Course: KOU001EEC BSc (Hons) Informatics

Module Code: KOL304CR

Module Title: Games and AI

Module Leader: Lena Erbs

Hand out date: February 2025

Submitted by: PAUL MICKY D COSTA

Submission deadline: 26/03/2025

Portfolio: AI Techniques in Real-Time Games

Table of Contents

1. A* Algorithm

Description

A* (A-star) is one of the most widely used pathfinding algorithms in games. It is designed to efficiently find the optimal path from a starting position to a goal while considering cost efficiency. A* builds upon Dijkstra’s algorithm by incorporating a heuristic function, which enables it to prioritize more promising paths rather than evaluating every possible route. This makes it particularly useful for real-time applications where computational efficiency is crucial.

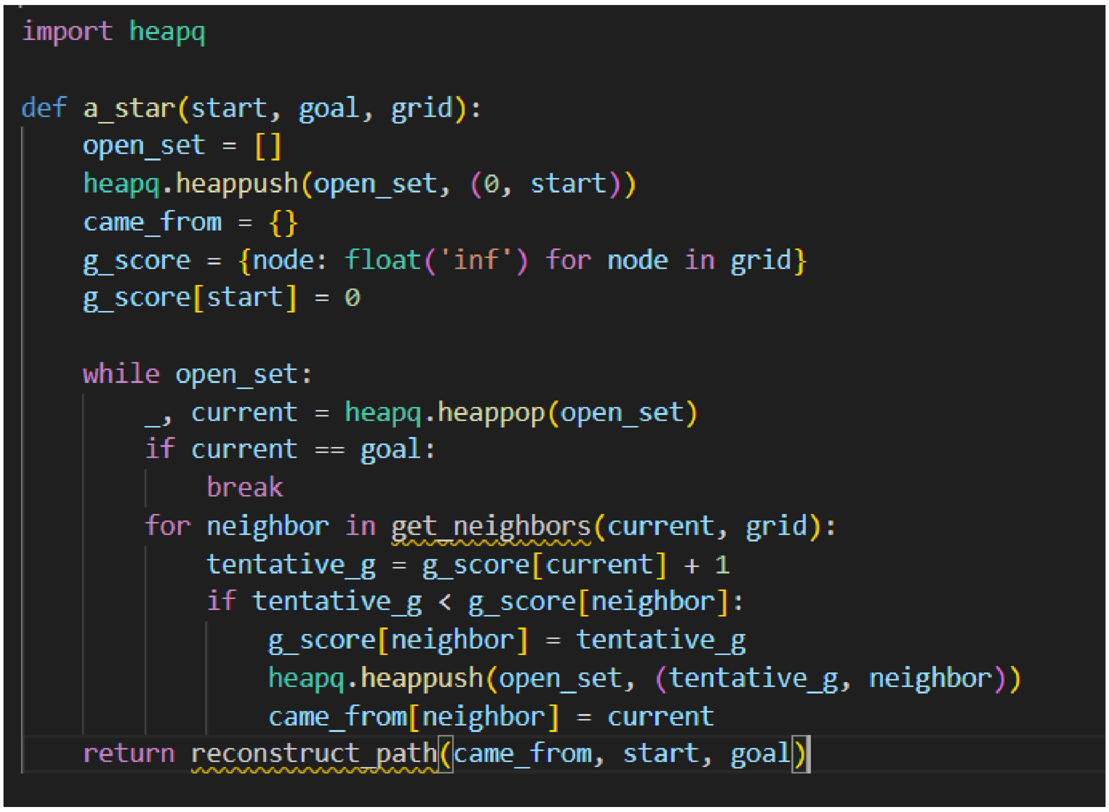
A* is fundamental in game AI for guiding NPC movement in open-world and strategy games. It enables characters to navigate obstacles dynamically while ensuring the shortest possible route is taken. Many modern games, such as strategy and role-playing games, use A* for character movement, enemy chasing, and other real-time decision-making processes.

Implementation

To implement A*, I used Python with a grid-based representation of the game environment. The algorithm works as follows:

- Define a grid where each node represents a traversable space within the game world.
- Assign three key values to each node:
  - g: The cost from the start node to the current node.
  - h: The heuristic estimate of the cost from the current node to the goal.
  - f: The total estimated cost (f = g + h).
- Utilize a priority queue to always expand the node with the lowest f-value first.
- Continue the process until the goal is reached, at which point the shortest path can be reconstructed.

Code Snippet:



```python
import heapq

def a_star(start, goal, grid):
    open_set = []
    heapq.heappush(open_set, (0, start))
    came_from = {}
    g_score = {node: float('inf') for node in grid}
    g_score[start] = 0

    while open_set:
        _, current = heapq.heappop(open_set)
        if current == goal:
            break
        for neighbor in get_neighbors(current, grid):
            tentative_g = g_score[current] + 1
            if tentative_g < g_score[neighbor]:
                g_score[neighbor] = tentative_g
                heapq.heappush(open_set, (tentative_g, neighbor))
                came_from[neighbor] = current
    return reconstruct_path(came_from, start, goal)
```

Reflection

The most significant challenge in implementing A* was choosing the right heuristic function. A good heuristic dramatically improves efficiency, while a poor one can make the algorithm perform worse than Dijkstra's. I experimented with several heuristics, including Manhattan distance, Euclidean distance, and diagonal distance.

Manhattan distance performed well in grid-based maps without diagonal movement, while Euclidean distance was more suited to maps allowing diagonal traversal. Overestimating heuristics led to suboptimal paths, whereas underestimating heuristics increased computation time.

A* excels in dynamic game environments where real-time pathfinding is required. However, it can become computationally expensive for large-scale maps, especially in open-world games with complex terrain. Optimizations such as hierarchical pathfinding and jump-point search can help mitigate performance issues.

Ultimately, A* remains one of the most effective algorithms for pathfinding in games due to its balance between accuracy and efficiency.

2. Dijkstra's Algorithm

Description

Dijkstra's algorithm is a fundamental pathfinding technique used in games and computational graph theory. It finds the shortest path between nodes in a weighted graph by evaluating all possible routes systematically. Unlike A*, Dijkstra's algorithm does not rely on heuristics, making it an exhaustive but slower approach, particularly in large-scale environments.

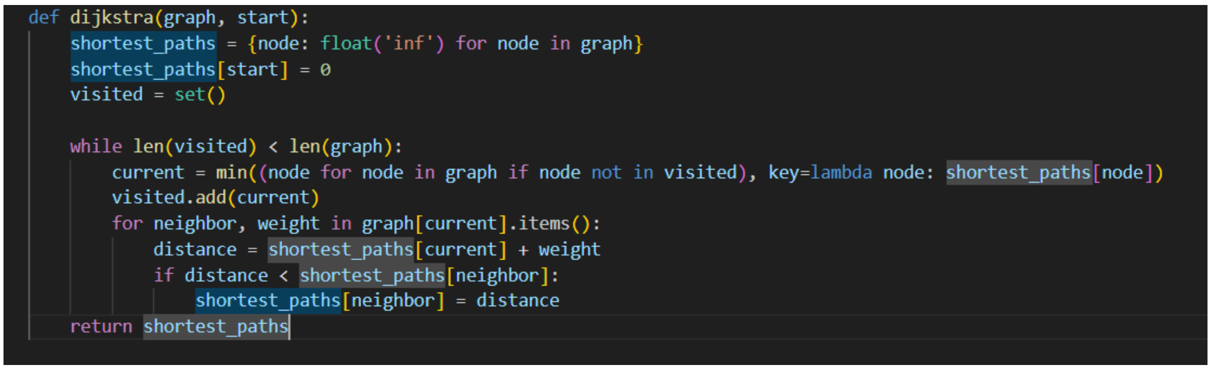
In gaming, Dijkstra's algorithm is commonly used in turn-based strategy games, where finding the exact shortest path is crucial for unit movement and tactical positioning. It is also employed in game maps where precomputed navigation graphs optimize real-time decision-making.

Implementation

To implement Dijkstra's algorithm, a priority queue is used to process nodes in ascending order of distance from the starting position. The key steps include:

- Assigning an initial distance of zero to the start node and infinity to all others.
- Iteratively selecting the node with the smallest known distance and updating neighboring nodes accordingly.
- Marking nodes as visited once their shortest path is determined.
- Continuing until all reachable nodes have been processed.

Code Snippet:



```
def dijkstra(graph, start):
    shortest_paths = {node: float('inf') for node in graph}
    shortest_paths[start] = 0
    visited = set()

    while len(visited) < len(graph):
        current = min((node for node in graph if node not in visited), key=lambda node: shortest_paths[node])
        visited.add(current)
        for neighbor, weight in graph[current].items():
            distance = shortest_paths[current] + weight
            if distance < shortest_paths[neighbor]:
                shortest_paths[neighbor] = distance
    return shortest_paths
```

Reflection

Dijkstra's algorithm is highly reliable because it guarantees finding the shortest path. However, its biggest drawback is the lack of heuristic optimization, making it computationally expensive in large and complex maps. Compared to A*, which directs its search using heuristics, Dijkstra evaluates all possibilities equally, often leading to unnecessary calculations.

One way to enhance performance is through optimization techniques such as priority queues implemented with binary heaps, which significantly reduce processing time. In grid-based games, precomputing paths using Dijkstra's algorithm allows for fast lookups during real-time gameplay.

Despite its slower nature, Dijkstra's algorithm remains an essential tool in game AI, particularly when accuracy is more important than speed. It is especially useful in environments where paths must be determined ahead of time, such as in tactical role-playing games or navigation systems requiring exhaustive route evaluation.

3. Boids Flocking

Description

Boids flocking is an AI technique used to simulate the natural movement of groups, such as birds, fish, or swarms of insects, in a realistic manner. The behavior is governed by three fundamental rules:

- **Separation**: Avoiding overcrowding by maintaining an optimal distance from nearby boids.
- **Alignment**: Matching the general direction of neighboring boids.
- **Cohesion**: Moving towards the center of the group to maintain unity.

This technique is widely used in real-time strategy and open-world games to create dynamic, lifelike crowd behaviors without requiring direct player control. The balance between these three principles dictates how cohesive or dispersed a flock appears, making parameter tuning crucial for realistic simulation.

Boids are frequently utilized in large-scale simulations where emergent behavior enhances realism, such as simulating enemy patrols, NPC crowds, and organic movement patterns in wildlife-based games. Game developers use boid algorithms to create visually engaging and believable group movement patterns that adapt dynamically to changes in the environment.

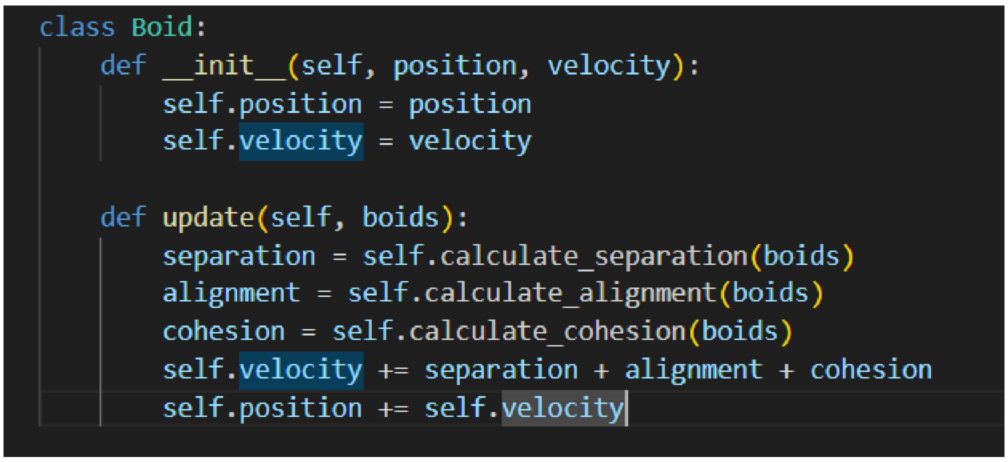
Boids flocking is inspired by real-world flocking behavior, first modeled by Craig Reynolds in 1986. The algorithm allows groups to move cohesively without central control, making it a powerful tool for simulating natural behaviors in gaming environments.

Implementation

The implementation of boids flocking involves iterating through each boid in a system and adjusting its velocity based on nearby boids. Each boid follows three primary rules that dictate its movement:

- **Separation:** Each boid calculates the distance to its neighbors and moves away if they are too close, preventing overcrowding.
- **Alignment:** Boids adjust their velocity to match the average direction of their neighbors, ensuring a synchronized movement pattern.
- **Cohesion:** Each boid moves towards the center of mass of nearby boids, promoting group unity.

Code Snippet:



```python
class Boid:
    def __init__(self, position, velocity):
        self.position = position
        self.velocity = velocity

    def update(self, boids):
        separation = self.calculate_separation(boids)
        alignment = self.calculate_alignment(boids)
        cohesion = self.calculate_cohesion(boids)
        self.velocity += separation + alignment + cohesion
        self.position += self.velocity
```

Additional optimizations, such as spatial partitioning, help reduce computational overhead by limiting the number of boids each agent must consider at every frame.

Without optimization, each boid would check all others in the simulation, leading to poor performance in large-scale environments.

Reflection

One of the major challenges in implementing boids flocking is fine-tuning the weighting of separation, alignment, and cohesion forces. Excessive separation results in boids scattering too far apart, while strong cohesion causes them to clump together unnaturally. A well-balanced system produces visually compelling and responsive group behavior that adapts to environmental changes dynamically.

Another challenge is handling obstacles within a game environment. Without additional logic, boids may collide with walls, trees, or other structures, breaking the illusion of intelligence. Implementing obstacle avoidance mechanisms, such as raycasting or steering behaviors, helps enhance the realism of boids navigation.

Performance optimization is another critical consideration. When simulating thousands of boids, computational efficiency becomes a challenge. Techniques such as quadtree partitioning or spatial hashing can significantly improve performance by reducing the number of interactions each boid processes per frame.

The boids model is a great example of emergent behavior, where complex group movement arises from simple rules. It is widely used beyond gaming, including in robotics, crowd simulations, and animations. In game development, boids flocking helps create immersive worlds where NPCs, animals, or enemy units move in a lifelike manner, adding depth and realism to the experience.

Ultimately, boids flocking remains a valuable AI technique in real-time simulations. Properly tuning parameters, optimizing performance, and integrating obstacle avoidance are essential to achieving realistic and engaging group movement in games.

Conclusion

Implementing A*, Dijkstra's algorithm, and Boids flocking provided valuable insights into AI techniques for real-time games. Each method serves a unique role, making it essential to choose the right approach based on game requirements and computational constraints.

A* proved to be a highly efficient and scalable pathfinding algorithm, balancing accuracy and speed by using heuristics. However, its computational cost increases with larger maps, necessitating optimizations for extensive environments. In contrast, Dijkstra's algorithm guarantees the shortest path by systematically evaluating all possible routes, making it ideal for turn-based strategy games. Although slower, it excels in scenarios where precision is more important than speed.

Boids flocking, unlike pathfinding algorithms, focuses on emergent behaviors, enabling lifelike and adaptive group movements. This technique is widely used in simulating crowds, enemy formations, and NPC behavior, adding realism to game environments. Fine-tuning separation, alignment, and cohesion is crucial to maintaining fluid movement, and optimization techniques such as spatial partitioning improve efficiency in large-scale simulations.

Beyond gaming, these AI techniques have broader applications in robotics, simulations, and real-world optimization. A* is commonly used in autonomous navigation, Dijkstra's algorithm aids network routing, and Boids flocking contributes to crowd dynamics and drone swarm coordination. Understanding these methods strengthens AI development skills and enhances problem-solving in various domains.

In conclusion, selecting the appropriate AI technique depends on game complexity, performance constraints, and desired realism. Mastering these algorithms allows developers to create intelligent, responsive, and engaging game experiences. As AI technology advances, integrating machine learning and adaptive AI models will further enhance game behavior, leading to more immersive and dynamic gameplay.