Parallelizing a Simple Chess Program

Brian Greskamp
ECE412, Fall 2003
E-mail: greskamp@crhc.uiuc.edu

Abstract

In a closely watched match in 1997, the parallel chess supercomputer Deep Blue *defeated then world champion Garry Kasparov* $3\frac{1}{2}$ *to* $2\frac{1}{2}$*. The fruits of Moore's law combined with steadily improving chess algorithms now allow programs such as* Deep Fritz *to challenge human grandmasters while running on nothing more than a commodity 4-way SMP workstation. This paper provides a short overview of the parallel chess algorithms that help achieve these feats and details the implementation of a simple parallel chess program. It also muses about possible methods for extracting additional parallelism using reconfigurable hardware.*

1. Introduction

Computer chess has been a compelling research topic since the very inception of the artificial intelligence field. Claude Shannon's seminal 1950 paper [12] made crucial early contributions, including an evaluation function for quantifying the quality of a given board position. The alpha-beta search strategy, due to Newell *et. al.* in 1958 [10], was another groundbreaking development. Although newer search algorithms have been proposed, some with great success, the venerable alpha-beta algorithm remains at the heart of most implementations. Augmenting alpha-beta with heuristic enhancements and knowledge databases has extended its power considerably. Of course, improved computer hardware has also contributed to the growing strength of computer players. The shift toward parallel machines in the 1980s and 90s saw many parallel chess implementations mapping alpha-beta to both shared and distributed-memory systems. *CilkChess* and **Socrates* demonstrated scalability on systems with 1000 nodes or more [6], while recently *Deep Fritz* has achieved grandmaster strength on as few as four processors.

This paper investigates the most basic serial chess algorithms and techniques for parallelizing them. It also presents an extremely simple implementation aimed at small-scale, commodity SMP workstations. Preliminary performance results are provided, and shortcomings of the current system are evaluated. The paper is organized as follows: section 2 introduces the basic theory and algorithms of game tree searching and briefly discusses how they might be parallelized. In section 3, implementation details for a simple parallel search engine are discussed. Some preliminary results and analysis follow in section 4. Next, section 5 covers unfinished work and suggested enhancements, including the use of reconfigurable hardware. Finally, section 6 concludes.

2. Background

This section provides a brief introduction to game tree searching and the all-important alpha-beta algorithm as well as some techniques for parallelizing it. Also included are some enhancements to alpha-beta that any credible chess program must implement. Later, these enhancements will raise some difficult design questions in the parallel engine.

2.1. Game Tree Search

Fundamental to chess and any other turn-based perfect-information game is the game tree. The tree represents all possible paths through the game's state space. Each node in the tree is a game state (*i.e.* a board configuration), and each arc represents a legal move for the player on-move at that state. The example in Figure 1 shows a portion of the game tree for Tic-Tac-Toe. An important characteristic of the tree is its branching factor, the average number of out-edges per non-leaf node. In chess, the branching factor is approximately 32 [8]. It is also important to note that for many games, including chess, the game "tree" is in reality a directed acyclic graph, where different sequences of moves can transpose into the same board position. However, tree-based analysis of such a graph is facilitated by cloning those nodes that have in-degree greater than one to create a tree where all nodes except the root have in-degree exactly one.

In order to choose its moves, a game-playing program searches for a path through the tree that terminates with the

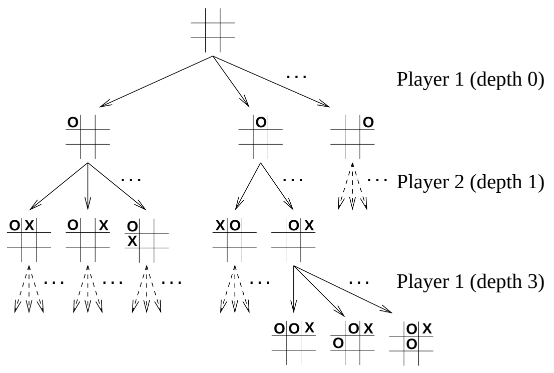

Figure 1. A fragment of the Tic-Tac-Toe game tree.

most favorable result. In performing this search, it is fundamental to assume that the program's opponent will also be searching for its best path and that both sides agree on the valuation of the leaf nodes. In other words, the program must assume best play by both sides. Leaves with low (or negative) value are desirable outcomes for one player (the "minimizing player"), while those with high value are desirable for the other (the "maximizing player"). A function called `Evaluate` is provided to determine the value of each leaf. Assuming that each player will attempt to respectively minimize or maximize its score at every turn leads directly to the recursive algorithm of Figure 2. This simple strategy, called MiniMax due to its alternating minimizing and maximizing phases, examines every node in the game tree and is guaranteed find an optimal solution.

2.2. Alpha-Beta Pruning

Fortunately, it is usually possible to achieve the same result as `MiniMax` while searching far fewer nodes. For a tree with branching factor b, the *alpha-beta pruning* technique [10] reduces the effective branching factor to $O(\sqrt{b})$ on average [7], allowing trees of depth $d \times 2$ to be searched in the same time MiniMax would require to search to depth d. The alpha-beta algorithm works by "cutting off" the search at a position if it becomes evident that the opponent would never allow the player to reach that position because its value is too great.

At each step the search function retains the best score it has found so far. When the maximizing player is searching a node and finds a path from that node having a higher score than the minimizing player's best score, then he knows the node from which he is currently searching is 'too good to be true' since the minimizing player will never give him the opportunity to play the move to that node. At that point, the search is said to "fail high" and the player can stop examining the current node. Likewise, if during his search, the minimizing player finds a path score that is less than the maximizing player's best score, then he can abandon searching that branch since the maximizing player will never allow him to move there. A straightforward implementation of `AlphaBeta` is given in Figure 3. In practice, the equivalent code from Figure 4 is preferred. Negating the result of the recursion at each step eliminates the need to handle the minimizing and maximizing cases separately.

```
int MiniMax(Node cur_node)
{
  if(cur_node.is_leaf)
    return Evaluate(cur_node);

  if(cur_node.type == MIN)
    best_score = INFINITY;
  else
    best_score = -INFINITY;

  while(HasMoreSuccessors(cur_node)) {
    succ_node = GetNextSucc(cur_node));
    score = MiniMax(succ_node);
    if(cur_node.type == MIN) {
      if(score < best_score)
        best_score = score;
    }
    else {
      if(score > best_score)
        best_score = score;
    }
  }
  return best_score;
}
```

Figure 2. The MiniMax algorithm.

In order to use the `AlphaBeta` routine, one invokes it on the root of the search tree with arguments $\alpha = -\infty$ and $\beta = \infty$, which correspond to the worst possible scores for the maximizing and minimizing players respectively. It can then be shown that for every call in the recursion, $\alpha < score \leq \beta$ where $score$ is the return value. Commonly, an additional *depth* argument is added so that the recursion can be stopped and the evaluation function applied at a preset non-leaf level in the tree, because searching all 30 to 50 levels in a typical chess tree would be far too costly. A final note about real alpha-beta implementations is that they must of course return the actual move that leads to the best score, along with the score itself.

2.3. Heuristic Enhancements

It is important to note that the order in which branches are evaluated under `AlphaBeta` has a major impact on running time. If at each ply the best moves are evaluated first, traversal simply proceeds down the left side of the tree, following the best line, and searches of the remaining sub-

```
int AlphaBeta(Node cur_node, int alpha,
              int beta)
{
  if(cur_node.is_leaf)
    return Evaluate(cur_node);

  if(cur_node.type == MIN)
    best_score = beta;
  else
    best_score = alpha;

  while(HasMoreSuccessors(cur_node)) {
    succ_node = GetNextSucc(cur_node));
    if(cur_node.type == MIN) {
      score = AlphaBeta(succ_node, alpha,
                        best_score);
      if(score <= alpha)
        return alpha;
      if(score < best_score)
        best_score = score;
    }
    else {
      score = AlphaBeta(succ_node,
                        best_score, beta);
      if(score >= beta)
        return beta;
      if(score > best_score)
        best score = score;
    }
  }
  return best_score;
}
```

Figure 3. The alpha-beta algorithm.

```
int AlphaBeta(Node cur_node, int alpha,
              int beta)
{
  if(cur_node.is_leaf)
    return Evaluate(cur_node);

  while(HasMoreSuccessors(cur_node)) {
    succ_node = GetNextSucc(cur_node);
    score = -AlphaBeta(succ_node, -beta,
                       -alpha);
    if(score >= beta)
      return beta;
    if(score > alpha)
      alpha = score;
  }
  return alpha;
}
```

Figure 4. The standard alpha-beta formulation.

trees quickly cut off. If on the other hand, the worst moves are evaluated first, then no cutoffs occur and the algorithm performs no better than `MiniMax`. This fact has given rise to a number of heuristics for estimating which moves are best in order that they can be tried first.

Iterative Deepening Iterative deepening is a technique wherein the tree search is first attempted with a shallow depth in order to obtain score estimates for each branch at the root node. Subsequent searches to greater depths can use the score estimates from previous iterations for move ordering. Iterative deepening is also applied in conjunction with a technique called “aspiration”, which allows for the selection of smaller search windows based on the node’s score from the preceding iteration.

History Table The history table is a small hash table that assists move ordering at all levels in the tree. When search completes at each node, the best move from that node has been found. The from-square and to-square of the best move serve as indices in the history table, and a counter for the corresponding entry is incremented. Before searching the moves at each node, the search routine first counsults the counter value in the history table entry for each available move. The routine examines moves with higher counts first, since those moves have been good at other nodes in the tree and they are likely to be good in the current position as well.

Transposition Table As discussed in subsection 2.1, the game tree for chess is in reality a DAG. Thus, positions which have already been searched are likely to be reached again. The transposition table memoizes the results of searching each node. Before searching any node, the search routine examines the transposition table. If the move has already been searched, the routine simply returns the memoized result without further consideration.

2.4. Parallel Alpha-Beta

In order to parallelize the search, it is necessary to assign processors to simultaneously search multiple subtrees of the game tree. The question of how to partition the tree is deceptively complex. One of the earliest and most intuitive techniques is *principal variation splitting* [9]. At each node, it first recursively searches the left-most branch to find an α-bound for the remaining branches which can then be searched with parallel invocations of `AlphaBeta`. For example, Figure 5 shows how PVSplit partitions a shallow tree between two processors. Pseudocode for `PVSplit` appears in Figure 6.

An apparent error in the `PVSplit` code is that a cross-iteration dependence on `alpha` still exists in the parallel loop. However, as long as the value of `alpha` used in each

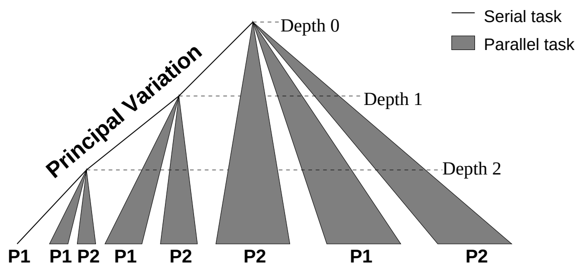

Figure 5. Parallel and sequential tasks in PVS-plit.

```
int PVSplit(Node cur_node, int alpha,
            int beta)
{
  if(cur_node.is_leaf)
    return Evaluate(cur_node);

  succ_node = GetFirstSucc(cur_node);
  score = PVSplit(cur_node, alpha,
                  beta);
  if(score > beta)
    return beta;
  if(score > alpha)
    alpha = score;

  //Begin parallel loop
  while(HasMoreSuccessors(cur_node)) {
    succ_node = GetNextSucc(cur_node);
    score = AlphaBeta(succ_node, alpha,
                      beta);
    if(score > beta)
      return beta;
    if(score > alpha)
      alpha = score;
  } //End parallel loop

  return alpha;
}
```

Figure 6. The PVSplit algorithm.

iteration is less than or equal to the value that would occur in the sequential version, the iterations can execute independently and the result will be correct. Nevertheless, keeping `alpha` as up-to-date as possible is critical for performance. Whenever one of the parallel iterations completes, it updates `alpha` if it has found a new best score. When the next iteration following the update begins, it reads the updated value. In the case that the best move is always searched first, the parallel algorithm visits the same number of nodes as the sequential algorithm, since α does not change during the parallel search. In practice, the parallel algorithm usually visits more nodes that the sequential one as a consequence of the weaker α-bound given to each iteration.

Another shortcoming of `PVSplit` is that *all* of the parallel searches must complete before the current `PVSplit` invocation may return. This leads to inefficient processor usage when most of the search threads have completed. Some processors will remain idle while the last few processors finish their searches. One solution is to implement work-stealing, and experiments indicate that the benefit is significant. The *Younger Brothers Wait Concept* [2] achieves speedups of over 140 on 256 nodes, and the previously-mentioned *CilkChess* uses the *Jamboree* strategy to achieve similar speedups. In contrast, the reported speedup for `PVSplit` is a modest 3.0 on four processors, with an asymptotic limit of approximately five [4]. For the purposes of this paper, the low asymptotic speedup is not a major concern.

3. Implementation

In order to implement PVSplit in the time allotted, it was necessary to build upon a simple, serial chess engine. For this purpose, version 1.2 of *mscp*, "Marcel's Simple Chess Program" was an ideal choice. The parallel varsion is referred to as *mscp-par*. Written by Marcel van Kervinck, *mscp* includes search enhancements such as transposition and history tables. It also uses iterative deepening and applies null-window and aspiration searching [11] in a limited way, but its most attractive feature is that the entire C language implementation comprises only 2000 lines of code. Furthermore, it is freely available under the GNU *General Public License*. The relative simplicity of *mscp* made it preferable to much stronger but more complex programs including *GNUChess* and *Crafty*. Still, there is little doubt that the more advanced serial programs would handily defeat *mscp-par*.

3.1. Architectural Considerations

Since the target machine is an SMP, it can be assumed that each processor has equally fast access to a single shared memory. Communication between parallel tasks takes place using the familiar threaded model and its associated synchronization primitives. Large shared data structures such as the transposition table can simply reside in the shared memory. Targeting a distributed-memory machine would complicate the design. Although most parallel search algorithms do not require much communication bandwidth for the basic alpha-beta search, shared data structures such as the history and transposition tables must be physically distributed across the nodes.

3.2. Threading Model

The parallel search is built on the POSIX threads ("pthreads") API. The pthreads library is available on nearly all UNIX-like systems, and most implementations are capable of spreading threads across an SMP system's multiple CPUs. For optimal search efficiency, one thread should exist per CPU. Although there is no way to specify a mapping between threads and CPUs, it is reasonable to assume that each thread will quickly migrate to a different CPU assuming the system is otherwise un-loaded.

The actual *mscp-par* code is kludgy and difficult to follow in part because it includes two separate and slightly different parallel search routines. They remain separate because searches at the root node behave slightly different from searches at other levels. For the sake of clarity, the following explanation assumes a single search routine, the familiar `PVSplit`. With this simplification, the threads operate as follows: For each `PVSplit` invocation, the main thread first serially searches the branch having the highest history table score. Then, before entering the parallel search loop, the main thread atomically pushes all remaining legal moves (except the first one which has already been searched serially) onto a worklist. It then creates n worker threads, where n is the number of system processors and calls `pthread_join` on each one to wait for their completion.

Each worker then begins consuming moves from the worklist and searching them one at a time until all moves in the list are exhausted. A mutex ensures atomic accesses to the list for all threads. Whenever a worker consumes a move from the list, it atomically reads the current value of `alpha` from another mutex-protected shared data structure. Next, it invokes `AlphaBeta` to search the move. If the search returns a new best score, the worker atomically updates the shared value of `alpha` so that the next thread to consume a move will see the new value. When a worker reaches the end of the list, it returns. When all worker threads have terminated, the main thread's call to `pthread_join` compltes and `PVSplit` returns.

3.3. Reentrancy and Shared Data

Most serial chess engines are not designed to be reentrant, and *mscp* is no exception. Almost every procedure updates a global variable, and the game state (board configuration, move stack, *etc.*) are all global. A quick fix for this problem is to encapsulate all of the program's global variables in a new C `struct` and pass a pointer to the new type into each function that formerly accessed globals. Now multiple procedures may be active simultaneously as long as each operates on a different instance of the structure. This approach led to the creation of a structure called `game_state`, which is copied into each new thread that is created. Each thread then has a private copy of the board state.

Data structures that are shared between parallel invocations of the search function, such as the transposition and history tables, remain global. A read-write lock (using the `pthread_rwlock` functions) governs access to the transposition table. Synchronization is crucial since corrupt data in the transposition table can corrupt the board state. Whenever the search routine does a transposition table lookup, it will compare a hash tag stored with the entry to a hash of the current board position. A match indicates that the current board position is *definitely* the one in the table. The search routine will then return the memoized `best_move` field, which might actually be played. If the `best_move` field was corrupted, the move may be illegal and result in an illegal board state!

Although some type of synchronization appears necessary, locking is extremely costly because the transposition table is read every time a move is searched and it is updated with the best move and score at the end of each non-leaf call to `AlphaBeta`. Consequently, the ratio of reads to writes is approximately equal to the branching factor of the search tree. The resultant lock contention will be quantified in the experiments of section 4. In contrast, no synchronization whatsoever is applied to the history table. Whereas the integrity of the transposition table can affect program correctness, the integrity of the history table affects only running time. At worst, a corrupt history table can cause bad moves to be searched before good ones.

4. Results

This section presents preliminary performance and scalability results and evaluates the effectiveness of the synchronization decisions reached in the previous section. Unfortunately, data are only available for an older version of the engine which contained a major error: Searches at the root node proceeded in the correct order (according to the iterative deepening process), but parallel searches further down the tree searched their subtrees in *pessimistic* order. The newest version corrects this problem, but suffers from unresolved concurrency bugs so meaningful numbers for it are not yet available. Therefore, the numbers reported here reflect the performance of searching the left subtree of the root serially then searching the remaining branches from the root in parallel. Since the program expends about $\frac{1}{3}$ of its CPU time searching the first subtree of the root, the maximum possible speedup is 3. Even assuming linear speedup on the remaining subtrees, one would expect only $(\frac{1}{3}+\frac{2}{3p})^{-1}$ speedup on p processors or 2 on four processors.

Two test cases are provided, both of which executed on a Sun SMP system running SunOS 5.9 on 32 UltraSPARC-III processors. In the first case, the human player makes the nonsensical opening move `a3` and records the time elapsed before the computer completes a depth 8 search and responds with `Nf6`. The running times for the unmodified, serial mscp-1.2 program appear under the label "original" in Table 1 alongside times for the parallel version with varying numbers of threads. The second case, shown in Table 2 tests the equally senseless sequence `a3 Nf6, b4 e6, Bb2 a5`. Note, however, that the stupidity of the sequence makes no difference to the `AlphaBeta` algorithm, which does not incorporate any positional knowledge.

Program	Processors	Time (m:ss)	Speedup
original	1	0:54.7	1
parallel	1	0:56.4	0.97
parallel	2	0:36.0	1.5
parallel	4	0:28.6	1.9
parallel	8	1:24	0.65

Table 1. Time to respond to opening `a3`.

Program	Processors	Time (m:ss)	Speedup
original	1	5:19	1
parallel	1	5:56	0.90
parallel	2	3:37	1.5
parallel	4	2:53	1.8

Table 2. Time to respond to opening `a3 b4 Bb2`.

Despite the major flaw, these intermediate results do provide some valuable information. First, it is clear that the parallelized code degrades performance only slightly. As seen in Table 1, speedup scales roughly as expected (given the broken move ordering) up to four processors, then falls quickly below unity. Between four and eight processors lies the point where transposition table lock contention comes to dominate the computation. Table 3 shows the amount of program execution time spent in "system" mode as reported by the UNIX `time` utility. It is reasonable to assume that almost all of the system time and some proportion of the user time is due to lock contention. The contention increases roughly exponentially with the number of processors. Clearly, an alternative to transposition table locking is necessary for greater scalability, but since `PVSplit` does not scale well past four processors anyway, this is not a major issue. Nevertheless, the following section does discuss some alternatives to the current locking scheme.

Processors	User CPU (s)	System CPU (s)
4	80	8
5	97	27
6	110	66
7	114	100
8	130	168

Table 3. Transposition table lock contention.

5. Future Work

Resolving the concurrency bugs in the latest version of *mscp-par* that corrects the move-ordering problems is a first priority. Porting the `PVSplit` algorithm to a stronger program such as *Crafty* is the next logical step. Beyond that, it is interesting to consider changes to facilitate better scalability. Clearly, the transposition table locking is a limiting factor. Fortunately, this issue has been well studied; one innovative proposal for cluster implementation [1] has even investigated reprogramming each node's network interface card to implement a fast distributed storage mechanism. As for shared-memory implementations, the authors of *CilkChess* ignore transposition table locking, just as *mscp-par* does for the history table. With an extremely large hash table (32GB), they find that conflicts are rare. Without locking, one can simply test the legality of any move read from the table before playing it to ensure that board corruption does not result. Another lockless approach that elegantly solves the corruption problem is due to Robert Hyatt [5]. It hashes the data in each entry such that corruption can be detected without the need to test the legality of the retrieved move.

Taking a much broader view, it would be interesting to investigate *automatic* methods for uncovering parallelism in the chess search. Since parallelizing alpha-beta reqired relaxing true data dependencies, it would be extremely difficult for compilers to produce parallel code from the sequential alpha-beta specification. On the other hand, a good deal of parallelism is hidden in the `Evaluate` function which assigns scores to board positions. A typical evaluation function loops over each piece on the board, computing threats, mobility, and board control. These functions account for approximately half of *mscp*'s execution time.

Interestingly, the evaluation functions can be implemented very efficiently in hardware as bit-vector operations — a fact which *Deep Blue's* 256 custom ASICs exploited [3]. The evaluation function is further suited for implementation in a hardware coprocessor because a relatively small amount of data must be communicated between the evaluation function and the rest of the (software) program, but large amounts of computation take place on that data. The entire board state can be transferred in 24 bytes, while the evaluation function may execute the equivalent of tens of thousands of general purpose instructions on that data to return only a one-byte score.

Unfortunately, great effort is usually required to design such hardware. Luckily, the relatively new field of Reconfigurable Computing (RC) might provide a simpler way to realize the benefit of custom hardware. One goal of RC is to *automatically* generate hardware designs from high-level source code and install them in reprogrammable logic (*i.e.* FPGAs) where they can assist a traditional CPU in computation. Numerous efforts, including the *Amalgam* project from UIUC have produced tools capable of generating such hybrid hardware-software implementations from C source. In fact, one of *Amalgam*'s demo applications is a solution to the n-Queens problem, a simple chess puzzle that computes attacks among several queens on the board.

6. Conclusions

Thus far, the development of *mscp-par* has been somewhat of a toy project; it is certainly not breaking any new ground. In fact, it is not even working correctly yet. One small benefit of the project is the development of a simple, freely available, albeit very weak, parallel chess program. Additionally, studying chess tree search has suggested some future investigations which *would* contribute new results. Specifically, the prospect of automatically extracting parallelism from the chess program's `Evaluate` routine is compelling. It is already known that more powerful evaluation functions significantly increase the effectiveness of the search. With the nearly unbounded parallelism available in reconfigurable hardware, it may be possible to execute these more complex functions even faster than executing the simpler function in software. Thus, although *mscp*'s evaluation function only accounts for half of the compute time, huge speedups and stronger play might be realized with more thoughtful evaluation functions.

References

[1] R. A. F. Bhoedjang, J. W. Romein, and H. E. Bal. Optimizing distributed data structures using application-specific network interface software. In *ICPP*.

[2] R. Feldmann, B. Monien, P. Mysliweitz, and O. Vornberger. Distributed game tree search. 12(2):65–73.

[3] F. Hsu. IBM's deep blue chess grandmaster chips. 19(2):70–81, March 1999.

[4] R. M. Hyatt. The dynamic tree-splitting parallel search algorithm. 20(1):3–19.

[5] R. M. Hyatt and T. Mann. A lockless transposition-table implementation for parallel search. 25, March 2002.

[6] C. Joerg and B. Kuszmaul. Massively Parallel Chess. In *Third DIMACS Parallel Implementation Challenge Workshop*, Rutgers University, 1994.

[7] D. E. Knuth and R. W. Moore. An anlaysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975.

[8] V. Manohararajah. Parallel alpha-beta search on shared memory multiprocessors. Master's thesis, University of Toronto, 2001.

[9] T. A. Marsland and M. S. Campbell. Parallel search of strongly ordered game trees. *ACM Computing Surveys*, 14(4):533–551, 1982.

[10] A. Newell, H. Simon, and J. Shaw. Chess playing programs and the problem of complexity. *IBM Journal of Research and Development*, 2:320–335, October 1958.

[11] J. Schaeffer and A. Plaat. New advances in alpha-beta searching. In *ACM Conference on Computer Science*, pages 124–130, 1996.

[12] C. E. Shannon. Automatic chess player. *Scientific American*, 182, 1950.