A Workflow Based Implementation of the Minimax Algorithm for Chess Using Google App Engine

Final Report

Team #2
Geetika Garg
David Smith
Zayd Hammoudeh

Table of Contents

List of Figures

List of Tables

A Workflow Based Implementation of the Minimax Algorithm for Chess Using Google App Engine

1. Team #2 Members

a. Geetika Garg #128
b. David Smith #228
c. Zayd Hammoudeh #877

2. Project Topic

Chess is a two player, zero sum[1] game that is both fully observable[2] and deterministic[3]. For adversarial search problems like chess, the Minimax algorithm is commonly used to determine a player's optimal strategy[4] [3].

The classic version of the Minimax algorithm has only two agents and is predicated upon assigning each player one of two classifications. First, the "Max" player selects those actions that are expected to maximize his/her own utility; in contrast, the "Min" player chooses actions that are expected to minimize the expected utility for the "Max" player. By minimizing the "Max" player's utility, the "Min" player is necessarily maximizing his/her own expected utility.

In classic Minimax, the "Min" and "Max" player alternate moves. Hence, each player's later moves are contingent upon the moves made earlier in the game. This leads to a large decision tree with each player making a single move at alternating levels of the tree.

The Minimax decision tree is classically built recursively running on a single core. Given the number of moves available to a player at any point in Chess, the search tree can be very large and time consuming to calculate. By distributing the Minimax calculations across multiple processing nodes, significant speed-up may be achieved.

[1] Zero-sum entails that the results of the game for each player is equal in magnitude but opposite in sign/direction. For example, if one player wins, the other player loses.
[2] A fully observable environment is one where an agent can see all relevant aspects of the environment's state at once.
[3] In this usage, deterministic is contrasted with stochastic. Hence, any legal move that an agent attempts is guaranteed to succeed.
[4] An optimal strategy is a sequence of moves/actions that will lead to a result at least as good as any other sequences of moves/actions given that the opponent is infallible.

3. Project Proposal

The traditional Minimax algorithm is implemented in a recursive fashion where decisions at the upper levels of tree are dependent on values calculated at lower levels of the tree. Our team developed a cloud based implementation of the Minimax algorithm for chess using workflows. This new paradigm eliminates the need for the algorithm to be run on a single core and opens up possibilities for task parallelism and scalability.

Since the complete search tree for standard chess is too large to realistically be constructed using existing computer hardware (for an average game with n moves the search tree is bounded by $O(30^n)$ [2]), our project will focus on smaller scale variants of the game like Minichess [1]. This simplification of the problem is intended to enable a clearer proof of concept and does not prevent our algorithm being used for a standard game.

4. Survey of Previous Works

Work into computer strategies for chess has been ongoing since the 1970s, and there has been significant progress in particular in the area of tree based searches (e.g. Minimax). Most initial efforts were implemented on computer architectures which had only a single processing unit. This paradigm creates an upper bound where the system's performance is limited by the contemporaneous hardware capabilities of a single machine. In order to remove this limiting factor, subsequent research has been done to parallelize chess; for example, IBM's Deep Blue chess system, which defeated Garry Kasparov in 1996, was a highly specialized, distributed computer [2].

[2] is the only paper we found which successfully implemented distributed Minimax for chess in the cloud. Their application was developed using Heroku's Platform as a Service (PaaS) runtime environment; it also utilized the open source database platform, MongoDB, as the task queue. Their algorithm follows a more traditional top-down approach for determining the best move; it did not take advantage of an alternate tree paradigm (e.g. a Directed Acyclic Graph – DAG) to improve the performance or simplify the flow.

5. Intended Users and Consumers

As with any chess game, the set of possible users is large and diverse; it includes any person with a reliable internet connection who enjoys playing chess on the computer, in particular different chess variants such as Minichess. Users would only need to run the client locally on their PC, and client would communicate directly with a stateless server in the cloud.

6. Theoretical Foundations

This section describes the theoretical foundations for Minimax as well as the new paradigm we developed for simplifying the cloud implementation of recursive decision trees.

6.1 Minimax Algorithm Overview

Minimax is a decision tree based artificial intelligence algorithm for determining a player's optimal strategy in a zero-sum game. Figure 1 is an example decision tree. Each node (shown as a blue circle) in the tree represents a possible state of the game's environment; in the case of chess, a state corresponds to a single turn in the game and has two accompanying data components: a game board and the current player. Nodes in the decision tree may have a set of successor states; the successor/child nodes correspond to new states that result from an action (i.e. chess piece move) performed by the active player.

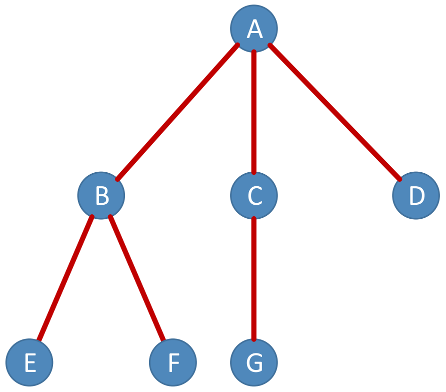


Figure 1 – Example Decision Tree

Leaf nodes in the Minimax decision tree correspond to either the end of the game (e.g. checkmate) or the maximum allowed tree depth. To determine the quality of a given move, each leaf node is assigned a utility, which numerically quantifies the value to the player of being in that resultant state. For example, a winning game state would have the maximum utility while a losing state would have the minimum (i.e. maximum negative) utility.

Figure 2 is pseudocode for the Minimax algorithm [3]. The active player at the top of the decision tree (e.g. node "A" in figure 1) is "Max". For the current "`state`", the Max player has a set of valid actions (i.e. chess moves); this set of valid moves is returned by the function "**`Actions`**". The function "**`Result`**" applies a specific action "`a`" to the current state and returns a new successor state. The function "**`Terminal-Test`**" checks whether the current state corresponds to the end of the game (e.g. checkmate) or whether the maximum allowed tree depth has been reached. If "**`Terminal-Test`**" returns true, then the "**`Utility`**" function is called, which calculates the utility value of `state`.

It is important to note that the functions "**`Max-Value`**" and "**`Min-Value`**" return utility (i.e. numerical) values while the function "**`Minimax-Decision`**" returns the optimal action (i.e. best chess move). This is because when a player invokes the Minimax algorithm, s/he is specifically seeking the best action to perform. In contrast, the other nodes in the tree are used by the algorithm to quantify the value of those initial moves, hence the need to return a number.

```
function Minimax-Decision(state) returns optimal_action
   return argmax_{a ∈ Actions(state)} Min-Value(Result(state, a, 0))

// Max function is called for the "Max" player
function Max-Value(state, depth) returns a utility value
   if (Terminal-Test(state, depth)) then return Utility(state)
   v = -∞

   for each a in Actions(state):
         v = Max(v, Min-Value(Result(state, a), depth + 1))
   return v

// Min function is called for the "Min" player
function Min-Value(state, depth) returns a utility value
   if (Terminal-Test(state, depth)) then return Utility(state)
   v = ∞

   for each a in Actions(state):
         v = Min(v, Max-Value(Result(state, a), depth + 1))
   return v
```

Figure 2 – Classic Minimax Algorithm [3]

6.1.1 Recursive Search through a Decision Tree

The Minimax pseudocode in figure 2 searches through the tree in a manner similar to recursive, depth first search. Figure 3 shows how Minimax would traverse the decision tree in figure 1. Next to each edge in figure 3 is an ordered pair (*x*, *y*), where *x* is the step number when the associated child state is first explored while *y* is the step number when the associated child state returns its utility value.

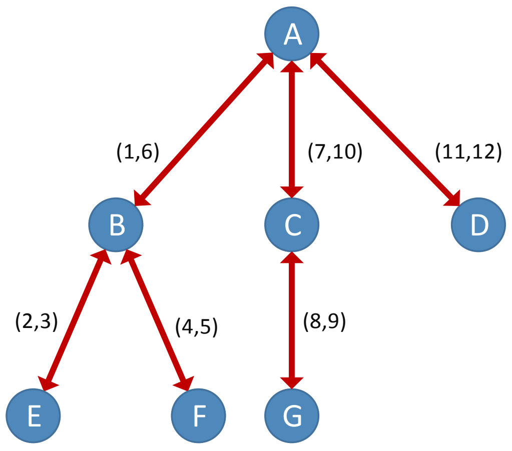


Figure 3 – Minimax Traversal of the Decision Tree in Figure 1

6.1.2 Relationship between Recursion, Scalability, and Space Complexity in Minimax

Since recursive implementations of Minimax rely on a stack to store previous states, its execution is traditionally modeled as if it is run on a single core on a single machine. When computerized Minimax was first described by Claude Shannon in 1950, parallel systems did not exist. What is more, the bounding limitation on Minimax's execution was (and to some extent remains) the space complexity since time complexity can be considered infinite for theoretical purposes.

Parallelizing Minimax will inherently require a greater memory footprint. Due to its use of a stack to store previous states, the recursive implementation of Minimax has a space complexity of $O(d)$ (where d is the maximum tree depth). In contrast, depending on the implementation, massively parallel Minimax could have a space complexity as high as $O(b^d)$ (where d is the maximum tree depth and b is the maximum branching factor).

In a cloud environment, Minimax's space complexity can remain manageable due to resource pooling. What is more, the cloud provides significant opportunities for computational parallelism. However, running Minimax in the cloud is not without its disadvantages. As the level of parallelism increases, the key limiting factor quickly becomes the inter-process communication and synchronization. This limitation will become dominant if the computation time in a given state is relatively low (i.e. has only fine grain parallelism).

6.2 A Cloud Implementation of a Recursive Tree

At first glance, it may appear that there are a large set of different paradigms that could be used to model a recursive tree. The following subsections provide greater insight into this problem including why different architectures are not feasible and why a workflow naturally lends itself to this type of problem.

6.2.1 Observations Regarding Recursive Search and their Consequences

When selecting a paradigm to model the recursive tree, there were a few noteworthy observations.

Observation #1: In Minimax, the number of child nodes generated by each node is not fixed. For example, in figure 1, node "A" has three children; node "B" has two children, and node "C" has only one child.

Consequences of Observation #1: In a decision tree with a predictable branching factor (b), a finite state machine based architecture, while sometimes cumbersome, can be used in place of recursive search. However, given that the number of leaf nodes in chess cannot be known in advance, a finite state machine is impractical.

Observation #2: When a parent node generates one or more child nodes, the behavior is very similar to traditional, nested fan-out architectures. For example, in figure 1, node "A" fanned out three child nodes: "B", "C", and "D"; some of these child nodes subsequently fanned out child nodes of their own.

Consequences of Observation #2: MapReduce is a commonly used cloud paradigm that allows for fan-out (mapping) and subsequent analysis of the results (reducing). However, the scheme is not traditionally used for arbitrary nested fan-outs with implicit data dependencies. As such, any MapReduce based implementation of Minimax would require significant and fundamental changes to MapReduce's typical use model.

Observation #3: As shown in figure 3, edges in a decision tree are traversed twice; the first time is when the child node is created, and the second time is when the child returns its results to the parent. This essentially makes edges bidirectional. If rather than returning its results to a parent, all of that parent's child nodes returned their results to a new node, recursion could be eliminated. However, this would require a synchronization/joining mechanism.

Consequences of Observation #3: By eliminating the need for child nodes to return their results to their parent, the inter-task dependency model is changed. This change allows for new paradigms to be used.

6.2.2 Modeling a Recursive Tree with as a Directed Acyclic Graph via a Workflow

A workflow is a tool used to model and manage task interdependencies in a complex activity. Workflow patterns constitute the mechanisms that define the relationships between these tasks [10]. For this application, the two most important workflow patterns are:

d. **AND Split** – When a specific task completes, it spawns two or more successor tasks, which are executed concurrently (or placed in the queue to be executed). Figure 4 shows a simple AND Split. Note that once task A completes, tasks B and C can be executed.

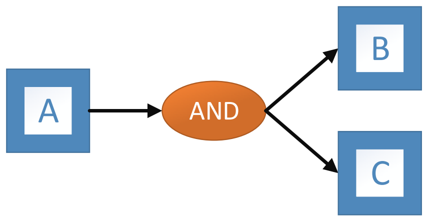


Figure 4 – AND Split Workflow Pattern

e. **Synchronization Join** – Before a successor task can begin execution, all of its predecessors must have terminated. Figure 5 shows a synchronization join. Note that tasks X and Y must both complete before task Z can begin.

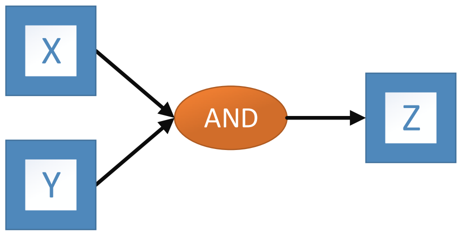


Figure 5 – Synchronization Join Workflow Pattern

Through the use of AND splits, the nested fan-outs described in observation #2 in section 6.2.1 can be trivially modeled. In contrast, modeling observation #3 using synchronization joins requires additional steps. Consider that a decision tree is mirrored along its leaf nodes; if this was done, then the need for child nodes to return data to a parent can be eliminated. Rather, data can be passed to a new node which will handle the post processing previously handled by the parent node. Synchronization joins are used in this architecture to ensure the algorithm's correctness. Figure 6 is an example of this technique applied to the decision tree in figure 1. With the exception of the leaf nodes (e.g. "E", "F", "G", and "D"), all nodes have been mirrored; the mirrored nodes are: A', B', and C'. The mirrored nodes are joined to their child nodes (or the mirrors of their child nodes) through synchronization joins. This makes the system into a type of directed acyclic graph (DAG) which can be trivially traversed in parallel.

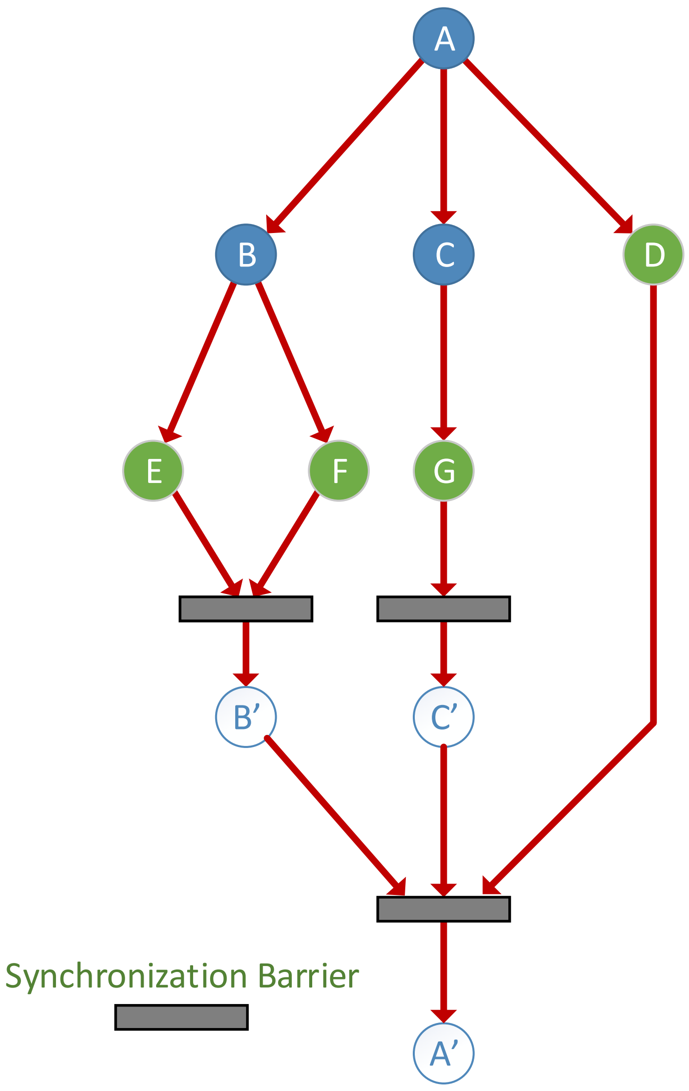


Figure 6 – The Minimax Decision Tree in Figure 1 Transformed into a Directed Acyclic Graph (DAG) through Tree Mirroring and Synchronization Joins

6.2.3 Advantages of Transforming a Recursive Tree into a Directed Acyclic Workflow

Transforming a recursive tree into a directed acyclic graph (DAG) (as shown in figure 6) necessarily changes the data dependencies between nodes. Each state in the workflow now becomes a task that can be allocated to processing nodes in the system; once a set of tasks have been completed, the results can be placed into queues while the system awaits the completion of other tasks that share the same parent. This enables the task level parallelism that was not possible when Minimax was implemented recursively.

6.3 Chess State Utility Modeling

The function “**Terminal-Test**” in figure 2 takes two arguments: the current state and the recursion depth. The need for “`state`” as a function parameter is obvious as the algorithm must determine if the end of the game has been reached. The reason “`depth`” is required is slightly more subtle.

For standard chess, the game-tree complexity is at least 10^{123} [7]. Similarly, for standard chess, the tree size is over 988 million after just 8 moves in the game [8]. Clearly, neither of these trees can be realistically constructed on existing hardware. As such, the algorithm needs a way to limit the tree size while still quantifying the relative quality of a given move; this is done by limiting the search depth (via the `depth` variable) and estimating the overall utility of a given board using a heuristic.

The essential qualities of a heuristic are that it is fast to calculate yet still provides a reasonable estimate of the state’s utility. The most commonly used heuristic is the Chess Piece Relative Value system proposed by [9]; it is shown in table 1.

Piece	♟	♞	♝	♜	♛	♚
Name	Pawn	Knight	Bishop	Rook	Queen	King
Value	1	3	3	5	9	0

Table 1 – Chess Piece Relative Value System Weights

The utility of a given board for a specific player is found by summing the relative value of all of that player’s remaining pieces and subtracting from that the relative value of the opponent’s remaining pieces. Using this heuristic, a larger utility value entails a greater likelihood of *winning* the game while a smaller utility value is an indication of a greater likelihood of *losing* the game.

6.4 Alpha-beta Pruning

Given a Minimax decision tree with branching factor, b, and a tree depth of d, the computational complexity to search the whole tree is bounded by $O(b^d)$. With a large branching factor (e.g. up to 30 or more in the case of chess), the computation complexity can be prohibitively high given the exponential growth.

No technique currently exists that can completely eliminate the exponential growth in Minimax. However, algorithms do exist that can reduce the size of the exponent, which can have tremendous impacts on the runtime of the algorithm. For example, if the branching factor is 30 and the depth of the tree is 6, then the number of states in the decision tree is 729 million. If the branching factor is reduced by a factor of 2 (i.e. from 6 to 3), the number of the states in the tree is 27,000 (in turn a reduction of 27,000 times).

In many cases, the Minimax algorithm can determine the optimal action without exploring all successor nodes. Figure 7 is an example Minimax decision tree. After exploring its first child node, the root node (which is a max) knows that the utility of whatever successor it selects will have a utility of at least 5 (by definition of it being a max node). When the second successor node (shown in the center) searches its first child, it sees that the child has a state utility of "$-\infty$." Since a min node always takes the minimum result value and given that $-\infty$ is less than 5, the min node knows it does not need to check its additional two successor nodes as there is no way the root max node will select it. Similarly, the root's third successor node can prune the tree after seeing a state utility of "2." Hence, the three pruned nodes can be ignored. This paradigm serves as the basis of Alpha-beta pruning. When Alpha-beta pruning is used in conjunction with Minimax for chess, the computational complexity is reduced from $O(b^d)$ to an average bounding of $O\left(b^{\frac{d}{2}}\right)$ [3].

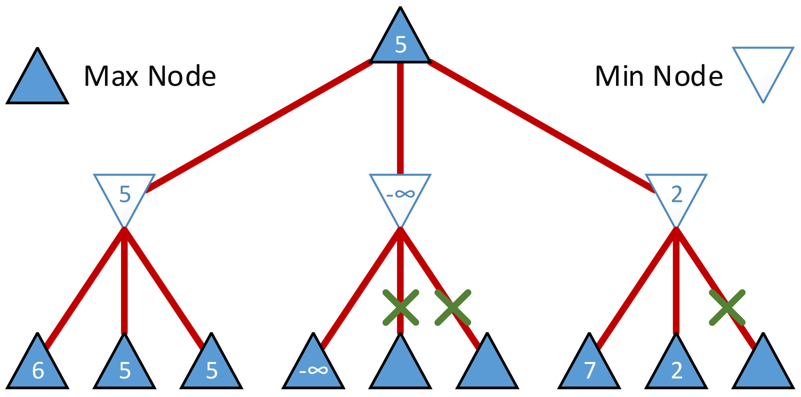


Figure 7 – Example Minimax Decision Tree with Alpha-beta Pruning

7. Project Architecture and Design

There are three primary components required to implement the game of chess in the cloud. They are: a client side application/viewer, application server, and the communication interface. Figure 8 shows the high level interconnects between these three modules. The following subsections describe the architectural details of these components as well as the rationale behind the key design decisions for each module.

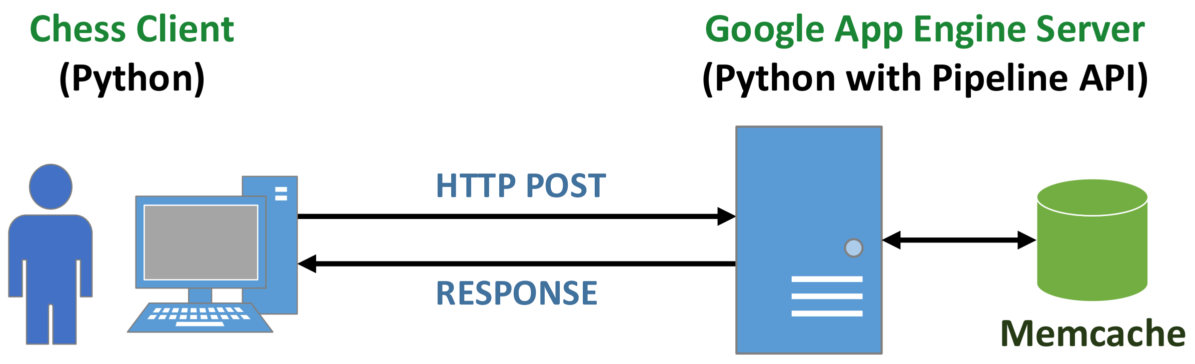


Figure 8 – Simplified Model of Our System's Architecture

7.1 Client Application

One of our key design decisions was to keep the entire architecture and design as lightweight as possible. As an extension of that, we tried to limit the burden on the server where reasonable. For instance, rather than have the client constantly connected to the server, the client application was designed to be to a degree independent, and only make requests of the server when specific data was needed. In a real-world deployment, this would allow the server to manage more requests with the same amount of resources.

The client application serves as the human player's interface with the system. It displays the current board, accepts user inputs, and displays the computer's move. The only time the client application communicates with the server is when it asks the server to decide its next move. The client application transmits to the server over the internet the current state of the game, and the server responds with its selected move (as shown in figure 8). Other than the move request and response, the client application has no other interaction with the server. The client application uses the same functions to process human and computer moves with the exception that at the end of a turn it alternates players.

7.2 Stateless Server

There are multiple different server architectures that could have been used for this project. To simplify the discussion, we will classify these architectures into two categories: lighter-weight, stateless servers and heavier-weight, stateful servers. Both categories of servers have specific advantages and disadvantages that are described in the following subsections.

7.2.1 Advantages and Disadvantages of a Lightweight, Stateless Server

A stateless server treats each client request as an independent transaction. As such, the server does not need to allocate memory resources to manage user sessions. This approach greatly reduces the complexity and memory footprint of the server process by transferring all of state management to the client application. What is more, this paradigm significantly simplifies the debug of server-side issues as it eliminates external dependencies on individual transactions. Finally, a stateless server can serve as an intermediary milestone if the final goal is a stateful server; such intermediary milestones are often useful in identifying key architectural deficiencies earlier in the process.

Stateless servers are not without their limitations. Chess by its nature is a stateful game. Each move is inherently dependent on all of the previous moves that came before it. What is more, a stateless server necessarily will need to rebuild the Minimax tree on subsequent, related transactions; this will invariably have a performance impact on the system.

7.2.2 Advantages and Disadvantages of a Heavyweight, Stateful Server

There are multiple different forms a stateful server could take for this type of application. In this section, we will discuss the heaviest weight style as it would provide the most contrast with a stateless server.

A stateful server could be implemented as a complete Software as a Service (SaaS) application. The server could store the state of the game in its memory and then transmit only the minimum amount of data needed by the client. This model shifts the system complexity from the client side to the server side. Hence, the client becomes much more tightly coupled with the server; this necessarily entails that the client becomes far less immune to server failures. However, because the server is storing state information, the server has the ability to pre-calculate potential client requests greatly improving overall system response time.

7.2.3 Selection of a Stateless Server Paradigm with Caching

When deciding whether to use a stateless or stateful server, we needed to be pragmatic regarding the overall scope of the project. If time and resources were limitless, a stateful architecture's superior response time would have been the deciding factor; if a user waits more than a few seconds for the computer to make its move, s/he may find the program sluggish and stop using it. However, at the same time, we needed to ensure we set achievable goals for ourselves.

Caching on a stateless server provides an acceptable medium between the superior performance and feature set of a stateful server versus the reduced complexity of a stateless architecture. Specific benefits of caching include:

a. **Performance** – Depending on the size of the cache, the most recent requests and frequently used boards will remain in the cache. This will allow for much of the performance benefits of a stateful server, but with greatly reduced complexity and overhead.

b. **Bounded Memory Footprint** – As more players and games are added to the stateful server, the memory footprint will also grow. By using caching, the designer of the stateless server is able to place an upper bound on the memory footprint of the application.

7.3 RESTful Communication

Regardless of the implementation of the client and server, the two modules will invariably need to communicate. Since these two processes do not reside on the same physical machine, the best paradigm for communication is through the internet.

One of our design goals was to avoid adding unnecessary complexity to the system unless there was a very compelling reason to do so. While SOAP based communication was always a possibility, we chose to instead use a REST based architecture; specific benefits and reasons for using REST include:

c. **Sufficient Feature Set** – REST provided all of the necessary features our application needed. The additional features available in WSDL-based web services did not add compelling value.

d. **Lightweight** – REST's lightweight design improves system performance while at the same time having less implementation complexity.

e. **Data Security** – For a chess application of this type, the security requirements for individual transactions are essentially zero. As such, SOAP's superior security capabilities were not needed.

f. **Stateless Server** – Section 7.2 details the design decisions behind our selection of the stateless server paradigm. As such, the communication protocol did not need to manage sessions between the client and server.

g. **Control Over Client-Server Interface** – Since we developed both the client and server side of the application, we had complete control over the requirements and implementation of the communication interface. This reduces the need for a more verbose and rigorous communication protocol like SOAP.

Given these factors, we saw very tangible benefits associated with using REST for this project. What is more, we did not see that the additional benefits provided by a WSDL-based protocol sufficiently offset the associated implementation complexity and overhead such a protocol introduced. This is the primary reason we selected a RESTful-style protocol for our design.

8. Project Implementation

Section 7 enumerates the three major components in our system specifically: the client application, stateless server, and communication interface. When implementing our system, a fourth component was required which handled the chess specific functionality. All four of these blocks are described in the following subsections.

8.1 Chess Legal Moves and "Check" Functionality

Implementer: David Smith
Programming Language Used: Python 2.7.8
Filename: chess_utils.py

For chess to be playable on any computer system, it is necessary to implement all the game rules. While the fundamentals of this implementation do not vary significantly between a cloud-based architecture and a local system, this does not in any way detract from the importance and complexity of this portion of the project.

Given a game board and the current player, this component of the system determines two key pieces of information:

a. **Check** – A player is in "check" when the opponent has a piece that is attacking that player's king. When a player is in check and not checkmate, the player is required to make a move which removes his/her king from check.

b. **Legal Moves** – The rules of chess dictate how each piece can move. The chess system must determine the set of valid moves for an individual player on a given game board.

The functions used to determine whether a player is in check as well as the set of legal moves for that player are implemented in the `ChessUtils` class in the module "chess_utils.py". Since the system must verify the validity of a human player's move as well as determine the set of valid moves for the computer's turn, this module is used by both the client and server applications.

8.2 Client Application

Implementers: Geetika Garg & Zayd Hammoudeh
Programming Language Used: Python 2.7.8
File Name: chess_client.py

The client application is intended to have two primary functions. First, it serves as the user interface for the player; the player enters his/her moves into the client and then uses it to view the progression of the game. Second, the client application invokes the commands to communicate with the Google App Engine (GAE) server.

The client application is written in Python and is purely terminal based. To run the client on a computer which already has Python installed, the command is:

```
python chess_client.py BoardFileName
```

Note "BoardFileName" is the path to a chess board file (see section 8.2.1 for more information on the file format). Figure 9 shows the initial loading of a 4x4 Minichess board in the client application on a Windows 8 PC. White pieces are prefixed with the letter "W" while black pieces are prefixed with the letter "B". Table 2 details the character used when printing each piece on the board. For example, a black knight would be displayed as "BN" while a white king would be displayed as "WK".

After loading the board, the client prompts the human player whether they want to be the white or black player. To be white, the user enters "0", and to be the black player, the user enters "1".

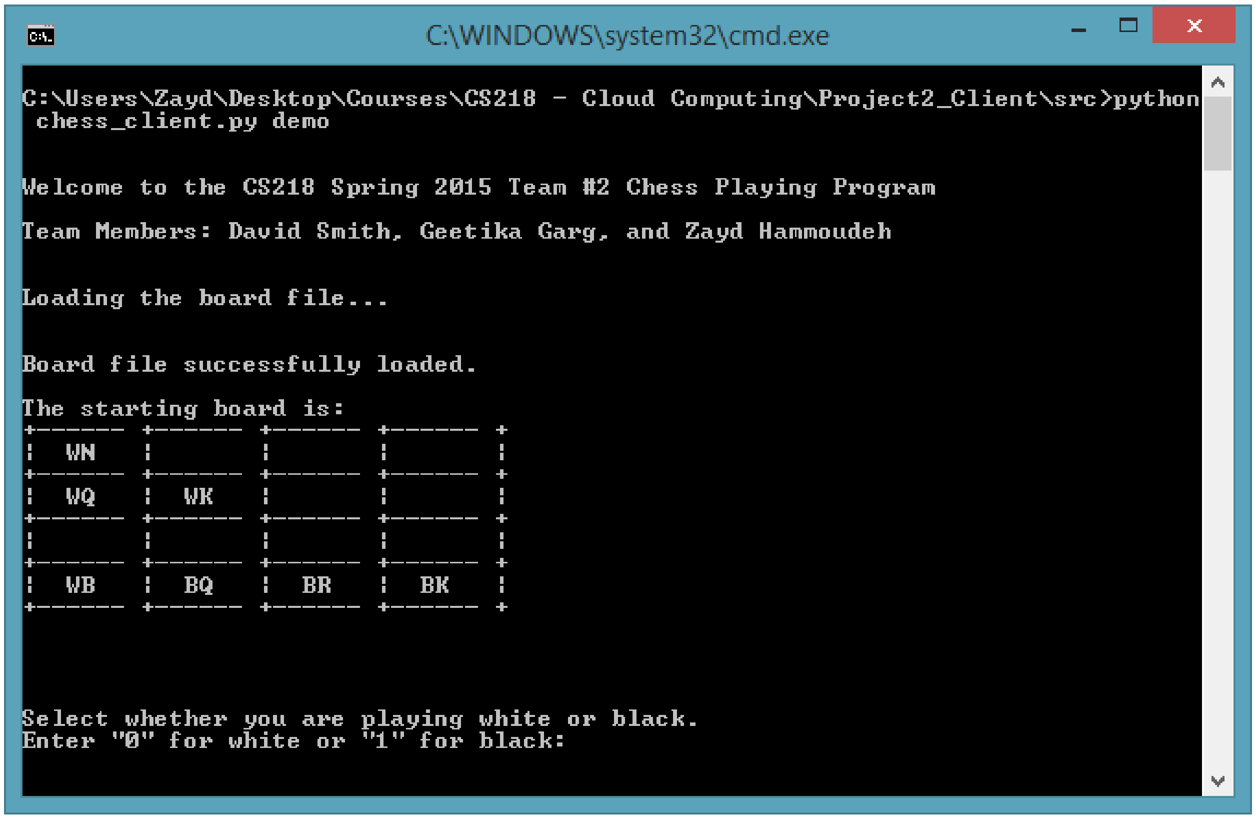
Piece	♟	♞	♝	♜	♛	♚
Name	Pawn	Knight	Bishop	Rook	Queen	King
Character in Terminal Print	P	N	B	R	Q	K

Table 2 – Characters Used to Represent the Chess Piece Types when Printing to the Terminal

Each square on the board is assigned an identification number. Given a square board whose side is length l, the cells in the top row are numbered from 0 on the left to $l - 1$ on the right. Similarly, the cells in the second row are numbered from l on the left to $2l - 1$ on the right; this numbering scheme continues for all subsequent board rows.

Moves for both the human and the computer players are in the format "*CurrentSquare, DestinationSquare*". For example, to move the black king in figure 9 up one square, the move would be "15, 11". Similarly, move "0, 9" would allow the white knight (in the top left corner of the board) to remove the white king from check.

When it is the computer's turn to make a move, the client transmits the board to the server and waits for it to respond with a selected move. The client then applies the server's selected move at which point it allows the human player to select another move. This continues until either one player is checkmated, or no moves are possible (resulting in a draw).



```
C:\Users\Zayd\Desktop\Courses\CS218 - Cloud Computing\Project2_Client\src>python
 chess_client.py demo

Welcome to the CS218 Spring 2015 Team #2 Chess Playing Program

Team Members: David Smith, Geetika Garg, and Zayd Hammoudeh


Loading the board file...


Board file successfully loaded.

The starting board is:
+------- +------- +------- +------- +
|  WN    |        |        |        |
+------- +------- +------- +------- +
|  WQ    |  WK    |        |        |
+------- +------- +------- +------- +
|        |        |        |        |
+------- +------- +------- +------- +
|  WB    |  BQ    |  BR    |  BK    |
+------- +------- +------- +------- +




Select whether you are playing white or black.
Enter "0" for white or "1" for black:
```

Figure 9 – Chess Client Running on a Windows 8 PC

8.2.1 Chess Board File

As mentioned previously, the chess client is passed a board file name. This board file is an ASCII text file in Comma Separated Values (CSV) format. The first row in the file indicates the player whose turn is next; "0" indicates the white player, and "1" represents the black player.

The remaining rows in the file represent individual rows on the chess board. Each comma separated value corresponds to a single cell/square on the board. The board file requirements are:

a. **Board Size** – The board must be a square (i.e. have the same number of rows as columns)

b. **Single King** – Each player must have exactly one king.

c. **Piece Placement** – White starts at the top of the board and moves towards the bottom of the board while black starts at the bottom of the board and moves towards the top. This requirement is most important for pawns.

Individual pieces on the chess board are denoted by integers. The pairing of integer values to pieces is shown in table 3; note that if a square has no piece in it (i.e. is a "blank square"), then the comma separated value is a "0". To differentiate a black piece from a white piece, a "1" is prefixed before the piece number for black pieces; there is no prefix for a white piece. Figure 10 shows an example board file for a standard game of chess. Note that the white player plays first, and that the board is 8x8.

Blank Square	Pawn	Rook	Knight	Bishop	Queen	King
0	1	2	3	4	5	6

Table 3 – Integer Values for Piece Types in a Board File

```
0
2,3,4,6,5,4,3,2
1,1,1,1,1,1,1,1
0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0
0,0,0,0,0,0,0,0
11,11,11,11,11,11,11,11
12,13,14,16,15,14,13,12
```

Figure 10 – Example Board File for the Initial Move of a Game of Standard 8x8 Chess

8.3 Server Application

Implementer: Zayd Hammoudeh & Geetika Garg
Programming Language Used: Python 2.7.8 with Google App Engine (GAE), GAE's Pipeline API, and GAE's Task Queue
Deployed Address: http://cs218-team2-chess-minimax.appspot.com
Filename: cloud_chess_Minimax.py

The third component of our architecture is the server, which runs in the Google App Engine (GAE) environment. The primary class in our application is named "`MainPage`", which inherits the Python `webapp2` class; `webapp2` comes bundled with the GAE Python 2.7 environment and is intended as lightweight web framework that allows users to quickly build and deploy simple web applications [12]. Our application overrides the `webapp2`'s `post` method to allow our system to process any HTTP POST requests that are sent to the server (the communication architecture is described in more detail in section 8.4).

Our server uses GAE's Pipeline API to create a Minimax workflow. In the following subsections, we provide an overview of the Pipeline API as well as how it is used to create a

Minimax workflow. Once a workflow has terminated, the Pipeline API returns what the system has determined is the best move; this move is then transmitted to the client application.

8.3.1 Google's Pipeline API

Implementer: Google with Very Minor Modifications by Zayd Hammoudeh
Programming Language Used: Python 2.7.8 with Google App Engine API
Package Name: pipeline
Original Source Code Link: https://github.com/GoogleCloudPlatform/appengine-pipelines

Google's Pipeline API was introduced in 2011 and is available in both Python and Java. Its architecture relies on a set of assumptions:

a. Non-deterministic task structure at the start of the process
b. A large number of concurrent tasks
c. Available task queue to store and manage the different tasks

The architecture's key goals were:

a. Enable coordination between tasks through passed arguments and the directing of outputs/results
b. Simplify the joining of parallel tasks
c. Elastic scaling

Since our application is Python based, we will exclusively discuss the Pipeline API's Python architecture, although the Java architecture is similar [13].

The foundation of the Pipeline API is the `Pipeline` class. All user tasks are encapsulated within a user defined class, which inherits `Pipeline`; any specific user code is placed inside the class' "`run`" method, which the user is overriding. Figure 11 shows one of our user defined classes "`DetermineBestMove`" that inherits the `Pipeline` class.

```
class DetermineBestUtilAndMove(pipeline.Pipeline):
    def run(self, is_max, current_best_util_and_move, valid_moves,
             *utilities_and_moves_out_of_states):

        # Extract head of the list and use for comparison.
        best_move = valid_moves[0]
        best_util = move_utilities[0]
        …
```

Figure 11 – Example "`DetermineBestUtilAndMove`" Class that Uses the GAE Pipeline API

In most cases, the "`run`" method is not a normal function, which when invoked executes. Rather, it usually is a special type of Python function known as a generator. Generators allow the system to improve performance and reduce data dependencies through lazy evaluation (i.e. results are only calculated when they are actually needed) [14]. If a value is not presently needed, then the value's associated calculation can be delayed by placing the task in a queue.

The following is an example[5] showing how a Pipeline API workflow could be used to calculate a second-order polynomial function $f(x)$ defined via the equation:

$$f(x) = ax^2 + bx + c$$

Note that this example is deliberately trivial. Its role is to explain the necessary concepts and is not intended as a demonstration of the tool's full power.

The calculation of $f(x)$ can be broken down into three tasks. The first two are multiplication tasks, which determine ax^2 and bx, and the third task is an addition where the three terms (e.g. ax^2, bx, and c) are summed together. Tasks in these figures are represented as green circles, and the results of these tasks are stored in memory slots, which are shown as gray cans; the red rectangle is a barrier synchronization (i.e. join).

In the first stage of execution shown in figure 12, the two "Multiply" tasks are run; note that their execution can be done in any order (including in parallel) since they do not share mutable data dependencies. Once a task's execution has completed, the task fills its associated memory slot with its results and notifies the synchronization barrier.

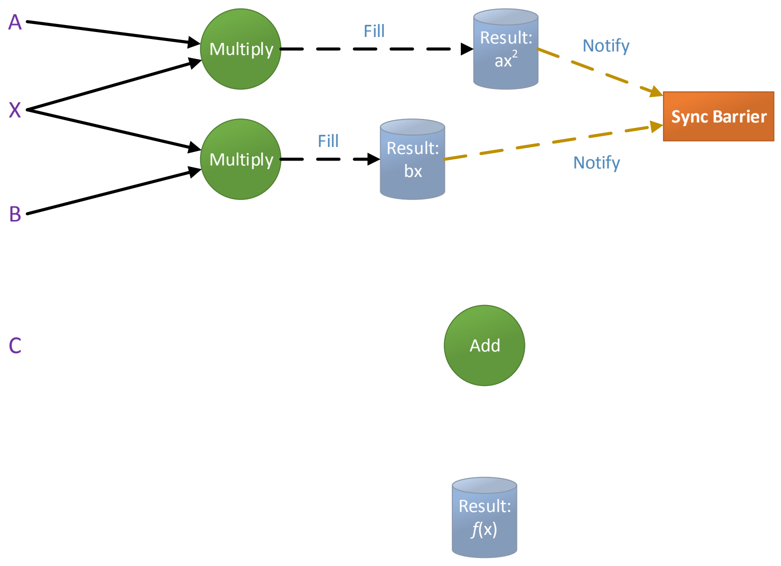


Figure 12 – Start of the Pipeline API Workflow for a Second-Order Polynomial Function

[5] This Pipeline API example was presented in [13]. It is only slightly modified for this report.

Once the synchronization barrier has been notified by all of its dependent tasks, it notifies the "Add" task. The "Add" task then retrieves all of its required data and begins execution. When this second stage of computation has completed, the "Add" task stores its results in its own memory slot for retrieval by another process. The complete workflow, including this second stage is shown in figure 13.

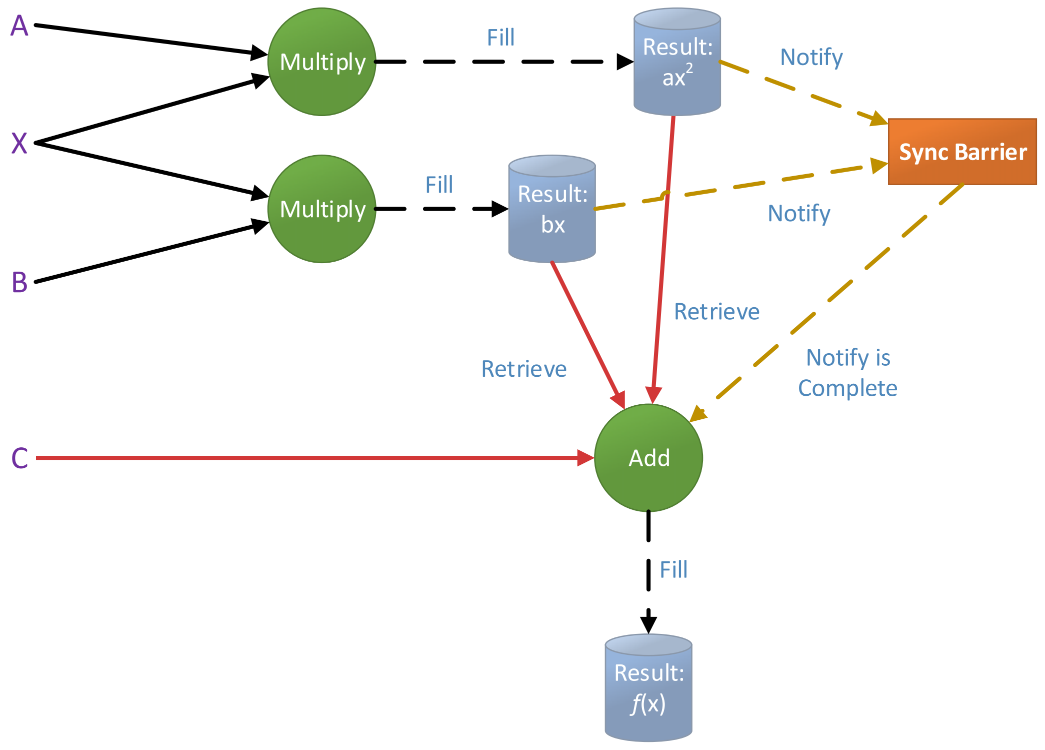


Figure 13 – Completion of the Pipeline API Workflow for a Second-Order Polynomial Function

The Pipeline API automatically manages the filling of the memory slots, task notifications, and synchronization barriers. The programmer only needs to define the task modules and input data streams. This framework greatly simplifies what would otherwise be very complex and onerous.

8.3.2 Using the Pipeline API to Create a Minimax Workflow

Tasks in the Pipeline API are objects. The objects' types are derived from (i.e. inherit) Google's `Pipeline` class. Our architecture has five primary task classes; they are:

a. `MakeMove` – This class constitutes the root of the Minimax decision tree. It spawns (fans-out) the initial set of successor tasks at depth 1 of the decision tree. Once all of the successors have completed processing, `MakeMove` returns a move, which is a two element Tuple of integers.

b. `MakeMoveIter` – For Alpha-beta pruning to be possible, the algorithm must have the results from previous states to determine whether it can prune the tree. This class divides Minimax successor nodes into "batches" that are processed sequentially; this is done by spawning additional objects of type `MakeMoveIter`. Based off previous results (either calculated or in cache), this class can prune the search tree and not process subsequent batches unnecessarily.

c. `InnerMakeMove` – An inner processing class. It is performs the "`Terminal-Test`" described in figure 2. If the test returns true, it runs the "`Utility`" function to determine the state's utility, otherwise, it generates the successor objects of type "`MakeMoveIter`".

d. `DetermineBestUtilAndMove` – This class serves as the join/synchronization mechanism. It waits for all of the `InnerMakeMove` objects to return their utility values at which point it selects the move with either the minimum or maximum utility as dictated by whether the current node is of the "Min" or "Max" type.

e. `AddToMemCache` – This class writes a state's utility and best move to the cache. While this item could have been placed inside the `MakeMoveIter` class, performing the operation inside a Pipeline object allows it to be done off the critical path, improving system performance.

Through the combination of the first four task classes, we are able to model the complete Minimax tree shown in figure 6. The `MakeMove` class would map to node "A" in the figure, while nodes "B", "C", "D", "E", "F", and "G" are represented by `MakeMoveIter` and `InnerMakeMove` together. Synchronization joins are managed by the Pipeline API architecture itself. The mirrored tasks A′, B′, and C′ would be of type `DetermineBestUtilAndMove`. What is more, since the links between tasks are formed in real time, any arbitrary decision tree structure can be supported.

8.3.3 Server-Side Caching

Implementer: Geetika Garg
Programming Language Used: Python 2.7.8 with GAE's Memcache Datastore
Filename: cloud_chess_Minimax.py

As described in section #7.2.3, caching allows a stateless server to have better performance by storing the results of recent and/or frequent requests. In Minimax, states will often be re-explored multiple times as the algorithm searches the state space. Without caching, each time a node is revisited, the system would need to waste time and resources recalculating the results.

To eliminate the need to recalculate results, we implemented a caching architecture in our application using Google App Engine's (GAE) Memcache datastore. The specific benefits of GAE's Memcache are:

a. Highly scalable and global so regardless of the number of instances, all tasks can share a single datastore repository.

b. High throughput – Up to 10,000 operations per second.

c. Scalable Capacity – From 1 to 20GB. [15]

Each item in Memcache has a dedicated key, which has three attributes:

a. `next_player` – Player that will make the next move.

b. `board` – The current game board.

c. `is_max` – True if `next_player` is a "max" node and false otherwise.

Entries in Memcache are a tuples of two parameters, which are:

a. `utility_and_move` – Itself a tuple of two parameters. The two parameters are the state utility and best move associated with the combination of the variables "`next_player`", "`board`", and "`is_max`" used in the key.

b. `tree_depth` – In a decision tree, a heuristic is used to calculate the utility of leaf node that are not terminal states. The utility calculations for nodes higher in the tree are much more accurate than for those lower in the tree; this is because the higher nodes have many successor nodes that are explored to inform the algorithm about the upper node's quality. Due to this, our algorithm only uses the Memcache value if it was determined at the equivalent level in the tree or higher. Otherwise, the algorithm calculates the utility of the node again (this time exploring more successors) and upon completion updates the Memcache.

Before determining a state's successor, our algorithm first checks if the solution already exists in Memcache. If it does and the tree depth is acceptable, the algorithm immediately returns the stored result without the need to perform additional processing. Otherwise, it continues processing the results as normal.

8.3.4 Alpha-beta Pruning

Implementer: Geetika Garg & Zayd Hammoudeh

Programming Language Used: Python 2.7.8 with Google App Engine (GAE) and the GAE Pipeline API
Filename: cloud_chess_Minimax.py

As mentioned previously in section #8.3.1, Alpha-beta Pruning relies on the results from previous calculations in order to prune the search tree. Our algorithm is able to generate/extract these results via two different approaches:

a. `Batch Processing` – In traditional, recursive Minimax, nodes are processed one at a time. This allows nodes at the same level in the tree or higher to determine if any successors states can be ignored. As such, our algorithm is designed to allow successor nodes to be processed based off a user specified batch size, with each batch processed sequentially (if at all).

b. `Caching` – As described in section #8.3.3, the results of previous state calculations are stored in Memcache. Given the previously calculated values of alpha and beta, the algorithm first checks in the cache if any of its *successors* (i.e. not itself) show that the tree can pruned. If so, the algorithm does not expand any of the successors (even those not in Memcache) and returns the cached result.

Some may incorrectly argue that segmenting the search tree into batches will cause our distributed Minimax algorithm to run slower. If the GAE engine had unlimited processing nodes and there was no overhead to starting new tasks and retrieving their results, then the assertion would be correct. However, there is a finite quantity of computational resources, and significant time can be wasted processing nodes that have no chance of being optimal. What is more, while the computation time at each level of the tree increases linearly due to batch processing, the potential speed-up is exponential. As such, while our implementation of Alpha-beta pruning may run slower in some niche cases, it will run substantially faster in most others.

It is also important to note that the advantages and importance of Alpha-beta Pruning cannot be fully appreciated when transactions are considered individually. Rather the primary benefits manifest themselves when there are a large number of requests competing for the system's finite resources. In such cases, the efficiency improvements provided by Alpha-beta Pruning become even more critical.

8.3.5 Randomization of Poor Moves

In traditional Minimax, the computer always selects the best move. Since human players are by nature infallible creatures, we experimented with including a randomization error to make the algorithm behave more like a human opponent. Given that this invariably made the algorithm's performance worse, this approach was quickly abandoned. However, we have included this code in the file "cloud_chess_minimax_with_randomization.py" with our submission. It was not kept up to date with the rest of the application so it is no longer supported. Rather, we included it for documentation purposes.

8.4 Client-Server Communication

Implementer: Zayd Hammoudeh
Programming Language Used: Python 2.7.8
File Name: chess_client.py

Since the client runs locally on the human player's computer, it needs to communicate with the Google App Engine server through the internet. Since the requests are very simple by nature, we wanted to keep the communication paradigm as lightweight as possible. Table 4 contains the details regarding the protocol as well as the data formats used in the client-server communication.

Request Protocol	HTTP
Request Type	POST
Request Format	Percent Encoded String of Tuples
Response Format	Percent Encoded String

Table 4 – Client-Server Communication Protocol and Format Information

8.4.1 Client HTTP POST Request

To determine the computer's move, the client sends an HTTP POST request to the server. Figure 14 shows the request source code; it uses Python's "`urllib`" and "`urllib2`" libraries.

In the HTTP request, there are four fields:

c. `current_player` – The player (e.g. white or black) that will make the next move.

d. `board` – The current game board (i.e. all remaining pieces and their location) encoded as a Python list of lists (i.e. a two dimensional matrix) of integers.

e. `tree_max_depth` – Maximum depth of the minimax search tree.

f. `batch_size` – Used in Alpha-beta pruning to specify the number of child nodes that are processed simultaneously before attempting to prune the search tree. By setting `batch_size` to 1, the algorithm would duplicate the execution of traditional, recursive Minimax. In contrast, setting `batch_size` to infinity disables Alpha-beta Pruning as the nodes in the tree are unable to share predecessor values, which is necessary for pruning.

The function "`url.urlencode`" converts these data fields which were originally stored in a dictionary (i.e. hash table) data structure into a string in percent encoded format; the string is composed of a sequence of two element Tuples with the first element in each Tuple being a dictionary key and the second being the value associated with that key [11]. The command "`urllib2.urlopen`" then transmits the data string to the server and waits for a response.

```
def execute_server_command(current_player, board, queue):

    # This data is passed to the Google App Engine Server via a post.
    board_data_dict = {'current_player': str(current_player),
                       'board' : str(board) }
    # Encode the board data for sending to the server.
    post_board_data = urllib.urlencode(board_data_dict)
    # Build the server request.
    server_request = urllib2.Request(server_url, post_board_data)

    # Make the request and wait for a response.
    try:
        server_response = urllib2.urlopen(server_request)
    except:
        print "\n\nNetwork communication error.  Exiting..."
        sys.exit(0)

    # Get the server request
    unparsed_move = server_response.read()
    parsed_move = unparsed_move.split(",")
    computer_move = (int(parsed_move[0]), int(parsed_move[1]))
    # Put the move onto the queue.
    queue.put(computer_move)
```

Figure 14 – Client HTTP POST Request Source Code

To improve the responsiveness of the client application, the HTTP POST request is issued inside a dedicated thread. This allows the GUI to be updated every few seconds to indicate to the human player that the application has not hung or crashed. While this is more of an aesthetic feature, it is important to show to the human player that the system is still alive since calculating the computer move can take many seconds.

8.4.2 Server Request Processing

When the server receives the request, it needs to parse the request information. As mentioned in the previous section, Python converted the data fields into a string that encoded a set of two element Tuples. The `webapp2` Python framework can automatically extract the POST request's field values using the command "`self.request.POST.get`". However, all of the values are in the format of a string. As such, before the server application can use these values, it needed to convert them back to their original format (i.e. integer and list of lists for the "`current_player`" and "`board`" fields respectively). Figure 15 contains our source code for processing the fields in the client's request.

```
def post(self):

    # Extract the POST information from the client.
    current_player = int(self.request.POST.get("current_player"))
    board_str = self.request.POST.get("board")
    tree_max_depth = int(self.request.POST.get("tree_max_depth", 2))
    use_memcache = self.request.POST.get("use_memcache", "True")=="True")
    # Default batch size is fully parallel
    batch_size = int(self.request.POST.get("batch_size", sys.maxint))

    # Rebuild the board from the string passed in the message.
    split_on_header = board_str.split("[[")
    board_row_strings = split_on_header[1].replace("]]","").split("], [")
    board = []
    # Parse the rows
    for row_string in board_row_strings:
        split_row = row_string.split(", ")
        new_row = []
        # Parse the individual cells in a row.
        for cell in split_row:
            new_row.append(int(cell))
        board.append(new_row)
```

Figure 15 – Server Source Code for Processing Client Request Data

8.4.3 Server Response Creation

Once the server has finished processing the client's request, it sends back its selected move. Similar to the client request, we choose to keep the server response lightweight. As such, the server only responds with two integers separated by comma (i.e. the format of a move as described in section 8.2). For example, if the client's move was to move from square 0 to square 3, the server response to the client's HTTP POST request would simply be "0,3". The function "`self.response.write`" in figure 16 is the GAE command to append a string to the HTTP response. Once the GAE "`post`" method terminates, the response is automatically transmitted to the client.

```
# Return the best move to the client.
self.response.write(str(best_move[0]) + "," + str(best_move[1]))
```

Figure 16 – Server Response Source Code

8.4.4 Client Processing of the Server Response

Upon receipt of the server's response, the client application converts the comma separated string into a two element Tuple of integers; the application then processes the computer's move as it would the human player's move. The application next gives the human player an opportunity to make a move creating a cycle.

9. Test Plan Methodology and Execution

To ensure the robustness of our system, our team implemented a multi-tier testing strategy. The four primary levels of testing in our architecture are: assertion based testing, method unit testing, module level testing, and complete system testing. By using a bottom-up testing methodology, we were able to detect, fix, and correct most bugs at the method level, which greatly simplified the module and system level testing [5].

9.1 Testing Reduction through Reduced Code Duplication

Thoughtful architecture design can significantly reduce the amount of testing that a system requires. One of the steps we took when designing our system was to write both the client and server programs in Python even though some of us felt more comfortable with other languages (e.g. Java). By using the same language for both components of our system, we were able to share significant amounts of code between these two system modules. That not only reduces the development time, but it also reduces the verification time because a module only needed to be tested once for both blocks.

9.2 Assertion and Exception Based Testing

Implementer: Zayd Hammoudeh

Assertion checks are a useful tool to verify the computational integrity of key sections of a software program; they act as an invariant check and ensure that a section of code's actual results match the programmer's expectations. Since these assertion statements can be inserted anywhere in the code, they provide the finest level of granularity when checking for correctness; in the simplest terms, the assert statement acts as a "first line of defense" for detecting a wide variety of code issues.

Figure 17 is a section of code from our "`get_state_utility`" method where the Python "`ValueError`" exception is raised if either of the following two criteria is not met:

a. `piece_id` matches either one of the six valid chess piece types (e.g. pawn, rook, bishop, knight, queen, king) or a blank square on the board.

b. `piece_id` belongs to one the two possible chess colors (e.g. white and black)

```
# Go through all the possible pieces and assign a value
if(piece_id == ChessPiece.BLANK): piece_value = 0
elif(piece_id == ChessPiece.PAWN): piece_value = 1
elif(piece_id == ChessPiece.ROOK): piece_value = 5
elif(piece_id == ChessPiece.BISHOP): piece_value = 3
elif(piece_id == ChessPiece.KNIGHT): piece_value = 3
elif(piece_id == ChessPiece.QUEEN): piece_value = 9
elif(piece_id == ChessPiece.KING): piece_value = 0
else:
    raise ValueError("Invalid board piece: " + str(piece_id))

if(piece_color == PlayerType.WHITE):
    white_score += piece_value
elif(piece_color == PlayerType.BLACK):
    black_score += piece_value
else:
    raise ValueError("Invalid board piece color: " + str(piece_id))
```

Figure 17 – Exception Based Testing Example from the "`get_state_utility`" Method

Additional examples where assertions and exceptions are used in our system to verify correctness are:

a. `print_board` – Method for displaying the current chess board's state
b. `make_move` – Method for applying a player's selected move to the board
c. `MakeMove.run` – Primary Google App Engine Pipeline API class. Ensures that the server does not receive an invalid input from the client.
d. `parse_board` – Function used to parse and verify the correctness of strings read in from a user specified board file.

9.3 Method and Function Unit Testing

Implementers: David Smith & Zayd Hammoudeh

Similar to Javadoc for Java, Python allows programmers to include a documentation string for each function/method. Figure 18 is an example documentation string for the function "`query_server`". Note that the documentation string (shown in green) is bookended by a set of three quotations marks and that it immediately follows the function definition.

```
def query_server(current_player, board):
    """
    Makes a request of the server to get the computer's
    move.

    Params:
    current_player : Integer - 0 for white and 1 for black.
    board - List of list of integers of the pieces on the board.

    Returns: Tuple of two integers.  Index 0 in the Tuple is the
    current (i.e. source) location of the piece to be moved
    while index 1 is the new (i.e. destination) location of the piece.
    """
```

Figure 18 – A Python Documentation String for Function "`query_server`"

`doctest` is a module that comes bundled with both Python 2.7 and 3.0; it allows the programmers to embed automated unit testing capability inside a function's documentation string [6]. Through the use of these method/function unit tests, we were able to screen for bugs in the functions themselves, which eliminates the burden to test these use cases at the system level. What is more, because `doctest` is automated, we are able to easily and quickly verify that no new bugs are introduced into the code whenever changes are made.

To insert a doctest, a new line is added to the documentation string with three consecutive "greater than" signs (i.e. ">>>") followed by a space, the method name, and any input parameters. The next line in the documentation string must contain the function's expected results. When the function "`doctest.testmod()`" is called, `doctest` runs all of a module's unit tests and performs a string compare between the method's actual result and the expected result specified in the documentation string. After the completion of `doctest`, the system alerts the programmer of any result mismatches.

Figure 19 is an example documentation string for the method "`get_player`" with five `doctest` unit tests included. The first three unit tests verify that the function returns the results we expect. The second two unit tests are used to verify the assertion/exception error handling that was described in section 9.2.

```
def get_player(piece_player_num):
    """
    Given the specified piece number, this function returns whether it
    belongs to white or black.

    Param:
    piece_player_num - Integer - Piece number the combines player and piece ID

    Returns: Color corresponding to the piece with number "piece_player_numb".

    >>> ChessUtils.get_player(0)
    0
    >>> ChessUtils.get_player(10)
    1
    >>> ChessUtils.get_player(19)
    1

    >>> ChessUtils.get_player(20)
    Traceback (most recent call last):
     ...
    ValueError: The number "20" does not correspond to a valid color.

    >>> ChessUtils.get_player(-1)
    Traceback (most recent call last):
     ...
    ValueError: The number "-1" does not correspond to a valid color.
          """
```

Figure 19 – Python Documentation String with `doctest` Included for Method "`get_player`"

Functions/methods in our system that have `doctest` unit tests are:

a. `get_piece` – Method that determines the type of chess piece (e.g. pawn, knight, king, etc.) associated with a specified integer.
e. `get_player` – Method that determines the color associated with a specified integer.
f. `make_move` – Method that applies a specified move to a specified game board.
g. `in_bounds` – Method that determines whether a specified square (as an x, y pair) is valid for a specified game board.
h. `in_check` – Method that determines whether a specified player is in check for a specified game board.
i. `get_valid_moves` – Method that determines the set of a legal moves for a specified player on a specified board.
j. `get_state_utility` – Method that determines the utility for a specified player given a specified board.
k. `print_board` – Method that prints a specified board to the console.
l. `parse_board` – Function that parses strings read in from a user specified board file.

We also endeavored to have someone other than a function's author generate unit tests so as to increase the likelihood of finding errors.

9.4 Module Level Testing

As part of our multi-tiered test plan, we developed our client and server application code in such a way as to enable stand-alone testing of the individual modules before attempting to test the full system. The following subsections describe these code development and testing strategies.

9.4.1 Module Level Testing of the Client Application

Implementer: Zayd Hammoudeh

As described previously, our application allows a human to play against the computer. The human interacts with the computer through the client application.

One possible strategy to test and debug the client is to wait for the server application to be fully developed and then to debug the client code with the server in the loop. This approach makes it significantly more difficult to track down issues, as the location of a particular bug may not be initially obvious.

We decided a much better strategy was to develop the chess client to support a human to play against another human. This approach allows us to test and debug the client code with as few outside variables as possible. In addition, this methodology allows us to test vastly more use cases as we are not restricted to whatever moves the computer makes. Hence, this strategy allowed us to increase both the test coverage and test flexibility for the client application.

9.4.2 Module Level Testing of the Server Application

Implementer: Zayd Hammoudeh

Implementing and testing the translation of Minimax from a recursive algorithm to a workflow based implementation using Google App Engine's Pipeline API by itself is a significant challenge. As such, we delayed introducing the additional complexities associated with the game of chess until we had verified the correct implementation of the Minimax algorithm.

Initially, we developed our server application to generate a ternary tree of height 3. Figure 20 shows the specialized state utility function we used to calculate the utility of a leaf node for testing purposes. By reseeding the random number generator for each leaf node and then assigning the *j*th random number for the specified seed to the *j*th leaf node, we are able to generate a predictable, verifiable, yet easily changeable (i.e. through a new seed) number sequence.

```
random.seed(88) # Reseed the random number generator
j = 3 * board[0] + board[1] # Determine the leaf node number

# Select the jth random number for the specified seed
for i in xrange(0, j+1):
    k = random.randint(0, 9) # Generate a random integer between 0 and 9 inclusive
return k
```

Figure 20 – Simplified Python Implementation of a Randomly Generated State Utility

Figure 21 shows an example tree that was created using this testing implementation. By selecting a height of three, we were able to verify that both the "Min" and "Max" components of Minimax worked as expected. What is more, we were able to verify that the Minimax algorithm selects the correct initial move (in this case the second move with utility "6" is the best move). Only once we had fully verified this implementation, we moved on to incorporating the chess specific functions, which we tested using unit testing as explained in section 9.3.

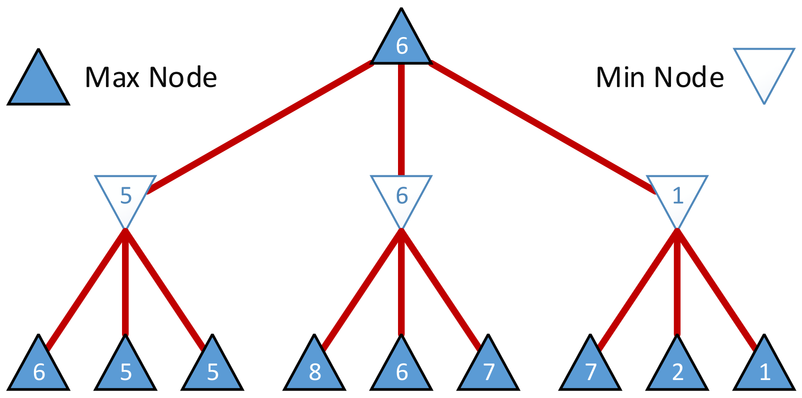


Figure 21 – Example Minimax Test Tree with a Random Seed of 88

9.5 Complete System Testing

Implementer: Zayd Hammoudeh & Geetika Garg

Once all of the tiers in our system had been tested using the previously described approaches, the amount of system level testing required was greatly reduced. However, we still performed extensive system level testing.

One of the powerful features of Google App Engine (GAE) is that Google provides tools that enable the user to simulate the GAE environment in Eclipse. This enabled us to do much of the system level debugging locally on our own machines. The client application would be running in one Eclipse instance while the GAE server would be running in a second Eclipse instance. We could then monitor, control, and debug the communication between the two application components without introducing the variable of a network. Once this implementation had been fully debugged, we then deployed our application to the cloud environment for final verification.

```
0
3,0,0,0
5,6,0,0
0,0,0,0
14,15,12,16
```

Figure 22 – Example System Test Board File

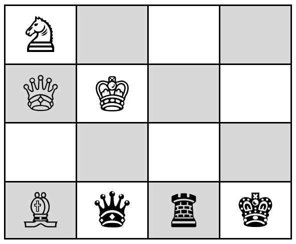


Figure 23 – Example System Test Board

Figures 22 and 23 show an example board we used when testing the system (note this is the same board file that will be used in our demo). While the white player makes the first move for this board file (as denoted by the "0" in the first line in figure 22 as described in section 8.2.1), we experimented with the human player being both black and white. When playing, we would make both good and bad moves for the human player and verify the resulting move of the computer. The biggest issue we ran into when running these tests is that often the computer would make moves that *seemed* illogical to the tester but were shown to actually be a better strategy once the game was played out. This is most likely more a testament to the limited chess skills of the tester as opposed to the masterful chess skills of our tool.

10. Potential Improvements

Many artificial intelligence algorithms, including Minimax, rely on heuristics. Superior heuristics allow the system to not only return superior solutions but to also have potentially faster performance. Currently, the algorithm uses a "one-size-fits-all" approach when it comes to Minimax tree depth and Alpha-beta pruning batch size. Using intelligent heuristics, both of these could be made more adaptive. Examples of variables the heuristics could use include:

1. **Number of Existing Server Requests** – A greater number of standing requests implies a greater burden on the server. Google App Engine (GAE) has an upper bound on the number of tasks that can be dequeued per second. If the server saw that it was lightly loaded, it could allow the search tree to proceed deeper.

2. **Expected Branching Factor** – Depending on the number, type and position of the remaining pieces, a heuristic could provide a rough estimate of the branching factor for subsequent moves. If the branching factor was small, the algorithm could choose to increase the search depth.

3. "**Playbooks**" – A playbook is a known, strategy that is guaranteed to lead to a winning outcome on specific board configurations. By placing these configurations in a persistent datastore, the algorithm could refer to them to determine an optimal solution. This approach can be superior to the caching architecture we used because the game board does not need to exactly match a previous board. Rather, the board must only have a generalized placement for the playbook to be usable.

In addition, section #7.2.3 describes the advantages of using a stateful server for this application. Given the scope of the project and the time, we were forced to default to a stateless architecture. If more time and resources were available, we would transition the stateless server to a stateful one. Once the server was made stateful, it would open up the possibility to transition the client application from a Python program that is terminal based to a web based application that is served by our GAE infrastructure. This would greatly expand the number of users who could use the program as it would require no additional software on the client-side other than a web browser, which is now standard on essentially all modern computer systems.

One final idea we explored but never implemented was the ability to automatically return a winning move once it was found. Currently, the algorithm waits for all Pipeline objects to complete as GAE does not provide an interface to gracefully terminate all related tasks in a queue. Instead, what it provides is an error handling system that allows a program to force kill all queued and running tasks up to the root and return an error message. If we were to embed a move tuple inside the error message, the program could parse the error message to extract the specified move. Such a scheme has obvious inherent risks if not diligently checked and verified. However, it is an attractive option to improve the algorithm's performance.

11. Conclusions and Final Thoughts

A massively distributed implementation of a tree-based algorithm has merit from both a theoretical and educational perspective. For example, our design's translation of a recursive tree into a DAG is something we have not seen described in previous literature. However, that fact alone does not necessarily make it the right approach for Minimax and chess.

Any workflow based management system, including Google's Pipeline API, has overhead. Tasks must be enqueued, dequeued, and managed by the task queue; input data must be sent to different processing nodes throughout the network, and the results are then sent back to assigned processing nodes. All of these steps have a cost and take time.

For any state in the chess decision tree, the time required to generate the set of valid moves is relatively small. As such, this essentially yields a type of fine grain parallelism where the benefits of massive parallelism are overshadowed by the costs associated with distributing the algorithm. Advanced techniques like caching can make our algorithm appear faster than the traditional recursive implementation. However, such techniques only serve to obscure the reality as had the same technique been applied to the recursive approach, then this revised version of the standard approach may have been faster than our implementation.

When a lighter weight version of the Pipeline API (or a similar architecture) exists, the techniques we describe in this paper would need to be revisited. Until that time however, the additional overhead introduced by computation in a distributed system can outweigh the speed-ups the same system provides.

References

[1] Pritchard, D. Brine. *The Encyclopedia of Chess Variants*. Godalming: Games & Puzzles, 1994. Print.

[2] Badhrinathan, G.; Agarwal, A.; Anand Kumar, R., "Implementation of distributed chess engine using PaaS, "*Cloud Computing Technologies, Applications and Management (ICCCTAM), 2012 International Conference on"*, vol., no., pp.38,42, 8-10 Dec. 2012 doi: 10.1109/ICCCTAM.2012.6488068

[3] Russell, Stuart J., Peter Norvig, and Ernest Davis. *Artificial intelligence : a modern approach*. Upper Saddle River, NJ: Prentice Hall, 2010. Print.

[4] Braude, Eric J. *Software Engineering : An Object-Oriented Perspective*. New York: Wiley, 2001. Print.

[5] Roebuck, Kevin. *Application Testing as a Service (TaaS): High-impact Technology – What You Need to Know: Definitions, Adoptions, Impact, Benefits, Maturity, Vendors*. Brisbane: Emereo Publishing, 2012. Print.

[6] "25.2. Doctest — Test Interactive Python Examples." 25.2. *Doctest*. Web. 29 Mar. 2015. <https://docs.python.org/2/library/doctest.html>.

[7] Allis, Louis Victor. *Searching for Solutions in Games and Artificial Intelligence*. Wageningen: Ponsen & Looijen, 1994. Print.

[8] Hoffman, Paul. *King's Gambit: A Son, a Father, and the World's Most Dangerous Game*. New York: Hyperion, 2007. Print.

[9] Seirawan, Yasser, and Jeremy Silman. *Play Winning Chess: An Introduction to the Moves, Strategies, and Philosophy of Chess from the U.S.A.'s #1-ranked Chess Player*. Redmond, WA: Tempus of Microsoft, 1990. Print.

[10] Marinescu, Dan C. *Cloud Computing Theory and Practice*. Boston: Morgan Kaufmann, 2013. Print.

[11] "20.5. Urllib — Open Arbitrary Resources by URL." 20.5. *Urllib*. Web. 29 Mar. 2015. <https://docs.python.org/2/library/urllib.html#urllib.urlencode>.

[12] "The Webapp2 Framework." *Google Developers*. Web. 29 Mar. 2015. <https://cloud.google.com/appengine/docs/python/tools/webapp2>

[13] "Google I/O 2011: Large-scale Data Analysis Using the App Engine Pipeline API." *YouTube*. Google Developers, 12 May 2011. Web. 30 Mar. 2015. <https://www.youtube.com/watch?v=Rsfy_TYA2ZY>.

[14] "Generators." Python Software Foundation. Web. 30 Mar. 2015. <https://wiki.python.org/moin/Generators>.

[15] "Memcache." *Google Developers*. Web. 20 Apr. 2015. <https://cloud.google.com/appengine/docs/adminconsole/memcache. [Accessed: 20- Apr- 2015>