GamesmanPuzzles: A Leap Into the Puzzles Domain

Anthony Ling
Dan Garcia, Ed.
Joshua Hug, Ed.

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2021-146
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2021/EECS-2021-146.html

May 21, 2021

Copyright © 2021, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Acknowledgement

Prof. Dan Garcia, who gave me this opportunity to develop something truly special in the last 2 years.

Prof. Joshua Hug, for agreeing to be the second reader and giving me advice to improve my work.

My fellow members of the GamesmanPuzzles group, who helped contribute Puzzles and advice on the package.

Mark Presten (https://github.com/mpresten) implemented Peg Solitaire and the Command Line Interface (CLI) in the “Results” section.

Arturo Olvera (https://github.com/arturoolvera) implemented N-Puzzle and designed Figure 3.

The GamesCrafters group, who supported me in integrating GamesmanPuzzles into GAMESMAN.

My friends and family, who supported my education throughout it all.

GamesmanPuzzles: A Leap Into the Puzzles Domain

by Anthony Ling

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Teaching Professor Dan Garcia
Research Advisor

(Date)

* * * * * * *

Teaching Professor Joshua Hug
Second Reader

(Date)

Abstract

Puzzles are one-person "games", with the player executing moves in a problem state. The goal is for the player to reach a winning state and avoid losing states, if they exist. *GamesmanPuzzles* is a system designed to strongly solve the Puzzle by finding the remoteness values of every reachable position, perform analysis, and allow the puzzle to be played. This software has a Python interface, and is built on top of Professor Dan Garcia's GAMESMAN project, which strongly solves two-player games and provides powerful functionality such as analysis tools and a graphical game playing interface.

GamesmanPuzzles was developed as an effort to expand the intellectual convex hull of the GamesCrafters computational game theory research and development group into the domain of puzzles. It does so by providing functionality for playing and solving Puzzles, as well as integrating Puzzles into the GAMESMAN ecosystem. By providing tutorial material for newcomers, we hope this will serve as a resource for future development.

Acknowledgements

- Prof. Dan Garcia, who gave me this opportunity to develop something truly special in the last 2 years.
- Prof. Joshua Hug, for agreeing to be the second reader and giving me advice to improve my work.
- My fellow members of the GamesmanPuzzles group, who helped contribute Puzzles and advice on the package.
    - Mark Presten (https://github.com/mpresten) implemented Peg Solitaire and the Command Line Interface (CLI) in the "Results" section.
    - Arturo Olvera (https://github.com/arturoolvera) implemented N-Puzzle and designed Figure 3.
- The GamesCrafters group, who supported me in integrating GamesmanPuzzles into GAMESMAN.
- My friends and family, who supported my education throughout it all.

Abstract **2**

Acknowledgements **3**

1. Introduction **6**

2. Background **8**

2.1 Puzzles 8

2.1.1 Value Classification 10

2.1.2 Solving Puzzles 12

Symmetries 14

2.1.3 Variants 14

2.2 GamesmanPuzzles Group 14

2.3 GAMESMAN Frontend and Backend APIs 15

2.3.1 GamesmanUni 15

2.3.2 GamesCraftersUWAPI 15

3. Programming Model, Architecture, and Implementation **16**

3.1 Puzzles 16

3.1.1 String Representations 17

3.1.2 Moves 17

3.1.3 Primitives 18

3.1.4 Default Positions and Solutions 19

3.1.5 Symmetries 19

3.2 Solvers 19

3.3 Backend Server 20

3.4 Players 20

3.5 Curriculum 21

3.5.1 Tutorial (2 weeks) 21

3.5.2 Design (1 week) 21

3.5.3 Development (2 weeks) 22

4. Results **23**

4.1 Puzzles 25
4.1.1 Example: Towers of Hanoi 25
4.1.2 Example: Lights Out 26
4.1.3 Example: N Puzzle 27
4.1.4 Example: Peg Solitaire 28
4.2 Solvers 29
4.2.1 GeneralSolver 29
4.2.2 Solver Wrappers 29
Relative Performance 30
4.3 Players 31
4.4 Curriculum 32
5. Future Work **33**
5.1 Additional Puzzles 33
5.2 Randomized Starting Positions 33
5.3 Forwards Moves Solver 33
5.4 Binary Extensions 34
5.5 Distributed Computing 34
5.6 Additional Documentation 34
6. Conclusion **35**
References **36**
Appendix **38**
A.1 Puzzle Functionality 38
A.2 Puzzle Base Class 39
A.3 ServerPuzzle Base Class 44
A.4 Hanoi Puzzle implementation 46
A.5 Solver Base Class 52
A.6 Code to Generate Solver Graphs 53
A.7 Server Puzzle Assignment (Design part) 55
A.8 Server Puzzle Assignment (Develop part) 57

1. Introduction

GamesCrafters is an undergraduate research and development group formed by Professor Dan Garcia in 2001 with the purpose to solve two-player games using combinatorial and computational game theory. It was built on top of the GAMESMAN project. GAMESMAN was originally developed by Prof. Garcia in 1990 as an effort to provide an open-source architecture for encoding, solving, analyzing, and playing games. Users simply need to define game modules in order to access the full functionality of the library, such as a graphical or command line interface [11].

GamesmanPuzzles adds to the GAMESMAN project by expanding the GamesCrafters group into the domain of puzzles, one-player games. Similarly, we aim to provide powerful functionality for puzzles similar to how GAMESMAN provides powerful functionality for games. We first proposed the GamesmanPuzzles project in the spring of 2020 [8]. During that time, the GamesCrafters group, established by UC Berkeley Professor Dan Garcia, had the following active projects:

- GamesmanClassic, a collection of games encoded and solved in C and based on Professor Garcia's original Gamesman Masters Thesis [5].
- GamesmanUni, an online web GUI [10].
- GamesmanJava, a parallel solver using Apache Spark [7].

This project satisfies the following key requirements that were identified through our time in GamesCrafters:

- *Build an application that strongly solves Puzzles:* This is based on the original ideals of the GamesCrafters group and the GAMESMAN project.
- *Simple to develop and easy to build upon:* GamesCrafters is largely an undergraduate student organization, and new members often have little or no programming experience. It is beneficial

to create a project that follows a simple programming model and is coded in a language that many students at UC Berkeley would understand, which in this case is Python. The project's main feature is the collection of Puzzles, so adding more Puzzles must be possible with limited guidance.

- *Be relevant to GamesCrafters applications:* There are many GamesCrafters projects that are often discontinued either due to lack of relevancy or lack of support. Integration with the main GAMESMAN applications ensures relevancy in the unforeseen future.
- *Relatively performant:* The goals of this project is to create the foundations and interfaces for solving Puzzles and project integration, so optimizing for performance is not a major focus. However, it can be a focus in the future, and Puzzles should be able to be solved in a reasonable amount of time.

2. Background

2.1 Puzzles

Puzzles aren't new to GAMESMAN. Between 2008 and 2010, many students contributed time and effort into authoring code that would solve and play puzzles [2, 3, 4]. However, due to an unfortunate server crash, lack of documentation, and student interest, development was discontinued. Our goal with GamesmanPuzzles is to provide a simple and powerful package that will have continued support in the unforeseen future.

A "Puzzle" can be defined as the following:

- A discrete set of states Loading…. This represents every possible "board state" for a Puzzle.
- A discrete set of **forward** moves Loading… for every state Loading…. Each move maps a state Loading… to another state Loading…, which can be represented as a function on the state Loading… s.t. Loading…. This represents all the moves that can be made for any given "board state" of a puzzle.
- A discrete set of solution states Loading…. This represents the *primitive* states an agent playing would like to reach.
- A discrete set of **backwards** moves Loading… for every state Loading…. It performs similar functionality as a **forward** move Loading…, but performs the inverse operation.

An example of a popular Puzzle would be the Towers of Hanoi. Each state of this Puzzle is represented by the arrangement of the Puzzle, while the act of moving discs from one rod to another represents a move. The Towers of Hanoi puzzle traditionally begins with three rods in a line and a stack of Loading… differently shaped disks on the leftmost rod, ordered by size of the disk (smallest on the top, largest on

the bottom). The goal is to move all the discs onto the rightmost rod. This is done by moving the topmost disk of any stack of a rod onto the stack of another rod. The disk can only be placed either on the floor of the rod or on top of a disk that is bigger than it [17].

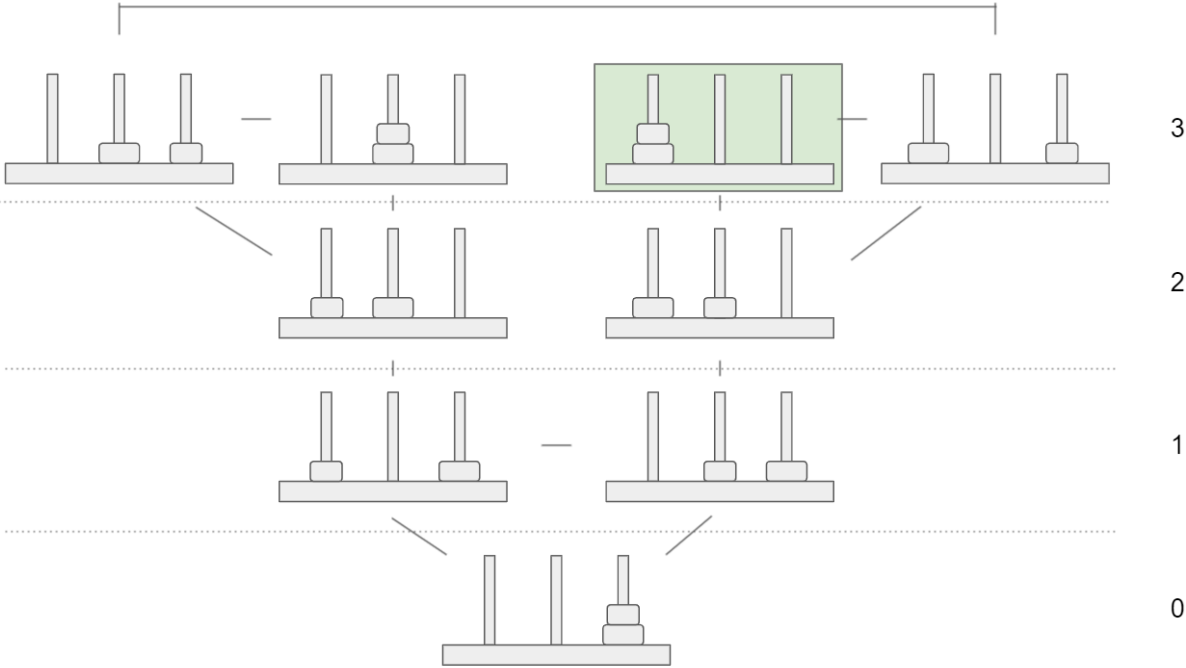

Figure 1*:* Visualization of all of possible positions of Hanoi, variant 2 disks, 3 rods. The remoteness is indicated by the numbers on the right. The green rectangle represents the starting position.

The remoteness Loading… of a Puzzle is defined to be the minimum number of moves to reach the solution state. Figure 1 shows each possible position of Towers of Hanoi with 3 rods and 2 disks. Each position is organized in layers split by the dotted lines. The numbers on the right represent the remoteness values of all positions in the layer. The remoteness is calculated recursively; it is the minimum of the remoteness of all child positions (all the positions the current position can reach in one move) plus 1. For example, the starting position of this Puzzle (indicated by the green rectangle) has two possible moves, one to a position with remoteness 3 and to another position with remoteness 2. Since

min(2,3) + 1 = 3, we set the remoteness of the starting position to be 3. The process of assigning remoteness values will be detailed more in the “Solving Puzzles” subsection.

2.1.1 Value Classification

We define two Puzzle categories. Depending on the type, the classification of positions differs. All Puzzles listed will be defined later in the “Results” section.

- *Always-winnable* Puzzles: The most common Puzzle type, the formal definition is that there is a path to the solution state from all positions (reachable by the initial position). Example Puzzles include Towers of Hanoi and Lights Out.
- *Not-always-winnable* Puzzles: The formal definition is that is a path from the initial position to the solution state, but there is not necessarily a path from all positions (reachable by the initial position) to the solution state. Example Puzzles include Peg Solitaire and Chair Hopping.

We classify positions and moves differently based on which type of Puzzle is being solved. This is to allow players to know which moves are optimal when interacting with a visualization (i.e. GamesmanUni). As a visual, **winning** moves are colored in green, **tieing** moves are colored in yellow, and **losing** moves are colored in red.

In an Always-winnable Puzzle, every position is a **winning** position. Moves that lower the current remoteness are classified as **winning** moves, moves that maintain the same remoteness are classified as **tieing** moves, and moves that raise the current remoteness are classified as **losing** moves.

In a Not-always-winnable puzzle, positions that cannot reach a solution state in any sequence of moves are classified as **losing** positions, while every other position is a **winning** position. Moves that lower the current remoteness are classified as **winning** moves, and moves that maintain or raise the remoteness are classified as **tieing** moves. Moves that lead to a losing position are classified as **losing** moves.

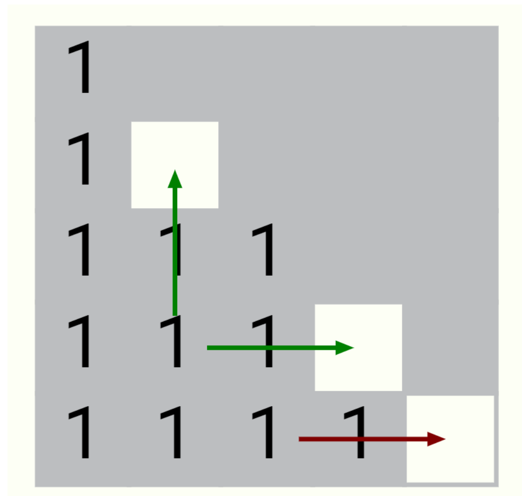

Figure 2: Example of Peg Solitaire in GamesmanUni displaying the colors of moves. Peg Solitaire is a Not-always-winnable Puzzle, meaning the red move will result in a losing position.

2.1.2 Solving Puzzles

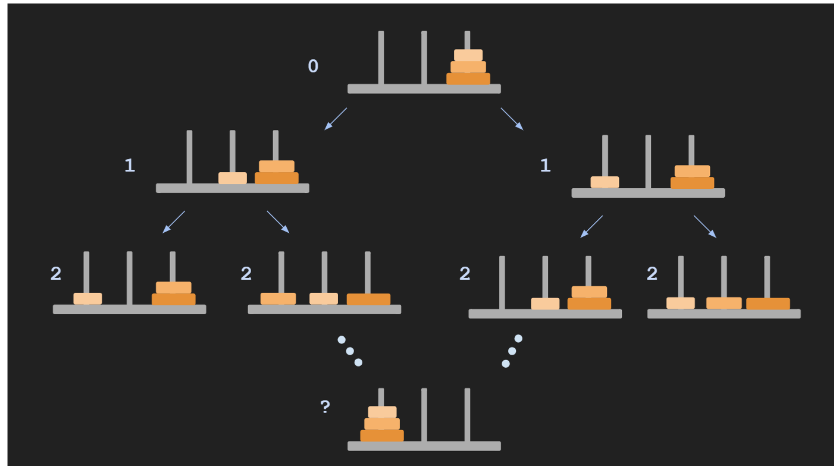

Figure 3: A visualization of the backwards pass for solving the Towers of Hanoi. Figure designed by Arturo Olivera.

Our algorithm attempts to strongly solve the puzzle by finding the remoteness of every possible state, as follows:

- If the solution states of a Puzzle are not known, identify those solution states by using BFS (i.e. forwards pass) on a defined starting state, usually the position from `generateStartPosition`. Initialize the remoteness of all solution states to 0, and add all solution states to our "frontier" (those just labeled as having remoteness R).
- Start by initializing Breadth First Search (BFS) on every solution state (i.e. backwards pass) Loading…. For every state Loading… examined by BFS, we exhaustively search the entire state space reachable through backwards moves Loading….
- The BFS proceeds forward in "waves", at each round expanding out from the frontier, labeling the newest previously-unlabeled positions, and replacing the frontier with these new positions.
- Initialize the "next frontier" to empty. For all states reachable in one backward move from the frontier (positions with remoteness R), if the state has not been labeled yet and is not a symmetry (defined below), set the remoteness to R+1 and add it to the next frontier (once this

step is done, replace the frontier with the next frontier). Continue until the previous step adds no new states to the frontier.

States that aren't able to reach solutions are classified as **losing** positions. As stated before, this allows for the convenient traversal of an agent through a puzzle towards a solution.

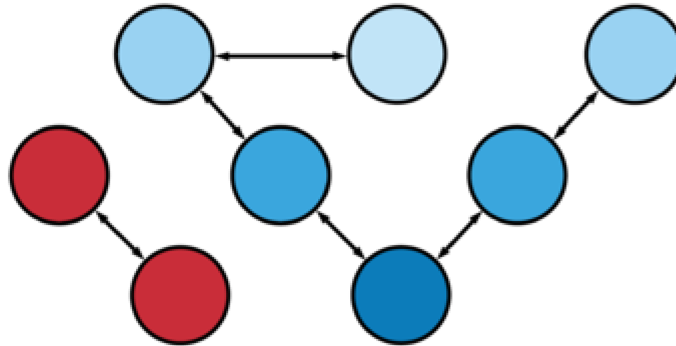

Figure 4: Demonstrating how a puzzle is solved. The bottom-most blue node is the solved position and the nodes connecting it are solvable positions with decreasing remoteness (indicated by decreasing shades of blue). The red nodes are unsolvable positions.

Symmetries

Symmetries are defined as positions that share the same remoteness. By reducing positions to symmetries, we reduce the number of positions needed to be traversed by the solver algorithm. Some symmetry reduction techniques include flipping or rotating boards in 2D puzzles. Figure 5 demonstrates an example of a symmetry in Hanoi.

Figure 5: Both of these positions in Hanoi have the same remoteness and have been encoded in Hanoi to share the same hash value.

2.1.3 Variants

Puzzles can have multiple *variants*, which are Puzzles that have similar rulesets with a major difference. For instance, the default Towers of Hanoi variant can have 3 rods and 3 disks, but a variant can have a differing amount of disks or rods (i.e. 5 disks, 10 rods). Variants allow for the reuse of existing code to create an entirely new Puzzle.

2.2 GamesmanPuzzles Group

The GamesmanPuzzles group is a subgroup of GamesCrafters, which focuses on projects involving the Puzzle domain (mainly the development of the GamesmanPuzzles project). It was founded in Fall 2019 by the establishment of the GamesmanPuzzles project and had 7 members during its two year period.

Members first join the GamesCrafters group and then are given the choice to choose one of its subgroups. Newcomers in the GamesmanPuzzles are tasked with a 5-week onboarding assignment to develop and explore how to develop Puzzles (described in Programming Model, Architecture, and Implementation).

2.3 GAMESMAN Frontend and Backend APIs

2.3.1 GamesmanUni

GamesmanUni is a GamesCrafters project that serves to provide GUIs and analysis for games on the internet in the form of web applications. It provides two major features:

- Integration of many GamesCrafters projects into our HTML server.
- Conversion of backend Games/Puzzles into automatic GUIs, dictated by GamesCraftersUWAPI [6].

Before, GamesmanUni only supported regular two-player Games, and didn't have any support for Puzzles. Our contribution to GamesmanUni is the addition of Puzzles to the GamesmanUni server, including GUI generation and the Visual Value History (VVH) [10].

2.3.2 GamesCraftersUWAPI

GamesCrafters Universal Web API is a GamesCrafters project that defines a standard and connects GamesCrafters projects together under one web API [6]. This is the medium where GamesmanUni accesses game data such as remoteness and position values. Other than being a step into integrating Puzzles into GamesmanUni, adding Puzzles into GamesCraftersUWAPI allows for future frontend applications to access Puzzles and integrate them into their systems as well.

3. Programming Model, Architecture, and Implementation

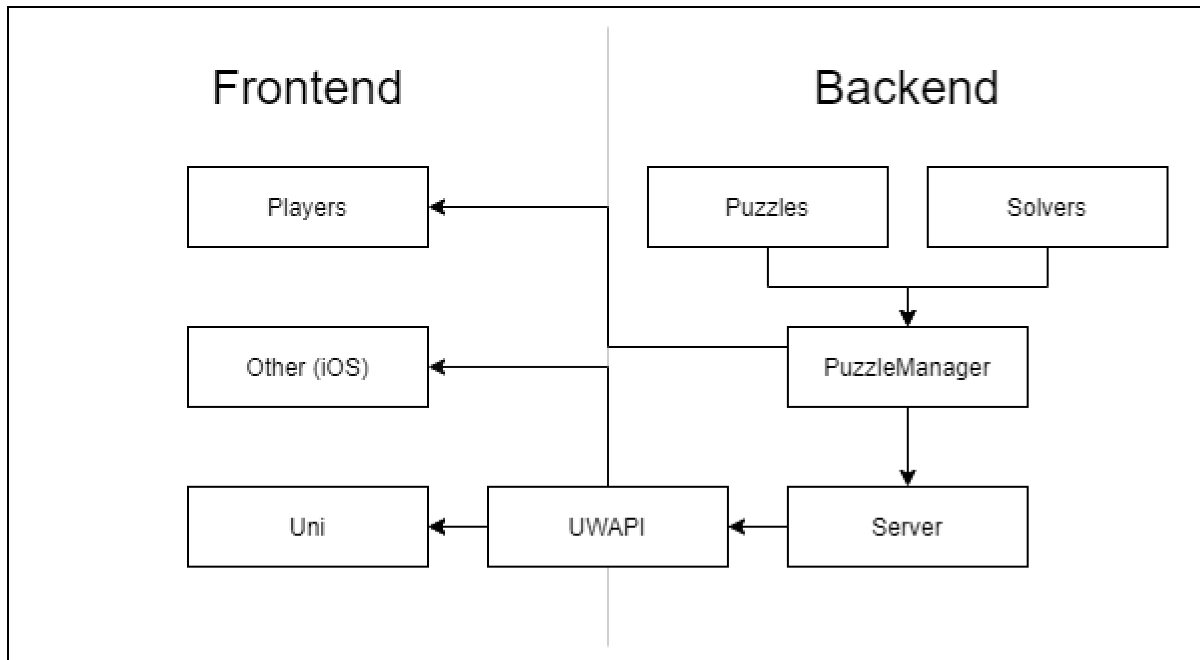

Figure 6: The System Design of GamesmanPuzzles

GamesmanPuzzles core functionality implements a puzzle-Solver programming abstraction and follows the Python object-oriented programming (OOP) model.

3.1 Puzzles

A *Puzzle* object represents the abstraction of a puzzle state, as discussed in the "Motivations and Requirements" section. Towers of Hanoi with 3 disks and 3 rods will be used as an example for all of the example functionality. Much of the functionality described below is available for review in Appendix A.1.

3.1.1 String Representations

Puzzles support string representations to indicate the current state for user and developer interactions. There are two modes of string representation, each define the mode keyword argument in `toString(mode)`:

- *"minimal"* provides a url-friendly representation, which is used in the backend server.
- *"complex"* is a multi-line string representation resembling ASCII art. It is the default behavior of `__str__`.

For example, Puzzles support the following methods:

- `toString(mode)` takes in a *mode* keyword argument and supports two *modes*.
- `fromString(puzzle_string)` takes in a *minimal* string and returns a Puzzle object.
- `__str__` the built-in method for string representation that can be defined by any Python object and is called when print is called. Default is the *complex* string.

As an example, the starting position of the three-disk Towers of Hanoi puzzle is when all the disks are stacked on the leftmost rod. The minimal string representation is "7-0-0", where each number represents the disks on the rod. This can be further visualized by representing each number in binary, where the index of each bit represents the type of disk and the value represents whether the disk is on that rod. Another example is "6-1-0", where the string representation indicates the two larger disks on the leftmost rod while the smallest disk in the middle.

3.1.2 Moves

Puzzle objects follow the abstraction of a puzzle and are immutable objects. They are able to generate possible forward moves to next states as well as generate backgrounds moves to previous states.

- `generateMoves(movetype)` generates moves from the current position and is able to generate forwards and backwards moves based on the movetype.
- `doMove(move)` executes a move on the current position and returns the resulting state after the move is made.

For example, in Appendix A.1, calling `generateMoves` on the starting position (printed as "7-0-0") produces {(0, 1), (0, 2)} and `doMove` on move (0, 1) produces a new Hanoi object (printed as "6-1-0").

3.1.3 Primitives

Solvers need to know whether a position has reached the set of solutions and whether there are any possible moves left. Essentially, it asks "is the puzzle over?". If so, and we've achieved the solution, return a "win", otherwise return a"lose". If the puzzle is not over, return "undecided".

- `primitive` returns the following strings:
    - *"win"* indicates that the position is at a solution state.
    - *"undecided"* indicates that the position is not at a solution state and still has possible moves.
    - *"lose"* indicates that the position is not at a solution state and does not have any possible moves

For example, in Appendix A.1, calling `primitive` on Hanoi with three disks on the leftmost rod ("7-0-0") returns "undecided" while on Hanoi with three disks on the rightmost rod ("0-0-7") returns "win".

3.1.4 Default Positions and Solutions

Puzzles must generate a starting position of the puzzle to allow players a starting position when playing the Puzzle. When `generateSolutions` is not defined in a Puzzle, the solver uses the starting position to find the solutions states as described in 2.1.2.

- `generateStartPosition(variant)` returns the starting position of the puzzle based on the variant
- `generateSolutions` (optional) returns the solution states of the puzzle

For example, in Appendix A.1, calling `generateStartPosition` on variant "3_3" (3 disks, 3 rods) will produce Hanoi with three disks on the leftmost rod ("7-0-0"), while calling `generateSolutions` will return a list of solution states. In Hanoi, that solution state is when all three disks are on the rightmost rod ("0-0-7").

3.1.5 Symmetries

Puzzle objects support symmetries by allowing Puzzles positions that have the same remoteness value have the same "hash" value. This can be defined by any Puzzle by overwriting the `__hash__` function and defining an algorithm to detect those symmetries. Hanoi handles symmetries through reindexing and reduction. This process is described in 3.1 Advanced Hashing Techniques in the GamesmanPuzzles tutorial [9].

3.2 Solvers

A *Solver* object solves a Puzzle object through the process described in Motivations and Requirements. Once a Puzzle is solved, the Solver is able to determine the remoteness of any Puzzle object that shares the same Puzzle and variant. It is also possible to perform analysis on the Puzzle, such as determining

the number of positions that has to be traversed or the maximum remoteness of the Puzzle. Solvers are also able to be persistent by storing their remoteness data structure into a file, then by saving the file into local storage. Another Solver of the same type can access the file for reusability. Solvers that are integrated into the system must support a minimal API, which can be viewed in Appendix A.5.

- `__init__` The initialization of the solver, anything that has to be initialized can be called here.
- `solve` Solves the puzzle, normally by finding and storing the remoteness values of all positions of a Puzzle.
- `getRemoteness(position)` Returns the remoteness of the position.

3.3 Backend Server

In addition to the regular Python API, we also have a backend web API communicating using JSON files. This results in adding additional functionality for each Puzzle object, creating a new type called a ServerPuzzle. ServerPuzzles mainly differ from regular Puzzle types through the inclusion of metadata, as well as user input validation and type methods for Puzzle object generation. The backend server is powered by a simple script and utilizes the Python Flask library. It runs Solvers and Puzzles, which are managed by an object known as the PuzzleManager.

3.4 Players

For local user interaction, Player types were introduced. Player objects are able to take in a Puzzle object and optionally a Solver object, and users are able to interact with the Player such as inputting moves or viewing analysis of the Puzzle object.

3.5 Curriculum

As part of a major requirement, it must be easy to develop in GamesmanPuzzles in order to ensure relevance. As such, there is a major emphasis on documentation and in-depth tutorials for future developers. For all newcomers to GamesmanPuzzles, an onboarding stage is initiated. The number of weeks next to each stage signifies the expected number of weeks to complete that stage, but the entire process is flexible based on the student's schedule. It's important to note that curriculum was distributed over remote learning due to the COVID-19 pandemic, and thus the timing may differ in an in-person setting.

3.5.1 Tutorial (2 weeks)

Students are given 2 weeks to read over an in-depth tutorial into developing a Puzzle and a Solver. The tutorial consists of 8 pages of implementing Hanoi and 4 pages of developing a basic GeneralSolver. The pages are stored in GitHub as Markdown files. These tutorials are meant to provide students a greater understanding of how GamesmanPuzzles works as well as how to develop a Puzzle or Solver [9].

The tutorial provides all of the code in the complete implementation and splits it into different functions. It explains what each of the functions does, as well as any core concepts they fulfill. Students complete their own implementation of Hanoi by copying the segments of code and executing the final result with commands given.

3.5.2 Design (1 week)

Students are given a task to design their own Puzzle and create a writeup listing requirements to fulfill. This writeup serves multiple purposes; one, it gives the project leader understanding of the student's

current knowledge of the Puzzle as well as a chance to clear up any misconceptions. Two, it provides additional documentation for the Puzzle for future developers to look on.

The properties of the Puzzle as required in the writeup:

- **Puzzle name**
- **Puzzle ID**: A simple identifier of a puzzle
- **Puzzle visual**: Picture of a physical example of a Puzzle
- **Description**: Short description of Puzzle
- **Position**: The string representation of a Puzzle
- **Moves**: Types of moves as well as string representation of a move
- **Variants**: Include a default variant with a position limit to ensure feasible solving times
- **Optimization** (optional): Methods to improve solving times further

After the Design write-up, it is reviewed by other GamesCrafters and commented on for improvements. Once the review process is completed, the Development assignment starts. This assignment can be viewed in raw Markdown in Appendix A.7.

3.5.3 Development (2 weeks)

Students are given a task to develop their own Puzzle based on their Design document, as well as integrate it directly into GamesmanPuzzles. Students are asked to take inspiration from previous implementations of Puzzles as well as their experiences in the Tutorial stage.

The assignment also requires students to work on tests, as a way to test correctness as well for future use. The testing framework used for testing is pytest, and its correctness is maintained through TravisCI. The assignment can be viewed in raw Markdown in Appendix A.8.

4. Results

As of the time of writing, GamesmanPuzzles currently supports the following:

- 10 Puzzles
- 1 in-memory Solver
- 3 persistent Solver wrappers
- 1 Command Line Interface
- 1 Backend server implementation as well as 1 Frontend integration

Name	Variant	Always-winnable?	Default Remoteness	Max Remoteness	Number of Positions (w/o symmetries)	Number of Positions (w/ symmetries)
Towers of Hanoi	3 rods, 1 disk	Yes	1	1	2	3
Towers of Hanoi	3 rods, 2 disk	Yes	3	3	5	9
Towers of Hanoi	3 rods, 3 disk	Yes	7	7	14	27
Towers of Hanoi	3 rods, 8 disk	Yes	255	255	3281	6561
Towers of Hanoi	4 rods, 3 disk	Yes	5	5	15	64
N Puzzle	4	Yes	N/A	6	12	12
N Puzzle	9	Yes	N/A	31	181440	181440
Lights Out	2x2	Yes	4	4	16	16
Lights Out	3x3	Yes	5	9	512	512
Lights Out	4x4	Yes	4	7	4096	4096
Peg Solitaire	Triangle of side 5	No	13	13	13935	16384
Chair Hopping	10	No	35	35	476	2772
Bishops	5x4 board, 2 bishops	Yes	18	19	100	1260
Bishops	7x4 board, 2 bishops	Yes	12	12	972	6006
Bishops	7x6 board, 3 bishops	Yes	16	16	26566	1085250
Rubiks	2x2x2 Cube	Yes	N/A	14	3674160	$3.24 * 10^{15}$

Figure 7: An analysis of all thePuzzles currently in GamesmanPuzzles

4.1 Puzzles

There are 10 puzzles that have been implemented in our collection. The following examples describe example Puzzles. Appendix A.1 showcases the functionality of Puzzles in Python, while Figure 7 details all the results from solving these Puzzles.

4.1.1 Example: Towers of Hanoi

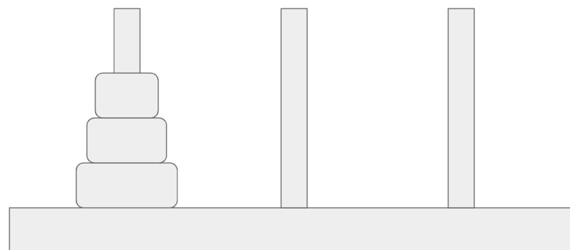

Figure 8: A visualization of the Towers of Hanoi

Towers of Hanoi was one of the first Puzzles to be implemented in GamesmanPuzzles and was known for its simplicity and convenience in Fall 2019. The maximum remoteness for any 3-rod puzzle can be easily calculated through a simple equation: Loading..., where Loading... is the number of disks.

It consists of a number of rods and different size disks. Disks can move from one rod to another with the restriction that larger disks cannot be placed on top of smaller disks. The goal of this Puzzle is to move all of the disks from the leftmost rod towards the rightmost rod [17].

This Puzzle is used in multiple examples, such as providing the benchmarking for Solver performance and being chosen as the Puzzle to be implemented in the tutorial. It has yet to receive a frontend GUI.

4.1.2 Example: Lights Out

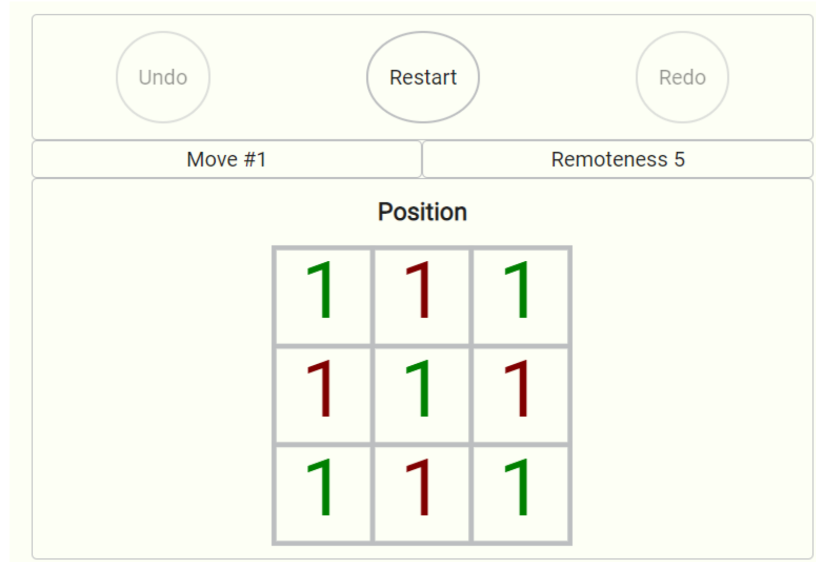

Figure 9: The Lights Out, 3 x 3 variant, displayed on GamesmanUni

Lights Out is a Puzzle involving a 2D grid of Lights. The player is able to switch a Light's state by selecting a Light and switching the state of the Light and its 4-way adjacent Lights. All of the Lights are initially lit up and the goal is to have all of the Lights off [15]. Its development and front-end implementation was done in Spring 2020.

This puzzle also supports a frontend GUI hosted by GamesmanUni. The GUI itself displays a 2D grid filled with 1s and 0s, where 1s indicate a Light being on while 0s indicate a Light being off. Players can select any square in the grid to execute a move.

4.1.3 Example: N Puzzle

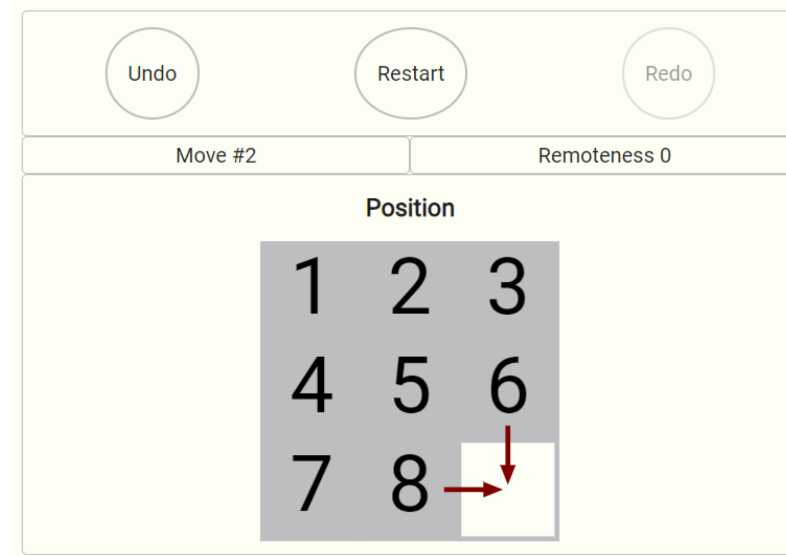

Figure 10: The N Puzzle, 3x3 variant, displayed on GamesmanUni

N Puzzle (also known as 16 Puzzle, 15 Puzzle) is a Puzzle involving a 2D grid of numbers with one slot open. Numbers adjacent to the slot can slide into the open slot, The goal is to order all the numbers row by row [14]. The GamesmanPuzzles implementation was developed by Arturo Olvera in Spring 2020, while the front end implementation was developed by Anthony Ling in Spring 2021.

This puzzle also supports a frontend GUI hosted by GamesmanUni. The GUI itself displays a 2D grid filled with numbers and an open slot. Arrows indicate where a number can be moved to fill in a slot.

4.1.4 Example: Peg Solitaire

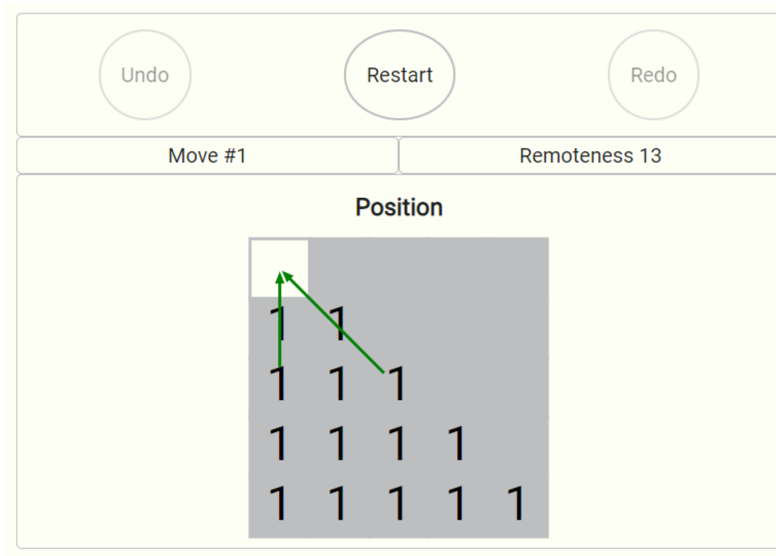

Figure 11: Peg Solitaire, Regular variant, displayed on GamesmanUni

Peg Solitaire (also known as Triangular Peg Solitaire) is a Puzzle with 14 pegs on a triangular board with 5 pegs on a side. Moves consist of jumping a peg over an adjacent peg into a hole two positions away; the "jumped-over" peg is then removed from play. The goal of the Puzzle is to leave the board with only one peg remaining [16]. The GamesmanPuzzles implementation and front end implementation was developed by Mark Presten in Spring 2020.

This puzzle also supports a frontend GUI hosted by GamesmanUni. The GUI itself displays a 2D grid, with half of the grid filled with 1s. Each block can jump over another block by using arrows indicating a jump, and players select these arrows to execute a move. At the time of this writing, the "AutoGUI" feature of GamesmanUni did not support triangular boards, so we "sheared" our triangular board to the left to live within its rectangular framework.

4.2 Solvers

There is 1 in-memory solver that was implemented and 3 persistent solver wrappers.

- GeneralSolver
- PickleSolver
- IndexSolver
- SQLiteSolver

4.2.1 GeneralSolver

The GeneralSolver is our main solver algorithm used to solve Puzzles. As described in the process of solving Puzzles in Background, GeneralSolver supports both Backward passing and Forward passing, as well as querying remoteness values and position values. Forward passing is initialized if there are no defined solution states and `generateSolutions` doesn't return any positions.

4.2.2 Solver Wrappers

PickleSolver, IndexSolver, and SQLiteSolver are Solver wrappers around GeneralSolver that allow for persistence. All three of them store the remoteness values in a local file; the main difference between all three of them is how they store the remoteness values in local files.

- PickleSolver directly dumps the remoteness dictionary as a Python object into a Pickle file (Pickle is a standard Python library that supports object serialization) [12].
- IndexSolver stores a byte array, with each index of the byte array being the hash value of the position. This solver indicates a need for a tight hash function.
- SQLiteSolver utilizes the sqlitedict for persistence and stores the remoteness values in a SQLite file [13].

Relative Performance

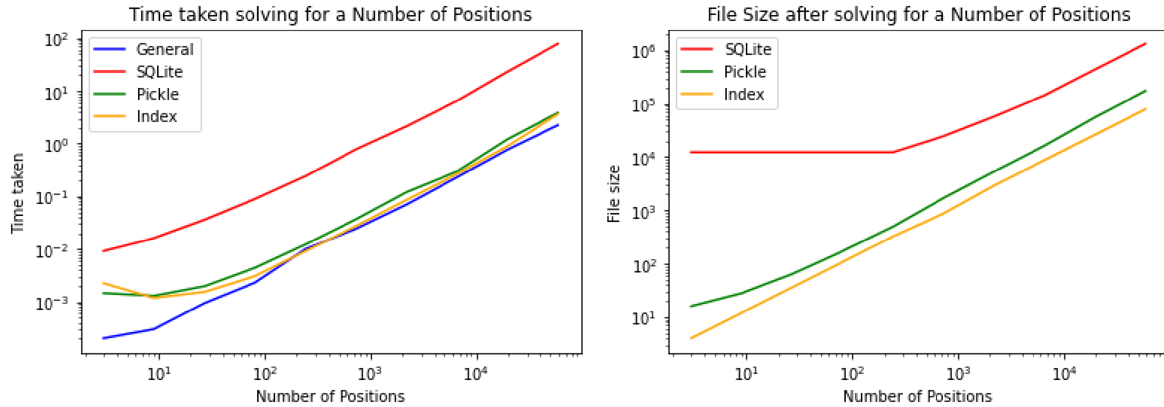

Figure 12: Time taken and File Size as a function of the Number of Positions, executed on Towers of Hanoi

We compared the performance and size of files between our solvers. SQLiteSolver was the least performant in both time taken and file size, while PickleSolver and IndexSolver was competitive in those fields. One should note that IndexSolver's file size varies depending on the hash function. A suboptimal hash function may result in larger file sizes.

4.3 Players

```
ant1ng@DESKTOP-BO5M3MR: ~/GamesmanPuzzles/puzzlesolver/players
Turn:                       0
Primitive:                  SOLVED
Position:                   WIN
Remoteness:                 7
Winning Moves:
- (0, 2)                        6
Tieing Moves:
- (0, 1)                        7
Losing Moves:
- <None>
Puzzle:
   A   |   |
   B   |   |
   C   |   |
---------------
   0   1   2
Possible Moves:
0 -> (0, 2)
1 -> (0, 1)
Enter Piece:
```

Figure 13: Command Line Interface of Towers of Hanoi. Shown in the figure is the "complex" string representation of Hanoi with possible moves to make, followed by possible Winning and Tieing moves.

We have one implementation of a Command Line Interface for local playing. It was developed by Anthony Ling and Mark Presten in Fall 2019. It displays the string representation returned by the `__str__` function and solver information for the best moves and remoteness values. Moves can be executed by specifying the index of the move displayed.

4.4 Curriculum

Out of the 10 Puzzles in our system, 3 of the puzzles (Bishops, TopSpin and N-Queens) were developed under the assignment structure. During the tutorial stages, most of the problems involved finding bugs with the example code, which were quickly resolved and fixed in the tutorial.

The Design stage proved invaluable to both students and the project lead. The project lead was able to share tips regarding potential design flaws, such as an incorrect number of positions possible for a variant or an unattractive string representation.

The Development stage was the most difficult, with most students asking the project lead questions regarding programming errors and missing details required in a Puzzle implementation. While some of these errors were due to student error, other errors showed crucial details missing from the documentation.

5. Future Work

5.1 Additional Puzzles

As a major feature of GamesmanPuzzles and the fundamental goal of our project (as depicted through our implementation of the curriculum), we wish for future members of GamesmanPuzzles to continue developing more Puzzles and adding them into the system, such as Klotski, Sokoban, WayOut, SnakeBird, or even their own Puzzles! Adding more Puzzles will increase the richness and functionality of the project.

5.2 Randomized Starting Positions

Users often tire of starting from the same position. Puzzles like N-Puzzle don't have a defined starting position in literature, so introducing randomized starting positions could allow players to interact with Puzzles in new and innovative ways. An example implementation would use the solver to discover all the possible positions then map those positions to remoteness values. The solver would need to be able to generate random starting positions for each remoteness value to allow the user a chance to change difficulty.

5.3 Forwards Moves Solver

We wish to make the API more minimal by defining a Solver that is able to solve Puzzles without defined backwards moves. This would not only place less burden on the developer but also make GamesmanPuzzles more flexible to more Puzzles.

5.4 Binary Extensions

As part of the requirements, Python was used as the language of choice. The intent was due to Python being an easier language for newcomers to GamesCrafters to understand due to their background. As a side effect however, much of the codebase remains unoptimized and unsuitable for large puzzles with more than 10^7 positions.

Performance can increase using the inherent speedup of binary extensions onto CPython, which allows Python to access C functions. Cython is also known to be a good and simpler alternative but is not as flexible as CPython.

5.5 Distributed Computing

GamesCrafters often deals with games that have nearly 10^{10} positions, which are infeasible to be solved using a single computer. Expanding solver functionality to work on multiple machines would be beneficial, such as utilizing the Message Passing Interface (MPI) directly through binary extensions or the mpi4py Python library. Utilizing the UPC++ libraries for a global shared hash table is also another possibility.

5.6 Additional Documentation

While the procedure to develop a Puzzle has been well documented (the tutorial), there are additional aspects that still need further explanation as indicated by the results of the curriculum. In order to ensure familiarity with GamesmanPuzzles, more documentation and student testing is necessary.

6. Conclusion

This paper introduces GamesmanPuzzles, a collection of Puzzles bundled together in a simple yet powerful Python interface. It was developed as an effort to expand the GamesCrafters group into the domain of Puzzles, and provides functionality similar to its predecessor project, GAMESMAN. It does so by providing functionality for playing and solving Puzzles, as well as integrating Puzzles It attempts to foster development of Puzzles beyond the scope described in this report by providing introductory material for newcomers to continue development. Through our efforts, we hope the GamesmanPuzzles is poised to be a big part of GamesCrafters in the future.

References

1. GamesCrafters. (n.d.). *GamesCrafters*. GamesCrafters. http://gamescrafters.berkeley.edu/
2. The GamesCrafters Group. (2008, November). *Fa2008Puzzles*. GamesCrafters Wiki. https://nyc.cs.berkeley.edu/wiki/Fa2008Puzzles
3. The GamesCrafters Group. (2010, November 28). *GamesmanWeb/PythonPuzzles*. GitHub. https://github.com/GamesCrafters/GamesmanWeb/tree/master/PythonPuzzles
4. The GamesCrafters Group. (2010, December 23). *Puzzle Writeup Fall 2010*. GamesCrafters Wiki. https://nyc.cs.berkeley.edu/wiki/Puzzle_Writeup_Fall_2010
5. The GamesCrafters Group. (2021). *GamesmanClassic*. GitHub. https://github.com/GamesCrafters/GamesmanClassic
6. The GamesCrafters group. (2021). *GamesCraftersUWAPI*. GitHub. https://github.com/GamesCrafters/GamesCraftersUWAPI
7. The GamesmanJava Group. (2021). *GamesmanJava*. GitHub. https://github.com/GamesCrafters/GamesmanJava
8. The GamesmanPuzzles Group. (2021, April 10). *GamesmanPuzzles*. GitHub. https://github.com/GamesCrafters/GamesmanPuzzles
9. The GamesmanPuzzles Group. (2021, April 10). *GamesmanPuzzles Tutorial*. GitHub. https://github.com/GamesCrafters/GamesmanPuzzles/tree/master/guides/tutorial
10. The GamesmanUni Group. (2021). *GamesmanUni*. GitHub. https://github.com/GamesCrafters/GamesmanUni
11. Garcia, D. D. (1990). *GAMESMAN* [A finite, two-person, perfect-information game generator]. GamesCrafters. https://people.eecs.berkeley.edu/~ddgarcia/software/gamesman/GAMESMAN.pdf

12. Python Software Foundation. (2021, May 16). *pickle - Python object serialization*. Python 3.9.5 Documentation. https://docs.python.org/3/library/pickle.html
13. Rehurek, R., Escobar, V. R., Usov, A., Swaminathan, P., & Quast, J. (2020, October 9). *sqlitedict -- persistent dict, backed-up by SQLite and pickle*. GitHub. https://github.com/RaRe-Technologies/sqlitedict
14. Wikipedia. (2021). *15 Puzzle*. Wikipedia. https://en.wikipedia.org/wiki/15_puzzle
15. Wikipedia. (2021). *Lights Out (game)*. Wikipedia. https://en.wikipedia.org/wiki/Lights_Out_(game)
16. Wikipedia. (2021). *Peg solitaire*. Wikipedia. https://en.wikipedia.org/wiki/Peg_solitaire
17. Wikipedia. (2021, May 15). *Tower of Hanoi*. Wikipedia. https://en.wikipedia.org/wiki/Tower_of_Hanoi

Appendix

A.1 Puzzle Functionality

Here’s example code demonstrating how to interact with Puzzles.

```
>>> from puzzlesolver.puzzles import Hanoi
>>> puzzle = Hanoi() # Equivalent to Hanoi(variant={rod_variant : 3, disk_variant : 3})
>>> puzzle = Hanoi.generateStartPosition("3_3") # Equivalent to the previous line
>>> puzzle = Hanoi.fromString("7-0-0") # Equivalent to the previous line
>>> print(Hanoi().toString(mode="minimal"))
7-0-0
>>> print(puzzle)
   A   |   |
   B   |   |
   C   |   |
---------------
   0   1   2
>>> puzzle.generateMoves()
{(0, 1), (0, 2)}
>>> print(puzzle.doMove((0,1)))
   |   A   |
   B   |   |
   C   |   |
---------------
   0   1   2
>>> print(puzzle.primitive())
undecided
>>> puzzle = Hanoi.generateSolutions()[0] # The solution state of Hanoi
>>> puzzle = Hanoi.fromString("0-0-7") # Equivalent to the previous line
>>> print(puzzle)
   |   |   A
   |   |   B
   |   |   C
---------------
   0   1   2
>>> print(puzzle.primitive())
win
>>> from puzzlesolver.solvers import GeneralSolver
>>> solver = GeneralSolver(puzzle) # Initializing Solver object with Puzzle
>>> solver.solve() # Solving the Puzzle
>>> solver.getRemoteness(Hanoi.fromString("7-0-0"))
7
>>> solver.getRemoteness(Hanoi.fromString("0-0-7"))
0
```

A.2 Puzzle Base Class

This is the Puzzle Base Class for all Puzzles defined in the GamesmanPuzzles system.

```
# These are general functions that you might want to implement if you are to use the
# PuzzlePlayer and the GeneralSolver
from ...util import classproperty, depreciated
import progressbar
import warnings

class Puzzle:

    #################################################################
    # Background data
    #################################################################

    id   = None
    auth = None
    name = None
    desc = None
    date = None

    #################################################################
    # Intializer
    #################################################################

    def __init__(self):
        """Returns an instance of a Puzzle. Board state of the Puzzle
        should be a Puzzle returned from `generateStartPosition`
        """
        pass

    #################################################################
    # Variants
    #################################################################

    @property
    def variant(self):
        """Returns a string defining the variant of this puzzleself.

        Example: '5x5', '3x4', 'reverse3x3'
        """
        return "NA"

    @classmethod
    def generateStartPosition(cls, variantid):
        """Returns a Puzzle object containing the start position.

        Outputs:
```

```
            - Puzzle object
        """
        raise NotImplementedError

    #################################################################
    # String representations
    #################################################################

    def toString(self, mode="minimal"):
        """Returns the string representation of the Puzzle based on the type.

        If mode is "minimal", return the serialize() version
        If mode is "complex", return the printInfo() version

        Inputs:
            mode -- "minimal", "complex"

        Outputs:
            String representation -- String"""

        if mode == "minimal" and hasattr(self, "serialize"):
            return self.serialize()
        if mode == "complex" and hasattr(self, "printInfo"):
            return self.printInfo()
        return "No string representation available"

    def __str__(self):
        """Returns the toString representation in "complex" mode

        Returns
        -------
        str
            self.toString(mode="complex")
        """
        return self.toString(mode="complex")

    #################################################################
    # Gameplay methods
    #################################################################

    def primitive(self):
        """If the Puzzle is at an endstate, return PuzzleValue.SOLVABLE or
PuzzleValue.UNSOLVABLE
        else return PuzzleValue.UNDECIDED

        PuzzleValue located in the util class. If you're in the puzzles or solvers directory
        you can write from ..util import *

        Outputs:
            Primitive of Puzzle type PuzzleValue
        """
        raise NotImplementedError
```

```python
def doMove(self, move):
    """Given a valid move, returns a new Puzzle object with that move executed.
    Does nothing to the original Puzzle object

    NOTE: Must be able to take any move, including `undo` moves

    Raises a TypeError if move is not of the right type
    Raises a ValueError if the move is not in generateMoves

    Inputs
        move -- type defined by generateMoves

    Outputs:
        Puzzle with move executed
    """
    raise NotImplementedError

def generateMoves(self, movetype="legal"):
    """Generate moves from self (including undos)

    Inputs
        movetype -- str, can be the following
        - 'for': forward moves
        - 'bi': bidirectional moves
        - 'back': back moves
        - 'legal': legal moves (for + bi)
        - 'undo': undo moves (back + bi)
        - 'all': any defined move (for + bi + back)

    Outputs:
        Iterable of moves, move must be hashable
    """
    raise NotImplementedError

#################################################################
# Solver methods
#################################################################

def __hash__(self):
    """Returns a hash of the puzzle.
    Requirements:
    - Each different puzzle must have a different hash
    - The same puzzle must have the same hash.

    Outputs:
        Hash of Puzzle -- Integer

    Note: How same and different are defined are dependent on how you implement it.
    For example, a common optimization technique for reducing the size of key-value
    pair storings are to make specific permutations of a board the same as they have
    the same position value (i.e. rotating or flipping a tic-tac-toe board).
```

```
        In that case, the hash of all those specific permutations are the same.
        """
        raise NotImplementedError

    @property
    def numPositions(self):
        """Returns the max number of possible positions from the solution state.
        Main use is for the progressbar module.
        Default is unknown length, can be overwritten
        """
        return None

    def generateSolutions(self):
        """Returns a Iterable of Puzzle objects that are solved states.
        Not required if noGenerateSolutions is true, and using a CSP-implemented solver.

        Outputs:
            Iterable of Puzzles
        """
        return []

    #################################################################
    # Player methods
    #################################################################

    def playPuzzle(self, moves):
        """Default playPuzzle method uses indices to chose which
        move to play."""

        print("Possible Moves:")
        for count, m in enumerate(moves):
            print(str(count) + " -> " + str(m))
        print("Enter Piece: ")
        index = int(input())
        if index == '':
            return "BEST"
        elif index >= len(moves):
            return "OOPS"
        else:
            return moves[index]

    #################################################################
    # Number representation
    #################################################################

    def __add__(self, other):
        """Equivalent to doMove, can only add moves together

        Parameters
        ----------
        other : "Move"
            Custom defined Puzzle move
```

```
        Returns
        -------
        Puzzle
            Puzzle instance with move executed
        """
        return self.doMove(other)

    def __radd__(self, other):
        """Reverse add (same as __add__)

        Parameters
        ----------
        other : "Move"
            Custom defined Puzzle move

        Returns
        -------
        Puzzle
            Puzzle instance with move exectuted
        """
        return self.doMove(other)

    def __repr__(self):
        return "<{} object with {}>".format(self.__class__.__name__,
self.toString(mode="minimal"))

    #################################################################
    # Depreciated methods
    #################################################################

    def printInfo(self):
        """Prints the string representation of the puzzle.
        Can be custom defined"""

        return str(self)
```

A.3 ServerPuzzle Base Class

This is the ServerPuzzle Base Class for specific Puzzles to support server functionality.

```
from ...util import PuzzleException, classproperty, depreciated
from . import Puzzle

class ServerPuzzle(Puzzle):

    #################################################################
    # Variants
    #################################################################

    @classproperty
    def variants(cls):
        """A Collections object that holds all the supported variants
        that a Puzzle will support.
        """
        return {}

    @classproperty
    def test_variants(cls):
        """
        Same as variants, except for testing purposes
        """
        return {}

    #################################################################
    # Deserialization
    #################################################################

    @classmethod
    def fromString(cls, positionid):
        """Returns a Puzzle object based on "minimal"
        String representation of the Puzzle (i.e. `toString(mode="minimal")`)

        Example: positionid="6-1-0" for Hanoi creates a Hanoi puzzle
        with two stacks of discs ((3,2) and (1))

        Must raise a TypeError if the positionid is not a String
        Must raise a ValueError if the String cannot be translated into a Puzzle

        NOTE: A String cannot be translated into a Puzzle if it leads to an illegal
        position based on the rules of the Puzzle

        Inputs:
            positionid - String id from puzzle, serialize() must be able to generate it

        Outputs:
```

```
        Puzzle object based on puzzleid and variantid
    """
    if hasattr(cls, "isLegalPosition"):
        if not isinstance(positionid, str):
            raise TypeError("PositionID must be type str")
        if not cls.isLegalPosition(positionid):
            raise ValueError("PositionID could not be translated into a puzzle")
    if hasattr(cls, "deserialize"):
        return cls.deserialize(positionid)
    raise NotImplementedError

#################################################################
# Depreciated Methods
#################################################################

@depreciated("serverPuzzle.serialize is depreciated. See serverPuzzle.fromString")
def serialize(self):
    """Returns a serialized based on self

    Outputs:
        String Puzzle
    """
    return str(self)

@classmethod
@depreciated("serverPuzzle.deserialize is depreciated. See puzzle.toString")
def deserialize(cls, positionid):
    """Returns a Puzzle object based on positionid

    Example: positionid="3_2-1-" for Hanoi creates a Hanoi puzzle
    with two stacks of discs ((3,2) and (1))

    Inputs:
        positionid - String id from puzzle, serialize() must be able to generate it

    Outputs:
        Puzzle object based on puzzleid and variantid
    """

    raise NotImplementedError

@classmethod
@depreciated("isLegalPosition is depreciated")
def isLegalPosition(cls, positionid, variantid=None):
    """Checks if the positionid is valid given the rules of the Puzzle cls.
    This function is invariant and only checks if all the rules are satisified
    For example, Hanoi cannot have a larger ring on top of a smaller one.

    Outputs:
        - True if Puzzle is valid, else False
    """
    raise NotImplementedError
```

A.4 Hanoi Puzzle implementation

This is an implementation of Hanoi demonstrating an example implementation of a Puzzle.

```
"""Game for Tower of Hanoi
https://en.wikipedia.org/wiki/Tower_of_Hanoi
"""

from copy import deepcopy
from . import ServerPuzzle
from ..util import *
from ..solvers import IndexSolver

def ffs(num):
    """Helper function to return the index of the LSB.
    For the 0 case, return `float('inf')`
    """
    output = (num & -num).bit_length() - 1
    output = output if output != -1 else float('inf')
    return output

class Hanoi(ServerPuzzle):

    id      = 'hanoi'
    auth    = "Anthony Ling"
    name    = "Towers of Hanoi"
    desc    = """Move smaller discs ontop of bigger discs.
        Fill the rightmost stack."""
    date    = "April 2, 2020"

    variants =  ["2_1"]
    variants += ["3_1", "3_2", "3_3", "3_4", "3_5", "3_6", "3_7", "3_8"]
    variants += ["4_1", "4_2", "4_3", "4_4", "4_5", "4_6"]
    variants += ["5_1", "5_2", "5_3", "5_4"]

    test_variants = ["3_1", "3_2", "3_3"]

    def __init__(self,  variantid=None, variant=None):
        """Returns the starting position of Hanoi based on variant first, then
        variantID. By default it follows "3_3"

        Inputs
            - (Optional) variantid: string
            - (Optional) variant: dict

        Outputs
            - A Puzzle of Hanoi
        """
        self.rod_variant = 3
```

```
        self.disk_variant = 3
        if variant:
            if not isinstance(variant, dict):
                raise TypeError("Variant keyword argument is not of type dict")
            if "rod_variant" not in variant:
                raise ValueError("Variant keyword argument does not contain rod_variant")
            if "disk_variant" not in variant:
                raise ValueError("Variant keyword argument does not contain disk_variant")
            self.rod_variant, self.disk_variant = variant["rod_variant"],
variant["disk_variant"]
        elif variantid:
            if not isinstance(variantid, str):
                raise TypeError("VariantID is not of type str")
            strlist = variantid.split("_")
            if len(strlist) != 2:
                raise ValueError("Invalid variantID")
            self.rod_variant = int(strlist[0])
            self.disk_variant = int(strlist[1])

        self.rods = [2 ** self.disk_variant - 1] + [0] * (self.rod_variant - 1)

    @property
    def variant(self):
        """Returns the variant of the Puzzle

        Outputs:
        - Variant : str
        """
        return "{}_{}".format(self.rod_variant, self.disk_variant)

    @property
    def numPositions(self):
        """Returns the upperbound number of possible hashes

        Outputs:
        - numPositions : int
        """
        return self.rod_variant ** self.disk_variant

    def __hash__(self):
        """Returns the reduced hash of the Puzzle

        Outputs:
        - hash : int
        """

        # Except for the last rod, sort all the rods in descending order by size
        rodscopy = self.rods[:-1]
        rodscopy.sort(reverse=True)

        # Hash calculation is the sum of the:
        #   rod index of a disk * rod_variant ** disk size
```

```
        # over all disks
        output = 0
        for i in range(len(rodscopy)):
            rod = rodscopy[i]
            j = 0
            while rod != 0:
                mod = rod % 2
                output += mod * (i + 1) * self.rod_variant ** j
                j += 1
                rod = rod >> 1
        return output

    def toString(self, mode="minimal"):
        """Returns the string representation of the Puzzle based on the type.

        If mode is "minimal", return the serialize() version
        If mode is "complex", return the printInfo() version

        Inputs:
            mode -- "minimal", "complex"

        Outputs:
            String representation -- String"""

        if mode == "minimal":
            return "-".join([str(rod) for rod in self.rods])
        elif mode == "complex":
            output = ""
            for j in range(self.disk_variant):
                for rod in self.rods:
                    output += " " * 3
                    if (rod >> j) % 2 == 1: output += chr(j + 65)
                    else: output += "|"
                output += "\n"
            output += "----" * (self.rod_variant) + "---\n"
            output += "   " + "   ".join(str(i) for i in range(0, self.rod_variant))
            return output
        else:
            raise ValueError("Invalid keyword argument 'mode'")

    @classmethod
    def fromString(cls, positionid : str):
        """Returns a Puzzle object based on "minimal"
        String representation of the Puzzle (i.e. `toString(mode="minimal")`)

        Example: positionid="6-1-0" for Hanoi creates a Hanoi puzzle
        with two stacks of discs ((3,2) and (1))

        Must raise a TypeError if the positionid is not a String
        Must raise a ValueError if the String cannot be translated into a Puzzle

        NOTE: A String cannot be translated into a Puzzle if it leads to an illegal
```

```
        position based on the rules of the Puzzle

        Inputs:
            positionid - String id from puzzle, serialize() must be able to generate it

        Outputs:
            Puzzle object based on puzzleid and variantid
        """
        if not isinstance(positionid, str):
            raise TypeError("PositionID is not type str")

        rod_strings = positionid.split("-")
        if not rod_strings:
            raise ValueError("PositionID cannot be translated into Puzzle")

        try:
            rods = [int(rod) for rod in rod_strings]
        except ValueError:
            raise ValueError("PositionID cannot be translated into Puzzle")

        sum_rods = sum(rods) + 1
        if sum_rods & -sum_rods != sum_rods:
            raise ValueError("PositionID cannot be translated into Puzzle")

        newPuzzle = Hanoi(variant={
            "rod_variant" : len(rods),
            "disk_variant" : sum_rods.bit_length() - 1})
        newPuzzle.rods = rods
        return newPuzzle

    def __repr__(self):
        """Returns the string representation of the Puzzle as a
        Python object
        """
        return "Hanoi(board={})".format(self.toString())

    def primitive(self):
        """If the Puzzle is at an endstate, return PuzzleValue.SOLVABLE or
PuzzleValue.UNSOLVABLE
        else return PuzzleValue.UNDECIDED

        PuzzleValue located in the util class. If you're in the puzzles or solvers directory
        you can write from ..util import *

        Outputs:
            Primitive of Puzzle type PuzzleValue
        """
        if self.rods[-1] != 2 ** self.disk_variant - 1:
            return PuzzleValue.UNDECIDED
        return PuzzleValue.SOLVABLE

    def doMove(self, move):
```

```
        """Given a valid move, returns a new Puzzle object with that move executed.
        Does nothing to the original Puzzle object

        NOTE: Must be able to take any move, including `undo` moves

        Raises a TypeError if move is not of the right type
        Raises a ValueError if the move is not in generateMoves

        Inputs
            move -- type defined by generateMoves

        Outputs:
            Puzzle with move executed
        """

        if not isinstance(move, tuple) and \
            len(move) != 2 and \
            isinstance(move[0], int) and \
            isinstance(move[1], int):
            raise TypeError("Invalid type for move")

        if move not in self.generateMoves():
            raise ValueError("Move not possible")

        newPuzzle = Hanoi(variantid=self.variant)
        rods = self.rods.copy()

        lsb_index = ffs(rods[move[0]])
        assert lsb_index != float('inf')
        rods[move[0]] = rods[move[0]] - (1 << lsb_index)
        rods[move[1]] = rods[move[1]] + (1 << lsb_index)
        assert sum(rods) == 2 ** self.disk_variant - 1
        newPuzzle.rods = rods
        return newPuzzle

    def generateMoves(self, movetype="all"):
        """Generate moves from self (including undos).
        NOTE: For Hanoi, all moves are bidirectional, so movetype doens't matter

        Inputs
            movetype -- str, can be the following
            - 'for': forward moves
            - 'bi': bidirectional moves
            - 'back': back moves
            - 'legal': legal moves (for + bi)
            - 'undo': undo moves (back + bi)
            - 'all': any defined move (for + bi + back)

        Outputs:
            Iterable of moves, move must be hashable
        """
        moves = set()
```

```
        rods = list(map(ffs, self.rods))
        for i in range(len(rods)):
            for j in range(len(rods)):
                if rods[i] < rods[j]:
                    moves.add((i, j))
        return moves

    def generateSolutions(self):
        """Returns a Iterable of Puzzle objects that are solved states.
        Not required if noGenerateSolutions is true, and using a CSP-implemented solver.

        Outputs:
            Iterable of Puzzles
        """
        puzzle_string = "0-" * (self.rod_variant - 1)
        puzzle_string += str(2 ** self.disk_variant - 1)

        return [self.fromString(puzzle_string)]

    @classmethod
    def generateStartPosition(cls, variantid, variant=None):
        """Returns the starting position of Hanoi based on variant first, then
        variantID. Follows the same functionality as __init__

        Inputs
            - (Optional) variantid: string
            - (Optional) variant: dict

        Outputs
            - A Puzzle of Hanoi
        """
        return Hanoi(variantid, variant)
```

A.5 Solver Base Class

This is the base Solver class for all Solvers in GamesmanPuzzles.

```
#These are general functions that you might want to implement if you are to use the
PuzzlePlayer
from ..util import *

class Solver:

    def __init__(self, puzzle, **kwargs):
        """Creates a Solver object initialized with puzzle

        Inputs
        puzzle -- the puzzle to be solved on
        """
        raise NotImplementedError

    def solve(self, *args, **kwargs):
        """Solves the puzzle initialized in the init function
        """
        raise NotImplementedError

    def getRemoteness(self, puzzle, **kwargs):
        """Finds the remoteness of the puzzle

        Inputs:
        puzzle -- the puzzle in question

        Outputs:
        remoteness of puzzle
        """
        raise NotImplementedError

    # Built-in functions
    def getValue(self, puzzle, **kwargs):
        """Returns solved value of the puzzle

        Inputs
        puzzle -- the puzzle in question

        Outputs:
        value of puzzle
        """
        remoteness = self.getRemoteness(puzzle, **kwargs)
        if remoteness == PuzzleValue.MAX_REMOTENESS: return PuzzleValue.UNSOLVABLE
        return PuzzleValue.SOLVABLE
```

A.6 Code to Generate Solver Graphs

This is the code to generate the Figure 12 graphs.

```
from puzzlesolver.puzzles import Hanoi
from puzzlesolver.solvers import GSolver, SQLSolver, ISolver, PSolver

import time
variants = ["3_%i" % i for i in range(1, 11)]
pos_num = [Hanoi.generateStartPosition(variantid=variant).numPositions for variant in
variants]
def timeit(solve_cls, dir_path="/tmp/puzzles/"):
    arr_handle = []
    for variant in variants:
        print("Solving variant: %s " % variant, end="")
        start = time.time()
        puzzle = Hanoi.generateStartPosition(variant)
        if dir_path:
            solver = solve_cls(puzzle, dir_path=dir_path)
        else:
            solver = solve_cls(puzzle)
        solver.solve()
        length = time.time() - start
        arr_handle.append(length)
        print("Took %f seconds" % length)
    print("Done")
    return arr_handle
print("General")
general = timeit(GSolver, None)
print("SQL")
sql = timeit(SQLSolver)
print("Index")
index = timeit(ISolver)
print("Pickle")
pickle = timeit(PSolver)
import matplotlib.pyplot as plt
plt.loglog(pos_num, general, label="General", color="blue")
plt.loglog(pos_num, sql, label="SQLite", color="red")
plt.loglog(pos_num, pickle, label="Pickle", color="green")
plt.loglog(pos_num, index, label="Index", color="orange")
plt.xlabel("Number of Positions")
plt.ylabel("Time Taken")
plt.title("Time Taken solving for a Number of Positions")
plt.legend();
pickle_size = [16, 28, 64, 172, 496, 1698, 5345, 16284, 55662, 173800]
index_size = [4, 12, 36, 108, 324, 872, 2916, 8748, 26244, 78732]
sql_size = [12288, 12288, 12288, 12288, 12288, 24576, 57344, 143360, 438272, 1323008]

import matplotlib.pyplot as plt
```

```
plt.loglog(pos_num, sql_size, label="SQLite", color="red")
plt.loglog(pos_num, pickle_size, label="Pickle", color="green")
plt.loglog(pos_num, index_size, label="Index", color="orange")
plt.xlabel("Number of Positions")
plt.ylabel("File size")
plt.title("File Size after solving for a Number of Positions")
plt.legend();
```

A.7 Server Puzzle Assignment (Design part)

This is the Design assignment given out to students during the 5-week tutorial.

```
# Server Puzzle Assignment (Design part)
Alright, now it's time to take the training wheels off and develop your own ServerPuzzle.
Your assignment is to design and develop a ServerPuzzle based on the tutorials and format
set up in GamesmanPuzzles.

Before developing the ServerPuzzle, you must visualize how your Puzzle would work. What
should be the default variant? How many positions must be hashed? How will the puzzle
progress?

This design process will be represented with a writeup. You must submit the writeup in PDF
form. Include these in your writeup:

- Your name/Team names
- Puzzle
    - Puzzle Name
    - Puzzle ID
        - Simple identifier of a Puzzle. (Example: 'hanoi')
    - Puzzle Visualization
        - A picture of the Puzzle.
        - Must match default Variant
    - Short Description of Puzzle
        - About 1-2 paragraphs
        - Should contain how to play and win.
        - State why you think it's a good addition to GamesmanPuzzles
- Position
    - Position representation (Check Example A below)
- Moves
    - The type of Legal moves in the Puzzle
        - Forward, Bidirectional, or Both
    - Move representation (Check Example A below)
        - Moves should be represented as a tuple with two entries.
        - You should represent complex entries as single numbers or letters.
- Variants (Must have at least two Variants, including the default Variant)
    - Variant Name
    - Number of possible positions
        - Also include calculation
    - A Default Variant should have a small minimum remoteness (5-20 moves) and be easy to
solve (10000 positions at max). You wouldn't have any problems solving it multiple times.
- (Optional) Optimization
    - Example topic: Reduced number of positions with Hash tricks

[//]: # "Submit your writeup in the shared Google Drive folder by the listed time
(10/21/20). The Google Drive link will be posted on Slack."

## Examples
```

````
### Example-A:
The Tower of Hanoi board can be represented in this String representation:
```
[[3, 2, 1], [], []]
```
A move can be represented as a tuple with Whole Numbers. For example, a move from the first rod to the second rod can be represented as:
```
(0, 1)
```
Another example is chess. A white knight move can be represented as
```py
("b1", "c3")
```
````

A.8 Server Puzzle Assignment (Develop part)

This is the Develop assignment given out to students during the 5-week tutorial.

```
# Server Puzzle Assignment (Develop part)
Now that you have a general idea of what kind of Puzzle you want to implement, it is time to
develop!. Similar to how you implemented Hanoi, implement your ServerPuzzle and follow the
[tutorial steps](../tutorial). You may refer to the already existing puzzles
([Hanoi](../../puzzlesolver/puzzles/hanoi.py)) for guidance.

### Testing
You are also responsible for implementing test sets following the format, located in
`GamesmanPuzzles/tests/puzzles/test_<your_puzzle_name here>`.

- `testHash()`
    - Tests the expected behavior of the hash function on the puzzle states.
- `testSerialization()`
    - Tests if serialization and deserialization works both ways.
- `testPrimitive()`
    - Tests if the start state and end state outputted the right primitives.
- `testMoves()`
    - Tests a specific scenario and checks if the moves inputted resulted in the expected
state, generated moves, and expected invalid moves.
- `testPositions()`
    - Tests the default start state and finish positions matches the expected
serializations.
- `testValidation()`
    - Tests four invalid serializations and checks if it raises an error.
- `testServerPuzzle()`
    - Tests server functionality by trying out a series of inputs.

 You are EXPECTED to take much inspiration from the [example test suite of
Hanoi](../../tests/puzzles/test_Hanoi.py).

 To run your tests, execute in the GamesmanPuzzles directory:
```
pytest --cov puzzlesolver
```
Submit this project by creating a pull request to the Master branch. Refer to
[Contributing](../Contributing.md) for more info.

### Additional Steps and Tips To Consider
- A real ServerPuzzle should not be using GeneralSolver as its main solver, as each request
for the remoteness of a position for our server would have the GeneralSolver solve the
puzzle. Consider using one of our persistence solvers like SqliteSolver or IndexSolver. The
hash used in the tutorial should NOT be used for IndexSolver.
- Files should be placed properly in their respected directories. Refer to [Where To Put My
Stuff](../wheretoputmystuff.md) for more info. You should also adjust your dependencies
based on the location of the file.
```