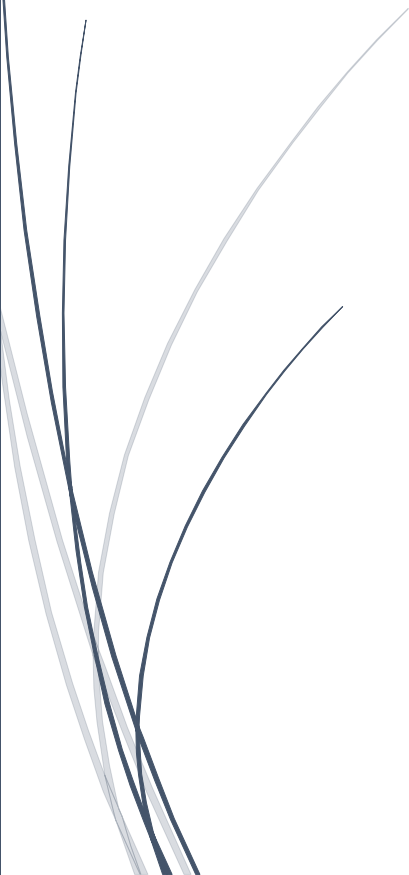




Stand: 3301

Where Brute Force meets AI

BTYS Project Report



Andrew Finnerty
THE 'BISH' GALWAY

Table of Contents

Abstract.....	3
Introduction	4
Literature Review.....	6
Methodology	10
Methodology Phase I	10
Methodology Phase 2	10
Methodology Phase 3	10
Brute Force Algorithm Design.....	10
Source an AI	11
Automated Game Driver.....	11
Generalization form 3x3 grid to NxN grid.....	12
Experiments and Testing	12
Results.....	14
Fixed and Execution Costs	14
Accuracy.....	16
Optimal Point of Transition	18
Transition by Game Iterations	18
Transitions by Accuracy	18
Overall Transitions	21
Discussion	22
Conclusion.....	25
Appendix A - Environment for tests	28
Appendix B - Tic-tac-toe game theory	30
Appendix C - Methodology	37
Detailed description of precautions and pre-experimental process to set up test	37
Detailed description of the experiments.....	38
Appendix D - Results	40
Table of results (3x3 grid)	40
Analytics charts for 3x3 grid data	41
Table of results (4x4 grid)	42
Appendix E – Python Code.....	45

Abstract

This project, titled “Where AI meets Brute Force”, is an innovative exploration into the intersection of artificial intelligence (AI) and Brute Force algorithms. The primary objective was to pitch the two - AI and Brute Force – harnessed as game engines against each other in a consistent game arena and find the lowest environmental cost / best overall calculation efficiency.

This is realized by finding the optimal point of transition (*POT*) between the two game engines. Both AI and Brute Force methods are notorious for their computing power requirements. I am demonstrating how pitching these against each other will help to find the optimal point of transition for their computing costs.

The optimal point of transition is the specific point at which the game engines are transitioned maximizing computational efficiency. This point will be specific to the complexity of the task, the level of accuracy required, the number of iterations or how many times the algorithms need to be run. This means that the optimal point of transition will vary drastically from task to task even by changing certain parameters slightly as seen in the results I have collected. Although the results are very specific and cannot be taken as a general case of the problem the methodology, process and concept will remain similar, to find the optimal point of transition generically.

For the experiments I used three types of algorithms: AI Q-learning algorithm, Brute Force algorithm in the form of a tablebase (look-up-table), and a combination of both algorithms at different points of transition.

The overall cost has been compared to the accuracy of the algorithm.

The results clearly show that an optimal point of transition can be calculated by combining algorithms for a specific task. The two algorithms are very different yet transitioning between them at a specific point of transition to meet the requirements of the task optimizes the cost.

The results themselves are not particularly useful. However, the main contribution of the project is proving that the key concept of optimizing computational efficiency through combining and transitioning between algorithms is valid. Naturally the next step would be to combine more algorithms to increase efficiency again or even calculate the optimal number of algorithms to assign to a project and where to transition between them.

This project increases awareness of the damage of inefficient algorithms, the way to go about increasing efficiency, the need to find alternative algorithms where it is more efficient and to use the right tools for the job. It sets out a path forward for further research in this area, and highlights the potential applications of the findings.

Introduction

This project aims to address a significant problem in the field of AI - the high computational cost associated with complex games such as chess. While Brute Force computing is expensive, it is a one-off cost, whereas the cost of AI is repeated with each execution. By combining elements of both AI and Brute Force computing, this project seeks to increase computational efficiency and reduce environmental impact. In this project, I demonstrate how to find the right combination of algorithms for the task and how to find the optimal point of transition between them.

This problem is part of a bigger issue which is that the inefficient use of algorithms wastes computational resources. My project focuses on one aspect of this problem in game theory, which is a useful experimental setting due to the self-imposed limitations of games.

The inspiration for this project comes from the game of chess, where a 'tablebase' concept can show the result of any given position with a certain number of pieces. AI algorithms in chess struggle in the endgame, losing accuracy. These can be improved by applying tablebases, but there are limitations on generating tablebases due to the exponential increase in computing cost as complexity increases. This means that up until the point of reaching a tablebase position, AI and other heuristic algorithms must be used instead. I was curious whether advanced chess engines such as Stockfish factor the pre-calculated tablebase positions into their algorithms. This curiosity inspired this project and my idea of combining algorithms to optimize computational efficiency and accuracy.

In computer science class we learned that the four key techniques to computational thinking are: decomposition, abstraction, pattern recognition and algorithms. To solve any task computationally these must be applied.

It seemed logical to me that, when faced with a task like building a chess engine, you would break a problem down into smaller sections (decomposition), and assign each section to a niche algorithm, specific to the specific section of the game. This way you could optimize each algorithm to do its specific job with the highest degree of success. To find the optimal point, first you must find the best algorithms. Intuitively some algorithms will be more appealing than others, and understanding the maths behind the problem can be very useful for selecting the right algorithm as shown in the discussion and the appendix.

In my study it made sense for me to implement the Brute Force algorithm at the end as there is a more finite number of positions to calculate, and to implement the AI at the start due to its pattern recognition abilities so it could quickly eliminate possibilities while not being required to play with 100 per cent accuracy. Understanding the maths of the game is key to selecting the right algorithm here.

While my project talks a lot about increasing computational efficiency of algorithms. However, I do not at any point attempt to increase the efficiency of any one algorithm, by changing its

logic, scalability or code. Instead, my project simply wishes to find the most efficient use of given algorithms possibly by transitioning between them to optimize cost and accuracy to meet the user's needs. The AI and Brute Force algorithms are not the most efficient algorithms for the task. They simply represent any algorithms capable of solving the task and how this can be measured.

My project involves a series of experiments using an N-Dimensional Tic-tac-toe grid as the experimental grounds. Different grid sizes were tested, along with different points of transition from AI to Brute Force. I was satisfied with the results of the project as they have shown that an optimal point of transition can be calculated between two algorithms given specific parameters.

My project involved a specific case of the problem of inefficient algorithmic use applied to game theory and the results have proven the concept. There are so many applications of this project in areas which require large quantities of data and selecting the best algorithm for the task. Within game theory alone, I would like to run similar tests in chess and other more complex games after proving in a simpler domain that this concept works. However, this idea is not be limited to games, in any sectors with complex processes such as climate and financial modeling there are applications, and I will discuss this later.

Literature Review

Here are a few key concepts related to my project:

Brute Force algorithm:

An algorithm which systematically checks and iterates through all possible combinations usually through some sorting algorithm often by divide and conquer or backtrack and recursion.

Q-Learning AI:

Q-learning is a model-free reinforcement learning algorithm to learn the value of an action in a particular state. It does not require a model of the environment. Reinforcement Learning has been hugely successful in many areas especially in game theory with some examples like AlphaGo.

‘reinforcement learning systems are trained from their own experience, in principle allowing them to exceed human capabilities, and to operate in domains where human expertise is lacking’ (Silver et al., 2017, p. 354)

This type of reinforcement learning involves an agent, a set of *states* and a set of *actions* per state. By performing an action, the agent transitions from state to state. Executing an action in a specific state provides the agent with a *reward* (a numerical score).

‘The essence of RL is learning through interaction. An RL agent interacts with its environment and, upon observing the consequences of its actions, can learn to alter its own behaviour in response to rewards received. This paradigm of trial-and-error learning has its roots in behaviourist psychology, and is one of the main foundations of RL [135]. The other key influence on RL is optimal control, which has lent the mathematical formalisms (most notably dynamic programming [13]) that underpin the field. ((Arulkumaran et al., 2017, p.2)

The goal of the agent is to maximize its total reward. It does this by adding the maximum reward attainable from future states to the reward for achieving its current state, effectively influencing the current action by the potential future reward. This potential reward is a weighted sum of expected values of the rewards of all future steps starting from the current state.

‘Deep reinforcement learning is poised to revolutionise the field of AI and represents a step towards building autonomous systems with a higher level understanding of the visual world.’ (Arulkumaran et al., 2017, p.1)

(Optimal) Point of Transition:

The specific point at which the algorithm transitions from one program to the next. The optimal point refers to the point which maximizes the accuracy of the task for a certain cost given the parameters of the task.

This project best fits under the category of increasing computational efficiency. Surprisingly I found very little research and experimentation specific to game theory involving AI and Brute Force.

There were quite a few papers on the famous AlphaGo and Alpha Zero dominations in Go and chess respectively.

‘Amongst recent work in the field of DRL, there have been two outstanding success stories. The first, kickstarting the revolution in DRL, was the development of an algorithm that could learn to play a range of Atari 2600 video games at a superhuman level, directly from image pixels [84]. Providing solutions for the instability of function approximation techniques in RL, this work was the first to convincingly demonstrate that RL agents could be trained on raw, high-dimensional observations, solely based on a reward signal. The second standout success was the development of a hybrid DRL system, AlphaGo, that defeated a human world champion in Go [128], paralleling the historic achievement of IBM’s Deep Blue in chess two decades earlier [19] and IBM’s Watson DeepQA system that beat the best human Jeopardy! players [31]. Unlike the handcrafted rules that have dominated chess-playing systems, AlphaGo was composed of neural networks that were trained using supervised and reinforcement learning, in combination with a traditional heuristic search algorithm’ (Arulkumaran et al., 2017, p.1)

These engines utilized pattern recognition and reinforcement learning to accomplish these feats. Successful chess engines have often relied on heuristics and learning as there are simply too many combinations on a chess board for Brute Force calculation. The exception to this is at the end of the game when there are few pieces left on the board. In this case Brute Force methods are used to generate all possible positions in the form of a look-up-table known as a tablebase.

In the source code for the chess engine “Stockfish”, one of the leading chess engines for many years, I found that Stockfish does utilize these tablebases, stopping calculating when it reaches these positions. However, it doesn’t adjust the algorithm before this point or apply specific heuristics or other algorithms leading up to this point as my project suggests.

I have also found lots of evidence of the growing cost of machine learning on the environment and the problems that come with this level of data.

‘The data mining area requires high-performance computing. The research often lacks computational resources capable of processing current algorithms.

Nevertheless, there is a decisive factor in whether or not to use technology, tool, or algorithm since machines containing a high-performance processor can be very expensive.’ (Novais et al., p 15)

Novais et al. (2021) compared a Brute Force algorithm with a smarter sequential algorithm and found that the Brute Force algorithm has greater energy efficiency

‘reaching at least 1.79x more operations per energy consumption than other algorithms on different architectures explored in this work’ (p. 1)

‘Processing time and energy consumption toward knowledge discovery, big data, and other areas are closely related to the evolution of architectures’ (Novais et al., 2021, p.2)

Han and Dally (2018) focus on the efficiency of deep learning computing and discuss problems of memory bandwidth, network bandwidth and engineering bandwidth.

‘Due to the complexity of the solution domain often, meta-heuristic methods such as genetic algorithms are applied to driving speed curve optimization by searching for the best train coasting points. Other advanced methods such as artificial intelligence neural net-work and fuzzy logic have also been employed to improve the efficiency and results of the optimisation. However, exhaustive search (exact algorithms) such as the Brute Force algorithm provides a more straightforward approach than meta-heuristics, and, importantly, they are guaranteed to find optimal solutions and to prove the optimality. However, the performance of these algorithms is not satisfactory in real-time train control as the computation time grows exponentially with the problem size. In order to overcome this drawback, and Enhanced Brute Force searching method has been used to constrain the solution domain, therefore reducing the computational cost.’ Zhao, Roberts & Hillmansen (2012) focus on driving speed curve for trains and find that ‘the Brute Force algorithm provides a more straightforward approach than meta-heuristics’ (p.159)

Thinke et al. (2016) discuss:

‘the topology optimisation of wireless networks using a Brute Force algorithm’.

They are concerned with reducing computational complexity of algorithms.

Oh et al. (2020) are also concerned with efficiency of algorithms and compare a deep learning algorithm with a Brute Force algorithm for renewable energy sources in a distribution network.

In a blog post by Bitvore, it discusses the use of Brute Force in AI. It uses the example of a Rubik's cube to illustrate the difference between AI and Brute Force. The post suggests that for problems with enumerable moves or well-defined configurations, Brute Force can be effective. However, for problems without these characteristics, an algorithmic approach may be more suitable.

'Not every job needs a hammer. Not every problem looks like a nail. Not every text and content analysis problem needs AI. Building AI models comes with an operational and computational cost. It's important to know where that benefit kicks in on the return-on-investment curve. Sometimes small, well-defined problems are better suited to small(ish), deterministic solutions. It's important to pick the right tool for the task.'

Methodology

After initial research on the idea, the first steps in the project involved setting up the general execution framework environment for the project, followed by testing.

Methodology Phase I

- Python Environment
- Game Loop Design
 - TicTacToe Game Logic
 - Create Grid
 - Display Grid
 - Textual Layout
 - Custom GUI (later)
 - Get Player Move
 - Text based Coordinate input
 - Custom GUI (later)
 - Get Computer Move
 - Check Win
 - Check Draw
- Validate game logic through manual test, correcting errors

Methodology Phase 2

- Brute Force Algorithm Design (computer move)
- Source an AI (computer move)
 - Integrate with game framework
- Create an automated game driver (player move)
- Testing of BF and AI

Methodology Phase 3

- Generalization form 3x3 grid to NxN grid
- Add 'Point of Transition' using *both* Brute Force and AI.
- Experiments and Testing (continuous refinement)
- Collect & Analyze Results

Brute Force Algorithm Design

Create a Brute Force algorithm capable of generating every position and evaluating the outcome of that position based on optimal play. There were many examples online of minimax

algorithms similar to this, but they were not exactly what I was looking for this is discussed further in appendix C. The way I designed the tablebase was modelled off the chess tablebase and thus requiring generating all positions, Cartesian product (grid size choose 3 to the power of the grid size), filtering these down to only allowing legal positions, assigning results to these positions based on optimal moves being played in a top – down approach. This worked very well for a 3x3 grid; however, the computational cost was realized as it was visible in human time to see the results generated.

However, the primary function of this sort of a tablebase is as a look-up-table, and for this reason when I designed an algorithm to pick a move it simply views all moves as equal as they all lead to a draw with best play. For this reason, I introduced a scoring system whereby the function to pick a move would find the number of possible results available in a following position which does not assume the opponent will play the best move.

This algorithm is a perfect example of an exponential increase when it gets scaled up to a bigger grid size. When I implemented the tablebase generation for a 4x4 grid the CPU and memory of the machine immediately skyrocketed, and I patiently waited for 30 minutes before it had finally classified and assigned 20 million positions into the tablebase. This only increases for a 5x5 grid.

Source an AI

From reading AI literature, I determined that a reinforcement learning AI was ideally suited to the problem due to its ability to learn from experience, and its use of tasks (input/output). I originally consider using a page-rank algorithm (teacher recommendation) to evaluate moves as a form of reinforcement learning, however I decided to use the more sophisticated/adaptable Q-learning AI algorithm. I sourced an implementation of this AI that was part of an opensource GitHub AI repository [tbd], and modified this for used with an NxN tictactoe grid. The AI came with a useful trainer that implemented a set of prioritized heuristics mimicking a human-intuitive strategy.

Automated Game Driver

I originally modelled the tester using a custom variant the trainer; however this led to very limited results due to the repetitive nature of the tester. It meant that only a slight aspect of the algorithm's ability was being tested. This is discussed in more detail in appendix C. Ultimately I replaced this with a random mover generator to get broader coverage and a truer general comparison between AI and brute-force.

Generalization form 3x3 grid to NxN grid

The next step was to change both algorithms to work for an NxN board rather than just the standard 3x3 board. This proved particularly challenging in the case of the teacher as most of the heuristics and strategies were hard-coded (specific 3x3 grid coordinates were coded). This meant creating abstract mathematical operations to convert these based on the grid size.

After a significant amount of beta testing to ensure there were no bugs, the next step was to set up the environment for experiments and testing.

Experiments and Testing

For the experiments I was fundamentally concerned with two outputs of the algorithms:

- ***Efficiency (cost)***
- ***Accuracy***

Efficiency is usually measured in one of three ways: time, space and energy. These can be measured as functions of cost. Energy and space (in memory) were ruled out for me due to a lack of equipment for precise measurements. Another common measurement of algorithms is how it deals with scaling up the input also known as Big-O Notation. I wasn't concerned with this because these algorithms were only designed for this specific task and due to the nature of the problem it would be exponential.

The basic idea is:

- implement the algorithm
- measure the time it takes to implement and run
- repeat for averages to maximize accuracy
- plot as a function of the cost.

For the experiments I decided to test the wall time, CPU time and the percentage CPU used. However, on my computer, like most computers, the algorithm I was trying to benchmark wasn't the only thing running. There are many other things using processor resources such as operating system tasks, other user processes etc.

This means that the results will be imprecise due to the noise of other processes, and the measured elapsed time will contain other factors than the algorithm running.

Noise Reduction:

- Repeat every test three times and got an average.
- Close all other applications running and disabled some background process.
- Measured CPU time as well as wall time

Although, in hindsight the percentage CPU didn't add any significance to the project as the utility used to measure would just randomly measure the percentage CPU being used during the process, potentially when the algorithm wasn't doing any work leading to a wide variety of results with no obvious connection. Furthermore, I could only have one thread assigned to the task at a time, so it was maxed at 25% (1 or 4 CPU core threads) the whole time. For these reasons I decided to use the wall time for accurate cost comparison.

Please see appendix C for a more detailed description of preparing for experimentation and testing.

Results

For this investigation 3 types of game engines were used

- Q-learning AI
- Brute Force algorithm
- Hybrid game engine
 - N engines – corresponding to grid squares
 - alternative points of transition between constituent engines.

I conducted tests with grid size 3, 4, and 5 (partial) for all the experiments.

A detailed table of the raw recorded results is in [Appendix D]

The graphs in this section show results obtained 4x4 grid

The following graphs demonstrate the optimal cost of executing any game engine depends on three factors: complexity of task, accuracy required and the number of iterations.

optimal cost = function(task-complexity, accuracy, #iterations)

Fixed and Execution Costs

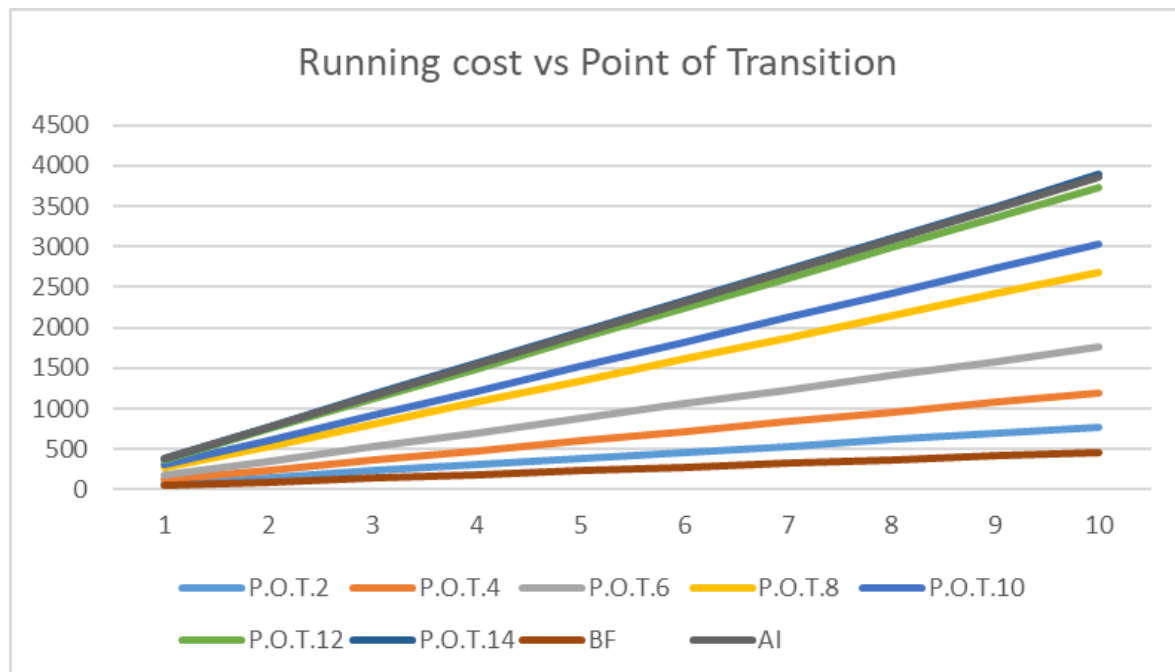


Chart: Linear running cost (uS) v Game Engine – (POT=Point/Move of Transition)

On a per iteration basis running the AI is many times as expensive as the Brute Force.

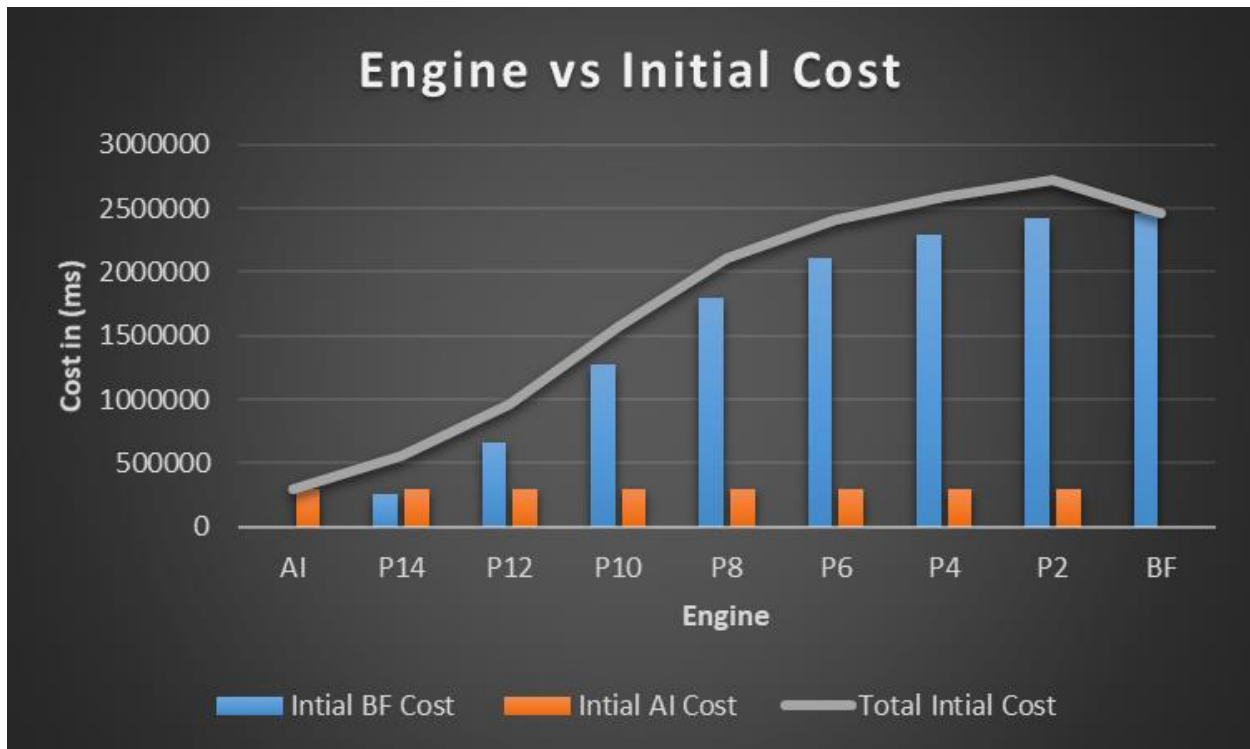


Chart: Total fixed costs (ms) for each game-engine combination/transition

The Brute Force has a colossal initial cost, making it less appealing if only one iteration is needed.

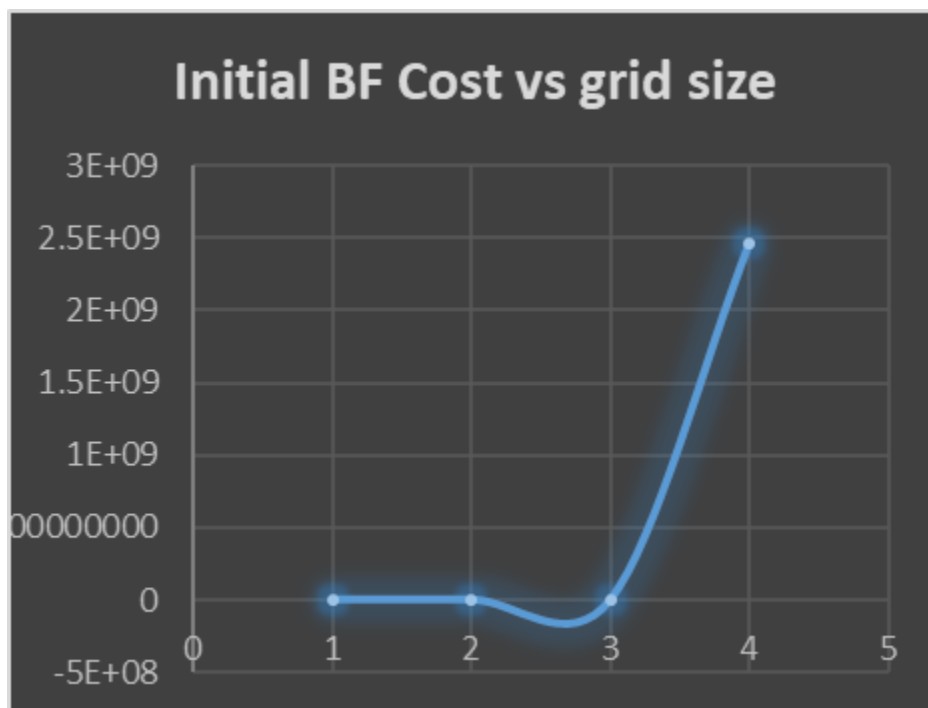


Chart: Fixed costs for Brute Force grow exponentially with larger grids

In terms of the complexity of the task, again we can see that Brute Force search suffers from exponential growth as the complexity scales up.

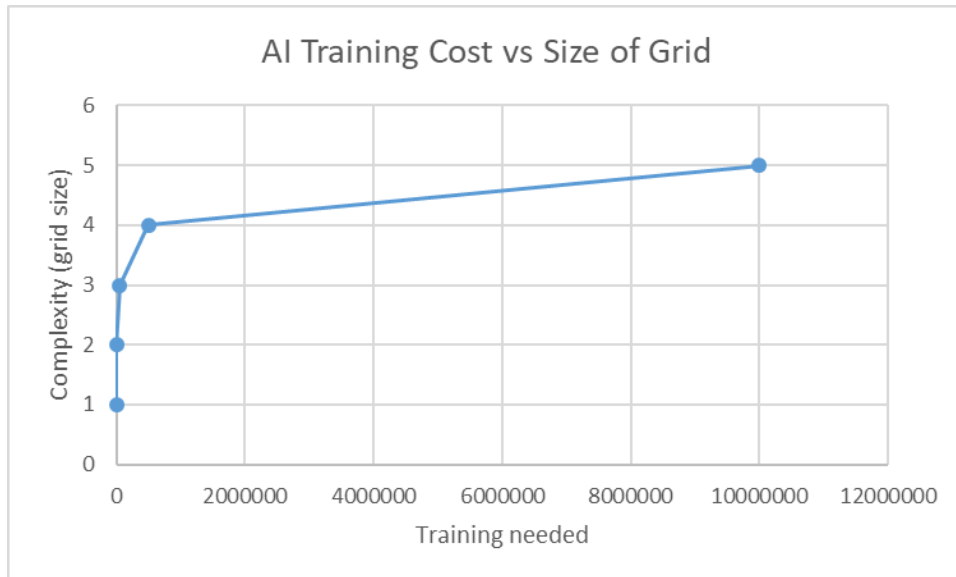


Chart: The AI-training to active a target level of accuracy is exponential

The AI Q-learning model also increases exponentially as the grid size increases.

Its accuracy didn't improve much until 50,000 games were played for the 4x4. On the 5x5, even with 100,000 games its accuracy did not improve at all.

Accuracy

The accuracy required is another critical component in determining the cost. If you want the algorithm to be 100% accurate all the time, Brute Force is the only solution. However as mentioned, due to its exponential increase in computational cost it is not always practical and in games like chess, impossible.

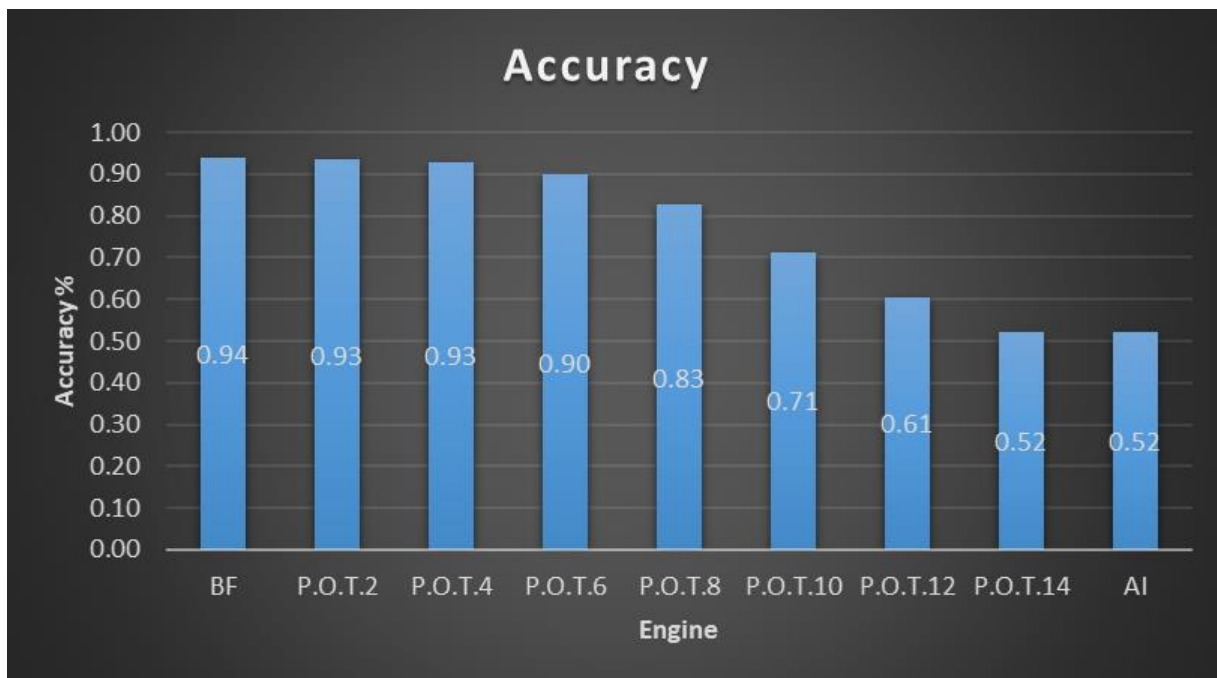


Chart: Game engine accuracy falls-off with less Brute Force and more AI

The Accuracy here weights a win and a loss as equivalent, and equal to 2 draws. The Brute Force engine achieves the best accuracy, and AI the worst. Hybrid engine accuracy falls off as the number of moves executed by brute force goes down.

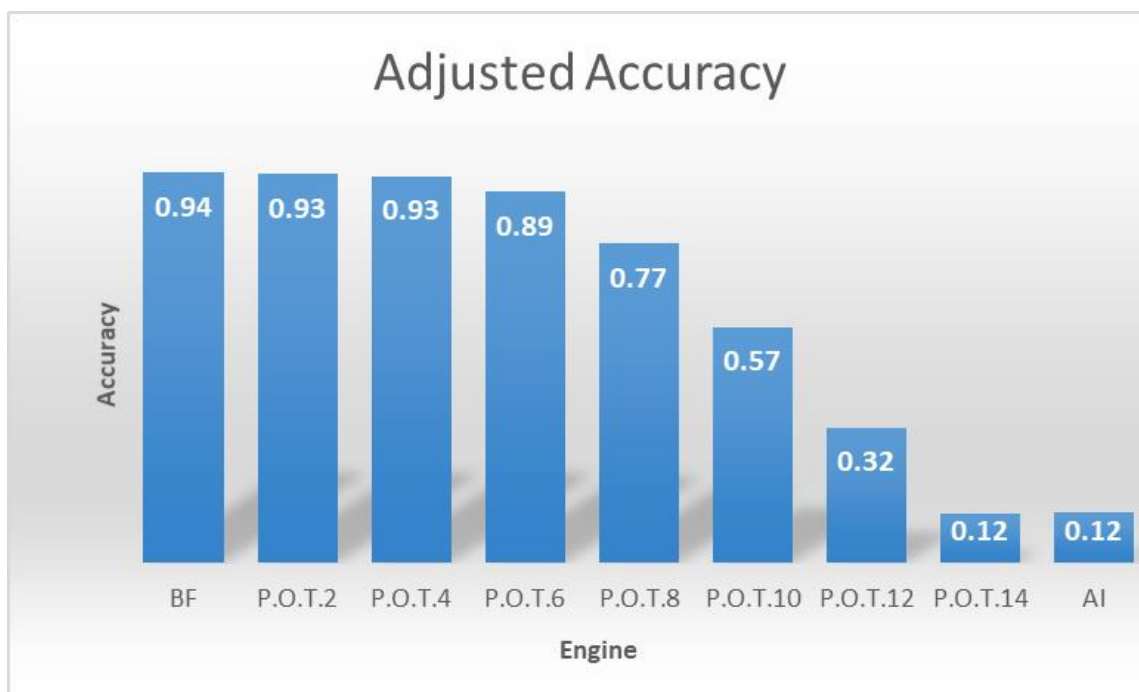


Chart: Loses outweighing wins in accuracy calculation (alt. game result criteria)

The Accuracy here weights a loss much less favorably than a win (x4) – representing a change in game outcome evaluation criteria that in perhaps more intuitive. This view further exaggerates the fall off in accuracy as more game moves are made by AI.

Optimal Point of Transition

Transition by Game Iterations

In order to calculate the optimal point of transition between methods the number of iterations must be known.

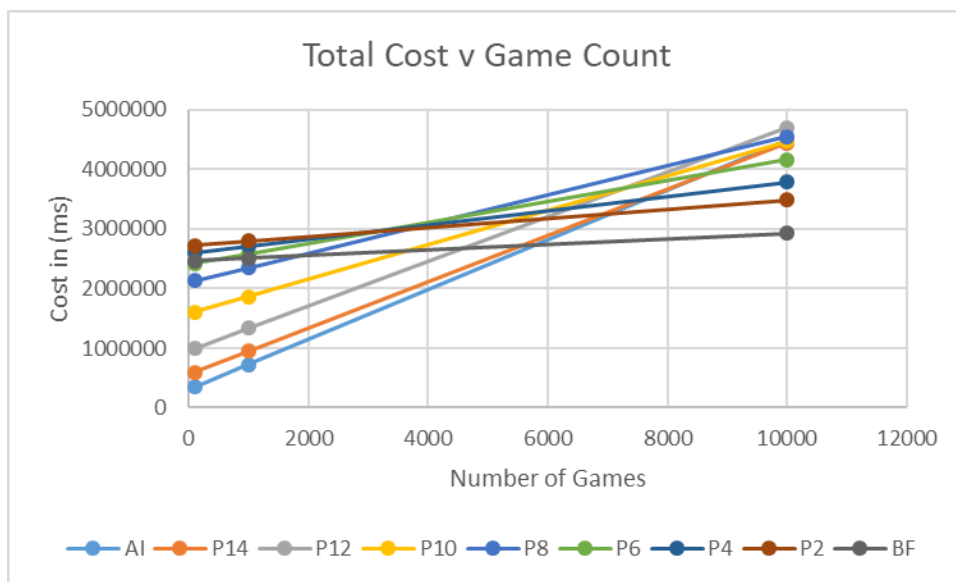


Chart: Fixed+Running costs for more games – AI cost grows most quickly

This graph demonstrates at what point it becomes economical to switch between AI and Brute Force purely based on the number of iterations. The precise point of intersection between the AI (light blue) and Brute Force (grey) can be shown.

Up until this intersection point the AI is the cheapest however it also has the lowest accuracy, so this does not tell the full picture. Also, for more complex problems Brute Force cost will skyrocket and may not be available. The alternative engines which transition at different points between the AI and Brute Force do not look very appealing. However, their accuracy is better as pointed out on the next graph for a specific number of iterations.

Transitions by Accuracy

Game Iteration Count: 1

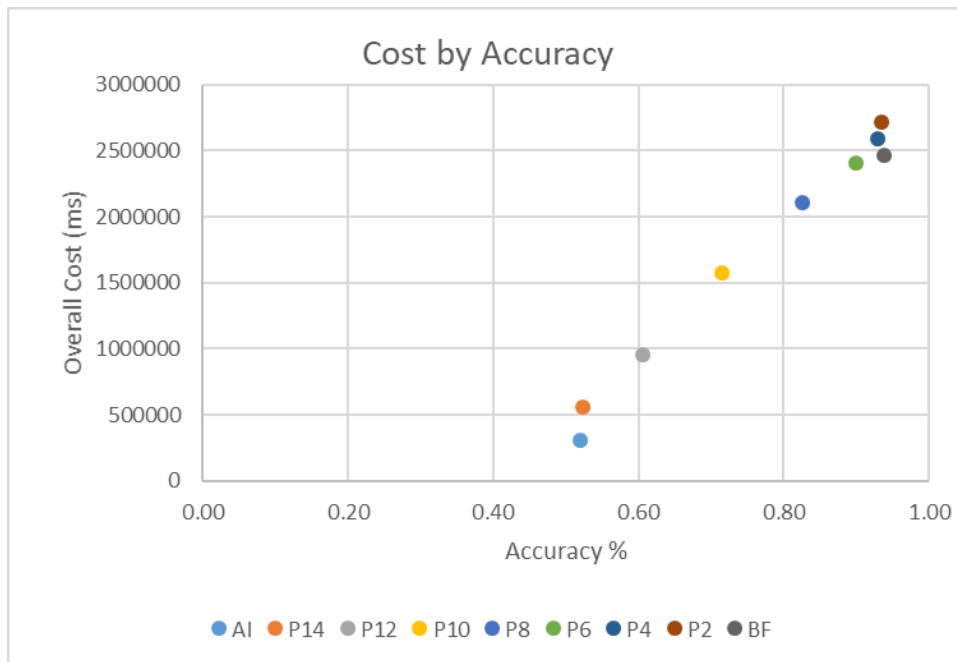


Chart: Given usage & minimum accuracy, select game-engine hybrid by cost

This graph represents the options available, and the cost-accuracy trade-off is clearly visible. If accuracy isn't a big concern, AI is clearly the best option, by contrast the Brute Force algorithm is the best to maximize accuracy. A compromise between the two would be to switch at 12 (grey), 10 (yellow) or 8 (blue). This graph was taken for one iteration.

Game Iteration Count: 500

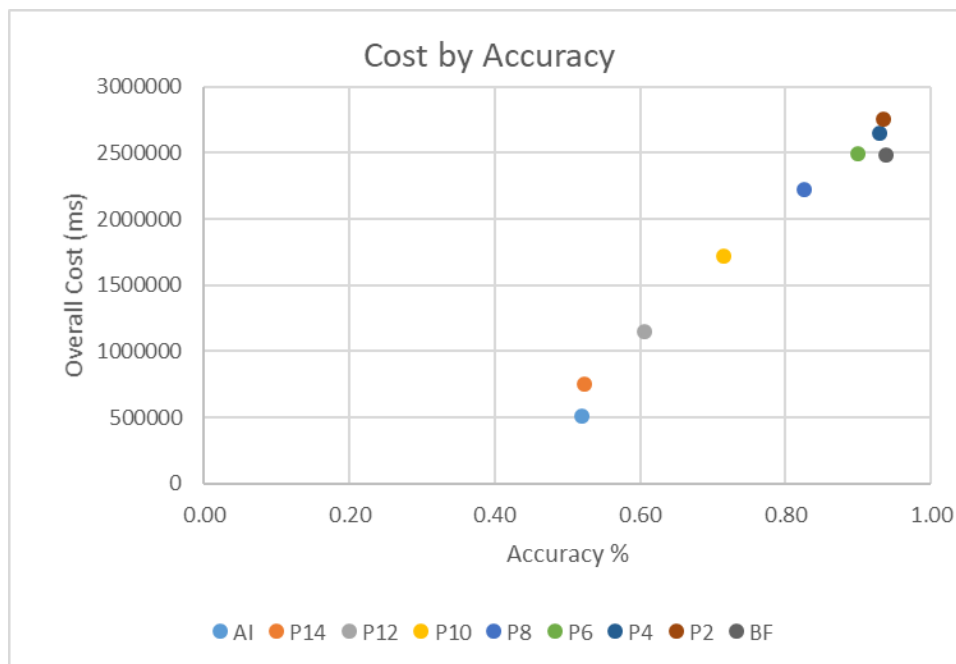


Chart: Given usage & minimum accuracy, select game-engine hybrid by cost

The engine cost grow as the iteration count increases. The AI is most affected due to its higher per-game execution cost.

Game Iteration Count: 5000

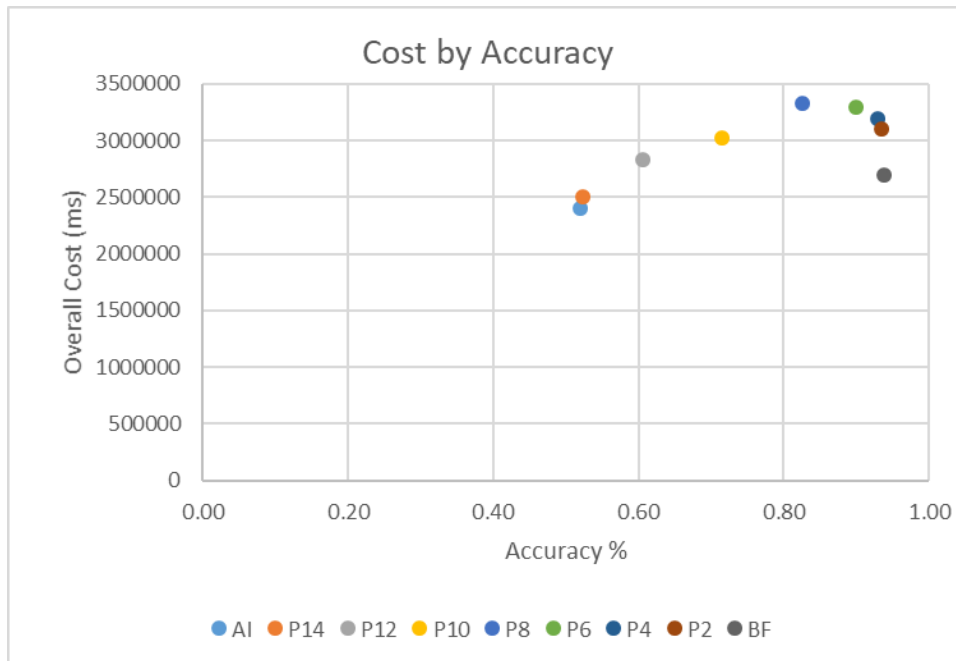


Chart: Given usage & minimum accuracy, select game-engine hybrid by cost

Suddenly the initial cost means much less and Brute Force becomes a lot more appealing almost economical.

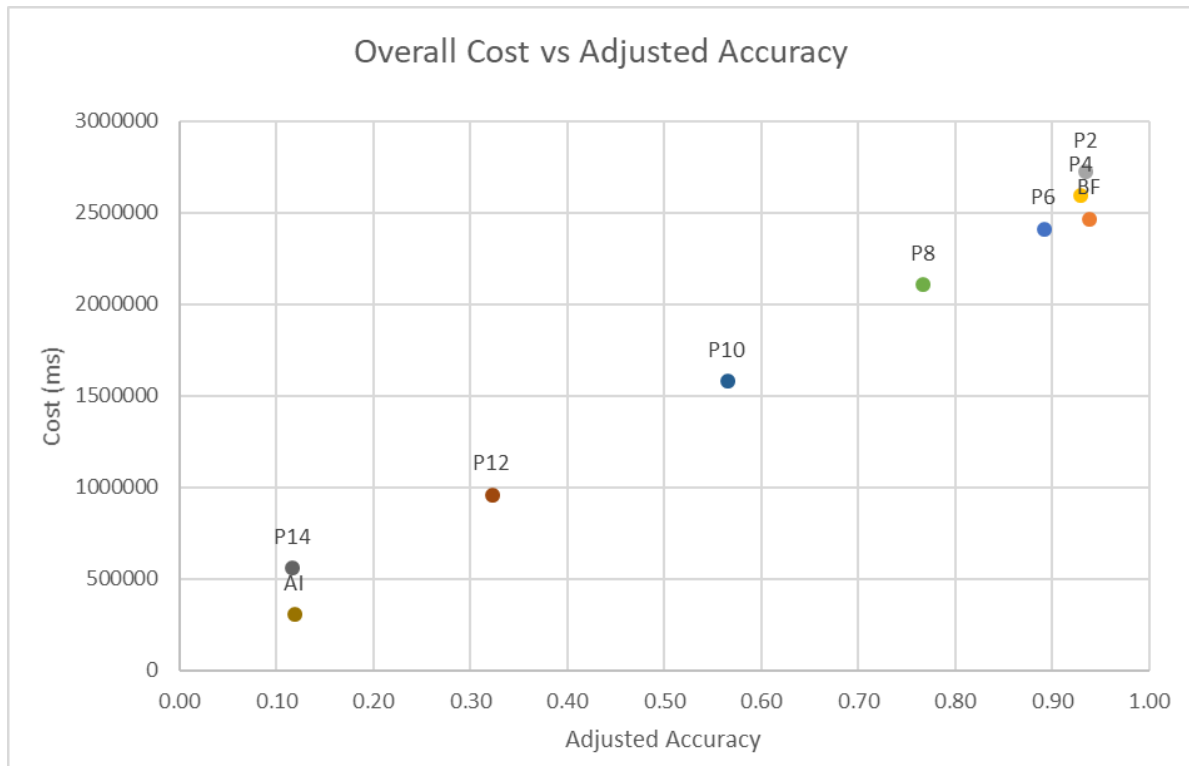


Chart: Given usage & minimum accuracy (weighted), select game-engine hybrid by cost

As I mentioned earlier, for some tasks the accuracy test may not represent a true picture due to the randomness of the AI. For this reason, I have included a graph with the adjusted accuracy of the engines.

Overall Transitions

Based on these graphs it is possible to recommend the best engine for this task, based on three input parameters: accuracy, complexity and the number of iterations. If someone says they want to get the best engine for a 4x4 grid for 500 iterations and an accuracy requirement of 70% I can recommend using the algorithm that switches at move 10 (P10), as this is the cheapest engine which meets all their requirements.

All the tables from where this data came from are in [Appendix D] as well as the graphs for a 3x3 grid.

Discussion

The results clearly demonstrate that an optimal point of transition can be calculated for this generalized tictactoe experimental space. If the intersection point occurs after the required number of iterations the most cost efficient strategy may be to stick with the one algorithm the whole time. However, the method I have shown can reveal if you should transition between algorithm, at what point is the optimal for saving cost and exactly how much you will save by following this combination.

As mentioned, this area of research is hugely important at a time when AI algorithms and massive quantities of data are easily accessible. It is important that we understand the environmental damage of inefficient computing and wasting resources. It is more important than ever that we maximize computational efficiency by optimizing the algorithms that are used and that we look for straightforward and creative solutions to these problems.

These results clearly highlight the energy that can be saved on a few small optimizations and show how a few small changes to the algorithm can result in monumental savings. For any given task it is important to analyze all possibilities and consider transitioning between algorithms if the task can be decomposed into more niche subsections. This can not only increase computational efficiency and save resources but also increase accuracy and productivity. In the latter half of the discussion, I will talk about how the maths can assist in choosing the correct algorithm and making educated guesses.

The Brute Force algorithm is a perfect example of how having access to pre-calculated data, at a certain point where the computational cost can be justified saves energy in the long term and increases accuracy. In game theory, usually regardless of how complex the game is at the end of the game some form of Brute Force can be applied and both increases accuracy for engines while also saving energy. Brute Force is the perfect algorithm to exploit this niche section of the game. This could be taken a step further by finding the point where Brute Force can no longer be applied. In a tictactoe game this could look like a position between move 5 and 7, before the Brute Force is applied.

Another niche algorithm could be applied here to capitalize on a specific task. A supervised learning reinforcement AI algorithm could serve well here, only calculating until it hits the results of the Brute Force. It could potentially spot patterns in this small section due to it only having to calculate a limited number of moves ahead and increase accuracy instead.

If computational cost is to be restrained it is essential to cut some of the work and rely on AI algorithms, heuristics, alpha-beta pruning and pattern recognition to optimize cost.

This leads me to believe that for some problems many different algorithms can be used and transitioned between to maximize efficiency and accuracy. The experiments I conducted were applied to a basic problem domain. However, they proved the concept that transitioning

between algorithms can optimize cost. The methodology demonstrated here to calculate the optimal combination of AI and Brute Force algorithms to maximize cost efficiency can also be applied in other domains. This concept is simple, yet it can have huge implications for many areas involving large quantities of data which could use AI and Brute Force algorithms.

These could include other areas in game theory like chess, online gaming, financial and climate modelling and cryptography.

Conclusion

I first set out on this project to quench my curiosity about the use of combining and transitioning between algorithms in the game of chess. The hypothesis I proposed at the start of this project was that an optimal point of transition can be calculated between AI and Brute Force in a game of N dimensional tic-tac-toe. I have proven that it can be based on the size of N, the number of iterations and the accuracy required. Once I had proven the concept, I started exploring with bigger grid sizes and this quickly revealed the limitations of the Brute Force algorithm, and how if it is to work at all it must be combined with other algorithms.

This project has really opened my eyes to the massive quantities of data and computational power so many algorithms require that are running all around us. The idea of breaking a problem down into smaller sections and handing those sections off to more niche algorithms is not new. In fact, it is done all the time often referred to as specialization. Finding the optimal amount of specialization, and at what point to pass on to the next algorithm is the concept I am trying to emphasize. This is not to say that it has not been thought of before, however I simply wish to look at this from a different perspective and point out the huge potential this concept has in other areas of data science.

In conclusion, I have found that the optimal point of transition can be calculated between AI and Brute Force for this specific task and can be calculated based off the complexity of the task, the number of iterations, and the accuracy required. This algorithm may involve transitioning between algorithms at specific points of transition. Now that I have proven this concept, I hope to highlight its potential to save computational resources and improve accuracy in many applications.

The experiments I conducted were applied to a basic problem domain. However, they proved the concept that transitioning between algorithms can optimize cost. The methodology demonstrated here to calculate the optimal combination of AI and Brute Force algorithms to maximize cost efficiency can also be applied in other domains. This concept is simple, yet it can have huge implications for many areas involving large quantities of data which could use AI and Brute Force algorithms.

These could include other areas in game theory like chess, online gaming, financial and climate modelling and cryptography.

As I mentioned in the literature review, I was incredibly surprised at how little I could find on this topic both in terms of research and experimentation. It was noted that major chess engines do use transition to tablebases or Brute Force and factor this into their calculations. This supports the idea of transitioning between AI and Brute Force does work. I am very curious as to how far this concept can be taken and where it could be implemented in more complex processes.

References

1. Arulkumaran, K., Deisenroth, M. P., Brundage, M., & Bharath, A. A. (2017). A brief survey of deep reinforcement learning. *arXiv preprint arXiv:1708.05866*.
2. Han, S., & Dally, W. J. (2018, June). Bandwidth-efficient deep learning. In *Proceedings of the 55th Annual Design Automation Conference* (pp. 1-6).
3. Jones, M., Nikovski, D., Imamura, M., & Hirata, T. (2016). Exemplar learning for extremely efficient anomaly detection in real-valued time series. *Data mining and knowledge discovery*, 30, 1427-1454.
4. Novais, J. P., Maciel, L. A., Souza, M. A., Song, M. A., & Freitas, H. C. (2021). An open computing language-based parallel Brute Force algorithm for formal concept analysis on heterogeneous architectures. *Concurrency and Computation: Practice and Experience*, 33(18), e6220.
5. Oh, S., Yoon, Y., & Kim, S. (2020). Online reconfiguration scheme of self-sufficient distribution network based on a reinforcement learning approach. *Applied Energy*, 280, 115900. <https://doi.org/10.1016/j.apenergy.2020.115900>.
6. Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., ... & Hassabis, D. (2017). Mastering the game of go without human knowledge. *nature*, 550(7676), 354-359.
7. Thike, A., Lupin, S., & Vagapov, Y. (2016). Implementation of Brute Force algorithm for topology optimisation of wireless networks. *2016 International Conference for Students on Applied Engineering (ICSAE)*, 264-268. <https://doi.org/10.1109/ICSAE.2016.7810200>.
8. Zhao, N., Roberts, C., & Hillmanssen, S. (2014). The application of an enhanced Brute Force algorithm to minimise energy costs and train delays for differing railway train control systems. *Proceedings of the Institution of Mechanical Engineers, Part F: Journal of Rail and Rapid Transit*, 228(2), 158-168.
9. GitHub account from where I sourced the open source AI - <https://github.com/rfeinman/tictactoe-reinforcement-learning>

I used all of the following websites at various points to gather information for my project:

10. <https://blog.bitvore.com/where-does-artificial-intelligence-use-Brute-Force-instead>
11. <https://realpython.com/tic-tac-toe-ai-python/>
12. <https://github.com/official-stockfish/Stockfish/blob/master/src/search.cpp>
13. <https://chess.stackexchange.com/questions/19233/do-stockfish-and-the-tablebase-work-together-or-independently>
14. <https://twice22.github.io/tictactoe/>
15. <https://blog.enterprisedna.co/how-to-generate-all-combinations-of-a-list-in-python/>
16. <https://note.nkmk.me/en/python-itertools-product/>
17. [Using Functions in a Sketch | Arduino Documentation](#)
18. [Intelligent Arduino Uno & Mega Tic Tac Toe \(Noughts and Crosses\) : 5 Steps - Instructables](#)
19. <https://www.instructables.com/Tic-Tac-Toe-6/>
20. <https://www.instructables.com/Tic-Tac-Toe-on-Arduino-With-AI-Minimax-Algorithm/>
21. <https://cseweb.ucsd.edu/~kube/cls/12.s13/Lectures/lec06/lec06.pdf>
22. <https://cseweb.ucsd.edu/~carter/260/260class05.pdf>

23. <https://ieeexplore.ieee.org/abstract/document/8649493>
24. [Brute Force vs. Smart Design: An Important Choice Between Raw Computational Power and Care in Mobility AI | by Evgeny Klochikhin | DataSeries | Medium](#)
25. <https://studyalgorithms.com/theory/how-do-you-compare-two-algorithms/>
26. <https://stackoverflow.com/questions/15455048/releasing-memory-in-python>
27. <https://pynative.com/python-get-execution-time-of-program/>
28. <https://towardsdatascience.com/the-fundamentals-of-the-big-o-notation-7fe14210b675>

Appendix A - Environment for tests

1.1 Developing the environment for the test.

This involved designing the experimental ground for the test, which was tic-tac-toe game logic. The full code for this is contained in appendix E 1. I also had to include a timing utility for the whole game and the length of time (both wall time and CPU time) it took for the relevant algorithm to run. I also added an optional GUI display board which I took from the pygame library.

1.2 Developing the Brute Force Algorithm

The Brute Force algorithm code is contained in Appendix E 2. The Brute Force algorithm was modelled off a chess tablebase (look-up-table). The simple idea behind it is to generate every possible position. Discard the illegal positions and assign the result to the remaining position. This would start by looking at the positions with no blanks and assigning a result to all these positions (all draws in this case as wins aren't included). Then the algorithm works backwards assigning results to the next set of positions (with one blank). Assigning positions is based on subbing the next piece into each blank and evaluating what the next position will be based on optimal play. This process continues populating all the tablebases until they have all been generated.

In order for this to function as an engine I also included the best move in each position that will transfer to the next position, which selected the move that was best for the player based on a scoring system visible in the code.

1.3 Finding a suitable reinforcement learning AI algorithm

After a bit of research the Q-learning model seemed very appropriate for my project as it didn't require a massive knowledge of machine learning or updating the algorithm to a specific task. The underlying idea of the Q-learning algorithm is quite simple: There are possible states, and each state there are actions which will transfer it from one state to another. Executing an action in any state will lead to a reward (numerical value, good or bad). The goal of the AI is to maximize its reward, it does this both by calculating the greatest possible reward attainable in future states and the results of actions it has carried out in other states. This way it learns as it is playing and is highly adaptable.

1.4 Transitioning the algorithm to NxN

This turned out to be quite complex in some cases particularly for the Brute Force and AI teaching algorithm. One of the problems I faced in the Brute Force was iterating through each position for each row in the grid size. Originally, I used 3 for loops to do this, however when changing to N dimensional this was impossible. The idea that I used was based on indexing each iteration to the power of the grid size to iterate through each position and put the combination to the power of the index and took the modulus value of each combination by the number of combinations until the index was increased and I

floor divided by the number of combination. Code snippet below, please see Appendix E for full code.

```
numCombs = len(combs)
for c in range(0, numCombs**gridSize, 1):
    game = []
    for i in range (gridSize-1, -1, -1):
        game.append( combs[(c // numCombs**i) % numCombs] )
```

There were also many challenges in converting the teacher algorithm for the AI to N by N as I mentioned, previously, coordinates had been hard-coded into it. Appendix E also contains the code for that.

1.5 Setting up relevant options:

As I mentioned in the document I ran all the code through the command line so I set up some options for convenience. There are three options on display off, text and GUI which shows a graphical board. There is an option to choose which algorithm plays as which player or whether the user wants to play. There is also a choice to get both the Brute Force and AI algorithms to transition at a specified point of transition, and there was an option to pick the number of games played. These options were all very helpful to efficiently run the experiments.

Appendix B - Tic-tac-toe game theory

I believe it is essential to understand the underlying mathematics behind the task to pick the right algorithm and test the robustness of the solution.

I will first introduce concepts related to tictactoe, then how this would apply when converting it to N dimensions and finally I will show why this is important for the Brute Force and AI algorithms.

The first question that must be asked is how many possible Tictactoe board positions are there? If you imagine there is a blank board, there are 9 possible choices to make on the first turn. Each choice will lead to its own unique position. Now it is Os turn to move. For all the 9 possible positions O can choose from 8 remaining squares, when the turn goes back to x it will have 7 choices available. The total number of combinations would be $9 \times 8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 362,880$ positions. This is known as 9-factorial or $9!$.

However, given that in each square there are only 3 possible choices in any given position X, O or blank, we can use permutations to calculate the maximum possible positions = $3^9 = 19683$

This is because each player can only play an X and O, so they aren't different pieces. Note that both order and repetition are allowed here as an X played in square on one move is technically a different position than an X played on that square on a different move.

However, many of these are illegal because this includes positions with any quantity of one piece, when there can only be a maximum of 5 Xs and 5 Os in each position.

To find the correct number of combinations another combinatorics method is used so I can count the number of combinations of Xs and Os at each move, explained later on. Which gives 5478 legal combinations.

Many of these combinations are simply just other positions rotated or reflected. To find the true number of unique combinations we must utilize a lemma (Burnside's Lemma) which is a consequence of group theory. Please see appendix B for how this Lemma works and the calculations for this project.

If this method is followed for all game positions, there are 765 distinct (orbit) positions.

Make it NxN:

To give a rough approximation for the number of positions permutations can be used as there are 3 choices for each square and the grid size squared number of squares. The formula would look something like this: $3^{(n^2)}$.

For a 4x4 this would give just over 43 million positions and for a 5x5 this would give close to 850 billion positions.

Again, these numbers don't include positions where an incorrect and illegal quantity of one piece exists.

To find only positions which only contain legal quantities of each piece, a bit more combinatorics is required.

On any given move the total number of positions that contain the correct number of Xs can be given by $n^2 \text{ choose } x$. Where x is the number of Xs on the board. Similarly, the number of combinations of Os in any given position can be given by $(n^2 - x) \text{ choose } o$.

Hence the number of total combinations as any combination Os can be assigned with any combination of Xs is $(n^2 \text{ choose } x) * ((n^2 - x) \text{ choose } o)$.

For each move from 0 to grid size squared this can be applied. The x will always equal $(\text{moves} + 1) // 2$ and O will always equal $(\text{moves}) // 2$ and therefore for each move the number of combinations can be calculated.

Here is a sample table:

Gridsize N:	1	2	3	4	5
Move 0	1	1	1	1	1
Move 1		4	9	16	25
Move 2		12	252	240	600
Move 3		12	72	1680	6900
Move 4			756	10920	75900
Move 5			1260	43680	531300
Move 6			1520	160160	3542000
Move 7			1140	400400	16824500
Move 8			390	895950	75710250
Move 9			78	1433520	257414850
Total	1	29	5478	...9722011	...161995031226

This table gives us a pretty accurate overview of all the combinations and the first $2n - 3$ moves it matches the numbers that are in the tablebase generated by the Brute Force. However, after $2n - 3$ moves it is possible for X to win which is included in these results and not in the tablebases. At $2n - 2$ moves one win is possible. At $2n - 1$ to $4n - 3$ two wins are possible and so on.

To calculate the number of positions that should be in the tablebase and factor out illegal combinations it is essential to not to include the wins. A win is given by n of either an X or an O in the same direction. First count the number of wins. There are n rows, n columns and 2 diagonals on any NxN board. Each win can happen in the same place with a different order of Xs. E.g. On a 3x3 grid, you can imagine a situation where 3 Xs occur in a row ($X_1X_2X_3$) however for the purpose of combinatorics this win would be different ($X_1X_3X_2$) and so to ($X_2X_1X_3$) ($X_2X_3X_1$) ($X_3X_1X_2$) ($X_3X_2X_1$). So, for each combination of wins on a 3x3 grid there are 6 combinations or 3! Or n!. This means that the total number of wins can be given by $2n(n!) + 2(n!)$

Up to $2n-3$	$0*(2n(n!)+2(n!))$	0 wins
$2n-2$	$1*(2n(n!)+2(n!))$	1 win (X)
From $2n-1$ to $4n-3$	$2*(2n(n!)+2(n!))$	2 wins (X) and (O)
$4n-2$	$3*(2n(n!)+2(n!))$	3 wins 2(X) and (O)
From $4n-1$ to $6n-3$	$4*(2n(n!)+2(n!))$	4 wins 2(X) and 2(O)
$6n-2$	$5*(2n(n!)+2(n!))$	5 wins 3(X) and 2(O)

Adding all these wins up for:

3x3 grid there are:	$11(2n(n!) + 2(n!))$
4x4 there are:	$26(2n(n!) + 2(n!))$
5x5 there are:	$46(2n(n!) + 2(n!))$

These calculations form the basis for the tablebase calculations and are essential to understand to predict the optimal point of transition between converting between the two approaches. As I was generating the 3x3 tablebases I printed the tablebase, the number of positions in the tablebase and the time it took:

```

Tablebase(N) - entries - generation time (ms): 9 16 47233
Tablebase(N) - entries - generation time (ms): 8 222 65170
Tablebase(N) - entries - generation time (ms): 7 696 97034
Tablebase(N) - entries - generation time (ms): 6 1372 144476
Tablebase(N) - entries - generation time (ms): 5 1140 120975
Tablebase(N) - entries - generation time (ms): 4 756 95678
Tablebase(N) - entries - generation time (ms): 3 252 65034
Tablebase(N) - entries - generation time (ms): 2 72 38083
Tablebase(N) - entries - generation time (ms): 1 9 39104
Tablebase(N) - entries - generation time (ms): 0 1 2521
PS C:\andrew\my\btvs>

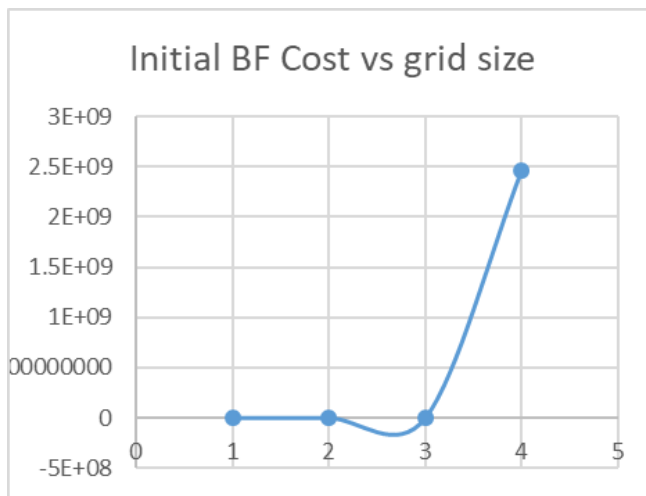
```

As you can see this is very similar to the combinations table above, only the wins are factored out. This task clearly scales up massively as the complexity or grid size increases and the 4x4 tablebases are much bigger:

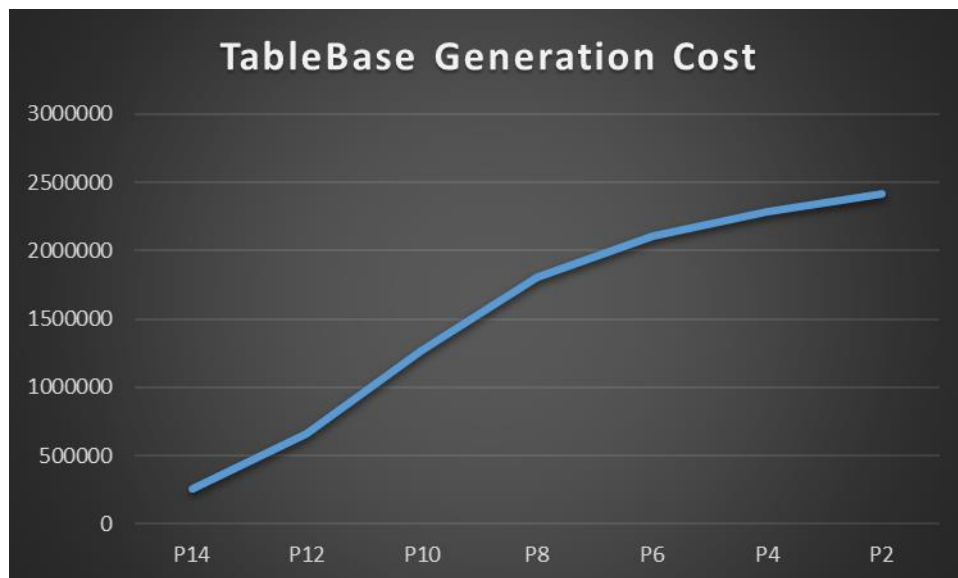
```

Tablebase(N) - entries - generation time (ms): 16 5356 53092962
Tablebase(N) - entries - generation time (ms): 15 53440 93195017
Tablebase(N) - entries - generation time (ms): 14 270304 115870643
Tablebase(N) - entries - generation time (ms): 13 714208 174593718
Tablebase(N) - entries - generation time (ms): 12 1415400 256507132
Tablebase(N) - entries - generation time (ms): 11 1800624 313941257
Tablebase(N) - entries - generation time (ms): 10 1908592 349365176
Tablebase(N) - entries - generation time (ms): 9 1394128 341990666
Tablebase(N) - entries - generation time (ms): 8 891026 268312713
Tablebase(N) - entries - generation time (ms): 7 398200 187765530
Tablebase(N) - entries - generation time (ms): 6 160160 140486014
Tablebase(N) - entries - generation time (ms): 5 43680 106012307
Tablebase(N) - entries - generation time (ms): 4 10920 93496799
Tablebase(N) - entries - generation time (ms): 3 1680 75404068
Tablebase(N) - entries - generation time (ms): 2 240 62762019
Tablebase(N) - entries - generation time (ms): 1 16 52150580
Tablebase(N) - entries - generation time (ms): 0 1 2312

```



This graph gives some indication of the exponential nature of the task and Brute Force algorithm when the grid size increases.



This graph is based off the tablebase generation cost data with respect to the point of transition. As shown in the maths above, because the quantities of data in the tablebase are almost appear quadratic in shape (figure) this means that this curve is not smooth and begins to look more like a Z shape indicating what could be potentially good value for cost.

The 5x5 grid simply requires too much memory and computational resources to Brute Force and generate all the tablebases. However, this is a perfect example of how algorithms can be transitioned between to optimize accuracy. Generating the last few tablebases for a 5x5 grid improves the accuracy of the algorithm.

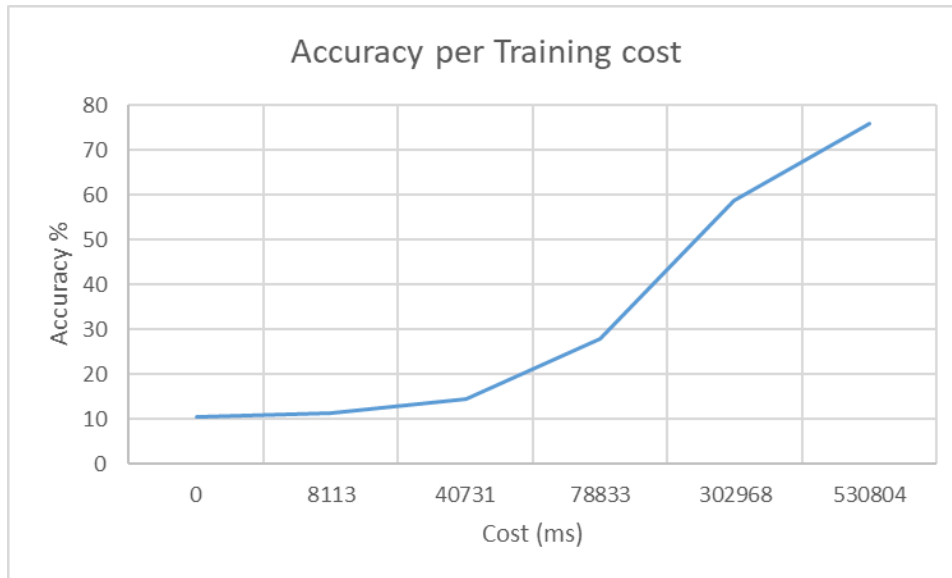
We can consider the maths above for the number of possible positions in each tablebase a robustness test for the Brute Force algorithm. It both verifies that the algorithm has correctly generated the right number of positions and provides insights into the limitations and uses of the algorithm.

The next algorithm to consider is the Q-learning reinforcement algorithm. This algorithm does not do anything special specific to this task and can represent any algorithm being considered. I am just providing some background information so we can look at some of the maths behind it.

Reinforcement learning involves an agent, a set of *states* and a set of *actions* per state. By performing an action, the agent transitions from state to state. Executing an action in a specific state provides the agent with a *reward* (a numerical score).

The goal of the agent is to maximize its total reward. It does this by adding the maximum reward attainable from future states to the reward for achieving its current state, effectively influencing the current action by the potential future reward. This potential reward is a weighted sum of expected values of the rewards of all future steps starting from the current state.

As I have already calculated the number of states based on the grid size, it is evident that AI will not be able to reach every state particularly on the bigger grids. As seen in the graph below:



At first, there is a very slight and steady rise as the AI is still very much learning and hasn't encountered sufficient states yet to learn enough. This is followed by a sharp rise in accuracy as the AI accumulates enough states to pass a threshold from where its model rapidly increases. From there the rise slows right down and the AI is no longer able to learn as much from each run. For the project's purpose, I tried to pick the AI at the optimal point in terms of accuracy versus cost, i.e. at the top of the steep section. This is very much dependent on the size of the grid, for the 3 I approximated 50,000 and the 4 I approximated 500,000 games.

The underlying maths is very useful for selecting the correct algorithm and preparing educated guesses as to where the optimal point of transition may lie.

Extra Detail

Calculating the number of orbits:

To show how this works for a let's take just the final positions where there are only Xs and Os on the board.

This means that there are maximum of 2^9 positions but really only $9!/(5!*4!)$ or 9 choose 5 positions which equals 126.

Burnside's Lemma states that the number of orbits (true unique positions) is equal to the average number of fixed points (Positions that are unaffected when they are rotated or reflected).

There are 4 orbits: 0-degree turn, 90-degree turn, 180-degree turn, and 270-degree turn.

There are 4 reflections: horizontal, vertical, north-east south-west diagonal, and the north-west south-east diagonal.

0-degree turn: **126**

All 126 positions are fixed

90 and 270 degree turns: **2 and 2**

If a position is to be fixed for a quarter rotation the corners must all be the same. Because there are only 4 Os, this means that this can only occur in two combinations for each rotation.

180-degree rotation: **6**

For a position to remain fixed on a half turn the diagonally opposite corners must be the same and the opposite sides must be the same. Once again because there is a maximum of four Os there are only 6 combinations where this is possible.

For all the reflections: **12 + 12 + 12 + 12**

Each side of the reflection must be the same. There are 3 squares on each side affected by the reflections because the axis runs down the remaining 3. On one side there are 2 elements (pieces) to be chosen from. Because there can only be 4 Os both an X and an O must be chosen. These can be arranged in 6 ways. However, there are also two positions with only 1 orbit which contain 4 fixed reflections each. For this reason, $6 + (4-1) + (4-1) = 12$ fixed reflections per axis.

Therefore the total number of fixed positions is $126 + 2 + 2 + 6 + 12 + 12 + 12 + 12 = 184$

To get the average divide by 8 and the answer is 23.

If this method is followed for all game positions, there are 765 distinct (orbits) positions.

2.3 AI maths

I have already shown the graph showing the accuracy of AI versus the cost of training it making an 'S' shape curve. Here I will show the formula for updating Q values the algorithm I borrowed and

Appendix C - Methodology

Detailed description of precautions and pre-experimental process to set up test

The first step was researching how best to measure the computational cost of implementing and running the algorithms. With the help of many useful websites I have referenced in the reference section, I decided the most appropriate measurements to take for the equipment I had access to was the run time.

I did a lot of research on the best algorithm for the project. For the Brute Force algorithm, most websites were recommending a simple Brute Force minimax algorithm, which is a very effective because of the limitations of the 3x3 grid. However, this wasn't quite what I was looking for because it doesn't scale up well at all. I wanted to base my design off the chess tablebases that can both act as a look-up-table and an engine. I could not find an example of this online so I had to design it myself which was a very time consuming process.

For the AI algorithm I was unsure what algorithm to use. I knew I wanted a reinforcement learning AI because this would be essential for contrast. Many of the reinforcement AIs available were highly complex and specific to a few tasks. I considered attempting to design my own based on a SARSA (State, Action, Reward, State, and Action) with a page rank algorithm to select the best move, but this would have taken a long time. Then I found a Q-learner AI algorithm on an open source GitHub page (mentioned in the references). This was perfect for the project as it is model free, can be adapted easily, achieves good results, and saves me a lot of time. My work on the AI was almost all converting it and its trainer from a 3x3 grid from which it was designed to work for an N by N grid.

This was very difficult in the case of the teaching algorithm as it was all based on human-like heuristics which were hard coded into the algorithm with specific coordinates.

When all the algorithms were finally in order it was time for me to start preparing for the experiments.

Many precautions were taken to ensure testing was as fair and accurate as possible. After a significant amount of beta testing to ensure there were no bugs, the next step was to set up the environment for experiments and testing.

For the experiments I was only concerned with two outputs of the algorithms, their accuracy and efficiency. Efficiency is usually measured in one of three ways: time, space and energy. These can be measured as functions of cost. Energy and space (in memory) were ruled out for me due to a lack of equipment for precise measurements. Another common measurement of algorithms is how it deals with scaling up the input also known as Big-O Notation. I wasn't concerned with this because these algorithms were only designed for this specific task and due to the nature of the problem it would be exponential.

For the experiments I decided to test the wall time, CPU time and the percentage CPU used. I soon realized that I could only assign one thread to the problem so the CPU percentage was capped at 25%, and the CPU time had less meaning. However, on my computer, like most computers, the algorithm I was trying to benchmark wasn't the only thing running. There are many other things using processor resources such as operating system tasks, other user processes etc.

This means that the results will be imprecise due to the noise of other processes, and the measured elapsed time will contain other factors than the algorithm running.

To ensure the results were as accurate as possible:

I repeated every test three times and got an average. I rigorously checked these tests to search for outliers, or excessive deviation and ran the results through Excel. I had to abandon the CPU time due to excessive deviation. Even though it was only three tests average each test consisted of 1000 or 5000 iterations so there was lots of data to ensure the sample wasn't too small.

I closed all other applications running and disabled some background process. I ran all the tests through the command line. Between each test I would delete old files created and copy across any relevant files for the next test. Ideally I would have liked to have used a Linux box designed for to run a single program as a windows machine has many background process and I couldn't fairly test the CPU time.

I ran the garbage collector before the experiments to free up memory. This was quite a computational intensive process so I had to make sure it wasn't running during the process.

[Detailed description of the experiments](#)

I ran all the experiments through the command line to ensure that there were no distractions by other applications running. Each test was time stamped and I copied it into an Excel sheet, from where I cross-checked the time stamp of the test and updated the table. The testing in all took over 12 hours to run as all files needed to be copied over and old files deleted between each test. I ran these results for both the 3x3 grid and the 4x4 grid.

Each experiment was ran 1000 times in the case of the 3x3 grid or 5000 times in the case of the 4x4 grid. This process was then repeated 3 times for each test and the average values were used in the graphs.

Initially when I ran these test I discovered that the tests were unfair due to the design of the tester. The tester was designed to minimize the randomness to ensure the tests were as fair as possible. Even though objectively this way of testing may be considered 'fairer as there is no degree of randomness or luck, the results it was producing were not a true representation of the algorithms ability as only a limited variety of scenarios were tested due to tendency to repeat games and positions. This was a huge problem and threw the results way off and I was wondering why the algorithms were reaching so many of the same result positions. After a bit

of investigation I noticed that a lot of the positions were repeating. This forced me to reconsider and change the testing process. Instead, I used a random algorithm to test the AI and Brute Force algorithms. To compensate for the degree of randomness I increased the game count to ensure that all elements of the game were covered and an average could be calculated.

Appendix D - Results

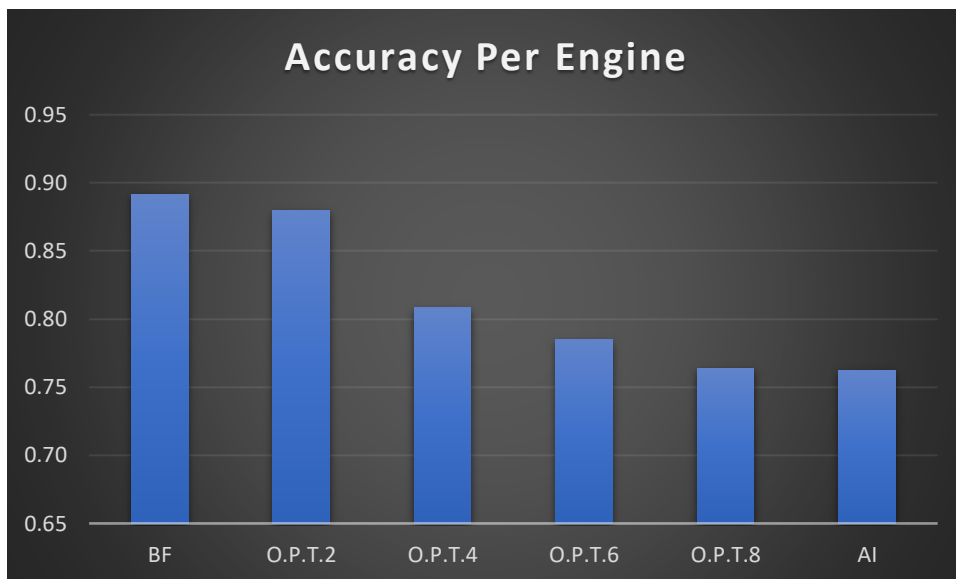
Table of results (3x3 grid)

Test Nr	Games	Grid size	engine	transition	total time	ave - time	Total CPU	Ave CPU	% CPU	win	lose	draw	ai training
14.1	1000	3	qLearner	0	231473	231	234375	234	25	38 3	50 7	110	0
14.2	1000	3	qLearner	0	235192	235	218750	218	24.7	33 2	53 7	131	0
14.3	1000	3	qLearner	0	242750	242	250000	250	28.6	36 1	52 7	112	0
15.1	1000	3	qLearner	0	122539	122	78125	78	25	82 3	76	101	10000
15.2	1000	3	qLearner	0	121892	121	140625	140	25	82 3	76	101	10000
15.3	1000	3	qLearner	0	123680	123	109375	109	25	82 3	72	105	10000
16.1	1000	3	qLearner	0	120338	120	62500	62	25	82 7	66	107	50000
16.2	1000	3	qLearner	0	119505	119	109375	109	27.1	83 1	64	105	50000
16.3	1000	3	qLearner	0	118843	118	109375	109	25	83 0	71	99	50000
17.1	1000	3	bf	0	21276	21	0	0	30	88 4	0	116	
17.2	1000	3	bf	0	20458	20	46875	46	26.3	88 3	0	117	
17.3	1000	3	bf	0	21240	21	46875	46	25	90 7	0	93	
18.1	1000	3	both	2	44059	44	31250	31	25.9	89 1	14	95	50000
18.2	1000	3	both	2	44043	44	31250	31	28.6	88 4	15	101	50000
18.3	1000	3	both	2	45666	45	46875	46	25	90 5	11	84	50000
19.1	1000	3	both	4	77872	77	46875	46	27.8	85 5	49	96	50000
19.2	1000	3	both	4	83071	83	62500	62	30	85 1	37	112	50000
19.3	1000	3	both	4	76525	76	46875	46	25	85 5	50	95	50000
20.1	1000	3	both	6	108604	108	140625	140	25	82 7	70	103	50000
20.2	1000	3	both	6	118820	118	46875	46	27.1	84 7	62	91	50000
20.3	1000	3	both	6	108249	108	140625	140	26.2	85 5	43	102	50000
21.1	1000	3	both	8	124724	124	171875	171	25	83 2	66	102	50000
21.2	1000	3	both	8	121685	121	140625	140	25	82 5	62	113	50000
21.3	1000	3	both	8	120987	120	171875	171	25	82 8	65	107	50000

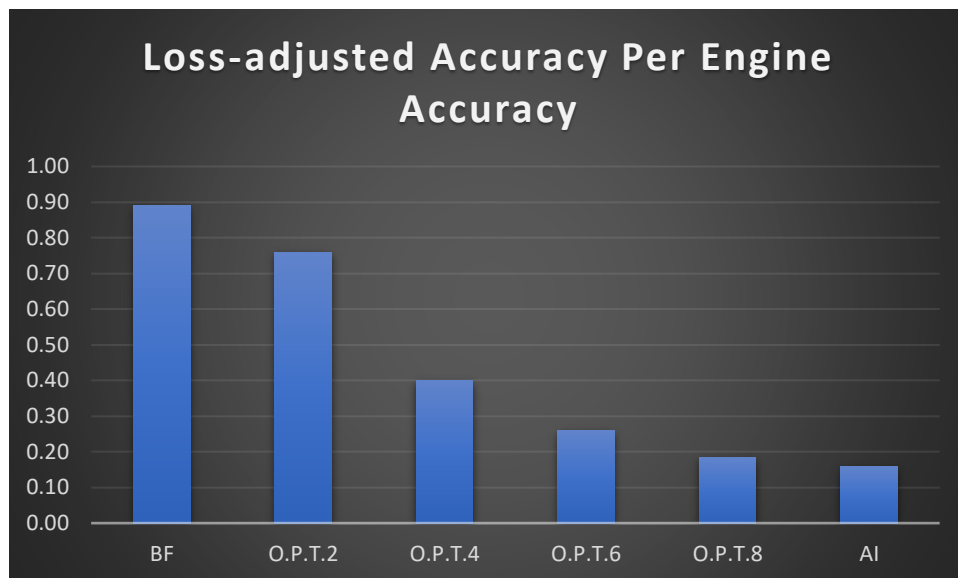
Analytics charts for 3x3 grid data



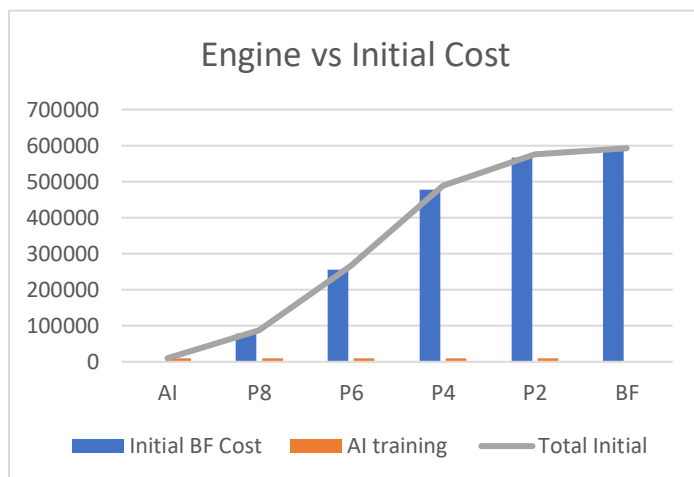
Run cost per iteration



Accuracy per engine



Loss adjusted cost per engine



Engine vs Initial cost

Table of results (4x4 grid)






























Test Nr	Gam es	gridSi ze	engin e	transiti on	total time	ave - time	Total CPU	Ave CPU	% CPU	win	los e	dra w	ai trainin g
1.1	5000	4	qLear ner	0	3042879	608	2953125	590	26	1333	1557	2110	0
1.2	5000	4	qLear ner	0	3030423	606	3015625	603	26	1306	1591	2103	0
1.3	5000	4	qLear ner	0	3182062	636	3312500	662	29.7	1309	1568	2123	0

2.1	5000	4	qLearner	0	3316462	663	3328125	665	25.7	1329	1561	2110	10000
2.2	5000	4	qLearner	0	3333332	666	3359375	671	25.8	1362	1564	2074	10000
2.3	5000	4	qLearner	0	3293959	658	3250000	650	25.1	1345	1550	2105	10000
3.1	5000	4	qLearner	0	3081887	616	3078125	615	26	1478	1551	1971	50000
3.2	5000	4	qLearner	0	3019713	603	3078125	615	25.6	1483	1493	2024	50000
3.3	5000	4	qLearner	0	2986515	597	3046875	609	27.2	1503	1511	1986	50000
4.1	5000	4	qLearner	0	2545772	509	2390625	478	26.6	1994	1357	1649	100000
4.2	5000	4	qLearner	0	2518932	503	2578125	515	25.7	2109	1257	1634	100000
4.3	5000	4	qLearner	0	2550297	510	2578125	515	27	2049	1345	1606	100000
5.1	5000	4	qLearner	0	2435023	487	1765625	353	26	3264	639	1097	500000
5.2	5000	4	qLearner	0	1925069	385	1906250	381	26.2	3285	675	1040	500000
5.3	5000	4	qLearner	0	1936674	387	1984375	396	26.7	3260	692	1048	500000
6.1	5000	4	bf	0	236653	47	203125	40	31.4	4688	0	312	0
6.2	5000	4	bf	0	227902	45	218750	43	25	4699	0	301	0
6.3	5000	4	bf	0	236091	47	296875	59	27.3	4688	0	312	0
7.1	5000	4	both	2	362660	72	375000	75	26	4632	0	368	500000
7.2	5000	4	both	2	384372	76	375000	75	33	4700	0	300	500000
7.3	5000	4	both	2	422796	84	421875	84	34.9	4691	0	309	500000
8.1	5000	4	both	4	620239	124	625000	125	25.9	4640	0	360	500000
8.2	5000	4	both	4	594281	118	656250	131	24.8	4641	0	359	500000
8.3	5000	4	both	4	591859	118	593750	118	24.9	4663	0	337	500000
9.1	5000	4	both	6	855242	171	953125	190	27.1	4541	15	444	500000
9.2	5000	4	both	6	864255	172	703125	140	26.1	4484	12	504	500000
9.3	5000	4	both	6	929062	185	953125	190	28.4	4504	12	484	500000
10.1	5000	4	both	8	1062500	212	1171875	234	25.2	4246	98	656	500000
10.2	5000	4	both	8	1241681	248	2187500	437	77.9	4228	102	670	500000
10.3	5000	4	both	8	1375265	275	1359375	271	38.2	4211	94	695	500000
11.1	5000	4	both	10	1435068	287	1390625	278	25.3	3809	258	933	500000
11.2	5000	4	both	10	1455310	291	1375000	275	26.3	3826	255	919	500000
11.3	5000	4	both	10	1462609	292	1640625	328	33.8	3820	232	948	500000
12.1	5000	4	both	12	1850383	370	1812500	362	34.2	3516	455	1029	500000
12.2	5000	4	both	12	1829649	365	1859375	371	33	3484	500	1016	500000
12.3	5000	4	both	12	1640625	328	1796875	359	37.7	3502	461	1037	500000
13.1	5000	4	both	14	1850383	370	1984375	396	37.1	3293	663	1044	500000

13.2	5000	4	both	14	188118 6	376	198437 5	396	27.2	327 2	687	104 1	500000
13.3	5000	4	both	14	205713 0	411	195312 5	390	34.6	330 4	679	101 7	500000

Appendix E – Python Code

Source Files:

 testPlayer Python Source File 10.7 KB	 teacherN Python Source File 10.0 KB	 teacher Python Source File 8.32 KB
 game Python Source File 8.06 KB	 tictactoe Python Source File 7.28 KB	 genericTb Python Source File 7.15 KB
 agent Python Source File 5.38 KB	 play Python Source File 4.73 KB	 guiDisplay Python Source File 4.65 KB
 tablebaseGenerator Python Source File 4.35 KB	 checkfork Python Source File 3.83 KB	 checkLegal Python Source File 2.81 KB
 bruteForce Python Source File 2.35 KB	 compVComp Python Source File 1.69 KB	 checkActive Python Source File 1.47 KB
 tableBaseData Python Source File 1.39 KB	 options Python Source File 1.24 KB	 convertGame Python Source File 1.11 KB
 userInput Python Source File 1012 bytes	 bruteForceAssignResult Python Source File 754 bytes	 bruteForceCheckForWin Python Source File 646 bytes
 BruteForceCheckWin Python Source File 646 bytes	 compInput Python Source File 437 bytes	 testing Python Source File 406 bytes
 randomCompMove Python Source File 400 bytes	 displayGrid Python Source File 334 bytes	 checkDraw Python Source File 280 bytes
 createGrid Python Source File 195 bytes	 blockfork Python Source File 0 bytes	

TicTacToe game file:

```
# Tic-tac-toe
import datetime
import time
import createGrid
import displayGrid
import userInput
import checkActive
import checkDraw
import compInput
import tableBaseData
#import tablebaseGenerator
import genericTb
import agent
import pickle
import os
import options
```

```

import psutil
import guiDisplay
import testPlayer
import randomCompMove

def main():
    opts = options.options()

    gridSize = opts.gridSize

    tablebase = []
    if opts.x == "bf" or opts.o == "bf" or (opts.switch0 != -1):
        tablebase = tableBaseData.loadTablebase(gridSize)

    display = None
    if opts.display == "gui":
        display = guiDisplay.displayGrid(gridSize)

    if opts.o == "ai":
        ag = agent.initQAgent(opts)

    if opts.x or opts.o == "tester":
        tester = testPlayer.TestPlayer(0.9, gridSize)

    totalTime = 0
    totalCpuTime = 0
    winXCount = 0
    winOCount = 0

    psutil.cpu_percent(interval=None)

    stt = time.perf_counter_ns()

    for i in range(opts.games):
        game = createGrid.createGrid(gridSize)
        move = 0
        gameTime = 0
        gameCpuTime = 0
        gameIsActive = True

        if opts.switch0 != -1:
            opts.o = "ai"

        row = -1
        col = -1

```



```

while gameIsActive == True:
    reward = 0
    if opts.display == "text":
        displayGrid.displayOGrid(game)
    if opts.display == "gui":
        display.displayGrid(game, row, col)

    if (opts.switch0 != -1) and move >= opts.switch0:
        opts.o = "bf"

    if move == 0:
        if opts.display == "text":
            print('***** New Game *****')
        if opts.display == "gui":
            display.draw_msg("New Game")

    # prompt user to input move
    if opts.x == "tester":
        row, col = tester.makeMove(game)
    elif opts.x == "player" and opts.display == "test":
        row, col = userInput.userInput(game, gridSize)
    elif opts.x == "player" and opts.display == "gui":
        gameIsActive, row, col = display.getMove()
        if gameIsActive == False:
            continue
    elif opts.x == "bf":
        row, col = genericTb.getMove(game, tablebase)
    elif opts.x == "ai":
        row, col = ag.get_action(agent.getStateKey(game))
    elif opts.x == "random":
        row, col = randomCompMove.randomCompMove(game, gridSize)

    # Check the user move
    game[row][col] = 'X'
    won, line = checkActive.checkActive(game, gridSize, row, col)
    if won == True:
        gameIsActive = False
        reward = -1
        winXCount += 1
        if opts.display == "text":
            print("Game is Over - you won")
        if opts.display == "gui":
            display.displayWin(game, line)
            display.draw_msg("Game Over - You Win")
            time.sleep(30)

```

```

elif checkDraw.checkDraw(game, gridSize) == True:
    gameIsActive = False
    if opts.display == "text":
        print("Game is Over - Draw")
    if opts.display == "gui":
        display.draw_msg("Game Over - Draw")
        time.sleep(3)

# Update position
if opts.display == "text":
    displayGrid.displayXGrid(game)
if opts.display == "gui":
    display.displayGrid(game, row, col)
    time.sleep(1)

move += 1

if gameIsActive:

    # Request computer move
    st = time.perf_counter_ns()
    sct = time.process_time_ns()
    if opts.o == "basic":
        row, col = compInput.compInput(game, gridSize)
    if opts.o == "bf":
        row, col = genericTb.getMove(game, tablebase)
    if opts.o == "tester":
        row, col = tester.makeMove(game)
    if opts.o == "ai":
        row, col = ag.get_action(agent.getStateKey(game))
    gameTime += time.perf_counter_ns() - st
    gameCpuTime += time.process_time_ns() - sct

    # update Q-values
    if move > 1 and opts.o == "ai":
        st = time.perf_counter_ns()
        sct = time.process_time_ns()
        ag.update(prevState, agent.getStateKey(game), prevMove, (row, col), 0)
        gameTime += time.perf_counter_ns() - st
        gameCpuTime += time.process_time_ns() - sct
        prevState = agent.getStateKey(game)
        prevMove = (row, col)

    # Check the computer move
    game[row][col] = 'O'

```

```

        won, line = checkActive.checkActive(game, gridSize, row, col)
        if won == True:
            gameIsActive = False
            reward = 1
            winOCount += 1
            if opts.display == "text":
                print("Game is Over - you lost")
                displayGrid.displayOGrid(game)
            if opts.display == "gui":
                display.displayWin(game,line)
                display.draw_msg("Game Over - You Lose")
                time.sleep(3)
        elif checkDraw.checkDraw(game, gridSize) == True:
            gameIsActive = False
            if opts.display == "text":
                print("Game is Over - draw")
                displayGrid.displayOGrid(game)
            if opts.display == "gui":
                display.displayGrid(game,row,col)
                display.draw_msg("Game Over - Draw")
                time.sleep(3)

        move += 1

    if opts.o == "ai":
        st = time.perf_counter_ns()
        sct = time.process_time_ns()
        ag.update(prevState, None, prevMove, None, reward)
        gameTime += time.perf_counter_ns() - st
        gameCpuTime += time.process_time_ns() - sct

    totalTime += gameTime
    totalCpuTime += gameCpuTime

trainingTime = time.perf_counter_ns() - stt

print(datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S"))
print("Total player-0 time (us): ",end='')
print(totalTime // 1000)
print("Average player-0 Time (us): ",end='')
print(totalTime/opts.games // 1000)
print("Total player-0 CPU - time (us): ",end='')
print(totalCpuTime // 1000)
print("Average player-0 CPU - Time (us): ",end='')
print(totalCpuTime/opts.games // 1000)

```

```

print("CPU percentage for run: ",end='')
print(psutil.cpu_percent(interval=None))
print("Games/Wins/Losses/Draws for player-0: ",end='')
print(opts.games, win0Count, winXCount, opts.games - (winXCount+win0Count))
print("Overall/Training time (ms): ",end='')
print(trainingTime // 1000000)

if opts.o == "ai" or (opts.switch0 != -1):
    ag.save(opts.aiDataFile)

main()

```