ECE496 Final Report

Distributed Minimax

Project No. 307

Authors:

Steven Blenkinsop (997673888)

Jonathan Gray (998198588)

Supervisor:

Prof. Jorg Liebeherr

Administrator:

John Taglione

Submitted March 19, 2015

1 Executive Summary

This document outlines the design for and results of developing a software framework which allows the user to distribute the evaluation of a zero-sum sequential game using the minimax algorithm over an arbitrary number of computers. Minimax is an optimal algorithm developed in the late 1940s which plays a game by searching for the best possible move under the assumption that the opponent plays perfectly. Developing this framework will enable users to speed up the evaluation of minimaxable problems by leveraging existing capacity on general-purpose commodity computer systems.

The resulting framework allows a user to deploy a game-specific web service which computes the best move to make from a game position supplied by a client of the service. Clients of a web service created using the framework are also able to supply a time limit for the evaluation of the problem, which forces the service to provide a best guess if it is unable to fully evaluate the problem in the allotted time.

The design uses a master/slave architecture to distribute the web service over a cluster of computers. The master node in the cluster receives requests from the client, breaks the requested minimax computation into a set of jobs which can be executed in parallel, and allocates those jobs to slave nodes in the cluster. Once the slaves have completed their jobs, the master consolidates the results in order to determine what the best move is and returns it to the client.

At the time of writing of this document, the framework has been fully implemented and tested. The framework has been verified to meet all functional requirements and constraints, including correctly implementing the minimax algorithm, being able to distribute the computation over two to ten computers, being able to be limited by a user-supplied time limit, and running on all target operating systems. Additionally, the key objective that a cluster of computers using the framework should be able evaluate more game states in the same amount of time as a single computer was met. A second objective, that the framework should optimally use processors of different speeds was not incorporated into the design, and therefore has not been met. Because the project has met all requirements and constraints, as well as the first objective, we consider the project to be a success. Future work on this project would be directed toward redesigning the job allocation module in order to meet the second objective.

2 Group Highlights and Individual Contributions

2.1 Jonathan's Individual Contributions

Jonathan focused on the master-slave communication protocol, job pool creation and allocation algorithms, and master-slave coordination. He designed, implemented, and tested the bulk of the code for creating a job pool from a single game state, coordinating the allocation of jobs to slaves, concurrently working jobs slave-side, and aggregating the slaves' results master-side.

In addition, he performed the 2-slave and 10-slave speed testing using Amazon AWS virtual machines, familiarizing himself to a new industry-standard tool.

2.2 Steven's Individual Contributions

For the project, I was responsible for implementing the initial version of the minimax algorithm, as well as implementing game logic and game state serialization for tic-tac-toe and checkers. I was also responsible for validating the correctness of the minimax algorithm and testing the compatibility of the final implementation with Windows 8.1.

The minimax algorithm is defined in terms of a set of operations that can be implemented on any value representing a state in a game. These operations tell the implementation of the algorithm whose move it is, what the heuristic value of the current game state is, what moves are possible from this state, and how to serialize the state for transmission over the network. This implementation has been validated by running it using the tic-tac-toe game logic and generating a table of optimal moves (Appendix E). Over 10% of this table has been manually verified for correctness in holding with the relevant validation test, and no errors were found.

This implementation of the algorithm was then used to build the slave tasks and forms an integral part of their functioning, along with the code Jonathan wrote to optimize the execution of the code.

3 Table of Contents

4 Introduction

This document outlines the design, implementation, and testing of a software framework for distributing the evalutation of the minimax algorithm across a cluster of computers. The motivation and goal of the design is introduced first, followed by a set of requirements and objectives that the project should fulfill, and a description of the final design at both a system- and module-level. A set of tests was designed and executed for each requirement and objective, and the results of these tests are presented and discussed. Lastly, this document provides an overview of the conclusions drawn from this project, and suggestions for future work.

4.1 Background and Motivation

Some of the earliest software written for programmable computers sought to enable them to play abstract strategy games such as chess and checkers [1]. In the late 1940s, artificial intelligence pioneers like Claude Shannon, John von Neumann, and Alan Turing independently developed game-playing algorithms that relied on the concept of minimaxing, an optimal algorithm for game-playing which searches for the most advantageous move assuming the opponent plays perfectly [2].

The Minimax algorithm's analysis of a game state (e.g. the particular configuration of chess pieces mid-game) relies on analysis of all possible games that could be played out from this position. For games too complex for this complete analysis (chess, for example, is estimated to have more possible games than there are atoms in the universe[3]), the game is analyzed to a limited depth, and a heuristic is used to estimate which player is winning at a given state. If more game states are analyzed, the heuristic evaluation will be based on more information, and the resulting analysis is more likely to be accurate.

The purpose of this project is to build software that facilitates the coordination of multiple computers to work together to run the Minimax algorithm, in order to analyze more game states per unit time than would otherwise be possible with just a single computer.

Specifically, this project aims to produce a software framework for distributing the execution of the minimax algorithm over an arbitrarily-sized cluster of computers. The framework allows an end-user to describe the game tree and provide heuristics for evaluating the state of a game that has not yet terminated, while abstracting away the mechanics of the distributed nature of the computation. It also provides a simple interface to add, remove, and replace computers participating in the minimax evaluation.

The primary motivation for this project was the group members interest in game artificial intelligence and distributed systems.

4.2 Project Goal

The goal of this project was to create a software framework with which a user can distribute the evaluation of a user-defined perfect-information zero-sum sequential game using the minimax algorithm over an arbitrary number of computers.

5 Project Requirements

5.1 Functional Requirements

The design shall:

1. correctly implement the minimax algorithm for user-defined perfect-information zero-sum sequential games
2. allow for distributing evaluation over a number of nodes between 2 and 10
3. allow the user to set a time limit (at least 1 second) on the computation, and enforce this time limit within 0.5 seconds

5.2 Constraints

The resulting software must be executable on all of:

1. Mac OS 10.9 or above
2. Windows 8.1 or above
3. Linux 3.14 or above

5.3 Objectives

The resulting software should:

1. evaluate more game states per unit of time compared to an equivalent implementation of the minimax algorithm running on single computer

2. efficiently utilize processors of different speeds

6 Final Design

Section 6.1 provides an overview of the final design and the interactions between the various modules.

Following that, Section 6.2 provides a more detailed description about the major design components and processes described in the system overview.

6.1 System-Level Overview

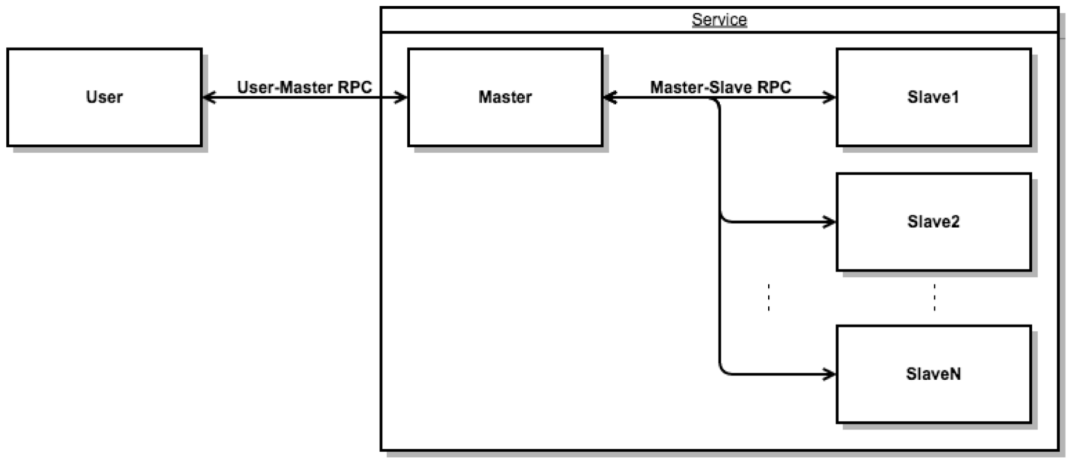


Figure 1: Component diagram for the distributed minimax service showing the separation of *user*, *master*, and *slave* processes

A single *master* process and a number of identical *slave* processes communicate with each other to provide a network *service* to a *user* process. As illustrated in Figure 1, the *user* and the *slaves* communicate only with the *master* (over the *User-Master RPC* and *Master-Slave RPC*, respectively) and not with each other.

On receiving a request from the *user* over the *User-Master RPC*, the *master* creates a *job pool* by breaking down the Minimax algorithm into discrete jobs that can be performed concurrently, and distributes these jobs to the slaves over the *Master-Slave RPC*. The *slaves* execute their jobs and return the results of their analyses to the *master*. The *master* consolidates these results into a single *best move* suggestion and returns this to the *user*.

6.2 Module-Level Descriptions

This section gives a formal description of the major modules referenced in the previous section. Each module's inputs and outputs are listed along with a brief description of the module's purpose. The final design of each module is also described, and areas for potential improvement are noted in some cases.

6.2.1 User-Master RPC

Module Inputs:

- Serialized game state
- Time limit

Module Outputs:

- Suggestion for best move

Module Description:
This module represents network interaction between the *user* and the *master*.

The user makes a request to the master with a serialized game state and a maximum time allowed. The Master node is expected to confer with the slaves and respond within the time limit with its suggestion for a move to be played.

Final Design:
The open-source *Protobuf* library[4] was used to describe the message formats and generate RPC code. For an exact specification of the RPC, see Section 12.1 in Appendix C.

6.2.2 Job Pool Creation

Module Inputs:

- Game state to be analyzed
- A set of slaves that are available to do work

Module Outputs:

- A set of jobs that can be worked independently and concurrently

Module Description:

This module is a component of the *master*.

Given a game state, this module generates a set of jobs that can be worked independently. The number of jobs created will depend on the number of slaves available to do work. In particular, it should produce at least as many jobs as there are slaves, so no slave is left with no work to do.

Final Design:

A *job* in the final design is a game state that is a descendent[1] of the game state to be analyzed. This module will examine the direct descendents of the viven game state, then each of their direct descendents, etc., until the number of descendents is greater than the number of slaves. When that condition is met, these game states are returned as jobs.

6.2.3 Job Allocation

Module Inputs:

- A set of jobs that can be worked independently and concurrently
- A set of *slaves* that are available to do work

Module Outputs:

- Exactly one *slave* assigned to each job

Module Description:

This module is a component of the *master*.

Given a set of jobs and a set of *slaves*, this module decides which *slaves* will work on which jobs.

[1]The ”children” of a game state are the game states that can be reached by making a single move. For example, an empty tic-tac-toe board has nine children, representing a piece played in each of the nine empty spots. The ”descendents” of a game state are its children, its children’s children, etc.

Final Design:

The final design assigns jobs to slaves in a round-robin fashion. For example, if there are 10 jobs and 7 slaves, then the first three slaves will be assigned two jobs each, and the other four slaves will be assigned only one job. This method ensures that the *number* of jobs assigned to each slave is fairly even, but is agnostic about *which* jobs are assigned to which slave.

This is a major area for improvement. Given more time, this module should attempt to measure the relative importance of jobs and the effectiveness of slaves at performing work, and assign more important jobs to more effective slaves.

6.2.4 Master-Slave RPC

Module Inputs:

- A set of jobs
- Time Limit

Module Outputs:

- Result of analysis for each job

Module Description:

This module represents network interaction between the *master* and the *slaves*.

The master delivers to each slave a set of jobs to be performed and a maximum time allowed. The slaves return their results to the Master within the allowed time.

This module deals primarily with serialization and communication with the master node, while the *Slave Work* module below deals primarily with performing analysis.

Final Design:

The open-source *Protobuf* library[4] was used to describe the message formats and generate RPC code. For an exact specification of the RPC, see Section 12.1 in Appendix C.

6.2.5 Slave Work

Module Inputs:

- A set of jobs
- Time Limit

Module Outputs:

- Result of analysis for each job

Module Description:

This module is a component of the *slaves*.

This module performs executes the jobs that were allocated by the master. This involves running the Minimax algorithm on subsets of the game tree. The results of the jobs are returned to the *Master-Slave* RPC module to be returned to the master.

Final Design:

This module receives a set of game states to analyze and a time limit, and produces a heuristic evaluation of each game state. For each game state, concurrently, it will run an iterative-deepening Minimax[5] algorithm[2], analyzing to increasing depths until time runs out. When the time limit has elapsed, the results of the last complete iteration for each game state are returned.

6.2.6 Result Consolidation

Module Inputs:

- Results of analysis for each job

Module Outputs:

- Suggestion for best move

Module Description:

This module is a component of the *master*.

This module combines the results of the analyses that the slaves performed to produce a single best move suggestion to return to the user.

[2]The Minimax algorithm is run first with a cut-off depth of 1, then 2, then 3, etc. until time runs out.

Final Design:

Given a set of game states that are descendents of the "parent" game state for which the user requested analysis, this module runs the Minimax algorithm rooted at the parent, using the slaves' heuristic evaluations. The output of this Minimax evaluation is returned to the user as the suggested best move.

7 Testing and Verification

This section introduces a test for each requirement, constraint, and objective introduced in Section 5. The results of each test are included along with a short discussion concluding whether or not each requirement was met.

7.1 Requirement: *shall correctly implement the minimax algorithm*

7.1.1 Test: Manually Validate Tic-Tac-Toe

Test description: Generate a table of optimal moves for each possible starting tac-tac-toe position using the minimax implementation. Manually verify that the generated moves are optimal.

Pass criteria: Each validated entry in the table is correct. At least 10% of the table must be validated.

Test result: PASS

Tic-tac-toe has 628 possible start positions (i.e. positions where neither player has won nor have all the squares been filled). Moves for well over 63 (10%) of these were manually validated. A table of validated moves is included in Appendix E.

7.2 Requirement: *allow for distribution over a number of nodes between 2 and 10*

7.2.1 Test: Two-Node Cluster

Test description: Analyze a Connect-4 game state using a cluster of 2 slaves

Pass Criteria: The analysis completes without error, and the result of the analysis is an optimal move.

Test Result: PASS

Included here is the raw log output from the user process (Figure 2) and the master process (Figure 3) when this test was run.

```
2015/03/15 23:25:48 asking about state:

| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | |O| | |
| | | |O|X| | |
| | |O|X|X|X| |
2015/03/15 23:25:48 making request, time limit 3000 millis
2015/03/15 23:25:52 rpc finished in 3.180523374s
2015/03/15 23:25:52 recommended move:

| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | |O| | |
| | | |O|X| | |
| | |O|X|X|X|O|
```

Figure 2: The raw output of the user process with a 2-node slave cluster

```
 1  2015/03/16 03:25:35 Slave service listening on port 14782
 2  2015/03/16 03:25:35 User service listening on port 14783
 3  2015/03/16 03:25:39 slave connected with 0 past results
 4  2015/03/16 03:25:41 slave connected with 0 past results
 5  2015/03/16 03:25:49 DoWork called
 6  2015/03/16 03:25:49 expanding rootState for 2 slaves
 7  2015/03/16 03:25:49 created 7 jobs from root state
 8  2015/03/16 03:25:49 distributing 7 jobs to 2 slaves
 9  2015/03/16 03:25:52 slave connected with 3 past results
10  2015/03/16 03:25:52 AddResult {X:1928440315904 Y:208339468288 IsXMove:true}
```

Figure 3: The first 10 lines of raw output of the master process with a 2-slave cluster. The complete log is too long to list here.

We can verify that this is a 2-slave cluster in line 8 of Figure 3. We can further verify that the analsyis completed without error and that the result of the analysis (at the bottom of Figure 2 was an optimal move[3]. Since the analysis completed successfully and without

[3]The rules of Connect-4 are not included in this document, but are widely available. In this case, the recommended move is optimal because any other move would lead to an immediate win by X playing in the same spot.

error for a cluster of 2 slaves, this test is considered to have passed.

7.2.2 Test: Ten-Node Cluster

Test description: Analyze a Connect-4 game state using a cluster of 10 slaves

Pass Criteria: The analysis completes without error, and the result of the analysis is an optimal move.

Test Result: PASS

A cluster of 10 slaves was used to analyze the same Connect-4 state, and the same (optimal) result was returned. Relevant lines of the master process's log output are shown in Figure 4.

```
2015/03/16 02:55:22 DoWork called
2015/03/16 02:55:22 expanding rootState for 10 slaves
2015/03/16 02:55:22 created 13 jobs from root state
2015/03/16 02:55:22 distributing 13 jobs to 10 slaves
```

Figure 4: An excerpt of the raw output of the master process with a 10-slave cluster. The complete log is too long to list here.

It can be seen in this log output that 10 slaves are used. Since the analysis completed successfully and without error using a cluster of 10 slaves, this test is considered to have passed.

7.3 Requirement: *allow the user to set a time limit on the computation; enforce within 0.5 seconds*

7.3.1 Test: Time limit of 3 seconds

Test description: Analyze a Connect-4 game state using an imposed time limit of 3 seconds.

Pass Criteria: The result is returned in at most 3.5 seconds.

Test Result: PASS

Figure 2 shows the output of the user process when this test was run. Line 9 shows that the request was made with a limit of 3 seconds, and line 10 shows that the response was returned in 3.18 seconds. Since the time limit was enforced within 0.5 seconds, this test is considered to have passed.

7.3.2 Test: Time limit of 10 seconds

Test description: Analyze a Connect-4 game state using an imposed time limit of 10 seconds.

Pass Criteria: The result is returned in at most 10.5 seconds.

Test Result: PASS

Figure 5 shows the output of the user process when this test was run. Line 9 shows that the request was made with a limit of 10 seconds, and line 10 shows that the response was returned in 10.16 seconds. Since the time limit was enforced within 0.5 seconds, this test is considered to have passed.

```
 1  2015/03/15 22:51:42 asking about state:
 2
 3  | | | | | | | |
 4  | | | | | | | |
 5  | | | | | | | |
 6  | | | | |O| | |
 7  | | | |O|X| | |
 8  | | |O|X|X|X| |
 9  2015/03/15 22:51:42 making request, time limit 10000 millis
10  2015/03/15 22:51:52 rpc finished in 10.160189283s
11  2015/03/15 22:51:52 recommended move:
12
13  | | | | | | | |
14  | | | | | | | |
15  | | | | | | | |
16  | | | | |O| | |
17  | | | |O|X| | |
18  | | |O|X|X|X|O|
```

Figure 5: The output of the user process with a specified time limit of 10 seconds.

7.4 Constraint: *must be executable on Mac OS 10.9, Windows 8.1, and Linux 3.14*

7.4.1 Test: Windows 8.1

Test Description: Analyze a game state with all master and slave processes running on Windows 8.1.

Pass Criteria: The analysis completes without error.

Test Result: PASS

Figure 11 in Appendix D is a screenshot of a master, a user, and two slave processes running successfully in a Windows 8.1 environment.

7.4.2 Test: Mac OS 10.9

Test Description: Analyze a game state with all master and slave processes running on Mac OS 9.

Pass Criteria: The analysis completes without error.

Test Result: PASS

Figure 12 in Appendix D is a screenshot of a master, a user, and two slave processes running successfully in a Mac OS 10.9 environment.

7.4.3 Test: Linux 3.14

Test Description: Analyze a game state with all master and slave processes running on Linux 3.14.

Pass Criteria: The analysis completes without error.

Test Result: PASS

Figure 13 in Appendix D is a screenshot of a master, a user, and a slave process running successfully on an Amazon Linux AMI[6] using kernel version 3.14.

7.5 Objective: *evaluate more game states than the non-distributed implementation*

7.5.1 Test: Distributed vs. Non-Distributed

Test description: Analyze the same Connect-4 state using (1) a cluster of 10 slaves, and (2) a single computer. Ensure that the single computer is given the same processing time as the cluster took for communication, coordination, and processing combined. Measure the number of game states analyzed during the computations.

Pass Criteria: The cluster of 10 slaves analyzes more game states than the single computer.

Test Result: PASS

The number of states analyzed was measured for four analyses: with 10 slaves, given 3 seconds and given 10 seconds, and with 1 slave given the same time constraints. The results are shown in Figure 6.

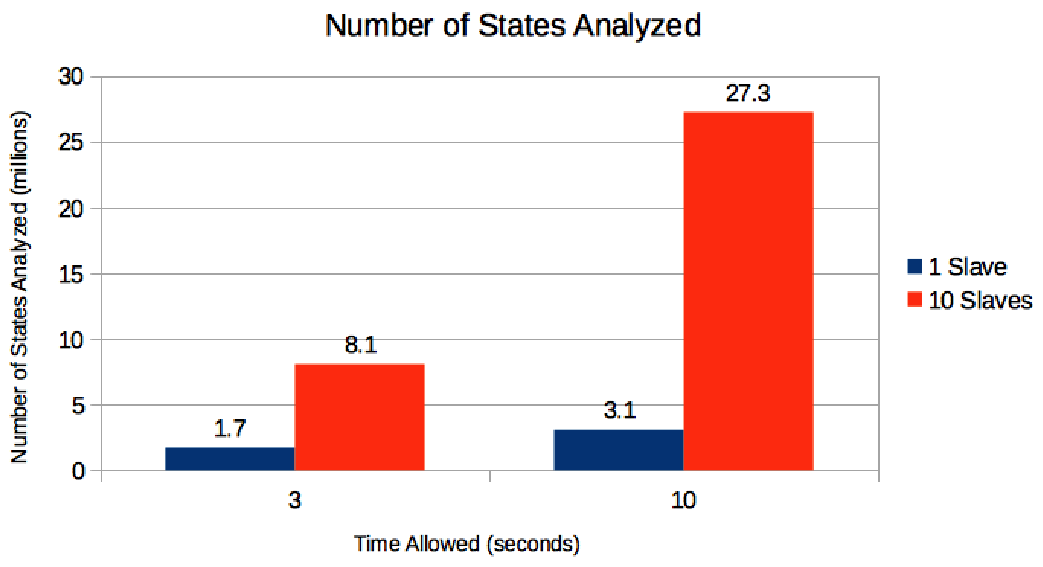


Figure 6: The number of states analyzed by two slave clusters (one with a single computer, the other with 10 slaves) given 3 seconds and 10 seconds

Note that for both time limits, the cluster of 10 slaves analyzed more states than the single computer. For this reason, this test is considered to have passed, and the objective met.

7.6 Objective: *should maximize throughput on different combinations of fast and slow processors*

7.6.1 Test: Analyze different combinations of fast and slow CPUs

Test Description: See Section 3.6 of Appendix B for a description of the originally-proposed test.

Test Result: FAIL

The work required for this test to pass was not completed, and the test was abandoned and assumed to fail. In particular, as mentioned in the "Final Design" paragraph of Section 6.2.3, the master's job allocation algorithm is only a simple round-robin assignment, agnostic to the processing capabilities of the different slaves. Given more time, this is a potential area for improvement of this project.

8 Summary and Conclusions

The overarching purpose of this project was to provide a method to speed up the Minimax algorithm by making use of all available computing power. A user with many computers at his disposal should easily be able to coordinate them to work together on the Minimax problem, and provide better results than he would otherwise achieve with only a single computer.

The first functional requirement presented in Section 5 ensures that the Minimax algorithm is implemented correctly, without which any improvements to execution speed would be worthless. This requirement was met, as demonstrated in Section 7.1.

The second requirement ensures that the number of computers is variable, i.e. that the system can will work successfully with different cluster sizes[4]. This requirement was also met by demonstrating that the same evaluation worked with a cluster of 2 slaves and a cluster of 10 slaves.

[4]Though the original project proposal required that the system work with up to 50 computers, this was reduced to 10 to make testing cheaper and easier. This change is not evidence that the system would *not* work with 50 computers (in fact, due to the encouraging results seen in Section 7.5.1, we predict that the system would work quite well with clusters of this size, and would like to test this prediction in the future).

The third requirement ensures that the user can vary the amount of time available to do work. The framework produced for this project is written in a general, extensible way that would easily allow developers to run the Minimax algorithm on a wide variety of game-theoretic scenarios. Considering the range of possible applications, this requirement was added to ensure that the resulting framework is not overly restrictive in its scope. For similar reasons, the sole constraint in Section 5.2 (that the software should be executable on a range of operating systems) was added as well. Both requirements were met.

However, as stated at the beginning of this section, the utility of this project is tied to its ability to provide *better* results on a cluster of computers than it could otherwise achieve. If the overhead of network communication and coordination inherent in a distributed system makes cluster perform worse than a single computer, then this project – while meeting requirements – is not useful. This primary objective is laid out in Section 5.3 and tested in Section 7.5.1. The results, summarized in Figure 6 in that section, demonstrate that this objetive was met – for two different time limits, the cluster of 10 slaves analyzed significantly more game states than a single computer running the Minimax algorithm.

Since the requirements and constraints were met, and this primary "usability" objective was achieved, we consider this project a success.

A second objective was included Section 5.3 addressing users who might have heterogeneous clusters. Ideally, the software should recognize differences in processing power between different computers in the cluster, and assign more work to more powerful computers. Work to this end was not completed by the time this report was writte, and so this objective is left unmet. A complete re-design of the job allocation module (Section 6.2.3) in pursuit of this objective would likely be the starting point for future work on this project.

9 References

[1] Russell, S., Norvig, P. & Davis, E. (2010). Artificial intelligence: a modern approach. Upper Saddle River, NJ: Prentice Hall. (p. 122)

[2] Russell, S., Norvig, P. & Davis, E. (2010). Artificial intelligence: a modern approach. Upper Saddle River, NJ: Prentice Hall. (p. 142)

[3] Steinerberger, S. (2014). On the number of positions in chess without promotion. International Journal of Game Theory.

[4] Google (2008). Protocol Buffers - Google's data interchange format.
`https://github.com/google/protobuf`

[5] Chess Programming Wiki (2015). Iterative Deepening.
`https://chessprogramming.wikispaces.com/Iterative+Deepening`

[6] Amazon (2015). Amazon Linux AMI.
`http://aws.amazon.com/amazon-linux-ami/`

10 Appendix A: Gantt Chart History

This appendix includes the Gantt charts from the Project Proposal, the Test Document, and the Progress Report, as well as an updated Gantt chart which includes the work completed up to the day this report was submitted.

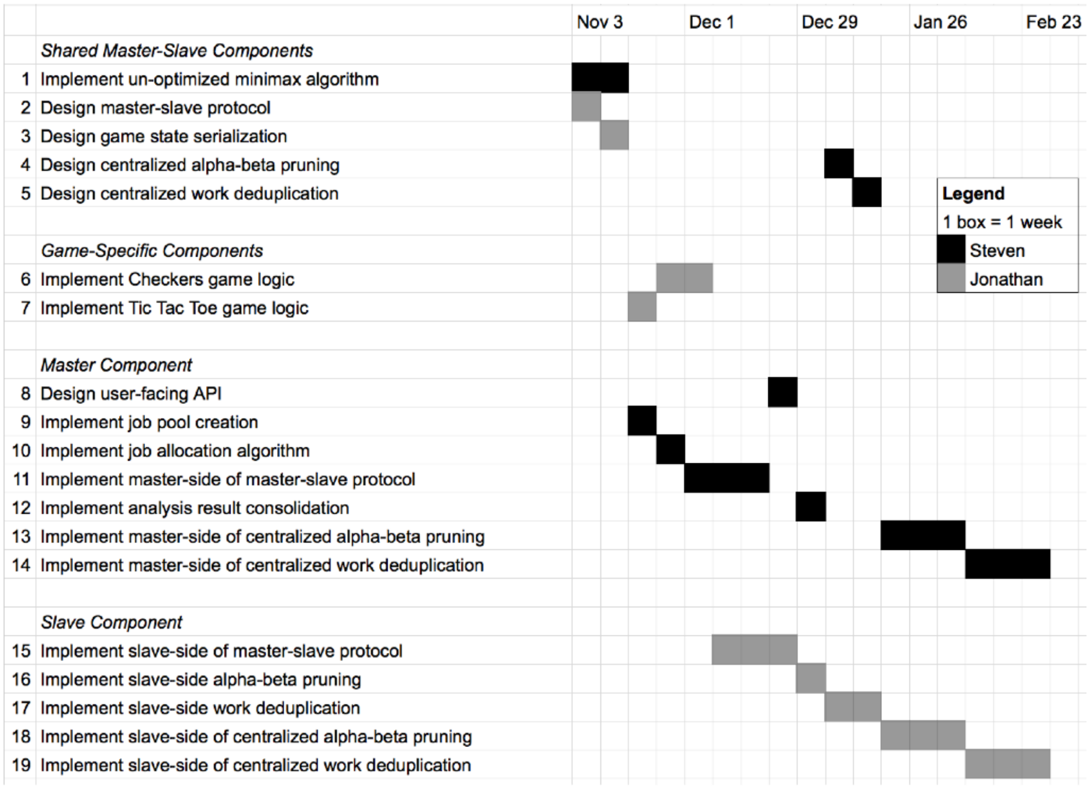


Figure 7: The initial Gantt chart, as presented in the Project Proposal.

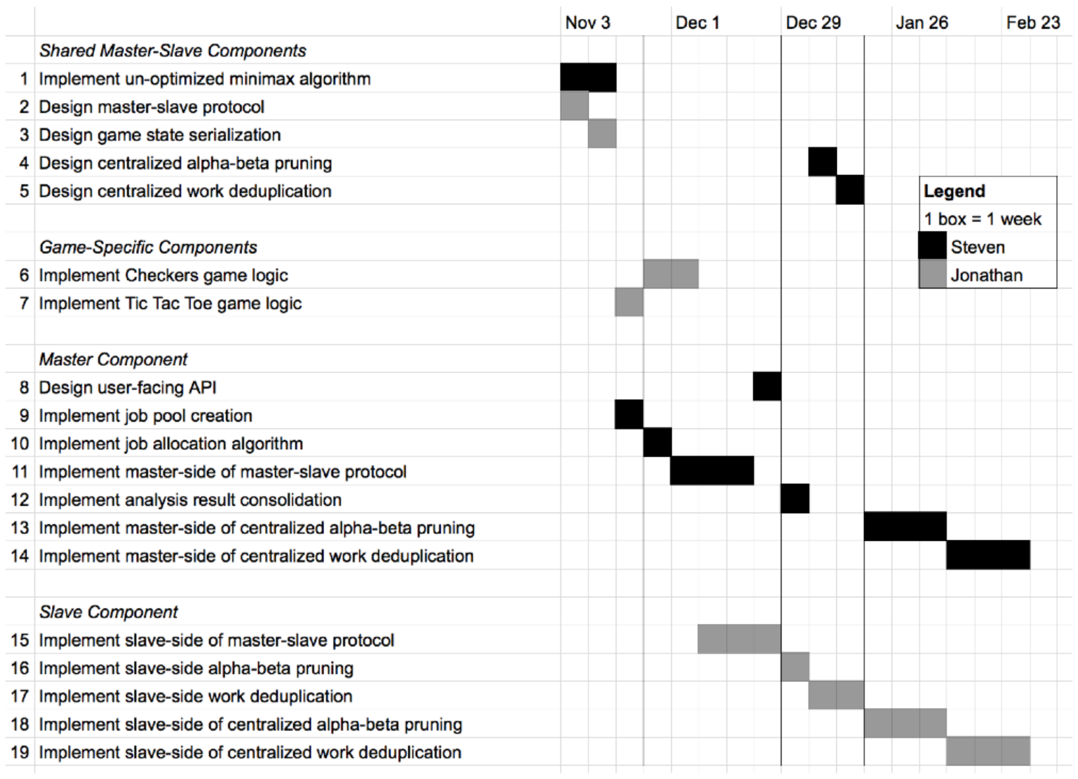


Figure 8: The Gantt chart from the Test Document

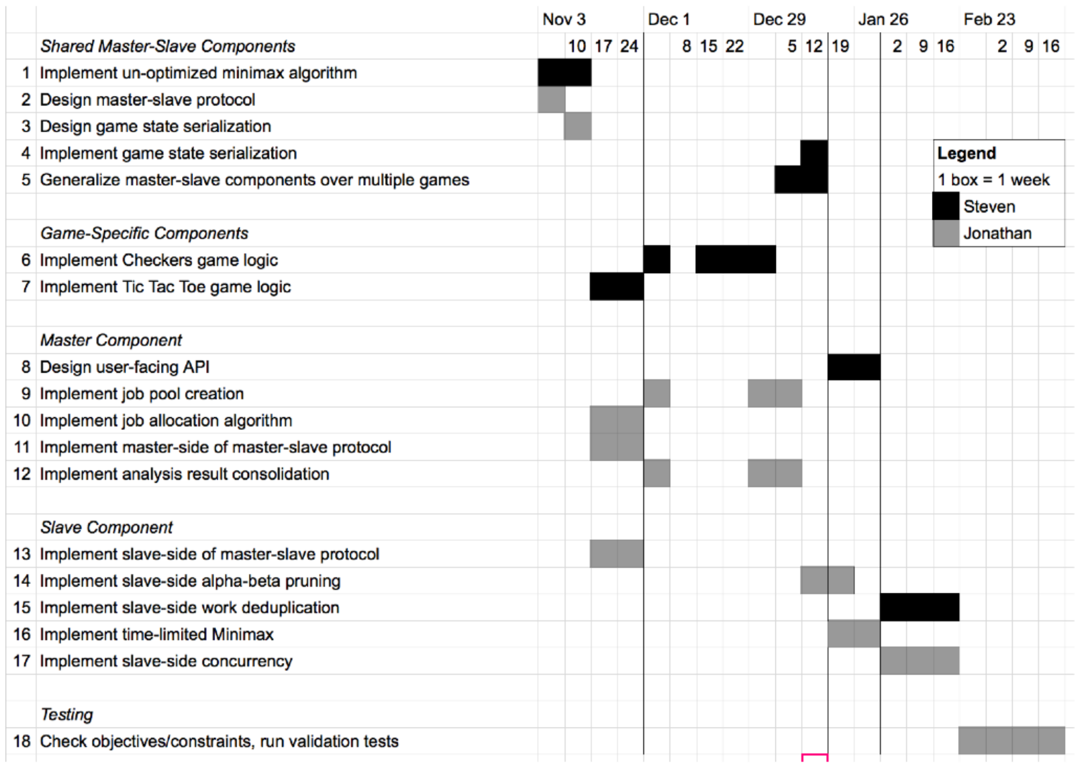


Figure 9: The Gantt chart from the Progress Report

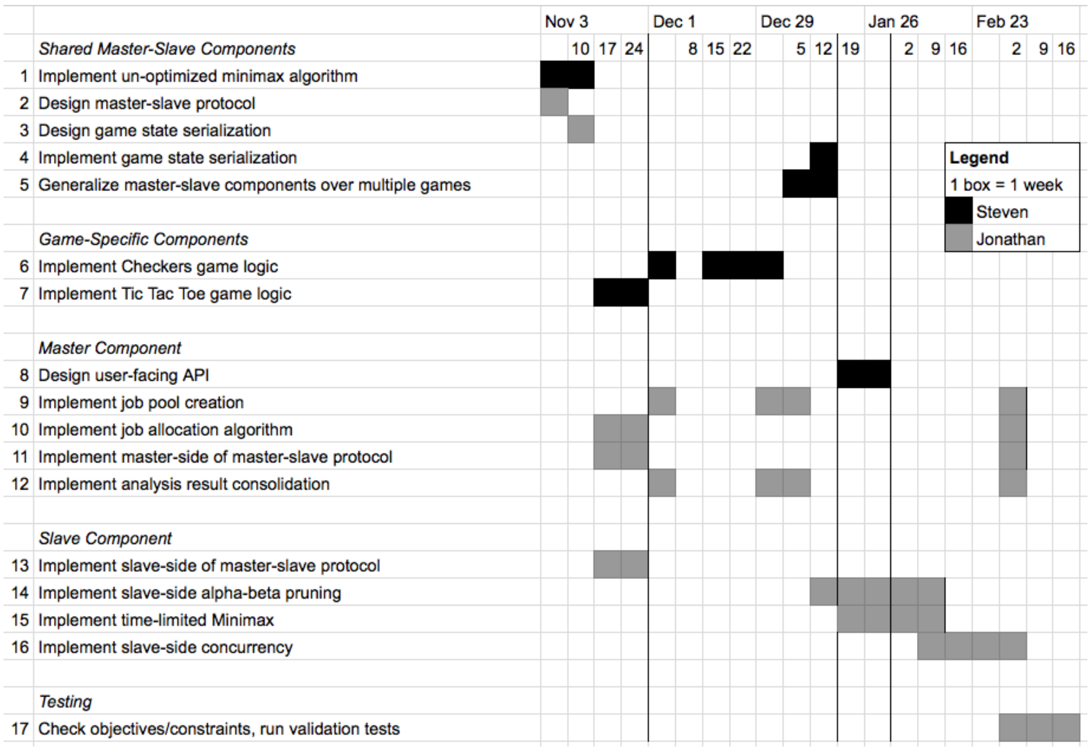
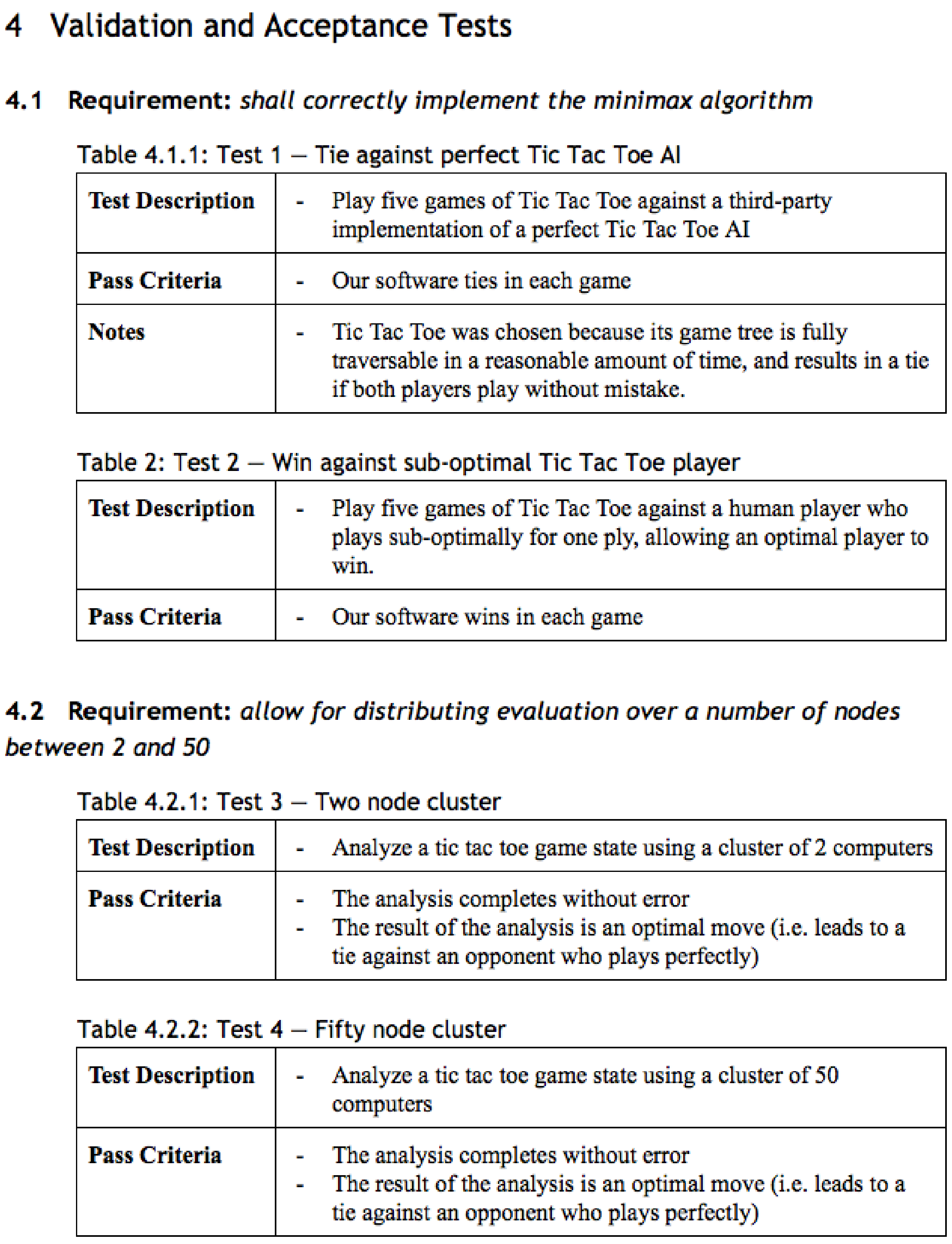


Figure 10: The final Gantt chart, representative of work done to this day.

11 Appendix B: Original Validation and Acceptance Tests



4 Validation and Acceptance Tests

4.1 Requirement: *shall correctly implement the minimax algorithm*

Table 4.1.1: Test 1 – Tie against perfect Tic Tac Toe AI

Test Description	- Play five games of Tic Tac Toe against a third-party implementation of a perfect Tic Tac Toe AI
Pass Criteria	- Our software ties in each game
Notes	- Tic Tac Toe was chosen because its game tree is fully traversable in a reasonable amount of time, and results in a tie if both players play without mistake.

Table 2: Test 2 – Win against sub-optimal Tic Tac Toe player

Test Description	- Play five games of Tic Tac Toe against a human player who plays sub-optimally for one ply, allowing an optimal player to win.
Pass Criteria	- Our software wins in each game

4.2 Requirement: *allow for distributing evaluation over a number of nodes between 2 and 50*

Table 4.2.1: Test 3 – Two node cluster

Test Description	- Analyze a tic tac toe game state using a cluster of 2 computers
Pass Criteria	- The analysis completes without error - The result of the analysis is an optimal move (i.e. leads to a tie against an opponent who plays perfectly)

Table 4.2.2: Test 4 – Fifty node cluster

Test Description	- Analyze a tic tac toe game state using a cluster of 50 computers
Pass Criteria	- The analysis completes without error - The result of the analysis is an optimal move (i.e. leads to a tie against an opponent who plays perfectly)

4.3 Requirement: *allow the user to set a time limit (at least 1 second) on the computation*

Table 4.3.1: Test 5 – Time limit of 1 second

Test Description	- Analyze a checkers game state using an imposed time limit of 1 second
Pass Criteria	- The analysis completes without error - The result is returned in at most 1 second

Table 4.3.2: Test 6 – Time limit of 30 seconds

Test Description	- Analyze a checkers game state using an imposed time limit of 30 seconds
Pass Criteria	- The analysis completes without error - The result is returned in at most 30 seconds

4.4 Constraint: *must be executable on OS 10.9, Windows 8.1, and Ubuntu 14.04*

Table 4.4.1: Test 7 – OS Compatibility

Test Description	- Analyze a checkers game state using a cluster of 3 computers, running each of the three operating systems listed in the constraint
Pass Criteria	- The analysis completes without error

4.5 Objective: *should evaluate more game tree nodes than the non-distributed implementation*

Table 4.5.1: Test 8 — Better throughput

Test Description	- Analyze five states in a checkers game using a cluster of 10 computers - Analyze the same five states using an equivalent non-distributed version of the algorithm running on a single computer - In all analyses, measure the number of game tree nodes analyzed
Pass Criteria	- The distributed version analyzes more game tree nodes than the single-computer version in all five tests

4.6 Objective: *should maximize throughput on different combinations of fast and slow processors*

Note: *This test presents a criterion for the optimization of an objective, rather than a binary test.*

Table 4.6.1: Test 9 — Efficiency with fast and slow processors

Test Description	- Analyze the same five states of a checkers game with four different clusters: 1. 8 CPUs of equal speed 2. 4 fast CPUs and 4 slow CPUs 3. 6 fast CPUs and 2 slow CPUs 4. 2 fast CPUs and 6 slow CPUs - In all analyses, measure the number of game tree nodes analyzed
Optimization Target	- Let N be the average number of game tree nodes analyzed across each of the five states for a given cluster. (e.g. let N_1 be the average number of game tree nodes analyzed across all five states when all CPUs are equal). Let $M = (N_1 + N_2 + N_3 + N_4) / 4$ - Maximize the measured value of M

12 Appendix C: Code Samples

This appendix includes raw samples of code from the project to serve as further information for interested developers, and as proof of work for ECE496 administrators.

12.1 Protobuf Specifications

User-Master RPC

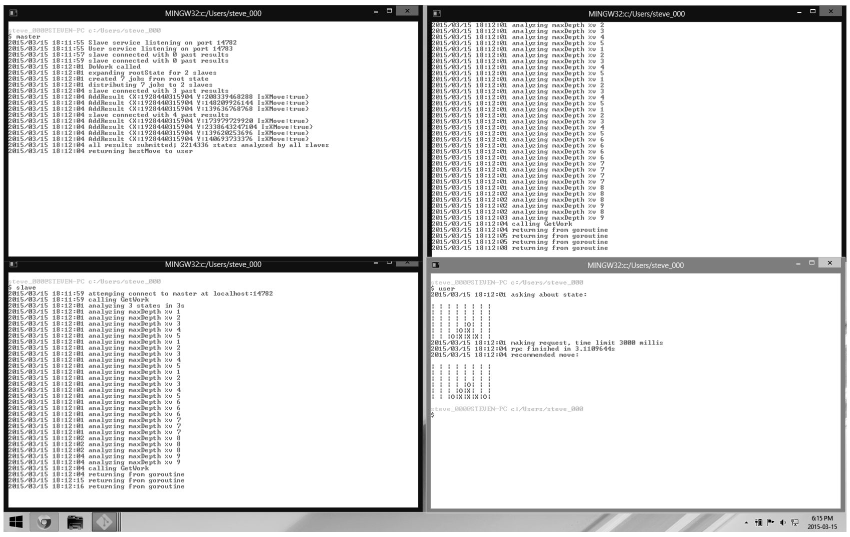
```
message DoWorkRequest {
    required bytes state = 1;
    required uint64 timeLimitMillis = 2;
}

message DoWorkResponse {
    required bytes move = 1;  // suggestion for next state
}

service UserService {
    rpc DoWork(DoWorkRequest) returns (DoWorkResponse);
}
```

Master-Slave RPC

```
message GetWorkRequest {
    message Result {
        required bytes state = 1;
        required int32 value = 2;  // should match Value type in game/game.go
        required int64 numStatesAnalyzed = 3;
    }
    // optionally return the results of a previous workload
    repeated Result result = 1;
}

message GetWorkResponse {
    repeated bytes state = 1;
    required uint64 timeLimitMillis = 2;
}

service SlaveService {
    rpc GetWork(GetWorkRequest) returns (GetWorkResponse);
}
```

13 Appendix D: Screenshots

This appendix includes screenshots taken during testing to serve as proof of work.



MINGW32:c:/Users/steve_000

```
steve_000@STEVEN-PC c:/Users/steve_000
$ master
2015/03/15 18:11:55 Slave service listening on port 14782
2015/03/15 18:11:55 User service listening on port 14783
2015/03/15 18:11:57 slave connected with 0 past results
2015/03/15 18:11:59 slave connected with 0 past results
2015/03/15 18:12:01 DoWork called
2015/03/15 18:12:01 expanding rootState for 2 slaves
2015/03/15 18:12:01 created 7 jobs from root state
2015/03/15 18:12:01 distributing 7 jobs to 2 slaves
2015/03/15 18:12:04 slave connected with 3 past results
2015/03/15 18:12:04 AddResult {X:1928440315904 Y:208339468288 IsXMove:true}
2015/03/15 18:12:04 AddResult {X:1928440315904 Y:148209926144 IsXMove:true}
2015/03/15 18:12:04 AddResult {X:1928440315904 Y:139636768768 IsXMove:true}
2015/03/15 18:12:04 slave connected with 4 past results
2015/03/15 18:12:04 AddResult {X:1928440315904 Y:173979729920 IsXMove:true}
2015/03/15 18:12:04 AddResult {X:1928440315904 Y:2338643247104 IsXMove:true}
2015/03/15 18:12:04 AddResult {X:1928440315904 Y:139620253696 IsXMove:true}
2015/03/15 18:12:04 AddResult {X:1928440315904 Y:140693733376 IsXMove:true}
2015/03/15 18:12:04 all results submitted; 2214336 states analyzed by all slaves
2015/03/15 18:12:04 returning bestMove to user
```

MINGW32:c:/Users/steve_000

```
2015/03/15 18:12:01 analyzing maxDepth %v 2
2015/03/15 18:12:01 analyzing maxDepth %v 3
2015/03/15 18:12:01 analyzing maxDepth %v 4
2015/03/15 18:12:01 analyzing maxDepth %v 5
2015/03/15 18:12:01 analyzing maxDepth %v 1
2015/03/15 18:12:01 analyzing maxDepth %v 2
2015/03/15 18:12:01 analyzing maxDepth %v 3
2015/03/15 18:12:01 analyzing maxDepth %v 4
2015/03/15 18:12:01 analyzing maxDepth %v 5
2015/03/15 18:12:01 analyzing maxDepth %v 1
2015/03/15 18:12:01 analyzing maxDepth %v 2
2015/03/15 18:12:01 analyzing maxDepth %v 3
2015/03/15 18:12:01 analyzing maxDepth %v 4
2015/03/15 18:12:01 analyzing maxDepth %v 5
2015/03/15 18:12:01 analyzing maxDepth %v 1
2015/03/15 18:12:01 analyzing maxDepth %v 2
2015/03/15 18:12:01 analyzing maxDepth %v 3
2015/03/15 18:12:01 analyzing maxDepth %v 4
2015/03/15 18:12:01 analyzing maxDepth %v 5
2015/03/15 18:12:01 analyzing maxDepth %v 6
2015/03/15 18:12:01 analyzing maxDepth %v 6
2015/03/15 18:12:01 analyzing maxDepth %v 6
2015/03/15 18:12:01 analyzing maxDepth %v 6
2015/03/15 18:12:01 analyzing maxDepth %v 7
2015/03/15 18:12:01 analyzing maxDepth %v 7
2015/03/15 18:12:01 analyzing maxDepth %v 7
2015/03/15 18:12:01 analyzing maxDepth %v 7
2015/03/15 18:12:01 analyzing maxDepth %v 8
2015/03/15 18:12:02 analyzing maxDepth %v 8
2015/03/15 18:12:02 analyzing maxDepth %v 8
2015/03/15 18:12:02 analyzing maxDepth %v 9
2015/03/15 18:12:02 analyzing maxDepth %v 8
2015/03/15 18:12:03 analyzing maxDepth %v 9
2015/03/15 18:12:04 calling GetWork
2015/03/15 18:12:04 returning from goroutine
2015/03/15 18:12:05 returning from goroutine
2015/03/15 18:12:05 returning from goroutine
2015/03/15 18:12:08 returning from goroutine
```

MINGW32:c:/Users/steve_000

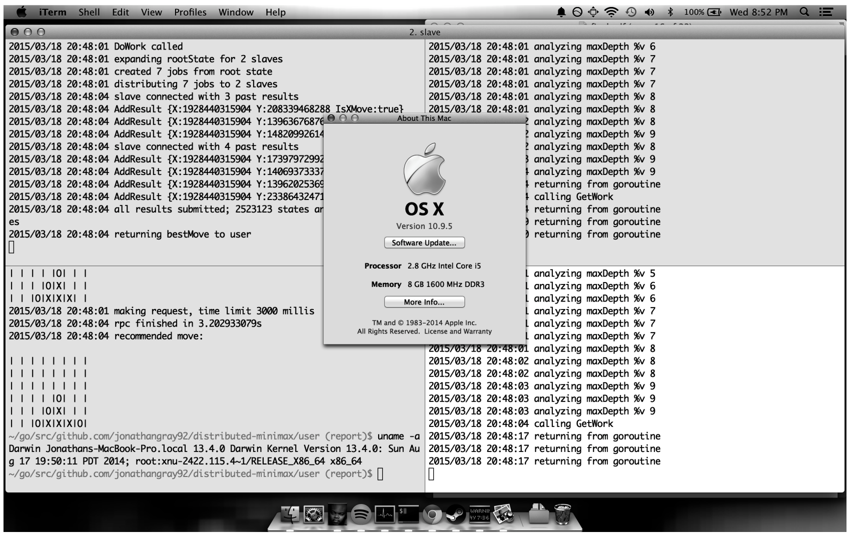
```
steve_000@STEVEN-PC c:/Users/steve_000
$ slave
2015/03/15 18:11:59 attemping connect to master at localhost:14782
2015/03/15 18:11:59 calling GetWork
2015/03/15 18:12:01 analyzing 3 states in 3s
2015/03/15 18:12:01 analyzing maxDepth %v 1
2015/03/15 18:12:01 analyzing maxDepth %v 2
2015/03/15 18:12:01 analyzing maxDepth %v 3
2015/03/15 18:12:01 analyzing maxDepth %v 4
2015/03/15 18:12:01 analyzing maxDepth %v 5
2015/03/15 18:12:01 analyzing maxDepth %v 1
2015/03/15 18:12:01 analyzing maxDepth %v 2
2015/03/15 18:12:01 analyzing maxDepth %v 3
2015/03/15 18:12:01 analyzing maxDepth %v 4
2015/03/15 18:12:01 analyzing maxDepth %v 5
2015/03/15 18:12:01 analyzing maxDepth %v 1
2015/03/15 18:12:01 analyzing maxDepth %v 2
2015/03/15 18:12:01 analyzing maxDepth %v 3
2015/03/15 18:12:01 analyzing maxDepth %v 4
2015/03/15 18:12:01 analyzing maxDepth %v 5
2015/03/15 18:12:01 analyzing maxDepth %v 6
2015/03/15 18:12:01 analyzing maxDepth %v 6
2015/03/15 18:12:01 analyzing maxDepth %v 6
2015/03/15 18:12:01 analyzing maxDepth %v 7
2015/03/15 18:12:01 analyzing maxDepth %v 7
2015/03/15 18:12:01 analyzing maxDepth %v 7
2015/03/15 18:12:02 analyzing maxDepth %v 8
2015/03/15 18:12:02 analyzing maxDepth %v 8
2015/03/15 18:12:02 analyzing maxDepth %v 8
2015/03/15 18:12:04 analyzing maxDepth %v 9
2015/03/15 18:12:04 analyzing maxDepth %v 9
2015/03/15 18:12:04 calling GetWork
2015/03/15 18:12:04 returning from goroutine
2015/03/15 18:12:15 returning from goroutine
2015/03/15 18:12:16 returning from goroutine
```

MINGW32:c:/Users/steve_000

```
steve_000@STEVEN-PC c:/Users/steve_000
$ user
2015/03/15 18:12:01 asking about state:

| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | |O| | |
| | | |O|X| | |
| | |O|X|X|X| |
2015/03/15 18:12:01 making request, time limit 3000 millis
2015/03/15 18:12:04 rpc finished in 3.1109644s
2015/03/15 18:12:04 recommended move:

| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | |O| | |
| | | |O|X| | |
| | |O|X|X|X|O|

steve_000@STEVEN-PC c:/Users/steve_000
$
```

6:15 PM
2015-03-15

Figure 11: A screenshot taken in a Windows 8.1 environment showing one master, one user, and two slave processes running successfully. Note that the colours have been modified to save ink if this document is printed.



iTerm Shell Edit View Profiles Window Help — 100% Wed 8:52 PM

2. slave

```
2015/03/18 20:48:01 DoWork called
2015/03/18 20:48:01 expanding rootState for 2 slaves
2015/03/18 20:48:01 created 7 jobs from root state
2015/03/18 20:48:01 distributing 7 jobs to 2 slaves
2015/03/18 20:48:04 slave connected with 3 past results
2015/03/18 20:48:04 AddResult {X:1928440315904 Y:208339468288 IsXMove:true}
2015/03/18 20:48:04 AddResult {X:1928440315904 Y:13963676876
2015/03/18 20:48:04 AddResult {X:1928440315904 Y:14820992614
2015/03/18 20:48:04 slave connected with 4 past results
2015/03/18 20:48:04 AddResult {X:1928440315904 Y:17397972992
2015/03/18 20:48:04 AddResult {X:1928440315904 Y:14069373337
2015/03/18 20:48:04 AddResult {X:1928440315904 Y:13962025369
2015/03/18 20:48:04 AddResult {X:1928440315904 Y:23386432471
2015/03/18 20:48:04 all results submitted; 2523123 states ar
es
2015/03/18 20:48:04 returning bestMove to user
```

```
| | | | |O| | |
| | | |O|X| | |
| | |O|X|X|X| |
2015/03/18 20:48:01 making request, time limit 3000 millis
2015/03/18 20:48:04 rpc finished in 3.202933079s
2015/03/18 20:48:04 recommended move:

| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | |O| | |
| | | |O|X| | |
| | |O|X|X|X|O|
~/go/src/github.com/jonathangray92/distributed-minimax/user (report)$ uname -a
Darwin Jonathans-MacBook-Pro.local 13.4.0 Darwin Kernel Version 13.4.0: Sun Au
g 17 19:50:11 PDT 2014; root:xnu-2422.115.4~1/RELEASE_X86_64 x86_64
~/go/src/github.com/jonathangray92/distributed-minimax/user (report)$
```

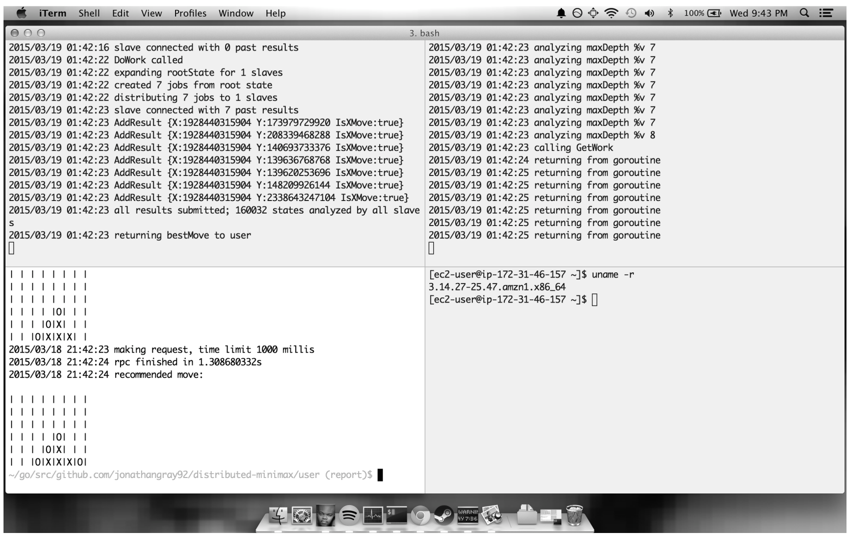
```
2015/03/18 20:48:01 analyzing maxDepth %v 6
2015/03/18 20:48:01 analyzing maxDepth %v 7
2015/03/18 20:48:01 analyzing maxDepth %v 7
2015/03/18 20:48:01 analyzing maxDepth %v 7
2015/03/18 20:48:01 analyzing maxDepth %v 8
2015/03/18 20:48:01 analyzing maxDepth %v 8
2 analyzing maxDepth %v 8
2 analyzing maxDepth %v 9
2 analyzing maxDepth %v 8
3 analyzing maxDepth %v 9
4 analyzing maxDepth %v 9
4 returning from goroutine
4 calling GetWork
4 returning from goroutine
9 returning from goroutine
0 returning from goroutine
```

```
1 analyzing maxDepth %v 5
1 analyzing maxDepth %v 6
1 analyzing maxDepth %v 6
1 analyzing maxDepth %v 7
1 analyzing maxDepth %v 7
1 analyzing maxDepth %v 7
2015/03/18 20:48:01 analyzing maxDepth %v 8
2015/03/18 20:48:02 analyzing maxDepth %v 8
2015/03/18 20:48:02 analyzing maxDepth %v 8
2015/03/18 20:48:03 analyzing maxDepth %v 9
2015/03/18 20:48:03 analyzing maxDepth %v 9
2015/03/18 20:48:03 analyzing maxDepth %v 9
2015/03/18 20:48:04 calling GetWork
2015/03/18 20:48:17 returning from goroutine
2015/03/18 20:48:17 returning from goroutine
2015/03/18 20:48:17 returning from goroutine
```

About This Mac

OS X
Version 10.9.5
Software Update...
Processor 2.8 GHz Intel Core i5
Memory 8 GB 1600 MHz DDR3
More Info...
TM and © 1983–2014 Apple Inc.
All Rights Reserved. License and Warranty

Figure 12: A screenshot taken in an OS 10.9 environment showing one master, one user, and two slave processes running successfully. Note that the colours have been modified to save ink if this document is printed.



```
2015/03/19 01:42:16 slave connected with 0 past results
2015/03/19 01:42:22 DoWork called
2015/03/19 01:42:22 expanding rootState for 1 slaves
2015/03/19 01:42:22 created 7 jobs from root state
2015/03/19 01:42:22 distributing 7 jobs to 1 slaves
2015/03/19 01:42:23 slave connected with 7 past results
2015/03/19 01:42:23 AddResult {X:1928440315904 Y:173979729920 IsXMove:true}
2015/03/19 01:42:23 AddResult {X:1928440315904 Y:208339468288 IsXMove:true}
2015/03/19 01:42:23 AddResult {X:1928440315904 Y:140693733376 IsXMove:true}
2015/03/19 01:42:23 AddResult {X:1928440315904 Y:139636768768 IsXMove:true}
2015/03/19 01:42:23 AddResult {X:1928440315904 Y:139620253696 IsXMove:true}
2015/03/19 01:42:23 AddResult {X:1928440315904 Y:148209926144 IsXMove:true}
2015/03/19 01:42:23 AddResult {X:1928440315904 Y:2338643247104 IsXMove:true}
2015/03/19 01:42:23 all results submitted; 160032 states analyzed by all slave
s
2015/03/19 01:42:23 returning bestMove to user
```

```
2015/03/19 01:42:23 analyzing maxDepth %v 7
2015/03/19 01:42:23 analyzing maxDepth %v 7
2015/03/19 01:42:23 analyzing maxDepth %v 7
2015/03/19 01:42:23 analyzing maxDepth %v 7
2015/03/19 01:42:23 analyzing maxDepth %v 7
2015/03/19 01:42:23 analyzing maxDepth %v 7
2015/03/19 01:42:23 analyzing maxDepth %v 7
2015/03/19 01:42:23 analyzing maxDepth %v 8
2015/03/19 01:42:23 calling GetWork
2015/03/19 01:42:24 returning from goroutine
2015/03/19 01:42:25 returning from goroutine
2015/03/19 01:42:25 returning from goroutine
2015/03/19 01:42:25 returning from goroutine
2015/03/19 01:42:25 returning from goroutine
2015/03/19 01:42:25 returning from goroutine
2015/03/19 01:42:25 returning from goroutine
```

```
| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | |O| | |
| | | |O|X| | |
| | |O|X|X|X| |
2015/03/18 21:42:23 making request, time limit 1000 millis
2015/03/18 21:42:24 rpc finished in 1.308680332s
2015/03/18 21:42:24 recommended move:

| | | | | | | |
| | | | | | | |
| | | | | | | |
| | | | |O| | |
| | | |O|X| | |
| | |O|X|X|X|O|
~/go/src/github.com/jonathangray92/distributed-minimax/user (report)$
```

```
[ec2-user@ip-172-31-46-157 ~]$ uname -r
3.14.27-25.47.amzn1.x86_64
[ec2-user@ip-172-31-46-157 ~]$
```

Figure 13: A screenshot taken in an Amazon Linux AMI environment showing one master, one user, and one slave process running successfully. Note that the Linux kernel version is printed in the bottom-right, and is 3.14.

14 Appendix E: Validated-Tic-Tac-Toe Results (Author: Steven Blenkinsop)

		Input	Output
1	✓	Player: x Board: oo- xxo --x	Player: o Board: oox xxo --x
2	✓	Player: x Board: o-- x-- -xo	Player: o Board: o-- xx- -xo
3	✓	Player: o Board: o-- x-x ox-	Player: x Board: o-- xox ox-
4	✓	Player: x Board: o-- -ox xox	Player: o Board: o-x -ox xox
5	✓	Player: x Board: --- -o- xox	Player: o Board: -x- -o- xox
6	✓	Player: x Board: oox -xo -x-	Player: o Board: oox -xo xx-
7	✓	Player: x Board: -ox --o -x-	Player: o Board: -ox --o xx-
8	✓	Player: x Board: oo- -x- xox	Player: o Board: oox -x- xox
9	✓	Player: x Board: oo- --x xox	Player: o Board: oox --x xox

		Input	Output
33	✓	Player: x Board: oox x-o oxx	Player: o Board: oox xxo oxx
34	✓	Player: o Board: -ox x-- ---	Player: x Board: -ox xo- ---
35	✓	Player: x Board: oo- --x --x	Player: o Board: oox --x --x
36	✓	Player: o Board: oox xo- xx-	Player: x Board: oox xo- xxo
37	✓	Player: x Board: oox xox ---	Player: o Board: oox xox --x
38	✓	Player: x Board: o-x -oo xx-	Player: o Board: o-x -oo xxx
39	✓	Player: o Board: -ox xxo -ox	Player: x Board: oox xxo -ox
40	✓	Player: o Board: -o- x-x -ox	Player: x Board: -o- xox -ox
41	✓	Player: o Board: o-x xxo ---	Player: x Board: o-x xxo o--

		Input	Output
10	✓	Player: x Board: ox- xoo x--	Player: o Board: ox- xoo x-x
11	✓	Player: x Board: oox x-x --o	Player: o Board: oox xxx --o
12	✓	Player: x Board: oxo x-x oxo	Player: o Board: oxo xxx oxo
13	✓	Player: x Board: o-- xo- x--	Player: o Board: o-- xo- x-x
14	✓	Player: o Board: ox- x-o -x-	Player: x Board: ox- xoo -x-
15	✓	Player: o Board: ox- xox xo-	Player: x Board: ox- xox xoo
16	✓	Player: o Board: oox xxo --x	Player: x Board: oox xxo o-x
17	✓	Player: x Board: oox --- oxx	Player: o Board: oox --x oxx
18	✓	Player: x Board: oo- -x- --x	Player: o Board: oox -x- --x

		Input	Output
42	✓	Player: o Board: o-- xxo -x-	Player: x Board: oo- xxo -x-
43	✓	Player: o Board: oo- xx- xox	Player: x Board: ooo xx- xox
44	✓	Player: o Board: o-- -ox xx-	Player: x Board: o-- -ox xxo
45	✓	Player: x Board: oox -ox -x-	Player: o Board: oox -ox -xx
46	✓	Player: o Board: oo- x-x --x	Player: x Board: ooo x-x --x
47	✓	Player: o Board: o-o -x- -xx	Player: x Board: ooo -x- -xx
48	✓	Player: o Board: o-x x-- xo-	Player: x Board: o-x xo- xo-
49	✓	Player: o Board: oo- -xx xxo	Player: x Board: ooo -xx xxo
50	✓	Player: o Board: --- -ox xox	Player: x Board: -o- -ox xox

		Input	Output
19	✓	Player: x Board: oox o-x xxo	Player: o Board: oox oxx xxo
20	✓	Player: x Board: -ox --o xox	Player: o Board: -ox -xo xox
21	✓	Player: o Board: ox- xxo -ox	Player: x Board: oxo xxo -ox
22	✓	Player: o Board: oo- x-x oxx	Player: x Board: ooo x-x oxx
23	✓	Player: o Board: o-x x-o x--	Player: x Board: o-x xoo x--
24	✓	Player: o Board: o-- xx- o-x	Player: x Board: o-- xxo o-x
25	✓	Player: x Board: oo- --x x--	Player: o Board: oox --x x--
26	✓	Player: o Board: oo- -xx x--	Player: x Board: ooo -xx x--
27	✓	Player: o Board: o-o --x x-x	Player: x Board: ooo --x x-x

		Input	Output
51	✓	Player: x Board: o-- x-- ox-	Player: o Board: o-- xx- ox-
52	✓	Player: o Board: oo- x-- -xx	Player: x Board: ooo x-- -xx
53	✓	Player: o Board: oox xox -x-	Player: x Board: oox xox -xo
54	✓	Player: x Board: o-x xoo -x-	Player: o Board: o-x xoo -xx
55	✓	Player: x Board: -o- x-x -o-	Player: o Board: -o- xxx -o-
56	✓	Player: x Board: oo- xx- o-x	Player: o Board: oo- xxx o-x
57	✓	Player: x Board: oox xoo x-x	Player: o Board: oox xoo xxx
58	✓	Player: o Board: oox xx- ox-	Player: x Board: oox xxo ox-
59	✓	Player: x Board: o-- -ox x--	Player: o Board: o-- -ox x-x

		Input	Output
28	✓	Player: x Board: oo- x-x -xo	Player: o Board: oo- xxx -xo
29	✓	Player: x Board: oo- x-- oxx	Player: o Board: oox x-- oxx
30	✓	Player: x Board: -ox -oo -xx	Player: o Board: -ox -oo xxx
31	✓	Player: o Board: o-- xxo x--	Player: x Board: o-o xxo x--
32	✓	Player: o Board: --- -x- ---	Player: x Board: o-- -x- ---

		Input	Output
60	✓	Player: x Board: ox- xoo -x-	Player: o Board: ox- xoo -xx
61	✓	Player: x Board: o-- xo- -x-	Player: o Board: o-- xo- -xx
62	✓	Player: x Board: o-o -ox -xx	Player: o Board: o-o -ox xxx
63	✓	Player: o Board: o-- xox --x	Player: x Board: o-o xox --x
64	✓	Player: x Board: o-- xoo x-x	Player: o Board: o-- xoo xxx