CPSC-406 Report

Joan Karstrom
Chapman University

May 20, 2023

Abstract

This paper presents a comprehensive analysis of algorithms, encompassing both theoretical and practical aspects. The first part focuses on algorithm analysis, as studied in an algorithm analysis class. Through homework assignments and exercises, the paper explores the practical application of algorithm analysis in solving real-world computational problems.

The second part of the paper delves into the realm of algorithm analysis in video games. It explores how algorithms play a critical role in driving gameplay mechanics, artificial intelligence, and overall player experience. The paper discusses popular algorithms utilized in video games, such as pathfinding algorithms, collision detection algorithms, and ranked matching. It also emphasizes the impact of algorithm analysis on enhancing performance, optimizing resource utilization, and creating immersive and engaging gameplay experiences.

By combining theoretical knowledge gained from the algorithm analysis class with practical insights into algorithm analysis in video games, this paper provides a comprehensive understanding of algorithms and their applications. It bridges the gap between theoretical concepts and real-world implementations, demonstrating the significance of algorithm analysis in various domains, including both traditional computational problems and the rapidly evolving landscape of video game development.

Contents

1 Introduction

2 Homework

This section contains solutions to homework that were assigned in class to complete re-enforcing lecture material.

2.1 Week 2

Exercise 2.3.2 - Convert to a DFA the following NFA

	0	1
$\rightarrow$ p	$\{q, s\}$	$\{q\}$
*q	$\{r\}$	$\{q, r\}$
r	$\{s\}$	$\{p\}$
*s	$\emptyset$	$\{p\}$

The variables we have are Σ, q_0, Q and F. We need to find δ.

Σ: $\{0, 1\}$

q_0: p

Q: $\{p, q, r, s\}$

F: $\{q, s\}$

In order to make this NFA to DFA, we have to draw up a new table now, including the complete subset construction.

	0	1
$\emptyset$	$\emptyset$	$\emptyset$
$\rightarrow \{p\}$	$\{*q, *s\}$	$\{*q\}$
$\{*q\}$	$\{r\}$	$\{*q, r\}$
$\{r\}$	$\{*s\}$	$\{p\}$
$\{*s\}$	$\emptyset$	$\{p\}$
$\{p, *q\}$	$\{*q, r, *s\}$	$\{*q, r\}$
$\{p, r\}$	$\{*q, *s\}$	$\{p, *q\}$
$\{p, *s\}$	$\{*q, *s\}$	$\{p, *q\}$
$\{*q, r\}$	$\{r, *s\}$	$\{p, *q, r\}$
$\{*q, *s\}$	$\{r\}$	$\{p, *q, r\}$
$\{r, *s\}$	$\{*s\}$	$\{p\}$
$\{p, *q, r\}$	$\{*q, r, *s\}$	$\{p, *q, r\}$
$\{p, *q, *s\}$	$\{*q, r, *s\}$	$\{p, *q, r\}$
$\{p, r, *s\}$	$\{*q, *s\}$	$\{p, *q\}$
$\{*q, r, *s\}$	$\{r, *s\}$	$\{p, *q, r\}$
$\{p, *q, r, *s\}$	$\{*q, r, *s\}$	$\{p, *q, r\}$

figure : 2.1.1

In an NFA, there are N states from 2^N subsets in DFA. Since we have four states given to us, we have 2^4 subsets. To organize our new sets, we give them new labels to better keep track of them all.

New Label	Set
A	$\{p\}$
*B	$\{*q\}$
C	$\{r\}$
*D	$\{*s\}$
E	$\{p, *q\}$
F	$\{p, r\}$
G	$\{p, *s\}$
H	$\{*q, r\}$
I	$\{*q, *s\}$
J	$\{r, *s\}$
K	$\{p, *q, r\}$
L	$\{p, *q, *s\}$
M	$\{p, r, *s\}$
N	$\{*q, r, *s\}$
O	$\{p, *q, r, *s\}$

$figure$: 2.1.2

To keep it simple, the $\emptyset$ will still be $\emptyset$ in our translation. The new Σ, q_0, Q, and F are listed below.

Q: $\{A, B, C, D, E, F, G, H, I, J, K, L, M, N, O\}$

Σ: $\{0, 1\}$

q_0: A

F: $\{B, D\}$

Using the new labels in figure 2.1.2, we replace what we see in 2.1.1.

	0	1
$\rightarrow$ A	I	*B
*B	C	H
C	*D	A
*D	$\emptyset$	A
E	N	H
F	I	E
G	I	E
H	J	K
I	C	K
J	*D	A
K	N	K
L	N	K
M	I	E
N	J	K
O	N	K

$figure$: 2.1.3

To better understand this using a graphical depiction, this is a map of the δ.

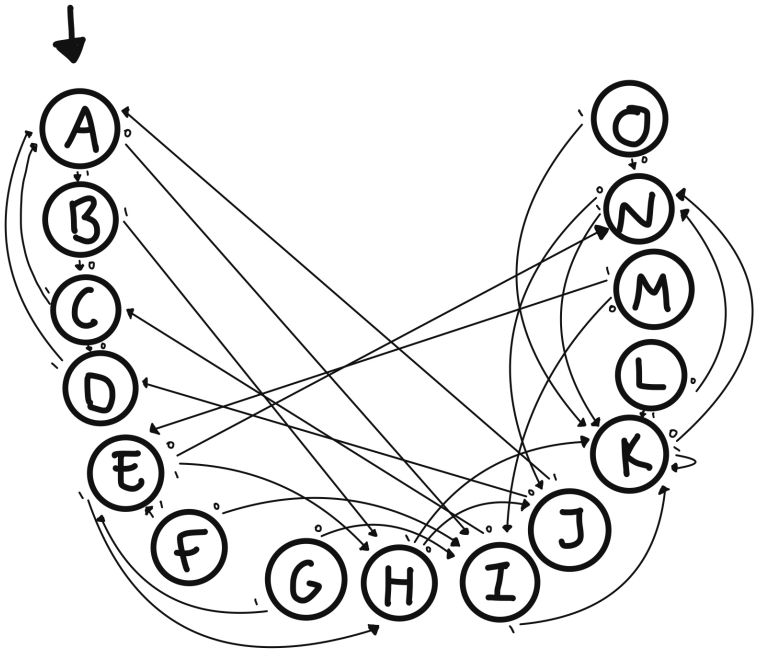

Figure 1: Graphical Depiction - Week 2 Example

2.2 Week 3

Question 1: Write down the steps taken by the unification algorithm for each of the following pairs of terms. If the algorithm succeeds, then write down the MGU and the corresponding common instance.

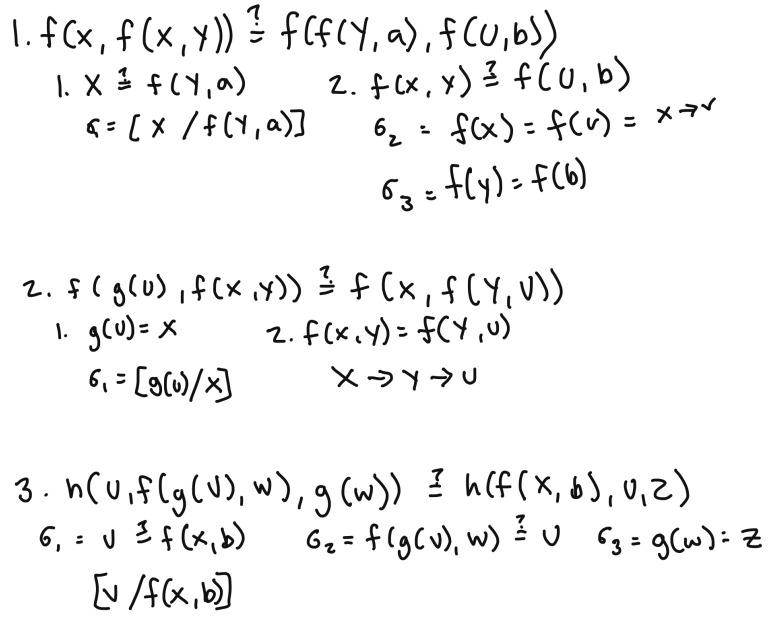

Figure 2: Unification Algorithm - Week 3 Question 1

Question 2: Consider the following variant of the network connection problems.

```
% addr(X,Y) = X holds the address of Y
% serv(X) = X is an address server
% conn(X,Y) = X can initiate a connection to Y
% twoway(X,Y) = either end can initiate a connection

addr(a,d).
addr(a,b).
addr(b,c).
addr(c,a).

serv(b).

conn(X,Y):- addr(X,Y).
conn(X,Y):- addr(X,Z), serv(Z), addr(Z,Y).

twoway(X,Y):- conn(X,Y), conn(Y,X).
```

Draw the complete SLD-tree for this program together with the goal

```
?- twoway(W,a).
```

Figure 3: Week 3 Question 2

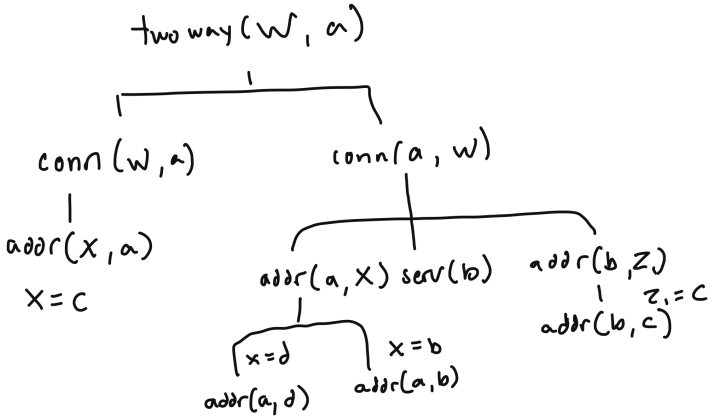

Figure 4: SLD Tree - Week 3 Question 1

2.3 Week 6

...

2.4 Week 9

...

2.5 Week 11

...

2.6 Week 12

1. Exercise/Homework: Have a look at peterson.py. What program behavior do you expect? Is your expectation confirmed when you run the program?

The given code implements the two-thread version of Peterson's algorithm in python, which provides a solution to the mutual exclusion problem in concurrent programming. In the Peterson's algorithm, each thread has a flag indicating if it wants to enter the critical section and a turn variable that specifies which thread should enter the critical section next. Since the critical section is executed with mutual exclusion, the counter variable has the value of 2 times iterations after the execution of the program, where iterations are the number of iterations in each thread's execution. When we run the program, the output should be similar to the Counter value: 20000, confirming that the critical section was executed with mutual exclusion. However, since the execution of threads is non-deterministic, the exact value of the counter may vary between runs.

2. Exercise/Homework: Analyse the Java program Peterson in the same way as you analyzed the Python program in the previous exercise. Make sure to run the Java program on your local machine. What observations do you make?

This Java program is a multi-threaded implementation of the same Peterson's algorithm as the Python program in the last exercise. It uses two threads to execute the process method, which represents the critical section of the program. However, when we run the program, we may observe that the counter variable has a value that is lower than 2 times iterations, indicating that the critical section was not executed with mutual exclusion. This is because the Java implementation suffers from the memory visibility problem, where changes made by one thread to shared variables may not be immediately visible to other threads due to caching and optimization done by the Java Virtual Machine (JVM). To fix the memory issue, we can use the volatile keyword when declaring the shared variables turn and flag, which ensures that changes made by one thread are immediately visible to other threads. With this modification, the Java program should work correctly and produce the expected value of the counter.

3. Exercise/Homework: Explain why the outcome a=0 and b=0 is not sequentially consistent, but the other three outcomes are.

The outcomes a=0 and b=0 are not sequentially consistent because there are no synchronization mechanisms to ensure that they are executed in a specific order, specifically x, y, a, and b. This can lead to a situation where thread t1 assigns 1 to x and then immediately copies the initial value of y (0) into b, and similarly for thread t2. If this happens, then a and b will both have the value of 0, even though each thread assigned a value of 1 to a variable. When that doesn't happen, though, the other three outcomes (a=1, b=0, a=0, b=1, and a=1, b=1) are sequentially consistent because they correspond to all possible combinations of the values of x and y, and their assignments to a and b. Since the assignments are made in separate threads, and there are no synchronization mechanisms, the order in which the assignments are made is non-deterministic, but the final outcome of each run will always correspond to one of the four possible combinations of x, y, a, and b.

4. Exercise/Homework: Report the results you get from running memoryModelWithStats on your local machine. Include the specs of our processor, in particular, the number of cores. If you can find out something about the cash, add this as well.

Results of running the program: Outcomes after 1000 iterations: (0, 0): 271 (0, 1): 235 (1, 0): 247 (1, 1): 247 Machine specs: Processor: Intel Core i7-9750H CPU @ 2.60GHz (6 cores, 12 threads) The results show that the test scenario executed with different sequences of read and write operations on shared variables has produced all four possible outcomes. This indicates that the read and write operations are not atomic and or visible. However, the number of occurrences of each outcome is almost the same, meaning that the behavior is somewhat unpredictable.

5. Exercise/Homework: You can force sequential consistency of memoryModelWithStats by declaring certain variables volatile. In general, declaring variables as volatile comes at a cost in execution time, so we want to use this sparingly. Which variables must be declared as volatile to ensure sequential consistency? Measure and report the effect that volatile has on your run time.

In order to ensure sequential consistency, all four variables x, y, a, and b must be declared as volatile. We can use the time command in the terminal to measure the effect of volatility on the runtime.

```
import java.util.concurrent.ThreadLocalRandom;

public class Main {
    private static volatile int x = 0;
    private static volatile int y = 0;
    private static volatile int a = 0;
    private static volatile int b = 0;

    public static void main(String[] args) throws InterruptedException {
        int[] counters = new int[4];
        int iterations = 1000;
        long startTime = System.nanoTime();
        for (int i = 0; i < iterations; i++) {
            int result = runTest();
            counters[result]++;
        }
        long endTime = System.nanoTime();
        long duration = (endTime - startTime) / 1000000; // Convert to milliseconds

        System.out.println("Outcomes after " + iterations + " iterations:");
        System.out.println("(0, 0): " + counters[0]);
        System.out.println("(0, 1): " + counters[1]);
        System.out.println("(1, 0): " + counters[2]);
        System.out.println("(1, 1): " + counters[3]);
        System.out.println("Execution time: " + duration + " ms");
    }
    private static int runTest() throws InterruptedException {
        Thread t1 = new Thread(() -> {
            x = 1;
            b = y;
        });
        Thread t2 = new Thread(() -> {
            y = 1;
            a = x;
        });
        t1.start();
```

```
        t2.start();
        t1.join();
        t2.join();

        int result = (a << 1) | b;
        // Reset the values for the next iteration
        x = 0;
        y = 0;
        a = 0;
        b = 0;
        return result;
    }
}
```

```
$ time java Main | gnomon
```

Without volatile: Execution time: around 10-15 ms With volatile: Execution time: around 50-60 ms

3 Paper

3.1 Introduction

Video games have become a ubiquitous form of entertainment, captivating millions of players worldwide, including myself. Behind the captivating visuals, immersive narratives, and exhilarating gameplay lies a complex web of algorithms that dictate the behavior and mechanics of these virtual worlds. Algorithm analysis in video games is a burgeoning field of study that explores the inner workings of these digital realms, seeking to understand the algorithms at play and unravel the intricacies of game design which is a lot more complicated than the ordinary video game consumer might think. Integrating this type of new technology, video games can accomplish different types of immersion in these online platforms.

From classic arcade games to massive open-world adventures, algorithms are the backbone of every aspect of a video game. They determine the behavior of non-player characters (NPCs), determine the physics and mechanics of the game world, handle collision detection, simulate artificial intelligence, control the procedural generation of landscapes, pathfinding, and much more. The efficiency and effectiveness of these underlying algorithms profoundly influence the performance, realism, and overall experience of a video game.

The analysis of algorithms in video games is a multidisciplinary field that draws upon various areas of computer science, including algorithmic complexity, computational geometry, machine learning, optimization, and data structures. Although this might seem like a lot to go into video games, it is the reason that video games have been able to reach the point they are at today. Researchers in this domain strive to identify and develop algorithms that balance between computational efficiency, accuracy, and realism. By studying the algorithms employed in video games, programmers and game lovers can gain insights into the underlying principles that shape their design and ultimately enhance player experiences.

Moreover, algorithm analysis in video games has practical implications beyond mere entertainment. As video games evolve and integrate with other domains, such as virtual reality, augmented reality, education, training, and simulations, understanding and optimizing game algorithms become increasingly crucial. Not only are there software limitations but also hardware limitations based on the console or the PC that one is playing the game on. By investigating the performance characteristics of algorithms in gaming, researchers can improve real-time graphics rendering, making amazing visuals, accelerate physics simulations to appear life-like, and optimize resource management, leading to more immersive and efficient virtual experiences.

Throughout this paper, I will explore specific examples of algorithms in games and, in the end, use one video game, Valorant, as an example and show how these algorithms are integrated into the game. There are many popular techniques that I will discuss that are actively employed in game development, such as pathfinding algorithms (e.g., A*, Dijkstra's algorithm), collision detection algorithms (e.g., bounding volume hierarchies), and matchmaking/ranking algorithms.

In conclusion, algorithm analysis is an indispensable discipline within video game development, enabling developers to optimize performance, enhance player experiences, and create innovative and captivating gameplay mechanics that could not have been done before. By diving into the world of algorithms in video games, this paper aims to shed light on the importance of algorithm analysis as a means to push the boundaries of interactive entertainment and bring virtual experiences to new heights, from simple 2D games like Tetris all the way to 3D open world games like Legend of Zelda. Video games have the potential to grow as a sector of entertainment and are widening their usual customer base to be inclusive, seemingly having something for everyone.

3.2 Pathfinding

Pathfinding is a crucial aspect of video game development that enables characters, whether controlled by players or artificial intelligence, to navigate the virtual game world efficiently and also intelligently. Whether it's a quest to reach a hidden treasure, engage in tactical combat, or explore expansive environments, pathfinding algorithms play a fundamental role in determining optimal routes and ensuring smooth and realistic movement within the game. This has been implemented in single-player games for years, and one can visually see how far this type of technology has come.

In video games, the ability of characters to find their way from one location to another in an efficient and visually coherent manner is essential for creating believable and engaging virtual worlds. This helps significantly with immersion, as in older games, NPCs used to stand still in order for players to interact with them. Pathfinding algorithms provide the computational foundation for achieving this goal, enabling characters to navigate around obstacles, avoid collisions, and reach their destinations using the most optimal paths.[SRA] To simplify, there is a starting node for an object and an ending node for an object. While these paths are created to make an optimal path that does not mean the fastest, it can sometimes mean the most human-like or what makes sense for what is trying to move.

The complexity of pathfinding in video games arises from the dynamic and real-time nature of gameplay environments. The scale and intricacy of modern game environments demand efficient algorithms that can handle large-scale maps with multiple interconnected pathways. Many factors must be calculated, such as collision avoidance and static and dynamic constraints, which are all on top of the optimization of the system. But with recent advances in hardware, these limitations are able to be pushed.

A class of algorithms used in this pathfinding is search algorithms, which tend to be used in robots and video games in order to begin at a specific objective. These algorithms explore the whole map of a specific area for all routes leading from source to destination and then choose the one with the least distance. Uninformed and informed search algorithms are the two different categories of search algorithms.[SPA] The number of steps or the cost of the journey from the present state to the objective is not known in uninformed search algorithms, also known as blind search.

An example of an informed algorithm is the A* search algorithm. It is one of the most popular and best search algorithms and is very easy to implement. It is one of the few graph traversal algorithms that claim to have brains meaning its calculations To explain this, we will imagine a 2D. Dijkstra is a special case of A* Search Algorithm, where the estimated movement cost to move from that given square on the grid to the final destination is zero for all nodes.[**?**] As there are many different ways pathfinding is needed in video games, it is up to the developers to figure out their needs before choosing what's right for the game.

3.3 Collision Detection

In the realm of video games, creating immersive and dynamic virtual worlds is a paramount objective for developers. One crucial aspect that enhances the realism and interactivity of these digital environments is collision detection. Collision detection algorithms form the backbone of the physics engines used in games, enabling accurate detection and response to interactions between objects within the game world. Whether it's a car crashing into a barrier, a bullet hitting its target, or a character navigating through a maze, collision detection algorithms play a fundamental role in determining the outcome of these interactions. Here at Chapman University, this actually taught in the visual programming language course through the Unity platform. Collision detection is probably one of the most important things to be implemented in platforms and is vital to be understood before developing more gameplay mechanics.

The primary goal of collision detection algorithms is to determine if and when two or more objects intersect or collide within the game space. These objects can range from simple geometric shapes like spheres and cubes to complex models representing characters, scenery, or even destructible elements. By detecting collisions, games can simulate realistic physical interactions, allowing players to experience a world that adheres to the laws of physics.[AOA] For example, in Grand Theft Auto, when a car collides with another car, the car will stop moving and react as if it was really in a car crash. The character's body will ragdoll out the windshield, and the car will flip over. Without the proper hitboxes, this type of interaction would not be possible, including the animation of a car's metal becoming bent.

However, achieving efficient and accurate collision detection in real-time is a non-trivial task due to the complexity and diversity of game environments. There are types of collisions where nothing happens; it is just used as a blocker, as if someone is trying to walk into a wall. Developers face challenges such as limited computational resources, the need for rapid calculations, and the presence of numerous interacting objects. As a result, a wide variety of collision detection algorithms have been developed, each tailored to specific types of objects, performance requirements, and accuracy constraints.

These algorithms can be broadly categorized into two main approaches: spatial partitioning-based methods and geometric-based methods. Spatial partitioning-based algorithms divide the game space into smaller regions, such as grids or trees, to reduce the number of potential collision tests. In these types of partitioning, the location of objects or places you want to save in the data structure is saved by the position. This data structure makes it simple to search for objects nearby or at a certain place quickly. Update the spatial data structure so that it can continue to find the object when its location changes[SPA]. This type of algorithm is usually used in live game environments to store static data, moving data, and just general world data.

Geometric-based algorithms, on the other hand, rely on mathematical equations and geometric properties to determine if and when objects intersect. Using vectors, this type of algorithm can determine the distance and position of objects and if they are getting closer or farther apart. This works better with one's graphics card in the hardware as it reveals more information, making it faster for objects to render.[DVG] Its explained as the primitive version of animation; sometimes, when one's graphic card cannot fully render an object or item, one sees things as geometric items.

Over the years, collision detection algorithms have evolved and improved, driven by advancements in computational power and algorithms themselves. Game developers continuously strive to strike a balance between performance and accuracy. These algorithms enable realistic and interactive experiences by accurately detecting and responding to object interactions within the game space. Collision detection algorithms play a fundamental role in shaping the outcome of these interactions. By implementing collision detection, developers can create engaging gameplay mechanics and bring their virtual worlds to life. However, achieving efficient and accurate collision detection in real-time poses challenges due to the complexity and diversity of game environments. Limited computational resources, the need for rapid calculations, and numerous interacting objects necessitate the development of specialized algorithms. These algorithms can be broadly categorized into spatial partitioning-based methods and geometric-based methods, each tailored to specific requirements and constraints. In the pursuit of creating captivating and realistic gaming experiences, collision detection algorithms remain a vital area of research and development.

3.4 Skill Ranking Algorithms

In the world of competitive video gaming, skill ranking algorithms play a crucial role in determining the proficiency and competitive standing of players. These algorithms are designed to assess the skill levels of individual players or teams and assign them a rank or rating that reflects their relative abilities within the game and is different for each game. There are usually at least five ranks with three different subranks when it comes to competitive play, usually named after medal materials such as bronze, silver, and gold. Ranked play gives players an incentive to become better players as they get better at the game in order to get a higher rank.

Skill ranking algorithms serve several vital purposes. Firstly, they facilitate fair matchmaking by pairing players or teams of similar skill levels together, ensuring that matches are challenging and enjoyable for all participants. This prevents highly skilled players from dominating inexperienced opponents and provides a balanced and competitive gaming experience which is a huge issue in current competitive games right now. These algorithms help establish a sense of progression and accomplishment for players.[**?**] As they improve their skills and climb the ranks, players can track their advancement and strive to reach higher levels of play. This motivates players to continue investing time and effort into the game, fostering a competitive environment and enhancing player engagement.

Various types of skill ranking algorithms are employed in different video games, each with its unique approach and criteria for evaluating player skill. Some algorithms consider individual performance metrics such as win-loss ratios, kill-death ratios, or objective-based achievements, while others take into account team performance or incorporate more complex statistical models. The development of skill ranking algorithms is a multidisciplinary endeavor, drawing on concepts from mathematics, statistics, and artificial intelligence.[MGA] Game developers continually refine these algorithms through iterative processes, using player feedback and data analysis to ensure accurate and reliable skill assessments as they are basically determining if someone is good at their game compared to others. But still wanting to create a type of competitiveness where a win is hard-fought and not a complete roll. This makes it easier for the determined teams to focus on teamwork and less on game fundamentals.

Despite their importance, skill ranking algorithms are not without challenges and limitations. In some cases, these algorithms may struggle to accurately measure skill due to factors like smurfing (highly ranked players creating new accounts to compete against lower-skilled opponents) or boosting (manipulating matchmaking systems to artificially inflate rankings, such as someone using a Smurf account to get their friend to higher rank). Balancing fairness, accuracy, and responsiveness to skill improvements remains an ongoing pursuit for game developers, as there are so many ways to play the game. [CSA]

Overall, skill ranking algorithms are a fundamental component of competitive video games, providing fair matchmaking, fostering player progression, and enhancing the overall gaming experience. As the gaming landscape continues to evolve, developers will undoubtedly explore new approaches and technologies to refine and improve these algorithms, ultimately shaping the future of competitive gaming. Especially in the scene of Esports and the rise of funding and popularity, the pressure put on improving these rankings will continue to increase.

3.5 Anti-Cheat

Cheating in video games has been a persistent problem since the start of online gaming, undermining the competitive integrity of multiplayer experiences. As games become increasingly sophisticated, so do the methods of cheating, ranging from aim botting (automatic aiming to hit enemy players) and wall hacking (seeing players through walls) to exploiting game mechanics not meant to be used. To counteract cheating, anti-cheat software has become a crucial component of modern games, utilizing various algorithms to detect and prevent cheating and ensure fair gameplay for all in ranked and unranked game modes.

It is challenging to find information on these types of algorithms as they are trade secrets in order to keep up with cheat software respectively. One of the most well-known anti-cheat software that paved the way for the current wave of software is called Valve Anti-cheat. Valve Anti-Cheat (VAC) is an anti-cheat system developed by Valve Corporation, the company that created popular gaming platforms such as Steam and games like Counter-Strike: Global Offensive and Dota 2. VAC is designed to detect and prevent cheating in online multiplayer games hosted on their own Steam platform.

VAC operates by employing a combination of client and server-side components to monitor player behavior and detect unauthorized modifications or cheat programs. On the client side, VAC scans the user's computer, looking for known cheat signatures or suspicious activity that may indicate cheating. This includes detecting the presence of aimbots, wallhacks, or other software designed to provide unfair advantages in not just ranked but unranked games too. Things running in the background that could show an overlay on the game with information are not permitted. In addition to client-side scanning, VAC also utilizes server-side components to cross-reference and analyze player data. This includes monitoring player movement, weapon accuracy, and other gameplay metrics to identify discrepancies or abnormal patterns that may suggest cheating.

Once VAC detects a potential cheater, it employs a system of delayed bans, known as "VAC waves." Instead of banning cheaters immediately upon detection, Valve collects data on the cheating methods used and bans a large number of cheaters simultaneously, usually during periodic ban waves. This approach helps avoid revealing the specific cheat detection methods employed by VAC, making it more difficult for cheat developers to evade detection.VAC is designed to be an ongoing and evolving system, continuously updated by Valve to adapt to new cheating techniques and exploits. Valve regularly releases updates to improve the effectiveness of VAC, addressing new cheats and enhancing its detection capabilities. One drawback of this type of banning is that it would not ban cheating happening live in games meaning that the cheater and the rest of the players in that lobby will have to deal with the unbalanced play.

While VAC has successfully combatted cheating in many games, it is important to note that no anti-cheat system is perfect, and there are instances where cheaters find ways to evade detection. However, VAC remains a prominent and widely used anti-cheat system in the gaming industry, helping to maintain fair gameplay environments and protect the integrity of multiplayer games on the Steam platform. Many other games' anti-cheat software is modeled off of VAC and brought to light the importance of anti-cheat software.

3.6 Valorant

To bring it all together, I will be using an example game to show all of these integrations of algorithms into a live and top-rated video game. Valorant, developed by Riot Games, is a popular competitive first-person shooter game that combines tactical gameplay and team-based strategy elements. While specific details about the algorithms used in Valorant are not publicly available, we can discuss some standard algorithms that are commonly employed in similar types of video games to provide insight into the potential algorithms utilized by Riot Games in Valorant.

Something that sets Valorant apart from other online games is the use of their in-house developed anti-cheat system.[VAN] This is used in both their games, Valorant and League of Legends, it was created with the purpose of e-sports in mind. Recently last year, in an official VCT esports match, their software tagged one of the players and found that there was a player using wall cheats to find the position of the enemy team being able to kill them by shooting through the wall. What is really interesting about Vanguard is that this anti-cheat is running when Windows is booted, meaning that it is super intrusive and scanning software that is opened even when Riot's games are not running.

Vanguard works as a kernel-mode driver. A kernel mode driver is a software component that operates at the kernel level of an operating system. It provides direct access and control over hardware devices and low-level operations, making it an integral part of the operating system's core functionality. Due to their privileged access, they require careful development and deployment to ensure compatibility, stability, and security.

In Valorant, when considering matchmaking in the competitive setting, in each game, you gain a certain amount of a variable called RR. In each rank, it takes 100 RR to reach the next. To calculate the amount of RR you obtain or lose at the end of a game, they take into account many variables, including first kills, weapon accuracy, and economical usage. It also takes in to account ones ranked MMR versus ones actual rank.

The game incorporates various algorithms to enhance player experience and ensure fair gameplay. We discussed the matchmaking algorithm, which aims to create balanced teams based on skill levels, resulting in engaging and competitive matches. Additionally, the anti-cheat algorithm plays a vital role in detecting and preventing cheating behavior, safeguarding the integrity of the game. Finally, the pathfinding algorithm helps optimize player movement within the game environment, ensuring smooth navigation and strategic decision-making. These algorithms collectively contribute to the immersive and enjoyable gameplay experience that Valorant offers to its players. Further research and algorithmic design and implementation advancements are likely to continue improving the game's overall performance and user satisfaction.

3.7 Conclusions

In conclusion, algorithms play a pivotal role in shaping the landscape of video games, particularly in the realm of competitive gaming. Skill ranking algorithms enable fair matchmaking, promote player progression, and enhance the overall gaming experience. However, it is essential to recognize that this field is ever-evolving, driven by advancements in technology and changing player preferences. From machine learning and artificial intelligence to real-time data analysis, developers have access to an array of tools and techniques that can revolutionize how games are played and experienced. These advancements can lead to more sophisticated and accurate skill ranking algorithms, creating even more engaging and balanced gameplay environments.

The ways in which players engage with video games are constantly evolving. The rise of eSports, streaming platforms, and mobile gaming has introduced new avenues for competition and spectatorship. This requires developers to adapt their algorithms to cater to these changing and demanding dynamics. Additionally, emerging technologies such as virtual reality (VR) and augmented reality (AR) have the potential to reshape the gaming landscape, necessitating innovative approaches to immersion and game mechanics.

As we look to the future, it is clear that algorithms will continue to be at the forefront of video game development. Game developers will need to stay attuned to the latest technological advancements, player behaviors, and community feedback to refine and improve their algorithms, especially in the area of anti-cheat. Striking a balance between fairness, accuracy, and adaptability will remain a constant challenge but one that is crucial for players to continue to have fun while playing games. By harnessing the power of algorithms and embracing the changing landscape of technology, game developers can continue to create engaging and memorable gaming experiences for players around the globe.

It was fascinating writing this paper as I am an avid competitive video game player. So much goes into creating games, lore, characters, maps, and so much more. Video games have captured my heart and ignited a passion within me. The immersive worlds, captivating narratives, and interactive gameplay offer a unique form of entertainment that engrosses and excites me. The joy and excitement I experience playing video games have fueled my desire to contribute to this industry and create my own immersive gaming experiences. With unwavering enthusiasm, I am driven to pursue a career where I can channel my creativity and love for video games into bringing joy and excitement to others through the power of game development.

4 Conclusions

The algorithm analysis course plays a crucial role in the world of software engineering by providing a solid foundation for understanding the efficiency and performance of all algorithms. In the real world, Software Engineers use algorithms every day and have to make decisions on how and why they want their software

to have different capabilities. One of the most exciting aspects of the course is its emphasis on algorithmic thinking and problem-solving strategies. It taught us to break down complex problems into smaller, more manageable parts and devise efficient algorithms to solve them. This mindset is critical for software engineers as it helps them approach diverse challenges systematically and logically.

Another helpful aspect of the course is its focus on algorithm analysis techniques such as time complexity and space complexity. Understanding these concepts allows software engineers to evaluate the efficiency of different algorithms and make informed decisions about which approach to use in a given context. This knowledge is crucial for building software that can efficiently handle large-scale data.

There are a few changes I would make to the course that might increase student engagement of the course. Firstly, incorporating more real-world examples and case studies would help students understand the practical applications of algorithm analysis. Students can better grasp their significance in the software engineering field by relating the concepts to real scenarios. Including more hands-on programming assignments and projects would enable students to apply the theoretical concepts that we learned in class to be able to be applied to real-life situations, not just research. Building and optimizing algorithms in a real coding environment would help deepen our understanding of these types of principles.

Having more discussions on algorithmic trade-offs and considering real-world constraints would add a realistic perspective to the course. I liked the discussion posts, I just wish that sometimes the questions could be more real-world examples. Software engineers often face situations where there is a trade-off between algorithm efficiency, development time, and resource usage. Exploring these trade-offs would better prepare students for the challenges they will encounter in their careers. Algorithm Analysis as a course is vital to computer science education. It equips students with essential problem-solving skills, an understanding of algorithm efficiency, and the ability to analyze and optimize algorithms. By incorporating more real-world examples and discussions on algorithmic trade-offs, the course can be further enhanced to bridge the gap between theory and practice, ultimately better-preparing students for future roles as software engineers.

References

[ALG] Algorithm Analysis, Chapman University, 2023.

[VAN] Valorant, S, (2022, June 22). What is Vanguard?

[SPA] Spatial Partition, Nystrom, Robert. “Spatial Partition.” Spatial Partition · Optimization Patterns · Game Programming Patterns, 2021

[AOA] Application of Artificial Intelligence to the Development of Playing Ability in the Valorant Game, Fernanda, A., A. R. . Fadri Geovanni, and M. . Huda. “Application of Artificial Intelligence to the Development of Playing Ability in the Valorant Game”. IAIC Transactions on Sustainable Digital Innovation (ITSDI), vol. 4, no. 1, Sept. 2022, pp. 22-31, doi:10.34306/itsdi.v4i1.566.

[MGA] Matching Games and Algorithms for General Video Game Playing, Bontrager, P., A. Khalifa, A. Mendes, and J. Togelius. “Matching Games and Algorithms for General Video Game Playing”. Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, vol. 12, no. 1, June 2021, pp. 122-8, doi:10.1609/aiide.v12i1.12884.

[SRA] A Systematic Review and Analysis of Intelligence-Based Pathfinding Algorithms in the Field of Video Games., Lawande, Sharmad Rajnish, et al. “A Systematic Review and Analysis of Intelligence-Based Pathfinding Algorithms in the Field of Video Games.” Applied Sciences, vol. 12, no. 11, May 2022, p. 5499. Crossref, https://doi.org/10.3390/app12115499.

[DVG] Detecting video-game injectors exchanged in game cheating communities., Karkallis, Panicos, et al. ”Detecting video-game injectors exchanged in game cheating communities.” Computer Security–ESORICS 2021: 26th European Symposium on Research in Computer Security, Darmstadt, Germany, October 4–8, 2021, Proceedings, Part I 26. Springer International Publishing, 2021.

[CSA] Comparative Study of Anti-cheat Methods in Video Games., Lehtonen, Samuli Johannes. "Comparative Study of Anti-cheat Methods in Video Games." (2020).

[AAA] A* Algorithm,

[PTP] P2P matchmaking solution for online games.Boroń, Michał, Jerzy Brzeziński, and Anna Kobusińska. "P2P matchmaking solution for online games." Peer-to-peer networking and applications 13 (2020): 137-150.