Fall 12-9-2016

Win Nim

Anna Carrigan
University of Wyoming, acarriga@uwyo.edu

Solving the Game of Nim

Anna Carrigan

University of Wyoming Honors Senior Project

Fall 2016

Introduction

Game theory: the branch of mathematics that people other than mathematicians find interesting. It's the "cool" side of math, the side that comes in handy during the Connect Four championship. But game theory is not just about winning games. Yes, game theory lets your computer win checkers every time. It can also help divvy up possessions in the event of a divorce (Matz 1339), or split up a cake fairly (Tannenbaum). John Nash, famous for his portrayal in the movie *A Beautiful Mind*, developed game theory concepts that are constantly used in economics and even biology (Babu, Krishnamurthy, and Parthasarathy 769-772). Those who study game theory never have to look hard for its applications.

My senior project is the combination of two courses I have recently taken here at the University of Wyoming: Game Theory and Mobile Application Development. I wanted to create a project with crossover between math and computer science, my two main fields of study. It is a great, simple project to show future employers when they request to see a previous project of mine. I created an Android application that uses game theory to help the user win a game they are playing against someone else. After some preliminary research, I narrowed the focus to Nim, a game well known to mathematicians but extremely simple to explain.

Nim starts out with at least one pile of chips. Players take turns choosing a number of pebbles to remove from a pile and whoever takes the last pebble wins. Nim is a completely solved game, so the winner is determined from the start. The application could coach the winner move-by-move to the end, or allow a determined loser to recover if the person they are playing against makes a mistake.

To complete my project, principles from mathematics courses, software design courses, and my mobile design course were implemented. Research and analysis on how to create the most

user-friendly interface as well as the most efficient and/or elegant algorithms for my solutions was also conducted. The completed product is highly demonstrative of my graphical user interface skills as well as my mathematics knowledge, but the development process is even more revealing – it demonstrates my organizational skills and my capability for planning and completing a major project.

Game Theory and Solved Games

There are too many different types of "games" that can be analyzed and solved with game theory to talk about in one paper. For this reason, our focus will be two-player, zero-sum games. A zero sum game is a game where one player wins what the other player loses (Ferguson). Many, many well-known games are zero-sum: Rock Paper Scissors, Connect Four, Checkers, Chess. In all of these games, if one player wins, then the other player loses. The outcome is what defines the game as zero-sum – not the in-between play. Zero-sum as a concept does not only apply to recreational games. Think of two customers arguing over the last bottle of detergent at the supermarket. This is a zero-sum game, because if one customer gets the detergent, the other customer loses it. Now, imagine two roommates at the supermarket arguing over which detergent to get. One roommate prefers Ajax, and the other prefers Tide. However, both roommates would prefer buying the same detergent over their preference so they can buy it together in bulk instead of buying two different bottles. So no matter what, each roommate will either get their preferred detergent or they will be glad to have the same detergent as their roommate. No one wins or loses! Non-zero sum games in the form of trades are a huge part of the study of economics. However, most recreational strategy games involve a win or a loss, and therefore the focus of this paper will be zero-sum, two person games that involve some form of strategy.

What does it mean to say a two-player, zero-sum game is "solved"? A first intuition may be to say a game is "solved" if a player has a be-all end-all strategy to win the game from the start, rendering the other player helpless and defeated. Mathematically, if a game is solved it means that a "property with regard to the outcome of the game has been determined" (Allis). In friendlier terms, a game is solved if at some point in the game you can prove the game will end

in a win, lose, or draw. This may seem like it's not very much information, and that's because it's not. Knowing who *can* win is not the same as knowing *how* to win. However, many mathematicians actually favor these games with very little determinable information. Their proofs tend to be deeper and more abstract.

A two-player, zero-sum game can be "solved" at three different levels: ultra-weak, weak, and strong. In ultra-weakly solved games, only the outcome of the game is known. For weakly solved games, not only is game value (Player 1 wins, draw, or Player 2 wins) known, but there is also an algorithm that one player can follow to secure a win or a draw. Donald Michie defines strongly-solved games as a game where from *any* position, a winner or a draw can be determined as well as a strategy to acquire that game value. These three types of "solved" games are defined by Allis and widely accepted in the mathematical community. It is probably easier to understand these terms in the case of examples. Connect Four is a strongly solved game. If two people were playing a game of Connect Four in a room, and a game theory expert were to walk in at any point in the game, he could determine both which player was in a position to win and what strategy they should follow to achieve this win (Tromp). English Draughts, a slight variation of regular checkers, is weakly solved. Two experts in game theory could coach two English Draughts player in "perfect play" from the beginning of the game, and the game is guaranteed to end in a draw. Hex, a game where the goal is to connect opposite sides of the board with pieces of the same color, has multiple board sizes. All square board sizes are ultra-weakly solved in that it is known that the first player can "steal" a winning strategy from the second player, and it is impossible to draw. However, a game theory expert could not coach the first player through a win. The game theorist is only able to tell the first player that they *can* win, (Henderson, Arneson, and Hayward 505-510).

Some games can be solved explicitly by mathematicians, without the use of a computer. A pencil and paper is enough to find the solution for games like Nim and its derivatives. Nim involves players taking turns removing a set number of pebbles from a pile (for example, on their turn a player can remove one, two, or three pebbles). The goal of the game is to take the last pebble. Combinatorial game theory allows for even an undergraduate student to determine who will win, and their winning strategy, from any position. In other words, Nim is strongly solved. The winning strategy may not be easily memorized (for instance, it may involve translating numbers into binary or writing down a chart), but nonetheless there is a winning strategy determined. Having a computer to help makes winning Nim a lot easier.

Other games cannot be solved so easily by a mathematician and his notebook. The game of checkers was solved by creating a database of every possible "end of game" position. Chinook, a checkers-playing program developed by Schaffer, has "perfect information about all positions involving eight or fewer pieces on the board, a total of 443,748,401,247 positions" (van den Herik, Uiterwijk, and van Rijswijck 277-311). A computer program like Chinook can look at a position in checkers and determine which player could win and how. The program takes a starting position and "plays" different moves until it reaches one of the end-game database positions. At this point, the computer knows the outcome of its stored position and can now determine the outcome of the initial position. When the program determines the outcome of a specific position, it stores it in a separate database to allow for quicker searches in the future. It is important to note that a Checkers player who is determined by the computer to be a winner at a certain position is only the winner if they proceed with *perfect play*. This concept will be discussed later on, but the general idea is that a player cannot win by just moving at random. The

player must play the perfect move every turn to guarantee the computer-identified outcome, which is why games are still fun: people make mistakes.

How does the computer get the information for the endgame solutions? The answer is a mathematician's least favorite kind of proof: brute force. For example, the Chinook endgame database generated and individually evaluated every single position with eight checkers remaining on the board, of either color (van den Herik, Uiterwijk, and van Rijswijck 277-311) by trying every possible move sequence and determining who could win every single time. Computerized chess engines also use these endgame "tablebases" and allow for software programs like Deep Fritz to beat world chess champion Vladimir Kramnik ("World Chess Champion Loses Match 4-2 to Deep Fritz Computer"). However, chess is not completely solved. There are too many possible outcomes and not enough computing power at this point in time to completely solve chess in the near future (Allis). Right now, chess-playing computers play strategically but not omnisciently while trying to get to a board position it has stored somewhere in its tablebase.

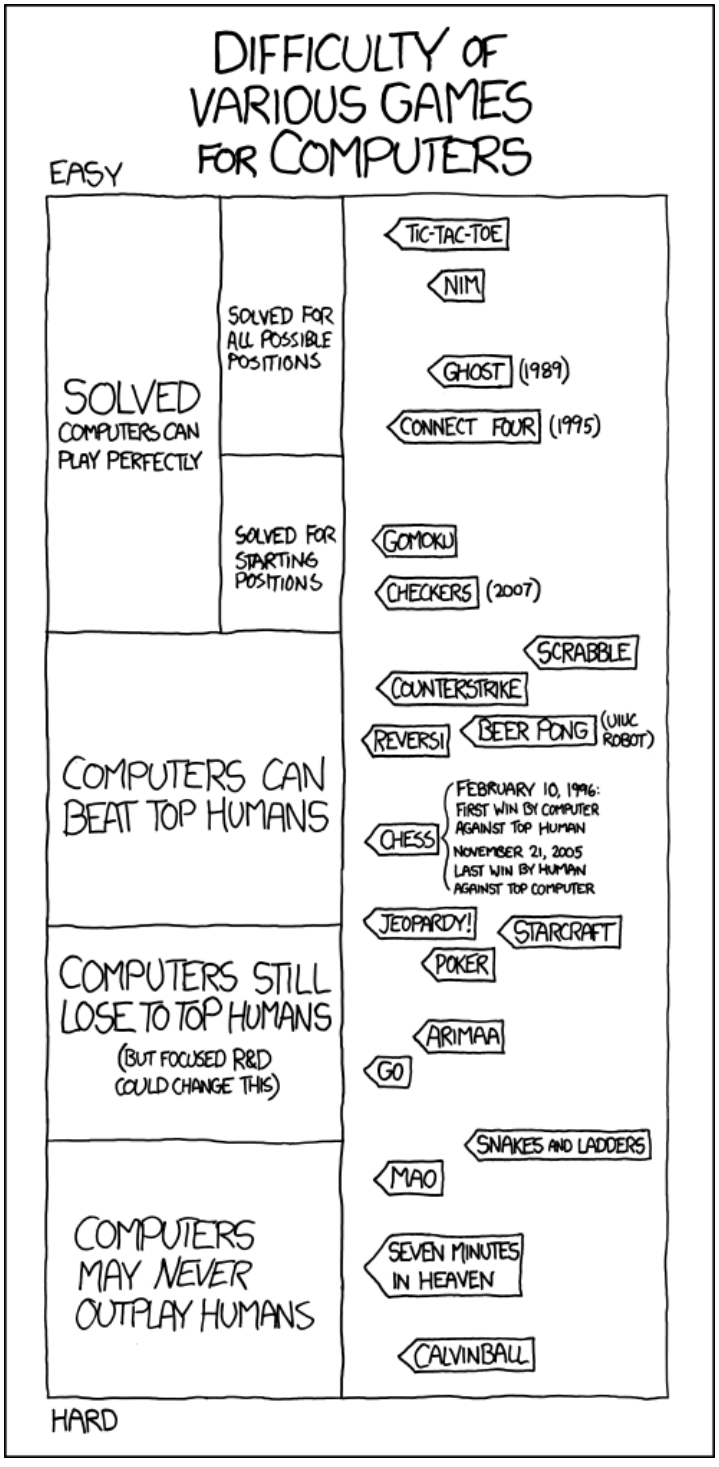
Even though chess is not solved, computer scientists and mathematicians have still produced a computer program "intelligent" enough to beat even the best human chess players. Is that good enough? Maybe. An "intelligent enough" artificial intelligence (AI) is often satisfactory for playing casual games. For example, rock-paper-scissors is a zero-sum game that simply cannot be solved – the players pick randomly each time. However, AIs have been created that analyze the behaviors of human players. These programs use strategies ranging from counting the opponent's moves to determine what move they play the most often, to creating a database of sequences of four moves that the opponent performs and matching current behavior to past sequences (Jordan 6-8). Tic-Tac-Toe is completely solved, but for lightweight

applications like a phone app, an AI is often suggested instead of a database search for the optimal move for both creative and storage reasons. Creating a Tic-Tac-Toe AI is a common exercise in computer science classes.

As mentioned earlier, even if a game is solved with Player 1 winning, Player 1 needs to demonstrate "perfect play" to guarantee their win. So what is perfect play? Knowing what move to take next to win requires something called "perfect information." A game with "perfect information" requires that both players know exactly what has happened so far, including the starting point of the game. A game like poker is not a game with perfect information, because players do not know what the other player has in his or her hand. A game like chess is a game with perfect information, because every player can see the entire board, and watch the other player move pieces. They have the exact same amount of information as the other player (Ferguson). Without perfect information, perfect play instead requires the player to play different moves with determined probabilities (Allis). Obviously, perfect play for most games is not possible for humans. Even computers that play chess only start using "perfect play" after they reach a board configuration already listed in their database. The task of a good game AI is to identify perfect play or near-perfect play for any and all game positions.

Nim is a game where perfect play is available from the start for a computer or even someone with just a pencil and paper. Perhaps the simplest of zero-sum games, Nim was the focus of my project because the strategies used to find a solution to win the game were a combination of brute force and the use of mathematical theorems. Meanwhile, humans continue to use computers to look for solutions to games that we have not quite solved yet.

Chart discussing the ability of computers to play games against humans (Munroe).



Creating a Nim Solver Application for Android

Solving a Game of Nim – The Algorithm

So how does the computer or phone solve the game of Nim? It all comes down to identifying winning and losing positions in the game. For example, having no chips left is a losing position for the current player, because it is their turn and they have nothing left to take. Having one chip left is a winning position, because you can take the last chip and force the next player into the losing position. Similarly, if players can only take one or two chips, then having three chips left is a losing position because the next player will just take whatever you leave and finish off the game. So how can we find these positions, especially if our game of Nim has more than one pile?

We can separate a game with many piles into smaller, individual games by separating out the piles. Once we have just one pile to handle, it is possible to assign what are called Sprague-Grundy numbers. Start at the last position in the game, where there are no chips left. This is a losing position for the current player. Label this position with a zero. All losing positions will be labeled with a zero. Now, think about how to win, you need to put the next player into a losing position. If it is *your* turn, you want to move to a losing position to set up the next player for failure. You want to move to a zero position. If you can move to a losing position from your position, you label it with how far it is to the next losing position. If you cannot move to it (like above, where you have three chips and the next losing position is zero chips, but you can only take one or two) then you label that position with a zero, because it is also a losing position. Eventually you have the entire pile mapped out as losing or winning positions.

If the game consists of only one pile, the algorithm is over. To see if you are going to win, check if you are in a winning position (the number labelled on the position is not zero) and

then move that number of chips out of your pile to put the next person in a losing position. If you have multiple piles, you take something called the Nim-Sum of the Sprague-Grundy numbers already assigned to piles to identify a winning or losing position. Take the Sprague-Grundy numbers of the current position in each pile you are in, and take their binary digital sum. That is, you turn the numbers into binary and add up the digits individually without carrying, mod two, to see if you have a zero (losing position) or a winning position.

My computational algorithm does just that. It takes your game, assigns Sprague Grundy numbers to the piles, and then calculates the Nim-Sum to determine if you are going to win. If you are, it can also tell you the optimal move by checking systematically for ways to get to that previously identified Nim Sum.

Planning Phase

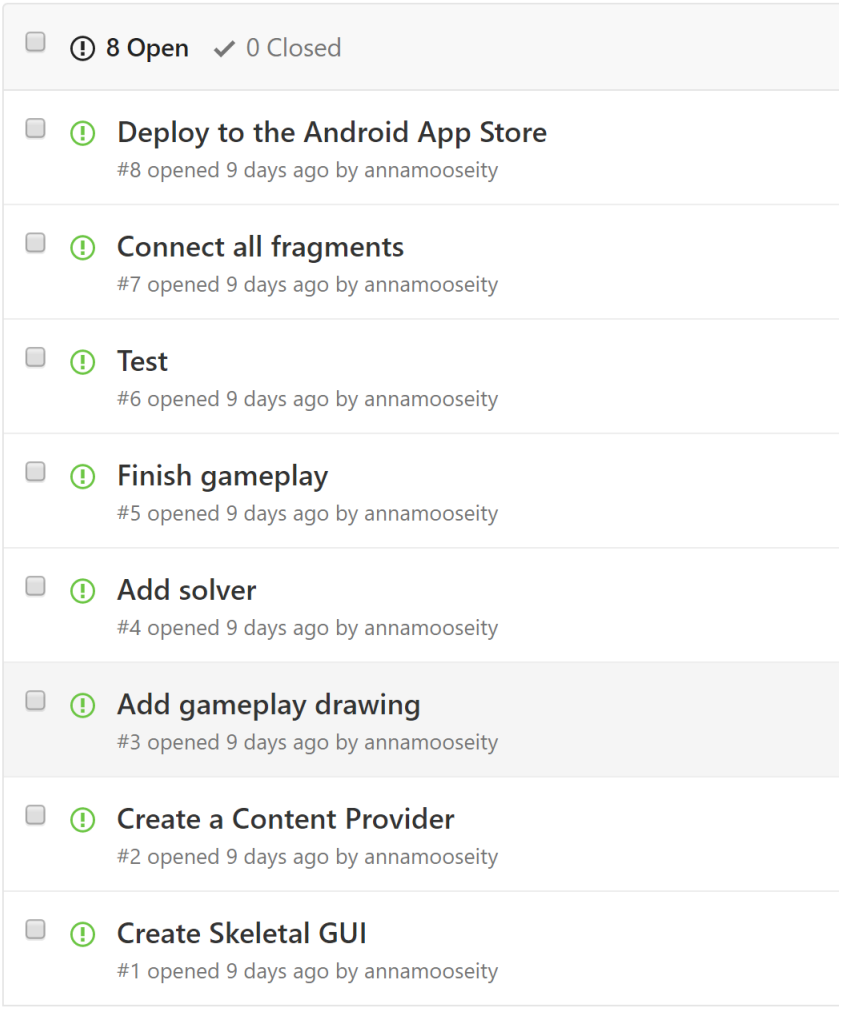
It may or may not surprise someone unfamiliar with software development that much of the coding process is not actually typing code. This is especially true for larger projects, because you cannot skate by on just a mental picture. At the beginning of the semester, I spent several weeks investigating good ways to organize my process, the Android creation and publishing cycle, and the computations involved in solving Nim.

All of my coding was done using Android Studio, an Integrated Development Environment created by Google specifically for creating Android applications. I already had some familiarity with this environment from a previous employer as well as my mobile development class this semester. During the development process I also discovered a lot of handy features for actual development that expedited the process, such as getter/setter generation, automatic refactoring, and suggestions for better code practice. Android studio would allow me to test my application in an emulator on many different sized phones to guarantee

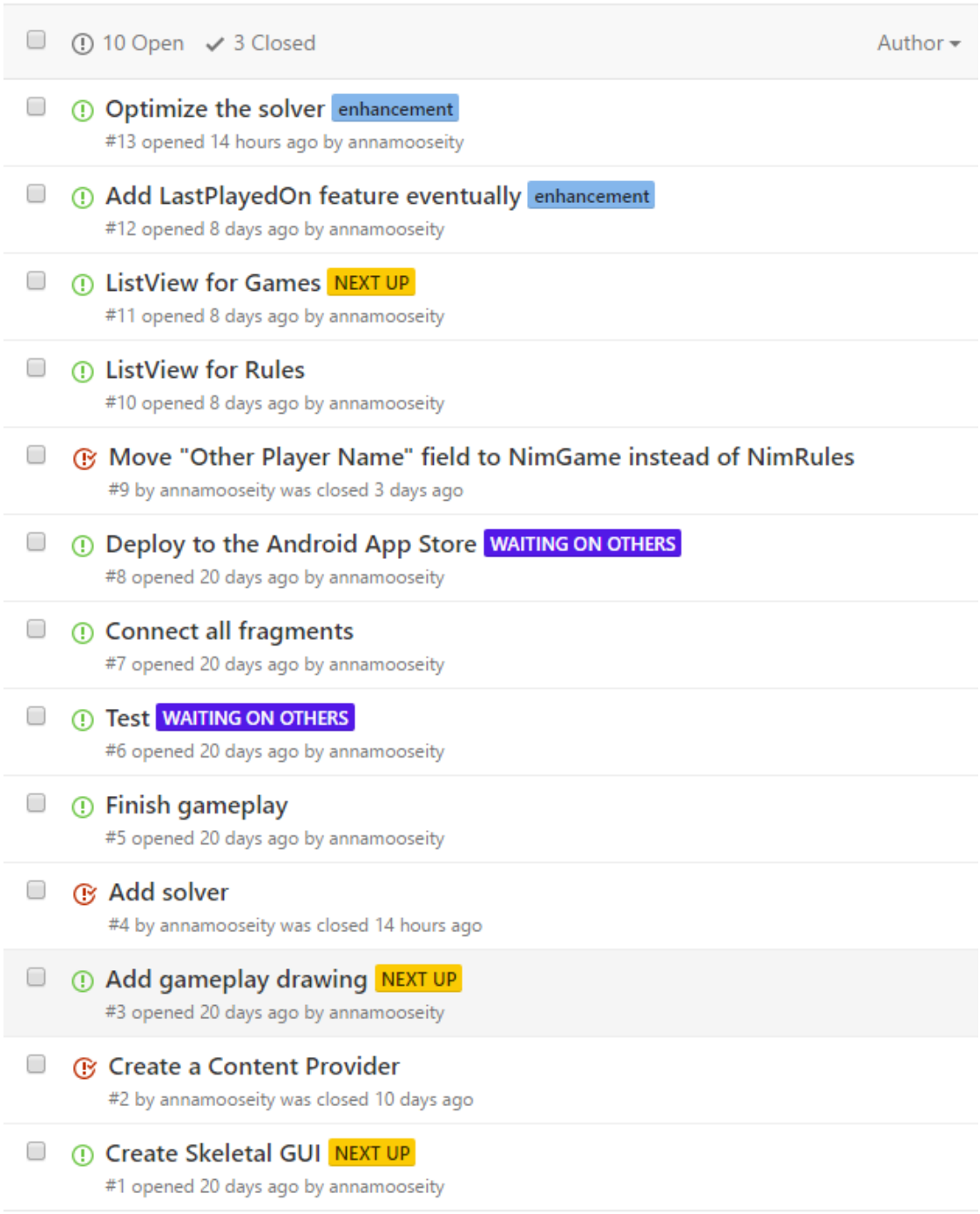
compatibility with as many Android phones as possible, especially because the phone I was testing on was must larger than average.

To keep my process streamlined, organized, and efficient, I created a public GitHub repository to use for both source control and for project planning. Source control is a way to "check in" changes to files (coding or not) and allows developers to document code modifications, record why they changed those things, and, if necessary, revert changes if something breaks badly. Github is especially important in team environments as it allows for team members to merge files that may have conflicts. For example, if two people are working on the same file at once and go to check in their changes, one person will not write over the changes of the other. However, it is extremely useful even for solo projects. If something goes terribly wrong, you do not have to start all over from the beginning – you just have to start from the last time it was working. Because of this, frequent (but not trivial) check-ins are recommended. Github also has a well-developed planning system available with its repositories. You can create issues, assign them to team members (or yourself), and link different check ins to those issues. I decided I was going to use this issues feature to organize and track the different steps I would need to create my application. I ended up adding ideas for enhancements to my program as well, tagged in purple in the second figure.

A screenshot of the GitHub issues for the Nim application soon after their creation.



A screenshot of the Github issues during development. Note that issues are tagged to keep the developer focused.



Each issue contained a bulleted list of steps that needed to be taken to close the issue. The issues were created so that it was not necessary to complete them in order (except for the last few issues). If one issue hit a roadblock, progress for the entire application would not need to be

stopped. I modeled this plan after my software development internship at Lockheed Martin, where issues were created in our software development management software (JIRA, by Atlassian) independent of other issues so that they could be assigned to any developer in any order.

For example, the "Create Skeletal GUI" bullet item included bullets for the aesthetic decisions (what color was everything going to be, and in what style), and then moved on to outline the creation of the user interaction elements without actually adding any data to them. "Create a screen that will list saved games" was on that list, even though "Create a way to save games" was not. Another item on the list was "create the backbone of the play screen." This involved creating a spot to put the gameplay view, but not actually doing any work on the view itself. This kept tasks from becoming overwhelming and allowed for more individualized testing of each piece, which prevented serious bugs later on in development. Blindly jumping into coding leads to weaknesses in the code structure, which leads to code "scaffolding," the equivalent of fixing weird code behaviors with code duct-tape. This makes it harder to improve code later, and makes it *very* hard to add more to the code without dealing with a massive cleanup. Planning forces developers to do it right the first time. I was very fortunate to be taking my Mobile Programming class at the same time as this assignment, as it answered a lot of questions I had up front. The class spent time on basic GUI elements, list views, and saving data outside of the application. Without this information, I am sure I would have figured out a way to code what I needed to code, but having the information already allowed me to hit the ground running and not hack together code pieces off of Google.

After outlining the issues in Github, creating my repository, opening a new Android project in that repository, and performing an initial check in to make sure my repository was indeed working, I was ready to start my Nim Solver application.

The Development Process – Finally, the Fun Part

I started with the "Create a Skeletal GUI" issue. It took me just a few hours to have a couple of my screens for Nim up and running. Of course they would need some tweaking, but testing is a lot more difficult when you do not have anything to look at. Since I am a stickler for a good, consistent GUI, I knew rearranging and making each screen perfect would use a large percentage of my time. I had to remind myself that I could also keep coming back to it once I had the necessary functionality.

One of the most difficult parts at the beginning was figuring out the slider bar that let the user decide how many piles there were going to be in the game. The trouble with a lot of development in Android Studio is that there are at least two different ways to do something, and usually one way is a lot easier than the other. Unfortunately, the easy way is often underrepresented in online computer science forums like StackExchange. After an hour or two of frustrated searching, I finally found the solution I was looking for (Senta). It was a one-line fix, but other sources had hinted at having to import an entire library. Fortunately, I refused to use that complicated of a solution without exhausting my search.

I tried to keep good coding practice in mind while I continued to develop my application. For example, I created a NimRules class that contained all the information you needed to know to set up a game of Nim: who plays first, how many piles there are, how many chips per pile there are, how many chips a player could take at a time, et cetera. Then I created a NimGame class that had its own set of rules, plus a number of moves counter and an array that contained

the current states of each of the piles. This made it easier for me to develop without typing the same thing over and over. It also made my code more readable and reusable. These NimGame and NimRules objects were extremely handy when I was developing my list view, as I could work with objects instead of just strings from a database.

The next big decision was how to store my data while the application was not in use. I decided to implement a content provider in the form of an SQLite database to store game rules and saved games. For a while, I thought keeping a list in my application of all the different games would work. I quickly realized that list goes away as soon as you close the application. Thus, a SQLite content provider database was born. This allowed me to store objects in the way I had intended the objects to be stored. I had two tables in my database, one for rules and one for games. The games table had a column that linked each game to the set of rules. After creating the content provider, I did not have the functionality to test it quite yet. When I did link the lists up with the content provider, I was pleasantly surprised at the lack of bugs that I had to correct. My Mobile Programming class did a great job familiarizing me with content providers.

After I got the content provider up and running along with a GUI to use to put data into the database, I was ready to start fleshing out some of the more complicated parts. First in line was the list view for the different sets of rules. For functionality purposes, the adapter for the list rows, which tells the list how to display different parts of the rules (like piles, options for number of chips to take, and who goes first) in the rows, was extremely bare bones at first. After getting everything else working I improved its aesthetic, but at first it would just print the different details in array format. The list was interpretable by me and me alone, and even I forgot what it was telling me occasionally. It was a big relief towards the end when I finally made the list descriptive. I did something very similar for the adapter for the list of games, even though that

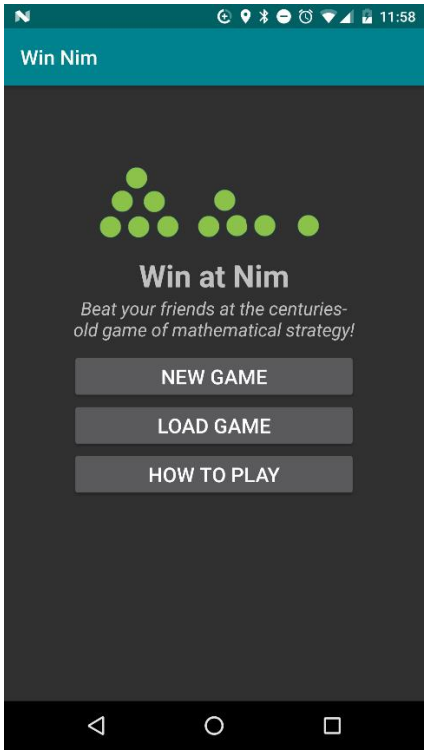
was developed much later. I wanted to avoid bogging myself down with deciding how everything should look right away when it would likely change anyway. In the end it was a good decision because I was able to make everything look cohesive. I made the decision to create a new game from the list of rules – to create a new game, the user simply taps on the rules they want to create the game with. A dialog will pop up and ask for the name of the opponent and, when you click "Let's go!", a copy of the new game will be saved and the game will be displayed. To delete a set of rules or a game from either of the list views, the user must press and hold on the game and then confirm they actually want to delete the game. I do worry that there is a more user-friendly way to have programmed my application, but I got feedback from friends and family and they did not seem to mind the layout. After publishing my application in the Android app store, though, any feedback will be taken into consideration. During this stage of development, I had to keep my eye on the original goal, however, as I was on a deadline.

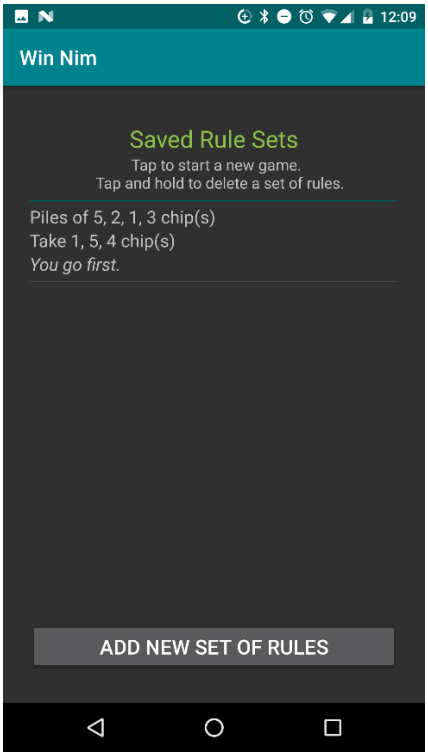
The gameplay board may sound like the most exciting part to develop, but it was actually the easiest and therefore the least satisfying because its animation is very minimalist. Since only up to six piles are allowed, I added six custom views (circles) which highlight when touched and display the number of chips in the pile in the middle of the circle. I allocated space to tell the user whose turn it was, allow the user to pick how many chips to choose, and space for the solving algorithm I developed to tell the user how to win if they can. The hardest part was getting a game to load into that play screen from the database, but after a lot of hard work I had it working, and saving. At that point it was time for a lot of intermediate testing to make sure I was still on the right track. I identified and fixed a couple issues like piles with negative chips, inconsistent loading, and one particularly picky bug where I had assumed that an item's position in a list was the same as the item's position in the database. Another bug was the possibility for

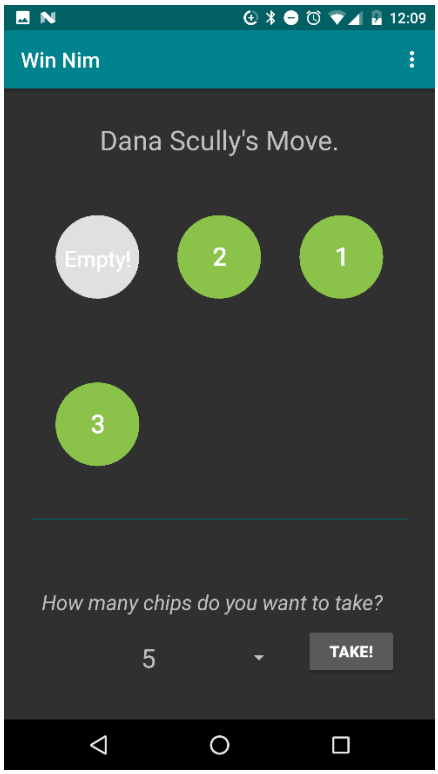
draws – if players were not allowed to take single chips, they could get stuck in a draw. I made the executive decision to completely eliminate that possibility. Users must always have the option to take just one chip from a pile. If rules were evaluated more closely, this requirement would not be as necessary. However, then I run the risk of having an angry user who cannot figure out why their rules are not being accepted. I finished adding the remaining functionality like checking if something has changed when the user clicks back, asking them if they want to save, game updates and deletions, and menus in the action bar. Many hours of testing both alone and with friends and family, I was ready for the last step.

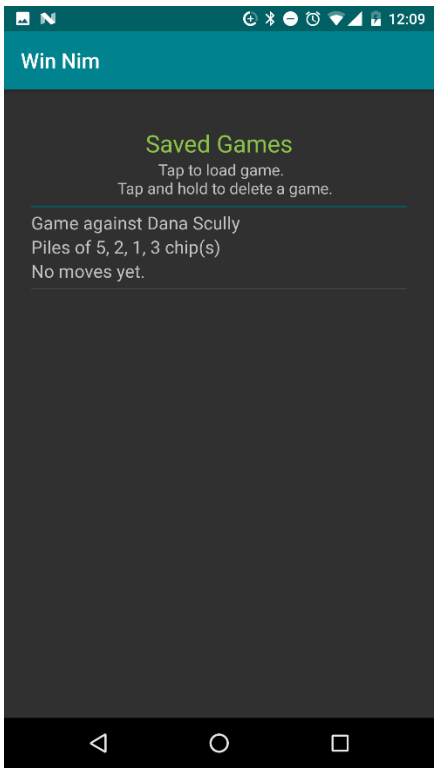
"Make things pretty" was actually the title of one of my issues in Github. I spent a lot of time going through and thinking about colors and usability to try to make it as user friendly as possible. I wanted users to be able to download the app and start playing even if they had never heard of Nim before. I have included screenshots of the final (for now) appearance of my application. A lot of time was spent on identifying professional-looking colors and layouts, and I am highly satisfied with the result.

Most recent views for the Win Nim application.









Opportunities for Improvement

The opportunities for more development with this application are limitless, although I have identified a few that I would enjoy addressing in the near future.

A “last played on” feature for the games list would be convenient. Games could be sorted by most-recent, making it easier for users to pick up where they left off. Adding more piles to the game would be a nice feature as well. This is not difficult to implement, and would just involve drawing more piles on the play screen. Another quick improvement would be adding a “play against an AI” feature, as the solver is already implemented. Playing against an AI would just involve a few changes in logic and a couple of extra user interfaces.

One of the more important, but less exciting opportunities for improvement is optimizing the solving and playing algorithms by using dynamic programming. This is a matter of time and concentration as well as design reflection. If I did make improvements on this, I would also want to do quite a bit of code refactoring to encapsulate functions into cleaner classes and make my code have even better design principles. These improvements would go unnoticed by the user but could improve the performance very slightly and allow bigger modifications to go more smoothly in the future.

Another improvement I had considered adding at the beginning is the ability to edit rule sets after they are created. I can see users becoming frustrated by having to enter in multiple, similar rule sets. However, there would need to be a way to choose if you want to edit the rules or just start a game, and a good way to do this is not immediately apparent. The other, bigger problem would be that this function would always have to operate as a “Save as new” after editing, because rules tied to an existing game should not change. This could be done, but it would take some investigation and design planning outside of the original application’s scope.

As discussed earlier, this application makes end-game draws impossible, which is a benefit in my eyes. However, I can see some mathematicians as finding this limiting. Adding handling for draws and updating the solving algorithm would be a potential improvement, although I do not want to address this unless specifically requested. What fun is a zero-sum game if it can end in a draw?

Other improvements, like adding animation or drawing the board differently are absolutely in the future of this application's development. However, I want to wait to get more feedback from customers in the Google Play store, friends, family, and teachers. What I think is a good design may not be what most of the population thinks is a good design, and I do not want to get tunnel vision while continuing my development. I do plan to keep working on this application and making improvements, and I am excited to see where this takes me.

Conclusion

My application is both fun and educational. It can be used to teach anyone with a basic understanding of math that some games are already solved. The concept of Nim Sums when talking about the application and "how does it do that", could be explained to students with a basic understanding of binary. These students could then use the app to "prove" to themselves that the strategy really does work. I myself was somewhat in disbelief that a game like Nim is completely solved, but this app has proven to me that if you follow directions, you will win (especially if the other person makes a mistake).

Developing this application has been a great learning process for me, and I believe that will continue as I continue to improve upon my design. Not only did I get to investigate more about Game Theory, which is a personal interest of mine, but I was able to develop and deploy my first application to the Google Play store. My Nim-solving application is available on the Google Play store as WIN NIM, for download by any Android device. I plan to upload the APK to Github so that even those with Android devices that cannot access the Google Play store (like Kindles and other third-party Android tablets) can download the application and beat their friends at Nim. This is also a great project to show to future employers. I am very proud of my progress and look forward to making it better and better.

Works Cited

Allis, L. V. "Searching for Solutions in Games and Artificial Intelligence." Rijksuniversiteit Limburg, 1994. Web.

Babu, Sujatha, Nagarajan Krishnamurthy, and T. Parthasarathy. "The Creative Genius: John Nash." *Resonance* 21.9 (2016): 769-72. Web.

Ferguson, Thomas. "Two-Person Zero-Sum Games." *Game Theory.* 2nd ed., 2014. Print.

Henderson, Philip, Broderick Arneson, and Ryan B. Hayward. "Solving 8×8 Hex*." *IJCAI* (2009): 505-10. Web.

Jordan, Michael B. "An Actuarial Artificial Intelligence for the Game Rock-Paper-Scissors." *South African Journal of Science* 112.5/6 (2016): 6-8. Print.

Matz, Jeremy A. "We'Re all Winners: Game Theory, the Adjusted Winner Procedure and Property Division at Divorce." *Brooklyn law review* 66.4 (6): 1339. Print.

Munroe, Randall. *Xkcd: Game AIs*.Web.

Senta, Nilesh. *Discrete Seekbar without Third Party Library*., 2016. Web. Nov 6, 2016.

Tannenbaum, Peter. *Excursions in Modern Mathematics*. 8th Edition ed. Harlow, Essex: Pearson, 2014. Web.

van den Herik, H. Jaap, Jos W. H. M. Uiterwijk, and Jack van Rijswijck. "Games Solved: Now and in the Future." *Artificial Intelligence* 134.1 (2002): 277-311. *CrossRef.* Web.

"World Chess Champion Loses Match 4-2 to Deep Fritz Computer." *Daily Bulletin*Dec 6, 2006. *Global Newsstream.* Web. <http://search.proquest.com/docview/357458994>.