**Domain:** Games

**Problem:** Optimizing pathfinding for non-player characters (NPC) to find player character in a grid based RPG style game.

**Goal:** To identify an optimal AI pathfinding technique to implement into an NPC for tracking the player character in a grid-based RPG. The objective is to achieve the shortest distance traveled, highest efficiency, and minimal traversal cost, accounting for various terrains with distinct difficulty levels or weights.

**Introduction**

All good grid-based RPG games employ specific pathfinding algorithms for their NPCs, to make the NPC movements look planned and deliberate giving a sense of realism. Pathfinding, also known as path planning, has many different approaches on how it can be tackled and used. For the domain of games, the author chooses to explore 3 of the most iconic pathfinding techniques and decide on the most optimal one as a solution for his grid based RPG game. The 3 Pathfinding techniques are: Depth-First Search (DFS), Dijkstra's Algorithm and A\* (A-star) Algorithm. Additionally, for a more comprehensive exploration of AI pathfinding, the author will discuss another commonly used AI technique in level design, touching upon

terrain generation and incorporating the use of Perlin noise.

**Greedy Best-First Search**

Greedy Best-First Search is a heuristic-based algorithm designed to efficiently navigate through complex environments. Unlike traditional search algorithms, Greedy Best-First Search prioritizes exploration based on heuristic information, emphasizing speed over optimality. This algorithm is particularly well-suited for scenarios where quickly finding an approximate solution is more crucial than ensuring the absolute shortest path.

Strengths and Weaknesses:

Greedy Best-First Search stands out for its simplicity and rapid decision-making. By relying on heuristic estimates, the algorithm swiftly explores the search space, making it suitable for real-time applications in game development where quick responses are vital. The simplicity of implementation is another notable strength, providing an accessible solution for various game environments.

The primary weakness of Greedy Best-First Search lies in its lack of optimality. Decisions are made solely based on heuristic estimates without considering the actual cost of reaching the current state. This can lead to suboptimal paths, particularly when the heuristic doesn't accurately reflect the true cost. Additionally, the algorithm may struggle in scenarios with backtracking, potentially getting stuck in local minima.

Advantages:

Speed is the primary advantage of Greedy Best-First Search. Its ability to quickly navigate through the search space makes it well-suited for scenarios where finding an approximate solution promptly is more critical than ensuring optimality. The simplicity of the algorithm contributes to its versatility, allowing it to be easily adapted to various game environments.

Disadvantages:

The lack of optimality is a significant drawback, especially in games where finding the shortest or most efficient path is crucial. Greedy Best-First Search might overlook paths that would lead to a better solution in the long run. The reliance on heuristic information can also introduce bias in decision-making, favoring paths that initially seem promising based on estimates.

Examples of Application:

Greedy Best-First Search finds application in scenarios where a quick but approximate solution is acceptable. In game development, it could be employed in real-time strategy games where units need to navigate through terrain swiftly, and an optimal path is not the primary concern.

Input Data and Expected Output:

Heuristic information is the key input for Greedy Best-First Search, guiding the exploration of the search space. In game development, this data could represent heuristic estimates of distances or costs between different locations. The expected output is a path to the goal, emphasizing speed over optimality.

In summary, Greedy Best-First Search offers a swift and simple solution for scenarios in game development where quickly finding an approximate solution is prioritized. However, its lack of optimality may limit its applicability in situations where ensuring the absolute shortest path is crucial. Now, let's transition to the exploration of Dijkstra's Algorithm, providing a comprehensive analysis to aid in choosing the most suitable pathfinding algorithm for game development.

**Depth-First Search (DFS)**

Depth-First Search (DFS) is a fundamental algorithm used in pathfinding within the domain of games. This algorithm explores a graph or tree by prioritizing deep exploration over breadth. Let's delve into the strengths, weaknesses, advantages, and disadvantages of DFS in the context of pathfinding for games.

**Strengths and Weaknesses of DFS**

DFS is known for its simplicity and ease of implementation. It is a versatile algorithm applicable to various types of graphs, making it suitable for different game environments. The memory requirements of DFS are minimal, as it only needs to store information about the current path being explored. In game scenarios, where computational resources are crucial, this efficiency is a significant strength.

One prominent weakness of DFS is its lack of optimality. Since it doesn't consider the weights of the edges, it might not always find the shortest path. In game development, finding the optimal path is often crucial for efficient navigation, and DFS may fall short in this regard. Additionally, DFS may get stuck in infinite loops if the graph contains cycles, posing challenges for real-time applications like games.

**Advantages and Disadvantages of DFS**

DFS is well-suited for scenarios where a solution path is likely to be deep within the graph. In game maps where long paths need to be traversed, DFS might outperform other algorithms. Its simplicity also makes it a good choice for quick prototyping and initial testing phases in game development.

The lack of optimality is a significant drawback in scenarios where finding the shortest path is essential. In games, especially those requiring precise navigation, suboptimal paths could lead to inefficient AI behavior. Additionally, DFS might not be suitable for dynamic environments where paths change frequently, as it does not adapt well to alterations in the graph.

**Examples of Application:** In games, DFS can be applied to character navigation in large, open-world maps. For instance, in an adventure game, DFS could be used by non-player characters (NPCs) to explore and find hidden locations. However, it might not be the best choice for scenarios where NPCs need to find the shortest path to a destination due to its non-optimality.

**Input Data and Expected Output:** DFS typically requires information about the adjacency of nodes in the graph. In the context of games, this translates to data about the connections between different locations or tiles. The expected output is the path taken, which might not always be the shortest but is the first path discovered by the algorithm.

In short, Depth-First Search is a simple and resource-efficient algorithm suitable for certain game scenarios. However, its lack of optimality and susceptibility to infinite loops limit its applicability in games where finding the shortest path is crucial. While it can be implemented for the authors RPG game, it is not ideal! so the next section will explore the Dijkstra's Algorithm.

**Dijkstra's Algorithm**

The Dijkstra's Algorithm is another widely used pathfinding algorithm with distinct strengths and weaknesses in the context of game development. Let's delve into its characteristics, advantages, disadvantages, and applications.

**Strengths and Weaknesses of Dijkstra's Algorithm**

One of the primary strengths of Dijkstra's Algorithm is its ability to find the shortest path in a weighted graph. This makes it highly suitable for scenarios in games where the distance between locations matters, ensuring optimal pathfinding. Dijkstra's Algorithm guarantees the optimality of the solution, providing confidence in the accuracy of the results.

While Dijkstra's Algorithm excels in finding the shortest path, it can be computationally expensive in terms of time and memory, especially in large graphs. This algorithm explores all possibilities without any heuristic guidance, which can be inefficient for certain game environments. In scenarios where computational resources are limited, Dijkstra's Algorithm might not be the most efficient choice.

**Advantages and Disadvantages of Dijkstra's Algorithm**

The primary advantage of Dijkstra's Algorithm lies in its optimality – it guarantees the shortest path. This is particularly crucial in games where precise navigation is essential. Additionally, Dijkstra's Algorithm is flexible and can be adapted for use in various game environments with weighted graphs.

One notable disadvantage is the computational cost associated with exploring all possible paths. In real-time games where quick decision-making is required, the time complexity of Dijkstra's Algorithm might be a drawback. Additionally, the algorithm doesn't consider any heuristic information, potentially leading to suboptimal performance in certain scenarios.

**Examples of Application:** In games, Dijkstra's Algorithm can be applied to scenarios where finding the optimal path is crucial, such as guiding characters through a maze or determining the best route for a moving object. For instance, in a strategy game, Dijkstra's Algorithm could be employed to calculate the most efficient path for units to reach a destination, taking into account terrain costs.

**Input Data and Expected Output:** Dijkstra's Algorithm requires information about the weighted connections between nodes in the graph, indicating the cost of moving from one location to another. In the context of games, this could represent the distance or difficulty of traversal between different game tiles. The expected output is the shortest path from the source to the destination.

In short, the Dijkstra's Algorithm is quite powerful for finding the shortest path in game scenarios that involve weighted graphs. Its optimality is a significant advantage, but the computational cost may limit its applicability in real-time, resource-constrained environments. While this algorithm can be implemented and sounds perfect for an RPG game the author feels there could be an even better pathfinding technique, so the next section will explore the A\* algorithm.

**A\* (A-star) Algorithm**

A\* (A-star) algorithm is a widely used and highly efficient pathfinding algorithm that combines the benefits of both Depth-First Search (DFS) and Dijkstra's Algorithm while mitigating their drawbacks. Let's explore the strengths, weaknesses, advantages, and disadvantages of A\* in the context of game development for the authors RPG game.

**Strengths and Weaknesses of A\* Algorithm**

A\* is known for its optimality, providing the shortest path in weighted graphs. This algorithm introduces a heuristic component, making it more efficient than Dijkstra's Algorithm by guiding the search towards the most promising paths. A\* strikes a balance between completeness and optimality, making it suitable for various game environments. Its ability to adapt to different heuristic functions contributes to its flexibility.

While A\* is a powerful algorithm, its main weakness lies in its potential computational complexity. The efficiency of A\* heavily depends on the quality of the heuristic function used. In scenarios where a poorly chosen heuristic leads to overestimation or underestimation of costs, A\* might not perform optimally. Additionally, A\* may struggle in scenarios with dynamic environments, as it doesn't adapt well to changes in the graph.

**Advantages of using A\* Algorithm**

One significant advantage of A\* is its ability to find the optimal path efficiently. By incorporating a heuristic function like the Manhattan distance, A\* narrows down the search space, making it more suitable for real-time applications in games. It also outperforms both DFS and Dijkstra's Algorithm in terms of computational efficiency, especially in scenarios where finding the shortest path is critical.

**Examples of Application:** In games, A\* is widely used for character navigation, pathfinding for moving objects, and determining optimal routes for game entities. For example, in a real-time strategy game, A\* could be employed to guide units through changing terrain efficiently, adapting to obstacles and avoiding unnecessary detours.

**Input Data and Expected Output:** Similar to Dijkstra's Algorithm, A\* requires information about the weighted connections between nodes in the graph, along with a well-designed heuristic function. In game development, this data could represent the distances or costs associated with moving between different locations. The expected output is the optimal path from the source to the destination.

**Manhattan Distance as a Heuristic:** One commonly used heuristic in A\* is the Manhattan distance, especially in grid-based games. This heuristic calculates the distance between two points on a grid by summing the absolute differences of their coordinates. Using the Manhattan distance as a heuristic in A\* is advantageous in scenarios where movement is constrained to grid positions, providing a good balance between accuracy and computational efficiency.

*Proving A Superiority:*

After comparing and analyzing the 3 pathfinding techniques the author feels A\* can be considered superior to DFS and Dijkstra's Algorithm in scenarios where optimality and computational efficiency are crucial. By incorporating a heuristic like the Manhattan distance, A\* significantly reduces the search space, focusing on the most promising paths. In comparison to DFS, A\* guarantees optimality, and compared to Dijkstra's Algorithm, it often achieves the same optimality with lower computational cost.

In conclusion, A\* stands out as a powerful and widely applicable pathfinding algorithm for the authors RPG game. Its ability to find optimal paths efficiently, guided by a well-designed heuristic like the Manhattan distance, makes it the perfect choice for the player tracking NPC in a grid-based RPG.

**Terrain Generation Using Perlin Noise**

Terrain generation using Perlin noise is a widely employed technique in game development to create realistic and visually appealing landscapes. Let's explore the strengths, weaknesses, advantages, and disadvantages of using Perlin noise for terrain generation.

**Strengths and Weaknesses of Perlin noise for Terrain Generation**

Perlin noise is renowned for its ability to generate smooth, continuous, and natural-looking patterns. This quality makes it particularly well-suited for terrain generation, as it mimics the organic variations found in natural landscapes. The algorithm is relatively simple to implement, providing a good balance between computational efficiency and visual quality.

While Perlin noise excels in creating organic patterns, its weakness lies in the potential for uniformity (this can be observed during Part C Implementation section). Without additional modifications or layers, Perlin noise might generate terrains that appear too smooth or lack the desired level of detail. To address this, developers often combine multiple layers of Perlin noise or apply additional algorithms for more intricate terrains.

**Advantages and Disadvantages of Perlin noise for Terrain Generation**

One of the primary advantages of Perlin noise is its versatility. It can be adjusted to generate a variety of terrains, from rolling hills to rugged mountains. The simplicity of the algorithm makes it accessible for developers at various skill levels. Additionally, Perlin noise provides a level of randomness that contributes to the uniqueness of each generated terrain, enhancing the overall gaming experience.

The main disadvantage is the potential for monotony without proper adjustments. While Perlin noise offers a good starting point, it might not capture all the nuances of complex natural landscapes. Developers often need to complement Perlin noise with additional algorithms or tweak parameters to achieve the desired level of realism and variety in terrain generation.

**Examples of Application:** Perlin noise is commonly used in open-world games to generate realistic landscapes. For instance, in a role-playing game (RPG), Perlin noise could be employed to create diverse terrains, including plains, forests, and mountains. The algorithm's adaptability allows developers to tailor generated terrains to suit the thematic requirements of their games.

**Input Data and Expected Output:** Perlin noise typically requires input parameters such as scale, octaves, and persistence to control the characteristics of the generated terrain. The expected output is a heightmap representing the elevations of the terrain, which can then be translated into 3D landscapes or 2D maps for game environments.

In summary, Perlin noise is a valuable tool for terrain generation in game development due to its simplicity, versatility, and ability to produce visually appealing landscapes. While it may have limitations in generating highly detailed terrains, developers can overcome these challenges by combining Perlin noise with other techniques or tweaking parameters to achieve the desired level of complexity and realism.

The Author chose to include terrain generation using Perlin noise in this section to further enhance the profoundness of pathfinding techniques. The next section will showcase different implementations of A\* algorithm for various situations and also how it stays optimal even in a procedurally generated terrain.