A Cloud-Based Path-finding Framework: Improving the Performance of Real-Time Navigation in Games

Jordan Rowe
Thrive Therapeutic Software
Nottingham, UK
mail@jordanneilrowe.co.uk

Amanda Whitbrook*
Department of Electronics,
Computing and Mathematics
University of Derby
Derby, UK
a.whitbrook@derby.ac.uk

Minsi Chen
School of Computing and Engineering
University of Huddersfield
Huddersfield, UK
m.chen@hud.ac.uk

ABSTRACT

This paper reviews current research in Cloud utilisation within games and finds that there is little beyond Cloud gaming and Cloud MMOs. To this end, a proof-of-concept Cloud-based Path-finding framework is introduced. This was developed to determine the practicality of relocating the computation for navigation problems from consumer-grade clients to powerful business-grade servers, with the aim of improving performance. The results gathered suggest that the solution might be impractical. However, because of the poor quality of the data, the results are largely inconclusive. Thus recommendations and questions for future research are posed.

ACM Reference Format:
Jordan Rowe, Amanda Whitbrook, and Minsi Chen. 2017. A Cloud-Based Path-finding Framework: Improving the Performance of Real-Time Navigation in Games. In *UCC '17: 10th International Conference on Utility and Cloud Computing Companion.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3147234.3148097

1 INTRODUCTION

The Cloud continues to be a focal point for research, yet there is limited literature available on the subject of Cloud utilisation within games; existing research typically targets Cloud gaming and Cloud MMOs. To this end, a proof-of-concept Cloud-based path-finding framework is proposed. This is a simple architecture, designed to compute the navigation routes for computer-controlled agents in the Cloud.

A further motivation for its development is the need to increase available resources to push the boundaries of game AI development, which is currently far from being able to outperform human experts [3], often as a result of a lack of memory. In addition, it is evident that real-time strategy (RTS) games are one of the more challenging genres to develop intelligent navigation for. This is a direct result of the tight constraints imposed, which include real-time and simultaneous moves, partial observability, nondeterministic play, and vast state space sizes.

The goal of this research is thus to evaluate the computational benefit of moving path-finding calculations from consumer-grade clients to powerful business-grade servers and to determine whether there is a potential application for Cloud based path finding within real-time games, i.e. can the Cloud perform at an acceptable level when the network bandwidth is assumed to be the largest variable affecting performance? Note that henceforth this paper refers to the server-computed path search as *Cloud Path-finding*, and any paths computed on the client as *Local Path-finding*.

1.1 RTS AI Research

There is a multitude of research on RTS AI covering the following topics as categorised by [3, 12]: adversarial planning [9, 11, 15], opponent modeling [17], decision making under uncertainty [12], spatial and temporal reasoning [3, 5, 12], resource management [4, 10, 13], collaboration [3, 12] and path-finding [1, 2, 7, 14]. Despite the diverse range of topics, none attempt to solve their problems using the Cloud, which provides additional rationale for the research conducted.

2 RELATED WORK

The only existing literature that proposes a concept similar to Cloud AI is [18]. This paper introduced the notion of a fast and scalable Cloud service to provide automated planning (AP) solutions. The rationale was to develop a system so that smartphones and other low-powered devices could utilise AP, with an Internet connection as the only requirement. The RESTful web service allows clients to connect to a Proxy & Scheduler, which distributes the available resources without exposing them to the technical details. The service is implemented using Planning Domain Description Language (PDDL) and 'Fast Downward' to accommodate the needs of researchers and engineers, and makes full use of Cloud facilities so that the Proxy & Scheduler can launch and destroy virtual hosts on demand. In comparison, the proposed Cloud AI differs in a few, but significant, ways:

- Connections are per request instead of per session;
- Data is transferred using Protocol Buffers - a faster and more lightweight format compared to the commonly used JSON [19];
- Only A* Search is specifically implemented, as opposed to the generic solution in [18], but this allows the data-intensive

*Corresponding author

UCC'17 Companion, December 5–8, 2017, Austin, TX, USA
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-5195-9/17/12…$15.00
https://doi.org/10.1145/3147234.3148097

algorithm to test the limits of the connection bandwidth - the expected largest factor in overall performance;
- Initial proof-of-concept does not utilise the Cloud; testing is carried out using a single server.

3 IMPLEMENTATION

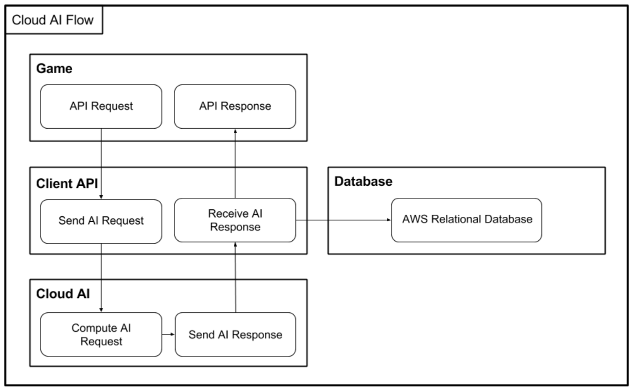

Figure 1: Cloud Path-finding concept flow chart

The Cloud Path-finding architecture is made up of four distinct parts; the server application, which responds to computer-controlled agents' requests, the client API, which is used to send requests to the server application, a simple test game that utilises the client API, and a database for storing analytics. Figure 1 shows the flow diagram for the architecture.

3.1 Server

The server application used to compute traversable paths is hosted on a free-tier Amazon Web Services (AWS) Elastic Compute Cloud (EC2) t2.micro virtual machine running Ubuntu Server 16.04 LTS with a single vCPU and 1GB RAM. It was chosen for its low cost and availability during the search for a suitable service. The application is developed from the ground up using C++14 for its speed and to ensure only the necessary components are implemented. To help handle networking logic the C++ Boost library is incorporated, making heavy use of Asio's TCP networking logic in particular - the protocol used between client and server; no additional networking logic is implemented. The data format of choice for communication between client and server is Google's Protocol Buffer [6]. This was chosen instead of, for example, JSON, because of its smaller data size, faster read/write speeds and generated source files, which are easier to use programmatically [6].

The logic for handling requests follows a simple four-step process:

(1) The client sends a request header, of exactly 7 bytes, containing the type of computation and the size of the request body, in bytes. For the research conducted the only available request header was for A* Search.
(2) The client then sends the request body containing all the data necessary to compute the requested calculation.
(3) Upon completing the calculation, the server sends a response header, of exactly 7 bytes, containing the type of computation and the size of the response body, in bytes.
(4) The server then sends the response body, which contains the results of the requested path calculation.

The request headers and bodies are serialised Protocol Buffers and are sent using the aforementioned TCP. For simplicity, for each new request received, a new thread is spawned to handle it and therefore the Cloud Path-finding is not currently designed as a scalable framework.

3.2 Artifical Intelligence

The popular A* search algorithm is sufficient for the purposes of this proof of concept; better search algorithms exist, but A* is simple to implement. Moreover, the overall performance is not the main concern of this paper, rather, it is the performance of the Cloud Path-finding against the Local Path-finding that is of interest. Therefore, implementing an easy search algorithm such as A* Search simplifies the design and development of the system.

To ensure high performance, the A* Search implemented on the server is highly modified to use as little data as possible. Each node in the grid has a unique ID, a heuristic, and a map containing pairs of IDs of neighbouring nodes and the cost to travel there. This is the minimal quantity of data necessary to perform A* Search in the Cloud.

3.3 Client API

The client has two main functions. The first is to act as the interface between the game and the server when sending A* Search requests. Cloud Path-finding requests made through the API are asynchronous. As part of the request, a callback must be provided, which is sent once the request has been completed. The second function of the API is to perform A* Search locally; this is a blocking call and only returns control once the calculation is complete. In both instances, data related to the calculation is gathered and stored in an AWS relational database for later analysis. The client was developed in C# and compiled as a DLL for use in other programs.

3.4 Test Game

A clone test game based on Spelunky and the SpelunkBots API [16] was created. Its path-finding computation was purposefully designed with the Cloud AI in mind and so it was expected to work seamlessly, as opposed to using an existing project which might have been incompatible or required significant modification to work effectively with the client API. An additional benefit of using a purpose-built clone is that it only includes the necessities, i.e. generating levels and basic AI to navigate them. The Spelunky clone was implemented using the MonoGame framework (http://www.monogame.net), which targets the .NET Framework 4.5 standard.

The level generation is computed in a similar manner to the original work, however, no enemies or objects are created. Reference [8] provides a detailed explanation of how Spelunky procedurally generates its levels. In contract, the following describes how levels are spawned for the Spelunky clone. Each level is comprised of several rows and columns of rooms, and to produce a variety of data, the number of rows and columns ranges from 4 to 10 rooms. Each room contains 8 rows by 10 columns of tiles, true to the original work. A selection of room templates with varying numbers of exits are hardcoded, each with a specific type associated with it:

- LeftRight - rooms with exits to the left and right;
- LeftRightTop - rooms with exits to the left, right and top;
- LeftRightBottom - rooms with exits to the left, right and bottom;
- All - rooms with exits in all directions.

The algorithm for generating levels works as follows:

(1) The number of rows and columns of rooms is randomly generated;
(2) A column in the top row is chosen at random to be the start location. This room is then given a room template of type LeftRight;
(3) A direction from the current room is randomly chosen from left, right, and down:
    - Left - if it is possible to move to the room to the left (it has not yet been set and is not out-of-bounds), move to the left and choose a room from the templates that contains an exit to the right;
    - Right - if it is possible to move to the room to the right, move to the right and choose a room from the templates that contains an exit to the left.
    - Below - there are two possibilities for rooms below. If the room below can be reached, move down and choose a room from the templates that contains an exit above; else, if the room below is out-of-bounds, the exit has been found.
(4) Repeat step 3 until the exit has been found;
(5) For the remaining unset rooms, randomly choose a room of any type.

Upon completing the level generation, five basic computer controlled agents are spawned into the level, and their only function is to choose a start and destination node randomly. This information is then used by the algorithm to calculate the path. Once the results are received, and if there is an acceptable path, they travel from the start to the destination, and then repeat the process. Every 20 seconds the entire level is destroyed and a new one is generated for the agents to navigate. Figure 2 shows the appearence of a typical level. There are three main elements to be seen:

- Dark grey tiles - these are the ground tiles;
- Light grey tiles - these are the waypoint tiles (used in the A* Search) that dictate where a computer-controlled agent can move to;
- Black tiles - these are the AI; there are five in the game that randomly navigate around.

The rationale for creating a clone of Spelunky was that the level generation provided a simple and scalable solution for generating levels of varying sizes and layouts. Extending the number of rows or columns, or designing more room templates was trivial once the base algorithm was in place. When the game executes, it continues to generate varying levels without any further work required, simplifying the process of data collection. Note that the levels are often larger than can be seen on screen, but the visualisation of the game is only for evaluation purposes to ensure that paths are correctly computed.

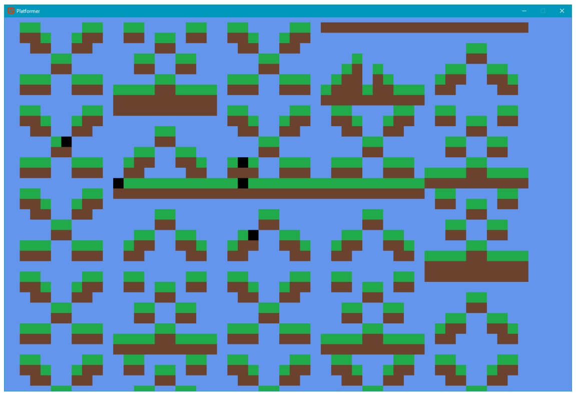

Figure 2: A randomly generated Spelunky-style level

3.5 Analytics

The data gathered by the client API needed storing for future analysis. It was not stored locally in a text file, as this could prove time consuming to merge and graph appropriately. As an existing service could not be found, an AWS relational database was set up as the central repository of data to store the following:

- Success - Whether a path was found or not;
- Sent Data - How much data the client transferred to the server. This refers to the request body size;
- Received Data - How much data the server sent back to the client. This refers to the response body size;
- Calculation Time - How long it took to calculate a path in the A* Search. For the server, this also includes the time taken to deserialise the protocol buffer containing the calculation data, but not the time to serialise the results;
- Total Time - How long the entire process of sending and receiving a request took. For the Local Path-finding, this is the same as the Calculation Time;
- Used Server - This indicates whether the Cloud Path-finding or Local Path-finding was used;
- Room Count - The number of rooms in the generated level.

A Microsoft Excel spreadsheet, which connects to the SQL Server database to pull down data, was used to help visual the results. In addition, IBM's Watson was used to generate the graphs found in Section 4.

3.6 Testing

The test game was run for 30 minutes to determine the performance of the Cloud Path-finding AI. This produced 90 randomly generated levels and 1,708 A* Search requests. For a comparison, the test game was run for another 30 minutes, this time utilising the Local Path-finding. This produced 90 randomly generated levels and 1,659 A* Search requests. The specification of the Local Path-finding (which also runs the test game), as well as a reminder of the Cloud configuration, can be seen in Table 1.

Table 1: Local and Cloud Hardware Specifications

Local Path-finding	Cloud Path-finding
Windows 10 Pro	Ubuntu Server 16.04 LTS
i7-4770K @ 3.5GHz	1 vCPU Intel Xeon @ 3.3GHz
16GB RAM	1GB RAM
200MiB/s NW	Low to Moderate NW Performance

A significant oversight with the testing process is acknowledged; the two systems possess different hardware, thus a direct comparison of the gathered data may seem illogical because of the impact of the hardware differences on the overall performance. It can be reasoned, however, that the data gathered is perfectly valid, and as such, a general comparison is still justifiable - the Local Path-finding is equivalent to a typical mid-to-high end 'gaming' PC; if the Cloud Path-finding cannot match or surpass its performance, then it can be surmised that Cloud Path-finding is not a practical solution, unless the server hardware is superior to the local hardware.

Another problem is that the only reasonable variables that can be used to compare the two path-finding implementations are the room count of the generated levels and the total calculation time. These can be analysed, but the results may be somewhat vague because room count is not a clear and comprehensible metric; a generated level with 10 rooms could have a different number of waypoints compared to another generated level with 10 rooms, because the room templates used are invariably different. A more suitable metric is the waypoint count, because this is an accurate and unambiguous value.

4 RESULTS

4.1 Cloud Path-finding Performance

Figure 3 shows the average total time to send, compute, and receive the results for a request, and the average calculation time taken plotted against the sent data size (in bytes) in the Cloud. The *x*-axis values are grouped together in steps of 15,646 bytes for clarity. Note that the calculation time in the Cloud includes the time taken to deserialise and compute A* Search.

It is clear to see that the Cloud Path-finding quickly exceeds the 16 milliseconds frame time required for 60 FPS. Granted, this does not affect the game performance itself, but it does affect the responsiveness of the computer-controlled agents; as the game generated larger rooms, the computation in the Cloud rapidly approached a 1 second total time, which is far too slow for the high responsiveness demanded in real-time games. From 0 - 15,646 to 62,587 - 78,233, the total average time increased by 1350%, whereas the sent data size only increased by about 500%.

Table 2 provides a side-by-side comparison of the average total time against the average calculation time. The table shows that it is the calculation time that is responsible for the poor Cloud AI performance, and not the time taken to send or receive data. Serialisation and deserialisation are known to be a slow process, however, the calculation time includes both deserialising the data and computing A* Search. Thus, an improvement to the experimental set-up would be to time each operation individually so that empirical evidence can exactly determine the main offender.

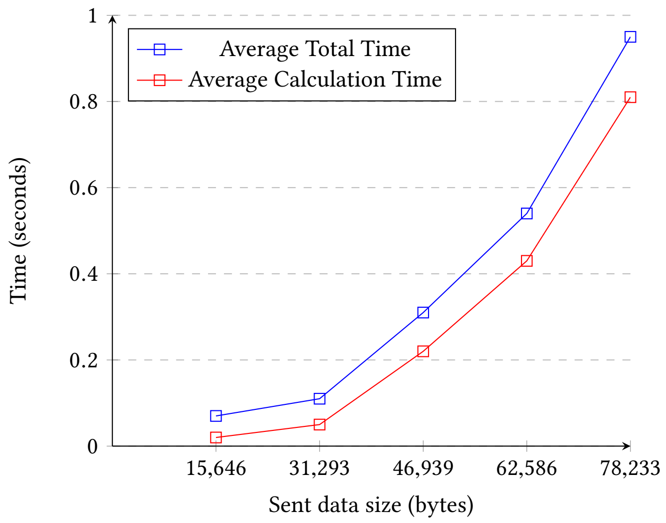

Figure 3: Cloud Path-finding average total time and calculation time against sent data size

Table 2: Cloud Path-finding performance comparison

Sent Data Size	Avg Calculation Time	Avg Total Time
0 - 15,646	0.02	0.07
15,647 - 31,293	0.05	0.11
31,294 - 46,939	0.22	0.37
46,940 - 62,586	0.43	0.54
62,587 - 78,233	0.81	0.95

4.2 Cloud vs Local

Figure 4 shows the previous Cloud Path-finding average total time and average calculation time, this time plotted against the room count. Figure 5 shows the Local Path-finding average total time (which is synonymous with the average calculation time) also plotted against room count. As expected, for both the Local and Cloud computation, there is an increase in the average total time as the size of the rooms rises; more rooms generally means more waypoints, thus more time is required to compute A* Search, and more data needs to be sent and received by the Cloud server. It is interesting to note that the slowest Local search time (100 rooms; 8,000 tiles, 0.028 seconds), is significantly faster than the fastest Cloud Path-finding time (16 rooms; 1,280 tiles, 0.08 seconds).

5 CONCLUSION & FUTURE WORK

Real-time strategy games provide a complex and challenging environment for real-time navigation research. This paper focused on performance, which is an area of RTS path finding that has been overlooked as merely a by-product of current research. The aim of the work was to determine the practicality of a proof-of-concept Cloud based AI framework with the goal of improving real-time path finding performance by relocating A* Search computations to a server. The results obtained suggest that such a solution is not currently practical, but the evidence is weak considering the local and server hardware differences and the ambiguity of the room

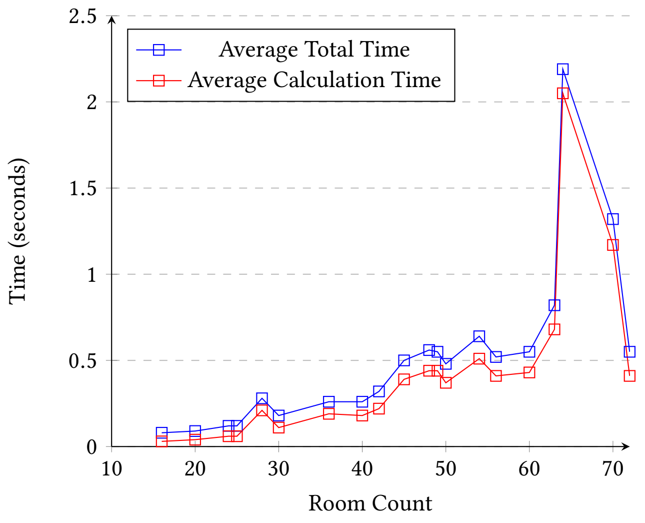

Figure 4: Cloud Path-finding average total time and calculation time against room count

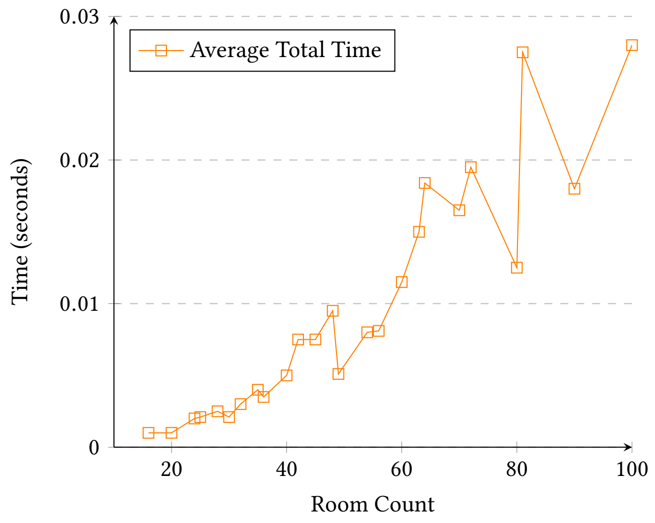

Figure 5: Local Path-finding average total time

count metric. More could be concluded from a set of results if these issues were addressed (see Section 5.1).

5.1 Future Research Recommendations

This paper acknowledges a weakness in the experimental design process which led to inconclusive results. Thus, the most prominent recommendation is to use waypoint count (the number of nodes used in the A* Search calculation, i.e. the number of tiles sent to the server), instead of the ambiguous room count. This would enable a more direct comparison between both systems to be achieved.

For a direct performance comparison both systems should ideally be run on the same hardware. In hindsight, it was unnecessary to design and develop the client API and test game for the Windows platform. Two AWS EC2 virtual machines could have been set up, one running the Cloud based computation as before, and the other making use of a combination of Local and Cloud computation, as was the case for the Windows platform. More reliable conclusions could be drawn using this set up, and by incorporating the previous recommendation.

Two trivial improvements remain. First, the test game should be redesigned to use the exact same generated levels and paths for both systems, instead of leaving it to chance with random number generation. Second, an analysis of the granularity between data transfer to the Cloud and the computation time of A* search would enable a more precise quantification of the true cause of Cloud based computation performance.

A final improvement for future research is to explore the scalability of the framework; it was hypothesised that the current thread-per-connection approach would scale poorly. Examining this empirically and devising a scalable Cloud based computation solution would thus be worthwhile. The scalability of the computation should concern both the size of the map as well as the number of computer-controlled agents.

REFERENCES

[1] Zeyad Abd Algfoor, Mohd Shahrizal Sunar, and Afnizanfaizal Abdullah. 2017. A new weighted pathfinding algorithm to reduce the search time on grid maps. *Expert Systems with Applications* 71 (2017), 319–331. https://doi.org/10.1016/j.eswa.2016.12.003
[2] Anon. 2009. Using Potential Fields in a Real-time Strategy Game Scenario (Tutorial). (Jan 2009). http://aigamedev.com/open/tutorials/potential-fields/
[3] Michael Buro and Timothy Furtak. 2004. RTS Games and Real-Time AI Research. *In Proceedings of the Behavior Representation in Modeling and Simulation Conference (BRIMS)* (2004), 51–58.
[4] David Churchill and Michael Buro. 2011. Build Order Optimization in StarCraft. *Proceedings of the Seventh AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment* (2011), 14–19.
[5] Kenneth Forbus, James Mahoney, and Kevin Dill. 2002. How Qualitative Spatial Reasoning Can Improve Strategy Game AIs. *IEEE Intelligent Systems* 17, 4 (2002), 25–30.
[6] Google. [n. d.]. Protocol Buffers | Google Developers. ([n. d.]). https://developers.google.com/protocol-buffers/
[7] Johan Hagelback. 2016. Hybrid Pathfinding in StarCraft. *IEEE Transactions on Computational Intelligence and AI in Games* 8, 4 (2016), 319–324. https://doi.org/10.1109/tciaig.2015.2414447
[8] Darius Kazemi. [n. d.]. Spelunky Generator Lessons. ([n. d.]). http://tinysubversions.com/spelunkyGen/
[9] Jerome Laforge, Silvia Garcia Diez, and Marco Saerens. 2013. Rminimax: An Optimally Randomized MINIMAX Algorithm. *IEEE Transactions on Cybernetics* 43, 1 (2013), 385–393.
[10] Josh MCCoy and Michael Mateas. 2008. An Integrated Agent for Playing Real-Time Strategy Games. *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence* (2008), 1313–1318.
[11] Santiago Ontanon. 2012. Experiments with Game Tree Search in Real-Time Strategy Games. *CoRR* abs/1208.1940 (2012). http://arxiv.org/abs/1208.1940
[12] Santiago Ontanon, Gabriel Synnaeve, Alberto Uriarte, Florian Richoux, David Churchill, and Mike Preuss. 2013. A Survey of Real-Time Strategy Game AI Research and Competition in StarCraft. *IEEE Transactions on Computational Intelligence and AI in Games* 5, 4 (2013), 293–311.
[13] Sergio Queiroz, Geber Ramalho, and Thiago Souza. 2014. Resource Managment in Complex Environments - An Application to Real Time Strategy Games. *Brazilian Symposium on Computer Games and Digital Entertainment* (2014), 21–30.
[14] Craig Reynolds. 1987. Flocks, Herds, and Schools: A Distributed Behavioral Model. *SIGGRAPH Conference Proceedings* (1987), 25–34.
[15] Frederik Sailer, Michael Buro, and Marc Lanctot. 2007. Adversarial Planning Through Strategy Games. *Proceedings of the 2007 IEEE Symposium on Computational Intelligence and Games* (2007), 80–87.
[16] Daniel Scales and Tommy Thompson. 2014. SpelunkBots API - An AI toolset for Spelunky. *2014 IEEE Conference on Computational Intelligence and Games* (2014). https://doi.org/10.1109/cig.2014.6932872
[17] Frederik Schadd, Er Bakkes, and Spronck Pieter. 2007. Opponent modeling in real-time strategy games. *Proceedings of the GAME-ON 2007* (2007), 61–68.
[18] Tommy Thompson and Dave Voorhis. 2014. Planning in the Cloud: Massively Parallel Planning. *2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing (UCC)* (2014), 493–494.
[19] W3Schools. [n. d.]. JSON vs XML. ([n. d.]). https://www.w3schools.com/js/js_json_xml.asp