{d-k}-1}{2}$.

Therefore the degree of non-central points $p \neq c$, such that $p \in \ell, c \in \ell$ is $\deg(p) = 2^k - 1 + \frac{3^{d-k}-1}{2}$, where $k = \left|\left\{i \in \{1, \dots, d\} | p_i \neq \alpha\right\}\right|$. The degree of the non-central point $q \neq c$ on main diagonal $m \in \mathbb{L}_m(n^d)$ is $\deg(q) = 2^d - 1$. We show, that if $k \neq d$ then $2^k - 1 + \frac{3^{d-k}-1}{2} \neq 2^d - 1$. For contradiction let us suppose that $2^d - 2^k = \frac{3^{d-k}-1}{2}$ and rewrite the formula into binary numbers:

$$\begin{array}{rr}
2^d & 1\overbrace{0\dots\dots\dots0}^{d} \\
-2^k & -1\overbrace{0\dots0}^{k} \\
\hline
\text{Thus } \frac{3^{d-k}-1}{2} & \overbrace{1\dots1\underbrace{0\dots0}_{k}}^{d} \quad = \beta
\end{array}$$

It is trivial to prove by induction, that 4 divides $3^{d-k} - 1$ if and only if $d - k$ is even. β must be even so $d - k$ must be also even. We use the divisibility by 3 test in the binary system[1] for $\gamma = 2\beta + 1$ (it should equal to 3^{d-k}). The binary

[1]The divisibility test is well known problem of Algebra.

number is divisible by 3 if and only if sum E of even order digits and sum O of odd order digits are equal (mod 3). Note that,

$$\gamma = \overbrace{1\dots1}^{d-k}\overbrace{0\dots0}^{k}1,$$

$d-k$ is even so digits sums of the orders 1 to d are equal, but $|E-O| = 1$ (because the 1 at order 0). Therefore γ is not divisible by 3, which is a contradiction. □

Lemma 4.5. *Let $\alpha \in \mathbb{T}_n^d$, e be the edge and c be the corner, such that $c \in e$. If corner c is fixed by α, then $\alpha(e) = e'$ is edge, such that $c \in e'$.*

Proof. First we prove the lemma for n odd. Let $\alpha \in \mathbb{T}_n^d$ be the automorphism, such that corner c is fixed by α. Let e be the edge, such that $c \in e$ and line $\ell = \alpha(e)$. Without loss of generality, $type(e) = (+, 0, \dots, 0)$ and

$$type(\ell) = (\overbrace{+, \dots, +}^{k}, 0, \dots, 0).$$

Let $p = [1, 0, \dots, 0] \in e$ and

$$q = \alpha(p) = [\overbrace{i, \dots, i}^{k}, 0, \dots, 0], \text{where } i \in [n], i \notin \{0, \frac{n-1}{2}, n-1\}$$

Degree of point q can be deduced by similar way as in the proof of Lemma 4.4. Each non-empty subset the set of coordinate indices i-block and 0-block defines a line, hence $\deg(q) = 2^k - 1 + 2^{d-k} - 1$. Degree of p is $\deg(p) = 2^{d-1}$. Note that $\deg(p) = \deg(q)$ if and only if $k = 1$ or $k = d-1$. Therefore dimension of the line ℓ is 1 or $d-1$. If $d = 2$ the proof is finished.

Let us suppose $d > 2$ and $\dim(ell) = d-1$. The second corner $c_1 \in e, c_1 \neq c$ has to be mapped on some corner. The center point c' of e (the point, which have some coordinates equal to $\frac{n-1}{2}$) has to be mapped on the center of ℓ, because it must be collinear with the cube center and there are only 3 points on e collinear with the cube center: 2 corners c and c_1 and the line center c'. We know the degree of line center: $d(k) = 2^{d-k} - 1 + \frac{3^k-1}{2}$, where k is the dimension of the line. Therefore $\deg(\alpha(c')) = 1 + \frac{3^{d-1}-1}{2}$. However, $\deg(c') = 2^{d-1}$ and $\deg(\alpha(c')) > \deg(c')$, if $d > 2$, which is a contradiction and line ℓ must have dimension 1.

Now we complete the proof for n even. Without loss of generality fixed corner c has coordinates $[0, \dots, 0]$. Let $L = \{\ell \in \mathbb{L}(n^d) | c \in \ell\}$. Lines from L has to be mapped on lines from L. We compute the size of star ℓ^* for each line $\ell \in L$. We take a point $p \in \ell \in L$. Without loss of generality, it has coordinates:

$$p = [p_1, \dots, p_d] = [\overbrace{i, \dots, i}^{k}, 0, \dots, 0], \text{where } i \in [n]$$

There are two blocks of coordinates: 0-block and i-block. Let I be the set of coordinate indices of i-block, $I = \{1, \dots, k\}$, and K be the set of coordinate indices of 0-block, $K = \{k+1, \dots, d\}$. There are 3 types of lines incident with p:

1. Lines with increasing coordinate sequences in i-block. Every $J \subset I, J \neq \emptyset$ defines line ℓ_J with type:

- $type(\ell_J)_j = p_j$, if $j \notin J$
- $type(\ell_J) = +$, if $j \in J$

For example:

$$type(\ell_J) = (\underbrace{\overbrace{+,\dots,+}^{}}_{J}\overbrace{,i,\dots,i}^{I},0,\dots,0)$$

There are $\sum\limits_{j=1}^{k-1}\binom{k}{j} = 2^k - 2$ choices for $J \subset I$. We cannot choose whole I as J, because we would get the original line ℓ. Every line adds $n-1$ points into the star and there are n choices for i. However, every point is counted twice in the following way. Let q be the point:

$$q = [\overbrace{c,\dots,c,d,\dots d}^{I},0,\dots 0]$$

Let C be the set of c-coordinate indices and D be the set of d-coordinate indices. Point q is added for the first time, when $C = J \subset I$ and q is added for the second time, when $D = J \subset I$. If $J \neq C, D$ point p cannot be added because $p \notin \ell_J$. So these lines add $n(n-1)(2^{k-1}-1)$ points to the star ℓ^*.

2. Lines with increasing sequences in 0-block. Every $J \subseteq K, J \neq \emptyset$ defines line ℓ_J with type:

   - $type(\ell_J)_j = p_j$, if $j \notin J$
   - $type(\ell_J)_j = +$, if $j \in J$

   Let $\ell_J^1 \in \mathbb{L}(n^d)$, such that

$$type(\ell_J^1) = (\overbrace{c,\dots c}^{I},\overbrace{+,\cdots +}^{J},0,\dots,0)$$

   Let $\ell_J^2 \in \mathbb{L}(n^d)$, such that

$$type(\ell_J^2) = (\overbrace{d,\dots d}^{I},\overbrace{+,\dots,+}^{J},0,\dots,0)$$

   Note that, lines ℓ_J^1 and ℓ_J^2 have an empty intersection when $c \neq d$. Therefore lines of this type add

$$n(n-1)\sum_{j=1}^{d-k}\binom{d-k}{j} = n(n-1)(2^{d-k}-1)$$

   points to star ℓ^*.

3. There are some remaining lines incident with corners $q = [0,\dots,0]$ and

$$r = [\overbrace{n-1,\dots,n-1}^{I},0,\dots,0]$$

   These lines are analogous to face diagonals from the set D in proof of the first part of Lemma 4.2. Let $J_1 \subset I, J_1 \neq \emptyset$ and $J_2 \subseteq K, J_2 \neq \emptyset$. Each pair of J_1 and J_2 defines lines $\ell^1_{J_1J_2}, \ell^2_{J_1J_2} \in \mathbb{L}(n^d)$ with type:

- $type(\ell^1_{J_1J_2})_j = +, type(\ell^2_{J_1J_2})_j = n-1$, if $j \in J_1$
- $type(\ell^1_{J_1J_2})_j = +, type(\ell^2_{J_1J_2})_j = -$, if $j \in J_2$
- $type(\ell^1_{J_1J_2})_j = 0, type(\ell^2_{J_1J_2})_j = +$, if $j \in I$ and $j \notin J_1$
- $type(\ell^1_{J_1J_2})_j = 0, type(\ell^2_{J_1J_2})_j = 0$, if $j \in K$ and $j \notin J_2$

Note that $q \in \ell^1_{J_1J_2}$ and $r \in \ell^2_{J_1J_2}$. And $\ell^1_{J_1J_2} \cap \ell^2_{J_1J_2} = [x_1, \dots x_d]$:

- $x_j = 0$, if $j \notin J_1 \cup J_2$
- $x_j = n-1$, if $j \in J_1 \cup J_2$.

For example:

$$\begin{array}{rcl} & & \overbrace{\qquad\qquad\qquad\qquad}^{I} \quad \overbrace{\qquad\qquad\qquad\qquad}^{K} \\ type(\ell^1_{J_1J_2}) & = & (0,\dots \quad +,\dots \quad 0,\dots \quad +,\dots) \\ type(\ell^2_{J_1J_2}) & = & (+,\dots \quad n-1,\dots \quad 0,\dots \quad -,\dots) \\ \ell^1_{J_1J_2} \cap \ell^2_{J_1J_2} & = & [0,\dots \quad \underbrace{n-1,\dots}_{J_1} \quad 0,\dots \quad \underbrace{n-1,\dots}_{J_2}] \end{array}$$

We cannot choose J_1 as whole I nor empty. If $J_1 = \emptyset, I$, we will get lines from step 2. We cannot choose J_2 empty. If $J_2 = \emptyset$ we will get lines from step 1. We have $\sum_{j=1}^{k-1} \binom{k}{j} = 2^k - 2$ choices how to select J_1 and $\sum_{j=1}^{d-k} \binom{d-k}{j} = 2^{d-k} - 1$ how to select J_2. Each pair of J_1 and J_2 adds $2n-3$ points to the star ℓ^*.

Two lines of different types can have an intersection only on line ℓ. Therefore to get $|\ell^*|$, we can simply add numbers of points from all steps and we get function s, which determines the size of star ℓ^* by dimension of the line $\ell \in L$:

$$s(k) = n(n-1)(2^{d-k} + 2^{k-1} - 2) + (2n-3)(2^d - 2^{d-k+1} - 2^k + 2)$$

We count the second derivative of the function s:

$$\begin{aligned} s''(k) &= \log^2(2)(2^{d-k}n(n-1) + 2^{k-1}n(n-1) - 2^k(2n-3) - 2^{d-k+1}(2n-3)) \\ &= \log^2(2)((2^{d-k} + 2^{k-1})(n-2)(n-3)) \end{aligned}$$

Hence $s''(k) > 0$ when $n > 3$. Function s is convex and $s(k) = s(d-k+1)$, therefore $s(1) \neq s(k)$, when $k \notin \{1, d\}$. Every automorphism $a \in \mathbb{T}^d_n$ has to preserve the size of the star of every line. Therefore the edge e incident with the fixed corner c has to be mapped on ℓ, such that ℓ is edge or main diagonal. However by Lemma 4.4, line ℓ can not be the main diagonal, therefore ℓ must be the edge.

□

4.5 Generator of the group $\mathbb{T}^d_n$

In this section we find the generator of the group $\mathbb{T}^d_n$. We use 3 types of automorphism:

1. Group of rotations of d-dimensional hypercube $\mathbb{R}_d$. Generators of this group are rotations:

$$R_{ij}\big([c_1, \dots, c_i, \dots, c_j, \dots, c_d]\big) = [c_1, \dots, n - c_j - 1, \dots, c_i, \dots, c_d]$$

   for every $i, j \in \{1, \dots, d\}$.

2. Group of permutation automorphisms $\mathbb{F}_n$ (same as in Section 4.3.2).

3. Group of axial symmetry $\mathbb{X}$. In this group there are two automorphisms: Id and $X\big([c_1, \dots, c_{d-1}, c_d]\big) = [c_1, \dots, c_d, c_{d-1}]$.

Definition 4.15. Group $\mathbb{A}_n^d$ is generated by elements from $\mathbb{R}_d \cup \mathbb{F}_n \cup \mathbb{S}$.

We prove that $\mathbb{A}_n^d = \mathbb{T}_n^d$ by the same two steps as we proved $\mathbb{A}_{2k}^3 = \mathbb{T}_{2k}^3$:

1. For any automorphism $t \in \mathbb{T}_n^d$ we find automorphism $a \in \mathbb{A}_n^d$, that $t \circ a$ fixes every point from certain set S.

2. If an automorphism $a' \in \mathbb{T}_n^d$ fixes every point from S, then a' is identity.

Theorem 4.4. *For all $t \in \mathbb{T}_n^d$ exists $a \in \mathbb{A}_n^d$ such that $t \circ a$ fixes every corner of the cube n^d and every point on line $\ell = \big\{[i, 0, \dots, 0] | i \in [n]\big\}$.*

Proof. First we create automorphism $a' \in \mathbb{A}_d^n$, such that $t \circ a$ fixes all corners. We start with point $p_0 = [0, \dots, 0]$. $t(p_0)$ has to be on the main diagonal (by Lemma 4.4). We choose $f \in \mathbb{F}_n$, such that $t \circ f(p_0)$ is corner. Then we choose $r \in \mathbb{R}_d$, such that $t \circ f \circ r(p_0) = p_0$. Now we choose rotations $x_i \in \mathbb{R}_d$ to fix points $p_i = [0, \dots \underset{i}{n - 1}, \dots 0]$ by induction by i.

1. $i = 0$, point p_0 is fixed by $x_0 = t \circ f \circ r$.

2. For $i > 0$, by induction hypothesis we have automorphism $x_{i-1} \in \mathbb{A}_n^d$, such that x_{i-1} fixes all points from set $P_{i-1} = \{p_k | 0 \leq k \leq i - 1\}$. Corner $p_i = [0, \dots, \underset{i}{n - 1}, \dots, 0]$ is mapped onto $p_j = [0, \dots, \underset{j}{n - 1}, \dots, 0]$, because edges incident with p_0 is mapped onto edges incident with p_0 (by Lemma 4.5) and $i \leq j$, because points from P_{i-1} are already fixed. If $x_{i-1}(p_i) = p_i$ we choose $x_i = x_{i-1}$. Otherwise we choose 2 rotations and compose them with x_{i-1}:

   (a) $R_{ji}\big([c_1, \dots, c_i, \dots, c_j, \dots, c_d]\big) = [c_1, \dots, c_j, \dots, n - c_i - 1, \dots, c_d]$:
   $R_{ji}(p_j) = R_{ji}\big([0, \dots, \underset{i}{0}, \dots, \underset{j}{n - 1}, \dots, 0]\big)$
   $= [0, \dots, \underset{i}{n - 1}, \dots, \underset{j}{n - 1}, \dots, 0]$

   (b) $R_{dj}\big([c_1, , \dots c_j, \dots, c_d]\big) = [c_1, \dots, c_d, \dots, n - c_j - 1]$:
   $R_{jd}\big([0, \dots, \underset{i}{n - 1}, \dots, \underset{j}{n - 1}, \dots 0]\big) = [0, \dots, \underset{i}{n - 1}, \dots, 0]$

   Hence $x_i \circ R_{ji} \circ R_{dj}$ fixes p_i and all points of P_i because rotations R_{ji} and R_{dj} do not effect first $i - 1$ coordinates. Note that it also fixes p_0.

In this way we can fix all corners p_i for $i \in \{0, \ldots, d-2\}$. If x_{d-2} fixes p_{d-1}, then $x_{d-1} = x_{d-2}$. Otherwise p_{d-1} is mapped onto p_d and then $x_{d-1} = x_{d-2} \circ X$, where $X \in \mathbb{X}$ and $X \neq Id$. So x_{d-1} fixes all points of P_{d-1} and corner p_d is fixed automatically because there is no other possibility, where corner p_d can be mapped.

We prove that $a' = x_{d-1}$ fixes all corners $c = [c_1, \ldots, c_d]$ by induction by $k(c) = \big|\{i \in [n], c_i = n-1\}\big|$:

1. Corners c, such that $k(c) \in \{0, 1\}$ are already fixed by a'.

2. Corners c, such that $k(c) > 1$, has coordinates without loss of generality:

$$c = [\overbrace{n-1, \ldots, n-1}^{k(c)}, 0, \ldots, 0]$$

   We take neighbours n_1, n_2 of corner c:

   (a) $n_1 = [\overbrace{n-1, \ldots, n-1}^{k(c)-1}, 0, \ldots, 0]$

   (b) $n_2 = [0, \overbrace{n-1, \ldots, n-1}^{k(c)-1}, 0, \ldots, 0]$

   Corners n_1 and n_2 have two common neighbours: c and

$$n_3 = [0, \overbrace{n-1, \ldots, n-1}^{k(c)-2}, 0, \ldots 0]$$

Corners n_1, n_2 and n_3 are fixed by induction hypothesis. Therefore corner c is also fixed, because it must be the neighbour of n_1 and n_2. The automorphism to fix points on line ℓ is constructed in the same way as in proof of Theorem 4.2. We find the automorphism a, which fixes all corners and the points on line ℓ by induction. We start with automorphism a'. In step i of induction we compose automorphism from step $i-1$ and automorphism $f_i \in \mathbb{F}_n$, which fixes points $[i, 0, \ldots, 0]$ and $[n-i-1, 0, \ldots, 0]$. □

Theorem 4.5. *If automorphism $\alpha \in \mathbb{T}_n^d$ fixes all corners of the cube n^d and all points of edge e, then α is identity.*

Proof. We prove it by induction by dimension d of the cube n^d:

1. $d = 2$. Main diagonals are fixed, because all corners are fixed (by Observation 4.2). Let point $p \in m \in \mathbb{L}_m(n^2)$, such that p is not a corner. p is collinear with only one point $q \in e$, such that q is not corner. Hence p is fixed. Therefore every line $\ell \in \mathbb{L}(n^2)$ is fixed, because every line intersects both main diagonals (by Observation 4.2). So every point $p \in n^2$ is fixed because every point is an intersection of at least two lines (by Observation 4.3).

2. $d > 2$ and suppose the theorem holds for all dimensions smaller then d. Without loss of generality, $e = \big\{[i, 0, \ldots, 0] | i \in [n]\big\}$ We take face

$$F = \big\{[x_1, \ldots, x_{d-1}, 0] | x_1, \ldots, x_{d-1} \in [n]\big\}$$

Face F has dimension $d-1$, so all points of F are fixed by induction hypothesis. Then we take all faces G of dimension $d-1$, such that $F \cap G \neq \emptyset$. Corners $c \in G$ are fixed. At least one edge f exists, such that $f \subseteq F \cap G$. Therefore points of f are also fixed and points $p \in G$ are fixed by induction hypothesis. By this argument we show that every outer point is fixed. Every line $\ell \in \mathbb{L}(n^d)$ is fixed, because every line contains at least 2 outer points (by Observation 4.2). Therefore every point $q \in n^d$ is fixed, because every point is an intersection of at least 2 lines (by Observation 4.3).

□

4.6 Order of the group $\mathbb{T}_n^d$

In the previous section we found the generator of the group $\mathbb{T}_n^d$. Now we compute the order of the group.

Lemma 4.6. *Orders of the basic groups are:*

1. $|\mathbb{R}_d| = 2d|\mathbb{R}_{d-1}| = 2^{d-1}d!, |\mathbb{R}_2| = 4$
2. $|\mathbb{F}_n| = \prod_{i=0}^{\lfloor \frac{n}{2} \rfloor - 1} (n-2i)$
3. $|\mathbb{X}| = 2$

Proof.

1. Size of hypercube rotation group $\mathbb{R}_d$ is well known. The first equality can be deduced as follows:
   (a) Hypercube n^d has $2d$ faces of dimension $d-1$ and each $(d-1)$-dimensional face can be rotated to the front.
   (b) Every $(d-1)$-dimensional face of the cube n^d has $|\mathbb{R}_{d-1}|$ rotations.

   The second equality can easily be proved by induction.
2. Size of the group $\mathbb{F}_n$ is the count of permutation $\pi \in \mathbb{S}_n$ with symmetry property. For image of the first element we have n possibilities how to choose, for the second element we have $n-2$ possibilities and so on.
3. Size of $\mathbb{X}$ is clearly 2.

□

Lemma 4.7. *Groups $\mathbb{R}_d$ and $\mathbb{F}_n$ commute and groups $\mathbb{X}$ and $\mathbb{F}_n$ commute.*

Proof. Let $R_{ij} \in \mathbb{R}_d$ and $F_\pi \in \mathbb{F}_n$, then:

1. $R_{ij} \circ F_\pi\big([c_1, \dots, c_i, \dots, c_j, \dots, c_d]\big) = F_\pi\big([c_1, \dots, n-c_j-1, \dots, c_i, \dots, c_d]\big)$
   $= [\pi(c_1), \dots, \pi(n-c_j-1), \dots, \pi(c_i), \dots, \pi(c_d)]$
   $= [\pi(c_1), \dots, n-\pi(c_j)-1, \dots, \pi(c_i), \dots, \pi(c_d)]$
2. $F_\pi \circ R_{ij}\big([c_1, \dots, c_i, \dots, c_j, \dots, c_d]\big)$
   $= R_{ij}\big([\pi(c_1), \dots, \pi(c_i), \dots, \pi(c_j), \dots, \pi(c_d)]\big)$
   $= [\pi(c_1), \dots n-\pi(c_j)-1, \dots, \pi(c_i), \dots, \pi(c_d)]$

Proof that $\mathbb{X}$ and $\mathbb{F}_n$ commute is analogous. □

Lemma 4.8. *If d is odd then $\mathbb{R}_d \cap \mathbb{F}_n = \{ \ Id \ \}$. If d is even then $\mathbb{R}_d \cap \mathbb{F}_n = \{ \ Id \ , F_\sigma\}$, where $F_\sigma \in \mathbb{F}_n$, such that $\sigma(i) = n - i - 1$.*

Proof. Every rotation preserve the order of points on line $\ell = \big\{[i, \ldots, i] | i \in [n]\big\}$. There are two permutation automorphisms, which preserve the order on the line ℓ: identity and F_σ. Let $\mathit{flip}(i) = n - i - 1$, where $i \in [n]$, then each rotation R can be written by operations *flip* and permutation $\pi_R \in \mathbb{S}_d$. For example rotation $R_{12}([c_1, c_2, \ldots, c_d]) = [n - c_2 - 1, c_1, \ldots, c_d]$ can be written as $R_{12}([c_1, c_2, \ldots, c_d]) = [\mathit{flip}(c_{\pi(1)}), c_{\pi(2)}, \ldots, c_{\pi(d)}]$, where $\pi = (12)$.

- If d is odd, $F_\sigma \notin \mathbb{R}_d$. For contradiction let us suppose that $F_\sigma \in \mathbb{R}_d$. For every rotation R_{ij} is permutation $\pi_{R_{ij}} \in \mathbb{S}_d$ a transposition and π_{F_σ} is identity. Therefore F_σ must be composed of even number of R_{ij}. Every R_{ij} does exactly one *flip* operation. Hence F_σ has to do even number of *flip* operations, which is a contradiction, because d is odd and F_σ must do an odd number of *flip* operations.

- If d is even, $F_\sigma \in \mathbb{R}_d$. We composed F_σ as follows: for each pair $\{i, j\}$, such that $i \in \{1, \ldots, d\}$ is odd and $j = i + 1$ we use rotation $R_{ij} \circ R_{ij} = R^2_{ij}([c_1, \ldots, c_i, c_j, \ldots c_d]) = [c_1, \ldots n - c_i - 1, n - c_j - 1 \ldots c_d]$. When we compose all these rotations we get automorphism F_σ. We know that there are not any other automorphism in $\mathbb{R}_d \cap \mathbb{F}_n$.

□

Lemma 4.9. *Group $\mathbb{X}$ can be generated by elements from groups $\mathbb{R}_d$ and $\mathbb{F}_n$ if and only if d is odd.*

Proof. Let $X \in \mathbb{X}$. The case $X = \mathit{Id}$ is trivial. Further suppose that $X\big([c_1, \ldots, c_{d-1}, c_d]\big) = [c_1, \ldots, c_d, c_{d-1}]$.

- If d is odd we use rotations:

  1. $R_{d-1,d}\big([c_1, \ldots c_{d-1}, c_d]\big) = [c_1, \ldots, n - c_d - 1, c_{d-1}]$
  2. Let $S = (1, 2, \ldots, d-2, d)$ and S_i be the i-th element of S. Note that sequence S has even length $d - 1$. We make pair $\{S_i, S_j\}$, such that $i \in \{1, \ldots, d-1\}$ is odd and $j = i + 1$. For each pair $\{S_i, S_j\}$ we use rotation $R^2_{S_i S_j}$ (like in proof of Lemma 4.8).

  After the composing of these rotations we get:

  $$\overline{R}\big([c_1, \ldots, c_{d-1}, c_d]\big) = [n - c_1 - 1, \ldots, n - c_d - 1, n - c_{d-1} - 1]$$

  We use F_σ (same as in Lemma 4.8) and get automorphisms $X = \overline{R} \circ F_\sigma$.

- If d is even, we use a similar argument as in proof of Lemma 4.8. Every automorphism generated by elements from $\mathbb{R}_d$ and $\mathbb{F}_n$ can be written as $R \circ F$, where $R \in \mathbb{R}_d, F \in \mathbb{F}_n$, because $\mathbb{R}_d$ and $\mathbb{F}_n$ commute (by Lemma 4.7). For contradiction let us suppose $X = R \circ F$. We can use only identity and F_σ (like in Lemma 4.8) as F, because no coordination change its value (only two coordinations switch their positions) and rotations can revert only *flip* operation (like in proof of Lemma 4.8). $\pi_X \in \mathbb{S}_d$ is the transposition on coordinates, therefore it must be composed of odd number of R_{ij} rotations. Therefore R must do an odd number of *flip* operation. After using identity or F_σ as F, there still is an odd number of *flipped* coordinates, which is a contradiction.

□

Lemma 4.10. *Let $X \in \mathbb{X}$, such that $X \neq Id$. Then for all $R_1 \in \mathbb{R}_d$ exists $R_2 \in \mathbb{R}_d$, such that $R_1 \circ X = X \circ R_2$.*

Proof. It is sufficient to prove it for rotation R_{ij}.

- If $\{i,j\} \cap \{d-1,d\} = \emptyset$, then the proof is obvious and $R_{ij} \circ X = X \circ R_{ij}$.
- Let us suppose $\big|\{i,j\} \cap \{d-1,d\}\big| = 1$ and $i = d-1$ (proofs of other cases are similar):
$R_{d-1,j} \circ X\big([c_1,\dots,c_j,\dots,c_{d-1},c_d]\big) = [c_1,\dots,c_{d-1},\dots,c_d,n-c_j-1]$
$= X \circ R_{dj}\big([c_1,\dots,c_j,\dots,c_{d-1},c_d]\big)$
- Let us suppose the last possibility $i = d-1$ and $j = d$:
$R_{d-1,d} \circ X\big([c_1,\dots,c_{d-1},c_d]\big) = [c_1,\dots c_{d-1},n-c_d-1]$
$= X \circ R_{d-1,d}\big([c_1,\dots,c_{d-1},c_d]\big)$

□

Theorem 4.6. *If d is odd, then $|\mathbb{T}_d^n| = |\mathbb{R}_d||\mathbb{F}_n|$.*

Proof. Every automorphism generated by elements from $\mathbb{R}_d$ and $\mathbb{F}_n$ can be written as $R \circ F$, where $R \in \mathbb{R}_d, F \in \mathbb{F}_n$ (by Lemma 4.7). Furthermore $\mathbb{R}_d \cap \mathbb{F}_n = \{\ Id\ \}$ (by Lemma 4.8), therefore these automorphisms form the direct product $\mathbb{R}_d \times \mathbb{F}_n$. The group $\mathbb{X}$ is a subset of $\mathbb{R}_d \times \mathbb{F}_n$ (by Lemma 4.9). Therefore the group $\mathbb{T}_d^n = \mathbb{R}_d \times \mathbb{F}_n = \{r \circ f | r \in \mathbb{R}_d, f \in \mathbb{F}_n\}$ and $|\mathbb{T}_d^n| = |\mathbb{R}_d||\mathbb{F}_n|$. □

Theorem 4.7. *If d is even, then $|\mathbb{T}_d^n| = 2(|\mathbb{R}_d|-1)(|\mathbb{F}_n|-1)$.*

Proof. Let $A_n^d = \big\{(R,F) | R \in \mathbb{R}_d, F \in \mathbb{F}_n\big\}$. If we create automorphism a for every pair $(R,F) \in A_n^d$, such that $a = R \circ F$, we generate some automorphisms twice, because $F_\sigma \in \mathbb{R}_d \cap \mathbb{F}_n$ (by Lemma 4.8). We remove the pairs, which contain F_σ. Let $B_n^d = A_n^d \setminus \big\{(R,F) \in A_n^d | R = F_\sigma \vee F = F_\sigma\big\}$. We have to add F_σ to the B_n^d, so all pairs form group $\mathbb{P}_n^d$ and $|\mathbb{P}_n^d| = |\mathbb{R}_d||\mathbb{F}_n| - |\mathbb{R}_d| - |\mathbb{F}_n| + 1 = (|\mathbb{R}_d|-1)(|\mathbb{F}_n|-1)$. However, non-identity automorphism $X \in \mathbb{X}$ is not in this group (by Lemma 4.9). Every automorphism generated by elements from $\mathbb{R}_d \cup \mathbb{F}_n \cup \mathbb{X}$, can be written as $R \circ F \circ X$, where $R \in \mathbb{R}_d, F \in \mathbb{F}_n, X \in \mathbb{X}$ (by Lemma 4.7 and Lemma 4.10). Therefore $\mathbb{T}_n^d = \{p \circ s | p \in \mathbb{P}_n^d, s \in \mathbb{S}\}$ and $|\mathbb{T}_n^d| = 2(|\mathbb{R}_d|-1)(|\mathbb{F}_n|-1)$. □

Corollary 4.1. *1. If d is odd, then:*

$$|\mathbb{T}_d^n| = 2^{d-1}d! \prod_{i=0}^{\lfloor \frac{n}{2} \rfloor - 1} (n-2i)$$

2. *If d is even, then:*

$$|\mathbb{T}_d^n| = 2(2^{d-1}d! - 1)(\prod_{i=0}^{\lfloor \frac{n}{2} \rfloor - 1} (n-2i) - 1)$$

3. *The groups $\mathbb{T}_{2k}^d$ and $\mathbb{T}_{2k+1}^d$ are isomorphic.*

Conclusion

We implemented some algorithms for Qubic solving and compared them. As the main algorithm we used Pn-search and we confirmed, that Weak Pn-search is faster on DAG than the original Pn-search. We used Db-search and λ-search as the evaluation method for new Pn-search node. Db-search is very efficient for Qubic. It is a very good algorithm for searching the threat winning sequences, which are very important in Qubic. However λ-search is not suitable for this problem. In comparison with Db-search, λ-search is extremely inefficient.

We parallelized Pn-search and found that it is not very difficult to implement it, when there are more than one starting positions. Only a few edits suffices in single-thread Pn-search. Transposition table has to be changed to handle multi-thread access. It is needed to implement how starting positions are distributed to each threads, where they are solved by Pn-search. There a lot of things to research in parallelization. We think that it would be interesting to use GPU for parallel Pn-search and other searching algorithms.

We tried to solve the game 5^3 and 3D connect four, which are open problems. However, our program did not solve these games. It is harder to create threats in these games so Db-search is not as efficient as it is for Qubic. Nonetheless, we only changed our program for Qubic. We think that 3D connect four can be solved by present computers, but it would need an algorithm which more resembles the rules of 3D connect four.

In Chapter 4 we characterized all automorphisms of combinatorial cube n^d with set of the line, which is used for multidimensional tic-tac-toe. This is a generalization of Silver [7]. As corollary we know:

- the number of automorphism of every cube n^d
- the groups of automorphisms of cubes with even and odd dimension have different structure
- the groups of automorphisms of cubes with the size $2k$ and $2k + 1$ are isomorphic

Hence automorphisms are known for every cube n^d with finite n and d. It would be interesting to find the automorphisms of the cube with infinite dimension. Would the automorphisms be same even for uncountable dimension? Lines of the cube n^d cannot be generalized to infinite n, because of decreasing coordinate sequences. We think that similar proof can be used to characterize the automorphisms of the toroidal grid, which is a combinatorial cube n^d with lines defined as follows:

Definition 4.16. *Line of toroidal grid ℓ* is every set of points of a combinatorial cube $n^d, \ell = \{p^1, \dots, p^n\}$, such that for every $1 \leq j \leq d$ set $S_j = \{p^1_j, \dots, p^n_j\}$ can be ordered into a sequence $T_j = (t^1_j, \dots, t^n_j)$, such that $\mathit{type}(T_j) \neq ?$.

Note that every line $\ell \in \mathbb{L}(n^d)$ is also the line of the toroidal grid and every automorphism $t \in \mathbb{T}^d_n$ is an automorphism of the toroidal grid as well.

Bibliography

[1] ALLIS, L. Victor. *Searching for Solutions in Games and Artificial Intelligence.* PhD thesis. University of Limburg, Maastricht, The Netherlands, 1994. ISBN 9090074880.

[2] BECK, Jószef. *Lectures on Positional Games.* Rutgers University, New Brunswick, New Jersey, U.S.A, 2003.

[3] KANEKO, Tomoyuki. *Parallel Depth First Proof Number Search.* Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence. The AAAI Press, Menlo Park, California, 2010. ISBN 978-1-57735-463-5.

[4] KISHIMOTO, A., KOTANI Y. *Parallel AND/OR tree search based on proof and disproof numbers.* Joho Shori Gakkai Shinpojiumu Ronbunshu, vol. 99, no. 14., pp 24–30. ISSN 1344-0640.

[5] PATASHNIK, Oren. Qubic: $4 \times 4 \times 4$ Tic-Tac-Toe. *Mathematics Magazine*, Vol. 53, No. 4 (Sep.,1980), pp. 202-216.

[6] SAITO, Jahn-Takeshi, WINANDS, H. M. Winands, VAN DEN HERIK, H. Jaap. *Randomized Parallel Proof-Number Search.* Advances in Computer games. Springer Berlin, Heidelber, 2010, pp. 75–87. ISBN 978-3-642-12992-6.

[7] SILVER, Roland. *The group of Automorphisms of the Game of 3-Dimensional Ticktacktoe.* The American Mathematical Monthly, Vol. 74, No. 3 (Mar., 1967), pp. 247–254.

[8] THOMSEN, Thomas *Lambda-Search In Game Trees – With Application To Go.* Computers and Games. Springer, Berlin, Heidelberg, 2001, pp. 19–38. ISBN 978-3-540-43080-3.

[9] UEDA, Toru, HASHIMOTO, Tsuyoshi, HASHIMOTO, Junichi, IIDA, Hiroyuki *Weak Proof-Number search.* Proceedings of the 6th international conference on Computers and Games pp. 157–168. Springer-Verlag Berlin, Heidelberg 2008. ISBN 978-3-540-87607-6.

[10] WU, I-Chen, LIN, Hung-Hsuan, LIN, Ping-Hung, SUN, Der-Johng, CHAN, Yi-Chih, CHEN, Bo-Ting. *Job-Level Proof-Number Search for Connect6.* Computer and Games. Springer, Berlin, Heidelberg, 2011, pp 11–22. ISBN 978-3-642-17927-3.

List of Figures

List of Tables

List of Abbreviations

- AI—Artificial Intelligence
- BFS—Breadth-first Search
- DAG—Directed Acyclic Graph
- Db-search—Dependency-based Search
- DFS—Depth-first Search
- MPN—Most Proving Node
- Pn-search—Proof-number Search

A. Solving program documentation

A.1 Introduction

Programs Qubic and ParallelQubic are solvers for 3-dimensional version of tic-tac-toe played in cube with size 4, known as Qubic. Both programs are written in language C and they are console applications. Qubic compares some solving algorithm, mainly the time needed for solving. It is easy to choose algorithm by macros. For ParallelQubic the fastest algorithm is chosen and parallelized.

A.2 Algorithms

A.2.1 Pn-search

Proof number search is the main solving algorithm in Qubic. All Pn-search methods are in *PnSearch.c*. The starting method is *ProofNumberSearch* with parameters:

- *qubic1* and *qubic2* are the tokens of the players
- *player* denotes which player will play (1 for first player, 0 for the second one)

It returns the result of the game:

1. 1 for the first player win
2. -1 for the second player win or a draw

Method *ProofNumberSearch* is called for every starting game in *Program.c*. It creates the root of the AND/OR graph and tries to solve it. It repeats 4 steps, until the root is proved:

1. Select the most proving node (MPN): method *selectMostProoving*
2. Develop MPN: method *developNode*
3. Set proof and disproof numbers to MPN: method *setProofAndDisproofNumbers*
4. Update all ancestors of MPN: method *updateAncestors*

Selecting the most proving node

Selecting is standard by proof and disproof numbers. It selects the first of node children with the same proof/disproof number (depends on the type of node). The selecting repeats, until the node does note have any children and this node is returned as the most proving one.

Developing the node

Developing the node means to create all its valid children. Creating children is in *generateAllChildren.* It makes next ply to game represented by node N (passed as parameter), searches if the new game is in the transposition table and according to the result, it inserts the node into the graph. There are 3 types of results of the searching transposition table:

- The game was solved before. If the value of the new game proves or disproves the node N, it is evaluated with true/false value and the child is not inserted into the graph. Otherwise the new child with value true or false is added to N.
- The game was not solved, but node M represented same game as node N was created before. Node N is added as a parent to the node M and node M is added as a child to the N (node M has more parents now). This option is possible only when DAG is created as Pn-search graph.
- The game was not found in the transposition table. New node K is created and added as a child to the node N. K is inserted into the transposition table as unsolved.

Creation of the node (allocating the structure and filling it with data) is in the *createChild.* After the children generating, each child is evaluated and then proof and disproof number is set to the child. Evaluation of the node is in method *evaluate.* There are several types of evaluation depending on the macro settings in *Settings.h*:

- Db-search
- λ-search
- Combined Db-search and λ-search: Db-search is started first. If it fails (node has value unknown), λ-search is started.
- No algorithm for evaluation: node gets value true/false if and only if the first/second player wins. It is in the method *evalGame* in *PnSearch.c.*

Set proof and disproof numbers

Setting numbers depends on type, value and state of the node:

1. The node has known value. If it is true, proof number is set to 0 and disproof number to INT_MAX (used as ∞). If it is false, proof number is set to INT_MAX and disproof to 0.
2. If the node is expanded (its children were generated), numbers are counted from its children. There are two formulas how to count the numbers: standard and weak. What is used depends on macro settings in *Settings.h* (macro WEAK_PN).
3. If the node has unknown value, heuristic numbers are set in *setHeuristicNumbers.*

Update ancestors

The method is simple. Method *setProofAndDisproofNumbers* is called to all ancestors of the node.

A.2.2 Db-search

Dependency-based search is used as the evaluation algorithm for Pn-search node. It tries to find winning path containing only threatening ply. That means the sequence of attacker plies, such that defender has only one possibility how to play and the last attacker ply is the winning ply. If this path exists, the Pn-search node is evaluated with true/false value, otherwise with unknown value. All Db-search methods are in *DbSearch.c*. The entry Db-search method is *DbSearch*. It takes one parameter *root*, where the starting game is saved, for which is searched the winning path. It returns 1 if winning path exists, 0 otherwise. First the root of the Db-search is created and it is controlled if a player wins. If a player does not win, it repeats 2 steps, until the path is found or any node is not inserted into the tree:

1. Add dependency stage: method *addDependencyStage*—returns 1 if winning path is found, 0 otherwise

2. Add combination stage: method *addCombinationStage*—returns 1 if winning path is found, 0 otherwise

Each run of the cycle is one level.

Add dependency stage

It recursively traverses the whole Db-search graph and looks for a combination node (or root) from the last level. For each founded node method *addDependencyChildren* is called. What method *addDependencyChildren* does, depends on the type of *node* (passed as parameter to the method):

1. The *node* is combination node. It makes new dependency node from combination node. If the node is winning node, it returns 1.

2. The *node* is dependency node. It looks for all possible attacker threats and makes a new dependency node with that threat and answer: attacker ply and only one defender ply.

If a new dependency node is created *addDependencyChildren* calls itself on the new node. Inserting the new dependency node into the graph is in *AddDependencyChild* in *DbVertex.c*. The method allocates new *DbVertex*, adds it to the *node* as its child, fills it with data and return pointer to it. Parameters of the method are:

1. *node*: pointer to *DbVertex*, which will be the parent of the new dependency node

2. array *positions*: operator

3. *level* and *type*: properties of the new dependency node
4. *player*: which player is attacker
5. *switchPoint*: if 1, it switch third and fourth argument of the operator
6. *win*: output parameter, if 1, the new dependency node is winning node and the method return NULL

If a winning dependency node is found the method *addDependencyChildren* returns 1, and then *addDependencyStage* returns 1 as well, otherwise they return 0.

Add combination stage

It recursively looks for pairs of nodes in Db-search graph, which can be combined. In *addCombinationStage* the first node of the pair is determined and in the *findAllCombinationNode* the second one is determined. Method *findAllCombinationNode* also traverses the graph recursively. If the nodes from the pair are not in conflict, it tries to combine the nodes. If nodes were not combined before and they do not have the same operator, it tries to find a threat, which can be made by the attacker (depending operator). If such an operator exists, it combines the node. There are 3 problems with combination node, which need to be solved before inserting the node into the graph:

- Defender four: If the defender has four tokens in line after the combination, the game is lost for the attacker and the combination node cannot be inserted into the graph.
- Defender open three: If the defender has three tokens in line and the last one is free, the defender make a threat and the attacker has to answer it.
- Defender closed three: If the defender has three tokens in line and the attacker one, the defender could make a threat and if the attacker did not answer it immediately, the threat sequence from root to the node is not valid.

If there are not any of these 3 problems, or if they are solved, the combination node is inserted into the graph. It is in *AddCombinationNode* in *DbVertex.c.* The method parameters are:

1. *node* and *partner*: combined nodes
2. array *qubic*: *qubic[0]*—second player marks, *qubic[1]*—first player marks
3. *level*: level of the new combination node
4. *freePos1* and *freePos2*: empty points (third and fourth argument of operator)
5. *defendPathLength*: length of defend path, which solve defender open threes

Defender open three

The problem with defender open threes are solved by *counterDefenderOpenThrees*. It tries to find a path, which defends the open threes. It returns:

- -1 if the path does not exist and defender win
- 0 if attacker win
- the length of the path (count of the operators), if the attacker counters the defender open threes

The path (if exists) is saved in global array *DefendPath* in *DbVertex.h*. The method *counterDefenderOpenThrees* check the path from root to combination node. If the defender makes open three, method *counterDefenderOpenThree* is called. It returns the same value as *counterDefenderOpenThrees* and it tries to defend the open three only by threats of the attacker. Its parameters are:

- array *qubic*: actual game
- array *combineQubic*: finish game—the plies from defend path must not be in conflict with this game
- *line*: line in which is the defender open three
- *freePos1* and *freePos2*: third and fourth argument of depending operator of combination node—plies from defend path must not be in conflict with these spaces either
- *defendPathLength*: length of the current defend path

Defender closed three

The problem with the defender closed three is solved by *solveDefenderClosedThrees*. It returns 0 if closed three is a problem, that means the attacker did not answer on open three immediately. It returns 1 if closed three is not problem. It creates a random path to the combination node, which is combined *node1* and *node2*. It checks the path and if the defender makes open three and the attacker does not answer it, it returns 0. If path is valid it returns 1. It can refuse a combination node where a valid path exists, but the random path was invalid. However, it does not happen very often and solving time is not affected by this randomize.

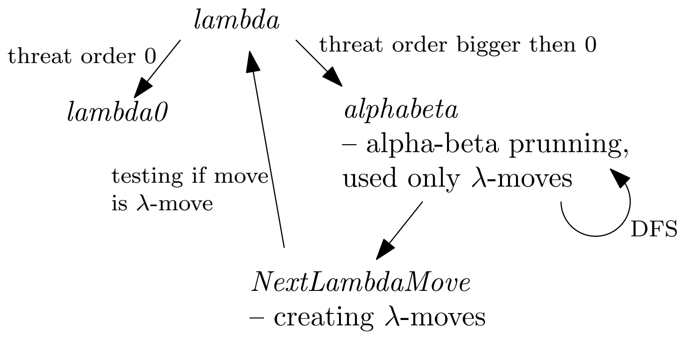
A.2.3 λ-search

λ-search is also used as an evaluation method. All methods are in *Lambda.c*. λ-search is a standard alpha-beta depth first search, but it uses only plies with threat order λ. The searching depth is limited by macro MAX_DEPTH. Entry method is *LambdaSearch*. It tries to find a winning path containing only λ-moves. If such path exists it returns 1, otherwise 0. λ is limited by macro MAX_LAMBDA. Method *LambdaSearch* calls *lambda* with increasing λ. Parameters of the method *lambda* are:

- n: limit for λ

- *d*: limit for searching depth
- *lastPly*: what ply was made last—it is for iterative evaluating of the game. If it is the first call of the method, the value of parameter must be −1.

Method *lambda* calls *lambda0* if threat order is 0 and *alphabeta* if threat order is bigger. There is standard alpha-beta pruning in *alphabeta* but only λ moves are used. λ moves are created in *NextLambdaMove*. Evaluation of the game (if a player wind) is in *eval* and *eval2*. Both methods return 4 if the attacker wins, 0 if the defender wins, or 2 if nobody wins. Method *eval* checks all lines, but *eval2* checks only lines going through the point *lastPly*, which is the parameter of the method, so method *eval2* quicker. The current game is in the array *qubic* and it is edited by methods *makePly* and *unmakePly*. Parameter *plyColor* of the method *makePly* has value 1 for the first player and −1 for the second player. In Figure A.1 is the diagram of the calling the λ-search methods.



lambda
threat order 0
threat order bigger then 0
lambda0
alphabeta
– alpha-beta prunning, used only λ-moves
testing if move is λ-move
DFS
NextLambdaMove
– creating λ-moves

Figure A.1: Diagram of λ-search methods

A.2.4 Transposition table

Transposition table is a hash table, where Pn-search nodes are saved to avoid solving some nodes more than once. All transposition table methods are in *Transposition.c*. Hash table data is in array *transposition*, the array has size about 1GB.

Searching the table

Searching the table is in *CheckHashTable1*. It makes (if the macro AUTOMORPHISM is defined in *Settings.h*) all automorphisms image of the game, which is searched. For each image the method *checkHashTable2* is called. After that it returns the result. There are several options, what is in array *transposition*:

- Free entry, *transposition[x][0]* is 0
- Solved game, in *transposition[x][0]* are the second player tokens and in *transposition[x][1]* are the first player tokens. If the first player win the game, in *transposition[x][0]* is 1 on the position, where is the first 1 from left in *transposition[x][1]*. For example: the left side of the binary code of first and second player tokens are: 0100... and 0010... and the first player win the game, so in *transposition[x][1]* is 0100... and in *transposition[x][0]*

is 0110... If the second player win the game, binary codes of the tokens are not modified when inserting into the table.

- Unsolved game, in *transposition[x][1]* is 0 and in *transposition[x][0]* is pointer to Pn-search node. This option is possible only, when macro DAG is defined in *Settings.h*.
- Invalid entry, in *transposition[x][0]* and *transposition[x][1]* is INT_MAX. This option is also possible, only if macro DAG is defined.

Method *checkHashTable2* counts hash (index to the array *transposition*) of binary codes of the players tokens (universal hash function in *getHash*) and tries to find the game in the array. It checks 100 array entry (macro HASH_TRY) after the counted index. If it finds:

1. Free entry: stops searching and returns the index of the free entry
2. Invalid entry: keeps searching, but it returns the index of invalid entry as free entry
3. Unsolved node: if it represents the searching game, stops searching returns the pointer in *node* parameter of *checkHashTable2*. This parameter is not defined, if macro DAG is not defined.
4. Solved game: if it represents the searching game, stops searching and returns the value of the game. -2 is for first player win, -3 is for the second player win.
5. Different solved or unsolved game: keeps searching.

The method *checkHashTable2* returns -4 if any free or invalid entry is not found.

Inserting into the table

There are three methods for inserting:

- *void InsertGame(uint64_t qubic1, uint64_t qubic2, int index)*: It inserts game *qubic1*, *qubic2* on position *index* into the array. It is used, when starting games for Pn-search is generated.
- *void InsertVertexPointer(struct Vertex* vertex, int index)*: It inserts pointer on position *index* into the array. It is used, when DAG is created as Pn-search graph.
- *void InsertVertexValue(struct Vertex* node)*: If DAG as Pn-search graph is created it inserts the solved game on position *node->TranspositionIndex* into the array (it rewrites the pointer to the *node*). If a tree is created, it counts the hash of the game and inserts in into the array.

Methods *InsertGame* and *InsertVertexPointer* are called right after *CheckHashTable1*, where the hash is counted, so the insert methods have the parameter *index* to avoid counting hash more than once.

Deleting from the table

When DAG is created as Pn-search graph, pointers are deleted from the table (*DeleteVertexPointer*), when the Pn-search root is solved and Pn-search tree is freed. These entries became invalid, and they do not stop the searching in *checkHashTable2*.

Reading from the table

When starting games are created, they are saved in the transposition table. After generating the starting positions, the method *Flush* is called. It returns dynamic array of all non-empty entries and then it clears the table. In the new array, even indices have entries *transposition[n][0]* (second player tokens) and odd indices entries *transposition[n][1]* (first player tokens) .

A.2.5 Automorphisms

There are 192 automorphisms (including identity) of Qubic. Counting the automorphism image of the game representation means to do permutation of bit from 64-bit integer. All automorphism methods are in *Automorphism.c* and in *AutomorphismConstant.c*. There are 3 ways how to count the automorphisms images.

Permutation array

It is the simplest and slowest method. Counting the images is in *Transform*. It has the following parameters:

- *qubic1* and *qubic2* are game representations
- *rotate1* and *rotate2* are output parameters and automorphism image of game representation is saved there *automorphism* is index of automorphism, which is used

Array *automorphisms* is used for counting. For counting the image of automorphism *a* every bit from game representation with position *i* in the number is moved to position *automorphisms[a][i]*. This method is used only for testing whether other methods return same results.

Shift array

Method *TransformShift* is quicker. It has the same parameters as *Transform*. It uses arrays *Automorphisms2* and *Shift*. For counting the image of automorphism *a* it is made & (bit operation) between game representation and numbers from *Automorphisms2*. After & it is shifted by number (with the same index as pattern in *Automorphisms2*) from *Shift*. Positive number in *Shift* means left shift, negative number right shift. Hence bits, which are moved by same positions, are moved at one time. Array *Automorphisms2* is ordered by count of patterns. Therefore images of automorphisms with less patterns are counted earlier. This method is used in searching the transposition table, when AUTOMORPHISM is defined and CONSTANT is not defined in *Settings.h*.

Constant methods

This way is the quickest. Each automorphism has the method *Transform#N* in *AutomorphismConstant.h*, where *#N* is the index of automorphism. These methods work the same as *TransformShift*, but all code is generated and they work with constants instead of values from arrays. There is also the method *TransformConst* in *AutomorphismConstant.c*, which calls methods *Transform#N* by value of parameter *automorphism*. This method is only used for testing. Methods *Transofrm#N* are called one by one in *applyAutomorphisms* during searching the transposition table, when when AUTOMORPHISM and CONSTANT are defined in *Settings.h*.

Creating automorphisms

Automorphisms are created in *CreateAutomorphisms*. First 7 basic automorphisms are created into the permutation array *automorphism*. Other automorphisms are created by joining the exists automorphisms. Items for arrays *Automorphisms2* and *Shift* are created by values from array *automorphisms*.

A.2.6 Code generating

Some parts of code are generated, because execution time shorten. If macro CODE_GENERATE is defined, these parts are generated and saved into the files and Qubic is not solved. If macro CODE_GENERATE is defined, CONSTANT must not be defined and AUTOMORPHISM must be defined. If CONSTANT is defined automorphisms are not counted, so automorphisms generating would not work. If AUTOMORPHISM is not defined, automorphism images are not counted during searching the transposition table. Therefore starting games generating would generate more games than with AUTOMORPHISM defined. Most of generating methods are in *CreateCode.c*:

- *createHashFunction*: creates *getHash* function for the transposition table and saves it into *getHash.txt*

- *createApplyAutomorphisms*: creates *applyAutomorphisms* function for constant counting automorphism images during searching the transposition table and saves it into *applyAutomorphisms.txt*

- *createAutomorphimMethods*: creates 191 (the method for identity is not generated) methods one for each automorphism with name *Transform#N* (where *#N* is number between 1 and 191) and *TransformConst* and saves it into *AutomorphismConstant.c.txt* and their declarations into *AutomorphismConstant.h.txt*

Code is generated also in *Game.c*. Here is the method *CreateWinLines*, which fills array *Lines* and *Positions* with data. When code is generated, these arrays are written into the file *lines.txt*.

Starting games generating

Starting games with depth 3 (2 crosses and 1 ring) are generated by the method *createStartGames* in *Program.c*. It fills array *startGames* with 7 non-isomorphic games with depth 3. Starting games with depth 4 are generated in *Generator.c*. For each game from *startGames* is called method *GenerateAllGames*. Generated games are stored in the transposition table with automorphisms, so only non-automorphic games are saved. The games are generated in two steps:

1. It generates all possible combination of D points in the game cube, where D is parameter *depth*. This generation is in *generateAllGames1*.
2. It generates all combination of tokens into each combination of points and saves it into the transposition table. This generation is in *generateAllPlies*.

Method *GenerateAllGames* has the following parameters:

- *depth* is the depth of generating games.
- *defenderPliesCount* is the count of defender plies in the generated plies (not in pattern).
- *pattern1* and *pattern2* are started games for generating.

In case of generating games with depth 4 from games with depth 3, the parameter *depth* is 4, the parameter *defenderPliesCount* is 1 (one ring is needed to generate, count of cross is counted in the method), and the parameters *pattern1* and *pattern2* are entries from *startGames* (*startGames[x][1]* is the for first player, *startGames[x][0]* for the second player). After generation the method *Flush* from *Transposition.c* is called and all entries are saved in array *starts*. If code is generated, the array *starts* is written into the *starts.txt*.

A.3 Data Structure

A.3.1 Game representation

The game cube is represented by two 64-bit unsigned integer. In the first are saved the first player tokens, the second player tokens are saved in the second one. If some point in the cube has coordinates $[x, y, z]$ it is on $4^2x + 4y + z$ position (from right) in integer. The Qubic game with coordinates can be written into console by method *WriteQubic*, which is saved in *Game.c*. The first player tokens are usually in variable *qubic1*, the second player tokens in *qubic2*. If they are saved in array, then *qubic[0]* are second player tokens, *qubic[1]* are first player tokens. The first player is represented by 1, the second player by 0. Each winning line is represented by 64-bit unsigned integer saved in array *Lines* in *Game.c*. Coordinates system is the same as in the game representation. Hence the result of the line ℓ representation & the tokens representation of some player is representation of the player tokens in the line ℓ. In array *Positions* are saved lines going through each point. For example: lines going through point 23 in cube are saved in *Positions[23][0]* and *Positions[23][1]*. The integers are indexed from right to left, so only 12 bits from *Positions[x][0]* are used (there are 76 lines and first 64 lines are saved in *Positions[x][1]*). For example: if right end of *Positions[23][0]* is ...0100 (in binary code), it means that the line saved in *Lines[66]* goes through position 23 in game representation.

A.3.2 Vertex

Vertex structure represents Pn-search node. It is saved in *PnVertex.h*. *Vertex* properties are:

- *ProofNumber* and *DisproofNumber*: 32-bit integers, proof and disproof number of the node.
- *Memory*: 8-bit unsigned integer. It is used for some small (1-bit and 2-bit) node properties. Meanings of bits (indexed from right):
  - 0: Type of node—0 is AND node, 1 is OR node
  - 1: Evaluated—1 is true, 0 is false
  - 2: Expanded—1 is true, 0 is false
  - $3-4$: Value of node—00 is Unknown, 01 is False, 10 is True
  - $5-6$: How the node was evaluated: 01 by evaluation algorithm, 10 by Pn-search (solving by values of children of the node), 11 by transposition table (same node was solved earlier)
  - 7: Not used

  There are methods in *PnVertex.c* to get and set these properties.
- *TranspositionIndex*: 32-bit integer. Index, where the node is in the transposition table. It is used only when DAG is used as Pn-search graph.
- *Ply*: 8-bit unsigned integer. Point, where player plies. It is used when Db-search nor λ-search is not used.
- *Qubic1*: 64-bit unsigned integer. First player marks.
- *Qubic2*: 64-bit unsigned integer. Second player marks.
- *ChildrenCount*: 8-bit integer. Count of the node children (count of the Children array item).
- *Children*: dynamic array of Vertex structure. Pointers to the node children.
- *Parent*: parent of the node. Type depends on settings:
  - Pointer to *Vertex* structure, if tree is used as Pn-search graph. In this case every node has only one parent.
  - Pointer to *Parent* structure, if DAG is used as Pn-search graph. In this case a node can have more than one parent. So all parents are saved as linked list. The structure *Parent* is very simple. It has two fields: pointer to *Vertex* structure and pointer to another *Parent* structure (next item in list). It is also saved in *PnVertex.h*.

A.3.3 DbVertex

DbVertex structure represents Db-search node. It is saved in *DbSearch.h*. *DbVertex* properties are:

- *Depth*: 8-bit unsigned integer. Depth of the node in Db-search graph (amount of nodes between this node and root).
- *Level*: 8-bit integer. Absolute value of the field is level of the node. Sign of the field is type. If it is 0 the node is root, if it is positive it is combination node, if it is negative it is dependency node.
- *Qubic*: 2-length array of 64-bit unsigned integer. *Qubic[0]* represents tokens of the second player, *Qubic[1]* represents tokens of the first player.
- *Child*: pointer to DbVertex structure, the first child of Db-search node.
- *LastChild*: pointer to DbVertex structure, the Last child of Db-search node.
- *Sibling*: pointer to DbVertex structure, sibling of the node (nodes have same depth).
- *Parent*: pointer to DbVertex structure, the first parent of the node.
- *Parent2*: pointer to DbVertex structure, the second parent of Db-search node. It is used when node is combination node.
- *DbOperator*: 4-length array of 8-bit integer. It represents Db-operator.
- *DefendPath*: dynamic array of 8-bit integer. Path to defend open threes in combined nodes. First item is length of the array. Every four items in array are one operator.

Db operator is represented by 4-tuple of integers. The first two numbers represent points, where the attacker must have his tokens. The third number represents point, where the attacker puts his token and creates threat. The fourth number represents point, where the defender puts his token to counter the threat.

A.4 Settings

There are several macros in *Settings.h* for control, what parts of code are compiled. Macro settings are only in program Qubic.

- DEBUG: If defined, algorithm soundness is tested. It does not change the algorithms, but the solving is slower. Error messages are written into console.
- DB and LAMBDA: If DB is defined, Db-search is used as evaluation method (same with LAMBDA and λ-search). If both are defined, Db-search is used first and if it fails, λ-search is used. If DB nor LAMBDA are not defined, nodes get their value only from their children.

- CODE_GENERATE: If defined, code is generated and game is not solved. If CODE_GENERATE is defined, CONSTANT must not be defined and AUTOMORPHISM must be defined, otherwise the generated code is not correct.
- CONSTANT: If defined, pre-generated code is used (i.e. winning line representations are integer constants, there are generated method for each automorphism and so on).
- AUTOMORPHISM: If defined, automorphisms are used in searching the transposition table.
- DAG: If defined, DAG is used as graph in Pn-search. If not defined regular tree is used.
- WEAK_PN: If defined, Weak Pn-search is used, otherwise standard Pn-search is used.
- STARTS_DEPTH_3: If defined, starting games have depth 3 (2 crosses and 1 ring), otherwise the depth is 4.

A.5 Inputs and Outputs

Both programs (Qubic and Parallel Qubic) do not need any input. After starting the program Qubic, the macro settings are written into the console. If macro CODE_GENERATE is defined, code is generating into the files. After that the application is closed. If CODE_GENERATE is not defined, Qubic solving starts. When a starting game is solved, the result is written into the console. The format is *Result #N: #R*, where *#N* is index of the starting game and *#R* is result, 1 for the first player win, -1 for the second player win or draw. After solving all starting games, macro setting are displayed again, and some statistics about solving are displayed:

1. *Result*: overall result of Qubic: 1 if every starting game had result 1 (the first player win), 0 if some starting game had result -1
2. *Node count*: count of created Pn-search nodes
3. *Hashed*: count of Pn-search nodes saved in the transposition table
4. *Not Hashed*: count of Pn-search nodes, which were not saved in the transposition table, because no free entry was found
5. *Hit count*: count of successful searching in the transposition table
6. *Evaluated*: count of Pn-search nodes evaluated with known value (True/False)
7. *Evaluation Failed*: count of Pn-search nodes, which evaluation algorithm did not evaluate with known value
8. *Evaluation Success*: count of Pn-search nodes, which evaluation algorithm evaluated with known value

9. *Expanded*: count of expanded Pn-search nodes (children were created for those nodes)
10. *Time*: solving time in format mm:ss

A.5.1 Outputs of parallel program

The program ParallelQubic has not any macro settings described in Section A.4. There is a macro NUM_THREADS, which defines the number of used threads. When the program is started, the value of the macro NUM_THREADS is written into the console and then it starts solving the games from array *starts*. When a thread solves the starting game, the result is written into the console (in the same format as in single-thread program). After solving all starting games, number of threads are displayed again, and some statistics are displayed:

1. *Result*: overall result of Qubic: 1 if every starting game had result 1 (the first player win), 0 if some starting game had result −1
2. *Nodes counted more than once*: count of nodes, which were simultaneously solved in different threads
3. *Time*: solving time in format mm:ss

A.6 Parallelization

In ParallelQubic, the fastest algorithm from Qubic is parallelized. It is Weak Pn-search on DAG with Db-search as the evaluation method, without automorphisms in searching the transposition table and starting depth 4 (i.e. DB, CONSTANT, DAG and WEAK_PN would be defined in Qubic other macros would not). The changes are not so large. *Program.c* is changed, here the threads are created. Starting games are read from array with lock to avoid solving some games more than once. Most of the global variables from Qubic are global only within the thread.

A.6.1 Transposition table

Most of the changes are in the transposition table. The table is divided into two tables. The first is global for the whole application and solved games are saved there. Data is written into the table with a lock. Reading is without lock. The second table is local for each thread and unsolved Pn-search nodes are saved there.