Self-playing Snake Game based on Pathfinding Algorithms

Yang Sui
ECE, School Of Graduate Studies
Rutgers University
hourunlinxa@gmail.com

Jin Xu
ECE, School Of Graduate Studies
Rutgers University
hourunlinxa@gmail.com

Runlin Hou
ECE, School Of Graduate Studies
Rutgers University
hourunlinxa@gmail.com

I. THE SNAKE GAME

Before we implement the algorithms, we first need to build a snake game as a platform. Our design is to use graph theory to implement the snake game.

We separate the snake game into three parts:

- map
- snake
- food

The map is where the snake takes its movement, so we will build the map as the fundamental graph. Considering the structure of the map, it is like a chessboard with blocks arranged in rows and columns that are perpendicular to each other. Each block of the map will be treated as a vertice, and the edges will be used to describe the connection of two vertices.

For the body of the snake, since the blocks that the body was taken are unreachable for the snake. So we decide to mark those vertices as taken.

The food of the position will be saved. But the vertice itself will be the same as the other vertices since the food must be reachable for the snake to take.

As a whole, the snake game will be built on an undirected graph with the same weight for all edges. Because the snake would be able to reach every vertice, and each vertice will only take one movement to move to its adjacent vertice.

In the implementation of our snake on graph theory, all three parts can be built as follow:

- map, an undirected and unweighted map
- snake, will be marked as unreachable vertice to its adjacencies
- food, normal vertice but the position will be saved

For the code implementation, we found that the map of the snake follows strict mathematical rules like every row has the same vertices, and also every column has the same vertices. Mathematically, each vertice's position can be calculated by adding or subtracting a certain value. Considering this situation, we save the whole map in a list and make a judgment of their connection according to their coordinates.

II. BFS APPROACH

Before I choose to use BFS as the algorithm, I actually think about some other ways to achieve my goals, like DFS or Dijkstra. For DFS, it will keep going on one path until it reaches the target. And renew this path if the other path is shorter. So with the map getting bigger, it will take much longer than the BFS. As for Dijkstra, when facing an undirected graph with the same weights for all edges, Dijkstra is exactly BFS. So I take BFS instead.

Introduction

We will start by talking about some common sense of the BFS algorithm. We all know that the BFS algorithm will traversal all of the vertices to find the target that we are looking for. If we are doing a BFS in a tree, and we search from the root node, the BFS algorithm can be seen as searching layer by layer from top to bottom. When we deal with a graph, we need to set an initial vertice for the algorithm and the process of the algorithm will spread like a water wave.

So generally speaking, the basic logic of the BFS algorithm is to set up an initial vertice and keep visiting the adjacencies until we find the target.

Implementation

The implementation of the BFS is simple. And since we are dealing with an undirected graph with the same weight for edges. The process becomes easier. Considering the attribute of the BFS itself, every path found by BFS would be exactly the shortest path to the target because every vertice will be immediately added to the queue right after their parents been visited.

I use two functions to implement the kernel goal of the BFS algorithm.

- `connected(vertice)`
- `bfs(head, target)`

`connect(ver)` will take a vertice as an input. And the output is a list of the adjacencies of this vertice. As we mentioned in the last section, the graph that our map mapped to is a list. Also, as we knew, the map of the Snake Game is like a chessboard, that each node has four edges and is vertical to each other. So the mathematical relationship of connected vertices can be described as following:

a, b are coordinates of two vertices, when a, b can fit the following equtoins, then they are connected. w is the length of the map.

$$a = b - 1 \tag{1}$$
$$a = b + 1 \tag{2}$$
$$a = b - w \tag{3}$$
$$a = b + w \tag{4}$$

Also during the judgment, we need to exclude the walls and the body of the snake, since they will be treated as unreachable vertices of the map. This can be easily achieved because we store the index of them as attributes in the `solver` class.

`bfs(head, target)` takes the head position of the snake and the target position as inputs. And it will return the shortest path from the snake's head to the target. To achieve the purpose of BFS, we first need to create a FIFO queue to record the incoming vertices. Each time we visited a new vertice, we will add its adjacencies into the queue for future visiting. This can be done easily combined with the `connected(ver)`. Basically, the process will be going as visiting the vertices whose distance is 1 to head, and visited the vertices whose distance is 2 to head and so on. Once we reach the target, we will stop the loop.

Strategy

The kernel of the algorithm is to find the shortest path to the food. But it is not enough to find the shortest path for the snake. Also, we need to tell the snake how it can try to be alive, so that snake can get more scores.

I have the snake to follow three rules for the entire game. The first one is to find the food. As long as the snake can reach the food, it will go for it directly. But of course, with the game going on, the body of the snake might block the food from the head. This is when the snake can not reach the food. And here comes the second rule that when the snake can not reach the food, it will chase the tail. And of course, there will also be a situation that the snake can not found the food nor the tail. When we are facing this situation, I'll ask the sanke to move randomly and pray for the food can be reached several movements later.

Also, it is obvious that the game is a dynamic process. But the path we found is always based on a map in a certain moment. This means that we may not get the shortest path cause with the movement of snakes body, the connection of each vertice might be updated, then some new path will be created. So I will have the snake find a new path after every movement, though the first path can lead the snake to destination.