Dillon Newell
1000407641

GameSolver.NET – Analyzing game theory solving in portable program applications

Abstract

Solving game theory problems can be useful in a variety of programs in use across many hardware types. GameSolver.NET attempts to create a portable library that can be consumed by such programs and perform in a quick and deterministic fashion independent of the hardware it is deployed on. We focus on video games as a use case and find that under certain scenarios it can indeed be a viable solution.

1. Introduction

In today's world, programs are in use on many different types of hardware such as computers, smartphones, or video game consoles. Individual programs may be developed for multiple hardware types simultaneously. In this scenario, it is desirable to reduce code redundancy by developing the program in a way that it is hardware agnostic. This speeds up development and debugging time by only requiring modifications to one code base. Microsoft recommends using code sharing for cross-platform mobile app development [1].

Application of game theory can be useful to many types of programs [2]. However, there is little research into a hardware agnostic approach that allows a program to be easily developed for multiple platforms. Thus, we introduce GameSolver.NET, a portable library that can solve a variety of game theory problems on many different hardware platforms.

To analyze GameSolver.NET's viability, we focus on a popular case that requires deployment to many platforms: video games. Video games can be deployed to computers, smartphones, consoles, or web browsers. They also frequently require a server to facilitate multiplayer functionality.

The following sections are broken down as follows:

2. Overview of the GameSolver.NET implementation.
3. In-depth explanation of hypothetical game theory problems that could be useful to video game developers.
4. Explanation on the GameSolver.NET implementation for the problems discussed in section 3.
5. Methodology for experiments on how GameSolver.NET performs with the problems in section 3, based on metrics such as processor time and memory usage.
6. Results from experiments.
7. Discussion of results.
8. Conclusion
9. Future work

2. GameSolver.NET overview

GameSolver.NET is written in C# and targets the .NET Standard API [3]. .NET standard is a platform agnostic API from Microsoft that can be consumed by any .NET runtime and runs on the following platforms:

- Windows based computers and servers, through .NET Framework [4] or .NET Core [5].
- Linux or macOS based computers and servers, through .NET Core.
- Smartphones through Xamarin [6].
- All the above plus in-browser and on video game consoles through Mono [7].

In addition to its wide platform compatibility, .NET Standard was chosen because it is particularly relevant for video games. A great deal of video games are programmed with the Unity engine [8], which runs Mono for game logic [9] and can easily reference a .NET Standard library [3]. ASP.NET is also a very popular platform for running servers [10], and it too can target .NET Standard libraries.

GameSolver.NET makes extensive use of functional programming through the LINQ API, which allows for two important optimizations relevant to video game applications. Firstly, it can greatly reduce the memory footprint [11]. In both client-side and server-side logic for video games, memory usage is an important limitation that must be kept in check. Secondly, it allows for easier multi-threaded processing [12]. Certain platforms such as smartphones and consoles typically have a large

amount of slow processor cores, as opposed to a small number of fast cores. Servers can typically have dozens of processor cores. Multi-threading allows for more of the cores to be used, instead of waiting on a single core to finish work while the others are idling.

3. Cases of game theory applications in video games

This section is broken down into two subsections. Firstly, we discuss a hypothetical example where game theory is used for server-side logic. Then, we discuss an example that could be used client-side inside the game executable itself.

3.1. Server-side case – map and gametype selection

A competitive multiplayer game may have several map and gametype combinations that a match uses. Maps are virtual areas that the match takes place in, and gametypes are the objectives each team must complete to win. An example of a gametype might be capture the flag or simply eliminate the enemies.

These competitive games typically employ one of two approaches to select a map/gametype combination for a match. Firstly, they can decide one combination and unilaterally enforce it on the players. Or, they could present a small number of combinations and allow the players to vote on them.

Say the multiplayer server has access to each player's relative performance for a given map/gametype combination. This could be obtained by observing the player's results while playing it. In some scenarios, this data can be used to choose combinations that are the most competitive for players, i.e. where their skills will be most evenly matched.

Suppose a simple competitive head-to-head match with two players. There are two gametypes and two maps available. Two combinations will be selected by the server to play back-to-back. For variation, the combinations must be unique. The server will try to select a pair that will result in closer outcomes for the players to maximize competition.

Say player 1 is better at gametype 1 on map 2, and gametype 2 on map 1. He also prefers map 1 overall. This could be illustrated with a cost vector:

$$\boldsymbol{a} = [6, 2, 1, 8]$$

Which has the form of map/gametype [1/1, 2/1, 1/2, 2/2]. Similarly, a cost vector for player 2 could be:

$$\boldsymbol{b} = [2, 4, 2, 7]$$

The server will try to select a pair of combinations where both players can be happy with their overall performances. It can do this by constructing a cost matrix for player 1, where each element is his own cost for the pair subtracted by the cost for player 2. Thus, a lower cost in the matrix will result in a better pair matchup for player 1. The diagonals are represented as infinity, since the matchups must be unique:

$$A = \begin{bmatrix} \infty & 2 & 3 & 5 \\ 2 & \infty & -3 & -1 \\ 3 & -3 & \infty & 0 \\ 5 & -1 & 0 & \infty \end{bmatrix}$$

The cost matrix for player 2 is simply the negative of *A*, but with the diagonals still positive infinity. This can be solved as a *nxn* bi-matrix game. The server will try to find pure Nash Equilibrium (NE) solutions to the bi-matrix problem. If there are multiple, it can randomly select from them. If there are none, it can randomly select a pair that would be close to an NE. If a voting system is used, the options to use for the vote can be selected from the pool of NE solutions.

This can be kept in a "2-player" fashion while extending to team-based games. The goal is to keep the teams competitive, so the cost vectors used to construct the matrix could simply be the sum of all the individual players' cost vectors on each team.

In this simple case there are only four combinations available, but most games will have more maps and gametypes available. For example, the shooter game *Halo 5* regularly rotates roughly 42 maps and 20 gametype variations [13] for a total of 840 combinations (however, not every gametype is available for every map).

3.2. Client-side case – computer-controlled enemies

Controlling a virtual enemy (commonly called AI) to play against the player is a common application for game theory. Many games are a natural fit for game theory, such as tic-tac-toe [14]. It is important

however to maximize enjoyment for the player. If the AI is too good, the player may become frustrated. If the AI is too predictable, the player may become bored.

Here we focus on a more complicated example that may occur in a three-dimensional game where the player has more options than placing an "X" or "O". The goal of the AI will be to eliminate the player, but in such a way that its actions are unpredictable.

Suppose the game is of the action genre, and both the player and AI are taking cover behind corners down a hall from one another. We give the AI two options: run up and attempt to ambush the player or remain by the corner. The AI's costs for each action is highly dependent on the player's actions, and we let the player choose from a wide variety of actions, such as:

- Remain by the corner. In this case we give a low cost to the AI if running up, and a neutral cost if it too stays back.
- Running up. This is a high cost for the AI if staying back, and a neutral cost if running up.
- Throwing a grenade to the AI's corner. Low cost for the AI if running up, high cost if staying back.
- Staying by the corner but shooting down the hall. High cost for the AI if running up, neutral if staying back.

The exact costs would be determined beforehand. Using simple values, this could be expressed in a zero-sum game with cost matrix:

$$A = \begin{bmatrix} 0 & -1 & -1 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$

Where the first row is the AI running up, and second staying back. Values are -1 if the AI eliminates the player, neutral if neither are eliminated or there is an equal chance, and +1 if the player eliminates the AI. Since we want the AI to be unpredictable, we can first find its mixed NE solution. Then, randomly select an action weighted by its mixed proportion.

4. Implementation details of game theory solutions

GameSolver.NET is able to solve a variety of game theory problems. The algorithms used to solve are detailed in this section and noted on their relevance to problems from the previous section.

Cost matrices can be input in one of two ways. Firstly, they can be passed directly from a referencing program. This would require that the referencing program be written for a compatible runtime, such as a .NET runtime or a native COM interop application (e.g. C++). Secondly, it can parse a comma-separated values string (CSV).

The solvers are implemented based on costs of the C# type "double". This is a 64-bit floating-point (FP64) data type based on the IEEE 754 specification [15]. In general, modern CPUs are well optimized for FP64 arithmetic and have only negligibly more performance for less precise types (sometimes, less precise types are slower) [16].

Every solver is thoroughly tested against all examples given in the ECE 1657 notes and problem set solutions.

4.1. Pure solutions for arbitrary-sized bi-matrix games

GameSolver.NET can find all pure solutions for bi-matrix games with matrices of any size. Of course, the processing time and memory usage will increase for larger matrices. Memory usage is optimized wherever possible. A 10000x10000 matrix of doubles already consumes at least 800 megabytes of memory space (memory usage calculations are given in Appendix 1.1).

The algorithm for this is relatively simple. First, two lists are constructed: one for the minimums across columns, and another for the minimums across rows. Then, it iterates over every cost in the matrix. If the cost is less than or equal to the minimums in the rows or columns for each player's respective matrix, it is selected for returning as a NE solution.

The memory footprint is equal to the footprint for the matrices plus $8(n+m)$, where n and m are the dimensions of the matrices. It is implemented in a way that allows for the matrices to be stored externally, e.g. in a database such as MySQL. In this case there is no memory footprint for the matrices.

The time complexity is:

$$O(nm)$$

Multi-threading is utilized where possible to make use of all CPU cores.

4.2. Mixed solutions for 2x*n* zero-sum matrix games

Mixed solutions can be found for 2x*n* zero-sum matrix games for arbitrarily large *n*. The algorithm employs the graphical method to find the solutions. Again, memory is optimized wherever possible.

The algorithm starts by iterating over each column and selecting a best response function from it. It then tries to find an intersect with every other best response function. If one is found, it checks to see if any of the other best response functions are above this point. If there is, then this is not a valid solution since it is below the top line drawn by the functions. Then, it finds the intersection point that has the lowest value and returns that as the NE solution.

The memory footprint is negligibly higher than that of the matrix. If the matrix is stored externally, the memory footprint is negligible.

The worst-case time complexity is:

$$O\left(n(0.5n^2 + n)\right) \approx O(n^3)$$

Although it is frequently less than this, as discussed in Section 7. Multi-threading is used for the entire algorithm.

5. Methodology

The implementations discussed above were empirically tested across three different platforms:

1. Desktop computer – Represents a typical gaming desktop
2. Server – Similar hardware to the desktop, but with slower processor cores and slightly better multithreading efficiency
3. Mobile phone – A typical smartphone that one could play games on

Exact specifications of the platforms are given in Appendix 2, along with specific software and framework versions. The following symbols will be used to express test parameters:

- *K* – The amount of different numbers that will be implemented for cost values. E.g. with *K* = 10, cost values will be selected from [0, 1, 2, …, 9]. The values benchmarked are 10, 100, and 2147483647 (the maximum value of a signed 32-bit integer).
- *N* – The size of the matrix. For pure solution tests, this will be an *N*x*N* matrix, for mixed it will be a 2x*N* matrix. The values benchmarked are 10, 100, 1000, and 10000.

For each platform, the following steps were taken to test performance:

1. A random number generator was instantiated with a specific seed. This is important to make sure the numbers it generates are the same across all platforms.
2. 2 arrays of length *N* are instantiated with random values between 0 and *K*. These will represent the players' costs for a map/gametype combination.
3. Two *N*x*N* matrices are created from the two cost vectors as described in Section 3.1.
4. 5 warmup tests are run without measuring anything. These get the system up to full speed before the actual benchmark.
5. The time, CPU time, and memory consumption are measured over 10 benchmarks as described in Appendix 1. The average value is returned and used for experiment results. Several other metrics such as average number of NE solutions found are returned as well. These serve to verify consistency across platforms.
6. Steps 2-5 are repeated for variations of *N* and *K*. The test program is restarted in between tests to ensure that the memory footprint of previous matrices is not still held and skewing total memory consumption results.
7. The test program is restarted again.
8. Mixed solutions are tested – a 2x*N* matrix is instantiated with random values between 0 and *K*.
9. Steps 4 and 5 are repeated using the mixed solution solver.
10. Steps 7 and 8 are repeated for variations of *N* and *K*. Again, the test program is restarted between tests.

Figure 1: Results from pure strategy games. (a) Total calculation times vs dimensions of matrix. (b) Calculation time scaling with limit of unique cost values. (c) Memory consumption vs dimensions of matrix. The two lines in the centre are actually three lines, the server and desktop results match very closely. These three lines use the left axis. The other three lines are memory consumption without the consumption to hold the matrix object; these use the right axis. (d) Thread multiplier vs matrix dimensions.

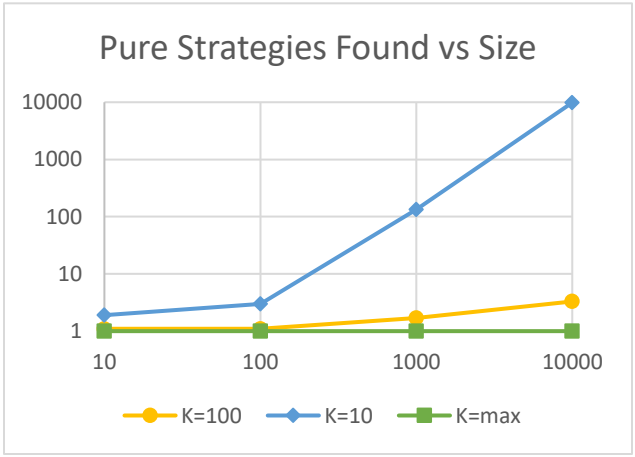

Figure 2: Number of pure strategies found vs matrix dimensions for different values of K.

Varying the *N* parameter is important to see how performance scales with more complex games. Varying *K* illustrates how performance can increase when solution intersections are sparse.

6. Results

6.1. Pure strategy results

This section contains results from benchmarks modelling the map/gametype selection problem described in Section 3.1. These are shown in Figures 1 and 2.

The solver finds solutions to the example given in Section 3.1 of (2, 1).

6.2. Mixed strategy results

This section contains the results from benchmarks modelling the AI control problem described in Section 3.2. These are shown in Figure 3.

7. Discussion

There are two important analyses to make from the results:

- How robust the code was to moving across the platforms.
- How the different platforms performed.

Since an important part to this project is creating a portable codebase that can be used across platforms, it is important to ensure that GameSolver.NET behaves the same no matter where it is deployed.

Select solutions were compared and indeed the library gave the same results on all platforms. The results in Figure 2 are also the same for each platform. Thus, we can conclude that the library behaves the same across the platforms. With thorough tests on the desktop platform for known games and solutions passing, we can assume that the results are accurate everywhere.

When analyzing performance, there is an important note to make regarding viability. Some problems do not have to run in real-time, such as the map/gametype selection shown in the pure strategy results. However, others are processed client-side and must run in real-time. The AI problem in the mixed strategy results is an example.

Typically, games will run at 30 or 60 frames per second (FPS). Logic is often run at similar times, and it would be unreasonable to run AI calculations that take longer than 33 milliseconds (the time for each frame at 30FPS). No platform is able to compute a mixed solution for $N = 10000$ in less than a second. For a sparse game (where K is very high), the best result is achieved on the desktop and is 1.53s. On the smartphone, it takes 25.1s. As can be seen in Figure 3(b), mixed solution time goes up as K gets smaller. This is because the algorithm does not run its inner loop if an intersection is not found between two best response functions. With high K, these intersections will be rare. So, no platform can viably compute mixed solutions for $N = 10000$.

Looking at $N = 1000$, the desktop can compute the solution in 24.3ms for sparse games. Therefore, it can be reasonable to use this problem to model AI behaviour in a desktop game. The smartphone requires 260ms, and so a drop to $N = 100$ (which it can compute in 6ms) is required for viability. At $N = 1000$ the desktop can compute the solution in under 225ms for all values of K, and at $N = 100$ the smartphone can find the solution in under 12ms for any K (the desktop can find any solution in under 3ms for $N = 100$).

We can conclude that the game described in Section 3.2 can be applied to video games on any platform if N is on the order of 10^2.

The memory footprint to run the algorithm is around 30MB for Windows-based programs and 100MB for smartphone. However, much of this memory is required by the test program itself. It can be assumed the extra overhead from solving a game from Section 3.2 is very small.

Multi-threading is efficient when finding mixed solutions. It is highest when $N = 10000$ for every platform. It is around 8 on the desktop and server platforms (which have 12 CPU threads and 6 true CPU cores each), and 5 on the smartphone. While the smartphone has 8 CPU cores, half of them are lower performing and used mainly in low-power calculations. These results are what would be

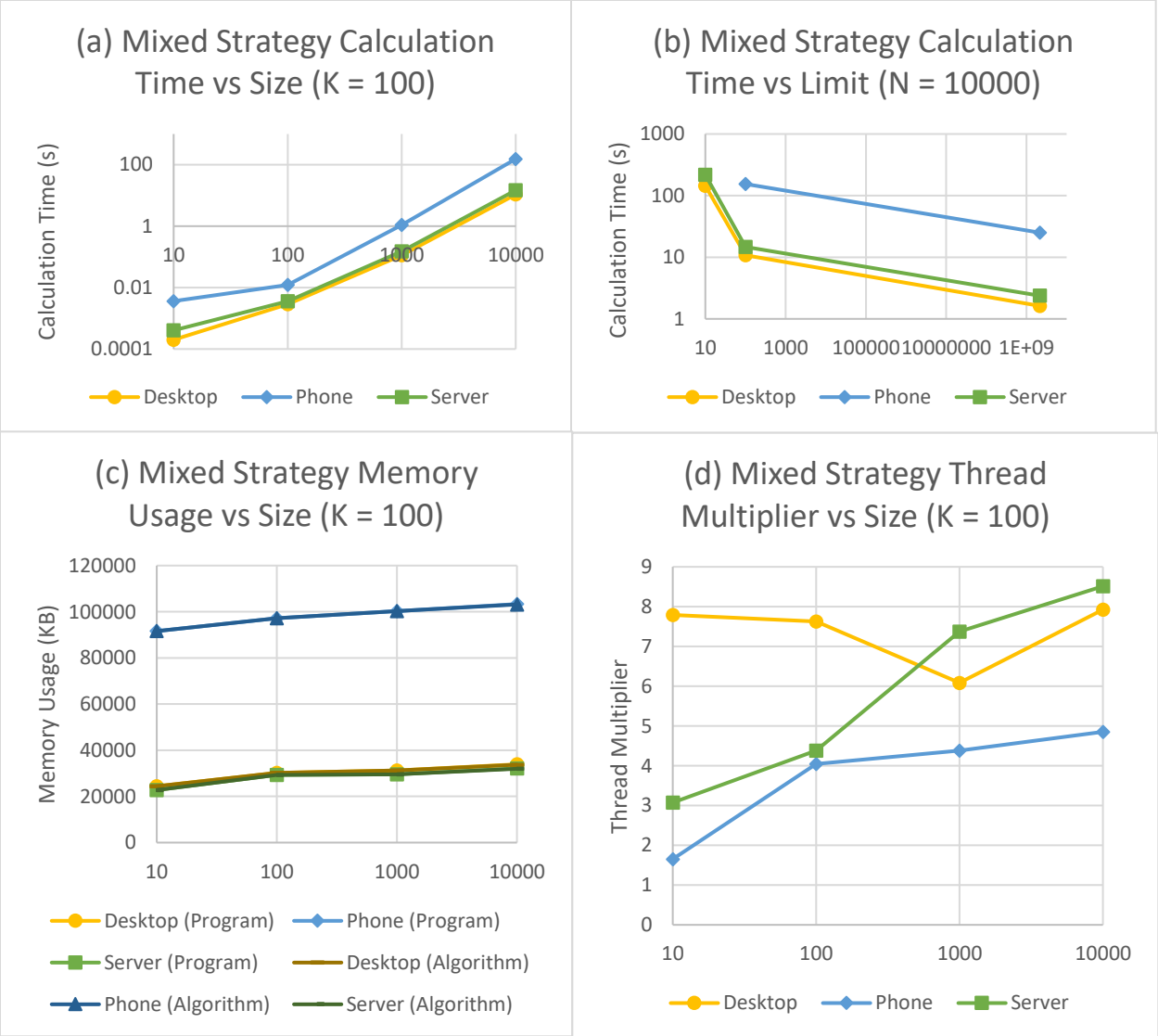

Figure 1: Results from mixed strategy games. (a) Total calculation times vs width of matrix. (b) Calculation time scaling with limit of unique cost values. (c) Memory consumption vs width of matrix. The lines two lines for phone very closely match, as do the 4 lines for desktop and server. (d) Thread multiplier vs matrix widths.

expected from excellent multi-threading scaling. Note however, that the results for $N = 100$ and less are not as accurate, since the internal CPU time clock does not have as high of a resolution as real-time measurement does.

The pure solution results are faster across the board. This is because it is an $O(N^2)$ operation instead of $O(N^3)$. There is however significantly more memory required to hold the cost matrix. This is not a problem if the matrix is stored externally as can be seen in Figure 1(c), but as a result the smartphone was unable to complete tests for $N = 10000$ since it exceeded Android's memory limits.

The game theory problem discussed in Section 3.1 would likely run on a server however. Also, this scenario would almost certainly have cost values stored in an external database. GameSolver.NET can be used in a way so that the map/gametype matrix is constructed on the fly from the players' individual cost vectors.

As can be seen from Figure 1(b), pure solution time does not appreciably depend on K. This is because sparsity of solutions does not affect its time complexity. The server can solve this game for $N = 10000$ in around 2.4s, and for $N = 1000$ (close to the example given in Section 3.1) in around 35ms. This

overwhelmingly supports the application of this problem to solve match/gametype voting, since a time of 35ms would put little stress on the multiplayer server.

One problem that could arise in the computation time is the access to an external database for the players' cost vectors. However, the user of the library could easily construct a system that caches the cost vectors (which would only take up $8 \times N$ bytes each) and constructs the total cost matrix on the fly. This eliminates the memory overhead of holding an NxN matrix, while not significantly increasing runtime since the database is only accessed once.

The multithreading scaling for pure solutions is not as high as it is for mixed. This is because part of the algorithm is not multi-threaded. It is however quite efficient, providing a scaling of around 4.3 for $N = 1000$.

As an extra result, the number of NE solutions found for the selected random number generator seed are shown in Figure 2. It is intuitive that the number of NEs will scale with the dimensions of the matrix. For $K = 10$ the scaling is well-described with N. However, as K gets higher, there will be less cost values that equal each other. This results in a drop of NEs, and when K is at its highest only one NE was found for every value of N.

8. Conclusion

We made the following observations from the results:

- GameSolver.NET can be viable to compute AI actions in a way that is both unpredictable to the player and maximizes difficulty. If the number of actions considered by the player is on the order of 10^3, it can be calculated in less than the time it takes the GPU to render a frame on the desktop. If it is on the order of 10^2, it can be calculated in a fraction of that time on both the desktop and smartphone.
- GameSolver.NET can be viable to compute maps to select for voting by a multiplayer server. The time required is negligible for the server if the number of map and gametype combinations is on the order of 10^3. If no NEs are found, the library can return the top candidates that are close to being NEs for the server to select from or present for voting.
- The library gives deterministic results regardless of what platform it is deployed on.

So, we have shown that two hypothetical (but realistic) applications of game theory to video games can viably be used by a cross-platform, hardware agnostic library such as GameSolver.NET. But what exactly are the benefits from the final product?

Every line of code in the core GameSolver.NET library is completely hardware agnostic. The only hardware-specific code used was that used by the host program (e.g., the boilerplate Android application logic to support smartphone benchmarks). From the referencing application, only one line of code is needed to get the solutions if a cost matrix is already constructed. This means that adding any features or fixing any bugs in the game theory logic can be done once and automatically applies to every platform.

The library can be used directly with matrices constructed from arrays. However, it also can be used with any array-like object. In .NET this is known as "IReadOnlyList". IReadOnlyList is an interface implemented by several array-like objects (including arrays themselves). However, a developer can construct a custom object that implements IReadOnlyList and use it with GameSolver.NET. This allows e.g. an object that is backed by a database instead of in-memory arrays. A custom IReadOnlyList may not be backed by an array at all, and instead calculate values on the fly. The only requirement is that it has a defined amount of elements and an element can be selected by its index.

Being hardware-agnostic and allowing such flexibility by developers means GameSolver.NET can be dropped into almost any .NET project that could make use of its game theory solvers.

9. Future work

GameSolver.NET can only solve a limited number of games as it stands. It can find the pure NEs for any bi-matrix game, and the mixed NE for any 2x*n* zero-sum game. It is implemented in a way that adding

other solver types should be easy – expanding to find e.g. solutions to population games could prove useful.

Further platforms can be tested for performance, such as video game consoles. These unfortunately require a developer subscription to test programs on.

The benchmarks provided here are all synthetic. To get the most accurate results, GameSolver.NET could be tested in a live application. E.g. a current video game developed with Unity can easily reference the library. Importantly, checking if the solution overhead impacts user experience (mainly reduced framerates) would be the final viability test.

It may be possible to solve 3x*n* games in mixed strategy by utilizing 3-dimensional lines instead of 2-dimensional.

Implementing learning after repeated games is a very interesting concept for controlling AI and should be explored further.

Appendix

1. Technical calculations

1.1. Memory footprint

Memory footprint calculations are relatively straightforward. If we let each cost value be associated with a 64-bit number (such as a double precision floating-point type of long integer), then each value will take up 8 bytes. In the GameSolver.NET implementation, all values are stored as 8-byte doubles. This is done so any consumer can use their value types with ease, since a smaller number type can be upgraded to a double. Ideally, we could allow the user to select a lower storage size. This could be accomplished with relative ease by duplicating the logic and using a smaller type such as a 4-byte floating-point value but was not explored.

With each cost value taking up 8 bytes, an *m*x*n* cost matrix takes up $8mn$ bytes.

During pure solution finding, the minimum values across rows and columns are cached for performance. These take up $8m$ and $8n$ bytes for a total of $8(m + n)$. The other overhead is negligible and consists of loop variables and return values.

During mixed solution finding, there is no cache. There is negligible overhead in the point and line structs used during the graphical analysis. However, due to functional programming with LINQ, no more than one of these are held at a time.

During experiments, memory footprint is obtained using diagnostic tools built into the .NET Standard API. Algorithm memory footprint is obtained by subtracting the memory required for the cost matrix from the total memory footprint.

1.2. Processor time

Processor time is measured in two values. First is in real-time, which is the time it takes the algorithm to complete from start to finish.

The second is in CPU time, which is the total aggregate time spent among all CPU threads. If 100% multi-threading efficiency is achieved. The thread multiplier is how much faster the real-time is compared to CPU time, i.e. real-time divided by CPU time. This value would ideally be equal to the number of hardware CPU threads. However, efficiency this high is rarely achieved.

During experiments, these values were obtained using analysis tools built into the .NET Standard API.

2. Setup configurations

2.1. Desktop

- Windows 10 x64 Build 17134.441
- Intel Core i7-8700k @4.8 GHz (6 cores, 12 threads)
- Gigabyte Aorus Gaming 7 Z370 Motherboard
- 32 GB DDR4-2400 RAM

2.2. Server

- Windows Server 2019 Build 17666.1000
- Intel Core i7-5820k @3.7GHz (6 cores, 12 threads)
- ASUS X99 Sabertooth Motherboard
- 32GB DDR4-2133 RAM

2.3. Phone

- Android 8.1 (Oreo)
- Samsung Galaxy S8+ G955W
- Qualcomm Snapdragon 835 (4 high power + 4 low power cores)
- 4 GB RAM

2.4. Software

- Visual Studio Enterprise 2017 15.8.8
- Codebase targets .NET Standard 2.0
- Windows tests target the .NET Core 2.1 framework
- Mobile tests target Xamarin Forms 3.1 and Android version 8.1 (API 27)

References

[1] Microsoft. 2018. Sharing code overview. https://docs.microsoft.com/en-us/xamarin/cross-platform/app-fundamentals/code-sharing.

[2] Mohammadi, A, Rabinia, S. 2018. A comprehensive study of Game Theory applications for smart grids, demand side management programs, and transportation networks. arXiv:1804.10712.

[3] Microsoft. 2018. .NET Standard. https://docs.microsoft.com/en-us/dotnet/standard/net-standard.

[4] Microsoft. 2018. .NET Framework Guide. https://docs.microsoft.com/en-us/dotnet/framework/.

[5] Microsoft. 2018. About .NET Core. https://docs.microsoft.com/en-us/dotnet/core/about.

[6] Microsoft. 2017. Get started with Xamarin. https://docs.microsoft.com/en-us/xamarin/cross-platform/get-started/.

[7] Mono Project. 2018. Mono. https://www.mono-project.com/.

[8] Unity Technologies. 2018. Games made with Unity. https://unity.com/madewith.

[9] Microsoft. 2014. Unity: Developing your first game with Unity and C#. https://msdn.microsoft.com/en-us/magazine/dn759441.aspx.

[10] W3Techs. 2018. Usage statistics and market share of ASP.NET for websites. https://w3techs.com/technologies/details/pl-aspnet/all/all.

[11] Microsoft. 2007. LINQ: .NET Language-Integrated Query. https://msdn.microsoft.com/en-us/library/bb308959.aspx.

[12] Microsoft. 2017. Introduct to PLINQ. https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/introduction-to-plinq.

[13] Halo Nation. 2015. Halo 5: Guardians multiplayer maps. http://halo.wikia.com/wiki/Category:Halo_5:_Guardians_Multiplayer_Maps.

[14] Cornell Blog. 2015. Game theory in video game AI and its limitations. https://blogs.cornell.edu/info2040/2015/09/13/game-theory-in-video-game-ai-and-its-limitations/.

[15] Microsoft. 2018. Double struct. https://docs.microsoft.com/en-us/dotnet/api/system.double.

[16] Nikolskiy, V, Stegailov, V. 2016. Floating-point performance of ARM cores and their efficiency in classical molecular dynamics. *J. Phys.: Conf. Ser.* 681 012049