Development of a
Generalized Game-Playing Framework

A Technical Report

in STS 4600

Presented to

The Faculty of the

School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment of the Requirements for the Degree

Bachelor of Science in Computer Science

Tazz Stieglitz

June 7, 2014

On my honor as a University student, I have neither given nor received unauthorized aid on this assignment as defined by the Honor Guidelines for Thesis-Related Assignments

Signed: ____ Date: ___________

Approved: ____ Date: ___________

Table of Contents

Executive Summary

The initial goal of this project was to produce a program that would allow the user to define a game, and then play that game with some degree of skill. Initially, an implementation was planned that would begin by attempting to play the game optimally, but fall back to an imperfect, but still ideally "good" algorithm if it was determined that the optimal algorithm was too slow for that specific game. Over the course of development, it was determined that a fast algorithm for playing completely arbitrary games could not be designed to achieve the desired level of skill. In order to meet the initial goals, the program was redesigned to allow user-provided algorithms for playing the games. In order to ensure that the initial goals of optimal and "better-than-nothing" play were met, algorithms which meet those criteria were developed, but overall, the program has shifted in purpose from that of a generalized game solver to that of a framework for designing game-playing algorithms.

For testing and demonstration, two game descriptions were developed -- Feed the Pig, and Tic-Tac-Toe. These games are relatively simple, and serve to allow easy analysis of tests on a quick turnaround. Additionally, a custom controller designed specifically to play Feed the Pig was developed, as a proof-of-concept for that functionality. Several other controllers were developed as well, to showcase the feature. In addition to the aforementioned controllers for optimal play and quick "better-than-nothing" play, controllers were developed to provide completely random play, play based on user input, and play based on a statistical sampling of simulated games.

Source code for this project is available at htp://github.com/truefire/TPS_Undergrad_Thesis

Acknowledgements

I would like to thank the following people for their assistance throughout the development of this project:

- my technical advisor, Gabriel Robins, for his assistance in the thesis process, and for his theory of computation class, which provided the foundation for much of the work required to complete this project.
- my STS4600 advisor, Deborah Johnson, for her assistance with the written portions and logistics of the thesis process, and also for putting up with my delays towards the end of the process.
- my STS4500 advisor, Joanne Cohoon, for helping acquaint me with the undergraduate thesis project.
- my good friends Kim Lindblom, Henry Mozer, and Ethan Treff, for general programming help, and for listening to my constant complaints about obscure compiler errors.
- my parents, for their support and encouragement throughout my study.

This project made use of the following libraries:

- The C++ Boost libraries, available under the Boost Software License
  Boost: http://www.boost.org/
  License: http://www.boost.org/LICENSE_1_0.txt
- Don Clugston's FastDelegate library, available under the Code Project Open License 1.02
  FastDelegate: http://www.codeproject.com/Articles/7150/
  License: http://www.codeproject.com/info/cpol10.aspx

Glossary

Controller - Encapsulates an algorithm for controlling a specific player in a game.

Minimax - A decision algorithm that seeks to minimize the maximum potential loss.

Perfect Information Game - A game in which all players are aware of the same portion of the game state.

GDF - Game Description Format. A file-format developed for describing games.

CDF - Controller Description Format. A file-format developed for describing controllers.

Generator program - An executable which itself generates another executable to be run.

Move space - The set of all valid moves available to a player at a given point in a game.

Game space - The set of all possible move sequences for a given game.

State space - The set of all possible states a given game could ever be in.

Section I: Design Report

A. Introduction

The initial intent of this project, as stated in the executive summary, was to develop a "generalized board-game solver" -- more specifically, a program that could read a game description, provided in a then-undecided language, and produce a system for managing actual execution of the game, as well as an algorithm that could provide optimal moves. While the final result certainly fits this description, the project has also, in response to certain technical limitations, taken on an additional purpose. The final product serves as a framework, taking both descriptions of games and descriptions of algorithms for playing said games, and combining them together to actually execute the algorithms. In addition to providing the necessarily slow or inaccurate generalized algorithms that the project initially sought to develop, the program also allows users to develop their own algorithms which can, by virtue of specialization, outperform the provided algorithms.

B. Evaluation of Options

The goal of this project falls into a category of artificial intelligence programming known as general game playing. Among practical research projects in this problem domain, the General Game Playing (GGP) Project of the Stanford Logic Group stands at the forefront. In initial stages of development, the Game Description Language (GDL) developed by the GGP Project was considered for use in this project (Love, 2008, and General Game Playing, 2014). Ultimately, however, the decision not to use GDL was made based on the following reasons:

- GDL is not bound to a particular implementation. Efficient execution of GDL games thus requires some amount of tailoring of the implementation to the game description. As our implementation is static, this is impossible. By choosing a language that is inherently linked to a specific implementation, game descriptions are instead required to be tailored to the implementation.
- GDL is a declarative language. For optimization reasons, descriptions based on an imperative language are preferable, as it more closely resembles the actual

computational process, and allows game-description writers to more carefully optimize their game.

- GDL is verbose. As a minimalistic predicate language, its syntax is inconvenient for any kind of enumeration or similar _____________. This leads to large amounts of repetition in even the simplest GDL descriptions.

Besides GDL, one other potential alternative was found in the initial planning stages: a program called *Zillions of Games*. *Zillions*, however, also did not prove to be what was required for the project. *Zillions* also uses games defined in a declarative language, and falls prey to the same verbosity that GDL does. Finally, and most importantly, it lacks support for many features which are considered standard in most practical languages, such as arithmetic, functions, and non-Boolean variables. *Zillions* also lacks any "perfect" algorithm for all supported games, and lacks any means of implementing that algorithm. (Zillions, n.d.)

Other existing solutions tend to hold similar properties to GDL and *Zillions of Games*. Most notably, general game description languages tend to be declarative, choosing to prioritize ease of writing a game description over practical usability within a framework. For this reason, it was decided that a proprietary description language and execution framework would be developed.

C. Program Design

The program functions in a two-stage process. In the first stage, the generator program is fed a game description file in GDF format, and a controller description file, in CDF format (see Section II). After processing these two files, the generator produces source code for and compiles an entirely new program which will execute the game described with the controllers provided. The generator program itself is a windows binary executable (.exe) that can be invoked via the command line. The generated source code is C++ code, which is compiled into an executable using the g++ compiler.

This two-step approach was decided on after an initial prototype using scripting languages proved to be too inefficient. The two-step compilation process allows the overall efficiency of the program to be improved significantly by allowing all scripts within the game descriptions and controllers to run as native code, and by taking advantages of compiler optimizations provided by g++ (in particular, the -O2 optimization flag is used).

The generator program is relatively straightforward. The first step is parsing the input files into the internal data structure used to hold them. This is relatively simple -- as the GDF and CDF formats are both XML-based, most of the work is done by an XML parser. The second step is to generate the source files for the final program from this data. This is mostly text formatting, and very little of it is of any technical interest. Almost all of the complexity present in the generator program stems directly from the complexity required in the generated program.

The core of the program's functionality can be seen in the generated code's structure. A look into the files generated will provide a sufficient understanding of the code flow and general functionality of the program as a whole.

common.h

This header file is included at the top of every other file. It provides some utility methods and typedefs used elsewhere in the code. More importantly, however, it includes the utility code defined in the game description and controller description files so that they are available elsewhere.

controller.h

This file contains the implementations of each of the controllers defined in the CDF, as well as a mapping from the names of the controllers to their actual implementations, so that the user can select which controllers are used at runtime.

main.cpp

This file serves as the entry point for the program. It sets up the selected controllers and maintains the main execution loop and main game instance.

game.cpp and game.h

The bulk of the program's work is done in the game class, defined in these two files. Instances of this class hold data necessary to specify the current state of the game, such as board data and the ID of the currently active player. The class also provides methods for controlling the game state -- methods for executing each possible move and checking win conditions. The class also provides many maps and lists useful to controllers for enumerating the move space. Finally, the class provides utility methods to make writing controllers easier -- simple wrappers for actions such as validating a move and printing the current game state.

In the development of the game class, it was important to provide a means to not only access and modify the game state, but also to fully enumerate and explore the game space. This gave rise to two design decisions worth mentioning.

The first decision was the inclusion of the copy method. In early development, when only primitive controllers were being used, game was a singleton class. It quickly became apparent during the development of more sophisticated controllers that the ability to have multiple game instances for simulation purposes would be necessary. This led to a refactoring of the game class to allow for multiple instances. The copy method was added to provide a convenient means of branching the game state into possible subsequent states.

The second design decision involved the logging of game progress. Logging of moves, board state, and win conditions was initially done by the game itself, via standard output. This proved problematic, however, when a controller would try to simulate a game. Logs from the simulated games would pollute the standard output channel, both confusing the actual happenings of the game, and significantly slowing down the simulation. The chosen solution was to move all essential logging from the game description to the game class, and to add a flag that would indicate whether logging should be performed for a given game or not.

D. Unmet Goals

During the initial planning phase, it was decided that an algorithm would be developed in order to determine whether it would be practical to use a slow but perfect solution to a game, or if a faster but less accurate heuristic-based approach should be used. Over the course of development, however, it was found that this judgment was more easily made by humans than by an algorithm, so the feature was dropped. That said, the extensible nature of the system does allow for such an algorithm to be implemented if it were deemed necessary in future use of the program.

E. Conclusion

In conclusion, the developed program meets all project goals, and met all initial implementation goals other than the development of the algorithm-switching functionality. Additionally, the program was made significantly more extensible by allowing custom user-written game-playing algorithms. If any future work on this project were done, it would likely need not make any changes to the systems in place. Focus instead would likely be put on development of new game descriptions, more sophisticated controllers, or a more user-friendly front-end program that would communicate with the main program via standard IO.

Section II: Program Specification

A. GDF (Game Description Format) Specification.

Overview

GDF is an XML-based format for formal specification of a game's rules. GDF “scripts” are simply embedded C++ code.

Composition

The following tags are supported by the GDF format. The root node of a GDF file is a <game> tag. Tags within indented sections are sub-elements of the preceding non-indented tag.

<game [name]>: The entire game specification is wrapped in this tag
- *<boards>:* Wraps the list of boards
- *<players>:* Wraps the list of players
- *<playerInstances>:* Contains a list of player types – one for each participating player
- *<util>:* Define subroutines which can be used by other parts of the code
- *<win>:* A function which returns the winning pid (-1 if the game is still going, -2 for a draw)

<board [label]>: Defines a game board– a collection of related game info
- *<space>:* Specifies the valid values that can be placed in a board space
- *<width>:* Specifies the width of the board (for display purposes only)
- *<init>:* Provides initial values for each space in the board

<player [type]>: Defines a type of player
- *<moves>:* Wraps the list of this player-type's moves

<move [name]>: Defines a move that a player can make
- *<params>:* Wraps the list of parameters for this move
- *<validate>:* A function that determines whether the move is valid on given parameters
- *<execute>:* A function that alters the game state to perform the move

. *<param [name]>:* Defines a parameter to a move and wraps the possible arguments.

Formatting

PlayerInstances should be specified as a comma-delimited list of player names with each name surrounded by quotes. For example:

```
<playerInstances> "Standard","Standard","DiceRoller" </playerInstances>
```

The board space should be specified as a comma-delimited list of integers. For example:

```
<space>1,2,3,4,5</space>
```

Board width should be specified as a single integer, for example:

```
<width>5</width>
```

Initial board state should be specified as a comma-delimited list of integers. For example:

```
<init>0,0,0,1,0,0,0 </init>
```

Parameter values should be specified as a comma-delimited list of integers. For example:

```
<param name="x">1,2,3</param>
```

Code Details

The executor and validator of a move provide the body of a C++ function for either executing or validating the move. Although the internal implementation is a bit more subtle, both functions can be seen as taking integer arguments of the names specified in the <params> field of the move. The validator is required to return a Boolean value indicating whether or not the move is valid for the current game state with the given parameters. Within the function, the "this" keyword points to the game instance which is performing the move or validation. The executor functions should manage the turn order (by setting current_pid of the game object), since turn order may be dependent on moves executed.

The win section contains the body of a function that takes no parameters and returns an int. This int should be the pid of the winning player, -1 if the game has not yet terminated, or -2 in the case of a draw. The win condition is required to be checked by the controllers in between every move executed.

The util code will be embedded at the beginning of the program -- accessible to nearly all other code. For this reason, it is recommended to make use of namespaces to avoid conflicts with other code -- for example, the util functions provided by the controller descriptions.

Reserved Words

Certain words are reserved for internal use. Specifically, the namespaces "common", "fastdelegate", and "main_execution"; the class "game"; and the global function "main".

<u>B. CDF (Controller Definition File) Specification</u>

Overview

CGF is an XML based format for defining routines that control a player's move choices. CGF routines are simply embedded C++ code.

Composition

The following tags are supported by the CGF format. A GDF file does not contain one root node, but a *<controllers>* and *<util>* section.

<controllers>: Wraps the list of controllers
*<controller [type]>:*Defines a controller type with the given name.
<util>: Defines additional subroutines to be used by controllers.

Code Details

Each controller represents the body of a function that takes a game object as it's only parameter and returns void. The parameter is named _game. It is the responsibility of the controller to print the win message and end the program if it executes a move which puts the game in a terminal state. The win condition is required to be checked in between every move executed (even if executing a move results in a repeated turn for the same player).

The util code will be embedded at the beginning of the program -- accessible to nearly all other code. For this reason, it is recommended to make use of namespaces to avoid conflicts with other code -- for example, the util functions provided by the game description.

Reserved Words

Certain words are reserved for internal use. Specifically, the namespaces "common", "fastdelegate", and "main_execution"; the class "game"; and the global function "main".

C. Controller Descriptions for Developed Controllers

PlayerController:

Allows a human user (or another program, if piping is set up correctly) to pass instructions on which move to use via standard input. Move choices are passed in the following format:

move: *move_name [parameters]*.

RandomController:

Randomly chooses valid moves from the moveset available to the player. This particular implementation does not keep track of which moves have been validated – it simply generates and tries independent random moves until one is accepted.

AcyclicMinimax

Performs a brute-force minimax algorithm on the state space. This algorithm guarantees a forced when whenever possible. If no force wins are possible, but a forced draw is, the algorithm will guarantee a forced draw. The algorithm's behavior is undefined (other than that it will continue to choose valid moves) when a forced win or draw is not available. This controller does not do any cycle detection – to effectively detect cycles in such a simulation requires an inordinate amount of space or time, which significantly decreases the applicability of the algorithm for most cases.

Statistical

This controller simulates random games branching from each possible immediate move it could take. It chooses the move that maximizes the quantity (victories – defeats). This particular implementation spends 1 second per turn simulating games. This algorithm has a significant weakness in that it does not consider the goals of players in its simulations – all opponents – and even the controller itself – are assumed to act completely randomly. This leads to some bad moves, most notably moves that will give the opponent easy chances to force a win. This flaw is corrected in the StatSim controller.

StatSim

This controller uses an algorithm that can be seen as a combination of the Statistical and Minimax algorithms. The algorithm runs a simulation of a sample set of games for each potential move, and chooses the move that led to the highest win rate. Unlike in the Statistical controller, however, these simulations assume that all players are themselves running statistical simulations (with a smaller sample size, to prevent exponential increase of the number of simulations) to determine their moves.

D. Output Specifications

The following is an enumeration of the output that will be produced by the generated program. Output will be done via the standard output channel (stdout).

- When a player makes a move, output: “move: <pid> | <move name>(<param 1>, <param 2> …)\n”, followed by each of the boards in sequence
- A board is output as follows: “board: <board name> | <board width> |<board data> \n"
- Board data is output as a list of integers, in the format [v1, v2, v3, v4 …]
- Game completion will output “win: <winner pid> \n” or “draw\n”
- Other messages will be output with "msg: <message> \n"

Section III: User Manual

A. Overview

This section serves as a guide to the average user on operation of the program.

B. Using the program

Running the program is done in two steps. First, the generator program is invoked, with three arguments passed in: the filename of a game description file in GDF format, the filename of a controller description file in CDF format, and the name of the output file to be produced. For example:

```
> generator.exe TicTacToe.gdf Controllers.cdf playTicTacToe.exe
```

The second step is to invoke the generated program, passing in the name of the controller to be used in sequence, for each player. For example, the following will start the tic-tac-toe game with a human-controlled first player, and a random-choice second player (assuming PlayerController and RandomController were specified in the cdf file provided in step one).

```
> playTicTacToe.exe PlayerController RandomController
```

C. Writing GDF files

*A more formal specification of the GDF format can be found in Section II

Essential to writing a GDF file is knowledge of the structure of the game class. In particular, a GDF-writer should be aware of the following properties and methods:

- winner: An int that represents the pid of the game winner (-1 for non-terminal states, -2 for a draw)
- current_pid: An int value that represents the pid of the current player
- active: A bool which tells whether a game is the 'main game' of an execution session, or a simulated branch
- boards_[board name]: A pointer to the board of the specified name. Boards are stored as std::vector<int>;

Remember that the game description, not the controller or framework, is responsible for keeping track of turn order. Also, if you want controllers to be able to simulate your game, refrain from using unqualified output -- it is recommended that all non-essential output statements be wrapped in a conditional checking that the game is the currently active game (via game::active).

It is worth noting that a given game may have more than one possible GDF representation, and that not all encodings are equal. When writing a GDF, it is important to consider the intended use. Some encodings may be suited to brute-force solution, while others may provide more readable results for human users, for example.

D. Writing CDF files

*A more formal specification of the CDF format can be found in Section III.

In writing a CDF, one should be familiar with all the necessities of writing a GDF, as well as the following properties and methods:

- common::validator: A wrapper for validator functions. Can be invoked with operator().
- common::executor: A wrapper for executor functions. Can be invoked with operator().
- game::boards_map: A map from board names to (pointers to) boards themselves.
- game::boards_list: A vector<string> containing the names of every board.
- game::player_types: A string array containing the type of each player, indexed by pid.
- game::num_players: The number of players in the game (fixed).
- game::move_map: A map from player names to movesets (as vectors of move names).
- game::domain_map: A 2D map from player names and move names to possible values for move arguments. The result is a vector<vector<int>>, where the Nth internal vector lists possible values for the Nth argument.
- game::validator_map: A 2D map from player names and move names to the validator function for that move.
- game::executor_map: A 2D map from player names and move names to the executor function for that move.
- game::copy: Creates a copy of the current game state, and returns a pointer to that game object.
- game::win_check(): Evaluates win conditions, and returns the pid of the winner. (-1 for non-terminal state, -2 for draw)

- game::try_move(), execute_move(), validate_move(), validate_move_unsafe(): These method names are fairly straightforward. Each takes 3 parameters -- a player type, a move name, and arguments to the move, as an int[]. try_move, validate_move, and validate_move_unsafe each return a bool indicating whether the move was valid. validate_move_unsafe is faster than validate_move, but does not do any error checking, so it will likely cause an exception if passed malformed data or a move that doesn't exist.

References

Love, Nathaniel, et al. (2008.) General Game Playing: Game Description Language Specification. Retrieved from http://logic.stanford.edu/classes/cs227/2013/readings/gdl_spec.pdf

General Game Playing. (2014). Introduction to GDL. Retrieved from http://games.stanford.edu/index.php/intro-to-gdl

Zillions Development. (n.d.) "Can Zillions Support This Game?" FAQ. Retrieved from http://www.zillions-of-games.com/supportedFAQ.html

Everyone. (n.d.) Wikipedia. Retrieved from http://en.wikipedia.org