GTQ: A Language and Tool for Game-Tree Analysis

Jónheiður Ísleifsdóttir and Yngvi Björnsson

School of Computer Science, Reykjavík University, Reykjavík, Iceland
`{jonheiduri02,yngvi}@ru.is`

Abstract. The search engines of high-performance game-playing programs are becoming increasingly complex as more and more enhancements are added. To maintain and enhance such complex engines is a challenging task, and the risk of introducing bugs or other unwanted behavior during modifications is substantial. In this paper we introduce the Game Tree Query Language (GTQL), a query language specifically designed for analyzing 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), a software tool that allows efficient execution of GTQL queries on game-tree log files. The tool helps program developers to gain added insight into the search process of their engines, as well as making regression testing easier. Furthermore, we use the tool to analyze and find interesting anomalies in search trees generated by a competitive chess program.

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 [8, 18, 7, 1]. To maintain and enhance such complicated software is a challenging task, and the risk of introducing bugs or other unwanted behavior during modifications is substantial. A standard software-engineering approach for verifying that new modifications do not break existing code is to use *regression testing*. To a large extent that approach is what game-playing program developers use. They keep around large suites of test positions and verify that a modified program evaluates them correctly and plays the correct move. 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; they may stay hidden for a long time without surfacing. They 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 may instead degrade the efficiency of the search unnecessarily.

To detect these anomalies one must typically explore and gather statistics about the search process.

In this paper we introduce *Game-Tree Query Language (GTQL)*, a language specifically designed for querying game trees, and expand on previous work [5] by doing a thorough empirical analysis where GTQL queries are used to analyze and look for anomalies in search trees generated by a third-party competitive chess program. As demonstrated, the query language allows the game-program developers to gain better insight into the behavior of the search process of their programs and makes regression testing easier. The developer 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 containing the search trees. The queries are then run against the logs to verify that the search is behaving in accordance with expectations. This has the potential of substantially shorten the testing process as unwanted behaviors can be detected early, as opposed to after playing hundreds of test games or, in the worst case, never.

The paper is organized as follows. In the next section we describe the syntax and semantics of the language, followed by a section giving an overview of the usage and implementation of *Game-Tree Query Tool (GTQT)*, a tool for effectively executing GTQL queries. The efficiency and scalability of the tool is then empirically evaluated, and the tool used to detect anomalies in the search of the chess program FRUIT. Finally, we present conclusions and discuss future work.

2 The Game-Tree Query Language

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. The query parts must be listed in the order given above, separated by a semi-colon, but any unwanted parts can be omitted.

2.1 Query and Expression Evaluation

To be valid, expressions must be formed such that they evaluate to either true or false. By default a query returns the set of nodes in a game tree that fulfil the query, that is, the nodes for which all query parts evaluate to true. An example query is provided below:

```
node: type = PVNode;
child: count([]type = type) >= 5;
subtree: count(*) > 1000
```

Table 1. Operators listed by precedence

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

The query asks for nodes where the principal variation (PV) of the search is changing frequently (this could, e.g., be an indication of a bad move-ordering mechanism). The node expression evaluates to true only at PV nodes; the child expression counts the number of child nodes that are of the same type as the parent (i.e., also PV nodes) and returns true if there are at least five such child nodes; the subtree expression further limits the set of PV nodes that can fulfill the query by demanding their subtree being of a minimum size — this is done to exclude PV nodes close to the leaves where frequent PV changes may occur naturally.

The language is case sensitive and its expressions consist of *attributes*, *constants*, *operators*, and *functions*. *Attributes* refer to data fields associated with the nodes stored in the game-tree 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 above example *type* is an attribute telling whether a node is a PV, CUT, or an ALL node. Attribute names 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 (e.g., *PVNode* in our example query). The same naming convention is used for constant names as for attribute names. Information about attribute and constant names available to a query are stored in the game-tree file being queried.

The language operators fall into four categories: *hierarchical*, *attribute*, *relational*, and *logical* operators. They are listed in Table 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, [ ], is prefix referring to the child node currently evaluated. In our example, the child expression has such an operator for comparing the type of the child nodes to the type of the node evaluated by the node expression (the parent). The second hierarchical operator [<], not shown in the example, stands for the previously evaluated child. It can be used to compare two consecutive child nodes (e.g., to see if a node is being examined). The *attribute* operator, &, is essentially an inclusive bitwise *and*, and is used to extract flag bits from attribute fields. For example, a single node may be flagged simultaneously as a *quiescence* node and as belonging to a *null-move* search. The *relational* operators test for equality or inequality of attributes, constants, function results, and numbers, and the *logical* operators allow one to form arbitrarily complex expressions by combining Boolean expressions. Parentheses can be used to control precedence and order of evaluation.

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 wildcard character * may be used within 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 only exception is when the function is used stand-alone in a node expression. In that case, the query returns the actual count as opposed to a set of nodes. This is useful for gathering statistics about the tree, e.g., as in the example below where the total number of PV nodes in the tree is being counted:

```
node: count(type = PVNode)
```

More query examples are provided later in the paper. However, for a more thorough explanation of the syntax and semantics of GTQL, as well as for additional query examples, we refer interested readers to [14, 5].

3 Game-Tree Query Tool

The *Game-Tree Query Tool (GTQT)* is a software for parsing and executing GTQL queries. It is a console application that runs 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).

Below we give a brief overview of the one-pass algorithm used for executing the queries. The algorithm is capable of answering any single query, no matter how complex, in a single traversal of the game tree. The input to the program is a set of queries and a game-tree log file. In addition to the game-tree data (the attribute values of the nodes) the file stores meta-data, such as the names of the attributes and constants available to the query and information about the layout of the file. The tool, after processing and validating the meta-data, parses and syntactically checks the queries before executing them. For a more detailed discussion of the query execution algorithm, the logging mechanism, and the usage of the GTQL tool we refer readers to [14].

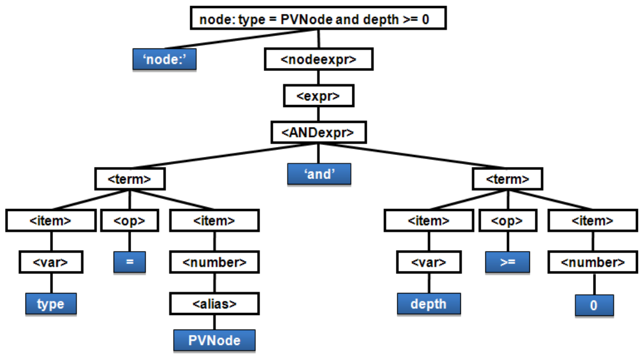


Fig. 1. An example parse tree

3.1 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 Fig. 1, 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 parse nodes return a Boolean value when evaluated, representing whether the corresponding expression evaluated to true or false for any given node in the game tree. Typically, the result of an evaluation on a game-tree node depends on the attribute values stored with the node. For example, in Fig. 1 the values of both the *type* and *depth* fields are required for evaluating the query; 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 function *count*, as it returns a count based on data accumulated over many nodes. Such queries cannot 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*. 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 required. Note that the counter lists and counter structures are not stored as a part of the parse tree because our query execution algorithm may have to execute several counter based *query instances* concurrently. Instead multiple instances of the query are created, each using its own set of counters (see later).

3.2 Executing a Query

Although the tool is used for post-processing game-tree logs, time is still of some essence when evaluating large game trees. The query execution algorithm makes only one pass through the game tree, during which it collects all information needed to answer the query. The algorithm is presented as Algorithm 1.

Algorithm 1 DFT-QUERY-EVAL(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-QUERY-EVAL(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)
```

It first checks if the node expression evaluates to true (line 2), and if so a new query instance is created and put onto a so-called query instance stack (line 4). The stack keeps track of active query instances. A single query may have several query instances active at the same time. More specifically, during the recursive depth-first traversal (line 8) of the game tree, each node on the path from the root to the current node where the node expression evaluates to true adds a new query instance. The need for having many query instances open at the same time is because separate counters are required for evaluating the child- and subtree-expressions of each instance, as their subtree scopes differ as shown in Fig. 2. Child- and subtree-expressions can be evaluated (line 13) only when the search backtracks after their corresponding tree scope has been fully traversed. If both expressions evaluate to true, the node is added to set of results (line 14) and the query instance then popped off the stack (line 15). The subroutine *evalCounter-Exprs* updates the counters associated with the `count(<expr>)` expressions. It is called for both child expressions (line 9) and subtree expressions (line 17). In the latter case, counters in all query instances on the stack are updated.

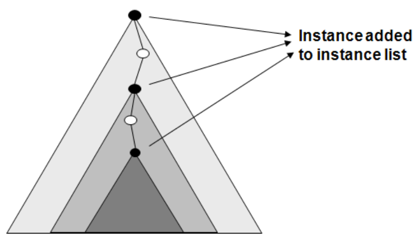
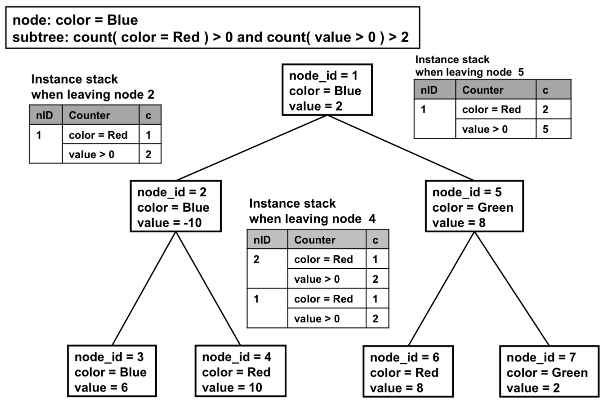


Fig. 2. Subtree scope of different nodes in the same line



Instance stack
when leaving node 2

nID	Counter	c
1	color = Red	1
	value > 0	2

Instance stack
when leaving node 4

nID	Counter	c
2	color = Red	1
	value > 0	2
1	color = Red	1
	value > 0	2

Instance stack
when leaving node 5

nID	Counter	c
1	color = Red	2
	value > 0	5

Fig. 3. Example of one-pass query evaluation

An example of the one-pass query-evaluation process is given in Fig. 3. 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 because 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-QUERY-EVAL 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 Experiments

To demonstrate the potentials of GTQL we used it to analyze game-tree logs generated by a competitive chess program. In this section we report our findings.

Table 2. Game trees used in the 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

4.1 Experimental Setup

For our experiments we used the chess program Fruit [15], 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 [6]. We used version 2.1 of the program, which is the strongest open-source chess engine available (subsequent versions of the program were not open source). The only modification we made to the program was to augment its search engine with code for collecting attribute values and with calls to the game-tree log library.

The chess program was instructed to search nine tactical chess positions taken from the LCT II test suite (positions number 15-23) [16]. This suite is one of several frequently used standard test suites to measure chess programs' performance. On each of the problems the chess program was run until the correct solution was found. For each position, a separate game-tree log was generated for each search iteration. The solution (best move played) was found on iterations varying from the second to the tenth ply, as shown in Table 2. The first column indicates the position within the suite; the second column, SD, shows the search depth of the iteration where 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 for each position the game-tree log from the iteration where the solution was first found (SD). The experiments were run on a 3GHz Linux-based computer with 2GB of main memory.

4.2 Processing Throughput

We start by measuring the throughput of the query tool. It can process around 500 to 600 thousand nodes per second from the game-tree log, depending on the complexity of the query. The average time complexity of our one-pass query algorithm is $O(n * log(n))$ where n is the number of nodes, so the throughput degrades only slightly with increasingly larger trees [14]. The throughput speed is in the ballpark of how fast chess programs search and log the game trees.

Table 3. 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%

4.3 Node Type Statistics

Next we asked queries for collecting statistics about the game trees, more specifically the ratio of PV, CUT, and ALL nodes. The queries are shown below:

```
node: count(type = PVNode)
node: count(type = CUTNode)
node: count(type = ALLNode)
```

and the result in Table 3. The result looks as one would expect: very low ratio of PV nodes, and 2 to 3 times more CUT than ALL nodes. The only deviation from this is in Pos_{16} where there is a unusually high ratio of PV nodes, but that is not much of a concern because of the fact that the tree is small (PV changes are not too uncommon in shallow trees). This example, although not providing much additional insight, is good for a sanity check to confirm the expected behavior. The statistics were provided as a demonstration of the type of statistics that can be collected. One must also be a little cautious when working with accumulated statistics, as they may overlook individual anomalies. We thus look more carefully at PV changes in the next subsection.

4.4 Principal Variation Changes

The query below was executed for different values on n:

```
node: type = PVNode;
child: count([]type = type) >= n
```

The result is shown in Table 4. As can be seen, frequent PV changes are uncommon, although there are a few problematic nodes in Pos_{22} that might warrant a further investigation (there are 5 positions with 9 or more PV child nodes).

4.5 Large Quiescence-Search Trees

Quiescence searches are essential in chess programs for evaluating unstable positions — such searches typically include selected captures and even checks.

Table 4. Number of PV 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
9	0	0	0	0	0	0	0	5	0

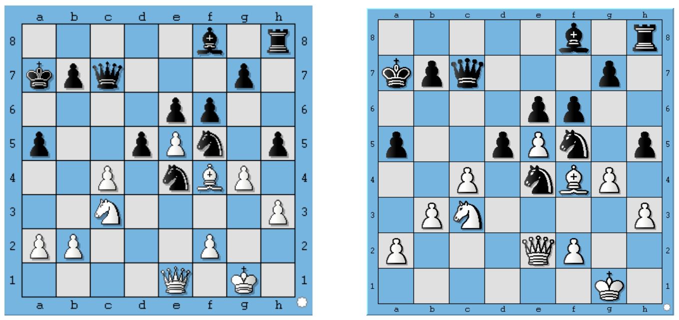


Fig. 4. Chess positions with a large quiescence search

However, because of the frequency of quiescence searches, it is important to keep their size under control, a process that takes a careful tuning. To obtain more insight into the size of the quiescence searches of FRUIT we executed the following query for different values of n:

```
node: flags & QRootNode;
subtree: count(*) > n
```

The attribute *flags* is used in the chess program to mark various node properties, such as whether a node belongs to a research, a null-move search, or a quiescence search. The root of a quiescence search tree is marked as *QRootNode*.

For all positions except one, the quiescence searches were all under 200 nodes (and most much smaller). In Pos_{18} (from the game Vanka - Jansa, Prag 1957), however, four of the quiescence-search subtrees had more nodes. For example, from the two positions shown in Fig. 4 the generated quiescence-search trees were of size 840 and 486 nodes, respectively. This is quite excessive compared to a normal search, and should raise a flag as something that warrants further investigation. This is a good example of how the tool can be used to help identify problems with the search performance.

5 Related Work

To the best of our knowledge GTQL is the first query language specifically designed for game trees. However, several query languages for tree structures exists, including XPath [10] for querying XML data. The navigational abilities of XPath have been used in subsequent languages either by directly supporting

XPath like XQuery [9] does or extending its syntax like is done in LPath [2]. XSQuirrel [17] is a related language for making sub-documents out of existing XML documents. None of the aforementioned languages are well suited for our purpose and do for example not allow aggregation. However, there does exist a chess-specific query language, Chess Query Language (CQL) [11], but it is designed for matching chess positions, not tree structures.

On a further account, there do exist some tools that can be helpful in visualizing game trees. Rémi Coulom presented a visualization technique for search trees using treemaps [12]. Treemaps are based on the idea of taking a rectangle and dividing it into sub-rectangles for each subtree. The first rectangle is split vertically, one rectangle per child. Those rectangles are then split horizontally for each of their children and so on. Although such a technique can give some insight into where the search mainly spends it effort, it is insufficient for detecting most search abnormalities. There do also exist browsers that allow one to navigate through game-tree logs and look at resulting game positions [3, 4, 13].

6 Conclusions

From the above results we may fairly conclude that the new query language and software can aid researchers and game-playing program developers in verifying the correctness of their game-tree search algorithms. The syntax and semantics of the language are explained in such terms that GTQL can be used by others. The GTQ tool expresses queries about complex game-tree structures, including hierarchical relationships and aggregated attributes over subtree data. Last but not least, in this paper we demonstrated the usefulness of the GTQ tool by analyzing and finding abnormalities in the search trees of the competitive chess program Fruit. These are just a few examples of the usefulness of GTQ. The tool is quite flexible as the users decide which information about the game trees to log. For example, by logging static node evaluations one can envision the tool being useful to researchers working on search, e.g., for finding pathological behaviors or for measuring diminishing returns of deeper search.

GTQL is the first query language specifically designed for game trees. There are still many additions and improvements that could be made in future versions of both GTQL and GTQT. For example, the expressiveness of the language could be enhanced, e.g., to include parent relations (and more generally ancestor relations), as well as an extended sibling relation. Also, other functions like *min* and *max* would be useful. Moreover, there are improvements to be made to the implementation; the two most prominent ones are: (1) allowing many queries to be answered simultaneously, and (2) introducing run-time compression/decompression to the game-tree log files as they can quickly grow large. As of now, the tool cannot handle game trees built in parallel. This limitation is worthwhile to be addressed in future versions as multi-core processors are becoming mainstream.

Finally, it is our hope that this work will aid 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 http://cadia.ru.is.

Acknowledgments. This research was supported by grants from The Icelandic Centre for Research (RANNÍS) and by a Marie Curie Fellowship of the European Community programme *Structuring the ERA* under contract MIRG-CT-2005-017284. We also thank the anonymous referees for their insightful comments.

References

1. D. Billings and Y. Björnsson. Search and knowledge in Lines of Action. In H.J. van den Herik, H. Iida, and E.A. Heinz, editors, *ACG*, volume 263 of *IFIP*, pages 231–248. Kluwer, 2003.
2. S. Bird, Y. Chen, S.B. Davidson, H. Lee, and Y. Zheng. Extending XPath to support linguistic queries. In *Proceedings of Programming Language Technologies for XML (PLANX)*, pages 35–46, Long Beach, California, January 2005. ACM.
3. Y. Björnsson. *Selective Depth-First Game-Tree Search.* PhD thesis, University of Alberta, Canada, June 2002.
4. Y. Björnsson and J. Ísleifsdóttir. Tools for debugging large game trees. In *Proceedings of The Eleventh Game Programming Workshop*, Hakone, Japan, 2006.
5. Y. Björnsson and J. Ísleifsdóttir. GTQL: A query language for game trees. In *Proceedings of The Twelfth Game Programming Workshop*, pages 205–216, Amsterdam, The Netherlands, 2007.
6. Y. Björnsson and H.J. van den Herik. The 13th world computer-chess championship. *ICGA Journal*, 28(3):162–175, 2005.
7. M. Buro. How machines have learned to play Othello. *IEEE Intelligent Systems*, 14(6):12–14, November/December 1999. Research Note.
8. M. Campbell, A.J. Hoane Jr., and F.-h. Hsu. Deep blue. *Artificial Intelligence*, 134(1-2):57–83, January 2002. Special Issue on Games, Computers and Artificial Intelligence.
9. D. Chamberlin. XQuery: An XML query language. *IBM Systems Journal*, 41(4):597–615, 2002.
10. J. Clark and S. DeRose. XML path language (XPath) 1.0. Technical report, W3C Recommendation, 1999.
11. G. Costeff. The Chess Query Language: CQL. *ICGA Journal*, 27(4):217–225, 2004.
12. R. Coulom. Treemaps for search-tree visualization. In J.W.H.M. Uiterwijk, editor, *The 7th Computer Olympiad Computer-Games Workshop Proceedings*, 2002.
13. A. Fortuna. Internet Resource http://chessvortex.com/chant, 2003. CHANT: A Tool to View Chess Game Trees.
14. J. Ísleifsdóttir. GTQL: A Game-Tree Query Language. Master's thesis, Reykjavik University, Iceland, January 2008. http://www.ru.is/?PageID=7094.
15. F. Letouzey. Internet Resource http://www.fruitchess.com, 2005. Fruit Chess.
16. F. Louguet and La Puce Échiquéenne. Internet Resource http://perso.orange.fr/lefouduroi/testlct2.htm, 2007. LCT II v. 1.21.
17. A. Sahuguet and B. Alexe. Sub-document queries over XML with XSQirrel. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*, pages 268–277, New York, NY, USA, 2005. ACM Press.
18. J. Schaeffer. *One Jump Ahead: Challenging Human Supremacy in Checkers.* Springer-Verlag New York, Inc., 1997.