BUILDING THE CHECKERS 10-PIECE ENDGAME DATABASES

Jonathan Schaeffer, Yngvi Björnsson, Neil Burch,
Robert Lake, Paul Lu and Steve Sutphen
Department of Computing Science,
University of Alberta,
Edmonton, Alberta
Canada T6G 2E8
{jonathan,yngvi,burch,lake,paullu,steve}@cs.ualberta.ca

Abstract In 1993, the CHINOOK team completed the computation of the 2 through 8-piece checkers endgame databases, consisting of roughly 444 billion positions. Until recently, nobody had attempted to extend this work. In November 2001, we began an effort to compute the 9- and 10-piece databases. By June 2003, the entire 9-piece database and the 5-piece versus 5-piece portion of the 10-piece database were completed. The result is a 13 trillion position database, compressed into 215 GB of data organized for real-time decompression. This represents the largest endgame database initiative yet attempted. The results obtained from these computations are being used to aid an attempt to weakly solve the game. This paper describes our experiences working on building large endgame databases.

Keywords: Retrograde analysis, endgame databases, checkers

1. Introduction

Endgame databases have had an enormous impact in computer games research. They have been instrumental in building world championship programs (e.g., the World Man-Machine Checkers Champion CHINOOK (Schaeffer, 1997)), solving games (e.g., Nine Men's Morris (Gasser, 1996) and Awari (Romein and Bal, 2003; Romein and Bal, 2002)), and uncovering new insights into games.

For converging games, where the number of pieces on the board reduces as the game progresses, larger endgame databases are a performance asset to a game-playing program, both in terms of reducing the size of the search tree and by replacing heuristic evaluations with perfect

knowledge. However, there are practical considerations to building large databases, including the time required to compute them, and the resulting size of the (compressed) databases. Few researchers and developers have the expertise, motivation, patience, and computing resources to push database technology to its limit (a recent exception is the solution to the game of Awari (Romein and Bal, 2003; Romein and Bal, 2002)). This means, for example, that the 6-piece chess endgame databases are unlikely to be completed in the near future.

CHINOOK is the World Man-Machine Checkers Champion (Schaeffer, 1997).[1] The 8-piece endgame databases were a critical part of the program's success against the top human players. The databases contained secrets that were well beyond the understanding of even the premier players in the world. These databases were started in 1989 and completed in 1993—444 billion positions compressed into 5.6 GB of data. These numbers may seem small by today's standards, but were impressive back in the early 1990s when a state-of-the-art CPU was an Intel 486, 32 MB was considered to be a lot of memory, and 1 GB disks were new technology and very expensive.

Beginning in November 2001, we started production runs for computing the 9- and 10-piece checkers endgame databases. The databases are not needed to improve the playing strength of checkers programs; there are currently at least five checkers programs that are superior to all human players. Rather, there is a more enticing goal: solving the game of checkers (or, more precisely, weakly solving the game (Allis, 1994)). The total search space for the game is 5×10^{20}, a seemingly prohibitively large number. However, most of the search space is likely to be irrelevant to the proof, and resulting estimates of the proof-tree size are well within what is possible to compute with current technology. Building the 10-piece databases (specifically the key 5×5 subset, where each side has the same number of pieces) is a key stepping stone to solving checkers.

This paper describes our experiences building the 9- and 10-piece checkers endgame databases. The task was daunting, given the need for 64-bit addressing, large computations (up to 171 billion positions in one computation), large intermediate disk needs (over 1 TB), verification of the results, and fault tolerance. In 10 years, even these numbers will seem trivial, but the techniques will be useful for the next large database computation.

This paper makes the following contributions:

1 the practical considerations that complicate any long-term data-intensive computation,

2 the system issues that need to be addressed, including memory constraints, concurrency, compression, and fault tolerance,

3 improved data compression techniques,

4 data on the 9- and 10-piece checkers databases, and

5 speculation on the likelihood of solving checkers in the near future.

Section 2 describes the algorithms used to compute the 8-piece databases. Section 3 discusses the enhancements needed to move to the larger 10-piece databases. The results from building the databases and the implications for solving the game of checkers are in Section 4. Section 5 concludes with perspectives on building larger databases.

2. Algorithms

The important application-specific properties that influence the database algorithms are (Goldenberg et al., 2003) (the "Properties"):

1 The game starts with 12 white checkers and 12 black checkers on the board.

2 When a piece is captured, it is removed from the board and cannot return.

3 Checkers can be promoted to become kings (when the checker moves to the back rank of the opponent).

4 Checkers can only move forward; kings can move forward and backward.

The algorithms used for the checkers computation are updated versions of those used to compute the CHINOOK 8-piece databases (Lake et al., 1994). This code had not been touched since the completion of the databases in 1993.

The most common format of an endgame database stores for each position a distance metric. This metric is typically either the number of moves to win (if appropriate) or the number of moves to convert to another database. This level of detail is tremendously useful in practice since it allows a game-playing program to play the "best" database moves without needing any search. However, this representation requires (at least) a byte of data per position, and the resulting database does not compress well. The philosophy adopted for building checkers databases has been to build the largest databases possible. To do this necessitates storing the minimal amount of information per position in

the database—recording only whether a position is a win, a loss or a draw. The result facilitates the creation of large endgame databases that compress extremely well.

For database calculations, each position is represented by 2 bits, representing the values win (W), loss (L), at least a draw (D), and unknown (U). Using D to mean at-least-a-draw instead of exactly a draw is useful, since it reduces the amount of disk I/O done by the program (see the Lookups phase described below). A portion of the endgame database (a *slice*) is computed by resolving all positions as wins, losses or draws. The final result is compressed, verified, and then added to the master copy of the completed databases.

The 10-piece databases are huge (8.5 trillion positions for just the 5-piece versus 5-piece subset), and it is not practical to do the entire calculation as one big computation. Instead, the problem is broken down into smaller slices that can be solved more easily. The databases are broken down as follows:

- By pieces: The N-piece database can be computed once the N-1-piece database is done (by Property #2).

- By material: An N-piece database is further divided so that subsets with a different number of pieces per side can be computed in parallel (Property #2). For example, in the 9-piece database computation, the 8 pieces versus 1, 7 versus 2, 6 versus 3, and 5 versus 4 subsets can be computed in parallel.

- By number of kings: The material division is further broken down by the number of kings for each side (exploiting Property #3). For example, after 5 kings versus 4 kings have been computed, then the subset 4 kings and 1 checker versus 4 kings can be computed (the one checker might promote, thus the 5 king versus 4 king database must be computed first).

- By leading rank: A sub-database is further *sliced* into pieces by considering the position of each side's most advanced (leading) checker (from ranks 1 to 7). Positions where the leading checker is on rank R must be computed before those where the leading checker is on rank $R-1$ (Property #4). For example, in the 4 kings and 1 checker versus 4 kings endgame, all positions where the checker is on the seventh rank must be computed before tackling all positions where the checker is on the sixth rank. For databases where each side has a checker, this technique results in dividing the computation into 49 (not-necessarily-equal) slices, dramati-

Table 1. Database slices for 10-piece database (5 versus 5 pieces).

Database	Total Positions	Slices
5500	16,257,084,480	1
5401	142,249,489,200	7
5302	247,789,432,800	7
5203	214,750,841,760	7
5104	92,565,018,000	7
5005	15,868,288,800	6
4411	311,375,610,000	28
4312	1,085,553,705,600	49
4213	941,518,468,800	49
4114	406,152,630,000	49
4015	69,686,136,000	42
3322	946,853,107,200	28
3223	1,643,753,217,600	49
3124	709,688,460,000	49
3025	121,877,184,000	42
2233	714,003,388,800	28
2134	617,101,500,000	49
2035	106,080,312,960	42
1144	133,467,390,552	28
1045	45,934,129,104	42
0055	3,956,576,472	21
Total	8,586,481,972,128	630

cally reducing the size of the biggest computation that has to be performed.

More details on the decomposition can be found in (Lake et al., 1994).

Table 1 shows how the 5-piece versus 5-piece subset of the 10-piece database can be subdivided into smaller pieces. The first column gives the number of kings and checkers for the sub-database using the notation "*bk wk bc wc*", where *bk* is the number of black kings, *wk* is the number of white kings, *bc* is the number of black checkers and *wc* is the number of white checkers. The 8.5 trillion positions are divided into 21 subsets based on the number of kings and checkers. The 3223 subset (3 kings and 2 checkers for black; 2 kings and 3 checkers for white) is the largest, with roughly 1.6 trillion positions. This is subdivided into 49 slices based on the leading checker.

The largest slices in the 5 piece versus 5 piece subset of the 10-piece database are shown in Table 2. To specify a slice, we use the notation "*bk wk bc wc . br wr*" where *br* is the rank of the leading black checker and *wr* is the rank of the leading white checker. The largest slice is 171

Table 2. Largest 10-piece database slices.

Slice	Size		
3223.77/2332.77	85,515,674,400	×2 =	171,031,348,800
2233.76/2233.67	73,228,209,600	×2 =	146,456,419,200
3223.67/2332.76	71,823,866,400	×2 =	143,647,732,800
3223.76/2332.67	59,656,240,800	×2 =	119,312,481,600
3322.76/3322.67	58,741,300,800	×2 =	117,482,601,600
3223.57/2332.75	58,132,058,400	×2 =	116,264,116,800
2134.77/1243.77	56,491,266,000	×2 =	112,982,532,000
2233.77	104,558,625,600	=	104,558,625,600
3223.66/2332.66	50,304,477,600	×2 =	100,608,955,200

billion positions (3223.77 with black to move and its *mirror* database 2332.77 with white to move). Using 2 bits per position, this slice requires almost 40 GB of storage during its computation phase. In total, there are only 9 slices that have a size of over 100 billion positions.

Note that slices can be further sub-divided. Ed Trice and Gil Dodgen have experimented with using both the rank of the leading checker and the configuration of checkers on the first rank to achieve further subdivisions (Dodgen and Trice, 2002). The finer granularity of the slices reduces the RAM needs and increases the computation's concurrency. For the work reported here, additional subdivisions were not needed. However, with current technology they might be needed if one wanted to compute the 11-piece databases (currently not in our plans).

The endgame database solving programs were designed with the following objectives in mind:

1 Reduce the amount of disk I/O needed.

2 Reduce the memory requirements for the largest jobs.

3 Use as many machines as possible.

The computation of a database slice consists of 5 phases.[2] The phases iterate over the data, where each position value in the slice has been initialized to unknown (U). The database construction phases are summarized in Table 3.

1 **Captures**: The rules of checkers require that a capture move, if present in a position, must be played. A capture move removes one or more pieces from the board. All capture moves are looked up in previously computed databases and the maximum of the resulting values (W/L/D) is assigned to the position. For an N-piece

database calculation, this phase only requires the 2 through N-1-piece databases. This is important because the N-1-piece databases are considerably smaller than the N-piece databases. For example, the 9-piece databases are only 25 GB in size. Thus the capture phase for all 10-piece database calculations can be computed well in advance of when the data is needed.

2 **Lookups**: The databases are sliced according to the leading checker. When the leading checker advances, it will result in a position that has already been computed. The Lookups phase resolves all moves by the leading checker. By handling this I/O in a separate phase, we can guarantee that the next phase (non-captures) does not have to access any previously computed database results.

The advance of the checker may result in the current position being resolved as a win. In rare cases the only moves possible in a position are those of the leading checker. If all these moves lead to losing positions, then the current position can be resolved as a loss. If the leading checker advances and the resulting position leads to a draw, then we have a lower bound on the value of the position. The position might still be a win (a king move or non-leading checker move could lead to a winning position). Thus, if a leading checker move results in a draw score, this position is marked as a D but with the semantics being that the value is $\geq$ a draw. For this phase, only the N-piece database is needed (but, as explained below, because of the compression scheme used, the 2 through N-1-piece databases might also be required).

3 **Non-captures**: The preceding phases resolved all requests for information from previously computed database slices. In the non-captures phase, only moves by kings and non-leading checkers are considered. Hence there is no need to access the previously-computed databases. In contrast to the previous phases, the non-captures phase is compute-intensive.

This phase iterates over all positions in the slice, skipping over capture positions (their values are fixed) and W/L positions (their value cannot change). Only unresolved positions and draw positions are considered; the former to discover whether the position is a W/L/D and the the latter to see if the D can become a W. This phase only resolves wins and losses. When no more changes occur during an iteration, the non-captures phase is complete. Any position that has a U or D value must be a real draw.

This phase may require iterating over the data 100 or more times (the maximum number of ply needed to force a winning position into another database slice). To reduce the cost, the program only iterates over all positions until a "small" number of changes occurs in an iteration. The positions that change value are saved in a queue. For subsequent iterations, the only positions whose value can be resolved are those that are a predecessor of a queue position.

4 **Compression**: The endgame databases are needed in a real-time searching program (such as CHINOOK). Hence the data has to be compressed in a way that supports real-time decompression. The compression scheme used is described in Section 3.3.

5 **Verification**: Errors are a fact of life in any long-running computation. Since one result depends on another, it is critical that the computations be verified for correctness. There is an easy way to do this: after the non-captures phase, a quick scan of the data can verify if the resulting set of values is internally consistent (self-consistency). This is quick, but does not catch all possible errors. Instead, our verification phase operates not on the 2-bit-per-position representation but on the compressed database. All positions are verified that they are consistent not only within the slice, but also with respect to previously computed data. The latter point dramatically increases the cost of the verification, but can find errors not caught by the fast scheme. Besides, it makes it easier to sleep at night!

The database construction phases are summarized in Table 3. The time column is a generic average that represents the percentage of wall clock time spent in each phase. These numbers can vary significantly depending on the data set used. The verification phase is the most expensive since, in effect, it has to repeat most of the work done in the previous phases.

The breakdown of the computation into multiple phases assists in planning how to effectively acquire and use computing resources. The captures, lookups, and verification phases are I/O bound. These phases need to be run on machines with a minimum of 300 GB of disk storage, and they benefit from the fastest possible disk drives. The non-capture phase is compute bound. It should be run on the fastest possible processor. The computation is easily parallelized to run on a shared-memory computer, and the performance scales well to a large number of processors.

Table 3. Database construction summary.

Name & Needs	Description	Databases Used	Values Set	Time (%)
Captures I/O	Resolve capture moves. Sequential pass over the data.	2—(N-1)	W,L,D	15
Lookups I/O	Resolve non-capture moves that result in database positions. Sequential pass over the data.	N	W,L,≥D	24
Non-Captures CPU	Resolve non-capture moves. Repeated passes over the data, both sequential and random access, until no more changes.	None	W,L	20
Compress I/O	Convert to final compressed format. Sequential pass over the data.	None	D	1
Verify I/O	Verify that the new results are consistent internally and with pre-existing databases. Sequential pass over the data.	2—N	None	40

3. Moving from Eight to Ten Pieces

This section discusses the issues that had to be addressed to enhance the CHINOOK database calculations to accommodate the larger size of the 10-piece databases.

3.1 64-bit Indices

By subdividing the databases into slices, the original CHINOOK code could get by with using 32-bit numbers for position indices. For the 10-piece databases, the largest individual slice was 104 billion positions (the symmetric database 2233.77), necessitating at least 37 bits for addressing.

The CHINOOK code was converted to use 64-bit indices. By-and-large this was easy to do, but there were some subtleties that were initially overlooked. For example, most C compilers do automatic conversion between 32- and 64-bit numbers (both ways), possibly losing precision (and usually not getting a compiler warning). Another danger was intermediate expression results. Some expressions combined 32- and 64-bit data with implicit data conversions that could lead to errors.

Note that simply converting all numbers to use 64 bits was not an option. The tables used for computing position indices occupy a lot of

memory. Using 32-bit numbers wherever possible reduced the memory footprint of the program, freeing up more space for disk caching.

3.2 64-bit File Sizes

When we started the project, support for 64-bit file sizes was not fully integrated in Linux. However, we were fortunate in that the experimental kernels we used fully supported the two routines that we needed: `open64` and `lseek64`. Support for large files has limited other groups wanting to build large databases on Windows' platforms.

3.3 Compression

Many endgame databases associate a distance metric with a database position (the number of moves to win or the number of moves to convert to another database slice). For checkers, this was impractical. Our goal was to build the largest database possible. For this to happen, disk space and the execution overhead of accessing the data could not be a limitation. For example, if a byte was associated with each of the 13 trillion database positions computed, then 13 TB of disk would be needed. Even a generous 10:1 compression ratio would still leave the database size at an awkward 1.3 TB. The large disk size will dramatically slow down database computations since it will be difficult to achieve spatial and temporal disk locality (this was elegantly addressed for smaller databases by (Lincke and Marzetta, 2000)).

Allowing only win-loss-draw values in the database enables 5 position values to be encoded in a byte ($3^5 = 243 < 256$). Using this trivial compression would result in 13 trillion positions being encoded into 2.6 TB. This is still too large (and expensive) to be practical. Further data compression is needed.

The checkers data has to be available for use in a real-time search. Hence any compression scheme has to support rapid real-time decompression. The databases were compressed by using two techniques:

1 removing information that can be easily re-computed, and

2 run-length encoding.

Any position where either side to move could result in a capture would have the position result removed from the database (i.e., capture and threatened capture positions). It is easy to re-compute the value of a capture position: play the capture move(s) and look up the resulting position(s) in the database. Removing values for positions where a capture is threatened is more problematic. To re-compute this value, the side to move must try all possible moves and, in some cases, in the result-

ing position the opponent has a forced capture or there is a threatened capture—all these positions must be looked up in the database. Hence positions with a threatened capture may require an expensive search to resolve. It quickly became clear that with our compression algorithms, simply removing capture position values was not good enough; we had to remove threatened capture positions to make the compressed database size reasonable. Our estimate is that removing threatened capture positions improves the compression by a factor of 4.

All capture and threatened capture positions would have their value replaced by the dominant value in the database slice. Then run-length encoding would be used to compress the data. The original CHINOOK algorithm encoded 5 positions into a byte (Lake et al., 1994). That left 13 values for the run-length encoding ($256 - 3^5 = 13$). These values were used to represent runs of the dominant value, for runs of length 10 to 3,200. For example, a database slice might be dominated by wins. The capture and threatened capture positions (typically 75% of the positions) would have their values replaced by a win. Run-length encoding would then find many long stretches of wins and encode them into one (or a few) bytes.

The original CHINOOK databases, 444 billion positions (all the 2 through 8-piece databases), were compressed into 5.6 GB. This works out to an average of roughly 77 positions encoded in a byte. This is misleading since the lop-sided databases (e.g., 6 pieces versus 2) compress very well (they are almost all wins for the strong side), whereas the even material databases (e.g., 4 pieces versus 4 pieces) have a mix of win, loss and draw values, resulting in poorer (but still good compression). The 4 pieces versus 4 pieces database averaged 22 positions per byte.

For the 10-piece databases, our initial estimates were that the above scheme would result in a final database size of 400 GB. Thus it was important to find a better compression scheme.

The new algorithm is based on Huffman coding and consists of the following steps:

1 Replace capture and threatened capture positions with the win/loss/draw value that continues the current run. For example, if we have a run of wins and then come across a capture position, its value becomes a win.

2 Convert the string of win/loss/draw into a string of *(win/loss/draw, run_length)* pairs. There will not be two consecutive runs with the same first value.

3 Predict the value of a run based on the value of the run before the previous run. For example, given runs *(draw, X)* and *(loss, Y)* we

would predict the value of the next run to be draw. In the small databases, the prediction is wrong roughly 1% of the time. In the 8-, 9-, and 10-piece databases, the miss ratio is about 5%. Now convert the string so that a *(value, length)* pair simply becomes *length*, preceded by a special *miss* symbol if value is not correctly predicted.

4 If a maximum run length of N is chosen, we then have $N-1$ length symbols, one escape symbol that states that an integer length follows, and one symbol that states that the value of this run is predicted incorrectly. Given the frequencies of these symbols, an optimal length limited prefix free code (length limited Huffman code (Turpin and Moffat, 1995)) can be generated. While a different code could be generated for every file, it turns out that the codings are quite stable from the 8-piece database file and beyond, so we can use a fixed code generated from the largest database file to get most of the savings. Twenty bits was chosen as a reasonable limitation on the length of the bit strings, as a table 1,048,576 entries wide used for decoding seemed reasonable and larger string lengths provided minimal improvements. Given this maximum, empirical testing on the databases showed a number around 10,000 to be the best choice for the maximum run length allowed before escaping to a 32-bit integer description. Increasing the number of symbols overly crowded the space of bit strings available for compression by too much, and decreasing the maximum run length increased the number of escaped symbols by too much.

5 The previous types used to predict the types of the first two runs are set by looking ahead at these two symbols and using the values that will correctly predict them. These values are stored at the front of the compressed bit-string using three bits.

Using this new scheme resulted in a reduction in size of the complete 2-piece through 8-piece databases from 5.6 to 2.7 GB, cutting the database in half (averaging out to 155 positions per byte). The complete 9-piece databases is 16.8 GB, an average of 227 positions per byte. The 10-piece databases (5 pieces versus 5 pieces) compress to 125 GB, 65 positions per byte. This represents a substantial improvement over the 22 positions per byte seen for the 4 pieces versus 4 pieces subset of the 8-piece databases.

3.4 Disk I/O

Table 3 shows that the wall clock time is dominated by the I/O-intensive phases. The captures, lookups, and verify phases all sequentially proceed through the data. However, each may result in a (usually small) search to resolve the value of the position by looking up values in previously computed databases. This search is a consequence of the data compression scheme used (which removes the value for any capture and threatened capture position). The alternative was to keep the uncompressed data on disk and use that instead. This was not done because of the possibility of introducing an error; the values based on I/O operations (e.g., capture positions) have not been verified for correctness. Rather than trust unverified data, we preferred the (slower) use of the compressed data.

The capture phase runs quite quickly. Surprisingly, typically over 60% of the positions get resolved in this phase. Each position has slightly more than one legal capture move per position. The remaining positions need to have a lookup performed. These positions average roughly 3 moves by the leading checker(s), each of which has to be looked up. Each of these searches is, on average, considerably more expensive than a simple capture position. Thus, even though the lookup resolves only typically 10-15% of database, it runs slower than the captures phase because of the increased amount of I/O.

The database calculation has been set up so that each position has I/O performed on it a maximum of two times. Capture positions are visited only in the captures phase; they are not included in the final compressed database, so no verification has to be done. All the remaining positions may have to have I/O done twice: once to do a lookup of any leading checker moves, and once to verify the position value if there is no threatened capture.

The databases have been organized to increase data locality. Databases that are likely to lead into one another are located physically close to each other on disk. As well, the program maintains its own internal disk paging, allowing the program to prioritize the database pages kept in memory. The result is that the program, using 200 MB of page buffers, ends up doing one disk I/O for an average of 500 database position value requests. In other words, the hit rate is 499/500.

I/O could be significantly reduced if the database construction program used slices selectively. For example, some of the databases are relatively small, and slicing them into 49 pieces incurs a lot of unnecessary overhead. These databases could be constructed as one big computation. For example, the 1045 database has only 45 billion positions—

using roughly 10.5 GB. Rather than slicing this piece into 42 slices—each with a lookups phase—the entire 1045 sub-database could be done as a single computation. Then the lookups would only be required for part of the database—where there was a leading checker on the 7th rank. This has not been done.

It may seem that the non-captures phase should require the most computational effort, given that this phase must make repeated passes over the data. Further, some of databases are too large to be resident in RAM, requiring costly disk paging. Fortunately, this was not a problem in our implementation. The non-captures phase was set up so that references to values in other databases (requiring I/O operations) were not needed. The position indexing scheme was organized to facilitate spatial and temporal locality. This allowed a (relatively) small working set of data to be resident in memory during the non-captures phase. This was facilitated by having an internal paging mechanism, allowing the program to take advantage of application-dependent properties to minimize the I/O. On our machines, 200 MB of RAM was allocated for pages. With this, we have been able to complete the non-captures phase on files as large as 25 GB in only a few days.

It is interesting to note that the profile of the database computation has changed significantly since we did this work in the early 1990s. Some parts of the program that were previously I/O bound are now CPU bound (more memory to eliminate costly I/O), while other parts that were CPU bound are now I/O bound (CPU speed has improved more than disk speed). This meant that we had to re-profile the program and use additional optimization techniques.

3.5 Errors

Given that this computation takes many CPU years to run and terabytes of data transferred from and to disk, it is critical that an error not be allowed to creep into the calculation. An error early on in the computation, for example, may result in the entire calculation having to be repeated. For example, in October 2001, Gil Dodgen and Ed Trice calculated the 8-piece databases. We compared the CHINOOK results with theirs and discovered a difference in the 7-piece results (Dodgen and Trice, 2002). It eventually turned out that the CHINOOK databases were wrong (a few thousand positions). However, even with the error *the databases still passed all our verification tests*! This may seem strange, but it can happen. The computed data can be internally consistent, but wrong. The best way to verify the correctness of the databases is to have them independently computed and then the results compared—as

we did with the Dodgen/Trice data.[3] Needless to say, we are hoping that this experience is not repeated with our 9- and 10-piece calculations.

During the course of the calculations, we had to contend with a faulty CPU, bad memory, a disk crash, network errors and operator errors. In some cases, these errors were trivial to spot (dead disk), while others proved more subtle (faulty memory chip). Several precautions were taken to reduce the likelihood of introducing an error into the computation:

1 All calculations for all phases were logged. This was useful if a post-mortem was needed to identify the reason(s) for a computation failure.

2 All data copied over a network was verified. The source and destination files had a cyclic redundancy check (CRC) value computed, and the two had to match. In practice, most copies worked correctly. However, at least once a month the CRC check would fail signaling a copy error. This allowed us to redo the copy and prevent an error from being introduced.

3 The database files were augmented with a 32-bit CRC number for each block of 1024 bytes. Whenever a disk read (local or over the network) was performed, the data read would be verified for consistency with the CRC number. This enhancement allowed us to find a subtle bug in the program, and occasionally would uncover a read failure that was not reported by the operating system.

4 All data computed—databases in their original and compressed form—were archived to tape. Thus, if a catastrophic event occurred (e.g., an error was discovered in the early part of the computation), we would be able to recover by repairing the faulty data rather than having to re-compute it from scratch. The need to retrieve data from tape occurred only once.

Despite all the above precautions, occasionally the computation of a database slice failed to verify, even though the logs showed no record of any error occurring.

Are the databases 100% correct? We do not know, but hope that someone will soon repeat our calculations and confirm the correctness of our results.

3.6 System Issues

For the checkers computation, keeping many machines 100% busy is a difficult task. It is complicated by the calculation dependencies

(some databases must be computed before others), hardware specialization (run I/O-intensive jobs on machines with fast disks; run CPU-intensive jobs on machines with fast processors), and disk management (transferring files; making sure that disks do not fill up). We developed tools that can automate most of the computation dependency and hardware specialization issues (Goldenberg et al., 2003). However, managing the data turned out to be labour intensive and a source of potential errors. We were unable to find or build a usable tool that could properly manage the data file dependencies, taking into account disk space constraints, in such a way as to maximize throughput. This appears to be a very difficult problem, but one that needs to be solved if one is to fully automate data-intensive computations such as building checkers endgame databases.

4. Results

This section discusses the results of computing all the 9-piece databases and the 5 pieces versus 5 pieces subset of the 10-piece databases.

4.1 Computation

Table 4 shows the sizes of the databases computed.[4] Obviously the more lop-sided the material balance in the computation (e.g., 6 pieces versus 3), the less useful it will be in practice (for CHINOOK) and for solving the game. 13.1 trillion positions have been computed. We claim that this is the largest endgame database (in terms of number of positions) yet computed for any game.

The computation took 18 months. The 9-piece computation began in November 2001 and the 10-piece in January 2002. These computations ended in June 2003. Most of the work was completed on dual-processor AMD machines. The memory used ranged from 1 to 4 GB. Older, slower (800 MHz) computers were used to pre-compute the captures phase of the computation. The lookups, non-captures, and verification phases were done using an average of 3 machines, with an average speed of 1.5 GHz. All phases used both processors to speed up the computation.

We had infrequent access to a 64-processor SGI O3000 (500 MHz) with 32 GB of RAM. The machine was used to run the non-captures phase of many of the largest database slices. The database program was parallelized using POSIX threads so that the range of positions could be equally divided between the processors and computed in parallel. The largest computation (171 billion positions) took 2.3 days of SGI time to resolve. The length of time was due to the relative slowness of the

Table 4. Databases completed.

Num Pieces	Pieces/Side	Size	Total Completed
1	1 – 0	120	120
2	2 – 0	3,484	
	1 – 1	3,488	6,972
3	3 – 0	65,192	
	2 – 1	196,032	261,224
4	4 – 0	883,458	
	3 – 1	3,546,384	
	2 – 2	2,662,932	7,092,774
5	5 – 0	9,237,424	
	4 – 1	46,409,320	
	3 – 2	93,041,488	148,688,232
6	6 – 0	77,526,288	
	5 – 1	467,999,856	
	4 – 2	1,174,279,692	
	3 – 3	783,806,128	2,503,611,964
7	7 – 0	536,417,856	
	6 – 1	3,782,903,904	
	5 – 2	11,404,950,960	
	4 – 3	19,055,258,760	34,779,531,480
8	8 – 0	3,118,957,920	
	7 – 1	25,172,147,520	
	6 – 2	88,657,111,920	
	5 – 3	177,982,456,720	
	4 – 4	111,378,534,401	406,309,208,481
9	9 – 0	15,455,930,880	
	8 – 1	140,531,639,040	
	7 – 2	566,442,589,440	
	6 – 3	1,328,448,083,840	
	5 – 4	1,997,749,399,776	4,048,627,642,976
10	10 – 0	65,975,569,920	
	9 – 1	0	
	8 – 2	0	
	7 – 3	0	
	6 – 4	0	
	5 – 5	8,586,481,972,128	8,652,457,542,048
Total			13,144,833,586,271

processors (500 MHz) and the number of passes over the data that were required to resolve all the positions.

The total amount of computing done is difficult to estimate given that a varying number of machines were used, with different number of processors, and with differing processor speeds. Normalized to a 1.5 GHz processor, a ballpark estimate is that the complete 2 through 9-piece databases and the 5 versus 5 piece subset of the 10-piece databases required 15 CPU years of computing.

Since a few of the 6 versus 4 piece database slices have been computed (low priority on a single machine), we could actually start computing the 11-piece database (6 versus 5 subset). This computation is roughly 10-fold bigger (117 trillion) than what has already been accomplished. We will not pursue this unless we determine that the 10-piece databases are not sufficient for solving the game of checkers in a reasonable amount of time.

4.2 Statistics

Because of the concurrency used in the non-captures phase (2 processors would iterate on a slice in parallel), it is hard to know the exact number of ply required to resolve a slice. There were some slices that needed over 180 iterations to resolve, a lower bound that is probably very close to the actual number. Consider what this number means. There were slices where over 180 ply were needed before a capture could be forced *or* the leading checker could safely advance one square. In the latter case, one wonders how many more ply would be needed to win the game once that checker had safely advanced a single square—it could be huge! This gives rise to the speculation that there are 10-piece positions that may require many hundreds of ply to solve. For example, Gil Dodgen and Ed Trice have built a perfect-play 7-piece database, and they report the longest win (against best play) to be 253 ply (127 moves) (Trice and Dodgen, 2003). There must be 10-piece positions that are considerably longer than that.

The previous discussion illustrates the disadvantage of computing only W/L/D values. A checkers program such as CHINOOK could reach a 10-piece position and not know how to win it. The search could flounder, not being able to choose between winning moves to find a quick path to victory. The (real) danger is that the program will end up cycling around, not knowing how to make progress (although this has not been seen in practice).

4.3 Solving Checkers

The total possible search space for the game of checkers is 5×10^{20} (see Table 5)—a daunting number. But how much of this search space has to be explored to solve checkers?

Three assumptions can be used to get a rough upper bound on the effort required to solve checkers. The following heuristics are used to identify the key search space for the proof tree; parts that are excluded may be needed in the case of proving trivially won positions.

- **Material Balance:** An advantage of 2 or more pieces is huge; equivalent to roughly a rook or more in chess. It seems reasonable to assume that a proof would not have to go through positions with lop-sided material. The useful positions are those where the material balance is even, or one side has a single piece advantage.

- **King Balance:** One side having 3 or more kings than the other rarely occurs in practice. Hence we limit the search space of interest to those subsets where the number of kings for each side differs by at most 2.

- **Number of Kings:** Kings only appear on the board later in the game. For example, although it is theoretically possible to have 24 pieces on the board with one of them being a king, this scenario is highly contrived. A reasonable assumption is to limit the number of kings to being 6 when there are 10 or less pieces on the board, 4 with 12 or more pieces, 2 with 14 or more pieces, and zero with 24 or less pieces.

Table 5 shows the results of applying the above assumption. From $O(10^{20})$ the potential search space drops to $O(10^{14})$. Of this, the databases computed thus far represent roughly 7.5 trillion—5% of the reduced search space. It is too early to know the full impact of the 10-piece databases in the checkers proof.

5. Conclusions

Disks are getting larger and cheaper; terabyte systems are affordable and petabyte systems exist. Moore's law continues to hold and multiprocessors systems are ubiquitous. RAM is inexpensive, and hardware and operating systems are gradually moving to accommodate large memories. In effect, there is no technological limit to pushing database technology to even greater heights. The endgame databases reported here contain over 10^{13} data points, a 30-fold increase over what seemed pos-

Table 5. Reducing the checkers search space.

Pieces	Database Size	Plausible Bound
1	120	120
2	6,972	3,488
3	261,224	196,032
4	7,092,774	2,662,932
5	148,688,232	89,972,128
6	2,503,611,964	759,865,120
7	34,779,531,480	17,681,009,520
8	406,309,208,481	103,706,534,351
9	4,048,627,642,976	1,551,749,730,336
10	34,778,882,769,216	5,862,356,551,488
11	259,669,578,902,016	21,456,015,775,392
12	1,695,618,078,654,976	46,262,266,685,096
13	9,726,900,031,328,256	22,268,142,277,920
14	49,134,911,067,979,776	29,879,692,089,280
15	218,511,510,918,189,056	802,158,318,720
16	852,888,183,557,922,816	723,777,011,100
17	2,905,162,728,973,680,640	2,169,968,941,008
18	8,568,043,414,939,516,928	1,527,822,346,512
19	21,661,954,506,100,113,408	3,587,090,153,856
20	46,352,957,062,510,379,008	1,959,596,777,424
21	82,459,728,874,435,248,128	3,564,284,669,088
22	118,435,747,136,817,856,512	1,489,690,180,992
23	129,406,908,049,181,900,800	2,057,391,420,240
24	90,072,726,844,888,186,880	641,335,986,590
Total	500,995,484,682,338,672,639	145,925,579,158,733

sible a decade ago. High-end technology that is available today could be used to push this to 10^{14}.

The reason for computing the 10-piece databases was to solve the game of checkers. The databases eliminate the bottom of the search tree. A separate project is building the top of the proof tree, searching forward from the root towards the databases. When the two search frontiers meet, checkers will be solved. At this point in time, it is too early to tell how soon this will happen.

6. Acknowledgments

This work benefitted from the frequent email interactions with Ed Trice and Gil Dodgen. Their independent calculation of the 8-piece database uncovered an error in the CHINOOK databases. It was better

to find this error *before* starting the 9- and 10-piece calculation than afterwards!

This research was supported by grants from the Natural Sciences and Engineering Council of Canada (NSERC) and Alberta's Informatics Center of Research Excellence (iCORE). This project used resources funded by the Canada Foundation for Innovation, including the Multimedia Advanced Computational Infrastructure (MACI) resources at the University of Alberta. The efforts of the Software Systems Group and the Hardware Support Group of the Department of Computing Science, University of Alberta, are greatly appreciated.

Notes

1. There are over 100 checkers variants. The variant used here is played on an 8×8 board and is popular in the former British Commonwealth and in North America. So-called International Checkers is played on a 10×10 board and is popular in Russia, Europe, and Africa.

2. The original program had a sixth phase (Winning Moves) that was not used. The benefits of including this phase were small and they did not out-weight the additional coordination required in the computation.

3. We are aware of another effort to compute the 9-piece databases and (apparently) the 10-piece databases. We have made two offers to exchange information with this party so that the correctness of both of our efforts could be verified. The offers have been declined.

4. Note that some 6 piece versus 4 piece slices have been computed.

References

Allis, V. (1994). *Searching for Solutions in Games and Artificial Intelligence*. PhD thesis, Department of Computing Science, University of Maastricht.

Dodgen, G. and Trice, E. (2002). Personal communication.

Gasser, R. (1996). Solving Nine Men's Morris. *Computational Intelligence*, 12:24–41.

Goldenberg, M., Lu, P., Pinchack, C., and Schaeffer, J. (2003). TrellisDAG: A System for Structured DAG Scheduling. In *9th Workshop on Job Scheduling Strategies for Parallel Processing*. To appear.

Lake, R., Schaeffer, J., and Lu, P. (1994). *Advances in Computer Chess 7*, chapter Solving Large Retrograde Analysis Problems Using a Network of Workstations, pages 135–162. University of Limburg.

Lincke, T. and Marzetta, A. (2000). Large endgame databases with limited memory space. *Journal of the International Computer Games Association*, 23(3):131–138.

Romein, J. and Bal, H. (2002). Awari is solved. *Journal of the International Computer Games Association*, 25(3):162–165.

Romein, J. and Bal, H. (2003). Solving the game of awari using parallel retrograde analysis. *IEEE Computer*. To appear.

Schaeffer, J. (1997). *One Jump Ahead: Challenging Human Supremacy in Checkers*. Springer-Verlag.

Trice, E. and Dodgen, G. (2003). The 7-piece perfect play lookup database for the game of checkers. *Advances in Computer Games 10*. To appear.

Turpin, A. and Moffat, A. (1995). Practical length-limited coding for large alphabets. *The Computer Journal*, 38(5):339–347.