MISD Timed Chess to simulate Parallel Functioning of the Brain

ECE 5904: Project and Report

April 27, 2023

Advised by Dr. JoAnn Paul

Virginia Polytechnic Institute and State University,
Blacksburg, VA 24061

1. Introduction

We are currently reaching the limits of our parallel architecture. The Brain is the only truly intelligent system known to man. Traditional computer architecture has historically been limited in achieving parallel processing. But while the Brain uses significantly less power, it is also highly parallelizable [27]. Taking inspiration from the Brain, we propose using a novel MISD (multiple instruction single data stream) architecture that emulates independent processing of the same set of inputs to emulate various levels of thinking with selection of response forced by a time constraint. This report will summarize the objectives, solutions, and experimentation for the proposed method.

2. Motivation

S. Nandakumar, S. Kulkarni, A. Babu, and B. Rajendran, in the paper, "Building Brain-Inspired Computing Systems: Examining the Role of Nanoscale Devices" hinted at the Brain being a source of inspiration to solve current computer architectural limitations.

> "Brain-inspired computing is attracting considerable attention because of its potential to solve a wide variety of data-intensive problems that are difficult for even state-of-the-art supercomputers to tackle. The ability of the human Brain to process visual and audio inputs in real time and make complex logical decisions by consuming a mere 20 W makes it the most power-efficient computational engine known to man. While state-of-the-art digital complementary metal– oxide–semiconductor (CMOS) technology permits the realization of individual devices and circuits that mimic the dynamics of neurons and synapses in the Brain, emulating the immense parallelism and event-driven computational architecture in systems with comparable complexity and power budget as the Brain, and in real time, remains a formidable challenge" - [27]

2.1. Limitations of Parallelism

In 1967, Gene Amdahl gave the literal description of what is today known as Amdahl's Law [1]. It is typically stated as follows:

> The speedup of a system is limited by the number of times the sped-up part is used as well as its frequency.

The equation is given by:

$$S = \frac{1}{f_s + \frac{f_p}{N}}$$

Here,
S is scalability
f_p is the fractional part that is parallelizable.
f_s is the fractional part that is serializable.
N is the number of processors used.
$\therefore 1 - f_p = f_s$ is the part of the problem that does not use the speedup.

In fact, when this is plotted on Graph 2.1, we notice something interesting. The Graph is a plot of Amdahl's Law. The X axis is the number of processors, Y axis is the speedup. We plot the Scalability for different values of the Sequential portion, f_s. As we start to decrease the sequential part and increase the number of processors, the scalability tapers off. This means that using conventional forms of multiprocessing, the limits of parallelism will be reached well below levels of complexity found in the Brain. Indeed, as C. Batten has shown in [19] and Figure 2.2, scalability has leveled off over the years even though our transistor count has increased substantially. Note, the X axis shows the years and Y axis, the Scalability. Moreover, the graph also shows Frequency and Wattage have leveled off even though the number of cores in the CPU have increased.

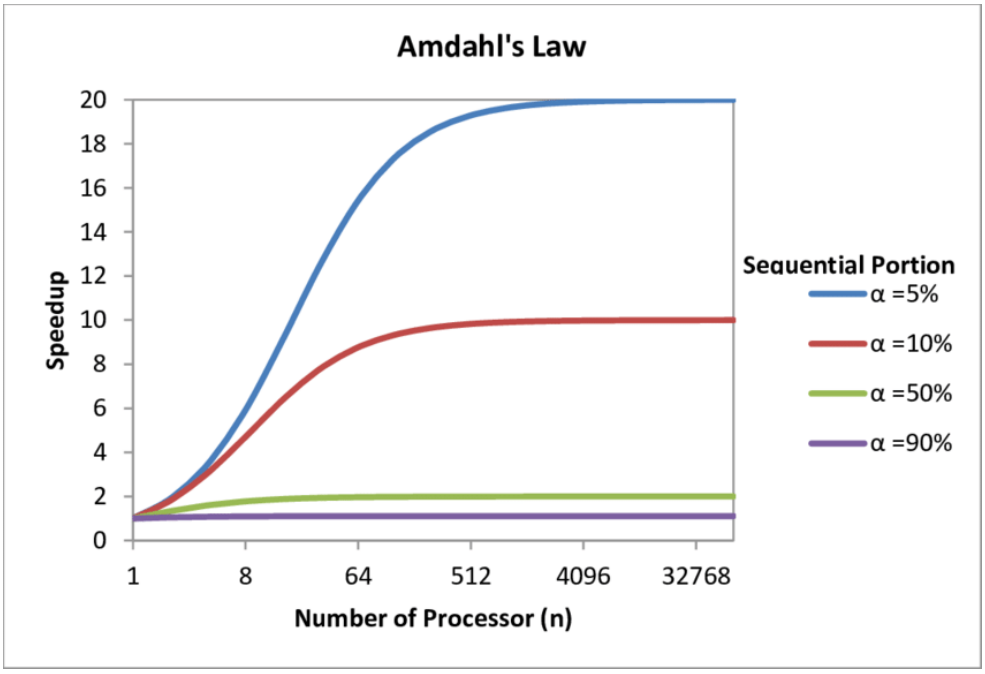

Figure 2.1: Plot of Amdahl's Law. [22]

Many computer scientists are now researching alternative ways of implementing parallelism. Given that the Brain has a very large neuronal interconnection [34], we would expect parallelism to be inherent to such a dense network and we see this parallelism in our daily lives as well.

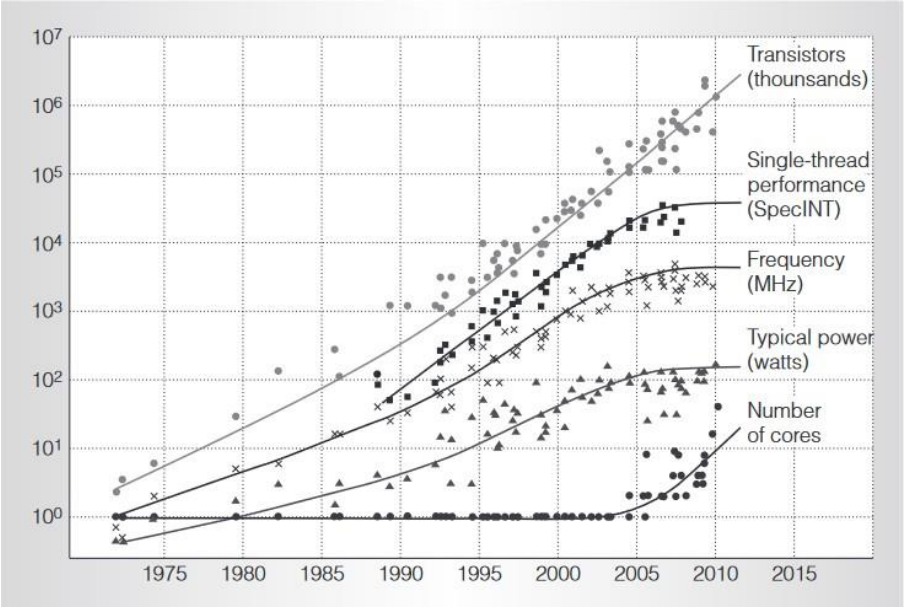

Figure 2.2: Variation of Scalability over time. [19]

3. Insight

Inspired by the way the Brain responds to the same set of stimuli with different responses, depending upon the time given to respond, we propose to investigate how multiple, different outputs might allow for greatly increased levels of parallelism in computing, inspired by the Brain.

3.1. Review of Flynn's Taxonomy

Michael Flynn in 1966 proposed the classification of computer architecture. Called Flynn's Taxonomy [9], in it he gave SISD, SIMD, MISD and MIMD as different classifications.

Following is the table:

	Single Data stream	Multiple Data Streams
Single Instruction stream	SISD	SIMD
Multiple Instruction Streams	MISD	MIMD

Table 3.1: Flynn's Taxonomy

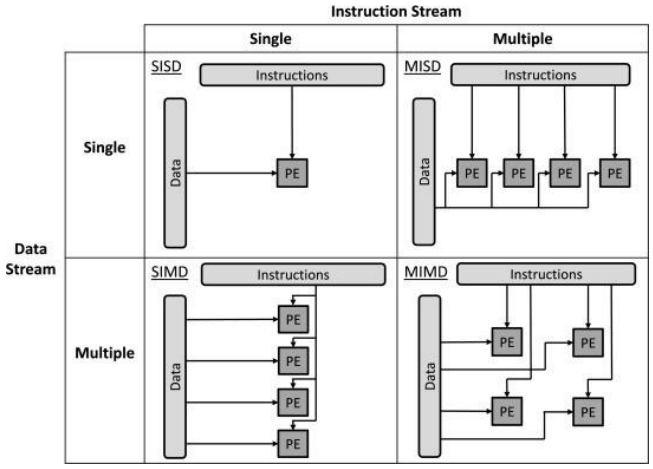

Figure 3.1: Flynn's Taxonomy [29]

1. SISD - Single Instruction single Data stream is a type of computer architecture wherein all data is acted on by 1 instruction stream in a sequential order. This is the most basic kind of computer architecture.
2. SIMD - Single Instruction Multiple Data Stream performs the same instructions on multiple data streams. This type of architecture is typically used for the processing of arrays in signal processing.
3. MISD - Multiple Instruction Single Data Stream can be thought of as an array like processors that work on the same input data stream. But they have different instructions and hence can have different outputs as well.
4. MIMD - Multiple Instruction Multiple Data streams are the most general kind of parallel processing and are widely employed in current computer architecture. It consists of multiple processor elements working on multiple streams of data, each having different instructions. Processors are typically coordinated by an operating system or multithreaded program that is designed to share resources.

Figure 3.1, shows a pictorial representation of SISD, SIMD, MISD and MIMD. As can be

seen, SISD has a single data stream and single instruction for the processing element, while SIMD has multiple data streams, common in array processors that act upon multiple data streams using the same operation. MIMD uses multiple instructions, multiple processing elements and multiple data streams and MISD has a single input data stream and multiple processing elements, acting independently in parallel.

Given that most current parallel architectures use MIMD [15], and MIMD does not scale to levels of complexity found in the Brain, novel parallel architectures should be considered. We are inspired by how MISD may implement the parallelism and conflict found in the subconscious mind.

3.2. Parallel thinking in the Brain

The Brain never processes the same input in the same way [35]. In fact, these different inputs evolve into different states that the Brain can use, and it is attributed to the highly dense and interconnected neural network of the Brain. Even a single Neuron as seen in Figure 3.2 can have hundreds of interconnections. The dendrites connect to thousands of different cells as well as Axon terminals.

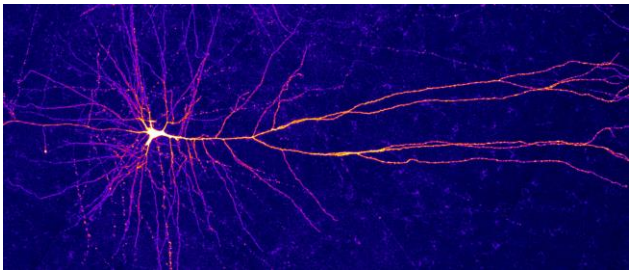

Figure 3.2: Image of a single neuron and all its interconnections [4]

This inspires us to investigate Multiple Instructions, Single Data stream type architecture. But, if different networks of neurons process the same input in diverse ways, the output will be different as well. This could lead to conflict. Consider this in terms of playing chess. Figure 3.3 provides a conceptual representation of this. It is well known that Novice players can think of only 1-2 moves ahead while Grandmasters can think of many more moves. This has the potential for Grandmasters to hold very advanced chess board states in their Brain as seen in Figure 3.3. Each of the different states shown are evolution of the same input states but they differ temporally. Note that all the evolved states might not be used, and their usage depends on the time at which those states should be resolved into a single state that the Brain can output.

Consider how this relates to the process of thinking. If the Brain thinks deeply for longer periods

of time, then the result will almost always be different if in contrast it finished thinking too quickly. Thus, the Brain must evolve different states that can resolve into something. We hypothesize that the Brain resolves this based on the time it has to think.

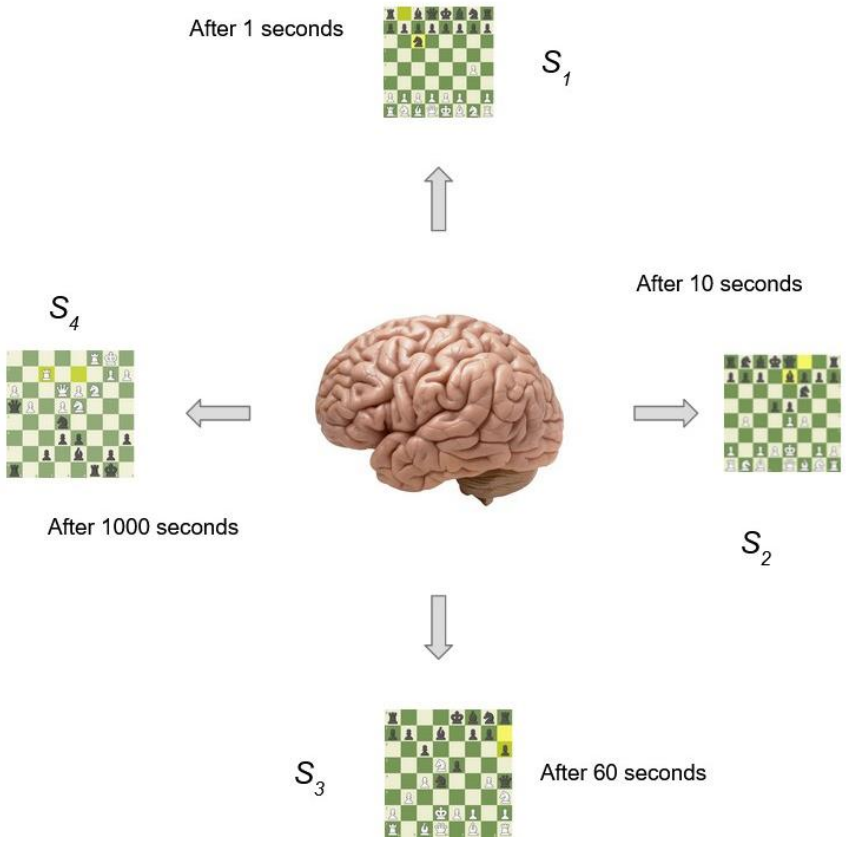

Figure 3.3: Conceptual process of parallel thinking in the Brain

Proposed Experimentation

Inspired by the chess example, we propose to implement MISD on a conventional Von Neuman Architecture.

4.1. Why games?

Games, such as 2 player games, are tractable and have many ways of playing. Different opening moves may or may not lead to winning. In fact, no typical move is the best move in a game since such moves can have counter moves that can change the course of a game. Humans, when playing games, tend to have various levels of play. Quick thinking about different moves

almost always leads to losing but when thought deeply, for extended periods of time, it is possible to win games. This means, if we simulate games and their various levels of play, we might be able to simulate the human Brain.

Since the Brain thinks in parallel, it must also evolve different solutions to the game at the same time and pick the one based on time. Thus, given n states S_1, S_2, S_3....S_n, each state must be evolved independent to the other to prevent interference. The Brain can easily do this since it already has a dense network with high degree of interconnectivity [34].

4.2. Zero Sum Games and their Algorithms

A zero sum game is a game in which the player and his opponent have equal win and loss i.e., a player wins in equal amount to the opponent's loss [23]. Many board games are zero sum game [36]. For zero sum games, their algorithms tend to be simpler, and the results are easier to analyze. Therefore, such games are considered for experimentation. Among the many board games, we consider chess due to its popularity, tractability, and ample literature on grandmaster bots.

In order to implement chess on an MISD architecture, each instruction must represent different algorithms that take as input a particular game move and evolve different states. These states must also be resolved since they are in conflict with each other.

4.3. A Review of Chess

Chess is a 2-player game and uses a 64 square 8x8 board alternating between white and black [20]. Each player controls 16 pieces:

- 1 King
- 1 Queen
- 2 Rooks
- 2 Bishops
- 2 Knights
- 8 Pawns

White moves are followed by black and vice versa. Several ways exist to end a game in a draw [20]. Every chess piece has legal moves and illegal moves. Due to this, chess has an exceptionally large branching factor. The average branching factor is 35 [21]. This means that for a given chess piece, on average there are at most 35 ways to select a move. After that, there are 1225, 42875, 1500625 and so on. As seen in 4.1, $O(35^x)$ rises faster than $O(e^x)$ and hence after a depth of

just 4, there are more than a million moves. This means, 2 moves can have 2 independent sub trees and each subtree can be explored by an algorithm. This large branching factor is perfect for board state evolution since different Algorithms will have lower probability of exploring the same subtree.

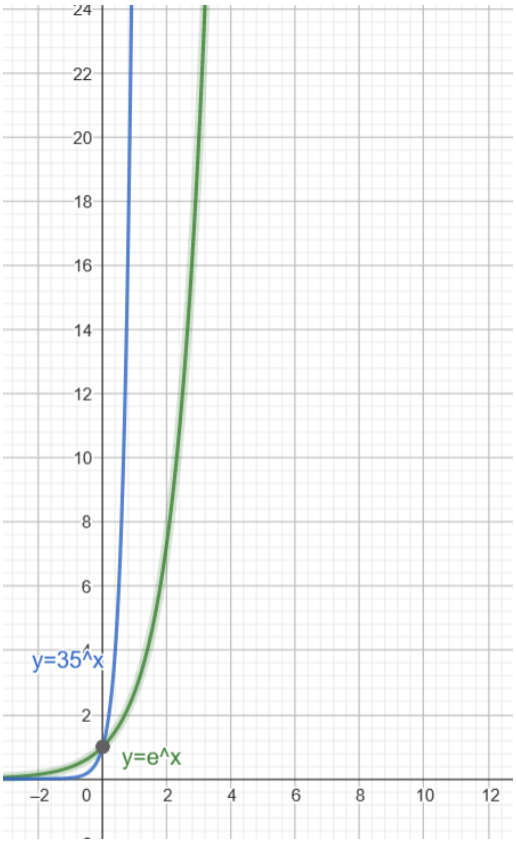

Figure 4.1: Graph of Time Complexity of $O(35^x)$ vs $O(e^x)$.

4.4. Multiprocessing and Multithreading

In order to explore these subtrees in parallel, multiprocessing can be used. This requires a CPU with at least 2 or more processors. Algorithms take a specific move as an input and are processed independently on each core of a CPU.

It is important to provide reasons why multithreading was not chosen. Multithreading is the execution of multiple threads by a hardware executor [3]. Threads are either user defined, which are managed by the Operating System or implicit, which are statically generated by a compiler or dynamically generated by hardware. Before superscalar architecture, each core of a CPU could only process 1 thread at a time and context switched in order to appear as if threads were being executed in parallel [37]. This 'false' parallelism undercuts our efforts to achieve true parallelism in order to simulate the Brain. Hence, we do not use multithreading.

5. Background

5.1. Chess Algorithms

Chess algorithms can be divided into 2 main types:

1. Search and Evaluation based
2. Static or dictionary based

Search and Evaluation based make use of search algorithms and evaluation algorithms to determine the best move. In static or dictionary based, a dictionary of the best moves and resulting moves are kept and looked up during play. We will focus on search based.

5.1.1. Search Algorithms in Artificial Intelligence

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  inputs: percept, a percept
  static: seq, an action sequence, initially empty
          state, some description of the current world state
          goal, a goal, initially null
          problem, a problem formulation

  state ← UPDATE-STATE(state, percept)
  if seq is empty then do
      goal ← FORMULATE-GOAL(state)
      problem ← FORMULATE-PROBLEM(state, goal)
      seq ← SEARCH(problem)
  action ← FIRST(seq)
  seq ← REST(seq)
  return action
```

Figure 5.1: A simple problem solving agent that uses a Search method to scan an optimal path [28]

Search algorithms in Artificial Intelligence generate a path from a node in a 'tree' that represents the optimal path. The path generated from input to output is called the Action Sequence and the execution cycle is called the Execution Sequence [28]. The action sequence guides the execution sequence. As seen in Figure 5.1, the agent first formulates a goal, formulates the problem given the goal and state and finally does a search. The action sequence generated is then executed. This Agent runs for only 1 state. In chess, this search method will go down the game tree, and try to determine if a chosen move leads to winning or losing a game.

5.1.2. Minimax Algorithm

The Minimax algorithm attempts to minimize the maximum loss due to a certain move in a game [17]. Minimax is a well-known algorithm and has been used for years to solve 2-person computer games, perfect information games to choose the best move. Shannon and Turing first proposed the basic concepts of Minimax [17]. Minimax works by recursively travelling the nodes of a tree, till it reaches the leaf and then backtracks, at each step calculating alternatively the minimum of the maximum and maximum of the minimum. Figure 5.2 shows such an example. The terminal nodes represent the values. At each level, we alternatively take the Maximum of the children and Minimum of the children. Consider child nodes -2, -1 and +4, -5. We take the Maximum of them, and we get -1 and +4. Now we switch to minimum and get -1. We continue this process till we reach the root node of that subtree. Figure 5.3 shows the Minimax Algorithm in Pseudo code. We alternatively maximize and minimize the score for each child node of the current node and once we reach depth 0 or a terminal node, we return the heuristic value. Typically, Minimax uses depth first search in combination with backtracking in order to reduce storage requirements. Better performance and simplicity are obtained with the use of Alpha Beta Pruning [18].

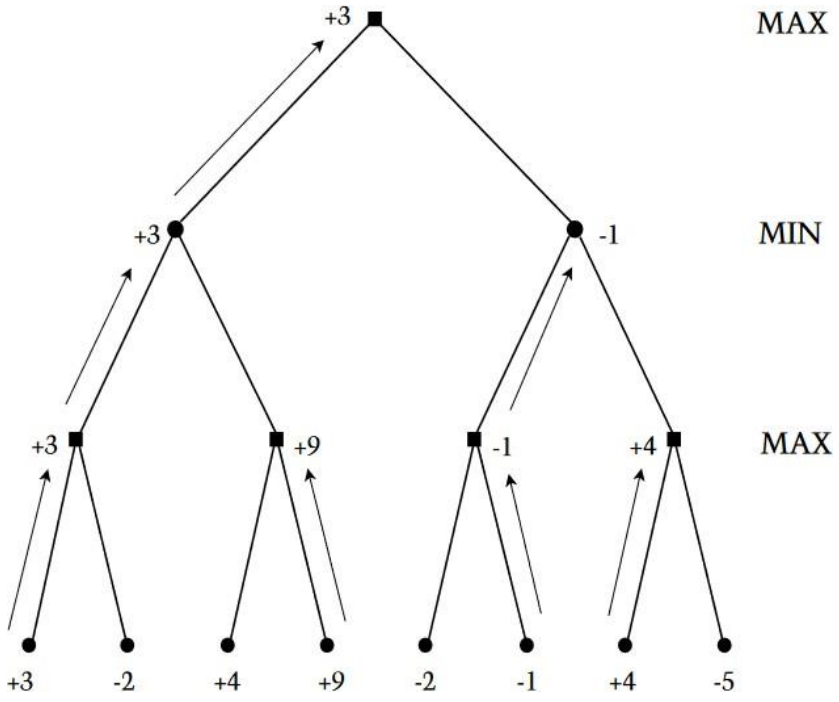

Figure 5.2: An example of a Minimax tree. Note that the value obtained depends on whether we wish to minimize the loss or maximize the score [17].

Since Minimax does a depth first search till the depth d, the entire search can take a very long time as the game tree of chess is very large. Claude Shannon approximated that there were 1040 board position [31] and hence a naive depth first search on all node is infeasible.

Alpha Beta Pruning

Although Minimax guarantees to find the best move, the time taken for the entire tree is not feasible. Alpha Beta Pruning is an extension that decreases the search time without sacrificing accuracy. The value of the current node and its parent are the pruning parameters. The MAX node is the alpha value while the min node is the beta value. The pruning rules states that whenever the alpha value becomes more than the beta value, the state can be pruned [25]. As seen in Figure 5.4, the value of N_7 is unknown. N_3 is a MIN node and N_1 is a MAX node and thus their values must be at most 1 and at least 2 respectively. Hence the value of N_3 will not be backpropagated to N_1 and any additional calculations are useless. Figure 5.5 shows the Pseudo code of this method. The pseudo code is similar to Figure 5.3 but we calculate the alpha values and beta values at each iteration and if the beta value is less than equals to the alpha value then we break the loop.

```
function minimax(node,depth, maximizingPlayer)
if depth = 0 or node is a leaf node then
    return heuristic value of node
end
if maximizingPlayer then
    value = -∞
    while every child of node do
        value = max(value, minimax(child, depth-1,
          FALSE)
    end
    return value
end
else
    value = +∞
    while every child of node do
        value = min(value, minimax(child, depth-1,
          TRUE)
    end
    return value
end
```

Figure 5.3: The Minimax Algorithm [32]

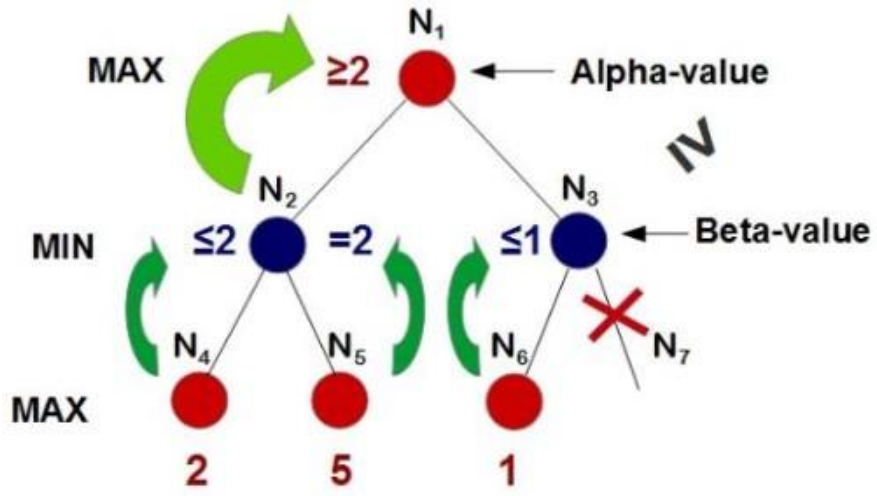

Figure 5.4: Alpha Beta Pruning [25]

Time Complexity

Given that naive Minimax is a depth first search algorithm which scans each node, the time complexity of this algorithm is $O(b^d)$ where b is the branching factor and d is the depth [24]. With Alpha Beta Pruning applied, each node would examine $2b - 1$ grandchildren and therefore, the best-case complexity is $O(b^{d/2})$ [24].

5.1.2. Monte Carlo Tree Search

Monte Carlo Tree Search, first coined by Rémi Coulom [6], is a randomized approach to solving zero sum games. First used in Go playing programs, it quickly spread and is now used in Google DeepMind's AlphaGo program in combination with 2 deep neural networks [11]. It all culminated in it winning against reining Go champion Lee Sedol in March of 2016 [11].

The basic Monte Carlo Tree Search involves a tree being built in an incremental and asymmetric manner. For each iteration, the most urgent node is found by the tree policy. The policy attempts to balance considerations of exploration and exploitation. A simulation is then run from the selected node and a search tree is formed. The child node is added to the current node corresponding to the action taken and then the tree statistics are updated. Moves are made according to a policy that is typically about uniform randomness [2].

```
function ALPHA-BETA(node, depth, α, β,
 maximizingPlayer)
if depth = 0 or node is a leaf node then
 | return heuristic value of node
end
if maximizingPlayer then
 | value = -∞
 | while every child of node do
 | | value = max(value, ALPHA-BETA(child,
 | | depth-1,α, β, FALSE)
 | | α = max(α, value)
 | | if β ≤ α then
 | | | break
 | | end
 | end
 | return value
end
else
 | value = +∞
 | while every child of node do
 | | value = min(value, ALPHA-BETA(child,
 | | depth-1,α, β, TRUE)
 | | β = min(β, value)
 | | if β ≤ α then
 | | | break
 | | end
 | end
 | return value
end
```

Figure 5.5: Alpha Beta Algorithm Pseudo Code [32]

The following are the 4 main operations that take place [11]:

- Selection - This corresponds to choosing a move at a node or action and the choice is based on the Upper Confidence Bound (UCB) which is a function of the current estimate plus another factor.
- Expansion - This corresponds to an opponent's move in the tree and is modeled by a probability distribution that is a function of the state after the action has been taken.
- Simulation/Evaluation - This returns the estimated value at a given node which could be the real value corresponding to the end of the game.
- Backup - This corresponds to the backward dynamic programming algorithm employed in decision trees.

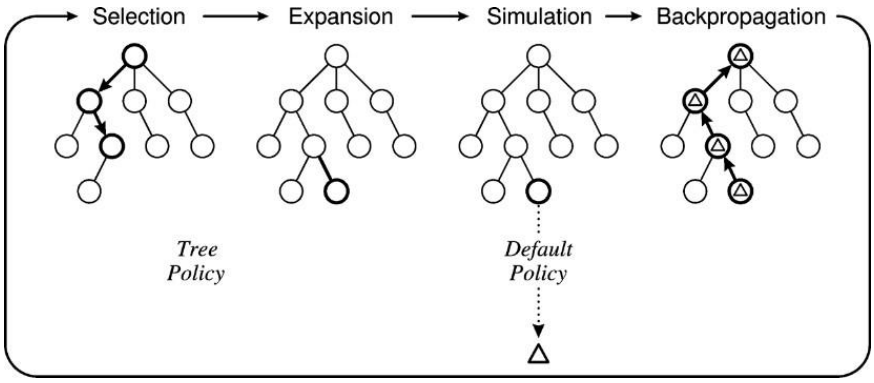

Figure 5.6: Monte Carlo Tree Search Iteration [2]

Algorithm 1 Monte Carlo Tree Search Algorithm

```
1:  function MCTS(root)
2:      root: Root of Tree
3:      EXPAND(root)
4:      while RESOURCESREMAIN( ) do                    ▷ Time or Iterations
5:          ℓ ← TRAVERSE(root)          ▷ Identify a leaf node via best UCB score
6:          if ROLLEDOUT(ℓ) then                       ▷ Previously rolled out node.
7:              EXPAND(ℓ)
8:              ℓ ← RANDOMCHILD(ℓ)
9:          SF_ℓ ← ROLLOUT(ℓ)              ▷ Monte Carlo game simulation from ℓ
10:         BACKPROP(ℓ, SF_ℓ)          ▷ Back-propagate result from ℓ up to root
11:     return BESTCHILD(root)
```

Figure 5.7: Monte Carlo Tree Search pseudo code [13]

Figure 5.6 shows visualization of an iteration of the Algorithm. The selection process takes place by calculating the UCB value. We then expand based on a probability distribution. Simulation uses the default tree policy to return the estimated value. In backpropagation, we send the value back to the ancestors by using backward dynamic programming methods. The iteration covers the 4 methods detailed above and Figure 5.7 shows the pseudocode. For a given root node, we first get all the children of the input nodes and then the selection of the child nodes based on the UCB score. From there we expand, select a random child, and then simulate/roll out with the selected child. Finally, we backpropagate.

Time complexity

The time complexity of the algorithm is $O(mKI/C)$ where m, K, I, C are the number of random

children to consider for each search, number of parallel searches, number of Iterations and the number of cores used, respectively [16]. In our implementation, $K = 1$, $I = 5$ and C = 1 since there is only 1 core and hence 1 parallel search. Thus, our resulting complexity is $O(m)$.

6. Implementation

The implementation consists of 2 main parts - Implement chess algorithms in parallel and find a way to resolve their states. The program is implemented in Python.

6.1. Simulate MISD on non MISD architecture

MISD which stands for Multiple Instructions Single Data Stream is an uncommon computer architecture [29] and has been used in space shuttles by NASA [33] in order to verify calculations by running the same input on 2 different, independent computers and verifying the results. IBM's Power 4 CPU was the first CPU to introduce multi core processing to the commercial market [14] back in 2001. Multiprocessing can thus enable us to simulate MISD to a certain degree. This is achieved by running each algorithm separately on a single core. Finally, the results from each core are returned.

To implement multiprocessing in python, the multiprocessing module is used [10] as seen in Figure 6.2 It has native APIs that can be used for processor pools, closing pools and terminating pools. The apply function takes as input a function reference and the argument parameter of the apply function is where the board state and cut off time are passed. Figure 6.1 is a conceptual representation of the MISD implementation and parallels Figure 3.3. Because Grandmasters think many moves in advance, each processor if left to process, will continue to evolve advanced board states in memory. This effectively simulates deep thinking. In order avoid accidentally pooling more CPU's than the number that exists on the system, we take the minimum of 3 and the CPU count as seen in Figure 6.2.

```
pool = Pool(min(3, cpu_count())
r1 = pool.apply(bot1.move, args=(board, cut_of_time, ))
r2 = pool.apply(bot2.move, args=(board, cut_of_time, ))
r3 = pool.apply(mcts_main, args=(board, cut_of_time, ))
```

Figure 6.1: Python multiprocessor module usage

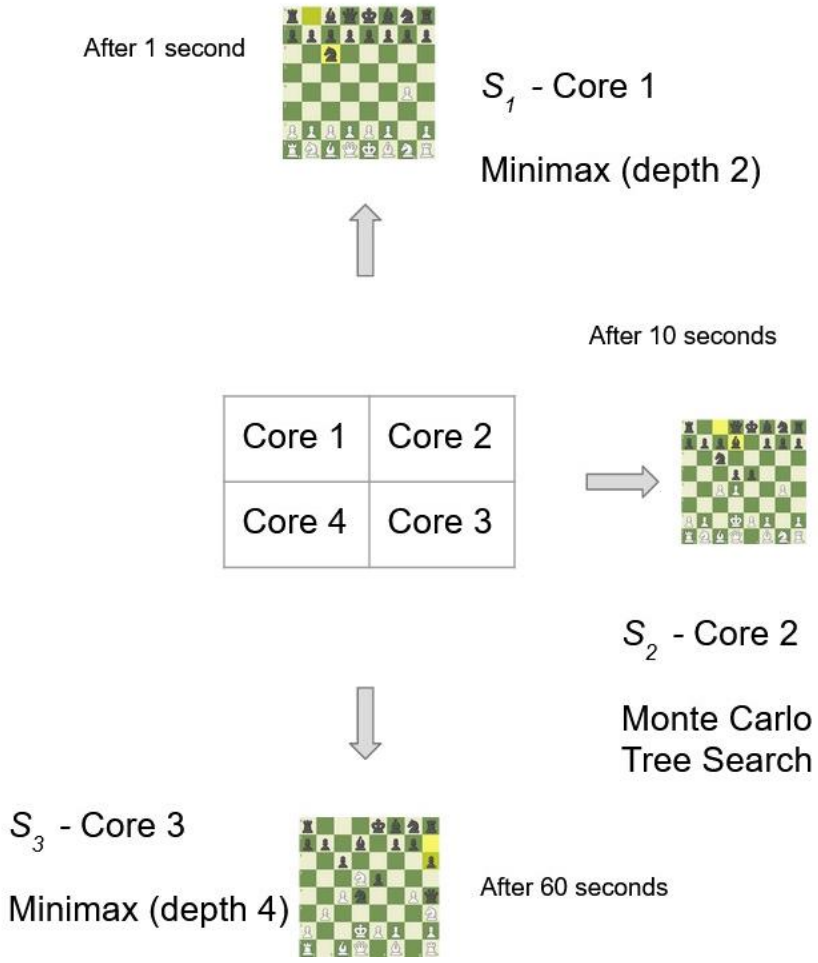

Figure 6.2: Conceptual Representation of MISD and independent game states evolving in each core. Note the parallels to the Brain's thinking in Figure 3.3

6.2. Chess Algorithms

Minimax and Monte Carlo Tree Search is implemented in Python. For reusability, Object Oriented Programming was used which means both the algorithms were represented by classes. Figures 6.3 and 6.4, shows the code snippets for the same. For Monte Carlo Tree Search, we define 4 functions, ucb, expand, rollout and rollback. ucb calculates the UCB value of a given node. Expand will expand the tree from the given input node. Roll out will simulate the selection using a the randomly selected node as input and roll back backpropagates to the ancestors. Minimax had 2 functions. The main function starts the timer and the _minimax function is the main recursive function that runs the program.

```
68  class MonteCarlo:
69      def __init__(self):
70          return
71
72      def ucb(self, node: Node) -> float:
73          ans = node.w_Score + 2 * (sqrt(log(node.N + e + (10 ** -6)) / (node.n + (10 ** -10))))
74          return ans
75
76
77
78
79      def expand(self, node: Node, white: int) -> Node:  ...
99              return self.expand(selected_child, 1)
100
101
102     def rollout(self, node: Node):  ...
123         return self.rollout(rnd_state)
124
125     def rollback(self, node: Node, reward):  ...
131         return node
132
133     def main(self, node: Node, cut_of_time: int, iterations: int):  ...
169         return (original_board.parse_san(selected_move).uci(), time() - t1)
170
```

Figure 6.3: The Monte Carlo Tree Search Class in Python

```
class MiniMaxPlayer(Player):
    def __init__(self, player, depth, verbose=False):  ...
        self.verbose = verbose

    def _minimax(self, board: Board, player: bool, depth: int, t1: float, cut_of_time: int, alpha: float = -inf, beta: float = inf):
            return [minScore, bestMove]

    def move(self, board: Board, cut_of_time: int):  ...
        return (best_move[1].uci(), time() - t1)
```

Figure 6.4: The Minimax Class in Python

6.3. Chess Board, legalities, and representation

Since chess has an extensive set of rules such as a set of legal moves for each piece, tie conditions etc., all these need to be implemented in order for a game to start. Fortunately, Python has a chess library called Chess [8] which takes care of all the rules, move conversion and verification. In order to communicate moves more effectively to the player, the UCI protocol is used [12]. UCI, which stands for Universal Chess Interface, is used to communicate between chess

engines. In its most simplistic form, it communicates the current chess position and the next chess position in co-ordinate form. For example, e4d5 is a UCI form which says that move the current piece located at e4 to d5 on the board.

To communicate board state, the FEN notation is used. FEN is a notation invented by newspaper journalist David Forsyth in the 19th century. A single FEN record uses one text line of variable length composed of six data fields. Six fields comprise a FEN description. The en passant target square, the active color, the castling availability, the piece placement, the halfmove clock, and the fullmove number are all specified by FEN [7]. A FEN position description's length varies a little depending on the position. The description may not properly fit on some devices if it is 80 characters or longer in some circumstances. These jobs are not that common [7]. The only characters allowed in each field are non-blank printable ASCII characters. A single ASCII space character is used to demarcate adjacent fields [7]. The following FEN represents the board in Figure 6.5.

rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR b KQkq - 0 1

Figure 6.5: Standard Chess Board in start state [5]

6.4. Randomized Cut Off Time

In order to implement a randomized cut off time, we first generate a random integer in a range (Figure 6.6) starting from 0 since when rounded up, Minimax with depth 2 finishes execution in 0.01 seconds on average. The generated integer is passed as an argument to all 3 algorithm calls. At each iteration/recursion, the current wall clock time is measured and subtracted from the start time to determine the time taken for execution. If this time is greater than the cut off time, then the recursion stack is exited as shown in Figure 6.7.

```
cut_of_time = random.randint(0, 60) if random_time else -1
```

Figure 6.6: Cut off time is generated by the randint function which generates a random integer in the given range

```
diff = time() - t1
if(diff>cut_of_time and cut_of_time!=-1):
    return []
```

Figure 6.7: If the current wall clock time difference exceeds the cut off time, then the recursion stack is exited

During a move, when all 3 algorithms give their output, to resolve their output, this cut off time is used. In the Brain, as these game states evolve independently of each other, time is used to resolve their differences and then output the best answer. The implementation of the cut off time provides a resolution similar to that of the Brain.

7. Experimentation

To show that this new architecture is a better model of the Brain's neural network, experiments must be carried out.

7.1. Time complexity and measurements

As stated in 4.3, since the complexity of each depth rises faster than exponential time, we expect that the average run time for each succeeding depth will take b times longer where b is the average branching factor. Since the average branching factor for every depth is not the same, we expect variation such that consecutive depths are at most 35 times slower.

We hypothesize that Monte Carlo will be at most 35 times slower than Minimax with Depth 2 and Minimax with Depth 4 will be at most 35 times slower than Monte Carlo since the complexity of Minimax and Monte Carlo are $O(b^{d/2})$ and $O(m)$ as stated in 5.1.2 and 5.1.3.

Algorithm	Average Runtime in Seconds
Minimax with Depth 2	0.140407741
Monte Carlo Tree Search	4.573309207
Minimax with Depth 4	82.41738934

Table 7.1: Average Runtime over 40 games for all 3 algorithms

Table 7.1 is experimental data for 40 random games. The runtime for every turn is measured and averaged over the number of bot turns. This average is then averaged over the number of games which in this case is 40. As seen in the table, Monte Carlo is on average approximately 33 times slower than Minimax with depth 2. And Minimax with depth 4 is 18 times slower than Monte Carlo thus validating our hypothesis.

7.2. Addressing process interference

As seen in Figure 7.1, the graph represents how the average runtime varies for 40 random games. The X axis shows the number of games, and the Y axis shows the Time in seconds. The blue, orange and green line shows the average runtime variation for Minimax with depth 2, Monte Carlo Tree Search and Minimax with depth 4. If seen closely, we realize that the average runtime graphs of each algorithm never intersect with each other and the slowest is the many orders of magnitude slower than the other 2. This means, each algorithm's process has no interference on other algorithms. This shows, experimentally, that the algorithms run independently and thus evolve board states in parallel and hence, represents MISD well enough.

7.3. Inherent Conflict Resolution – Board state convergence

Given that we know the Brain resolves conflicting game states depending on the time 3.2, it now becomes contingent to show that such conflict resolution naturally arises because of our implementation.

To show this, we can hypothesize that given a certain board, there exists only 1 best move. Consider for instance Figure 7.2, there clearly is one best move, move Black's King at e8 to d8, thus capturing White's Queen. We thus need to show that our algorithms, although evolved independently on each core of the CPU, will converge to this specific move e8 to d8.

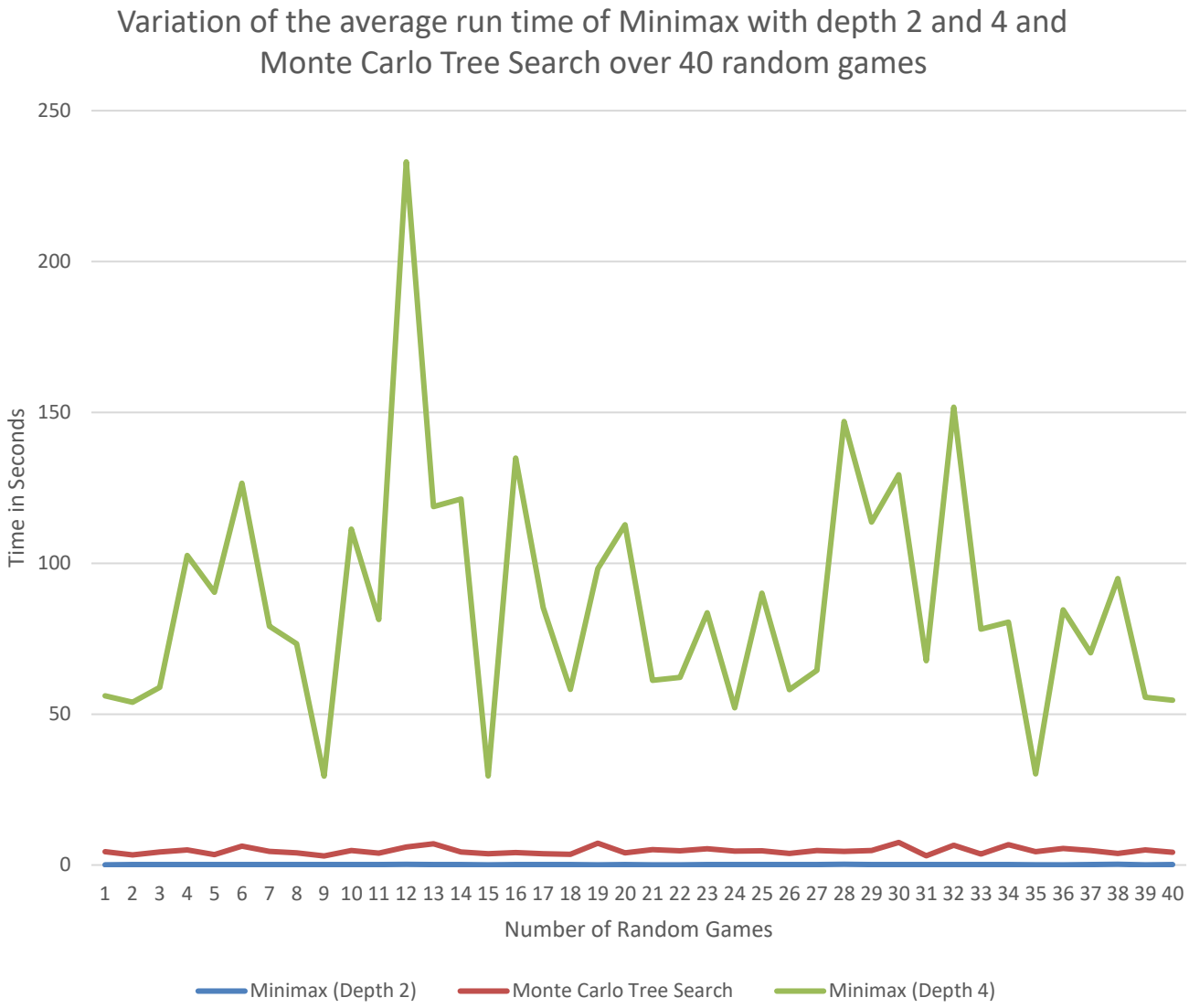

Figure 7.1: Variation of the average runtime across 40 random games

Given the board state in Figure 7.2, the output is seen in Figure 7.3. The table gives the algorithms run, the depth if applicable, the move generated, and the time taken by each. All 3 runs show the same output. Thus, the algorithms converged independently of each other which is an accurate representation of the Brain as well.

Figure 7.2: A game state with only 1 best move

ID	Bot Name	Move Generated	Depth	Time	Select
0	Minimax	e8d8	2	0.01	Select
1	Monte Carlo Tree Search	e8d8		5.84	Select
2	Minimax	e8d8	4	12.42	Select

Figure 7.3: Output of the Algorithms with board state in Figure 7.2

7.4. Temporal Conflict Resolution - Speed chess players have only marginal win percentage against an un-timed weak player

The core concept of speed chess lies in its ability to make quick decisions against a set timer that starts at every round. This means, board states can no longer evolve for longer periods of time in the Brain's neural network. When the timer is up, in a round of speed chess, the Brain is forced to give an answer.

Win Percentage	Tie Percentage	Loss Percentage
35	65	0

Table 7.2: Win, Tie and Loss Percentage of the time constrained algorithm against a baseline for 20 random games

In order to simulate this, the cut off time is detailed in Section 6.4 of the Implementation Chapter. We hypothesize that this cut off time will make the player win less often. To simulate the other player, we set a baseline metric. The baseline in this case is Minimax with depth 2. This independent version of Minimax will be the other player and since the depth is only 2, it is on average less accurate when compared to depth 4.

After experimenting with 20 random games, Table 7.2 shows the Win, Tie and Loss percentage. We notice that while our time constrained algorithms have never lost, the tie percentage is nonetheless higher than the win percent. This indicates that our time constrained algorithms are only marginally better than the baseline.

8. Conclusion

The Brain has a massively parallel network of highly interconnected neurons. Motivated by the way the Brain produces different responses, if given more time to think, we showed that an MISD architecture with variable time constraints may mimic the way the Brain processes in parallel. We developed experiments and showed that a timer using a randomized cut off time is only marginally better against a novice player. In the future, we propose to investigate our implementation on a true MISD architecture and to utilize different depths to gain even more levels of thinking.

Bibliography

[1] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), page 483–485, New York, NY, USA, 1967. Association for Computing Machinery. doi:10.1145/1465482.1465560.

[2] Cameron B. Browne, Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012. doi:10.1109/TCIAIG.2012.2186810.

[3] Horia V. Caprita and Mircea Popa. Design methods of multithreaded architectures for multicore microcontrollers. In *2011 6th IEEE International Symposium on Applied Computational Intelligence and Informatics (SACI)*, pages 427–432, 2011. doi:10.1109/SACI.2011.5873041.

[4] Jordana Cepelewicz. Hidden computational power found in the arms of neurons. 2020. URL: https://www.quantamagazine.org/neural-dendrites-reveal-their-computational-power-20200114/.

[5] CHESS.com. How to set up a chessboard, 2023. URL: https://www.chess.com/article/view/how-to-set-up-a-chessboard.

[6] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *Proceedings of the 5th International Conference on Computers and Games*, CG'06, page 72–83, Berlin, Heidelberg, 2006. Springer-Verlag.

[7] Steven Edwards. Standard: Portable game notation specification and implementation guide, 1994. URL: https://www.thechessdrum.net/PGN_Reference.txt.

[8] Niklas Fiekas. Chess, 2023. URL: https://pypi.org/project/chess/.

[9] M.J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, 1966. doi:10.1109/PROC.1966.5273.

[10] Python Foundation. multiprocessing — process-based parallelism, 2011. URL: https://docs.python.org/3.11/library/multiprocessing.html.

[11] Michael C. Fu. A tutorial introduction to monte carlo tree search. In *2020 Winter Simulation Conference (WSC)*, pages 1178–1193, 2020. doi:10.1109/WSC48552.2020.9384090.

[12] Rudolf Huber and Stefan Meyer-Kahlen. Uci protocol, 2004. URL: https://www. wbec-ridderkerk.nl/html/UCIProtocol.html.

[13] Joshua Hutson, George Lamb, and Chris Alvin. Monte carlo tree search applied to the game abalone. *J. Comput. Sci. Coll.*, 37(5):21–30, oct 2021.

[14] IBM. Power 4 - the first multi-core, 1ghz processor, 2011. URL: https://www.ibm.com/ibm/history/ibm100/us/en/icons/power4/.

[15] Intel. Mimd, 2013. URL: https://web.archive.org/web/20131016215430/http://software.intel.com/en-us/articles/mimd.

[16] Yifa Jin and Shaun Benjamin. Cme 323, report, 2015. URL: https://stanford.edu/~rezab/classes/cme323/S15/projects/montecarlo_search_tree_report.pdf.

[17] Hermann Kaindl. Minimaxing: Theory and practice. *AI Magazine*, 9(3):69, Sep. 1988. URL: https://ojs.aaai.org/aimagazine/index.php/aimagazine/article/ view/944, doi:10.1609/aimag.v9i3.944.

[18] Donald E. Knuth and Ronald W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975. URL: https://www.sciencedirect.com/science/article/pii/0004370275900193, doi:https://doi.org/10.1016/0004-3702(75)90019-3.

[19] Christos Kozyrakis, Aman Kansal, Sriram Sankar, and Kushagra Vaid. Server engi-

neering insights for large-scale online services. *IEEE Micro*, 30(4):8–19, 2010. doi:10.1109/MM.2010.73.

[20] Emanuel Lasker and M. I. Dvoretskiĭ. *Lasker's manual of chess*. Russell Enterprises, 2010. URL: https://worldcat.org/title/778368272.

[21] ALAN LEVINOVITZ. The mystery of go, the ancient game that computers still can't win, 2014. URL: https://www.wired.com/2014/05/the-world-of-computer-go/.

[22] Khaled Mashfiq. *NONLINEAR EARTHQUAKE ENGINEERING SIMULATION US- ING PARALLEL COMPUTING SYSTEM*. PhD thesis, 12 2012. doi:10.13140/RG.2.2.21215.87208.

[23] MasterClass. Zero-sum game meaning: Examples of zero-sum games. 2022. URL: https://www.masterclass.com/articles/zero-sum-game-meaning.

[24] Vasilis Megalooikonomou. Cis603 s03, 2003. URL: https://cis.temple.edu/~vasilis/.

[25] Sajjad Mozaffari, Bardia Azizian, and Mohammad Hadi Shadmehr. Highly efficient alpha-beta pruning minimax based loop trax solver on fpga. In *2015 18th CSI International Symposium on Computer Architecture and Digital Systems (CADS)*, pages 1–4, 2015. doi:10.1109/CADS.2015.7377789.

[26] H.J.R. Murray. *History of Chess*. DigiCat, 2022. URL: https://books.google.com/books?id=B_hyEAAAQBAJ.

[27] S.R. Nandakumar, Shruti R. Kulkarni, Anakha V. Babu, and Bipin Rajendran. Building Brain-inspired computing systems: Examining the role of nanoscale devices. *IEEE Nanotechnology Magazine*, 12(3):19–35, 2018. doi:10.1109/MNANO.2018.2845078.

[28] Stuart Jonathan Russell and Peter Norvig. *Artificial intelligence : a modern approach ; [the intelligent agent book]*. Prentice Hall, 2003. URL: https://virginiatech.on.worldcat.org/oclc/249298222.

[29] Bertil Schmidt, Jorge González-Domínguez, Christian Hundt, and Moritz Schlarb. Chapter 3 - modern architectures. In Bertil Schmidt, Jorge González- Domínguez, Christian Hundt, and Moritz Schlarb, editors, *Parallel Pro- gramming*, pages 47–75. Morgan Kaufmann, 2018. URL: https://www.sciencedirect.com/science/article/pii/B9780128498903000034, doi:https://doi.org/10.1016/B978-0-12-849890-3.00003-4.

[30] ScienceDaily. Human Brain region functions like digital computer, 2006.

[31] Claude E. Shannon. *Programming a Computer for Playing Chess*, pages 2–13. Springer New York, New York, NY, 1988. URL: https://doi.org/10.1007/ 978-1-4757-1968-

0_1.

[32] Shubhendra Pal Singhal and M. Sridevi. Comparative study of performance of parallel alpha beta pruning for different architectures. In *2019 IEEE 9th International Confer- ence on Advanced Computing (IACC)*, pages 115–119, 2019. doi:10.1109/IACC48062.2019.8971591.

[33] Alfred Spector and David Gifford. The space shuttle primary computer system. *Commun. ACM*, 27(9):872–900, sep 1984. doi:10.1145/358234.358246.

[34] Olaf Sporns, Giulio Tononi, and Rolf Kötter. The human connectome: A structural description of the human Brain. *PLOS Computational Biology*, 1(4):null, 09 2005. doi:10.1371/journal.pcbi.0010042.

[35] Tilman Stephani, Gunnar Waterstraat, Stefan Haufe, Gabriel Curio, Arno Villringer, and Vadim V. Nikulin. Temporal signatures of criticality in human cortical ex- citability as probed by early somatosensory responses. *Journal of Neuroscience*, 40(34):6572–6583, 2020. URL: https://www.jneurosci.org/content/40/34/6572, arXiv:https://www.jneurosci.org/content/40/34/6572.full.pdf, doi:10.1523/JNEUROSCI.0241-20.2020.

[36] John von Neumann, Oskar Morgenstern, and Ariel Rubinstein. *Theory of Games and Economic Behavior (60th Anniversary Commemorative Edition)*. Princeton University Press, 1944. URL: http://www.jstor.org/stable/j.ctt1r2gkx.

[37] Zhifeng Zhang, Dongqiang Pan, and Guochun Wan. Modeling and simulation of chip multithreading processor architecture. In *2010 Second International Conference on Computer Modeling and Simulation*, volume 2, pages 85–88, 2010. doi:10.1109/ICCMS.2010.207.