Answer Set Programming in Proofdoku

Adam M. Smith
amsmith@ucsc.edu
UC Santa Cruz
Department of Computational Media

Abstract

Proofdoku is an AI-based game design that extends Sudoku. In addition to playing by the rules of the traditional logic puzzle, players must explain their reasoning. An AI system checks this reasoning and provides hints that guide the player to discover new reasoning patterns for themselves. Co-developing the game design and the AI system, implemented using the technology of Answer Set Programming (ASP), guided us to include features that depend on high-complexity combinatorial search and optimization. We developed Proofdoku to better understand the implications of designing and deploying game systems that depend on ASP for live interaction. This paper offers design tradeoffs and makes suggestions for future deployments of ASP-backed game designs.

Introduction

Proofdoku is an extension to the traditional Japanese logic puzzle Sudoku, developed within the practice of *AI-based game design* (Eladhari et al. 2011). The Proofdoku project aims to uncover new player experiences unreachable without the affordances of artificial intelligence (AI) systems as well as to understand how the context of a deployed game pushes back on those systems.

In Proofdoku, the player works under the traditional rules of Sudoku with one key twist: they must explain their reasoning (and it must be valid). A small AI system, built using the technology of answer set programming (ASP) (Gebser et al. 2012) and co-developed with the design of the game, plays several roles during gameplay including checking the validity of player arguments and computing hints for the various phases of play. Through interaction with Proofdoku, players can learn and generalize new deductive inference patterns for Sudoku (including those unknown to the designers).

The Proofdoku project was initiated to answer specific questions about applying ASP: What practical engineering concerns arise when deploying an ASP-backed gameplay experience? Which aspects of live play might be enabled or enhanced using ASP? These questions are answered here in necessarily game-specific ways. However, the concerns that arose in design, development, and deployment touch on much broader issues: maintaining responsiveness of the AI system for players; the level of project-specific engineering; software licensing; and monetary costs associated with centralization.

This paper makes the following contributions:

- We describe an AI-based puzzle game in which ASP is used to enable several aspects of the player experience.
- We show how hinting systems and other feedback mechanisms emerge from the architectural surplus of ASP.
- We compare of ASP deployment alternatives in the concrete context working within a modern web browser.
- We make suggestions for caching and network architecture that overcome many of the costs associated with provisioning a centralized solving service.

Background

We first review some context for the Proofdoku project.

AI-based Game Design

In introducing the practice of AI-based game design, Eladhari et al. (2011) argue that “the development of innovative artificial intelligence systems plays a crucial role in the exploration of currently unreachable [game design] spaces.” AI-based game designs are those that have an AI system closely integrated into the core mechanics, aesthetics and story. For example, *Prom Week* (McCoy et al. 2013) is a game that intimately depends on the CiF (McCoy et al. 2010) AI system for social physics modeling. In *Sarah and Sally* (Černý 2015), an action planning system allows for cooperative puzzle solving in the context of a single-player game. In Proofdoku, the player manipulates sets of evidence (selected cells on the board) that they think will convince the AI system. In the commercial setting, *Third Eye Crime* (Moonshot Games 2014) and *The Sims* (Maxis 2000) are AI-based game designs that also revolve around understanding and manipulating AI-driven character behaviors.

In AI-based game design, co-development of an AI system and a game design allows for productive mutual influence: the affordances of the AI system suggest new game design options and the context of the game design push back with new requirements for the AI system. Proofdoku's design was shaped through several cycles of this influence.

Copyright © 2017, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

Answer Set Programming

ASP is a declarative logic programming paradigm oriented at solving complex (NP-hard) search and optimization problems (Gebser et al. 2012). ASP technology combines a modeling language used for describing domain-specific problems with domain-independent algorithms for solving them. In particular, ASP uses a Prolog-like modeling language called AnsProlog. Most answer-set solvers rely on the latest algorithms emerging from the SAT/SMT[1] literature.

We use the Clingo-5 system,[2] a modern ASP system with support for defining custom search heuristics, externally checked constraints interleaved with the search process, and hooks for scripting languages in the service of integrating the solvers with outside environments. Proofdoku makes use some key features in Clingo not found in non-ASP systems such as the SAT solver MiniSat (Eén and Sörensson 2003). For example, we rely on *disjunctive* ASP (Gebser, Kaufmann, and Schaub 2013) to express a search problem outside of the complexity class NP when searching over arguments the player might make. We combine this with optimization criteria to find the best argument. In hinting modes, we make use of *brave* and *cautious* reasoning modes (Eiter, Ianni, and Krennwallner 2009) to efficiently access the union and intersection of all possible solutions to a puzzle or arguments for a given conclusion.

ASP in Game Design Applications

ASP is not commonly used in game design applications. Against this background, proposals for using ASP to analyze game mechanics (Smith, Nelson, and Mateas 2010) or to model design spaces for procedural content generation (Smith and Mateas 2011) stand out. Both of these emphasize offline applications of ASP that avoid interactive contact with players, instead yielding design insights or generating datasets that impact the player experience later on, as in *Infinite Refraction* (Butler et al. 2015).

Butler et al. (2013) describe a system built on live interaction with ASP-based content generators and gameplay analyzers. However, this mixed-initiative design tool still emphasizes design-time use by a single user. Here, responsiveness is valued but not critical. Likewise, scalability to many users on diverse devices and networks is not relevant.

Compton et al. describe *Anza Island* (Compton, Smith, and Mateas 2012), an AI-based game design in which the player interacts with an ASP-driven agent by adding and removing constraints represented by collectible cards. While *Anza Island* breaks ground with novel gameplay, it does so at the level of a prototype. The Proofdoku project brings ASP-backed gameplay out of the lab where a broader external context can push back on the design of the AI system.

Game Design

In this section, we describe Proofdoku in game design and implementation terms, saving details of the AI system for the subsequent section.

[1]Boolean satisfiability or Satisfiability Modulo Theories

[2]https://potassco.org/clingo/

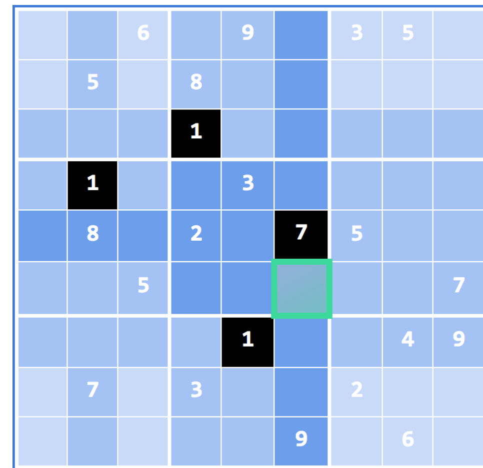

Figure 1: Initial state of a puzzle in Proofdoku. White numbers represent clue cells. Empty cells must be filled by the player by making arguments. Four cells of evidence (black background) are selected towards arguing for conclusion cell (green outline). Other background colors visualize local ambiguity: the number of different values that cell might take on across all evidence-consistent solutions (ignoring other clues). These four evidence cells are sufficient to argue that the conclusion cell must be filled with a 1.

Sudoku Basics

In traditional Sudoku puzzles, the player is faced with a 9×9 grid of cells. The cells will be filled with numbers from 1 to 9. Although many cells start blank, *clue* cells come prefilled, restricting the choices for the blank cells in such a way that there is one unique way to fill in the blanks. The player's job is to fill in the blanks so that each number is used exactly once in each row, once in each column, and once in each 3-by-3 sub-grid (sometimes called a cage).

Figure 1 shows the initial state of a relatively difficult Sudoku puzzle (as visualized within Proofdoku with an optimal opening argument displayed). Of the 81 total cells, 23 are provided as clues and 58 must be filled by the player.

Players access and play Sudoku puzzles in a variety of media including physical books and newspapers or digital apps and websites. Proofdoku seeks the attention of players who play the daily puzzles on websites such as New York Times[3] or USA Today[4] using desktop computers and mobile devices (smartphones and tablets).

Proofdoku Phases and Flow

As a digital adaptation of a traditionally pen-and-paper puzzle, play in Proofdoku moves through a number of phases of interaction with our game website.[5] The overall flow between these phases is visualized in Figure 2.

[3]https://www.nytimes.com/crosswords/game/sudoku/easy

[4]http://puzzles.usatoday.com/sudoku/

[5]http://proofdoku.com/

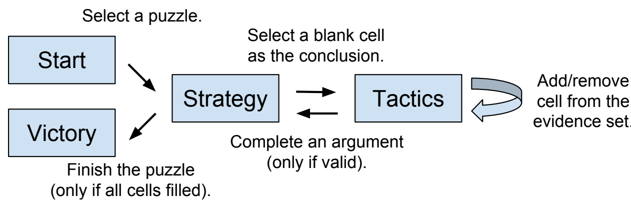

Figure 2: Gameplay phases in Proofdoku.

In the *Start* phase, the player selects a puzzle to play. Our game currently offers a fixed tutorial puzzle (accompanied by additional guidance not present on other puzzles) and fixed easy, medium, and hard difficulty puzzles. These are complemented with a small set of dynamic (updating daily) puzzles. We currently offer the daily easy, medium, and hard puzzles from the New York Times website as well as the USA Today puzzle of the day.

Upon selecting a puzzle, the player enters the *Strategy* phase. In this phase, the player needs to make a choice about which blank cell they will fill next. Selecting a blank cell registers it as the intended conclusion and carries the player to the next phase. Strategy choices can be made based on player intuition or guidance from our hint system. No strategy is invalid, but few choices in any given state set the player up for making an argument of reasonable (small) size.

In the *Tactics* phase, the player selects filled cells build the an evidence set—these are cells are the premises in the players argument. After each change, the game quickly responds to tell the player if their argument is valid. If so, the player may complete the argument, resulting in a transition back to the Strategy phase now with one more cell filled. Players may also abandon an argument and transition back to the Strategy phase without filling the conclusion. In this case, the game remembers the last state of their argument for the given conclusion so they can recover their attempt later.

Once all cells are filled, no more choices in the Strategy phase are available. In this case, the player transitions to the *Victory* phase where they see a visualization of all of the cells they have filled in using arguments. They player may have long forgotten which cells they filled versus were provided originally, so this reminds them of their achievement.

Evergreen Design and Sustainable Deployment

A key design goal in Proofdoku is evergreen design: the game should always feel fresh and up-to-date, even years after initial deployment with no recent developer attention. To support this, our puzzle selection emphasizes daily puzzles from outside sources as well as the ability to import puzzles from new sources that become popular in the future. Additionally, we aim to make system architecture choices consistent leaving the game deployed with near-zero running costs.

Proofdoku is built on Google App Engine[6] (GAE) and primarily implemented as a static webpage that can be served with zero application server instances. The daily puzzle selection represents a tiny amount of nearly static content (changing just once a day) served from a Cloud Storage bucket. These components of the game can be scalably accessed by millions of players per day while still falling well within the free tier of Google's cloud services.

Outside of the game's static webpage, additional logic is needed to keep the game fresh. For a few seconds per day, puzzle scraping scripts run to fetch the daily puzzles from remote sites. Beyond this, our AI system (a general purpose ASP web service described later) also runs on the same cloud platform in a way that fits into the free tier.

Arguments

Arguments for the value of a blank cell are a core concept in Proofdoku. An argument is defined with respect to a puzzle board and is composed of an evidence set (a subset of the filled cells) and a conclusion cell (one of the blank cells). An argument may be valid or invalid. An argument is valid if and only if any board configuration consistent with the evidence set agrees with the conclusion. Amongst valid arguments for a given conclusion, optimal arguments are those with the least possible amount of evidence (by cell count).

Feedback Mechanisms

In order to help the player share their reasoning with the game, the game tries to continually share its reasoning with the player. After each change to the evidence set (in the Tactics phase) we show how much the range of legal possibilities for every other cell was impacted. This can help the player spot long range implications as well as when adding a certain cell of evidence has no impact on a cell of interest.

For the conclusion cell, we give the player visual feedback as to whether their valid argument is optimal or not. They may proceed back to the Strategy phase with any valid argument, or they may continue searching to find an optimal argument if they like.

Hints

Despite having the same core rules as traditional Sudoku, Proofdoku's argument mechanic is generally unfamiliar to our players. As such, our hinting systems revolve around guiding the player through the argument making process rather than underlying Sudoku reasoning. Hints of the style provided in *SquareLogic* (TrueThought LLC, 2009) would compliment the hinting design currently used in Proofdoku.

In the Strategy phase, the player may want guidance for which cell to use as a conclusion. Our hint system (upon player request) will add a glinting animation to one of the blank cells that can be proven with the least amount of evidence (often there are several possibilities). Arguments involving fewer cells of evidence are often conceptually simpler than those that use more, but this is not always the case. There are a number of size-8 arguments (selecting all the cells but the conclusion in a row, a column, or a cage) that feel quite simple despite using more evidence than less structured size-6 arguments.

In the Tactics phase, we can suggest which cells must be included in an evidence set in order to be part of an optimal argument as well as those which should never be selected in

[6] `https://cloud.google.com/appengine/`

```
num(1..9).

cage(I,J,((((I-1)/3)*3)+((J-1)/3))+1) :- num(I); num(J).

1 { fill(I,J, D): num(D) } 1 :- num(I); num(J).
1 { fill(I,J, D): num(I) } 1 :- num(D); num(J).
1 { fill(I,J, D): num(J) } 1 :- num(I); num(D).
1 { fill(I,J, D): cage(I,J,C) } 1 :- num(C); num(D).

fill(I,J, D) :- clue(I,J, D).

#show fill/3.
```

Figure 3: `board.lp` defines the rules of Sudoku.

any optimal argument. If there is only one optimal argument (uncommon), this hinting design will reveal the full details of that argument. Otherwise, it will focus the player's attention away from irrelevant cells on the board.

Formulation in ASP

Our AI system performs several inference tasks in support of core gameplay, feedback mechanisms, and hinting systems in Proofdoku. In this section, we describe how these tasks are formulated in ASP. Each inference task in our system is resolved by a single execution of Clingo. An execution involves a program–a combination of a general problem encoding and a specific problem instance–and some command line arguments which alter the behavior of the solver.

board.lp

When the player starts a puzzle the game only knows about the set of clues that define the puzzle. Before transitioning out of the Start phase, we verify that the clues imply a unique solution for the board and record the details of that solution for later. This is a basic Sudoku solving task.

Our first program simply states the rules of Sudoku: there are numbers 1 to 9; each number is used exactly once in a row, column, and cage (where the structure of cages is defined by a formula); and if a cell has a clue, the number filled in that cell must match the clue. We store this general problem encoding in a file called `board.lp` (see Figure 3).

To find the initial solution to the player's puzzle selection, we wrangle the puzzle's clues into logical facts like `clue(2,4,9)` which says that the cell on row 2 and column 4 holds a clue value of 9. These facts define a specific problem instance. Combining `board.lp` with this instance, we execute Clingo with command line arguments asking for two distinct solutions. If the solver returns no solutions, the clue set was invalid. If the solver returns two solutions, the clue set was ambiguous. If just one solution is returned, it is because the solver has verified that there is a unique solution (the most common case).

Later during play in the Tactics phase, we want to check the validity of the player's argument as well as to visualize the ambiguity of every cell. We accomplish this by combining `board.lp` with a different problem instance. Instead of using all of the puzzle's known cells as clue facts, we include only those cells marked as evidence. Instead of asking for two distinct solutions, we as the solver to compute the union of all possible solutions with brave reasoning. If we group the resulting outputs for the `fill` predicate by cell we can compute the level of ambiguity as the count of different assignments for that cell. Examining the ambiguity of the conclusion cell, we can quickly see if there was only one legal possibility (hence if the argument was valid or not).

argument.lp

The program above considered just one given evidence set. A different approach must be used if we want to reason across all evidence sets. The first place this occurs is in the computation of hints in the Strategy phase. Of all possible valid arguments that could be made in a given board state, we want to highlight one of the conclusion cells associated with arguments with a least-size evidence set. The notion of argument optimality in Proofdoku is defined solely in `argument.lp` (not in the interaction and graphics code), so it could be locally altered to support alternative notions of simplicity or elegance of arguments.

To model the space of valid arguments, we want to say that there exists a selection of premises and conclusion such that for all legal ways of filling the Sudoku grid if the filling agrees with the premises then it must agree with the conclusion. Effectively, we want to the solver to show the following where p, c, and f are sets of assignments of cells to numbers representing the premises, conclusion, and any complete filling respectively.

$$\exists\, p, c\, \forall\, f : \text{Legal}(f) \rightarrow (p \subseteq f \rightarrow c \subseteq f)$$

The general problem of showing the truth of an $\exists\forall$ quantified boolean formula (known as 2-QBF) has complexity beyond the class NP, and thus it cannot be solved by tools such as SAT solvers which only cover NP. 2-QBF is in fact the canonical problem for the class Σ_2^P on the second level of the Polynomial Hierarchy. Disjunctive answer set solvers (such as Clingo) can express problems of this complexity, but only with considerable hassle for the programmer by applying the saturation technique introduced by Eiter (2009). Although conventional ASP wisdom suggests avoiding saturation encodings in favor of metaprogramming approaches as in `metasp` (Gebser, Kaminski, and Schaub 2011)—also applied game content generation (Smith and Butler 2013))—the rules of Sudoku are simple enough to express.

Our second answer set program, `argument.lp` (see Figure 4), uses choice rules to nondeterministically guess an argument and then a saturation-encoding restatement of the rules of Sudoku to express the nonexistence of solutions that would falsify the argument. Additionally, it states optimization criteria: one point of cost for each cell selected as a premise in the argument.

To compute hints in the Strategy phase, we wrangle the currently filled cells into `clued(I,J)` facts and the contents of the puzzle's previously-computed solution into `fill(I,J,D)` facts. We then execute Clingo with `argument.lp` and these facts with command line flags asking it to emit only solutions that are proven to be optimal. Upon entering the Tactics phase, we execute Clingo

```
num(1..9).
cage(I,J,((((I-1)/3)*3)+((J-1)/3))+1) :- num(I); num(J).
clued(I,J) :- clue(I,J,D).

1 { premise(I,J): clued(I,J) }.
1 { conclusion(I,J): num(I), num(J), not clued(I,J) } 1.

altfill(I,J,D): num(D) :- num(I); num(J).
bot :- altfill(I,J,D); premise(I,J); not fill(I,J,D).
bot :- altfill(I,J,D); conclusion(I,J); fill(I,J,D).
bot :- 2 { altfill(I,J,D): num(D) }; num(I); num(J).
bot :- 2 { altfill(I,J,D): num(I) }; num(D); num(J).
bot :- 2 { altfill(I,J,D): num(J) }; num(I); num(D).
bot :- 2 { altfill(I,J,D): cage(I,J,C) }; num(C); num(D).
altfill(I,J,D) :- bot; num(I); num(J); num(D).
:- not bot.

#minimize { 1,I,J: premise(I,J) }.
#show conclusion/2.
```

Figure 4: `argument.lp` defines valid arguments subject to minimizing of the size of the evidence set.

with a similar setup (now including the known conclusion as a fact) to find the size of an optimal argument for that specific conclusion.

Hints in the Tactics phase are computed as the intersection of all optimal evidence sets. With known cells and conclusion wrangled into facts as before, we execute Clingo with flags instructing it to perform cautious reasoning over optimal solutions.

Deployment Options

Even with the rest of the game deployed as a static webpage, there are several options for how to deploy our AI system. In this section, we explain our reasoning towards our chosen model: a centralized ASP web service.

Our game is intended to be played inside of a modern web browser with no additional downloads. We want gameplay to feel responsive on desktop machines as well as low-end smartphones. Although we expect our players to have network access, we appreciate that mobile players may not have reliable connectivity throughout the play session.

In-process

Clingo can be utilized as a software library via either its Python or C/C++ APIs. Used as a library, the solver would tightly bind with the rest of our game code running within the same (operating system) process. In the context of JavaScript code running in a web browser, we work with Clingo.js,[7] a cross-compilation from the Clingo's original C++ code into pure JavaScript using Emscripten.[8]

Until very recently, Clingo was distributed under the GNU Public License, a copyleft license that made it impractical to do in-process deployments of Clingo for commercial games. As of May 3, 2017,[9] Clingo is now distributed under the much more permissive MIT License.

Executing the solver locally in the player's browser has the strong benefit of not requiring any centralized infrastructure for solving combinatorial search and optimization problems. As the game scales in popularity, individual players pay the cost (mostly obviously in increased battery usage on mobile devices).

The pure-JS solver does not execute as efficiently as a native compiled solver. However, for the modest scale of inference tasks about Sudoku puzzles, we found response times in desktop web browsers to be reasonable.

Although an in-process deployment strategy is conceptually the simplest approach, it implies some responsiveness and stability hazards. While the solver is executing, no other scripts on the page can run to continue animations or respond to additional player inputs. JavaScript provides no thread-like abstraction to avoid this blocking while still executing within a single process. Additionally, if the browser decides to halt the seemingly unresponsive script, it is difficult for the rest of the game code to recover.

Out-of-process

In a desktop application (outside of the browser), it would be natural to address the limitations of the in-process approach by simply executing Clingo as a child process (by means of a `spawn` syscall). Again, no such abstraction is provided inside of the browser (the local filesystem and native executables cannot be touched).

The WebWorker API[10] provides the abstraction of an invisible background page that can execute JavaScript code asynchronously from the player-visible page. Workers communicate with the main page by passing messages. Clingo.js can be used from inside a web worker. Indeed, this was the approached used in the original Proofdoku prototype.

In either the in-process or out-of-process mode of using Clingo.js, the player must wait for the full 5.22 MB of Clingo.js to be downloaded before any AI-supported interactions with a puzzle can occur. Over a slow or unreliable mobile network, even this download time could be detrimental to the critical first moments of player experience.

On these same mobile devices, the responsiveness of the pure-JS solver is also significantly reduced. Even if we could hide the longer solving times with distracting animations, we wanted to avoid unnecessary consumption of battery resources. Our players should not have to decide between playing Proofdoku for a few more minutes or saving their battery for important communication later in the day.

Remote service

If we give up the scalability benefits associated with running the solver locally, we can regain control over responsiveness even for our players on mobile devices. Instead of running

[7] `https://potassco.org/clingo/run/`

[8] `http://emscripten.org/`

[9] `https://github.com/potassco/clingo/releases/tag/v5.2.0`

[10] `https://html.spec.whatwg.org/multipage/workers.html`

the solver on the player's device, we considered running it on (virtual) machines in the cloud.

In Proofdoku, we specifically examined using Google Cloud Functions[11] to implement a Clingo web service. In this design, a native compiled solver runs in response to web requests made by remote clients.

This approach avoids up-front downloads and allows much closer to native solving speeds (modulo virtualization and other overheads of the shared infrastructure). As a commercial service, this design implies per-CPU-second costs that threaten our intent to leave the game running for several years. To stay within the service's free tier as our player population scales up and down, we introduced layers of caches so that the solver is only run when necessary instead of in response to every single player action in a puzzle (details below).

In exchange for requiring the player's device to be able to make small, periodic requests to a central service, we are able offer both desktop and mobile players the same responsiveness from our AI system.

Inference Caching

As our game is oriented around a small set of fresh daily puzzles, most players will be asking our centralized service to solve the same inference tasks. Even at the scale of an individual player there is duplication as they explore different evidence sets. By caching the results of inference tasks, we can improve responsiveness over a design that needs to execute Clingo after every interaction.

Our caching strategy operates at the level of Clingo calls. In this setup, the caches store key–value mappings where the key is a combination of the AnsProlog text and command line flags, and the value is the text output of Clingo. This makes the system independent of Proofdoku and suitable even for non-game interactive ASP deployments.

Local caching

The first layer of caching works locally in the web browser. This cache provides an instant response when removing evidence returns the player to a previously-seen state in the Tactics phase or when abandoning an argument returns the player to a previously-seen state in the Strategy phase.

By nature, the local cache cannot help when the player encounters a state that has never been seen on their device before. Filling the local cache requires a network transaction. Although this local cache could be primed with the popular inference results of the day, we did not consider the complexity of this design to be warranted.

We used a Least-Recently-Used cache with 100 slots implemented using a plain JavaScript object as the key–value store. Most states are either seen again after just a few clicks or never again, so we selected the cache capacity primarily as a way to avoid unbounded memory usage. Although we could use the Web Storage API[12] to persist the cache across page loads or play sessions, our evergreen design intent leads us to not expect players revisiting old puzzles.

Global caching

As different players independently pass through similar states while they work through the puzzles of the day, we expect our solving service to answer many duplicate requests. Although Google Cloud Functions (GCF) will readily spawn several copies of Clingo to match the request load, this translates into costs for us to keep the game deployed. Our intent is to stay within the free tier of this service even as popularity of the game rises and falls.

We created a thin wrapper around the GCF Clingo service using Python running on GAE. This wrapper tries to answer requests out of a global memcache[13] service before handing the requests to the GCF Clingo service. The focus on daily puzzles ensures that relatively few keys are active each day (compared to the number seen during the whole deployment).

Although we do not have sufficient data to report informative cache hit rate statistics, we can report one interesting qualitative phenomenon resulting from the use of the global cache. Certain inference tasks (particularly computing the Strategy hints for a hard puzzle) can take even the native solver a few seconds to compute rather than a few tens of milliseconds to serve from memcache. Even if the global cache only manages to serve responses for queries about the initial state of a puzzle, the use of a centralized service with a global cache contributes significantly to game's feeling of responsiveness in the very first moments of interaction with a puzzle compared to uncached or decentralized inference.

Conclusion

Proofdoku is an AI-based game which draws out the otherwise unseen implications of building and deploying interactive systems based on ASP. Although many of the design decisions we made in the deployment of our game are specific to project and the intended player experience, the questions we faced in making those decisions are very general.

With the software license associated with a state-of-the-art answer set solver having recently changed to MIT (from GPL), a number of different modes of integrating ASP into commercial applications has just opened up in 2017. We hope the exploration of Proofdoku's design choices sparks informed conversations about how ASP can be deployed in games and other interactive media applications in the future.

Acknowledgements

The author would like to thank Nicholas Warren and Mason Reed for their game design and software engineering contributions to Proofdoku, the Potassco team at the University of Potsdam for providing Clingo liberally licensed and free of charge, and Google, Inc. for providing a generous free tier for the Google Compute Platform products.

[11] https://cloud.google.com/functions/

[12] https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API

[13] https://cloud.google.com/appengine/docs/standard/python/memcache/

References

Butler, E.; Smith, A. M.; Liu, Y.-E.; and Popovic, Z. 2013. A mixed-initiative tool for designing level progressions in games. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology*, UIST '13, 377–386. New York, NY, USA: ACM.

Butler, E.; Andersen, E.; Smith, A. M.; Gulwani, S.; and Popović, Z. 2015. Automatic game progression design through analysis of solution features. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, CHI '15, 2407–2416. New York, NY, USA: ACM.

Černý, M. 2015. Sarah and sally: Creating a likeable and competent ai sidekick for a videogame. In *Experimental AI in Games: Papers from the AIIDE 2015 Workshop*, EXAG 2. AAAI.

Compton, K.; Smith, A.; and Mateas, M. 2012. Anza island: Novel gameplay using asp. In *Proceedings of the The Third Workshop on Procedural Content Generation in Games*, PCG'12, 13:1–13:4. New York, NY, USA: ACM.

Eiter, T.; Ianni, G.; and Krennwallner, T. 2009. Reasoning web. semantic technologies for information systems. Berlin, Heidelberg: Springer-Verlag. chapter Answer Set Programming: A Primer, 40–110.

Eladhari, M. P.; Sullivan, A.; Smith, G.; and McCoy, J. 2011. Ai-based game design: Enabling new playable experiences. Technical Report UCSC-SOE-11-27, UC Santa Cruz, Baskin School of Engineering.

Eén, N., and Sörensson, N. 2003. An extensible sat-solver. In Giunchiglia, E., and Tacchella, A., eds., *SAT*, volume 2919 of *Lecture Notes in Computer Science*, 502–518. Springer.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2012. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan and Claypool Publishers.

Gebser, M.; Kaminski, R.; and Schaub, T. 2011. Complex optimization in answer set programming. *Theory and Practice of Logic Programming* 11(4-5):821–839.

Gebser, M.; Kaufmann, B.; and Schaub, T. 2013. Advanced conflict-driven disjunctive answer set solving. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, IJCAI '13, 912–918. AAAI Press.

McCoy, J.; Treanor, M.; Samuel, B.; Tearse, B.; Mateas, M.; and Wardrip-Fruin, N. 2010. Authoring game-based interactive narrative using social games and comme il faut. In *Proceedings of the 4th International Conference & Festival of the Electronic Literature Organization: Archive & Innovate (ELO 2010)*.

McCoy, J.; Treanor, M.; Samuel, B.; Reed, A. A.; Mateas, M.; and Wardrip-Fruin, N. y. 2013. Prom week: Designing past the game/story dilemma.

Smith, A. M., and Butler, E. 2013. Quantifying over play: Constraining undesirable solutions in puzzle design. In *In Proceedings of ACM Conference on Foundations of Digital Games*.

Smith, A. M., and Mateas, M. 2011. Answer set programming for procedural content generation: A design space approach. *IEEE Transactions on Computational Intelligence and AI in Games* 3(3):187–200.

Smith, A. M.; Nelson, M. J.; and Mateas, M. 2010. Ludocore: A logical game engine for modeling videogames. In *Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG 2010)*.