REYKJAVÍK UNIVERSITY
HÁSKÓLINN Í REYKJAVÍK

GTQL: A Query Language for Game Trees

Jónheiður Ísleifsdóttir

Master of Science
December 2007

Reykjavík University - School of Computer Science

M.Sc. Thesis

GTQL: A Query Language for Game Trees

by

Jónheiður Ísleifsdóttir

Thesis submitted to the School of Computer Science
at Reykjavík University in partial fulfillment
of the requirements for the degree of
Master of Science

December 2007

Thesis Committee:

Dr. Yngvi Björnsson, supervisor
Associate Professor, Reykjavík University

Dr. Martin Müller
Associate Professor, University of Alberta

Dr. Björn Þór Jónsson
Associate Professor, Reykjavík University

The undersigned hereby certify that they recommend to the School of Computer Science at Reykjavík University for acceptance this thesis entitled **GTQL: A Query Language for Game Trees** submitted by Jónheiður Ísleifsdóttir in partial fulfillment of the requirements for the degree of **Master of Science**.

_____

Date

_____

Dr. Yngvi Björnsson, supervisor
Associate Professor, Reykjavík University

_____

Dr. Martin Müller
Associate Professor, University of Alberta

_____

Dr. Björn Þór Jónsson
Associate Professor, Reykjavík University

The undersigned hereby grants permission to the Reykjavík University Library to reproduce single copies of this thesis entitled **GTQL: A Query Language for Game Trees** and to lend or sell such copies for private, scholarly or scientific research purposes only.

_____
Date

_____
Jónheiður Ísleifsdóttir
Master of Science

GTQL: A Query Language for Game Trees

by

Jónheiður Ísleifsdóttir

December 2007

Abstract

The search engines of high-performance game-playing programs are getting increasingly complicated as more and more enhancements get added. To maintain and further enhance such complex engines is an involved task, and the risk of introducing bugs or other unwanted behavior during modifications is substantial. In this thesis we introduce the Game Tree Query Language (GTQL), a query language specifically designed for game trees. The language can express queries about complex game-tree structures, including hierarchical relationships and aggregated attributes over subtree data. We also discuss the design and implementation of the Game Tree Query Tool (GTQT), software that allows efficient execution of GTQL queries on game-tree logs. This tool helps program developers to gain added insight into the search process of their engines, as well as making regression testing easier. Empirical results are presented measuring the tool's efficiency as well as its effectiveness in finding anomalies in large game trees.

GTQL: Fyrirspurnarmál fyrir Leikjatré

eftir

Jónheiður Ísleifsdóttir

Desember 2007

Útdráttur

Leitarvélar háþróaðra leikjaforrita verða sífellt flóknari eftir því sem fleiri viðbótum er bætt við þær. Það er margslungið verk að viðhalda og bæta þessar flóknu vélar og hætta á villum og annarri óæskilegri hegðun er töluverð á meðan á breytingum stendur. Í þessari meistararitgerð kynnum við Game Tree Query Language (GTQL), sérhannað fyrirspurnarmál fyrir leikjatré. Í fyrirspurnarmálinu er hægt að búa til fyrirspurnir varðandi flókna byggingu leikjatrjáa, eins og sambönd milli hnúta og talningu gagna úr undirtré. Við ræðum einnig hönnun og útfærslu á Game Tree Query Tool (GTQT), hugbúnaði sem leyfir skilvirka keyrslu GTQL fyrirspurna á leitartré úr skrám. Þessi hugbúnaður eykur innsýn þeirra sem þróa leikjaforrit í þá leit sem leikjavélarnar framkvæma auk þess að auðvelda endurteknar prófanir. Kynntar eru niðurstöður tilrauna sem sýna fram á skilvirkni og árangur við að finna afbrigðileika í stórum leikjatrjám.

To Guðrún Hulda, who made me realize that I could do anything.

Acknowledgements

I would like to thank my supervisor Yngvi Björnsson for his patience, support and good advice during this project. Thanks to everyone at CADIA for stimulating conversations and wonderful company, especially to Guðný, Hilmar, Pálmi, Marta and Gunni for their insightful comments. Last but not least I want to thank my family and friends for always being there for me and supporting me in everything I do.

Publications

A part of the material in this thesis was published as an extended abstract *"Tools for Debugging Large Game Trees"* (Björnsson & Ísleifsdóttir, 2006) and presented as an invited talk at the 11th Games Programming Workshop in Japan in 2006 and as *"GTQL: A Query Language for Game Trees"* (Björnsson & Ísleifsdóttir, 2007) a paper and lecture presented at the 12th Games Programming Workshop in Amsterdam in June 2006.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

The development of high-performance game-playing programs for board games is a large undertaking. The search engine and the position evaluator, the two core parts of any such program, become quite sophisticated when all the necessary bells and whistles have been added. To maintain and further enhance such complicated software is an involved task, and the risk of introducing bugs or other unwanted behavior during modifications is substantial. For example, different search enhancements affect each other in various ways, and changing one may decrease the effectiveness of another. Similarly, tuning an evaluation function to better evaluate specific types of game positions may have adverse side effects on others.

A standard software-engineering approach for verifying that new modifications do not break existing code is to use *regression testing*. To a large extent this approach is what game-playing program developers use. They keep around large suites of test positions and verify that the modified program evaluates them correctly and that the search finds the correct moves. Additionally, new program versions play a large number of games against different computer opponents to verify that the newly added enhancements result in genuine improvements. Nonetheless, especially when it comes to the search, it can be difficult to detect abnormalities and they can stay hidden for a long time without surfacing. These can be subtle things such as the search extending useless lines too aggressively, or poor move-ordering resulting in unnecessarily late cutoffs. Neither of the above abnormalities result in erroneous results, but instead seriously degrade the effectiveness of the search. Such anomalies can be hard to detect, especially because contemporary game-playing programs typically explore hundreds of thousands of possibilities per second. To detect these anomalies one typically must explore and/or gather statistics about the search process. Until now the main way to do that is to go through logs and traces of

game trees, a very tedious and time consuming process. The contribution of this thesis is threefold:

- The Game-Tree Query Language
- The Game-Tree Query Tool
- Experiments in chess

The *Game-Tree Query Language (GTQL)* is a language for querying game-tree log files. It is a part of a larger suite of tools intended to alleviate the difficulty of debugging large game trees. The query language allows the game-program developers to gain better insight into the behavior of the search process and makes regression testing easier. A programmer can now keep around a set of pre-defined queries that check for various wanted or unwanted search behaviors (such as too aggressive extensions or large quiescence searches). When a new program version is tested, it can be instructed to generate log files with search trees, and the queries are then run against the logs to verify that the search is behaving in accordance with expectations.

The language was implemented in connection with an already existing library that facilitates logging of search trees to binary files and reading from these files. Because typical game trees have tens of millions of nodes we constructed a one-pass algorithm. It gathers all information needed to answer a query in a single pass through the tree, and it is implemented in the *Game-Tree Query Tool (GTQT)*. Results of efficiency testing of GTQT on generated trees of different shapes and sizes show that the implementation is efficient and scales well with large trees.

Experiments were run using GTQT where selected GTQL-queries were used to query actual game trees generated by the state-of-the-art chess-playing program Fruit (Letouzey, 2005). The result of these queries reveal several interesting anomalies in the game trees that could possibly be used to improve the program.

The remainder of this thesis is organized as follows. Chapter 2 gives an insight into the algorithms and extension methods used in current game-playing programs. In Chapter 3 the syntax and semantics of the Game-Tree Query Language is described and examples of queries provided. In Chapter 4 the implementation of the Game-Tree Query Tool for executing GTQL queries is described and the one-pass algorithm presented. We also measure the efficiency and scalability of the implementation on artificially generated game trees. Chapter 5 shows results of queries made on game trees from chess and presents several interesting search anomalies found using GTQT. In Chapter 6 we conclude and propose future work.

Chapter 2

Background

In this chapter we give a brief overview of game-tree search. This helps the reader to better understand the semantics of the game-tree queries we present in later chapters as well as the intricateness of our new one-pass algorithm. First, we describe the minimax rule and how it is used to find the minimax value of a game tree. Second, we discuss the $\alpha\beta$ algorithm and the minimal tree as well as the Principal Variation Search algorithm and last we talk about some additional search enhancements.

2.1 Minimax Search

The search scope of a zero-sum, perfect-information, two-player game can be represented as a game tree. The starting position of the search becomes the root of the tree and each edge represents a move from that position. The moves result in other positions, that are child nodes of the root. These nodes make up the first *ply* of the tree. The nodes on the first ply have other moves that result in new child nodes and so on.

Because game trees grow exponentially with depth they are usually not searched all the way to a terminal node but only to a certain depth. A terminal node is a position where we can say for certain that one player either wins, draws or loses the game. The fact that we do not search until reaching a terminal node means that we do not always know the true value of the leaf positions in the tree, i.e. the positions where the depth limit has been reached. They must therefore be evaluated in some way. This is done with a *heuristic evaluation function* that takes many things into account, such as material advantage, placement of material, and other domain specific attributes. The evaluation function returns a value,

Algorithm 1 negamax(node, depth)

```
1: Children ← getChildren(node)
2: if depth <= 0 or Children is empty then
3:    return f(node)
4: highest ← -∞
5: for all child in Children do
6:    value ← -negamax(child, depth-1)
7:    if value > highest then
8:       highest ← value
9: return highest
```

which is a measure of the ”goodness” of this position relative to other positions in the game.

When searching a game tree from the root position, the goal is to use the values from the leaf nodes to determine the heuristic *minimax* value of the root. This is done by traversing the tree in a depth-first fashion and backing up the values from the leaves to the root according to the minimax rule. The edge that leads to the child with the highest value is then considered to be the best move.

The minimax rule assumes that both players in the two-player game, called *min* and *max*, are out to maximize their own gain. If we assume that the evaluation function always returns the values according to how beneficial they are for max, then max always chooses the maximum value available from its children and min chooses the minimum value. Another formulation of this is called *negamax* (Knuth & Moore, 1975) and it is shown as Algorithm 1. There the values are negated with each ply so both max and min can maximize over their child values and the evaluation function takes into account who’s turn it is to play from the position to be evaluated. This formulation simplifies the algorithm and we will use it in the text from now on.

2.2 The Minimal Tree and $\alpha\beta$

Although minimax works in theory its practicality suffers from its brute-force nature. When we expand a game tree using the minimax rule there are many nodes that are expanded that cannot change the minimax value of the root. These nodes can be pruned from the tree, leaving behind a so called *minimal tree* (Marsland & Campbell, 1982). The minimal tree, sometimes called the *critical tree* (Björnsson & Marsland, 2001), returns the same minimax value to the root as a fully expanded tree, but has much fewer nodes.

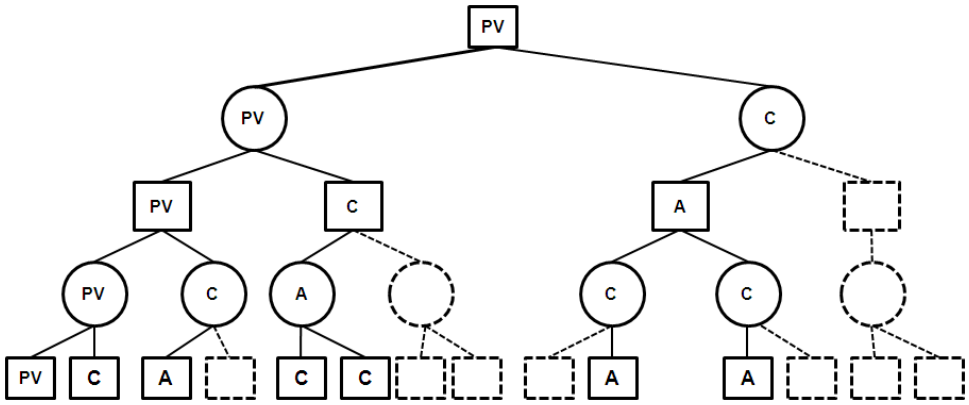


Figure 2.1: The structure of a minimal game tree

The *Alpha-Beta ($\alpha\beta$)* algorithm adds *pruning* to minimax and if a best move is always tried first it prunes the game tree to a minimal tree.

The minimal tree has a certain structure, that is shown in Figure 2.1. A formal definition (Björnsson & Marsland, 2001) states that the minimal tree has three types of nodes called *pv-, cut- and all-nodes* (Marsland & Popowich, 1985). For a tree to qualify as a minimal tree the structure must fulfill certain conditions where the different kinds of nodes play a big part. The root of the game tree is a *pv-node*. At each *pv-node* the child that has the highest minimax value is also a *pv-node* but all other children are *cut-nodes*. If several child nodes have the same minimax value it does not matter which one is chosen. A *cut-node* must have at least one child that has a lower minimax value than the next *pv-node* predecessor of that child. That child is an *all-node*. If there are several nodes that fulfill this condition it does not matter which one is picked. Other children of the *cut-node* are not searched. At an *all-node* on the other hand all children are *cut-nodes* and they are all searched.

If the $\alpha\beta$ algorithm has perfect move ordering it will only expand this minimal tree, constructed of the principal variation (best line of moves) and the least amount of nodes needed to make sure that it is the best path. In theory this works well but in practice it is rare that the move ordering of $\alpha\beta$ is sophisticated enough to expand only the minimal tree and if the moves are expanded in worst case order, $\alpha\beta$ expands the same number of nodes as minimax.

Algorithm 2 uses the bounds α and β as a search window, to guide the search and decide where cutoffs occur. As the search goes on the bounds are adjusted so nodes with values that don't change the minimax value are not expanded. We see that Algorithm 2 is similar to minimax but now the search terminates if the value returned from a child is higher than the β value. The β value is the upper bound on the window we are searching and no value

Algorithm 2 $\alpha\beta(\alpha, \beta, \text{node}, \text{depth})$

```
1:  Children ← getChildren(node)
2:  if depth <= 0 or Children is empty then
3:      return f(node)
4:  highest ← -∞
5:  for all child in Children do
6:      value ← -αβ(-β, -max(α, highest), child, depth-1)
7:      if value > highest then
8:          highest ← value
9:          if highest >= β then
10:             return highest
11: return highest
```

higher than that will ever be chosen by the opponent. It would therefore be pointless for the player whose turn it is now to try to find an even higher value and best to return this value immediately. This is how cutoffs occur and nodes where only a part of the children are expanded are called *cut-nodes*. If a cutoff does not happen the algorithm continues to expand all children and then returns the highest value.

Many enhancements have been added to the $\alpha\beta$ algorithm over the years. They include algorithms such as NegaScout (Reinfeld, 1983) and the Principal Variation Search (Marsland & Campbell, 1982) algorithm, as well as more external enhancements that can be used by any $\alpha\beta$ algorithm such as transposition tables, iterative deepening and selective quiescence search.

2.3 Principal Variation Search

Principal variation search (PVS) was first presented by Marsland and Campbell (Marsland & Campbell, 1982). PVS consists of two separate functions. The main one is the PVS function that drives the search and then there is the null-window search (NWS) function. Algorithm 3 shows both of these functions. PVS starts, like $\alpha\beta$, by checking for terminal nodes. It then continues and searches the first child with the original window, assuming moves have been ordered such that the first move is the best. The node resulting from that move is therefore expected to be the node with the highest value called the *pv-node*. The algorithm then goes on to search the rest of the moves with a null-window, calling the NWS function.

The null-window function is similar to $\alpha\beta$, but instead of using the usual window increases β by ϵ (the smallest granularity of values returned by the evaluation function). If

Algorithm 3 PVS(α, β, node, depth)

```
1:  function PVS(α, β, node, depth)
2:  Children ← getChildren(node)
3:  if depth <= 0 or Children is empty then
4:     return f(node)
5:  highest ← -PVS(-β, -α, child_1, depth-1)
6:  for all child_i in Children where i > 1 do
7:     if highest >= β then
8:        return highest
9:     α ← max(α, highest)
10:    value ← -NWS(-α, child_i, depth-1)
11:    if value > α and value < β then
12:       value ← -PVS(-β, -value, child_i, depth-1)
13:    if value > highest then
14:       highest ← value
15: return highest
16:
17: function NWS(β, node, depth)
18: Children ← getChildren(node)
19: if depth <= 0 or Children is empty then
20:    return f(node)
21: highest ← -∞
22: for all child in Children do
23:    value ← -NWS(-β+ϵ, child, depth-1)
24:    if value > highest then
25:       highest ← value
26:       if highest >= β then
27:          return highest
28: return highest
```

the null-window search returns a value that is greater than α (i.e. it fails high) the node needs to be *re-searched* with the larger window because a new *pv-node* might have been found with a higher value than the previous one. If a higher value is found it is saved in the variable highest and the re-searched node is a new *pv-node*.

Figure 2.2 shows a tree that has been searched with PVS. The gray nodes are the *pv-nodes* and the dotted lines represent cutoffs. The values inside the nodes are the backed up minimax values of the tree. The changes to the α and β values during the search are displayed inside the parenthesis beside the nodes and when there is only one value in the parenthesis it represents the β value in the NWS. The figure shows that nodes 7, 9, 13, 15 and 17 are all *cut-nodes* since they only search some (in this case one) of their children before cutoff occurs. Nodes 10 and 14 on the other hand are *all-nodes* because they search

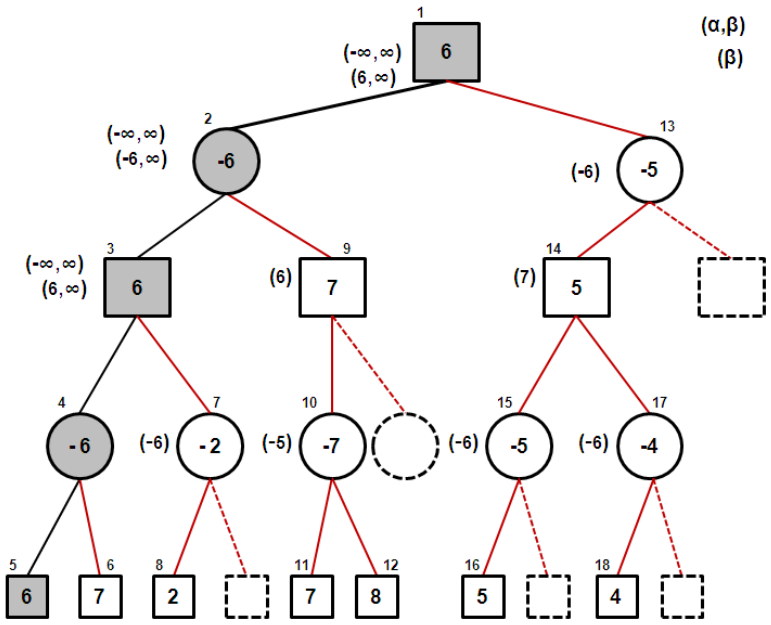


Figure 2.2: A Principal Variation search tree with cutoffs

all of their children. No *re-search* was needed in this tree, but if the move ordering is not sufficiently good *re-searches* sometimes occur.

2.4 Search Enhancements

State of the art game-playing programs not only use the algorithms previously described but enhance them with several methods. We describe the most important ones.

2.4.1 Quiescence Search

The term *quiescent position* refers to a position that can be safely evaluated by a static evaluation function and expected to return an accurate assessment (Marsland, 1986). Sometimes when searching in game trees we encounter positions that are not quiescent. In chess these are positions where there are impending captures, promotions or checks. When this happens we do not want to evaluate the position directly because some moves might lead to positions with values that are very different from the value that the evaluation would return for this position. To solve this, the moves that cause the unpredictability are searched deeper until their more quiescent descendants are found (Marsland & Reinefeld, 1993). They can then be evaluated and a more trusted value backed up to the position that was to be evaluated. Because of the added expense of these extra searches, it is im-

portant that the search is only done on the moves that cause the volatility of the position and that those moves are only expanded until a quiescent position is found.

2.4.2 Transposition Table

A transposition table is a structure that holds some or all of the positions that have been searched so far. It is usually implemented as a hash table; in chess a Zobrist (Zobrist, 1970) hash key is the most common way to code a position into a key to the table. Having a unique key for each position gives us direct access to it. The usual fields that are stored for each position are: the best move; value of the subtree; height of the subtree that produced the value; and two flags that indicates whether the value is an upper or lower bound or the true value of the position (Marsland, 1986).

A transposition table prevents us from searching subtrees that have been fully searched again and is also an important factor in keeping track of the principal variation moves when using iterative deepening.

2.4.3 Iterative Deepening

The term iterative deepening is used when the search depth is increased incrementally, usually by one ply, until the nominal search depth is reached (Marsland & Campbell, 1982). This means that the algorithm first searches to depth one, next to depth two and so on. Searches often have a fixed nominal depth or time limit. One of the original thoughts behind using iterative deepening was to control the time a search takes. Many game-playing programs have a fixed time to "think" and we don't want to start a search that will not be able to finish in time, although we still want to search as deep as we can to get the best possible move from our position. Iterative deepening does this by generating a separate tree for each iteration and therefore the algorithm can return the best possible value at any time.

The added overhead of searching the top plies over and over again is evened out by the fact that we can use the information gathered in the previous iteration to guide the search better and prune more branches from the tree. If we can cause cutoffs sooner with better move ordering we end up saving time. The PVS algorithm is frequently run with iterative deepening and takes advantage of this feature. To be able to utilize this we need some method of keeping track of the best moves from the previous iteration. This is usually done with a transposition table that is shared between iterations. By doing this the search can use all the information gathered during the previous iteration to guide the next one

straight to the best moves and prevent subtrees that have already been searched from being searched again.

2.5 Summary

We have now explained algorithms and enhancements commonly used by contemporary game-playing search engines. This is important background information that we will draw from in the following chapters to give realistic examples of useful GTQL-queries and their result when querying game trees. Next we introduce the Game Tree Query Language.

Chapter 3

Game-Tree Query Language

In this chapter we first describe the syntax and the semantics of GTQL. Then, to highlight the expressiveness of the language, we give several examples of GTQL queries that one might want to ask about game trees generated by $\alpha\beta$ based search engines. We also discuss the expressiveness of the language and give an overview of related work. A complete EBNF description of the GTQL syntax is provided in an appendix.

3.1 Syntax and Semantics

A GTQL-query consists of three parts: a *node-expression* part, a *child-expression* part, and a *subtree-expression* part:

```
node:<node-expression>;
child:<child-expression>;
subtree:<subtree-expression>
```

The keywords *node*, *child*, and *subtree* indicate the type of the expression that follows. If the expression is empty then the keyword (along with the following colon) may be omitted. A semicolon is used to separate the expressions and can be omitted if there is only one expression in the query. Valid expressions must be formed such that they evaluate to either true or false. The language is case sensitive and its expressions consist of *attributes*, *constants*, *operators*, and *functions*.

```
node:type & PVNode;
child:count([]type & PVNode) > 7
```

Figure 3.1: GTQL-query

The query in Figure 3.1 has: the attribute *type*, the constant *PVNode*, the number 7, both hierarchical and relational operators, and a count function. It has a node-expression and a child-expression but omits the subtree-expression. It is an example of a query that is e.g. useful to developers of chess-playing programs. It collects all nodes that have more than seven children marked with the flag *PVNode*. This means that the search is repeatedly re-searching nodes and changing the principal variation, which indicates a bad move ordering.

A GTQL-query is used to narrow down the set of all nodes in a game-tree log file to the subset of nodes that fulfill all aspects of a query. A query is performed on a game-tree file. The corresponding game tree is traversed in a left-to-right depth-first manner and the node-expression part of the query is evaluated for each node in a pre-order fashion (i.e. on the way down). If the node-expression evaluates to true, then the child and subtree parts of the query are evaluated as well (we use a one-pass algorithm for this as described in the next chapter). A node fulfills the query if all expression parts evaluate to true for the node. The nodes that fulfill the query are collected or counted and this collection or count is returned as the result of the query.

3.1.1 Attributes and Constants

Attributes refer to the attribute values of the nodes stored in the file being queried. For each node several attributes are stored, two of which are always present ($node\_id$ and $last\_move$) while others are optional. The optional attributes are typically algorithm and domain dependent and may contain whatever information the users decide to log in their game-playing programs (e.g. information about the search window passed to a node, the value returned, the type of the node, etc.). In the query in Figure 3.1 the attribute is *type*. The attribute names must follow a naming convention where a name starts with a letter and is then optionally followed by a series of characters consisting of letters, digits, and the underscore (_) character. Also, an attribute name may not be the same as a reserved keyword in the language.

Constants are either numeric integral types (i.e. integer numbers) or user-defined names that refer to numeric integral types. In Figure 3.1 the user-defined constant *PVNode* is

Table 3.1: Operators listed by precedence

Operator	Type	Arity
[ ], [<]	Hierarchical	unary
&	Attribute	binary
<,>, >=, <=, =, ! =	Relational	binary
not	Logical	unary
and	Logical	binary
or	Logical	binary

used. The same naming convention is used for constant names as for attribute names. Information about attribute and constant names available in a query are stored in the game-tree file being queried. In the current version of the language, attribute values, like constants, can only be numeric integral types.

3.1.2 Operators

The language operators fall into four categories: *hierarchical*, *attribute*, *relational*, and *logical* operators. They are listed in Table 3.1 in a decreasing order of precedence. The evaluation of operators of equal precedence is left-to-right associative.

The *hierarchical* operators are used as prefixes to attribute names, and identify the hierarchical relationship of the referenced node in relation to the current node (the one being evaluated in the node expression). Currently there are two such operators defined, and they may be used only in child expressions. The first operator, [ ], stands for the child node of the current node that is being evaluated. For example, the child expression `count([ ]type=type)` counts the number of children that are of the same type as the current node (in child-expressions, attributes without a prefix refer to the current node). This operator is used in the query in Figure 3.1. The second operator, [<], stands for the previously evaluated child. The sub-expression [<]`type=[ ]type` thus asks about two consecutive child nodes of the same type.

The *attribute* operator "&" is essentially an inclusive bitwise *and*, and is used to extract flag bits out of attribute fields. For example, a single node may be flagged as being simultaneously a *pv-node* and a *quiescence-node*. Because we have two bits set at once we can not use the = operator to extract these flags and must therefore use the "&" operator instead to get the correct result. The *relational* operators test for equality or inequality of attributes, constants, function results, and numbers, and the *logical* operators allows one to form combined Boolean expressions. Parentheses can be used to control precedence and order of evaluation.

3.1.3 The Count Function

There is only one function in the language, the `count(sub-expression)` function, and it returns the number of nodes in the expression scope (i.e. tree, children, or subtree) that evaluate to true. Functions cannot be used recursively, that is, the expression inside *count* cannot contain a call to *count*. The wild-card character * may be used with the function instead of an expression to refer to the empty expression, which always evaluates to true. Note that because expressions must evaluate to either true or false, the count function must be used with a relational operator, e.g. `count(*)>0`. The example query in Figure 3.1 has a call to the *count* function from the child-expression and the count must be higher than seven for the expression to return true for a node.

The only exception to this is when the function is used in a node-expression, for example `node:count(type&PVNode)`. In that case, the function not only counts the nodes that fulfill its sub-expression but changes the result of the query from a collection to a count of nodes that fulfill the query. This enables the user of GTQT to format the result through the query, and gather statistics about our trees without taking up memory to hold information about every node by using the *count()* in the node-expression. This difference in result presentation is discussed in Chapter 4. It is not required that node-expressions use *count()* but child- and subtree-expressions on the other hand must contain the aggregate function *count()* to be meaningful. The word *count* is a reserved keyword in the language.

3.1.4 Expressions

Dividing the query into three separate expressions enables us to query nodes based on attributes of the current node, its children, and nodes in its subtree. That way we can make complex queries based on the structure of our entire search tree. We now go through the structure of each expression type and give examples of the expression syntax.

Node-Expression

The node-expression is the basic part of the query. It evaluates expressions based on the attribute values of the current node. When a query is evaluated, the node-expression, if not omitted, is the first query evaluated. If a node does not evaluate to true for the node-expression there is no point in continuing to evaluate the child-expression and/or subtree-expression for this particular node.

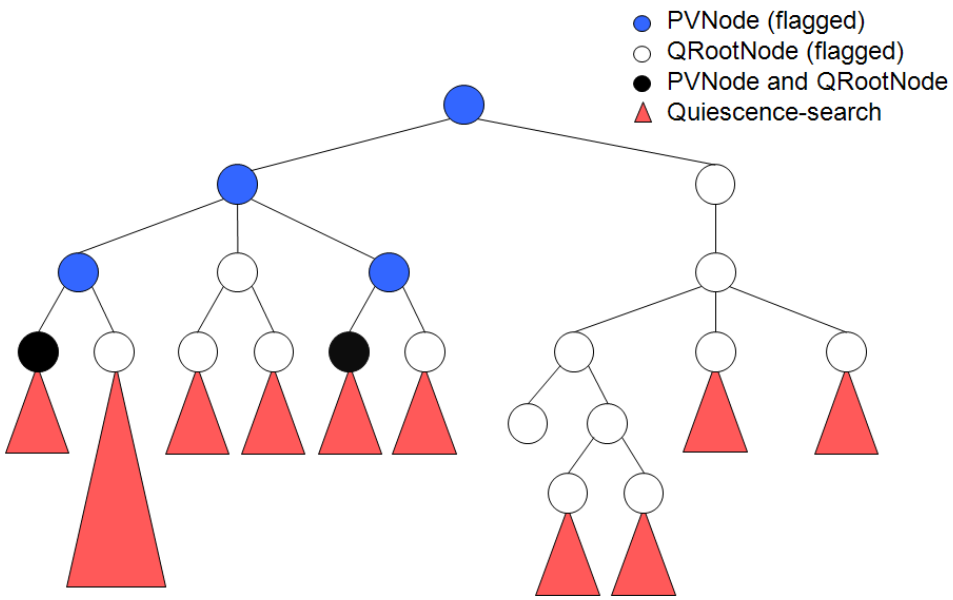


Figure 3.2: Game tree with quiescence searches

If the node-expression is a call to the aggregate function `count(type&PVNode)`, the total query returns the number of nodes that evaluate to true instead of the nodes themselves. The query `node:count(type&PVNode)` returns the number of *pv-nodes* in the whole search tree. It compares the value in the attribute *type* in each node to the constant *PVNode*. For the tree in Figure 3.2 it would return the value six.

If the node-expression is not aggregate it consists of terms comparing attributes, numbers or constants with other attributes, numbers or constants using relational operators. These terms can then be combined using the logical operators. Next are examples of legal node-expressions:

E_1: `*`
E_2: `alpha>beta`
E_3: `best>-4`
E_4: `type&PVNode`
E_5: `not type&PVNode`
E_6: `(type&PVNode or alpha>10) and beta<40`

Using the wild-card as an expression such as in E_1 makes the expression return true for all nodes. This gives the same result as omitting the node-expression. In E_2 we compare two different attributes of the node, *alpha* and *beta*, to each other. E_2 would be true only for nodes where the value of *alpha* is higher than the value of *beta* (which should never happen). E_3 is similar but here we compare the attribute *best* with a negative integral number. In E_4 we compare the attribute *type* to the constant *PVNode* with the bitwise flag

operator and get all nodes that have the bit *PVNode* set. E_5 shows how negation is used with an expression and E_6 is an example of a complex expression where parentheses are used to change the precedence of operators.

Subtree-Expression

A subtree-expression compares a `count(sub-expression)`, number, or constant with another `count(sub-expression)`, number, or constant. One or more subtree-expressions can be combined with a logical operator to form a new one. The sub-expression that the *count* function takes as a parameter is a node-expression and it is evaluated for each node in the current node's subtree. Below are examples of legal subtree-expressions. Note that multiple *count* functions are allowed, although not recursive.

SE_1: `count(*)>300`
SE_2: `count(type&AllNode)<count(best=5)`
SE_3: `count(type&AllNode)<20 and not count(*)>30`

Expression SE_1 counts the number of nodes that have subtrees with over 300 nodes. SE_2 is an example of a subtree-expression where we compare the result of two *count* functions. One of the functions checks whether the value of the *type* attribute has the constant *AllNode* bit set, while the other one checks whether the value of *best* is exactly 5. SE_3 is an expression where the results of two *count* functions are compared with numbers and then combined together with a logical operator and negation.

Child-Expression

A child-expression is the same as a subtree-expression except it allows hierarchical operators in the sub-expression. The unary hierarchical operators `[ ]` and `[<]` mark which node is to be queried for which attribute. If there is no hierarchical operator in front of an attribute, it refers to the parent. Here are three examples of child-expressions:

CE_1: `count([<]best>[ ]best)=2`
CE_2: `count([ ]type=QRootNode)>0 and count([<]type=2)>1`
CE_3: `count([ ]alpha>[<]beta or [ ]type=type)>0`

Expression CE_1 is a child-expression where we compare the *best* attribute of the previous sibling to the same attribute of the current child node. In CE_2 the counts are compared with numbers and combined with a logical operator. CE_2 is not true unless both of the combined expressions are true. The last child-expression CE_3 shows how we count a

combined sub-expression and refer to two consecutive child nodes and parent in one sub-expression.

Queries

A GTQL query is made from one or more of the expressions described above. The simplest queries have only one expression part and omit the others, but more complex queries have two out of three or all three expressions. Each query part (node, child, subtree) can only occur once in a query, and then in the above order.

Q_1: `node:type=PVNode;`
`child:count([ ]alpha>[<]beta or [ ]type=type)>0`
Q_2: `node:type&CutNode;`
`subtree:count(type&AllNode)>100`
Q_3: `child:count([ ]alpha<3)>2;`
`subtree:count(type&CutNode)>50`
Q_4: `node:type&CutNode;`
`child:count([ ]type>[<]type)>0;`
`subtree:count(best>4)>40`

Query Q_1 shows a query with a node-expression and a child-expression. Queries Q_2 - Q_4 show the other possible expression combinations.

3.2 Practical Example Queries

In the examples below we assume that for each node the search is logging information about its ply-depth in the game tree (*depth*) and flags indicating the node type (*type*). A node can be simultaneously flagged as being of several types, e.g. a *pv-node* and a *quiescence-node*.

Query 1

In this first example query we want to find whether the game-playing program is extending the search too aggressively. We form the query by asking for nodes where the *depth* attribute has a value greater than some high threshold value, excluding nodes in the quiescence search.

```
node:depth>=10 and not type&QNode
```

Query 2

As in the previous query, we are interested in identifying subtrees where too aggressive expansion occurs. However, now we want to identify places where the quiescence search is too aggressive. We form the query by asking for quiescence-root nodes having a subtree larger than 100 nodes.

```
node:type&QRootNode;
subtree:count(*)>100
```

Query 3

In this example, we want to identify nodes where the principal variation of the search changes frequently. Note that the node-expression part is not necessary for retrieving the answer, however, it is beneficial to include it as it constrains the search space of the query such that the child-expression is evaluated only at *pv-nodes*.

```
node:type&PVNode;
child:count([ ]type&PVNode)>2
```

3.3 Expressiveness of GTQL

The language can express a wide variety of different queries. Combining the three types of expressions, node, child and subtree, into one query gives us a chance to flag nodes that are of specific interest and relevance to the search algorithm we are using. Another thing worth mentioning is that all attributes to be queried with the language have to be integral numbers or user defined constants that map to integral numbers. This entails that we cannot have an attributes that takes a value such as: 2.345, true, or ”some string”. Expressions such as `type=true` or `depth=2.5` are therefore not valid. The implementation of the log library is what caused this.

The expressiveness of the language is limited because all expressions are evaluated in one depth-first traversal through the game tree. The algorithm doing this will be explained in detail in the next section. The language does not support nested calls to the *count* function in any part of the query. This means that we cannot query on attribute values of the current node’s grandchildren or subtrees and children of nodes in the current node’s subtree.

Additional functions would enhance the expressiveness of the language. The most important functions to add would be *min()* and *max()*. These functions would find the highest and lowest value of a certain attribute in the scope of the expression that contained them. An example of a query containing *max()* would be: `node:type&QRootNode; subtree:max(best)>10`. In this case the query would return all nodes that are quiescence root nodes and where the maximum value of *best* in all nodes in the subtree is higher than ten. We could also imagine a query using *min()* such as: `node:min(alpha)`, which returns the smallest value of the *alpha* attribute found in the tree. Adding functions such as these would not compromise the one-pass quality of the algorithm.

Also, we do not support queries on the parent of the current node and thus cannot query nodes on a certain path to the root of the tree. In addition the language does not support comparison or finding the difference between two game trees. It could be useful to do this when the search algorithm uses iterative deepening. The logging mechanism currently logs each search separately thus not allowing comparison.

Despite these limitations, the language can still express a vast amount of interesting queries about the structure of the game trees which search algorithms generate.

3.4 Related Work - Query Languages

To the best of our knowledge GTQL is the first language specifically designed for querying game trees. However, several query languages for tree structures exists.

XPath (Clark & DeRose, 1999) is a small query language that queries XML data by giving a path of a node and returning either the nodes found at the end of the path or objects from these nodes. XPath is primarily used for pattern matching and addressing parts of XML documents. Its similarity to GTQL is mainly that it can return an unstructured collection of nodes that fulfill a query. XPath can navigate an XML document horizontally and vertically and reference siblings, children, parents, descendants, and ancestors of the context node. The navigational abilities of XPath have been used in other languages either by directly supporting XPath like XQuery (Chamberlin, 2002) does or extending its syntax like is done in LPath (Bird, Chen, Davidson, Lee, & Zheng, 2005). XQuery includes XPath for navigation but is more expressive. It was designed for making human readable documents from XML documents and for complex data extraction and manipulation. XQuery is frequently used in transforming and integrating data from different sources, but because of its complexity and expressiveness using it for simple tasks sometimes seems like overkill.

The complexity of XQuery was one of the motivations for the design of XSQuirrel (Sahuguet & Alexe, 2005), a language for making sub-documents out of existing XML documents. Sub-documents are useful for many things, like distributed query processing and view creation, and XSQuirrel was created specifically to facilitate low overhead sub-document creation. It finds nodes based on a path query and reconstructs the tree from a specific node by adding its ancestors up to the root and all the node's descendants, thus keeping the structure of the tree intact even though some subtrees in the document have been removed. XSQuirrel queries can be translated into XPath or XQuery queries and thus a new evaluation method is not needed. XPath has been extended in other directions. It was used as a basis for the LPath language that is used for querying linguistic data. LPath adds subtree scoping, (i.e. restricting the query to the subtree of a specified node) and the concept of horizontal immediate precedence to other already defined methods of navigation.

The Chess Query Language (Costeff, 2004) is different from the languages discussed above. It is used for querying complex themes and positions in chess games and is very useful to composers and judges of study competitions. It can express a variety of different queries that describe chess positions but this unfortunately does not help the game-playing programmer when debugging a search algorithm.

Although the above languages all provide a query mechanism for referencing children, parents, descendants, and ancestors, they do not allow aggregation. A query such as, `subtree:count(type&ResearchNode)>50` could therefore not be formed and answered. Also, they are primarily designed for use on relatively small and shallow trees, and consequently can afford complex expressions. GTQL, however is designed for use on large trees, and the query expressiveness is designed such that the queries can be evaluated in a one-pass left-to-right tree traversal.

3.5 Summary

We have now described the syntax and semantics of GTQL and showed several examples of how queries are constructed and interpreted. We also discussed the limitation of the language and some ideas for enhancing its expressiveness in light of what is needed to express queries relevant to game playing. Next we present the implementation of the language in the Game-Tree Query Tool and how it uses a one-pass algorithm to collect all the information needed to answer a query in a single pass through the game tree.

Chapter 4

Game-Tree Query Tool

The *Game-Tree Query Tool (GTQT)* is a software for parsing and executing GTQL queries. It is an integral part of a larger collection of tools, called *Game-Tree Tools (GT-Tools)*, intended for aiding researchers in the analysis and visualization of large game trees. We start by giving a brief description of this suite of tools. Next we describe the design and implementation of the GTQT software in detail. We describe how queries are parsed and parse trees generated. We then introduce the one-pass query execution algorithm, designed to efficiently answer queries on large game trees, in an incremental fashion. First discussing the evaluation of a node-expression, then adding the evaluation of a subtree-expression and finally adding the evaluation of the child-expression that completes the algorithm. This designed makes GTQT capable of answering any query, no matter how complex it is, in a single traversal of the game tree.

4.1 Game-Tree Tools

The GT-Tools collection consists of a library for logging game-tree information, the GTQT program for processing GTQL queries, and a *Game-Tree Viewer* (GTV) for graphically viewing game-tree log files and query results. Game-playing program developers can enable logging of the search trees their programs generate by augmenting their programs with a handful of function calls to the log library (using a provided API showed in an appendix). The logging mechanism can be switched on or off with either a run-time or a compile-time flag. The log files generated this way can then be read either by GTQT for analysis, or the GTV tool for visualization. Figure 4.1 is an overview diagram of the tools and how they interact. The contribution of this thesis to GT-Tools is the GTQT software

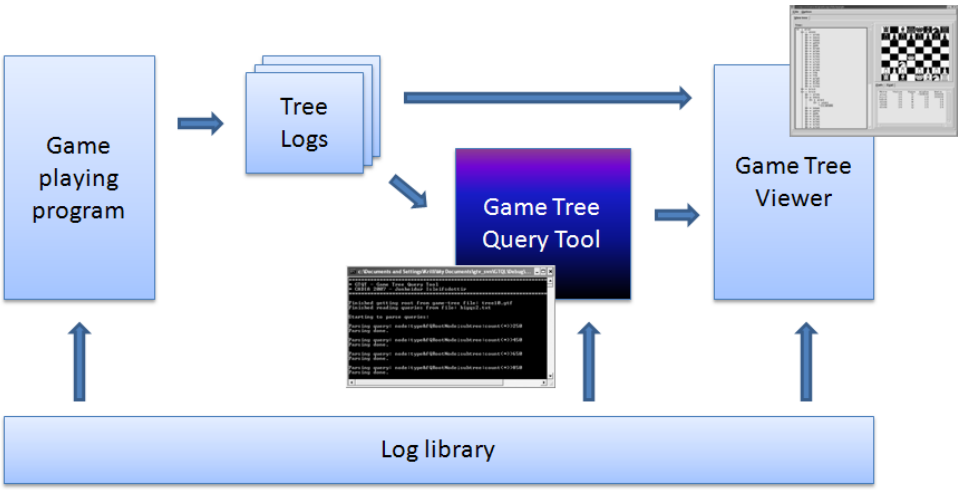


Figure 4.1: Collection of game-tree tools

that adds querying of the game-tree log files to the collection, and is shown as the dark component in Figure 4.1.

4.2 Using GTQT

The GTQT program is a console application that is run from a command line. It is implemented in C++ and runs on both Linux and Windows (as well as other platforms that support ANSI compliant C++ compilers). It links to the log library and runs as a stand alone process. The program takes two arguments: the name of a text file containing the queries to be executed, and the name of a game-tree log file. For example, in the command; `GTQL queries.txt tree10.gtf` the argument `queries.txt` is the name of a query file and the second argument `tree10.gtf` the name of a game-tree log file.

In addition to the game tree, the game-tree log file stores various meta-data in its header, such as the GT-Tool version used to generate the log file, a table of aliases (to allow symbolic names to be used for constants in queries, e.g. *PVNode* and *QNode*), and a description of the layout of the node data records. After processing the meta-data and validating it, the queries are parsed and their syntax checked. If invalid queries are encountered the program will report an error and terminate, otherwise it executes the queries on the supplied game-tree file, one query at a time. The result is written out to a special result file. An example result file is given in an appendix.

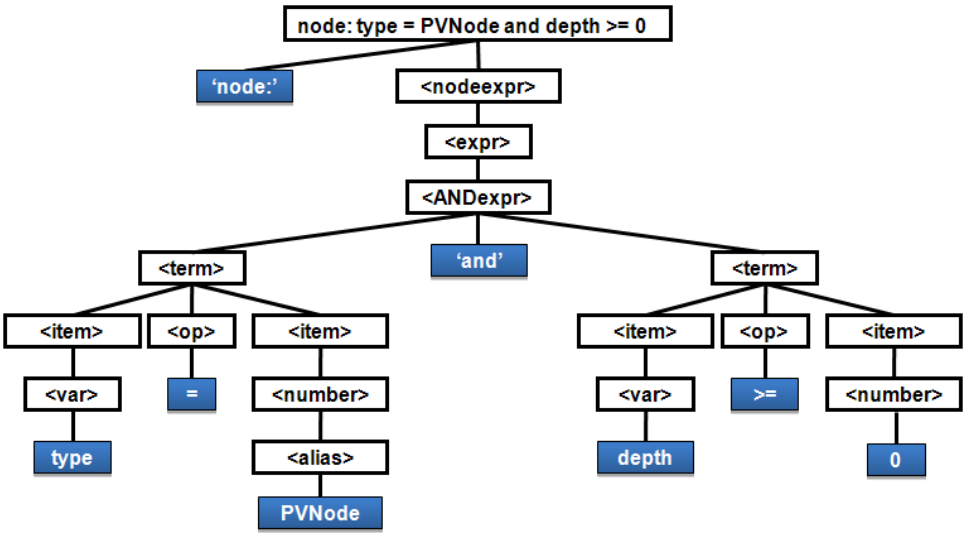


Figure 4.2: An example parse tree

4.3 Parsing a Query

Queries are parsed using a recursive-decent parser. A separate parse tree is built for each query. An example parse tree is shown in Figure 4.2, along with the query it represents. A parse tree consists of several different types of parse nodes, depending on the type of operator (e.g. relational or logical), term, or expression being evaluated. Most types of parse nodes return a Boolean value when evaluated, representing whether the corresponding expression evaluated to true or false for a particular node in the game tree. Typically the result of an evaluation on a game-tree node depends on the attribute values stored in the game-tree node. For example, in the above example the attribute values of both the *type* and *depth* fields are required for evaluating the query, and for nodes where *type* is equal to *PVNode* and *depth* is greater or equal to zero the query evaluates to true, but to false for all other nodes.

A special provision must be taken for queries containing the aggregate function *count*, as it returns a counter based on data accumulated over many records. Such queries can-

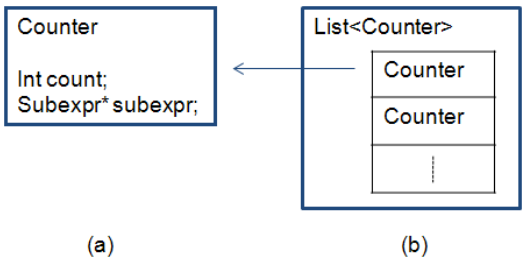


Figure 4.3: The counter structure and a list of counters

not be evaluated until after all data nodes in the expression scope have been traversed. In that case, in addition to the attribute values, a special structure containing count information accumulated over the scope (e.g. a subtree) of the query must be provided. This structure is called a *counter* and is shown in Figure 4.3(a). It stores a pointer to the count sub-expression (more specifically, to the parse-node representing the root of the sub-expression found within the counter function), and a counter variable initialized to zero. The pointer allows us to evaluate the *count* sub-expressions for all nodes in the corresponding scope (e.g. all nodes in a subtree or all children of a node), and the counter variable accumulates information about how many of them evaluated to true.

Node-expressions can only contain one count function, whereas both subtree- and child-expression can contain many such functions; for such expressions a list of counter structures is stored as shown in Figure 4.3(b). There are separate lists for child and subtree counters, as they are used differently in the evaluation. Note that the counter lists and counter structures are not stored as a part of the parse tree because our one-pass query execution algorithm may have to execute several counter based *query instances* concurrently on the same parse tree when evaluating subtree- and child-expressions (see later); for each query instance separate counters are needed.

4.4 Executing a Query

Time is of essence when evaluating large game trees. The query execution algorithm thus makes only one *depth-first traversal (DFT)* of the game tree, during which it collects all the information needed to answer the query. We introduce the one-pass DFT algorithm in steps. By one-pass we mean that we only read each node in the tree once from the log for each query. Since the one-pass traversal becomes complicated when the query has all three kinds of expressions, we start by describing the evaluation of the node-expression. We then add the subtree-expression evaluation with query instances and last we add the child-expression evaluation. The last algorithm is the one used in GTQT.

4.4.1 Evaluating a Node-Expression

The evaluation procedure shown in Algorithm 4 takes a game-tree node as a parameter. The first call to the algorithm is with the root of the game-tree being evaluated. The node-expression (*nodeExpr*) is a part of the query, which is global, so we can refer to it directly from within the algorithm. The same holds for the other expressions we refer to

Algorithm 4 DFT-SIMPLE(node)

```
1: if nodeExpr.evaluate(node) then
2:     addToResult(node)
3: children = node.getChildren()
4: for all child in children do
5:     DFT-SIMPLE(child)
```

later in this chapter. The algorithm then feeds the node into the parse tree for evaluation (*nodeExpr.evaluate(node)*) and if the evaluation returns true the node is added to *result*. If the node-expression has a call to the *count* function a counter is updated, otherwise the node identification is added to the list of result nodes. The children of the node are then read from the file and traversal of the tree continues in a depth-first left-to-right recursive manner. If a node is a leaf the algorithm evaluates it and then backtracks.

4.4.2 Evaluating a Subtree-Expression

Since a subtree-expression must contain the aggregate *count* function it cannot be evaluated until the depth-first traversal algorithm backtracks and the sub-expression information from a node's subtree has been collected. This is done with the *query instance*. It is a small data structure that holds two lists of counters: One for counters made from subtree sub-expressions and the other for child sub-expressions. The structure of the query instance is shown in Figure 4.4. Whenever a node evaluates to true for a node-expression a new query instance is made for that node. The instances are pushed onto a stack and then updated according to information from the subtree and children of that node. We must collect statistics individually for each node because they have different subtree scopes. One node's subtree scope may include all or a part of other nodes' subtree scope, but it is never exactly the same. This is depicted in Figure 4.5. The figure shows that when three query instances are added to the stack there are three different subtree scopes concurrently active.

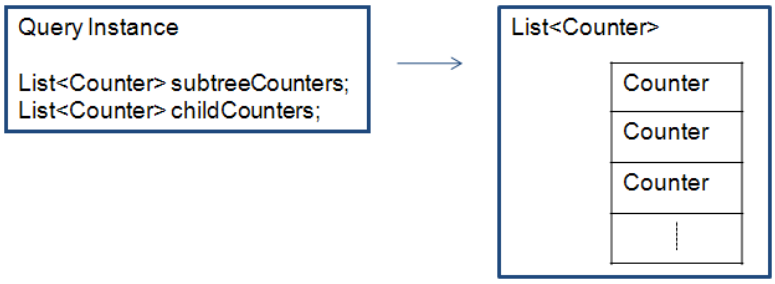


Figure 4.4: The structure of a query instance

Algorithm 5 DFT-SUBTREE(node)

```
 1: queryInst = null
 2: if nodeExpr.evaluate(node) then
 3:     queryInst = new QueryInstance(subtreeExpr)
 4:     queryInstStack.push(queryInst)
 5: children = node.getChildren()
 6: for all child in children do
 7:     DFT-SUBTREE(child)
 8: if not queryInstStack.empty() then
 9:     if queryInst == queryInstStack.top() then
10:         if subtreeExpr.evaluate(queryInst) then
11:             addToResult(node)
12:         queryInstStack.pop()
13:         delete queryInst
14: evalCounterExprs(queryInstStack, node)
```

Algorithm 5 expands Algorithm 4 to also handle subtree-expressions. Now, instead of adding the node to result when the node-expression evaluates to true, a new query instance is created, and pushed onto the stack (line 4). The algorithm then continues to traverse the tree until it reaches a leaf and then backtracks. On the way up the subtree-expression must be evaluated based on the information that has been accumulated in the query instance. If there is no instance on the stack or it does not match the node there is no need to evaluate because if the node-expression is false the query will never be true. If the current node is on top of the stack the subtree-expression is evaluated using the information from the query instance instead of the node (line 10).

For example if we were evaluating the subtree-expression `count(type=PVNode)>5` the evaluation of the count function works like this: Find the pointer to the expression `type=PVNode` in the subtree-counters list in the query instance and return the value in the count variable. This return value is then compared to 5 with the relational operator > in the evaluation of the subtree-expression. The outcome of this evaluation is then either

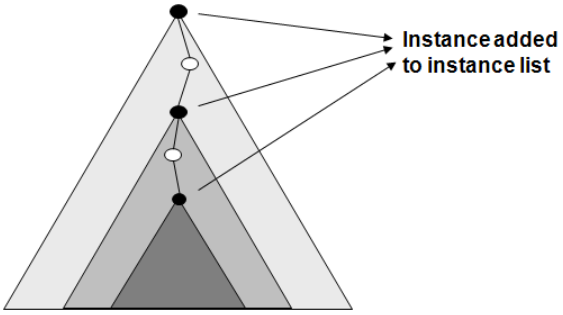


Figure 4.5: Subtree scope of different nodes in the same line

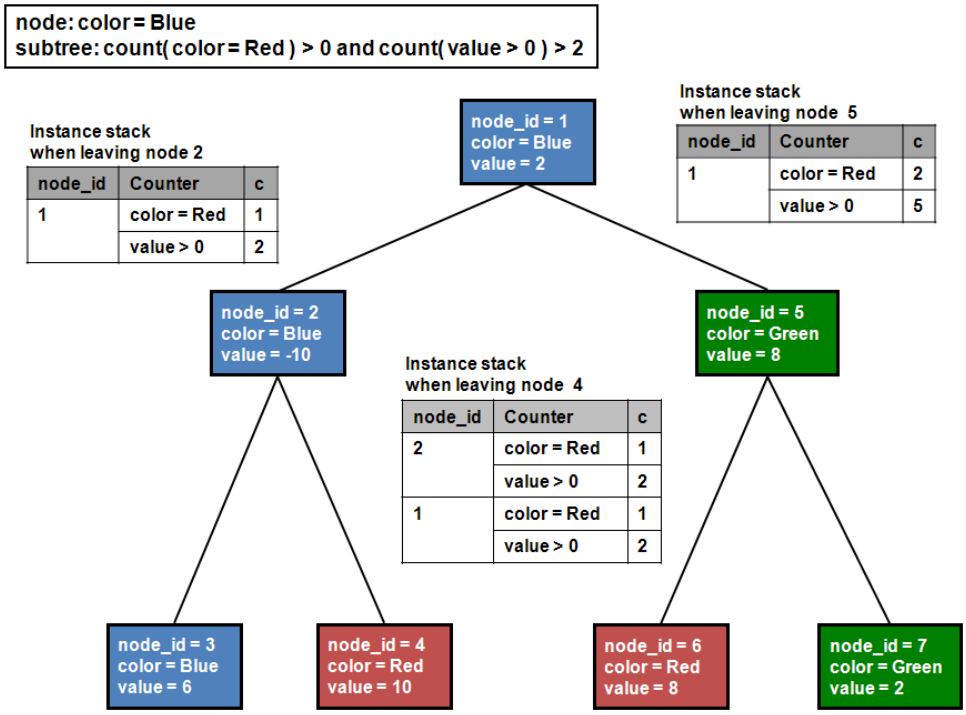


Figure 4.6: Traversing the data tree using query instances

true or false and decides whether the node is added to result or not (line 11). The query instance has now served its purpose, and is popped off the stack and deleted (lines 12 and 13).

After the evaluation we then need to update the counters of all the other query instances on the stack based on evaluation of the current node (line 14). The function in the last line of Algorithm 5 works like this: Each sub-expression from the subtree-counters list is evaluated for the current node, and if it evaluates to true we traverse the instance stack, and for every instance increase the count variable in the corresponding counter by one. This must be done separately for each query instance because they all have different subtree scopes.

An example of a tree traversal and parse-tree evaluation is given in Figure 4.6. The node-expression part of the query looks for nodes with the color blue. In this example we refer to the nodes by their $node\_id$. The node with $node\_id$=`1` becomes Node_1. The root is blue so a new query instance is created on the stack. This instance contains two counters: one for the sub-expression `color=Red` and one for `value>0`. The counter stores a pointer to the parse tree of the count sub-expression, and a counter variable initialized to zero (the c field in the figure). The traversal continues down the left branch, and as the node-expression is also true for Node_2, an instance is created on the stack for that node as well. An instance is also added for Node_3. Now, because a leaf has been reached, the DFT

Algorithm 6 DFT-FINAL(node)

```
 1: queryInst = null
 2: if nodeExpr.evaluate(node) then
 3:    queryInst = new QueryInstance(subtreeExpr, childExpr)
 4:    queryInstStack.push(queryInst)
 5: children = node.getChildren()
 6: prev = null
 7: for all child in children do
 8:    DFT-FINAL(child)
 9:    evalCounterExprs(queryInst, node, child, prev)
10:    prev = child
11: if not queryInstStack.empty() then
12:    if queryInst == queryInstStack.top() then
13:       if subtreeExpr.evaluate(queryInst) and childExpr.evaluate(queryInst) then
14:          addToResult(node)
15:       queryInstStack.pop()
16:       delete queryInst
17: evalCounterExprs(queryInstStack, node)
```

algorithm evaluates the subtree-expression for Node_3 based on the instance (the evaluation is false in this case) and backtracks. However, before backtracking the remaining query instances on the stack are updated according to evaluation of Node_3 (the counter for `value>0` is increased by one for both instances). The instances for Node_1 and Node_2 have now been updated and the algorithm has backtracked to Node_2. From there it continues to traverse the children and explores Node_4. This process continues until the entire tree has been traversed. A snapshot of the query instance stack is shown in the figure at selected points (text above the stacks in the figure). The rightmost snapshot shows the stack when the algorithm backtracks back to Node_1 for the last time. We can see that the instance for Node_1 is the only one left on the stack and its counters have been updated several times. Node_1 is now evaluated based on the query instance, the subtree-expression is true, so the node is added to result.

4.4.3 Evaluating a Child-Expression

We now add the evaluation of a child-expression to Algorithm 5, resulting in Algorithm 6. All the required expression evaluations have now been added and Algorithm 6 is the final version of the *DFT-algorithm* as it is used in the game-tree query tool. The algorithm has the same structure but there are a few additions. Now we add both the counter list from the *subtreeExpr* and the *childExpr* to the query instance before pushing it onto the stack (line 3). When the algorithm backtracks from exploring a child, the *evalCounterExprs()*

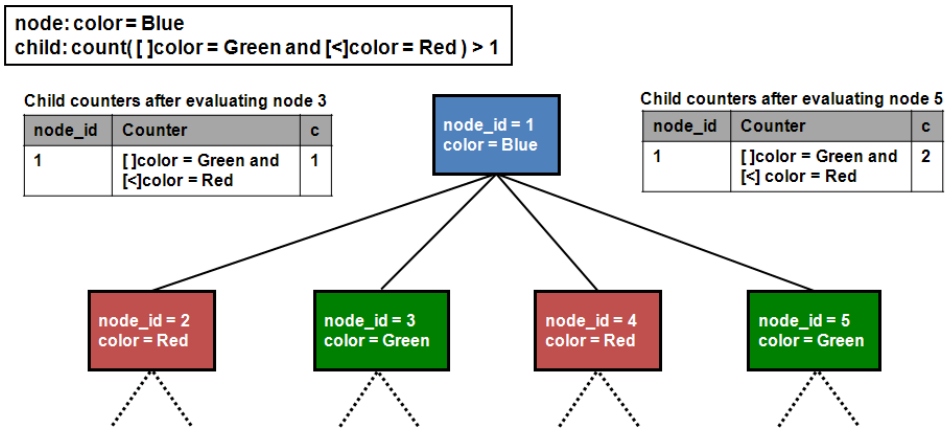


Figure 4.7: Traversing children using a query instance

function (line 9) is called and the query instance counters are updated according to the evaluations of the sibling queries using; *child*, *node*(as the parent to the child) and *prev* (previous sibling). It is because of this that we need to retain the previous sibling node in the variable *prev* (line 6). While we are traversing the children we keep updating *prev* (line 10). When the updating is finished (all children have been traversed) both the subtree- and child-expressions are evaluated and the node added to result based on that (line 14). When we send a query instance as a parameter to an evaluation function, the function knows which counter list to use in the evaluation of the expression.

Figure 4.7 is an example of a node with children. The child query is: `count([ ]color=Green and [<]color=Red)>1`. We evaluate it for the children of Node_1. After getting all the children of Node_1 we call Algorithm 6 recursively for each child. When the algorithm backtracks to the parent node it updates the child-counter list in the query instance by calling the function *evalCounterExpr(queryInst, node, child, prev)*. The function evaluates the sub-expression in the child-counters list (in this case only `[ ]color=Green` and `[<]color=Red`) for the child node. For the first child, Node_2, there is no previous child so *prev=null*. Since this child-expression involves a previous sibling the query returns false for Node_2. Node_2 is then made the previous node and the algorithm continues. When all the children have been evaluated like this and the counter in the query instance increased (by Node_3 and Node_5 in this case), the child-expression is evaluated for Node_1 using the resulting query instance list. In this case it evaluates to true because the counter is 2.

4.5 Algorithmic Complexity

It is important when designing algorithms to analyze the algorithmic complexity. We need to know what the normal and worst case space and time requirements for the algorithm are and whether they can cause trouble for the regular user of the program implementing the algorithm, in this case GTQT. Let us start by defining the necessary terminology:

- *n* = size of game tree (in number of nodes)
- *d* = actual depth of game tree
- *sc* = size of counter
- *sl* = size of counter lists

We now analyze the space and time complexity of the DFT-FINAL algorithm presented in subsection 4.4.3.

4.5.1 Space

The space complexity of the algorithm depends on how many query instances can be active on the stack simultaneously. The size of the tree is not an issue because the nodes are read from the file as the tree is traversed in a depth-first manner. The query instance stack is therefore the only memory needed. The algorithm adds at most the number of nodes that correspond to the depth of the game tree to the stack. Whenever it reaches a leaf it backtracks to the previous node, so the stack is never larger than d. In the worst case scenario our tree has branching factor one and then $d = n$. Each query instance contains lists of counters, varying for each query, the size of a counter is fixed (a pointer and an integer variable).

$$WCSC = n * sc * sl = O(n)$$

The equation above shows the worst case space complexity (WCSC). Both *sc* and *sl* would classify as constants when analyzing computational complexity, *sc* because it is fixed and *sl* because it is independent on number of nodes in tree.The space requirements are therefore in the worst case only dependent on the number of nodes in the game tree. Actual game trees normally have a branching factor much greater than one (in chess it is thirty six) and trees with a branching factor of at least two have a depth that is logarithmic in the

number of nodes in the tree ($\log n$). This means that the average case space complexity of DFT-FINAL is O($\log n$).

4.5.2 Time

The time complexity of the algorithm is dependent on the size of the game tree we are traversing as the tree must always be traversed once. The other factors that play a role in the time complexity is the updating of counters in the query instances on the stack. Each time we backtrack from a node we must evaluate it for each counter and if it is true update all counters on the stack. In the worst case where our tree has branching factor one and all sub-expression in the query return true for every node we have to update *n* query instances for the lead node $n - 1$ for its parent node and so on.

$$WCTC = (n + (n-1) + (n-2) + ...) * sl = O(n^2)$$

The equation for worst case time complexity (WCTC) of DFT-FINAL is shown above. Like we mentioned earlier *sl* is a constant so again the worst case complexity is only dependent on the number of nodes in the game tree, but in this case we have to update the query instance stack again and again so the worst case time complexity becomes $O(n^2)$. In the average case, the stack is approximately as deep as the depth of the tree ($\log n$) instead of *n*, but we still have to traverse the whole tree (*n*). This makes the average case time complexity of DFT-FINAL *O(n* $\log(n)$*)*.

4.6 Related Work - Tools

There have been other efforts toward making tools to visualize and analyze large search spaces. Rémi Coulom presents a visualization technique for search trees (Coulom, 2002) based on treemaps. Treemaps are based on the idea of taking a rectangle and dividing it into sub-rectangles for each subtree (Shneiderman, 1992). The first rectangle is split vertically, one rectangle per child. Those rectangles are then split horizontally for each of their children and so on. The problem with doing this when dealing with search trees, especially $\alpha\beta$ trees where every other layer has many nodes and the other only one, is that the map becomes full of very narrow lines that all lie in the same direction. Coulom presents a new layout algorithm that is similar to squarified treemaps. Squarified treemaps

split the rectangles in a way that enforces an aspect ratio close to one. The strip treemap algorithm goes even further and preserves the move ordering information in the map as well. The visualization tool can be used to make treemaps out of search trees and the user can zoom in on individual subtrees. As an improvement it is suggested that comparing two trees with a "diff" would be a nice addition to the program.

A tool like this certainly helps researchers to visualize large search trees and colors can be used to increase the descriptive power of maps like these. But even with increased visualization the problem with representing trees as figures is that they always need to be interpreted and can not deliver specific statistical information about the three. Figures also become increasingly harder to interpret when the information presented increases. A query language can therefore deliver more precise information about game trees.

Computer games is not the only field in computer science where researchers are dealing with huge state spaces. Jan Friso Groote and Frank van Ham (Groote & Ham, 2003) describe a method for visualizing huge state spaces in the field of modeling and verification of complex systems using state transition systems. When analyzing complex parallel processes these systems can have millions of nodes and it is very hard to visualize them using regular graph drawing techniques. The current methods used to gain insight into the structure of these state spaces are good for answering whether two different processes are doing the same thing but they can not answer questions such as; how many states are in each phase of the process?

Groote and van Ham claim their technique is highly scalable and computationally inexpensive and it is mostly limited by the capacity of the current graphics hardware. The reduction process assigns each node a non negative rank depending on its distance from the root taking edge direction into account. The edges going against the flow of the process are called backpointers and the user of the application can choose whether they are displayed or hidden. The nodes are also clustered together based on how similar they are. Each cluster is then displayed as a disk in three dimensional space where the diameter is in proportion to the number of nodes in the cluster. The clusters are then connected with cones and clusters with the same rank displayed in the same horizontal plane.

Processes containing over 1 million states can be clearly visualized and the structure analyzed using this method. The software also supports zooming into subsections of the state space for a closer look. This application is useful for many aspects of visualizing state spaces but it does by no means answer all questions related to analyzing such data.

Both methods described above help researchers to visualize huge state spaces and thereby gain an insight into the work they are doing. What they are not capable of is answer-

ing specific questions about the internal structure of these spaces like we are doing with GTQT, where we are taking an extra step toward easier debugging and regression testing of search algorithms.

4.7 Summary

We have now gone through the implementation of GTQL in the Game-Tree Query Tool. We briefly described the GT-Tools suite and how GTQT fits into it and then went through the parsing and the one-pass DFT-algorithm step by step. The use of GTQT, how it works, and the complexity of the one-pass algorithm were discussed. In Chapter 5 we evaluate empirically both the efficiency and the effectiveness of the language and the tool.

Chapter 5

Experiments

In this chapter we present the setup and results of empirical evaluation of GTQL and GTQT. In the first section testing of the efficiency of the implementation of GTQL and the one-pass algorithm in GTQT is tested by running a set of GTQL-queries on artificial game trees. In the second section we use GTQT to find anomalies in chess trees by running GTQL-queries on game trees generated by a chess engine.

5.1 Efficiency of GTQT

To test the efficiency of the implementation and the one-pass algorithm we ran the same set of queries on several artificially generated trees. The trees have different depths and branching factors. The nodes are identical apart from the best attribute that is randomly assigned a value between 0 and 9 and the type attribute is always set to 1. The *best* attribute is used when we needed to narrow our queries to return true for a specific ratio of the tree. The log library code prevented us from making tree files that were larger than 2 GB. The queries were run on a 3GHZ Linux-based computer with 2GB of main memory.

Table 5.1 shows the depth, branching factor and size (in number of nodes) of each tree. Tree_1 through Tree_5 all have varying depths, but their relative sizes are kept in the same ballpark. Tree_6 through Tree_{10} on the other hand all have the depth five but different sizes and branching factors. Tree_1 is the deepest tree and one of the larger ones. Tree_6 is the largest but it is relatively shallow and has a high branching factor. Dividing the trees into these two categories allows us to study the effects of size and depth individually.

Table 5.1: Game-tree logs used in efficiency testing

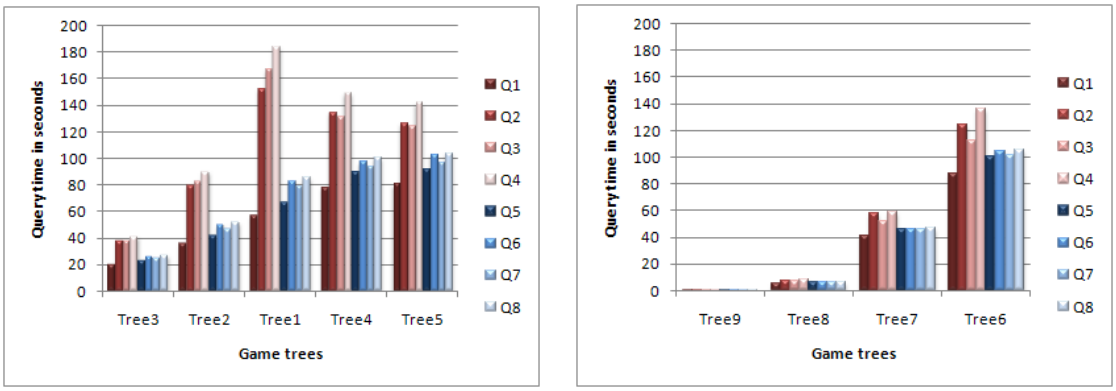
Tree	Depth	BF	Number of nodes
1	24	2	33,554,431
2	15	3	21,523,360
3	10	5	12,207,031
4	9	7	47,079,208
5	8	9	48,427,561
6	5	35	54,066,636
7	5	30	25,137,931
8	5	20	3,368,421
9	5	10	111,111

We ran eight queries on the trees to check the efficiency of GTQT. The queries are:

Q_1: `node:count(*)`
Q_2: `node:count(*);child:count(type=[ ]type)>0`
Q_3: `node:count(*);subtree:count(*)>300`
Q_4: `node:count(*);child:count(type=[ ]type)>0;`
`subtree:count(*)>300`
Q_5: `node:count(best=0)`
Q_6: `node:count(best=0);child:count(type=[ ]type)>0`
Q_7: `node:count(best=0);subtree:count(*)>300`
Q_8: `node:count(best=0);child:count(type=[ ]type)>0;`
`subtree:count(*)>300`

In queries Q_1 - Q_4 we use * in the node-expression but in queries Q_5 - Q_8 the expression `best=0`, otherwise they are identical. The effect of this difference is that for queries Q_1 - Q_4 a query instance is made for all nodes but in queries Q_5 - Q_8 query instances are only created for 10% of the nodes in the tree. The nodes that have `best = 0` are randomly distributed throughout the tree. Since all nodes have the same type, the child-expression used in the queries is true for all nodes that have one or more children. While testing the queries we also ran them with collecting instead of counting (this is done by omitting the count in the node query) and found that it had no notable effect on the query time.

We timed how long it took for GTQT to answer each query for each tree. The results are shown in Figure 5.1. The trees in the graphs in Figure 5.1 are ordered by size, because of the assumption that query time would increase linearly by size of the game tree. This assumption holds for all queries in Figure 5.1(b) where the depth of the trees is fixed. In Figure 5.1(a) on the other hand we notice that the columns for queries Q_1 and Q_5 - Q_8 grow with the size of the tree, but query times for queries Q_2 - Q_4 are much higher for



(a) $Tree_1$-$Tree_5$ (b) $Tree_6$-$Tree_9$

Figure 5.1: Query times for all trees and queries

$Tree_1$ than $Tree_4$, and $Tree_4$ also has higher times than $Tree_5$ even if $Tree_5$ is the largest. There must therefore be something other than size that is also affecting the time it takes to answer a query when the queries have child- and subtree-expressions.

Query Q_1 is our baseline query. It has essentially no evaluation because when the * is used GTQT returns immediately true for each node and the time it takes to process the query thus consists only of traversing the game tree by reading the nodes from the file. Figure 5.2 and Figure 5.3 show how the throughput (measured in nodes per second) drops below the baseline for more complicated queries. The baseline query processes around 600 thousand nodes per second. This is approximately the same throughput as the chess program that generated the tree. When evaluation of a node-expression is added, as in Q_5, the throughput decreases by 17% to around 500 thousund nodes per second in all trees regardless of depth and branching factor. This means that the cost of reading the tree from a file and traversing it without any evaluation is around 83% of the total cost of evaluating a query with only a node-expression.

If we go further and look at how many nodes are processed per second when a child- and/or subtree expression are added to the query, as in Q_2-Q_4 and Q_6-Q_8, we see in Figure 5.2 that the type of node-expression has an effect on how much the throughput changes. For queries Q_2-Q_4 the throughput decreases by 10-15% by adding child- and/or subtree-expression. This is mainly the added overhead of updating counters on the query instance stack. We do not see the same level of decrease for queries Q_6-Q_8, as they add 90% fewer instances on the stack than Q_2-Q_4. We can also see in Figure 5.3 that the different branching factors of the trees have no effect on the throughput of nodes within a query (the columns within each query are very even). The columns for $Tree_9$ are always a little shorter than the others but that can be explained by the fact that $Tree_9$ is by far the smallest tree consisting of only around 100 thousand nodes.

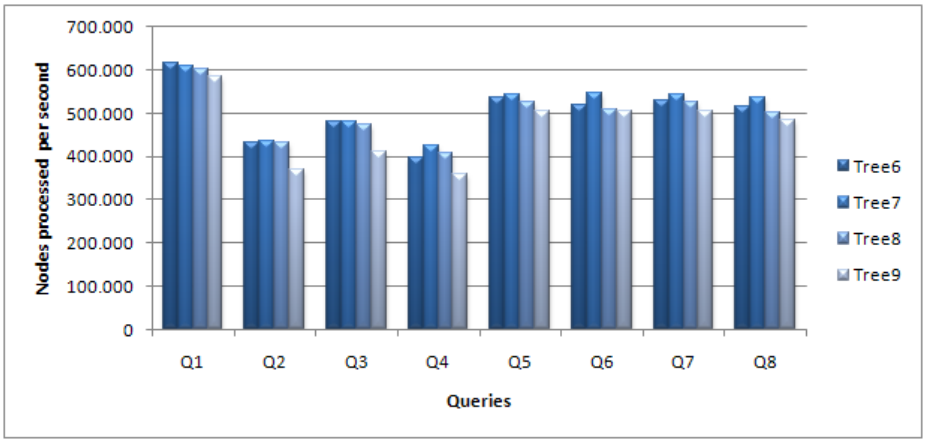


Figure 5.2: Number of nodes processed per second for all queries on Tree_6-Tree_9

In Figure 5.3 the evenness of the columns within queries is not present for queries Q_2-Q_4. In these queries the columns form a step ladder. The lowest step being the deepest tree and so on. This is caused by the same effect we saw as the spikes in query time for these queries in Tree_1-Tree_3 in Figure 5.1(a). The ladders can also be seen in queries Q_6-Q_8 in Figure 5.3 but there they are not as steep. This decrease in throughput is in direct relation with the depth of the tree. The decrease is over 60% for Tree_1 that is 24 plies and then gets less and less as the trees get shallower. This is caused by the same reason as before, the added overhead of updating the counters on the stack. In this case the stack is larger and updating takes more and more time with increasing depth. The effect of this is present but has less impact on the throughput in queries Q_6-Q_8 because they add only about 10% of the nodes in the tree on the query instance stack. Queries Q_5 - Q_8 are much more representative of normal use of GTQT than queries Q_1 - Q_4.

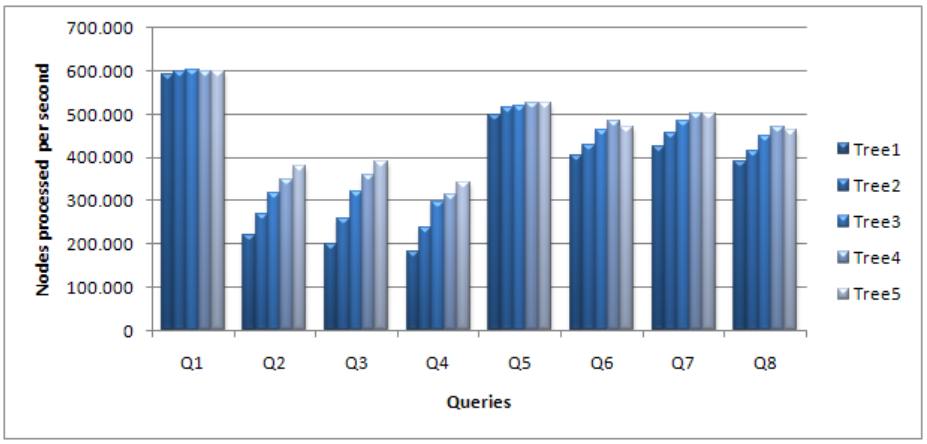


Figure 5.3: Number of nodes processed per second for all queries on Tree_1-Tree_5

The results presented here show that the one-pass algorithm is working well. In the more representative use case the query time grows close to linearly with respect to the size of the tree, even though we are counting and aggregating information about both the children and the subtrees of nodes. There is no exponential growth in the time it takes to answer a query. This is consistent of with the theoretical analysis that given $O(n \log(n))$ average time complexity. However the logarithmic factor is small and thus does not show clearly in our experimental data.

5.2 Experiments in Chess

To demonstrate the usefulness of the GTQT software, and to gain additional practical experiment with the tool, we used it to analyze game-tree logs generated by the chess program Fruit (Letouzey, 2005). In this section we report the anomalies found in the game trees. We do not attempt to analyze them further or look for a cause. That task is left to others.

5.2.1 Fruit Chess

Fruit Chess (Letouzey, 2005) is a chess engine developed by Fabien Letouzey. It was first released in March 2004, and subsequently made a strong appearance in the 2005 World Computer Chess Championship held in Reykjavík. For our experiments we used version 2.1 of the program, which is the strongest open-source chess engine available (unfortunately, subsequent versions of the program were not made open source). The only modification we made to the program was to augment its search engine with calls to the game-tree log library.

5.2.2 Game Trees

The query tool was tested on game-tree logs generated by Fruit Chess when searching nine tactical chess positions from the LCT II test suite (Louguet & Échiquéenne, 2007). This suite is one of several frequently used standard test suites to measure chess program performance. The suite consists of 35 test positions: 14 strategic, 12 tactical, and 9 endgame. We chose to use nine of the twelve tactical positions (numbers 15-23 in the suite) as they are the most relevant for evaluating search performance. On each of the chess problems the chess program was run until the correct solution, given in the test suite, was found. For

Table 5.2: Game-tree logs used for experiments

LCT II Position	SD	SSD	Number of Nodes
15	8	15	205,199
16	2	19	2,383
17	6	18	197,803
18	10	30	1,671,866
19	5	25	78,165
20	8	24	580,158
21	9	45	2,821,292
22	9	41	5,135,007
23	9	35	1,009,011

each position, a separate game-tree log was generated for each iterative-deepening search iteration. The solution (best move played) was found on iterations varying from the second to the tenth ply, as shown in Table 5.2. The first column indicates the position within the suite; the second column, SD, shows the search depth in the iteration when the best move was first returned; the third column, SSD, is the maximum search depth reached in that iteration; and the final column is the number of nodes searched in that iteration. In our experiments we used the game-tree log from the iteration where the solution was first found for each position.

From here on we will refer to the game trees by using their LCT II position number, e.g. the game tree searched from position 18 will be referred to as Pos_{18}.

5.2.3 Queries

The queries we executed on the game trees are listed in Table 5.3. They are constructed for providing useful information about the different types of nodes in the search tree, and for discovering possible pitfalls the search can fall into. For example: Is the structure of the tree as expected? Are we searching too deep or extending the search too aggressively? Are the quiescence searches too big? Is the principal variation changing frequently? The queries can be divided into four categories depending on which question they are trying to answer.

Queries Q_1 - Q_4 are targeted to check the frequency of different type of nodes in the tree. Query Q_1 counts all nodes, while queries Q_2 - Q_4 count nodes with specific attributes. The latter three queries should add up to the same number of nodes as the result from Q_1 since the constants PVNode (*pv-node*), CutNode (*cut-node*) and AllNode (*all-node*) are mutually exclusive, but all inclusive types.

Table 5.3: Queries used in experiments

ID	Query
Q_1	node:count(*)
Q_2	node:count(type & PVNode)
Q_3	node:count(type & CutNode)
Q_4	node:count(type & AllNode)
Q_5	node:type & QNode
Q_6	node:ply<=*x* and not type & QNode
Q_7	node:ply>*x* and not type & QNode
Q_8	node:ply>*x+5* and not type & QNode
Q_9	node:ply>*x+10* and not type & QNode
Q_{10}	node:ply>*x+20* and not type & QNode
Q_{11}	node:ply>*x+25* and not type & QNode
Q_{12}	node:ply>*x+30* and not type & QNode
Q_{13}	node:type & QRootNode subtree:count(*)>0
Q_{14}	node:type & QRootNode subtree:count(*)>50
Q_{15}	node:type & QRootNode subtree:count(*)>100
Q_{16}	node:type & QRootNode subtree:count(*)>150
Q_{17}	node:type & QRootNode subtree:count(*)>200
Q_{18}	node:type & QRootNode subtree:count(*)>250
Q_{19}	node:type & PVNode child:count([ ]type & PVNode)>2
Q_{20}	node:type & PVNode child:count([ ]type & PVNode)>5
Q_{21}	node:type & PVNode child:count([ ]type & PVNode)>7
Q_{22}	node:type & PVNode child:count([ ]type & PVNode)>8
Q_{23}	node:type & PVNode child:count([ ]type & PVNode)>9
Q_{24}	node:count(type & ResearchNode)

The second category, queries Q_5 - Q_{12}, firstly counts how many quiescence nodes there are in the tree, that is, number of nodes in all of the quiescence searches(query Q_5). The next two queries count the nodes that are not quiescence nodes and lie beneath (query Q_6) and above (query Q_7) a certain search depth. Since the trees generated are of different depths we counted nodes that were not quiescence nodes on regular intervals below the nominal search depth. These queries gather all nodes below a certain depth that are not flagged as quiescence search nodes. The *x* represents the nominal search depth of that particular tree.

Queries, Q_{13} through Q_{18}, are used to find large quiescence-search trees. A quiescence search should only search selected moves and only until a quiescent position is reached. They are frequent and should therefore be as small as possible. It is therefore interesting to find cases where the searches "run wild" and become excessively large. We used a few queries to get closer to the outliers. These queries gather all nodes that are quiescence search roots and have subtrees that have more than a specific number of nodes.

Table 5.4: Ratio of node types in the game trees

Tree	Num. of Nodes	pv-nodes	cut-nodes	all-nodes
Pos_{15}	205,199	0.44%	69.71%	29.86%
Pos_{16}	2,382	14.60%	63.58%	21.82%
Pos_{17}	197,803	0.03%	74.25%	25.73%
Pos_{18}	1,671,866	0.06%	75.87%	24.07%
Pos_{19}	78,165	0.52%	73.32%	26.16%
Pos_{20}	580,158	0.08%	73.19%	26.72%
Pos_{21}	2,821,292	0.10%	72.10%	27.79%
Pos_{22}	5,135,007	0.12%	76.93%	22.95%
Pos_{23}	1,009,011	0.15%	67.52%	32.33%

The fourth category of queries, Q_{19} through Q_{23}, was used to check for changing principal variations and also to see how large a portion of the nodes had to be re-searched after being searched with the null window. These queries gather all *pv-nodes* that have more than a specified number of children that are also *pv-nodes*. Query Q_{24} counts the number of nodes that have the re-search flag set which means that they were originally searched with a null window and had to be re-searched with a larger window because the null-window test failed.

5.3 Results

In this section we report our findings from running each of the above mentioned categories of queries on the game trees.

5.3.1 Ratio of node types in the game trees

Table 5.4 shows the result of running queries Q_1, Q_2, Q_3 and Q_4 on the game trees. The first column is the total number of nodes in the tree (returned by Q_1) and the next three columns show the ratio of each node-type in the tree (query Q_2 - Q_4).

In Table 5.4 we see that the ratios between types are relatively consistent. The only exception to this is Pos_{16}. However the search from that position is small, so deviations like the one present are not necessarily abnormal. The *pv-nodes* are only a small part of the whole tree, because in the ideal case there should be only one pv-node in each ply of the tree. In practice there are more *pv-nodes* in the tree because of re-searches but if the search algorithm has good move ordering this should not occur frequently. The majority of the nodes in the tree are *cut-nodes*. The reason behind this lies in the structure of

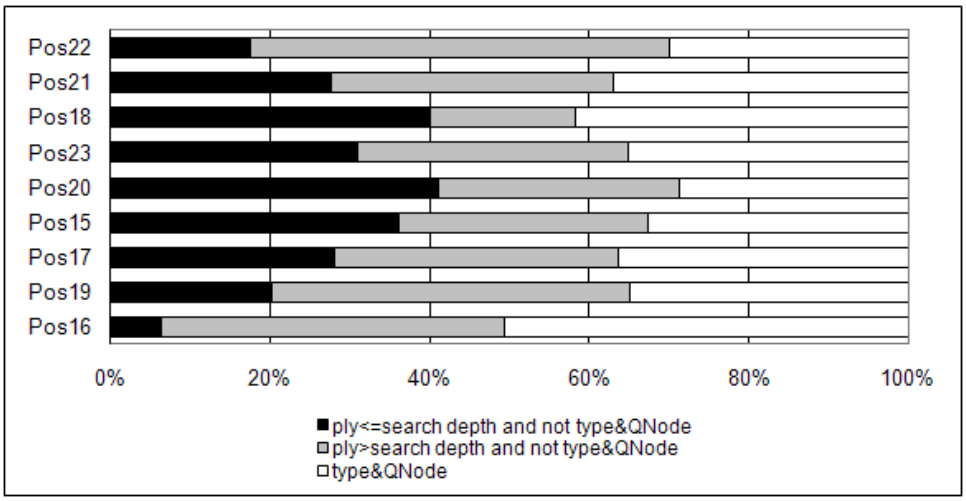


Figure 5.4: Ratio of regular search extension nodes compared to nodes above search depth and quiescence-nodes

the minimal tree (Figure 2.1), each *cut-node* has one *all-node* child that has all *cut-node* children. This structure produces 2-3 times more *cut-nodes* than *all-nodes* in the tree, so the result looks normal.

5.3.2 Deep Lines

In queries Q_8 - Q_{12} we look for non-quiescence search nodes that are well above the nominal search depth. The results in Table 5.5 show that several of the trees have lines that extend as deep as +10 plies above the nominal search depth (query Q_{10}). Fewer trees have lines that extend deeper but Pos_{21} has extensions as deep as +30 (query Q_{12}). That is aggressive extending of the search tree.

It is not only interesting to see how deep the search extensions go but also how big a part of the tree they are. If we sum up the results of queries Q_5 - Q_7 we get the total

Table 5.5: Number of nodes at different depths in the game trees

Query	Pos_{15}	Pos_{16}	Pos_{17}	Pos_{18}	Pos_{19}	Pos_{20}	Pos_{21}	Pos_{22}	Pos_{23}
Q_5	66,602	1,205	71,610	694,582	27,267	166,095	1,040,497	1,532,847	352,744
Q_6	74,282	153	55,841	673,702	15,987	239,674	786,366	908,701	314,201
Q_7	64,315	1,025	70,352	303,582	34,911	174,389	994,429	2,693,459	340,666
Q_8	12,783	369	412	15,931	14,129	2,143	100,186	564,116	76,098
Q_9	691	172	0	445	547	126	13,619	81,124	7,275
Q_{10}	0	0	0	0	0	0	3,577	3,564	48
Q_{11}	0	0	0	0	0	0	653	276	0
Q_{12}	0	0	0	0	0	0	6	0	0

number of nodes in the tree. These numbers are used to calculate the ratio of different types of nodes compared to the total number of nodes in the tree. The result of these calculations is shown in Figure 5.4. About 30% of the total number of nodes in the search trees are quiescence search nodes (QS-nodes) and this ratio is consistent in all trees. Of the other 70% of the nodes, usually over half are due to other kinds of search extensions, so only around 30% - 40% of the nodes in the tree are nodes we would search if we were not using extensions. There are two trees that diverge from the others, Pos_{18} and Pos_{22}. Pos_{18} is a rather large search tree and a 40% ratio of nodes above the original search tree. Only about 18% of the nodes are extensions. In Pos_{22} on the other hand the nodes above original search depth are only about 18% of the total number of nodes and extensions account for about 45% of the nodes. In Pos_{22} the algorithm is obviously extending the search very aggressively.

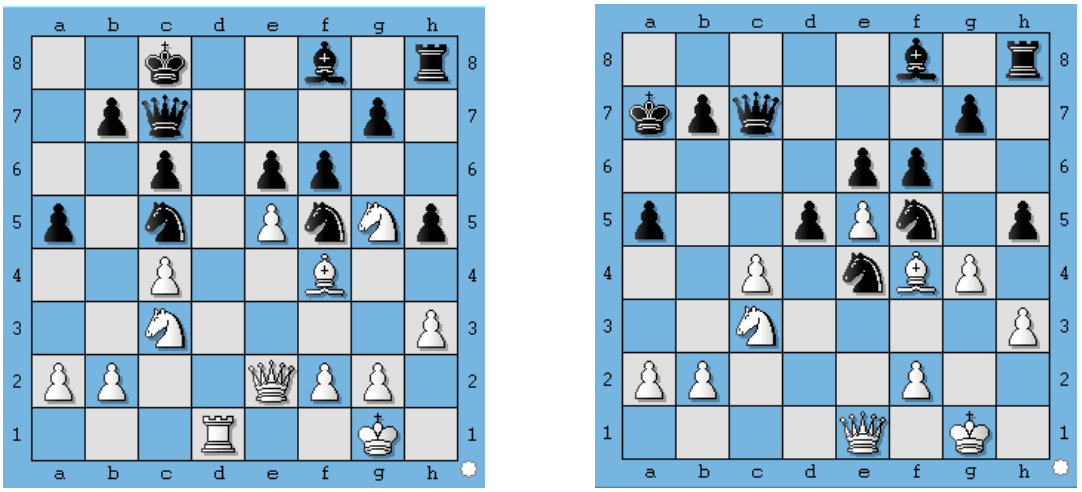
5.3.3 Large Quiescence Trees

Queries, Q_{13} - Q_{18} were used to query for very large quiescence-search trees (QS-trees) in the game-trees. For some trees we had to add more queries with a higher limit to find the biggest trees. The limit was raised to 850 for Pos_{18}.

Table 5.6 shows the size range of the largest QS-trees in each of the test position searches. For example, the four largest quiescence search trees of Pos_{15} have between a 100 and 150 nodes. The result shows that trees from several positions have QS-trees that are larger than 100 nodes. This could be of a concern as QS-trees should be kept as small as possible. In particular, the game tree that was searched from Pos_{18} had several large QS-trees, the largest having 840 nodes, quite excessive compared to typical QS-trees. Pos_{18} (a position from the game Vanka - Jansa, Prag 1957) is shown in Figure 5.5(a). The large quiescence search occurs after the move sequence: g5e4, c8b8, e2e1, b8a7, g2g4, c5e4, d1d5, c6d5 (not a particularly relevant sequence as most lines explored by the search, but nonetheless using valuable CPU resources), resulting in the position shown in Figure 5.5(b).

Table 5.6: Number of nodes in large quiescence trees

Trees	Number of nodes in 4 largest QS-trees
Pos_{16}, Pos_{17}, Pos_{19}, Pos_{20}	$50 < n < 100$
Pos_{15}, Pos_{23}	$101 < n < 150$
Pos_{21}, Pos_{22}	$151 < n < 200$
Pos_{18}	$201 < n < 850$



(a) Pos_{18} (b) Position with large quiescence search

Figure 5.5: Chess positions

This is good example of how the tool can be used to improve the search performance: now a program developer could look more deeply into what is going on with the quiescence search from this position.

5.3.4 Principal Variation Changes

We also checked the trees for changes in the Principal variation with Q_{19} - Q_{23}. The idea behind the PVS algorithm is to expand the best move first, refuting other lines with the least effort and thereby only expand a minimal tree. The algorithm therefore always starts by searching other nodes than the *pv-node* with a null-window search. If the null-window search fails the node has to be re-searched with a larger window and if a better value is found during that search the principal variation changes. It is an indicator that the move-ordering in the game-playing program is bad if this happens frequently.

Table 5.7 shows that all trees except Pos_{17} had between 2 and 59 nodes that had more than five children of type *pv-node* (query Q_{20}) and the tree from Pos_{22} had five nodes that

Table 5.7: Number of nodes in each tree with several *pv-node* children

#pv-children	Pos_{15}	Pos_{16}	Pos_{17}	Pos_{18}	Pos_{19}	Pos_{20}	Pos_{21}	Pos_{22}	Pos_{23}
>2	91	29	1	87	32	26	243	556	160
>5	7	3	0	3	3	2	15	59	20
>7	0	0	0	0	0	0	0	6	6
>8	0	0	0	0	0	0	0	6	0
>9	0	0	0	0	0	0	0	5	0

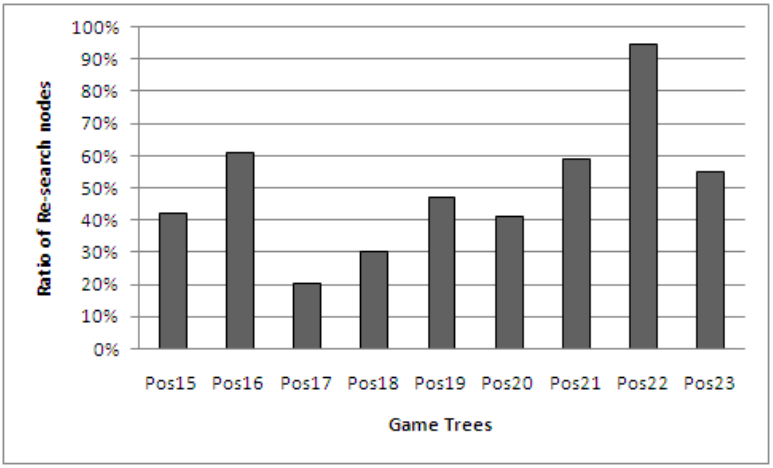


Figure 5.6: Ratio of nodes that have been re-searched compared to all nodes

had over nine *pv-node* children (query Q_{23}). This might indicate that the move ordering, i.e. the order in which the algorithm expands the nodes, is not doing a good job from this position. There could be several reasons for why it is difficult to expand the nodes in the best order. The evaluation function could be returning a poor estimate on the ”goodness” of the positions and if they are searched deeper the values change a great deal. This is consistent with our finding that the search from Pos_{22} was extended aggressively.

Query Q_{24} counts all nodes that were marked as re-search nodes. This means that these nodes were first searched with a null-window and failed and had to be searched again. If we look at the ratio of re-search nodes compared to all nodes in the tree in Figure 5.6 we can see that it is consistent with the result above. Pos_{22} has by far the largest ratio of re-search nodes as well as the most nodes with changing principal variation.

5.4 Summary

We have now showed that GTQT is both efficient and effective in running complex queries and finding anomalies in game trees. The normal case complexity of the algorithm has been verified by efficiency testing and several anomalies where found querying chess trees with GTQL queries. This concludes our discussion on GTQL and GTQT and in the next section we conclude and present ideas on future work.

Chapter 6

Conclusions

The contributions of this thesis are a new query language and a software tool for executing queries formed in the language. These contributions aid researchers and game-playing program developers in verifying the correctness of their game-tree search algorithms. This is a much needed addition to the arsenal of methods that are already in use for this purpose today.

In Chapter 3 we presented the new query language, GTQL. We explained the rationale behind the language and how it was designed to allow expressive questions about game trees to be formed and answered efficiently. The syntax and semantics of the language were explained and also how user-defined attributes and constants are embedded in the queries. The language allows hierarchical node relations to be expressed as well as supporting the aggregate function count(), which allows statistical information about trees to be gathered. These features collectively provide the language with sufficient expressiveness to form various types of interesting queries about game-tree structures.

We also designed and implemented a software tool for executing GTQL queries. The tool, GTQT, was described in Chapter 4. It uses a one-pass algorithm for executing pre-parsed queries, thus allowing even the most complex query to be answered in a single traversal of the game tree. We thoroughly tested the efficiency and scalability of the algorithm on synthetic game trees and presented the results in Chapter 5. Our experiments show that the complexity of the query is not a significant factor in the algorithm's processing time, and that the query time grows in a linear relation with the tree size. This allows us to answer even the most complex queries about huge game trees in a matter of only a few minutes. In Chapter 5 we also experimented with the tool on realistic game trees formed by the chess program Fruit. The performance was also good there, but more importantly the tool was used to discover several interesting anomalies in the game trees.

There are several additions and improvements that could be made for future versions of both GTQL and GTQT. The expressiveness of the language could be enhanced to enable querying of parent relations as well as extending the sibling relations. Also, other aggregation functions like min() and max() would be useful to have in the language. Implementation wise there are also improvements to be made. For example, a useful implementation feature is to be able to answer several queries simultaneously a single pass through the tree.

Also, run-time compression/decompression of the log files is something that needs to be considered since the log files can quickly become huge. Another way to address this would be to additionally offer real-time query processing inside a game program, thus totally bypassing the need for a log file where that is more applicable. Integrating the query tool into the Game Tree Viewer would also greatly add to the visualization of the result.

Even though we only discuss the language here in the context of game trees, there is nothing in its design that prevents it from being used in other search domains as well (e.g. planning). Currently, the only requirement is that the trees are generated in a depth-first fashion. It would, with additional work though, be possible to relax this requirement. But mostly, it is our sincere hope that this work will aid many researchers in the field of search algorithms with the tedious process of debugging and verifying the correctness of their programs, thus saving them countless hours of frustration and grief.

The Game-Tree Query Tool is available for download at cadia.ru.is.

Bibliography

Bird, S., Chen, Y., Davidson, S. B., Lee, H., & Zheng, Y. (2005, January). Extending XPath to support linguistic queries. In *Proceedings of programming language technologies for xml (planx)* (p. 35-46). Long Beach, California: ACM.

Björnsson, Y., & Ísleifsdóttir, J. (2006). Tools for debugging large game trees. In *Proceedings of the eleventh game programming workshop.* Hakone, Japan.

Björnsson, Y., & Ísleifsdóttir, J. (2007). GTQL: A query language for game trees. In *Proceedings of the twelfth game programming workshop* (p. 205-216). Amsterdam, Holland.

Björnsson, Y., & Marsland, T. A. (2001). Multi-cut alpha-beta-pruning in game-tree search. *Theor. Comput. Sci.*, *252*(1-2), 177-196.

Chamberlin, D. (2002). XQuery: An XML query language. *IBM Systems Journal*, *41*(4), 597–615.

Clark, J., & DeRose, S. (1999). *XML path language (XPath) 1.0.* (Tech. Rep.). W3C Recommendation.

Costeff, G. (2004). The Chess Query Language: CQL. *International Computer Games Association Journal*, *27*(4), 217–225.

Coulom, R. (2002). Treemaps for search-tree visualization. In J. W. H. M. Uiterwijk (Ed.), *The seventh computer olympiad computer-games workshop proceedings.*

Groote, J. F., & Ham, F. van. (2003). Large state space visualization. In *Proceedings of tools and algorithms for construction and analysis of systems (tacas 2003)* (Vol. 2619 of LNCS, p. 585-590).

Knuth, D. E., & Moore, R. W. (1975). An analysis of alpha-beta pruning. *Artificial Intelligence*, *6*(4), 293–326.

Letouzey, F. (2005). Internet Resource http://www.fruitchess.com. (Fruit Chess)

Louguet, F., & Échiquéenne, L. P. (2007). Internet Resource http://perso.orange.fr/lefouduroi/testlct2.htm. (LCT II v. 1.21)

Marsland, T. A. (1986). A review of game-tree pruning. *International Computer Chess Association Journal*, *9*(1), 3–19.

Marsland, T. A., & Campbell, M. (1982). Parallel search of strongly ordered game trees. *ACM Computing Survey*, *14*(4), 533–551.

Marsland, T. A., & Popowich, F. (1985). Parallel game-tree search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *7*(4), 442–452.

Marsland, T. A., & Reinefeld, A. (1993). *Heuristic search in one and two player games* (Tech. Rep. No. TR 93-02). Edmonton, Canada: Department of Computing Science, University of Alberta.

Reinfeld, A. (1983). An improvement of the scout tree-search algorithm. *International Computer Chess Association Journal*, *6*(4), 4–14.

Sahuguet, A., & Alexe, B. (2005). Sub-document queries over XML with XSQirrel. In *Www '05: Proceedings of the 14th international conference on world wide web* (pp. 268–277). New York, NY, USA: ACM Press.

Shneiderman, B. (1992, January). Tree visualization with tree-maps: A 2-d space-filling approach. *ACM Transactions on Graphics*, *11*(1), 92–99.

Zobrist, A. L. (1970). *A new hashing method with application for game playing* (Tech. Rep. No. TR 88). Madison: Computer Science Department, University of Wisconsin.

Appendix A

EBNF for GTQL

This appendix presents the Extended Backus Naur form that describes the complete grammar of the *Game Tree Query Language*, including keywords and syntax rules.

<query> := <node>
| <child>
| <subtree>
| <node>;<child>
| <node>;<subtree>
| <child>;<subtree>
| <node>;<child>;<subtree>

<node> := 'node' ':' <nodeexpr>

<child> := 'child' ':' <childexpr>

<subtree> := 'subtree' ':' <treeexpr>

<nodeexpr> := <expr>
| 'count' '(' <expr> ')'

<expr> := <ANDexpr> {'or' <ANDexpr>}
| <wildcard>

<ANDexpr> := <term> {'and' <ANDexpr>}
| <term>

<term> := <item> <op> <item>
| 'not' <term>
| '(' <expr> ')'

<item> := <var>
| <number>

<treeexpr> := <treeANDexpr> {’or’ <treeANDexpr>}

<treeANDexpr> := <treeterm> {’and’ <treeterm>}

<treeterm> := <treecountitem> <op> <treecountitem>
| ’not’ <treeterm>
| (<treeexpr>)

<treecountitem> := ’count’ ’(’ <expr> ’)’
| <number>

<childexpr> := <childANDexpr> {’or’ <childexpr>}

<childANDexpr> := <childterm> {’and’ <childANDexpr>}

<childterm> := <childcountitem > <op> <childcountitem>
| ’not’ <childterm>
| ’(’ <childexpr> ’)’

<childcountitem> := ’count’ ’(’ <siblingexpr> ’)’
| <number>

<siblingexpr> := <siblingANDexpr> {’or’ <siblingANDexpr>}
| <wildcard>

<siblingANDexpr> := <siblingterm> {’and’ <sibingterm>}

<siblingterm> := <siblingitem> <op> <siblingitem>
| ’not’ <siblingterm>
| ’(’ <siblingexpr> ’)’

<siblingitem> := <sibling><var>
| <var>
| <number>

<number> := - <digits>
| <digits>
| <alias>

<sibling> := ’[<]’ | ’[]’

<op> := ’=’ | ’!=’ | ’<’ | ’>’ | ’>=’ | ’<=’ | ’&’

<wildcard> := ’*’

<digits> := [’0’- ’9’] [’0’- ’9’]*

<var> := [’A’-’Z’, ’a’-’z’] [’A’-’Z’, ’a’-’z’, ’_’, ’0’-’9’]*

<alias> := [’A’-’Z’, ’a’-’z’] [’A’-’Z’, ’a’-’z’, ’_’, ’0’-’9’]*

Appendix B

Logging Game Trees

This is an example to show how to use the game-tree log interface in a game program. We use here a simple iterative deepening and minimax procedure for demonstration purposes (and omit various details that are not relevant for our demonstration purposes). Essentially, one must create a handle in the beginning (and delete in the end), open a new file for each search iteration, and call special functions when entering/exiting a node. The user collects the information he or she want to log in the structure $data$.

```
/*  Example TicTacToc program. */
#include "gt_log.h"
...
GTDataDescript gtDataDescr  =      /* <= GTL data description */
  { "TicTacToe", sizeof(Data_t), 0, {}, 5,
       { ...
         { "depth", offsetof(Data_t,depth), sizeof(int) },
         { "best" , offsetof(Data_t,best),  sizeof(int) },
         { "type" , offsetof(Data_t,type),  sizeof(int) } }
  };
GTLogHdl hGTL;   /* <=  Game-tree log handle. */

Value Minimax( Position_t *pos, int depth, Move_t move_last ) {
   Data_t  data;  /* <= GTL data record, user defined. */
   ...
   data.depth = depth;
   gtl_enterNode( hGTL, move_last );   /* <= GTL enter node.*/
   ...
   n = generateMoves(pos, moves);
   for ( i=0 ; i<n ; i++ ) {
      makeMove( pos, moves[i] );
      value = -Minimax( pos, depth-1, moves[i] );
      ...
      retractMove( pos, moves[i] );
   }
   ...
   data.best = best;
   gtl_exitNode( hGTL, &data );   /* <= GTL exit node */
```

```
   return best;
}

Value IterativeDeepening(  ... ) {
   ...
   for ( iter=1 ; iter<=maxIter ; ++iter ) {
       ...
       gtl_startTree( hGTL, filename, strFEN );   /* <= GTL open new game-tree log file */
       ...
       value = Minimax( &pos, iter, NULL_MOVE );
       ...
       gtl_stopTree( hGTL );     /* <= GTL close tree */
   }
   ...
}

int main() {
  ...
  hGTL = gtl_newHdl( "TicTacToe", &gtDataDescr ); /* <=  GTL new handle */
  if ( hGTL == NULL ) exit(EXIT_FAILURE);
  ...
  gtl_deleteHdl( &hGTL );    /* <=  GTL delete handle */
  ...
}
```

Appendix C

Result Files

GTQT returns two kinds of Game-tree index (.gti) result files. The type that is returned is decided by the query i.e. if the node-expression has a call to the *count()* function GTQT returns a result file with statistics, otherwise a result file with *node_ids*. Below are examples of the two files. The first four lines of the files are the same; on top is the date and time the file was created; next the the name of the game-tree log; there after the query; and last we have the query time. If we have a file with statistics line five displays on one hand "stat:" and then displays the number of true nodes in the next line. If on the other hand the result is a collection line five has the word "nodes:" and then the number of nodes that are true and it then has the *node_ids* of the result nodes in the following lines.

C.1 Result file with statistics

10:48 12.6.2007
tree12.gtf
node:count(type>0)
18.016
stat
1014195

C.2 Result file with node IDs

11:27 13.6.2007
tree12.gtf
node:type|1; child:count([ ]type|1)>2
15.157
nodes:28
39
3299
3886
6183
6189
6338
6503
6761
7164
7193
7916
8855
9992
10020
11567
12007
19332
19760
19980
20332
20430
20451
20600
21285
23909
47793
72451
86685

REYKJAVÍK UNIVERSITY

HÁSKÓLINN Í REYKJAVÍK

School of Computer Science
Reykjavík University
Kringlan 1, IS-103 Reykjavík, Iceland
Tel: +354 599 6200
Fax: +354 599 6301
http://www.ru.is