

Transposition-Driven Scheduling in Parallel Two-Player State-Space Search

Jan Renze Steenhuisen



Delft University of Technology

Transposition-Driven Scheduling in Parallel Two-Player State-Space Search

Master's Thesis in Computer Science

Parallel and Distributed Systems group
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology

Jan Renze Steenhuisen

June 13, 2005

Author

Jan Renze Steenhuisen

Title

Transposition-Driven Scheduling in Parallel Two-Player State-Space Search

MSc presentation

June 20, 2005

Graduation Committee

prof. dr. ir. H. J. Sips (chair)	Delft University of Technology
ir. dr. D. H. J. Epema	Delft University of Technology
prof. dr. H. J. van den Herik	Maastricht University
dr. K. van der Meer	Delft University of Technology

Abstract

This thesis describes the design and implementation of a parallel two-player state-space searcher called DARKSIGHT. The program uses principal variation search as its main search algorithm with numerous additional techniques and heuristics. The young-brothers-wait concept is used as the splitting strategy. Transposition-driven scheduling is used as its load-balancing scheme, which achieved near-linear speedups in parallel one-player state-space search. Our experiments show that transposition-driven scheduling is used successfully with performance results that are comparable to those achieved by other parallel two-player state-space searchers. Additionally, we present new experimental results on the deep search behaviour of two-player state-space searchers, and results that indicate that the commonly used constant-sized aspiration windows are not optimal.

“Ich habe ein leises Gefühl des Bedauerns für jeden, der das Schachspiel nicht kennt, so wie ich jenen bedauere, der die Liebe nicht kennengelernt hat. Das Schachspiel hat wie die Liebe, wie die Musik die Fähigkeit, den Menschen glücklich zu machen.” – Siegbert Tarrasch

Preface

The work lying before you is my Master of Science thesis, in which a description is given of the work done and the results achieved since January 2004 at the Parallel and Distributed Systems group, Faculty of EEMCS, Delft University of Technology. For more than ten years I have been interested in computer chess. Unfortunately, it was mere reading about computer chess that kept me occupied during these years. Although I had the urge to build my own program, my other occupations kept me from coding. Having the opportunity to spend ‘eight months’ on one project, I decided to take this time to write my own chess-playing program.

Before continuing with my thesis, I would like to thank some people who have been of great help during my Master’s project. First of all, I would like to thank Anke, for her patience, comments and support. I would like to thank my parents for their patience and financial support throughout my years at the university. I want to thank the graduation committee members for their collective and individual contributions. My last thanks go to the many people I have spoken with on many coffee breaks, in which thoughts were shared on a variety of topics.

Renze Steenhuisen

Delft, The Netherlands
June 13, 2005

Contents

Preface	vii
1 Introduction	1
2 Problem Setting	3
2.1 Sequential Game-Tree Search	3
2.1.1 Minimax Search	4
2.1.2 Alpha-Beta Search	4
2.1.3 Principal Variation Search	4
2.1.4 Transposition Table	5
2.2 Parallel Game-Tree Search	5
2.2.1 Preliminaries	5
2.2.2 Splitting Strategies	7
2.2.3 Load Balancing	8
2.2.4 Distributed Transposition Tables	9
2.3 Transposition-Driven Scheduling	9
2.3.1 The Algorithm	10
2.3.2 Wide-Area Transposition-Driven Scheduling	11
3 Design and Implementation of a Sequential Game-Tree Searcher	13
3.1 Global Design	13
3.2 The Framework	14
3.2.1 Board Representation	14
3.2.2 Move Generator	14
3.2.3 Making and Unmaking Moves	15
3.3 The Search	15
3.3.1 Transposition Tables	16
3.3.2 Move Ordering	17
3.3.3 Extension Heuristics	19
3.3.4 Pruning Heuristics	20
3.4 The Evaluation Function	21
3.4.1 Patterns and Values	21
3.4.2 Endgame Tablebases	23

3.4.3	Interior-Node Recognition	23
3.4.4	Quiescence Search	25
4	Design and Implementation of a Parallel Game-Tree Searcher	27
4.1	Global Design	27
4.2	The Framework	28
4.2.1	Communication	28
4.2.2	Supporting Data Structures	29
4.3	The Search	30
4.3.1	Young Brothers Wait Concept	31
4.3.2	Transposition-Driven Scheduling	31
4.3.3	Additional Search Techniques	32
4.4	Determining the Granularity	33
5	Experiments in Game-Tree Search	35
5.1	The Aspiration Window Experiment	35
5.1.1	Experimental Setup	36
5.1.2	Results	36
5.2	The Deep Search Experiment	38
5.2.1	Background	38
5.2.2	Experimental Setup	39
5.2.3	Confidence Bounds	39
5.2.4	Results	39
5.3	Speedup Experiments with DARKSIGHT	42
5.3.1	Experimental Setup	43
5.3.2	Results	44
6	Conclusions and Future Work	51
6.1	Conclusions	51
6.2	Future Work	52
A	Deep Search	59
B	Bratko-Kopec Test Set	63

Chapter 1

Introduction

In daily life, people are often in a situation where choices have to be made in which they want to choose the best alternative according to some criteria. In computer science terminology, such a situation is called a state and the choices are referred to as state-transitions or moves. Before choosing one of the alternatives, the resulting states are evaluated, and the best is selected. This search process is called *state-space searching*.

The same problem of making choices in every day life arises when playing a game. In games, the states and the possible transitions are clearly defined by rules, making them an ideal test environment for research on state-space searching. When the situation can be searched thoroughly, the best choice is selected with certainty. In general, however, state spaces are too large to search extensively within a reasonable amount of time.

The problem of searching faster and deeper is elementary to computer science. It is especially of great importance in the field of artificial intelligence. No wonder that state-space searching has a rich history in which many sequential algorithms have been proposed and studied thoroughly. Despite this rich history, parallel state-space searching still holds many unsolved research challenges.

State-space searching can be divided into multiple problem domains. For instance, a distinction can be made between *one-player* and *two-player* games. One-player games, such as Rubik's cube, are characterised by having a goal state in which the problem is solved. Two-player games, like chess, have numerous final states which are either drawn or won by one of the players.

Recent developments in parallel one-player state-space searching have resulted in near-linear speedup [86]. The search for an efficient parallel algorithm for searching faster and deeper in two-player games is still continuing, although DEEP BLUE II won against the reigning human world chess champion Gary Kasparov. However, the more fundamental question remains unanswered: Does searching more deeply actually result in stronger play? Although both intuition and experiments show [9, 24, 51, 56, 98, 101] that it does so, the question why remains [7]. Even the actual gain in performance of searching more deeply is unknown.

As a preparation to this MSc project, a thorough study on sequential and parallel two-player search algorithms has been performed [95]. In one-player games, a new scheduling scheme has successfully been applied to a parallel search algorithm called *transposition-driven scheduling* [86]. Although this new scheme shows very promising, the application to two-player games introduces some problems that need to be solved. The goal of this MSc project is to apply transposition-driven scheduling to a parallel two-player state-space searcher, or *game-tree searcher*, in order to measure the achievable speedup.

To this end, we developed a chess-playing program from scratch and called it DARKSIGHT, which consists of approximately 26,500 lines of code. An efficient algorithm has to be used as the basis of a parallel state-space searcher, because using a poor searcher may result in too optimistic speedups [16]. DARKSIGHT is an efficient searcher, although the actual playing strength is not considered. The key characteristics of DARKSIGHT, such as the search algorithm and the expansion speed, are comparable to those of strong chess programs. This makes it a representative program for measuring the achievable speedup of applying transposition-driven scheduling to a parallel two-player state-space searcher.

The results of three experiments are presented, two of which use the chess-playing program CRAFTY and one uses DARKSIGHT. First, the *aspiration window* [94] experiment provides insight into the fluctuation of the search values returned by CRAFTY. Second, experimental results on the *deep search* behaviour of CRAFTY are presented which answer some questions raised in the literature [42, 47, 55] with a high level of confidence. Finally, the achieved speedup is presented of DARKSIGHT in which transposition-driven scheduling is used successful.

The remainder of this thesis contains five chapters and its contents is as follows. In Chapter 2, some important algorithms and ideas in sequential and parallel game-tree searching are described. In Chapter 3, a detailed description of the sequential part of DARKSIGHT is given. In Chapter 4, the parallel part of DARKSIGHT is described in detail. In Chapter 5, the results of the aspiration windows experiment, the deep search experiment, and the speedup experiments using transposition-driven scheduling in the domain of chess are discussed. Finally, conclusions are drawn from the results and are summarised in Chapter 6, together with some suggestions for future research.

Chapter 2

Problem Setting

This chapter presents prior work in state-space searching that is relevant to the implementation of the sequential and parallel chess-playing program DARKSIGHT. In Section 2.1, some sequential algorithms for searching a two-player state space are described. In Section 2.2, the splitting strategies and load-balancing schemes used in parallel two-player state-space searching are presented. In Section 2.3, an explanation is given of Transposition-Driven Scheduling.

2.1 Sequential Game-Tree Search

In two-player zero-sum games with complete information, like chess, the players have opposite goals that can only be reached at the expense of the other. Both players have the same game-relevant information and are assumed to try to play optimally. Game-tree searching deals with finding the best transition from a certain state to another. An evaluation function values a state, with smaller values representing better states for one player and higher values better for the other. A two-player zero-sum game amounts to the problem of one player trying to maximise and the other player trying to minimise this value. An example of a limited game tree is given in Figure 2.1.

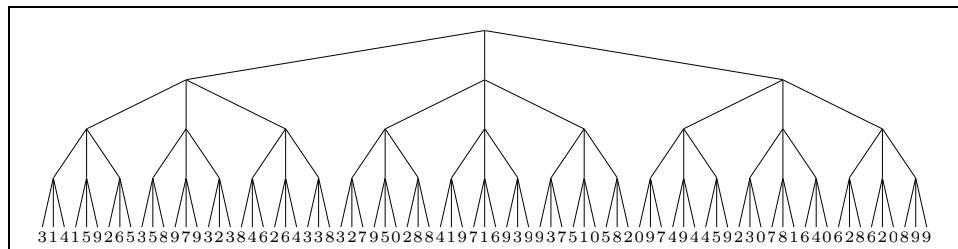


Figure 2.1. An example of a limited game tree.

2.1.1 Minimax Search

Assuming passing is not allowed, so a move must be chosen, the maximising and minimising players are at alternating levels of the tree. *Minimax* search [92] solves the problem in a straightforward way by recursively taking either the maximum or minimum value of its successors. Minimax constructs a full-width *search tree* by expanding every legal move at *interior nodes*, until either a certain depth (*horizon*) or a *terminal node* (mate or draw) is reached. The algorithm's complexity is therefore $\Theta(w^d)$, with w the effective branching factor and d the search depth. This algorithm makes it almost impossible to reach a reasonable depth in games like chess with $w \approx 37$ [28].

Negamax is a different formulation of the minimax search algorithm, which maximises the negated values of its successors. Due to its shorter form, the negamax version of algorithms is mostly used.

2.1.2 Alpha-Beta Search

The *Alpha-Beta* search algorithm [60] returns the same value as minimax, while expanding a smaller search tree. The algorithm maintains a *search window* $[\alpha, \beta]$, with α being a lower bound and β an upper bound on the overall value, which is initialised at $[-\infty, \infty]$ and narrows due to the progressing search. Instead of expanding every node full-width, it prunes sub-trees that cannot change the overall value (*cutoffs*). *Move ordering* is measured by the probability of the first considered move being the move that causes a cutoff. The algorithm's runtime depends on move ordering and is $\Omega(w^{\lfloor d/2 \rfloor} + w^{\lceil d/2 \rceil} - 1)$ and $O(w^d)$.

When every state at level d is assigned a sequence of positive integers $a_1 a_2 \dots a_d$. The root position corresponds to the empty sequence and all w successors of a position $a_1 a_2 \dots a_d$ are assigned the sequences $a_1 a_2 \dots a_d 1, \dots, a_1 a_2 \dots a_d w$. Let a position be *critical* if $a_k = 1$ for all odd or even k , then the critical positions $a_1 a_2 \dots a_d$ of a perfectly ordered tree are labeled:

- *PV-node*: if $a_j = 1$ for $1 \leq j \leq d$.
- *CUT-node*: if a_j is the first entry that is larger than 1 and $d - j$ is even.
- *ALL-node*: otherwise

2.1.3 Principal Variation Search

Principal variation search (PVS) [66, 67, 80, 81] is an Alpha-Beta-like searcher that assumes very good move ordering, and uses the fact that a smaller window results in smaller search trees. The algorithm works like the Alpha-Beta algorithm, but assumes perfect move ordering. Therefore, PVS merely searches the *principal variation* (PV), sequence of PV-nodes, with a full search window and all alternatives with a *minimal window* or *null-window*.

2.1.4 Transposition Table

Although, a state space is commonly referred to as a tree it often is a directed acyclic graph instead. *Transpositions* often occur in these graphs, which are equal states except for the sequences of moves leading to them. Although discarding the move history introduces the *graph history interaction* problem [12], it is generally omitted. *Transposition tables* are used to reuse the results of sub-trees that have already been searched, which results in large time savings [79]. Transposition-table entries should provide sufficient information in the least number of bytes, in order to optimise the number of positions stored in limited sized memory. An entry should at least provide the hash key, best move, value, bound (*lower*, *upper* or *exact*) and a quality measure, where depth is most commonly used [15, 76]. Hash tables implement transposition tables efficiently, needing only an efficient way of constructing hash keys as the unique *state signatures*. The method for constructing hash keys described by Zobrist [107] is the most commonly used. Additionally, a replacement scheme has to be used for situations in which different states are mapped onto the same table entry because limited memory is used. Many replacement schemes have been formulated and studied extensively throughout the years [13, 14, 76].

2.2 Parallel Game-Tree Search

Although sequential search algorithms have become more and more efficient, searching the game tree faster and deeper is needed. Parallel computation has a great potential in fulfilling this need, because of the additional resources. The problem, however, is how to use these additional resources efficiently. While the minimax algorithm can be parallelised trivially, Alpha-Beta-like algorithms are harder to parallelise efficiently because they are inherently sequential.

2.2.1 Preliminaries

A good understanding of the most important issues in parallel algorithms is needed before using them. First, some measures are given for parallel performance. Then, an overview is given of the factors that contribute to the overhead in parallel search algorithms.

Parallel Performance

First of all, a measure is needed for the performance gain achieved by parallelising a given application over a sequential implementation. The *speedup* (S) captures the relative benefit of solving a problem in parallel by taking the ratio of the serial runtime T_S of the best sequential algorithm to the time T_P taken to solve the same

problem with P identical processors [39],

$$S = \frac{T_S}{T_P}. \quad (2.1)$$

The maximum speedup theoretically achievable is P (*linear speedup*), although *super-linear speedup* might occur in practise due to a non-optimal sequential algorithm or hardware characteristics. The *cost* (C) of solving a problem with a parallel system is the sum of the times that all processors spent solving the problem [39]. Cost is used for comparing the performance of parallel algorithms to their sequential counterparts, and is defined by

$$C = P \cdot T_P. \quad (2.2)$$

The *efficiency* (E) is another performance measure, very similar to speedup, which is the fraction of the time for which each processor is used usefully [39],

$$E = \frac{S}{P} = \frac{T_S}{P \cdot T_P} = \frac{T_S}{C}. \quad (2.3)$$

There are two ways for measuring performance of parallel algorithms. On the one hand, both the sequential and parallel runs can be done with a constant amount of memory, thereby measuring the scalability of the number of processors. On the other hand, the same amount of additional memory is used for every additional processor, thereby measuring the practical performance of distributed-memory systems.

Overhead

The performance depends on the overhead inflicted on a parallel algorithm during execution. The overhead of the parallel algorithm is the difference between its cost and the runtime of the fastest known serial algorithm for solving the same problem. The major sources that cause overhead in a parallel search algorithm are *communication overhead*, *search overhead*, and *synchronisation overhead* [22, 88].

Communication overhead is caused by the time taken to exchange information between processors, which obviously does not occur in sequential algorithms.

The search overhead (SO) is caused by the parallel algorithm expanding a larger search tree than the sequential algorithm [22]. When N_S represents the number of states expanded by the sequential algorithm and $N_{p(i)}$ is the number of states expanded by processor i , search overhead is defined by

$$SO = \frac{\sum_{i=1}^P \{N_{p(i)}\}}{N_S} - 1. \quad (2.4)$$

There are several causes for additional computation in parallel game-tree search. First, the parallel algorithm may use wider search windows than the sequential

search. Secondly, additional sub-trees are expanded in parallel that would have been pruned in the sequential algorithm.

Synchronisation overhead is caused by idling processors, because there is no work in the work queue or a processor has to wait at a synchronisation point. The most important factor in synchronisation overhead is the *load imbalance* (LI) [22], defined by

$$LI = \frac{\max_{i=1}^P \{N_{p(i)}\}}{\text{avg}_{i=1}^P \{N_{p(i)}\}} - 1. \quad (2.5)$$

The *granularity* is the number and size of basic tasks into which a problem is decomposed [39]. It therefore bounds the maximum level of concurrency and is of great influence to the overhead factors. A decomposition into a large number of small tasks is called *fine-grained* and a decomposition into a small number of large tasks is called *coarse-grained*. In parallel game-tree search, the remaining depth is used for expressing granularity and should be determined carefully. Generally, coarser-grained reduces communication overhead, but increases synchronisation and search overhead.

2.2.2 Splitting Strategies

An important issue in parallel two-player state-space search is the order in which states are expanded, and the manner in which these states are distributed. Previous work on this topic has resulted in numerous algorithms, that either belong to the *synchronous* or the *asynchronous* class [16].

The synchronous algorithms make use of the good move ordering in modern programs. In these well-ordered trees, the first move has a high probability of causing a cutoff in CUT-nodes. Therefore, these algorithms wait for the first move to be searched before initiating parallelism. *Principal Variation Splitting* (PV-Splitting) [68] makes use of this idea and calls itself recursively while traveling down the principal variation. Together with its successors (*Enhanced PV-Splitting*, *Dynamic PV-Splitting*, and *Dynamic Multiple PV-Splitting*) it suffers from much synchronisation overhead [16]. The *Young Brothers Wait Concept* (YBWC) [34] generalises the idea to split only at PV-nodes into splitting at all nodes.

Unsynchronised Iteratively Deepening Parallel Alpha-Beta Search (UIDPABS) [78] is the first attempt at an asynchronous algorithm in order to lose many synchronisation points. However, the resulting speedup was even worse than with PV-Splitting. *Asynchronous Parallel Hierarchical Iterative Deepening* (APHID) [17, 18, 19] is a generalisation of UIDPABS which outperforms YBWC in Othello and checkers, and achieves comparable results in chess. The biggest advantage of APHID is that no synchronisation overhead nor communication overhead occurs among the slave processors.

Young Brothers Wait Concept

The *Young Brothers Wait Concept* (YBWC) [34] is a splitting scheme for parallel two-player game-tree search. The scheme dictates a specific order in which sub-trees are to be expanded. Basically, it is the generalised version of PV-Splitting, in which the left-most move was expanded first only at PV-nodes. In YBWC, all nodes are handled equally and are expanded in two sequential phases, with the left-most sub-tree being searched before the other sub-trees (younger brothers).

In a perfectly ordered tree, the result returned by the first move causes the cutoff in a CUT-node. YBWC causes no search overhead when expanding a perfectly ordered tree, because YBWC always waits for the result of the left-most branch. Even if move ordering is near perfect, the first move will cause the cutoff in a CUT-node with high probability, while the first move is likely to narrow the search window when expanding a PV-node or performing a re-search.

Although YBWC causes relatively little search overhead in near perfect ordered trees, it still initiates parallelism too quickly at CUT-nodes when the first branch does not result in a cutoff. *YBWC** [32, 34] is a generalised version of YBWC. It uses domain dependent information to handle situations with multiple candidate cutoff moves before initiating parallelism. Furthermore, all nodes become synchronisation points when YBWC is applied at every node. Therefore, synchronisation overhead can hurt the performance of YBWC when not handled carefully. The number of synchronisation points is increased even further when iterative deepening is used. Because the degree of parallelism at synchronisation points is very limited, the waiting time must be compensated by doing the other branches in parallel. Synchronisation points become a serious problem when the average branching factor of the search tree is small.

*YBWC** was used in the chess program ZUGZWANG. It searched all moves in parallel at ALL-nodes in order to reduce the number of synchronisation points. DEEP BLUE searched all the moves of ALL-nodes in parallel as well [20]. However, a method for determining whether a node is an ALL-node is needed. The achieved efficiency of ZUGZWANG is 0.55 with 256 processors and 0.33 with 1024 processors, but these performance numbers are criticised [17]. The achieved efficiency by DEEP BLUE is reported to be approximately 0.30 on 256 processors [90]. Due to the success of YBWC, numerous variants have been tried, such as Dynamic Tree Splitting [88], Jamboree [63] and ABDADA [106].

2.2.3 Load Balancing

Load balancing for a parallel system is one of the most important problems which has to be solved in order to enable the efficient use of parallel computer systems. While the splitting strategy defines the expansion of the search tree, load-balancing schemes are used to distribute the work. The better the distribution process, the smaller the load imbalance and therefore smaller synchronisation overhead.

In synchronous parallel search algorithms, the load-balancing scheme relies on a

static processor tree [68] or on *work stealing* [88]. The work stealing scheme works by keeping a local work queue, implemented as a normal priority queue. Whenever a processor needs a new job, it first looks in its local work queue. If the local queue is empty, a processor is chosen at random from which it tries to steal a job to work on.

The asynchronous parallel search algorithms rely on the master processor to handle the load balancing for the complete system [17, 18, 19, 78]. In this scheme, workers are used for processing the work directly received from a master processor as in a master-slave relation.

2.2.4 Distributed Transposition Tables

In parallel algorithms, the transposition table can be used either non-shared (i.e., *local*) or shared (i.e., *replicated* or *partitioned*) [82]. First, it is possible not to share the transposition table at all, and only maintain a table locally. This approach does not need any communication overhead, but normally leads to a high increase in search overhead [82]. Second, the table can be replicated, making all transposition lookups local but all updates global [82, 88]. Drawbacks of this scheme are that it has a high communication overhead, because table updates cause large amounts of data being broadcast. Due to replication of the data, the additional available memory is effectively not being used, which causes search overhead to increase. Replication only performs well when there are just a few processors being used. Finally, a partitioned table can be used in which an equal part of the table is stored at each processor, such that there are no duplicated entries [32, 35, 82]. Updates can be done with asynchronous communication, but remote lookups need synchronous communication which results in high communication overheads even on low-latency networks [82].

2.3 Transposition-Driven Scheduling

Traditional distributed state-space searching algorithms distribute work over processors without caring to which processor it is sent. Work stealing is an efficient way of handling the distribution process, because it involves little communication overhead. However, when a transposition table is used, work stealing either results in significantly higher search overhead by using local tables, or in higher communication overhead when replicated or partitioned tables are used.

Transposition-Driven Scheduling (TDS) is an algorithm, originally developed for one-player search, that uses a partitioned transposition table for work distribution [86]. Instead of stealing work from other processors and doing remote table lookups, TDS migrates states to the processors where their transposition table entry is being managed. Although this reversed scheme looks expensive at first, it outperforms traditional work stealing algorithms by a wide margin when applied

to one-player search. TDS achieves near-linear speedups up to 128 processors for the *15-puzzle*, the *double-blank puzzle*, and *Rubik's cube* [82, 85].

2.3.1 The Algorithm

TDS takes the opposite approach on how to deal with transpositions as compared to the traditional approach of work stealing and remote synchronous lookups. Instead of performing work-stealing actions, every processor assigns work to other processors in a predefined way, which is integrated with the remote lookup mechanism.

Each state is uniquely assigned a *home processor*, which manages the transposition table entry for this state. The home processor is computed from the state signature (see Section 2.1.4). Then, the state is *asynchronously* sent to its home processor, where it will be processed.

TDS assumes that the state signatures are integer values that are distributed uniformly over their value domain, which is the case with Zobrist keys [107]. The home processor of a state is then simply calculated by taking the result of the state signature modulo the number of processors. Load balancing is done implicitly when the grain size is fine enough, because each processor has an equal probability of being the state's home processor. Part of the signature indicates the home processor number, while the remaining part is used to find its index in the transposition table at that processor. Therefore, it is possible to reduce the size per transposition table entry by only storing the part of the state signature that is not used for indexing.

Each processor manages part of the partitioned transposition table, and maintains a work queue. The work queue is a priority queue that contains the states that need expansion. As long as the queue is non-empty, the processor expands the next state to its successor states. After expansion, all child-nodes are sent to their home processors. Upon arrival the state is looked up in the transposition table. If the state is found in the table and was searched sufficiently, this result can be used immediately. If the state is not there, the entry is written to the local work queue.

TDS is a new approach in parallelising state-space search algorithms, and has proven to be efficient in parallel one-player search [86]. It has six advantages [85]:

1. All transposition table accesses are local.
2. All communication is asynchronous.
3. No duplicate searches are performed.
4. It efficiently uses the extra memory when more processors are added.
5. It produces more stable execution times for trees with many transpositions than the work-stealing algorithm.
6. No separate load-balancing scheme is needed on homogeneous systems.

Although these advantages sound very promising, the algorithm was originally developed for parallel one-player search. Research is needed to evaluate the applicability of TDS to two-player state-space searching, because a number of problems need to be resolved:

1. Backpropagation of the children's search results to the parent is needed, because its value depends on them.
2. A mechanism is needed to prevent searching transpositions concurrently.
3. The search windows in two-player search algorithms cause two problems:
 - (a) Narrower windows can become available after expansion.
 - (b) Transpositions of states can be searched with different windows.
4. A mechanism is needed to efficiently abort the search of sub-trees, which are likely to be spread over multiple processors.
5. The move ordering should not be disturbed too much, because it is of fundamental importance to the performance of two-player search algorithms.

2.3.2 Wide-Area Transposition-Driven Scheduling

Constructing an efficient parallel search algorithm for a parallel machine, even with fast interconnects, is a challenge. To make the algorithm run efficiently on a wide-area network that is characterised by high latency and low bandwidth is an even bigger challenge.

Wide-Area Transposition-Driven Scheduling (WA-TDS) [83] has been applied successfully to the one-player search algorithm IDA*. The required bandwidth is the main problem with TDS on multiple clusters interconnected with a wide-area network. By partitioning the search tree into sub-trees and distributing these over the clusters, TDS can be used per cluster on the sub-trees without wide-area remote lookups. Although this introduces a significant duplicated search overhead, it deals with the problems that are introduced by the wide-area connection. Remote lookups cannot be done over the wide-area network because this introduces too much communication overhead.

Experimental results [83] show that on wide-area interconnected cluster systems, WA-TDS performs much better than traditional distributed search algorithms. On a four-cluster system, where each cluster contained 16 processors, speedups were obtained that were 27-430% better than traditional work-stealing based algorithms, while the wide-area bandwidth per link was only a few KB/s, but it searched 38-90% more states than on a single cluster system with 64 processors.

Chapter 3

Design and Implementation of a Sequential Game-Tree Searcher

In this chapter, the sequential part of DARKSIGHT is described in detail. In Section 3.1, the global design of the sequential part of DARKSIGHT is given. In the three succeeding sections, detailed descriptions are given of the framework in Section 3.2, of the search algorithm in Section 3.3, and of the evaluation function in Section 3.4.

3.1 Global Design

The program in its basic form consists of three main modules:

1. The framework, which deals with board representation, move generation, and bookkeeping when making and unmaking moves.
2. The search, which deals with expanding a tree from the root position in order to find the best line of play.
3. The evaluation function, which deals with assigning a value to a position that represents the judgment of which side is stronger.

These three modules all have their own characteristics, but they are very much dependent on each other. The search module is the least domain-dependent one, although there are pruning heuristics, extensions, and reductions that are formulated in a domain-dependent way. Second, the framework module has an interface which is common for all state-space searchers. However, the function definitions differ completely from one domain to another. Finally, the evaluation function is completely domain-dependent, because the rules of the game implicitly define what should be valued in the evaluation function.

To speed things up, there are some gray areas in which two modules are implemented together. Strictly separating the modules is possible, but will result in a drastic speed reduction.

3.2 The Framework

One of the most disputed topics in computer chess newsgroups is the choice of a framework, because it is the core of the program. The choice of a specific framework is based on the possible speed of the move generator and the functions for updating the state. First, insight is given into DARKSIGHT's board representation. Thereafter, the move generator and the update routines are described.

3.2.1 Board Representation

DARKSIGHT uses normal bitboards [26], rotated bitboards [53] and an 8x8 board representation. The choice to use bitboards as the board representation is made because most people at the Computer-Chess Club forum [23] think it is the fastest framework available for computer chess, although it is a topic of ongoing debate. The 8x8 board representation is mainly used for quick lookups and board display, but whether maintaining and using the 8x8 board is worthwhile can be disputed. The bitboards are 64-bit vectors that have a bit for every field on the board. Every bitboard is used for giving a complete overview of the truth-value of a certain feature per square, for instance, whether the square contains a black piece. Representing a complete board is done by having one bitboard per piece per colour. The advantage of this bitboard representation stems from the fact that 64 is a power of 2, and that computers with a 64-bit architecture can do bit operations on the complete board in just one operation. Another advantage is that most operations needed for updating bitboards and using them can be formulated as fast bit operations. Another framework that is worth looking into, is the so called 0x88 board representation. There is very fast and short code available for move generation, but due to time constraints this has not been looked into.

3.2.2 Move Generator

The move generator is responsible for generating all possible state-transitions. Designing a move generator is a delicate task, because there are multiple issues to address:

- Speed: The average time needed for move generation.
- (Pseudo-)legal moves: Checking for legality is a relative expensive operation, making it worthwhile to postpone legality checking until it is needed.
- Multiple move generators: Promotions, captures, check-evasions, and normal moves are very different and can be optimised separately.
- Board representation: The board representation influences the speed of move generation and updating.

Optimal speed is needed in DARKSIGHT, and therefore pseudo-legal for capture and normal moves are generated separately.

		Time (sec.)	
d	$\text{perft}(d)$	DARKSIGHT	CRAFTY
1	20	0.00	0.00
2	400	0.00	0.00
3	8902	0.00	0.00
4	197281	0.04	0.04
5	4865609	0.99	0.92
6	119060324	25.47	24.27
7	3195901860	639.72	618.92

Table 3.1. Timed $\text{perft}(d)$ for DARKSIGHT and CRAFTY from the starting position.

3.2.3 Making and Unmaking Moves

Next to generating the pseudo-legal moves, it is needed to actually perform the transition. The bookkeeping includes updating the variables that are used in the state representation, such as the bitboards, the 8x8 board, castling rights, and en-passant field. Additionally, to speed up DARKSIGHT, some variables are updated incrementally, such as the state signature (see Section 3.3.1), numbers of types of pieces, material balance, and the positional scores for the pieces (see Section 3.4.1). Because bookkeeping and move generation together are responsible for the achievable expansion speed, they are a key characteristic. In Table 3.1, the raw expansion speeds of DARKSIGHT and CRAFTY are compared by $\text{perft}(d)$. This performance test is a well-defined full-width and constant-depth d search function for counting the number of expanded states. The results show that DARKSIGHT is comparable to the strong program CRAFTY, which can be partially explained by the fact that DARKSIGHT updates more variables. Additional tests have shown that the speed of DARKSIGHT nearly doubles when using a 64-bit instead of a 32-bit architecture.

3.3 The Search

DARKSIGHT uses PVS (see Section 2.1.3) as its main search routine, because it is one of the fastest algorithms available for searching game-trees and is suitable for parallelisation. The algorithm performs best when the move ordering is near perfect, which is the case for many modern chess programs such as DARKSIGHT. The search routine is enhanced with a two-level transposition table (see Section 3.3.1), dynamic move ordering (see Section 3.3.2), selective extension heuristics (see Section 3.3.3), and state-of-the-art forward-pruning heuristics (see Section 3.3.4). At the root, DARKSIGHT employs *Iterative Deepening* (ID) [62, 67] with a static *aspiration window* (see Section 5.1) of 0.4 of a pawn [94].

Description	Size in Bits
Hash key	64
Value	17
Draft	15
Bound	2
Best move	21

Table 3.2. Information in a transposition table entry.

3.3.1 Transposition Tables

DARKSIGHT uses separate transposition tables for white and black, to avoid problems with white-moves being returned from the table while it is black’s turn.

The state signature is constructed using Zobrist keys¹ [107], because they are fast and easy to implement. DARKSIGHT uses 64-bit random numbers as its keys, generated by a special purpose random generator that is only called in the initialisation phase. The state signature is updated incrementally in the routines for making and unmaking moves instead of being re-generated every time it has to be used.

The size of the table is chosen to be a power of two, determining an entry’s table index in just one single logical operation. The index of a certain entry therefore equals the last $\log(|entries|)$ bits from the state signature. The information contained in one single transposition table entry in DARKSIGHT is shown in Figure 3.2. Although highly unlikely when using 64-bit keys, it might occur that two different positions have the same state signature. DARKSIGHT therefore always checks the legality of the returned move before using it, which is done by a fast specialised routine.

Replacement Scheme There is a chance that different state signatures hash onto the same table entry. When such a hash collision occurs, a choice has to be made whether to replace or retain the position in the transposition table. When enough memory is available and the position needs to be stored, chaining or probing (linear or quadratic) can be used. In general, however, available memory is limited and choices need to be made which positions to keep in memory, which is governed by a *replacement scheme* [12, 13, 14, 31, 67].

DARKSIGHT uses the *two-deep* replacement scheme for handling hash collisions, which uses table entries that contain two positions each. When the new position has been searched deeper, the first entry overwrites the second entry and the new entry is put in the first, else the new position is stored in the second entry.

¹Most chess programs use random numbers for their Zobrist keys. However, the average and minimum Hamming Distances have to be maximised, because they represent the relative interdependency of the bit-vectors, or Zobrist keys. Using the math from coding theory and cryptography, there are ways to generate M numbers of length N that satisfy these constraints. These codes are called *Bose-Chaudhuri-Hochquenghem* (BCH) codes [105].

3.3.2 Move Ordering

Because DARKSIGHT uses an Alpha-Beta like search algorithm, it is of great importance to have a high degree of move ordering. Move ordering is measured by the probability of the first move considered being the move that causes a cutoff. It is, for example, better to sort the capture moves in a *most valuable victim/least valuable aggressor* (MVV/LVA) order [10], than have no predefined order.

DARKSIGHT uses the same mechanisms as many other chess-engines for move ordering and therefore orders them approximately in the same way. Additionally, the *Enhanced Transposition Cutoff* (ETC) [79] is incorporated. The rationale of utilising ETC is that it is relatively cheap and causes a reduction in tree size with its early cutoffs. The following list of stages gives a complete ordered description of DARKSIGHT's move ordering ($\approx 90\%$):

1. *Hash move*: When the current position is in the transposition table, but the depth is not sufficient to return a score directly, the transposition table supplies a move which is considered first [66].
2. *ETC captures*: After generating all capturing moves, all their succeeding positions are quickly looked-up in the transposition table. If one such capture results in a position with a score exceeding beta, that move is considered next, causing an early cutoff.
3. *Winning captures*: All captures are considered in most gain to most loss order using a *Static Exchange Evaluator* (SEE)². Only the captures with non-negative SEE values are the suggested moves in this stage.
4. *ETC normal moves*: After generating the remaining moves, the procedure of stage 2 is repeated.
5. *Killer moves*: Next, priority is given to the two moves from the killer table, which is maintained throughout the search to hold the two best moves on a certain depth in the search tree [1].
6. *History moves*: Thereafter, the two normal moves with the highest score in the history heuristic table are suggested [87, 89].
7. *Losing captures*: The remaining captures are returned in order of decreasing SEE value.
8. *Remaining moves*: All remaining moves are considered last in the static order in which they are generated by the move generator (castling, knight, bishop, rook, queen, king, pawn, and underpromotions).

²The SEE procedure returns the expected gain or loss of initiating the first capture in a exchange sequence on a specific square.

The effectiveness of trying the hash move first rises from the effect that the best move for the next iteration equals the current best move. Deep search experiments on *diminishing returns* in chess [48, 55] suggested that the probability that the resulting best move from a d -ply and a $(d - 1)$ -ply search differ decreases from 35 – 40% at $d = 2$ to 10 – 15% at $d = 14$ in state-of-the-art chess programs. In Section 5.2, new results are presented of two large scale experiments that achieve higher statistical significance and push the horizon from depth 14 to depth 20.

When a list of n moves has to be sorted based on their history value for instance, *insertion sort* is used. Although this algorithm’s complexity ($O(n^2)$) is worse than that of quicksort ($O(n \lg n)$) for instance, Figure 3.1 shows insertion sort to be the fastest algorithm near 35, which is a typical number of moves to be sorted.

In addition to the move ordering scheme just discussed, the technique of *Internal Iterative Deepening* (IID) [2] is applied when the position was not found in the transposition table and the null-move heuristic (see Section 3.3.4) was not applied. IID performs a 2-ply shallower search in order to come up with a good move suggestion, like Iterative Deepening.

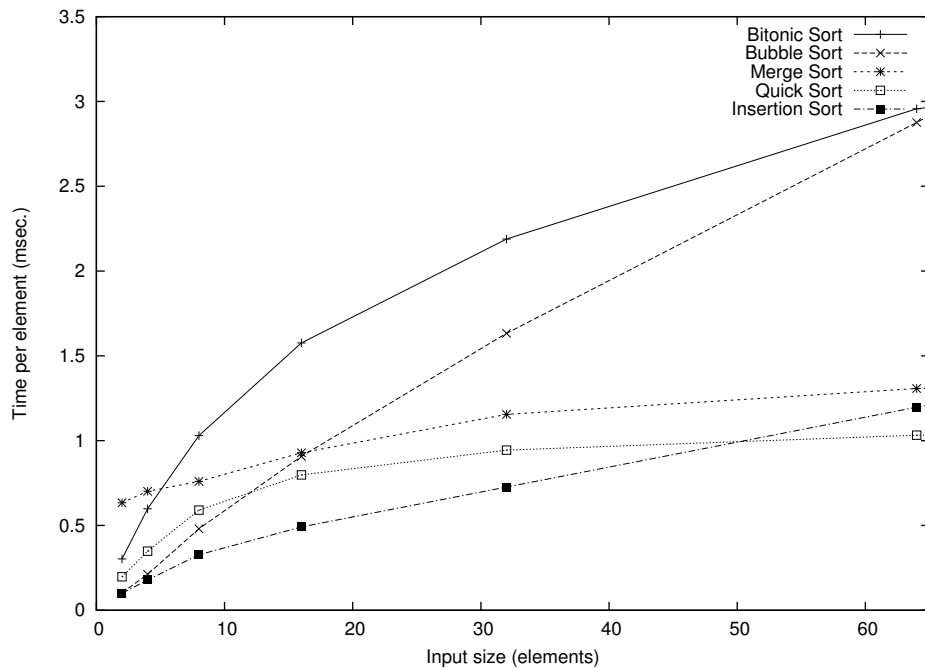


Figure 3.1. Performance of five different sorting algorithms.

3.3.3 Extension Heuristics

DARKSIGHT tries to implement all those techniques that have been reported to influence the playing strength in a positive way. However, search extensions should be applied carefully because they introduce the possibility of explosive growth of the search tree, which can be solved in multiple ways:

- Limit the search extension to at most one full ply extension per ply.
- Limit the application of search extensions to a part of the search tree.
- Fractional extensions, for extending the search with a fraction of one ply.

DARKSIGHT uses fractional extensions, which can be summed when multiple heuristics apply, and are limited to one ply per ply. The following six extension heuristics are currently incorporated:

1. *Check extension*: When a move puts the opponent in check, the number of counter moves is very limited in general [4, 65, 94].
2. *Deep search extension*: When a player's current score is greater than or equal to beta, but that the score would drop significantly when doing nothing [30].
3. *Pushed-pawn extension*: When a pawn is moved to its fifth, sixth, or seventh rank, it may be running towards promotion and is therefore extended [65].
4. *Recapture extension*: When pieces are being exchanged, moves are almost always forced and therefore do not need much time to be searched. A capture is considered a recapture when the previous move was a capture and the captured pieces differ up until half a pawn's value [4, 8].
5. *Single-reply extension*: When there is only one legal move in the current situation, the move is chosen instantly. Therefore, searching that move should not be considered a ply, resulting in extending the search a full ply [65].
6. *Mate-threat extension*: A player's king is under threat when doing nothing leads to being checkmated. It has been shown worthwhile to extend the search in situations where the player's king is under significant threat [2]. An alternative is to extend when a pseudo-mate threat is detected, which occurs when (after moving) the moving side's queen and at least one other piece or pawn of that side are attacking a square adjacent to the enemy king and if this was not true for the same square before this move. DARKSIGHT has incorporated the first alternative.

It remains future work to implement and test additional extension heuristics (i.e., singular extension [2, 3, 4, 65], Botvinnik-Markoff extension [23]) and *reduction heuristics* (i.e., fail-high reduction [33], backwards move reduction [65]).

3.3.4 Pruning Heuristics

Basically, there are two ways of pruning: Theoretically sound backward pruning as in Alpha-Beta search (see Section 2.1.2), and forward pruning that breaks with the full-width search paradigm. The following forward-pruning techniques are all unsound, which means that their outcome is not necessarily the correct value of the pruned sub-tree. This possibly changes the search result.

Null-move Pruning *Null-move* pruning is a dynamic pruning technique that fails-high when returning the right to move to the opponent and performing a shallow search results in a score larger than beta [6, 30, 37, 38]. The rationale of null-move pruning is that the right to move represents a value on its own. Although, this is working in the larger part of the game-tree, it does not hold in so called *zugzwang* positions in which the player to move can only reach a position that has a worse value than the present [30, 37, 38].

Verification Search can be used to solve the problem null-move pruning has in *zugzwang* positions [100]. However, the effectiveness in DARKSIGHT has yet to be verified.

In normal null-move pruning, a shallow search is performed with a depth reduction factor (R), which is normally a fixed 2 or 3 plies. Using $R = 3$ causes tactical performance loss, but reduces the number of nodes searched considerably compared to $R = 2$. The idea of *adaptive null-move* pruning is to combine the merits of the tactical save $R = 2$ and the unsafe aggressive pruning of $R = 3$ [45]. In DARKSIGHT, the following adaptive reduction factor R_{adpt} (dependent on depth d and the maximum number of pieces per side γ) is used instead of a fixed R :

$$R_{adpt} = \begin{cases} 2 & \text{if } (d \leq 6) \text{ or } ((d \leq 8) \text{ and } (\gamma < 3)) \\ 3 & \text{if } (d > 8) \text{ or } ((d > 6) \text{ and } (\gamma \geq 3)) \end{cases} \quad (3.1)$$

Futility Pruning The rationale of the futility-pruning mechanism is to prune whenever a situation occurs that seems completely futile. DARKSIGHT utilises three futility-pruning mechanisms: *Limited Razoring*, *Extended Futility Pruning* and *Normal Futility Pruning* [44]. They are applied in the given order but at different depths (d), and use different margins by which a position has to be futile:

1. **Limited Razoring:** The remaining depth is decreased by one ply, when exactly 3 plies remain to be searched and the position needs to improve at least a queen to be of influence to the outcome of the search.
2. **Extended Futility Pruning:** The estimated value of the current position is returned immediately, when exactly 2 plies remain to be searched in which at least a rook must be gained.
3. **Futility Pruning:** The estimated value of the current position is returned immediately, when exactly 1 ply remains to be searched in which an improvement is needed comparable to 4 pawns.

3.4 The Evaluation Function

The evaluation function gives a representative value for a specific position. Although a game like chess only has three theoretical values (win, draw, loss), the evaluation function should give a heuristic estimate. In Figure 3.2 the possible return values and their meanings are displayed and described, in which a loss is represented by $-M$, a draw by 0, a win by M , the evaluation value by E , and the number of plies by d .

$$\frac{-M + d \quad \parallel \quad -E \dots 0 \dots E \quad \parallel \quad M - d}{\text{Mated-in-}d \quad \parallel \quad \text{Eval. Scores} \quad \parallel \quad \text{Mate-in-}d}$$

with

$$-M + d < -E < 0 < E < M - d$$

Figure 3.2. The basic score bounds.

3.4.1 Patterns and Values

Patterns are distinguishable features of a chess position that are used to estimate the value of the position. Although DEEP BLUE II was able to detect over 8000 different patterns [20, 90], most chess programs detect fewer than 100 different patterns. All given evaluation weight values are in *centi-pawns* (cp).

Material Balance The material balance is the most dominant factor in the evaluation function. The rewards given by DARKSIGHT’s evaluation function for the mere existence per pieces on the board are listed in Table 3.3. These values are

Piece	Weight
Queen	$\omega_Q = 900$
Rook	$\omega_R = 500$
Bishop	$\omega_B = 300$
Knight	$\omega_N = 250$
Pawn	$\omega_P = 100$

Table 3.3. The values for pieces used in DARKSIGHT.

very close to the textbook values used for teaching people to play chess. Chess experts and statistical research have shown that certain combinations of pieces should be valued, irrespective of their location on the board [96, 104]. Table 3.4 shows the combinations of pieces that are recognised by DARKSIGHT, with pieces of the same colour (unless stated otherwise). Additionally, variable scoring of pawns $f(|P|)$ is used by rewarding a bonus dependent on the number of pawns on the

Piece	Combined with	Weight
Queen	Knight	$\omega_{QN} = +16$
Rook	Bishop	$\omega_{RB} = +10$
Rook	Knight	$\omega_{RN} = -6$
Rook	Pawn	$\omega_{RP} = -3$
Bishop	Bishop	$\omega_{BB} = +20$
Bishop	Pawn (of either side on the same colour)	$\omega_{BP} = -2$
Knight	Pawn	$\omega_{NP} = +10$

Table 3.4. The values for piece combinations used in DARKSIGHT.

board, with $0 \leq |P| \leq 8$ and $f(|P|) = (0, 50, 90, 120, 140, 150, 150, 140, 120)$. The values of ω_{QN} , ω_{RN} and ω_{NP} are based on [104]. The remaining values are taken from other chess-playing programs and were checked against the literature [96] when possible.

Piece Placement Although the material balance is very important, it does not yet say anything about the location of the material. In DARKSIGHT, *piece-square-tables* are used as guidelines for positioning of the material. The value per square takes into account the *mobility* and the *centralisation* of the piece on that square. The summed value for the complete board is updated incrementally in the routines for making and unmaking moves.

Closely related to the placement of pieces is the importance for development. It basically means that pieces have to be brought into play, instead of remain silent and inactively waiting in the back. In DARKSIGHT, the *Levy Development* function is used [40, 64].

Patterns per Piece Next to the general patterns discussed above, there are patterns for every piece separately that contribute to the evaluation value. Many of these patterns are described in the chess literature, but are mostly not defined precisely nor are given an estimated value. DARKSIGHT recognises the following patterns:

- Pawn structure³: Doubled/tripled (a-file: -10, b-file: -8, c-file: -8, d-file: -13), passed (+8, +16, +32, +60, +120, +160), connected abreast (+2), connected diagonally (+1), isolated (a-file: -8, b-file: -10, c-file: -12, d-file: -14), backward (-3), and blocked center pawns (-12).
- Bishop: Trapped (-175), outpost (+12), Fianchetto (+5), and mobility (+3).

³Because the pawn structure is a relatively static structure, the value is stored in a hash table which effectively has a hit-rate of 95 – 99%. This results in better pawn structure scoring almost for free.

	# of EGTBs:	EGTB (KByte)	W/D/L (KByte)
3-men EGTBs	5	62	225
4-men EGTBs	30	30.260	15.881
5-men EGTBs	110	7.372.374	1.030.333
6-men EGTBs	247/365	556.223.689	

Table 3.5. The memory consumption for perfect evaluation.

- Knight: Outpost (+12), blocks c-pawn (-10), king proximity (+3).
- Rook: On open file (+15), on semi open file (+5), on seventh rank (+30), connected (+5), battery (+8), king proximity (+2).
- King safety: On open file (-4), on semi open file (-2), next to open file (-2), castling bonus (king side: +15, queen side: +20), and crippled shelter (-5).

3.4.2 Endgame Tablebases

The use of perfect evaluation, by means of *Endgame Tablebases* (EGTBs), causes better evaluation results and hence better choices to be made by the search process. The idea behind these databases is that part of the game can be completely solved by using *retrograde analysis*⁴. The multiple metrics in use for infallible endgame play are *distance-to-conversion* (DTC) [102, 103], *distance-to-mate* (DTM) [46], *distance-to-zeroing* (DTZ), and the *distance-to-rule* (DTR) [41]. DARKSIGHT uses the Nalimov EGTBs [74, 75], which are a large set of compressed DTM tables. Although the use of this information results in better play, it needs a large amount of resources. To reduce the memory consumption *knowledgeable encoding* can be used, which only stores whether the position is won, drawn or lost instead of the DTM [46]. Table 3.5 shows the memory consumption of the available Nalimov EGTBs on disk (the 6-men are partially done), together with the memory consumption of knowledgeable encoding (W/D/L) representation of the same EGTBs.

3.4.3 Interior-Node Recognition

Interior-node recognition is a trivial but extremely powerful idea [43, 46, 52, 93]. The idea is to stop searching and cut the whole sub-tree when the game-theoretical value is known (win/draw/loss). Although this technique looks easy to implement, the integration into a search algorithm is highly complex:

- Checking availability and selecting the correct recogniser function efficiently is a challenging task, which is solved by using tables that are indexed by a *material signature*.

⁴Awari is solved by applying this technique [84].

Database	Knowledgeable Scoring for 4-Men Late Endgames (X=Q,R and Y=B,N/Z=X,Y,P/l=losing and w=winning)
$[KBB]_w[K]_l$	$ mbal + \frac{5}{2} - \frac{1}{4}corner_dist(K_l) - \frac{1}{8}dist(K_l, K_w)$
$[KBN]_w[K]_l$	$ mbal - \frac{1}{2} - \frac{1}{4}b_corner_dist(K_l) - \frac{1}{8}dist(K_l, K_w)$
$[KP]_w[KP]_l$	$4 + \frac{1}{4}adv(P_w) + \frac{1}{16}dist(K_l, P_w) - \frac{1}{8}dist(K_w, P_l)$
$[KPP]_w[K]_l$	$8 + \frac{1}{4}max_adv(PP) - \frac{1}{8}edge_dist(K_l) - \frac{1}{16}dist(K_l, K_w)$
$[KQ]_l[KR]_w$	$8 + \frac{1}{8}dist(K_l, Q_l) - \frac{1}{4}edge_dist(K_l) - \frac{1}{16}dist(K_l, K_w)$
$[KQ]_w[KR]_l$	$4 + \frac{1}{8}dist(K_l, R_l) - \frac{1}{8}edge_dist(K_l) - \frac{1}{8}corner_dist(K_l) - \frac{1}{8}dist(K_l, K_w)$
$[KX]_l[KP]_w$	$ mbal - 2 + \frac{1}{4}dist(K_l, X) + \frac{1}{4}adv(P) - \frac{1}{8}edge_dist(K_l) - \frac{1}{16}dist(K_l, K_w)$
$[KX]_w[KP]_l$	$ mbal - \frac{3}{2} + \frac{1}{8}dist(K_l, P) - \frac{1}{4}adv(P) - \frac{1}{8}edge_dist(K_l) - \frac{1}{16}dist(K_l, K_w)$
$[KX]_w[KX]_l$	$8 + 4is_Q(X) + \frac{1}{4}dist(K_l, X_l) - \frac{1}{8}edge_dist(K_l) - \frac{1}{16}dist(K_l, K_w)$
$[KX]_w[KY]_l$	$ mbal + 2 + is_Q(X) + is_RN(XY) + \frac{1}{4}dist([K]_l, Y) - \frac{1}{8}edge_dist(K_l) - \frac{1}{16}dist(K_l, K_w)$
$[KXZ]_w[K]_l$	$ mbal + 4 + \frac{1}{8}is_P(Z)adv(Z) - \frac{1}{4}edge_dist(K_l) - \frac{1}{16}dist(K_l, K_w)$
$[KY]_l[KP]_w$	$2 + \frac{1}{4}adv(P) + \frac{1}{16}dist(K_l, Y) + \frac{1}{16}dist(K_l, P) - \frac{1}{8}edge_dist(K_l) - \frac{1}{16}dist(K_l, K_w)$
$[KYP]_w[K]_l$	$ mbal + 4 + \frac{1}{4}adv(P) + \frac{1}{16}dist(K_l, P) - \frac{1}{8}edge_dist(K_l) - \frac{1}{16}dist(K_l, K_w)$

Table 3.6. Knowledgeable scoring for 4-men endgames.

$$\begin{array}{c}
-M + d \parallel -R \parallel -E \dots 0 \dots E \parallel R \parallel M - d \\
\text{Mated-in-}d \parallel \text{Recog.-Loss} \parallel \text{Eval. Scores} \parallel \text{Recog.-Win} \parallel \text{Mate-in-}d \\
\text{with} \\
-M + d < -R < -E < 0 < E < R < M - d
\end{array}$$

Figure 3.3. The accommodated interior-node score bounds.

- When a position is recognised as a win for one side, by using knowledgeable encoding, the engine needs an appropriate evaluation function to guarantee to make progress in playing out that recognised game-result. Examples of such functions are listed in Table 3.6 [46].
- The values of recognised positions should not interfere with mate and static evaluations, but still give good estimates of the recognised game-theoretical values. Therefore the original evaluation bounds were adjusted to accommodate recogniser values, in which a recognised loss is represented by $-R$, and a recognised win by R (see Figure 3.3).
- Scoring inconsistencies that are caused by incomplete recogniser coverage of sub-games of a position for which a recogniser exists, such as with pawn (under-) promotions.

These problems have been solved over the years. Furthermore, there are additional advantages to the use of interior-node recognisers. First, when exact recognition results cannot be given, recognition bounds can be used. Secondly, the time spent on evaluating positions is reduced in the case of successful recognition, because recognisers are normally much faster than heuristic evaluations. Thirdly, material signatures can be used for other important tasks in the static evaluation, to speed things up a little. Finally, it should be mentioned that the use of EGTBs can be

integrated easily, making extension of perfect play with new recognisers very easy when new EGTBs become available.

Although the problems have been solved and additional advantages have been uncovered, the big challenge remains to formulate interior-node recognisers for new configurations. The problem of constructing these recognisers is as hard as solving the sub-domain itself. While complete databases with perfect play information are growing exponentially, the size of recognisers will hopefully remain extremely small in comparison.

DARKSIGHT also utilises the recognisers in horizon-nodes and quiescence-nodes, instead of merely using them in interior nodes. Material signatures [43] are used for detection and selection of interior-node recognisers. The material signatures that are recognised are all 3- and 4-men late endgames [46], some 5-men late endgames [52], and pawn blockades [99].

3.4.4 Quiescence Search

Quiescence search is a dynamic evaluation function that is actually an aggressive forward-pruning search function that overcomes the problem of severe misjudgment of *unstable* states [5, 10, 57, 58, 91]. In an unstable position the static evaluation will change drastically due to a capture.

DARKSIGHT merely expands check evasions and winning captures that result in a material evaluation above a certain margin below α (Δ -pruning), and considers a situation to be stable when there are no such moves to consider.

Chapter 4

Design and Implementation of a Parallel Game-Tree Searcher

In this chapter, the parallel part of DARKSIGHT is described in detail. In Section 4.1, the global design of the parallel part of DARKSIGHT is given. In the three succeeding sections, detailed descriptions are given of the framework in Section 4.2, of the search algorithm in Section 4.3, and of how to determine the optimal granularity in Section 4.4.

4.1 Global Design

The parallel part of DARKSIGHT is an extension of its sequential counterpart, and consists of the following two modules:

1. The framework, which provides the necessary communication functionality and the supporting data structures.
2. The search, which is concerned with splitting, distributing and scheduling of jobs at processors.

These modules are implemented in a domain independent way, and are strictly separated from each other. However, the parallel search module depends on the parallel framework and the modules of the sequential search. As in the sequential situation, optimisations are possible by introducing domain dependency. For instance, programs that use application-specific code to run on the network interface have been reported to be up to 2.5 times as fast as those that use generic message-passing software [11].

DARKSIGHT is an interactive parallel program, which is either waiting for user input or performing a search. These two states have different demands on the functionality. Operations in the interactive part are not time-critical, while those in the search are. For convenience, one processor is assigned to be the master, which handles the interaction with the user and initiate the parallel search.

Message Type	Size in Bytes
New Job	296
Result	48
Abort	40

Table 4.1. The message sizes in DARKSIGHT.

4.2 The Framework

The parallel framework of DARKSIGHT provides an interface to the basic operations. Therefore, the used implementation is hidden and can be optimised separately. It provides the basic communication operations, and the needed functionality for keeping track of the work.

4.2.1 Communication

Often, more operations are needed in addition to the basic communication operations, such as the packing and unpacking of data. Therefore, an interface is used in DARKSIGHT to hide these additional function calls that are not essential to the operation's functionality.

The *Message Passing Interface* (MPI) [71] is used as the basis for the communication operations. Because message passing is a widely used paradigm and MPI is an established standard, its use is supported on most modern parallel machines. By using MPI it is likely that DARKSIGHT can run on most modern parallel systems. Although MPI is used, all communication operations are implemented as functions in the communication module and therefore can be implemented using another library while still providing the same interface and functionality. In this way, it is easy to incorporate the possible usage of other communication libraries, such as the *Parallel Virtual Machine* (PVM) [36], or even application-specific code that runs on the network interface [11].

The performance of parallel programs depends to a great extent on its communication. The message structure is therefore important to the overall performance. Table 4.1 shows all messages that are used in the search phase. The messages are relatively small, because they merely contain the necessary information.

To get better insight into the actual time needed for sending messages of different sizes on a specific architecture, a *ping-pong test* can be performed. This test basically gives the best-case performance on the time spent, as the sum of latency and transmission time, for a roundtrip of messages of different sizes. In Figure 4.1, the results are shown of an asynchronous ping-pong test on a system of Pentium-III machines interconnected by a Myrinet-2000 [72] network (i.e., the DAS-2 system).

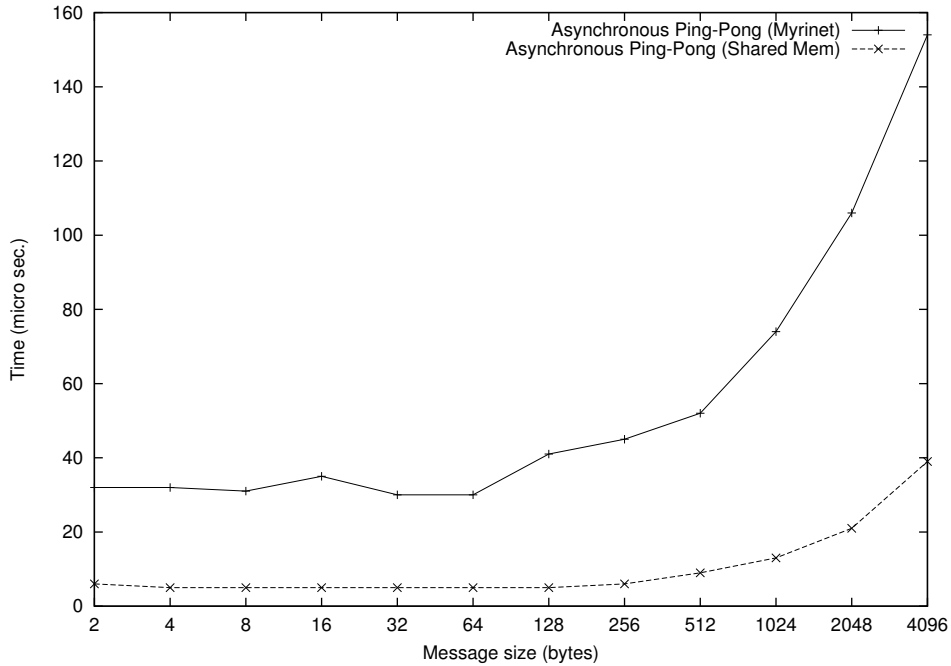


Figure 4.1. The asynchronous round-trip time in the DAS-2 system.

4.2.2 Supporting Data Structures

The three data structures described here support the parallel algorithm used in DARKSIGHT. First, a description is given of the system used to uniquely distinguish states in the parallel part of the search. Next, the implementation and use of the work queue, and of the busy list are described in detail.

Signatures

In the parallel part of the search tree, a system is needed to uniquely identify states. Although the system used for the hash keys (see Section 3.3.1) has a very small probability of two states having the same hash key, this risk is too big in the parallel part of the search tree. To overcome this problem, the *Dewey decimal system* [59] is used to assign coordinate numbers to the states. In this system, all states at level d are assigned a sequence of positive integers $a_1 a_2 \dots a_d$. The root position corresponds to the empty sequence and all w successors of a position $a_1 a_2 \dots a_d$ are assigned the sequences $a_1 a_2 \dots a_d 1, \dots, a_1 a_2 \dots a_d w$.

By using this system, the states in the parallel part of the search tree are uniquely identified by a sequence of integers. The sequence represents the move sequence that leads from the root position to the current state, when the index in a list of legal moves is used for every sequence entry. In a chess position, there are at most 219 legal moves possible [23]. Therefore, DARKSIGHT implements signatures as a small array of 32-bit numbers with each number containing four moves of the

move sequence. This identification system of states can be used in many ways which will be discussed in the following sections.

Work Queue

Incoming jobs need to be stored in a work queue and sorted based on their priorities. Therefore, the work queue has to be a *priority queue* into which work can be inserted, and from which the job with the highest priority can be extracted. In DARKSIGHT, this priority queue is implemented as a *binary heap* [25].

In addition to this data structure, a mechanism is needed that assigns a priority value to the jobs. This priority should be based on the location in the search tree. The sequential search order is assumed to be the perfect priority order.

The mechanism used in APHID [16] is the most commonly used, which assigns a priority to a state based on its move history. The categories and associated priorities used in APHID are PV-node (4), eldest-brother non-PV node (2), and the remaining nodes (0) [16]. The priority of a state is the summation of the associated priorities of the moves in its move history. Note that this mechanism causes multiple states to be associated with the same priority, for instance all permutations of a list of moves.

In DARKSIGHT, the signatures are used as priorities for the priority queue. A function is used that compares two signatures and returns which of the two comes first in the sequential search order (i.e., has the highest priority). This mechanism extracts the stored states in the sequential search order, which is assumed to be the optimal order.

Busy List

When a job is selected from the work queue, it has to be searched. Because the value of the selected state depends on the value of its children, their results have to be propagated back. Therefore, a busy list is maintained that contains all states that are currently being searched and waiting for results of their children. A *balanced binary tree* can be used for implementing such a busy list.

In DARKSIGHT, the busy list is implemented as a *red-black tree* [25], while the signatures are used as ordering criterion.

4.3 The Search

The parallel search is built on top of the functionalities of the sequential search modules. For instance, the move ordering used in the parallel search is the same as the one used in the sequential search (see Section 3.3.2). The basic search algorithm used in the parallel part is the PVS algorithm (see Section 2.1.3). In DARKSIGHT, the Young Brothers Wait Concept (YBWC) (see Section 2.2.2) and Transposition-Driven Scheduling (TDS) (see Section 2.3) are used for parallelising PVS.

In the following two sections, detailed descriptions are given of the YBWC in Section 4.3.1 and TDS in Section 4.3.2, as implemented in DARKSIGHT. In Section 4.3.3, the details are given of the implementation problems when applying all techniques from the sequential search (see Sections 3.3 and 3.4) in the parallel part, and the solutions used in DARKSIGHT.

4.3.1 Young Brothers Wait Concept

Efficiently parallelising an Alpha-Beta-like searcher like PVS is not trivial, because the efficiency of the algorithm largely depends on the sequential order. First, move ordering is of fundamental importance to the performance of the pruning mechanism of the PVS algorithm. Second, those children that are most likely to narrow the search window should be searched first, because search overhead is increased when larger search windows are used.

In DARKSIGHT, the YBWC is used as the balance between efficiently parallelising PVS while maintaining the sequential order. The same move-ordering scheme is used as in the sequential search (see Section 3.3.2), which has proven to cause a cutoff with high probability at CUT-nodes. When the first move does not cause a cutoff, DARKSIGHT searches the promising moves first (i.e., winning captures, killers and history moves), to reduce search overhead. In DARKSIGHT, YBWC is applied at all nodes, which causes an increased synchronisation overhead. It remains future work to apply the YBWC only at CUT-nodes.

4.3.2 Transposition-Driven Scheduling

The design of the original TDS algorithm [85, 86] is roughly maintained. The main loop of the parallel part consists of a phase in which messages are received and new work is put in the work queue, and a phase in which work from the work queue is done. Furthermore, the home processor of a position is determined by the state signature modulo the number of processors, to which it is sent asynchronously. This implicitly balances the workload when a sufficient amount of work is distributed, because each processor has an equal probability of being a state's home processor. Although the basic algorithm remains, the problems listed in Section 2.3 need to be solved before applying TDS to a parallel two-player state-space searcher. In DARKSIGHT, these problems have been solved in the following ways.

1. Backpropagation of children's search results to their parent is made possible by putting the expanded states in a busy list. The received result messages from the children are used to update the state's value, which is stored in the busy list (see Section 4.2.2). When all results are received and the state is searched completely, the state is removed from the busy list and its value is reported back to its parent.
2. Because transpositions are sent to the the same home processor, the concurrent searching of transpositions can be prevented. When a new state is a

transposition of a state in the work queue or busy list, which is not a predecessor, the new state is put in a waiting list. When a state has been searched that has transpositions waiting, the shallowest transposition is searched first.

3. The problems with search window sizes have been dealt with as follows.
 - (a) Because a child's search result can narrow the search window, the YBWC is used to wait for the child's result with the highest probability of narrowing the search window.
 - (b) Because transpositions of states can be searched with different search windows, only one transposition is searched at a time. When that transposition is searched, the transposition with the highest priority is selected and looked up in the transposition table in order to adjust the search window to the most recent information. This solution is equal to the adjustments done to the search window in the sequential search, in which transpositions are searched sequentially with possibly different search windows.
4. Although the YBWC often prevents searching sub-trees unnecessary, it needs to be possible to abort searching sub-trees which are likely to be spread over multiple processors. In the current version of DARKSIGHT, this is implemented as a broadcast and keeping a list of previously sent abort messages. However, letting an abort message propagate through the complete system, like the global state marker in an algorithm for determining the global state in distributed systems [21], may be more efficient.
5. The application of the YBWC is assumed not to disturb the move ordering too much, because the first move is still searched first.

4.3.3 Additional Search Techniques

PVS, YBWC and TDS together form the basis of the parallel searcher. However, when heuristics and forward-pruning techniques are not applied in the parallel part of the search, a totally different search tree is expanded. The sequential and parallel search cannot be compared when totally different search trees are expanded.

In DARKSIGHT, all additional search techniques have been implemented in the parallel search, as described in Chapter 3. Although this seems a trivial task, code needs to be inserted on multiple places in order to include all techniques. For instance, the null-move is searched before the first legal move, and special code was needed for handling the null-move search results. Furthermore, DARKSIGHT applies all extensions of Section 3.3.3 to the states in the parallel part of the search tree.

4.4 Determining the Granularity

As described in Section 2.2.1, the granularity used in a parallel algorithm is crucial to its performance in practice. The granularity should be chosen carefully, dependent on the demands of the algorithm (i.e., TDS) and the hardware characteristics of the used system. First, TDS needs a granularity that is sufficiently small such that load balancing is done implicitly (see Section 2.3). However, choosing a grain size that is too fine results in too high search and communication overhead. Secondly, the system, on which the algorithm's performance is tested, is of influence to the best granularity. For instance, changing a fast network into a slow one generally means that a larger granularity performs better.

Although in chess the number of plies is commonly used, granularity can be defined either as the number of remaining plies or as the number of searched plies. Whether the granularity is fine enough depends on the number of processors, because more processors simply need more jobs. However, a trade-off should be made between good load balance by TDS and search speed. Speedup is still the success factor of the algorithm, not the optimal load balance. The grain size on a specific system needs to be chosen dependent on the number of processors and the requested search depth (see Figure 4.2). Better speedups may be achieved by a coarser grain size which reduces the communication overhead, while limiting both synchronisation and search overhead.

In DARKSIGHT, the remaining depth is used for expressing granularity and should be determined carefully for the specific hardware it is run on. The results of determining the best granularity on the DAS-2 is described in Section 5.3.

Some future work on determining an even better granularity is to differentiate between null-window searches and full-window searches. This differentiation probably results in better performance results, because the average time needed for searches to a depth of d ply differs strongly.

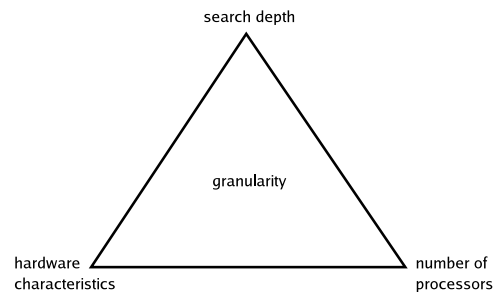


Figure 4.2. The factors of influence to the optimal granularity value.

Chapter 5

Experiments in Game-Tree Search

In this chapter, three experiments are described that zoom in on three different topics in two-player state-space search. In Section 5.1, the results are presented of the aspiration window sizes that are needed in the chess program CRAFTY. In Section 5.2, new results of the deep search experiment with CRAFTY are presented. In Section 5.3, the performance results of transposition-driven scheduling are presented, as applied to our parallel chess program DARKSIGHT.

5.1 The Aspiration Window Experiment

Alpha-Beta-like algorithms search within a *search window* $[\alpha, \beta]$. They guarantee to give the correct value when called with the full window $[-\infty, \infty]$. However, when the search result is outside the search window a re-search is needed, which is costly and reduces the efficiency. The search window tightens as the search progresses, which results in higher cutoff rates. We assume that the value of a search to depth d (V) can be used as an approximation of the search to depth $d + 1$. A margin (ϵ) can be used as the assumed maximum difference of two search values to succeeding depths, with high probability. Calling the search algorithm with an *aspiration window* $[V - \epsilon, V + \epsilon]$ results in higher cutoff rates compared to searching with a full window, and therefore in smaller search trees [66, 67, 94].

Although it is essential to choose ϵ with great care, to our knowledge no study has been published on the required aspiration window sizes (ϵ). The goal of this aspiration-window experiment is to gain insight into the differences between the search values of a search to depth $d + 1$ compared to search values to depth d for the chess program CRAFTY. The results indicate that search values differ less with increasing d , which implies that smaller aspiration windows are needed with increasing d .

5.1.1 Experimental Setup

Our experiment was run from September 2003 till March 2005. The chess program CRAFTY version 19.6 was used with a 768 MByte transposition table and a 24 MByte pawn hash table, equipped with all 3-4-men EGTBs with 1377 KByte for indices and decompression tables, and no openingbook. Fourteen stand-alone computers were used (3.06 GHz Intel Pentium 4, 512 KByte cache, 1 GByte RAM). The chosen test positions are the first 3800 positions of the *Encyclopedia of Chess Openings* (ECO), obtained from `ftp://darkside.its-s.tudelft.nl/Test-Suites/ECO.epd`. These positions were spread over the available computers and searched to a depth d of 18.

5.1.2 Results

From the experimental results, a simple program is used to extract the search values for depths d from 1 until 18. Next, we determine the required aspiration window sizes (ϵ) for the searches to $d + 1$, without a re-search being needed. This is done by taking the difference of the search value of the search to depth d and the value of the first sub-tree of the search to a depth of $d + 1$. The resulting density of the sufficient values of ϵ for different values of d is shown in Figure 5.1. In Figure 5.2, the cumulative distribution is given for the same data.

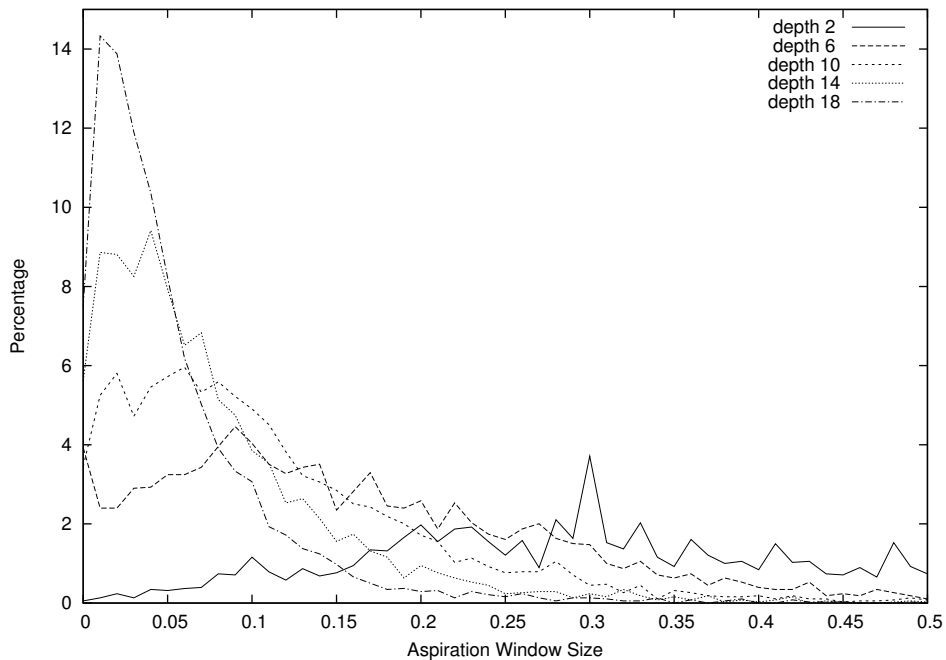


Figure 5.1. The density of the aspiration window sizes.

It can be observed from Figure 5.1 that the mass moves towards the smaller aspiration window sizes with increasing depth. In Figure 5.2, the same effect is visible with the lines growing faster with increasing depth. The program used in this experiment, CRAFTY, uses a constant-size aspiration window of 0.40. Because smaller window sizes result in smaller expanded search trees and therefore faster searches, it would be better to use smaller values of ϵ (aspiration window sizes) at increasing depths. The value of ϵ at depth d should have an acceptable risk for having to do a re-search, based on the cumulative distribution. What this acceptable risk is, remains future work.

In Figure 5.2, it can be observed that the cumulative distributions for depth d and $d + 1$ are comparable, for uneven d with $d > 2$. This may suggest that the accuracy of search values of CRAFTY are not increasing continuously with d .

Note that these results were obtained with one specific program, and may therefore be different when another program is used. It remains to be seen how these results differ when using different programs. Also note that all test positions are taken from the opening phase. It remains to be seen whether the results are different when the aspiration-window experiment is repeated with middlegame or endgame positions.

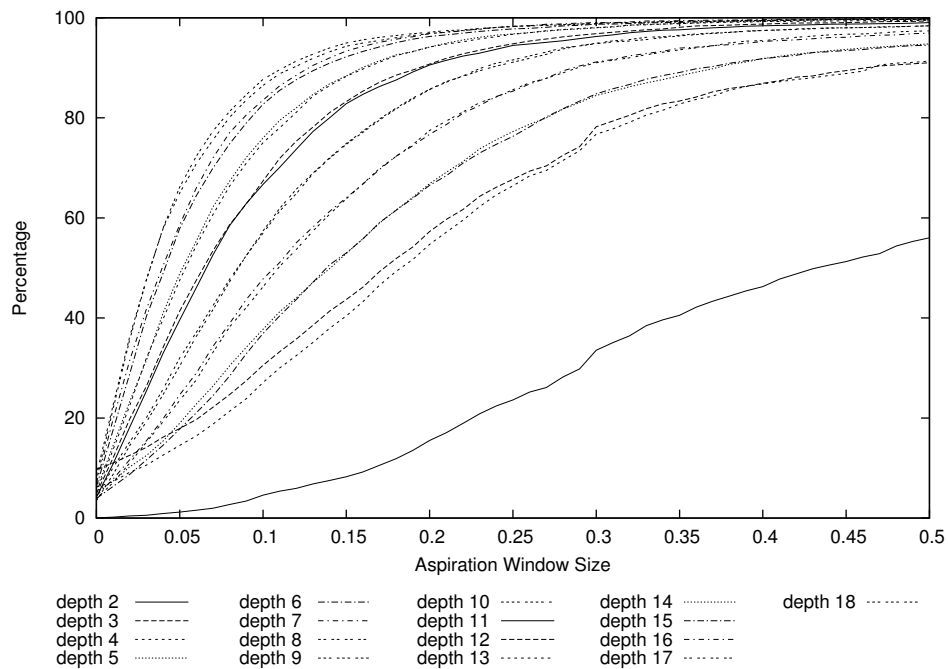


Figure 5.2. The cumulative distribution of the aspiration window sizes.

5.2 The Deep Search Experiment

Although searching more deeply promises to achieve stronger play, the relation between search depth and playing strength is not exactly known. In this section, new results are presented on the behaviour of Alpha-Beta-like searchers when searching to increasing depths. These results provide answers to questions raised in the literature on the relation between search depth and playing strength with higher level of confidence than ever presented before.

5.2.1 Background

The research on the relation between search depth and playing strength started in the early 1980s with *self-play* experiments [24, 101]. Playing strength was reported to increase almost linearly with search depth, which was confirmed by other researchers using different programs [9, 56, 73, 98]. Unfortunately, these self-play experiments could not quantify the differences in playing strength with statistical confidence [48, 50, 73]. New self-play experiments showed the existence of diminishing returns for additional search with 95% statistical confidence [49, 51].

When $B(d)$ denotes the best move of iteration at depth d , the *best-change rate* $BC(d)$ is defined as the probability that $B(d) \neq B(d - 1)$. Newborn [77] discovered that results of self-play experiments are closely correlated with $BC(d)$, and he was able to formulate the following hypothesis that predicts the increase in playing strength for searching to $d + 1$, $RI(d + 1)$:

$$\text{Newborn's Hypothesis: } RI(d + 1) = \frac{BC(d+1)}{BC(d)} \cdot RI(d).$$

The *go deep* experiment was introduced for determining $BC(d)$ for higher values of d . Hyatt and Newborn [55] used CRAFTY and showed that $BC(d)$ decreases, but stabilises at 15 – 17% at higher search depths. Heinz repeated their go deep experiment with his chess program DARKTHOUGHT and reported similar results [42]. However, a drop at the end of the data curve of DARKTHOUGHT raised the question whether this was a trend or a mere fluctuation. After applying the least squares method to fitting several functions, Heinz found that the constant function was the best approximation for the plies 11-14 when checking against the extrapolation of the model to depth 16 [47]. Furthermore, Heinz introduced a new hypothesis, which links the best-change rate and the *fresh-best rate* $FB(d)$, being defined as the probability that $B(d) \neq B(j)$ for all $j < d$:

$$\text{Heinz's Hypothesis: } \frac{BC(d+1)}{BC(d)} \approx \frac{FB(d+1)}{FB(d)}.$$

Although much research has been done, $BC(d)$ and $FB(d)$ need to be quantified for $d \geq 14$, with high level of statistical confidence. An answer is needed whether $BC(d)$ remains constant or continues to decrease. Furthermore, Heinz's hypothesis remains to be proven.

5.2.2 Experimental Setup

The experimental setup is equal to the one described in Section 5.1.1. Additionally, the original set of positions used by Hyatt and Newborn (and by Heinz) was included. The original set of positions was obtained from `ftp://ftp.cis.uab.edu/pub/hyatt/plytest/positions.gz` and corrected according to [42]. In addition to those corrections, position #163 was removed because it has only one legal move, leaving 342 remaining test positions.

The 342 original positions were spread over the available computers and searched to a depth of 20, while the set of 3800 opening positions was searched to a depth of 18. The original test positions have not all been searched to $d = 20$, for two reasons. One, the set contains positions with mating sequences that are shorter than 20 plies. Two, it took CRAFTY too long to complete the search in some positions in-between power shortages or system reboots. For the ECO test set, there was only one position unable to finish searching to depth 18 within two months. Only the missing results of these incomplete searches are not included.

5.2.3 Confidence Bounds

Although the results are based on 342 or 3800 samples, they are not necessarily the exact values. Therefore, lower and upper bounds need to be defined on the true values with high level of statistical confidence. Assuming a normal distribution, bounds can be given by $\bar{x}_n(d) \pm z_{\%} \cdot SE(\bar{x}_n(d))$ for any level of statistical confidence, with n the sample size. Here, $\bar{x}_n(d)$ represents the average value, and $z_{\%}$ denotes the upper critical value of the normal distribution $\mathcal{N}(0, 1)$ ($z_{95\%} = 1.96$). The standard error is given by $SE(\bar{x}_n(d)) = \sqrt{\bar{x}_n(d) \cdot (1 - \bar{x}_n(d)) / n}$, with $\bar{x}_n(d)$ being either $BC(d)$ or $FB(d)$.

After applying this theory to obtain actual numbers for the experimental results, decreasing standard errors are observed. For the original test set, the standard error of $BC(d)$ decreases from 0.0265 ($d = 2$), 0.0184 ($d = 14$) to 0.0169 ($d = 20$), while for the ECO test set the standard error changes from 0.0054 ($d = 2$), 0.0059 ($d = 14$) to 0.0051 ($d = 18$). Compared to the standard errors of the previous go deep experiments of Hyatt and Newborn with 0.0263 ($d = 2$) to 0.0195 ($d = 14$), and Heinz with 0.0258 ($d = 2$) to 0.0186 ($d = 14$), the results from the ECO test set are much more reliable. Comparing the standard errors for $FB(d)$ gives a similar result.

5.2.4 Results

The values for $BC(d)$ and $FB(d)$ are extracted from the search results by a small program. In Figures 5.3 and 5.4, the new $BC(d)$ and $FB(d)$ results are shown for the original set, CRAFTY (2004), and the ECO test set, ECO (2004). The previous results of Hyatt and Newborn [55], CRAFTY (1997), and Heinz [42], DARKTHOUGHT (1998), are also provided for comparison. A complete numerical overview of all data, including standard errors, is given in Appendix A.

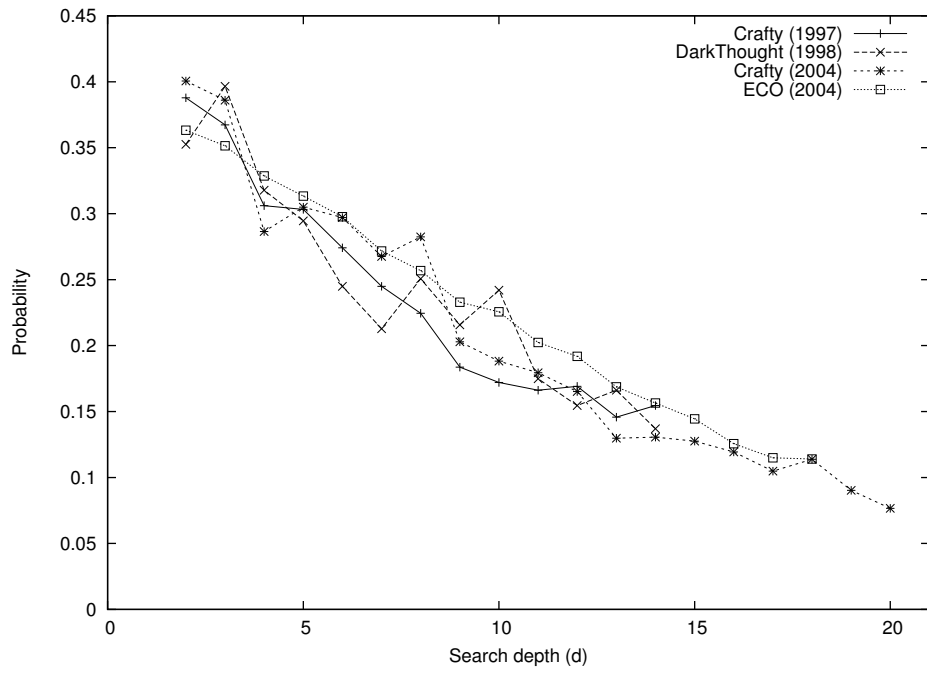


Figure 5.3. The best-change rates of four deep search experiments.

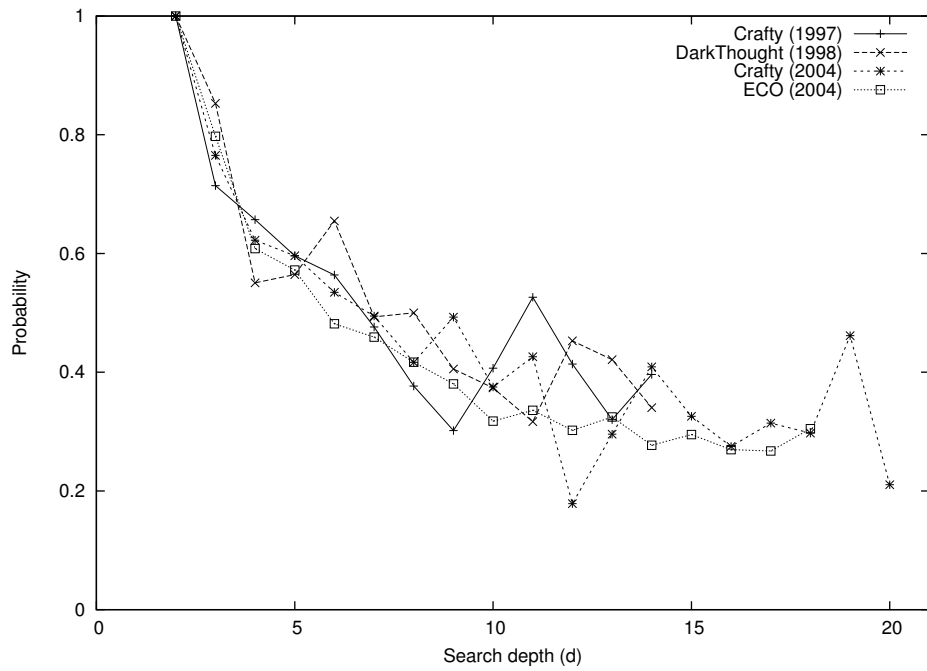


Figure 5.4. The fresh-best rates of four deep search experiments.

As expected, the best-change and fresh-best values closely resemble previous results. Note that slightly different results are achieved by using different versions CRAFTY on the original set of positions, with the versions used in 1997 and 2004, respectively. Looking at the results beyond $d = 14$, it can be observed that $BC(d)$ continues to descend. Therefore, $BC(d)$ does not remain 15–17% at higher search depths [55], and DARKTHOUGHT’s drop at the end of the curve is likely not to be a mere fluctuation but a trend [42].

Figure 5.5 shows the $BC(d)$ and $FB(d)$ from the ECO test set with the 95% confidence bounds, together with the result of applying least squares fitting of an exponential function. Both functions fit the data remarkably well, and it is therefore clear that the suggested constant function is not the best fit to the $BC(d)$ data [47]. However, the $FB(d)$ does seem to stabilise at 25 – 30%.

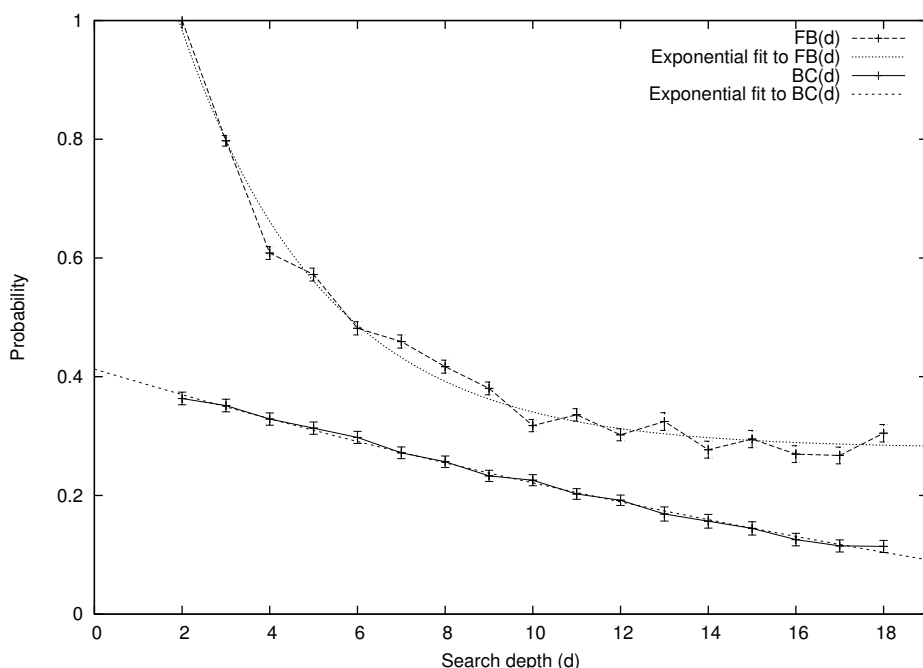


Figure 5.5. Least squares fitting to the best-change and fresh-best rates of CRAFTY with the ECO test set.

Finally, Figure 5.6 gives some insight into the correctness of Heinz’s hypothesis by using the results obtained with the ECO test set. It shows $\frac{BC(d+1)}{BC(d)}$ and $\frac{FB(d+1)}{FB(d)}$ together with the 95% confidence bounds, which were obtained by taking ten-thousand bootstraps for every data point [27]. Unfortunately, no conclusions can be drawn from this figure concerning Heinz’s hypothesis. The figure, however, does show that most of the values are below 1, which hints at diminishing returns when the provided ratios are correct estimators of the increase in playing strength.

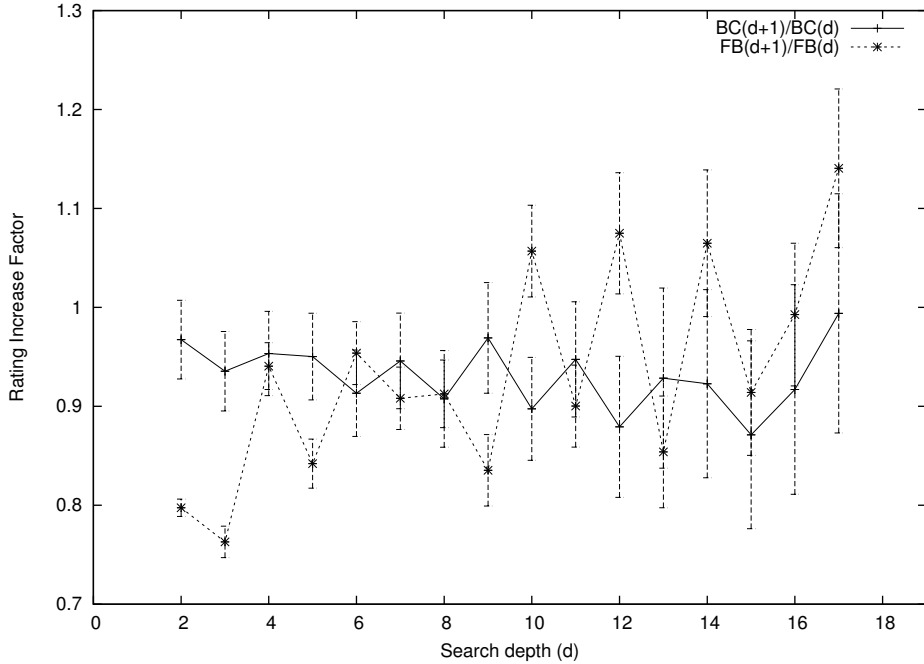


Figure 5.6. Bootstrap results of $\frac{BC(d+1)}{BC(d)}$ and $\frac{FB(d+1)}{FB(d)}$ with 95% confidence bounds for CRAFTY with the ECO test set.

5.3 Speedup Experiments with DARKSIGHT

The application of TDS to a one-player state-space searcher has resulted in near-linear speedup [86]. However, applying TDS to a two-player state-space searcher introduces a number of problems (see Section 2.3). In this section, we present the speedup achievable by our parallel two-player state-space searcher, DARKSIGHT, in which TDS is used for load balancing.

The design and implementation of the sequential and parallel part of DARKSIGHT are described in Chapters 3 and 4, respectively. The solutions to the problems of applying TDS in a two-player state-space searcher, as used in DARKSIGHT, have been described in Section 4.3.2.

Two things should be mentioned concerning the achievable speedup. First, the achievable speedup depends on the granularity used in the parallel algorithm, optimal value of which needs to be determined (see Section 4.4). Second, the best move in a certain state is not as well defined as the correct output of a sorting algorithm. Selecting the best move in a state depends on many things, such as on the expansion order, or on the order in which result messages are received. A different order of receiving results effectively expands a different search tree, sends different results up the search tree, and searches a different number of states. However, a set of positions is used to minimise this effect.

5.3.1 Experimental Setup

The achievable performance of DARKSIGHT was measured on the second-generation *Distributed ASCII Supercomputer* (DAS-2) in May 2005. The DAS-2 is a multicluster containing five clusters, each of which is located at one of the following Dutch universities: Vrije Universiteit (72 nodes), Leiden University (32 nodes), University of Amsterdam (32 nodes), Delft University of Technology (32 nodes), and University of Utrecht (32 nodes). The clusters are interconnected by SurfNet [97], the Dutch university backbone for wide-area communication (currently 10 Gbit/s bandwidth). Nodes within a local cluster are inter-connected by a Myrinet-2000 LAN with 1200 Mbit/s bandwidth [72]. Additionally, Fast Ethernet is used for the operating system and file transport. The DAS-2 is a homogeneous multicluster (each node has approximately the same specification) with the following specification per node:

- Two 1 GHz Intel Pentium III processors, 256 KByte cache.
- 1 GByte RAM except 1.5 GByte for Leiden University and University of Amsterdam, 2 GByte for Vrije Universiteit.
- A 20 GByte local IDE harddisk (80 GByte for Leiden University and University of Amsterdam).
- A Myrinet-2000 network interface.
- A Fast Ethernet network interface (on-board).

DARKSIGHT was used with a 64 MByte transposition table and a 4 MByte pawn hash table. The 24 positions of the well-known Bratko-Kopec test set were used (see Appendix B).

Because the best granularity values for the parallel algorithm are not known, they need to be determined in this experiment (see Section 4.4). Therefore, every position is searched to a depth of 10 ply multiple times, for all combinations of the numbers of processors and the granularity values. Although an arbitrary number of processors can be used by DARKSIGHT, we ran the parallel tests on 2, 4, 8, 16, and 32 processors. Four granularity values for the parallel algorithm have been tested thoroughly (granularity values 4, 5, 6, and 7). All performance results are collected at every depth, for every combination of processor count and granularity value. The results per processor count with the largest speedup determine the achievable speedup of DARKSIGHT.

For running all the jobs on multiple sites of the DAS-2, the KOALA Co-Allocating Grid Scheduler [69, 70] has been used. This caused the total experiment to be completed within 4 hours, which can easily be repeated with a new version of the program.

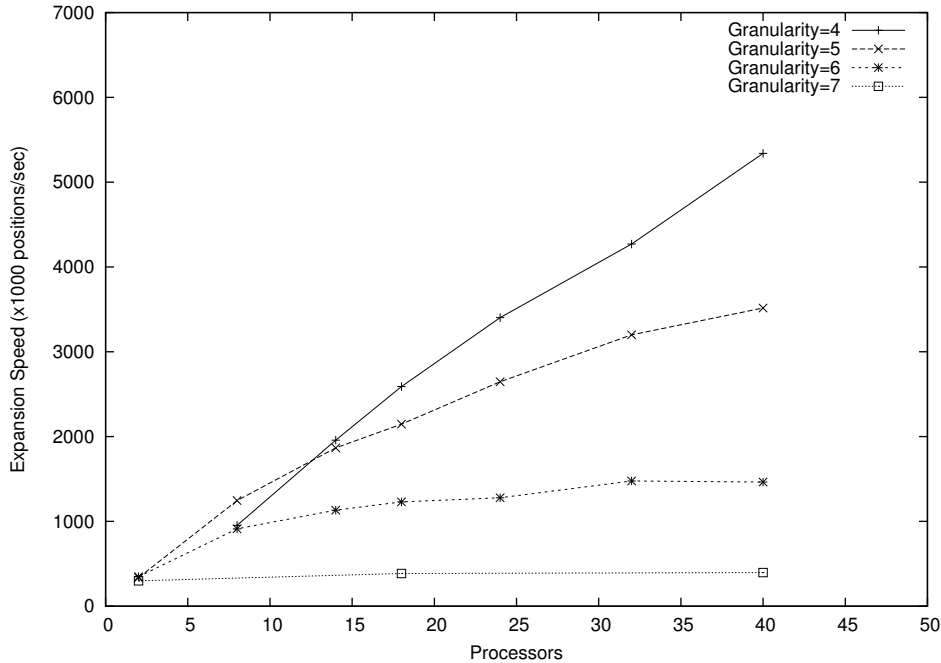


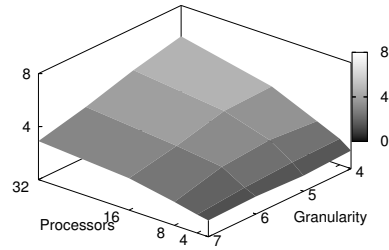
Figure 5.7. Expansion speedup results for different granularity values.

5.3.2 Results

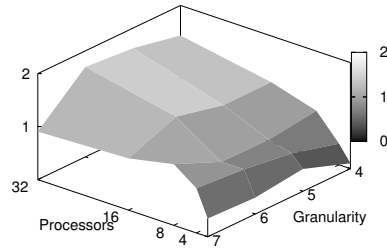
Figure 5.7 shows the speedup of expanding positions when searching to a depth of 8 ply, for different granularity values. These results show that the expansion speed increases until a certain maximum is reached. When the granularity is decreased, the maximum level of parallelism increases and therefore a higher maximum expansion speed can be reached. Increasing the level of parallelism results in an increased communication overhead, which has a negative impact on the expansion speed.

An increasing expansion speed is merely a prerequisite for achieving speedup. The actual speedup of a state-space searcher depends on search overhead, communication overhead, and synchronisation overhead (see Section 2.2.1 for definitions). A small program is used to extract these overhead values, the speedup, and the load imbalance from the search results. All extracted values are the averaged values from all positions of the test set used. The sequential reference values are taken from a sequential run of DARKSIGHT on the same test set.

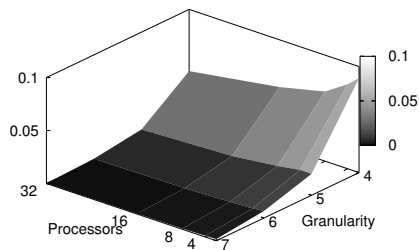
Performance results differ when searching to different search depths. As an example, Figure 5.8 gives an overview of the performance results achieved at a search to a depth of 10 ply. This figure shows that the achieved performance depends on the number of processors and on the granularity used. Because we want to achieve maximum speedup, the best granularity value is selected for every number of processors that gives the largest speedup (see Figure 5.8(a)). Table 5.1 shows



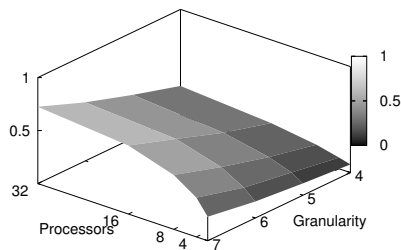
(a) Speedup



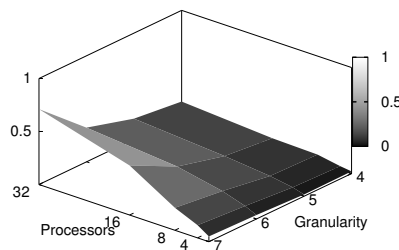
(b) Search Overhead



(c) Communication Overhead



(d) Synchronisation Overhead



(e) Load Imbalance

Figure 5.8. Parallel performance results for DARKSIGHT searching to a fixed depth of 10 ply.

Number of Processors	Granularity	Speedup	Search O. (%)	Synch. O. (%)	Comm. O. (%)
2	6	1.33	31.1	15.6	0.7
4	5	1.91	64.5	18.0	2.0
8	5	3.02	76.1	25.3	1.7
16	4	4.19	112.9	21.6	5.6
32	4	5.67	143.3	27.8	4.2

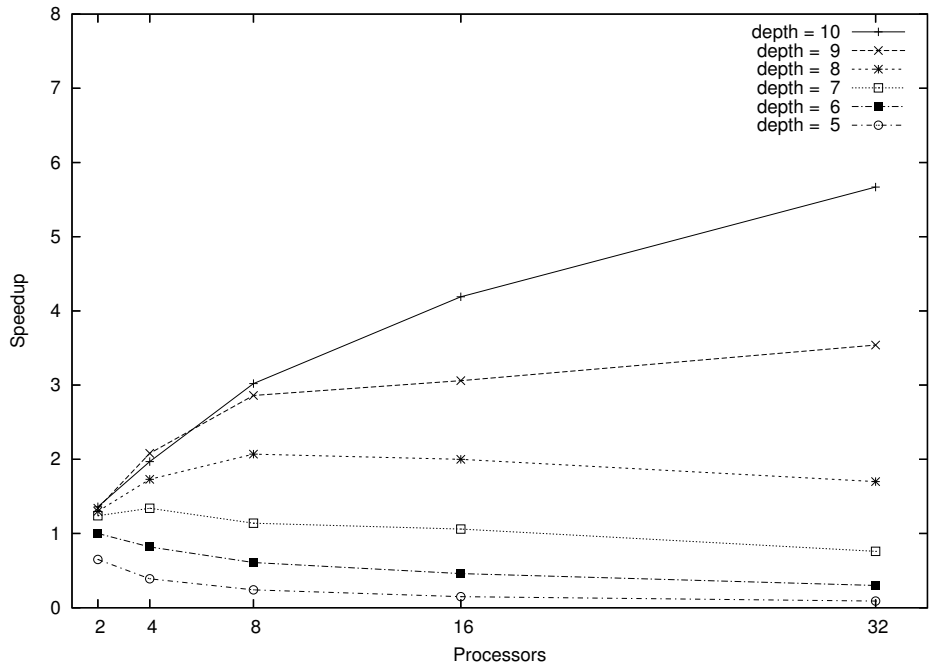
Table 5.1. Speedup results for DARKSIGHT searching to a fixed depth of 10 ply.

the achieved speedup results by DARKSIGHT at a search to a depth of 10 ply, after choosing the optimal granularity value. We have determined the optimal granularity values for the parallel algorithm for the used processor counts, and for search depths up until 10 ply. The achievable speedups are shown in Figure 5.9.

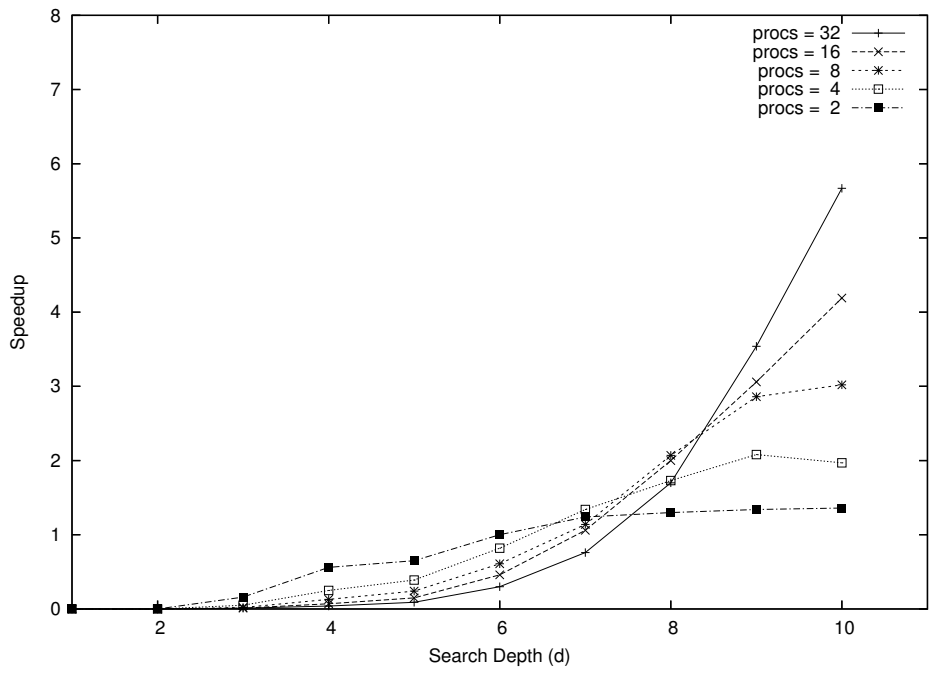
The results show that the speedup increases with the search depth in DARKSIGHT until a certain maximum is reached. The maximum achievable speedup increases with the number of processors used (see Figure 5.9(a)), but a larger number of processors needs a larger search depth before reaching its maximum speedup (see Figure 5.9(b)). The speedup is not linear due to non-negligible search, synchronisation, and communication overhead. Due to a relative large communication overhead, the speedup drops below 1 for too shallow search depths, and when too many processors are used. The communication overhead is relatively small, compared to the synchronisation overhead, on larger numbers of processors. The search overhead increases rapidly, which is the main problem of the current implementation of DARKSIGHT. Therefore, the problem is to get the processors to work instead of waiting for incoming jobs, without doing too much unnecessary work. Despite these problems, these preliminary results look promising for future improvements. Searching to depths beyond 10 ply promises to achieve larger speedups than presented here.

Others have tested other algorithms, like APHID and ABDADA (see Section 2.2.2), using different chess programs. A 13.5-fold speedup with CRAFTY, and an 11-fold speedup with THETURK was achieved in a search to a depth of 11 ply, using APHID on 32 processors of the shared-memory SGI Origin 2000 (on 64 processors the speedups were 15 and 14, respectively) [17]. The reported speedup with ABDADA search was 16 on 32 processors [106], and the speedup achieved with Multigame was 28 on 64 processors of the DAS-2 [82]. However, comparing reported speedups is hard because the results depend on factors like hardware characteristics, implementation details, test set used, and the sequential counterpart.

Although the performance results of TDS are not as good, they are achieved with a rapidly increasing search overhead. This search overhead is to a great extent caused by a known but unsolved problem with the basic Alpha-Beta pruning mechanism in the parallel part of the search tree. Solving this problem will improve performance significantly, but remains future work. It is expected that the communication and



(a) Speedup results for increasing numbers of processors.



(b) Speedup results for increasing search depth.

Figure 5.9. Achieved speedups by DARKSIGHT at different search depths.

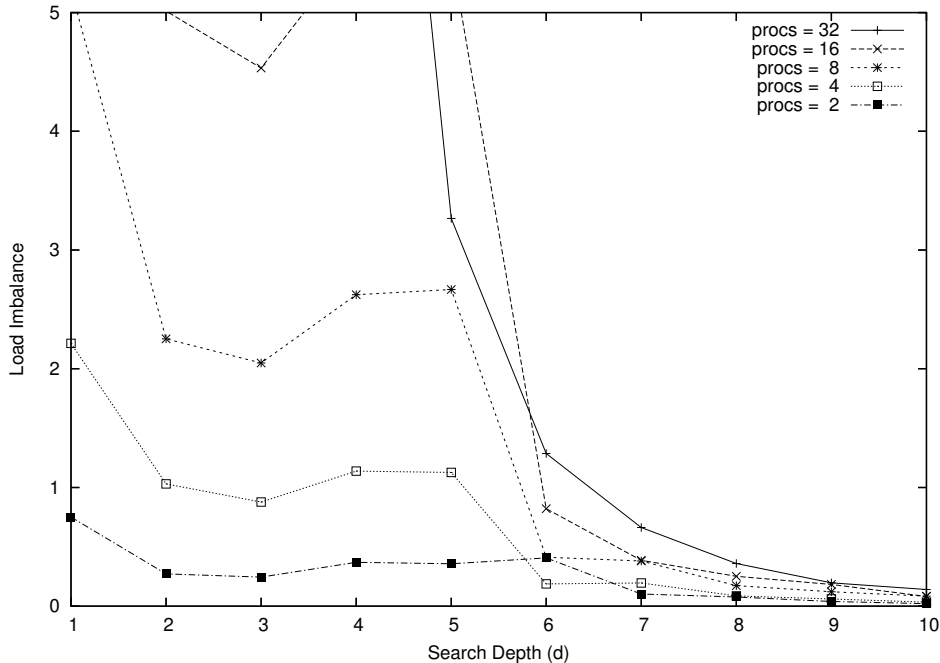


Figure 5.10. Load imbalance for DARKSIGHT at different search depths.

synchronisation overhead will hardly change after this problem has been solved, because the basic algorithm remains the same. Of course, a re-calibration of the granularity values of the parallel search is needed. The search overhead is expected to decrease to some stable level, which depends on the implemented splitting strategy (i.e., the YBWC). Although it is hard to estimate the search overhead, we assume the search overhead to stabilise near 50%, based on two observations. One, a search overhead of 70% was achieved with APHID in THE TURK at a search depth of 8 ply [19]. Two, the search overhead was observed to decrease at increased search depths with the YBWC in ZUGZWANG [32]. Reducing the search overhead to approximately 50% increases the efficiency to approximately 50%.

Larger speedups than presented here can be achieved in numerous ways, besides solving the problem with the pruning mechanism. First, the synchronisation overhead can be reduced by applying YBWC only at CUT-nodes [32], instead of applying it at every interior node (see Section 4.3.1). Second, full window and null-window searches are handled equally, while the latter expand significantly smaller search trees. Third, the symmetric multiprocessing functionality on every node of the DAS-2 system was not used, which will result in larger speedups [29, 54]. Finally, speedups below 1 can be prevented by not initiating parallelism for search depths that are too shallow.

Although no improvement can yet be reported on applying TDS to a parallel two-player state-space searcher, there is reason for optimism. TDS is a load-balancing scheme whose task it is to evenly distribute load over all available processors, in

order to minimise the load imbalance (see Section 2.2.1). Figure 5.10 shows the load imbalance for the experimental results with DARKSIGHT that achieved the largest speedups, as described above. This graph gives insight into the effectiveness of the application of the TDS load-balancing scheme in DARKSIGHT. It can be observed that the load imbalance decreases with increasing search depths, and with a smaller number of processors. For instance, the load imbalance decreases from a 0.04 - 0.20 range at a search depth of 9 ply, to a rang of 0.02 - 0.14 at a search depth of 10 ply.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this thesis, the design and implementation of our two-player state-space searcher DARKSIGHT have been described. This program has been used to address the question raised in Romein’s thesis [82], whether transposition-driven scheduling (TDS) can successfully be applied to parallel two-player state-space searching. Furthermore, the aspiration window and the deep search behaviour of the two-player state-space searcher CRAFTY have been studied.

The results of our aspiration-window experiment show that the search values become more accurate when searching to increased depths. Therefore, searches to succeeding depths differ less in value with increasing search depth. This suggests that smaller aspiration windows should be used for deeper searches instead of constant-sized windows, as is done in most modern programs.

Using the results of our experiment on the deep search behaviour of CRAFTY, the best-change rate and fresh-best rate have been quantified with a high level of statistical confidence. The results show that the best-change rate continues to decrease with increasing search depth, instead of stabilising as was suggested by others [42, 47, 55]. However, the results also show that the fresh-best rate seems to stabilise at 25 – 30%.

The results of our last experiment prove that TDS has successfully been applied in DARKSIGHT, with achievable speedups that are comparable to those of other programs with efficiencies near 50%. The load imbalance is shown to be nearly absent, which is the primary task of TDS. Furthermore, the communication overhead is shown to be small, while the synchronisation and especially the search overhead cause the performance bottleneck. This indicates that a better splitting strategy is needed to improve the efficiency of a parallel two-player state-space searcher.

6.2 Future Work

Because DARKSIGHT has been implemented from scratch, many opportunities exist for improvement. For instance, its playing strength can be improved by applying machine learning on the values used in the evaluation function. Although TDS has been applied successfully, many improvements and scientific challenges remain concerning the parallel algorithm:

- Currently, the YBWC is applied at all interior nodes. Applying the YBWC only at CUT-nodes will result in more parallelism, reduced synchronisation overhead, and increased speedup [32].
- Shared memory, available on all the nodes of DAS-2, is not yet being used. Using the available shared memory may both reduce the search overhead, and gain speedup. The reported near-linear speedup results show very promising [23].
- Searching with a null-window results in expanding a significantly smaller search tree compared to searching with a full window. However, currently they use the same granularity value in DARKSIGHT. Synchronisation overhead might be reduced when different values are used.
- Performance might be improved significantly by optimising distributed data structures, like the partitioned transposition table, by running application-specific software on the network-interface card [11].
- Although the processing power of the nodes is used, neither the additional main memory nor disk space is exploited. However, the additional memory can be used (e.g., additional endgame tablebases), which would not be possible otherwise.
- Implementing an efficient parallel searcher already has been a challenge on a single cluster. An even bigger challenge is to make it run efficiently on a multicluster interconnected by a high-latency and low-bandwidth network, like wide-area connections. A wide-area variant of transposition-driven scheduling has already been applied successfully to the one-player search algorithm IDA* [83].

Bibliography

- [1] S. G. Akl and M. M. Newborn. The Principal Continuation and the Killer Heuristic. In *ACM '77 National Conference, Proceedings*, pages 466–473. ACM Press, 1977.
- [2] T. S. Anantharaman. Extension Heuristics. *ICCA Journal*, 14(2):47–65, 1991.
- [3] T. S. Anantharaman, M. S. Campbell, and F.-h. Hsu. Singular Extensions: Adding Selectivity to Brute-Force Searching. *ICCA Journal*, 11(4):135–143, 1988.
- [4] D.F. Beal and M.C. Smith. Quantification of Search Extension Benefits. *ICCA Journal*, 18(4):205–218, 1995.
- [5] D. F. Beal. Mating Sequences in the Quiescence Search. *ICCA Journal*, 7(3):133–137, 1984.
- [6] D. F. Beal. Experiments with the Null Move. In D. F. Beal, editor, *Advances in Computer Chess 5*, pages 65–79. Elsevier Science Publishing, 1989.
- [7] D. F. Beal. *The Nature of Minimax Search*. PhD thesis, Universiteit Maastricht, The Netherlands, 1999.
- [8] H. J. Berliner. Some Innovations Introduced by HiTech. *ICCA Journal*, 10(3):111–117, 1987.
- [9] H. J. Berliner, G. Goetsch, M. S. Campbell, and C. Ebeling. Measuring the Performance Potential of Chess Programs. *Artificial Intelligence*, 43(1):7–21, 1990.
- [10] P. Bettadapur. Influence of Ordering on Capture Search. *ICCA Journal*, 9(4):180–188, 1986.
- [11] R. A. F. Bhoedjang, J. W. Romein, and H. E. Bal. Optimizing Distributed Data Structures Using Application-Specific Network Interface Software. In *International Conference on Parallel Processing*, pages 485–492, 1998.
- [12] D. M. Breuker. *Memory Versus Search in Games*. PhD thesis, Universiteit Maastricht, The Netherlands, 1998.
- [13] D. M. Breuker, J. W. H. M. Uiterwijk, and H. J. van den Herik. Replacement Schemes for Transposition Tables. *ICCA Journal*, 17(4):183–193, 1994.
- [14] D. M. Breuker, J. W. H. M. Uiterwijk, and H. J. van den Herik. Replacement Schemes and Two-Level Tables. *ICCA Journal*, 19(3):175–180, 1996.
- [15] D. M. Breuker, J. W. H. M. Uiterwijk, and H. J. van den Herik. Information in Transposition Tables. In H. J. van den Herik and J. W. H. M. Uiterwijk, editors, *Advances in Computer Chess 8*, pages 199–211. Universiteit Maastricht, 1997.
- [16] M. G. Brockington. A Taxonomy of Parallel Game-Tree Search Algorithms. *ICCA Journal*, 19(3):162–174, 1996.
- [17] M. G. Brockington. *Asynchronous Parallel Game-Tree Search*. PhD thesis, University of Alberta, Canada, 1998.
- [18] M. G. Brockington and J. Schaeffer. APHID Game-Tree Search. In H. J. van den Herik and J. W. H. M. Uiterwijk, editors, *Advances in Computer Chess 8*, pages 69–92. Universiteit Maastricht, 1997.
- [19] M. G. Brockington and J. Schaeffer. APHID: Asynchronous Parallel Game-Tree Search. *Journal of Parallel and Distributed Computing*, 60:247–273, 2000.

- [20] M. Campbell, A. J. Hoane, and F.-h. Hsu. Deep Blue. *Artificial Intelligence*, 134:57–83, 2002.
- [21] K. M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.
- [22] P. Ciancarini. A Comparison of Parallel Search Algorithms based on Tree Splitting. Technical Report 94-14, University of Bologna, Laboratory for Computer Science, Bologna, Italy, May 1994.
- [23] Computer-Chess Club. <http://www.talkchess.com/>, Mar. 2005.
- [24] J. H. Condon and K. Thompson. Belle. In P. W. Frey, editor, *Chess Skill in Man and Machine*, pages 201–210. Springer-Verlag, New York, N.Y., 1983.
- [25] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, United States of America, 2000.
- [26] S. M. Cracraft. Bitmap Move Generation in Chess. *ICCA Journal*, 7(3):146–153, 1984.
- [27] A. C. Davison and D. V. Hinkley. *Bootstrap Methods and Their Application*. Cambridge University Press, 1997.
- [28] A. D. de Groot. *Thought and Choice in Chess*. Mouton publishers, The Hague, The Netherlands, 1978.
- [29] V. Diepeveen. Personal communication, 2004.
- [30] C. Donninger. Null Move and Deep Search: Selective Search Heuristics for Obtuse Chess Programs. *ICCA Journal*, 16(3):137–143, 1993.
- [31] C. Ebeling. *All the Right Moves: A VLSI Architecture for Chess*. PhD thesis, Carnegie-Mellon University, 1986.
- [32] R. Feldmann. *Game Tree Search with Massive Parallel Systems*. PhD thesis, University of Paderborn, Germany, 1993.
- [33] R. Feldmann. Fail-High Reductions. In H. J. van den Herik and J. W. H. M. Uiterwijk, editors, *Advances in Computer Chess 8*, pages 111–127. Universiteit Maastricht, 1997.
- [34] R. Feldmann, B. Monien, P. Mysliwicz, and O. Vornberger. Distributed Game Tree Search. *ICCA Journal*, 12(2):65–73, 1989.
- [35] R. Feldmann, P. Mysliwicz, and B. Monien. Game-Tree Search on a Massively Parallel System. In H. J. van den Herik, I. S. Herschberg, and J. W. H. M. Uiterwijk, editors, *Advances in Computer Chess 7*, pages 203–218. University of Limburg, 1994.
- [36] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine (A Users' Guide and Tutorial for Networked Parallel Computing)*. MIT Press, 1994. online version at <http://www.netlib.org/pvm3/book/pvm-book.html>.
- [37] G. Goetsch and M. S. Campbell. Experiments with the Null Move Heuristic in Chess. In *Proceedings AAAI Spring Symposium on Computer Game Playing*, pages 14–18, 1988.
- [38] G. Goetsch and M. S. Campbell. Experiments with the Null-Move Heuristic. In T. A. Marsland and J. Schaeffer, editors, *Computers, Chess, and Cognition*, pages 159–168. Springer, 1990.
- [39] A. Grama, A. Gupta, G. Karypis, and V. Kumar. *Introduction to Parallel Computing*. Addison Wesley, 2003.
- [40] D. Hartman. Notions of Evaluation Functions Tested against Grandmaster Games. In D. F. Beal, editor, *Advances in Computer Chess 5*, pages 91–141. Elsevier Science Publishing, 1989.

- [41] G. McC. Haworth and R. B. Andrist. Model Endgame Analysis. In H. J. van den Herik, H. Iida, and E. A. Heinz, editors, *Advances in Computer Games: Many Games, Many Challenges*, pages 65–80. Kluwer Academic Publishers, 2003.
- [42] E. A. Heinz. DarkThought Goes Deep. *ICCA Journal*, 21(4):228–244, 1998.
- [43] E. A. Heinz. Efficient Interior-Node Recognition. *ICCA Journal*, 21(3):156–167, 1998.
- [44] E. A. Heinz. Extended Futility Pruning. *ICCA Journal*, 21(2):75–83, 1998.
- [45] E. A. Heinz. Adaptive Null-Move Pruning. *ICCA Journal*, 22(3):123–132, 1999.
- [46] E. A. Heinz. Knowledgeable Encoding and Querying of Endgame Databases. *ICCA Journal*, 22(2):81–97, 1999.
- [47] E. A. Heinz. Modeling the “Go Deep” Behaviour of Crafty and DarkThought. In H. J. van den Herik and B. Monien, editors, *Advances in Computer Chess 9*, pages 59–71. Universiteit Maastricht, 1999.
- [48] E. A. Heinz. Self-Play Experiments in Computer Chess Revisited. In H. J. van den Herik and B. Monien, editors, *Advances in Computer Chess 9*, pages 73–91. Universiteit Maastricht, 1999.
- [49] E. A. Heinz. A New Self-Play Experiment in Computer Chess. Technical Report 608, Laboratory of Computer Science, Massachusetts Institute of Technology, United States of America, 2000.
- [50] E. A. Heinz. *Scalable Search in Computer Chess (Algorithmic Enhancements and Experiments at High Search Depths)*. Vieweg, Wiesbaden, Germany, 2000.
- [51] E. A. Heinz. Self-Play, Deep Search and Diminishing Returns. *ICGA Journal*, 24(2):75–79, 2001.
- [52] E. A. Heinz. Static Recognition of Potential Wins in KNNKB and KNNKN. In H. J. van den Herik, H. Iida, and E. A. Heinz, editors, *Advances in Computer Games: Many Games, Many Challenges*, pages 45–64. Kluwer Academic Publishers, 2003.
- [53] R. M. Hyatt. Rotated Bitmaps, a New Twist on an Old Idea. *ICCA Journal*, 22(4):213–222, 1999.
- [54] R. M. Hyatt. Personal communication, 2004.
- [55] R. M. Hyatt and M. M. Newborn. Crafty Goes Deep. *ICCA Journal*, 20(2):79–86, 1997.
- [56] A. Junghanns, J. Schaeffer, M. G. Brockington, Y. Björnsson, and T. A. Marsland. Diminishing Returns for Additional Search in Chess. In H. J. van den Herik and J. W. H. M. Uiterwijk, editors, *Advances in Computer Chess 8*, pages 53–67. Universiteit Maastricht, 1997.
- [57] H. Kaindl. Dynamic Control of the Quiescence Search in Computer Chess. In R. Trappl, editor, *Cybernetics and Systems Research*, pages 973–977. North-Holland, Amsterdam, 1982.
- [58] H. Kaindl. Quiescence Search in Computer Chess. *SIGART Newsletter*, 80:124–131, 1982.
- [59] D. E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, 1968.
- [60] D. E. Knuth and R. W. Moore. An Analysis of Alpha-Beta Pruning. *Artificial Intelligence*, 6(4):293–326, 1975.
- [61] D. Kopec and I. Bratko. The Bratko-Kopec Experiment: A Comparison of Human and Computer Performance in Chess. In M. R. B. Clarke, editor, *Advances in Computer Chess 3*, pages 57–72. Pergamon Press Ltd., 1982.
- [62] R.E. Korf. Iterative Deepening: An Optimal Admissible Tree Search. *Artificial Intelligence*, 27(1):97–109, 1985.
- [63] B. C. Kuszmaul. *Synchronized MIMD Computing*. PhD thesis, Massachusetts Institute of Technology, United States of America, 1994.

- [64] D. N. L. Levy. *The Chess Computer Handbook*. B.T.Batsford Ltd., London, England, 1984.
- [65] D. N. L. Levy, D. C. Broughton, and M. Taylor. The SEX Algorithm in Computer Chess. *ICCA Journal*, 12(1):10–21, 1989.
- [66] T. A. Marsland. Relative Performance of the Alpha-Beta Algorithm. *ICCA Newsletter*, 5(2):21–24, 1982.
- [67] T. A. Marsland. A Review of Game-Tree Pruning. *ICCA Journal*, 9(1):3–19, 1986.
- [68] T. A. Marsland, M. Olafsson, and J. Schaeffer. Multiprocessor Tree-Search Experiments. In D. F. Beal, editor, *Advances in Computer Chess 4*, pages 37–51. Pergamon Press Ltd., 1985.
- [69] H. H. Mohamed and D. H. J. Epema. The design and implementation of the koala co-allocating grid scheduler. In *European Grid Conference*, 2005.
- [70] H. H. Mohamed and D. H. J. Epema. Experiences with the koala co-allocating scheduler in multiclusters. In *Proc. of the 5th IEEE/ACM Int'l Symp. on Cluster Computing and the GRID (CCGrid2005)*, 2005.
- [71] Message passing interface standards. <http://www.mpi-forum.org/docs/docs.html>, Mar. 2005.
- [72] Myricom, inc. website. <http://www.myrinet.com/>, Mar. 2005.
- [73] P. Mysliwicz. *Konstruktion und Optimierung von Bewertungsfunktionen beim Schach*. PhD thesis, University of Paderborn, Germany, 1994.
- [74] E. V. Nalimov, G. McC. Haworth, and E. A. Heinz. Space-Efficient Indexing of Endgame Tables for Chess. In H. J. van den Herik and B. Monien, editors, *Advances in Computer Chess 9*, pages 93–113. Universiteit Maastricht, 1999.
- [75] E. V. Nalimov, G. McC. Haworth, and E. A. Heinz. Space-Efficient Indexing of Chess Endgame Tables. *ICGA Journal*, 23(3):148–162, 2000.
- [76] H. L. Nelson. Hash Tables in CRAY BLITZ. *ICCA Journal*, 8(1):3–13, 1985.
- [77] M. M. Newborn. A Hypothesis Concerning the Strength of Chess Programs. *ICCA Journal*, 8(4):209–215, 1985.
- [78] M. M. Newborn. Unsynchronized Iteratively Deepening Parallel Alpha-Beta Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, 10(5):687–694, 1988.
- [79] A. Plaat, J. Schaeffer, W. Pijls, and A. de Bruin. Exploiting Graph Properties of Game Trees. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI'96)*, volume 1, pages 234–239, 1996.
- [80] A. Reinefeld. An Improvement to the SCOUT Tree-Search Algorithm. *ICCA Journal*, 6(4):4–14, 1983.
- [81] A. Reinefeld. *Analyse von Baum - Suchalgorithmen*. PhD thesis, University of Hamburg, Germany, 1987.
- [82] J. W. Romein. *Multigame - An Environment for Distributed Game-Tree Search*. PhD thesis, Vrije Universiteit Amsterdam, The Netherlands, 2001.
- [83] J. W. Romein and H. E. Bal. Wide-Area Transposition-Driven Scheduling. In *IEEE International Symposium on High Performance Distributed Computing*, pages 347–355, 2001.
- [84] J. W. Romein and H. E. Bal. Solving Awari with Parallel Retrograde Analysis. *IEEE Computer*, 36(10):26–33, 2003.
- [85] J. W. Romein, H. E. Bal, J. Schaeffer, and A. Plaat. A Performance Analysis of Transposition-Table-Driven Work Scheduling in Distributed Search. *IEEE Transactions on Parallel and Distributed Systems*, 13(5):447–459, 2002.
- [86] J. W. Romein, A. Plaat, H. E. Bal, and J. Schaeffer. Transposition Table Driven Work Scheduling in Distributed Search. In *16th National Conference on Artificial Intelligence (AAAI)*, pages 725–731, 1999.

- [87] J. Schaeffer. The History Heuristic. *ICCA Journal*, 6(3):16–19, 1983.
- [88] J. Schaeffer. Distributed Game-Tree Searching. *Journal of Parallel and Distributed Computing*, 6(2):90–114, 1989.
- [89] J. Schaeffer. The History Heuristic and Alpha-Beta Search Enhancements in Practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(11):1203–1212, 1989.
- [90] J. Schaeffer and J. van den Herik. *Chips Challenging Champions (Games, Computers and Artificial Intelligence)*. Elsevier Science B.V., Amsterdam, The Netherlands, 2002.
- [91] G. Schrüfer. A Strategic Quiescence Search. *ICCA Journal*, 12(1):3–9, 1989.
- [92] C. E. Shannon. Programming a Computer for Playing Chess. *Philosophical Magazine*, 41(4):256–275, 1950.
- [93] D. J. Slate. Interior-Node Score Bounds in a Brute-Force Chess Program. *ICCA Journal*, 7(4):184–192, 1984.
- [94] D. J. Slate and L. R. Atkin. CHESS 4.5 - The Northwestern University Chess Program. In P. W. Frey, editor, *Chess Skill in Man and Machine*, pages 82–118. Springer-Verlag, 1977.
- [95] J. R. Steenhuisen. A Survey of State-Space Searching. Jan. 2004.
- [96] M. Sturman. Beware the Bishop Pair. *ICCA Journal*, 19(2):83–93, 1996.
- [97] Surfnet. <http://www.surfnet.nl/>, Mar. 2005.
- [98] A. Szabo and B. Szabo. The Technology Curve Revisited. *ICCA Journal*, 11(1):14–20, 1988.
- [99] O. D. Tabibi, A. Felner, and N. S. Netanyahu. Blockage Detection in Pawn Endings. *ICGA Journal*, 27(3):150–158, 2004.
- [100] O. D. Tabibi and N. S. Netanyahu. Verified Null-Move Pruning. *ICGA Journal*, 25(3):153–161, 2002.
- [101] K. Thompson. Computer Chess Strength. In M. R. B. Clarke, editor, *Advances in Computer Chess 3*, pages 55–56. Pergamon Press Ltd., 1982.
- [102] K. Thompson. Retrograde Analysis of Certain Endgames. *ICCA Journal*, 9(3):131–139, 1986.
- [103] K. Thompson. Chess Endgames Volume 1. *ICCA Journal*, 14(1):22, 1991.
- [104] G. Timoshchenko. Bishop or Knight? *ICCA Journal*, 16(4):209–216, 1993.
- [105] T. Warnock and B. Wendorff. Search Tables in Computer Chess. *ICCA Journal*, 11(1):10–13, 1988.
- [106] J.-C. Weill. The ABDADA Distributed Minimax-Search Algorithm. *ICCA Journal*, 19(1):3–16, 1996.
- [107] A. L. Zobrist. A Hashing Method with Applications for Game Playing. *ICCA Journal*, 13(2):69–73, 1990.

Appendix A

Deep Search

This appendix lists the exact numerical data of the deep search experiments of Section 5.2. In addition to these values, their standard errors (see Section 5.2.3) are provided.

Search Depth	Best Change	Fresh Best	(d - 2) Best	(d - 3) Best
2	0.4006 (2.65)	1.0000 (0.00)		
3	0.3860 (2.63)	0.7652 (2.29)	0.2348 (2.29)	
4	0.2865 (2.44)	0.6224 (2.62)	0.2857 (2.44)	0.0918 (1.56)
5	0.3050 (2.49)	0.5962 (2.66)	0.2308 (2.28)	0.0865 (1.52)
6	0.2971 (2.48)	0.5347 (2.71)	0.2970 (2.48)	0.0693 (1.38)
7	0.2676 (2.40)	0.4945 (2.71)	0.3407 (2.57)	0.0549 (1.24)
8	0.2824 (2.44)	0.4167 (2.67)	0.3333 (2.56)	0.1042 (1.66)
9	0.2029 (2.18)	0.4928 (2.71)	0.2609 (2.38)	0.0580 (1.27)
10	0.1882 (2.12)	0.3750 (2.63)	0.2656 (2.40)	0.1250 (1.79)
11	0.1794 (2.08)	0.4262 (2.68)	0.2295 (2.28)	0.0820 (1.49)
12	0.1652 (2.02)	0.1786 (2.08)	0.3393 (2.57)	0.1429 (1.90)
13	0.1298 (1.83)	0.2955 (2.48)	0.2727 (2.42)	0.0909 (1.56)
14	0.1306 (1.84)	0.4091 (2.68)	0.1818 (2.10)	0.0000 (0.00)
15	0.1276 (1.82)	0.3256 (2.55)	0.2093 (2.22)	0.0930 (1.58)
16	0.1194 (1.77)	0.2750 (2.44)	0.2000 (2.19)	0.1500 (1.95)
17	0.1048 (1.68)	0.3143 (2.54)	0.2000 (2.19)	0.0571 (1.27)
18	0.1138 (1.76)	0.2973 (2.54)	0.1622 (2.04)	0.1351 (1.90)
19	0.0903 (1.69)	0.4615 (2.94)	0.1538 (2.13)	0.1154 (1.88)
20	0.0766 (1.69)	0.2105 (2.59)	0.3158 (2.95)	0.0526 (1.42)

Table A.1. Results of CRAFTY (2004) for the original test positions.

Search Depth	Best Change	Fresh Best	(d - 2) Best	(d - 3) Best
2	0.3684 (0.54)	1.0000 (0.00)		
3	0.3505 (0.54)	0.8592 (0.45)	0.1408 (0.45)	
4	0.3253 (0.53)	0.6303 (0.55)	0.2749 (0.51)	0.0948 (0.33)
5	0.3134 (0.52)	0.5484 (0.56)	0.3038 (0.52)	0.0788 (0.27)
6	0.2774 (0.51)	0.4935 (0.56)	0.2931 (0.52)	0.0983 (0.32)
7	0.2393 (0.50)	0.4409 (0.56)	0.3269 (0.53)	0.0968 (0.30)
8	0.2306 (0.49)	0.4621 (0.55)	0.2478 (0.52)	0.1161 (0.33)
9	0.2141 (0.47)	0.3942 (0.54)	0.3269 (0.53)	0.1034 (0.33)
10	0.2115 (0.47)	0.3309 (0.52)	0.3504 (0.54)	0.1071 (0.33)
11	0.1992 (0.45)	0.3049 (0.53)	0.3333 (0.54)	0.0879 (0.32)
12	0.1863 (0.44)	0.3398 (0.52)	0.3177 (0.53)	0.0884 (0.33)
13	0.1642 (0.61)	0.3260 (0.76)	0.3229 (0.75)	0.1129 (0.46)
14	0.1498 (0.59)	0.2955 (0.73)	0.3471 (0.76)	0.0928 (0.52)
15	0.1401 (0.57)	0.2831 (0.74)	0.3309 (0.75)	0.1176 (0.49)
16	0.1185 (0.54)	0.2913 (0.72)	0.3043 (0.75)	0.1000 (0.51)
17	0.1096 (0.52)	0.2736 (0.72)	0.2925 (0.74)	0.1226 (0.53)
18	0.1172 (0.51)	0.3244 (0.74)	0.2311 (0.71)	0.1067 (0.49)

Table A.2. Results of CRAFTY (2004) for the ECO test positions.

Search Depth	Best Change	Fresh Best	(d - 2) Best	(d - 3) Best
2	0.3878 (2.63)	1.0000 (0.00)		
3	0.3673 (2.60)	0.7143 (2.44)	0.2857 (2.44)	
4	0.3061 (2.49)	0.6571 (2.56)	0.2571 (2.36)	0.0857 (1.51)
5	0.3032 (2.48)	0.5962 (2.65)	0.3077 (2.49)	0.0673 (1.35)
6	0.2741 (2.41)	0.5638 (2.68)	0.2447 (2.32)	0.0851 (1.51)
7	0.2449 (2.32)	0.4762 (2.70)	0.3095 (2.50)	0.0714 (1.39)
8	0.2245 (2.25)	0.3766 (2.62)	0.3117 (2.50)	0.1169 (1.73)
9	0.1837 (2.09)	0.3016 (2.48)	0.3810 (2.62)	0.0476 (1.15)
10	0.1720 (2.04)	0.4068 (2.65)	0.3220 (2.52)	0.0508 (1.19)
11	0.1662 (2.01)	0.5263 (2.70)	0.2456 (2.32)	0.0526 (1.21)
12	0.1691 (2.02)	0.4138 (2.66)	0.2414 (2.31)	0.1034 (1.64)
13	0.1458 (1.91)	0.3200 (2.52)	0.2200 (2.24)	0.1400 (1.87)
14	0.1545 (1.95)	0.3962 (2.64)	0.3208 (2.52)	0.0566 (1.25)

Table A.3. Results of CRAFTY (1997) for the original test positions.

Search Depth	Best Change	Fresh Best	(d - 2) Best	(d - 3) Best
2	0.3528 (2.58)	1.0000 (0.00)		
3	0.3965 (2.64)	0.8529 (1.91)	0.1471 (1.91)	
4	0.3178 (2.51)	0.5505 (2.69)	0.3119 (2.50)	0.1376 (1.86)
5	0.2945 (2.46)	0.5644 (2.68)	0.2475 (2.33)	0.1089 (1.68)
6	0.2449 (2.32)	0.6548 (2.57)	0.1905 (2.12)	0.0595 (1.28)
7	0.2128 (2.21)	0.4932 (2.70)	0.2877 (2.44)	0.1096 (1.69)
8	0.2507 (2.34)	0.5000 (2.70)	0.2442 (2.32)	0.0465 (1.14)
9	0.2157 (2.22)	0.4054 (2.65)	0.2838 (2.43)	0.1351 (1.85)
10	0.2420 (2.31)	0.3735 (2.61)	0.3494 (2.57)	0.0843 (1.50)
11	0.1749 (2.05)	0.3167 (2.51)	0.3667 (2.60)	0.1000 (1.62)
12	0.1545 (1.95)	0.4528 (2.69)	0.2076 (2.19)	0.0943 (1.58)
13	0.1662 (2.01)	0.4211 (2.67)	0.2807 (2.43)	0.1053 (1.66)
14	0.1370 (1.86)	0.3404 (2.56)	0.2553 (2.35)	0.1277 (1.80)

Table A.4. Results of DARKTHOUGHT for the original test positions.

Appendix B

Bratko-Kopec Test Set

This appendix lists the test set of 24 positions used in the experiments discussed in Section 5.3, better known as the Bratko-Kopec test set [61].

