Applying Hamiltonian Cycles to Solving the Classic Game of Snake

Matthew Kurniadi, Alex Shoulders, Arsh Suri

10 November 2022

Abstract

The classical game of Snake, where a player maneuvers a continuously growing line on a grid, seems simple on the surface, but it has many layers of complexity when it comes to achieving a victory in the game. The main focus of this project is to apply graph theory concepts to develop an algorithm to always obtain a victory in the classic variation of Snake, given the size of an NxN playing field. By analyzing this grid the game takes place on as a connected graph, we transform the game into a Hamiltonian cycle problem to find a path our player character can follow to always win. Playing on a 9 x 10 test grid, we used a backtracking algorithm to find a Hamiltonian cycle. We then followed that generated cycle with our player character and repeatedly obtained a victory, by reaching the maximum length of our character, demonstrating that Hamiltonian cycles, although inefficient, can be used to consistently win the game of Snake.

1 Introduction

Invented in 1976, Snake is a simple genre of games involving a line, sometimes referred to as a “snake”, on a grid. The aim of the game is simple: to increase the length of your snake, achieving a “win” when your snake covers the entire game board. Although starting out as a humble arcade game called Blockade, it has now been downloaded on over 400 million phones and a variation of it even has been created by the tech giant Google (Blog). However, despite its widespread popularity, the game is notoriously difficult to win; most players often opting to beat their own length high score rather than the game itself. A solution to win every time is not common knowledge and we wanted to develop our own algorithm to do so.

The game of snake often serves as a benchmark for machine learning and path-finding algorithms (Sebastianelli et al.), as it presents a multi-faceted problem with many unique approaches and play styles. However, machine learning methods, while usually achieving victory, often can’t achieve it every single trial. While there are no direct benefits to the application of the algorithm we created, besides completing Snake, it served as the perfect base for students to learn the essentials and applications of different mathematical concepts. Additionally, parts of the algorithm for routing the snake could also be used to route other objects in the real world, such as delivery bots on city blocks or public transportation aiming to travel through each of its stations. Though not obvious, many different aspects of this algorithm involve complex combinatorial concepts.

To tackle this issue, we first determined the exact ”game of snake” we wanted to solve, which was closest to the Google variant. We then tried to apply graph theory concepts and attempted to use them to form an algorithm we then tested in multiple trials.

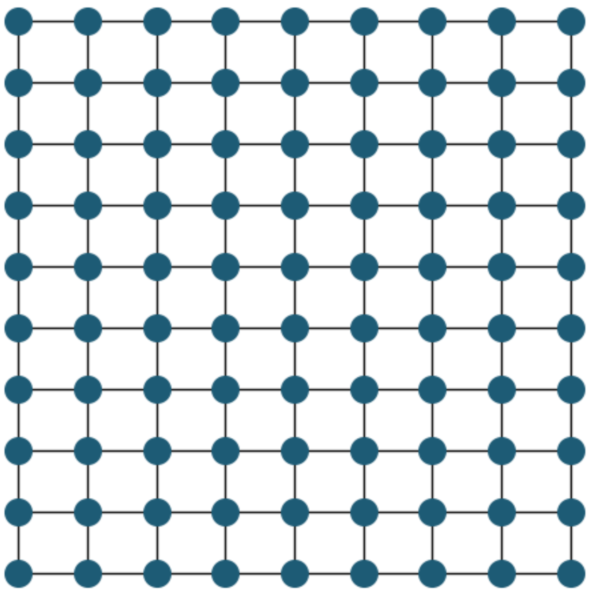
2 Background

Throughout all the variations of the game Snake, there are a few mechanics are consistent throughout all of them. The player's length grows with each successive object, or "apple", it passes through, or "collects". Apples occupy one square on the playing grid and are placed randomly in a space that the snake doesn't currently occupy. The snake continuously moves relative to its head, a designated end of the snake. Players can then use button presses, like the arrow keys or "WASD", to change the direction the head moves to collect apples to achieve the maximum length. Players can only change the head's movement in a direction perpendicular to its current movement, or left and right relative to the head's orientation. The rest of the snake's body follows the path the player takes with the head. The player loses if the snake hits the edge of the play space, referred to as the "wall", or intersects itself. We want to develop an algorithm that efficiently and quickly grows the snake to occupy every cell in the grid and win (Sebastianelli et al.).

As there are several variants of Snake, it is important to specify the version we want to solve. Firstly, Snake takes place in a square grid with N x N cells. For our own program it will be developed in a 9 x 10 grid, but we will address applications regarding larger grids. The starting length of the snake usually varies as well, but more modern versions use four cells, so we will as well. Additionally, we will force our player snake character to start in the same location every time, relative to the walls of the grid, as is common practice for modern games. We will also constrain the 1st apple spawn position to a few units, relative to the size of the grid, above the head of the snake. For reference, a step is when the snake moves one tile thus crossing a grid line. We will not address solving times in seconds as different variants and versions have the snake move at different speeds. Our algorithm's performance will be measured with two factors: completion rate, how often it wins Snake, and the average number of steps it takes.

3 Approach

In order to apply graph theory concepts, we must transform the snake play space into a graph. For each cell, our snake, no matter which direction it enters the cell, can only move to adjacent cells, not including diagonals. By representing each cell as a single vertex and adding edges between each of its adjacent non-diagonal cells, we can form a connected graph. Doing this process for a $9x10$ play space, our primary test size, yields the graph below.

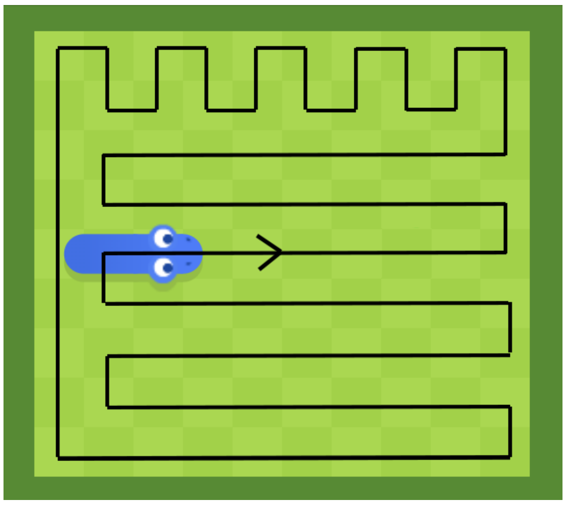


This forms a connected graph. Using the experience we gained playing Snake ourselves, we realized that the steps to to take the first few apples are have little significance. However, the difficulty begins to arise as the snake takes up more room and has runs the risk to trap itself, eventually losing by hitting the wall or on self. The crucial part is setting up a situation to the snake can collect the last apple to reach max length. To reach max length, as apples can't spawn on top of snake, the snake must cover the entire grid or apples will spawn in inaccessible places surrounded by the snake's body. Additionally, in order to reach max length, as the snake can't intersect with itself, the player must navigate the snake in a way to travel to each cell exactly once on the last NxN steps before victory.

Applying this to our connected graph, the path the player is travelling in the last NxN can be represented as a sequence of vertices. This sequence must contain all vertices, to reach max length, and each vertex must only be present once in the sequence, as the snake can't pass through itself. According to graph theory, a sequence of vertices satisfying these conditions for a graph G is called a Hamiltonian Path (Keller & Trotter). As the last NxN steps must follow a Hamiltonian path, we need to find a route for all for previous steps. If we were to determine a Hamiltonian Cycle, a Hamiltonian path where the final vertex in the sequence is has an edge with the first vertex, we can theoretically keep following that forever until we win, as the last NxN steps will be Hamiltonian. That is the crux of our snake-winning algorithim.

4 Solution

Now, for our algorithm, we had to find a way to determine a possible Hamiltonian cycle for the player to follow. One naive way to do this would be go through all the permutations of each vertex on our graph, checking it satisfies the conditions of a Hamiltonian Cycle. That means our program would have to go through, for our test grid, 90! combinations. We decided to go with an backtracking algorithm. This algorithm would start with some initial point. It would then only add vertices to form a path that were adjacent to the latest point and not previously added. This would result in us eliminating many unreasonable combinations. We found implemented versions of a backtracking algorithm in Python. We then modified that code to only take a Python list of every vertex and its connections. Using that functions, we then programmed a way to generate a connected vertex graph list for any NxN grid. Using both of these features together to find a Hamiltonian cycle on = 9 x 10 grid results in this path:



5 Drawbacks

By creating a path that hits every cell once and then repeats the pattern, the snake can cover all cells without worrying about intersections. Because the same path is being traced over and over, you are guaranteed to run over the apples eventually. This graph is a Hamiltonian cycle because it goes over every vertex exactly once and returns to the starting point.

This method is horribly slow though. With this method, the average steps taken are equal to one-half of the empty cells. Thus at the beginning, there are 9 x 10 = 90 cells, and minus the four that the snake occupies would be 86 empty cells. This means the average time to collect the next apple would be 43 steps. This can be used to create a formula for the average steps to complete a round. It is empty cells at the start times the average time to collect the first apple divided by two. This means that in a 50 x 50 grid, this method would take nearly 1.6 million steps.

One of the hardest parts of using our Snake algorithm is its reliance on Hamiltonian cycles. For the snake to not intersect itself, the user must follow a Hamiltonian cycle that is larger than the snake at all times. The trick about finding Hamiltonian paths is that it is an NP-complete problem. For our purposes, this means two major things. There are no concrete properties of a graph that allow us quickly tell if it has a Hamiltonian cycle, and brute force checks for Hamiltonian paths take extremely long amounts of time. A good, efficient computer could check a graph of size 9 x 10 in a reasonable time frame, but any grid of size over 25 x 25 would not be reasonable for any computer.

Another problem with using Hamiltonian paths is the inability to find said path(s) when the dimensions of the playspace are both odd-numbered. To prove this statement, we will use the fact that a bipartite graph, in order to a Hamiltonian cycle, must have an even number of vertices. The graph can be easily converted into a bipartite graph, since there is a chromatic number of 2 for the graph. Imagine a chess board with two colors: black and white. This distribution leads to only two different colors, without any of the same colors touching. Each coloring leads to a set of one side of the bipartite graph.

To prove that it must have an even number of vertices, we can state that a graph is bipartite if and only if all cycles of said graph has even length (Wilson 42). Let us assume a bipartite graph with an odd amount of vertices has a Hamiltonian cycle. This must mean that the cycle has an odd length, but then it can't be a bipartite graph. This proves that a bipartite graph cannot have an odd amount of vertices if it wants to also have a Hamiltonian cycle.

Since our graph is essentially a grid, the number of vertices can be found by multiplying the dimensions of the grid. A grid, and its corresponding graph, when both dimensions are odd, leads to an odd number of vertices. Therefore, a Hamiltonian cycle cannot be found on grids with both odd dimensions, so our method of completion will not work on said Snake games with the dimensions.

6 Conclusion

The Snake game can be completed using a variety of simple or innovative methods and algorithms but with widely varying completion times relative to each other. Our program utilizes a Hamiltonian cycle, found from the dimensions of the grid, between all the cells, therefore ensuring that the snake can collect every apple without intersecting itself. However, our program turns out to be relatively slow compared to other, more sophisticated programs that use other parts of combinatorics, such as lattice paths. Our reliance on using a Hamiltonian path led to exceedingly slow processing times as the dimensions of the grid increased due to the nature of said pathways.

7 Appendix

7.1 Vertex List Generation for a Grid

```
def GridGraphGen(height,width):
    totalnodes = height*width
    graphlist = []
    for node in range(totalnodes):
        tempnode = [0] * totalnodes
        row = node // width
        col = node - row*width

        firstrow = row == 0
        lastrow = row == height-1

        firstcol = col == 0
        lastcol = col == width - 1

        if firstrow:
            if firstcol:
                tempnode[node + width] = 1
                tempnode[node + 1] = 1
                #print(node, tempnode)
            elif lastcol:
                tempnode[node + width] = 1
                tempnode[node - 1] = 1
                #print(node, tempnode)
            else:
                tempnode[node - 1] = 1
                tempnode[node + 1] = 1
                tempnode[node + width] = 1
                #print(node, tempnode)
        elif lastrow:
            if firstcol:
                tempnode[node - width] = 1
                tempnode[node + 1] = 1
                #print(node, tempnode)
            elif lastcol:
                tempnode[node - width] = 1
                tempnode[node - 1] = 1
                #print(node, tempnode)
            else:
                tempnode[node - 1] = 1
                tempnode[node + 1] = 1
                tempnode[node - width] = 1
                #print(node, tempnode)
        else:
            if firstcol:
                tempnode[node - width] = 1
                tempnode[node + width] = 1
                tempnode[node + 1] = 1
            elif lastcol:
                tempnode[node - width] = 1
                tempnode[node + width] = 1
                tempnode[node - 1] = 1
            else:
                tempnode[node - width] = 1
                tempnode[node + width] = 1
                tempnode[node - 1] = 1
                tempnode[node + 1] = 1
        graphlist.append(tempnode)
    print("graph generated")
    return graphlist
```

7.2 Backtracking Algorithim

```
#/***********************************************************************************
#*    Title: Hamiltonian Backtracking
#*    Author: Geeks for Geeks
#*    Availability: https://www.geeksforgeeks.org/hamiltonian-cycle-backtracking-6/
#*    Source code partially modified to fit circumstances.
#************************************************************************************
```

```
class Graph():
    def __init__(self, vertices):
        self.graph = [[0 for column in range(vertices)]
                      for row in range(vertices)]
        self.V = vertices

    ''' Check if this vertex is an adjacent vertex
        of the previously added vertex and is not
        included in the path earlier '''

    def isSafe(self, v, pos, path):
        # Check if current vertex and last vertex
        # in path are adjacent
        if self.graph[path[pos - 1]][v] == 0:
            return False

        # Check if current vertex not already in path
        for vertex in path:
            if vertex == v:
                return False

        return True

    # A recursive utility function to solve
    # hamiltonian cycle problem
    def hamCycleUtil(self, path, pos):

        # base case: if all vertices are
        # included in the path
        if pos == self.V:
            # Last vertex must be adjacent to the
            # first vertex in path to make a cycle
            if self.graph[path[pos - 1]][path[0]] == 1:
                return True
            else:
                return False

        # Try different vertices as a next candidate
        # in Hamiltonian Cycle. We don't try for 0 as
        # we included 0 as starting point in hamCycle()
        for v in range(1, self.V):

            if self.isSafe(v, pos, path) == True:

                path[pos] = v

                if self.hamCycleUtil(path, pos + 1) == True:
                    return True

                # Remove current vertex if it doesn't
                # lead to a solution
                path[pos] = -1
        print(path)
        return False

    def hamCycle(self):
        path = [-1] * self.V

        ''' Let us put vertex 0 as the first vertex
            in the path. If there is a Hamiltonian Cycle,
            then the path can be started from any point
            of the cycle as the graph is undirected '''
        path[0] = 0

        if self.hamCycleUtil(path, 1) == False:
            return "Solution Does Not Exist"

        self.printSolution(path)
        return path

    def printSolution(self, path):
        print("Solution Exists: Following",
              "is one Hamiltonian Cycle")
        for vertex in path:
            print(vertex, end=" ")
        print(path[0], "\n")
```

7.3 Displaying Connected Grid Graph

```
import matplotlib.pyplot as plt
import networkx as nx

graph = nx.grid_2d_graph(10,9)
plt.figure(figsize=(5,5))
pos = {(x,y):(y,-x) for x,y in graph.nodes()}
nx.draw(graph, pos=pos,
        node_color='#1d5b75',
        with_labels=False,
        node_size=200)
plt.show()
```

References

G. T. (n.d.). *History of snake (video game): 26 facts / events (creators, versions,...).* Retrieved 2022-11-11, from https://www.gamesver.com/history-of-snake-video-game-facts-events-creators-versions/

Blog, M. D. (n.d.). *Snake charmed! 10 fascinating facts about the world's most popular game.* Retrieved 2022-12-03, from https://blogs.windows.com/devices/2012/02/02/snake-charmed-10-fascinating-facts-about-the-worlds-most-popular-game/

Chin, N. (n.d.). *Snake: Hamiltonian cycle.* Retrieved 2022-11-11, from https://kychin.netlify.app/snake-blog/hamiltonian-cycle/

Haidet, B. (n.d.). *How to win snake: The UNKILLABLE snake AI.* Retrieved 2022-11-11, from https://www.youtube.com/watch?v=TOpBcfbAgPg&ab_channel=AlphaPhoenix

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., ... Oliphant, T. E. (2020, September). Array programming with NumPy. *Nature*, *585*(7825), 357–362. Retrieved from https://doi.org/10.1038/s41586-020-2649-2 DOI: 10.1038/s41586-020-2649-2

Hunter, J. D. (2007). Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, *9*(3), 90–95. DOI: 10.1109/MCSE.2007.55

Keller, M. T., & Trotter, W. T. (n.d.). *Applied combinatorics* (2017.2 ed.). Retrieved 2022-11-11, from https://www.appliedcombinatorics.org/book/frontmatter-1.html

Sebastianelli, A., Tipaldi, M., Ullo, S. L., & Glielmo, L. (n.d.). A deep q-learning based approach applied to the snake game. In *2021 29th mediterranean conference on control and automation (MED)* (pp. 348–353). (ISSN: 2473-3504) DOI: 10.1109/MED51440.2021.9480232

Van Rossum, G., & Drake Jr, F. L. (1995). *Python reference manual.* Centrum voor Wiskunde en Informatica Amsterdam.

Wilson, R. J. (n.d.). *Introduction to graph theory* (5th ed ed.). Longman. (OCLC: ocn562773056)