Chapter 6

Evaluations and conclusions

In this thesis we have presented research on the balance between memory and search in two-player zero-sum games. As example domains we have used the games of chess and domineering. The trade-off between memory and search led to the formulation of three problem statements. In this chapter the problem statements are re-addressed and evaluated.

6.1 More memory and less search

We investigated whether we can exploit the large amount of memory currently available. The underlying idea is that storing more knowledge into memory may result in a decreasing need for search. A depth-first search algorithm only stores the path from the root to the node under investigation, and hence it uses little memory. However, many depth-first search algorithms use the available memory for keeping a transposition table. The transposition table eliminates the need for search at identical nodes, because the results of previous search processes have been saved in the table. The first problem statement addresses decreasing the need for search by increasing the use of memory.

Problem statement 1: Which methods exist to improve the efficiency of a transposition table?

In Chapter 2 we investigated three methods of improving the efficiency of a transposition table. Irrespective of the size of the transposition table, collisions (cf. subsection 2.4.2) are bound to occur. When a collision occurs, a choice has to be made which of the two positions involved should be preserved in the transposition table. Such a choice is governed by a replacement scheme.

The first method to improve the efficiency of the transposition table is *to improve the replacement scheme*. Experiments have shown that a two-level scheme works significantly better than a traditional one-level scheme. Further, the concept *Big*

(based on the number of nodes of the subtree) works better than the most widely used concept *Deep* (based on the depth of the subtree).

The second (obvious) method of improving the efficiency of a transposition table is *to increase the number of positions in the table*. In most implementations the number of positions usually is a power of two. Hence, increasing the number of positions means doubling the number of positions. However, after a certain transposition-table size has been reached it turns out that doubling the number of positions in the table has a limited benefit. Moreover, doubling the number of positions in the table can cause the table to take up too much memory.

When doubling the number of positions has a limited benefit the memory can be used *to store additional information in an entry*. This is the third method for improving the efficiency of the transposition table. We first have performed experiments to investigate which information is more important to store in a transposition-table entry: the best move in a position, or the score of that move. Experiments show that the score is more important than the move. Further, we have investigated the effect of storing an n-ply PV ($n = 2..5$) in an entry, instead of only the best move (a 1-ply PV). Our results show that storing additional information in an entry is a profitable way of using the available memory, which outperforms the benefit of doubling the number of positions in the table. We believe that this is a fruitful domain for future research (cf. Section 6.4).

6.2 Less memory and more search

We investigated whether we can exploit the increase in computer speed. The underlying idea is that more speed enables more search, thereby acquiring more knowledge, and hence decreasing the need for memory. Best-first search needs a large amount of memory to store the entire search tree. At present computer speeds, the memory available is quickly filled. Since the quality of a best-first search algorithm depends on the quality of the directing knowledge, ways have to be found to use the increase in speed to acquire more knowledge per node, hence also improving the directing knowledge. Consequently, the search process will search the state space more efficiently, reducing the need for memory at the cost of more search. The second problem statement addresses decreasing the need for memory by increasing the use of search.

Problem statement 2: Which methods exist for best-first search to reduce the need for memory by increasing the search, thereby gaining more knowledge per node?

In Chapter 4 we introduced the pn^2-search algorithm. This is a best-first search algorithm (pn search), using a second search (also pn search) as evaluation of a leaf, thereby adding more (directing) knowledge to every node in the search tree. Experiments with this algorithm (listed in section 4.4) show that pn^2 search is a good method of reducing the need for memory by increasing the search. The pn^2-search algorithm uses roughly twice as much search time compared to the traditional

pn-search algorithm, leading to a decrease in the need for memory. A further advantage of the pn^2-search algorithm is that it solves test positions not solvable (due to memory constraints) by a standard pn-search framework.

6.3 Less memory and less search

In the first problem statement we tried to reduce the need for search by increasing the use of memory. Analogously, in the second problem statement we tried to reduce the need for memory by increasing the use of search. An attempt to combine the advantages of both approaches (reducing the need for search *and* reducing the need for memory) is the following. In a search tree it is profitable to recognize transpositions and to ensure that for each set of identical nodes, only one subtree is expanded. If a best-first search algorithm (which stores the whole search tree in memory) is used, the search tree is converted into a search graph, by joining identical nodes into one node. This causes subtrees to be merged, decreasing the need for memory. Since the graph contains fewer nodes than the tree, less searching is needed as well. However, joining identical nodes into one node introduces the so-called *graph-history-interaction* (GHI) problem, since determining whether *nodes* are identical is not the same as determining whether the *search states* represented by the nodes are identical. The third problem statement addresses decreasing the need for memory *and* decreasing the need for search.

Problem statement 3: Is it possible to give a solution for the GHI problem for best-first search?

In Chapter 5 we have given a solution to the GHI problem for best-first search, resulting in a Directed-Cyclic-Graph (DCG) algorithm for pn search, called the BTA (Base-Twin Algorithm) algorithm. This algorithm is based on the distinction of two types of nodes, termed *base nodes* and *twin nodes*. The purpose of these types is to distinguish between equal positions with different history. By transferring the search tree into our implementation of a search DCG, less memory is needed, since only the roots of equal subtrees are duplicated. Furthermore, less search is needed, since the DCG contains fewer nodes than the tree. It is shown that our algorithm is hardly less efficient than other, not entirely correct DCG algorithms in terms of numbers of nodes searched. One drawback of our solution is the cost of finding the node to be expanded, in the case that many transpositions occur. We are convinced that the advantage of solving the GHI problem outweighs this drawback.

6.4 Future research

In this section several recommendations for future research on the trade-off between memory and search are given.

6.4.1 More memory and less search

This subsection provides some ideas for future research on $\alpha\beta$ search in combination with a transposition table. The idea of using an n-ply principal variation in an entry, instead of only the best move (cf. subsection 2.7.3), seems worthy of further investigation. Based on the experiments concerning $\alpha\beta$ search with a transposition table (cf. Chapter 2) it is advised to concentrate on using additional information affecting the number of cut-offs generated by bound values.

A second recommendation is to store the best n moves with their respective values (exact values, upper bounds, or lower bounds) in an entry, instead of only storing the best move.

As a third recommendation it may be worthwhile investigating whether an entry is still effective in the table. To this end we store in a transposition-table entry the last time the position from this entry has been *read* in the search[1], and we use this stamp for the decision what to do when a collision occurs.

The transposition table can also be used to store results of partial game boards. when using partition search (Ginsberg, 1996). After a certain number of moves played in the game of domineering, the board is usually divided into separate (and smaller) regions. The search time will decrease considerably if the results of these regions can be found in the transposition table. In this case it is not sufficient to store only the values of *win* and *loss* in the table, since it has to be known by what margin a player can win a region (Conway, 1976; Berlekamp, 1988).

6.4.2 Less memory and more search

This subsection lists some ideas for future research on modifications of the pn-search algorithm (or other best-first search algorithms), decreasing the need for memory. The fraction function used in Section 4.2 works well. Still, it would be interesting to investigate whether other fraction functions perform even better. After every initialization of a most-proving node in the first-level tree, pn^2 search deletes the second-level tree. If the next most-proving node is one of the children of the previously expanded node, then the second-level tree is recreated. Therefore, it could be advantageous to store the last N second-level trees in a cache to reduce this overhead, a proposal already suggested by Schaeffer (mentioned by Allis, 1994).

The pn^2-search algorithm can be seen as a pn-search algorithm with another pn search for evaluation. Other combinations are worthwhile to be investigated, such as the combination of pn search and $\alpha\beta$ search, leading to two variants: (1) use the pn-search algorithm with $\alpha\beta$ search for evaluation, and (2) use the $\alpha\beta$ algorithm with pn search for evaluation. The first variant can be used e.g., in a chess tactical analyzer: pn search uses $\alpha\beta$ search at the leaves to get a more accurate evaluation. The second variant can be used e.g., in a chess program: a (positional) $\alpha\beta$ search uses pn search at the leaves to check for forced mates.

[1] We note that this is a method different from time stamping, where the last time a position has been *written* into an entry is stored.

6.4.3 Proof-number search

In this subsection we give two recommendations for improving pn search as it is used in the game of chess.

First, pn search often finds a longer mate than the optimal shortest one. If it is desired to urge pn search to find a shorter mate than it does at present, the following two solutions are suggested: (1) after a mate has been found, try searching for a shorter mate by only examining nodes in the search tree at a lower depth than the depth of the shortest mate found so far, or (2) the proof and disproof numbers in the leaves are initialized to values over unity, say at the depth of the node in question in the tree; this deters deep searches and hence long mates.

Second, in Chapter 3 it is shown that pn search is a good searcher for mates, especially when the winning variation contains forcing moves. When considering extending pn search to other tactical problems, say as a tactical analyzer for gaining material, a difficulty arises: the condition for suspending search (recognizing the proved or disproved nature of a node) is not easy to formulate. Temporary gains should be discarded, and proved or disproved should hold only when the material gain is permanent. Then and only then the goal is reached and the node should be evaluated to *true*, as is a win in standard pn search. A possible definition, worthwhile testing, is: the gain value of a node is stable if the attacker is to move and has gained at least the material expected. Since this definition of a stable gain is a heuristic, it may be incorrect. To prevent unwanted effects, the variation found by pn search can be checked by an $\alpha\beta$ search. The variation can also be used to sort the moves in the $\alpha\beta$ search, resulting in deeper searches than a standard full-width search, because of the additional cut-offs of pn search.