Predictive Dynamic Shortest Path Algorithms

David Buyck

May 2021

Abstract

The game Snake found a large audience when users found it preloaded on Nokia phones in 1998. Many people played Snake to pass the time, and and the more time that passed in the game of Snake, assuming the snake continued to eat objects, the more challenging the game got. This takes the game of snake from the simple game of finding a shortest path between the snake and the food object to a game where the tail creates a dynamic terrain and the food object is continuously placed on a random position. In this paper, I explore various shortest path algorithms (SPA) and how adaptive and predictive utilities can allow for an algorithm to not only successfully complete a game of Snake but also to reach optimal game completion times. The easiest algorithm to use to complete a game of Snake is a Hamiltonian Cycle where the snake follows a path that touches every position on the game grid and returns where it started. This can complete the game of Snake, but the time required to complete the game this way can be immense. I explored ways to speed up the completion time by using one of the most powerful and efficient SPAs, the A* search. Using the Euclidean distance between the snake and the food object as a heuristic, I found great success improving the time it takes for the snake to reach the food object. However, A* search fails when the snakes tail blocks the path to the food object, so I experimented with adding local cycles to the A* search that allow the snake to wait until a path to the food object becomes available. This allowed for the continuation of the game, but it was slow and did not prepare the snake for the next food object's position. The solution I found used a predictive algorithm that adapted to the dynamic terrain created by the snake's tail, which allowed for the snake to find the optimal path to the food object each time while ensuring future paths to food objects were available. In this paper, I will discuss the algorithms I used and created. I will detail how each utility of the algorithms is used to ensure success and increase completion speed. I will then show the test results of these algorithms when they are tested on a 10x10 Snake grid and discuss the test environment and how the numbers can be interpreted. Finally, I will discuss the difficulty with the algorithm and how it can be improved.

1 Introduction

The game of Snake can be reduced to the very common, real world problem of efficiently navigating dynamic terrains. An algorithm that quickly and efficiently finds a solution to dynamic terrain pathfinding can be used in countless real world applications, from a robot used in unmanned rescue to a tugboat in a port [2] [9]. Optimal solutions to pathfinding on dynamic terrains can reduce the time it takes for a rescue robot to reach an injured person who has limited time or engage a mechanism at a nuclear powerplant before a meltdown occurs. These algorithms can be helpful beyond our imagination. Because of this, there are many different algorithms and adaptions to those algorithms that allow for them to be improved and more widely used. Algorithms from A* search, D* search, and Dijkstra's algorithm all excel at pathfinding [10]. The real challenge these algorithms face is the uniqueness of each application as a dynamic terrain can take on many different forms.

It is easy to imagine how SPAs that run on dynamic terrains can differ. A tugboat would need an SPA that searches the two-dimensional port for a path to a goal position across the flat plane of water [2]. There are other SPAs that are designed to handle three-dimensional dynamic terrains like those for rescue robots that may need to navigate cave systems. Both situations involve changing terrains from other ships sailing across the harbor to falling rocks and debris in a tunnel cave-in. An algorithm that cannot handle the changes to its terrain will find a suboptimal path or, in the worst case, it will fail and the agent will become

immobile. This issue is extremely prevalent for traversing the dynamic terrain in the game Snake. Every move the snake makes the terrain change as the snake's tail follows the head. The goal position changes every time the snake reaches the food object. The majority of the time the game is played there is no direct route from the snakes head to the food object which, if not handled, would cause basic SPAs like A*, D*, and Dijkstra's algorithm to fail and the snake to lose the game. There are also challenges where the food can randomly be placed in a position directly in front of the snake in a way where the game is lost instantly if there is not ample room between the snake and its tail. There can be situations where the snake wraps around a single cell and considering the path the tail makes that position can become unreachable for the snake. All these dynamic search hazards must be factored into a dynamic SPA for the search to even be possible and a path must be chosen quickly, as the game of Snake requires the agent to move every second.

To overcome all the hazards of Snake's dynamic terrain, modifications must be made to basic SPAs. The first and most important modification is the algorithm must account for the tail moving every time the snake head moves. This requires an algorithm that uses the deterministic nature of the snake's movement to calculate a projection efficiently and accurately of what the terrain will look like many moves ahead since a bad move can trap the snake or force the snake into its tail or a wall which can force the snake to lose and the game to be unsuccessful. Using a payoff function that allows the agent to look moves ahead can allow the payoff function to determine if that move can create a successful outcome for future move [1]. With the future state of the board known to the agent, an SPA can then calculate the optimal route based on where the tail will be for every move of the snake. After a guaranteed successful and optimal run is achieved, the only thing left for an optimal solution to Snake is optimally anticipating the random placement of the food object.

In this paper, I will be focusing on the application of A* search to a 2-dimensional, 4-neighbor grid. I will discuss the various payoff functions that can be used alongside predictive algorithms that allow the snake to see every position many moves ahead. I will also discuss the use of local cycles that allow the snake to consolidate its tail and allow the snake to buy time for a path to the food object to become available. From the research I have done, I have found no optimal solution to the Snake game or any dynamic terrain problem for that matter. All my algorithms aid the snake in finding and optimal solution to the game Snake, but the randomness of the food objects placement makes these algorithms far from perfect, which is a subject that requires much discussion. I will detail each algorithm and explain how it benefits the agent's navigating as well as discuss its flaws and challenges.

In the following sections I will discuss how other researchers have approached solving the problem of pathfinding on both static and dynamic terrains. Then I will explain what and how I decided to use from their research to develop the dynamic search algorithms I used for the Snake game. Finally, I will show the results from my performance tests, compare the results to an agent following a basic Hamiltonian cycle, and discuss the success and possibilities of improvement of my algorithms.

2 Related Work

The game of Snake is centered around the problem of pathfinding on a two-dimensional grid. Since the dynamic terrain changes with every move of the snake, the SPA must be extremely efficient if it is to calculate a path before the snake moves every second. "A Comparative Study of A-star Algorithms for Search and rescue in Perfect Maze" by Liu and Gong discussed the performance of A* searches relative to non-heuristic searches such as Depth First Search (DFS) [5]. In their paper, they detailed three different A* searches based off different heuristics and directly measured them to each other as well as a DFS on a perfect maze. They tested the three A* searches and DFS on ten perfect mazes and analyzed the results. Their tests showed that A* searches outperform DFS in all but a few instances. They concluded that A* search is a better search for perfect mazes and the most efficient heuristic for navigating a perfect maze is using the Euclidian distance from the agent to the maze exit. So, A* would be an excellent choice for finding a path quickly and the Euclidian distance would be an excellent heuristic since measuring the distance between two points on the snake's grid is very possible.

A* is an efficient SPA that can be used to solve the game of snake, but an optimal path is required to find

an optimal solution. "Finding paths on real road networks: the case for A*" by Zeng and Church showed that A* search was more efficient than other SPAs and, with larger networks, it also outperformed SPAs [10]. SPAs can outperform A*, like the Gallo-Pallpttino (GP) class of algorithms, by adding complexity. However, that complexity makes them far less efficient than A*. Zeng and Church showed that A* can outperform the GP class of algorithms when applied to larger problems involving larger path networks. Efficient algorithms like Dijkstra's algorithm get outperformed by A*s ability to use spatial coordinates to efficiently find the shortest path. This makes A* one of the best algorithms to use when spatial coordinates can be accessed from both an efficiency and performance aspect. This reinforced my belief that A* was the most efficient SPA to use but it also began to convince me that it will also find the optimal path.

The grid the snake uses as its terrain is a uniform cost grid, which can make searches simpler as the edges between vertices are all of equal weight. This allows for several shortcuts to be used to increase A* searches performance. "Efficient Optimal Search of Uniform-Cost Grids and Lattices" by Kuffner proposed a technique for optimal searches on Euclidian cost grids which reduced the runtime relative to A* or Dijkstra's algorithm [4]. The technique that he proposed exploits the know constant edge weights of uniform grids to reduce the number of nodes searched. Specifically, he recognized the triangle inequality can allow the search to visit diagonally adjacent grid points 2 ways at the same cost. Since the cost is always the same for both routes, the search can recognize a diagonally adjacent grid and only take one of the 2 possible direct routes. This technique can reduce the runtime of optimal searches, but he admitted this reduces the runtime by a constant and the runtime still grows exponentially as the size of the grid grows. To supplement the runtime reduction, he suggested running the search on a reduced graph to find a course path and then to use that path on the finer, real graph to reduce the runtime even more. This was helpful as paths that are not strictly in front of the snake or to the side of the snake can utilize this corner cutting technique to save some time each time A* runs.

With the solid foundation of A* search, I can now confidently find an efficient, optimal path when the terrain is static and the goal is reachable. The next question is how to handle a dynamic terrain. "Comparison of A* and Dynamic Pathfinding Algorithm with Dynamic Pathfinding Algorithm for NPC on Car Racing Game" by Sazaki, Satria, and Syahroyni discussed and tested the performance of an NPC race car using A* search, the dynamic pathfinding algorithm (DPA), and a combination of the two to determine the best way to design a race car NPC [6]. The way they implemented the combination of A* and DPA was to first run A* and save the optimal path. Then as the NPC car navigated the path, the DPA was run to detect changes in the dynamic terrain. When a change was detected, like an obstacle in its way, the NPC followed the DPA until the obstacle was cleared. Then the NPC went back and followed the A*. The result of tests in A*, DPA, and a combination of the two showed that a combination of the two gave the NPC better performance both for time of completion and the ability to complete races with obstacles present. This paper used a DPA to supplement their A* search for handing obstacles. Considering that the snake's tail is determined on the movement of the head, I used a DPA to maneuver around the dynamic terrain created by the tail.

Another version of an SPA that accounts for a dynamic terrain is a tactical pathfinding algorithm (TPA). "Route Optimization Movement of Tugboat with A* Tactical Pathfinding in SPIN 3D Simulation" by Anisyah, Rusmin, Hindersah discussed the application of A* search to the problem of finding the most efficient paths for tugboats at a port [2]. They first showed the need for an efficient tugboat navigation algorithm as each port can be very dense with ships, reefs, and other obstacles. Tugboats are constantly needed to move ships and to find the optimal path for a tugboat could a lot of time and fuel. To test the ability of A* on a real port, they mapped a grid to Tanjung Priok port in Indonesia. To compliment A*, they added a TPA to their A* heuristic. Their TPA ran in three stages, got the destination node's coordinates, calculated the shortest route based on port factors and the tugboats' attributes, then navigated the route. The TPA, combined with the Euclidian distance, allowed for their modified A* search to find a path that was 37 percent more efficient than A* alone. This used a heuristic that was well beyond the Euclidian distance as it factored in the movements of other objects in the port. This TPA can be used for Snake as the agent can calculate the position of the tail for each move of the snake as the tail's movement is completely deterministic.

An SPA that is supplemented by a DPA and a TPA can have information that conflicts with each other.

Where an SPA like A* looks for the optimal path, a TPA will look for a future path. An algorithm must be used to decide which path takes precedence, which path should it be the current optimal path, or a path that preserves the future optimal path. This is where a payoff function can help decide which path is most beneficial. "The Robot Control Strategy in a Domain with Dynamic Obstacles" by Alferov and Malafeyev discussed an agent's ability to pursue a moving goal position [1]. In this paper they discussed how each agent can define states in the conflict process that can be used to evaluate the agent's next move where one agent pursues, and the other agent evades. A payoff function was constructed by using strategy pairs of the two agent's trajectories where the evading agent tries to maximize its payoff function and the pursuing agent tries to minimize its payoff function. It mentioned skipping sequential steps of the conflict process can be beneficial to an agent. The paper concluded that if both agents use their payoff functions there would be no optimal strategy as there would be an equilibrium between the results of the two functions. A payoff function can be applied to the SPA, DPA, and TPA to ensure that each path selected has the optimal outcome in mind without sacrificing future moves or missing a current optimal path.

When the payoff function decides that the current SPA's optimal path will cause the snake to become trapped and thus lose the game, many paths become available to waste moves until an optimal path that allows a successful completion of the game exists. This brings up new challenges beyond the SPA, DPA, TPA, and payoff functions. "Summary of Pathfinding in Off-Road Environment" by Kapi, Sunar, and Algfoor discussed the challenges that are presented with pathfinding algorithms used on off-road terrain [3]. They started by describing the many possible uses for off-road terrain pathfinding algorithms (ORPA) from use in robotics, military use, archeology, and search and rescue. They described several existing ORPA that are based off A*, D*, Improved Ant Colony, and Dijkstra algorithms that used a 3D grid to model sloping terrain to calculate off-road paths. They noted that an Adaptive Dijkstra and Adaptive A* search appeared to be superior as they had lower processing times from the other algorithms by employing dynamic resolution and clustered optimization. They concluded their paper by discussing that to truly compare these ORPAs we need standardized instruments that can be used to test for optimization. An adaptive A* search can be used to continuously calculate suboptimal paths to the goal that would allow for time to be wasted until an optimal path opens that allows a successful game completion.

Running an A* search for multiple suboptimal paths can become very time intensive. Other options would have to be used to meet the one second time requirement each time. "Grid Based 2D Navigation by a Decentralized Robot System with Collison Avoidance" by Yang described the benefit of using more than one robot for 2D navigation [9]. He noted that multiple robots have an advantage over one robot as they can work in unison as a team or can be dynamically decoupled and thus work independently to reach a goal. He proposed an algorithm that allows robots to move around an equilateral triangular grid using shared data to efferently search for a goal. His algorithm sensed the environment, judged the environment based on its current location, communicated its current data, and waited for all other robots to communicate their data then moves. His proposed algorithm allowed for multiple robots to complete the search without collision even if one robot became broken or malfunctioned. The stability of this algorithm comes from the constant communication of data between robots to allow for efficiency and to ensure that the robots do not collide. If the snake can predict future runs, it can collect terrain data that would be like multiple robots sharing data. This shared data can eliminate the need for A* to be run many times to find suboptimal paths, and instead we can calculate a path based on predictive, dynamic path.

There will still be times when it is beneficial for the snake to eliminate as many open spaces as possible. Towards the end of a game of Snake, if all the open spaces are in front of the snake the game ends very quickly. "An Effective Method of Pathfinding in a Car Racing Game" by Wang and Lin described a navigation strategy of an AI controlled race car that vastly outperformed A* for navigating a racetrack [8]. Their strategy involved tracing rays originating from the front 2 corners of the car protruding outward. These rays will eventually collide with the edges of the track. By calculating the distance and the angle of these rays the AI can determine when the car needs to turn to stay on the track. This eliminated the need for dynamic search algorithms and allowed the car to navigate completely unknown courses. To determine what was the track and what was not they used color detection from an overhead camera over the course where the track was a different color than areas that were not the track. This allowed constant sensing of where the track was

along with the ability to determine where the rays collide with the edges of the track. The way this racecar can hug a wall based on its distance from the wall is useful to the Snake algorithm since the agent can easily calculate the closest locations of the tail and traverse the empty spaces nearby to consolidate empty space.

To test the snake to see if the SPA, DPA, TPA, payoff function and predictive, dynamic path are finding an optimal solution, we will need to test the algorithms against a method of solving Snake that already exists. "Hamiltonian Cycles in Solid Grid Graphs" by Umans and Lenhart discussed and proved strategies for creating, proving, and analyzing Hamiltonian Cycles (HC) in solid grid graphs [7]. Solid grid graphs are NP-complete where an entire cycle can be created through nodes where the start is connected to the end of the cycle. Creating 2-factor sub graphs (2FSG) can be used to create an HC by reducing the number of components. Four types of boundary cells make up the 2FSGs. These can be used to identify how best to reduce the 2FSG into an HC. Alternating strip sequences and static alternating strip sequences can be used to identify how to reduce 2FSGs into HCs more efficiently. A Hamiltonian cycle can be created on the snake's grid and used as a path that allows for the snake to traverse every position without every getting trapped by its tail. This will allow us to find a very basic solution to the Snake game and create a baseline average time to compare my tests.

3 Approach

In order to test various pathfinding algorithms on a Snake game, I needed to create the Snake game in an environment that allowed me to both control the snake from code as well as collect data from the environment. For this I relied on the Unity Game engine where I created the game objects and the game code. From the code I was then able to access position of the snake, snake's tail, and food objects at all times. I was then able to create an algorithm that moved the snake in a Hamiltonian cycle, which I used as a baseline for testing. I then modified an A* search and implemented DPAs, TPAs, payoff functions and predictive dynamic pathfinding to develop an algorithm that finds an optimal solution to the Snake game. I finally ran tests on the baseline Hamiltonian Cycle Snake and the A* snake to then compare and analyze the results.

In the Unity Game engine, I created a cube object for the snake head, snake tail, and snake food object. I then created a Grid script (a C# class) that is used for setting the size of the grid that the snake can move on and for keeping track of open spaces for placing the food objects. I created a SnakeTail script that is used to store the locations of the tail on the grid. Finally, I created a HamiltonianSnake and AStarSnake script that is used to control the snake, implement the algorithms, collect data on the snake objects location and movements, and ensure the snake follows the rules of the game. Once the scripts were attached to the game objects, I was ready to begin creating the algorithms.

For the HamiltonianSnake script I found a simple Hamiltonian cycle that is scalable for any grid size n x n such that n is even. This HamiltonianSnake script cycles through the grid, touching every point once before reaching its starting point and starting again. I ran tests on this snake agent and collected game completion times as well as the ration of wins vs losses. Since this algorithm touches every point and the tail is always behind the snake (at previous positions), it was no surprise this snake agent wins every time.

```
while snakeHasNotWon() and snakeHasNotLost():
    followHamiltonianCycle()
```

Figure 1: HamiltonianSnake Algorithm

The AStarSnake algorithm is very complex as it needs to keep track of the current tail's location, the previous positions of the snake agent, and the current path it is on. The AStarSnake essentially collects the location data of the game objects in play, tail and food, and then passes this information to a modified

A* search. The modified A* search runs a regular A* search except with each additional space moved from the original snake position a tail position is removed. This allows for the search to see into the future and remove the tail that will be gone when the snake arrives. This significantly improved the snake's ability to navigate around its tail and made it possible for an A* search to be run when the tail will eventually open a path to the food during the search.

```
while snakeHasNotWon() and snakeHasNotLost():
    removeSectionsOfSnakeTailSnakeConnotReach()
    runAStarSearch()
    if aStarWasSuccessful():
        useFuturePositionToCheckIfTailTipIsReachable()
        if tailTipIsReachableWithFutureSnake():
            setNewPathAsTheAStarResult():
        else:
            followTail()
    else:
        followTail()
```

Figure 2: AStarSnake Algorithm

The next issue was that there are times when using the previous A* search will send the snake into an enclosed position within its tail and cause the game to be lost. In order to overcome this possibility, after each successful calculation of the A* search, the future position of the snake will be stored. If there is a route from the snake's new position to the snake's new tail's tip position, then it takes the path. This is because if the snake head can make it to the tails tip position it can always follow its tail and never be trapped. This ensures that the snake never choses the current optimal path over a successful game completion. This head to tail tip path is used like a TPA that uses the predictability of the snake's dynamic tail to make an intelligent move both for optimal time as well as ensuring success.

An issue arose when the snake would follow its tail. Since when a food object is eaten the tail grows in length there are times where the food object can be placed several times directly in front of the snake. If the snake was following the tail too closely, there was a chance that the food placements would cause an unavoidable loss. This means that there needs to be a payoff function to ensure that the optimal path does not put the snake in this situation when it is likely to face this issue. So, the payoff function I used would cause the snake to take small Hamiltonian paths when the game is near an end (there are few open spaces for the food to be placed). This would in effect be the opposite of an optimal path to the food object, but as the open spaces would become consolidated in front of the snake, the last few food placements cause the game to be won very quickly and it removes the possibility of the snake to be forced to lose the game.

4 Experiment Design and Results

I first ran the HamiltonianSnake algorithm 25 times on a 10x10 grid with a 1 second move time to get an average completion time of 5411 seconds for the Snake game. I also collected the maximum completion time of 6382 seconds and a minimum completion time of 4571 seconds for the 25 runs of the HamiltonianSnake.

With these values as a standard marker I then ran my predictive, dynamic AStarSnake algorithm on the same 10x10 grid for 25 successful games and collected an average completion time of 1325 seconds. I collected the maximum completion time of 1577 seconds and a minimum completion time of 1100 seconds for the 25 runs of the HamiltonianSnake. These results proved that the predictive, dynamic AStarSnake algorithm can complete a game of Snake in less than one third the time it takes the HamiltonianSnake to complete the game.

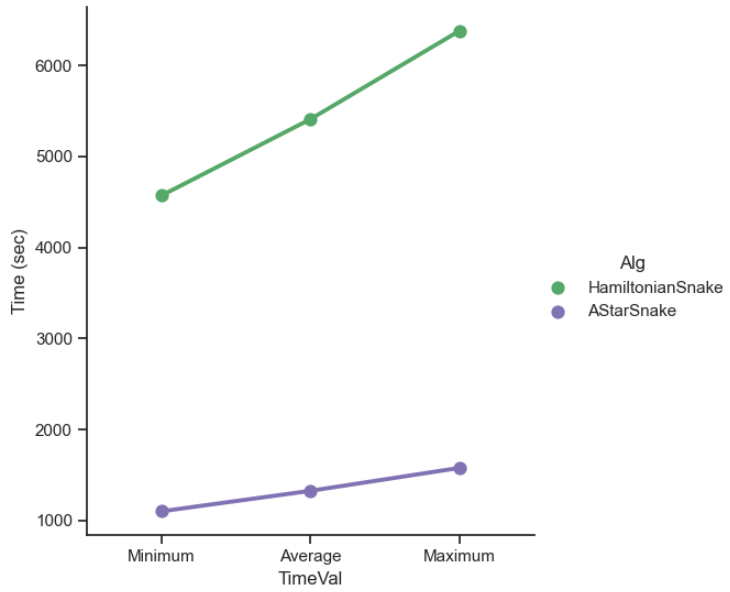

Figure 3: Completion Time Comparison

Table 1: Results from Testing to 25 Completions on 10x10 Grid

	HamiltonianSnake	AStarSnake
Avg. Completion (sec)	5411	1325
Min. Completion (sec)	4571	1100
Max. Completion (sec)	6382	1577
Total Runs To reach 25 Wins	25	46
Wins	25	25
Loses	0	9
Infinite Cycles	0	12

The HamiltonianSnake never lost a game, which is not surprising, and if completing the game is the only metric then this would have perfect results. However, this was far from the case for the AStarSnake algorithm. As the AStarSnake had to run 46 times before it completed the game 25 times. This was due to 9 losses which occurred when the snake became cornered by its tail when food objects appeared in front of the snake immediately after reaching a food item or a food item was placed on the path between the snake and tail and then prevented the snake from continuing its miniature Hamiltonian cycle and caused the game to be lost. In addition to the 9 losses, there were 12 infinite cycles that occurred when the snake would endlessly repeat a cycle that never reached the food. This would happen late in the game when there were few open spaces and there would be one or a few empty spaces buried deep inside of the snake's tail. Then when the snake came to approach the food there was no way for the snake to reach the food without the tail growing and causing the snake to fail. This resulted in the snake cycling indefinitely and never resulted in

a win or loss.

5 Analysis

Without a doubt the predictive, dynamic AStarSnake algorithm completed snake games faster than the HamiltonianSnake. After watching the AStarSnake, I can say it would outperform me as it took very risky moves cutting directly behind the tail as the tail passed by. But the optimality of the algorithm is questionable.

In the beginning the AStarSnake absolutely did run optimally as there was no shorter path to the food object. Even when there was no direct route to the food, the TPA and payoff function ensured that the path chosen was the fastest path as it used the tails future positions to calculate the A* search. The only part of the algorithm that I would say was not optimal was the small Hamiltonian paths that the snake used at the end to consolidate empty space. This behavior was first used when 80% of the spaces were filled by the tail and there was no direct route to the food object. Then when 90% of the spaces were filled by the tail, the AStarSnake exclusively used the small Hamiltonian paths and no longer used the A* search. These chosen points of 80% and 90% could easily be improved on as this was both what caused an infinite cycle when the small Hamiltonian paths started too late and what increased completion times when they were started too early. Since the food objects are placed randomly, there is no determinism of where the empty spaces will be at the end of the game. To find a truly optimal solution, this function should be improved on.

Was this approach intelligent? I would say without a doubt this algorithm shows both incredible intelligence and also foresight that a human player would not have. The AStarSnake's ability to see many moves into the future ensured that each move was calculated to ensure the best chance for success while also finding many optimal paths throughout the game. It is not until the end of the game when there is little room between the snake and its tail when it either loses or ends up in an infinite cycle.

6 Conclusion and Future Work

This project was a complete success. In the beginning it looked to be an absolute disaster as A* search failed as soon as there was no direct path from the snake to the food object. Then there was an issue on how to solve the snake movement when there was no path to the food object even if the snake used all the open spaces around it. This required me to adopt the key method of success in the HamiltonianSnake algorithm where the agent would essentially follow its tail endlessly until success. This inspired me to use the position of the tails tip as a marker that guarantees the snake will never become trapped.

With the A* search powering optimal searches and optimal paths the AStarSnake was extremely successful. I could not find any peer reviewed research papers done on the Snake game, so I have no other data than my own to compare my work to. I would like to have some other intelligent solutions to compare my times with to see if my algorithms were getting relatively close to optimal times. Also, since AStarSnake would lose and find infinite cycles, I wonder if there is any intelligent solution that does not lose on occasion and if there is an easy solution to preventing infinite cycles. I do believe since there is the random placement of the food object there is likely no way that there would be a guaranteed optimal solution as the placement of the food can always appear in in front of the snake enough times to lose the game unless it is running in a perfect Hamiltonian cycle. I am proud of my algorithms as they have performed very well, but I know there is much more work to do with the late game stage of the algorithm, specifically the small Hamiltonian paths.

References

[1] G. V. Alferov and O. A. Malafeyev. The robot control strategy in a domain with dynamical obstacles. In L. Dorst, M. van Lambalgen, and F. Voorbraak, editors, *Reasoning with Uncertainty in Robotics*, pages 209–217, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[2] A. S. Anisyah, P. H. Rusmin, and H. Hindersah. Route optimization movement of tugboat with a tactical pathfinding in spin 3d simulation. In *2015 4th International Conference on Interactive Digital Media (ICIDM)*, pages 1–5, 2015.

[3] A. Y. Kapi, M. S. Sunar, and Z. A. Algfoor. Summary of pathfinding in off-road environment. In *2020 6th International Conference on Interactive Digital Media (ICIDM)*, pages 1–4, 2020.

[4] J. Kuffner. Efficient optimal search of uniform-cost grids and lattices. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, volume 2, pages 1946–1951 vol.2, 2004.

[5] X. Liu and D. Gong. A comparative study of a-star algorithms for search and rescue in perfect maze. In *2011 International Conference on Electric Information and Control Engineering*, pages 24–27, 2011.

[6] Y. Sazaki, H. Satria, and M. Syahroyni. Comparison of a and dynamic pathfinding algorithm with dynamic pathfinding algorithm for npc on car racing game. In *2017 11th International Conference on Telecommunication Systems Services and Applications (TSSA)*, pages 1–6, 2017.

[7] C. Umans and W. Lenhart. Hamiltonian cycles in solid grid graphs. In *Proceedings 38th Annual Symposium on Foundations of Computer Science*, pages 496–505, 1997.

[8] J.-Y. Wang and Y.-B. Lin. An effective method of pathfinding in a car racing game. In *2010 The 2nd International Conference on Computer and Automation Engineering (ICCAE)*, volume 3, pages 544–547, 2010.

[9] X. Yang. Grid based 2d navigation by a decentralized robot system with collison avoidance. In *2017 36th Chinese Control Conference (CCC)*, pages 8473–8478, 2017.

[10] W. Zeng and R. L. Church. Finding shortest paths on real road networks: the case for A*, Apr. 2009.

7 Appendix

All code was written by me. I used the wikipedia.org A* search algorithm pesudocode as a foundation for my predictive, dynamic A* search.

7.1 HamiltonianSnake

```
using System.Collections;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using UnityEngine;
using UnityEngine.SceneManagement;

public class HamiltonianSnake : MonoBehaviour
{
    [SerializeField] GameObject snakeTail;
    Grid grid;
    List<Vector3> previousPositions = new List<Vector3>();

    List<GameObject> tail = new List<GameObject>();

    HamiltonianPath hamPath;

    static int wins = 0;
```

```
static int losses = 0;
static List<float> times = new List<float>();

float time;
float waitTime = .001f;

List<Vector3> path;

bool removeWorked = false;

bool isGameOver = false;

float startTime;

public List<GameObject> Tail { get => tail; set => tail = value; }


// Start is called before the first frame update
void Awake()
{
    hamPath = GetComponent<HamiltonianPath>();
    grid = FindObjectOfType<Grid>();
}

void Start()
{
    List<GameObject> tailCopy = new List<GameObject>(tail);
    path = new List<Vector3>(hamPath.Path);
    time = Time.time;
    startTime = Time.time;
}

// Update is called once per frame
void Update()
{
    if (!isGameOver)
    {
        if (Time.time > time + waitTime)
        {
            if (path.Count > 0)
            {
                time = Time.time;
                MoveSnake(gameObject, path, previousPositions, true);
                if (tail.Count < Grid.WIDTH * Grid.WIDTH - 1)
                {
                    IsGameLost();
                }
                TailAndGoalState(gameObject, grid.Goal.transform.position, path, tail,
                    previousPositions, true, new List<GameObject>());
            }
            else
            {
                List<GameObject> tailCopy = new List<GameObject>(tail);
                path = new List<Vector3>(hamPath.Path);
            }
        }
```

```
    }

    if (tail.Count == 99)
    {
        print("You win!");
        isGameOver = true;
        times.Add((Time.time - startTime));
        wins++;
        File.AppendAllText("HamPathData.txt", " Time for competion: " + (Time.time - startTime)
            + ", with waitime: " + waitTime + ", Win. Win/loss: " + wins + "/" + losses + ",
            average time: " + times.Sum() / times.Count + ", Max time: " + times.Max() + ", Min
            time: " + times.Min() + "\n");

        SceneManager.LoadScene("SampleScene");
    }
}

public void SimulateSnake(GameObject simSnake, Vector3 goalPosition, List<GameObject> simTail,
    List<Vector3> simPrevPositions, List<Vector3> simPath, List<GameObject> addedTailsToDelete)
{
    while (simPath.Count > 0)
    {
        MoveSnake(simSnake, simPath, simPrevPositions, false);
        TailAndGoalState(simSnake, goalPosition, simPath, simTail, simPrevPositions, false,
            addedTailsToDelete);
    }
}

private void IsGameLost()
{
    foreach (GameObject t in tail)
    {
        if (t.transform.position == transform.position)
        {
            isGameOver = true;
            print("Game Over");
            times.Add((Time.time - startTime));
            losses++;
            File.AppendAllText("HamPathData.txt", " Time for competion: " + (Time.time -
                startTime) + ", with waitime: " + waitTime + ", Fail. Win/loss: " + wins + "/"
                + losses + ", average time: " + times.Sum()/times.Count + ", Max time: " +
                times.Max() + ", Min time: " + times.Min() + "\n");
            SceneManager.LoadScene("SampleScene");
        }
    }
}


private void TailAndGoalState(GameObject localSnake, Vector3 goalPosition, List<Vector3>
    localPath, List<GameObject> localTail, List<Vector3> localPreviousPositons, bool
    isRealSnake, List<GameObject> addedTailsToDelete)
{
    if (localSnake.transform.position == goalPosition)
    {
        GameObject firstTail;
        if (isRealSnake)
```

```
        {
            firstTail = Instantiate(snakeTail, localPreviousPositons[0], Quaternion.identity);
        }
        else
        {
            firstTail = new GameObject();
            firstTail.name = "First tail";
            firstTail.transform.position = localPreviousPositons[0];
            addedTailsToDelete.Add(firstTail);
        }
        localTail.Add(firstTail);
        firstTail.transform.position = localPreviousPositons[0];
        if (isRealSnake)
        {
            grid.PlaceFood();
            CheckIfFoodOverlapsTail();
        }
    }
    else
    {
        if (localTail.Count > 0)
        {
            if (isRealSnake)
            {
                grid.AddSpace(localTail[0].transform.position);
            }
            localTail[0].transform.position = localPreviousPositons[0];
            GameObject recycledTail = localTail[0];
            localTail.RemoveAt(0);
            localTail.Add(recycledTail);
        }
    }
}

private void CheckIfFoodOverlapsTail()
{
    foreach (GameObject t in tail)
    {
        if (t.transform.position == grid.Goal.transform.position)
        {
            print("Food overlapped tail!!!!!!!!!!!!!!!!");
            isGameOver = true;
        }
    }

    if (grid.Goal.transform.position == transform.position)
    {
        print("Food overlapped snake!!!!!!!!!!!!!!!!");
        isGameOver = true;
    }
}

private void MoveSnake(GameObject localSnake, List<Vector3> localPath, List<Vector3>
     localPreviousPositons, bool isRealSnake)
{
    localPreviousPositons.Insert(0, localSnake.transform.position);
```

```
        if (localPreviousPositons.Count > Grid.WIDTH * Grid.HEIGHT)
        {
            localPreviousPositons.RemoveAt(localPreviousPositons.Count - 1);
        }


        localSnake.transform.position = localPath[0];
        if (isRealSnake && tail.Count == 0 && transform.position != grid.Goal.transform.position)
        {
            grid.AddSpace(localPreviousPositons[0]);
        }
        localPath.RemoveAt(0);
        if (isRealSnake)
        {
            removeWorked = grid.RemoveSpace(localSnake.transform.position);
            if (!removeWorked)
            {
                print("Remved failed");
            }
        }
    }
}
```

7.2 Hamiltonian Path

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class HamiltonianPath : MonoBehaviour
{
    List<Vector3> path = new List<Vector3>();
    Grid grid;

    public List<Vector3> Path { get => path; set => path = value; }

    // Start is called before the first frame update
    void Awake()
    {
        bool down = true;
        grid = FindObjectOfType<Grid>();
        for(int i = 0; i < Grid.WIDTH; i++)
        {
            if(i == 0 || i == Grid.WIDTH - 1)
            {
                if (down)
                {
                    for (int j = Grid.WIDTH - 1; j >= 0; j--)
                    {
                        Path.Add(new Vector3(i, 0, j));
                    }
                }
                else
                {
```

```
                for (int j = 0; j < Grid.WIDTH; j++)
                {
                    Path.Add(new Vector3(i, 0, j));
                }
            }

            // end pieces
        }
        else
        {
            if (down)
            {
                for (int j = Grid.WIDTH - 2; j >= 0; j--)
                {
                    Path.Add(new Vector3(i, 0, j));
                }
            }
            else
            {
                for (int j = 0; j < Grid.WIDTH - 1; j++)
                {
                    Path.Add(new Vector3(i, 0, j));
                }
            }
            // center cuts
        }
        down = !down;
    }
    // last top strip
    for (int i = Grid.WIDTH - 2; i > 0; i--)
    {
        Path.Add(new Vector3(i, 0, Grid.WIDTH - 1));
    }
  }
}
```

7.3 AStarSnake

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using UnityEngine;
using UnityEngine.SceneManagement;

public class AStarSnake : MonoBehaviour
{
    [SerializeField] GameObject snakeTail;
    Grid grid;
    List<Vector3> previousPositions = new List<Vector3>();

    List<GameObject> tail = new List<GameObject>();
```

```
PredictiveDynamicAStar aStar = new PredictiveDynamicAStar();

float time;
float waitTime = .1f;

List<Vector3> path;

bool isChasingTail;

bool removeWorked = false;

bool isGameOver = false;

static int wins = 0;
static int losses = 0;
static List<float> times = new List<float>();

public List<GameObject> Tail { get => tail; set => tail = value; }

int tailChaseCount = 0;

static int infiniteLoops = 0;

float startTime;


// Start is called before the first frame update
void Awake()
{
    grid = FindObjectOfType<Grid>();
}

void Start()
{
    List<GameObject> tailCopy = new List<GameObject>(tail);
    (path, isChasingTail) = aStar.AStarSearch(transform.position, grid.Goal.transform.position,
        tailCopy, gameObject, false, false);
    time = Time.time;
    startTime = Time.time;
}

// Update is called once per frame
void Update()
{
    try
    {
        //print("Path size: " + path.Count);
        if (!isGameOver)
        {
            if (Time.time > time + waitTime)
            {
                if (path.Count > 0)
                {
                    time = Time.time;
                    MoveSnake(gameObject, path, previousPositions, true);
                    if (tail.Count < Grid.WIDTH * Grid.WIDTH - 1)
```

```
                {
                    IsGameLost();
                }
                TailAndGoalState(gameObject, grid.Goal.transform.position, path, tail,
                    previousPositions, true, new List<GameObject>());
            }
            else
            {
                List<GameObject> tailCopy = new List<GameObject>(tail);
                (path, isChasingTail) = aStar.AStarSearch(transform.position,
                    grid.Goal.transform.position, tailCopy, gameObject, false, false);
            }
        }
    }

    if (tail.Count == 99)
    {
        print("You win!");
        isGameOver = true;
        times.Add((Time.time - startTime));
        wins++;
        File.AppendAllText("AStarData.txt", "Time for competion: " + (Time.time - startTime)
            + ", with waitime: " + waitTime + ", Win. Win/loss: " + wins + "/" + losses +
            ", average time: " + times.Sum() / times.Count + ", Max time: " + times.Max() +
            ", Min time: " + times.Min() + ", Infinte loops: " + infiniteLoops + "\n");
        SceneManager.LoadScene("SampleScene");
    }
}
catch
{
    try
    {
        if (tail.Count == 99)
        {
            print("You win!");
            isGameOver = true;
            times.Add((Time.time - startTime));
            wins++;
            File.AppendAllText("AStarData.txt", "Time for competion: " + (Time.time -
                startTime) + ", with waitime: " + waitTime + ", Win. Win/loss: " + wins +
                "/" + losses + ", average time: " + times.Sum() / times.Count + ", Max time:
                " + times.Max() + ", Min time: " + times.Min() + ", Infinte loops: " +
                infiniteLoops + "\n");
            SceneManager.LoadScene("SampleScene");
        }
        else
        {
            losses++;
            File.AppendAllText("AStarData.txt", "Time for competion: " + (Time.time -
                startTime) + ", with waitime: " + waitTime + ", Loss. Win/loss: " + wins +
                "/" + losses + ", average time: " + times.Sum() / times.Count + ", Max time:
                " + times.Max() + ", Min time: " + times.Min() + ", Infinte loops: " +
                infiniteLoops + "\n");
            SceneManager.LoadScene("SampleScene");
        }
    }
```

```
        catch
        {
            print("Some wack bullshit");
            SceneManager.LoadScene("SampleScene");
        }
    }
}

public void SimulateSnake(GameObject simSnake, Vector3 goalPosition, List<GameObject> simTail,
    List<Vector3> simPrevPositions, List<Vector3> simPath, List<GameObject> addedTailsToDelete)
{
    while (simPath.Count > 0)
    {
        MoveSnake(simSnake, simPath, simPrevPositions, false);
        TailAndGoalState(simSnake, goalPosition, simPath, simTail, simPrevPositions, false,
            addedTailsToDelete);
    }
}

private void IsGameLost()
{
    foreach (GameObject t in tail)
    {
        if (t.transform.position == transform.position)
        {
            isGameOver = true;
            print("Game Over");
            losses++;
            File.AppendAllText("AStarData.txt", "Loss\n");
            SceneManager.LoadScene("SampleScene");
        }
    }
}


private void TailAndGoalState(GameObject localSnake, Vector3 goalPosition, List<Vector3>
    localPath, List<GameObject> localTail, List<Vector3> localPreviousPositons, bool
    isRealSnake, List<GameObject> addedTailsToDelete)
{
    if (localSnake.transform.position == goalPosition)
    {
        tailChaseCount = 0;
        GameObject firstTail;
        if (isRealSnake)
        {
            firstTail = Instantiate(snakeTail, localPreviousPositons[0], Quaternion.identity);
        }
        else
        {
            firstTail = new GameObject();
            firstTail.name = "First tail";
            firstTail.transform.position = localPreviousPositons[0];
            addedTailsToDelete.Add(firstTail);
        }
        localTail.Add(firstTail);
        firstTail.transform.position = localPreviousPositons[0];
```

```
        if (isRealSnake)
        {
            List<Vector3> pathReceiver;
            grid.PlaceFood();
            CheckIfFoodOverlapsTail();
            List<GameObject> tailCopy = new List<GameObject>(localTail);

            (pathReceiver, isChasingTail) = aStar.AStarSearch(transform.position,
                grid.Goal.transform.position, tailCopy, localSnake, false, false);
            localPath.AddRange(pathReceiver);
        }
    }
    else
    {
        if (localTail.Count > 0)
        {
            if (isRealSnake)
            {
                grid.AddSpace(localTail[0].transform.position);
            }
            localTail[0].transform.position = localPreviousPositons[0];
            GameObject recycledTail = localTail[0];
            localTail.RemoveAt(0);
            localTail.Add(recycledTail);
        }
    }
}

private void CheckIfFoodOverlapsTail()
{
    foreach (GameObject t in tail)
    {
        if (t.transform.position == grid.Goal.transform.position)
        {
            print("Food overlapped tail!!!!!!!!!!!!!!!!");
            isGameOver = true;
        }
    }

    if (grid.Goal.transform.position == transform.position)
    {
        print("Food overlapped snake!!!!!!!!!!!!!!!!");
        isGameOver = true;
    }
}

private void MoveSnake(GameObject localSnake, List<Vector3> localPath, List<Vector3>
    localPreviousPositons, bool isRealSnake)
{
    if (isChasingTail)
    {
        tailChaseCount++;
        if (tailChaseCount > Mathf.Pow(Grid.WIDTH, 2) * 3)
        {
            infiniteLoops++;
            File.AppendAllText("AStarData.txt", "Infinte Loop\n");
```

```
                SceneManager.LoadScene("SampleScene");
            }
        }
        else
        {
            tailChaseCount = 0;
        }

        localPreviousPositons.Insert(0, localSnake.transform.position);
        if (localPreviousPositons.Count > Grid.WIDTH * Grid.HEIGHT)
        {
            localPreviousPositons.RemoveAt(localPreviousPositons.Count - 1);
        }


        localSnake.transform.position = localPath[0];
        if (isRealSnake && tail.Count == 0 && transform.position != grid.Goal.transform.position)
        {
            grid.AddSpace(localPreviousPositons[0]);
        }
        localPath.RemoveAt(0);
        if (isRealSnake)
        {
            removeWorked = grid.RemoveSpace(localSnake.transform.position);
            if (!removeWorked)
            {
                print("Remved failed");
            }
        }
    }
}
```

7.4 PredictiveDynamicAStar

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

// https://en.wikipedia.org/wiki/A*_search_algorithm
    ------------------------------------------------------------------------

public class PredictiveDynamicAStar : MonoBehaviour
{
    private List<Vector3> ReconstructPath(Dictionary<Vector3, Vector3> cameFrom, Vector3 current)
    {
        List<Vector3> totalPath = new List<Vector3>();
        totalPath.Add(current);
        while (cameFrom.ContainsKey(current))
        {
            current = cameFrom[current];
            totalPath.Insert(0, current);
        }
        totalPath.RemoveAt(0);
        return totalPath;
```

```
}

private float Heuristic(Vector3 current, Vector3 goal)
{
    float euclideanDistance = Vector3.Distance(current, goal);
    return euclideanDistance;
}

private List<Vector3> GetOpenNeighbors(Vector3 position, List<GameObject> tail)
{
    List<Vector3> neighbors = new List<Vector3>();

    Vector3 up = new Vector3(0, 0, 1);
    Vector3 right = new Vector3(1, 0, 0);


    // check right
    bool isRightAvailabe = true;
    if (Grid.WIDTH > position.x + 1)
    {
        foreach (GameObject t in tail)
        {
            if (Vector3.Distance(position + right, t.transform.position) < 0.1)
            {
                isRightAvailabe = false;
            }
        }
        if (isRightAvailabe)
        {
            neighbors.Add(position + right);
        }
    }

    // check left
    bool isLeftAvailabe = true;
    if (position.x > 0)
    {
        foreach (GameObject t in tail)
        {
            if (Vector3.Distance(position - right, t.transform.position) < 0.1)
            {
                isLeftAvailabe = false;
            }
        }
        if (isLeftAvailabe)
        {
            neighbors.Add(position - right);
        }
    }

    // check up
    bool isUpAvailabe = true;
    if (Grid.HEIGHT > position.z + 1)
    {
        foreach (GameObject t in tail)
        {
```

```
            if (Vector3.Distance(position + up, t.transform.position) < 0.1)
            {
                isUpAvailabe = false;
            }
        }
        if (isUpAvailabe)
        {
            neighbors.Add(position + up);
        }
    }

    // check down
    bool isDownAvailabe = true;
    if (position.z > 0)
    {
        foreach (GameObject t in tail)
        {
            if (Vector3.Distance(position - up, t.transform.position) < 0.1)
            {
                isDownAvailabe = false;
            }
        }
        if (isDownAvailabe)
        {
            neighbors.Add(position - up);
        }
    }

    return neighbors;
}

private float EdgeWeight(Vector3 current, Vector3 neighbor)
{
    // If I add weights to the edges I will do so here
    return 1;
}



private List<Vector3> CollectAllOpenMoves(GameObject snake, List<GameObject> originalTailCopy,
    Vector3 tailTip)
{
    List<Vector3> openMoves = GetOpenNeighbors(snake.transform.position, originalTailCopy);
    openMoves.Sort((a, b) => Vector3.Distance(b, tailTip).CompareTo(Vector3.Distance(a,
        tailTip)));
    if (openMoves.Count > 0)
    {
        foreach (Vector3 v in openMoves)
        {
            List<Vector3> newPath = new List<Vector3>();
            newPath.Add(v);

            if (CheckIfTailIsReachableFromGoalPosition(newPath, originalTailCopy, snake))
            {
                return newPath;
            }
```

```
        }
    }
    return new List<Vector3>();
}

private bool CheckIfTailIsReachableFromGoalPosition(List<Vector3> newPath, List<GameObject>
    tail, GameObject snake)
{
    List<Vector3> simulatedPath = new List<Vector3>(newPath);
    List<GameObject> simulatedTail = new List<GameObject>();
    List<GameObject> gameObjectsToDelete = new List<GameObject>();
    List<Vector3> simPreviousPositions = new List<Vector3>();
    foreach (GameObject t in tail)
    {
        GameObject tailToDel = new GameObject();
        tailToDel.name = "Tail to del1";
        tailToDel.transform.position = t.transform.position;
        gameObjectsToDelete.Add(tailToDel);
        simulatedTail.Add(tailToDel);
        simPreviousPositions.Add(t.transform.position);
    }
    GameObject simulatedSnake = new GameObject();
    simulatedSnake.name = "Simulated snake";
    simulatedSnake.transform.position = snake.transform.position;
    gameObjectsToDelete.Add(simulatedSnake);
    AStarSnake s = new AStarSnake();
    s.SimulateSnake(simulatedSnake, simulatedPath[simulatedPath.Count - 1], simulatedTail,
        simPreviousPositions, simulatedPath, gameObjectsToDelete);
    Destroy(s);
    List<Vector3> pathToTailTip;
    bool isChasingTail;
    (pathToTailTip, isChasingTail) = AStarSearch(simulatedSnake.transform.position,
        simulatedTail[0].transform.position, simulatedTail, simulatedSnake, false, true);

    foreach (GameObject g in gameObjectsToDelete)
    {
        Destroy(g);
    }

    if (pathToTailTip.Count != 0)
    {
       // print("Passed");
        return true;
    }
    //print("Failed");
    return false;
}

private void RemoveUnreachableTail(Dictionary<Vector3, Vector3> cameFrom, Vector3 current,
    List<GameObject> tail)
{
    List<Vector3> currentPath;
    currentPath = ReconstructPath(cameFrom, current);

    foreach (Vector3 v in currentPath)
    {
```

```
        if (tail.Count > 0)
        {
            tail.RemoveAt(0);
        }
    }
}

public (List<Vector3>, bool) AStarSearch(Vector3 start, Vector3 goal, List<GameObject> tail,
    GameObject snake, bool isChasingTail, bool isTailCheck)
{
    List<GameObject> tailOriginal = new List<GameObject>(tail);

    List<Vector3> openSet = new List<Vector3>();
    openSet.Add(start);
    Dictionary<Vector3, Vector3> cameFrom = new Dictionary<Vector3, Vector3>();

    // gScores not in list should be treated as infinity
    Dictionary<Vector3, float> gScore = new Dictionary<Vector3, float>();
    gScore.Add(start, 0);

    Dictionary<Vector3, float> fScore = new Dictionary<Vector3, float>();
    fScore.Add(start, Heuristic(start, goal));

    while (openSet.Count > 0)
    {
        Vector3 current = openSet[0]; // This should be the vector3 with the lowest fScore in
            the list
        tail = new List<GameObject>(tailOriginal);
        RemoveUnreachableTail(cameFrom, current, tail);


        if (Vector3.Distance(current, goal) < 0.1f)
        {
            // This section is where the goal is reachable
            //-------------------------------------------------------------

            List<Vector3> newPath = ReconstructPath(cameFrom, current);

            bool isTailReachable = true;
            if (!isChasingTail && !isTailCheck && tail.Count > 0)
            {
                isTailReachable = CheckIfTailIsReachableFromGoalPosition(newPath, tailOriginal,
                    snake);
            }
            if (isTailReachable)
            {
                if (!isTailCheck && snake.GetComponent<AStarSnake>().Tail.Count > Grid.WIDTH *
                    Grid.HEIGHT * UniversalConstans.HOMESTRETCH_TAIL_LENGTH)
                {
                    List<GameObject> originalTailCopy = new List<GameObject>(tailOriginal);
                    List<Vector3> deviation = CollectAllOpenMoves(snake, originalTailCopy,
                        tailOriginal[0].transform.position);
                    if (deviation.Count > 0) // && Vector3.Distance(snake.transform.position,
                        tailOriginal[tailOriginal.Count - 1].transform.position) < 1.1f)
                    {
                        return (deviation, true);
```

```
                }
            }

            return (newPath, isChasingTail);
        }
        else
        {
            // This is where the goal is reachable but the tail is not
            List<GameObject> originalTailCopy = new List<GameObject>(tailOriginal);
            List<Vector3> deviation = CollectAllOpenMoves(snake, originalTailCopy,
                tailOriginal[0].transform.position);
            if (deviation.Count > 0)
            {
                return (deviation, true);
            }
            return PredictiveDynamicAStar(snake.transform.position,
                tailOriginal[0].transform.position, tailOriginal, snake, true, false);
        }
    }

    openSet.Remove(current);

    foreach (Vector3 neighbor in GetOpenNeighbors(current, tail))
    {
        float tenativeGScore = gScore[current] + EdgeWeight(current, neighbor);
        if (!gScore.ContainsKey(neighbor) || tenativeGScore < gScore[neighbor])
        {
            cameFrom[neighbor] = current;

            if (gScore.ContainsKey(neighbor))
            {
                gScore[neighbor] = tenativeGScore;
            }
            else
            {
                gScore.Add(neighbor, tenativeGScore);
            }
            fScore[neighbor] = gScore[neighbor] + Heuristic(neighbor, goal);
            if (!openSet.Contains(neighbor))
            {
                openSet.Add(neighbor);
            }
        }
    }
}

// This is when there is no route to the goal

if (isTailCheck)
{
    return (new List<Vector3>(), false);
}
else
{
    List<GameObject> originalTailCopy = new List<GameObject>(tailOriginal);
```

```
            List<Vector3> deviation = CollectAllOpenMoves(snake, originalTailCopy,
                tailOriginal[0].transform.position); ;
            if (deviation.Count > 0) // && Vector3.Distance(snake.transform.position,
                tailOriginal[tailOriginal.Count - 1].transform.position) < 1.1f)
            {
                return (deviation, true);
            }
            //print("Tail not reachable2");

            return PredictiveDynamicAStar(snake.transform.position,
                tailOriginal[0].transform.position, tailOriginal, snake, true, false); // this
                empty list represents failure
        }
    }
}
```

7.5 Game Images

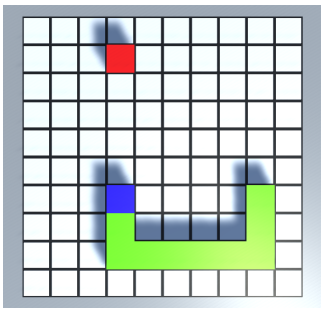

Figure 4: Direct Path A* Search

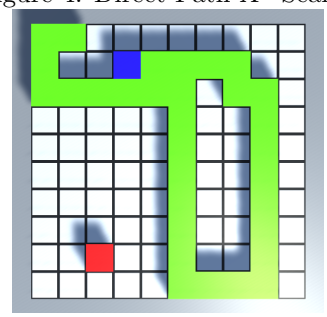

Figure 5: Indirect Path A* Search

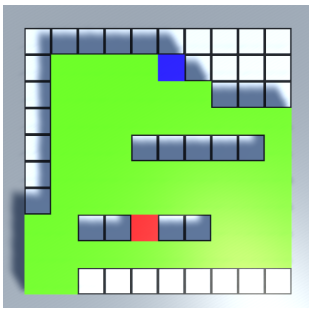

Figure 6: No Indirect Path

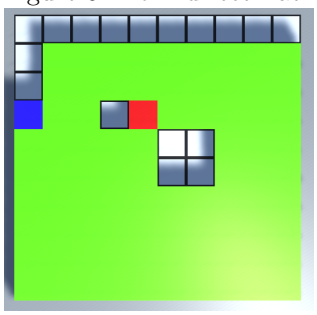

Figure 7: Infinite Cycle