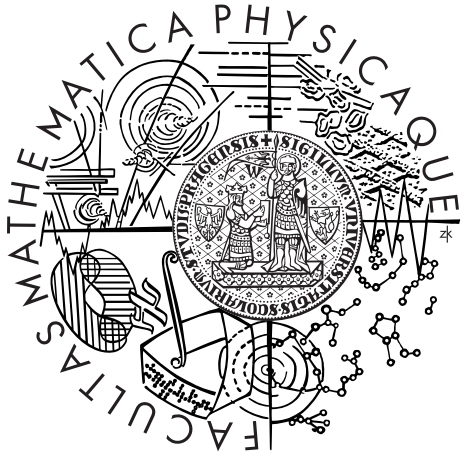
Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS



Jitka Novotná

Algorithms for solving strong positional games with high symmetry

Computer Science Institute of Charles University

Supervisor of the bachelor thesis: Tomáš Valla

Study programme: Informatika

Specialization: Obecná informatika

Prague 2012

I want to thank a lot of people who helped me with this thesis. First I want to thank Jan Kulveit, Tomáš Gavenčiak and Martin Mareš - I'm grateful for their time, support, good advice and also proofreading. I thank Pavel Veselý, Pavel Dvořák and Roman Smrž for helpful discussions and aid with debugging my code. Finally I thank my advisor, TV, for an unrelenting supply of fresh ideas what can be extended and improved.

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date ..... signature of the author

Název práce: Algorithms for solving strong positional games with high symmetry

Autor: Jitka Novotná

Ústav: Informatický ústav Univerzity Karlovy

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

Abstrakt:
V této práce představujeme několik algoritmů pro počítačové řešení silných pozičních her a to především algoritmů založených na algoritmu PN-search.
Zaměřujeme se na hry s vysokou symetrií herního plánu. Spojováním mnoha izomorfních pozic jsme dosáhli velkého zmenšení části herního strumu, kterou je potřeba prohledat.
Představujeme již známa vylepšení a také navrhujeme vlastní. Ověřujeme jak tyto heuristiky funguji na příkladu klikové hry.
K práci je přiložen software pro řešení silné klikové hry pro $K = 4$ a $N = 5 \dots 8$. Software zvládl vyřešit (6,4) a (7,4)-klikovou hru a dokázal, že druhý hráč má neprohrávající strategii, což se očekávalo, ale dosud nebylo potvrzeno.

Klíčová slova: PN-search, silné poziční hry, vysoká simetrie, kliková hra

Title: Algorithms for solving strong positional games with high symmetry

Author: Jitka Novotná

Department: Computer Science Institute of Charles University

Supervisor: RNDr. Tomáš Valla, Informatický ústav Univerzity Karlovy

Abstract:
In this thesis, we analyse several algorithms for solving strong positional games, mostly based on PN-search.
We focus on games with high symmetry of the game plan, where it is possible to substantially reduce the partial game tree by joining isomorph positions.
We review several known enhancements of PN-search and also propose some of our own design. We measure the effect of the enhancements on the clique game.
A part of the thesis is a software solver for clique game for $K = 4$ and $N = 5 \dots 8$. We were able to solve (6,4) and (7,4) clique games and prove that the second player has a draw strategy, which was expected but not shown previously.

Keywords: PN-search, strong positional games, high symmetry, clique game

Contents

Introduction

This thesis describes an algorithm for solving strong positional games with high symmetry. We describe the most important algorithm for solving combinatorial games – PN-search and its various enhancements, which change it so much, that they can be considered new algorithms.

PN-search was introduced by Allis in 1994[1]. It has become well a know algorithm and a basis for a lot of further work. It was successfully applied for a wide class of games, and many enhancements were invented. We review some of them and how they can be useful in case of a game with high symmetry. We consider which enhancements can be used together and what problems can arise with these combinations. We also introduce some enhancements which we discovered independently.

Outline

In the first chapter, we informally introduce some combinatorial game terminology. We suppose the reader has general knowledge about graph theory, see Diestel [2]. In the first chapter we also define clique game and give solutions of some cases of a clique game.

Known algorithms and their enhancements are introduced in the second chapter. Usually, algorithms are designed for a general combinatorial game. We apply them to strong positional games, which simplifies the descriptions. We start with a game solving meta algorithm. Some specific algorithms are described after. We choose PN-search to implement and to be described in detail.

Theoretical results, which we found independently, are described in the third chapter. In subsections 3.1.1 to 3.1.5 are enhancements of PN-search. Then we introduce some rules for deleting from the cache which are designed for strong positional games. We consider a problem which can occur when we combine some enhancements, and we propose a solution for the problem. In the end of the chapter, we focus on a graph of a clique game. We describe several ways of storing the game state and also how to normalize them.

The last chapter introduces our software and presents experimental results. First, we describe the partial game tree created by our solver during solving (6,4) and (7,4)-clique games. Then we compare some mentioned enhancements. In the end of the chapter, we describe the software implementation in more detail.

1. Games

In this chapter, we introduce combinatorial games and define several types of them. We will describe one such game in more details.

1.1 Definitions

We explain the basic combinatorial game definition informally on the example of chess. There are two *players*, which alternate in their moves. There is a board with 64 squares, some pieces placed on some squares and some player is on turn. We call this the *game state*. The player *turns* by moving some pieces to some other game state. He decides where to move – all decisions from each game state where the player is on turn are called a *strategy*. When one of the players moves to a *winning state* called mate, he wins. Both players want to win. There are also *draw states*, this is for example when there is no legal move.

1.1.1 Positional games

In *positional games*, the game state is set of states of *positions* which are either *free* or *claimed* by a player. Tic-tac-toe is a typical positional game. There are nine square positions. In each turn, one of the two players claims one free position, for example, by drawing a cross. The rules of the game define a set of positions called the *winning set*. In tic-tac-toe, every straight line with three squares is a winning set. Winning set is *claimed* by a player when the player claimed all positions from the set. Winning set is *free* for a player when no position in it is claimed by his opponent. The situation, when all except one position of a winning set is claimed by a player and the last position is free, is a *threat*.

There are two types of positional games. In the last paragraph, we described a *strong game*. In a strong game, both players want to claim a winning set and who claims first wins. In the other variant, called *maker-breaker game*, only the first player wants to claim a winning set. The second player wins in the case that every position is claimed and the first player has not claimed a winning set. There is no draw in the maker-breaker variant.

There are three possible endings of a strong positional game: the first player wins, the second player wins, there is a draw. As we will show in Theorem 1, if both players play their best possible strategies, the second player cannot win. The main question of this thesis is to find an algorithm which *solves* the game, i.e., determine if there is a winning strategy for the first player or a draw strategy for the second player in the finite strong positional game.

Theorem 1. *In a strong positional game the second player cannot has a winning strategy.*

Proof. We prove it by contradiction. Suppose that the second player has a winning strategy s. We show a winning strategy s' for the first player.

The first player claims a random position p in his first turn. In later turns, the first player imagines that the position p is free, and he plays according to the

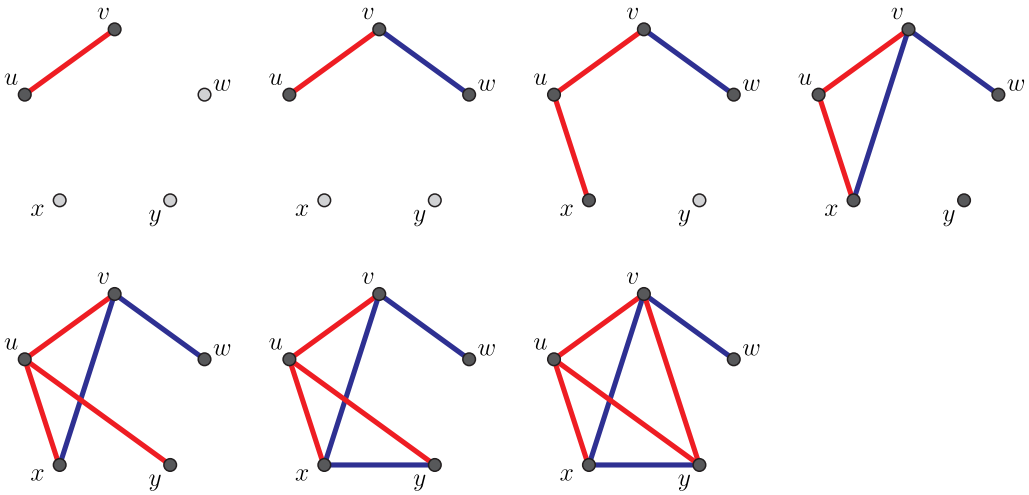


Figure 1.1: Two players play (5,3)-clique game.

strategy s. When the strategy s says to claim p, he claims another random free position p' and from this moment, he will use p' instead of p.

The first player also has a winning strategy, but both players cannot win. Contradiction.

□

There are several algorithms for solving general combinatorial games. We apply them to a strong positional game. Description of these algorithms is simpler. The algorithms are usually designed for games with *high branching factor* (that is the average number of possible moves in each game state is high – a typical example of such a game is Go) and *low symmetry* (that is only few or no distinct game states are symmetrical).

1.2 Clique game

In this section we define the clique game which we are going to use as the main example in this thesis.

1.2.1 Rules

The clique game is a strong positional game.

Two players play on a clique graph with N vertices. In each turn, one player colors one edge by his color. The first player has red color and the second player blue color. The player who colored a clique subgraph on K vertices wins.

We call it (N,K)-clique game. See an example of (5,3)-clique game game for in Figure 1.1.

Solution of (N,4)-clique game for $N < 18$ are open problem mentioned by Beck [3].

1.2.2 Solution for some N, K

For some N and K, the game can be solved by purely theoretical argument without computer backtracking.

We will use Ramsey's theorem. There are many variants of Ramsey theorem, the weakest one is enough for us. For proof see, e.g. Diestel [2].

Theorem 2. *For any positive integer k, there exists a positive integer n such that for any clique on at least n vertices whose edges are colored red or blue, there exists either a clique on k vertices with has all edges red or has all edges blue.*

Theorem 3. *For fixed K, there exists N_0 such that in each (N,K)-clique game where $N > N_0$ is a winning strategy for the first player.*

Proof. We chose N_0 as a n in Ramsey theorem. We already proved that either the first player has a winning strategy or the second player has draw strategy. It suffices to prove that there is no draw in (N,K)-clique game. Draw game state is a game state where every edge is claimed – it has a red or blue color and there is no clique on K vertices with all edges of the same color. However, this is not possible since according to the Ramsey's theorem, there is always such a clique. So there is no draw in (N,K)-clique game.

□

The case $K = 2$

For every $N \geq 2$, the first player has a winning strategy. He just claims one edge and wins.

The case $K = 3$

The number n from Ramsey's theorem 6 for $k = 3$. So, for $N \geq 6$, the first player has a winning strategy, as we proved in Theorem 3.

When $N = 5$, the first player also has a winning strategy. He plays similarly as in Figure 1.1. First he marks edge (u, v). When we count symmetry the second player has two possible moves to (v, w) and to (w, y). The first player turns to (u, x) and creates a threat. The second player must play (x, v). The first player turns to (u, y) which creates two threats (y, v) and (y, x) and he is going to win.

When $N \leq 4$ the second player has a draw strategy which can be found by simple backtracking over all position manually.

The case $K = 4$

When $N \leq 5$ the second player has a draw strategy. There are not enough edges to claim the whole K_4.

When $N \geq 18$, the first player has a winning strategy, as we proved in Theorem 3.

An interesting question is what is the lowest N for which the first player has a winning strategy. Our solver proved that for $N = 6$ and $N = 7$ the second player has a draw strategy.

1.2.3 Time and space complexity

We do not determine neither the asymptotic time complexity nor the asymptotic space complexity of the algorithms in this thesis. It is not clear what time/space

complexity means in game solving – if the input of the algorithm are rules of the game, most of the games can be solved in constant time.

On the other hand when the input of the algorithm is a game state of a positional game with n positions, solving the game is at least hard problem. ($n = \binom{N}{2}$ in (N,K)-clique game). The number of game states in such a game is $\mathcal{O}(2^{\binom{N}{2}})$. It is not necessary to visit them all, however, the number of the game states we must visit even for proving that a strategy is winning is also $\mathcal{O}(2^{\binom{N}{2}})$. From this view, all the improvements in this thesis are just heuristic.

The size of the partial game tree, which we must create for solving the game, is much more important than asymptotic complexity.

2. Solver algorithms and their enhancements

In this chapter, we introduce some heuristic algorithms for solving the combinatorial game and their enhancements.

2.1 Overview of algorithms

We will describe a meta algorithm how to prove that a finite two-player positional game is determined, that is if there is a winning strategy for the first player or draw strategy for the second.

2.1.1 Meta algorithm

All game states can be represented as a rooted tree. Each game state is a node and if someone can turn from game state u to game state v, there is an edge between them, and we will denote it (u, v). The *level* of a node is the distance from the root.

We will not try to build a *complete game tree*, representing every possible game state, because it would be too large to be represented in memory. Instead, we will incrementally build a *partial game tree* starting from the empty tree, trying to keep the partial game tree as small as possible.

We assign to each node one of three possible values – true, false, unknown. A node will have value true when we prove that there is a winning strategy for the first player and value false when we prove that there is a draw strategy for the second player. Otherwise, the node has value unknown. We say that node is *solved* if it has value true or false.

The algorithm starts with one node with value unknown. In each iteration, the algorithm does one of the two things:

- **Develop node.** We choose one leaf node with value unknown and *develop* it, which means that we create its sons. When a son is created we evaluate it (we check whether one of the players won).
- **Update node.** We choose one developed node with value unknown and examine its sons. We check if we can assign value true or false to the node.

Algorithm ends when the root is solved, and so game is determined.

There are two types of nodes. When the first player is on turn it is an or node. To assign value true for it, it is enough if one of its sons have value true. Then, the first player has a winning strategy which starts by the turn to the node with value true. On the other hand, when the first player has no winning strategy after any turn he has no winning strategy at all. So we can assign value false when all sons have value false. If none of above holds, the value of the node remains unknown. Hence value of an or node can be found by an operation which is very similar to logical *or*.

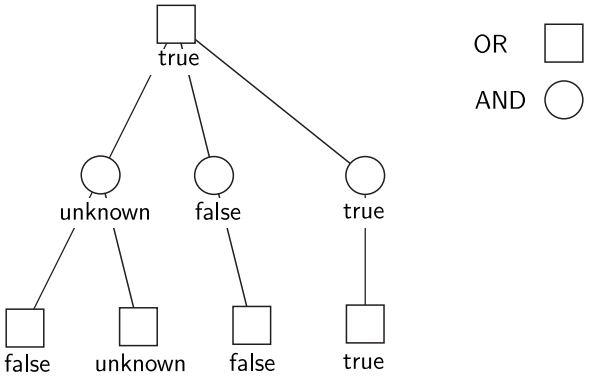


Figure 2.1: A partial game tree with assigned values.

The second type of node is an and node. The second player is on turn. Assigning values works in a similar way, we just use *and*-like operation instead of *or*-like.

See how it works in Figure 2.1.

2.1.2 Depth-first search

We can use ordinary depth-first search for searching the complete game tree and solving the root. We start by creating the root. We develop the root and choose one of it sons. We solve the son recursively and update the root. We continue solving the sons and updating until the root is solved.

This algorithm is a suitable solution for the clique game. The depth of the tree is $\binom{N}{2}$ so it does not take much memory. This is obviously slow but if we use some of enhancements of PN-SEARCH which are described later, this algorithm can find a solution for clique game with $N = 5, K = 4$ and maybe more.

2.1.3 Alpha-beta

Alpha-beta is a well known algorithm for finding good strategies. It is described for example in [4]. We can use it for any node n. It searches a part of the subtree under n and for nodes in some depth uses a rating function and returns the rating of n. The rating which ALPHA-BETA returns can be used for deciding to which game state to turn. It is useful as a heuristic and can be used for example in Artifical Intelligence and also in the clique game. We can use the depth-first search, and when we make a decision which sons to solve, we first rate all its sons by ALPHA-BETA and then solve them in the order determined by the ratings.

2.1.4 PN-search

PN-SEARCH was introduced by Allis [1]. We will describe an immediate evaluation variant of it which is better suited for our problem, because a test if a game state is winning is quick.

PN-SEARCH is a best-first search[1]. We need to specify which node is the best for our problem. We start with some definitions, which were used in [1].

[1]We traverse the tree in the order prescribed by a priority function indicating which node helps us most to find the solution.

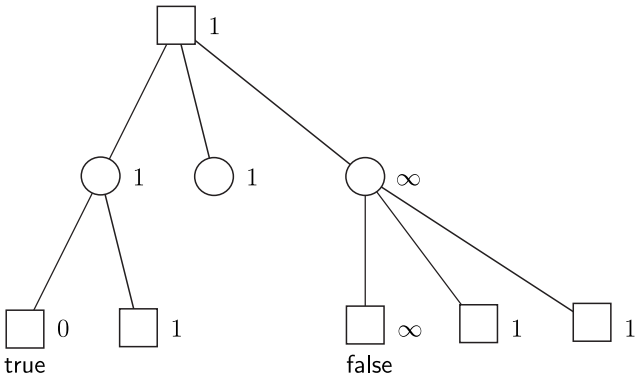


Figure 2.2: A partial game tree with proof numbers.

Definitions.

- We say that we *prove* a node if we prove that the first player has a winning strategy from it, and we can assign true to it.
- We say that we *disprove* node if we prove that the second player has a draw strategy from it, and we can assign false to it.
- For any partial game tree T, a set of leaf nodes with value unknown S is a *proof set* if proving all nodes within S proves T.
- For any partial game tree T, a set of leaf nodes with value unknown S is a *disproof set* if disproving all nodes within S disproves T.
- For any partial game tree T, the *proof number* of T is defined as the cardinality of the smallest proof set of T.
- For any partial game tree T, the *disproof number* of T is defined as the cardinality of the smallest disproof set of T.

We will show how proof and disproof numbers work and how they can be calculated on some examples. At Figure 2.2, we show a game tree with proof numbers. Nodes with value true have proof number 0 – there is nothing left to prove. Nodes with value false have proof number ∞ – there exists no set of nodes which proves it. Leaf nodes with value unknown have proof number 1 because it is enough to prove the node itself. An internal and node has proof number, which is equal to sum of the proof numbers of its sons, because for proving the and node we must prove all its sons, by proving their proof sets. An internal or node has proof number equal to the minimum of proof numbers of it sons because we need to prove at least one of its son.

Determining the disproof number works in a similar way. See Figure 2.3.

Definition. *For any partial game tree T a most proving node of T is a node, which by obtaining the value* true *reduces T's proof number by 1, while by obtaining the value* false *reduces T's disproof number by 1.*

Most proving node is a node which helps solve the root in both cases (proving or disproving it) so we will use it as the best node in best-first search. We will show that such a node exists in every partial tree.

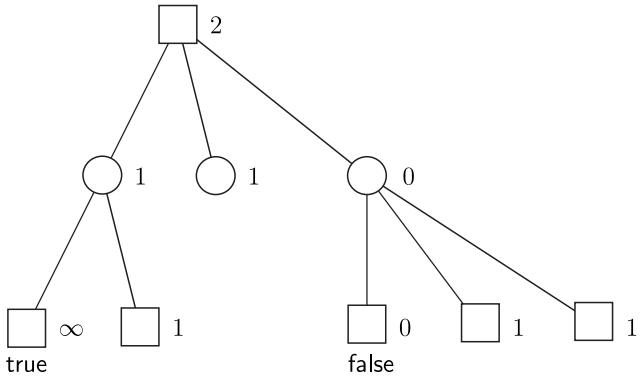


Figure 2.3: A partial game tree with disproof numbers.

Theorem 1. *(Allis [1]) In every partial game tree there is a most proving node.*

Proof. We will prove it by induction on depth of partial game tree. It is enough to show that there is a minimal proof set and a minimal disproof set having a common node. Proving the node reduces the proof set and disproving it reduces the disproof set.

- *Basic:* If the root is a leaf node, both the proving and the disproving set contain just the root.

- *Induction step:* Let the root be an and node. Minimal proof set of the root is the minimal proof set of its son s with minimal proof number. Minimal disproof set of the root is the union of minimal disproof sets of its sons, so it contains the minimal disproof set of s.

  Imagine a game (sub)tree where s is the root. By induction, his minimal proof set and minimal disproof set have a common node, which is also in the intersection of minimal proof and minimal disproof sets of the original root and is the most proving node.

  Proof for or root proceeds analogously.

□

Now we informally describe the whole PN-SEARCH, as detailed in pseudocode in Table 2.1 which was taken from [1].

We store a partial game tree and proof and disproof numbers of each node. We start with one node – the root. Then we repeat the following three steps until the root is solved.

The first step is *selectMostProving(root)* (see Table 2.2). This function works similarly to the proof that a most proving node exists, see Figure 2.4 We start in the root and continue by choosing the son which has the minimal proof number when we are in and node, respectively the minimal disprove number in or node, until we are in a leaf node. When there are more sons with a minimal number we choose the leftmost one (we fix an arbitrary ordering).

The second step is *developNode(mostProvingNode)* (see Table 2.4). There, we create sons of the most proving node and evaluate them.

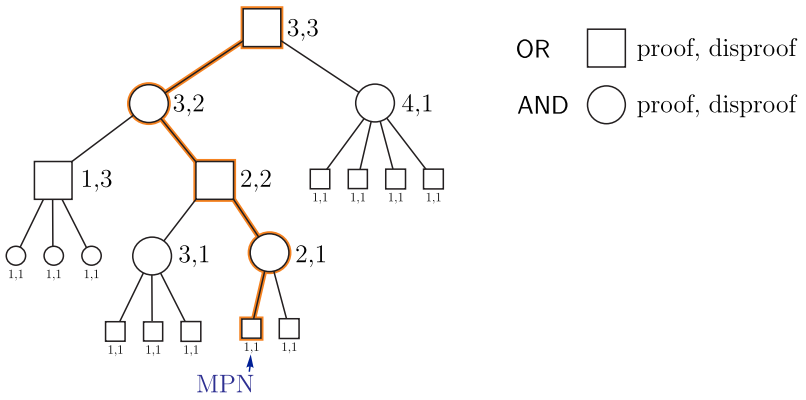


Figure 2.4: Selecting the most proving node.

```
procedure proofNumberSearch(root)
    evaluate(root)
    setProofAndDisproofNumbers(root)
    while root.proof ≠ 0 and root.disproof ≠ 0 and resourcesAviable() do
        mostProvingNode := selectMostProving(root)
        devolopNode(mostProvingNode)
        updateAncestors(mostProvingNode)
    od
    if root.proof = 0 then root.value := true
    elseif root.disproof = 0 then root.value := false
    else root.value := unknown

end
```

Table 2.1: PN-search algorithm.

The third step is *updateAncestors(mostProvingNode)* (see Table 2.5). Proof and disproof numbers in partial game tree have changed so we must update their stored values. We start updating most proving node and continue by updating its ancestors up to the root. At the end of these three steps, the (dis)proof numbers are again correct, because only numbers above most proving node could have changed.

2.1.5 Choice of solver algorithm

We chose PN-SEARCH for solving clique game, as it is widely known to outperform ALPHA-BETA algorithms in many domains, as shown by Allis [1] and [5] on several games.

Regardless of the algorithm used, it is highly desirable to work on a game DAG (as defined in section 2.2.2) instead of a plain tree. However, using DAG deforms some of the criteria used by both algorithms. In case of PN-SEARCH, we know methods to overcome those problems.

Additionally, ALPHA-BETA can be used as a heuristic in some of the PN-SEARCH variants.

```
function selectMostProving(node)
    while node.expanded do
        case node.type of
        or:
            i:=1
            while node.children[i].proof ≠ node.proof do
                i := i+1
            od
        and:
            i:=1
            while node.children[i].disproof ≠ node.disproof do
                i:= i+1
            od
        esac
        node := node.children[i]
    od
    return node
```

Table 2.2: Most-proving node selection algorithm.

```
procedure setProofAndDisproofNumbers(node)
    if node.expanded then
        case node.type of
        and:
            node.proof := ∑_{N∈children(node)} N.proof
            node.disproof := min_{N∈children(node)} N.disproof
        or:
            node.proof := min_{N∈children(node)} N.proof
            node.disproof := ∑_{N∈children(node)} N.disproof
        esac
    elseif node.evaluated then
        case node.value of
            false: node.proof := ∞; node.disproof := 0
            true: node.proof := 0; node.disproof := ∞
            unknown: node.proof := 1; node.disproof := 1
        esac
    else node.proof := 1; node.disproof:=1
    fi

end
```

Table 2.3: Proof and disproof numbers calculation algorithm.

```
procedure develop(node)
    generateAllChildren(node)
    for i:=1 to node.numberOfChildren do
        evaluate(node.children[i])
        setProofAndDisproofNumbers(node.children[i])
    od

end
```

Table 2.4: Node-development algorithm.

```
procedure updateAncestors(node)
    while node ≠ nil do
        setProofAndDisproofNumbers(node)
        node := node.parent
    od

end
```

Table 2.5: Ancestor-updating algorithm.

2.2 Enhancements of PN-search

We show several enhancements to the basic PN-SEARCH algorithm. Each of them is basically a new algorithm, and study them separately, unless noted otherwise. For a reference for most of the enhancements in this chapter, see [1].

Some of the enhancements use proof and disproof numbers and other concepts in a slightly different sense and treat them more like variables.

2.2.1 Path from the root

Proof number search is best-first search. Its main disadvantage is that searching for a most proving node takes a lot of time.

Theorem 4. *A current path is the path from the root to the most proving node. This path was traversed in the last search for the most proving node.*

In PN-SEARCH, each iteration starts at the root and descends down. After developing the most proving node, we update the proof and disproof numbers, starting in the most proving node and returning all the way back to the root. So we traverse the current path twice.

There are two enhancements which reduce the number of nodes traversed to select the most proving node.

Last changed node

We make two basic observations.

- When there is no value change during node update, traversal up can be stopped even before the root.

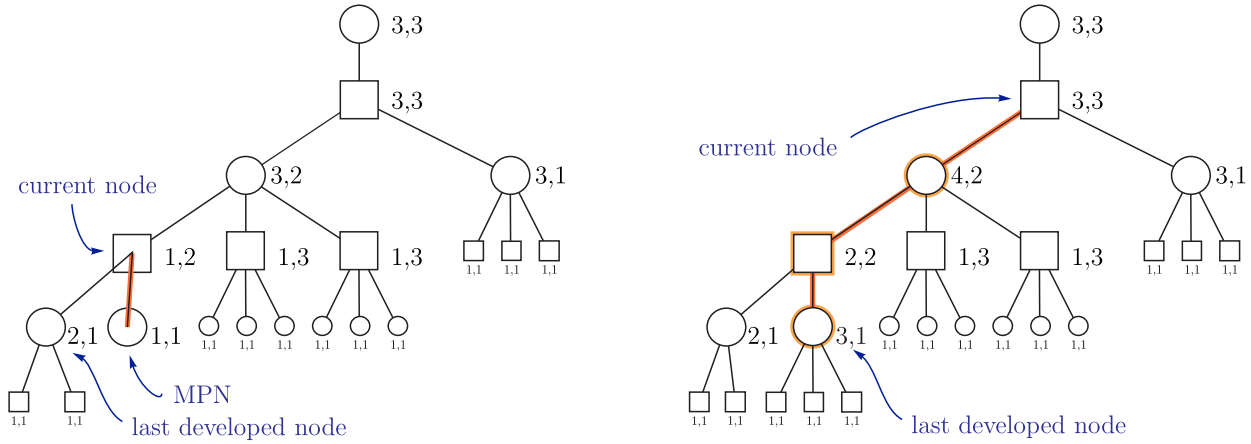


Figure 2.5: Enhancement LAST CHANGED NODE.

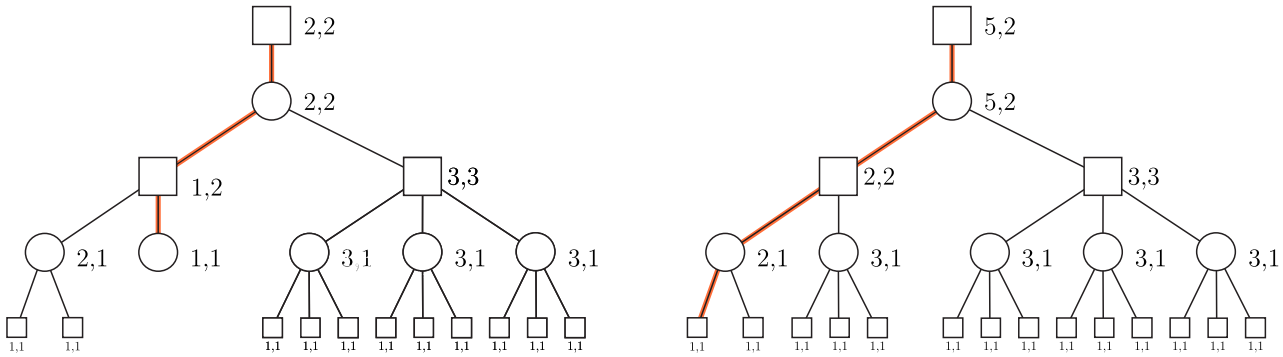


Figure 2.6: Two consecutive paths with the same beginning.

- Current paths of two consecutive iterations have the same beginning. They can differ only in part where proof or disproof number was updated.

Definition. *A current node is the lowest unchanged node from the current path, or the root if everything were changed.*

Enhanced PN-SEARCH works like ordinary PN-SEARCH. It has one variable called *currentNode*, at the beginning the value of *currentNode* is the root. Each *selectMostProving()* starts in *currentNode*. And each *updateAncestors()* ends at the node where proof and disproof numbers have not been changed and sets *currentNode* to the first unchanged node. See an example in Figure 2.5.

Depth-first PN-search

Here we describe depth-first PN-SEARCH, first introduced by Nagai et al. [7], which saves even more traversing. In two consecutive iterations, the current paths have a common beginning. See Figure 2.6. We reduced the part when proof and disproof numbers stay same, but we can also postpone some updates as shown in 2.7. We need to know the highest node which will not be in the new current path.

We introduce *proof and disproof thresholds* – two new numbers for each node on the current path. They are set so that when (dis)proof number is larger or equal, we must also update its parent. When it is smaller than its threshold, the next most proving node is in the same subtree.

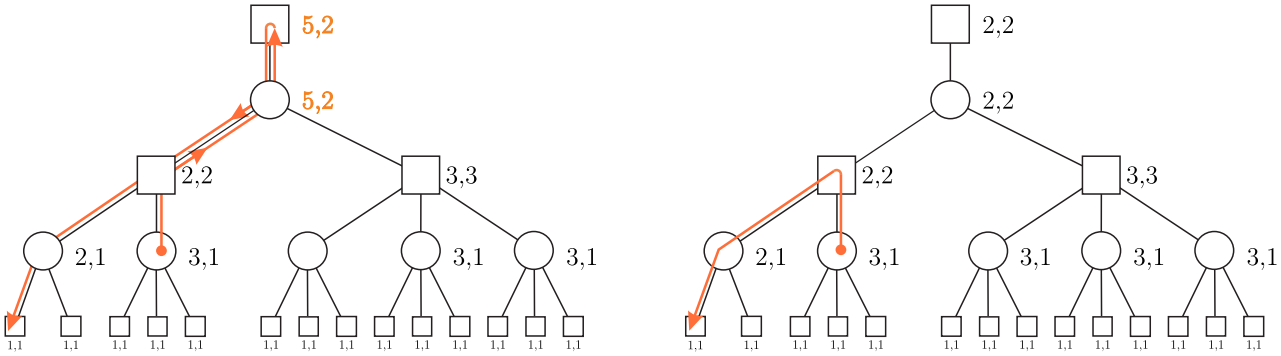


Figure 2.7: Postponed updates in depth-first PN-search

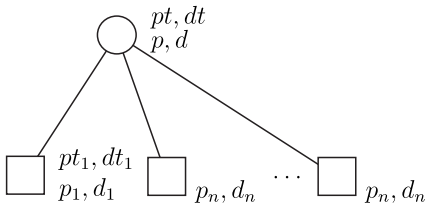


Figure 2.8: Setting proof and disproof thresholds.

Let p and d be proof and disproof numbers of node n. Let pt and dt be their thresholds. When n is and node we assume that its sons $1 \ldots k$ are ordered so that $p_1 < p_2 < \ldots < p_k$. You can see an example in Figure 2.8. Rules for setting new thresholds were deduced in [6]:

$$\begin{aligned} pt_1 &= min(pt, p_2 + 1), \\ dt_1 &= dt - d + d_1. \end{aligned}$$

When n is or node we assume that $d_1 < d_2 < \ldots < d_k$. Rules are symmetrical:

$$\begin{aligned} pt_1 &= pt - p + p_1, \\ dt_1 &= min(dt, d_2 + 1). \end{aligned}$$

When we delay updates to the time when $p > pt$ or $d > dt$ we can make more changes to the algorithm.

Main point is that we do not need to store the whole partial game tree we have traversed. It is enough to store nodes from the current path. Such algorithm could be implemented recursively and for this reason it is called depth-first. It is not a typical depth-first search because one node can be visited more than once.

The so called DF-PN search is typically implemented with a cache used for storing as many nodes from the partial game tree as possible.

2.2.2 Transpositions to DAG

We are visiting some game states many times in the game tree. A basic enhancement is to join these nodes into one as shown in Figure 2.9 of the tic-tac-toe game tree.

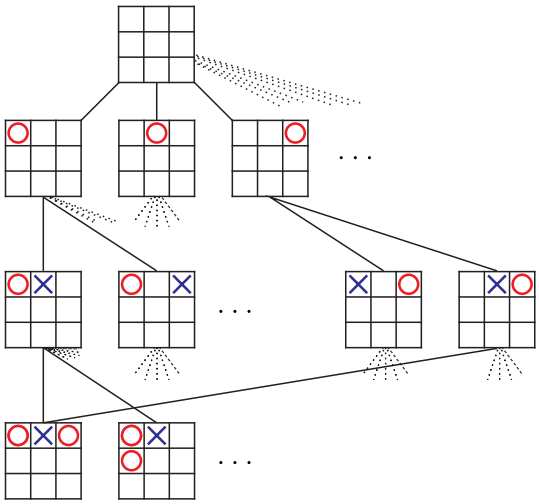


Figure 2.9: Identical game states are joined in the tic-tac-toe game.

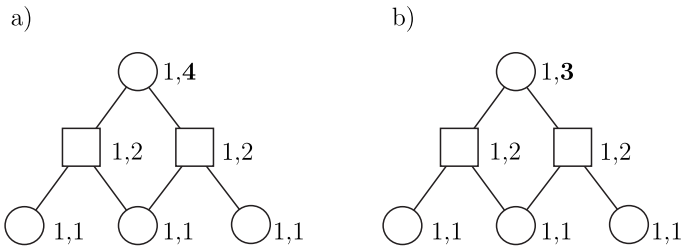


Figure 2.10: Proof and disproof numbers a) computed as usually b) by definition

The problem is that we now have a game DAG [2] instead of a game tree. There, proof and disproof numbers cannot be computed like above, because one node can increase the proof number of another node more than once, see Figure 2.10 for example. However, proof and disproof numbers computed by PN-SEARCH can be useful even though they can be higher than proof and disproof numbers as they was defined.

We can modify PN-SEARCH to work with a DAG. First, when we generate sons we check whether they already exist. We can use a hash table for this. Second, we need to update *all* the parents. This could cause problems if used together with other enhancements such as LAST CHANGED NODE defined in Section 2.2.1 but if we use ordinary PN-SEARCH it works. Allis shows it in [1].

One game state in turn t can be visited in the worst case approximately $t!$ times because positions can be claimed in any order. So joining them is a good idea and there exist many enhancements of this enhancement.

In this thesis, we try to use the advantage of symmetry as much as possible. Additionally, we join the game states which are isomorph. See 2.11 for an example on tic-tac-toe game tree. However, in clique game we cannot join all isomorphic game state into one, because finding canonical representation of a colored graph is a difficult problem (no polynomial algorithm is known [9]). Instead, we use a heuristic to find a normalized graph representation (me may not united *all* the isomorphic game states). See section 3.3 for details.

[2] There are no directed cycles because in a positional game, all edges lead to game states with more positions claimed.

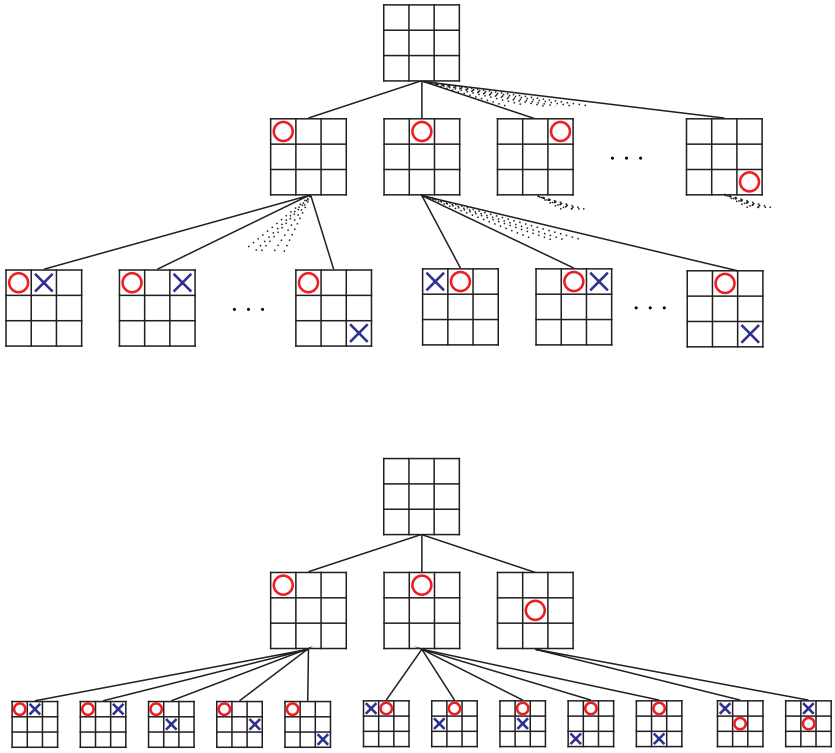


Figure 2.11: Game tree of the tic-tac-toe before and after joining isomorph position.

Note: We use "game tree" even if we talk about a game DAG. It can be a little confusing, but in many cases there is no reason to distinguish between them. We use "leaf node" when we are talk about an undeveloped node.

2.2.3 Weak PN-search

When we use transposition to DAG, one node can be counted in a (dis)proof number many times. We show an enhancement from [8] which tries to minimize this disadvantage.

In weak PN-SEARCH, we count the proof and disproof numbers in a different way:

1. Let n be a leaf node.
   - If the value of n is true, let $p(n) = 0$, $d(n) = \infty$.
   - If the value of n is false, let $p(n) = \infty$, $d(n) = 0$.
   - If the value of n is unknown, let $p(n) = 1$, $d(n) = 1$.
2. Let n be an internal or node with k sons. Then
   $p(n) = \min_{1 \le i \le k}(n_i)$,
   $d(n) = \max_{1 \le i \le k}(n_i) + (k - 1)$.
3. Let n be an internal and node with k sons. Then
   $p(n) = \max_{1 \le i \le k}(n_i) + (k - 1)$,
   $d(n) = \min_{1 \le i \le k}(n_i)$.

2.2.4 Deleting solved subtrees

As a minor optimization to save memory, we can delete a node after it is solved, together with the subtree under it. We discuss this enhancement below in 4.2.2.

2.2.5 No free winning set

In PN-SEARCH, we assign value false to a node in three cases. When it is winning state for the second player, when all positions are occupied, and finally when we determined from its sons that node is false.

It is obvious that when there is no free winning set we can also assign value false and cut it's subtree without visiting it.

3. Further theoretical improvements

In this chapter, we examine various improvements to pn-search and clique game. All results were independently discovered by the author of this thesis.

3.1 Other enhancements of PN-search

Here, we describe several enhancements and also show some solutions for problems which arise when we combine several PN-SEARCH enhancements together.

3.1.1 Ordering of sons

The ordering of sons of some node could be important. When there are two or more sons with a minimal (dis)proof number we usually choose the leftmost, but there could be better variant so we order sons by some heuristic.

3.1.2 Deleting solved sons

When we disproof a son of or node (and symmetrically when we proof a son of and node) we do not solve the node, but we can we can delete the edge between them. The node will have fewer sons so the sum or max or min of numbers could be counted faster, and update ancestors will also run faster. A bigger improvement is when using the WEAK PN-SEARCH. WEAK PN-SEARCH depends on the branching factor, which decreases.

3.1.3 Deleting parallel edges

When we use TRANSPOSITION TO DAG multiple parallel edges may be created, for example on the beginning of clique game. The root has $\binom{N}{2}$ sons. Sons, all of them represent K_N with one edge colored. When we normalise them they will all be the same, and so we have $\binom{N}{2}$ parallel edge from the root to the same node. For a reason similar to the previous subsection, it is a good idea to change parallel edges to a single edge.

3.1.4 Developing a node without creating sons

This enhancement saves memory. The partial game tree is largest when the root is solved. In this state, there is a lot of nodes which are undeveloped, see Figure 3.1a). We could reduce memory requirements if we do not create part of them.

When we are developing a node n, we create and evaluate all n sons, in order to get proof and disproof numbers, as usually. Afterwards, we delete these sons. When we find that the most proving node is a son of n we create all sons of n again, and now let them stay in the partial game tree, see Figure 3.1b). Every node is created at most twice so time requirements rise at most twice.

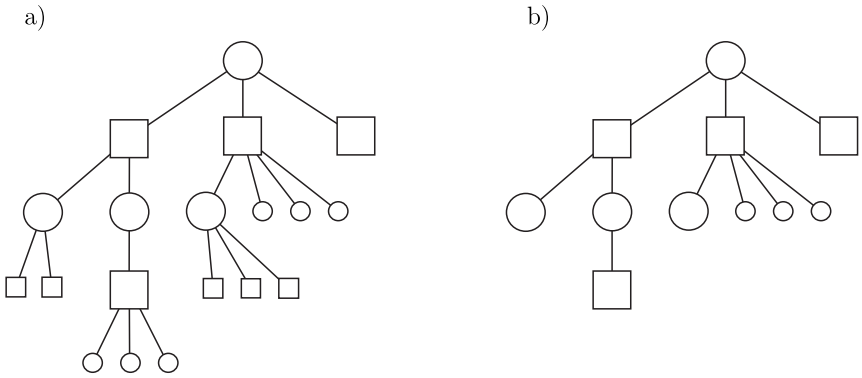


Figure 3.1: The partial game tree a)without b)with enhancement 3.1.4

Perhaps we can save even more memory when we use some counter in nodes and save only the developed sons or all the sons if the percentage or number of developed sons is above some threshold.

3.1.5 Even more symmetry

We can join even more nodes by considering additional symmetry, which may be called "swap of players": it connects game states where positions occupied by the first player in state a are occupied by the second player in state a', and positions occupied by the second player in a are taken by the first player in state a'. We can join the states to a provided we make some other changes - we introduce a new type of edge for this purpose. We call a swap edge an edge between a parent p and a son s, if there is a turn from p to s', that is, we can get to the game state by a move and a swap of players. This is possible by modification of functions *updateAncestors()*, *setProofAndDisproofNumbers()* and when we count thresholds in DF-PN-search. When we follow the swap edge, we have to use proof numbers in place of disproof numbers, and vice versa. The same is done with thresholds.

3.1.6 Cache

There is a possibility to use a node cache, instead of the representation of the whole tree, when we use DF-PN-search. We can use a cache also when we use ordinary PN-search, but we must do some modifications in this case:

1. Do not delete nodes on the current path.
2. Stop *updateAncestors()* when an ancestor is missing from the cache.
3. When a son is missing in the cache, create it again.

Both DF-PN-search and modified PN-search perform poorly when the cache is really small (slightly more than the maximal current path length). In this case, it could happen that algorithm never solves the root. It could happen that the root has two sons s_1 and s_2 and algorithm runs for a while in the subtree of s_1 and then in the subtree of s_2, deleting all the nodes of the subtree of s_1 from the cache. When it returns back below s_1 to improve it, it cannot because

it must create the nodes again (hence deleting the nodes from subtree of s_2) and the situation repeats ad infinitum. Hopefully this does not happen with a big cache.

Here, we specify how to decide which node to delete from the cache. We cannot delete a node from the current path and sons of a node which is just being developed. We can set other rules about which nodes cannot be deleted or set priorities for deleting. This can have significant impact on the performance of the algorithm as we might delete results which took a long time to compute.

When we set too many rules, it can happen that there is no node which can be deleted, and in this case we halt whole program. In our experience with our solver is that it happens very soon after program starts or never.

List of possible rules for deleting from cache:

- Do not delete a node on the current path and sons of a node which is being developed.
- Do not prefer nodes with value true or false – they are more important than the nodes with value unknown (but we need a really big cache for it).
- Do not prefer nodes with many parents – these are more important than nodes with few parents.
- Prefer nodes which are not developed – we do not lose any important information when we delete these.

3.1.7 Variants of *updateAncestors()*

There are three enhancements which influence the function *updateAncestors()*.

The first is using LAST CHANGED NODE to shorten the path.

The second is DF-PN-SEARCH (even if there is no function named *updateAncestors()* and it is updating lazily).

The third is TRANSPOSITION TO DAG. We found no article considering changes in *updateAncestors()* when node has more than one parent.

When we use the ordinary *updateAncestors()* together with TRANSPOSITION TO DAG and one of the other two enhancements, it can happen that some numbers are changed on the current path above an unchanged node. For an example, see Figure 3.2 – the root was changed, but its son on current path was not. Genesis of such situation is illustrated in Figure 3.3 It could also happen that the most proving node found by the algorithm is different than the most proving node as per the definition – see Figure 3.4. This is not as big problem as it seems to be – the algorithm is still correct. An unusual situation can happen when we also use DELETING SOLVED SUBTREES – as a Czech proverb warns we could "saw off a branch which is holding us"

To avoided, it we must use new *updateAncestors()* which updates *all* paths up to root, and better *selectMostProving()* as we show in the next subsection.

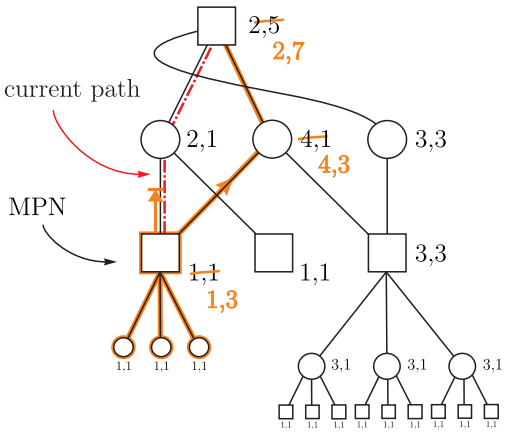


Figure 3.2: Numbers are changed on the current path above an unchanged node.

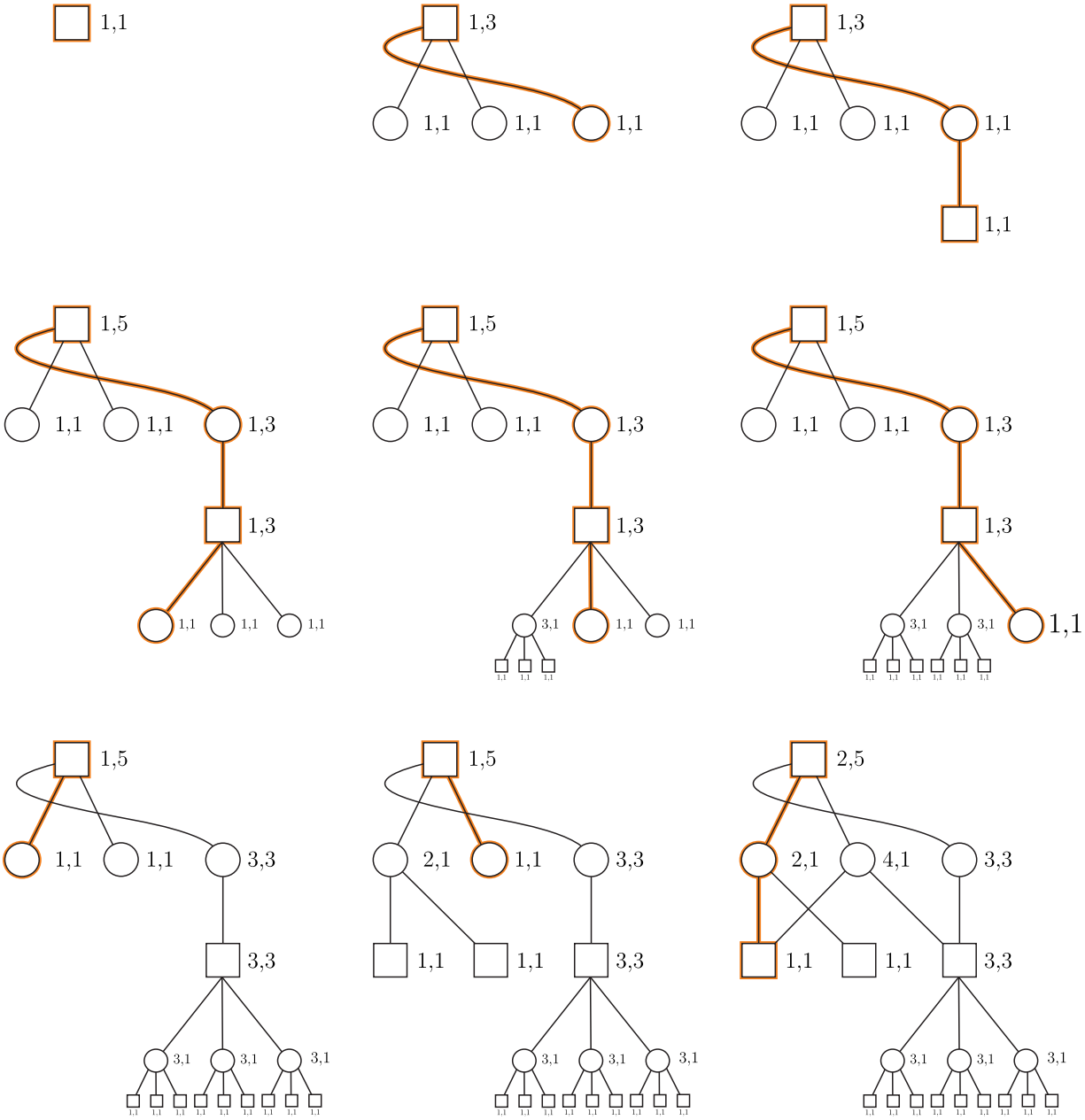


Figure 3.3: Genesis of situation in Figure 3.2.

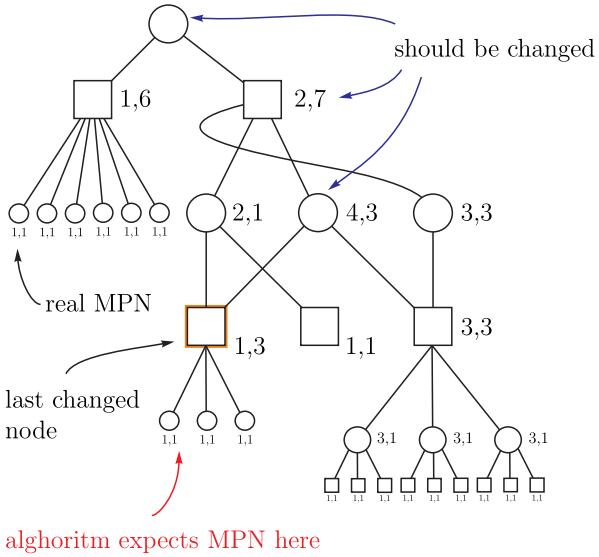


Figure 3.4: The most proving node is in different place than expected.

Simple updateAncestors()

A simple way how to define such *updateAncestors()* is recursively. The function updates a node, and if it changed either proof or disproof number, we call the function recursively on parents. It could happen that some nodes are updated many times. So its parents are also updated many times and the number of updates is growing with the number of possible paths.

Level updateAncestors()

A better solution is to update each node on some level and create a list of parents which need to be updated. After we dispose of duplicates in the list, we do the same on a higher level.

One-visit updateAncestors()

Another solution is to store more information, namely in which iteration of PN-SEARCH was the node last updated. In this case, we can update recursively and stop when we would update some node twice. It could happen that this algorithm does not count right values, but we can still use it as an effective approximation.

3.1.8 Variants of *selectMostProving()*

As shown in Figure 3.2, it could happen that some numbers are changed on the current path above an unchanged node. So the observation from Section LAST CHANGED NODE (2.2.1) does not hold, and it can happen that the algorithm select a most proving node wrongly. There is an example in Figure 3.4. We found several solutions.

Start from the root

The simplest solution is to use neither enhancement LAST CHANGED NODE nor DF-PN-SEARCH. In each iteration, we start the search from the root.

Start at the highest changed level

We can use this solution when we have the current path in a stack, which usually happens when we use recursion. Function *updateAncestors()* will compute the level of the highest changed node. We pop enough nodes from the stack so that the stack has the length equal to the level of the highest changed node. In the next iteration, we can start searching from the node on top of the stack. This way we find the same most proving node as when we are searching from the root. We can use the same argument as in Section LAST CHANGED NODE (2.2.1) to prove it.

Remember the current path

Another variant is to start at the level of the highest changed node of the current path. We hold the whole current path in an array. In every node, which is updated, we test if it is in the current path, which can be done quickly. The level of the updated node can be found as a number of positions claimed by all players or we can save this information in the node.

3.2 Graph representation for the clique game

We need to store a game state during searching the partial game tree. In other words, we need to save which edges are claimed and which color they have. In this section, we introduce several suitable solutions and their properties.

Generally, we use an adjacency matrix, but there are several ways how to represent it in memory. An adjacency matrix is preferred over other structures (adjacency list etc.) because most game states we search are in depth where the colored subgraph is dense.

3.2.1 Basic representation

The adjacency matrix is straightforwardly represented as two dimensional array $N \times N$ of numbers where the value on the position (i, j) is zero if edge (i, j) is free, or color number of the player who claimed it.

3.2.2 Triangle in a line

The previous representation is unnecessarily large. For saving space, we need just two-bit information for every edge, and we can use the symmetry of the adjacency matrix.

In this solution, we store the lower triangular part of the adjacency matrix in a line of bits. Supposing that the size of the computer word is at least $\binom{N}{2} \cdot 2$ bits we can use a single word. If this does not hold, we can use more words instead. For each edge (i, j), we set two bits. First, if the edge is free we set the bit at the

position $(\binom{u}{2} + v) \cdot 2$. Second, to store the color of the edge we set the bit at the position $(\binom{u}{2} + v) \cdot 2 + 1$.

Two small improvements

There is still space for improvement. Imagine that every two bits in a line form a binary number – it can be 0, 1 or 2, never 3. So, the structure from the last paragraph is a number in base three. We can convert this number into base two before storing and convert it back before reading. In this case, the structure has size $\log_2(3)/\log_2(4) \doteq 79\%$ of the original structure. It is the best solution if we care only about space, but converting between bases is slow.

A faster possibility is to break the number into parts by three edges (stored in six bits) and convert these short numbers into binary, which can be done quickly using a translation table. This structure has the size $5/6 \doteq 83\%$ of the original. A similar procedure can be used for parts of n edges. If $n = 5$ the structure has size $8/10 = 80\%$. For a bigger n, the reduction in space usage is not much better.

3.2.3 Adjacency matrix in a line

In our algorithm, we store the whole adjacency matrix of red edges in one line of bits and blue in the second line of bits. Again, we suppose that we have word of size at least N^2. Information if an edge has red color is stored at position $i \cdot N + j$ and also at position $j \cdot N + i$ in the red line, and correspondingly with blue edges and the blue line.

This structure is slightly more than twice as large than TRIANGLE IN A LINE, but it has several advantages in speed of operations, and it is easier for programming which is also important.

A trick: Detect common neighbours

We often need to know which nodes v have red (resp. blue) edges to nodes i and j simultaneously. If we use basic representation, there is one obvious way to find it. We look if red edges (i, v) and (j, v) exist for every node v ($v \neq i$ & $v \neq j$).

If we use ADJACENCY MATRIX IN LINE, there is a better trick. We can use binary mask and binary shift to get i-th and j-th row of the matrix, that is, bits on positions $i \cdot N, \ldots, (i+1) \cdot N - 1$ and $j \cdot N, \ldots, (j+1) \cdot N - 1$. Then we just apply logical *and* on these rows. Every nonzero bit indicates such vertex v.

Finding a claimed winning set

When the red player claims the edge (i, j) we want to know if he wins (determine if he created red K_4). A straightforward solution is to test every edge between every set of four nodes, which contain i and j.

We can also use the trick from the previous subsection to do it faster. We find all the nodes which have red edges to both i and j. Then we test if in this set are distinct nodes u and v such that red edge (u, v) exists. This is illustrated in Figure 3.5a).

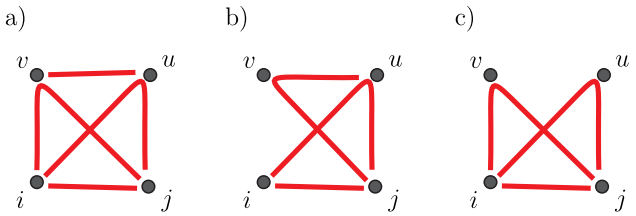


Figure 3.5:

Finding free winning sets

It can be useful to find out if there is a free winning set and how many such sets exist. We can stop searching the partial game tree if there are no free winning sets and we can prefer to turn to edges which are contained in most free winning sets.

Free winning sets can be found in a similar way as the claimed winning sets. We just use complement of opponents line instead of players line.

Finding threats

We want to know if the player who just played created a threat. If he created one threat his opponent has only one possible turn and if he created several threat disjoint on the other two vertices, he wins.

Threat is a state where there are five edges with the same color and one free edge between four nodes. Suppose that the last turn was to (i, j). There are two types of threats.

1. The free edge begins on node i or j (without loss of generality i).
2. The free edge does not contain any of them.

In the first case, we use the trick to find all nodes u which are neighbours of both i and j, and again to find all nodes v which are neighbours of both j and u. If there are such u and v, we check if there is free edge (i, v). If yes then $\{i, j, u, v\}$ is a threat. See Figure 3.5b).

The second case is similar to finding a claimed winning set. We just check if there is a free edge between u, v instead of a red edge. See Figure 3.5c).

Computing vertex degree

We can compute the color degree of any vertex v very quickly. We precompute table of size 2^N containing all the answers for degree queries. Then we just index this table by the v-th row of the adjacency matrix.

3.3 Normalizing function for the clique game

Many nodes with isomorphic graphs and their subtrees are traversed during an ordinary PN-SEARCH. If we want to use the advantage of a high symmetry of the game, we need to join these positions, and hence reduce the number of subtrees we must search.

k	1	2	3	4	5	6	7	8	9	10	
1	1	10	90	360	1260	2520	4200	4200	3150	1260	252
2	1	1	2	9	31	60	96	96	69	28	22
3	1	1	2	7	24	36	63	63	55	28	22
4	1	1	2	6	17	30	47	47	37	16	6

Table 3.1: Efficiency of normalise functions

In each node is a clique game state which is an undirected complete graph with edges colored by three colors (red, blue, free). In an ideal case, we would search the subtree of isomorphism class only once. When we are creating a new graph G we compute a canonical representation H of the isomorphic class where G belongs, and then use H instead of G. The enhancement TRANSPOSITIONS TO DAG causes that the subtree of two isomorphic graphs will be searched at most once.

However, in computing a canonical representative of a graph is a difficult problem (see [9]). So we compute an almost representative graph instead. We call them a normalised graphs.

Definition. *A normalising function is a function, that for every graph G, returns a graph H, such that H is isomorph with G. We we will call H a normalised graph.*

We want to find a normalise function which can be computed fast and which is effective. Here, efficiency means if we apply the function to every graph of an isomorphism class we receive as a small set of normalised graphs as possible, or in other words there are only few distinct normalised graphs which are isomorphic.

We introduce several such functions and compare their effectiveness and speed on graphs[1] with five nodes and k edges colored as in clique game. See Table 3.1.

1 Identity

Trivial to compute but ineffective.

2 Sorting by vertex degree

We can compute the degree of each node and sort the nodes by degree. A basic improvement is to count every red edge M times where M is the number of edges in a complete graph. This improvement decreases the number of normalised graphs which are isomorphic.

3 Sorting by vertex and neighbours degrees

Previous function fails in the case shown in Figure 3.6: first two graphs are normalized using the previous function, but in fact isomorph. In case of a tie, it helps to consider the degree of adjacent nodes. We define functions f and g on the nodes.

[1] undirected 3-colored complete graphs

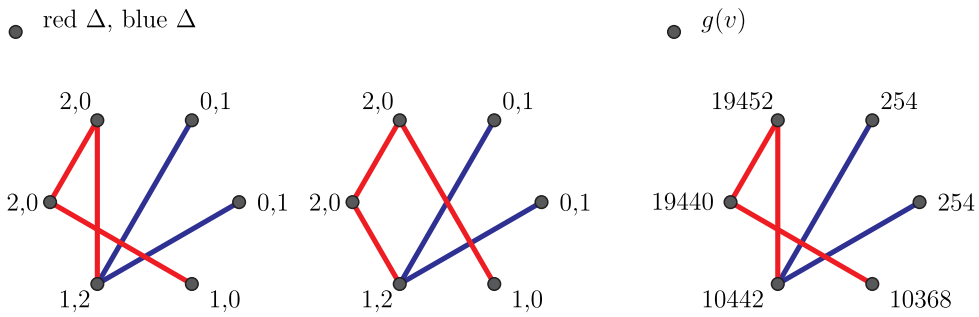


Figure 3.6: Normalised graphs.

$$\begin{aligned} f(v) &= \text{blueDeg}(v) + \text{redDeg}(v) \cdot M^3. \\ g(v) &= f(v) \cdot M^2 + \sum_{(u,v)\text{is red}} f(u) \cdot M + \sum_{(u,v)\text{is blue}} f(u). \end{aligned}$$

Then, the normalising function just sorts the vertices by g. See third graph in Figure 3.6

This function is asymptoticly slower than the previous one, but still polynomial and more effective.

4 Ideal case

We assign the canonical representation of the graph. This is the most effective solution but hard to compute as we noted before. This function was not implemented so we cannot measure its speed. Efficiency was computed in the following way: We use the previous normalising function and for every two graphs with the same values of g we straightforwardly test if they are isomorphic.

4. Experimental results

In this chapter, we describe directly our solver and results illustrating which enhancements help with solving the clique game.

4.1 Clique game

We solved the clique game for $K = 4, N = 6$ and $N = 7$. In this section, we describe the partial game trees which were created during solving that games. We use WEAK PN-SEARCH and TRANSPOSITION TO DAG with normalising function SORTING BY VERTEX AND NEIGHBOURS DEGREES and NO FREE WINNING SET. We call this combination K4WEAK.

In the first case ($N = 6$), the algorithm needs 337 iterations to solve the game. In the second case, the algorithm needs 16489 iterations and solves the game in three seconds on a desktop PC (AMD Athlon 3GHz).

Tables 4.1 and 4.2 show the properties of the game tree for each level.

The column ALL illustrates how many game states could exist on some level L. This number was computed as the number of edge-colored complete graphs with N nodes with $\lfloor L/2 \rfloor$ red edges and $\lfloor (L-1)/2 \rfloor$ blue edges. Numbers from the first column are not exact because they count also game states which cannot happen in a real game, for example two K_4 with the same color.

In the column CREATED we indicate the number of nodes which were actually created on each level. In the parentheses we indicate how many of the nodes were first evaluated as false respectively marked as a threat. There was no node which was first evaluated as true – this is a good sign, because nodes with value true never help to disprove anything and the algorithm avoids them.

In the column SOLVED we indicate how many nodes were solved on level L. All of them were solved as an false.

4.2 Comparing enhancements

In this section, we compare the changes caused by various enhancements, as measured by several tests. We prepare some combination of enhancement and describe each whole combination by one name.

The criteria we use for comparison:
- time needed to solve the game (in seconds)
- space which the program uses (number of created nodes)
- number of iterations
- level of the deepest developed node

The number of created nodes is a representative value for space requirements when we do not use any enhancement which deletes nodes. In the time test we set the time limit for solving the game to 60 seconds.

level	ALL	CREATED	SOLVED
$\sum$	10 165 779	1 241 (0,12)	110
0	1	1 (0, 0)	1
1	16	1 (0, 0)	1
2	240	2 (0, 0)	1
3	1 680	7 (0, 0)	6
4	10 920	26 (0, 0)	5
5	43 680	73 (0, 0)	28
6	160 160	190 (0, 0)	14
7	400 400	306 (0, 0)	31
8	900 900	273 (0, 0)	23
9	1 441 440	363 (0, 12)	0
10	2 018 016	0	0
11	2 018 016	0	0
12	1681 680	0	0
13	960 960	0	0
14	411 840	0	0
15	102 960	0	0
16	12 870	0	0

Table 4.1: Properties of the game tree for (6,4)-clique game.

level	ALL	CREATED	SOLVED
$\sum$	6 377 181 825	65 979 (92, 605)	2 356
0	1	1 (0, 0)	1
1	22	1 (0, 0)	1
2	462	2 (0, 0)	2
3	4 620	7 (0, 0)	7
4	43 890	29 (0, 0)	7
5	263 340	107 (0, 0)	44
6	1 492 260	396 (0, 0)	26
7	5 969 040	1 247 (0, 0)	242
8	22 383 900	3 812 (0, 0)	76
9	62 674 920	9 108 (0, 49)	752
10	162 954 792	16 938 (0, 96)	226
11	325 909 584	17 303 (9, 113)	452
12	597 500 904	8 252 (0, 119)	293
13	853 572 720	7 280 (43, 203)	159
14	1 097 450 640	825 (0, 0)	22
15	1 097 450 640	672 (40, 25)	46
16	960 269 310	0	0
17	640 179 540	0	0
18	355 655 300	0	0
19	142 262 120	0	0
20	42 678 636	0	0
21	7 759 752	0	0
22	705 432	0	0

Table 4.2: Properties of the game tree for (7,4)-clique game.

N	name	time	iterations	created nodes	max. level
6	K4	1.41s	413	1 406	10
6	K4Weak	1.03s	337	1 241	8
7	K4	12.34s	38 716	128 256	15
7	K4Weak	2.79s	16 489	65 979	14
6	basic	5.38s	5 541	6 353	14
6	weak	2.48s	4 237	6 079	14

Table 4.3: Comparing weak PN-search variants.

4.2.1 Weak PN-search

The enhancement weak PN-search 2.2.3 helps in every combination with other enhancements which we tried. The results are in Table 4.3.

K4Weak and K4

These two combinations use PN-search with last changed nodes, no free winning set and normalise function sorting by vertex and neighbours degrees. Basic also uses weak PN-search.

weak and basic

These two combinations are similar to previous ones but they both do not use the enhancement no free winning set. They both do not solve game for $N = 7$ within the time limit.

4.2.2 Deleting solved subtrees

This enhancement 2.2.4 does not help, not even when we use it only with an ordinary PN-search. It does not even considerably save memory.[1]

The reason is that most of the time the game tree is being developed and not many nodes are being solved. In the later part of the game when nodes are being solved and deleted we do not need the memory which is saved. The result is that when we run the program with and without this enhancement it needs approximately the same amount of memory and program with delete solved subtrees runs longer. For this reason it is now not implemented in our code. Other reason is that it causes many complications when we use it together with transposition to DAG, because deleting solved subtrees can delete node n which we might need because there is another path to n.

4.2.3 No free winning set

This enhancement radically decreases the level of the deepest developed node. The number of nodes in subsequent levels of the complete game tree exponentially rises. For this reason the enhancement no free winning set is necessary to solve the clique game for $N = 7$. Results for $N = 6$ are in Table 4.4.

[1] In this case we measured actual memory allocation, not number of created nodes.

N	name	time	iterations	created nodes	max. level
6	WEAK	2.48s	4 237	6 079	14
6	K4WEAK	1.03s	337	1 241	8
6	BASIC	5.38s	5 541	6 353	14
6	K4	1.41s	413	1 406	10

Table 4.4: Comparing NO FREE WINNING SET variants.

4.2.4 Ordering of sons

This enhancement is used in each variant of our solver. If we want not to use it, we must randomly reorder sons. That is happening when we sort sons by their hash in Section 4.2.5.

A first heuristic is when we use the ordinary PN-SEARCH and *developNode*() creates sons this way that it tries adding the new edge in alphabetical order.

A second heuristic is when we use PN-SEARCH with TRANSPOSITION TO DAG with normalising function SORTING BY VERTEX DEGREE or SORTING BY VERTEX AND NEIGHBOURS DEGREES. We first try colored the new edge to area with a low degree. It is good for the second player when all vertices have similar degree. When the second player has draw strategy, most nodes are solved as false. The ordering of sons of an or node is not important because we must solve them all. Ordering of an and node is good so this heuristic works.

We try one other heuristic. When we are creating sons by adding the edge e we count the free winning sets including e by function in Section 3.2.3. Then we sort the sons so that the leftmost one has the largest number. This heuristic was worse than the second, so currently it is not implemented.

4.2.5 Deleting parallel edges

As we explain in Section 3.1.3 there could be parallel edges if we use the enhancement TRANSPOSITION TO DAG. We implemented DELETING PARALLEL EDGES, that is we delete duplicate sons in the function *updateAncestors()*. We first create all the sons then normalise then, second we sort them by their hash and then check if nodes with the same hash are actually same, then delete duplicities.

However, after this the ordering of sons is different and it causes the partial game tree to be different. This could be a disadvantage and could overweight described advantages. The differences of the partial game trees are the reason why we do not measure this enhancement.

4.2.6 DF-PN-search

DF-PN-SEARCH was described in Section 2.2.1. In our case, it does not help as much as we expected. The reason is that it cannot be easily combined with NO FREE WINNING SET.

To check if there are free K_4's in a node n is a slow operation. To make it faster, we postpone it to the time when n is developed. Then when we create a son by claiming an edge e we check if there is a free K_4 containing e by the function described in Section 3.2.3. When there is no free K_4 for any e, we assign the value

N	name	time	iterations	created nodes
6	WEAK	1.34s	4 237	6 079
6	DFPN	0.96s	1 018	2 228

Table 4.5: Comparing DF-PN-SEARCH variants.

false to n. When we use POSTPONE TESTING together with DF-PN-SEARCH, the algorithm continues searching the subtree of n even if n is solved.

NO FREE WINNING SET helps more then DF-PN-SEARCH. For this reason DF-PN-SEARCH is not included in the fastes variant. We show how it helps when we use it for $N = 6$ without NO FREE WINNING SET in Table 4.5.

4.3 More about our implementation

Our software was created in order to solve the clique game for $K = 4$ and as high N as possible. We knew that there is no chance to solve it for all N up to 17 because just to check if a strategy is winning needs checking so many states that it could not be done.

Our code is in C language and it is on github[2] licensed as a free software under GNU GPL 3.

During the creation of this project, we need to test if some heuristic helps or not. First we tried to implement something and later we decided whether to keep it or delete it. Later we decided to enclose enhancement into "ifdefs". The result is that we can try combinations of more enhancements and we can compare them easily. There are thousands of possible combinations, but not all of them works. List of combinations which were tested is in Makefile.

4.3.1 Implemented enhancements

We implemented almost all the enhancements which are mentioned in this thesis. The enhancements which we do not compare helped in all cases and currently it is not possible to switch them off in our code.

We implemented all the enhancements from the second chapter and the first four enhancements from the third chapter. We use the last described variant of *updateAncestors()* and *selectMostProving()*. We did not implement enhancements DEVELOPING A NODE WITHOUT CREATING SONS from Section 3.1.4. For storing nodes we use ADJACENCY MATRIX IN LINE. We implemented the first three normalise functions described in Section 3.3 – the third normalize function SORTING BY VERTEX AND NEIGHBOURS DEGREES is the best in terms of efficiency and still fast enough.

4.3.2 The fastest variant

Our fastest variant is K4WEAK. This variant is WEAK PN-SEARCH with enhancements LAST CHANGED NODES, NO FREE WINNING SET and normalise function SORTING BY VERTEX AND NEIGHBOURS DEGREES.

[2] https://github.com/jitka/rocnikac/tree/master/src

Most of the developed nodes are in levels in a middle of the complete game tree, see Tables 4.1 and 4.2.

Solver spends most of the time by developing nodes. The program spends 21% of execution time in *cacheFind()* and at least another 23% in *normalise()* function, which are both used in *developNode()*. Most of the rest time is consumed by different functions for working with game state, which are mainly used also in *developNode()*.

Only 6% is consumed by *updateAncestor* and less than 1% by *selectMostProving()*. This is not surprising, because in 13497 of 16489 iterations the new most proving node is a son of the previous one.

Time needed to solve the root depends on the size of the partial game tree. When we want to make the solver faster, we should try to reduce the size of the partial game tree.

Conclusion

In this thesis we introduced algorithms for solving strong positional games, mainly PN-search and its enhancements.

We focused on games with high symmetry. The high symmetry allows, by joining several isomorph game state into one, to narrow the partial game tree. When partial game tree is narrow, we can search in it deeper. The other feature which keeps the tree narrow is that we do not deal with high branching factor. For example, we can compare our game with the game Tzaar, which has high branching factor and low symmetry. Despite of clever heuristics and optimizations the best bot playing the game searches the game tree only into depth 4 [10]. For comparison, one variant of our solver for (7,4)-clique game solved the game tree to depth 16.

There are other features distinguishing games with high symmetry. It is much more important to join the nodes. We must treat carefully the lifetime of the nodes, that is, we do not want to delete nodes from cache especially when their solution was expensive because as we may find another ancestor of these nodes. Finally in games with high symmetry much more often happens situation described in Figure 3.2.

Further work

There are many enhancements which we did not implemented and even do not mention. For example $1+\epsilon$ trick.

It could also be useful to implement enhancement "dveloping a node without creating sons" in Section 3.1.4 and study if it can significantly reduce the partial game tree. Also to might it may be interesting to try it on other games.

Of course, there is space for more new enhancements and heuristics.

Bibliography

[1] Allis, Victor and others. *Searching for solutions in games and artificial intelligence*. PhD thesis. University of Limburg, Maastricht, The Netherlands, 1994. ISBN 9090074880.

[2] Diestel, R. *Graph Theory*. Electronic Edition, Springer-Verlag Heidelberg, New York, 2005.

[3] Beck, József. *Combinatorial games - Tic-Tac-Toe Theory*. Cambridge University Press, 2008. ISBN 0521461006.

[4] Knuth, Donald E., Moore, Ronald W. *An analysis of alpha-beta pruning.* Artifical Intelligente, 1975.

[5] Tizhoosh, H.R. *Oppositional Concepts in Computational Intelligence*. Springer, 2008. ISBN 9783540708261.

[6] Pawlewicz, Jakub; Lew Lukasz. *Imploving depth-first PN-search: 1+ϵ trick*. Springer-Verlag, Berlin, Heidelberg, 2007. ISBN 3540755373.

[7] Nagai, A., Imai, H. *Aplication of df-pn+ to Othello Endgames*. Game Programming Workshop in Japan, 1999, pp 16–23.

[8] Hashimoto, Junichi, Hashimoto, Tsuyoshi, Iida, Hiroyuki, Ueda, Toru. *Weak Proof-Number Search*. Springer, Berlin, Heidelberg, 2008. ISBN 9783540708261.

[9] Read, Ronald C., Corneil, Derek G. *The graph isomorphism disease*. Journal of Graph Theory 1 (4), 1977, pp 339–363.

[10] Personal communication with Pavel Veselý.