COMPUTER CHESS:

EXPLORING SPEED AND INTELLIGENCE

By

PAUL RAYMOND STEVENS

A Thesis Submitted to The Honors College

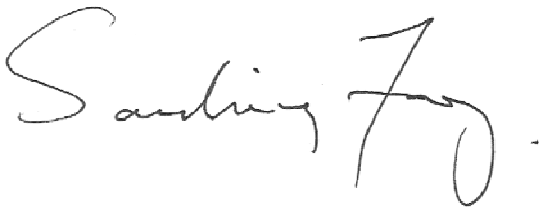
In Partial Fulfillment of the Bachelors degree
With Honors in

Computer Science

THE UNIVERSITY OF ARIZONA

May 2009

Approved by:



Dr. Sandiway Fong
Department of Computer Science

STATEMENT BY AUTHOR

This thesis has been submitted in partial fulfillment of requirements for a degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Signed: Paul R. Stevens

Abstract:

For over fifty years, computer scientists and chess players alike have been interested in the ability of computers to play chess. Every chess program is unique, and their designers continue to explore ways to expand and improve the field.

"PaulChess" is my contribution to that tradition. PaulChess is a computer chess program that I wrote over approximately seven months, starting in August 2008. My goal was to learn the techniques and concepts involved with building a chess program in its entirety. I would try and evaluate some ideas from other chess programs, as well as generate and explore many ideas of my own.

This thesis presents a first person description of PaulChess's creation. I explain how the program evolved from start to finish, sharing the technical details alongside my personal experiences. The purpose of this thesis is to provide readers with an account of what it took to make PaulChess from the perspective of a chess player and a programmer. I also reveal how successful the program ended up being in terms of its playing strength.

Origins of PaulChess

I have been playing chess on and off since my father taught me the game when I was about six years old. From then on, I have always loved chess. Unlike many other games, there is no luck involved. A win or a loss comes as a result of tactics and strategy, one well-planned move at a time.

In the summer of 2007, I received an email from the computer science advisor, informing CS students that there would be an experimental course offered in the fall of that year called *Chess and AI: Computation and Cognition.* At the time, I had no idea how huge an effect that one email would have on the next few years of my education. I enrolled with no expectations but a lot of curiosity.

Computer science professor Sandiway Fong co-taught the class with chess International Master Levon Altounian. As we learned chess rules and strategy from Mr. Altounian, Prof. Fong taught us move generation and search techniques. It ended up being one of the most difficult classes that I have ever taken. Each of us was required to build our own graphical chess program from scratch, as well as implement some challenging programming concepts. The assignments moved at an extremely fast pace, and most of the students struggled to keep up, including me.

The result of all of that hard work was well worth it. By the end of the class, I had a decent working chess program, written in Java, that I could truly call my own. It was not a perfect program, and the AI (artificial intelligence) made plenty of silly mistakes. Still, the program was my own work from start to finish, and it played surprisingly well for a first attempt. Although it made terrible choices in the opening and end game, Mr. Altounian judged it at a rating of 1500 in the middle game, which is approximately the equivalent of the average tournament player.

As the semester came to a close, I was happy with how much progress PaulChess Java had made in such a short time, but I knew that I could do better. And so I decided that I would commit to working on PaulChess for another two semesters as an honors thesis, starting in fall 2008.

PaulChess would turn out to be the most frustrating project that I have ever worked on in my entire life, but one of the most rewarding as well. My ultimate goal from the beginning was to develop a program that could beat me consistently. I wanted to teach it everything that I knew, to watch it grow before my eyes, and to see it finally play a solid game of chess.

This is the story of that program.

I have chosen to present this thesis as a personal account of how the program evolved, alongside technical concepts and analysis. This approach was suggested by my thesis advisor, Prof. Fong. As he once put it, creating an entire chess program is more of an art than a precise science, and I have composed this thesis with that idea in mind.

My goal is to explain how I made my chess program from a first person perspective—to explore the techniques and concepts involved, but also to communicate the experience itself, something that an interested chess programmer would never find in a technical paper or research article.

Note: Unless otherwise stated, "PaulChess" refers to PaulChess 1.0, the version of the program written in Slag and developed from fall 2008 to spring 2009. PaulChess Java is the version of the program that was created as a class assignment in fall 2007.

Early Choices – Programming in Slag

PaulChess Java had a lot of design strengths, but it had plenty of weaknesses too. One disadvantage was that it was written in Java. Java executes on a virtual machine, and so it runs slower than it would in a programming language that compiles to native code, such as C or C++. For a while I considered writing the new version of PaulChess in C, but decided against it. I simply did not have enough experience with C, and I concluded that the extra time it would take to learn the language would not justify the speed increase.

I decided on a compromise: program PaulChess in Slag. Slag is an object-oriented programming language, much like Java, written by NAU professor Abe Pralle. I already knew Slag very well because I had used it to program other games. Prof. Pralle suggested that Slag would probably be faster than Java because it has the option to cross-compile to C++, improving its performance with demanding programs. I am not sure if Slag actually ended up running faster than Java because I changed so many of the program's algorithms from the original version. However, I do know that Slag was easy to use and allowed me to focus on the program, not the programming language.

This meant that **I had to recode the entire program from scratch**. I knew that it would take some time, since PaulChess Java took about 4 months. But I was motivated, and so I began coding furiously to get it operating with all the features from the original.

Slag's official webpage is here: http://www.plasmaworks.com/slag

Early Choices – No Rotated Bitboards

The second major change that I made from the original program was the board representation. PaulChess Java used bitboards to store the state of the pieces, which I retained in PaulChess Slag. Bitboards are a way to represent the state of the game in binary using twelve 64-bit numbers, one for each piece type (e.g., white pawn, black knight). There are many motivations behind the bitboard approach. The main one is that bitboards make it much faster to calculate legal moves, due to the speed of bit operations.

I originally learned about bitboards from Prof. Fong's lectures. I have included a link to them here: http://dingo.sbs.arizona.edu/~sandiway/CSC483/lecture3.pdf

As a class requirement, PaulChess Java also maintained three rotated bitboard representations of the game to calculate moves for sliding pieces. This meant that the board state was stored as four bitboards for each piece type: no rotation, 90º rotation, and ±45º rotation.

Rotated bitboards were one of the toughest parts of PaulChess Java to get right. It takes quite a bit of extra work to support rotated bitboards at all. They require several conversion functions so that coordinates can be translated between bitboards. It is also a challenge to keep all four rotations in sync, because any change to the pieces on one board must occur on all of the others.

That may not sound too difficult, but it is important keep in mind that attacks are calculated and spread out across all four boards, as this is the entire purpose of using multiple rotations. It is not just that the boards must be kept in sync, but that the boards are not totally interchangeable, and some boards will always have information that others do not. It would be too slow to keep them all *totally* identical.

Also, I was concerned that having to copy and change all of these rotated bitboards every move would take a lot of processing (there are forty-eight bitboards total). Keeping them synchronized also requires four times the instructions, as does searching for any particular attack threat, since it may occur on any of the rotations.

All of these considerations caused me to make a major decision: **abandon rotated bitboards for PaulChess Slag**. I knew that it would save me a lot of testing and a lot of headaches. But how was I going to replace them? Sliding piece moves are tough to calculate without rotated bitboards, but I was convinced that I could do it just as fast using lookup tables. Unfortunately, I had no idea how it would be done.

Rotated Lookups Revealed!

After giving it a lot of consideration, **I figured out how to make sliding piece lookups without the use of rotated bitboards**. This is one aspect of PaulChess that I am particularly proud of, because I came up with the entire algorithm myself. I have never seen any other programs that tried to do it this way, and so I feel like it is my original idea. I am definitely glad that Prof. Fong taught the class how to use rotated bitboards, even though I did not use them in PaulChess, because I never would have come up with this idea if I did not fully understand how rotated bitboards work.

Here is the basic algorithm:

1. Mask off the relevant “line” of pieces (be it a row, column, or diagonal of eight bits). This mask is obtained from a lookup table, and it is based on the requested rotation.

2. Bit shift the line of pieces to a standard position. This is necessary because unless the line is in a standardized position, the lookup tables have to be much, much larger. The standard position for each rotation is represented by 1-bits in this diagram.

0º rotation	90º rotation	45º rotation	-45º rotation
11111111	00000001	00000001	**1**0000000
00000000	00000001	00000010	01000000
00000000	00000001	00000100	00100000
00000000	00000001	00001000	00010000
00000000	00000001	00010000	00001000
00000000	00000001	00100000	00000100
00000000	00000001	01000000	00000010
00000000	00000001	10000000	00000001

3. Perform a sliding piece lookup of the relevant type. Although the lookup tables work fine with 0º because all of the entries are adjacent, the other rotations require a perfect hash lookup. Surprisingly, I found out through trial and error that the table can have an almost perfect minimal hash with a simple hash function. The hash function is just modulus, and the rotated tables only take about two more entries each than the un-rotated table.

4. Bit shift the line of attack bits back to its original position.

5. Mask off the friendly pieces.

Repeat as needed, based on which lookups the piece requires (e.g., a queen uses all four lookups).

Rotated Lookup Example

Suppose that this is the current bitboard (all pieces are represented):

```
00010011
00010100
01100010
00110111
00001000
00000100
01011001
10000000
```

The bold, underlined 1 is a rook, and I want to look up its legal vertical moves.

1. Mask off the appropriate line:

```
00010011       00010000      00010000
00010100       00010000      00010000
01100010  OR   00010000  =   00000000
00110111       00010000      00010000
00001000       00010000      00000000
00000100       00010000      00000000
01011001       00010000      00010000
10000000       00010000      00000000
```

2. Shift the line to its "standard" position (in this case, the rightmost column):

```
00010000            00000001
00010000            00000001
00000000            00000000
00010000  >> 4 =    00000001
00000000            00000000
00000000            00000000
00010000            00000001
00000000            00000000
```

3. Perform the lookup. The first index is the position of the rook in the line, and the second index is the value that was just computed. *% 257* is the hash function, which works purely by coincidence:

```
Rot90[3][currBoard % 257] =

00000000
00000001
00000001
00000000
00000001
00000001
00000001
00000000
```

4. Rotate the result back to original position:

```
00000000            00000000
00000001            00010000
00000001            00010000
00000000  << 4 =    00000000
00000001            00010000
00000001            00010000
00000001            00010000
00000000            00000000
```

5. Last, mask of the friendly pieces (just one shown in this example).

```
00000000            00000000     00000000
00010000            00010000     00000000
00010000            00000000     00010000
00000000  AND NOT   00000000  =  00000000
00010000            00000000     00010000
00010000            00000000     00010000
00010000            00000000     00010000
00000000            00000000     00000000
```

And finally I have attack bits for the rook's column. It is the exact same procedure for the diagonals.

Sliding Piece Alternatives – What I Did Not Try

Although rotated bitboards are still a common option for generating sliding piece attacks, there are others. While trying to speed up board generation (later on in the project), I researched some fast alternatives, such as kindergarten and magic bitboards. They operate on the idea that is that it is possible mask off the relevant squares for a piece's attacks, and then multiply the result by a "magic" number, which can be used as a hash key to look up the solution in a table.

Here is an example: http://chessprogramming.wikispaces.com/Magic+Bitboards

Why did I never try the idea? I simply did not understand it well enough to implement it. The bit operations are complicated, and so are the concepts. Even with a few resources out there explaining the problem, I never felt like I had a strong enough command of the idea to actually apply it. It is starting to make more sense to me now, and so I may re-research the idea and try it in a future version of PaulChess.

Bit Sieve for Identifying Positions

The "bit sieve" is another original idea that I was not sure would work, but decided to try anyway. During move generation, a program constantly needs to figure out where the 1-bits are in a bitboard. For example, it needs to find individual piece bits to calculate their attacks, and it needs to find individual attack bits to generate next moves.

The classic loop of sixty-four is thought to be pretty slow, so I had another idea. What if I **pre-computed the positions of each 1 in every possible byte**, and stored this in a lookup table? The positions would be represented as a list of indices. This way, I can look up the positions for eight bits all at the same time, using minimal calculation.

Here is an example of one of the entries in the bit sieve. The array index is in binary (to show the whole byte), and the positions are expressed from least significant to most significant, which matches my bitboard representation:

Bits[01001011] = [0, 1, 3, 6]

Was it faster than the loop? I never tested the alternative. I was too busy building the program to try. In retrospect, I probably should have tried the alternatives at some point, but it was hard to want to spend a lot of time recoding something that already worked fine, especially since I felt like I was playing catch up with a program that I wrote a year ago. Also, it becomes harder to change the bit position algorithm as the program progresses, because it is used in so many places.

Master Position Lookup – I Wish!

The bit sieve was a cool idea for analyzing a whole byte in one operation. But it would save quite a few more operations if I could look up the positions for an *entire* bitboard at once, using the same concept as the bit sieve.

The problem is that this would take 2^{64} array entries (instead of 2^8). 2^{64} is otherwise known as 18,446,744,073,709,551,616. Multiply that by 8 bytes each for the key, and an average of about 128 bytes per entry, and the result is over 18 billion terabytes of data. No computer in the world could store this, not even close. Probably not even all of the computers in the world combined.

But then I realized that most bit position lookups are just searching for one or two bits, not sixty-four. Consider the lookups for the rooks, knights, bishops, queen, and king. In all of these cases, it is extremely rare that the population of the bitboard is more than two. So what if I **design a table that can identify the positions for one or two bits**, and then employ a slower technique if it turns out that this master lookups fails (which happens when there are 3+ "1-bits" in the bitboard). Sounds like a great idea, right?

The Algorithm That Could Have Been

Like my idea for rotated bit lookups, this idea for a small population-position lookup was entirely my own, and I thought that it was pretty good. So why did I choose not to use the idea in PaulChess? After starting to implement it, I realized that it would be very difficult to finish. I would have to insert the code to search the table, and more annoyingly, use an alternative method for finding positions when the table lookup fails. There would have been no way to keep the code from becoming messy and complicated.

Implementing the table also turned out to be more difficult than I thought it would be. Of course it uses a hash lookup, since the keys are not consecutive in any systematic way. But my usual hash function, %, was not good enough for this purpose. I realized that it would take a massive array for % to work as the hash function, and only about two percent of the array would have been populated. I needed a better way (at least out of principle), and I did not have one.

Taking the idea any further did not seem worth it. It would be one thing if I could somehow do the master lookup (with any arbitrary 64-bit board), but with only the ability to look up the positions for one or two bits, it would only end up being useful on about 25% of all of the position searches. **I was looking at a modest speed increase on 25% of position searches in exchange for obliterating the cleanliness of the code, and introducing a large chance of bugs**. I needed to draw the line somewhere, and I drew it there.

Still, I found it hard to give up on an idea that I spent a long time developing. It was something that I would have to get used to for this project.

Full Speed Ahead! …To What was Done a Year Ago

Having finished dealing with most of the board generation choices, I moved onward. I was only a few weeks into developing the program, but every day that it was not caught up to the Java version felt like another day that was wasted. I programmed at a frantic pace to add castling, en passant, promotion, etc. All of these special moves are a lot of trouble, but quite a bit easier to implement the second time around.

If memory serves me right, it took me one month to finish PaulChess version 0.1, which was the first fully working version. It had board editing and a 4-ply alpha beta game tree search (a game tree search is how the computer picks a move). PaulChess 0.1 was more or less equivalent to PaulChess Java, in both intelligence and speed. Fortunately, the code was much cleaner. The algorithms were faster, simpler, more intuitive, less buggy, and easier to expand.

It felt good to surpass something that took four months of work in just one, but I still could not shake the feeling that I was only just starting the project. As far as the end user was concerned, the programs were approximately the same at that point. Only a programmer would have been able to recognize the difference.

Automated History Feature

One of the first things that I needed to add was a game history, so that it would be possible to print a game to a file without having to record every move by hand. That would have taken a very long time, since over the course of its life, I have played at least a couple hundred games with PaulChess. Sure only about 25% of them are recorded, but 25% of a couple hundred games is still a few thousand moves.

It was not too hard to get a basic history in the game, because part of my board generation technique includes a “last move” object that is passed to a new board. This last move object really helped speed things up, because it provided all of the information that I needed to figure out which piece just moved and where it went. Beyond the data in this object, all I needed to figure out was if a piece got taken. This is calculated anyway as the next board generates.

Here is the real difficulty: making the history as terse as algebraic notation (the standard way to record chess games). Algebraic notation in chess is written in the shortest form possible without ambiguity. When I began to think about all that it would take implement this, I was intimidated.

I decided instead to compromise and err on the side of caution. **PaulChess always lists the starting square of a piece, so that it does not have to worry about ambiguity**. The notation looks like this:

```
1. e2e3 e7e5 2. Qd1g4 Nb8c6 3. Bf1b5 a7a6 4. Bb5a4 b7b5
```

It really should look like this:

```
1. e3 e5 2. Qg4 Nc6 3. Bb5 a6 4. Ba4 b5
```

The reason that I did not implement full algebraic notation appears when two of the same type of piece can move to the same square. For example, say there are two knights that can move to d5. This is not a problem for my program because it will always name the starting square of the knight, but real algebraic notation only names the row of the moving knight… unless this would not remove the ambiguity, in which case it names the column… unless this would not remove the ambiguity, in which case it names the row and column, which is exactly what my program does all of the time.

The algorithm for full algebraic notation is probably not as difficult as I made it out to be in my mind. Still, I did not want there to be any mistakes in the history, and full algebraic notation did not seem that important at the time. Plus, PaulChess's notation still resembles real algebraic notation closely enough that any chess player would understand what was going on (maybe even easier than normal).

This automated history ended up being much more useful than I thought it would at first. **When it came time to implement the draw by repetition rule and manually generate the opening book, the basic history feature would become invaluable**. I will explain those shortly.

Multiple AIs and Integrated Evaluation

I knew that I wanted my program to be able to play against itself so that I could have an easy way to test changes, and see if the program was improving relative to its earlier versions. In my mind, I made this much more difficult than it needed to be, due to an earlier idea I had to increase performance.

As is clear by now, most of my overcomplicated ideas start out as ways to increase speed. In this case, my thought was that **it would save a lot of time to gather evaluation data as a move generated**, rather than collecting all of the data later.

For example, to do a simple material evaluation requires looping through each bitboard and performing a population count (which, as I discussed earlier with the bit sieve, is not particularly fast). However, I realized that this already occurs when the board is generating legal moves. So why not just evaluate the material right then for free?

This idea probably improved performance, at least a little bit. I did implement it, and it has been a part of PaulChess ever since. The overcomplicated part comes next.

Every AI uses a different evaluation, and some AIs need to gather a lot more data than others. This raised two issues:

1. It did not seem fair to collect all of the data for a complicated AI and only use a little bit of it when running a simple AI. These extra, useless calculations would slow down the simple AI, and give an inaccurate reading of their relative speeds.

2. Assuming that some AIs would use evaluation techniques that others do not, even the most complicated AI would be wasting time with evaluations that it would never use from other AIs.

From these two points, I realized that data collection needs to change based on the AI. However, I still wanted to use my idea that data is collected as a new move generates (when possible).

I really racked my brain to think of a solution here, and a lot of questions were running through my head. Should I pass in some sort of "AI object" as each board was being created, that would tell the board what data it needed to collect, based on flags? If so, how would this object be passed in? By the GameTree as it generates boards? Will that take a lot of extra memory and a lot of trouble? How ugly is the code going to be if it needs switches to turn data collection on and off? Which entity would create the AI object, and what data would that object contain?

Finally, I came up with an idea: use a global variable (just an integer) indicating the current AI. The GameTree changes this variable right before it begins running a search. A board can turn on and off data collection switches based on this variable. No extra objects, no complicated interactions, no wasted memory.

It turned out to be **a great solution to a problem that did not really exist**, at least not for this version of PaulChess. As I would find out later, it is not a big deal to collect a little bit of extra data for AIs that do not need it—the performance hitch really is not that noticeable. But even more importantly, it is not a big deal to collect most of the data *after* board generation.

It probably does save a bit of time to collect relevant data during move generation when possible, but it overcomplicates the code. The performance increase depends on how many redundant operations there are between move generation and board evaluation. For my program, this was not a lot, so the speed increase was minimal. And to make it even less important, my most complicated (and best) AI uses basically the union of all my other evaluation functions. That AI would always run at full speed anyway, since it would be using at least as much data as any of my other ones.

This was just one more example of an idea that I spent a lot of time on, but that did not end up panning out. That was a pretty big theme for PaulChess in fall 2008. Luckily, I had so much to do that I did not have time to feel bad about it. I just moved on to other ideas, knowing full well how far PaulChess had to go before it would become a decent program.

My First Evaluation Concept – Material and Mobility

Now that I had a working chess computer with selectable AIs, I thought that all there was left to do would be to perfect the evaluation function. It would later turn out that I was seriously mistaken, but I would not realize this until the beginning of the next semester.

My first evaluation idea came directly from PaulChess Java: material and mobility. I decided to use the standard "human" values for material: pawn = 100, knight = 330, bishop = 330, rook = 500, queen = 900. The importance and definition of mobility changed a couple of times as I worked with the evaluation. I had two basic lines of thought:

1. Mobility is the number of squares that one side threatens (defense does not count) *taken as a whole*, regardless of the piece.

2. Mobility is a count of which squares *each piece* threatens, ignoring what other pieces are doing.

Both concept of mobility had their advantages. I found out through play that the first definition of mobility causes pieces to cover more total ground. However, because it does not matter *which* piece threatens a square, often times the queen ends up threatening most of the squares while the other pieces are doing nothing.

How about definition 2? Well, this got the computer to use each piece to the fullest… but in many cases, they would be covering the same squares. What good is each piece being highly mobile if they are all focused on the same, pointless squares?

The solution was to combine both mobility concepts. This way, I got a decent **balance of each piece being used intelligently, along with adequate overall board coverage**. Each square of the "combined" mobility (from type 1) is worth 10 points, and each individual mobility square (from type 2) is worth 5 points. This AI would come to be known as Simple1. It took many games to determine this balance, but it was worth it to find a good benchmark to compare against my other AIs.

Limitations of Material and Mobility

This simple evaluation idea can play a fairly strong game of chess, at least in the middle game. Although it would occasionally make mistakes, its ability to look four moves ahead and not miss anything allowed it to play consistently, and sometimes see sequences that a decent human player might miss.

Although it had its strengths, it certainly had its drawbacks. For one, **it played terribly in the opening**. Because mobility was the driving force behind most of its choices (assuming that it can keep the material even), its main strategy would always be to move out the queen as early as possible, and spend most of its time trying to increase the queen's influence. One thing that I can give it credit for is that it usually finds a way to keep its queen safe, despite it being out in the open.

However, even beginning chess players are taught that moving the queen out early is usually a bad idea. The queen quickly becomes a liability as it is chased around the board by other pieces, which develop into good locations and obtain strong attacks. Even when it was not moving the queen, it would often do odd moves that would never turn up in any opening book.

It had another problem in the end game. Unless a pawn will promote within four moves, Simple1 has no way to know that it is a good idea to advance pawns. So suppose it gets down to just a couple of pawns and the kings. Even a beginner could beat Simple1 at this point, because it has no idea what to do. Levon would later approximate its rating to be 1100 in the end game, compared with 1500 in the middle game.

Trying to Fix the Opening

I have always thought that the opening phase is very important. If I can gain a significant advantage early on, the rest of the game is that much easier. Unfortunately, Simple1 plays terribly in the opening. It would probably only do well in the opening against an inexperienced player who does not know how to deal with an extremely aggressive queen.

Keeping the queen from moving out in the opening turned out to be one of the hardest parts of PaulChess to perfect, and I am still not completely satisfied. On the one hand, I wanted to encourage the program to develop quickly and take over as much of the board as it can, making strong attacks and defending with any means necessary. On the other hand, I wanted to restrict the queen from advancing too early and being pushed around. It is not too hard to prevent the queen from moving at all in the early game, but when an early queen move is actually good, I wanted my computer to be able to find it. I also wanted to make sure that the queen moves out appropriately when the early opening phase is over.

After many games and tweaks to the evaluation, I realized that there are two main ways to keep the queen from moving out when it should stay back: **explicitly penalize early queen moves, and make other moves more attractive**. My first focus was on option one, but over time I realized that the second alternative is easier and more reliable. Instead of giving harsh penalties for a queen that moves out early, I needed to show the program that other moves are better, and override the mobility bonus that a queen gains in the opening.

With a little research on GNU chess (http://www.gnu.org/software/chess), I found out that a common way to help a computer play in the opening is called “piece-square evaluations.” Essentially, this involves “hard coding” the idea that particular squares are good for particular pieces in the opening. For example, c3 and f3 are both common knight locations early on, and b2 and g2 are common bishop moves. So I give a bonus when these pieces are in these squares. Similarly, d4 and e4 are strong pawn moves that deserve a bonus, while f3 is a terrible pawn move and incurs a penalty.

PaulChess makes fairly extensive use of piece-square evaluations in the opening for just about every piece, but this element of the evaluation is quickly scaled down as the game state advances

toward the middle phase. This way, the knights know to leave c3 and f3 when the time is right (that is, when material and mobility become more important than opening heuristics).

PaulChess would end up playing fairly strong in the opening, but not for a while. It took me a long time and many other additions to the program to get the balance right, and although it was playing a bit more conventionally in the opening, it still was not playing very well overall. At this point I moved on, hoping that more evaluation ideas would help me figure out the problem.

Trying to Fix the Pawns

A lot of chess programmers recommend adding in evaluations for the pawn structure. At first, I thought that it was not necessary. I was under the assumption that pawn structure weaknesses would be caught by the material/mobility evaluation, but I was wrong. Material/mobility only catches the problem if the problem is readily apparent within the search window (e.g., a doubled pawn gets taken in the next two moves). However, this is often not the case. For example, an isolated pawn may not become a liability until the late game, but it almost certainly will eventually. Additionally, mobility will only detect an issue with adjacent doubled pawns, not separated doubled pawns or isolated pawns.

Adding pawn structure considerations definitely helped in the opening. Intermediate players know not to double or isolate their pawns early on, and these considerations helped PaulChess play better overall, and particularly in the early game.

I also gave out advancement bonuses for pawns. As a pawn gets closer to promoting, it becomes more valuable, especially late game. This was another evaluation that I left out of Simple1 because I thought that the tree search would find the problem on its own. One of the most important evaluation concepts, that I would come to realize over and over again, is that **a game tree search will only find something, good or bad, if it is within the search window**!

2008 Comes to a Close and My Frustration Flares

Pawn structure and advancement bonuses helped, but the program was still not playing that well. It would miss good moves, play odd moves, and occasionally make blunders that it should have caught within the search window. I did not know what to do. I was frustrated, I was disappointed, and I felt like giving up. I continued to tweak the evaluation constants, adjust the piece-square evaluations, compensate for the game phase, but it was STILL playing poorly. It even struggled against PaulChess Java, which I had finished almost exactly a year before.

What was I supposed to do? Nothing seemed to help, even though I was trying everything that I could think of. Winter holiday began, and I decided to take a break.

2009: A New Year and a New Strategy

Finally, it came to me. I needed to stop agonizing over small adjustments to the evaluation and turn my attention to the tree search. I would come to realize that there are **three main aspects to a chess program**, each of which contributes to the program's speed and/or intelligence:

1. Move Generation
2. Evaluation Function
3. Tree Search

I had spent a couple months working on move generation and the evaluation function, but almost no time improving the tree search. The tree search is an important aspect of computer chess because it is the computer's method for picking a move. It does so by looking at hundreds of thousands of possible positions. The basic idea is that the computer generates all of the legal moves from its starting position, then it generates all of the legal responses that its opponent can make, then it generates all of its possible responses to that, etc. Finally, the search reaches a maximum depth, and the computer tries to find the most logical series of moves, assuming that both players are picking the best possible move at every turn.

The computer finally chooses its move based on a strategy of pursuing the best possible outcome no matter how smart its opponent plays. This is called a minimax search, and is employed by nearly all chess programs.

Unfortunately, the game tree search is not something that I know a lot about. It is complicated, difficult to debug, and there are a great number of different techniques out there. Still, I knew that I had to improve it. I needed to do some research, and fast.

Here are two of the best resources I found on chess programming in general, and the tree search in particular:

http://www.gamedev.net/reference/programming/features/chess1/
http://chessprogramming.wikispaces.com/

The first article by Francois-Dominic Laramée explains the basic step-by-step development of his chess program. It has a lot of great concepts that I tried in my own program during the second semester.

The second website is a large collection of articles, all related to chess programming.

Nearly all of the technical concepts that I discuss from here on out are inspired by articles from those websites.

Speed Improvement #1: The Transposition Table

My first goal of 2009 was to increase PaulChess's speed. I wanted to search deeper than 4-ply (a ply is a single move by one player), and my first strategy was to implement a transposition table. Basically, a transposition table is a cache of positions that have already been searched, along with the results of those searches. This is beneficial for two reasons:

- A tree search looks at a lot of redundant positions, even with alpha/beta pruning (alpha beta pruning is a way to search less of the game tree without affecting the results of the

search). For example, 1. Nc3 Nf6 2. Nb1 finishes in the exact same as position as 1. Na3 Nf6 2. Nb1, yet a normal tree search would fully evaluate both. The point here is that if the program searches one, there is no need to search the other. The results will be the same.

- Alpha beta pruning relies heavily on move ordering to increase cutoffs and reduce the number of searched positions (a "cutoff" is the term for what happens when the program is able to search less of the tree, due to other data that it has collected). If a program can try the strongest moves first, it can save a lot of search time. A great way to order moves is to do shallower searches and store the results in the transposition table, which the program can use to help with the move ordering of the deeper search and improve cutoffs. This technique is called iterative deepening, and is used by most chess programs.

The first step to creating a transposition table is to implement a hash function to index the board position, so that the program can look up transposition results in constant time. The standard method is called "Zobrist Hashing." It takes into account the twelve bitboards, the castling flags, the current turn, the en-passant row, and a lot of random numbers. The exact procedure can be found here: http://chessprogramming.wikispaces.com/Zobrist+Hashing

It is fairly simple, but it does take some time to implement and test. **The transposition table definitely sped up the program**, even though I am pretty sure that I am not using it to its full potential (due to my limited understanding of the tree search).

To give a basic idea of how much it helps, here is how long it takes PaulChess to evaluate a complicated middle game position using iterative deepening and the transposition table:

```
Elapsed time: 41.0620 seconds
```

And here is that same search without iterative deepening and no transposition table:

```
Elapsed time: 68.1560 seconds
```

This produced a savings of over twenty-seven seconds, or about 40% off. Not bad at all!

This program, like some others, actually uses a two-table scheme. When a hash collision occurs on the table query, one table favors storing data from deep searches, and the other table favors data from recent searches. I came across this strategy (called "2-Deep") from a research paper on transposition table collisions by Dennis M. Breuker and Jos W. H. M. Uiterwijk: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.53.5189

This turned out to be a significant win over the single-table schemes that I tried, just as the paper predicted. Incidentally, one of the reasons that I think my table is not being used to its full potential is that the paper also concludes that direct cutoffs dominate the speed improvements from using the table, whereas my table improves the program's speed mostly due to its move ordering aspect.

There is another huge benefit to transposition tables that I would find out shortly: they make it very easy to implement an opening book.

From Transposition Table to Opening Book

Zobrist hashing and a transposition table allowed me to easily create an opening book using exactly the same concepts. Although there are many ways to do the opening book, here is the method I chose:

1. The opening book is its own hash table of moves, just like the transposition table. However, all it needs to store is the starting position (its Zobrist hash key) and the Zobrist hash key of the response.

2. Before searching a position, check the opening book table. If the starting position is in the table, generate the next set of moves, and find the move that has the same Zobrist key as the book's response. This is the move suggested by the opening book.

This technique is easy to implement, reliable, extremely fast, and takes a trivial amount of memory. Now I just needed to populate the opening book with moves.

Generating the Opening Book: Surprisingly Fun and Easy!

I thought that there would be lots of great opening databases out on the internet that would be easy to use and integrate into a chess program. If there were, I never found them. I did not want to create the opening book on my own, but this ended up being a good thing. It gave me perfect control over how my program would play in the opening, and that was exactly what I wanted.

Still, manually generating an entire opening book seemed like a daunting task at first. But when I thought about it, I realized that there are not *that* many common opening lines. Adding just a few of the most conventional ones can really go a long way. Even so, there remained the problem of how to actually put these into the program.

I found a solution that that makes it fast and easy to put a lot of positions into the opening book manually. I am particularly proud of it, so I will describe the steps below. It is important to note that I had already created a history system so that a player can move backward and forward again within the game's history. All it takes is two stacks (a "back stack" and a "forward stack") that store board objects.

Here are the steps to generate a move for PaulChess's opening book:
1. Play moves to reach the starting position.
2. Play the move of the result position.
3. Press 'o'.

To back up a couple of moves and start fleshing out a new line, I can just press 'b' until I have reached the position that I want, and then continue as usual.

Why is it so easy? Since the program is already keeping a history, it has all of the data it needs once 'o' is pressed. The most recent position in the "back stack" is the starting position. The current position is the result (i.e., the "book move"). To increase the readability of the book file, I included a string representation of the current history. Here is what a few moves from the opening book look like:

```
332e2f2c77a8402d|fdf3d95385e5b7af|History: 1. d2d4
fdf3d95385e5b7af|84691e9fe2cbee66|History: 1. d2d4 Ng8f6
fb128a8b4ff1b087|5927b7b1098f7c37|History: 1. d2d4 Ng8f6 2. Ng1f3 e7e6
```

The first number is the Zobrist key of the starting move (in hexadecimal), the second number is the Zobrist of the result, and the last entry is the history. Since it is human readable, it is easy to remove positions by hand if I want to change the opening book. It is also easy to see which openings are well developed in the book, and which ones need more work.

A little trick: **when writing the book to a file, alphabetize the entries by their history strings**. This organizes them in a very useful way, with no human effort.

Currently, my opening book has 219 moves in it, all of which I added by hand. Thanks to this system, it only took a couple of hours to add all 219, and it was actually a lot of fun! To decide which moves to put in, I used an online openings database for help:
http://www.chessgames.com/perl/explorer

This website provides a graphical interface to explore openings. As a user makes moves, it presents a list of common responses, states how often each response is played (among professionals), and gives the win percentage of the response (white win, black win, or draw). To pick a move from any particular position, I would usually select a response based on a balance of popularity and win percentage. Some popular moves are not that likely to result in a win (according to the data), and some moves with a high win % are hardly ever played. I decided that the safest choice would be to consider a combination of both statistics.

Another great part about this type of opening book is that it relies on positions, not opening lines. This means that it handles transpositions (where the same position can be reached multiple ways) without any difficulty. It also consumes a lot less memory. The only disadvantage is that it requires some sort of hashing system to store the boards… but because I had previously implemented a transposition table, the hashing system was already finished!

Speed Improvement #2: Null-Move Pruning

PaulChess was getting faster, but still not fast enough to add extra plies to the search. In the gamedev.net article, I read that almost all modern chess programs employ null-move pruning because it dramatically increases speed, at the cost of a very small amount of search accuracy.

The basic assumption is that there will almost always be *some* move better than doing nothing in any given position. That is, doing nothing is rarely the best possible move (even if it were legal, which it is not). The idea behind null-move pruning is that a program can try doing nothing as if

it was a move, and then search the position less deeply than it normally would. If this modified search produces an alpha or a beta cutoff, the program can assume that the best move from this position would also have produced a cutoff, and treat the situation as if a cutoff occurred normally.

This saves a significant amount of search time. It takes less time to evaluate the null move than a normal move since it is searched to a shallower depth (usually two to three plies less the normal search). **If a cutoff occurs, the program will find it even faster than if it had searched the best move first**!

Just how much time does it save? Like with the transposition table, I do not think that I am implementing it perfectly. It is a lot easier to describe the idea than to code and debug it, especially considering that I only have a rudimentary understanding of the alpha/beta search in general. Still, the results speak for themselves.

Here is how long it takes PaulChess to evaluate a complicated middle game position using null-move pruning:

```
Elapsed time: 41.2030 seconds
```

Here is that same move without any null-move pruning.

```
Elapsed time: 133.8590 seconds
```

This is a savings of about 70%. Not bad. I owe just about all of the credit to Laramée's article on the topic: http://www.gamedev.net/reference/programming/features/chess5/page3.asp

By the way, I realized that I could store the results of null-move searches in the transposition table, and get a little extra speed out of it. I just had to make sure that I did not misuse the results by treating them to be more accurate than they actually are.

Speed Improvement #3: Two-Phase Move Generation

Some chess programs generate one legal move at a time during a tree search. Some programs generate all of them all at once. For a while, PaulChess was the latter. I started to think a little bit about the advantages and disadvantages of these two generation strategies in an effort to improve speed.

Generating all moves at once:

- If a cutoff occurs early on, the program wastes all of the time that it took to generate the discarded boards.
- Since all moves are fully generated at once, the program has all of the information it needs to make an intelligent decision about which move is likely to produce a cutoff. It is also possible to order the moves based on their static evaluation, which speeds up the search.

Generating moves one at a time:

- The program never generates wasted boards, even when an early cutoff occurs.
- It is somewhat difficult to decide on move ordering because the program may not know enough about moves that it has not yet generated. And there will definitely not be any move ordering based on static evaluation.
- It seems like quite a challenge to implement, because the state of the move generation must be stored.

Before I committed to trying one-at-a-time generation, which I was not even sure how to do, I decided on a compromise that seemed quite a bit better (at least in theory). Since material imbalances are by far the largest factors affecting cutoffs, **I chose to generate all captures first, and order these by static evaluation**. PaulChess searches these captures, and in the likely event of a cutoff, it does not bother generating any of the other moves. If there was no cutoff, all of the other moves are generated at once and ordered by static evaluation. I call this "two-phase generation." As far as I can tell, this approach yields just about all of the benefits of both generation methods, and greatly minimizes the disadvantages.

There is a simple way to implement this in code: after generating a piece's "move bits", just AND these bits with the opposing color to generate only captures. To generate only non-captures, use AND NOT instead.

The speed results were definitely noticeable. Although I did not retain the ability to turn this on and off in the most recent version of PaulChess, I did log the difference a while back when I was first trying the idea.

Here is a 5-ply middle game evaluation with two-phase generation and static ordering:

```
Elapsed time: 94.1250
```

That same move with one-phase generation and static ordering:

```
Elapsed time: 131.1710
```

The result was a savings of about 28%. I was happy with that! It is also interesting to note that there was an almost 100% speed increase using 4-ply moves instead (eight seconds improves to four seconds), suggesting that the savings depend quite a bit on the specifics of the searched boards.

I quickly discovered an addition benefit to the two-phase generation strategy: **it is necessary for the all-important quiescent search**, which is coming up next.

The Eternal Problem of the Search Horizon

I have said it once, and I will say it again: a game tree search will only find something, good or bad, if it lies within the search window.

Although my program was moving quite a bit faster, and could comfortably play at about 5-ply (on average), it would still make some odd moves every so often. So I decided to do a bit of investigation. It is impossible to tell exactly what positions are being searched and rejected (since any given search will look at hundreds of thousands), but I came up with the idea of examining the next few moves that the computer thought would be played.

When conducting a minimax search, the computer finds the most likely line of moves assuming "perfect play" by both players, where perfect play is evaluated from its perspective. Therefore, **I can look at the exact line that it considers to be perfect play by both players**, and discover weaknesses in its strategy when its idea of perfect play seems to be way off.

What I found was astonishing.

Often times the computer would make an incredibly bad capture as the deepest move in the search tree. Why? Because the next move, the move that makes the capture terrible, would be just beyond its search horizon. And in fact, the computer's final move in the search tree would *always* be to capture the highest value piece, as long as any capture was available. Of course, the vast majority of the time, just blindly taking the highest value piece is a terrible idea… it is exactly the same as a 1-ply search.

The computer was using an incredibly flawed idea about the state of the game at the edge of its search horizon to evaluate the entire game tree. This is where the quiescent search comes in.

Intelligence Improvement #1: Quiescent Search

When board evaluations are changing drastically from move to move (as is the case with captures), the program must search a little further to find a quiet or "quiescent" position. After reaching the bottom of the regular search, PaulChess searches captures only for up to a specified maximum additional depth. **The quiescent search turned out to be the biggest improvement to PaulChess's intelligence over the entire course of its development**.

The difference to its search horizon was immediately noticeable (that is, after I worked out plenty of bugs). Instead of making some wild capture after the 4-ply search, it would instead make an intelligent capture if it found one within the quiescent limit, or it would make a quiet move, having decided that none of the captures turns out to be a smart move.

Although it is difficult for me to say exactly how much the quiescent search helped, I have a basic idea. Before, it was losing to decent online computer chess programs, such as Shredder easy/normal, Thinking Machine 4, The Chess Machine, etc. Afterwards, it was able to beat these programs most of the time. Also, I could personally defeat PaulChess before adding the quiescent search, but after, it would beat me. In fact, I have never actually won a full game against it since adding the quiescent search.

Special tip: it is possible to store quiescent move data in the transposition table for a little speed boost. The program can also use null-move pruning within the quiescent search for a significant

speed boost. I came up with both of these ideas a bit later, and was very happy with the results. These two things combined allowed me to go from a 4-ply search with a max quiescent depth of four to a 4-ply search with a max quiescent depth of six, which is where the program stands right now.

PaulChess 1.0 uses a straight 4-ply search with a quiescent search of up to six plies deeper, combining for a maximum total of ten plies on lines with a lot of captures. Once again, Larameé's article was my primary resource for quiescent search: http://www.gamedev.net/reference/programming/features/chess5/page2.asp

Intelligence Improvement #2: Fixing Long-Overlooked Bugs

By this time I had added a fair amount of evaluation considerations to PaulChess. Normally, when I add something to the evaluation, I test the new feature in isolation (i.e., that the game correctly detects when the situation has occurred for the bonus or penalty). I then assign that feature a point value and put it into the overall evaluation. Sometimes, when I am in a hurry, I do a little bit less testing than that, assuming that a new feature works unless the computer really starts to play poorly.

Both of these tactics failed to produce accurate code on a couple of occasions. But that was not a huge surprise—of course there are going to be bugs once in a while. **What did surprise me was how long it would take me to notice those bugs**.

Here is an example. I programmed in a penalty for isolated pawns. I tested that the program correctly identified isolated pawns, assigned it a value, and assumed that it was working. Over time (a long time), I started to realize that it was isolating its pawns more than it should based on the penalty that I was giving it. Of course, given that the computer is doing a complex search with hundreds of thousands of positions, it is really hard to tell exactly what it *should* be doing based on its evaluation, since a human cannot search the game tree like it does.

Still, I decided to double check the isolated pawn code just in case. It turned out that I was actually giving a BONUS to isolated pawns, and had been doing so for several months! I really kicked myself for that one.

I found a different way to test that an evaluation element is working correctly. **I set up a position in which I have seen my program make a mistake** (for example, it makes a trade that isolates its pawns when it does not have to). **Then, I add the evaluation element and see if it makes the right move**.

If the program still makes the wrong move, I would know that following could be the problem:

- My evaluation element still has a bug and is not being detected correctly.
- My evaluation element is adding too small of a bonus or penalty compared to other considerations. In this case, I would need to make the bonus or penalty more extreme (either that, or adjust the other evaluations).

The key is that I needed all of my evaluation considerations to be scaled correctly based on how important they are compared to material. The material evaluation should be the baseline against which every other evaluation is scaled.

Intelligence Improvement #3: Miscellaneous Additions in the Last Frantic Push

At this point, the second semester was well over half-way finished, and I knew that if I ever wanted Levon to test the program, it would have to be soon. I tried a lot of ideas to improve PaulChess's game, keeping what worked and tossing what did not. Here is a summary of the strategies that I remember trying:

What worked:

- Adding a bonus for castling.
- Adding late game evaluations so that the computer can find a checkmate with just a rook or just a queen.
- Coding in the draw by repetition rule. The computer knows to avoid the draw if it is winning, and seek it if it is losing. This had the added benefit of keeping my program from playing in an infinitely loop.
- Various other tweaks, bug fixes, and small additions to the evaluation function.

What did not work:

- Singular extensions on check. The idea is that when one side is in check, it only has a couple of available moves, and the response to a check can change the game quite a bit. For some reason I could not get this to improve the computer's play, so I left it out of the final version. Now that I think about it, it would actually require looking two moves ahead of a check to notice important problems like royal forks.
- Dynamic search plies. In the end game and when in check, the game tree has far fewer positions to search, and the search does not take as long. In these situations, I figured that the computer should adjust its search and look even deeper to find a better move. Essentially, the computer would predict how long it would take to search deeper based on its previous passes of iterative deepening, and if its prediction was shorter than about thirty seconds (including the time that it had already been searching), it would perform another pass of iterative deepening. Unfortunately, I could never quite get it to predict the time of the next move correctly, no matter how many ideas I tried.

Around this same time I worked hard to add more options to the interface, such as the ability to reverse the board, turn on and off move information, control AI options, etc.

The Payoff

After working frantically during the previous couple of weeks, it was finally time to show the program to Levon and get his estimate of its strength. It was also time to begin writing this

report. But before giving away the results of its many games and my final thoughts on the project, here is a summary of what I have put in to the program.

PaulChess Overview

Move Generation:

- Board copying for move generation (the move is not "made" and "unmade" on the same board, as some programs do)
- Standard bitboard representation, single rotation
- Lookup tables for sliding pieces: row, column, leading, and trailing diagonals
- "Byte sieve" strategy for locating 1-bit positions
- "LastMove" object is passed to the new board, providing all of the relevant data about the last board and how it changed.

Evaluation Function:

- Standard material evaluation (pawn = 100, knight = 325, bishop = 330, rook = 500, queen = 900)
- Mobility bonus per piece, weighted toward the middle and late game
- Mobility bonus per square, weighted toward the middle and late game
- Doubled pawn penalty
- Isolated pawn penalty, weighted more heavily near the center files
- Blocked center pawn penalty (applies only when the pawn has not moved from its starting location)
- Pawn advancement bonus, weighted toward the middle and end game
- Piece-square bonuses in the opening phase for the following: advanced center pawns, common opening pawn moves, knights near the center, fianchetto and Italian bishops, and any piece that reaches the center
- Piece-square penalties in the opening phase for the following: odd pawn moves, knights on the edge of the board, bishops on the edge of the board, center pawns that have not moved, aggressive queen moves
- Castling bonus, weighted toward the early game
- Penalty for losing the ability to castle on either side (except when already receiving the castling bonus), weighted toward the early game
- Bonus for pawns near the opposing king
- Penalty for a king that is not adjacent to any friendly pawns
- Late game bonus for a king near other pieces
- Very late game bonus for moving the king toward the enemy king
- Very late game penalty for an enemy king that is near the center of the board

Tree Search Techniques:

- Standard 4-ply alpha beta search
- Quiescent search of captures, up to six additional plies
- Null-move heuristic for both the standard search and the quiescent search
- 2-level transposition table, following the 2-Deep scheme

- Two-phase move generation (captures and non-captures), ordered by static evaluation and the transposition table's suggestion (when applicable)

Opening Book:

- Created by hand, manually selected from an opening database
- Currently 219 moves
- White always opens with d4, favors the queen's gambit.
- Against e4 and d4, black always plays 1. e4 c4 and 1. d4 Nf6, respectively.
- The opening book can be turned on and off for any AI

Interface Features:

- Edit Board mode, which includes castling flags
- Auto position setup for Levon's test positions (hidden, use number keys in edit mode)
- Edit Players (and AI) mode
- AI options (all of these can be turned on and off): search depth, iterative deepening, transposition table, alpha beta pruning, dynamic plies, null move heuristic, opening book, singular extension, null move heuristic for quiescent search, max quiescent search depth
- Toggle reporting the computer's move time
- Ability to select different AIs, including Simple1 and PaulChess 1.0
- Write the history to a file
- Add positions to the opening book, write the opening book
- Game history is displayed
- Move backward and forward in the current game's history
- Explore the computer's projected line of play (hidden, use number keys with caps)
- Reverse the board
- Toggle music for the thinking computer
- Toggle "autoplay" for the computer (when off, it will only move when told to)
- Toggle the draw by repetition rule
- Full promotion options are available

Selection of Matches

While developing PaulChess, I tested it against other programs constantly. To give an idea of its strength, here are the results of its matches against other programs (using PaulChess version 1.0).

It is important to note that PaulChess takes approximately 20-30 seconds to move in an average middle game position, and most of these programs move faster. However, hardware differences and/or programming language choice can greatly affect the speed of a program, so it is really impossible to control for every variable.

For these reasons, this list of results is **not** intended to be a definite indication of which program is stronger. I am including it to give readers a *basic* idea of PaulChess's intelligence (played with normal parameters) compared to control programs.

Note: “WIN” means that PaulChess won, regardless of which color it was playing. All of these games are included in their entirety as Appendix I.

Excalibur Deluxe Talking Chess
www.ExcaliburElectronics.com
This is the same AI used in all portable Excalibur Chess devices. Reported to have a rating of 2000 when playing at full strength.

1. Hard (1 second) vs. PaulChess — DRAW (repetition) in 37 moves
2. PaulChess vs. Hard (2 seconds) — WIN in 82 moves
3. Hard (2 seconds) vs. PaulChess — LOSS in 88 moves

Shredder Computer Chess Online
http://www.shredderchess.com/play-chess-online.html
Shredder has a free online version of its program that is not as strong as the full version. It has three difficulty levels: easy, medium, and hard. These difficulty levels are supposed to be approximately as strong as equivalent human players.

4. PaulChess vs. Easy — WIN in 49 moves
5. Normal vs. PaulChess — WIN in 55 moves
6. PaulChess vs. Normal — LOSS in 75 moves
7. Hard vs. PaulChess — LOSS in 59 moves

The Computer Chess MACHINE 1.36
http://www.postcardchess.com/computerchess.html
This is a program that has evolved over 20 years. Its most current version is written in C++ and runs on a fast server. Its previous version was the Newton ChessPro program, running on the handheld Newton device.

8. PaulChess vs. Chess MACHINE — WIN in 71 moves
9. Chess MACHINE vs. PaulChess — DRAW (repetition) in 45 moves

Little ChessPartner
http://www.chessgames.com/perl/engine
This is a free online version of the much stronger ChessPartner program. It is written in Java, and is reported to have a 2000 ELO rating.

10. PaulChess vs. Little ChessPartner (1 second) — LOSS in 97 moves
11. Little ChessPartner (1 second) vs. PaulChess — DRAW (repetition) in 66 moves

Thinking Machine 4
http://www.turbulence.org/spotlight/thinking/chess.html
This program is written in Java and is designed to be as strong as the average opponent. It has been in development since 2002, and uses alpha/beta pruning with a quiescent search.

12. PaulChess vs. Thinking Machine 4 — WIN in 97 moves

(It is not possible to play as black against Thinking Machine 4)

GNU Chess 4.0
http://www.gnu.org/software/chess/
As I have mentioned, this program inspired many of the AI concepts and search techniques used in PaulChess. It is a very strong open source program with a lot of customizability, written in C.

13. PaulChess vs. GNU Chess 4.0 (1 second)	LOSS in 19 moves
14. GNU Chess 4.0 (1 second) vs. PaulChess	LOSS in 49 moves

Personal Assessment by Levon Altounian

Levon was the co-teacher of the Chess and AI class that I took in the fall of 2007. He is an International Master, and has worked evaluating computer chess programs. He also has experience catching people who cheat online using chess programs.

To be honest, I was terrified to have Levon take a look at the program. I knew that he could clobber it effortlessly if he played against it to win—that was never the question. I was just hoping that my program would have a decent rating, and have shown at least some improvement.

First he played against Simple1, which was version 0.1 of PaulChess, completed in approximately October 2008. It played terribly in the opening, moving its queen early and often. The middle game was quite a bit better due to its strong tactics, but it once again fell apart in the end game.

Levon's assessment of PaulChess 0.1 (called Simple1 in the interface):
1500-1600 middle game, 1100 end game

Then he played against PaulChess 1.0, completed late March. It played much more conventional openings, and a more solid (but conservative) middle game. I was amazed by how quickly Levon was able to find weaknesses in the program that I would never have been able to see. In a way, it was tough to watch how easily he could cause PaulChess to make mistakes, due to his incredible skill at the game and his extensive experience with other chess programs. Still, it was exciting when the PaulChess would make a good move, and ultimately it was a lot of fun to meet with Levon and learn from his evaluation.

Levon's assessment of PaulChess 1.0:
1800 middle game

How much did PaulChess improve over about five months? About 250 points in the middle game, and (I am guessing) about the same amount of improvement in the end game. Unfortunately, Levon and I did not have time to take PaulChess 1.0 to an end game.

As tough as it was to watch Levon play the program, I am so grateful that he was generous enough to meet with me. It was the highlight of the project, and the perfect way to end my work on PaulChess 1.0.

Ideas for PaulChess 2.0

As Levon and other programs have pointed out, PaulChess still has a long ways to go before it will be a truly competitive chess program. Here are some of the ways in which I believe that PaulChess could improve, given more time:

Speed Improvements:

- Additional search techniques, such as Aspiration Search
- Magic bitboards for sliding piece lookups
- Rewrite the program in C
- Figure out how to use transposition tables and the null-move heuristic *correctly*

Intelligence Improvements:

- King safety should be a bigger concern, especially in the middle game
- Improve the castling evaluation so that the computer only castles when it is safe
- Many end game considerations, especially what to do with kings and pawns
- Balance adjustments for existing evaluations (e.g., early king movement penalty vs. doubled pawn penalty)
- Improve the early queen evaluation so that it is more willing to move out when it is safe
- Expand the opening book, especially for the Sicilian opening
- Add an endgame tablebase

Concluding Thoughts

I can summarize almost all of PaulChess's strengths and weaknesses in one sentence:
It plays like I do.

From the very beginning, my goal was to create a chess program that could beat me consistently, and PaulChess 1.0 can. As Levon pointed out, under certain circumstances it still makes mistakes that a human would be unlikely to make. I have mentioned a few of these in the previous section. However, on the whole, I was able to teach it to think how I think. The main difference is that (like most programs) PaulChess is not good at developing long term strategies, but has a tactical advantage when it notices that a move is good within its search window.

Its biggest advantage when playing against me is that it does not get tired, it does not get distracted, it does not get frustrated, it never rushes to make a move, and it is never overconfident. All of these strengths, combined with its tactical advantage, are enough to outweigh my strategic edge in most situations.

I am proud that it plays like I do, even though it shares some of my weaknesses. When I think back to all of the countless hours that I spent testing and improving it, all of the adjustments that

I made to its evaluation function, and all of the search enhancements that I worked so hard on… ultimately, it was all aimed at making the program play more like I do, since that is the only style of play that I have the tools to teach it properly.

I wish that PaulChess could search 8-ply with a 16-ply quiescent search, instead of 4-ply with a 6-ply quiescent search. I wish that it knew how to play better with just a king and pawns. I wish that it could beat GNU Chess. But that is one great thing about PaulChess—it does not have to end here. I have started a project that I can continue to work on for the rest of my life. PaulChess 1.0 is only the very first serious version of the program, and I sure hope that it is not the last.

No matter how well it plays, PaulChess is my own creation, and I will always be grateful that I had the opportunity to work on it.

The story of PaulChess would not be complete until I thank Prof. Sandiway Fong and Levon Altounian for making this whole project possible.

PaulChess is dedicated to my father, Fred Stevens, who introduced me to chess at a young age and helped me cultivate a lifelong appreciation for the game.

Appendix I: Full Games

All of these games were mentioned in the "selected games" section. I have included the entire games here for a couple of reasons. First, as PaulChess improves in the future, I can evaluate its progress by having it play against these same programs and comparing the results. Second, a reader that is interested in how PaulChess *really* plays can get a pretty good idea from taking a look at these games.

All of the games listed here were played by PaulChess 1.0. They represent only a tiny fraction of the games that PaulChess has played in its lifetime, which probably number in the hundreds.

White: Computer (PaulChess3, 4/6 plies)
Black: Excalibur Deluxe Touch Chess (Fixed Time Hard, 2 seconds)

```
History:
1. d2d4 d7d5 2. c2c4 e7e6 3. Nb1c3 Ng8f6 4. Bc1g5 Bf8e7 5. Bg5:f6 Be7:f6 6.
e2e3 O-O 7. Ng1f3 d5:c4 8. Bf1:c4 Nb8d7 9. e3e4 Nd7b6 10. Bc4b3 Bc8d7 11. O-O
Bd7e8 12. e4e5 Bf6e7 13. Qd1c2 Nb6d5 14. Bb3:d5 e6:d5 15. a2a3 Be8c6 16. h2h3
Kg8h8 17. Nc3e2 f7f5 18. Ne2f4 Qd8d7 19. Rf1e1 Qd7c8 20. Qc2b3 Rf8d8 21.
Qb3c2 Bc6b5 22. a3a4 Bb5a6 23. Re1c1 c7c6 24. Rc1e1 Be7b4 25. Re1e3 b7b6 26.
Re3b3 Bb4e7 27. Rb3c3 c6c5 28. d4:c5 d5d4 29. Rc3b3 b6:c5 30. Ra1c1 c5c4 31.
Qc2d2 Be7c5 32. Qd2a5 c4:b3 33. Rc1:c5 Qc8b7 34. Nf4e6 Rd8d7 35. Ne6c7 Rd7:c7
36. Rc5:c7 Qb7b8 37. Nf3:d4 Ba6d3 38. Qa5d5 Bd3e4 39. Qd5f7 Qb8g8 40. Qf7:b3
Qg8:b3 41. Nd4:b3 Ra8b8 42. Nb3d2 Be4d5 43. Rc7d7 Bd5e6 44. Rd7:a7 Rb8:b2 45.
Ra7a8+ Be6g8 46. Ra8d8 Rb2a2 47. Rd8d4 Ra2a1+ 48. Nd2f1 Ra1a3 49. Nf1e3
Ra3a1+ 50. Ne3d1 Bg8b3 51. Rd4d8+ Bb3g8 52. Rd8d4 Bg8b3 53. Rd4d8+ Bb3g8 54.
e5e6 Ra1:d1+ 55. Rd8:d1 Bg8:e6 56. Rd1d8+ Be6g8 57. g2g4 f5:g4 58. h3:g4 g7g6
59. Kg1g2 h7h5 60. g4:h5 g6:h5 61. Rd8d7 Bg8e6 62. Rd7d6 Be6b3 63. a4a5 Kh8g7
64. a5a6 Bb3c2 65. a6a7 Bc2e4+ 66. f2f3 Be4a8 67. Rd6d8 Ba8c6 68. a7a8=Q
Bc6:a8 69. Rd8:a8 Kg7f6 70. Ra8a5 Kf6g6 71. Kg2g3 Kg6f6 72. Ra5:h5 Kf6g6 73.
Rh5e5 Kg6h6 74. Kg3g4 Kh6g6 75. Re5e1 Kg6g7 76. Kg4g5 Kg7f7 77. f3f4 Kf7g7
78. f4f5 Kg7f7 79. f5f6 Kf7g8 80. Kg5g6 Kg8f8 81. Re1e4 Kf8g8 82. 1-0
```

This was a very good game the whole way through. PaulChess was up two pawns by move 54, and due to a bishop pin and a passed pawn, forced Excalibur to throw its rook for a knight, giving PaulChess enough of a lead to promote pawns for a win.

White: Exaclibur Deluxe Talking Chess (Fixed Time Hard, 2 seconds)
Black: Computer (PaulChess3, 4/6 plies)

```
History:
1. d2d4 Ng8f6 2. Nb1c3 d7d5 3. Bc1f4 e7e6 4. Nc3b5 Bf8b4+ 5. c2c3 Bb4a5 6.
Ng1f3 a7a6 7. Nb5a3 Qd8e7 8. Qd1a4+ Nb8c6 9. Nf3e5 Ba5b6 10. Ne5:c6 Bc8d7 11.
Nc6:e7 Bd7:a4 12. Ne7:d5 Nf6:d5 13. Bf4g5 h7h6 14. Bg5h4 g7g5 15. Bh4g3 O-O
16. Na3c4 Ba4c2 17. Nc4:b6 c7:b6 18. Ke1d2 Bc2g6 19. Bg3d6 Rf8d8 20. Bd6a3
b6b5 21. h2h4 g5g4 22. h4h5 Bg6f5 23. f2f3 g4:f3 24. g2:f3 b5b4 25. Rh1g1+
Kg8h7 26. c3:b4 Nd5f4 27. e2e4 Bf5h3 28. Kd2e3 Bh3:f1 29. Ke3:f4 Bf1b5 30.
Kf4e3 Ra8c8 31. Rg1g2 Rd8g8 32. Rg2:g8 Rc8:g8 33. Ra1c1 Rg8g5 34. Rc1c7 Kh7g8
35. Rc7:b7 Rg5:h5 36. Rb7b6 Rh5h1 37. Rb6b8+ Kg8g7 38. Ke3f4 Rh1c1 39. Kf4e5
Rc1f1 40. f3f4 h6h5 41. Rb8c8 Rf1a1 42. b2b3 Ra1:a2 43. Ba3c1 Bb5d7 44. Rc8d8
Ra2c2 45. Rd8:d7 Rc2:c1 46. Rd7b7 Rc1c3 47. b4b5 a6a5 48. Rb7a7 Rc3:b3 49.
Ra7:a5 h5h4 50. f4f5 e6:f5 51. e4:f5 Rb3e3+ 52. Ke5d6 h4h3 53. Ra5a2 Kg7f6
```

```
54. Ra2f2 Kf6g5 55. b5b6 f7f6 56. b6b7 Re3b3 57. Kd6c7 Rb3c3+ 58. Kc7d7 Rc3b3
59. Kd7c6 Rb3b1 60. d4d5 Rb1c1+ 61. Kc6b6 Rc1b1+ 62. Kb6c7 Rb1c1+ 63. Kc7d6
Rc1b1 64. Kd6c6 Rb1c1+ 65. Kc6d7 Rc1b1 66. Kd7c8 Rb1c1+ 67. Kc8d8 Rc1b1 68.
Kd8c7 Rb1c1+ 69. Kc7b8 Rc1c5 70. Kb8a7 Rc5b5 71. b7b8=Q Rb5:b8 72. Ka7:b8
Kg5g4 73. d5d6 h3h2 74. Rf2:h2 Kg4g3 75. Rh2h8 Kg3g4 76. d6d7 Kg4:f5 77.
d7d8=Q Kf5e5 78. Rh8f8 f6f5 79. Qd8a5+ Ke5e4 80. Qa5:f5+ Ke4d4 81. Kb8c7
Kd4e3 82. Kc7d6 Ke3e2 83. Kd6e5 Ke2d2 84. Ke5d4 Kd2d1 85. Qf5d3+ Kd1c1 86.
Rf8f1+ Kc1b2 87. Qd3c3+ Kb2a2 88. 1-0
```

A very close game that almost ended in a draw around move 60. However, Excalibur was up an advanced pawn by the end game, which promoted, forcing PaulChess to give up a rook. This marked the end of the game.

White: Computer (PaulChess3, 4/6 plies)
Black: Exaclibur Deluxe Talking Chess (Fixed Time Hard, 1 second)

```
History:
1. d2d4 d7d5 2. c2c4 c7c6 3. Nb1c3 Ng8f6 4. Ng1f3 d5:c4 5. e2e4 Bc8e6 6.
Bf1e2 Nb8d7 7. Nf3g5 Nd7b6 8. Ng5:e6 f7:e6 9. e4e5 Nf6d5 10. Nc3e4 Ra8c8 11.
Ne4c5 Nb6d7 12. Nc5:b7 Qd8b6 13. Nb7c5 Nd7:c5 14. d4:c5 Qb6:c5 15. O-O Rc8b8
16. Be2g4 c4c3 17. b2b3 c3c2 18. Qd1d2 Nd5c7 19. Rf1e1 Rb8b4 20. Bg4f3 Nc7d5
21. Bf3e4 Rb4:e4 22. Re1:e4 Nd5c3 23. Re4c4 Qc5:e5 24. Qd2:c3 Qe5:c3 25.
Rc4:c3 g7g6 26. Rc3:c6 Bf8g7 27. Rc6c8+ Ke8d7 28. Rc8:c2 Bg7:a1 29. Bc1e3
Rh8a8 30. Rc2d2+ Kd7c6 31. Rd2c2+ Kc6d7 32. f2f3 h7h5 33. Rc2d2+ Kd7c6 34.
Rd2c2+ Kc6d7 35. Rc2d2+ Kd7c6 36. Rd2c2+ Kc6d7 37. a2a4 1/2-1/2
```

This game ended in a draw by repetition, although PaulChess made an unusual error in failing to recognize the draw, assuming that it was going to occur on the next move. PaulChess had a small positional advantage, and was not actively pursuing the draw.

White: Shredder Online (Hard)
Black: Computer (PaulChess3, 4/6 plies)

```
History:
1. e2e4 c7c5 2. Nb1c3 e7e5 3. Bf1c4 Ng8f6 4. Ng1f3 Nb8c6 5. Nf3g5 d7d5 6.
e4:d5 Nc6a5 7. Bc4b5+ Bc8d7 8. Qd1e2 Bf8d6 9. Ng5e4 Nf6:e4 10. Nc3:e4 Bd6f8
11. d5d6 b7b6 12. Ne4:c5 b6:c5 13. Qe2:e5+ Bf8e7 14. d6:e7 Qd8:e7 15. Bb5:d7+
Ke8:d7 16. Qe5:e7+ Kd7:e7 17. d2d3 Ra8b8 18. Bc1e3 Rb8:b2 19. Be3:c5+ Ke7e6
20. Ke1d2 Na5c6 21. Rh1e1+ Ke6d5 22. Bc5a3 Rb2b4 23. Ba3:b4 Nc6:b4 24. Re1e7
Rh8c8 25. c2c3 Nb4a6 26. Ra1b1 Rc8c7 27. Re7:c7 Na6:c7 28. Rb1b7 Kd5d6 29.
Rb7:a7 f7f5 30. Ra7:c7 Kd6:c7 31. d3d4 f5f4 32. c3c4 h7h5 33. h2h4 g7g6 34.
Kd2e2 Kc7b6 35. Ke2f3 Kb6c6 36. Kf3:f4 Kc6c7 37. Kf4e5 Kc7b6 38. c4c5+ Kb6c6
39. d4d5+ Kc6:c5 40. d5d6 Kc5c6 41. Ke5e6 Kc6b6 42. d6d7 Kb6c7 43. Ke6e7
Kc7b6 44. d7d8=Q+ Kb6c5 45. Qd8d6+ Kc5c4 46. Qd6:g6 Kc4d5 47. Qg6:h5+ Kd5d4
48. Qh5d1+ Kd4c5 49. h4h5 Kc5c4 50. g2g4 Kc4c5 51. Ke7e6 Kc5b4 52. g4g5 Kb4c3
53. f2f4 Kc3c4 54. g5g6 Kc4c3 55. g6g7 Kc3c4 56. Qd1d5+ Kc4b4 57. g7g8=Q
Kb4c3 58. Qg8g3+ Kc3b2 59. Qg3a3+ Kb2:a3
```

Shredder was only up by two pawns by move 22, which made PaulChess's idea to trade a rook for a bishop somewhat puzzling. PaulChess was losing three pawns to six by the end, and made little effort to defend against promotion.

White: Computer (PaulChess3, 4/6 plies)
Black: Shredder Online (Easy)

```
History:
1. d2d4 d7d5 2. c2c4 e7e6 3. Nb1c3 Ng8f6 4. Bc1g5 Bf8e7 5. Bg5:f6 Be7:f6 6.
e2e3 O-O 7. Ng1f3 d5:c4 8. Bf1:c4 Nb8c6 9. O-O Bc8d7 10. e3e4 Qd8e7 11. e4e5
Qe7b4 12. Qd1e2 Bf6e7 13. a2a3 Qb4b6 14. Qe2d2 a7a6 15. h2h3 f7f5 16. Nc3a4
Qb6a7 17. Na4c3 Ra8d8 18. Rf1e1 Qa7b6 19. Nc3a4 Nc6:d4 20. Na4:b6 Nd4:f3+ 21.
g2:f3 c7:b6 22. b2b4 Kg8h8 23. Qd2e3 Rd8c8 24. Qe3d2 Rf8d8 25. Bc4a2 Bd7b5
26. Qd2e3 Rc8c6 27. Qe3f4 Rd8d3 28. Ba2b1 Rc6c4 29. Qf4g3 Rd3d2 30. f3f4
Rc4d4 31. a3a4 Bb5c6 32. b4b5 a6:b5 33. a4:b5 Bc6:b5 34. Qg3c3 Be7h4 35.
Qc3c8+ Rd4d8 36. Qc8c1 Rd2:f2 37. Qc1e3 Rf2d2 38. Re1c1 Rd8d4 39. Ra1a8+
Bh4d8 40. Qe3b3 Bb5c6 41. Qb3:e6 Rd4:f4 42. Bb1:f5 Rf4f2 43. Ra8:d8+ Rd2:d8
44. Kg1:f2 Rd8f8 45. Kf2e3 b6b5 46. Bf5e4 Bc6:e4 47. Rc1c8 g7g5 48. Rc8:f8+
Kh8g7 49. 1-0
```

Shredder made a queen blunder on move 20, ending in a very lopsided trade. After that, it was just a matter of time before PaulChess was able to break past its defense.

White: Computer (PaulChess3, 4/6 plies)
Black: Shredder Online (Normal)

```
History:
1. d2d4 e7e6 2. e2e4 d7d5 3. Nb1c3 Bf8b4 4. e4e5 c7c5 5. Bf1b5+ Bc8d7 6.
Bb5:d7+ Nb8:d7 7. Ng1e2 Qd8h4 8. Bc1e3 Ng8e7 9. O-O c5:d4 10. Be3:d4 Bb4:c3
11. Ne2:c3 Ne7g6 12. g2g3 Qh4h3 13. Qd1e2 O-O 14. f2f4 Ng6e7 15. a2a3 Rf8c8
16. Rf1e1 Rc8c4 17. Nc3b5 Ra8c8 18. c2c3 a7a6 19. Nb5d6 b7b5 20. Nd6:c4 b5:c4
21. Bd4e3 Rc8b8 22. Be3a7 Rb8b7 23. Ba7f2 Rb7b8 24. Bf2a7 Rb8b7 25. Ba7f2
Rb7b8 26. Re1d1 a6a5 27. Bf2a7 Rb8b7 28. Ba7f2 Rb7b8 29. Bf2a7 Rb8b7 30.
Ba7e3 Rb7b8 31. Be3f2 Rb8b7 32. a3a4 Rb7b3 33. Bf2a7 Rb3b7 34. Ba7f2 Rb7b3
35. Bf2e3 Ne7f5 36. Be3a7 Nf5:g3 37. h2:g3 Qh3:g3+ 38. Kg1h1 Qg3:f4 39. Kh1g1
Nd7:e5 40. Ra1a2 Ne5f3+ 41. Kg1g2 Nf3h4+ 42. Kg2h3 Nh4f5 43. Ba7f2 Rb3b8 44.
Kh3g2 Rb8b3 45. Kg2g1 Qf4g5+ 46. Kg1h2 Qg5h6+ 47. Kh2g1 Qh6g5+ 48. Kg1h2
Qg5f4+ 49. Kh2g2 Rb3b8 50. Ra2a3 Rb8b7 51. Kg2g1 Rb7b8 52. Kg1g2 h7h5 53.
Kg2g1 Rb8:b2 54. Qe2:b2 Qf4g4+ 55. Kg1h2 Qg4:d1 56. Qb2b8+ Kg8h7 57. Kh2g2
Qd1g4+ 58. Kg2f1 Qg4d1+ 59. Kf1g2 g7g5 60. Qb8d8 Qd1g4+ 61. Kg2h2 Qg4f4+ 62.
Kh2g2 h5h4 63. Ra3a2 Qf4g4+ 64. Kg2f1 Qg4h3+ 65. Kf1g1 g5g4 66. Qd8:a5 Qh3f3
67. Qa5d8 h4h3 68. Bf2e1 g4g3 69. Qd8b8 Qf3e3+ 70. Ra2f2 h3h2+ 71. Kg1g2
Nf5h4+ 72. Kg2h3 h2h1=Q+ 73. Kh3g4 f7f5+ 74. Rf2:f5 e6:f5+ 75. Kg4h5 0-1
```

A close game with a lot of very redundant moves by both sides, but never a draw by repetition. Shredder made a mistake, trading a rook for a minor piece, but PaulChess left its defenses far too open and allowed pawns to advance. The two pawns attacked PaulChess's king and caused so many problems that one eventually promoted, turning the game in Shredder's favor.

White: Shredder Online (Normal)
Black: Computer (PaulChess3, 4/6 plies)

```
History:
1. e2e4 c7c5 2. Ng1f3 d7d6 3. Bf1c4 e7e5 4. O-O Bc8g4 5. h2h3 Bg4:f3 6.
Qd1:f3 Ng8f6 7. Nb1c3 Nb8c6 8. d2d3 Bf8e7 9. Qf3g3 O-O 10. Bc1h6 Nf6e8 11.
a2a3 Nc6d4 12. Ra1c1 Be7h4 13. Qg3g4 Bh4e7 14. Bc4d5 Qd8c7 15. a3a4 a7a6 16.
Bd5a2 Qc7d8 17. Nc3d5 Be7h4 18. c2c3 Nd4c6 19. Ba2b3 Nc6a5 20. Bb3a2 Na5c6
```

```
21. Ba2b3 Ra8a7 22. g2g3 Nc6a5 23. Bb3a2 Bh4f6 24. b2b4 c5:b4 25. c3:b4 Na5c6
26. Bh6e3 Ra7a8 27. Be3b6 Qd8b8 28. Qg4f5 Bf6d8 29. Bb6e3 g7g6 30. Qf5d7
Ne8f6 31. Nd5:f6+ Bd8:f6 32. Rc1b1 Bf6e7 33. Rf1c1 Rf8d8 34. Qd7g4 Qb8c7 35.
Qg4f3 Be7f8 36. Ba2d5 Bf8g7 37. Be3g5 Rd8e8 38. b4b5 a6:b5 39. a4:b5 Nc6d4
40. Qf3g4 Qc7b6 41. Rb1a1 Ra8:a1 42. Rc1:a1 Qb6:b5 43. Ra1a7 Qb5:d3 44. Qg4d7
Nd4f3+ 45. Kg1g2 Nf3e1+ 46. Kg2h2 Ne1f3+ 47. Kh2g2 Re8f8 48. Bg5e3 Nf3e1+ 49.
Kg2h2 Qd3f1 50. Qd7:f7+ Rf8:f7 51. Bd5:f7+ Kg8:f7 52. Ra7:b7+ Kf7e6 53.
Rb7e7+ Ke6:e7 54. Be3g5+ Ke7d7 55. Bg5e3 0-1
```

The sides had even material until move 42. PaulChess then managed to win some pawns when Shredder was unable to crack the defenses by PaulChess's castled king. Both sides put up strong attacks on each king's position, but PaulChess found a mate first.

White: Computer Chess MACHINE 1.36
Black: Computer (PaulChess3, 4/6 plies)

```
History:
1. f2f4 d7d5 2. Ng1f3 Ng8f6 3. e2e3 Bc8g4 4. Bf1d3 Bg4:f3 5. Qd1:f3 e7e6 6.
O-O Nb8d7 7. b2b3 Bf8c5 8. Bc1b2 O-O 9. Nb1c3 a7a6 10. a2a4 Nd7b8 11. g2g3
Nb8c6 12. Kg1h1 Nc6b4 13. Qf3e2 Nb4:d3 14. Qe2:d3 d5d4 15. Nc3e2 Qd8d5+ 16.
Kh1g1 d4:e3 17. Qd3:d5 Nf6:d5 18. d2d3 f7f5 19. Kg1g2 h7h6 20. Bb2e5 Bc5d6
21. Rf1h1 Bd6:e5 22. f4:e5 Rf8d8 23. Kg2f3 Nd5b4 24. Ra1c1 Rd8d5 25. d3d4
Nb4a2 26. Rc1a1 Na2b4 27. c2c3 Nb4d3 28. Kf3:e3 Nd3:e5 29. d4:e5 Rd5:e5+ 30.
Ke3f2 Ra8d8 31. Ne2d4 c7c5 32. Nd4e2 Rd8d2 33. Rh1e1 Re5e4 34. c3c4 b7b6 35.
Ra1b1 h6h5 36. Rb1a1 h5h4 37. Ra1c1 h4h3 38. a4a5 b6:a5 39. Kf2f1 e6e5 40.
Rc1a1 Rd2d3 41. Re1b1 Rd3d2 42. Ne2c3 Re4g4 43. Nc3e2 Rg4e4 44. Ne2c3 Re4g4
45. Nc3e2 1/2-1/2
```

Almost even the entire game. PaulChess traded a knight for two pawns in the middle game, which did not end up being too much of a problem. Unfortunately, PaulChess took a bad pawn sacrifice end game, which eventually led to a standoff. It sought and obtained a draw by repetition.

White: Computer (PaulChess3, 4/6 plies)
Black: Computer Chess MACHINE 1.36

```
History:
1. d2d4 Ng8f6 2. c2c4 g7g6 3. Nb1c3 d7d5 4. c4:d5 Nf6:d5 5. Nc3:d5 Qd8:d5 6.
Ng1f3 Bf8g7 7. e2e3 Bc8g4 8. Bf1e2 O-O 9. Qd1c2 Qd5c6 10. Qc2d1 Bg4:f3 11.
Be2:f3 Qc6b5 12. Qd1e2 Qb5b4+ 13. Qe2d2 Qb4b5 14. Qd2e2 Qb5b4+ 15. Qe2d2
Qb4b5 16. a2a4 Qb5b3 17. Ra1a3 Qb3b6 18. O-O a7a6 19. a4a5 Qb6b5 20. Qd2e2
Qb5:e2 21. Bf3:e2 Nb8c6 22. Be2f3 Nc6d8 23. Bc1d2 c7c6 24. Bd2b4 Rf8e8 25.
Rf1c1 Nd8e6 26. h2h4 f7f5 27. g2g4 f5:g4 28. Bf3:g4 Ne6c7 29. h4h5 Nc7d5 30.
Bg4e6+ Kg8f8 31. Be6:d5 c6:d5 32. h5:g6 h7:g6 33. Rc1c7 Ra8b8 34. Ra3a1 Kf8f7
35. Ra1c1 Bg7f6 36. Rc7d7 Kf7e6 37. Rc1c7 b7b5 38. b2b3 Re8h8 39. Bb4:e7
Bf6:e7 40. Rd7:e7+ Ke6f6 41. Re7f7+ Kf6e6 42. Rf7e7+ Ke6d6 43. Re7d7+ Kd6e6
44. Rd7f7 b5b4 45. Rf7e7+ Ke6f6 46. Re7f7+ Kf6g5 47. Rc7c6 Rb8f8 48. f2f4+
Kg5h5 49. Rf7:f8 Rh8:f8 50. Rc6:a6 Rf8e8 51. Kg1f2 g6g5 52. f4:g5 Kh5:g5 53.
Ra6b6 Re8f8+ 54. Kf2e2 Kg5h5 55. Rb6:b4 Rf8f6 56. Rb4b5 Rf6d6 57. e3e4 Kh5g6
58. Rb5:d5 Rd6:d5 59. e4:d5 Kg6f5 60. Ke2f3 Kf5g6 61. d5d6 Kg6f6 62. d4d5
Kf6f7 63. a5a6 Kf7e8 64. a6a7 Ke8d7 65. a7a8=Q Kd7:d6 66. Qa8c6+ Kd6e5 67.
Qc6c5 Ke5f5 68. Qc5e7 Kf5g6 69. Kf3g4 Kg6h6 70. Kg4f5 Kh6h5 71. 1-0
```

PaulChess was up a pawn by the endgame, and had its rooks behind the Chess MACHINE's pawn line. The Chess MACHINE was forced to take the defensive, and by the end PaulChess was up four pawns, gaining a promotion and a checkmate.

White: Computer (PaulChess3, 4/6 plies)
Black: Little Chess Partner (1s)

```
History:
1. d2d4 Ng8f6 2. c2c4 Nb8c6 3. Ng1f3 e7e6 4. Bc1g5 Bf8b4+ 5. Nb1d2 O-O 6.
Bg5:f6 g7:f6 7. e2e4 d7d5 8. c4:d5 e6:d5 9. e4:d5 Nc6e7 10. a2a3 Bb4:d2+ 11.
Qd1:d2 Ne7:d5 12. Bf1b5 c7c6 13. Bb5e2 Bc8e6 14. O-O b7b5 15. Rf1c1 Qd8d6 16.
Be2d3 Nd5f4 17. Bd3e4 Be6d5 18. Be4:d5 c6:d5 19. Rc1c5 a7a6 20. Qd2c2 Ra8e8
21. Rc5c6 Nf4e2+ 22. Kg1f1 Qd6d8 23. Rc6:a6 Re8e4 24. Ra1d1 Ne2f4 25. Kf1g1
Nf4e2+ 26. Kg1h1 Ne2f4 27. Kh1g1 Nf4e2+ 28. Kg1h1 Ne2f4 29. g2g3 Re4e2 30.
Rd1d2 Re2:d2 31. Nf3:d2 Nf4e6 32. Nd2f3 Qd8c8 33. Ra6c6 Qc8a8 34. Kh1g1 h7h5
35. h2h3 Qa8a7 36. Qc2d1 Rf8d8 37. Qd1d2 Qa7a4 38. Qd2d3 Rd8a8 39. Rc6b6
Qa4c4 40. Qd3d1 Ra8c8 41. Rb6b7 Qc4c6 42. Rb7a7 Qc6c1 43. Qd1:c1 Rc8:c1+ 44.
Kg1g2 Rc1c4 45. Ra7a8+ Kg8g7 46. b2b3 Rc4c3 47. b3b4 Rc3d3 48. h3h4 f6f5 49.
Nf3e5 Rd3:d4 50. Ra8a7 Kg7f6 51. Ne5d7+ Kf6e7 52. Nd7e5+ Ke7f6 53. Ne5:f7
Rd4d3 54. Nf7d6 Ne6d4 55. Nd6e8+ Kf6g6 56. Ra7a6+ Kg6f7 57. Ne8d6+ Kf7e7 58.
Nd6c8+ Ke7d7 59. Nc8b6+ Kd7d6 60. Nb6c4+ Kd6c7 61. Nc4e5 Rd3c3 62. Ra6a7+
Kc7d6 63. f2f4 Rc3c2+ 64. Kg2f1 Rc2a2 65. Ra7a6+ Kd6c7 66. Ra6a7+ Kc7d6 67.
Ra7a8 Ra2a1+ 68. Kf1g2 Ra1a2+ 69. Kg2f1 Ra2a1+ 70. Kf1g2 Ra1a2+ 71. Kg2h1
Nd4e2 72. Ra8a6+ Kd6c7 73. Ra6c6+ Kc7b7 74. Rc6g6 Ra2:a3 75. Kh1g2 Ne2d4 76.
Rg6g7+ Kb7b6 77. Ne5d7+ Kb6c6 78. Nd7e5+ Kc6d6 79. Rg7d7+ Kd6e6 80. Rd7c7
Ra3a2+ 81. Kg2f1 Ra2a1+ 82. Kf1f2 Ra1a2+ 83. Kf2f1 Ra2a1+ 84. Kf1f2 Ra1a2+
85. Kf2e3 Nd4c2+ 86. Ke3f2 Nc2:b4+ 87. Kf2e3 Ra2a3+ 88. Ke3f2 d5d4 89. Rc7c8
Ra3a2+ 90. Kf2g1 d4d3 91. Rc8e8+ Ke6d5 92. Re8d8+ Kd5e4 93. Rd8c8 d3d2 94.
Rc8d8 Nb4d5 95. Rd8:d5 Ke4:d5 96. Kg1f2 d2d1=Q+ 97. Kf2e3 0-1
```

The game was totally even until move 86. LCP won a pawn which created a hole, allowing it to promote. PaulChess's endgame strategy was seriously lacking, or it may have been able to prolong the game. However, being down two pawns with only kings left, it is unlikely that PaulChess would have been able to win after about move 90.

White: Little Chess Partner (1s)
Black: Computer (PaulChess3, 4/6 plies)

```
History:
1. d2d4 Ng8f6 2. Nb1c3 d7d5 3. Bc1f4 e7e6 4. Ng1f3 Bf8b4 5. a2a3 Bb4:c3+ 6.
b2:c3 Nb8c6 7. e2e3 O-O 8. Bf1d3 b7b6 9. O-O Bc8b7 10. h2h4 h7h6 11. Qd1e1
Qd8e7 12. a3a4 Nf6h5 13. Bf4h2 Nh5f6 14. Bh2f4 a7a6 15. Ra1d1 Nf6h5 16. Bf4h2
Rf8d8 17. c3c4 Nh5f6 18. c4:d5 e6:d5 19. c2c4 d5:c4 20. Bd3:c4 Nc6a5 21.
Bc4d3 Bb7:f3 22. g2:f3 Na5c6 23. Qe1c3 Qe7d7 24. Rd1c1 Nc6:d4 25. Qc3:d4
Qd7:d4 26. e3:d4 Rd8:d4 27. Bd3c4 c7c6 28. f3f4 b6b5 29. a4:b5 a6:b5 30.
Bc4e2 Ra8a6 31. Be2f3 Nf6e4 32. Rc1a1 Rd4a4 33. Ra1:a4 Ra6:a4 34. Rf1c1 c6c5
35. Bf3:e4 Ra4:e4 36. Rc1:c5 Re4e1+ 37. Kg1g2 b5b4 38. Rc5b5 Re1e4 39. Kg2f3
Re4d4 40. Kf3e3 Rd4c4 41. Ke3d3 Rc4c3+ 42. Kd3d4 Rc3c1 43. Rb5b8+ Kg8h7 44.
Rb8:b4 Rc1d1+ 45. Kd4e4 Rd1e1+ 46. Ke4d5 Re1h1 47. Bh2g3 Rh1d1+ 48. Kd5e4
Rd1e1+ 49. Ke4f5 f7f6 50. Rb4b3 h6h5 51. Rb3b4 Re1d1 52. f2f3 Rd1d2 53. Bg3e1
Rd2e2 54. Rb4e4 Re2c2 55. Kf5e6 Rc2b2 56. f4f5 Rb2b6+ 57. Ke6f7 Rb6b3 58.
Re4f4 Rb3b7+ 59. Kf7e6 Rb7b3 60. Be1f2 Kh7g8 61. Bf2c5 Rb3c3 62. Bc5d4 Rc3a3
63. Bd4b6 Ra3c3 64. Bb6d4 Rc3a3 65. Bd4b6 Ra3c3 66. 1/2-1/2
```

PaulChess was not able to capitalize on the hole in LCP's defences, and made a strange trade of a knight for two pawns on move 24. This ended up being a bad idea, because LCP was able to use its superior middle game position to win more pawns. Somehow, PaulChess managed to earn a draw by repetition.

White: Computer (PaulChess3, 4/6 plies)
Black: Thinking Machine 4

```
History:
1. d2d4 d7d5 2. c2c4 d5:c4 3. e2e4 Bc8e6 4. d4d5 Be6d7 5. Bf1:c4 b7b5 6.
Bc4d3 c7c6 7. d5:c6 Bd7:c6 8. Ng1f3 Ng8f6 9. Nf3e5 Nf6:e4 10. Ne5:c6 Nb8:c6
11. Bd3:e4 Qd8:d1+ 12. Ke1:d1 O-O-O 13. Kd1e1 Nc6e5 14. Bc1e3 Ne5d3+ 15.
Be4:d3 Rd8:d3 16. Be3:a7 Rd3d6 17. Nb1d2 Rd6c6 18. a2a4 b5:a4 19. Ke1e2 Kc8b7
20. Ra1:a4 e7e5 21. Ba7e3 Rc6a6 22. Ra4:a6 Kb7:a6 23. Rh1a1+ Ka6b5 24. Ra1a8
Kb5c6 25. f2f3 g7g6 26. Ra8a7 f7f5 27. Nd2c4 Kc6d5 28. b2b3 Bf8b4 29. Ra7b7
Bb4c3 30. Rb7d7+ Kd5c6 31. Rd7a7 Rh8b8 32. Ra7:h7 Rb8:b3 33. Rh7g7 Kc6d5 34.
Nc4b6+ Kd5e6 35. Rg7:g6+ Ke6f7 36. Rg6c6 f5f4 37. Rc6c7+ Kf7g6 38. Rc7c6+
Kg6g5 39. Be3c1 Bc3d4 40. Nb6d5 Bd4g1 41. h2h3 Kg5h4 42. Rc6c8 Rb3b5 43.
Rc8h8+ Kh4g3 44. Rh8g8+ Kg3h2 45. Nd5c3 Rb5c5 46. Bc1b2 e5e4 47. f3:e4 Rc5c4
48. e4e5 f4f3+ 49. g2:f3 Bg1a7 50. Rg8g4 Rc4c5 51. f3f4 Rc5c4 52. f4f5 Rc4c8
53. h3h4 Rc8c5 54. Nc3d1 Rc5c2+ 55. Ke2d3 Rc2c5 56. Nd1e3 Rc5b5 57. Bb2c3
Ba7:e3 58. Kd3:e3 Rb5b3 59. Rg4c4 Rb3b7 60. Ke3f2 Rb7a7 61. e5e6 Ra7a2+ 62.
Kf2f3 Ra2a8 63. Bc3e5+ Kh2h3 64. e6e7 Ra8a3+ 65. Kf3e2 Ra3a2+ 66. Ke2e3
Ra2a3+ 67. Ke3e2 Ra3a2+ 68. Ke2e3 Ra2a3+ 69. Ke3d2 Ra3a2+ 70. Kd2e1 Ra2a8 71.
Ke1d2 Ra8a2+ 72. Kd2e1 Ra2a8 73. Ke1f2 Ra8a2+ 74. Kf2g1 Ra2g2+ 75. Kg1f1
Rg2g8 76. h4h5 Rg8g5 77. e7e8=Q Rg5:f5+ 78. Rc4f4 Rf5:f4+ 79. Be5:f4 Kh3h4
80. Qe8g6 Kh4h3 81. 1-0
```

Although this game ended up being very long, PaulChess was up a minor piece within 11 moves, due to questionable openings by Thinking Machine 4. It continued to build upon this, gaining a huge material advantage by the end game, and promoting to a queen for the win.

White: GNUChess 4 (1 second)
Black: Computer (PaulChess3, 4/6 plies)

```
History:
1. d2d4 Ng8f6 2. c2c4 e7e6 3. g2g3 c7c5 4. Ng1f3 c5:d4 5. Nf3:d4 e6e5 6.
Nd4c2 d7d5 7. c4:d5 Qd8:d5 8. Qd1:d5 Nf6:d5 9. e2e4 Nd5b4 10. Nc2e3 Bc8e6 11.
Nb1c3 Bf8c5 12. Bf1b5+ Nb4c6 13. Ne3d5 Bc5d6 14. Bc1e3 O-O 15. Ra1d1 Rf8d8
16. O-O Be6h3 17. Rf1e1 h7h6 18. a2a3 Bh3e6 19. b2b4 a7a6 20. Nd5b6 Ra8a7 21.
Bb5e2 Be6b3 22. Rd1:d6 Rd8:d6 23. Nb6c8 Rd6d4 24. Nc8:a7 Nc6:a7 25. Be3:d4
e5:d4 26. Nc3d1 Na7c6 27. f2f4 Nb8d7 28. Kg1f2 Nd7f6 29. Be2f3 Bb3e6 30. e4e5
Nf6g4+ 31. Kf2g2 d4d3 32. h2h3 d3d2 33. Re1e2 Nc6d4 34. Re2:d2 Ng4:e5 35.
Rd2:d4 Be6:h3+ 36. Kg2:h3 Ne5:f3 37. Rd4d8+ Kg8h7 38. Rd8d7 b7b5 39. Rd7:f7
Kh7g8 40. Rf7a7 g7g6 41. Nd1c3 a6a5 42. b4:a5 Nf3d4 43. a5a6 Nd4c6 44. Ra7c7
b5b4 45. Rc7:c6 g6g5 46. Rc6c7 b4:c3 47. a6a7 g5g4+ 48. Kh3h4 h6h5 49. 1-0
```

A complicated series of moves led to a rook fork in the middle game that PaulChess would never recover from.

White: Computer (PaulChess3, 4/6 plies)
Black: GNUChess 4 (1 second)

```
History:
1. d2d4 Ng8f6 2. c2c4 g7g6 3. Nb1c3 Bf8g7 4. e2e4 d7d6 5. Ng1f3 O-O 6. Bc1g5
Nb8c6 7. d4d5 Nc6b8 8. Bg5:f6 Bg7:f6 9. Ra1c1 e7e5 10. Bf1d3 Nb8a6 11. O-O
Bc8d7 12. Qd1d2 Bd7g4 13. Bd3e2 Bg4:f3 14. g2:f3 Bf6g5 15. Qd2c2 Bg5f4 16.
Rc1a1 Qd8g5+ 17. Kg1h1 Qg5h4 18. h2h3 Qh4:h3+ 19. Kh1g1 0-1
```

Although the game started fine, PaulChess chose the wrong piece to take in an exchange and opened up its king, demonstrating one of its weaknesses: general king safety. This turned out to be disastrous. GNU chess quickly exploited the problem and found a mate.